ЗМІСТ

ВСТУП

Дана курсова робота реалізує візуалізацію алгоритму пошуку найкоротшого шляху в двовимірному лабіринті використовуючи алгоритм Флойда-Воршолла.

Метою даної курсової роботи є створення програми яка реалізує поставлене індивідуальне завдання і буде мати зручний інтерфейс, отримання практичних навичок програмування алгоритмів, що оброблюють графи, отримання навичок програмування графічного інтерфейсу користувача а також закріплення знань з курсу «Алгоритми та структури даних».

Тема даної курсової роботи має високу актуальність для студента, оскільки для її виконання треба мати знання у розділі дискретної математики «Теорія графів», а також мати представлення про об’єктно-орієнтоване програмування.

Висока актуальність полягає у тому, що теорія графів широко використовується у сучасних сферах життя та науках, таких як:

- двійковий пошук;

- інформатика та програмування (за допомогою теорії графів реалізується більшість алгоритмів, пов'язаних з деревами);

- комунікаційні та транспортні системи, зокрема для маршрутизації даних в Інтернеті;

- економіка;

- логістика;

- схемотехніка (топологія з’єднань елементів на печатній платі або мікросхемі представляє собою граф або гіперграф);;

Як можна побачити, теорія графів широко використовується і гарному спеціалісту у програмній інженерії необхідно не просто знати про такий розділ дискретної математики, а мати практичний досвід програмування алгоритмів що їх обробляють.

1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ТА АНАЛІЗУ АЛГОРИТМІВ. ПОСТАНОВКА ЗАДАЧІ

1.1Математичні основи аналізу алгоритмів

Для оцінки алгоритмів існує багато критеріїв. Найбільшу увагу приділяють порядку росту необхідних для розв'язання задачі часу та розміру пам'яті при збільшенні розміру вхідних даних.

Нехай A — алгоритм розв'язання деякого класу задач, а N — розмірність окремої задачі цього класу. N може бути, наприклад, розмірністю оброблюваного масиву, числом вершин оброблюваного графа тощо, Позначимо $f_A(N)$ функцію, що дає верхню межу максимального числа основних операцій (додавання, множення і т. д.), які повинен виконати алгоритм A, розв'язуючи задачу розмірності N. Говоритимемо, що алгоритм A поліноміальний, якщо $f_A(N)$ зростає не швидше, ніж деякий поліном від N. В іншому разі A — експоненціальний алгоритм [1].

Виявляється, що між цими класами алгоритмів є істотна різниця: при великих розмірностях задач (які, зазвичай, найцікавіші на практиці), поліноміальні алгоритми можуть бути виконані на сучасних комп'ютерах, тоді як експоненціальні алгоритми для практичних розмірностей задач, як правило, не можуть виконатися повністю. Зазвичай розв'язок задач, що породжують експоненціальні алгоритми, пов'язаний з повним перебором всіх можливих варіантів, і, зважаючи на практичну неможливість реалізації такої стратегії, для їх розв'язання розробляються інші підходи.

Наприклад, якщо навіть існує експоненціальний алгоритм для знаходження оптимального розв'язку деякої задачі, то на практиці застосовуються інші, ефективніші поліноміальні алгоритми для знаходження не обов'язково оптимального, а лише прийнятного розв'язку (наближеного до оптимального). Але навіть якщо задача допускає розв'язання за допомогою поліноміального алгоритму, його можна побудувати лише після глибокого вникання в суть поставленої задачі [2].

Розділяй та володарюй (англ. divide and conquer) в інформатиці — важлива парадигма розробки алгоритмів, що полягає в рекурсивному розбитті розв'язуваної задачі на дві або більше підзадачі того ж типу, але меншого розміру, і комбінуванні їх рішень для отримання відповіді до вихідної задачі. Розбиття виконуються до тих пір, поки всі підзадачі не виявляться елементарними.

Типовий приклад - алгоритм сортування злиттям. Щоб відсортувати масив чисел по зростанню, він розбивається на дві рівні частини, кожна сортується, потім відсортовані частині зливаються в одну. Ця процедура застосовується до кожної з частин до тих пір, поки сортовані частини масиву містять хоча б два елементи (щоб можна було її розбити на дві частини) [3].

Час роботи цього алгоритму складає n*ln(n) операцій, тоді як більш прості алгоритми вимагають n^2 часу, де n - розмір вихідного масиву.

Інші приклади важливих алгоритмів, в яких застосовується парадигма «розділяй та володарюй»:

- двійковий пошук;
- метод бісекції;
- швидке сортування;
- швидке перетворення Фур'є;
- алгоритм Карацуби та інші ефективні алгоритми для множення великих чисел;
- динамічне програмування;

У динамічному програмуванні для керованого процесу серед множини усіх допустимих управлінь шукають оптимальне у сенсі деякого критерію тобто таке, яке призводить до екстремального (найбільшого або найменшого) значення цільової функції — деякої числової характеристики процесу. Під багатоступеневістю розуміють або багатоступеневу структуру процесу, або розподілення управління на ряд послідовних етапів (ступенів, кроків), що відповідають, як правило, різним моментам часу. Таким чином, в назві «Динамічне програмування» під «програмуванням» розуміють «прийняття

рішень», «планування», а слово «динамічне» вказує на суттєве значення часу та порядку виконання операцій в процесах і методах, що розглядаються [4].

Методи динамічного програмування є складовою частиною методів, які використовуються при дослідженні операцій, і використовуються як у задачах оптимального планування, так і при розв'язанні різних технічних проблем (наприклад, у задачах визначення оптимальних розмірів ступенів багатоступеневих ракет, у задачах оптимального проектування прокладення доріг та ін.)

Методи динамічного програмування використовуються не лише в дискретних, але і в неперервних керованих процесах, наприклад, в таких процесах, коли в кожен момент певного інтервалу часу необхідно приймати рішення. Динамічне програмування також дало новий підхід до задач варіаційного числення.

Хоча метод динамічного програмування суттєво спрощує вихідні задачі, та безпосереднє його використання, як правило, пов'язане з громіздкими обчисленнями. Для подолання цих труднощів розробляються наближені методи динамічного програмування [4].

Амортизаційний аналіз — метод аналізу швидкодії алгоритмів, що розглядає усю послідовність операцій виконуваних програмою. Тут ми усереднюємо час необхідний для виконання операції над певною структурою даних. З амортизаційним аналізом ми можемо показати, що середній час необхідний на одну операцію нетривалий, хоча певні операції у послідовності вимагають багато часу. Амортизаційний аналіз відрізняється від аналізу середнього випадку тим, що ймовірність не має значення; амортизаційний аналіз гарантує середню швидкодію кожної операції в найгіршому випадку. Прикладами структур даних чиї операцію аналізуються за допомогою амортизаційного аналізу є хеш-таблиці, неперетинні множини, розширювані дерева [5].

Аналіз найгіршого випадку може дати занадто песимістичну оцінку для послідовності операцій, через те, що такий аналіз нехтує взаємодією між різними операціями на тій самій структурі даних . Амортизаційний аналіз може привести до більш реалістичної межі для найгіршого випадку завдяки врахуванню цих

взаємодій. Зауважимо, що межа представлена амортизаційним аналізом, насправді, є найгіршим випадком на середню операцію; деякі операції в послідовності можуть мати більшу трудомісткість ніж ця межа, але середня вартість в кожній правильній послідовності завжди коритиметься цій межі.

Амортизаційний аналіз подібний аналізу середнього випадку в сенсі, що він цікавиться вартістю усередненою на всю послідовність операцій. Однак, аналіз середнього випадку покладається на ймовірнісні припущення щодо структури даних і алгоритму для того, щоб обчислити сподіваний час виконання алгоритму. Тобто, його використання залежить від певних припущень щодо ймовірнісного розподілу вхідних даних, з чого випливає, що оцінка не правильна якщо припущення не дотримані (або ймовірнісний аналіз не можна використовувати взагалі, якщо не можна розподіл вхідних даних). Амортизаційний не потребує таких припущень [5].

1.2 Структури даних і класифікація основних алгоритмів

В програмуванні та комп'ютерних науках структури даних — це способи організації даних в комп'ютерах. Часто разом зі структурою даних пов'язується і специфічний перелік операцій, що можуть бути виконаними над даними, організованими в таку структуру [6].

Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх обробки. Добре побудовані структури даних дозволяють оптимізувати використання машинного часу та пам'яті комп'ютера для виконання найкритичніших операцій. Відома формула «Програма = Алгоритми + Структури даних» дуже точно виражає необхідність відповідального ставлення до такого підбору. Тому іноді навіть не обраний алгоритм для обробки масиву даних визначає вибір тої чи іншої структури даних для їх збереження, а навпаки.

Підтримка базових структур даних, які використовуються в програмуванні, включена в комплекти стандартних бібліотек найбільш розповсюджених мов

програмування, таких як Standard Template Library (STL) для C++, Java API, Microsoft.NET тощо.

Декілька типів структур даних :

– масив — впорядкований набір фіксованої кількості однотипних елементів, що зберігаються в послідовно розташованих комірках оперативної пам'яті, мають порядковий номер і спільне ім'я, що надає користувач;

– зв'язаний список— одна з найважливіших структур даних, в якій елементи лінійно впорядковані, але порядок визначається не номерами елементів, а вказівниками, які входять в склад елементів списку та вказують на наступний за даним елемент (в однозв’язних або однобічно зв'язаних списках) або на наступний та попередній елементи (в двозв'язних або двобічно зв'язаних списках). Список має «голову» — перший елемент та «хвіст» — останній елемент;

– хеш-таблиця — структура даних, що реалізує інтерфейс асоціативного масиву, а саме, вона дозволяє зберігати пари (ключ, значення) і здійснювати три операції: операцію додавання нової пари, операцію пошуку і операцію видалення за ключем;

– збалансоване дерево в загальному розумінні цього слова — це такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем є мінімальною. Ця властивість є важливою тому, що час виконання більшість алгоритмів на бінарних деревах пошуку пропорційний до їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних ситуаціях. Процедура зменшення (балансування) висоти дерева виконується за допомогою трансформацій, відомих як обернення дерева, в певні моменти часу (переважно при видаленні або додаванні нових елементів) [8];

Перелічимо деякі алгоритми на графах :

– пошук в глибину;

– пошук в ширину;

– топологічне сортування;

– фундаментальна множина циклів;

– Еейлерів цикл. Теорема Ейлера;

– Гамільтонів цикл;

– алгоритм Белмана-Форда;

– алгоритм Декстри;

– алгоритм Флойда-Воршолла;

Алгоритм пошуку в глибину— алгоритм для обходу дерева, структури подібної до дерева, або графа. Робота алгоритму починається з кореня дерева (або іншої обраної вершини в графі) і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

Пошук у ширину — алгоритм пошуку на графі. Алгоритм має назву пошуку в ширину, оскільки «фронт» пошуку (між пройденими та непройденими вершинами) одноманітно розширюється вздовж всієї своєї ширини. Тобто, алгоритм проходить всі вершини на відстані k перед тим як пройти вершини на відстані k+1 [7].

У динамічному програмуванні для керованого процесу серед множини усіх допустимих управлінь шукають оптимальне у сенсі деякого критерію тобто таке яке призводить до екстремального (найбільшого або найменшого) значення цільової функції — деякої числової характеристики процесу. Під багатоступеневістю розуміють або багатоступеневу структуру процесу, або розподілення управління на ряд послідовних етапів (ступенів, кроків), що відповідають, як правило, різним моментам часу. Таким чином, в назві «Динамічне програмування» під «програмуванням» розуміють «прийняття рішень», «планування», а слово «динамічне» вказує на суттєве значення часу та порядку виконання операцій в процесах і методах, що розглядаються.

Методи динамічного програмування є складовою частиною методів, які використовуються при дослідженні операцій, і використовуються як у задачах оптимального планування, так і при розв'язанні різних технічних проблем (наприклад, у задачах визначення оптимальних розмірів ступенів багатоступеневих ракет, у задачах оптимального проектування прокладення доріг та ін.) [4].

Методи динамічного програмування використовуються не лише в дискретних, але і в неперервних керованих процесах, наприклад, в таких процесах, коли в кожен момент певного інтервалу часу необхідно приймати рішення. Динамічне програмування також дало новий підхід до задач варіаційного числення.

Хоча метод динамічного програмування суттєво спрощує вихідні задачі, та безпосереднє його використання, як правило, пов'язане з громіздкими обчисленнями. Для подолання цих труднощів розробляються наближені методи динамічного програмування.

Обчислювальна геометрія (англ. computational geometry) — галузь комп'ютерних наук присвячена вивченню алгоритмів що описуються в термінах геометрії.

Основним стимулом розвитку обчислювальної геометрії як дисципліни був прогрес у комп'ютерній графіці та системах автоматизованого проектування та автоматизованих систем технологічної підготовки виробництва, проте багато задач обчислювальної геометрії є класичними за своєю природою, і можуть з'являтись при математичній візуалізації [1].

Іншим важливим застосуванням обчислювальної геометрії є робототехніка (планування руху та задачі розпізнавання образів), геоінформаційні системи (геометричний пошук, планування маршруту), дизайн мікросхем, програмування станків з числовим програмним управлінням.

Основними розділами обчислювальної геометрії є комбінаторна обчислювальна геометрія, чи також названа алгоритмічна геометрія, яка розглядає геометричні об'єкти як дискретні сутності [8].

1.3 Постановка задачі

За завдання треба реалізувати візуалізацію алгоритмом пошуку найкоротшого шляху в двовимірному лабіринті.

Вхідні дані будуть представляти набір з N стін, завдань двома парами цілочислених координат своїх вершин; N = 20 для тесту та N = 50 для демонстрації.

Джерело даних: довільно вибраний відеофайл, дані якого обробляються як послідовність однобайтових чисел-координат вершин.

Вимоги до інтерфейсу: користувач винен мати можливість переглянутися стіні на площині, переміщувати їх за допомогою миші; результати переміщення, можна Зберегти до файлу у форматі, обираності довільно; користувач винен мати можливість зверни дві точки, щоб програма побудувалося між ними найкоротшій шлях, що не перетинає жодної стіні, або повідомила, що такого шляху не існує; має буті можливість переглянутися етапи побудова маршруту [9].

Якщо граф не містить ребр з негативним вагою, то для вирішення цієї проблеми можна використовувати алгоритм Дейкстри для знаходження найкоротшого шляху від однієї вершини до всіх інших, запустивши його на кожній вершині. Час роботи такого алгоритму залежить від типу даних, який ми будемо використовувати для алгоритму Дейкстри, це може бути як проста чергу з пріоритетом, так і бінарна або Фібоначієва Куча, відповідно час роботи буде варіюватися від O (V^3) до O (V * E * log (V)), де V кількість вершин, а E - ребр. («O» -велике) [11].

Якщо ж є ребра з негативним вагою, можна використовувати алгоритм Беллмана - Форда. Але цей алгоритм, запущений на всіх вершинах графа, повільніше, час його роботи O (V^2 * E), а в сильно «густих» графах аж O (V4).

В основі алгоритму лежать дві властивості найкоротшого шляху графа.

Маємо найкоротший шлях p1k = (v_1, v_2, ..., v_k) від вершини v1 до вершини v_k, а також його підшлях p '(v_i, v_{i+1}, ..., v_j), при цьому діє 1 <= i <= j < = k.

Якщо p - найкоротший шлях від v_1 до v_k, то p 'також є найкоротшим шляхом від вершини v_i до v_j.

Це можна легко довести, так як вартість шляху p складається з вартості шляху p' і вартості інших його частин. Так от представивши що є коротший шлях p ', ми зменшимо цю суму, що призведе до протиріччя з твердженням, що ця сума і так вже була мінімальною [7].

Друга властивість є основою алгоритму. Розглядається граф G з пронумерованими від 1 до n вершинами $\{v_1, v_2, ..., v_n\}$ і шлях p_{ij} від vi до v_j, що проходить через певну безліч дозволених вершин, обмежене індексом k.

Тобто якщо k = 0, то розглядаються прямі з'єднання вершин один з одним, так як безліч дозволених проміжних вершин рано нулю. Якщо k = 1 - розглядаємо шляхи, що проходять через вершину v_1, при k = 2 - через вершини $\{v_1, v_2\}$, при k = 3 - $\{v_1, v_3, v_3\}$ і так далі.

Наприклад маємо такий граф і k = 1, тобто в якості проміжкових вузлів дозволений тільки вузол «1». У цьому графі при k = 1 немає шляху p43, але є при k = 2, тоді можна дістатися з «4» в «3» через «2» або через «1» і «2».

Розглянемо найкоротший шлях p_{ij} з дозволеними проміжковими вершинами {1..k-1} вартістю d_{ij}. Тепер розширимо безліч на k-тий елемент, так що безліч дозволених вершин стане {1..k}. При такому розширенні можливо 2 результата:

Випадок 1. Елемент k не входить в найкоротший шлях pij, тобто від додавання додаткової вершини ми нічого не виграли і нічого не змінили, а значить вартість найкоротшого шляху dk_{ij} не змінився, відповідно

dk_{ij} = dk-1ij - просто переймаємо значення до збільшення k.

Випадок 2. Елемент k входить в найкоротший шлях p_{ij}, тобто після додавання нової вершини в множину дозволених, найкоротший шлях змінився і проходить тепер через вершину v_k. Яку вартість отримає новий шлях?

Новий найкоротший шлях розбитий вершиною v_k на p_{ik} і p_{kj}, використовуємо перша властивість, згідно з ним, p_{ik} і p_{kj} також найкоротші шляхи від vi до v_k і від v_k до v_j відповідно. Значить

$dk_{ij} = dk_{ik} + dk_{kj}$

А так як в цих шляхах k або кінцевий, або початковий вузол, то він не входить в безліч проміжних, відповідно його з нього можна видалити:

$dk_{ij} = dk\text{-}1_{ik} + d_k\text{-}1_{kj}$ [11].

Як видно алгоритм дуже простий - спочатку відбувається ініціалізація матриці найкоротших відстаней D0, спочатку вона збігається з матрицею

суміжності, в циклі збільшуємо значення k і перераховуємо матрицю відстаней, з D^0 отримуємо D^1, з D^1 - D^2 і так далі до k = n.

Передбачається, що якщо між двома якимись вершинами немає ребра, то в матриці суміжності було записано якесь велике число (досить велика, щоб воно було більше довжини будь-якого шляху в цьому графі); тоді це ребро завжди буде невигідно брати, і алгоритм спрацює правильно. Правда, якщо не вжити спеціальних заходів, то при наявності в графі ребр негативного ваги, в результуючої матриці можуть з'явитися числа виду ∞-1, ∞-2, і т.д., які, звичайно, як і раніше означають, що між відповідними вершинами взагалі немає шляху.

Тому за наявності в графі негативних ребр алгоритм Флойда краще написати так, щоб він не виконував переходи з тих станів, у яких вже стоїть «немає шляху»

Алгоритмом потрібно $O(n^3)$ пам'яті, для збереження матриць. Однак кількість матриць можна легко скоротити до двох, щоразу переписуючи непотрібну матрицю або взагалі перейти до двомірної матриці, прибравши індекс k у dk_{ij}. Кращий варіант, який найчастіше використовується - писати відразу в матрицю суміжності, тоді нам зовсім не потрібна додаткова пам'ять, правда якщо відразу переписувати початкову матрицю, то потрібно додатково показати коректність алгоритму, так як класичний академічний доказ вірний тільки для випадку, коли матриця попередній ітерації не змінюється.

Що стосується часу роботи - трьох вкладених циклу від 1 до n - $\Theta(n^3)$ [2].

Якщо в графі є цикли негативного ваги, то формально алгоритм Флойда-Воршолла до такого графу непридатний. Але насправді алгоритм коректно спрацює для всіх пар, шляхи між якими ніколи не проходять через цикл негативної вартості, а для решти ми отримаємо якісь числа, можливо сильно негативні. Алгоритм можна навчити виводити для таких пар якесь значення, відповідне -∞

До речі після відпрацювання такого графа на діагональ матриці найкоротших шляхів виникнуть негативні числа - найкоротша відстань від вершини в цьому циклі до неї самої буде менше нуля, що відповідає проходу по цьому циклу, так що алгоритм можна використовувати для визначення наявності негативних циклів у графі.

Матриця відстаней покаже нам найкоротшу (найдешевшу) відстань для будь-якої пари вершин, а як же дізнатися шлях? Дуже просто, при розрахунку dk_{ij} потрібно розрахувати ще й πk_{ij}. πk_{ij} при цьому - попередник вершини v_j на шляху від v_i з безліччю дозволених проміжних вершин {1..k}.

Як і будь-який базовий алгоритм, алгоритм Флойда-Воршолла використовується дуже широко і багато де, починаючи від пошуку транзитивного замикання графа, закінчуючи генетикою і управлінням проектами. Але перше що приходить в голову звичайно ж транспортні і всякі інші мережі.

Скажімо якщо ви візьмете карту міста - її транспортна система це граф, відповідно привласнивши кожному ребру якусь вартість, розраховану скажімо з пропускної спроможності та інших важливий параметрів - ви зможете підвести попутника по самому короткому / швидкому / дешевому шляху [11].

2 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Структура даних і класифікація основних алгоритмів заданого класу

Алгоритм Флойда-Воршолла порівнює всі можливі шляхи в графі між кожною парою вершин. Він виконується в $\Theta(|V|^3)$порівнянь. Це доволі примітивно, враховуючи , що в графі може бути до $\Omega(|V|^2)$ ребер, і кожну комбінацію буде протестовано. Він виконує це шляхом поступового поліпшення оцінки по найкоротшому шляху між двома вершинами, поки оцінка не є оптимальною [11].

Розглянемо граф G з ребрами V, пронумерованими від 1 до N. Крім того розглянемо функцію shortestPath(i, j, k), яка повертає найкоротший шлях від i до j, використовуючи вершини з множини {1,2,...,k} як внутрішні у шляху.

Тепер, маючи таку функцію нам потрібно знайти найкоротший шлях від кожного i до кожного j, використовуючи тільки вершини від 1 до k + 1.

Для кожної з цих пар вершин, найкоротший шлях може бути або (1)- шлях, у якому є тільки вершини з множини {1, ..., k}, або (2)- шлях, який проходить від i до k + 1 а потім від k + 1 до j. Найкоротший шлях від i to j that only uses vertices 1 через k визначається функцією shortestPath(i, j, k),і якщо є коротший шлях від i до (k + 1 до j),то довжина цього шляху буде сумою(конкатенацією) найкоротшого шляху від i до k + 1 (використовуючи вершини{1, ..., k}) і найкоротший шлях від k + 1 до j (також використовуючи вершини з {1, ..., k}).

w(i,j)- це вага ребра між i та j.

Ця формула є основною частиною алгоритму Флойда-Воршолла. Алгоритм спочатку обчислює shortestPath(i, j, k) для всіх пар(i, j) де k = 1, потім k = 2, і так далі. Цей процес продовжується, поки k = N, і поки не знайдено всі найкоротші шляхи для пар (i, j). [17]

Найбільш відомим способом представлення графа на папері є геометричне зображення точок та ліній. Але нам необхідно якимось чином подавати граф для комп'ютерної обробки.

Класичним способом такого представлення графа є матриця інциденції. Для графа з n вершинами та m ребрами - це матриця A з n стрічками, що відповідають вершинам, та m стовпцями, що відповідають ребрам. Для орієнтованого графа стовпець, що відповідає ребру (x, y), містить -1 в стрічці, що відповідає вершині x, 1 в стрічці, що відповідає вершині y, і нулі в усіх інших стрічках. У випадку неорієнтованого графа стовпець, що відповідає ребру {x, y}, містить 1 в стрічках, що відповідають x та y, та нулі в інших стрічках.

З алгоритмічної точки зору матриця інциденції є, ймовірно, найгіршим способом представлення графа, який тільки можна собі уявити. По-перше, він вимагає mn комірок пам'яті, причому більшість цих комірок взагалі зайняті нулями. Незручним також є доступ до інформації. Відповідь на елементарне питання типу «чи існує дуга (x, y)?», «до яких вершин ведуть ребра з x?» вимагають в найгіршому випадку перегляду усіх стовпців матриці, а отже, m кроків.

Кращим способом представлення графа є матриця суміжності. Говорять, що вершини i та j у графі суміжні, якщо існує ребро, що їх з'єднує. Говорять, що два ребра суміжні, якщо вони мають спільну вершину. Тоді для графа з n вершинами та m ребрами матрицею суміжностей буде матриця $B=[b_{ij}]$ розміру n*n, де $b_{ij}=1$, якщо існує ребро, що йде від вершини i до вершини j, та $b_{ij}=0$ у іншому випадку. Оскільки у неорієнтованому графі ребро {i, j} веде як від i до j, так і від j до i, тому матриця суміжності такого графа завжди є симетричною [11].

2.2 Опис використаних алгоритмів

Алгоритм Флойда-Воршолла – динамічний алгоритм для знаходження найкоротших відстаней між всіма вершинами зваженого орієнтованого графа.

$w(i, j)$ - це вага ребра між *i* та *j*. Можна визначити shortestPath (*i*, *j*, *k* + 1) наступною рекурсивною формулою база shortestPath(i, j, 0)=w(i, j).

Ця формула є основною частиною алгоритму Флойда-Воршолла. Алгоритм спочатку обчислює shortestPath(*i*, *j*, *k*) для всіх пар(*i*, *j*) де *k* = 1, потім *k* = 2, і т.д.

Цей процес продовжується, поки $k = N$, і поки не знайдено всі найкоротші шляхи для пар (i, j) [17].

Псевдокод для цієї версії алгоритму:

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
```

Алгоритм Флойда-Воршолла зазвичай знаходить тільки довжину шляху між усіма парами вершин. За допомогою простих змін, можна створити функцію для відновлення фактичного шляху між будь-якими двома кінцевими точками вершин [17].

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of vertex indices initialized to null
procedure FloydWarshallWithPathReconstruction ()
```

```
  for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
    next[u][v] ← v
  for k from 1 to |V| // standard Floyd-Warshall implementation
    for i from 1 to |V|
      for j from 1 to |V|
        if dist[i][k] + dist[k][j] < dist[i][j] then
          dist[i][j] ← dist[i][k] + dist[k][j]
          next[i][j] ← next[i][k]
```

Псевдокод procedure Path(u, v)

```
  if next[u][v] = null then
    return []
  path = [u]
  while u ≠ v
    u ← next[u][v]
    path.append(u)
  return path
```

Приклад роботи алгоритму Флойда-Воршолла.

Необхідно знайти найкоротші шляхи між кожною парою вершин у графі, представленому на наступному рисунку

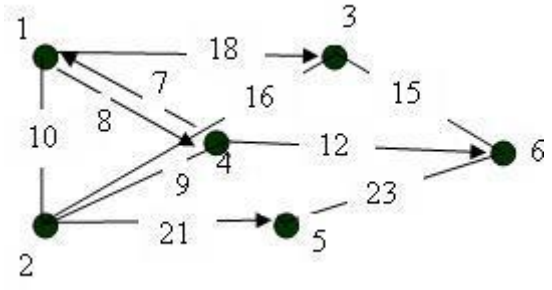


Рисунок 2.1 – Вихідний граф

Пронумеруємо всі вершини графа, і складемо матрицю довжин найкоротших дуг D^0, у випадку, якщо дуги між вершиною i і j не існує, елементу di, j матриці присвоюється значення ∞. Отримаємо матрицю D^0 [19].

На підставі матриці D^0, обчислимо послідовно всі елементи матриці D1. Для цього ми використовуємо рекурентне співвідношення $d_{i,j}^1=\min\{ d_{i,1}^0+ d_{1,j}^0; d_{i,j}^0\}$.

$d_{1,1}^1=\min\{d_{1,1}^0+d_{1,1}^0 d_{1,1}^0\}=\min\{0+0; 0\}=0$

$d_{1,2}^1=\min\{d_{1,1}^0+d_{1,2}^0 d_{1,2}^0\}=\min\{0+10; 10\}=10$

$d_{1,3}^1=\min\{d_{1,1}^0+d_{1,3}^0 d_{1,3}^0\}=\min\{0+18; 18\}=18$

$d_{1,4}^1=\min\{d_{1,1}^0+d_{1,4}^0 d_{1,4}^0\}=\min\{0+8; 8\}=8$

$d_{1,5}^1=\min\{d_{1,1}^0+d_{1,5}^0 d_{1,5}^0\}=\min\{0+\infty; \infty\}=\infty$

$d_{6,6}^1=\min\{d_{6,1}^0+d_{1,6}^0 d_{6,6}^0\}=\min\{\infty+\infty; 0\}=0$

Представимо матрицю D^1, включивши в неї розраховані елементи та на підставі матриці D^1, обчислимо послідовно всі елементи матриці D^2. Для цього ми використовуємо рекурентне співвідношення $d_{i,j}^2=\min\{ d_{i,2}^1+ d_{2,j}^1; d_{i,j}^1\}$.

Аналогічно представимо матрицю D^2, включивши в неї розраховані елементи.

На підставі матриці D^2 обчислимо послідовно всі елементи матриці D^3. Для цього ми використовуємо рекурентне співвідношення $d_{i,j}^3=\min\{ d_{i,3}^2+ d_{3,j}^2; d_{i,j}^2\}$.

Продовжуємо розрахунки, поки не отримаємо матрицю D^6 [19].

В результаті, нами отримана матриця довжин найкоротших шляхів між кожною парою вершин графа. Нижче представлена таблиця шляхів. Кожен елемент C_{ij} таблиці, це шлях з вершини i в вершину j:

i \ j	1	2	3	4	5	6
1	-	d_{1-2}=1-2	d_{1-3}=1-3	d_{1-4}=1-4	d_{1-5}=1-2-5	d_{1-6}=1-4-6
2	d_{2-1}=2-1	-	d_{2-3}=2-3	d_{2-4}=2-4	d_{2-5}=2-5	d_{2-6}=2-4-6
3	d_{3-1}=3-2-1	d_{3-2}=3-2	-	d_{3-4}=3-2-4	d_{3-5}=3-2-5	d_{3-6}=3-6
4	d_{4-1}=4-1	d_{4-2}=4-2	d_{4-3}=4-1-3	-	d_{4-5}=4-2-5	d_{4-6}=4-6
5	d_{5-1}=5-6-3-2-1	d_{5-2}=5-6-3-2	d_{5-3}=5-6-3	d_{5-4}=5-6-3-2-4	-	d_{5-6}=5-6
6	d_{6-1}=6-3-2-1	d_{6-2}=6-3-2	d_{6-3}=6-3	d_{6-4}=6-3-2-4	d_{6-5}=6-5	-

Рисунок 2.1 – Вихідна матриця шляхів

Таким чином ми отримали матрицю найкоротших шляхів між всіма вершинами зваженого орієнтованого графа.

3 ОПИС РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Визначення варіантів використання програмного забезпечення

Основний варіант використання полягає у демонстрації роботи алгоритму, тобто, безпосередньо, задачу курсової роботи.

Повний перелік варіантів використання:

– відкриття відео файл, для зчитування стін лабіринту;

– налаштування, тобто вибір кількості стін, які будуть зчитані;

– побудова найкоротшого шлях між початковою та кінцевою точкою;

– вибір початкової та кінцевої точки;

– відображення інформації про додаток [9].

На рисунку 3.1 показана діаграма використання до розробленого ПЗ.

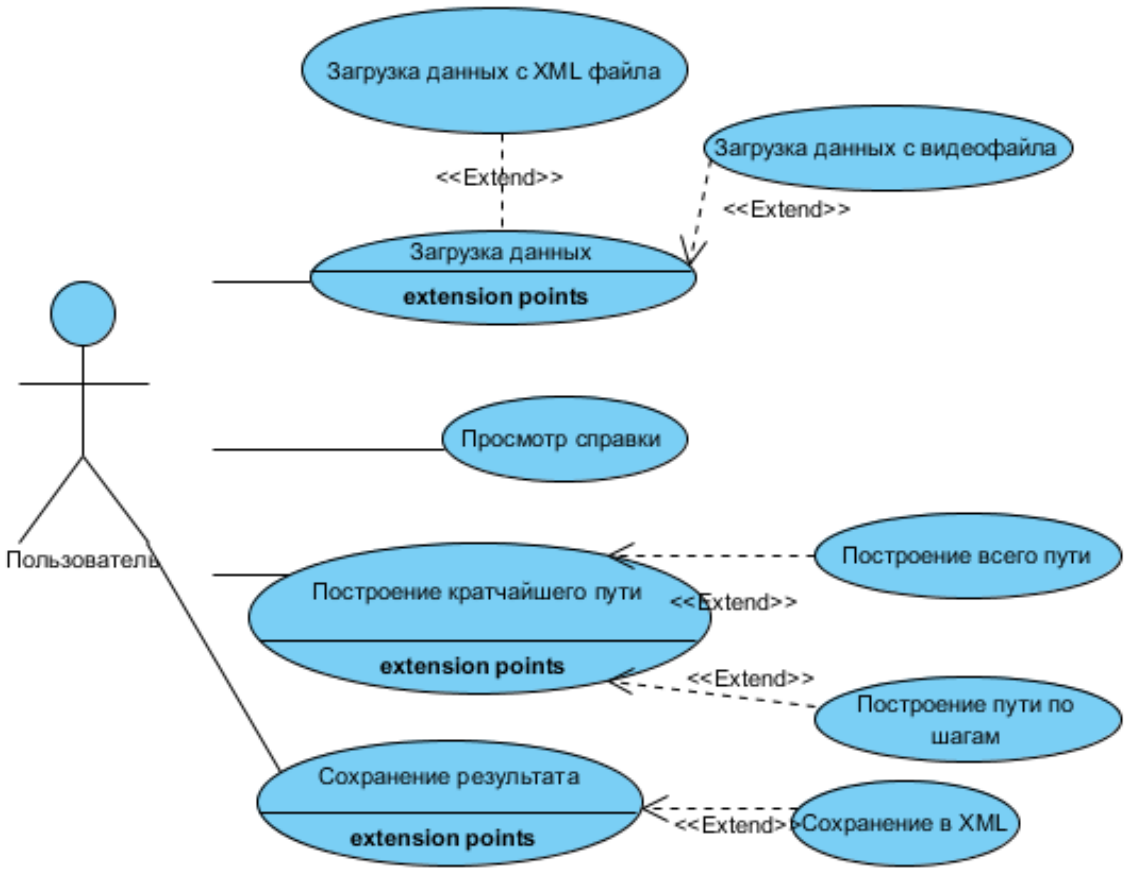


Рисунок 3.1 – Діаграма використання

3.2 Обґрунтування вибору ОС і засобів розробки ПЗ

В ролі платформи для програмного забезпечення була вибрана операційна система Windows. Вибор був обумовлений тим що ОС Windows встановлена на більшості комп’ютерів користувачів.

Microsoft Windows — узагальнююча назва операційних систем для ЕОМ, розроблених корпорацією Microsoft. Перші версії були не повноцінними операційними системами, а лише оболонками до ОС MS-DOS. На 2014 рік, за даними сайтів NetApplications та GoStats, Microsoft Windows встановлена більш як на 90% персональних комп'ютерів світу [13].

Для розробки програмного забеспечення було вирішено обрати мову C#.

C# (вимовляється Сі-шарп) — об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research (при фірмі Microsoft).

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. Перейнявши багато що від своїх попередників — мов C++, Delphi, Модула і Smalltalk — C#, спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад множинне спадкування класів (на відміну від C++).

C# є дуже близьким родичем мови програмування Java. Мова Java була створена компанією Sun Microsystems, коли глобальний розвиток інтернету поставив задачу роззосереджених обчислень. Взявши за основу популярну мову C++, Java виключила з неї потенційно небезпечні речі (типу вказівників без контролю виходу за межі). Для роззосереджених обчислень була створена концепція віртуальної машини та машинно-незалежного байт-коду, свого роду посередника між вихідним текстом програм і апаратними інструкціями комп'ютера чи іншого інтелектуального пристрою [12].

Java набула чималої популярності, і була ліцензована також і компанією Microsoft. Але з плином часу Sun почала винуватити Microsoft, що та при створенні свого клону Java робить її сумісною виключно з платформою Windows, чим суперечить самій концепції машинно-незалежного середовища виконання і порушує ліцензійну угоду. Microsoft відмовилася піти назустріч вимогам Sun, і тому з'ясування стосунків набуло статусу судового процесу. Суд визнав позицію Sun справедливою, і зобов'язав Microsoft відмовитися від позаліцензійного використання Java.

У цій ситуації в Microsoft вирішили, користуючись своєю вагою на ринку, створити свій власний аналог Java, мови, в якій корпорація стане повновладним господарем. Ця новостворена мова отримала назву C#. Вона успадкувала від Java концепції віртуальної машини (середовище .NET), байт-коду (MSIL) і більшої безпеки вихідного коду програм, плюс врахувала досвід використання програм на Java.

Нововведенням C# стала можливість легшої взаємодії, порівняно з мовами-попередниками, з кодом програм, написаних на інших мовах, що є важливим при створенні великих проектів. Якщо програми на різних мовах виконуються на платформі .NET, .NET бере на себе клопіт щодо сумісності програм (тобто типів даних, за кінцевим рахунком).

Станом на сьогодні C# визначено флагманською мовою корпорації Microsoft, бо вона найповніше використовує нові можливості .NET. Решта мов програмування, хоч і підтримуються, але визнані такими, що мають спадкові прогалини щодо використання .NET.

C# розроблялась як мова програмування прикладного рівня для CLR і тому вона залежить, перш за все, від можливостей самої CLR. Це стосується, перш за все, системи типів C#. Присутність або відсутність тих або інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльована у відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C#; подібної взаємодії слід чекати і надалі. (Проте ця закономірність буде порушена з виходом C# 3.0, що є розширеннями мови, що не

спираються на розширення платформи .NET.) CLR надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, збірка сміття не реалізована в самому C#, а проводиться CLR для програм, написаних на C# точно так, як і це робиться для програм на VB.NET, J# тощо [13].

Для написання коду було обране середовище розробки Visual Studio 2015 від Microsoft.

Microsoft Visual Studio — серія продуктів фірми Майкрософт, які включають інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів. Ці продукти дозволяють розробляти як консольні програми, так і програми з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight [13].

Visual Studio включає один або декілька з наступних компонентів (вибір робиться на етапі інсталювання):

– Visual Basic .NET, а до його появи — Visual Basic;

– Visual C++;

– Visual C#;

– Visual J#;

– Visual F# (входить до складу Visual Studio 2010);

– Visual Studio Debugger.

Багато варіантів постачання також включають:

– Microsoft SQL Server;

– MSDE Visual Source Safe — файл-серверна система управління версіями.

У минулому, до складу Visual Studio також входили продукти:

– Visual InterDev;

– Visual J++;

– Visual J#;

– Visual FoxPro;

– Visual Source Safe – файл-серверна система управління версіями.

Visual Studio 2012 представлений 2 серпня 2012 року. Включає .NET Framework 4.5. Головні нововведення це підтримка Windows RunTime, C++/CX(Component Extensions), бібліотека C++ AMP для GPGPU програмування, компілятор Visual C++ майже підтримує стандарт C++11. З'явився новий тип проектів, котрі дозволяють писати рідні застосунки (у стилі Windows Metro) для операційної системи Windows 8.

Також встановлений додаток до Visual Studio 2012 – ReSharper (R#) від компанії JetBrains.

ReSharper (R#) — додаток (add-on), розроблений компанією JetBrains для збільшення продуктивності роботи та автоматизації рефакторингу в середовищі Microsoft Visual Studio (підтримуються версії: 2003, 2005, 2008, 2010, 2012, 2013, 2015).

Здійснює миттєвий статичний аналіз коду (без потреби компіляції), передбачає додаткові засоби автозаповнення, навігації, пошуку, виділення синтаксису, форматування, оптимізації та генерації коду, надає близько 40 автоматизованих рефакторингів, спрощує модульне тестування в середовищах MSTest та NUnit [14].

Для розробки графічного інтерфейсу програми було обрано API Windows Forms.

Windows Forms - прикладний програмний інтерфейс (API), що відповідає за графічний інтерфейс користувача і є частиною Microsoft .NET Framework. Даний інтерфейс спрощує доступ до елементів інтерфейсу Microsoft Windows за рахунок створення обгортки для існуючого Win32 API в керованому коді. Причому керований код - класи, що реалізують API для Windows Forms, не залежать від мови розробки. Тобто програміст однаково може використовувати Windows Forms як при написанні ПЗ на C #, C ++, так і на VB.Net, J # та інші.

З одного боку, Windows Forms розглядається як заміна більш старої і складною бібліотеці MFC, спочатку написаної мовою C ++. З іншого боку, WF не

пропонує парадигму, порівнянну з MVC. Для виправлення цієї ситуації і реалізації даного функціоналу в WF існують сторонні бібліотеки. Однією з найбільш використовуваних подібних бібліотек є User Interface Process Application Block, випущена спеціальною групою Microsoft, що займається прикладами і рекомендаціями, для безкоштовного скачування. Ця бібліотека також містить вихідний код і навчальні приклади для прискорення навчання.

Усередині .NET Framework, Windows Forms реалізується в рамках простору назв System.Windows.Forms.

Як і Abstract Window Toolkit (AWT) (схожий API для мови Java), бібліотека Windows Forms була розроблена як частина .NET Framework для спрощення розробки компонентів графічного інтерфейсу користувача. Windows Forms побудована на основі застаріваючого Windows API і являє собою, по суті, обгортку низькорівневих компонентів Windows.

Windows Forms надає можливість розробки кроссплатформенного графічного інтерфейсу користувача. Однак, Windows Forms фактично є лише обгорткою Windows API-компонентів, і ряд її методів здійснюють прямий доступ до Win32-функціям зворотного виклику, які недоступні на інших платформах [15].

У .NET Framework версії 2.0 бібліотека Windows Forms отримала більш багатий інструментарій розробки інтерфейсів, toolstrip-елементи інтерфейсу в стилі Office 2003, підтримку багатопоточності, розширені можливості проектування і прив'язки до даних, а також підтримку технології ClickOnce для розгортання веб-додатків.

З виходом .NET Framework 3.0 Microsoft випустила новий API для малювання користувальницьких інтерфейсів: Windows Presentation Foundation, який базувався на DirectX 11 і декларативну мову опису інтерфейсів XAML. Однак, навіть незважаючи на все це, Windows Forms і WPF все ще пропонують схожу функціональність, і тому Windows Forms НЕ був скасований на користь WPF, а продовжує використовуватися як альтернативна технологія побудови інтерфейсів поряд з WPF.

Відповідаючи на питання на конференції Build +2014, Майкрософт пояснила, що Windows Forms буде підтримуватися, помилки будуть виправлятися, але нові функції додаватися не будуть. Пізніше, поліпшена підтримка високого дозволу для різних елементів інтерфейсу Windows Forms все ж була анонсована в релізі .NET Framework 4.5 [13].

3.3 Структура застосування

Для опису основних компонентів і взаємозв'язків розробленого ПЗ необхідно створити діаграму класів, варіантів використання та компонентів.

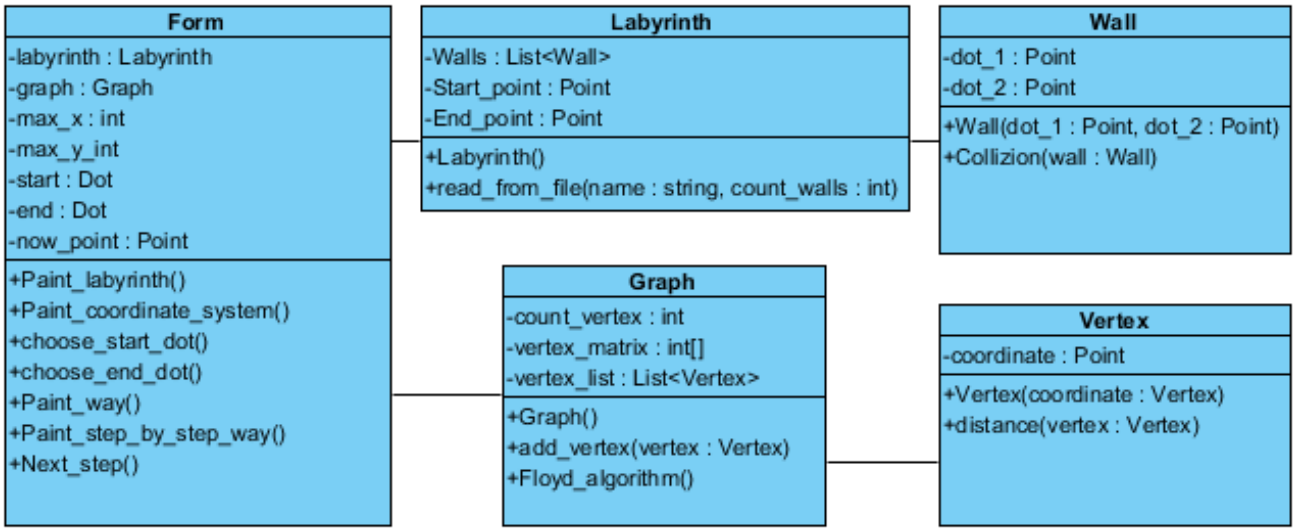


Рисунок 3.2 – Діаграма класів

На рисунку 3.2 зображено діаграму класів до створеного ПЗ. Так, як створення ПЗ було реалізовано мовою C# з допомогою .NET і використанням Windows Forms, весь функціонал програми знаходиться в класі Form.

В класі Form знаходяться методи Paint_labyrinth, який малює лабіринт, метод Paint_coordinate_system, який малює динамічну систем координат та методи choose_start та choose_end для задання початку та кінцю шуканого шляху відповідно.

Метод Paint_way малює найкоротший шлях від початкової до кінцевої точки, якщо такі задані, а метод Paint_step_by_step_way малює шлях по крокам де за визначення наступного кроку відповідає функція Next_step.

Методи choose_start_dot та choose_end_dot задають початкову та кінцеву точку лабіринту відповідно.

Клас Labyrinth представляє собою задання лабіринту у вигляді списку стін та в ньому зберігаються початкова та кінцева точка лабіринту у вигляді об’єктів класу Point.

Клас Graph представляє собою сутність для представлення графу, тобто даного лабіринту у вигляді графу. Він містить у собі матрицю суміжності з вагами ребер, кількість вершин графа та список цих вершин.

Клас Vertex – це клас для представлення вершини графу. Містить у собі координати вершини та метод визначення відстані до будь-якої іншої вершини.

Клас Wall представляє собою сутність для представлення стіни лабіринту. Він містить у собі дві точки – початок та кінець стіни в об’єктах класа Dot. Цей клас містить у собі інформацію про одну точку, а точніше її координату по осі x та y.

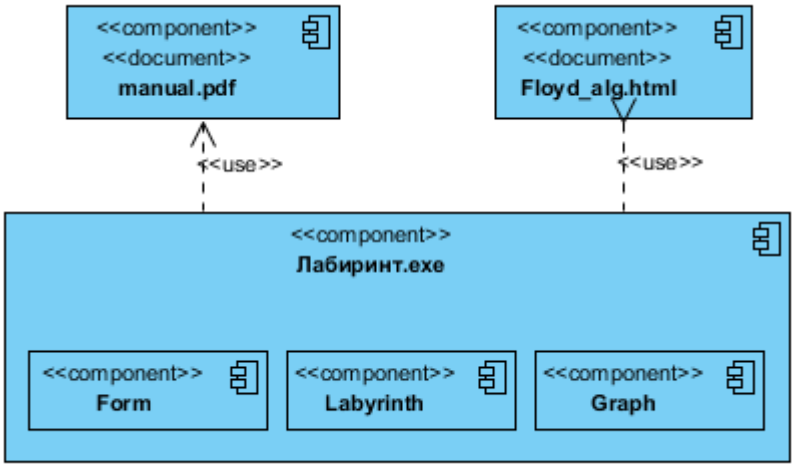


Рисунок 3.3 – Діаграма компонентів

На рисунку 3.3. зображена діаграма компонентів, яка відображає зв’язки між компонентами програми. Як видно з діаграми, головним файлом є файл

Лабиринт.exe, який вміщує в собі 3 компоненти, які є класами даного файлу. Докладніше про ці класі було розказано в описі діаграми класів. До цього файлу підключається файл, який при викликанні «About» у програмі відкриває *.html файл з необхідною інформацією та файл Manual.pdf, який являє собою інструкцію користувача.

4 ВИКОРИСТАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Установка програмного забезпечення

У процесі виконання курсової роботи було розроблена portable-версія програмного забезпечення, тобто воно не потребує установки, а лише копіювання його файлів на комп’ютер.

Вимоги для програмного забезпечення наведені нижче.

1) Операційна система:

– рекомендовані вимоги: Windows 7, Windows 8, Windows 10;

– мінімальні вимоги: Windows XP.

2) .NET Framework

– рекомендовані вимоги: .NET Framework 4.5.1;

– мінімальні вимоги: .NET Framework 4.0 (на більш старих версіях ПЗ може відображатись не зовсім коректно).

Вимоги до апаратного забезпечення наведені нижче.

1) Процесор:

– рекомендовані вимоги: 1.5 ГГц;

– мінімальні: 1 ГГц.

2) Оперативна пам'ять (RAM):

– рекомендовані вимоги: 1гб;

– мінімальні вимоги: 512 мб.

3) На жорсткому диску ПЗ займе не більше 10-ти мб.

Програма не потребує установки на комп'ютер користувача. Достатньо перемістити папку з програмою в будь-яке місце на жорсткому диску, після чого запустити Лабиринт.exe.

Для того, щоб закрити програму необхідно натиснути хрестик в верхньому правому куті вікна, або зайти в меню «Файл => Выход».

Обидва варіанти проілюстровані на рисунку 4.1.

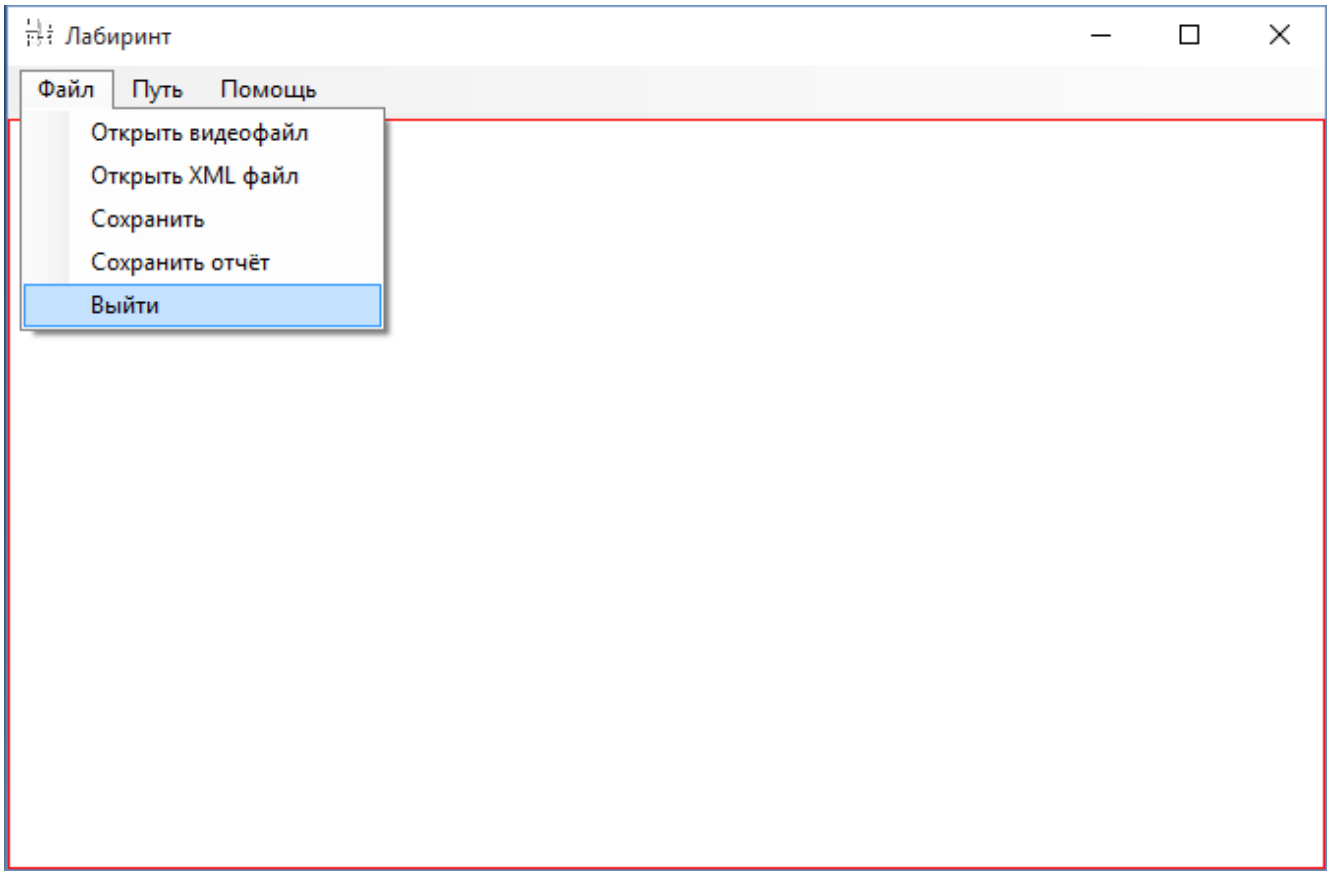


Рисунок 4.1 – Демонстрація виходу з програми

Більш детальніші інструкції щодо роботи з розробленим ПЗ буде наведено в наступному підрозділі.

4.2 Інструкція користувачеві

Лабіринт – програма візуалізації знаходження найкоротшого шляху в двовимірному лабіринті за допомогою алгоритму Флойда-Воршолла.

Вхідними даними є зчитані числа з відео файлу одного з запропонованих форматів: *.avi, *.mkv, *.mp4 – данні якого зчитуються як цілі числа у форматі Int.

Додаткові можливості: приближення та віддалення лабіринту, перегляд допомоги.

Інтерфейс програми зображений на рисунку 4.2.

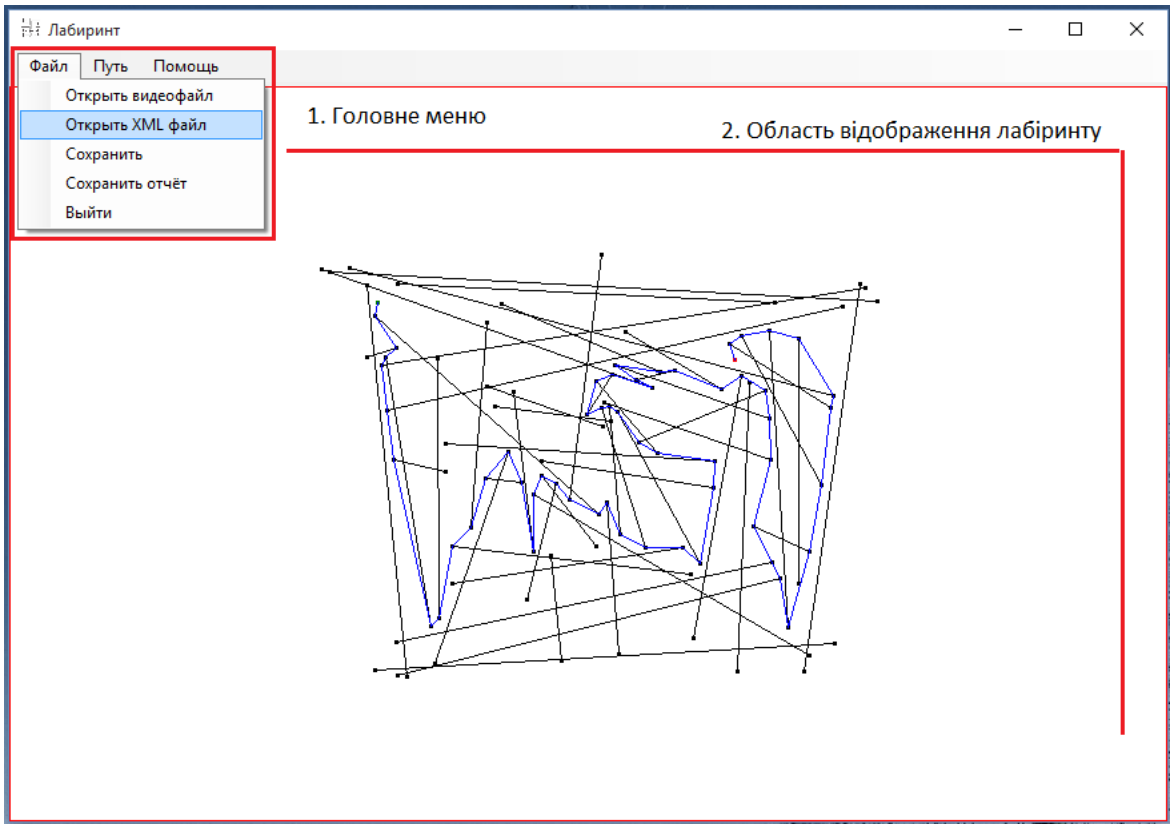


Рисунок 4.2 – Інтерфейс програми

Далі описано основні розділи роботи з програмою.

При натисканні на кнопку «Файл» відкривається меню (рисунок 4.3):

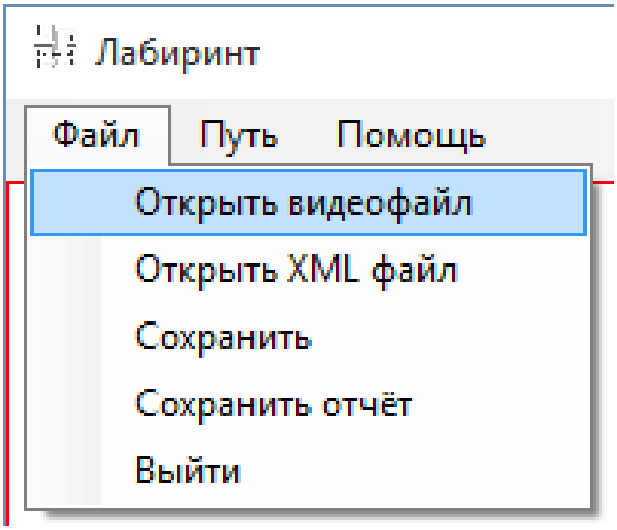


Рисунок 4.3 – Меню «Файл»

Маємо 3 функції: «Открыть видеофайл», «Открыть XML файл» , «Сохранить» , «Сохранить отчёт» та «Выход».

Функція «Открыть видеофайл » відкриває діалогове вікно Открытие файла для вибору кількості елементів для зчитування (рисунок 4.4).

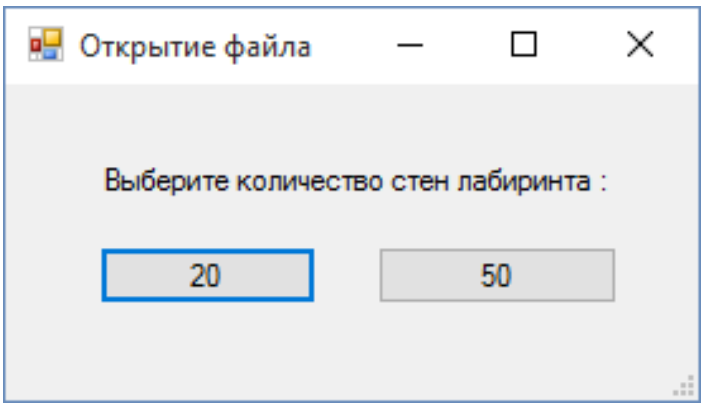


Рисунок 4.4 – Діалогове вікно «Открытие файла»

Після цього відкривається стандартний діалог відкриття файлу. Якщо файл буде успішно відкритий, на області відображення (рисунок 4.2 пункт 2) відобразиться лабіринт.

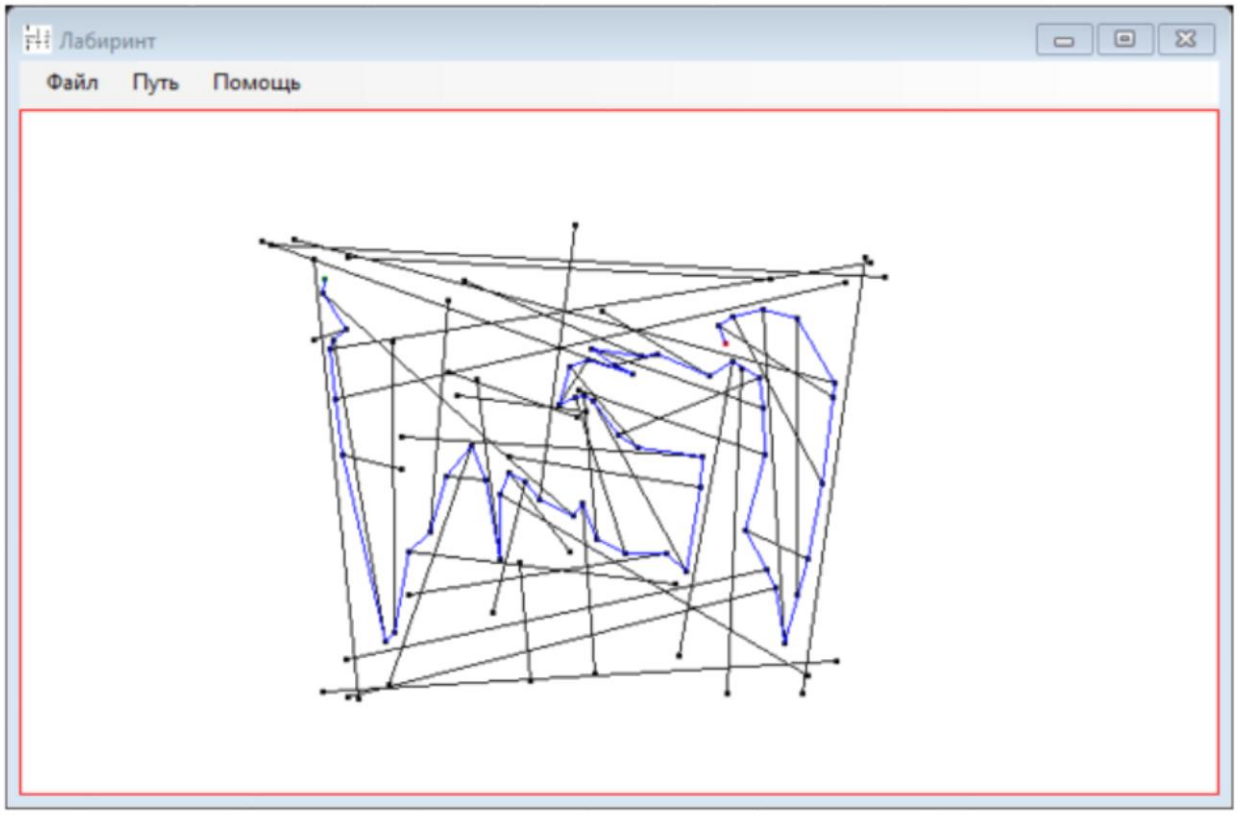
При виборі пункту меню «Открыть XML файл» відкривається діалогове вікно для вибору файлу формату XML для зчитування.

При виборі пункту меню «Сохранить» відкривається діалогове вікно для вибору файлу формату XML для запису у нього даних програми.

При виборі пункту меню «Сохранить отчёт» генерується звіт (рисунок 4.5), в якому відображені

Результат построения пути :

Снимок экрана :



Все точки	Точки, входящие в путь
-37 , 31	-37 , 31
10 , 207	-38 , 39

Рисунок 4.6 – Зразок звіту

Функція «Выход» зупиняє роботу програми.

При натисканні на кнопку «Путь» відкривається меню (рисунок 4.7):

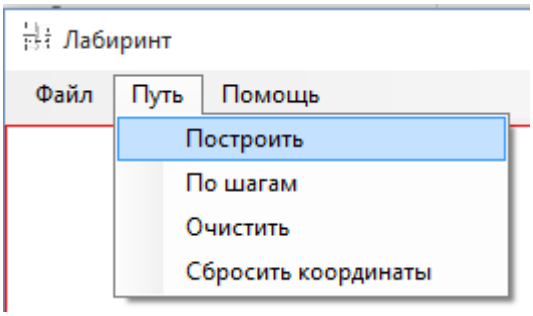


Рисунок 4.7 – Меню «Путь»

Маємо пункти меню «Построить», «По шагам» , «Очистить» , «Сбросить координаты».

При натисканні на елемент «Построить» буде побудований найкоротший шлях в лабіринті, якщо лабіринт був завантажений та вибрана початкова та кінцева точка.

При натисканні на елемент «По шагам» буде покроково побудований найкоротший шлях в лабіринті, якщо лабіринт був завантажений та вибрана початкова та кінцева точка.

При натисканні на елемент «Очистить» шлях буде очищений та залишено лише сам лабіринт.

При натисканні на елемент «Сбросить координаты» буде очищена динамічна система координат та повернута до початкового стану.

При натисканні на кнопку «Помощь» відкривається меню (рисунок 4.8):

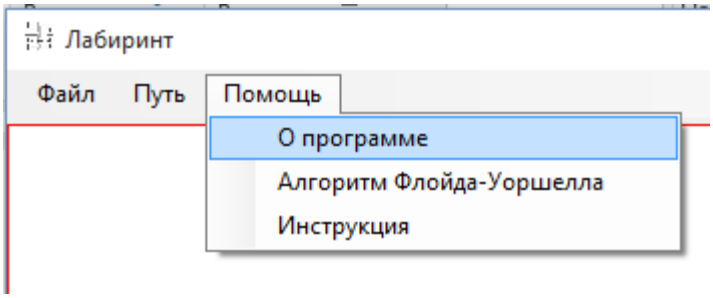


Рисунок 4.8 – Меню «Помощь»

Маємо три пункти меню: «О программе», «Алгоритм Флойда-Уоршелла» та «Инструкция».

При натисканні на пункт меню «О программе» відкривається вікно «Информация», в якому показана основна інформація по програмі (рисунок 4.9).

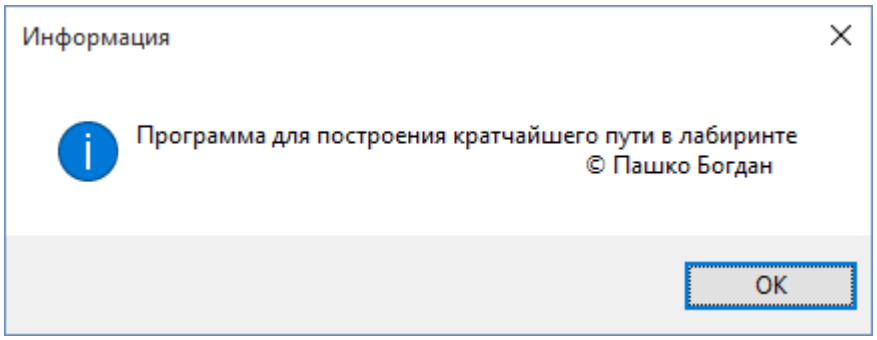


Рисунок 4.9 – Вікно «Информация»

При натисканні на пункти «Алгоритм Флойда-Уоршелла» відкривається сторінки в браузері по-замовчуванню із інформацією про алгоритм Флойда-Воршолла.

При виборі пункту «Инструкция» відкривається файл Manual.pdf – інструкція користувача.

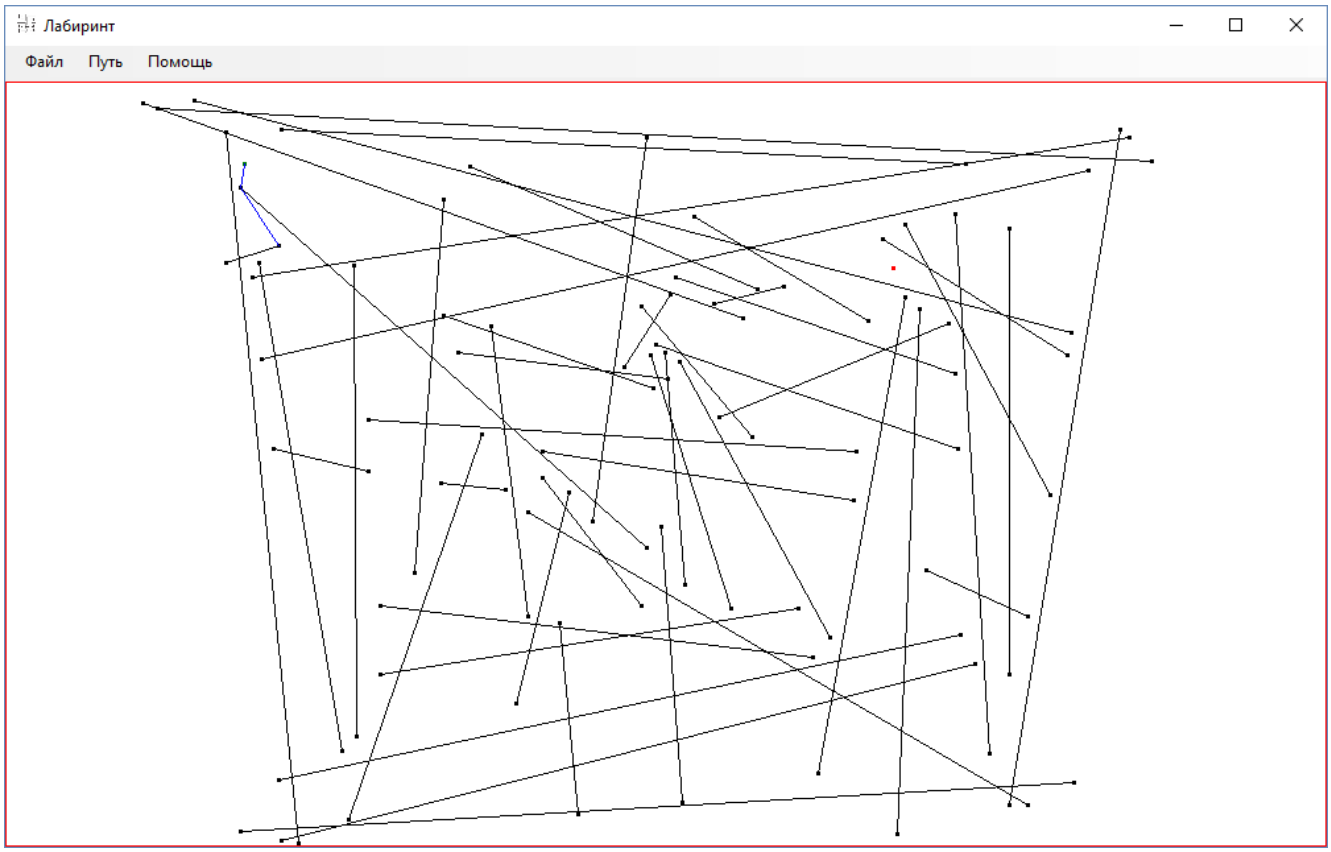
На області відображення (рисунок 4.2 пункт 2) розміщений лабіринт який ми зчитали з відео файлу.

Лабіринт представляє собою стінки з точками на кінцях, за які ми можемо тягнути мишкою та тим самим змінювати дану стінку та лабіринт.

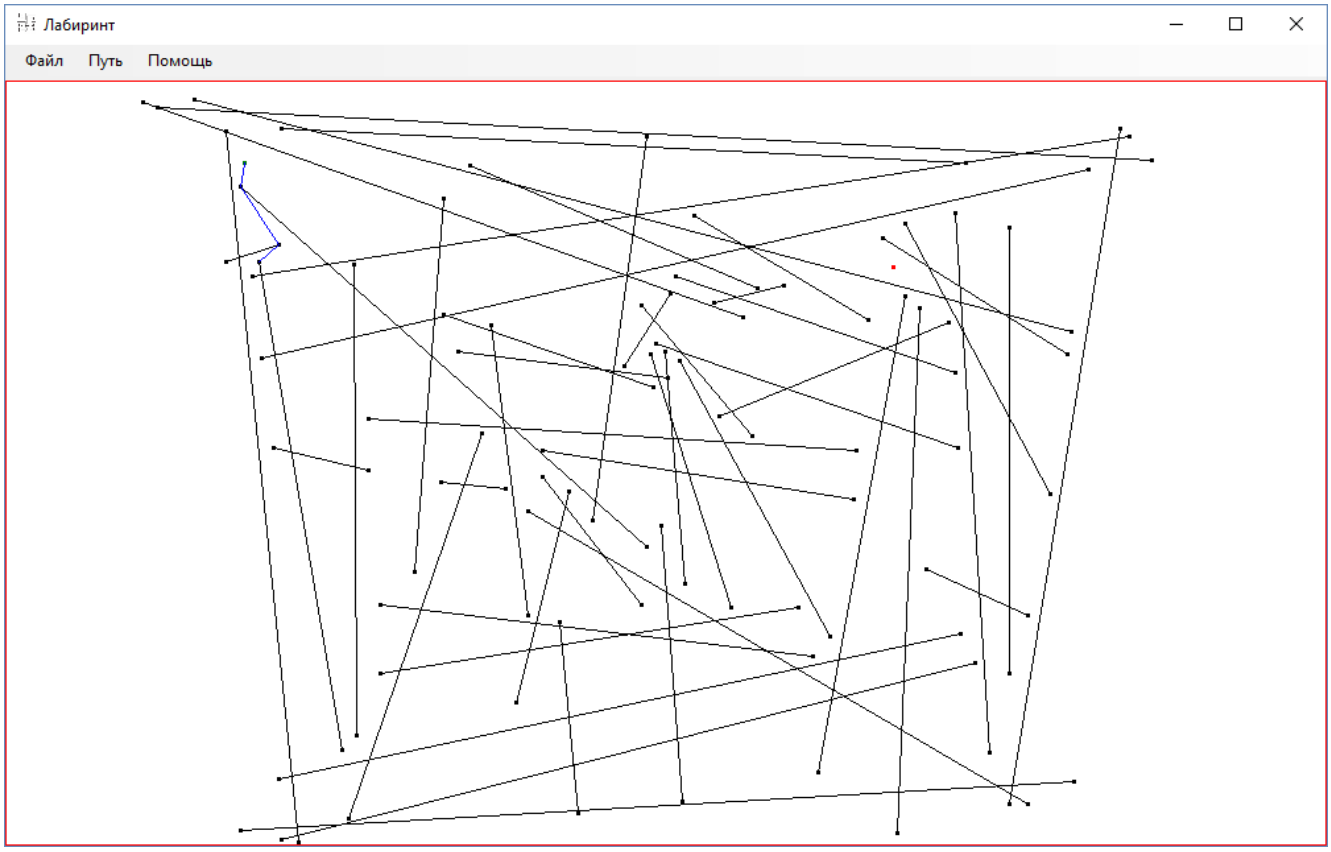
4.3 Аналіз результатів

Побудова найкоротшого шляху в лабіринті– проблема перебору всіх можливих варіантів шляху та вибору найкращого. Існує багато алгоритмів, які дозволяють вирішити цю проблему. Розроблене ПЗ реалізує алгоритми Флойда-Воршолла та його покрокову демонстрацію

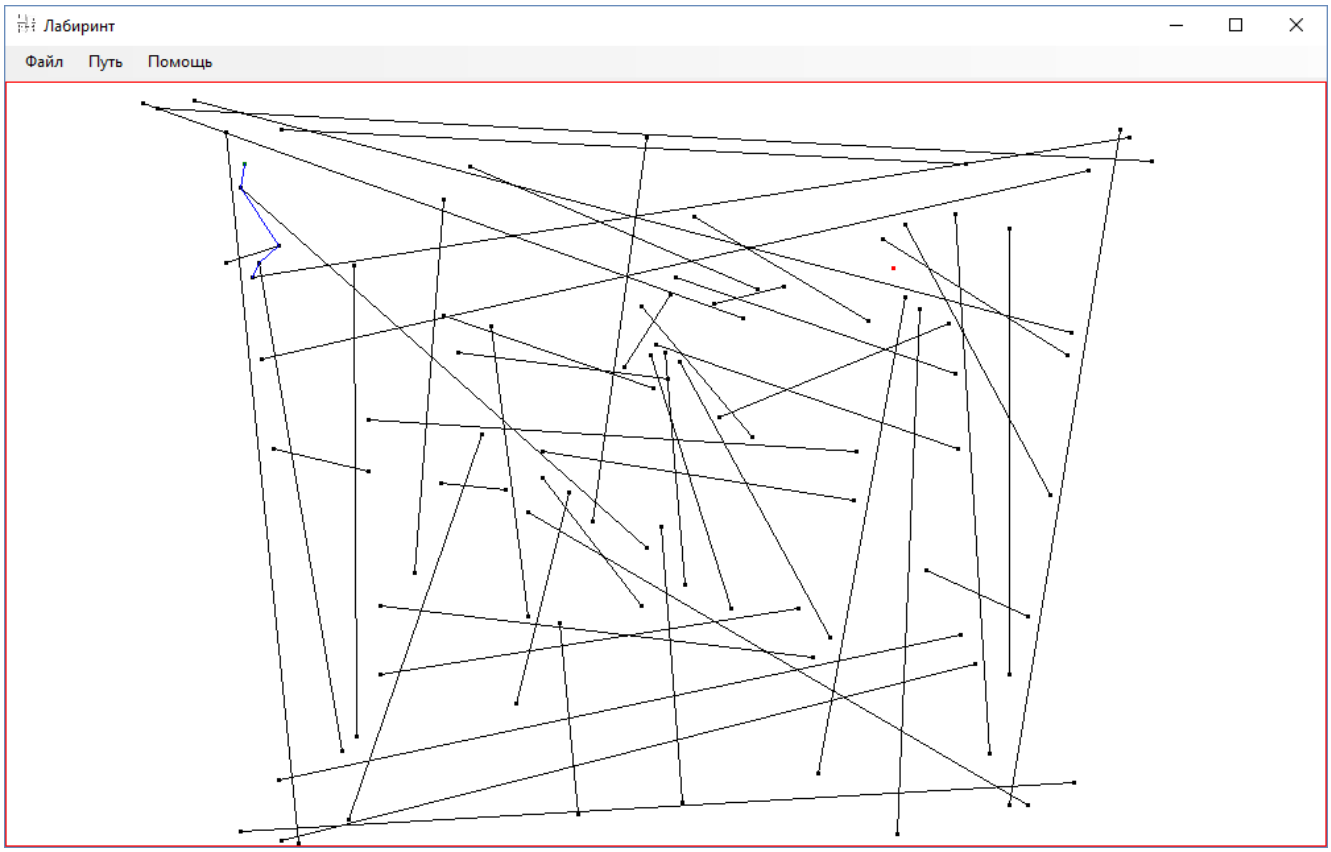
Покрокова побудова шляху відображена на Рисунку 4.11 (деякі кроки пропущені):



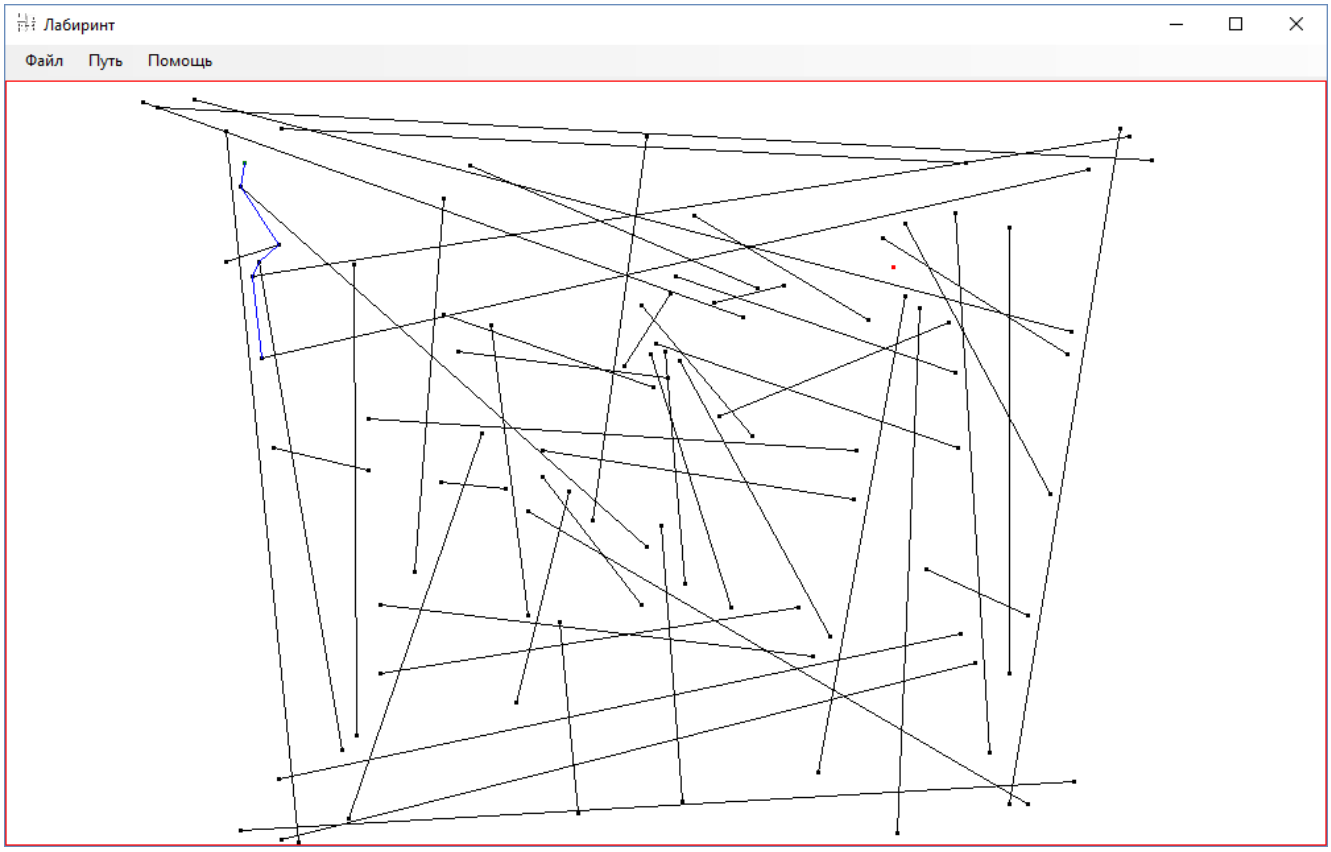
а)



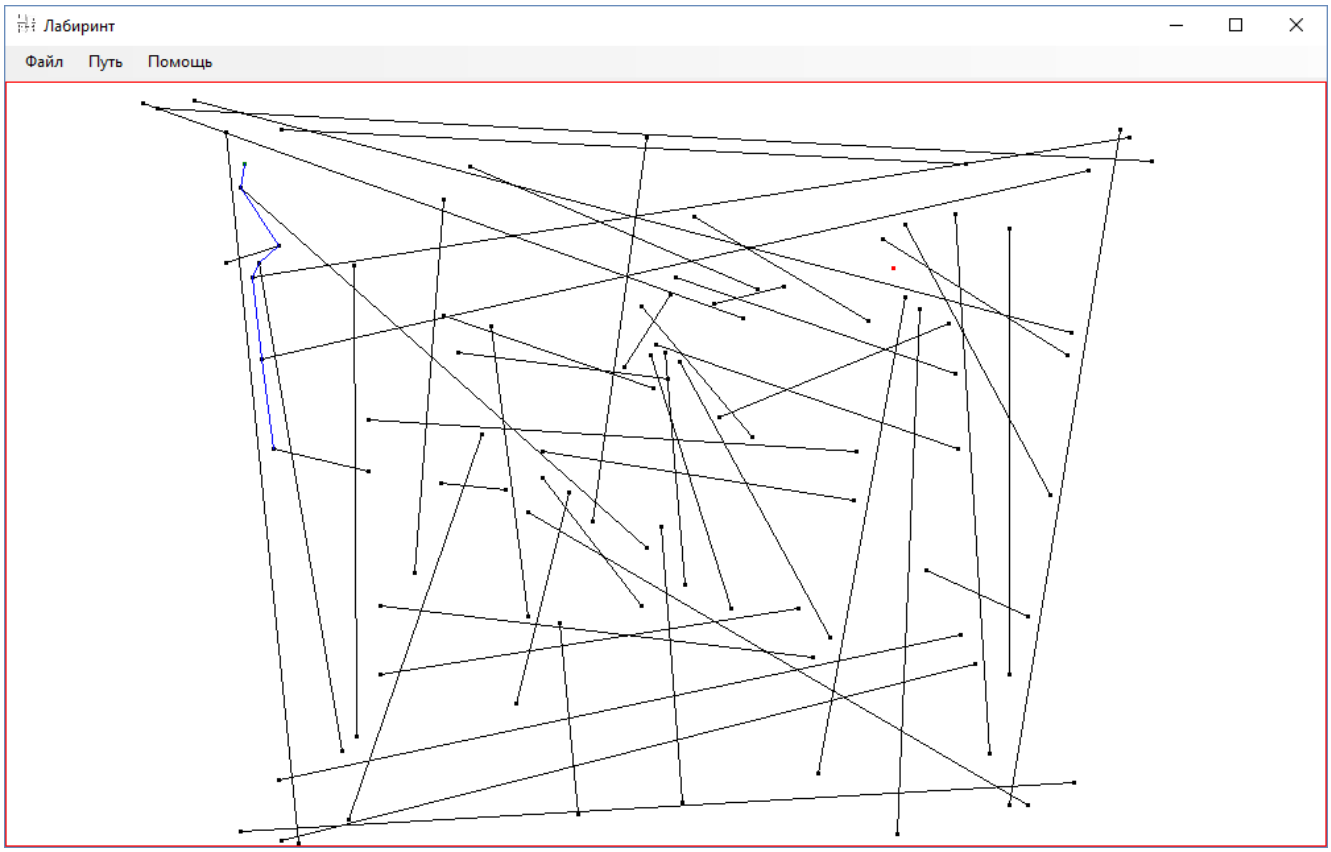
б)



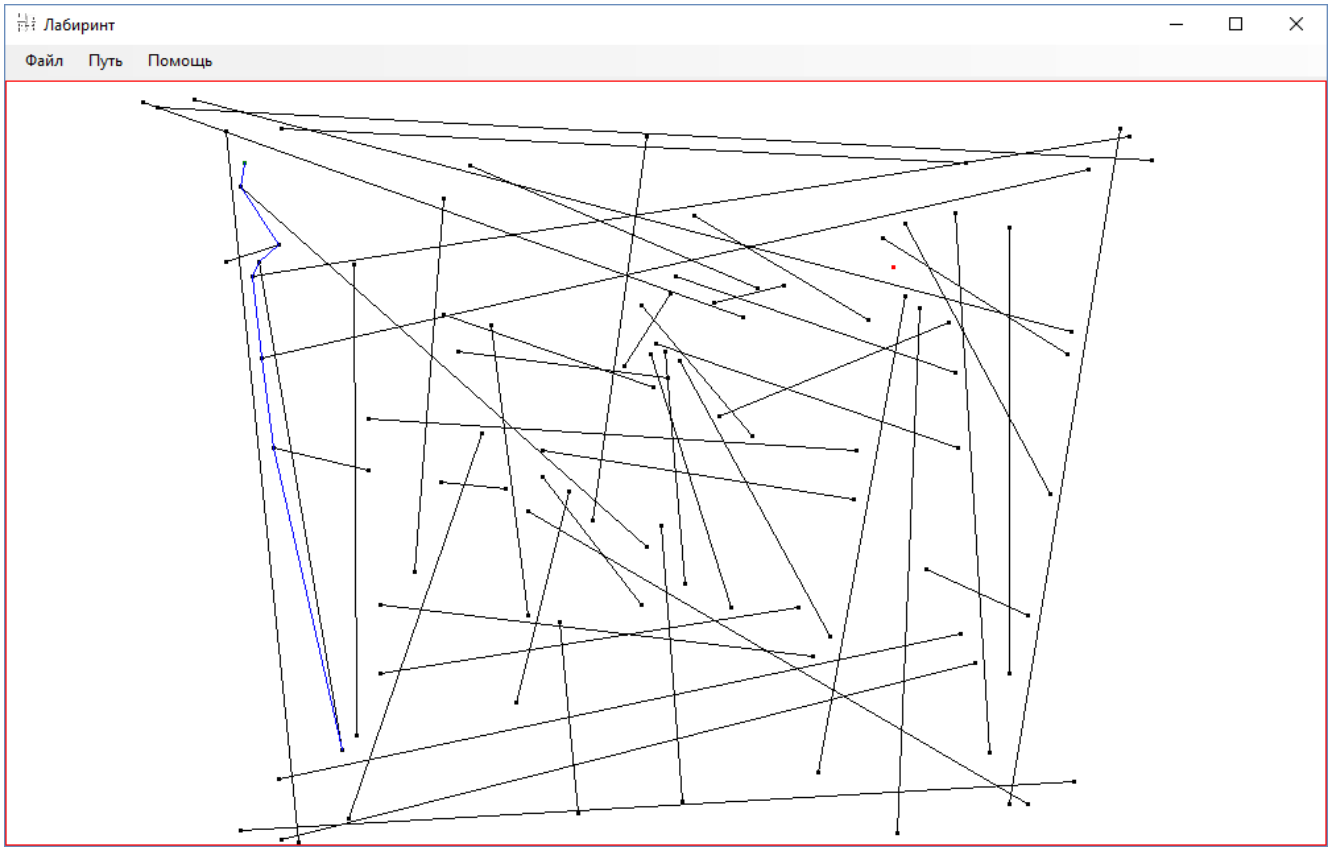
в)



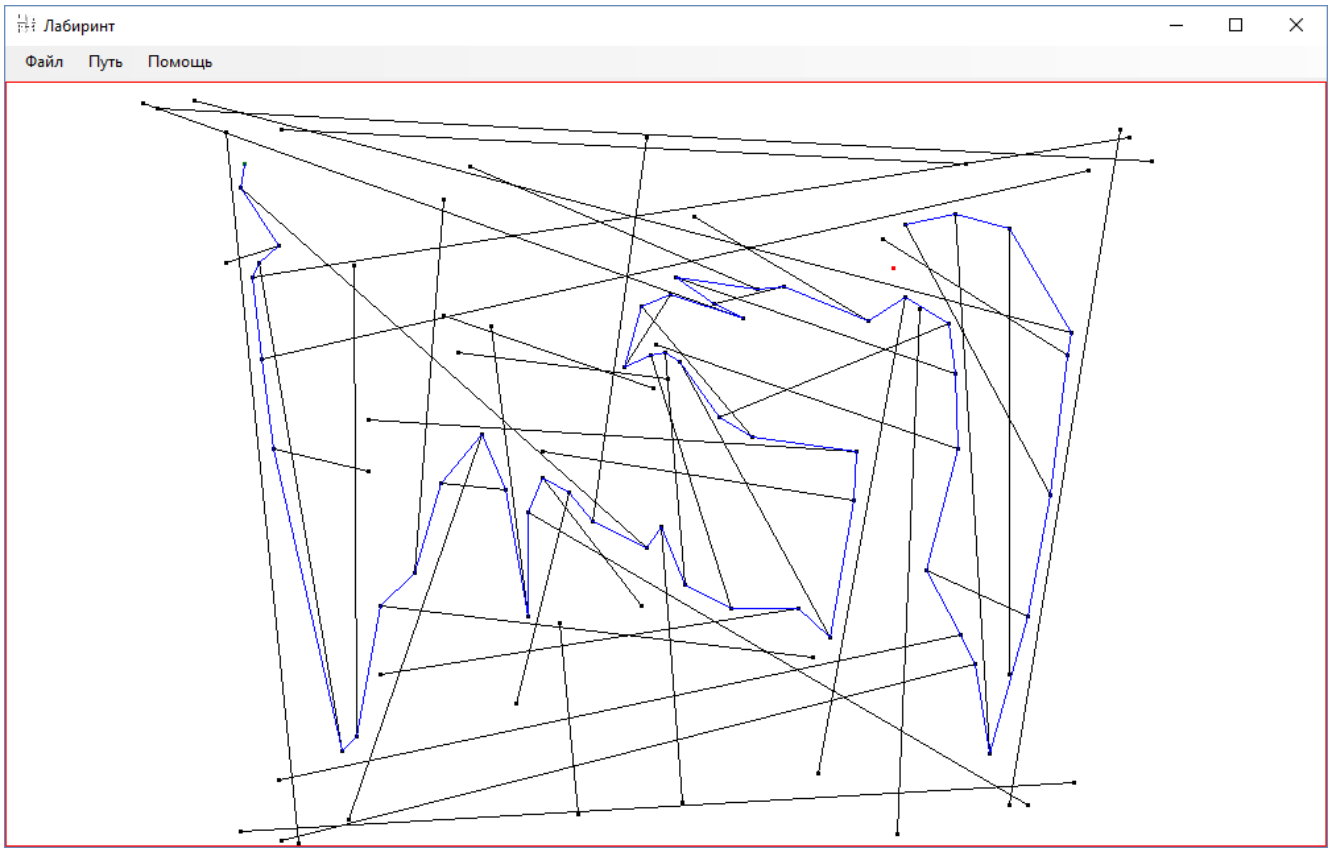
г)



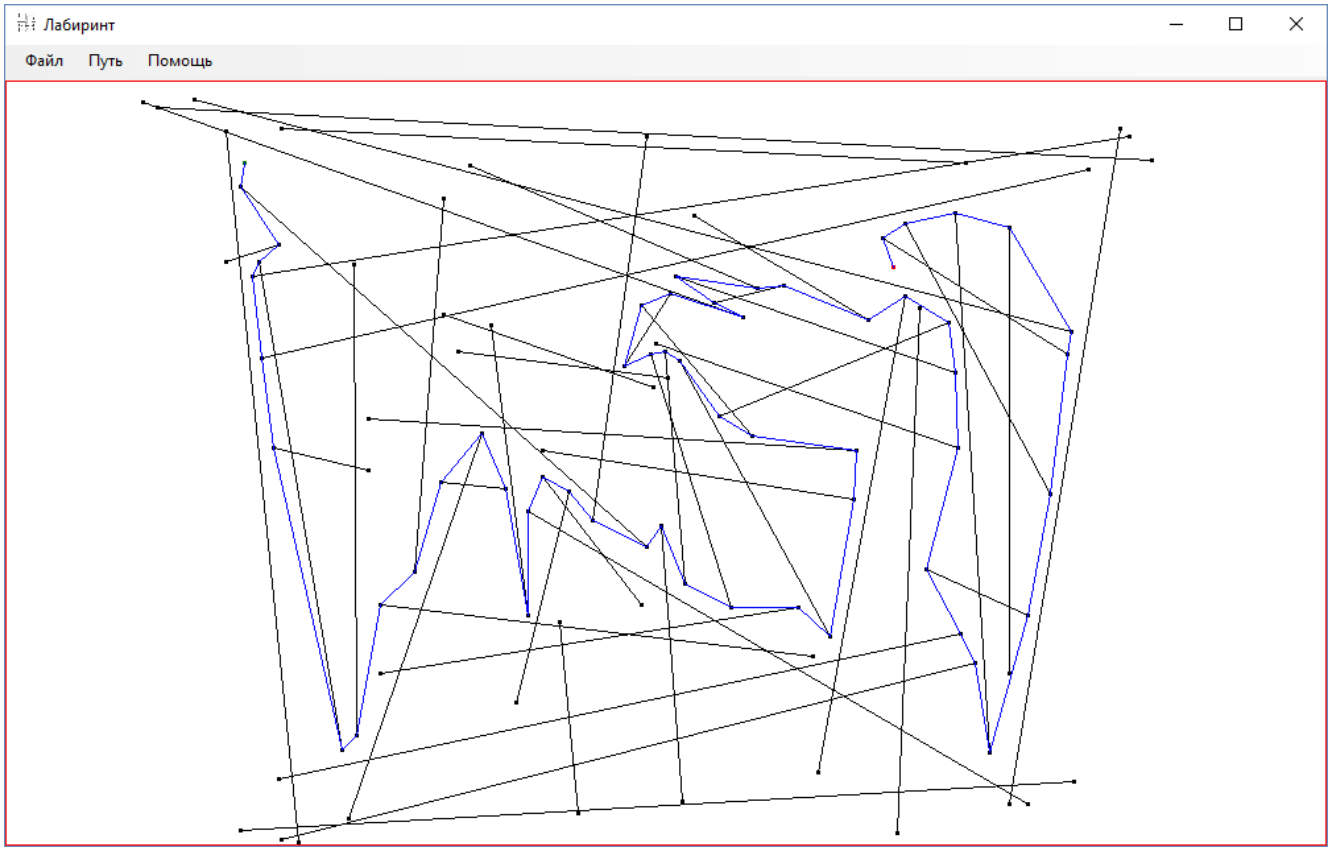
г)



д)



е)



ж)

Рисунок 4.11 – Покрокове знаходження шляху

ВИСНОВКИ

В цій роботі був розглянутий та досліджений алгоритм Флойда-Воршолла, основні принципи і нюанси цього алгоритму.

З виконаної роботи можна зробити наступні висновки:

– алгоритм доволі легкий у реалізації, але потребує доволі багато пам'яті при виконанні;

– було створене алгоритмічне ПЗ, а також набуто навички, щодо створення ПЗ, що включає нетривіальні алгоритми;

– були створені документи, супроводжуючі розроблене ПЗ;

Отже, у процесі виконання цієї роботи був зроблений перший крок у сфері вивчення алгоритмів на графах. Тобто маючи знання, здобуті у цій роботі, можна стверджувати, що фундамент вивчення цієї області покладено і можна рухатись далі в освоєнні загальних принципів побудови алгоритмів обробки даних. Також здобуті навички у проектуванні та розробці ПЗ з дружнім та гнучким графічним інтерфейсом.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1 Теорія алгоритмів як дисципліна // http://wiki.kspu.kr.ua/index.php/*_Теорія_алгоритмів_як_дисципліна, 19.09.2015

2 Обчислювальна складність // http://uk.wikipedia.org/wiki/Обчислювальна_складність, 19.09.2015

3 Алоритми розділяй та володарюй // https://uk.wikipedia.org/wiki/Розділяй_та_володарюй_(інформатика), 19.09.2015

4 Динамічне програмування // https://uk.wikipedia.org/wiki/Динамічне_програмування, 19.09.2015

5 Амортизаційний аналіз // https://uk.wikipedia.org/wiki/Амортизаційний_аналіз, 19.09.2015

6 Поняття основних структур даних // http://www.refine.org.ua/pageid-5325-1.html, 09.10.2015

7 Алгоритми на графах // https://uk.wikipedia.org/wiki/Категорія:Алгоритми_на_графах, 19.09.2015

8 Обчислювальна геометрія // https://uk.wikipedia.org/wiki/Обчислювальна_геометрія, 19.09.2015

9 Методичні вказівки по виконанню курсової роботи по курсу "Алгоритми і структури даних" для студентів, які навчаються за напрямом "Програмна інженерія" / Нац. техн. ун-т "Харківський політхнічний інститут"; [уклад.: Н. К. Стратієнко], 2013. – 34 с.

10 Масив (структура даних) // https://uk.wikipedia.org/wiki/Масив_(структура_даних), 04.10.2015

11 Алгоритм Флойда - Воршелла // https://uk.wikipedia.org/wiki/Алгоритм_Флойда__Воршелла, 09.10.2015

12 C# // https://uk.wikipedia.org/wiki/C_Sharp, 24.10.2015

13 Microsoft Visual Studio // https://uk.wikipedia.org/wiki/Microsoft_Visual_Studio, 24.10.2015

14 ReSharper // https://uk.wikipedia.org/wiki/ReSharper, 24.10.2014

15 Windows Forms // https://ru.wikipedia.org/wiki/Windows_Forms, 24.10.2014

16 Буч Г. Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин М.: ДМК Пресс, 2006. – 496 с.: ил.

17 Опис алгоритму Флойда-Воршелла // http://habrahabr.ru/post/105825/, 09.10.2015

18 Візуалізація алгоритму Флойда-Воршелла // http://uchimatchast.ru/teory/flojd_primer.php, 09.10.2015

19 Приклад роботи алгоритму Флойда-Воршелла // http://uchimatchast.ru/teory/flojd_primer.php, 09.10.2015