МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

СОЛОГУБ ВЛАДИСЛАВ АНДРІЙОВИЧ

Допускається до захисту
в.о. завідувача кафедри
інформаційних технологій
к.т.н, доцент
_____ Оксана ЗЕЛІНСЬКА
«____»_____ 2023 р.

ІНФОРМАЦІЙНА СИСТЕМА ОПТИМАЛЬНОГО КОМПЛЕКТУВАННЯ ЗАМОВЛЕНЬ В ГІПЕРМАРКЕТАХ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (магістерська) робота

Науковий керівник:
Сергій ШТОВБА,
професор кафедри
інформаційних технологій,
д.т.н., професор
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця – 2024

ЗМІСТ

АНОТАЦІЯ

Сологуб В. А. Інформаційна система оптимального комплектування замовлень в гіпермаркетах. Спеціальність 122 «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2023.

У кваліфікаційній (магістерській) роботі досліджено алгоритми пошуку оптимального шляху в спрощеній задачі комівояжера та процес розробки інформаційної системи, що представлена у вигляді комп'ютерної програми. Представлено модель зв'язаного графу. Розроблено алгоритм пошуку оптимального шляху в просторі на даній моделі. Встановлено на скільки швидше (~16%) відбувається прохід по гіпермаркету за використання алгоритму та моделі графу.

Ключові слова: шлях, метод, оптимізація, алгоритм, граф, вершина, простір.

ABSTRACT

Solohub V. A. Information System for Optimal Order Fulfillment in Hypermarkets. Specialty 122 "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia, 2023.

In the qualification thesis, algorithms for finding the optimal path in a simplified traveling salesman problem are explored, as well as the process of developing an information system presented in the form of a computer program. A model of a connected graph is presented, and an algorithm for finding the optimal path in the space of this model is developed. It is determined that the passage through the hypermarket using the algorithm and graph model is approximately 16% faster.

Keywords: path, method, optimization, algorithm, graph, vertex, space.

ВСТУП

Розглядається задача автоматизації комплектування у гіпермаркетах. Вона полягає в проході гіпермаркету задля докомплектування полиць. Дана проблема поширена в гіпермаркетах, при чому, чим більший гіпермаркет, тим складніше комплектувальнику обдумати шлях, за яким йому доведеться пройтись.

Традиційно вирішується за допомогою електронних або письмових списків та за допомогою людської логіки. Даний процес можливо оптимізувати. Швидше комплектування дозволить як комплектувальникам так і звичайним людям проходити по гіпермаркету витрачаючи на це менше часу. Для комплектувальників це означає більше виконаної роботи, для власників бізнесу — більше здобутків завдяки меншим витратам. Окрім цього дану роботу можна допрацювати так, що вона буде й корисна звичайним покупцям. Цим і обумовлюється актуальність дослідження.

Об’єктом дослідження – автоматизація процесів комплектування у гіпермаркетах.

Предметом дослідження – моделі, алгоритми та програмне забезпечення для мінімізації маршрутів комірників під час комплектування замовлень в гіпермаркетах.

Метою роботи є створення програмного продукту для прокладення оптимальних маршрутів в гіпермаркеті, необхідних для комплектування замовлень.

Для досягнення мети в роботі слід:

- Розглянути та проаналізувати поточний прогрес дослідження даної проблеми в публікаціях подібного типу;
- Дослідити проблему;
- Звіритись з публікаціями, та розробити власне рішення проблеми;
- Впровадити рішення у вигляді програмного продукту, так, щоб ПЗ повним чином би вирішувало проблему, та дозволяло легко підійти до неї;
- Описати роботу з програмним забезпеченням, представити результати роботи з програмним продуктом.

Новизна даного проекту полягає в його унікальності. У ході роботи було розроблено алгоритм для пошуку оптимального шляху в просторі,

тобто алгоритм розглядає вершини графу, які задано відповідно їх розташування у просторі, було розроблено модель ненапрямленого графу, зв’язки якого описуються самими вершинами. Використання цих концептів дозволяє зберігати зв’язки як посилання на інші вершини. Це спростило процес розробки програмного забезпечення, націленого на вирішення подібних проблем. Унікальність полягає в тому, що розглянута проблема, до цього моменту, розглядалась лише теоретично. Практична цінність роботи полягає в тому, що розроблено прототип програмного продукту, який має такі функції:

- Побудову, редагування та трекінг списків продуктів;
- Динамічну мапу гіпермаркету;
- Адаптований, під подану задачу, метод, що буде апроксимувати найкоротший шлях;
- Аналіз мапи на вершини та зв’язки графу.

Результатом проведеної роботи є Desktop додаток, що демонструє, описаний вище, функціонал.

РОЗДІЛ 1 АНАЛІЗ СТАНУ ПИТАННЯ

1.1 Аналіз об'єкту дослідження

В даній роботі розглядається процес походу до магазину, як задача оптимізації маршруту.

У ході роботи перед комплектувальником завжди стоїть питання “що та куди ?”. Зазвичай, складається список за яким орієнтуються, що мається у візку та куди його необхідно занести. В основному такі списки розробляються без будь-яких підходів, оскільки вони є частиною рутини. У більшості випадків, коли необхідно використовувати список, він містить продукти або товари різних категорій і має суттєвий розмір.

Для більш детального аналізу процесу було розглянуто професійні вимоги до кур’єрів-комплектувальників. Існує вимога “швидко та якісно збирати замовлення”, що може казати про відсутність інструментів для ефективного допакування. Більшість кур’єрів, скоріш за все, користуються "домашніми” застосунками компаній, що спрощує їм перегляд необхідних продуктів, але це лише допомагає частково закрити вимогу.

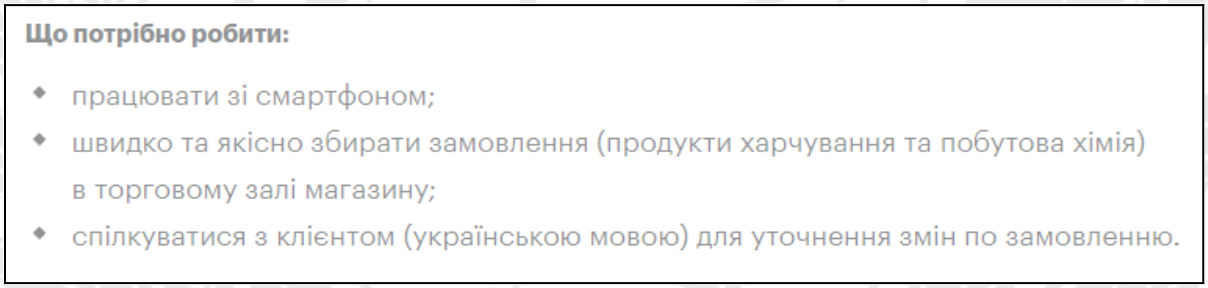

Рисунок 1.1 - Вимоги до кур’єрів-комплектувальників з вакансії А, robota.ua

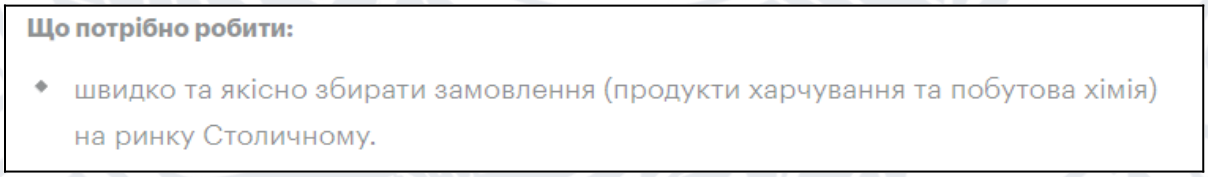

Рисунок 1.2 - Вимоги до кур’єрів-комплектувальників з вакансії Б, robota.ua

З цього прикладу видно, що потреба в такому додатку все ж таки є.

Для вирішення цієї задачі, необхідно представити рішення знайдене за допомогою алгоритму оптимізації шляху, тому дана робота пропонує категоризувати походи до магазину за розмірністю списку на унікальні

продукти та товари, та за, випливаючою з цього, кількістю унікальних розділів, що необхідно буде відвідати в тому чи іншому магазині. Такий підхід дозволить розглядати розділи магазинів як вершини графу, а відстані між ними як ваги ребер.

1.2 Проблема комівояжера, суть, способи вирішення

Оптимізація шляху — класична задача комп'ютерних наук. Існує велика кількість трактувань та проблем пов'язаних із цією дискретною проблемою. У контексті даного дослідження буде розглядатись саме проблема комівояжера.

Проблема комівояжера це алгоритмічна задача. Її суть можна розглянути так: "Якщо ми маємо список міст та відстані між парами цих міст, який шлях є найкоротшим, якщо потрібно відвідати кожне місто лише один раз та повернутись у початкове місто ?".[1] Вперше ця задача була математично сформульована в дев'ятнадцятому сторіччі ірландським математиком Віль'ямом Роувеном Гамільтоном.

Враховуючи те, що теоретичне рішення вимагає пройти не через усі вершини, але лише через ті, що вказано користувачем, та те, що кінець даного шляху може бути не лише на вході до гіпермаркету, а, наприклад, на касі, або взагалі в іншому місці, слід розглянути підвиди даної задачі.

Задача комівояжера має два спрощених види:

- Traveling purchaser problem (Проблема мандруючого покупця);
- Vehicle routing problem (Проблема транспортних шляхів).

Проблема мандруючого покупця має таку умову: "Маючи список ринків, вартостей мандрівок між ними, та маючи список доступних товарів разом з їх ціною на кожному ринку, необхідно знайти, для поданого списку, шлях, що включає до себе найменші витрати як на пересування, так і на покупки". Задача комівояжера є спеціальним випадком даної проблеми. TPP задача є саме спрощенням, тому що відбується пошук шляху не крізь усі вершини, як у TSP, а лише через ті, що містять продукт зі списку.

Задача транспортних шляхів, з іншого боку, описується так: "Який набір шляхів є оптимальним для n транспортних засобів, якщо необхідно зробити доставку лише деяким замовникам."

Подана проблема підлягає під поняття спрощеної задачі комівояжера, оскільки слід знайти найоптимальніший шлях через підмножину вершин, а не крізь усі, а закінчується знайдений шлях не виключно на початковій вершині.

Хоч задача і у спрощеному вигляді, вона все одно є NP-Hard (non-deterministic polynomial-time hardness) задачею теоретичних комп'ютерних наук та дослідження операцій.[2]

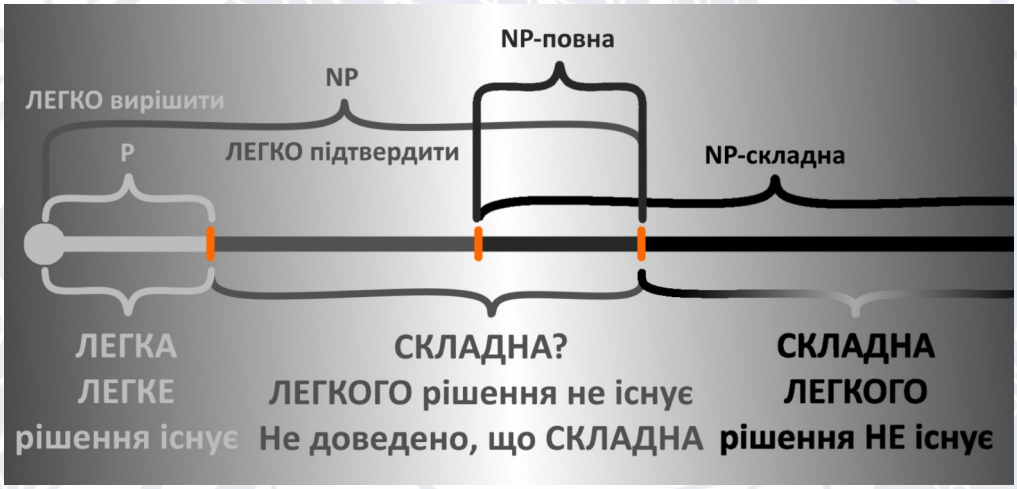

Рисунок 1.3 - Складності P задач у порівнянні

Дана проблема має масу математичних формулювань. В даній роботі розглядаються:

- Асиметрична та симетрична задачі;
- Задача на графі;

Асиметрична та симетрична задачі відрізняються тим, що шлях до міста може бути або не бути в один напрямок, наприклад:

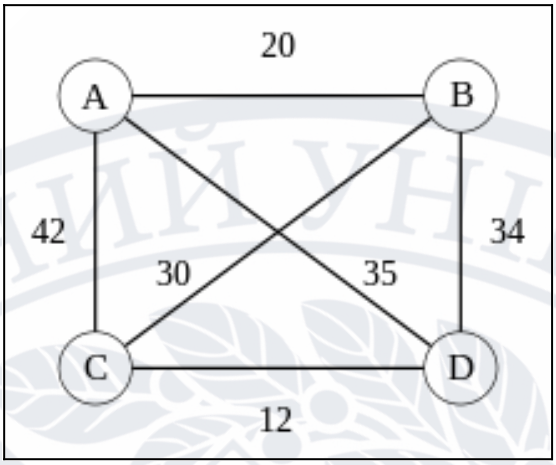

Рисунок 1.4 - Симетрична задача комівояжера з чотирма вершинами

Задача комівояжера на зваженому графі без напрямків має таке формулювання: “Знайти такий шлях, щоб пройти через кожну вершину графу лише раз та повернутись у початкову точку”. Зазвичай при такому формулюванні використовується повний граф (в якому кожна вершина містить шлях до кожної іншої вершини).

Також дана задача має кілька пов’язаних задач, до них входять:

- Обчислення Гамільтонового Шляху;
- MinIMax задача комівояжера;
- Загальна задача комівояжера;
- Проблема мандрівного покупця.

З описаних задач лише проблема мандрівного покупця має відношення до даної роботи, але в межах завдання було вказано вирішення саме проблеми комівояжера, тому розглядатися інший варіант не буде.

До способів вирішення NP-Hard задач, зазвичай, використовують такі підходи:

1. Точні методи;
2. Евристичні методи;
3. Унікальні випадки.

В рамках роботи було розглянути лише перші два підходи.

Точні методи дають можливість знайти точний розв'язок задачі комівояжера, тобто, обчислити довжини всіх можливих маршрутів та обрати маршрут з найменшою довжиною. Однак, навіть для невеликої кількості міст в такий спосіб задача практично нерозв'язна. Для простого

варіанта, симетричної задачі з n містами, існує (n - 1)! / 2 можливих маршрутів, тобто, для 15 міст існує 43 мільярдів маршрутів та для 18 міст вже 177 більйонів. Якщо існував би пристрій, що знаходив би розв'язок для 30 міст за годину, то для двох додаткових міст він витрачав би в тисячу раз більше часу; тобто, більш ніж 40 діб.

Найпрямішим підходом до вирішення задачі комівояжера (TSP) є - brute force search. Даний метод оглядає всі можливі комбінації вершин графу. Складність даного алгоритму складає O(n!). Цей пошук стає не практичним уже після 20 міст. [3]

Інші підходи включають до себе такі алгоритми:

- Алгоритм Хелда-Карпа;
- Алгоритми branch-and-bound (до 60 міст);
- Алгоритми прогресивного покращення (до 200 міст);
- Алгоритм branch-and-cut (до 85900 міст).

Останній алгоритм, branch-and-cut, реалізовано в програмі Concorde TSP Solver і вважається найшвидшим точним алгоритмом у вирішенні проблеми комівояжера для великої кількості вершин.

До складних, вибагливих до ресурсів, точних методів існують альтернативні евристичні. Вони дозволяють пожертвувати точністю заради пришвидшення пошуку.

Одним з перших для розв'язку застосували метод найближчого сусіда. Алгоритм найближчого сусіда (NN) дозволяє комівояжеру вибрати найближче невідвідане місто як наступний крок. Цей алгоритм швидко дає ефективно короткий шлях. Для N міст, розташованих випадковим чином на площині, алгоритм в середньому дає шлях, який на 25% довший за найкоротший можливий шлях[3].

До інших популярних алгоритмів входять:

- Алгоритм Кристофідеса-Сердюкова;
- Алгоритм Попарного обміну;
- Евристика Лін–Керніган;
- Оптимізовані ланцюги Маркова.

Головна перевага евристичних методів над точними полягає в тому, що вони знаходять оптимальний маршрут швидко, але цей маршрут не гарантовано найкоротший. Головна перевага точного методу над евристичними полягає в тому, що він завжди повертає найкоротший маршрут, але обрахування цього маршруту відбувається повільно з великою витратою ресурсів.

Дана проблема природньо поширена в багатьох галузях, наприклад:

- **Логістика:** дана проблема використовується для вирішення проблеми оптимального маршруту доставки для логістичних компаній. Відправники можуть знайти найкоротший маршрут для

відвідування кількох місць доставки, мінімізуючи витрати на паливо та час.

- **Мікроелектроніка:** дана проблема використовується для проектування оптимальних маршрутів проводів на мікрочіпах. Це допомагає знизити витрати на виробництво та забезпечити ефективне розташування компонентів на чіпі.
- **Дослідження геному:** дана проблема використовується для аналізу послідовностей ДНК та визначення найкоротшого шляху для послідовного читання генетичних кодів.
- **Маршрутизація мережі:** дана проблема використовується для оптимізації маршрутів у мережевому обладнанні, такому як маршрутизатори та комутатори, для ефективної передачі даних між вузлами мережі.
- **Виробництво:** дана проблема використовується для планування послідовності операцій у виробництві, мінімізуючи час та затрати на переміщення матеріалів та обладнання.

У ході дослідження проблеми було визначено напрям розвитку рішення даної задачі. Оскільки тематично, програмний продукт буде застосовуватись у гіпермаркетах, було сформульовано таку проблему: “У гіпермаркеті з визначеною кількістю розділів n, необхідно пройти підмножину n1 розділів таким чином, щоб кінцевий шлях w складав мінімальну можливу відстань від першого розділу $n1_0$, до його останнього $n1_i$.”. Простіше кажучи, необхідно знайти найоптимальніший шлях, що проходить крізь усі вершини n1 від входу до гіпермаркету до виходу на касі.

В даній задачі кількість вершин, що відповідає певному розділу не обмежена. Це задумано таким чином, тому що в кожному гіпермаркеті продукти з одного й того самого розділу можуть лежати в різних місцях. Хоча це, можливо, не є проблемою для власників магазинів, в даному контексті проблематика маркування точного місця знаходження брендових товарів заслуговує на власне дослідження.

В такій задачі розглядається зважений граф в якого, зв’язки є відстанями між розділами. Зв’язки не мають напряму. Кожна вершина має мінімум один зв’язок. Відстань вимірюється відповідно географічному розташуванню вершин. Кількість типів розділів обмежено тим, що типи

товарів є досить обширними. Наразі розробляється адаптація алгоритму Найближчого Сусіда для використання в технічному рішенні.

1.3 Огляд аналогів

Точно відповідних, публічних, аналогів не існує. Загалом більшість додатків такого типу є або органайзерами/списками (як, наприклад, Liki24 та CityRider), або додатками для формування замовлення, що, зазвичай, є фірмовими додатками і включають до себе функціонал пошуку, порівняння продуктів, перевірки наявності продуктів в магазинах мережі.

Найближчими до представленої реалізації є оптимізатори маршрутів, наприклад:

- **PTV Route Optimiser;**
- **MyOnlineRoute.**

PTV Route Optimiser – це логістичний застосунок компанії PTV, він використовується для планування оптимальних транспортних шляхів, та націлений на створення оптимальних графіків перевезення та на створення умов оптимального використання усіх транспортних засобів. Це великий застосунок для підприємств, впроваджується лише під замовлення, та з тим чи іншим функціоналом.

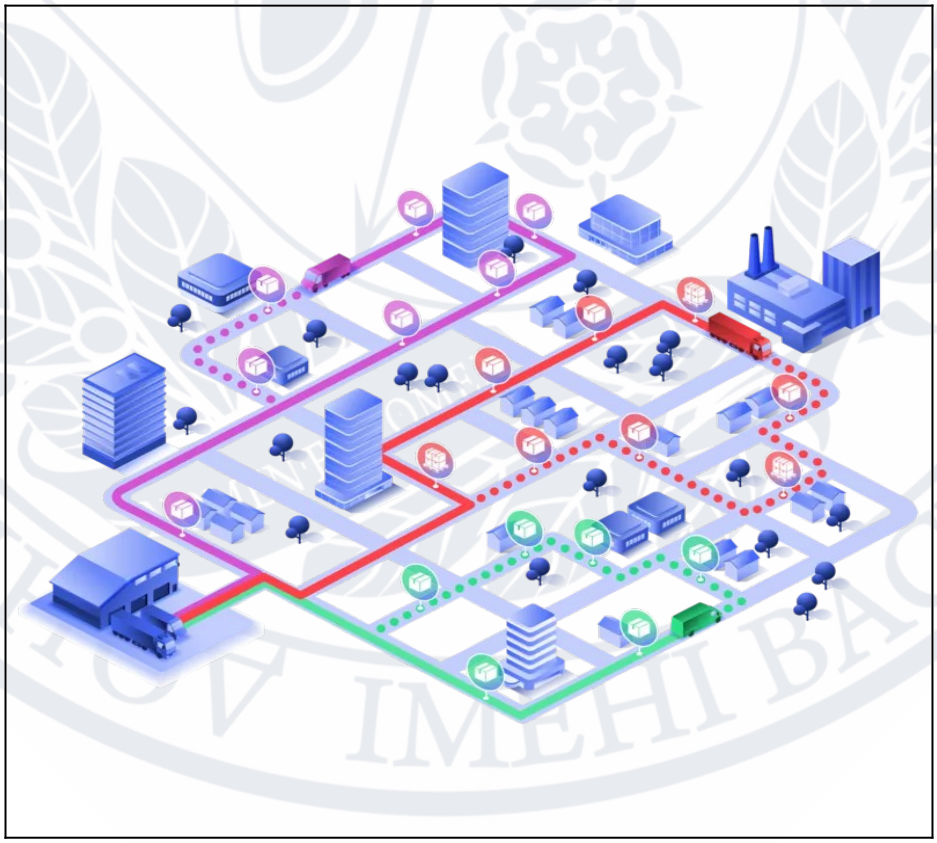

Рисунок 1.5 — Представлення додатку PTV Route Optimiser з їх веб-сайту

MyOnlineRoute, з іншого боку, застосунок для планування подорожей. Він включає до себе органайзер адрес, планувальник і оптимізатор. Поширюється у вигляді підписки, або оплачується в ході використання.

Хоч і певним чином дані застосунки є аналогами, але вони створені для вирішення дуже специфічної задачі, тому потреби в порівнянні даних застосунків — немає, методи та алгоритми будуть суттєво відрізнятися.

Головна перевага даного проекту полягає в його унікальності. Зазвичай процес орієнтації в крамницях інші застосунки залишають користувачеві. Дане рішення буде направляти користувача як компас, що повинно відміняти людську помилку в розглянутому процесі й позитивно впливати на час, що витрачається.

Головним недоліком додатку, ймовірно, буде складність використання. Оскільки даний проект є інструментом, користувачам доведеться вчитись ним користуватись. З цього випливає, що максимальна ефективність рішення буде досягатись лише через певний час, а в процесі навчання користуванням, теоретичний застосунок буде, навіть скоріше, сповільняти процес комплектування ніж пришвидшувати .

1.4 Підходи до вирішення

У ході дослідження було винайдено 3 підходи до вирішення проблеми комплектування:

1. Вирішення за допомогою інструментарію Google Maps;
2. Статистичний підхід із OpenStreetMap та їх датасетом України;
3. Створення власного, спеціалізованого інструменту.

Підхід №1

Включає до себе роботу з Android SDK (Java) або з Kotlin. Плюси цього підходу полягають у якості рішень і розмірі бібліотеки готових функцій. Maps SDK, також, включає в себе різноманітні статистичні відомості. Мінуси полягають у тому, що кількість запитів до Google Maps обмежено, та після певної кількості запитів з розробника почнеться стягуватись оплата. Для реалізації за цим підходом потрібно буде

розробити конвертацію та накладання зображення плану гіпермаркету поверх його географічних координат.

Даний підхід дозволив би пропустити етап впровадження певних математичних методів, а також спростив би роботу зі зв’язками графу. Додатково, робота з готовою мапою дозволила б пропустити етап розробки цієї мапи, та методів роботи з нею.

Підхід №2

Включає до себе роботу з даними від OpenStreetMap. Перевагами даного підходу є простота, широкий обсяг даних, безкоштовність. Недоліки полягають у потребі використання застосунків від третьої особи, робота з нестандартними розширеннями даних. Окрім цього створення парсеру потрібних, у цьому контексті, даних призвело б до набагато довшого циклу розробки.

Цей підхід можна використовувати як окремо, так і у зв’язці з іншими двома підходами. Його суть полягає в швидкому визначенні меж гіпермаркетів на мапі, оскільки території в OpenStreetMap завжди мають тип (це дозволяє відслідковувати де є гіпермаркет, де є магазин, а де не потрібна нам будівля), координати розташування входу, виходу та його меж. Причому, генерацію цих даних можна обмежити одним разом, а далі вже працювати з ними за потреби. Це дозволяє створити власну мапу гіпермаркетів, плани та розмітки яких створювали б користувачі або розробники. Це також надає можливість динамічно підвантажувати плани будівель за підходу до них.

Підхід №3

Має на увазі розробку власного рішення. Даний підхід надає великі переваги у вигляді відносно невеликої бази коду, гарної орієнтації в усіх впроваджених методах, інновації, необмеженості. З іншого боку неправильний план дій та незнання специфічності вирішення задач таких може призвести до стагнації. Ідея цього підходу полягає у впроваджені потрібного функціоналу лише тоді, коли він потрібен. Наприклад, оскільки плани гіпермаркетів досить легко представити малими зображеннями, то впровадження повноцінних функцій приближення та віддалення можна пропустити. Так само з географічними координатами, ми можемо впровадити лише методи, що потрібні для досягнення цілі.

Оскільки наш код ізольований і не залежить від інших факторів, при цьому підході, його можна перетворити на пакунок, яким, потім, можна поділитися з іншими розробниками.

1.5 Задачі дослідження

З**адачі дослідження**:

- Дослідити задачу комівояжера, її рішення, застосування та приклади реалізації алгоритмів, присутніх у рішеннях;
- Обрати або створити та обґрунтувати спосіб вирішення цієї задачі для реалізації;
- Розробити дизайн ПЗ, що дозволить швидко орієнтуватись у гіпермаркеті, та буде створювати маршрут для відповідного списку;
- Безпосередньо розробити MVP застосунку, що повинен включати такі функції:
    - Можливість будувати розмітки гіпермаркетів або магазинів;
    - Можливість задання списку продуктів і товарів;
    - Інтерактивне відстеження прогресу докомплектування.
- Порівняти час без використання додатку та з використанням додатку.

Дана система повинна забезпечувати швидше комплектування. Навіть невеличка перевага зможе покращити ефективність комплектувальників та прибутки власників бізнесів у довгочасній перспективі.

Оскільки задача комівояжера в даному випадку використовується для оптимізації шляху, саме час буде критерієм оцінювання. У ході тестування ПЗ буде сформовано такий список продуктів, щоб він максимально охоплював усі розділи магазину. Далі буде здійснено прохід по кільком магазинам із засіканням часу, з використанням додатку та без його використання. Це дозволить порівняти ефективність даного рішення зі звичайною людською логікою.

Висновок за першим розділом

У ході роботи було розглянуто процес закупки, було дано визначення задачі комівояжера, було оглянути її підвиди, алгоритми для вирішення, особливості рішення, типи алгоритмів. Було розглянути сильні та слабкі сторони теоретичного додатку, було оцінено його унікальність.

Було досліджено різні підходи до вирішення поданої проблеми. Розглянулись об’єкт та предмет дослідження, мета роботи. Було побудовано шлях до створення рішення, було поставлено задачі дослідження. Було поставлено за мету розробити додаток, що буде вигідний для комплектувальників.

РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ

2.1 Вимоги до алгоритму

У ході дослідження буде створюватись додаток для навігації в реальному часі, це створює вимогу до швидкодії. Через це необхідно дослідити саме евристичні алгоритми пошуку шляху. Оскільки задачею дослідження було визначено потребу в навігації — розглянемо алгоритми, що ідейно використовують простір і фізичні відстані.

Дана робота розглядає проблему комплектувальника, яку можна описати таким чином: “Маємо категоризований список продуктів. Знаючи можливі переходи з розділів у гіпермаркеті, необхідно знайти такий шлях, що буде проходити крізь кожен підрозділ, таким чином, що загальна довжина шляху буде мінімальна”.

У ході роботи гіпермаркет розглядається як граф. В даному графі вершини розташовні відносно одне одного. Зв’язки між цими вершинами є відстанями між ними. Існування зв’язку описує існування проході між двома вершинами. Таке представлення дозволяє розраховувати простим чином будувати зв’язки, та розраховувати ваги навіть із відсутніми зв’язками. Також такий підхід дозволяє відносно просто представити гіпермаркет графічно у вигляді зображення.

Комплектувальник входить у гіпермаркет в якійсь точці, дана точка в графі відома як початкова вершина. Дана вершина вирішує оптимальний шлях комплектувальника. Гіпермаркет може мати кілька початкових вершин, але алгоритм буде враховувати лишу ту, в якій комплектувальник знаходиться фізично. Тому в задачі початкова вершина — одна. Кур’єр повинен виходити через вихід, або через касу. Задача враховує це, і після пошуку оптимального шляху - знаходиться оптимальний вихід. Шлях до цього виходу додається до знайденого оптимального шляху.

У ході проходу по гіпермаркету комплектувальник повинен пройти крізь кожен розділ, що вказується у списку до побудови оптимального шляху. Даний список вирішує оптимальний шлях та розділи, крізь які комплектувальник обов’язково здійснить прохід. У випадках коли потрібний підрозділ недоступний, алгоритм повинен побудувати оптимальний шлях без врахування даного розділу.

Знаючи список підрозділів, алгоритм повинен приблизно знати, куди необхідно рухатись. Граф гіпермаркету, умовно, можна розбити на

підграфи розділів. За допомогою цих підграфів, на основі списку повинен генеруватись приблизний маршрут. Даний маршрут буде використовуватись для пошуку більш детального шляху. Також даний процес дозволить дізнатися, чи містить той чи інший магазин потрібні комплектувальнику розділи. Згенерований приблизний маршрут є відносним, вершини в даному кортежі можуть не мати зв'язків між одне одним.

2.2 Огляд існуючих алгоритмів

Найближчим до вимог є оригінальний алгоритм Дейкстри, який було створено в 1956 році. Його суть полягає в пошуку найкоротшого шляху між двома вершинами графу.[4] Даний алгоритм розглядає зв'язки графу як відстані між вершинами, а самі вершини — як точки. Він завжди пересувається в ту вершину, що, по-перше, не була відвідана та по-друге, є найближчою до поточної вершини. Оглянуті вершини помічаються, пройдені - запам'ятовуються. Даний алгоритм складається з таких кроків[5][6][7][9]:

1. **Початок:**
   a. Обираємо початкову вершину.
   b. Позначаємо відстані для всіх інших вершин як нескінченні, окрім початкової, для якої відстань встановлюється як 0.
   c. Позначаємо всі вершини як невідвідані.
2. **Оновлення відстаней:**
   a. Обираємо поточну вершину (початкова на початку).
   b. Для кожного невідвіданого сусіда поточної вершини:
      - Рахуємо відстань, пройшовши через поточну вершину.
      - Якщо ця тимчасова відстань менша за поточну відстань до сусіда, оновлюємо відстань до сусіда.
3. **Відмітка проходження:**
   a. Позначаємо поточну вершину як пройдену.
   b. Видаляємо поточну вершину зі списку невідвіданих.
4. **Умова завершення:**
   a. Якщо кінцева вершина вже відвідана або найменша тимчасова відстань серед невідвіданих вершин - нескінченність, алгоритм завершується.
5. **Вибір наступної вершини:**

a. Обираємо невідвідану вершину з найменшою тимчасовою відстанню як поточну.
b. Повторюємо кроки 2-4.

Цей процес триває до тих пір, поки відстані до всіх вершин не будуть оновлені, або поки не буде досягнуто кінцевої вершини. Алгоритм Дейкстри допомагає знаходити найкоротший шлях в графі, враховуючи довжину ребер між вершинами.[8]

Подальше вдосконалення цього підходу представляє алгоритм A*. Однак, на відміну від алгоритму Дейкстри, A* використовує додатковий елемент - евристичну оцінку. Ця оцінка додається до вартості пройденого шляху і визначається як евристична оцінка залишкового шляху до кінцевої вершини. Таким чином, A* надає пріоритет вершинам, які, ймовірно, приведуть до оптимального шляху, сприяючи ефективнішому пошуку.[11]

Алгоритм A* був представлений у 1968 році американськими вченими Пітером Хартом, Нілом Корменом та Рафаелем Міллером. Він відзначається здатністю знаходження найкоротших шляхів у графі, враховуючи вартість пройденої вершини та евристичну оцінку залишкового шляху до кінцевої вершини.[12]

Алгоритм A* є розширенням і вдосконаленням алгоритму Дейкстри, де враховується не тільки довжина вже пройденого шляху, але й евристична оцінка (чи "вартість") залишкового шляху до кінцевої вершини. Ця евристика допомагає алгоритму визначати пріоритетні вершини, щоб спрямовувати його в бік потенційно більш оптимального шляху.[10]

Цей алгоритм виглядає таким чином[13][14][15][16]:

1. **Початок:**
   a. Обираємо початкову вершину та встановлюємо відстані для всіх інших вершин як нескінченні, окрім початкової, для якої відстань встановлюється як 0.
   b. Позначаємо всі вершини як невідвідані.
2. **Оновлення відстаней та евристична оцінка:**
   a. Крім відстані від початкової вершини до кожної вершини, обчислюємо евристичну оцінку залишкового шляху до кінцевої вершини. Це може бути здійснено за допомогою функції евристики.

3. **Вибір та оновлення:**
    a. Обираємо поточну вершину, враховуючи як її відстань від початкової вершини, так і евристичну оцінку залишкового шляху.
    b. Для кожної сусідньої невідвіданої вершини:
        - Обраховуємо тимчасову відстань від початкової вершини до цієї сусідньої вершини через поточну вершину.
        - Обчислюємо евристичну оцінку залишкового шляху для цієї сусідньої вершини.
        - Об'єднуємо відстань та евристичну оцінку, і визнаємо, чи отримали покращення. Якщо отримали — оновлюємо значення відстані та евристики для сусідньої вершини.
4. **Відзначення відвіданих вершин та продовження:**
    a. Відзначаємо поточну вершину як відвідану та видаляємо її зі списку невідвіданих вершин.
    b. Продовжуємо до тих пір, поки не дійдемо кінцевої вершини або поки всі вершини не будуть оглянуті.
5. **Зупинка алгоритму:**
    a. Зупинка відбувається, якщо кінцева вершина відзначена як відвідана (у випадку планування маршруту між двома конкретними вершинами) або якщо найменша тимчасова відстань серед вершин у списку невідвіданих є нескінченною (у випадку повного обходу графа, коли немає з'єднання між початковою вершиною та рештою невідвіданих вершин).
6. **Оновлення поточної вершини:**
    a. Якщо алгоритм не зупинився, вибираємо невідвідану вершину з найменшою сумарною відстанню та евристичною оцінкою, і повертаємось до кроку 3.

2.3 Окреслення дій алгоритму

У ході роботи, методом проб та помилок, було створено першу версію необхідного алгоритму. Цей алгоритм базується на ідеї оцінюванні позитивних та негативних змін відстаней. Даний алгоритм виглядає так: [17]

1. **Перед початком:**
    a. Формуємо список.
    b. Розбиваємо граф на підграфи відповідно кожній унікальній категорії.

2. **Початок:**
   a. Обираємо початкову вершину.
   b. Додаємо цю вершину до оптимального шляху.
3. **Побудова приблизного шляху (початкова оцінка):**
   a. Від початкової вершини для кожного підграфу категорій:
      - Обраховуємо відстань від початкової вершини до кожної вершини підграфу. Якщо ця відстань найменша — додаємо її до приблизного маршруту.
      - Переходимо в цю найближчу вершину.
      - Повторюємо крок a доки не пройдемо по всім підграфам.
   b. Обираємо поточну приблизну вершину.
4. **Оцінка та оновлення:**
   a. Обчислюємо відстань від поточної до поточної приблизної вершини.
   b. Для кожного зв'язку поточної вершини:
      - Обчислюємо відстань від зв'язку до поточної приблизної вершини, виявляємо чи додатній має додатній знак вага цього зв'язку:
        - Якщо має — запам'ятовуємо, що є позитивна зміна, дивимось чи це найкраща зміна, якщо так — запам'ятовуємо зв'язок в буфері. Якщо не має — перевіряємо чи загалом були позитивні зміни. Якщо не було — обираємо найменш гіршу зміну, запам'ятовуємо в буфері.
5. **Відзначення відвіданих вершин та продовження:**
   a. Обираємо вершину з буферу як поточну вершину.
   b. Позначаємо, що вона відвідана.
   c. Додаємо поточну вершину в оптимальний шлях.
   d. Якщо поточна вершина міститься у графі приблизного шляху, видаляємо її в графі приблизного шляху (позначаємо відвіданою).
   e. Якщо поточна вершина також є поточною приблизною вершиною - обираємо наступну невідвідану поточну приблизну вершину. Інакше - повертаємось до кроку 4.
6. **Зупинка алгоритму:**
   a. Зупинка відбувається, якщо в графі приблизного шляху немає невідвіданих вершин.
7. **Оновлення поточної вершини:**
   a. Для кожної вершини гіпермаркету позначаємо, що вона не відвідана.

Якщо алгоритм не зупинився, повертаємося до кроку 4.

Головним недоліком даного алгоритму було те, що рано чи пізно він попадав у глухий кут знаючи лише те, що неможливо рухатись до наступної вершини. Таким чином і закінчувався. Дану проблему було вирішено за допомогою більш простого підходу, а саме - звірянні відстаней. Завдяки вже існуючим алгоритмам, що базуються на такій самій ідеї, на їх основі було дороблено модуль евристичної оцінки.[18][19]

У ході дороблення даного алгоритму його було перестворено у повноцінний евристичний алгоритм, що базується на ідеї оцінки відстаней для пошуку оптимального шляху, та в достатній мірі вирішує проблему комплектувальника. Згідно вимогам кінцевий алгоритм має такий вигляд: [20]

1. **Перед початком:**
   a. Формуємо список.
   b. Розбиваємо граф на підграфи відповідно кожній унікальній категорії.
2. **Початок:**
   a. Обираємо початкову вершину.
   b. Додаємо цю вершину до оптимального шляху.
3. **Побудова приблизного шляху (початкова оцінка):**
   a. Від початкової вершини для кожного підграфу категорій:
      - Обраховуємо відстань від початкової вершини до кожної вершини підграфу. Якщо ця відстань найменша — додаємо її до приблизного маршруту.
      - Переходимо в цю найближчу вершину.
      - Повторюємо крок а доки не пройдемо по всім підграфам.
   b. Обираємо поточну приблизну вершину.
4. **Оцінка та оновлення:**
   a. Обчислюємо відстань від поточної до поточної приблизної вершини.
   b. Для кожного зв’язку поточної вершини:
      - Обчислюємо модуль відстані від зв’язку до поточної приблизної вершини:
        - Якщо ця відстань менша за відстань від поточної, запам’ятовуємо її як найкращу зміну, запам’ятовуємо поточний зв’язок у буфері.

- У випадку якщо не знайшли позитивну зміну, обираємо найменшу негативну зміну. Запам’ятовуємо поточний зв’язок у буфері.

5. **Відзначення відвіданих вершин та продовження:**
   a. Обираємо вершину з буферу як поточну вершину.
   b. Позначаємо, що вона відвідана.
   c. Додаємо поточну вершину в оптимальний шлях.
   d. Якщо поточна вершина міститься у графі приблизного шляху, видаляємо її в графі приблизного шляху (позначаємо відвіданою).
   e. Якщо поточна вершина також є поточною приблизною вершиною - обираємо наступну невідвідану поточну приблизну вершину. Інакше - повертаємось до кроку 4.
6. **Зупинка алгоритму:**
   a. Зупинка відбувається, якщо в графі приблизного шляху немає невідвіданих вершин.
7. **Оновлення поточної вершини:**
   a. Для кожної вершини гіпермаркету позначаємо, що вона не відвідана.
   b. Якщо алгоритм не зупинився, повертаємося до кроку 4.

Перевагою поданого алгоритму є те, що він, завдяки особливості подання графу, дозволяє обрахувати оптимальний шлях не задаючи ваги зв’язків. Хоч і даний алгоритм було створено для застосування комплектувальниками, це не відміняє його пристосованість до задач подібного типу. Наприклад поданий алгоритм може використовувати і звичайний покупець. Представлення графу не обмежене лише гіпермаркетами і може бути загалом будь-яким магазином або навіть базаром. Даний алгоритм може бути покладеним в основу багатьох подібних проектів.

Окрім цього, представлення графу таким чином дозволяє швидко перетворювати його у зображення. Зображення утворене за допомогою цього алгоритму можна перетворити назад у граф, але необхідно знати масштаб. Зображення можна масштабувати простим чином — додавши 2 точки, та запам’ятавши відстань між ними. Знаючи це, можемо перетворити зображення на граф таким чином:

1. **Обраховуємо відстань між двома поданими географічними координатами. Обчислюємо відстань на піксель.**
2. **Пошук вершин:**

a. Робимо заливку (flood fill) всього окрім вершин n_i-го розділу;
b. Знайдені вершини записуємо в масив, при цьому запам’ятовуємо їх графічні координати;
c. Повторюємо доки не пройдемо n розділів;

3. **Пошук зв’язків:**
   a. Робимо заливку всього окрім зв’язків;
   b. З m_i-ої вершини від її координат у вісім сторін шукаємо пікселі зв’язків, Визначаємо кількість зв’язків.
      i. Йдемо по кожному зв’язку. Дійшовши до координати вершини записуємо зв’язок в m_i-у вершину. У випадку, коли знайдена вершина має кілька зв’язків у пріоритеті вибираємо вершину.
   c. Повторюємо доки не пройдемо m вершин;
4. **Розраховуємо ваги зв’язків використовуючи графічні координати, конвертуємо за допомогою відстані на піксель.**

Висновок за другим розділом

У цьому розділі було розглянуто методи оптимізації шляху, що базуються на ідею обрахування відстані, на їх основі було поставлено вимоги до алгоритму теоретичного ПЗ. Було розглянуто та представлено аналогічні алгоритми. Було подано першу версію теоретичного алгоритм до реалізації, а також окреслено його переваги та недоліки. Далі алгоритм було дороблено. Також було описано ідею перенесення зображення до графу.

РОЗДІЛ 3 РОЗРОБКА ДОДАТКУ

3.1 Дизайн ПЗ

Порівнявши всі можливі підходи було прийнято рішення використовувати підхід, що має на увазі розробку власного рішення. Це надає перевагу вибору платформи для розробки макету, в даному випадку це буде C#. Розроблений далі макет можна буде перенести до Xamarin або Xamarin.Forms.

Сама розробка є підтвердженням концепту, тобто буде демонструвати, що такий тип навігації можливий, та є ефективним.

У ході роботи було розроблено кілька макетів на різних стадіях розробки. Першим з них є цей макет:

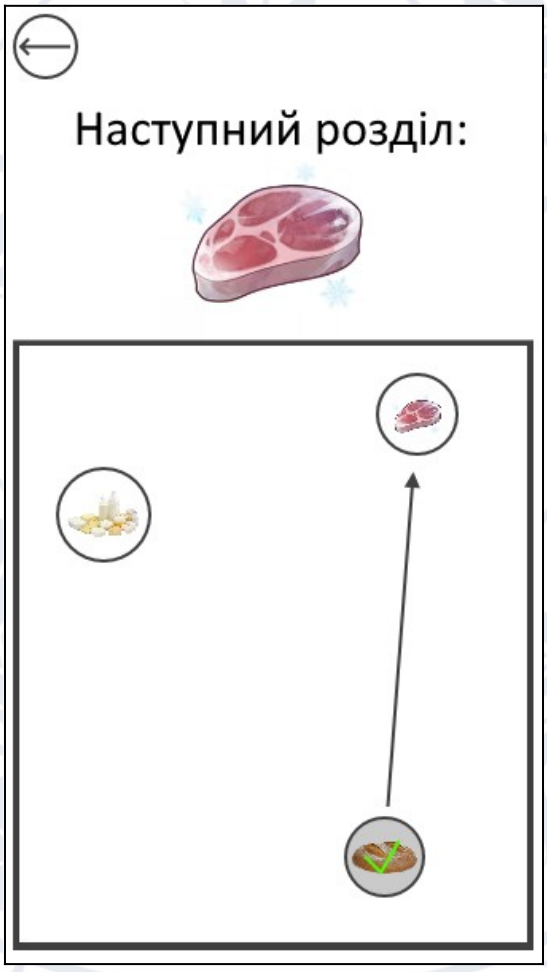

Рисунок 3.1 - Макет №1 користувацького інтерфейсу застосунку

Дане зображення демонструє базове бачення застосунку. Маємо мапу, вона вказує куди йти відповідно списку та алгоритму.

Далі макет було адаптовано під розмітку Android:

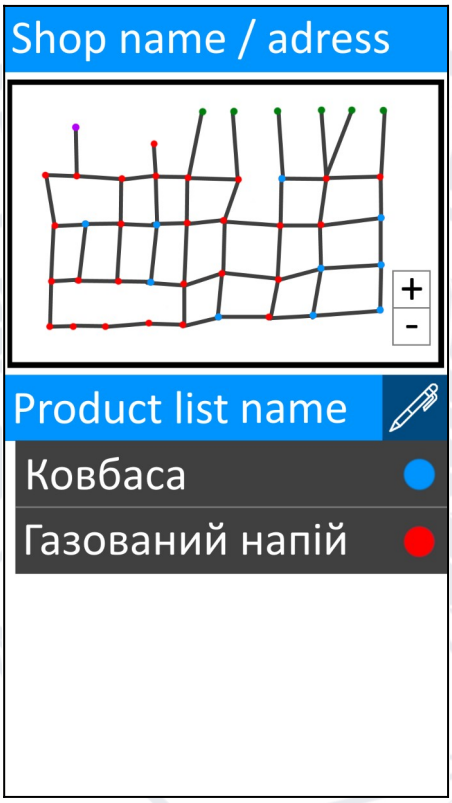

Рисунок 3.2 - Макет №2 користувацького інтерфейсу застосунку

Бачимо тут список змінив свій вигляд, тепер продукти повинні розташовуватись в режимі пріоритету - чим ближче потрібний продукт, тим вище він у списку. На вхід буде подаватись зображення такого типу:

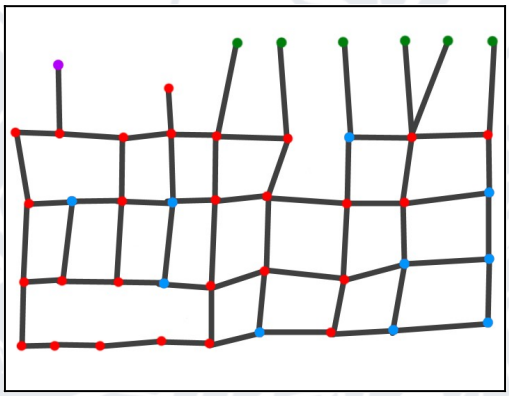

Рисунок 3.3 - Макет №2, мапа

Це дозволить графічно будувати граф, що досить сильно спростить користувачам процес створення мапи, оскільки на вхід програмі слід буде подати лише 3 значення та зображення. Ці три значення включають в себе дві точки, для пошуку масштабу мапи та третє значення - точка входу. Зображення та точки будуть запаковані в файл, що буде створюватись у програмі окремо.

На основі цього зображення було створено наступний макет, вже у виді форми WinForms:

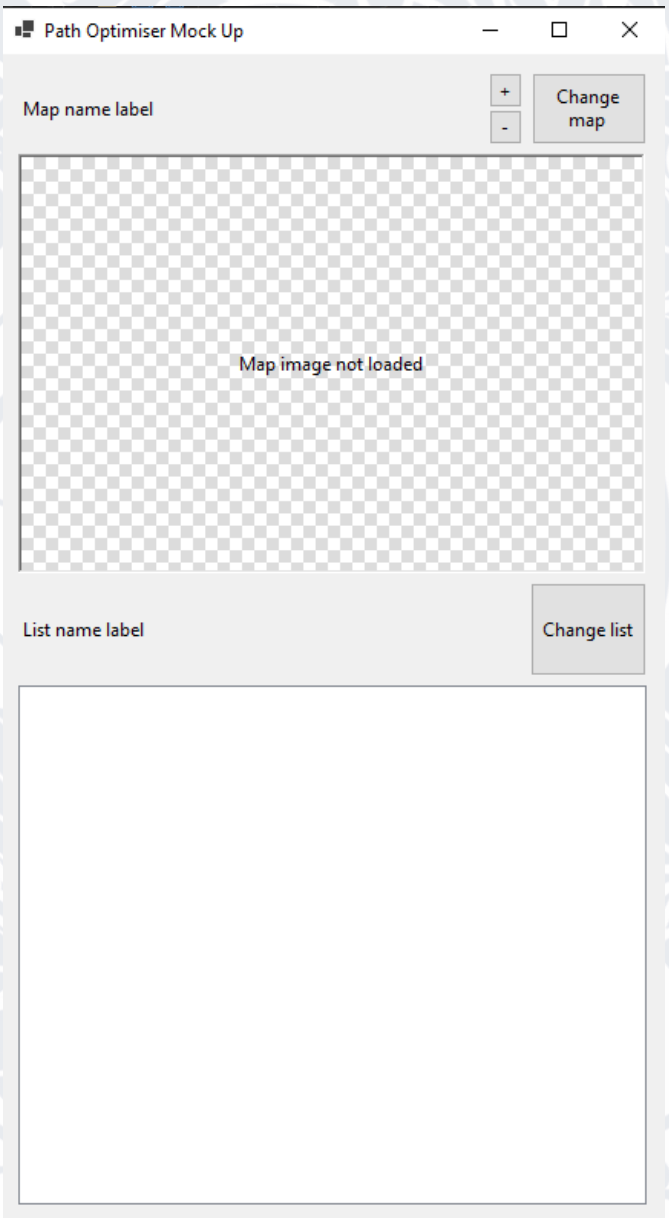

Рисунок 3.4 - Макет №3 користувацького інтерфейсу застосунку

Після підвантаження мапи, вона буде розкладатися на граф всередині програми. Після цього знайдеться найкоротший шлях до найближчого продукту. Далі програма направить користувача до цього продукту, паралельно з цим, слідкуючи за поточною позицією. При досягненні розділу, програма помітить, що продукт отримано, та обрахує новий найкоротший шлях до наступного найближчого продукту, при чому вона буде враховувати загальний пройдений шлях, та адаптувати його по ходу проходження графу.

Додатково до цієї форми, було побудовано макет створювача списків:

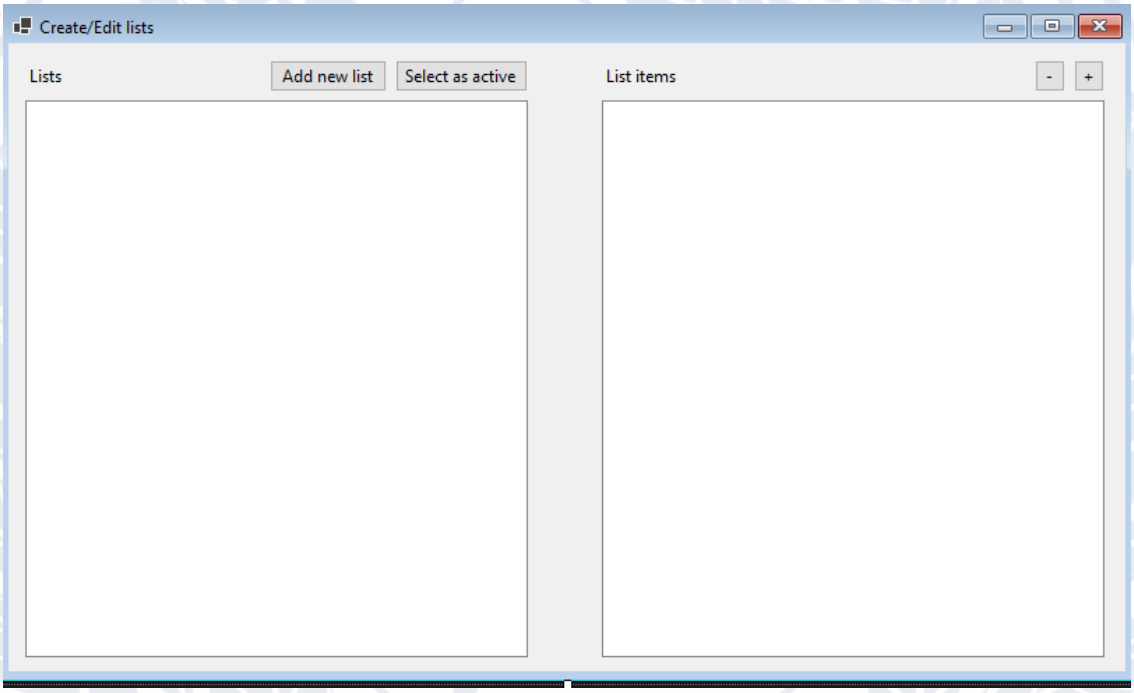

Рисунок 3.5 - Макет №3, створювач списків

Далі було створено кінцевий макет форми, він передбачає пошук шляху після обирання списку. Після того як користувач обрав список, для поточної мапи формується оптимальний шлях. Після формування він виводиться на мапу. При цьому список продуктів сортується таким чином, що чим вищий продукт у відображеному списку, тим ближчий він до користувача. Користувач може підвантажувати різні мапи, та форматовані списки. Дана форма має такий вигляд:

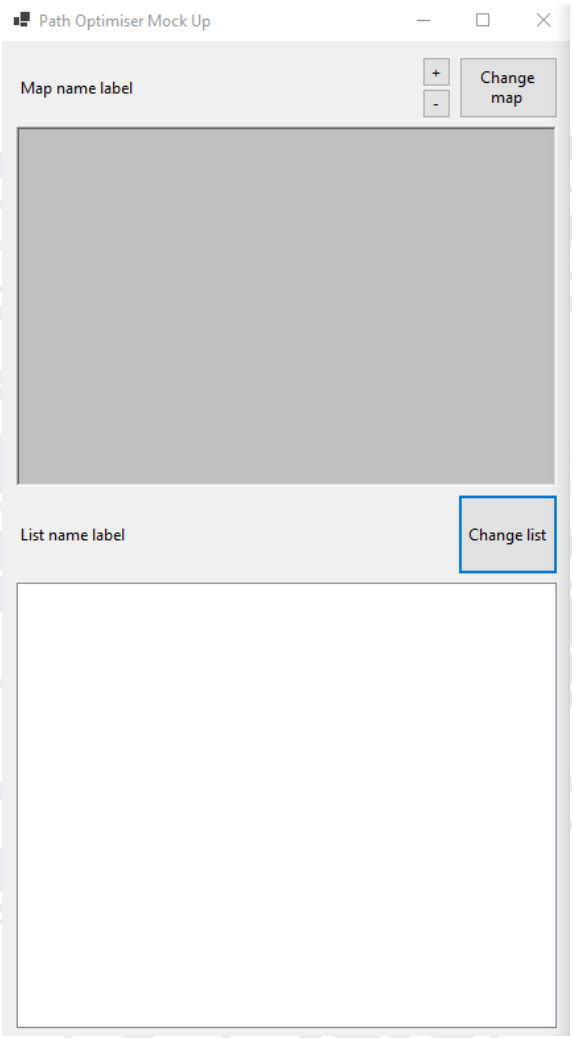

Рисунок 3.6 - Макет №4, фінальна форма

Окрім цього було довершено форму для роботи зі списками. Окрім базових функцій користувач матиме змогу зберігати та підвантажувати списки з файлів. Було покращено й можливості з видалення та редагування списків. Форма для списків також має кілька підформ, що дозволяють більш детально вводити дані, наприклад форма додавання речей до списку виглядає так:

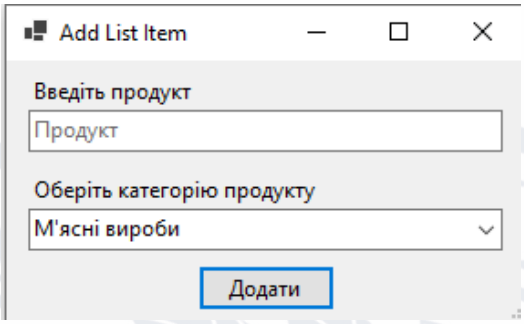

Рисунок 3.7 - Макет №4, форма додавання речей

Форма редактору списків була візуально адаптована під макет Android, та має такий вигляд:

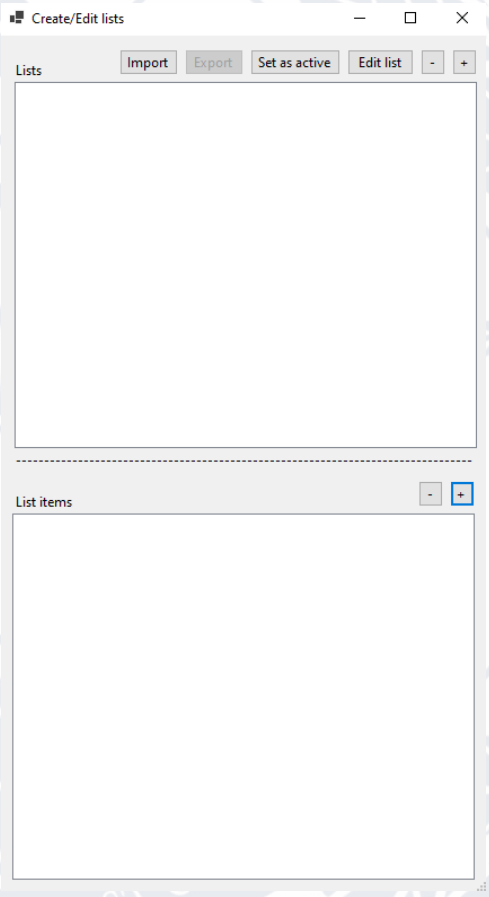

Рисунок 3.8 - Макет №4, форма редактору списків

Хоча в даній формі ще є багато місця для покращення, дана робота не присвячена дизайну форм, тому подалі даний варіант буде вважатися фінальним.

3.2 Архітектура

Дана робота використовує 1 пакет з NuGet:

CyotekImageBox - пакет для роботи із зображеннями. Розширений ImageBox, має інструменти для орієнтації на малюнку. Користувацький пакет, що дає можливість використання будь-де та для будь-чого.

У ході роботи, також, було створено 6 власних модулів, більшість з яких безпосередньо стосуються рішення задачі:

- `Product;`
- `ProductCategory;`
- `Node;`
- `Market;`
- `FormList;`
- `ProductCategoryList.`

Product

Об’єкт класу Product є представленням продукту в списку продуктів. Має 2 параметри та конструктор:

- `string _productName;`
- `string ProductName;`
- `ProductCategory _category;`
- `ProductCategory Category;`
- `Product(string name, ProductCategory category).`

Використовується для задання та відображення продуктів в списку продуктів, і для роботи з продуктами. Кількість об’єктів цього класу не обмежена.

ProductCategory

Об’єкт класу ProductCategory є представленням категорії продукту у списку продуктів, у вікні мапи. Має 2 параметри та конструктор:

- `string _categoryName;`
- `string CategoryName;`
- `Color _categoryColor;`
- `Color CategoryColor;`
- `ProductCategory(string name, Color color).`

Використовується для маркування обмеженої кількості розділів. Кожен розділ повинен мати унікальний колір. Розділи визначаються при запуску програми, та редагуються виключно розробником завдяки статичному класу ProductCategoryList.

Node

Об’єкт класу Node є представленням вершини на зображені, та вершини в географічних координатах. Об’єкт класу Node містить в собі зв’язки до інших вершин. Відповідає за роботу з вершинами графу. Складається з 5 параметрів, 2 конструкторів, 4 методів:

- `int _x;`
- `int X;`
- `int _y;`
- `int Y;`
- `List<Node> _connections;`
- `List<Node> Connections;`
- `ProductCategory _category;`
- `ProductCategory Category;`
- `bool _hasBeenVisited;`
- `bool HasBeenVisited;`
- `Node();`
- `Node(int i, List<Node> cons, ProductCategory category);`
- `int _y;`
- `int Y;`
- `List<Node> _connections;`
- `List<Node> Connections;`
- `List<double> CalculateDistanceBetweenConnections(Node node);`
- `bool GetPositiveNegativeLogic(double distanceOfApproximation, double distanceOfConnectionToApproximation);`
- `double GetDifference(double distance1, double distance2);`
- `double CalculateDistanceBetweenNodes(Node startPoint, Node endPoint);`

Даний клас відповідає за роботу з координатами та з вершинами графу одночасно. Таке представлення дозволяє не зберігати ваги зв’язків,

а розраховувати їх за потреби. Окрім цього даний клас дозволяє дізнатися, в який напрямок направлена точка по відношенню до іншої точки, дозволяє дізнатись різницю між відстанями, дозволяє дізнатись відстань між двома точками, дозволяє дізнатись відстані до усіх зв'язків поточної вершини.

Market

Екземпляр даного класу використовується для зберігання інформації про гіпермаркет, метод цього класу використовується для генерації гіпермаркетів. Загалом Market нараховує 2 параметри, 2 конструктори та 1 метод:

- `List<Node> _nodes;`
- `List<Node> Nodes;`
- `Node _startingNode;`
- `Node StartingNode;`
- `Market();`
- `Market(List<Node> newNodes, Node newStartingPoint);`
- `Market GenerateMarket().`

Даний клас використовується для зберігання гіпермаркетів, використовується у обрахуванні оптимального шляху, при тестуванні алгоритму.

Метод GenerateMarket() будує розмітку гіпермаркету випадковим чином. На виході отримується ненапрямлений граф розміром від 10 до 15 вершин, де кожна вершина містить від 2 до 4 випадкових зв'язків. Даний метод використовувався при формуванні алгоритму оптимального шляху. Також алгоритм тестувався саме на цьому методі.

FormList

Об'єкт класу FormList є представленням списків у програмі. Даний клас нараховує 2 параметри, 2 конструктори та 1 метод:

- `string _name;`
- `string Name;`
- `List<Product> _productList;`
- `List<Product> ProductList;`
- `FormList();`
- `FormList(string name, List<Product> productList);`

- `void SetFormList(FormList formList).`

Даний клас широко використовується в формах, на основі його об’єктів формуються оптимальні шляхи.

ProductCategoryList

Це статичний клас, що грає роль довідника продуктових категорій. Змінюється розробником та лише за потреби. Містить одну змінну (лише для читання) — масив категорій та параметр з гетером для її отримання:

- `List<ProductCategory> productCategories;`
- `List<ProductCategory> Categories;`

Клас ProductCategoryList націлений на чітке визначення можливих категорій у графі. Використовується при генерації маркетів, у роботі з об’єктами класу FormList, ProductCategory, Product. Має 7 стандартних категорій:

- "М'ясні вироби", `Color.Red`;
- "Риба та морепродукти", `Color.Blue`;
- "Овочі", `Color.Green`;
- "Фрукти", `Color.Yellow`;
- "Молочні вироби", `Color.Purple`;
- "Зернові вироби", `Color.Brown`;
- "Інше", `Color.DeepPink`.

3.3 Реалізація програми

Форма MainForm є головною формою програми, саме вона показує оптимальний шлях, та дає напрям руху комплектувальнику. Складається з 1 вікна для відображення списку, 1 вікна для відображення зображень. Також містить 4 кнопки:

- Кнопка зміни мапи — викликає діалогове вікно, що дозволяє підвантажити зображення мапи;
- Кнопка приближення зображення — відповідно наближує зображення до користувача;
- Кнопка віддалення зображення — відповідно віддаляє зображення від користувача;
- Кнопка редактору списків — відкриває форму ListForm, що відповідає за усі операції зі списками, та за генерацію оптимального шляху.

Клас форми складається з 1 конструктору, 6 подій та з 3 методів:

- `Form1()` – конструктор класу;
- `void SetListViewItems(List<Product> productList, string listName, List<Node> route)` – метод, що відповідає за відображення та сортування списку згідно оптимального шляху. Заповнює вікно відображення списку відсортованим списком у вигляді плитки;
- `void DrawGraph(Market market, List<Node> route)` – метод, що відображує поданий граф у вигляді зображення, та малює оптимальний шлях поверх цього зображення. Зображення графу зберігається в теці збережених списків;
- `SetListViewItemsAndDrawGraph(List<Product> productList, string listName, List<Node> route, Market market)` – метод, що комбінує роботу двох попередніх, для випадків, в яких необхідно викликати ці методи з інших форм;
- `void addZoomButton_Click()` – подія кнопки AddZoom, наближує зображення при натисканні;
- `void subZoomButton_Click()` – подія кнопки SubZoom, віддаляє зображення при натисканні;
- `void changeMapLabel_Click()` – подія кнопки ChangeMap, при натисканні відкриває діалогове вікно вибору файлу, після підтвердження файлу підвантажує його у вікно відображення мапи;
- `void changeListButton_Click()` – подія кнопки ChangeList, при натисканні відкриває підформу ListForm, що дозволяє редагувати та обирати списки;
- `void Form1_Load()` – подія, що відбувається при завантажені форми, конфігурує відображення списку до форми плиток.

SetListViewItems бере на вхід продукти списку, назву списку та оптимальний шлях. На виході отримуємо коректно заповнене вікно відображення списку. На самому початку цей метод очищує вікно відображення списку, запам’ятовує продуктовий список, та створює 2 змінні, що зберігають унікальні категорії з вершин шляху та зберігають унікальні категорії зі списку продуктів. Після отримання цих даних, метод створює третю змінну, що зберігає сортований список продуктів. Потім метод сортує продукти за оптимальними вершинами знайденими у ході оптимізації шляху. Оптимальний шлях є умовою сортування. Відповідно

нього розставляються й категорії продуктів. Якщо в шляху немає якоїсь категорії — можна сміло вважати, що дана категорія відсутня. В результаті сортування у вікні відображення списку отримуємо такий список:

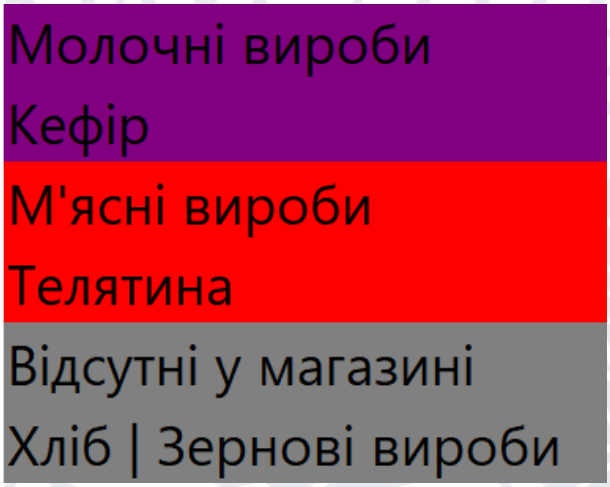

Рисунок 3.9 – Коректно відсортований список
(вищий продукт — пріоритетніший за нижчий, в самому низу — відсутні продукти)

Метод DrawGraph на вхід отримує граф магазину та оптимальний шлях. На виході отримуємо зображення графу з накладеним поверх нього оптимальним шляхом у вікні мапи, а також зображення в теці зберігання списків. Спочатку, для кожного зв’язку, малюються лінії, що поєднують їх. Для цього використовується стандартний метод для роботи з графікою — DrawLine(). Потім, подібним чином, розставляються точки за допомогою методу DrawEllipse(). У кінці в такому самому порядку малюється шлях. Отримана графіка зберігається у бітмапі, дана бітмапа зберігається як зображення, та підвантажується у вікно мапи.

Усі інші методи та події особливо не виділяються та служать для базових операцій. Виходячи з цього переходимо до форму редактору списку ListForm. Дана форма відповідає за усі операції зі списками, а саме:

- Операції імпорту/експорту списків та їх речей;
- Операція обрахування оптимального шляху;
- Операція виділення (активізації) списку;
- Операції додавання/видалення/редагування списків;
- Операції додавання/видалення речей списку.

Клас цієї форми складається з 6 змінних, 1 конструктору, 8 подій, 10 методів, а також кількох підформ та 3 дочірніх форм:

- `Form1 parent` – містить посилання на батьківську формуMainForm;
- `List<FormList> formList` – зберігає дані про усі списки;
- `List<string> nameList` – зберігає дані про усі назви списків;
- `string lastSelectedList` – зберігає дані про останній обраний список;
- `string lastSelectedItem` – зберігає дані про останній обраний предмет списку;
- `bool hasAtLeastOneList` – зберігає інформацію про наявність хоча б одного списку;
- `ListCreationForm(Form1 parentForm)` – конструктор класу, ініціалізує форму при цьому обираючи батьківську форму;
- `void addListButton_Click()` – подія, що відповідає за натискання кнопки додавання списку. Виводить на екран, дочірню форму, AddListForm.cs;
- `void importListButton_Click()` – подія, що відповідає за натискання кнопки імпорту списків. Відкриває файлове діалогове вікно, та пропонує користувачу підвантажити набір списків;
- `void exportListButton_Click()` – подія, що відповідає за натискання кнопки експорту списків. Відкриває файлове діалогове вікно, та пропонує користувачу зберегти усі поточні списки в файлі;
- `void listView_SelectedIndexChanged()` – подія, що відповідає за зміну обраного списку в вікні відображення списків. Зберігає та відображає інформацію про поточний обраний список. Простіше кажучи, коли обирається список, лише його речі також відображаються.;
- `void addItemButton_Click()` – подія, що відповідає за натискання кнопки додавання предмету до списку. Відкриває дочірню форму AddListItemForm.cs. Дозволяє задати продукт, його категорію для поточного обраного списку. Дозволяє додавати нові продукти до списку доки дочірня форма не зачиниться;
- `void subItemButton_Click()` – подія, що відповідає за натискання кнопки видалення предмету зі списку. Створює діалогову підформу, та запитує, чи насправді користувач бажає

видалити предмет, у випадку підтвердження видаляє предмет зі списку, інакше нічого не робить;

- `void editListButton_Click()` – подія, що відповідає за натискання кнопки редагування списку. Дозволяє відредагувати назву списку без створення нового;
- `void removeListButton_Click()` – подія, що відповідає за натискання кнопки видалення списку. Видаляє список і всі його предмети;
- `void selectActiveListButton_Click()` – подія, що відповідає за натискання кнопки обирання активного списку. Обраховує оптимальний шлях, переносить список у батьківську форму, відображає шлях та граф на мапі;
- `void SetFormList(List<FormList> list)` – метод-сеттер, що на вхід отримує змінну списків list та відповідає за присвоєння даних з list змінній formList. Простіше кажучи, замінюємо всі списки цим методом (що є дуже корисним при підвантажені з файлу);
- `List<FormList> GetFormList()` – метод-геттер, що відповідає за отримання даних зі змінної formList;
- `void AddProductToFormListName(string listName, Product listItem)` – метод, що відповідає за додавання нового предмету listItem за назвою списку listName. Використовується в дочірній формі для додавання предметів в поточний список;
- `void AddListNameToFormList(string listName)` – метод, що відповідає за створення нового списку з назвою listName. Використовується при додаванні нових списків. Список можна утворити за умови, що його назва не міститься в змінній formList. Кожен список має унікальну назву. Якщо було додано перший список, робить кнопку експорту активною та запам’ятовує, що маємо хоча б один список у змінній hasAtLeastOne, інакше видає діалогове вікно з помилкою;
- `void EditListName(string newListName)` – метод, що відповідає за зміну назви обраного списку на нову newListName. Так само перевіряє назву списку на унікальність, замінює назву. Використовується в дочірній формі EditListForm.cs;
- `void removeLastSelectedFormList()` – метод, що відповідає за видалення обраного списку. Якщо у вікні відображення списків не 0 списків, то видаляємо. Якщо в змінній formList 0

списків, запам’ятовуємо в змінну hasAtLeastOneList, деактивуємо кнопку експорту та видаляємо виділення останнього списку. Використовується кнопкою видалення списків, якщо останній обраний список не відсутній;

- `List<Node> CalculateRoute(Market market, List<ProductCategory> uniqueCategories)` – метод, відповідальний за фінальний пошук оптимального шляху;
- `List<Node> ApproximateRoute(Market market, List<ProductCategory> uniqueCategories)` – метод, відповідальний за пошук приблизного шляху;
- `List<List<Node>> GetSubgraphs(List<Node> nodes, List<ProductCategory> uniqueCategories)` – метод, що розбиває поточний список на підграфи згідно унікальних категорій в ньому;
- `List<ProductCategory>GetUniqueCategories(List<Node> marketNodes)` – метод, що шукає всі унікальні категорії продуктів;
- AddListForm.cs – дочірня форма, що відповідає за додавання нових списків в formList та в вікно відображення списків:
  - Має 1 змінну (посилання на батьківську форму), 1 конструктор, 1 подію;
  - Подія полягає в перевірці текстової колонки та виклику методу додавання списку в formList у батьківській формі.
- EditListForm.cs – дочірня форма, що відповідає за редагування списків у formList:
  - Має 1 змінну (посилання на батьківську форму), 1 конструктор, 1 подію;
  - Подія полягає в перевірці текстової колонки та виклику методу редагування назви списку в formList у батьківській формі.
- AddListItemForm.cs – дочірня форма, що відповідає за додавання речей до поточного списку.
  - Має 1 змінну (посилання на батьківську форму), 1 конструктор, 2 події;
  - Подія a полягає в тому, що після завантаження форми заповнюється локальний словник категорій;

- Подія b полягає в додаванні нового предмету з параметрами вказаними в цій формі до останнього обраного списку formList.

Алгоритм пошуку оптимального шляху реалізовано цими чотирма методами:

- `List<Node> CalculateRoute(Market market, List<ProductCategory> uniqueCategories);`
- `List<Node> ApproximateRoute(Market market, List<ProductCategory> uniqueCategories);`
- `List<List<Node>> GetSubgraphs(List<Node> nodes, List<ProductCategory> uniqueCategories);`
- `List<ProductCategory>GetUniqueCategories(List<Node> marketNodes).`

Дія алгоритму починається з натискання кнопки SetActiveList. В режимі тестування створюється граф. Даний граф складається з вершин, що було згенеровано випадковим чином:

```
Outputting nodes, adding connections
0; 0;       Фрукти
10; 20;         Фрукти
30; 30;         Інше
-120; -40;      Овочі
-50; -150;      Овочі
60; -120;       М'ясні вироби
140; 70;        Овочі
160; -80;       Інше
-90; -90;       Інше
-200; 300;      Риба та морепродукти
```

Рисунок 3.10 — тестовий граф

Далі, шляхом виклику методу CalculateRoute() запускається алгоритм. На вхід подаються гіпермаркет та список унікальних категорій. Унікальні категорії знаходяться методом GetUniqueCategories(). Цей метод, за допомогою LINQ, виводить всі унікальні категорії з поданого списку продуктів. Після запуску алгоритму першим кроком будується приблизний шлях, викликається метод AproximateRoute(). На вхід отримує гіпермаркет та унікальні категорії. При пошуці приблизного шляху генеруються підграфи. Викликається метод GetSubgraphs(), який на вхід отримує вершини графу гіпермаркету та унікальні категорії. В цьому методі програма просто звіряє категорію вершини з категоріями поданих списків, та складає з них підграфи:

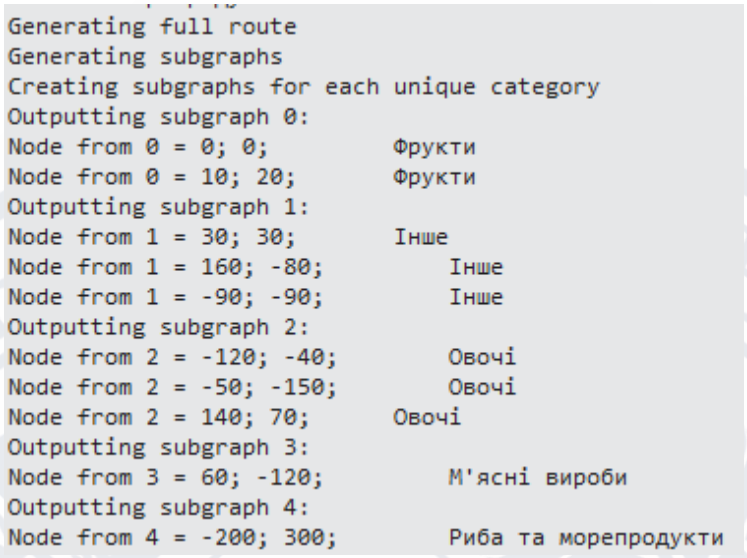

Рисунок 3.11 — підграфи категорій

Після цього продовжується робота AproximateRoute(). Даний метод ітерується цим кожним підграфом. Для кожного підграфу (тобто категорії) ми знаходимо мінімальну відстань до поточної вершини. Оцінивши кожну вершину підграфу — додаємо найближчу вершину до приблизного шляху. Переходимо до неї й оцінюємо вершини з наступної категорії. Робимо це до тих пір доки не закінчаться підграфи:

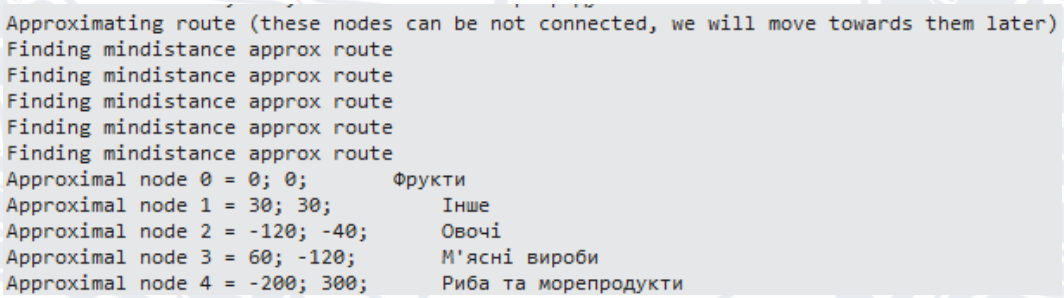

Рисунок 3.12 — приблизний шлях (5 вершин знизу)

Далі проводиться робота головного алгоритму. Шукаємо оптимальний шлях. Спочатку отримуємо посилання на вхід з гіпермаркету. Позначаємо цю вершину як початкову та додаємо до шляху. Вважаємо, що знаходимося в даній вершині. Перед запуском циклу дивимось, чи не знаходимось в приблизній вершині. Якщо знаходимось — вважаємо дану приблизну вершину пройденою, та видаляємо її зі списку приблизних вершин. Далі запускаємо цикл. Цикл триває доки змінна і не буде дорівнювати кількості приблизних вершин. Це робиться для того, щоб ми могли завершити цикл при проходженні всіх вершин. Всередині циклу

запускаємо ще один, який буде виконувати дії до тих пір, доки поточною вершиною не стане поточна приблизна вершина. Задаємо кілька змінних:

- `bool positiveChange` – змінна, що пам'ятає тип зміни відстані. Стандартне значення — false;
- `double approximalNodeCurrentDistance` – змінна, що зберігає відстань від поточної точки (точки де зараз знаходиться алгоритм) до апроксимованої оптимальної точки. Знаходиться за допомогою методу Node.CalculateDistanceBetweenNodes();
- `double bestChange` – змінна, що відповідає за збереження найкращої зміни. Стандартне значення — модуль відстані від поточної вершини до поточної приблизної вершини;
- `double leastWorstChange` – змінна, що відповідає за збереження найменш гіршої зміни. Використовується лише якщо не маємо позитивної зміни взагалі. Стандартне значення — максимальне значення типу double;
- `Node nextNode` – змінна, що зберігає посилання на наступну вершину, в яку алгоритму необхідно зайти.

Визначивши ці змінні перевіряємо зв'язки поточної вершини. Для кожної не пройденої вершини, що пов'язана з поточною вершиною ми знаходимо модуль відстані від цієї вершини до поточної приблизної вершини. Якщо цей модуль кращий за bestChange, вважаємо зв'язок пріоритетним (це означає, що пізніше в нього необхідно буде перейти), запам'ятовуємо, що є позитивна зміна, та призначаємо нову найкращу зміну. Якщо ця відстань не краща, дивимося, чи взагалі була позитивна зміна. Якщо не було — шукаємо найменш гіршу зміну. Так само запам'ятовуємо вершину та зміну. Переходимо до наступного зв'язку. Робимо так до тих пір доки не пройдемо всі зв'язки. В результаті оцінки обираємо фізично найближчу вершину, та переходимо в неї:

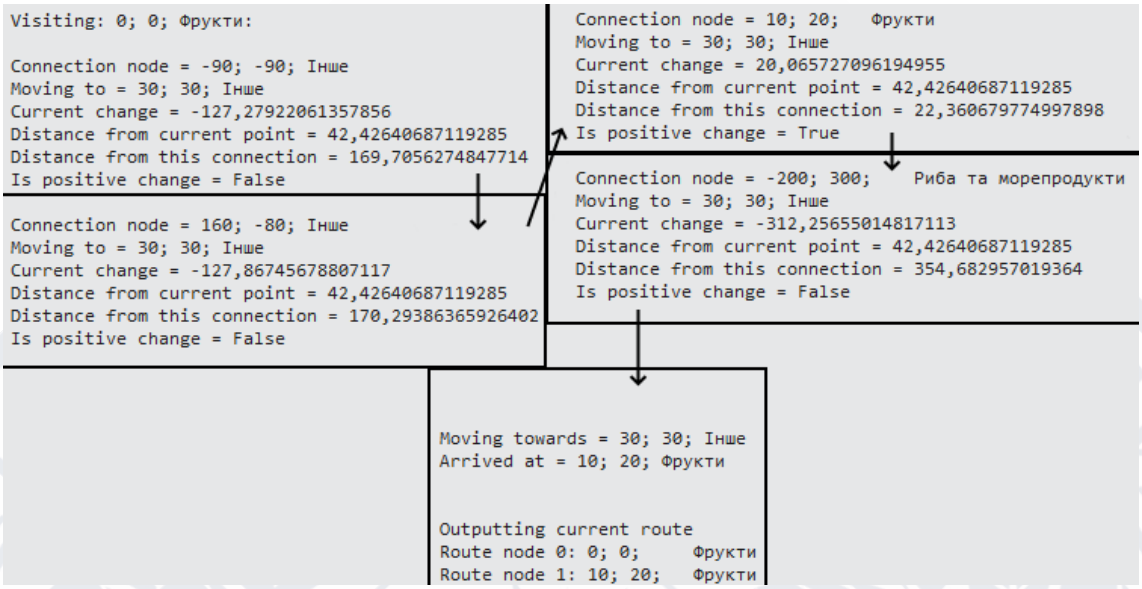

Рисунок 3.13 — оцінка зв’язків

Після оцінки переходимо в найкращий зв’язок nextNode, та додаємо до шляху й помічаємо дану вершину пройденою. Потім необхідно подивитись чи, випадково, ми не пройшли крізь приблизну вершину. В випадку якщо пройшли — оновлюємо кількість приблизних вершин, та видаляємо пройдені.

```
Outputting approximal route status: current amount of available nodes = 4
Approx route node 0: 30; 30;    Інше;   Visit status = False
Approx route node 1: -120; -40;     Овочі;  Visit status = False
Approx route node 2: 60; -120;  M'ясні вироби;  Visit status = False
Approx route node 3: -200; 300;     Риба та морепродукти;   Visit status = False
```

Рисунок 3.14 — список доступних приблизних вершин

Коли доступних вершин більше немає - алгоритм припиняє свою роботу.

```
Outputting approximal route status: current amount of available nodes = 0
Outputting generated route
0; 0;       Фрукти
10; 20;         Фрукти
30; 30;         Інше
-50; -150;      Овочі
-120; -40;      Овочі
60; -120;       M'ясні вироби
-200; 300;      Риба та морепродукти
```

Рисунок 3.15 — оптимізований та готовий, до використання, шлях

Отриманий шлях передається на mainForm за допомогою методу SetListViewItemsAndDrawGraph(), в результаті чого генерується таке зображення, яскраві лінії показують оптимальний шлях для проходу даного гіпермаркету використовуючи поданий список продуктів (вхід до гіпермаркету в жовтій точці, кінець шляху в синій точці):

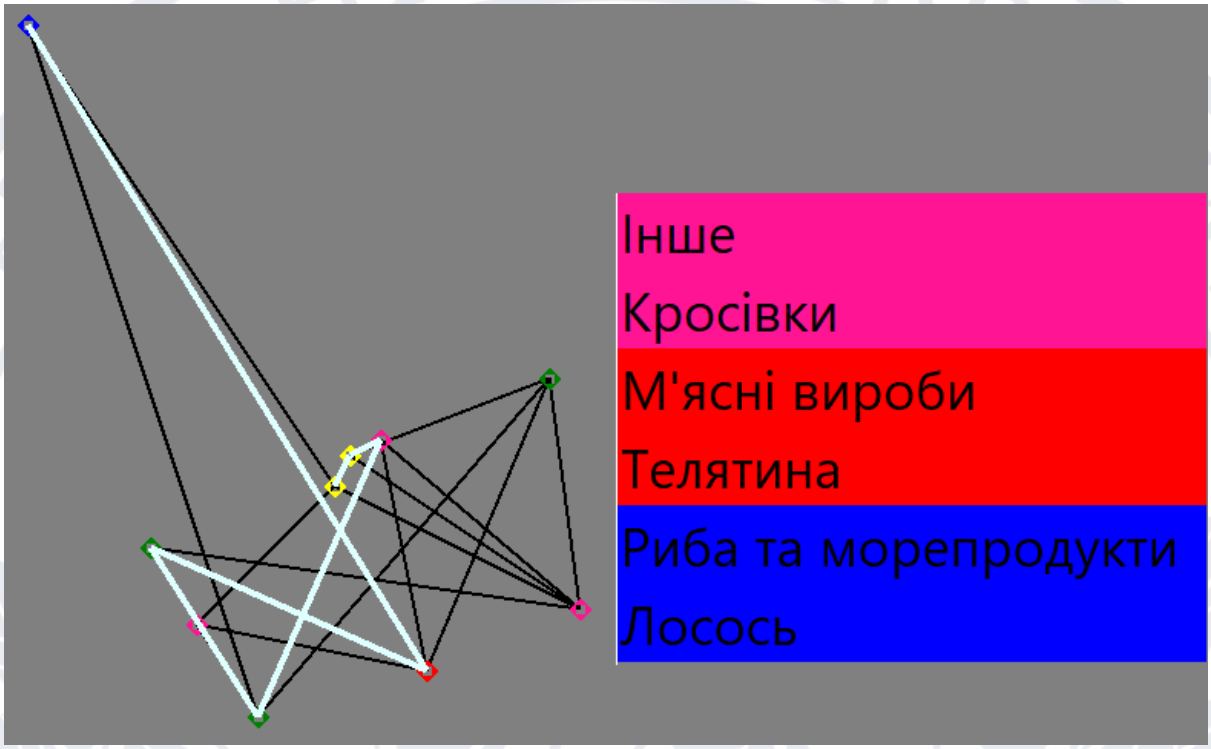

Рисунок 3.16 — оптимальний шлях на малюнку

Хоч і даний граф не є точним представленням маркету (візуальний перетин зв’язків у такому представленні графу нереальний у справжньому світі), немає жодних підстав чому цей алгоритм не можна пристосовувати до реальних планів гіпермаркетів.

3.4 Приклад застосування

При запуску програми, користувача зустрічає головна форма. Для початку роботи необхідно натиснути кнопку редактору списків. Після цього на вибір необхідно зробити одну з двох дій:

- Натиснути кнопку імпорту та обрати файл зі списками;
- Створити новий список та заповнити його речами.

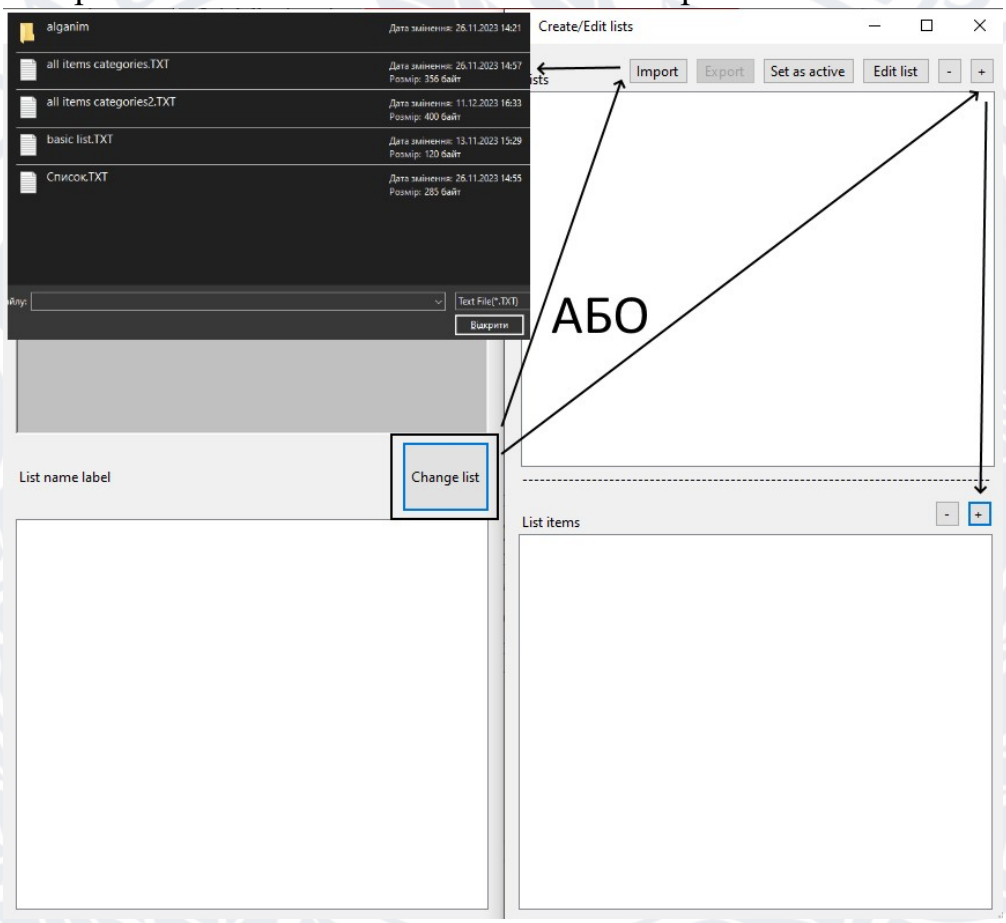

Рисунок 3.17 — початок роботи з додатком

Імпорт, при існуючих списках, підвантажить лише ті списки, що імпортують із файлу, всі інші списки буде видалено. Експорт зберігає список у вигляді текстового файлу. При експорті зберігаються предмети списків у форматі “Назва списку|Назва предмету|Назва його категорії”. Програма відображає предмети обраного списку. Якщо список не обрано, предмети не відображаються взагалі. Маючи готові списки, користувач має можливість їх редагувати, розширяти, зменшувати та зберігати.

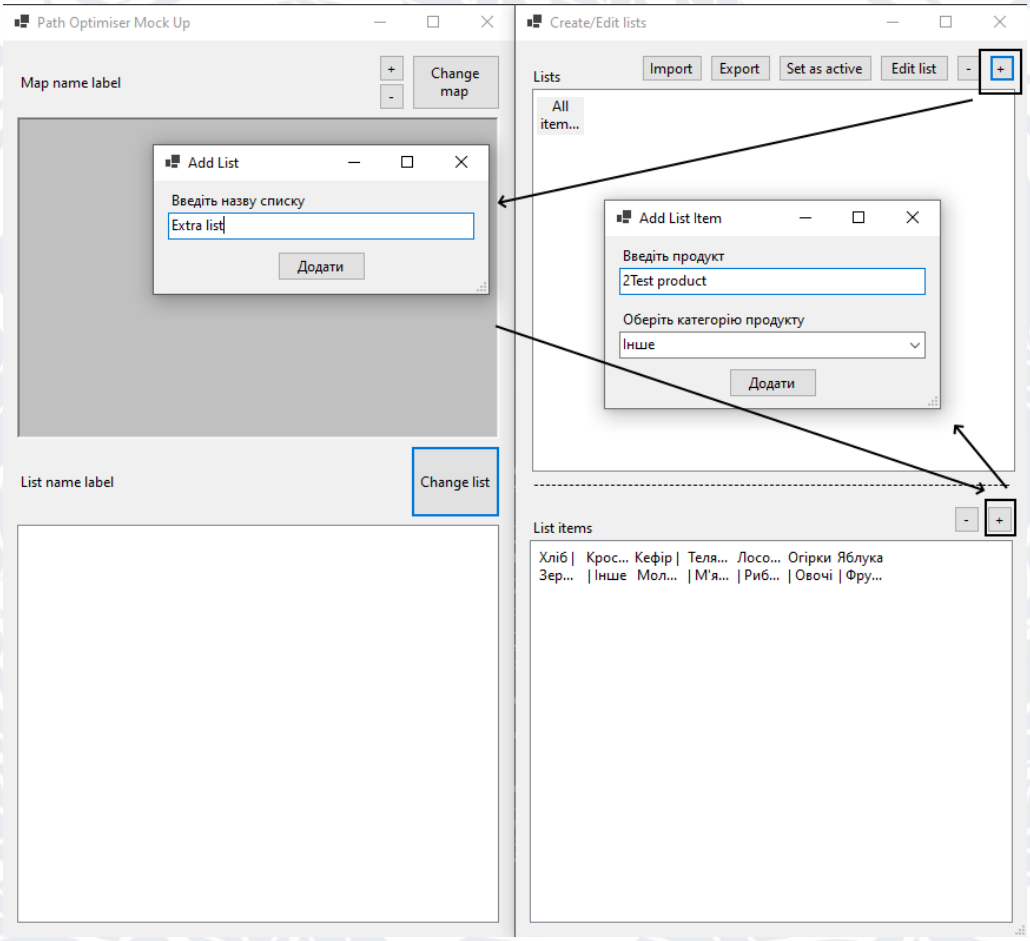

Рисунок 3.18 — робота зі списками

Завершивши редагування списків, користувачу рекомендується зберегти зміни. Лише після всіх цих дій користувач може згенерувати шлях. Для цього необхідно обрати список із рядку Lists, потім необхідно натиснути на кнопку “Set as active”. Обраний список стає активним. Це означає, що він переноситься на головну форму, після чого програма знаходить оптимальний шлях.

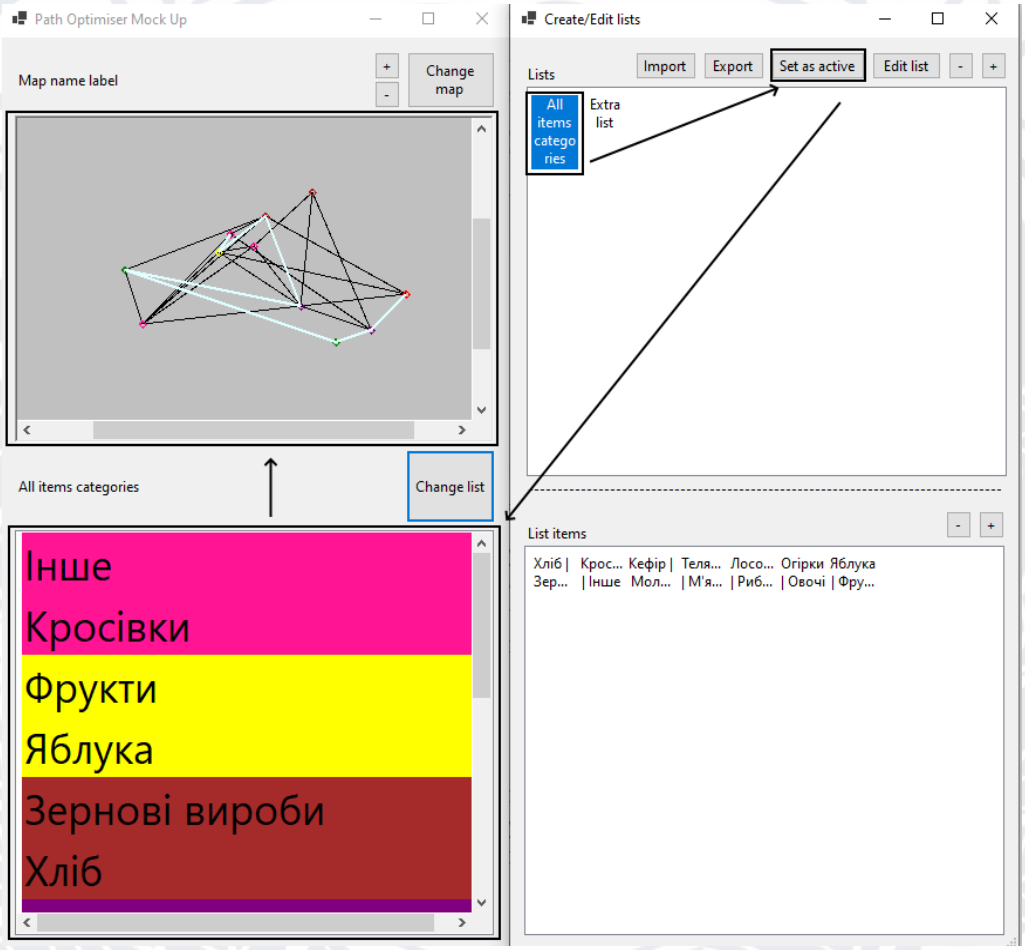

Рисунок 3.19 — активація списку

Після активізації списку, редактор списків можна спокійно зачиняти. Робота з додатком переноситься на головний екран. Тут користувач може наближувати та віддаляти мапу, звірятися з нею та переглядати стан комплектування. Програма також повідомляє для якого маркету було створено мапу, та повідомляє назву активного списку. Щоб побачити вже пройдені або відсутні продукти зі списку користувачу достатньо прокрутити список докомплектування донизу.

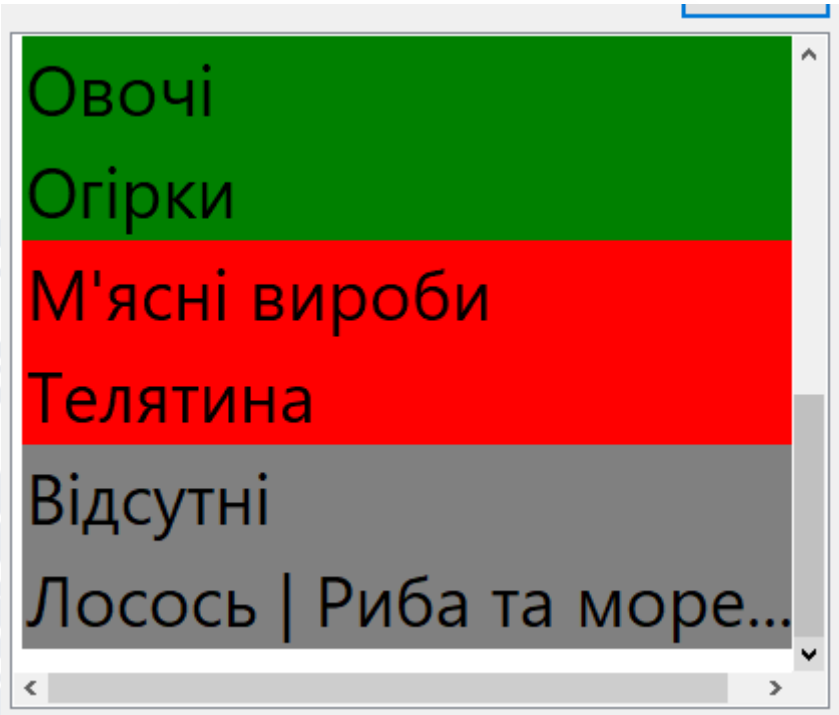

Рисунок 3.20 — пройдені/відсутні продукти

3.5 Експеримент

У ході дослідження було проведено експеримент. Суть цього експерименту полягала в проходженні гіпермаркету за заданим списком і порівнянні отриманого часу проходження зі звичайним проходом без використання додатку. Даний експеримент враховує додатковий час необхідний для того щоб знайти та покласти продукт до кошику (або відповідно задачі — щоб, навпаки, покласти продукт на полицю). Місцем проведення був локальний гіпермаркет. Для побудови графу було використано невеличкий додаток, що зберігає GPS координати при натисканні. Категорії записувалися в блокнот. Далі, на основі цих координат, було знайдено масштаб мапи. Після цього, в ручну, було задано граф.

Було виявлено, що процес побудови графу гіпермаркету — досить тривалий процес, навіть із інструментами, що полегшують його побудову. У ході дослідження було побудовано граф одного гіпермаркету. Даний гіпермаркет став майданчиком для тестування. Було проведено по 2 проходи гіпермаркету відповідно кожному списку. Списки містили повний набір категорій, напівповний набір категорій та повний набір продуктів (по 3 на категорію) при напівповному наборі категорій.

Після прогону графу через програму отримали такий результат для всіх категорій:

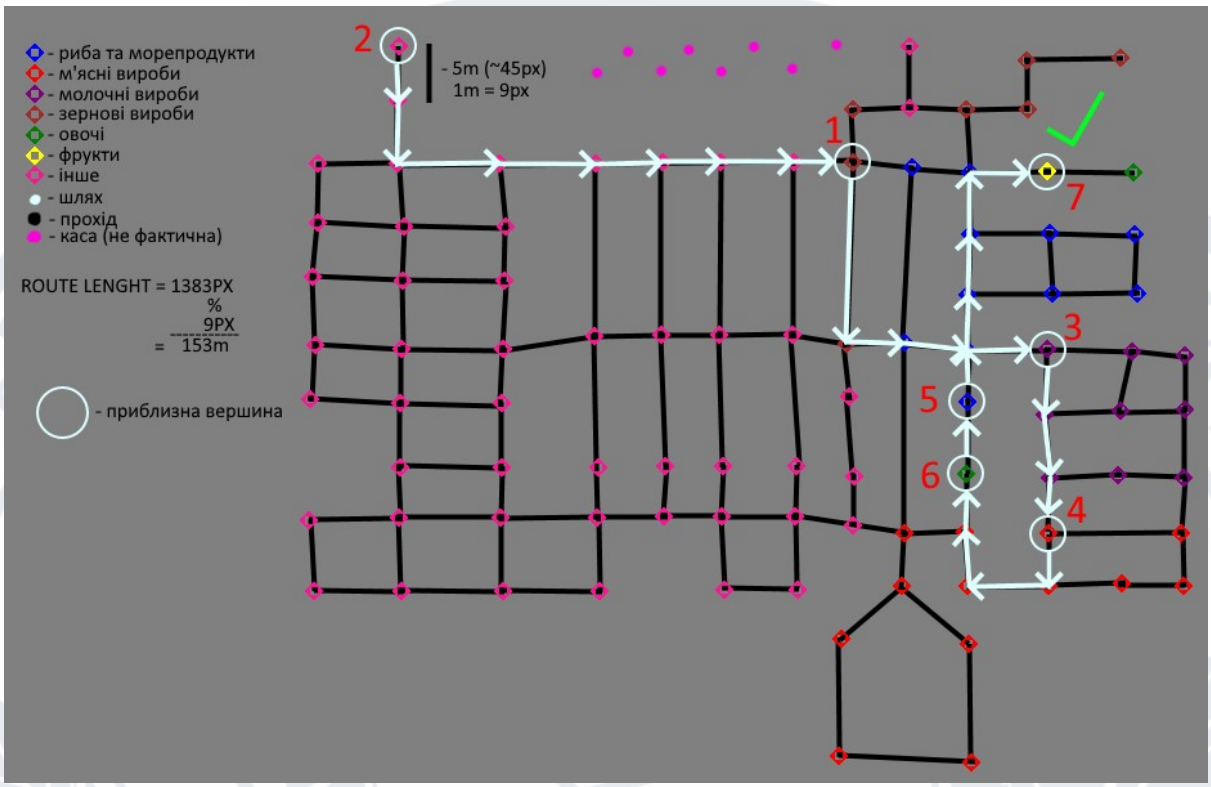

Рисунок 3.21 — експериментальний шлях

Після отримання даного шляху було здійснено прохід по гіпермаркету із та без його використання:

Користування шляхом	Використовуючи	Не використовуючи
Повні категорії (хв.)	~2:30 хв.	~3 хв.
Напівповні категорії (хв.)	~1 хв.	~1 хв.
Повні продукти (хв.)	~4:20 хв.	~5:10 хв.

З таблиці видно, що швидкість проходу із використанням шляху зменшилась на ~16%. Чим більше доводиться тримати в голові (або на папері), тим довше відбувається процес комплектування.

Висновок за третім розділом

У цьому розділі було розглянуто процес розробки додатку та його роботу. Було показано структуру модулів, дано визначення найголовнішим методам. Було реалізовано теоретичний алгоритм, відбулося його вбудування до програми. Було розроблено додаток для оптимізації шляху на графі. Було проведено дослідження ефективності оптимізованого шляху на реальному прикладі. Було покращено ефективність процесу комплектування програмою.

ВИСНОВКИ

У ході роботи було проведено повноцінне дослідження явища комплектування полиць гіпермаркетів. Було подано цінність та користь даної роботи. Було розглянуто задачу комівояжера, її суть, підходи до неї, складності рішень, типи цієї задачі, методи рішення, особливості методів, їх переваги та недоліки. Було розглянуто рішення конкурентів, функціонал, та проблеми, що вони вирішили. На основі цього було задано вектор дослідження, були визначені предмет, об’єкт, мета та задачі дослідження, було поставлено критерій оцінювання. Далі було розглянуто, кілька рішень задачі комівояжера, їх особливості реалізації, специфічність до використання, а також їх переваги та недоліки. На основі цього було поставлено проблему комплектувальника та було запропоновано алгоритм для вирішення поставленої проблеми. Було описано особливості новоутвореного алгоритму, його переваги та недоліки. Також було описано сферу його застосування. Далі, згідно задач дослідження, було розроблено програмний застосунок, що націлений на вирішення проблеми комплектувальника. Було розглянуто його програмні модулі, було подано приклади та макети дизайну. Було розглянуто особливості реалізації алгоритму в програмі, методи побудови зображення з графу, особливості використання програми. Було описано процес роботи з додатком. Було продемонстровано процес використання додатку на реальному прикладі. Було зібрано та порівняно дані критерію ефективності. В результаті виявилось, що додаток дійсно поліпшує ефективність комплектування при використанні. У ході роботи було розглянуто предмет та об’єкти дослідження, поставлені задачі було вирішено, а мету було цілком досягнуто.

СПИСОК ЛІТЕРАТУРИ

1. Anany Levitin, Introduction to The Design and Analysis of Algorithms, 2003, ISBN 0201743957;
2. Bernhard Korte, Jens Vygen, Combinatorial Optimization, 2006, ISBN 3540256849;
3. David S. Johnson, McGeoch, L. A., "The Traveling Salesman Problem: A Case Study in Local Optimization", 1997;
4. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7;
5. Dial, Robert B. (1969). "Algorithm 360: Shortest-path forest with topological ordering [H]". Communications of the ACM. 12 (11): 632–633. doi:10.1145/363269.363610. S2CID 6754003;
6. Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". Transportation Science. 32 (1): 65–73. doi:10.1287/trsc.32.1.65. S2CID 14986297;
7. Knuth, D.E. (1977). "A Generalization of Dijkstra's Algorithm". Information Processing Letters. 6 (1): 1–5. doi:10.1016/0020-0190(77)90002-3;
8. Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E. (April 1990). "Faster Algorithms for the Shortest Path Problem" (PDF). Journal of the ACM. 37 (2): 213–223. doi:10.1145/77600.77615. hdl:1721.1/47994. S2CID 5499589;
9. Thorup, Mikkel (2000). "On RAM priority Queues". SIAM Journal on Computing. 30 (1): 86–109. doi:10.1137/S0097539795288246. S2CID 5221089;
10. Thorup, Mikkel (1999). "Undirected single-source shortest paths with positive integer weights in linear time". Journal of the ACM. 46 (3): 362–394. doi:10.1145/316542.316548. S2CID 207654795;
11. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009). "Engineering Route Planning Algorithms". Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation. Lecture Notes in Computer Science. Vol. 5515. Springer. pp. 117–139. doi:10.1007/978-3-642-02094-0_7. ISBN 978-3-642-02093-3;
12. Zeng, W.; Church, R. L. (2009). "Finding shortest paths on real road networks: the case for A*". International Journal of Geographical

Information Science. 23 (4): 531–543. doi:10.1080/13658810801949850. S2CID 14833639;
13. Hart, P. E.; Nilsson, N.J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics. 4 (2): 100–7. doi:10.1109/TSSC.1968.300136;
14. Doran, J. E.; Michie, D. (1966-09-20). "Experiments with the Graph Traverser program". Proc. R. Soc. Lond. A. 294 (1437): 235–259. Bibcode:1966RSPSA.294..235D. doi:10.1098/rspa.1966.0205. S2CID 21698093;
15. Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*". Journal of the ACM. 32 (3): 505–536. doi:10.1145/3828.3830. S2CID 2092415;
16. De Smith, Michael John; Goodchild, Michael F.; Longley, Paul (2007), Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools, Troubadour Publishing Ltd, p. 344, ISBN 9781905886609;
17. Martelli, Alberto (1977). "On the Complexity of Admissible Search Algorithms". Artificial Intelligence. 8 (1): 1–13. doi:10.1016/0004-3702(77)90002-9;
18. Felner, Ariel; Uzi Zahavi (2011). "Inconsistent heuristics in theory and practice". Artificial Intelligence. 175 (9–10): 1570–1603. doi:10.1016/j.artint.2011.02.001;
19. Zhang, Zhifu; N. R. Sturtevant (2009). Using Inconsistent Heuristics on A* Search. Twenty-First International Joint Conference on Artificial Intelligence. pp. 634–639;
20. Pohl, Ira (1970). "First results on the effect of error in heuristic search". Machine Intelligence 5. Edinburgh University Press: 219–236. ISBN 978-0-85224-176-9. OCLC 1067280266.

Додаток 2 до наказу
від «31» березня 2023 року
№119/05

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я,_____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)