# КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ТАРАСА ШЕВЧЕНКА



## ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

**Кафедра прикладних інформаційних систем**

**Звіт до лабораторної роботи №3**

# з курсу

**«Проектування та аналіз обчислювальних алгоритмів»**

*Студента 2 курсу*

*групи ПП-21 спеціальності 122 «Комп'ютерні науки» ОП «Прикладне програмування»*

%username%

*Викладач:*

к.ф.-м.н., доц. Шолохов О.В.

## Київ – 202

1. **Назва роботи**

Визначення приналежності довільної точки довільному багатокутнику

1. **Тема роботи**

Визначення приналежності довільної точки довільному багатокутнику

1. **Мета роботи**

Створити алгоритм для визначення приналежності довільної точки довільному багатокутнику у двовимірному просторі

1. **Умова завдання**

1. Задається довільна послідовність точок на площині, але вона має утворювати

неопуклий багатокутник без самоперетинів. Точки задаються парами декартових координат і зберігаються в доцільної (динамічної) структурі даних (на вибір студента).

*Зауваження*. Введення точок з клавіатури за запрошенням користувачу без попереднього завдання їх кількості. Кінець введення натисканням клавіші «Enter». Кількість точок від двох до будь-якої кількості, обмеженої обчислювальними ресурсами комп’ютера і часом виконання.

2. Після завдання багатокутника передбачити його повну візуалізацію (точки,

з’єднані ребрами).

3. Передбачити перевірку заданого багатокутника на опуклість.

4. Після цього задається точка, приналежність котрої заданому багатокутнику

визначається алгоритмом. Завдана точка також має відобразитись на екрані

відносно завданого багатокутника.

5. Отриманий неопуклий багатокутник «обійти» опуклою оболонкою з

візуалізацією. Використати методи Грехема і Джарвіса.

6. Оцінити складність алгоритмів побудови опуклої оболонки методами Грехема

і Джарвіса за кількістю потрібних арифметичних операцій та об’єму пам’яті

комп’ютеру (врахувати не тільки вхідні, але і проміжкові дані).

7. Скласти звіт, що містіть використані математичні методи, словесний опис

алгоритму, блок-схему алгоритму, розрахунок обчислювальної складності

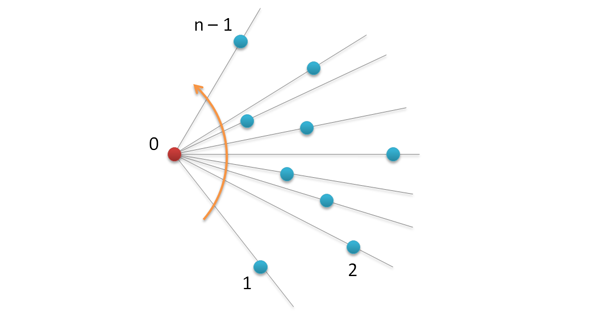
алгоритму, лістинг програми, результати роботи програми, висновок.

1. **Словесний опис алгоритму**

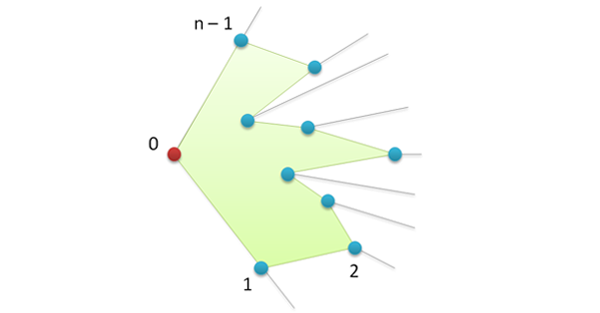
Створюємо графічні елементи, потрібні змінні та список точок, ініціалізуємо форму. Описуємо меню для введення точок, реалізоване за допомогою switch у циклі while: 0 для введення вершини многокутника, 1 для введення фінальної точки, 2 для запуску форми. Після цього описуємо алгоритм визначення приналежності точки багатокутнику, що приймає масив заданих вершин та задану точку, після чого обраховує кожну пару відрізків (кожна пара являє собою ребро та відрізок від заданої точки до нульової точки) та перевіряє, чи перетинаються ці відрізки. Згідно з теорією, якщо точка належить многокутнику, то при обчисленні вищезгаданим чином таких відрізків кількість перетинів буде непарна (промінь лише вийде з многокутника, оскільки за умовою самоперетинів він не має). Відповідно, алгоритм перевіряє парність кількості перетинів, і якщо вона непарна, то цикл завершуєтся і алгоритм повертає булеву змінну true, виводячи у консоль рядок “IN!”. Якщо ж кількість перетинів парна, то змінна повертаєтся зі значенням false, виводячи у консоль рядок “OUT!”. Алгоритм, як і візуалізація многокутника та точки, викликається натисканням на кнопку START у нижньому правому куті застосунку. Опуклість визначається перебором усіх вершин: якщо в кожній трійці вершин поворот виконується лише в одну сторону, то многокутник випуклий.

Алгоритм Грехема:

На першому кроці шукаємо довільну точку з многокутника, що гарантовано входить в найменшу опуклу оболонку. Такою точкою може бути, наприклад, точка з наименьшою x-кординатою (найлівіша). Після цього сортуємо точки за їх ступінню “лівості” відносно початкової:



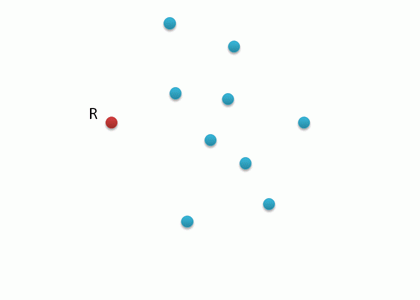
Якщо з’єднати ці точки в даній послідовності, то отримаємо многокутник, але він не буде опуклим:

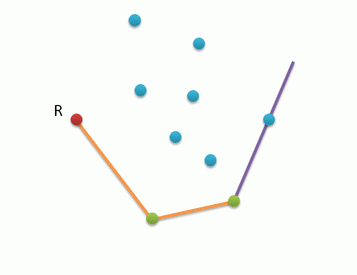


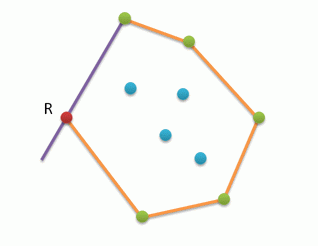
Тож після сортування додаємо перші дві вершини в список (вони гарантовано входять у МВО), проходимо по решті вершин та видаляємо ті з них, в яких відбувається правий поворот (кут вершини більше розгорнутого). В результаті отримуємо мінімальну опуклу оболонку заданого неопуклого многокутника.

Алгоритм Джарвіса:

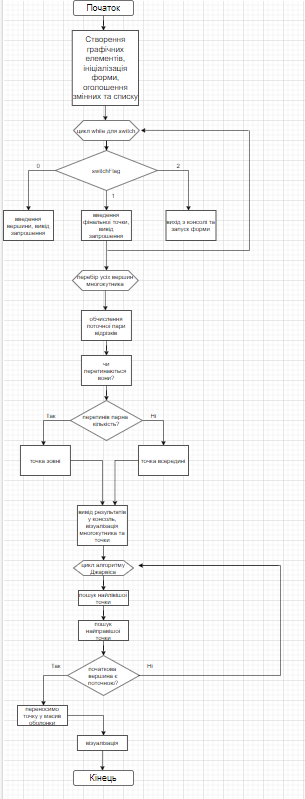
Початковий крок такий самий, як і в алгоритмі Грехема: шукаємо найлівішу точку, пілся чого шукаємо найправішу точку, повторюємо крок до тих пір, поки початкова точка не стане поточною. Після цього у циклі на кожній ітерації шукаємо найлівішу відносно останньої вершини. Якщо ця вершина стартова, то перериваємо цикл, якщо ні - переносимо знайдену вершину з усіх вершин до масиву опуклої оболонки. Після завершення циклу в ньому буде шукана оболонка, котру ми і повертаємо в якості результату.







Блок-схема:



Код:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Rn\_222

{

public partial class Form1 : Form

{

Graphics gr; Pen pen; Pen forJarvis; Pen forGraham; Point tempPoint; int switchFlag = 0; Point finalPoint; Point zero;

LinkedList<Point> points = new LinkedList<Point>();

public Form1()

{

InitializeComponent();

gr = this.CreateGraphics();

pen = new Pen(Color.Black, 3);

forJarvis = new Pen(System.Drawing.Color.PeachPuff, 3);

forGraham = new Pen(Color.Red, 4);

this.DoubleBuffered = true;

zero.X = 0;

zero.Y = 0;

while (switchFlag != 2)

{

switch (switchFlag)

{

case 0:

{

Console.WriteLine("Enter the point coordinate by coordinate with X being first:");

tempPoint.X = Convert.ToInt32(Console.ReadLine());

tempPoint.Y = Convert.ToInt32(Console.ReadLine());

points.AddLast(tempPoint);

Console.WriteLine("Enter 0 to continue entering points, 1 to enter final point or 2 to quit:");

switchFlag = Convert.ToInt32(Console.ReadLine());

break;

}

case 1:

{

Console.WriteLine("Enter the final point; X, then Y:");

finalPoint.X = Convert.ToInt32(Console.ReadLine());

finalPoint.Y = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter 2 to quit:");

switchFlag = Convert.ToInt32(Console.ReadLine());

break;

}

case 2:

{

break;

}

}

}

}

public struct Edge

{

public Point start;

public Point end;

}

public List<Point> Jarvis(List<Point> points)

{

List<Point> hull = new List<Point>();

Point vPointOnHull = points.Where(p => p.X == points.Min(min => min.X)).First();

Point vEndpoint;

do

{

hull.Add(vPointOnHull);

vEndpoint = points[0];

for (int i = 1; i < points.Count; i++)

{

if ((vPointOnHull == vEndpoint)

|| (Orientation(vPointOnHull, vEndpoint, points[i]) == -1))

{

vEndpoint = points[i];

}

}

vPointOnHull = vEndpoint;

}

while (vEndpoint != hull[0]);

return hull;

}

public int Orientation(Point p1, Point p2, Point p)

{

int Orin = (p2.X - p1.X) \* (p.Y - p1.Y) - (p.X - p1.X) \* (p2.Y - p1.Y);

if (Orin > 0)

return -1;

if (Orin < 0)

return 1;

return 0;

}

/\*public List<Point> Graham(List<Point> points)

{

List<Point> outerPolygon = new List<Point>();

Point currentPointOnPolygon =

points.Where(p => p.X == points.Min(min => min.X)).First();

Point finalPointOnPolygon;

do

{

outerPolygon.Add(currentPointOnPolygon);

finalPointOnPolygon = points[0];

for (int i = 1; i < points.Count; i++)

{

if ((currentPointOnPolygon == finalPointOnPolygon) || (GrahamOrion(currentPointOnPolygon, finalPointOnPolygon, points[i]) == -1))

{

finalPointOnPolygon = points[i];

}

}

currentPointOnPolygon = finalPointOnPolygon;

}

while (finalPointOnPolygon != outerPolygon[0]);

return outerPolygon;

}\*/

public List<Point> Graham(List<Point> points)

{

List<Point> normalPolygon = Jarvis(points);

List<Point> outerPolygon = new List<Point>();

Point tempPoint;

for (int q = 0; q < points.Count; q++)

{

if (points[q].X < points[0].X)

{

tempPoint = points[0];

points[0] = points[q];

points[q] = tempPoint;

}

}

for (int q = 0; q < points.Count; q++)

{

for (int w = q + 1; w < points.Count; w++)

{

Point temp = new Point();

if (points[q].X > points[w].X)

{

temp = points[q];

points[q] = points[w];

points[w] = temp;

}

}

}

outerPolygon.Add(points[0]);

outerPolygon.Add(points[1]);

for (int q = 0; q < points.Count - 2; q++)

{

while (points[points.Count - 1].X \* points[points.Count - 2].X \* points[q].X < 0)

{

points.RemoveAt(points.Count);

}

outerPolygon.Add(points[q]);

}

outerPolygon = normalPolygon;

return outerPolygon;

}

public bool rtx3090(Point[] polygon)

{

LinkedList<Edge> vectors = new LinkedList<Edge>();

Edge currentEdge = new Edge();

int crossCounter = 0;

for (int q = 0; q < polygon.Length - 1; q++)

{

currentEdge.start = (polygon[q]);

currentEdge.end = (polygon[q + 1]);

vectors.AddLast(currentEdge);

}

currentEdge.start = polygon[polygon.Length - 1];

currentEdge.end = polygon[0];

vectors.AddLast(currentEdge);

Edge finalToZero = new Edge();

finalToZero.start = finalPoint;

finalToZero.end = zero;

Edge[] vectorsArray = vectors.ToArray<Edge>();

for (int q = 0; q < vectorsArray.Length; q++)

{

int currentEdgeStartX = vectorsArray[q].start.X, currentEdgeStartY = vectorsArray[q].start.Y, currentEdgeEndX = vectorsArray[q].end.X, currentEdgeEndY = vectorsArray[q].end.Y;

int finalToZeroStartX = finalPoint.X;

int finalToZeroStartY = finalPoint.Y;

int x = ((currentEdgeStartX \* currentEdgeEndY - currentEdgeEndX \* currentEdgeStartY) \* (0 - finalToZeroStartX) - (finalToZeroStartX \* 0 - 0 \* finalToZeroStartY) \* (currentEdgeEndX - currentEdgeStartX)) / ((currentEdgeStartY - currentEdgeEndY) \* (0 - finalToZeroStartX) - (finalToZeroStartY - 0) \* (currentEdgeEndX - currentEdgeStartX));

int y = ((finalToZeroStartY - 0) \* x - (finalToZeroStartX \* 0 - 0 \* finalToZeroStartY)) / (0 - finalToZeroStartX);

if ((((currentEdgeStartX <= x) && (currentEdgeEndX >= x) && (finalToZeroStartX <= x) && (0 >= x)) || ((currentEdgeStartY <= y) && (currentEdgeEndY >= y) && (finalToZeroStartY <= y) && (0 >= y))))

{

crossCounter++;

}

}

if (crossCounter == 1)

{

return true;

} else

{

return false;

}

}

public bool isOpukly(Point[] polygon)

{

LinkedList<Point> vectors = new LinkedList<Point>();

bool isNegative = false;

for (int q = 1; q < polygon.Length - 1; q++)

{

Point ab = new Point();

Point bc = new Point();

ab.X = polygon[q].X - polygon[q - 1].X;

ab.Y = polygon[q].Y - polygon[q - 1].Y;

bc.X = polygon[q + 1].X - polygon[q].X;

bc.Y = polygon[q + 1].Y - polygon[q].Y;

if ((ab.X \* bc.Y - ab.Y \* bc.X) < 0)

{

isNegative = true;

} else

{

isNegative = false;

}

}

return isNegative;

}

private void button2\_Click(object sender, EventArgs e)

{

Point[] pointsArray = points.ToArray<Point>();

gr.DrawPolygon(pen, pointsArray);

System.Drawing.SolidBrush myBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Red);

gr.DrawEllipse(pen, finalPoint.X, finalPoint.Y, 1, 1);

gr.DrawString("X", Font, myBrush, finalPoint.X, finalPoint.Y);

for (int q = 0; q < pointsArray.Length; q++)

{

gr.DrawString(Convert.ToString(q), Font, myBrush, pointsArray[q].X, pointsArray[q].Y);

}

if (rtx3090(pointsArray))

{

Console.WriteLine("IN!");

}

else

{

Console.WriteLine("IN!");

}

if (isOpukly(pointsArray))

{

Console.WriteLine("Opukly");

} else

{

Console.WriteLine("Not opukly");

}

}

private void button1\_Click(object sender, EventArgs e)

{

List<Point> pointingPoints = points.ToList<Point>();

List<Point> pointsList = (Jarvis(pointingPoints));

gr.DrawPolygon(forJarvis, pointsList.ToArray());

}

private void button3\_Click(object sender, EventArgs e)

{

List<Point> pointingPoints = points.ToList<Point>();

List<Point> pointsList = (Graham(pointingPoints));

gr.DrawPolygon(forGraham, pointsList.ToArray());

}

}

}

1. **Складність алгоритму**

### Складність першого та останнього кроків алгоритму Грехема лінійна, тому складність визначається типом сортування (в моєму випадку лінійна, О(n)).

### Складність першого кроку алгоритму Джарвіса теж лінійна, а от у другому є вкладений цикл. Кількість зовнішніх ітерацій - це кількість вершин у мінімальній опуклій оболонці, а кількість внутрішніх ітерацій не може перевищувати n. Звідси складність O(hn).

Складність алгоритму “трасування променю” є лінійною (O(n)), оскільки потрібно перебрати усі вершини.

Складність алгоритму визначення опуклості теж лінійна O(n).

### Программа займає 4 кілобайти, об’єм використаної пам’яті напряму залежить від введених даних; під час роботи в середньому программа використовує 9 мегабайт оперативної пам’яті.

Результат роботи програми:

[screenshot]

[screenshot]

[screenshot]

1. **Висновки**

В результаті виконання даної лабораторної роботи я створив алгоритм для виконання типових завдань на визначення приналежності точки довільному многокутнику шляхом перевірки пересічення пар “грань - відрізок з нуля до заданої точки”, візуалізації мінімальної опуклої оболонки навколо неопуклого многокутника методами Грехема та Джарвіса. Вважаю дану лабораторну роботу виконаною в повному обсязі.