Звіт

Тема 45. Близькість. Усі найближчі сусіди

Шевченко Максим, МІ-3

**Вступ**

Сучасний розвиток комп'ютерних технологій дозволяє ставити і вирішувати нові складні завдання, які потребують побудові комплексних математичних моделей і способів їх вирішення. Такими завданнями є завдання геометричного і графічного моделювання явищ і процесів. Однією з ефективних стратегій вирішення зазначених завдань є визначення єдиної загальної методології їх вирішення в даній області. Тому природним є питання розробки спільних підходів, які б дозволяли ефективно вирішувати одночасно ряд геометричних задач на однакових вхідних даних: побудови політопів, тріангуляції, завдань близькості і ряду інших важливих завдань. Одним із методів їх вирішення є алгоритми на стратегії «Розділяй та Підкорюй», про які мова піде в основній частині.

**Умова**

Для заданої множини S із N точок в E2 просторі за допомогою стратегії «Р та П» (МЄАС) знайти усіх найближчих сусідів.

**Теорія**

**Задача пошуку найближчого сусіда** є задачею оптимізації, яка полягає у відшуканні у множині елементів, розташованих у багатовимірному метричному просторі, елементів, близьких до заданого, відповідно до заданої функції близькості. Формально ця задача ставиться наступним чином: надано множину точок S у просторі M та точку q ∈ M, необхідно знайти найближчу до q точку в S. Дональд Кнут в Мистецтві програмування (том 3, 1973) назвав це проблемою поштового відділення, посилаючись на застосування цієї задачі до пошуку найближчого поштового відділення. Прямим узагальненням задачі пошуку найближчого сусіда є алгоритм пошуку k-NN, який призначений для пошуку k найближчих точок.

Найчастіше M є метричним простором і запроваджується функція близькості, що визначається як метрика, яка є симетричною і задовольняє нерівності трикутника. Ще загальніше, M — це d-вимірний векторний простір, в якому близькість береться як Евклідова метрика, вулична метрика або інші метрики. Однак функція близькості може бути довільною. Одним з прикладів може бути метрика Брегмана, для якої нерівність трикутника не виконується.

“Найближчий сусід” – це відношення на множині точок S, яке визначається таким чином: точка b є найближчим сусідом точки a ( позначається a → b) , якщо dist(a,b) = mindist(a,c) , cЄS –a

Властивості відношення “найближчий сусід”

1. Відношення в загальному випадку не є симетричним.

2. Окрема точка на множині S із N точок може мати N-1 найближчого сусіда.

3. Окрема точка на множині S із N точок може бути найближчим сусідом не більше ніж для 6 точок на площині та не більше 12 у тривимірному просторі. Це число не відоме для просторів розмірності d > 12.

4. Якщо процедура вибору точок множини S здійснюється по нормальному закону, то математичне сподівання долі взаємних пар точок буде рівне (S)= 6π/(8π+3(3)1/2) ~ 0.6215 (число Піелоу).

Задача пошуку найближчого сусіда зустрічається у багатьох областях, наприклад:

* розпізнавання образів;
* класифікація документів;
* рекомендаційні і експертні системи;
* динамічне розміщення реклами в Інтернеті.

Перед вирішенням прикладної задачі необхідно обрати форму подання об'єктів і функцію близькості. У більшості випадків об'єкти подаються у вигляді багатовимірних векторів, а як функція близькості використовується скалярний добуток векторів, але можуть бути й інші форми подання даних, наприклад:

* множини — розмір перетину множин;
* рядки — відстань Левенштейна;
* граф — відповідність структур.

Існують численні варіанти задачі пошуку найближчих сусідів. Окрім класичної задачі знаходження найближчої до заданої точки, можуть бути поставлені задачі:

* знайти близьких сусідів (не обов'язково найближчого);
* знайти найближчого сусіда для групи елементів;
* знайти кількох найближчих сусідів;
* знайти усі пари елементів, відстань між якими менша за деяку задану;
* знайти найближчих сусідів у середі, що динамічно змінюється.

Було запропоновано багато алгоритмів вирішення задачі пошуку найближчого сусіда. Якість та корисність алгоритмів визначається часовою складністю запитів, а також складністю усіх структур пошуку інформації, що мають підтримуватися. Не існує загального вирішення задачі у багатовимірному Евклідовому просторі, яке б використовувало поліноміальний час попередньої обробки та полі-логарифмічний час пошуку даних.

Найпростішим способом розв'язання задачі, є обчислення відстані від заданої точки до кожної іншої точки в наборі даних, постійно відстежуючи найкращий результат на даний момент. Цей алгоритм називають прямими методом, і його складність виконання становить O(dN), де N — це потужність множини точок S, а d — це розмірність простору M. Для реалізації не потрібні ніякі додаткові структуру для пошуку даних, тому лінійний пошук не потребує додаткового простору даних крім початкового набору даних.

Розбиття простору:

* Діаграма Вороного;
* KD-дерева;
* BSP-дерева;
* Дерева покриттів;
* VP-дерево;
* R-дерево.

**Діаграма Вороного** — це особливий вид розбиття метричного простору, що визначається відстанями до заданої дискретної множини ізольованих точок цього простору. Вона названа на честь українського математика Георгія Вороного. Інші назви — теселяція Вороного, декомпозиція Вороного, чи теселяція Діріхле (на честь Лежена Діріхле).

У найпростішому випадку, ми маємо множину точок площини S, які називаються вершинами діаграми Вороного. Кожній вершині s належить комірка Вороного, також відома як комірка Діріхле, V(s), утворена з усіх точок ближчих до s ніж до будь-якої іншої вершини. Границі на діаграмі Вороного являють собою всі точки на площині, які рівновіддалені від двох найближчих вершин. Вузли Вороного — точки рівновіддалені від трьох і більше вершин.

Задачі, з визначеними оцінками складності, які можна звести до розглядуваних називаються задачами – прототипами. Вони є базовими для певних класів.

Задача НАЙБЛИЖЧА ПАРА зводима до УСІ НАЙБЛИЖЧІ СУСІДИ і задача НАЙБЛИЖЧА ПАРА зводима до ЕМКД, а отже задача НАЙБЛИЖЧА ПАРА є їх прототипом. Задача НАЙБЛИЖЧИЙ СУСІД зводима до k-НАЙБЛИЖЧИХ СУСІДІВ , а отже є її прототипом. Кожний найближчий сусід точки pi множини S визначає ребро у многокутнику Вороного Vi. Задача УСІ НАЙБЛИЖЧІ СУСІДИ зводима до ДІАГРАМА ВОРОНОГО і тому її можна розв’язати за оптимальний час O(NlogN).

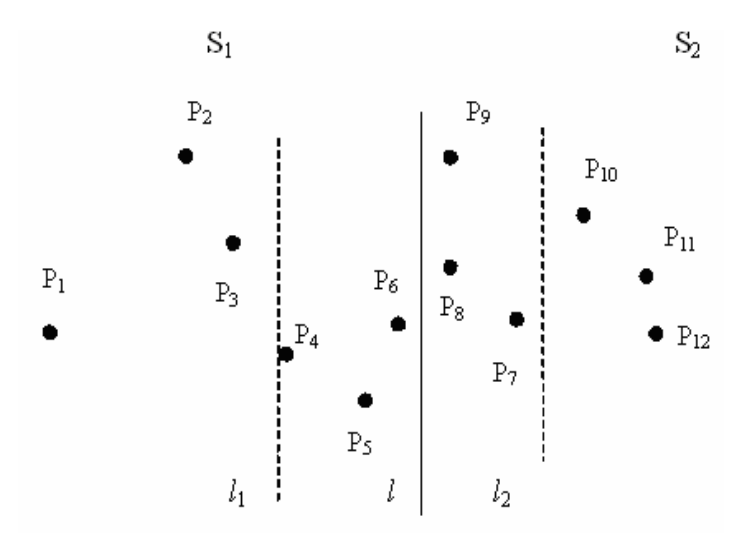
Для побудови алгоритмів з оцінкою складності O(NlogN) є два шляхи: сортування і використання методу ’’розподіляй та володарюй’’. Сортування можна зразу відкинути, так як воно зручне лише в умовах повної впорядкованості, яка полягає у проектуванні усіх точок на деяку пряму, але при цьому втрачається суттєва в даному випадку інформація.

Другий шлях для досягнення складності О(N log N) - стратегія «розділяй та пануй». При Цьому застосування такої стратегії буде успішним, якщо час кроку злиття буде лінійним. Позначивши через P(N, 2) час роботи алгоритму, який шукає найближчу пару точок на площині, отримаємо рекурентне співвідношення: P(N, 2) = 2P(N/2, 2) + O(N2). Розв’язком цього співвідношення є P(N, 2) = O(N2), яке не дає бажану оцінку. Але, враховуючи властивості відношення близькості, можна крок злиття зробити лінійним.

**Алгоритм**

Математична модель пропонованого алгоритму складається з таких основних етапів: попередня обробка, розбиття множини даних (рекурсивний спуск), рекурсивне злиття результатів для підмножин (рекурсивний підйом).

**Етап 1. Попередня обробка.** Нехай задана множина S з N точок на площині (рис. 1). На етапі попередньої обробки формується впорядкований масив точок U = {Pij, і, j = 1, N}. При цьому індекси i, j вказують номер точки в упорядкованому списку по х і y координаті відповідно. Сформований у такий спосіб масив подається на вхід алгоритму, який містить граф розбиття. У цьому графі (дереві) кожен вузол позначений цілим числом k, щодо якого розбивається список точок у вузлах на два рівнопотужних, щодо медіани, списки після порівняння перших індексів елементів масиву U. А кожен номер вузла k визначається за один прохід по дереву, якщо відомо кількість точок заданої множини.



Рисинук 1 – Розбиття вхідної множини точок

**Етап 2. Розбиття множини точок (рекурсивний спуск).** Цей етап алгоритму полягає в розбитті на кожному кроці рекурсії заданої множини точок у вигляді списку U на рівнопотужні підмножини U1, U2, пошуку медіани l і передачі U1, U2 на наступний крок рекурсії. Структура даних для цього процесу зображена на рисунку 2. Пошук медіани на упорядкованому по координаті х індексованому масиві точок U виконується за константний час О(1): l = (Pkj + Pk + 1j) / 2, де k - номер вузла. Час, необхідний на рекурсивний спуск паралельного алгоритму, визначається наступною лемою.

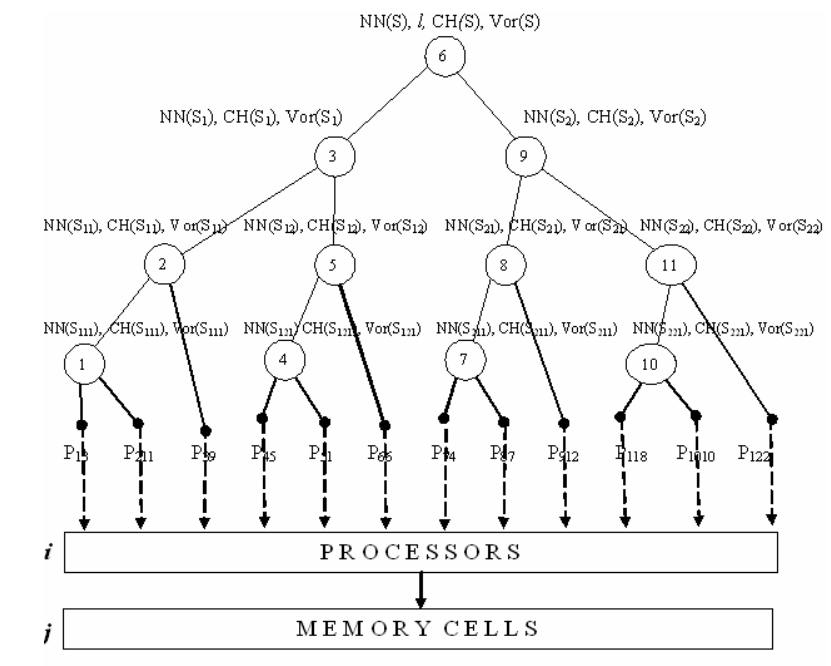


Рисунок 2 - Граф алгоритму. NN(S), Vor(S), CH(S) – процедури злиття; l – медіана

На вхід алгоритму подається безліч точок у вигляді індексованого двовимірного упорядкованого масиву U = {Pij, і, j = 1, N}. Таке уявлення множини точок дозволяє, знаючи кількість точок N в списку U, побудувати дерево розбиття.

**Етап 3. Рекурсивне злиття результатів для підмножин (рекурсивний підйом).** На цьому етапі, починаючи з листових вузлів дерева алгоритму, отримані результати рішення задачі подаються в батьківські вузли, де використовуються відповідні процедури злиття для побудови спільного рішення задачі. Процес завершується результатом злиття в кореневому вузлі.

На кожному кроці рекурсивного підйому, починаючи з другого, на вхід батьківського вузла v графа алгоритму подаються діаграми Вороного (ДВ) від лівого і правого синів vor(SL), vor(SR), а також найближчий сусід кожної точки підмножин SL, SR. Будується діаграма Вороного для вузла v і паралельно визначаються нові сусіди на границі підмножин SL, SR, щодо медіани l. На етапі злиття алгоритму, під час побудови розділяючого ланцюга діаграми Вороного, паралельно здійснюється пошук найближчих сусідів серед точок множин SL і SR, які утворюють поточну пару ребра ланцюга, σ(SL, SR). При цьому наступна пара, яка визначає нове ребро розділяючого ланцюга перевіряється на наявність нового найближчого сусіда. Позначимо як NN(S) множину пар найближчих сусідів для множини S, а NN(SL), NN(SR) - для множин SL і SR відповідно. Граф алгоритму в даному випадку - це двійкове дерево, з тією лише різницею, що кожен вузол дерева завантажується крім упорядкованого масиву точок медіани l діаграмами Вороного лівого vor(SL) і правого vor(SR) синів, множинами сусідніх пар точок синів NN(SL) і NN(SR) відповідно.

Для побудови злиття результатів пошуку найближчих сусідів для вузла v, необхідно:

1. Визначити верхні і нижні опорні вершини лівої і правої опуклих оболонок множини точок синів SL і SR, а отже і верхнє, і нижнє опорні ребра відповідно.

2. Верхні опорні вершини (qВОL, qВОR) утворюють першу пару точок, для яких необхідно виконати наступні дії:

- взяти верхню ліву опорну точку qВОL множини SL і порівняти відстань δ1 = dist(qВОL, qВОR) від неї до верхньої правої опорної точки qВОR множини SR з відстанню δВОL = dist (qВОL, PL) до її найближчого сусіда PL, знайденого на попередніх кроках;

- якщо це відстань менша, то найближчим сусідом для точки qВОL стає точка qВОR з відстанню δ1;

- інакше - найближчий сусід точки qВОL не змінюється;

- аналогічно визначається найближчий сусід точки qВОR.

3. Провести вхідне й вихідне ребро розділяючого ланцюга для опорних відрізків до перетину з одним з ребер діаграм vor(SL) або vor(SR).

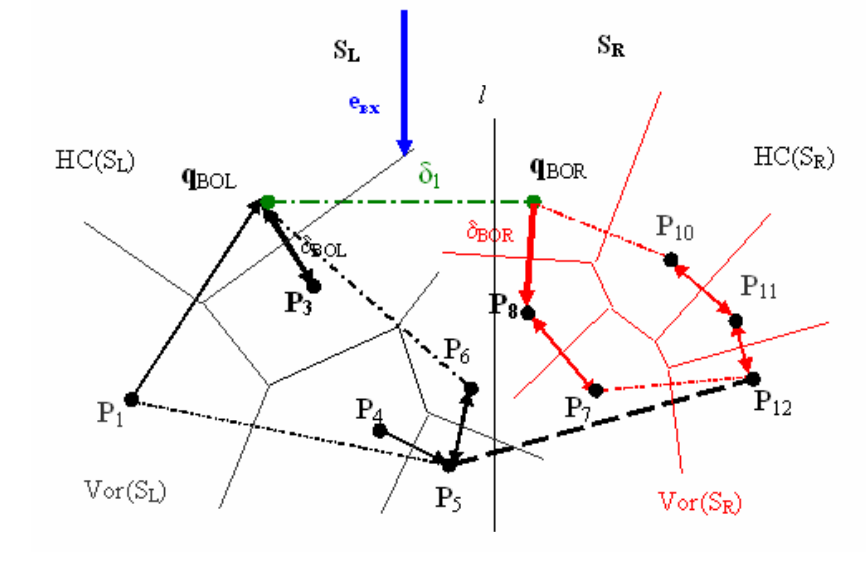


Рисунок 3 – Перевірка на наявність найближчих сусідів пари точок (qВОL, qВОR)

4. Проводиться процес знаходження найближчих сусідів (аналогічно пункту 2) для наступних пар точок (PiL, PjR) (де PiL Є SL, PjR Є SR, і, j = 1, N), які визначаються відповідними ребрами розділяючого ланцюга σ(SL, SR). Ланцюг будується для граничних множин точок щодо медіани l, розташованих зліва і праворуч від неї, які належать взаємно-опуклим ланцюгах опуклих оболонок синів, і точок, визначених ребрами діаграм vor(SL), vor(SR), які перетинають ребра цих ланцюгів.

5. Отримані після злиття таким способом множини пар найближчих сусідів і відповідні діаграми Вороного передаються на наступний рівень рекурсії.

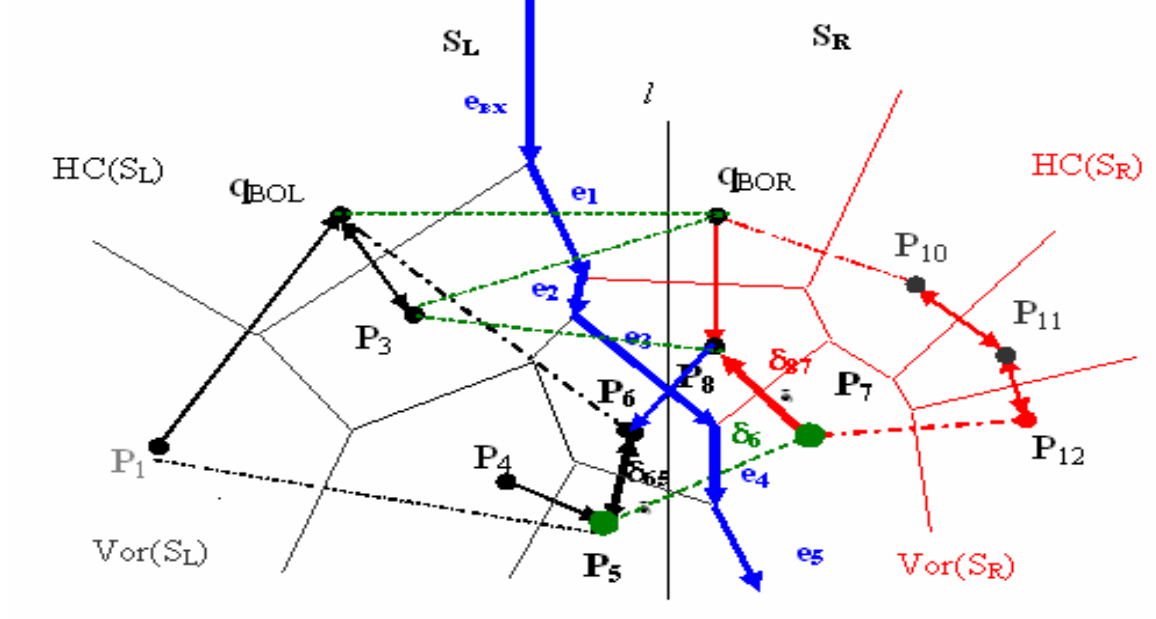


Рисунок 4 – Перевірка на наявність найближчих сусідів для пар точок (Р3, qВОR), (Р3, Р8), (Р6, Р8), (Р6, Р7), (Р5, Р7). Для Р8 Є SR знайдено нового найближчого сусіда P6 Є SL

6. Процес злиття завершується, коли буде побудований розділяючий ланцюг для кореня дерева, і тим самим визначено всі множини пар найближчих сусідів NN(S) множини точок S.

**Висновки**

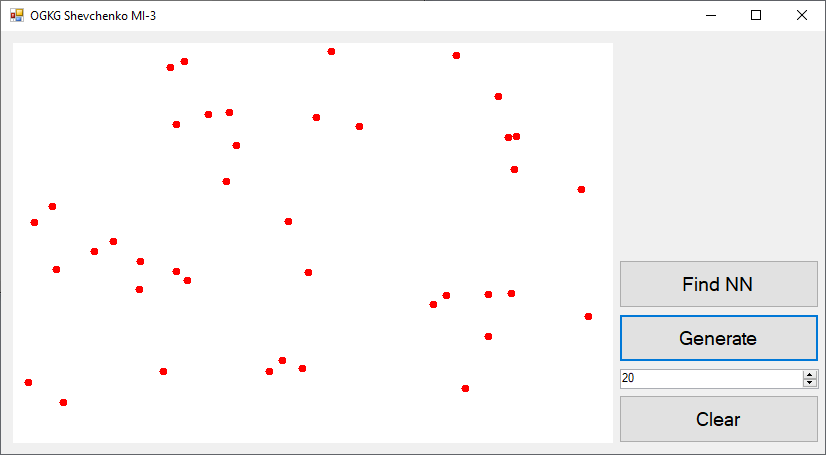
У даній лабораторній роботі було запропоновано алгоритм пошуку всіх найближчих сусідів за стратегією «Розділяй та Підкорюй» та виконано його програмну реалізацію.

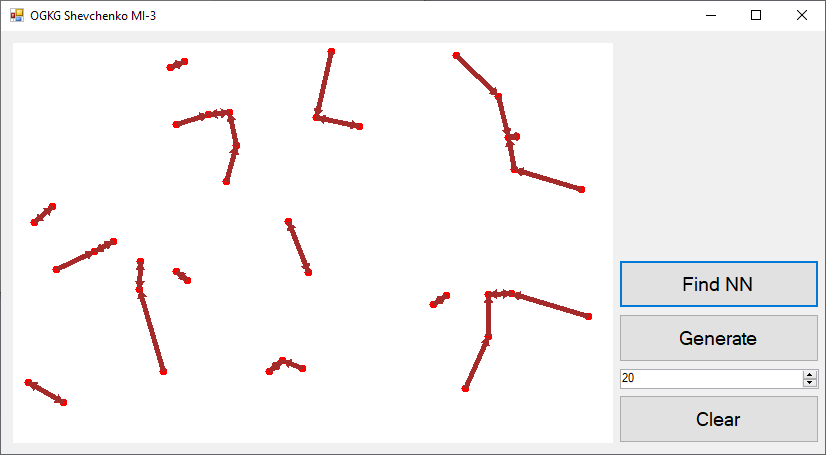
**Література**

1. В. М. Терещенко. «Подход к решению взаимосвязанных задач геометрического моделирования»
2. В. М. Терещенко. Презентація «Задачі Близькості»
3. Пошук найближчого сусіда. Електронний ресурс <https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D1%88%D1%83%D0%BA_%D0%BD%D0%B0%D0%B9%D0%B1%D0%BB%D0%B8%D0%B6%D1%87%D0%BE%D0%B3%D0%BE_%D1%81%D1%83%D1%81%D1%96%D0%B4%D0%B0>
4. Діаграма Вороного. Електронний ресурс <https://uk.wikipedia.org/wiki/%D0%94%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0_%D0%92%D0%BE%D1%80%D0%BE%D0%BD%D0%BE%D0%B3%D0%BE>

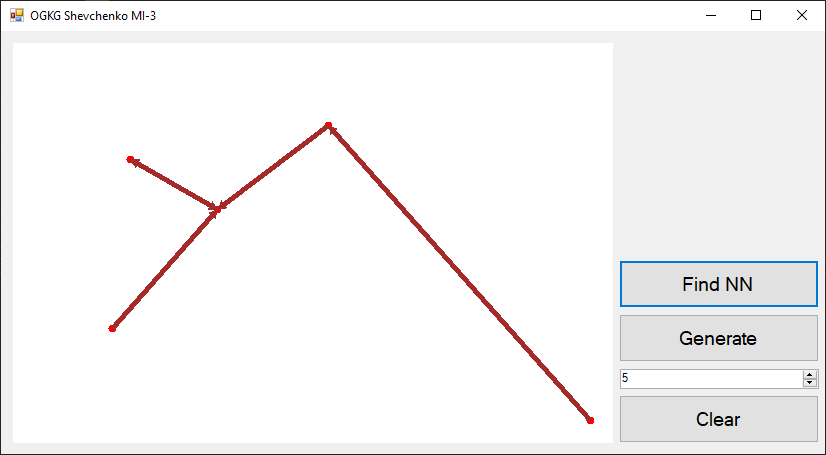
**Додатки**

Приклад виконання









Програмна реалізація

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;

using System.Xml.XPath;

namespace OGKGlab

{

public partial class Form1 : Form

{

bool visvor = false;

int maxPid = -1;

List<MyPoint> points = new List<MyPoint>();

List<NearestNeighbor> nearnei = new List<NearestNeighbor>();

List<VorNode> vor = new List<VorNode>();

List<int> Ux = new List<int>();

List<int> Uy = new List<int>();

List<int[]> Uij = new List<int[]>();

int DELTA = 1;

int RADIUS = 8;

int WIDTH = 0;//ширина 600

int HEIGHT = 0;//висота 400

Graphics g;

Pen arrow = new Pen(Color.Brown, 5);

Pen line = new Pen(Color.Blue, 3);

SolidBrush brush = new SolidBrush(Color.Red);

public Form1()

{

InitializeComponent();

WIDTH = pictureBox1.Width;

HEIGHT = pictureBox1.Height;

g = pictureBox1.CreateGraphics();

arrow.EndCap = System.Drawing.Drawing2D.LineCap.ArrowAnchor;

}

private void pictureBox1\_MouseClick(object sender, MouseEventArgs e)

{

AddPoint(e.X, e.Y);

VisualPoints();

}

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

{

Random rand = new Random();

int count = (int)nupdCount.Value;

int generated = 0;

while (generated < count)

{

if (AddPoint(rand.Next() % WIDTH, rand.Next() % HEIGHT))

++generated;

}

VisualPoints();

}

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

{

vor.Clear();

nearnei.Clear();

points.Clear();

maxPid = -1;

g.Clear(Color.White);

VisualPoints();

}

bool AddPoint(int x, int y)

{

if (x < RADIUS || x > WIDTH - 2 \* RADIUS) return false;

if (y < RADIUS || y > HEIGHT - 2 \* RADIUS) return false;

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

{

if ((x > points[i].x - RADIUS && x < points[i].x + RADIUS)

&& (y > points[i].y - RADIUS && y < points[i].y + RADIUS)) return false;

}

++maxPid;

points.Add(new MyPoint(maxPid, x, y));

return true;

}

void SortByX()

{

Ux.Clear();

List<MyPoint> buf = new List<MyPoint>();

buf.AddRange(points);

while (buf.Count > 0)

{

int min\_id = 0;

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

{

if (buf[i].x <= buf[min\_id].x) min\_id = i;

}

Ux.Add(buf[min\_id].id);

buf.RemoveAt(min\_id);

}

}

void SortByY()

{

Uy.Clear();

List<MyPoint> buf = new List<MyPoint>();

buf.AddRange(points);

while (buf.Count > 0)

{

int min\_id = 0;

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

{

if (buf[i].y <= buf[min\_id].y) min\_id = i;

}

Uy.Add(buf[min\_id].id);

buf.RemoveAt(min\_id);

}

}

void SortByXandY()

{

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

{

int[] buf = new int[2];

buf[0] = Ux.IndexOf(i);

buf[1] = Uy.IndexOf(i);

Uij.Add(buf);

}

}

double FindAngle(int xc, int yc, int xt, int yt, int x, int y)

{

int ax = xt - xc;

int ay = yt - yc;

int bx = x - xc;

int by = y - yc;

double asize = Math.Sqrt(ax \* ax + ay \* ay);

double bsize = Math.Sqrt(bx \* bx + by \* by);

int scalar = ax \* by - ay \* bx;

double angle = Math.Acos((ax \* bx + ay \* by) / (asize \* bsize));

if (scalar >= 0) return angle;

return 2.0 \* Math.PI - angle;

}

int[] FindCrossBorder(double A, double B, double C)

//return x1, y1, x2, y2

{

//Ax+By+C=0

int[] res = new int[4];

int count = -1;

//y=0

double Ab = 0;

double Bb = 1;

double Cb = 0;

int x = FindCrossX(A, B, C, Ab, Bb, Cb);

int y = FindCrossY(A, B, C, Ab, Bb, Cb);

if ((x >= 0 && x <= WIDTH) && (y == 0))

{

++count;

res[count] = x;

++count;

res[count] = y;

}

//y=HEIGHT

Ab = 0;

Bb = 1;

Cb = 0 - HEIGHT;

x = FindCrossX(A, B, C, Ab, Bb, Cb);

y = FindCrossY(A, B, C, Ab, Bb, Cb);

if ((x >= 0 && x <= WIDTH) && (y == HEIGHT))

{

++count;

res[count] = x;

++count;

res[count] = y;

}

//x=0

Ab = 1;

Bb = 0;

Cb = 0;

x = FindCrossX(A, B, C, Ab, Bb, Cb);

y = FindCrossY(A, B, C, Ab, Bb, Cb);

if ((x == 0) && (y >= 0 && y <= HEIGHT) && count < 3)

{

++count;

res[count] = x;

++count;

res[count] = y;

}

//x=WIDTH

Ab = 1;

Bb = 0;

Cb = 0 - WIDTH;

x = FindCrossX(A, B, C, Ab, Bb, Cb);

y = FindCrossY(A, B, C, Ab, Bb, Cb);

if ((x == WIDTH) && (y >= 0 && y <= HEIGHT) && count < 3)

{

++count;

res[count] = x;

++count;

res[count] = y;

}

return res;

}

int FindCrossX(double A1, double B1, double C1, double A2, double B2, double C2)

{

double zn = A1 \* B2 - A2 \* B1;

if (zn == 0) return -1;

double x = (-1.0) \* (C1 \* B2 - C2 \* B1) / zn;

if (0 - DELTA <= x && 0 + DELTA >= x) x = 0;

if (WIDTH - DELTA <= x && WIDTH + DELTA >= x) x = WIDTH;

return (int)x;

}

int FindCrossY(double A1, double B1, double C1, double A2, double B2, double C2)

{

double zn = A1 \* B2 - A2 \* B1;

if (zn == 0) return -1;

double y = (-1.0) \* (A1 \* C2 - A2 \* C1) / zn;

if (0 - DELTA <= y && 0 + DELTA >= y) y = 0;

if (HEIGHT - DELTA <= y && HEIGHT + DELTA >= y) y = HEIGHT;

return (int)y;

}

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

{

if (points.Count > 1)

{

SortByX();

SortByY();

SortByXandY();

List<MyPoint> sortedPoints = new List<MyPoint>();

foreach (int i in Ux)

{

sortedPoints.Add(points[i]);

}

g.Clear(Color.White);

VisualPoints();

Result res = FindAllNN(sortedPoints);

VisualVor(res.res\_vor);

VisuаlNN(res.res\_nn);

}

}

Result FindAllNN(List<MyPoint> p)//на вхід подавати відсортовані по х точки

{

if (p.Count == 2)

{

double xsp = (double)(p[0].x + p[1].x) / 2;

double ysp = (double)(p[0].y + p[1].y) / 2;

double kab = (double)(p[0].y - p[1].y) / (p[0].x - p[1].x);

double ksp = (-1.0) / kab;

double b = (double)ysp - (double)ksp \* xsp;

int[] res = FindCrossBorder(ksp, -1, b);

VisualVor(new List<VorNode> { new VorNode(p[0].id, p[1].id, res[0], res[1], res[2], res[3]) });

List<NearestNeighbor> nnbuf = new List<NearestNeighbor>();

nnbuf.Add(new NearestNeighbor(p[0].id, p[1].id, Math.Sqrt((p[0].x - p[1].x) \* (p[0].x - p[1].x) + (p[0].y - p[1].y) \* (p[0].y - p[1].y))));

nnbuf.Add(new NearestNeighbor(p[1].id, p[0].id, Math.Sqrt((p[0].x - p[1].x) \* (p[0].x - p[1].x) + (p[0].y - p[1].y) \* (p[0].y - p[1].y))));

List<VorNode> vorbuf = new List<VorNode>();

vorbuf.Add(new VorNode(p[0].id, p[1].id, res[0], res[1], res[2], res[3]));

return new Result(p, nnbuf, vorbuf);

}

if (p.Count == 3)

{

List<NearestNeighbor> nnbuf = new List<NearestNeighbor>();

List<VorNode> vorbuf = new List<VorNode>();

double[] xsp = new double[3];

double[] ysp = new double[3];

double[] A = new double[3];

double[] B = new double[3];

double[] C = new double[3];

for (int i = 0; i < 3; ++i)

{

xsp[i] = (double)(p[i].x + p[(i + 1) % 3].x) / 2;

ysp[i] = (double)(p[i].y + p[(i + 1) % 3].y) / 2;

double dx = (p[i].x - p[(i + 1) % 3].x);

double dy = (p[i].y - p[(i + 1) % 3].y);

if (dx == 0)

{

A[i] = 0;

B[i] = (-1.0);

C[i] = ysp[i];

}

else

{

if (dy == 0)

{

A[i] = (-1.0);

B[i] = 0;

C[i] = xsp[i];

}

else

{

double kab1 = dy / dx;

A[i] = (-1.0) / kab1;

B[i] = (-1.0);

C[i] = (double)ysp[i] - (double)A[i] \* xsp[i];

}

}

int nnid;

double size = Math.Sqrt((p[i].x - p[(i + 1) % 3].x) \* (p[i].x - p[(i + 1) % 3].x) + (p[i].y - p[(i + 1) % 3].y) \* (p[i].y - p[(i + 1) % 3].y));

double bsize = Math.Sqrt((p[i].x - p[(i + 2) % 3].x) \* (p[i].x - p[(i + 2) % 3].x) + (p[i].y - p[(i + 2) % 3].y) \* (p[i].y - p[(i + 2) % 3].y));

if ( size < bsize)

{

nnid = (i + 1) % 3;

}

else

{

nnid = (i + 2) % 3;

size = bsize;

}

nnbuf.Add(new NearestNeighbor(p[i].id, p[nnid].id, size));

}

double zn = A[1] \* B[2] - A[2] \* B[1];

if (zn == 0)

{

//вироджений випадок. точки на одній прямій. діаграма з паралелельних ліній

for (int i = 0; i < 3; ++i)

{

if ((p[i].x - p[(i + 2) % 3].x >= 0 == p[(i + 1) % 3].x - p[(i + 2) % 3].x >= 0) && (p[i].y - p[(i + 2) % 3].y >= 0 == p[(i + 1) % 3].y - p[(i + 2) % 3].y >= 0))

{

int[] res = FindCrossBorder(A[i], B[i], C[i]);

VisualVor(new List<VorNode> { new VorNode(p[0].id, p[1].id, res[0], res[1], res[2], res[3]) });

vorbuf.Add(new VorNode(p[i].id, p[(i+1)%3].id, res[0], res[1], res[2], res[3]));

}

else

{

continue;

}

}

}

else

{

double buf = (-1.0) \* (C[1]\*B[2] - C[2]\*B[1]) / zn;

int xc = (int)buf;

buf = (-1.0) \* (A[1] \* C[2] - A[2] \* C[1]) / zn;

int yc = (int)buf;

bool up = true;

if (FindAngle(p[0].x, p[0].y, p[2].x, p[2].y, p[1].x, p[1].y) < Math.PI)

up = false;

for (int i = 0; i < 3; ++i)

{

int[] res = FindCrossBorder(A[i], B[i], C[i]);

if ((FindAngle(xc, yc, p[i].x, p[i].y, res[0], res[1]) < FindAngle(xc, yc, p[i].x, p[i].y, p[(i + 1) % 3].x, p[(i + 1) % 3].y)) == up)

{

if ((FindAngle(xc, yc, p[i].x, p[i].y, res[2], res[3]) < FindAngle(xc, yc, p[i].x, p[i].y, p[(i + 1) % 3].x, p[(i + 1) % 3].y)) == up)

{

VisualVor(new List<VorNode> { new VorNode(p[0].id, p[1].id, res[0], res[1], res[2], res[3]) });

vorbuf.Add(new VorNode(p[i].id, p[(i + 1) % 3].id, res[0], res[1], res[2], res[3]));

}

else

{

VisualVor(new List<VorNode> { new VorNode(p[0].id, p[1].id, res[0], res[1], xc, yc) });

vorbuf.Add(new VorNode(p[i].id, p[(i + 1) % 3].id, res[0], res[1], xc, yc));

}

}

else

{

if ((FindAngle(xc, yc, p[i].x, p[i].y, res[2], res[3]) < FindAngle(xc, yc, p[i].x, p[i].y, p[(i + 1) % 3].x, p[(i + 1) % 3].y)) == up)

{

VisualVor(new List<VorNode> { new VorNode(p[0].id, p[1].id, xc, yc, res[2], res[3]) });

vorbuf.Add(new VorNode(p[i].id, p[(i + 1) % 3].id, xc, yc, res[2], res[3]));

}

else

{

continue;

}

}

}

}

return new Result(p, nnbuf, vorbuf);

}

if (p.Count == 1)//такого ніколи не має бути

{

return new Result(new List<MyPoint>(), new List<NearestNeighbor>(), new List<VorNode>());

}

List<MyPoint> bufLeft = new List<MyPoint>();

for (int i = 0; i < p.Count / 2; ++i)

{

bufLeft.Add(p[i]);

}

List<MyPoint> bufRight = new List<MyPoint>();

for (int i = p.Count / 2; i < p.Count; ++i)

{

bufRight.Add(p[i]);

}

return Merger(FindAllNN(bufLeft), FindAllNN(bufRight));

}

Result Merger(Result res1, Result res2)//процедура злиття результатів рекурсії

{

List<VorNode> vor\_buf = new List<VorNode>();

List<int> used\_id = new List<int>();

//знайти верхню ліву опорну

int vlop\_id = res1.res\_p[0].id;

for (int i = 1; i < res1.res\_p.Count; ++i)

{

if (res1.res\_p[i].y > points[vlop\_id].y)

{

vlop\_id = res1.res\_p[i].id;

}

else

{

if (res1.res\_p[i].y == points[vlop\_id].y && res1.res\_p[i].x > points[vlop\_id].x)

{

vlop\_id = res1.res\_p[i].id;

}

}

}

//знайти нижню ліву опорну

int nlop\_id = res1.res\_p[0].id;

for (int i = 1; i < res1.res\_p.Count; ++i)

{

if (res1.res\_p[i].y < points[nlop\_id].y)

{

nlop\_id = res1.res\_p[i].id;

}

else

{

if (res1.res\_p[i].y == points[nlop\_id].y && res1.res\_p[i].x > points[nlop\_id].x)

{

nlop\_id = res1.res\_p[i].id;

}

}

}

//знайти нижню праву опорну

int nrop\_id = res2.res\_p[0].id;

for (int i = 1; i < res2.res\_p.Count; ++i)

{

if (res2.res\_p[i].y < points[nrop\_id].y)

{

nrop\_id = res2.res\_p[i].id;

}

else

{

if (res2.res\_p[i].y == points[nrop\_id].y && res2.res\_p[i].x < points[nrop\_id].x)

{

nrop\_id = res2.res\_p[i].id;

}

}

}

//знайти верхню праву опорну

int vrop\_id = res2.res\_p[0].id;

for (int i = 1; i < res2.res\_p.Count; ++i)

{

if (res2.res\_p[i].y > points[vrop\_id].y)

{

vrop\_id = res2.res\_p[i].id;

}

else

{

if (res2.res\_p[i].y == points[vrop\_id].y && res2.res\_p[i].x < points[vrop\_id].x)

{

vrop\_id = res2.res\_p[i].id;

}

}

}

//знаходжу початкові координати розділяючого ланцюга

int xstart = (points[vlop\_id].x + points[vrop\_id].x) / 2;

int ystart = HEIGHT;

int xp = (points[vlop\_id].x + points[vrop\_id].x) / 2;

int yp = (points[vlop\_id].y + points[vrop\_id].y) / 2;

double A;

double B;

double C;

double dx = (points[vlop\_id].x - points[vrop\_id].x);

double dy = (points[vlop\_id].y + points[vrop\_id].y);

if (dx == 0)

{

A = 0;

B = (-1.0);

C = yp;

}

else

{

if (dy == 0)

{

A = (-1.0);

B = 0;

C = xp;

}

else

{

double kab1 = dy / dx;

A = (-1.0) / kab1;

B = (-1.0);

C = (double)yp - (double)A \* xp;

}

}

int[] res = FindCrossBorder(A, B, C);

if (res[1] > res[3])

{

xstart = res[0];

ystart = res[1];

}

else

{

xstart = res[2];

ystart = res[3];

}

//будую розділяючий ланцюг

while (vlop\_id != nlop\_id && vrop\_id != nrop\_id)

{

//шукаю індекс найближчого сусіда до лівої опорної

int lnn\_index = 0;

for (int i = 0; i < res1.res\_nn.Count; ++i)

{

if (res1.res\_nn[i].p1\_id == vlop\_id)

{

lnn\_index = i;

break;

}

}

//шукаю індекс найближчого сусіда до правої опорної

int rnn\_index = 0;

for (int i = 0; i < res2.res\_nn.Count; ++i)

{

if (res2.res\_nn[i].p1\_id == vrop\_id)

{

rnn\_index = i;

break;

}

}

//порівнюю довжину до нової точки та заміняю сусіда, якщо треба

double op\_size = Math.Sqrt((points[vlop\_id].x - points[vrop\_id].x) \* (points[vlop\_id].x - points[vrop\_id].x) + (points[vlop\_id].y - points[vrop\_id].y) \* (points[vlop\_id].y - points[vrop\_id].y));

if (op\_size < res1.res\_nn[lnn\_index].size)

{

res1.res\_nn[lnn\_index].size = op\_size;

res1.res\_nn[lnn\_index].p2\_id = vrop\_id;

}

if (op\_size < res2.res\_nn[rnn\_index].size)

{

res2.res\_nn[rnn\_index].size = op\_size;

res2.res\_nn[rnn\_index].p2\_id = vlop\_id;

}

//провести перпендикуляр до відрізку [vlop ; vrop]

xp = (points[vlop\_id].x + points[vrop\_id].x) / 2;

yp = (points[vlop\_id].y + points[vrop\_id].y) / 2;

dx = (points[vlop\_id].x - points[vrop\_id].x);

dy = (points[vlop\_id].y + points[vrop\_id].y);

if (dx == 0)

{

A = 0;

B = (-1.0);

C = yp;

}

else

{

if (dy == 0)

{

A = (-1.0);

B = 0;

C = xp;

}

else

{

double kab1 = dy / dx;

A = (-1.0) / kab1;

B = (-1.0);

C = (double)yp - (double)A \* xp;

}

}

//знайти найвищу точку перетину його з ребром однієї з діаграм воронного

int xc = 99999;

int yc = 99999;

int vlop\_buf = vlop\_id;

int vrop\_buf = vrop\_id;

bool left = true;

if (vlop\_id != nlop\_id)

for (int i = 0; i < res1.res\_vor.Count; ++i)

{

if ((res1.res\_vor[i].p1\_id == vlop\_id && !used\_id.Contains(res1.res\_vor[i].p2\_id)) || (res1.res\_vor[i].p2\_id == vlop\_id && !used\_id.Contains(res1.res\_vor[i].p1\_id)))

{

dx = (res1.res\_vor[i].x1 - res1.res\_vor[i].x2) / 2;

dy = (res1.res\_vor[i].y1 - res1.res\_vor[i].y2) / 2;

double Ab;

double Bb;

double Cb;

if (dx == 0)

{

Ab = (-1.0);

Bb = 0;

Cb = res1.res\_vor[i].x1;

}

else

{

if (dy == 0)

{

Ab = 0;

Bb = (-1.0);

Cb = res1.res\_vor[i].y1;

}

else

{

Ab = 1.0 / (res1.res\_vor[i].x2 - res1.res\_vor[i].x1);

Bb = (-1.0) / (res1.res\_vor[i].y2 - res1.res\_vor[i].y1);

Cb = res1.res\_vor[i].y1 / (res1.res\_vor[i].y2 - res1.res\_vor[i].y1) - res1.res\_vor[i].x1 / (res1.res\_vor[i].x2 - res1.res\_vor[i].x1);

}

}

int bufxc = FindCrossX(A, B, C, Ab, Bb, Cb);

int bufyc = FindCrossY(A, B, C, Ab, Bb, Cb);

if (Math.Sqrt((xstart - bufxc)\* (xstart - bufxc)+(ystart-bufyc)\* (ystart - bufyc)) < Math.Sqrt(Math.Abs((xstart - xc) \* (xstart - xc) + (ystart - yc) \* (ystart - yc))))

{

xc = bufxc;

yc = bufyc;

if (res1.res\_vor[i].p1\_id != vlop\_id)

{

used\_id.Add(vlop\_buf);

vlop\_buf = res1.res\_vor[i].p1\_id;

}

else

{

used\_id.Add(vlop\_buf);

vlop\_buf = res1.res\_vor[i].p2\_id;

}

}

}

}

if(vrop\_id != nrop\_id)

for (int i = 0; i < res2.res\_vor.Count; ++i)

{

if ((res2.res\_vor[i].p1\_id == vrop\_id && !used\_id.Contains(res2.res\_vor[i].p2\_id)) || (res2.res\_vor[i].p2\_id == vrop\_id && !used\_id.Contains(res2.res\_vor[i].p1\_id)))

{

dx = (res2.res\_vor[i].x1 - res2.res\_vor[i].x2) / 2;

dy = (res2.res\_vor[i].y1 - res2.res\_vor[i].y2) / 2;

double Ab;

double Bb;

double Cb;

if (dx == 0)

{

Ab = (-1.0);

Bb = 0;

Cb = res2.res\_vor[i].x1;

}

else

{

if (dy == 0)

{

Ab = 0;

Bb = (-1.0);

Cb = res2.res\_vor[i].y1;

}

else

{

Ab = 1.0 / (res2.res\_vor[i].x2 - res2.res\_vor[i].x1);

Bb = (-1.0) / (res2.res\_vor[i].y2 - res2.res\_vor[i].y1);

Cb = res2.res\_vor[i].y1 / (res2.res\_vor[i].y2 - res2.res\_vor[i].y1) - res2.res\_vor[i].x1 / (res2.res\_vor[i].x2 - res2.res\_vor[i].x1);

}

}

int bufxc = FindCrossX(A, B, C, Ab, Bb, Cb);

int bufyc = FindCrossY(A, B, C, Ab, Bb, Cb);

if (Math.Sqrt((xstart - bufxc) \* (xstart - bufxc) + (ystart - bufyc) \* (ystart - bufyc)) < Math.Sqrt(Math.Abs((xstart - xc) \* (xstart - xc) + (ystart - yc) \* (ystart - yc))))

{

xc = bufxc;

yc = bufyc;

if (res2.res\_vor[i].p1\_id != vrop\_id)

{

used\_id.Add(vrop\_buf);

vrop\_buf = res2.res\_vor[i].p1\_id;

left = false;

}

else

{

used\_id.Add(vrop\_buf);

vrop\_buf = res2.res\_vor[i].p2\_id;

left = false;

}

}

}

}

if (xc == 99999 || yc == 99999)

{

break;

}

//додати нове ребро між опорними точками

vor\_buf.Add(new VorNode(vlop\_id, vrop\_id, xstart, ystart, xc, yc));

//обрізати старе ребро й поміняти опорну точку

if (left)

{

for (int i = 0; i < res1.res\_vor.Count; ++i)

{

if ((res1.res\_vor[i].p1\_id == vlop\_id && res1.res\_vor[i].p2\_id == vlop\_buf) || (res1.res\_vor[i].p1\_id == vlop\_buf && res1.res\_vor[i].p2\_id == vlop\_id))

{

if (res1.res\_vor[i].y1 > res1.res\_vor[i].y2)

{

res1.res\_vor[i].y1 = yc;

res1.res\_vor[i].x1 = xc;

}

else

{

res1.res\_vor[i].y2 = yc;

res1.res\_vor[i].x2 = xc;

}

used\_id.Add(vlop\_id);

vlop\_id = vlop\_buf;

break;

}

}

}

else

{

for (int i = 0; i < res2.res\_vor.Count; ++i)

{

if ((res2.res\_vor[i].p1\_id == vrop\_id && res2.res\_vor[i].p2\_id == vrop\_buf) || (res2.res\_vor[i].p1\_id == vrop\_buf && res2.res\_vor[i].p2\_id == vrop\_id))

{

if (res2.res\_vor[i].y1 > res2.res\_vor[i].y2)

{

res2.res\_vor[i].y1 = yc;

res2.res\_vor[i].x1 = xc;

}

else

{

res2.res\_vor[i].y2 = yc;

res2.res\_vor[i].x2 = xc;

}

used\_id.Add(vrop\_id);

vrop\_id = vrop\_buf;

break;

}

}

}

//поміняти початкову точку перпендикуляра розділяючого ланцюга

xstart = xc;

ystart = yc;

}//теперішні опорні точки співпали з нижніми

//для нижніх опорних точок знаходжу перетин перпендикуляра з межами

if (xstart > 0 && ystart > 0)

{

xp = (points[nlop\_id].x + points[nrop\_id].x) / 2;

yp = (points[nlop\_id].y + points[nrop\_id].y) / 2;

dx = (points[nlop\_id].x - points[nrop\_id].x);

dy = (points[nlop\_id].y + points[nrop\_id].y);

if (dx == 0)

{

A = 0;

B = (-1.0);

C = yp;

}

else

{

if (dy == 0)

{

A = (-1.0);

B = 0;

C = xp;

}

else

{

double kab1 = dy / dx;

A = (-1.0) / kab1;

B = (-1.0);

C = (double)yp - (double)A \* xp;

}

}

res = FindCrossBorder(A, B, C);

if (res[1] < res[3])

{

vor\_buf.Add(new VorNode(nlop\_id, nrop\_id, xstart, ystart, res[0], res[1]));

}

else

{

vor\_buf.Add(new VorNode(nlop\_id, nrop\_id, xstart, ystart, res[2], res[3]));

}

}

//об'єдную й повертаю результат

res1.res\_p.AddRange(res2.res\_p);

res1.res\_nn.AddRange(res2.res\_nn);

res1.res\_vor.AddRange(res2.res\_vor);

res1.res\_vor.AddRange(vor\_buf);

return new Result(res1.res\_p, res1.res\_nn, res1.res\_vor);

}

void VisualPoints()

{

int half = RADIUS / 2;

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

{

g.FillEllipse(brush, points[i].x - half, points[i].y - half, RADIUS, RADIUS);

}

}

void VisualNN(List<NearestNeighbor> nearnei)

{

for (int i = 0; i < nearnei.Count; ++i)

{

g.DrawLine(arrow, points[nearnei[i].p1\_id].x, points[nearnei[i].p1\_id].y, points[nearnei[i].p2\_id].x, points[nearnei[i].p2\_id].y);

}

}

void VisualVor(List<VorNode> v)

{

if (visvor)

for (int i = 0; i < v.Count; ++i)

{

g.DrawLine(line, v[i].x1, v[i].y1, v[i].x2, v[i].y2);

}

}

}

}