

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра системного аналізу та обчислювальної математики

(повне найменування кафедри)

Пояснювальна записка

до дипломної роботи

бакалавра

(ступінь вищої освіти)

на тему Візуалізація спорідненості ліній соняшнику за фенотипом

(назва теми)

Виконав: студент 4 курсу, групи 810сп

Спеціальності 124 Системний аналіз

(код і найменування спеціальності)

Освітня програма (спеціалізація)

«Інтелектуальні технології та прийняття
рішень в складних системах»

Руслан СІРЕНКО

(ПРІЗВИЩЕ та ініціали)

Керівник Еліна ТЕРЕЩЕНКО

(ПРІЗВИЩЕ та ініціали)

Рецензент Наталя МИРОНОВА

(ПРІЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Інститут, факультет Факультет комп'ютерних наук і технологій
Кафедра Кафедра системного аналізу та обчислювальної математики
Ступінь вищої освіти бакалавр
Спеціальність 124 Системний аналіз
(код і найменування)
Освітня програма (спеціалізація) «Інтелектуальні технології та прийняття рішень в складних системах»
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри Еліна ТЕРЕЩЕНКО
«09» червня 2023 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА

Сіренку Руслану Володимировичу

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема роботи Візуалізація спорідненості ліній соняшнику за фенотипом,
Visualization of sunflower linear inbreeding by phenotype

керівник роботи Терещенко Елена Валентинівна

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від «18» квітня 2023 року №121

2. Строк подання студентом роботи 7 червня 2023

3. Вихідні дані до роботи статистичні дані польового експерименту, що агреговані в електронних таблицях MS Excel та надані лабораторією генетичних досліджень Інституту олійних культур НААН України

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вибір методу аналізу спорідненості ліній соняшнику та представлення родинних зв'язків; розробка програмного застосунку для аналізу на візуалізації спорідненості ліній соняшника за фенотипом; проведення експериментів на аналіз отриманих результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) презентація

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Прийняв виконане завдання
Розділ 1, Розділ 2	Терещенко Е.В., в.о. завідувача кафедри	21 лютого 2023	06 червня 2023
Нормоконтроль	Широкопад Д.В., ст. викладач	07 червня 2023	

7. Дата видачі завдання «21» лютого 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2 тиждень	Розділ 1
3	Вибір методів класифікації та візуалізації.	3 тиждень	Розділ 1
4	Роботою з Excel та базою даних.	4 тиждень	Розділ 2
5	Робота з алгоритмами аналізу.	5 тиждень	Розділ 2
6	Реалізація візуалізації даних у вигляді графа.	6 тиждень	Розділ 2
7	Проведення експериментів з візуалізацією графа.	7 тиждень	Розділ 2
8	Оформлення пояснювальної записки.	7-9 тиждень	Додатки
9	Нормоконтроль та рецензування.	9 тиждень	
10	Захист дипломної роботи.	10 тиждень	

Студент

(підпис)

(прізвище та ініціали)

Керівник роботи

(підпис)

(прізвище та ініціали)

РЕФЕРАТ

Дипломна робота: 57 с., 19 рис., 8 джерел, 1 додаток.

Об'єкт дослідження – методи представлення родинних зв'язків ліній соняшнику.

Предмет дослідження – представлення родинних зв'язків ліній соняшнику у вигляді графа.

Мета роботи – створення системи для дослідження спорідненості ліній соняшнику за фенотипом.

Методи дослідження – методи класифікації, методи теорії графів, методи теорії відношень.

У ході роботи було поставлено та вирішено такі задачі:

1. Аналіз задач предметної області , які розв'язує дослідник при аналізі спорідненості ліній соняшнику за фенотипом.
2. Вибір методу аналізу спорідненості ліній соняшнику та представлення родинних зв'язків.
3. Розробка структурної схеми системи для дослідження спорідненості ліній соняшнику за фенотипом.
4. Вибір інструментів розробки, програмна реалізація та тестування.
5. Проведення експериментів та аналіз отриманих результатів.

Результатом роботи є функціонуючий програмний застосунок, що дозволяє досліднику візуалізувати родинні зв'язки ліній соняшнику за статистичними даними польового експерименту, що агреговані в електронних таблицях MS Excel. Програмний застосунок реалізований на мові програмування C# в середовищі Visual Studio зі застосуванням бібліотек ExcelDataReader та Microsoft.Msagl та технологій ADO NET.

СПОРІДНЕНІСТЬ ЛІНІЙ СОНЯШНИКУ, ГРАФ, ВІЗУАЛІЗАЦІЯ, БАЗА ДАНИХ, C#, ADO.NET, EXCEL DATA READER, MSAGL

ЗМІСТ

Завдання	2
Реферат	4
Вступ.....	7
1 Теоретична частина.....	8
1.1 Аналіз предметної області.....	8
1.2 Методи класифікації для аналізу генетичної спорідненості ліній соняшнику	8
1.3 Теорія графів та її застосування для візуалізації родинних зв'язків ліній соняшнику	9
1.3.1 Поняття графа	9
1.3.2 Поняття візуалізації графів	10
1.3.3 Естетика візуалізації графів	11
1.3.4 Силкові алгоритми для візуалізації графів (Force-Directed).....	12
1.3.5 Енергетичні алгоритми (Energy-Based)	14
1.3.6 Алгоритми зменшення розмірності (Dimension Reduction).....	16
1.4 Розробка структурної схеми.....	18
1.5 Технічні засоби.....	19
1.5.1 Мова програмування C#	19
1.5.2 Інтерфейс програмування додатків Windows Forms	21
1.5.3 Середовище розробки Visual Studio	22
1.5.4 Excel Data Reader.....	23
1.5.5 ADO.NET	23
1.5.6 MSAGL	24
1.5 Аналоги	25
2 Практична частина	26
2.1 Постановка задачі.....	26

2.2 Робота з MS Excel та базою даних	27
2.3 Візуалізація даних	31
2.4 Експерименти з візуалізацією графа	34
Висновки	36
Перелік посилань.....	37
Додаток А. Програмний код	38

ВСТУП

Соняшник є одним з найважливіших культурних рослин, який відіграє важливу роль в сільському господарстві та харчовій промисловості. Розуміння спорідненості ліній соняшнику за фенотипом має велике значення для покращення селекційних програм та вибору найкращих генетичних ресурсів для покращення урожайності та якості соняшнику.

Спорідненість ліній соняшнику визначається на основі генетичної подібності, що може бути виявлена за фенотипом. Фенотип включає в себе зовнішній вигляд, морфологічні характеристики та інші спостережні ознаки рослини. Аналіз спорідненості соняшнику за фенотипом може допомогти встановити зв'язок між генетичними характеристиками та природними ознаками рослини.

Однак, визначення спорідненості соняшнику за фенотипом вимагає значної кількості даних та спеціалізованого аналізу. Традиційні методи оцінки спорідненості, такі як оцінка фенотипової схожості на основі спостережень або вимірювання морфологічних параметрів, можуть бути обмежені в своїй точності та об'єктивності. Тому важливим завданням є розробка нових методів, що дозволяють більш точно та ефективно визначати спорідненість ліній соняшнику за фенотипом.

Однією з перспективних технологій, яка може допомогти вирішити цю проблему, є візуалізація даних. Візуалізація спорідненості ліній соняшнику за фенотипом включає в себе використання графічних методів та інструментів для представлення даних про фенотип рослин. Це може допомогти дослідникам виявити взаємозв'язки та патерни, які не завжди очевидні за допомогою традиційних методів аналізу.

1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Аналіз предметної області

Для аналізу спорідненості ліній сояшнику за фенотипом використовуються різні методи. Одним з них є метод класифікації, який дозволяє групувати лінії сояшнику за подібними ознаками. Цей підхід допомагає виявити групи ліній, що мають спільні генетичні особливості, і встановити їх спорідненість [1].

Також використовується теорія графів для візуалізації родинних зв'язків між лініями сояшнику. У цьому підході лінії представляються у вигляді вузлів графа, а генетичні зв'язки між ними — у вигляді ребер. Це дозволяє дослідникам більш детально аналізувати та розуміти структуру спорідненості ліній.

Загальноприйнятою практикою також є використання методів теорії відношень для вивчення родинних зв'язків у генетиці сояшнику. Ці методи дозволяють встановити ступінь спорідненості між різними лініями на основі аналізу їх фенотипів та генетичних характеристик [2].

1.2 Методи класифікації для аналізу генетичної спорідненості ліній сояшнику

Методи класифікації є потужним інструментом у генетиці для аналізу генетичної подібності ліній сояшнику. Вони дозволяють групувати лінії за подібними ознаками та встановлювати їх спорідненість на основі генетичних характеристик.

Один з найпоширеніших підходів до класифікації ліній соняшнику є кластерний аналіз. Цей метод дозволяє групувати лінії в класи або кластери на основі схожості їх фенотипів та генетичних властивостей. Кластерний аналіз допомагає ідентифікувати групи ліній, що мають спільний генетичний початок, та визначити ступінь спорідненості між ними [3].

Крім кластерного аналізу, існують інші методи класифікації, такі як дискримінантний аналіз, регресійна модель, штучні нейронні мережі тощо. Вони базуються на математичних алгоритмах та статистичних підходах для визначення критеріїв класифікації та розподілу ліній за певними категоріями.

Застосування методів класифікації у генетиці соняшнику дозволяє вченим систематизувати та структурувати генетичну інформацію, що стосується ліній соняшнику. Це дає можливість краще розуміти генетичні зв'язки та спорідненість між лініями, що сприяє розвитку селекції та покращенню сортів соняшнику.

1.3 Теорія графів та її застосування для візуалізації родинних зв'язків ліній соняшнику

1.3.1 Поняття графа

Граф – це сукупність об'єктів довільної природи та зв'язки, що сполучають будь-які пари цих об'єктів (рис. 1.1):

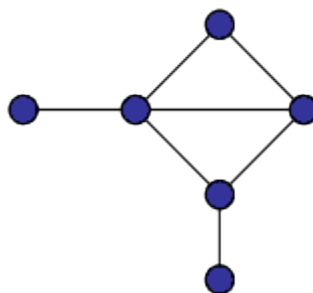


Рисунок 1.1 – Граф

Об'єкти розглядаються як вершини, або вузли графу, а зв'язки – як дуги, або ребра. Граф, який містить тільки ребра називається неорієнтованим, який містить тільки дуги — орієнтованим. Граф, що має як ребра так і дуги, називається мішаним. Іноді є потреба пару вершин з'єднати більше, ніж одним ребром. Ребра чи дуги одного напрямку, які з'єднують ту саму пару вершин, називаються кратними ребрами. Дуга чи ребро, що сполучає вершину саму зі собою називається петлею. Граф без кратних дуг і петель називається простим. Вершини сполучені ребром чи дугою називаються суміжними, також називаються суміжними ребра, що мають спільну вершину. Ребро (чи дуга) і її вершина називаються інцидентними.

Тобто, графи — це метод візуальної ілюстрації даних та відношень між ними. Мета графів — представити занадто численні, або складні дані, для їх адекватного опису в тексті, або алгоритмі. Забезпечення чіткості та коректність опису даних у графах забезпечують ефективність використання графів [4].

1.3.2 Поняття візуалізації графів

Візуалізація графу допомагає краще зрозуміти структуру зв'язків між вершинами графу, ніж простий перегляд даних у табличній формі. Завдання візуалізації полягає в тому, щоб подати граф у простому та зрозумілому вигляді, що підкреслює його структуру та топологічні властивості. Існують різні методи візуалізації графів, багато з яких спираються на певну топологію: планарні графи, дерева. Відповідно різні алгоритми призначені для різних видів графів і можуть накладати свої особливі обмеження на розташування вершин, на вигляд ребер (прямі, криві або ортогональні лінії) [5].

При візуалізації графів потрібно враховувати різні правила та принцип. Наприклад, гештальт-принцип близькості: вузли, які розташовані близько один до одного, інтерпретуватимуться користувачем як справжній зв'язок незалежно від того, чи він існує. Це означає, що розташування вузлів та ребер сильно впливає на те, як користувач сприймає взаємозв'язки на зображенні.

Хороша візуалізація може показати, чи є в графі кластери, мости, допоможе визначити чи є він однорідним, чи можливо він зливається в одну грудку до якогось головного центру тяжіння [6].

Незважаючи на те, що алгоритми візуалізації графів в основному розроблялися лише як інструменти для отримання зображень, їх можна застосовувати як методи зниження розмірності. Сам граф, якщо його зобразити матрицею суміжності — це дані у просторі високої розмірності, а при його відтворенні ми отримуємо дві координати кожної вершини (іноді три чи більше, але це виняток). Близькість вершин на візуалізації часто виражає схожість за властивостями.

1.3.3 Естетика візуалізації графів

Класичні підходи візуалізації графів застосовують набір правил, які створюють естетичні графіки. У цьому контексті естетика пов'язана з поліпшенням читабельності графа для користувача. Правильні естетичні властивості повинні забезпечувати ефективніше відображення графа та дозволяти користувачеві легко сприймати топологічну структуру графа [7].

Дотримання всіх цих естетичних правил одразу неможливе через їхній конкурентний характер, оскільки для досягнення одного часто потрібно зламати інший.

Деякі з них :

- Мінімум перетинів ребер. Покращує читабельність та допомагає користувачу в пошуку зв'язків. Перетин ребер також може приховати важливу інформацію та зробити графік менш доступним для користувача.
- Суміжні вершини позиціонувати близько одна до одної, несуміжні далеко. Зробити пов'язані вершини ближче один до одного — прямий та логічний спосіб висловити основні властивості графів.
- Групи вершин групуються у кластери. Якщо існують безліч вузлів, які сильніше пов'язані один з одним, ніж з рештою вершин графа, вони утворюють “групу” і на зображенні повинні виглядати як щільний кластер.
- Мінімум накладень вершин та ребер між собою. Якщо ми не зможемо розібрати один у певному місці об'єкт або декілька, то візуалізація погано зрозуміла для людини.
- Розміщення вершин та ребер рівномірно. Ця умова буває корисною, якщо властивості графа не дозволяють розглянути хоч якусь структуру в іншому випадку. Наприклад, якщо у нас весь граф — це один щільний кластер, то краще розмазати його по картинці, щоб розглянути хоча б нерівномірність зв'язків.

1.3.4 Силкові алгоритми для візуалізації графів (Force-Directed)

Ці методи відносяться до класу алгоритмів візуалізації графів естетично привабливим способом. Їх мета — розмістити вузли графа в двовимірному або тривимірному просторі так, щоб усі ребра мали приблизно однакову довжину, та звести до мінімуму число перетинів ребер між собою. Ці методи використовують симуляцію фізичних сил. Вершини подаються як заряджені частинки, які відштовхують одна одну, а ребра — як пружні струни,

які стягують суміжні вершини. Після цього моделюється рух вершин у такій системі, доки не встановиться стійкий стан. В інших підходах намагаються описати потенційну енергію такої системи та знайти становище вершин, яке відповідатиме мінімуму [8].

Пружинно-електричний алгоритм Ідса. Традиційний силовий алгоритм візуалізації графів розробив і опублікував Пітер Ідс у 1984 року. У цьому алгоритмі граф представляється у вигляді фізичної системи (рис. 1.2), в якій вершини графа відповідають зарядженим частинкам, а ребра — пружини між ними, в цій системі діють, як сили тяжіння, так і відштовхування.

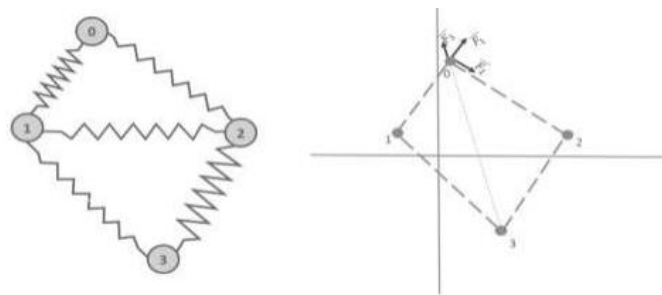


Рисунок 1.2 – Фізична модель графа

Система поміщається у випадкове початкове положення та залишає систему для досягнення стабільного стану, коли сила на кожному вузлі дорівнює нулю. У результаті експериментів Ідс вирішив використовувати пружини логарифмічної міцності, стверджуючи, що пружини лінійної міцності занадто сильні. Зв'язані вузли притягуються один до одного, тоді як всі інші вузли, змодельовані як електричні заряди, відштовхуються.

Алгоритм Фрюхтермана-Рейнгольда. Перша адаптація алгоритму Ідса, яка була створена для швидкості роботи та простоти використання, була розроблена Фрюхтерманом і Рейнгольдом. При цьому враховувалися властивості з'єднаних вузлів, що знаходяться близько один до одного (але не надто близько). Вони прагнули створити алгоритм, який відповідав би принципам рівномірного розподілу вузлів, малої кількості перетинів ребер,

рівномірної довжини ребер та симетрії графу. Для графів з менш ніж 100 вершин алгоритм гарантує виконання цих принципів.

Алгоритм поглиблення графа. Фрік, Людвіг та Мельдау хотіли створити алгоритм, який відповідає принципам візуалізації для більших і складніших графів, але з "інтерактивною швидкістю". Точні принципи не визначені явно, але вони схожі на концепції інших силових алгоритмів, а система оцінюється за принципом найменшого перетину ребер, рівних довжин ребер та рівномірного розподілу вузлів.

Алгоритм використовує сили, подібні до тих, що використовували Фрухтерман і Рейнгольд, а також гравітаційні сили, які притягують вузли до барицентру їхніх сусідів. Фрухтерман та Рейнгольд вважали, що покращений графік охолодження міг би значно покращити їхній алгоритм, тому Фрік та ін. реалізували адаптивний алгоритм з урахуванням локальних і глобальних температур. Чим нижча локальна температура вузла, тим ближче він повинен бути до свого кінцевого положення і тим на меншу відстань він може переміститися в кожній ітерації. Локальна температура вузла залежить від його температури на попередній ітерації, а також від того, чи коливається вузол між позиціями, чи є він частиною підграфа, що обертається. Глобальна температура є середнім значенням усіх локальних температур, і ітерації зупиняються, коли глобальна температура досягає заданого значення або коли пройшла фіксована кількість ітерацій.

1.3.5 Енергетичні алгоритми (Energy-Based)

Алгоритми, що базуються на енергії, розглядають компоновання як мінімум задачі оптимізації, в якій функція енергії кодує бажані властивості графа. Рішення, які призводять до локальних, а не глобальних мінімумів, не є

оптимальними рішеннянями і в цьому випадку не приведуть до оптимального компонування [8].

Деякі графи, створені за допомогою цих методів, можуть тимчасово переходити на конфігурації з вищою енергією, щоб досягти глобального мінімуму. Два основні методи, що використовують цей підхід, Камади-Кавай та Девідсона-Харела.

Алгоритм Камади-Кавай. У алгоритмі Камади-Кавай вершини розміщені таким чином, щоб їх візуальна відстань у межах графу була пропорційною їх теоретичній графічній відстані. Ключовою ідеєю алгоритму є використання пружинної моделі таким чином, щоб мінімізувати енергію системи.

У цій системі найбільш важливим є принцип симетрії, а функція вартості являє собою "ступінь дисбалансу" (відсутність симетрії) та ґрунтується на законі Гука (де сила, що діє на пружину, лінійна і пропорційна її зміщенню від її природної довжини). Оптимізація вимагає розв'язання рівнянь у приватних похідних на основі суми квадратів різниці між евклідовими і теоретико-графовими відстанями пар вузлів. На кожній ітерації переміщується тільки один вузол, і мінімізація виконується за допомогою ітерації Ньютон-Рафсона, результуючі макети є симетричними з кількома перетинами ребер.

Алгоритм Девідсона-Харела. Алгоритм імітації відпалювання. Відпалювання — це процес повільного охолодження рідини, так що вона утворює кристалічну структуру з мінімальною енергією. Їхній підхід полягає в побудові графів, які відповідають принципам відображення графів з рівномірно розподіленими вузлами, рівномірними довжинами ребер і мінімальним перетином ребер. Вони створили явну функцію вартості, що складається з частин, які керують розподілом вузлів, межами (відстанню від краю доступної області системи), довжинами, перетинами та сусіднім розташуванням ребер, які можуть бути зважені для визначення пріоритету конкретної цілі. Алгоритм починається з початкової конфігурації та

встановлення глобальної температури. На кожній ітерації переміщується тільки один вузол. Відстань, на яку може переміщатися вузол, зменшується з кожною ітерацією, а температура перераховується на кожному кроці. Процес триває доти, доки не буде виконано умову завершення, таку як повторення фіксованого числа етапів.

1.3.6 Алгоритми зменшення розмірності (Dimension Reduction)

Сутність алгоритмів у перетворенні даних із простору високої розмірності в простір низької розмірності таким чином, щоб представлення низької розмірності зберігало деякі значущі властивості вихідних даних, ідеально близькі до їх внутрішньої розмірності. Граф задається матрицею суміжності, тобто квадратною матрицею $N \times N$, де N — кількість вершин. Це можна інтерпретувати як N об'єктів у просторі розмірності N . Інший підхід ґрунтується на тому, щоб розрахувати теоретичні відстані між вершинами, ґрунтуючись на вагах ребер та знанні локальної топології, а потім спробувати зберегти співвідношення між цими відстанями при переході до простору меншої розмірності [8].

Більшість методів зменшення розмірності, що використовуються для компонування графів, використовують теоретико-графову відстань між парою вузлів в якості інформації, яка повинна бути збережена.

Метод головних компонент — це один з поширених методів обробки даних, метою якого є зменшення розмірності досліджуваних даних з мінімальними втратами корисної інформації. Винайдений Карлом Пірсоном у 1901 році та доповнений і розширений Гарольдом Хотелінгом в 1933 р. Цей метод полягає у пошуку додаткової системи прямих або площин простору змінних, які будуть відображати максимальну дисперсію вихідних значень —

така пряма або площина і називається головною компонентою. Основна лінійна техніка зменшення розмірності, метод головних компонент, здійснює лінійне відображення даних в менш вимірний простір таким чином, що максимізується дисперсія даних у маловимірному представленні.

Метод Харел-Корен. Один із найшвидших доступних методів компонування. Покращує минулий спосіб оптимізацією підпростору. Мета полягає в тому, щоб знайти підпростір низької розмірності, що може відображати граф з короткою довжиною ребер та рівномірним розподілом вузлів.

Побудування спектрального графа також зменшує розміри за рахунок використання власних векторів, пов'язаних із двома найбільшими власними значеннями теоретико-графової матриці відстаней, знайденої за допомогою спектрального розкладання. Цей метод швидко працює за рахунок використання тільки вибірки вузлів для обчислення теоретико-графових відстаней. Намагається реалізувати принципи побудови графів у просторі вищої розмірності.

1.3.7 Візуалізація на основі обмежень та особливостей (Feature-Based Layout)

Методи, засновані на обмеженнях, часто використовують алгоритми, орієнтовані на силу. Силкові алгоритми та їхні похідні використовують естетичні принципи як мотивацію під час їхньої реалізації та для їхнього оцінювання. Методи, засновані на обмеженнях, накладають визначені користувачем критерії розміщення на всі вузли або їх вибірку на додаток до використання алгоритму компонування [8].

Зазвичай дані приходять із реального світу, де в нас є не тільки інформація про суміжність вершин. Вершини є якимись реальними об'єктами

зі своїми властивостями. Згадуючи про це, ми можемо використовувати властивості вершин для відображення їх на площині. Для цього можна використовувати будь-які підходи, які зазвичай застосовують для табличних даних. Це вже згадані вище методи, або можна побудувати просту діаграму розсіювання для пар ознак і додавати ребра вже поверх отриманого результату.

1.4 Розробка структурної схеми

Структурна схема – це графічне зображення компонентів і зв'язків системи або процесу, що допомагає зрозуміти його функціонування та взаємозв'язки між його елементами.

Структурна схема включає компоненти системи (які можуть бути представлені у вигляді блоків, боксів або символів) та зв'язки між ними (які можуть бути показані у вигляді стрілок, ліній або ліній з певними позначеннями). Компоненти можуть включати елементи управління, бібліотеки, технології, вхідні/вихідні дані та інші.

Структурна схема допомагає візуалізувати архітектуру системи, виявити її основні елементи та зв'язки між ними. Вона також може допомогти ідентифікувати можливі проблеми, які можуть виникнути в системі, та використовуватися для аналізу та оптимізації її роботи.

У побудованій структурній схемі (рис 1.3) представлені основні етапи роботи програми для дослідження спорідненості ліній соняшнику за фенотипом, такі як завантаження даних з Excel, взаємодія з базою даних, аналіз даних за допомогою методів класифікації, налаштування вхідних параметрів, побудова графа родинних зв'язків, візуалізація результатів.

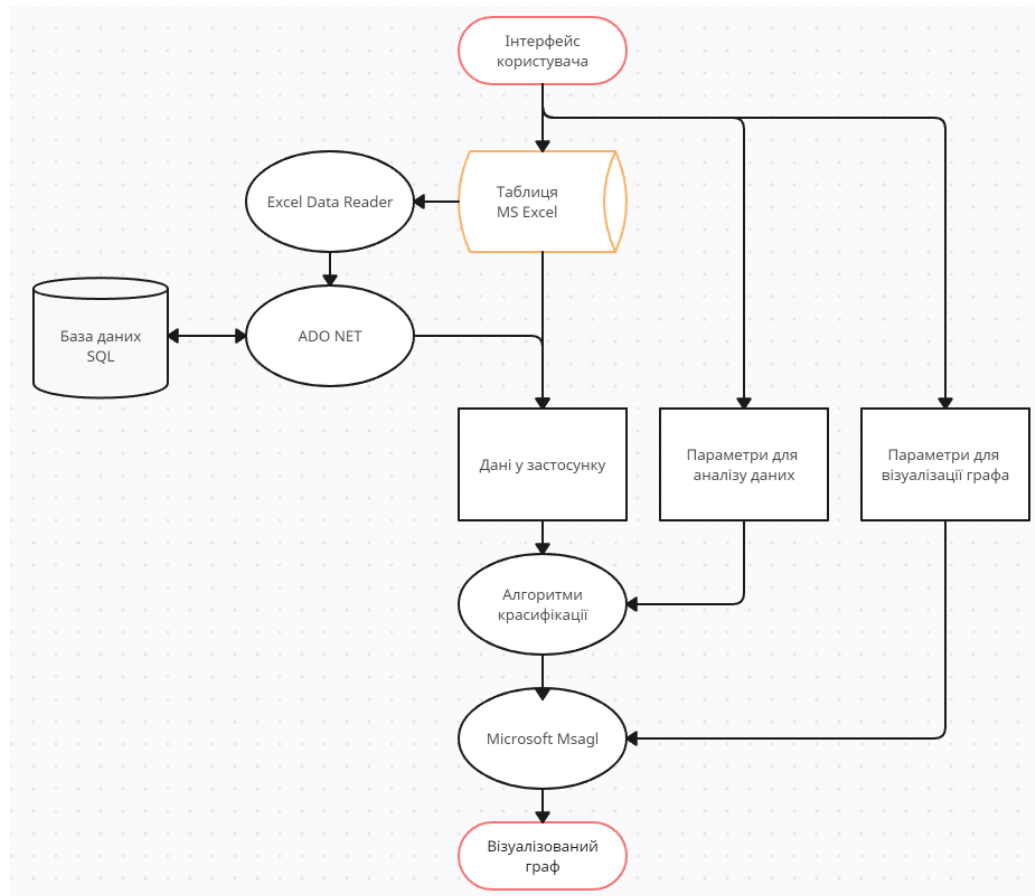


Рисунок 1.3 – Структурна схема системи

1.5 Технічні засоби

1.5.1 Мова програмування C#

Мова програмування C# є однією з популярних мов програмування, розроблених компанією Microsoft. Вона входить до складу платформи .NET і має широке застосування для розробки різноманітних програмних продуктів, включаючи десктопні додатки, веб-додатки, мобільні додатки та ігри.

C# є об'єктно-орієнтованою мовою програмування зі строгою типізацією. Підтримує такі концепції як спадкування, поліморфізм, інкапсуляція та абстракція. Вона також підтримує ряд інших мовних функцій, включаючи делегати, події, лямбда-вирази, LINQ та інші.

Основні переваги мови програмування C# включають:

- Простота вивчення: C# має простий і зрозумілий синтаксис, який легко вивчити навіть новачкам у програмуванні. Вона надає зручні конструкції для створення структурованого коду.
- Широкі можливості: C# має багатий набір бібліотек і фреймворків, що робить його потужним інструментом для розробки різноманітних програмних продуктів. Вона підтримує різні галузі розробки, включаючи веб-розробку, мобільну розробку, ігрову розробку та багато іншого.
- Інтеграція з .NET: C# є частиною платформи .NET, що відкриває широкі можливості для використання різноманітних бібліотек, фреймворків і сервісів, які розроблені для .NET. Це спрощує роботу з різними аспектами розробки, такими як робота з базами даних, взаємодія з мережею, криптографія та багато іншого.
- Кросплатформеність: Завдяки платформі .NET Core, C# може бути використаний для розробки кросплатформових додатків, які можуть працювати на різних операційних системах, таких як Windows, macOS і Linux.
- Підтримка розробки у середовищі Visual Studio: Microsoft надає потужне та зручне середовище розробки Visual Studio для програмістів, що дозволяє ефективно створювати, налагоджувати та управляти проектами на C#.

Мова програмування C# є потужним інструментом для розробки різноманітних програмних продуктів. Вона поєднує в собі простоту вивчення, широкі можливості, інтеграцію з .NET та підтримку розробки у зручному середовищі Visual Studio, що робить її популярним вибором серед програмістів.

1.5.2 Інтерфейс програмування додатків Windows Forms

Windows Forms є одним з фреймворків розробки графічних інтерфейсів користувача (GUI) у середовищі .NET. Він надає зручні засоби для створення десктопних додатків для операційних систем Windows. WinForms базується на мові програмування C# або інших мовах .NET, таких як VB.NET.

Основні переваги використання WinForms включають:

- Простота розробки: WinForms має просту архітектуру та інтуїтивний інтерфейс, що робить розробку десктопних додатків швидкою та простою. Розробнику не потрібно писати складний код для створення базових елементів управління, таких як кнопки, тексти, списки тощо.
- Багатий набір елементів управління: WinForms надає широкий вибір готових елементів управління, таких як кнопки, тексти, списки, таблиці, вкладки тощо. Це дозволяє розробнику швидко створювати різноманітні інтерфейси для своїх додатків.
- Легка інтеграція з .NET: WinForms безперервно інтегрується з .NET Framework, що дозволяє використовувати багато функціональних можливостей, що надаються .NET, таких як робота з базами даних, робота з мережею, криптографія та багато іншого.
- Сумісність з попередніми версіями Windows: WinForms підтримує сумісність з попередніми версіями операційної системи Windows, що дозволяє розробникам створювати додатки, які працюють на широкому колі різних версій Windows.

Всі ці переваги роблять WinForms популярним вибором для розробки десктопних додатків у середовищі .NET. Використання WinForms дозволяє швидко створювати функціональні та зручні інтерфейси користувача для додатків, що працюють під управлінням операційної системи Windows.

1.5.3 Середовище розробки Visual Studio

Visual Studio є інтегрованим середовищем розробки, розробленим компанією Microsoft. Це потужний інструмент для створення різноманітних програмних продуктів на базі платформи .NET, включаючи додатки для Windows, веб-додатки та мобільні додатки.

Основні переваги використання Visual Studio:

- Зручне середовище розробки: Visual Studio надає зручний та інтуїтивно зрозумілий інтерфейс для програмістів. Він має багатий функціонал, який сприяє підвищенню продуктивності розробника та спрощує розробку програмного забезпечення.
- Підтримка мов програмування: Visual Studio підтримує широкий спектр мов програмування, включаючи C#, VB.NET, C++, F#, Python та багато інших. Це дає розробникам можливість працювати з мовою, яка найкраще підходить для їх проекту.
- Розширюваність: Visual Studio надає можливість розширення функціональності за допомогою плагінів та розширень. Розробники можуть використовувати розширення, щоб додати нові інструменти, шаблони проектів, підтримку інших платформ та бібліотек.
- Універсальність: Visual Studio дозволяє розробникам створювати різноманітні типи програмного забезпечення, включаючи десктопні додатки, веб-додатки, мобільні додатки, хмарні рішення, ігри та інше. Це забезпечує гнучкість та масштабованість у розробці проектів.
- Інструменти для тестування та налагодження: Visual Studio надає широкий набір інструментів для тестування та налагодження програмного забезпечення. Розробники можуть використовувати вбудовані засоби для автоматизованого тестування, профілювання програми, аналізу продуктивності та відлагодження.

Visual Studio є потужним та комплексним середовищем розробки, яке допомагає розробникам створювати високоякісне програмне забезпечення для різних платформ та застосувань. Вона поєднує у собі зручний інтерфейс, багатий набір інструментів та широку підтримку мов програмування, що робить її популярним вибором серед розробників по всьому світу.

1.5.4 Excel Data Reader

Excel Data Reader – це відкрите програмне забезпечення, що надає можливість зчитувати дані з файлів Microsoft Excel без потреби встановлення Microsoft Office або використання COM-об'єктів. Вона підтримує формати файлів .xls і .xlsx і має простий API для отримання даних з аркушів, стовпців, рядків або конкретних діапазонів комірок. Бібліотека автоматично визначає типи даних, включаючи числа, рядки, дати, формули і гіперпосилання. Вона також підтримує обробку формул і може повертати їх результати. Excel Data Reader є кросплатформенною, що дозволяє використовувати її в проектах на .NET Framework та .NET Core.

1.5.5 ADO.NET

ADO.NET – це набір компонентів, що входить до складу .NET Framework і призначений для роботи з базами даних. Використовуючи ADO.NET, програмісти можуть підключатися до різних джерел даних, виконувати запити до бази даних, отримувати та зберігати дані.

ADO.NET надає провайдери для різних баз даних, таких як Microsoft SQL Server, Oracle, MySQL, SQLite і багатьох інших. Це дозволяє програмістам зручно підключатися до різних джерел даних без необхідності використовувати специфічні API для кожної бази даних.

За допомогою ADO.NET можна виконувати SQL-запити, зберігати процедури та обробляти дані. Він також надає можливості для роботи з наборами даних у вигляді DataSets, які дозволяють зберігати дані у пам'яті та здійснювати різні операції з цими даними, такі як фільтрація та сортування.

ADO.NET підтримує роботу з транзакціями, що дозволяє забезпечити цілісність даних. Ви можете групувати декілька операцій в одну транзакцію, яку можна комітити або відкатити.

ADO.NET є потужним та гнучким інструментом для роботи з базами даних в середовищі .NET, що дозволяє програмістам зручно та ефективно працювати з даними в різних типах проектів.

1.5.6 MSAGL

Microsoft Automatic Graph Layout (MSAGL) є бібліотекою, яка надає інструменти для автоматичного розміщення та візуалізації графів. Вона створена для спрощення роботи зі складними графовими структурами, такими як дерева, мережі або ієрархії.

MSAGL надає алгоритми автоматичного розміщення вузлів та ребер на площині, що дозволяє створювати графічне представлення даних. Бібліотека пропонує різноманітні методи розміщення, включаючи алгоритми, які враховують ваги, обмеження на відстані та інші фактори, що допомагають забезпечити зрозумілу та естетичну візуалізацію графів.

MSAGL підтримує різні типи візуалізації графів, включаючи направлені та ненаправлені графи, графи зі згортаними вузлами, графи з маркерами ребер та багато інших варіацій. Користувачі можуть налаштовувати вигляд графів, задавати кольори, розміри, стилі ребер та вузлів, що дозволяє створювати візуально привабливі та зрозумілі графічні представлення.

1.5 Аналоги

Наразі існує багато програм та бібліотек, які надають можливості для візуалізації даних у вигляді графів. Розглянемо деякі з них:

- Gephi: потужний інструмент для візуалізації та аналізу графів. Він надає широкий спектр функцій, таких як розміщення вузлів та ребер, фільтрація, кольорові схеми, анімація та інші. Gephi підтримує різні формати даних і забезпечує гнучкість при роботі з великими графами.
- Cytoscape: популярний інструмент для візуалізації та аналізу графів. Він має багатий набір функцій для маніпулювання графами, включаючи розміщення вузлів, стилізацію, фільтрацію, пошук шляхів та багато іншого. Cytoscape підтримує також розширення для розширення функціональності програми.
- Neo4j: графова база даних, яка розроблена спеціально для зберігання, керування та опрацювання графових структур. Вона базується на мові запитів Cypher і використовує модель властивостей-графів, що дозволяє ефективно працювати зі зв'язками та залежностями між елементами даних.

2 ПРАКТИЧНА ЧАСТИНА

2.1 Постановка задачі

Основним завданням дипломної роботи є розробка функціонального програмного застосунку, який дозволить досліднику візуалізувати родинні зв'язки ліній соняшнику за фенотипом. На виході з системи очікується отримання наступних результатів:

- Графічне представлення родинних зв'язків: Система повинна забезпечувати можливість побудови графа, в якому вершини будуть лініями соняшнику, а ребрами — спорідненості між ними. Граф має бути інтуїтивно зрозумілим та легко читатися, щоб дослідник міг аналізувати родинні зв'язки та знаходити залежності.

- Інтерактивність: Система має надавати можливість інтерактивного взаємодії з графом, дозволяючи задавати певні параметри для його побудови. Це сприятиме більш детальному вивченню родинних зв'язків та знаходженню конкретних паттернів.

- Інтеграція з базою даних та електронними таблицями MS Excel: Система повинна підтримувати зчитування даних про родинні зв'язки ліній соняшнику з електронних таблиць MS Excel та їх подальше зберігання у базі даних. Це дозволить досліднику легко і швидко оновлювати дані та аналізувати нові набори даних.

Отримання цих результатів дозволить досліднику краще розуміти спорідненість ліній соняшнику за фенотипом, виділяти групи ліній з подібними характеристиками та виявляти залежності між ними.

2.2 Робота з MS Excel та базою даних

У даному підрозділі розглядається робота з електронними таблицями MS Excel та базами даних. Для цього використовуються бібліотека Excel Data Reader та технологія ADO NET, що дозволяють нам забезпечити зручний та надійний доступ до даних з цих джерел.

Вихідні дані ми отримуємо у вигляді електронної таблиці MS Excel (рис 2.1).

	A	B	C	D	E	F	G	H	I
3	Назва	Ідентифікатор	Підгрупа	Олійність 2017	Маса 1000 шт 2017	Довжина листа	Ширина листа	Черешок	Дата
4	BA16	UE0100456	1	46.41	49.88	19.20	19.80	14.60	7-Jul
5	BA26	UE0100519	1	40.71	40.84	18.20	18.00	10.70	13-Jul
6	KГ16	UE0100523	2	21.71	72.00	19.40	15.40	16.80	14-Jul
7	InK1589		2	14.60	43.27	16.50	11.30	13.90	16-Jul
8	LD4(без ант)	UE0100566	3	22.61	38.52	17.30	15.80	8.60	17-Jul
9	LD72/2	UE0100568	3			20.33	18.17	11.33	10-Jul
10	LD4/1	UE0100565	3			10.67	11.67	5.50	
11	LD4/2	UE0100566	3	24.50	43.02	18.40	15.70	10.60	7-Jul
12	LD72/1	UE0100567	3	20.79	46.66	16.40	14.30	9.40	7.07; 14.07
13	LD72/3	UE0100569	3	24.24	38.86	19.20	17.00	10.80	10-Jul
14	InK404	UE0100533	3	24.18	46.82	18.80	15.60	11.30	7-Jul
15	2K224-2	UE0100595	3	28.98	51.03	16.30	13.00	10.00	7-Jul
16	InK235	UE0100919	4	24.20	43.71	14.20	12.60	9.20	3-Jul
17	APS 56	UE0100882	4	23.72	25.56	13.00	13.40	10.00	3-Jul
18	InK103	UE0100885	4	15.33	30.79	15.70	14.40	11.20	3-Jul
19	ВІР199	UE0100549	5	34.71	60.71	18.50	19.20	8.80	14.07; 12.07
20	ВІР501	UE0101066	5			20.20	21.90	11.00	17-Jul
21	InДН47	UE0100546	5	46.92	38.50	11.20	10.30	8.00	7-Jul
22	ДН47-2	UE0100613	5	26.52	35.53	24.20	26.00	14.10	17-Jul
23	ДН47-8	UE0100616	5	22.33	30.14	15.70	15.50	6.40	14-Jul
24	ДН47-9	UE0100617	5	32.36	43.72	15.70	15.80	8.00	16-Jul
25	ЗЛ226	UE0100696	6	36.50	44.29	18.16	18.56	11.82	13-Jul
26	НА896	UE0100460	6	41.32	44.38	16.48	16.74	11.92	12.07; 13.07
27	ЗЛ22/319	UE0101290	6	43.41	45.13	17.52	15.88	16.94	10-Jul
28	ЗЛ22/320	UE0101291	6	36.64	45.90	16.22	15.78	10.70	14-Jul
29	ЗЛ226/ ВС6	бахрома ВС6	6			18.92	16.74	12.56	17-Jul
30	ЗЛ22/434	UE0101292	6			21.84	23.36	12.32	17-Jul
31	ЗЛ226/ 2	UE0100696	6	38.01	30.60	14.92	14.32	11.46	17-Jul

Рисунок 2.1 – Дані про лінії соняшнику

Для того щоб завантажити ці дані до програмного застосунку ми повинні натиснути на пункт меню “Data”, а потім “Load” (рис 2.2).

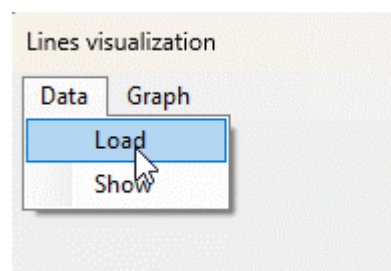


Рисунок 2.2 – Зовнішній вигляд контекстного меню “Data”

У вікні, що відкрилося, ми повинні обрати потрібний файл (рис 2.3)

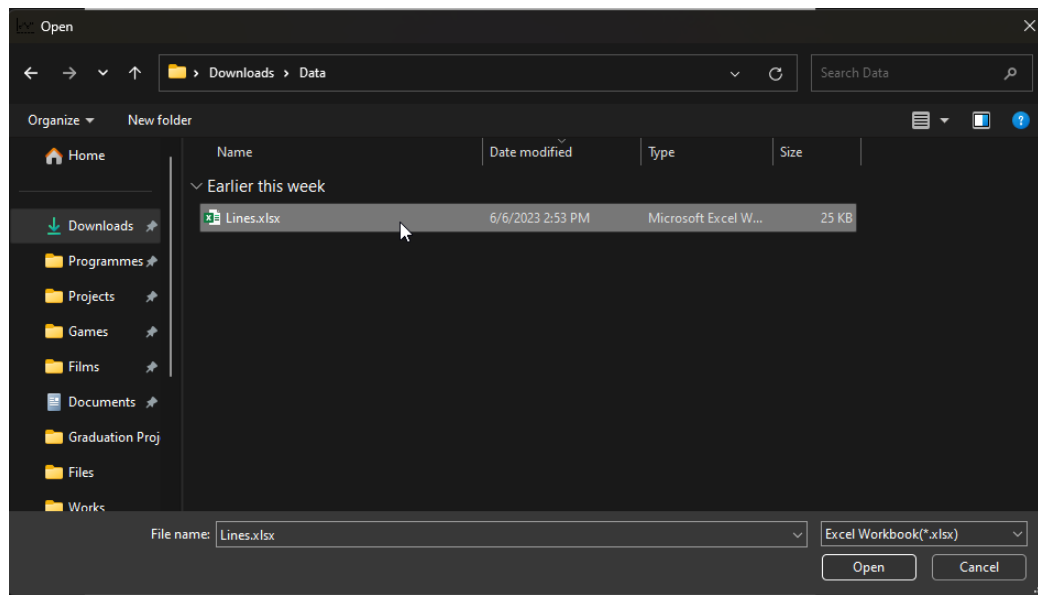


Рисунок 2.3 – Вибір файлу з даними

Після цього програма почне обробляти надану інформацію. Вибраний файл буде скопійований до папки “Files” у директорії проекту для подальшої взаємодії. Потім програма зчитує дані з першого листа MS Excel та створює базу даних з назвою “sunflower_lines” та таблицю “lines”. Для створення таблиці усі заголовки стовпців потрібно привести до стандартів SQL. Пробіли між словами повинні бути замінені на нижнє підкреслення, також заголовки не повинні містити спеціальних символів. Застосунок все це робить автоматично, після чого заносить дані про лінії соняшників до таблиці.

Після цього з’явиться вікно з повідомленням. У нашому випадку дані зчитані успішно (рис 2.4), але вікно також може повідомляти про помилку з описом того, що потрібно змінити у файлі Excel.

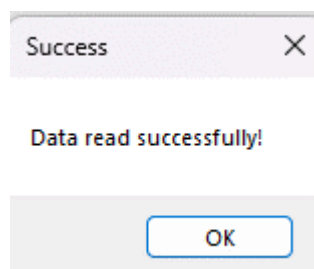
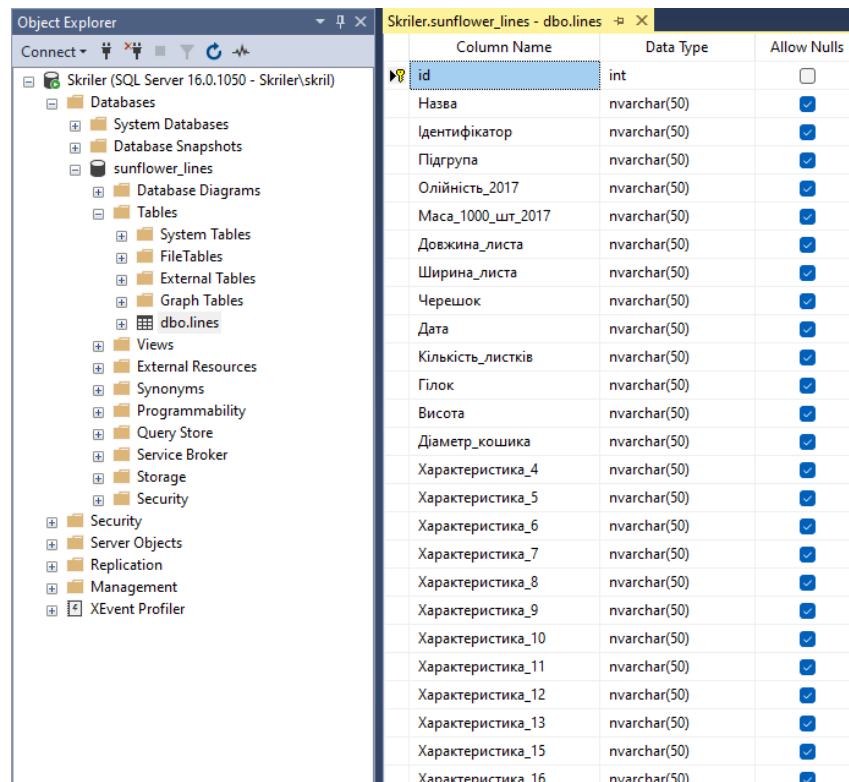


Рисунок 2.4 – Вікно, що повідомляє про успіх зчитування даних

Також за допомогою програми для роботи з SQL ми можемо побачити створену базу даних з назвою “sunflower_lines” (рис 2.5). Вона містить одну таблицю з назвою “lines”, яка зберігає дані про лінії соняшників, також можна побачити створені поля з типом nvarchar(50) для збереження строкових даних.



The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Object Explorer' pane displays the hierarchy: 'Skriler (SQL Server 16.0.1050 - Skriler\skril)' > 'Databases' > 'sunflower_lines' > 'Tables' > 'dbo.lines'. The main pane shows the table structure for 'Skriler.sunflower_lines - dbo.lines'.

Column Name	Data Type	Allow Nulls
id	int	☐
Назва	nvarchar(50)	☒
Ідентифікатор	nvarchar(50)	☒
Підгрупа	nvarchar(50)	☒
Олійність_2017	nvarchar(50)	☒
Маса_1000_шт_2017	nvarchar(50)	☒
Довжина_листа	nvarchar(50)	☒
Ширина_листа	nvarchar(50)	☒
Черешок	nvarchar(50)	☒
Дата	nvarchar(50)	☒
Кількість_листіків	nvarchar(50)	☒
Гілок	nvarchar(50)	☒
Висота	nvarchar(50)	☒
Діаметр_кошика	nvarchar(50)	☒
Характеристика_4	nvarchar(50)	☒
Характеристика_5	nvarchar(50)	☒
Характеристика_6	nvarchar(50)	☒
Характеристика_7	nvarchar(50)	☒
Характеристика_8	nvarchar(50)	☒
Характеристика_9	nvarchar(50)	☒
Характеристика_10	nvarchar(50)	☒
Характеристика_11	nvarchar(50)	☒
Характеристика_12	nvarchar(50)	☒
Характеристика_13	nvarchar(50)	☒
Характеристика_15	nvarchar(50)	☒
Характеристика_16	nvarchar(50)	☒

Рисунок 2.5 – Створена база даних

У свою чергу користувачу для того, щоб переглянути дані з бази даних потрібно натиснути на пункт меню “Data”, а потім “Show”(рис 2.6).

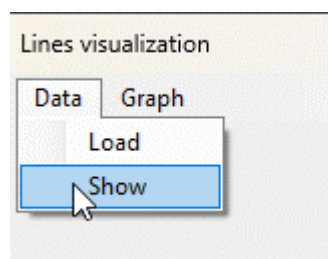


Рисунок 2.6 – Зовнішній вигляд контекстного меню “Data”

Якщо операцію завершено успішно з’явиться елемент Data Grid View, який відображає таблицю з отриманими даними (рис 2.7). Можна побачити, що дані збігаються з тими, що були у файлі MS Excel.

Lines visualization

Data

Graph

	id	Назва	Ідентифікатор	Підгрупа	Олійність_2017	Маса_1000_шт_2	Довжина_листа	Ширина_листа
▶	1	BA16	UE0100456	1	46.41	49.88	19.2	19.8
	2	BA26	UE0100519	1	40.713333333333...	40.836666666666...	18.2	18
	3	KG16	UE0100523	2	21.705	71.996666666666...	19.4	15.4
	4	InK1589		2	14.596666666666...	43.27	16.5	11.3
	5	LD4(без антоц)	UE0100566	3	22.606666666666...	38.52	17.3	15.8
	6	LD72/2	UE0100568	3			20.333333333333...	18.166666666666...
	7	LD4/1	UE0100565	3			10.666666666666...	11.666666666666...
	8	LD4/2	UE0100566	3	24.494999999999...	43.023333333333...	18.4	15.7
	9	LD72/1	UE0100567	3	20.790000000000...	46.656666666666...	16.4	14.3
	10	LD72/3	UE0100569	3	24.236666666666...	38.863333333333...	19.2	17
	11	InK404	UE0100533	3	24.18	46.82	18.8	15.6
	12	2K224-2	UE0100595	3	28.976666666666...	51.03	16.3	13
	13	InK235	UE0100919	4	24.2	43.706666666666...	14.2	12.6
	14	APS 56	UE0100882	4	23.72	25.556666666666...	13	13.4
	15	InK103	UE0100885	4	15.33	30.790000000000...	15.7	14.4
	16	ВІР199	UE0100549	5	34.713333333333...	60.706666666666...	18.5	19.2
	17	ВІР501	UE0101066	5			20.2	21.9
	18	InДН47	UE0100546	5	46.92	38.496666666666...	11.2	10.3
	19	ДН47-2	UE0100613	5	26.515	35.53	24.2	26

Рисунок 2.7 – Елемент Data Grid View з даними

При наступних запусках застосунку нам вже не потрібно обирати файл з даними, оскільки вони зчитуються з бази даних. Якщо ж нам потрібно внести інші дані, то ми можемо скористатися попереднім способом завантаження файлу у застосунок. При цьому старі дані будуть видалені (рис 2.8), а нові додані до бази даних.

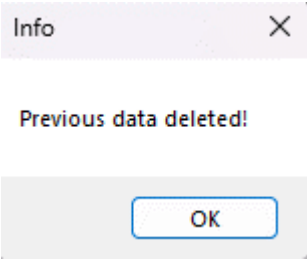


Рисунок 2.8 – Вікно, що повідомляє про видалення минулих даних

2.3 Візуалізація даних

У даному підрозділі розглядається робота з бібліотекою Microsoft.Msagl для візуалізації графа. За допомогою неї ми встановлюємо бажаний алгоритм для побудови графа, його зовнішній вигляд та колір вершин. Також ми можемо зберегти цей граф та у подальшому завантажити його до будь-якого застосунку, який підтримує цю бібліотеку, або роздрукувати його на аркуші паперу.

Для візуалізації даних користувачу потрібно натиснути на пункт меню “Graph”, а потім “Build”(рис 2.9).

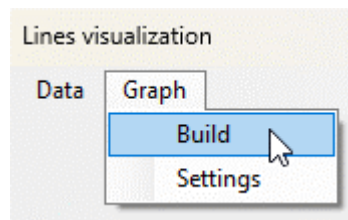


Рисунок 2.9 – Зовнішній вигляд контекстного меню “Graph”

Якщо додаток вже має завантаженні дані з таблиці MS Excel або бази даних з’явиться елемент Graph Viewer з побудованим графом (рис 2.10). Кожна вершина графу є однією з ліній соняшнику, а ребра між ними родинні зв’язки. Назва вершини є назвою відповідної лінії соняшнику, а лінії однієї групи мають однаковий колір. Для побудови ребра між вершинами нам необхідно визначити наявність родинних зв’язків між ними. Для цього ми порівнюємо характеристики кожної лінії з усіма іншими, використовуючи алгоритм повного перебору. Числові дані порівнюються з певною допустимою похибкою, яка за замовчуванням становить 10%. Крім того, ми повинні встановити мінімальну кількість схожих характеристик, яка потрібна для визначення родинного зв’язку. За замовчуванням це значення становить 40 з 75 наявних характеристик.

Settings

Min similar characteristics amount

Value: 40

Max deviation percent

Value: 10

Selected group

All

Submit

Selected characteristics

- ☒ Назва
- ☒ Ідентифікатор
- ☒ Підгрупа
- ☒ Олійність_2017
- ☒ Маса_1000_шт_2017
- ☒ Довжина_листа
- ☒ Ширина_листа
- ☒ Черешок
- ☒ Дата
- ☒ Кількість_листіків
- ☒ Гілок

Рисунок 2.12 – Вікно налаштувань

Змінимо значення параметрів на наступні (рис 2.13). Мінімальну кількість схожих характеристик оберемо 50, допустиму похибку виберемо 25%. З характеристик для перевірки виключимо такі: “Назва”, “Ідентифікатор”, “Підгрупа” і “Дата”. При цьому максимальна кількість для першого параметра зміниться у меншу сторону. Змінювати групу наразі не будемо для того, щоб у результаті побачити усі наявні лінії.

Settings

Min similar characteristics amount

Value: 50

Max deviation percent

Value: 25

Selected group

All

Submit

Selected characteristics

- ☐ Назва
- ☐ Ідентифікатор
- ☐ Підгрупа
- ☒ Олійність_2017
- ☒ Маса_1000_шт_2017
- ☒ Довжина_листа
- ☒ Ширина_листа
- ☒ Черешок
- ☐ Дата
- ☒ Кількість_листіків
- ☒ Гілок

Рисунок 2.13 – Вікно налаштувань зі зміненими параметрами

Побудуємо граф (рис 2.14) з новими параметрами. Можна побачити, що кількість вільних вершин стала більшою, а більшість ліній соняшнику з кожної підгрупи пов'язані між собою або ж не пов'язані з жодною.

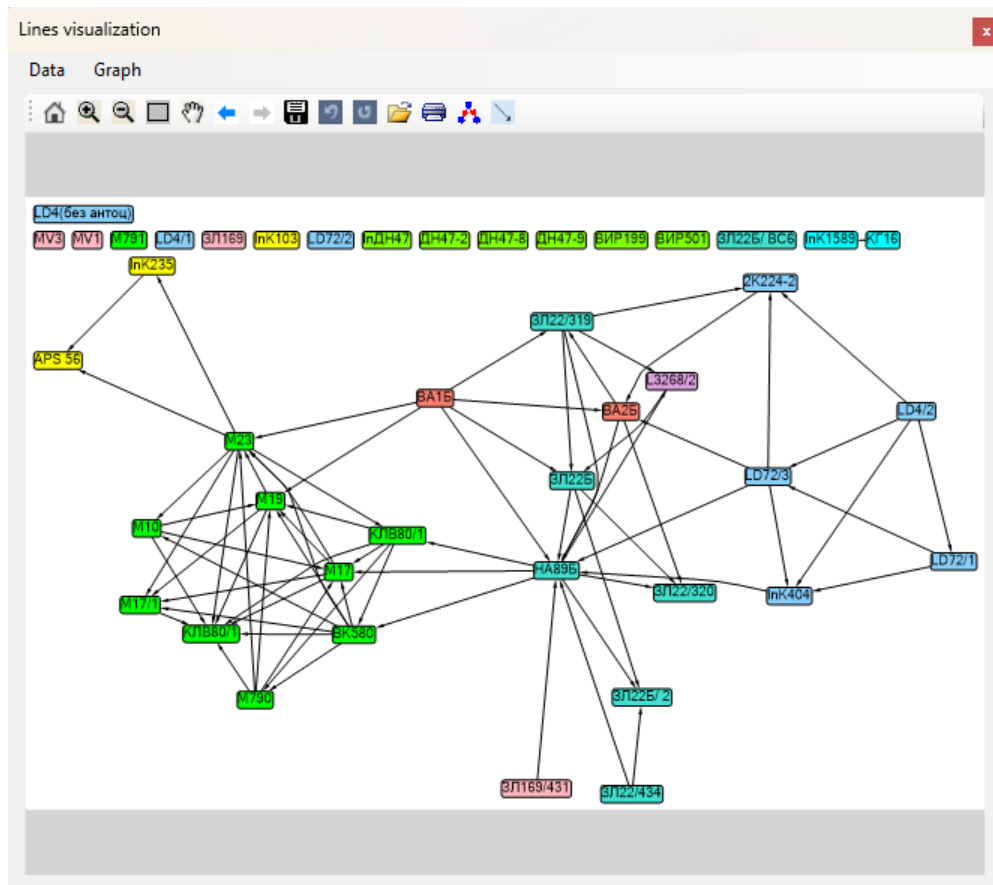


Рисунок 2.14 –Побудований граф зі зміненими параметрами

2.4 Експерименти з візуалізацією графа

Побудуємо граф (рис 2.15) для підгрупи номер 7 з такими параметрами: мінімальна кількість схожих характеристик — 40, допустима похибка — 15%. Характеристики для перевірки включають усі, окрім “Назва”, “Ідентифікатор”, “Підгрупа” і “Дата”. Для візуалізації графа будемо використовувати алгоритм багатовимірного шкалювання, який також був використаний для усіх графів з попереднього розділу.

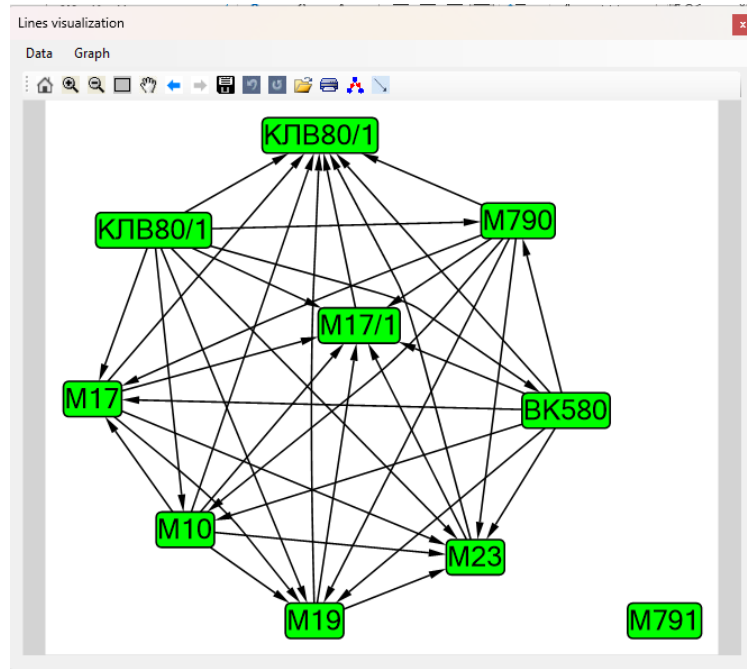


Рисунок 2.15 –Граф побудований за допомогою алгоритму багатовимірного шкалювання

Спробуємо візуалізувати цей самий граф за допомогою алгоритму Сугіями (рис 2.16). Бачимо, що вершини графу мають інше розташування.

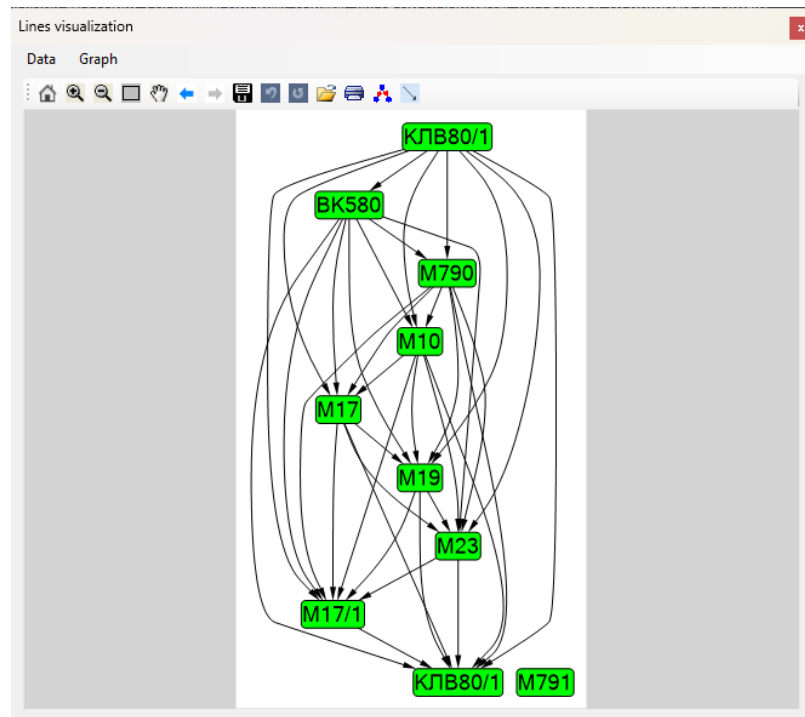


Рисунок 2.16 – Граф побудований за допомогою алгоритму Сугіями

ВИСНОВКИ

У ході даної роботи був розроблений функціонуючий програмний застосунок, що дозволяє досліднику візуалізувати родинні зв'язки ліній соняшнику за статистичними даними польового експерименту, які агреговані в електронних таблицях MS Excel. Також була побудована структурна схема системи для дослідження спорідненості ліній соняшнику за фенотипом.

Використано мову програмування C# та середовище Visual Studio для створення програмного застосунку. Використано бібліотеку Excel Data Reader для роботи з електронними таблицями MS Excel та бібліотеку Microsoft.Msagl для візуалізації графу. Для роботи з базами даних використано технологію ADO NET.

Результати проведених експериментів та аналізу підтверджують функціональність та ефективність розробленої системи. Завдяки їй, дослідники можуть отримати об'єктивну інформацію про генетичну спорідненість між лініями соняшнику, що дозволяє зробити обґрунтовані рішення щодо покращення селекційних програм та вибору найкращих генетичних ресурсів для покращення врожайності та якості соняшнику.

На мою думку, створена система для візуалізації родинних зв'язків ліній соняшнику є цінним інструментом для дослідників у галузі селекції соняшнику, також можливий подальший розвиток цієї системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. **Галузь, Я.** Методи класифікації і прогнозування [Текст] / Я. Галузь, О. Храмов, Ю. Серебренніков. – К. : Національний університет «Києво-Могилянська академія», 2014. – 198 с.
2. **Жук, Л. М.** Методи класифікації даних : навчальний посібник [Текст] / Л. М. Жук, В. Г. Курган. – К. : Центр учбової літератури, 2012. – 216 с.
3. **Іваницький, В. В.** Методи класифікації та регресійного аналізу [Текст] / В. В. Іваницький, І. В. Литвиненко. – К. : Видавництво КНУБА, 2013. – 328 с.
4. **Ляшенко, О.** Візуалізація даних. Дослідницька аналітика і презентація даних [Текст] / О. Ляшенко, О. Біленький. – Київ: Видавництво "Слово", 2018. – 192 с.
5. **Дяченко, О.** Візуалізація даних у вигляді графа для аналізу граф-моделей. Автоматика, вимірювання та керування [Текст] / О. Дяченко, І. Ігнатович. – 2015. – № 1. – С. 34–39.
6. **Козін, І.** Візуалізація даних: практичний підхід [Текст] / І. Козін. – Харків: "Корунд", 2017. – 224 с.
7. **Грищенко, І. І.** Візуалізація даних у вигляді графа для аналізу та дослідження великих даних. Проблеми програмування [Текст] / І. Грищенко. – 2016. – № 1–2. – С. 158–167.
8. **Сіренко, Р.С.** Порівняння алгоритмів візуалізації графів [Текст] / Р. Сіренко // Проблеми та напрямки розвитку, створення і використання інформаційних систем та технологій : матеріали Всеукраїнської науково-практичної конференції здобувачів вищої освіти та молодих учених, [Дніпро] 5 – 6 грудня 2022 р. – Дніпро, 2022. – С. 164–166.

ДОДАТОК А

Програмний код

MainForm.cs

```

public partial class MainForm : Form {
    private string filterPattern = "Excel Workbook(*.xlsx)|*.xlsx|Excel 97-
Excel 2003 Workbook(*.xls)|*.xls";
    private string filesDirectory = "..\\..\\..\\Files";
    private ApplicationSettings settings;
    private NodeObjectsList nodes;
    private ExcelReader excelReader;
    private DatabaseRepository dbManager;
    private DataComparer dataComparer;
    private GraphCreator graphCreator;
    public MainForm() {
        InitializeComponent();
        settings = new ApplicationSettings();
        dbManager = new DatabaseRepository(settings.DatabaseName);
        excelReader = new ExcelReader();
        dataComparer = new DataComparer(settings);
        graphCreator = new GraphCreator(settings);
    }

    private void MainForm_Load(object sender, EventArgs e){
        InitializeComponentsParameters();
        try {
            nodes = TryGetDataFromDatabase();
            if (nodes != null) {
                nodes.SetKeyCharacteristicsToNodes();
                settings.SetSelectedFields(nodes.ColumnNames);
            }
        }
    }
}

```

```

    } }
catch (Exception ex) {
    MessageBox.Show($"Error: {ex.Message}", "Error");
} }

private void InitializeComponentsParameters() {
    openFileDialog.InitialDirectory = "D:\\";
    openFileDialog.Filter = filterPattern;
    openFileDialog.RestoreDirectory = true;
}

private void loadToolStripMenuItem_Click(object sender, EventArgs
e){

    if (openFileDialog.ShowDialog() != DialogResult.OK)
        return;
    string filePath = openFileDialog.FileName;
    string fileName = Path.GetFileName(filePath);
    string destinationPath = Path.Combine(filesDirectory, fileName);
    try {
        if (File.Exists(destinationPath)) {
            File.Delete(destinationPath);
            dbManager.DropTable(settings.TableName);
            settings.SetDefaultValues();
            MessageBox.Show("Previous data deleted!", "Info");
        }
        File.Copy(filePath, destinationPath);
        nodes = excelReader.GetExcelData(fileName, filesDirectory);
        dbManager.CreateDatabase();
        dbManager.CreateTable(settings.TableName, nodes);
        dbManager.InsertDataIntoTable(settings.TableName, nodes);

        MessageBox.Show("Data read successfully!", "Success");
    }
}

```

```

        nodes.SetKeyCharacteristicsToNodes();
        settings.SetSelectedFields(nodes.ColumnNames);
    }
    catch (Exception ex) {
        MessageBox.Show($"Error: {ex.Message}", "Error");
    } }

private void showToolStripMenuItem_Click(object sender, EventArgs
e){

    try{
        if (!dbManager.IsDatabaseExist() ||
!dbManager.IsTableExist(settings.TableName)){
            MessageBox.Show($"Nothing to show", "Info");
            return;
        }
        DataSet dataSet =
dbManager.GetAllTableData(settings.TableName);
        if (dataSet != null){
            graphViewer.Visible = false;
            dgvTableData.Visible = true;
            dgvTableData.DataSource =
dataSet.Tables[settings.TableName]?.DefaultView;
        } }
        catch (Exception ex){
            MessageBox.Show($"Error: {ex.Message}", "Error");
        } }

private void buildToolStripMenuItem_Click(object sender, EventArgs
e){

    if (nodes == null) {
        MessageBox.Show($"No data to visualize", "Info");
        return;
    }

```

```

    }
    dgvTableData.Visible = false;
    graphViewer.Visible = true;
    try {
        List<NodeObject> selectedObjects;
        if (settings.SelectedGroup !=
ApplicationSettings.AllGenresSelectedCode) {
            selectedObjects = nodes.NodeObjects.Where(n => n.Group ==
settings.SelectedGroup).ToList();
        }
        else {
            selectedObjects = nodes.NodeObjects;
        }
        var similarObjects =
dataComparer.GetSimilarObjects(nodes.ColumnNames, selectedObjects);
        graphViewer.Graph =
graphCreator.CreateGraph(similarObjects);
    }
    catch (Exception ex) {
        MessageBox.Show($"Error: {ex.Message}", "Error");
    } }
    private void settingsToolStripMenuItem_Click(object sender,
EventArgs e) {
        if (nodes == null) {
            MessageBox.Show($"No data to configure", "Info");
            return;
        }
        SettingsForm settingsForm = new SettingsForm(settings, nodes);
        settingsForm.ShowDialog();
    }

```

```

private void MainForm_Resize(object sender, EventArgs e) {
    graphViewer.Width = dgvTableData.Width;
    graphViewer.Height = dgvTableData.Height;
}

private NodeObjectsList? TryGetDataFromDatabase() {
    NodeObjectsList nodes = null;
    if (!dbManager.IsDatabaseExist() ||
!dbManager.IsTableExist(settings.TableName))
        return nodes;
    DataSet dataSet =
dbManager.GetAllTableData(settings.TableName);
    string[] columnNames =
dbManager.GetTableColumnNames(settings.TableName);
    if (dataSet == null || columnNames == null)
        return nodes;
    DataTable dataTable = dataSet.Tables[settings.TableName];
    if (dataTable == null)
        return nodes;
    int rowsAmount = dataTable.Rows.Count;
    int columnsAmount = dataTable.Columns.Count;
    nodes = new NodeObjectsList(columnsAmount - 1);
    nodes.SetColumnNames(columnNames.Skip(1).ToArray());
    var dataRows = dataTable.Rows;
    NodeObject nodeObject;
    for (int i = 0; i < rowsAmount; ++i) {
        nodeObject = new NodeObject();
        for (int j = 1; j < columnsAmount; ++j) {
            nodeObject.AddValue(
                columnNames[j],
                dataRows[i][j]?.ToString()

```

```

        );
    }
    nodes.NodeObjects.Add(nodeObject);
}
return nodes;
} }

```

ExcelReader.cs

```

public class ExcelReader {
    public short RowHeadersIndex { get; private set; }
    public ExcelReader(short rowHeadersIndex = 2) {
        RowHeadersIndex = rowHeadersIndex;
    }
    public NodeObjectsList? GetExcelData(string fileName, string
fileDirectory) {
        NodeObjectsList nodes = null;
        Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);
        string filepath = Path.Combine(fileDirectory, fileName);
        using (var streamval = File.Open(filepath, FileMode.Open,
FileAccess.Read)) {
            using (var reader =
ExcelReaderFactory.CreateReader(streamval)) {
                var configuration = new ExcelDataSetConfiguration {
                    ConfigureDataTable = _ => new
ExcelDataTableConfiguration {
                        UseHeaderRow = false
                    } };
                var dataSet = reader.AsDataSet(configuration);
                if (dataSet.Tables.Count > 0)
                {

```

```

        nodes = ParseTableData(dataSet.Tables[0]);
    } } }
    return nodes;
}

private NodeObjectsList ParseTableData(DataTable dataTable) {
    NodeObjectsList nodes = new
NodeObjectsList(dataTable.Columns.Count);
    DataRow[] rows = dataTable.Select();
    DataRow currentRow;
    string temp;
    for (short i = 0; i < dataTable.Columns.Count; ++i) {
        nodes.ColumnNames[i] =
rows[RowHeadersIndex].ItemArray.ElementAt(i)?.ToString();
    }
    NodeObject nodeObject;
    for (short i = 0; i < dataTable.Rows.Count - RowHeadersIndex - 1;
++i) {
        currentRow = rows[i + RowHeadersIndex + 1];
        nodeObject = new NodeObject();
        for (short j = 0; j < dataTable.Columns.Count; ++j) {
            temp = currentRow.ItemArray.ElementAt(j)?.ToString();
            nodeObject.AddValue(nodes.ColumnNames[j], temp);
        }
        nodes.NodeObjects.Add(nodeObject);
    }
    return nodes;
} }

```

DatabaseRepository.cs

```

public class DatabaseRepository {

```

```

        private string serverConnStr = "Data Source=.;Integrated
security=True";

        public string DatabaseName { get; private set; }
        private Dictionary<string, string[]> tableHeaders;
        private SqlConnection serverConnection;
        private SqlConnection dbConnection;
        public DatabaseRepository(string dbName) {
            DatabaseName = dbName;
            string connStr = $"Data Source=.;Initial
Catalog={dbName};Integrated security=True";
            tableHeaders = new Dictionary<string, string[]>();
            serverConnection = new SqlConnection(serverConnStr);
            dbConnection = new SqlConnection(connStr);
        }
        public void CreateDatabase() {
            string queryCreateDB = $"Create database {DatabaseName}";
            SqlCommand createDBCommand;
            try {
                if (!IsDatabaseExist()) {
                    serverConnection.Open();
                    createDBCommand = serverConnection.CreateCommand();
                    createDBCommand.CommandText = queryCreateDB;
                    createDBCommand.ExecuteNonQuery();
                }
            }
            catch (Exception ex) {
                throw new Exception(ex.Message);
            }
            finally {
                serverConnection?.Close();
            }
        }
    }
}

```

```

public void CreateTable(string tableName, NodeObjectsList nodes) {
    string queryCreateTable;
    SqlCommand createTableCommand;
    try {
        if (!IsTableExist(tableName)) {
            string[] columnNames = new
string[nodes.ColumnNames.Length];
            Array.Copy(nodes.ColumnNames, columnNames,
nodes.ColumnNames.Length);
            tableHeaders.Add(tableName, columnNames);
            queryCreateTable = GetCreateTableQuery(tableName,
columnNames);
            dbConnection.Open();
            createTableCommand = dbConnection.CreateCommand();
            createTableCommand.CommandText = queryCreateTable;
            createTableCommand.ExecuteNonQuery();
        }
    }
    catch (Exception ex) {
        throw new Exception(ex.Message);
    }
    finally {
        serverConnection?.Close();
    }
}

public void DropTable(string tableName) {
    string queryDropTable = $"DROP TABLE {tableName}";
    SqlCommand dropTableCommand;
    try {
        if (IsTableExist(tableName)) {
            dbConnection.Open();

```

```

        dropTableCommand = dbConnection.CreateCommand();
        dropTableCommand.CommandText = queryDropTable;
        dropTableCommand.ExecuteNonQuery();
        tableHeaders.Clear();
    } }
    catch (Exception ex) {
        throw new Exception(ex.Message);
    }
    finally {
        serverConnection?.Close();
    } }

public void InsertDataIntoTable(string tableName, NodeObjectsList
nodes) {
    if (!tableHeaders.ContainsKey(tableName))
        throw new Exception("There is no such table");
    string queryInsertData;
    SqlCommand insertDataCommand;
    Try {
        string[] columnNames =
tableHeaders.GetValueOrDefault(tableName);
        queryInsertData = GetInsertDataQuery(tableName,
columnNames);
        dbConnection.Open();
        insertDataCommand = dbConnection.CreateCommand();
        insertDataCommand.CommandText = queryInsertData;
        for (int i = 0; i < columnNames.Length; ++i) {
            insertDataCommand.Parameters.Add($"@{columnNames[i]}",
SqlDbType.NVarChar, 50);
        }
    }
}

```

```

        NodeObject currentObject;
        string temp = string.Empty;
        for (int i = 0; i < nodes.NodeObjects.Count; ++i) {
            currentObject = nodes.NodeObjects[i];
            for (int j = 0; j < nodes.ColumnNames.Length; ++j) {
                temp = currentObject[nodes.ColumnNames[j]];
                insertDataCommand.Parameters[j].Value = temp;
            }
            insertDataCommand.ExecuteNonQuery();
        }
    }
    catch (Exception ex) {
        throw new Exception(ex.Message);
    }
    finally {
        serverConnection?.Close();
    }
}

public DataSet GetAllTableData(string tableName) {
    string getAllData = $"SELECT * FROM {tableName}";
    SqlDataAdapter getAllDataAdapter;
    DataSet dataSet = null;
    try {
        dbConnection.Open();
        getAllDataAdapter = new SqlDataAdapter(getAllData,
dbConnection);

        dataSet = new DataSet();
        getAllDataAdapter.Fill(dataSet, tableName);
    }
    catch (Exception ex) {
        throw new Exception(ex.Message);
    }
}

```

```

        finally {
            dbConnection?.Close();
        }
        return dataSet;
    }

    public string[] GetTableColumnNames(string tableName) {
        string[] columnNames = null;
        try {
            dbConnection.Open();
            DataTable dataTable = dbConnection
                .GetSchema("Columns", new string[] {
                    null,
                    null,
                    tableName
                });
            columnNames = (from row in dataTable.AsEnumerable()
                select row.Field<string>("COLUMN_NAME"))
                .ToArray();
        }
        catch (Exception ex) {
            throw new Exception(ex.Message);
        }
        finally {
            dbConnection?.Close();
        }
        return columnNames;
    }

    public bool IsDatabaseExist() {
        bool isDatabaseExist = false;
        string query = $"SELECT db_id('{DatabaseName}')";
    }

```

```

SqlCommand command;
try {
    serverConnection.Open();
    command = serverConnection.CreateCommand();
    command.CommandText = query;
    isDatabaseExist = command.ExecuteScalar() != DBNull.Value;
}
catch (Exception ex) {
    throw new Exception(ex.Message);
}
finally {
    serverConnection?.Close();
}
return isDatabaseExist;
}

public bool IsTableExist(string tableName) {
    bool isTableExist = false;
    DataTable dTable;
    try {
        dbConnection.Open();
        dTable = dbConnection
            .GetSchema("TABLES", new string[] { null, null, tableName
    ));

        isTableExist = dTable.Rows.Count > 0;
    }
    catch (Exception ex) {
        throw new Exception(ex.Message);
    }
    finally {
        dbConnection?.Close();
    }
}

```

```

    }
    return isTableExist;
}

private string GetCreateTableQuery(string tableName, string[]
columnNames) {
    if (columnNames == null || columnNames.Length == 0)
        throw new Exception("No headers to create the table");
    string query = $"Create table {tableName}(id int not null primary
key identity(1,1)";
    TransformSeporatorsIntoUnderscores(in columnNames);
    foreach (string columnName in columnNames) {
        query += $",[{columnName}] nvarchar(50)";
    }
    query += ")";
    return query;
}

private string GetInsertDataQuery(string tableName, string[]
columnNames) {
    string queryInsertData = $"INSERT INTO {tableName}(";
    string valuesStr = $"VALUES (";
    for (int i = 0; i < columnNames.Length; ++i) {
        queryInsertData += $"{columnNames[i]}";
        valuesStr += $"@{columnNames[i]}";
        queryInsertData += i != columnNames.Length - 1 ? ", " : ")";
        valuesStr += i != columnNames.Length - 1 ? ", " : ")";
    }
    queryInsertData += valuesStr;
    return queryInsertData;
}

```

```

private void TransformSeparatorsIntoUnderscores(in string[] strings){
    string separators = "[ ,;]";
    for (int i = 0; i < strings.Length; ++i) {
        strings[i] = Regex.Replace(strings[i], separators, "_");
    } } }

```

DataComparer.cs

```

public class DataComparer {
    private ApplicationSettings settings;
    public DataComparer(ApplicationSettings settings) {
        this.settings = settings;
    }
    public Dictionary<NodeObject, List<NodeObject>>
    GetSimilarObjects(string[] columnNames, List<NodeObject> objects) {
        int[,] amountOfSimilarCharacteristics =
    CompareObjects(columnNames, objects);
        var similarObjects = new Dictionary<NodeObject,
    List<NodeObject>>>();
        List<NodeObject> currentSimilarObjectsList;
        for (int i = 0; i < amountOfSimilarCharacteristics.GetLength(0);
    ++i) {
            similarObjects.Add(objects[i], new List<NodeObject>());
            similarObjects.TryGetValue(objects[i], out
    currentSimilarObjectsList);
            for (int j = 0; j < amountOfSimilarCharacteristics.GetLength(1);
    ++j) {
                if (amountOfSimilarCharacteristics[i, j] <
    settings.MinSimilarCharacteristicsAmount)
                    continue;

```

```

        currentSimilarObjectsList?.Add(objects[j]);
    } }
    return similarObjects;
}

public int[,] CompareObjects(string[] columnNames,
List<NodeObject> objects) {
    int objectsAmount = objects.Count;
    int[,] amountOfSimilarCharacteristics = new int[objectsAmount,
objectsAmount];
    for (int i = 0; i < objectsAmount; ++i) {
        for (int j = 0; j < objectsAmount; ++j) {
            if (i == j)
                continue;
            amountOfSimilarCharacteristics[i, j] =
                GetAmountOfSimilarCharacteristics(
                    columnNames,
                    objects[i],
                    objects[j]
                ); } }
    return amountOfSimilarCharacteristics;
}

public int GetAmountOfSimilarCharacteristics(string[] characteristics,
NodeObject firstObject, NodeObject secondObject) {
    if (firstObject.Count != characteristics.Length ||
secondObject.Count != characteristics.Length)
        throw new Exception("Objects have incorrect amount of
characteristics");
    int similarCharacteristicsAmount = 0;
    bool isCharcActive;
    string currentCharacteristic;

```

```

for (int i = 0; i < characteristics.Length; ++i) {
    currentCharacteristic = characteristics[i];
    settings.SelectedFields.TryGetValue(
        currentCharacteristic,
        out isCharcActive
    );
    if (!isCharcActive)
        continue;
    if (!IsCharacteristicsEquals(
        firstObject[currentCharacteristic],
        secondObject[currentCharacteristic]
    ))
        continue;
    ++similarCharacteristicsAmount;
}
return similarCharacteristicsAmount;
}

public bool IsCharacteristicsEquals(string firstCharc, string
secondCharc) {
    if (firstCharc == string.Empty && secondCharc == string.Empty)
        return false;
    double firstNumb;
    double secondNumb;
    bool isFirstCharcNumber = double.TryParse(firstCharc, out
firstNumb);
    bool isSecondCharcNumber = double.TryParse(secondCharc, out
secondNumb);
    if (isFirstCharcNumber && isSecondCharcNumber) {
        double minPossibleBorder = firstNumb *
settings.MinPossibleCoeff;

```

```

        double maxPossibleBorder = firstNumb *
settings.MaxPossibleCoeff;
        return minPossibleBorder <= secondNumb && secondNumb <=
maxPossibleBorder;
    }
    else {
        return firstCharc == secondCharc;
    } } }

```

GraphCreator.cs

```

public class GraphCreator {
    private static Random rand = new Random();
    private ApplicationSettings settings;
    private Dictionary<string, Color> groupColors;
    private List<Color> nodeColors;
    public GraphCreator(ApplicationSettings settings) {
        this.settings = settings;
        groupColors = new Dictionary<string, Color>();
        nodeColors = new List<Color>() {
            Color.Gold, Color.Chartreuse, Color.LightPink, Color.Plum,
Color.HotPink, Color.Salmon, Color.Snow, Color.Lime, Color.Tomato,
Color.LightSeaGreen, Color.Turquoise, Color.Brown, Color.GreenYellow,
Color.LightSkyBlue, Color.Yellow, Color.Silver, Color.Crimson, Color.Orchid,
Color.Peru, Color.Aquamarine, Color.Cyan, Color.DodgerBlue, Color.Teal,
Color.Violet, Color.RosyBrown,
        }; }
    public Graph CreateGraph(Dictionary<NodeObject,
List<NodeObject>> similarObjects, string graphTitle = "Lines") {
        Graph graph = new Graph(graphTitle);
        Node currentNode;

```

```

Edge currentEdge;
Color currentColor;
foreach (var item in similarObjects) {
    currentNode = graph.AddNode(item.Key.Name);
    if (groupColors.ContainsKey(item.Key.Group)) {
        currentColor =
groupColors.GetValueOrDefault(item.Key.Group);
    }
    else {
        currentColor = GetNewRandomColor(groupColors);
        groupColors.Add(item.Key.Group, currentColor);
    }
    currentNode.Attr.FillColor = currentColor;
}
foreach (var item in similarObjects) {
    foreach (var nodeObject in item.Value) {
        currentEdge = graph.Edges
            .FirstOrDefault(x => x.Source == nodeObject.Name &&
x.Target == item.Key.Name);
        if (currentEdge != null)
            continue;
        graph.AddEdge(item.Key.Name, nodeObject.Name);
    } }
return graph;
}

private Color GetNewRandomColor(Dictionary<string, Color>
groupColors) {
    Color randColor;
    do {
        randColor = nodeColors[rand.Next(nodeColors.Count)];
    }
}

```

```
    } while (groupColors.ContainsValue(randColor));  
    return randColor;  
} }
```