НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

«На правах рукопису»
УДК 004.021

«До захисту допущено»
Завідувач кафедри
_____ О.Л. Тимощук
«___»_____2018 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 124 Системний аналіз

на тему: «Мультиагентна система маршрутизації на основі алгоритмів пошуку найкоротшого шляху в графі»

Виконав:
студент ІІ курсу, групи КА-62м
Тішков Максим Олегович _____

Керівник:
Доцент, к.т.н., доц.,
Тимощук О.Л. _____

Рецензент:
Доцент кафедри системного проектування, к.т.н., доцент
Кисельов Г. Д. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ 2018

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 124 Системний аналіз (Системи і методи підтримки прийняття рішень)

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ О.Л. Тимощук

«___»_____2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Тішков Максим Олегович

1. Тема дисертації «Мультиагентна система маршрутизації на основі алгоритмів пошуку найкоротшого шляху в графі», науковий керівник дисертації Тимощук Оксана Леонідівна, доцент, к.т.н, затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом дисертації _____

3. Об’єкт дослідження: побудова мультиагентної системи маршрутизації на основі алгоритмів пошуку маршруту

4. Предмет дослідження: алгоритми пошуку маршрутів в графі

5. Перелік завдань, які потрібно розробити: огляд літератури з питань побудови систем маршрутизації для мобільних пристроїв та у вигляді клієнт-серверного застосунку, огляд існуючих алгоритмів пошуку маршрутів у графах та вибір алгоритмів для реалізації, огляд літератури з питання створення графу реальних доріг, генерація графу в базі даних, реалізація алгоритмів, тестування розробленої системи на маршрутах різної складності, аналіз отриманих результатів та порівняння якості роботи алгоритмів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграми для представлення роботи обраних алгоритмів, схеми бази даних та UML-діаграми класів розробленої системи, приклади побудованих маршрутів.

7. Орієнтовний перелік публікацій: -

8. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою дисертації	16.03.2018	Виконано
2.	Оформлення першого розділу магістерської дисертації	23.03.2018	Виконано
3.	Оформлення другого розділу магістерської дисертації	26.03.2018	Виконано
4.	Реалізація агенту генерації графу доріг	30.03.2018	Виконано
5.	Реалізація алгоритмів маршрутизації	10.04.2018	Виконано
6.	Оформлення третього розділу магістерської дисертації	24.04.2018	Виконано
7.	Тестування розробленої системи	27.04.2018	Виконано
8.	Оформлення четвертого розділу магістерської дисертації	03.05.2018	Виконано
9.	Оформлення висновків	09.05.2018	Виконано

Студент М.О. Тішков

Науковий керівник дисертації О.Л. Тимощук

РЕФЕРАТ

Магістерська дисертація: 145 с., 45 рис., 24 табл., 2 додатки, 25 джерел.

Дана робота присвячена дослідженню проблем побудови маршрутів на графах реальних доріг та створенню мультиагентної системи для маршрутизації та створення відповідного графу. В роботі досліджується можливість створення графу доріг з наявних картографічних даних, створюється граф доріг Києва, розглядаються існуючі методи пошуку оптимальних маршрутів на графах, реалізовуються 3 алгоритми та порівнюються результати їх роботи.

Метою даної роботи є побудова мультиагентної системи, що дозволить створювати графи доріг за даними сервісу OpenStreetMap та виконувати пошук маршрутів будь-якої складності на побудованих графах за допустимий час.

Об’єктом дослідження є побудова мультиагентної системи маршрутизації на основі алгоритмів пошуку маршруту.

Предметом дослідження є алгоритми пошуку маршрутів на графах.

Практична цінність: Розроблена система може слугувати веб-сервісом для мобільних додатків та веб сайтів та надавати можливості роутингу.

МУЛЬТИАГЕНТНА СИСТЕМА, МЕРШРУТИЗАЦІЯ, РОУТИНГ, OPENSTREETMAP, АЛГОРИТМ ДЕЙКСТРИ, АЛГОРИТМ A*, ДВОНАПРВЛЕНІ АЛГОРИТМИ ПОШУКУ МАРШРУТУ, CONTRACTION HIERARCHIES.

ABSTRACT

Master’s thesis: 145 p., 45 fig., 24 tab., 2 appendixes, 25 sources.

The theme: Multi agent routing system based on shortest path search algorithms in graph.

The paper is devoted to the problem of route construction on the real road network graphs and multi agent routing system creation. The paper considered by the possibility of road graph creating using cartographic data, creating road graph for Kyiv road network, researching methods for searching optimal routes on graphs and implementation of three routing algorithms.

The aim is to build multi agent system, that allows us to create road graphs using cartographic data from OpenStreetMap service and to build any complexity routes on created graphs for a reasonable time.

The object of research is multi agent routing system construction based on route finding algorithms.

The subject of research is route finding algorithms on graphs.

Practical value: The system can be used as web-service by mobile applications and web-sites to provide a routing opportunity

MULTI AGENT SYSTEM, ROUTING, OPENSTREETMAP, DIJKSTRA ALGORITHM, A* ALGORITHM, BIDIRECTIONAL ROUTE FINDING ALGORITHMS, CONTRACTION HIERARCHIES.

ЗМІСТ

ВСТУП

Високі темпи інформатизації різних видів діяльності в даний час привели до того, що з'явилася можливість комп'ютерного моделювання та проектування складних систем, вивчення їх властивостей і управління ними в умовах дефіциту часу, обмеженості ресурсів, неповноти інформації. Однак для дослідження характеристик будь-якої системи математичними методами повинна бути обов'язково виконана формалізація, тобто побудована математична модель. Дослідження за допомогою математичних моделей найчастіше є єдино можливим способом вивчення складних систем і вирішення найважливіших практичних завдань управління.

Графи виявилися гарною математичною моделлю широкого класу об'єктів і процесів. Теорія графів застосовується в таких областях, як фізика, хімія, теорія зв'язку, проектування ЕОМ, електроніка, машинобудування, архітектура, дослідження операцій, генетика, психологія, соціологія, економіка, антропологія і лінгвістика. При цьому зазвичай на графі вирішуються завдання пошуку оптимального маршруту, досяжності, завдання мережевого планування, потокова задача.

Першою роботою з теорії графів як математичної дисципліни вважається стаття Ейлера (1736 р.), у якій було розглянуто задачу про кенінгсбергські мости. Ейлер показав, що не можна обійти сім міських мостів та повернутися в початкову точку, пройшовши при цьому по кожному з мостів лише один раз. Наступний імпульс теорія графів отримала майже через 100 років з розвитком досліджень електричних мереж, кристалографії, органічної хімії та інших наук.

Зараз графи використовують аж ніяк не тільки як ілюстрацію. Наприклад, розглядаючи граф, що зображує мережу доріг між населеними пунктами, можна визначити маршрут проїзду від пункту А до пункту Б. Якщо

таких маршрутів виявиться декілька, то хотілося б обрати в певному сенсі оптимальний, наприклад, найкоротший або ж найбезпечніший. Для розв’язку задачі вибору вимагається провести певні обчислення над графами. При розв’язанні подібних задач виникає необхідність використання потужної обчислювальної техніки.

Зараз теорія графів охоплює велику кількість матеріалу та активно розвивається в багатьох напрямках. Нас же цікавить задача пошуку маршруту. Тому при описі алгоритмів та методів зупинимося на тих, що стосуються маршрутизації. Побудова математичного визначення графу здійснюється шляхом формалізації «об’єктів» та «зв’язків» як елементів деяких множин.

РОЗДІЛ 1. ІСНУЮЧІ ЗАДАЧІ ТЕОРІЇ ГРАФІВ ТА МЕТОДИ ЇХ РОЗВ'ЯЗАННЯ

1.1 Задача пошуку оптимального маршруту

Задача про найкоротший шлях – задача пошуку короткого шляху (ланцюгу) між двома точками (вершинами) на графі, у якому мінімізується сума ваг ребер, що утворюють шлях.

Найкоротший (простий) ланцюг часто називається геодезичним. Задача про найкоротший шлях є однією з найважливіших класичних задач теорії графів. Сьогодні відомо безліч алгоритмів для її вирішення [1].

Значимість даної задачі визначається її різними практичними застосуваннями. Наприклад в GPS-навігаторах, де здійснюється пошук найкоротшого шляху між двома точками на карті. В якості вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Сума відстаней всіх доріг між точками повинна бути мінімальною, тоді знайдено найкоротший шлях.

На рис. 1.1 зображено найкоротший маршрут в неорієнтованому графі без ваг. На рис. 1.2 зображено найкоротший маршрут на орієнтованому графі з вагами

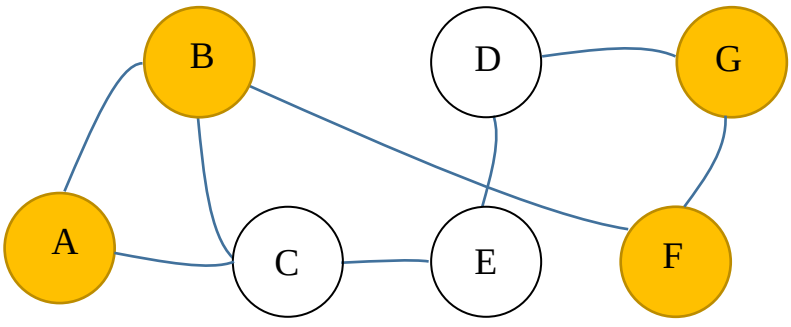


Рисунок 1.1 - Неорієнтований граф без ваг

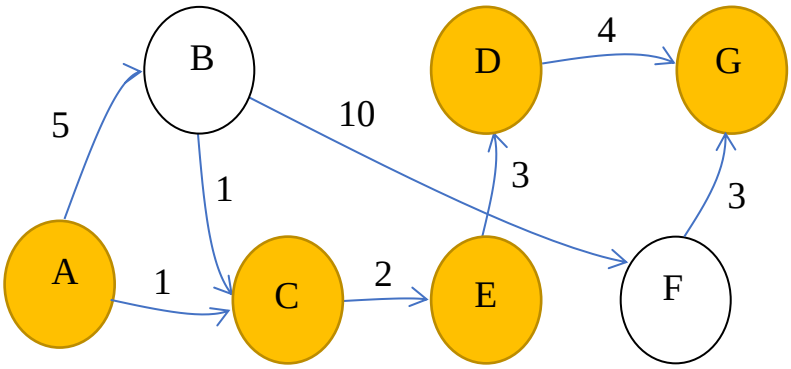


Рисунок 1.2 - Орієнтований граф з вагами

Задача пошуку найкоротшого шляху на графі може бути визначена для неорієнтованого, орієнтованого або змішаного графа. Далі буде розглянута постановка задачі в її найпростішому вигляді для неорієнтованого графа. Для змішаного і орієнтованого графа додатково повинні враховуватися напрямки ребер [2].

Граф являє собою сукупність непустої множини вершин та ребер (набір пар вершин). Дві вершини у графі є суміжними, якщо вони з'єднуються спільним ребром. Шлях у неорієнтованому графі представляє собою послідовність вершин $P = (v_1, v_2, \dots, v_n) \in V \times V \times \dots \times V$, таких що v_i суміжна з v_{i+1} для $1 < i < n$. Такий шлях P називається шляхом довжини n з вершини v_1 у v_n (i вказує на номер вершини шляху та не має жодного відношення до нумерації вершин у графі).

Нехай $e_{i,j}$ — ребро, що з'єднує дві вершини: v_i та v_j. Дана вагова функція $f: E \to R$, яка відображає ребра на їх ваги, значення яких виражається дійсними числами, та неорієнтований граф G. Тоді найкоротшим шляхом з вершини v до вершини v' буде називатися шлях $P = (v_1, v_2, \dots, v_n)$ (де $v_1 = v$ та $v_n = v'$), який має мінімальне значення суми $\sum_{i=1}^{n-1} f(e_{i,i+1})$. Якщо усі ребра у графі мають одиничну вагу, то задача задача зводиться до визначення найменшої кількості обійдених ребер [3].

1.2 Постановки задачі

Існують різні постановки задачі про найкоротший шлях:

1. Задача про найкоротший шлях в заданий пункт призначення. Потрібно знайти найкоротший шлях в задану вершину призначення t, який починається в кожній з вершин графа (крім t). Помінявши напрямок кожного ребра, що належить графу, це завдання можна звести до задачі про єдину вихідну вершину (у якій здійснюється пошук найкоротшого шляху з заданої вершини в усі інші);
2. Задача про найкоротший шлях між заданою парою вершин. Потрібно знайти найкоротший шлях із заданої вершини u в задану вершину v;
3. Задача про найкоротший шлях між усіма парами вершин. Потрібно знайти найкоротший шлях з кожної вершини u в кожну вершину v. Це завдання теж можна вирішити за допомогою алгоритму, призначеного для вирішення задачі про одну вихідну вершину, проте зазвичай вона вирішується швидше.

У різних постановках задачі, роль довжини ребра можуть грати не тільки самі довжини, але й час, вартість, витрати, обсяг витрачених ресурсів (матеріальних, фінансових, паливно-енергетичних і т. п.) або інші характеристики, пов'язані з проходженням кожного ребра. Таким чином, завдання знаходить практичне застосування у великій кількості областей (інформатика, економіка, географія та ін.) [3].

1.3 Можливі обмеження

Задача про найкоротший шлях дуже часто зустрічається в ситуації, коли необхідно враховувати додаткові обмеження. Їх наявність може значно підвищити складність завдання. Приклади таких завдань:

1. Найкоротший шлях, що проходить через задану множину вершин. Можна розглядати два обмеження: найкоротший шлях повинен проходити через виділену множину вершин, і найкоротший шлях повинен містити якомога менше невиділених вершин. Перше з них добре відоме в теорії дослідження операцій;

2. Мінімальне покриття вершин орієнтованого графа шляхами. Здійснюється пошук мінімального за кількістю шляхів покриття графа, а саме підмножини всіх s-t шляхів, таких що, кожна вершина орієнтованого графа належить хоча б одному такому шляху;

3. Задача про необхідні шляхи. Необхідно знайти мінімальну за потужністю множину s-t шляхів $P = p_1, \ldots, p_m$ таке, щоб для кожного $t_i \in R$ існував шлях $p_j \in P$, що накриває його. $R = t_1, \ldots, t_k$- множина певрних шляхів у орієнтованому графі G;

4. Мінімальне покриття дуг орієнтованого графа шляхами. Завдання полягає у знаходженні мінімальної за кількістю шляхів підмножини всіх шляхів, такої, щоб кожна дуга графа належила хоча б одному такому шляху. При цьому може мати місце додаткова вимога про те, щоб всі шляхи виходили з однієї вершини.

1.4 Застосування задачі пошуку маршруту

Задача про пошук найкоротшого шляху на графі може бути інтерпретована по-різному і застосовуватися в різних областях. Далі наведено приклади різних застосувань задачі. Інші застосування вивчаються в дисципліні, яка займається дослідженням операцій.

Алгоритми знаходження найкоротшого шляху на графі застосовуються для знаходження шляхів між фізичними об'єктами на таких картографічних сервісах, як карти Google або OpenStreetMap.

Якщо уявити недетерміновану абстрактну машину як граф, де вершини описують стан, а ребра визначають можливі переходи, тоді алгоритми пошуку найкоротшого шляху можуть бути застосовані для пошуку оптимальної послідовності рішень для досягнення головної мети. Наприклад, якщо вершинами є стани Кубика Рубіка, а ребро являє собою одну дію над кубиком, тоді алгоритм може бути застосований для пошуку рішення з мінімальною кількістю ходів.

Задача пошуку найкоротшого шляху на графі широко використовується при визначенні найменшої відстані в мережі доріг. Мережу доріг можна представити у вигляді графа з позитивними вагами. Вершини є дорожніми розв'язками, а ребра дорогами, які їх з'єднують. Ваги ребер можуть відповідати протяжності даної ділянки, часу необхідному для її подолання або вартості подорожі по ньому. Орієнтовані ребра можна використовувати для представлення односторонніх вулиць. У такому графі можна ввести характеристику , яка вказує на те , що одні дороги важливіші за інші для тривалих подорожей (наприклад автомагістралі). Вона була формалізована в понятті (ідеї) про магістралі. Для реалізації підходу, де одні дороги важливіші за інші, існує безліч алгоритмів. Вони вирішують задачу пошуку найкоротшого шляху набагато швидше, ніж аналогічні на звичайних графах [4]. Подібні алгоритми складаються з двох етапів:

1. Проводиться попередня обробка графа без урахування початкової та кінцевої вершини (може тривати до декількох днів, якщо працювати з реальними даними). Зазвичай виконується один раз і потім використовуються отримані дані;
2. Здійснюється запит і пошук найкоротшого шляху, при цьому відомі початкова та кінцева вершина.

1.5 Алгоритми пошуку маршруту

До найбільш популярних алгоритмув пошуку маршруту в графі можна віднести:

1. Алгоритм Дейкстри знаходить найкоротший шлях від однієї з вершин графа до всіх інших. Алгоритм працює тільки для графів без ребер з негативною вагою;
2. Алгоритм Беллмана-Форда знаходить найкоротші шляхи від однієї вершини графа до всіх інших у зваженому графі. Вага ребер може бути негативною;
3. Алгоритм пошуку A * знаходить маршрут з найменшою вартістю від однієї вершини (початкової) до іншої (цільової, кінцевої), використовуючи алгоритм пошуку по першому найкращому збігу на графі;
4. Алгоритм Флойда-Уоршелла знаходить найкоротші шляхи між усіма вершинами зваженого орієнтованого графа;
5. Алгоритм Джонсона знаходить найкоротші шляхи між усіма парами вершин зваженого орієнтованого графа;
6. Алгоритм Лі (хвильовий алгоритм) заснований на методі пошуку в ширину. Знаходить шлях між вершинами s і t графа (s не збігається з t), що містить мінімальну кількість проміжних вершин (ребер). Основне застосування - трасування електричних з'єднань на кристалах мікросхем і на друкованих платах. Так само використовується для пошуку найкоротшої відстані на карті в стратегічних іграх;
7. Contraction hierarchies. Алгоритм з переодбробкою графу для знаходження коротших шляхів і «віртуального» видалення вершин які можна пропустити при пошуку маршруту.

1.6 Алгоритм Дейкстри

Алгоритм голландського вченого Едсгер Дейкстри знаходить всі найкоротші шляхи з однієї наперед заданій вершини графа до всіх інших. З його допомогою, при наявності всієї необхідної інформації, можна, наприклад, дізнатися яку послідовність доріг краще використовувати, щоб дістатися з одного міста до кожного з багатьох інших, або в які країни вигідніше експортувати нафту тощо.

Мінусом даного методу є неможливість обробки графів, в яких є ребра з негативною вагою. Якщо, наприклад, деяка система передбачає збиткові для фірми маршрути, то для роботи з нею варто скористатися іншим алгоритмом [5].

Для програмної реалізації алгоритму знадобитися два масиви: логічний visited - для зберігання інформації про відвідані вершинах і чисельний distance, в який будуть заноситися знайдені найкоротші шляхи. Отже, є граф G = (V, E). Кожна з вершин входять в множину V, спочатку відзначена не відвіданою, тобто елементам масиву visited присвоєно значення false.

Оскільки найвигідніші шляхи тільки належить знайти, в кожен елемент вектора distance записується таке число, яке свідомо більше будь-якого потенційного шляху (зазвичай це число називають нескінченністю, але в програмі використовують, наприклад максимальне значення конкретного типу даних). В якості вихідного пункту вибирається вершина s і їй присвоюється нульовий шлях: distance [s] = 0, оскільки Немає ребра з s в s (метод не передбачає петель).

Далі, знаходяться всі сусідні вершини (в які є ребро з s). Нехай такими будуть t і u. І по черзі досліджуються, а саме обчислюється вартість маршруту з s черзі в кожну з них.

Але цілком ймовірно, що в ту чи іншу вершину з s існує кілька шляхів, тому ціну шляху в таку вершину в масиві distance доведеться переглядати, тоді найбільше (неоптимальне) значення ігнорується, а найменше ставитися у відповідність вершині.

Після обробки суміжних з s вершин вона позначається як відвідана і активною стає та вершина, шлях з s в яку мінімальний. Припустимо, шлях з s в u коротше, ніж з s в t, отже, вершина u стає активною і. Як і для вершини s досліджуються сусіди u, за винятком вершини s. Далі, u позначається як пройдена, активною стає вершина t, і вся процедура повторюється для неї. Алгоритм Дейкстри триває до тих пір, поки всі доступні з s вершини не будуть досліджені.

На рис. 1.3 зображена послідовність кроків пошуку найкоротших шляхів для всіх вершин з першої для конкретного графа.

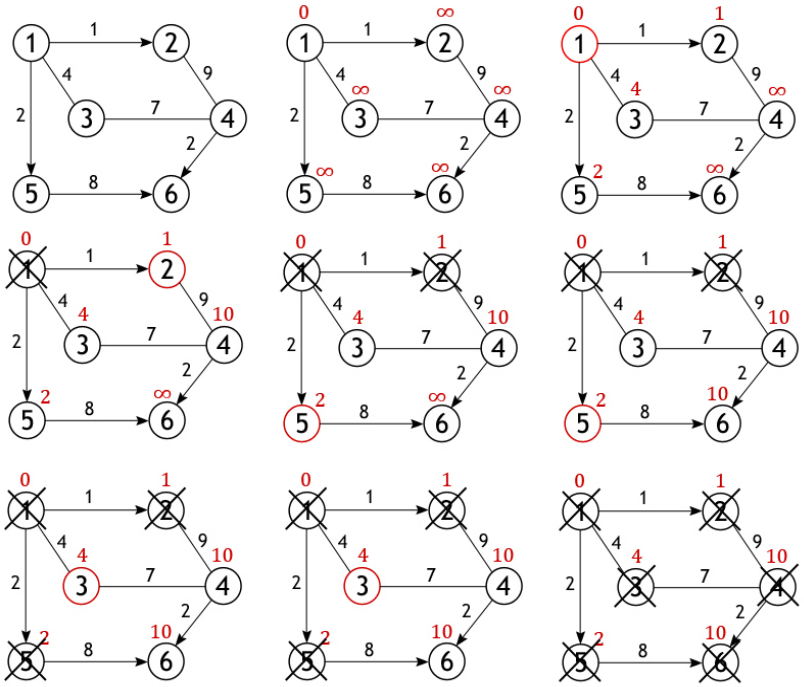


Рисунок 1.3 - Алгоритм Дейкстри

1.7 Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда - алгоритм пошуку найкоротшого шляху у зваженому графі. За час $O(|V| \cdot |E|)$ алгоритм знаходить найкоротші шляхи від однієї вершини графа до всіх інших. На відміну від алгоритму Дейкстри, алгоритм Белмана-Форда допускає ребра з негативною вагою. Запропоновано незалежно Річардом Белманом і Лестером Фордом [6].

Маємо орієнтований або неорієнтовані граф G зі зваженими ребрами. Довжиною шляху назвемо суму ваг ребер, що входять в цей шлях. Потрібно знайти найкоротші шляхи від виділеної вершини s до всіх вершин графа.

Зауважимо, що найкоротших шляхів може не існувати. Так, у графі, що містить цикл з негативною сумарною вагою, існує як завгодно короткий шлях від однієї вершини цього циклу до іншої (кожен обхід циклу зменшує довжину шляху). Цикл, сума ваг ребер якого негативна, називається негативним циклом.

Вирішимо поставлену задачу на графі, в якому свідомо немає негативних циклів. Для знаходження найкоротших шляхів від однієї вершини до всіх інших, скористаємося методом динамічного програмування. Побудуємо матрицю A_{ij}, елементи якої будуть означати наступне: A_{ij} — це довжина найкоротшого шляху з вершини s у вершину i, що містить не більше j ребер.

Шлях, що містить 0 ребер, існує лише до вершини s. Таким чином, A_{i0} рівне 0 при $i = s$, та +∞ у протилежному випадку.

Тепер розглянемо усі шляхи з s до i, що містять рівно j ребер. Кожен такий шлях є шляхом з $j - 1$ ребра, до якого додане останнє ребро. Якщо про довжини шляху $j - 1$ усі дані вже підраховані, то визначити $j -$ й стовпчик матриці не складає труднощів.

1.8 Алгоритм A*

Пошук A * (вимовляється «A зірка» або «A стар», від англ. A star) - у інформатиці та математиці, алгоритм пошуку по першому найкращому збігу на графі, який знаходить маршрут з найменшою вартістю від однієї вершини (початкової) до іншої (цільової, кінцевої).

Порядок обходу вершин визначається евристичної функцією «відстань + вартість» (зазвичай позначається як f(x)). Ця функція - сума двох інших: функції вартості досягнення розглянутої вершини (x) з початкової вершини (зазвичай позначається як g(x) і може бути як евристичною, так і ні) і евристичної оцінки відстані від розглянутої вершини до кінцевої (позначається як h (x)).

Функція h (x) повинна бути припустимою евристичної оцінкою, тобто не повинна переоцінювати відстані до цільової вершині. Наприклад, для завдання маршрутизації h(x) може являти собою відстань до цілі по прямій лінії, так як це фізично найменша можлива відстань між двома точками.

Цей алгоритм був вперше описаний в 1968 році Пітером Хартом, Нільсом Нільсоном і Бертрамом Рафаелем. Це по суті було розширення алгоритму Дейкстри, створеного в 1959 році. Він досягав більш високої продуктивності (за часом) за допомогою евристики. В їх роботі він згадується як «алгоритм A». Але так як він обчислює найкращий маршрут для заданої евристики, він був названий A* [7].

A * покроково переглядає всі шляхи, що ведуть від початкової вершини в кінцеву, поки не знайде мінімальний. Як і всі інформовані алгоритми пошуку, він переглядає спочатку ті маршрути , які «здається » ведуть до мети. Від кожного алгоритму (який теж є алгоритмом пошуку по першому кращому збігу) його відрізняє те, що при виборі вершини він враховує, весь пройдений до неї шлях (складова g(x) - це вартість шляху від початкової вершини, а не

від попередньої, як в жадібних алгоритмах). На початку роботи проглядаються вузли, суміжні з початковим; вибирається той з них, який має мінімальне значення f (x) , після чого цей вузол розкривається. На кожному етапі алгоритм оперує з безліччю шляхів з початкової точки до всіх ще не розкритих (листових) вершин графа («безліччю приватних рішень»), яка розміщується у черзі за пріоритетом. Пріоритет шляху визначається за значенням f (x) = g(x) + h(x) . Алгоритм продовжує свою роботу до тих пір, поки значення f(x) цільової вершини не виявиться меншим, ніж будь-яке значення в черзі (або поки все дерево не буде переглянуто) . З множинних рішень вибирається рішення з найменшою вартістю. Чим менше евристика h(x), тим більше пріоритет (тому для реалізації черги можна використовувати сортувальні дерева) [8].

1.9 Алгоритм Флойда Уоршела

Алгоритм Флойда-Уоршела - динамічний алгоритм для знаходження найкоротших відстаней між усіма вершинами зваженого орієнтованого графа. Розроблено в 1962 році Робертом Флойдом і Стівеном Уоршелом, хоча в 1959 році Бернард Рой (Bernard Roy) опублікував практично такий же алгоритм, але це залишилося непоміченим.

Нехай вершини графа $G = (V, E), |V| = n$ пронумеровані від 1 до n та введено позначення d_{ij}^{k} для довжини найкоротшого шляху від i до j, який окрім самих вершин i, j проходить лише через вершини $1 \dots k$. Очевидно, що d_{ij}^{k} — довжина (вага) ребра (i, j) якщо таке існує (в протилежному випадку його довжина може бути позначена як ∞).

Існує два варіанти значення $d_{ij}^{k}, k \in (1, \dots, n)$:

- найкоротший шлях між i, j не проходить через вершину k, тоді $d_{ij}^{k} = d_{ij}^{k-1}$

- існує більш короткий шлях між i, j, що проходить через вершину k, тоді він спочатку йде від i до k, а потім від k до j. В цьому випадку, очевидно, що $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$.

Таким чином, для знаходження значення функції достатньо обрати мінімум з двох відповідних значень. Тоді рекурентна формула для d_{ij}^{k} має вигляд:

$$d_{ij}^{k} = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}),$$

де d_{ij}^{0} - довжина ребра (i, j).

Алгоритм Флойда-Уоршела послідовно визначає усі значення $d_{ij}^{k}, \forall i, j, k \in [1, n]$. Отримані значення d_{ij}^{k} є довжинами найкоротших шляхів між вершинами (i, j) [9].

1.10 Алгоритм Джонсона

Алгоритм Джонсона дозволяє знайти найкоротші шляхи між усіма парами вершин зваженого орієнтованого графа. Даний алгоритм працює, якщо в графі містяться ребра з позитивною або негативною вагою, але відсутні цикли з негативною вагою.

Даний граф $G = (V, E)$ з ваговою функцією $\omega: E \to R$. Якщо ваги ребер ω у графі невід'ємні, можна знайти найкоротші шляхи між усіма парами вершин, запустивши алгоритм Дейкстри по одному разу для кожної вершини. Якщо в графі містяться ребра з негативною вагою, але відсутні цикли з

негативною вагою, можна обчислити нову множину ребер з невід'ємними вагами, що дозволяє скористатися попереднім методом. Нова множина, що складається з ваг ребер $\widehat{\omega}$, повинна задовольняти наступні вимоги:

- для усіх ребер (u, v) нова вага $\widehat{\omega}(u, v) > 0$;
- для усіх пар вершин $u, v \in V$ шлях P є найкоротшим шляхом з вершини u до вершини v з використанням вагової функції ω тоді і тільки тоді, коли P — також найкоротший шлях з вершини u до вершини v з ваговою функцією $\widehat{\omega}$ [10].

1.11 Алгоритм Лі

Алгоритм хвильового трасування (хвильовий алгоритм, алгоритм Лі) [9] - алгоритм пошуку шляху, алгоритм пошуку найкоротшого шляху на планарному графі. Належить до алгоритмів, заснованих на методах пошуку в ширину.

В основному використовується при комп'ютерному трасуванні (розводці) друкованих плат, з'єднувальних провідників на поверхні мікросхем. Інше застосування хвильового алгоритму - пошук найкоротшої відстані на карті в комп’ютерних стратегічних іграх.

Хвильовий алгоритм в контексті пошуку шляху в лабіринті був запропонований Е. Ф. Муром. Лі незалежно відкрив цей же алгоритм при формалізації алгоритмів трасування друкованих плат в 1961 році.

Алгоритм працює на дискретному робочому полі (ДРП) , що представляє собою обмежену замкнутою лінією фігуру, не обов'язково прямокутну, розбиту на прямокутні осередки, в окремому випадку - квадратні. Безліч всіх осередків ДРП розбивається на підмножини: «прохідні» (вільні), тобто при пошуку шляху їх можна проходити, «непрохідні» (перешкоди),

шлях через цей осередок заборонений, стартовий осередок (джерело) і фінішний (приймач). Призначення стартового і фінішного осередків умовно, достатньо вказівки пари осередків, між якими потрібно знайти найкоротший шлях.

Алгоритм призначений для пошуку найкоротшого шляху від стартового осередку до кінцевого осередку, якщо це можливо, або, за відсутності шляху видати повідомлення про непрохідність.

Робота алгоритму включає в себе три етапи: ініціалізацію, поширення хвилі і відновлення шляху.

Під час ініціалізації будується образ множини осередків оброблюваного поля, кожному осередку приписуються атрибути прохідності / непрохідності, запам'ятовуються стартовий і фінішний осередки.

Далі, від стартового осередку породжується крок до сусіднього осередку, при цьому перевіряється, чи прохідний він, і чи не належить раніше відміченому у дорозі осередку.

Сусідні осередки прийнято класифікувати двояко: в сенсі околиці Мура і околиці фон Неймана, які відрізняються тим, що в околиці фон Неймана сусідніми осередками вважаються тільки 4 осередки по вертикалі і горизонталі, в околиці Мура – усі 8 осередків, включаючи діагональні.

При виконанні умов прохідності і неналежності її до раніше позначених в дорозі осередків, в атрибут осередку записується число, рівне кількості кроків від стартового осередку, від стартового осередку на першому кроці це буде 1. Кожен осередок, мічений числом кроків від стартового осередку стає стартовим і з нього породжуються чергові кроки в сусідні осередки. Очевидно, що при такому переборі буде знайдено шлях від початкового осередку до кінцевого, або черговий крок з будь-якого породженого в дорозі осередку буде неможливий.

Відновлення найкоротшого шляху відбувається в зворотному напрямку: при виборі осередку від фінішного осередку до стартового на кожному кроці

вибирається осередок, що має атрибут відстані від стартового на одиницю менше поточного осередку. Очевидно, що таким чином знаходиться найкоротший шлях між парою заданих осередків [13].

1.12 Алгоритм Contraction hierarchies

Алгоритм призначений для швидкого пошуку маршруту між двома вершинами графа. Є алгоритмом переобробки графа. Тобто завдяки цьому алгоритму ми отримаємо оптимізований граф для пошуку маршруту. Після чого можна буде використати будь-який з відомих алгоритмів пошуку маршруту.

Суть алгоритму полягає в наступному: на кожному кроці ми обираємо одну вершину і для всіх сусідніх вершин додаємо ребра скорочення, де це можливо. На рис. 1.4 зображено приклад додавання ребра скорочення.

Як можемо побачити, найкоротший маршрут від A до E лежить через вершину C. Тому між ребрами A і E додається ребро скорочення AE. Аналогічно з вершинами A і B. В подальшому при пошуку маршруту вершина C ігноруватиметься.

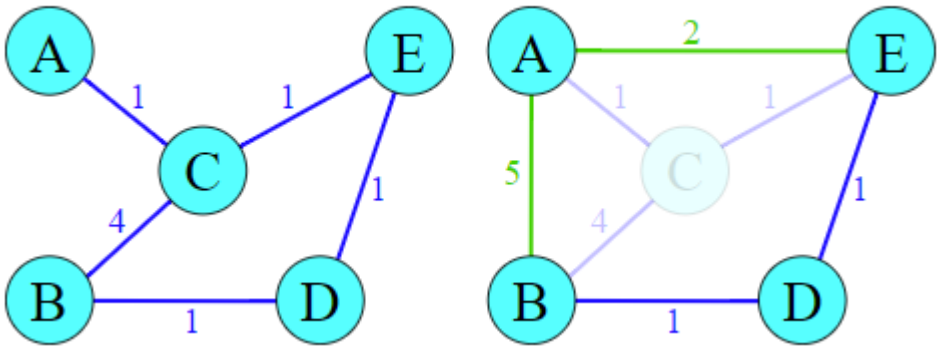


Рисунок 1.4 - Додавання ребер скорочення

Принципове значення для тривалості роботи алгоритму має порядок розгляду вершин. Результат від цього не буде залежати, але в залежності від вибору порядку обробки тривалість роботи може відрізнятися в рази.

Передобробка може займати значний час. Але для графа реальних доріг, який може включати мільйони вершин і ребер вона дає великий виграш у швидкості пошуку маршрутів [15].

Висновки за розділом 1

У розділі були описані основні задачі теорії графів пов'язані з пошуком шляху. Спочатку була розглянута задача пошуку найкоротшого шляху графі, наведена формальна постановка. Після були описані алгоритми, які найчастіше використовуються при вирішенні завдань пошуку найкоротшого шляху в графі. Зокрема: алгоритм Дейкстри, алгоритм Белмана - Форда, алгоритм A*, алгоритм Флойда-Уоршела, алгоритм Джонсона і алгоритм Лі. Починаючи від винайдення алгоритму Дейкстри в 1959 року і до сьогодні було винайдено багато алгоритмів на графах, що використовуються в багатьої сферах життя. Очевидно, що сучасні алгоритми пошуку шляхів у графах, що використовуються навігаторами є добре оптимізованими і винайти щось краще вкрай важко. Але базуючись на вже відомих алгоритмах можна будувати нові інтелектуальні системи, що стануть у нагоді багатьом користувачам.

РОЗДІЛ 2. МУЛЬТИАГЕНТНИЙ ПІДХІД ДО РОЗВЯЗАННЯ СКЛАДНИХ ПРОБЛЕМ ТА ЗАПРОПОНОВАНА МОДЕЛЬ

2.1 Агентний підхід

Інтелектуальні мультиагентні системи - один з нових перспективних напрямків штучного інтелекту, який сформувався на основі результатів досліджень в області розподілених комп'ютерних систем, мережевих технологій вирішення проблем і паралельних обчислень. У мультиагентних технологіях закладений принцип автономності окремих частин програми, спільно функціонуючих в розподіленій системі, де одночасно відбувається безліч взаємопов'язаних процесів. Такі програми називаються агентами [16].

Прикладами завдань, що вирішуються за допомогою МАС, є:

- управління інформаційними потоками і мережами;
- управління повітряним рухом;
- пошук інформації в мережі Інтернет;
- електронна комерція, навчання;
- колективне прийняття багатокритеріальних управлінських рішень та інші.

Агент - автономний штучний об'єкт, зазвичай комп'ютерна програма, що володіє активною мотивованою поведінкою здатна до взаємодії з іншими об'єктами в динамічних віртуальних середовищах. Кожен агент може приймати повідомлення, інтерпретувати їх зміст і формувати нові повідомлення, які або передаються в загальну базу, або направляються іншим агентам.

Інтелектуальним агентам притаманні такі основні властивості:

- автономність - здатність функціонувати без втручання з боку свого власника і здійснювати контроль власних дій і внутрішнього стану;
- активність - здатність до організації і реалізації дій;

- товариськість - взаємодія і комунікація з іншими агентами;
- реактивність - адекватне сприйняття стану середовища і реакція на його зміну;
- цілеспрямованість - наявність власних джерел мотивації;
- наявність базових знань про себе, про інших агентів і про навколишнє середовище;
- переконання - змінна частина базових знань, мінливих в часі;
- бажання - прагнення до певних станів;
- наміри - дії, які плануються агентом для виконання своїх зобов'язань та/або бажань;
- зобов'язання - завдання, які виконує один агент на прохання і / або дорученням інших агентів.

2.1.1 Класифікація агентів

Для класифікації агентних програм використовуються дві основні ознаки:

- ступінь розвитку внутрішнього уявлення про навколишній світ;
- спосіб прийняття рішення.

Найпростішим видом агента є простий рефлексний агент – рис. 2.1. Подібні агенти вибирають дії на основі потокового сприйняття стану середовища, ігноруючи всю решту історію сприйняття. Прості рефлексивні агенти надзвичайно прості, але мають обмежений інтелект.

На рис. 2.2 приведена структура агента, що діє з врахуванням внутрішнього стану. В умовах часткової спостережливості необхідно, щоб агент відстежував зміни середовища. Це означає, що агент повинен володіти

безліччю внутрішніх станів, зміна яких залежить від історії сприйняття. Поточне сприйняття комбінується з колишнім внутрішнім станом, в результаті відбуваються дії і відбувається зміна внутрішнього стану.

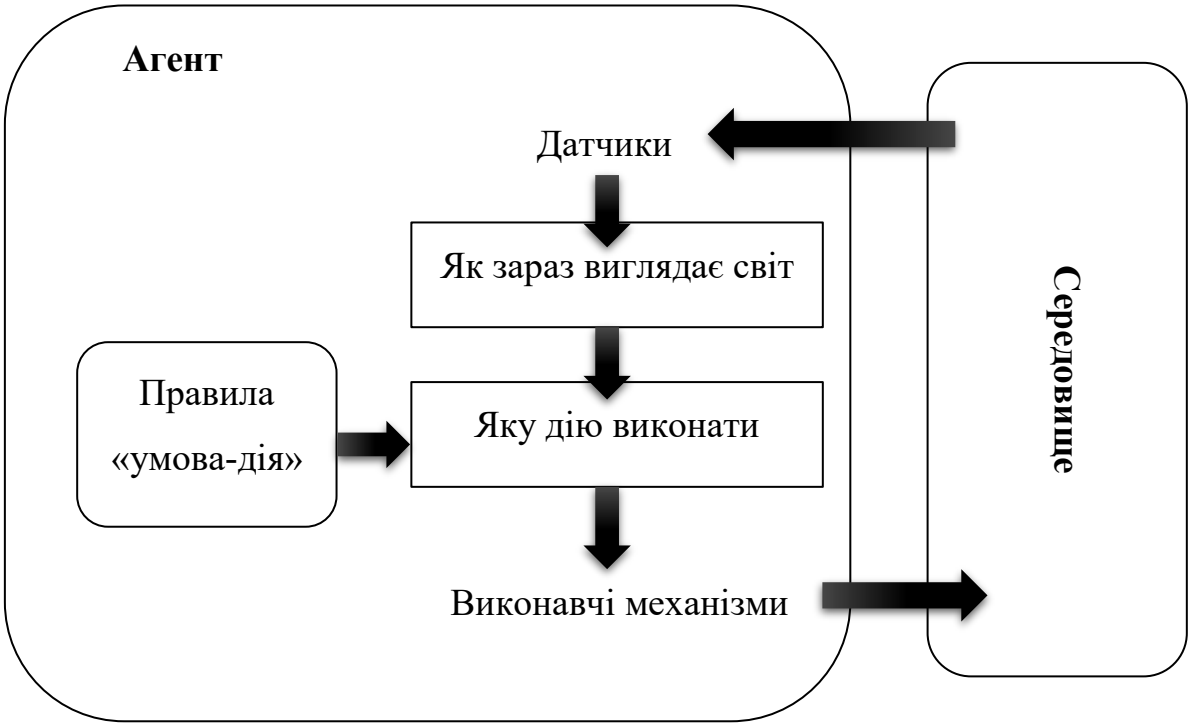


Рисунок 2.1 - Структура простого рефлексивного агенту

Знань про поточний стан середовища не завжди достатньо для прийняття рішення. Тоді агенту потрібно не тільки опис поточного стану, а й інформація про цілі, яка описує бажані ситуації. Структура агенту, що діє на основі цілей зображена на рис. 2.3. Він стежить за станом середовища, а також за кількістю цілей, яких він намагається досягти, і вибирає дію, спрямовану на досягнення цих цілей.

Часто бувають ситуації, коли для прийняття рішення недостатньо інформації тільки про цілі. По-перше, якщо є конфліктні цілі, такі, що можуть бути досягнуті тільки деякі з них (наприклад, швидкість, або безпека). По-друге, якщо є кілька цілей, до яких може прагнути агент, але кожна з них може бути досягнута з деякою ймовірністю успіху. В цьому випадку в програму

агента вводиться функція корисності яка ставить у відповідність станам агента дійсне число, що означає корисність даного стану. Агент вибирає дію, яка веде до найкращої очікуваної корисності.

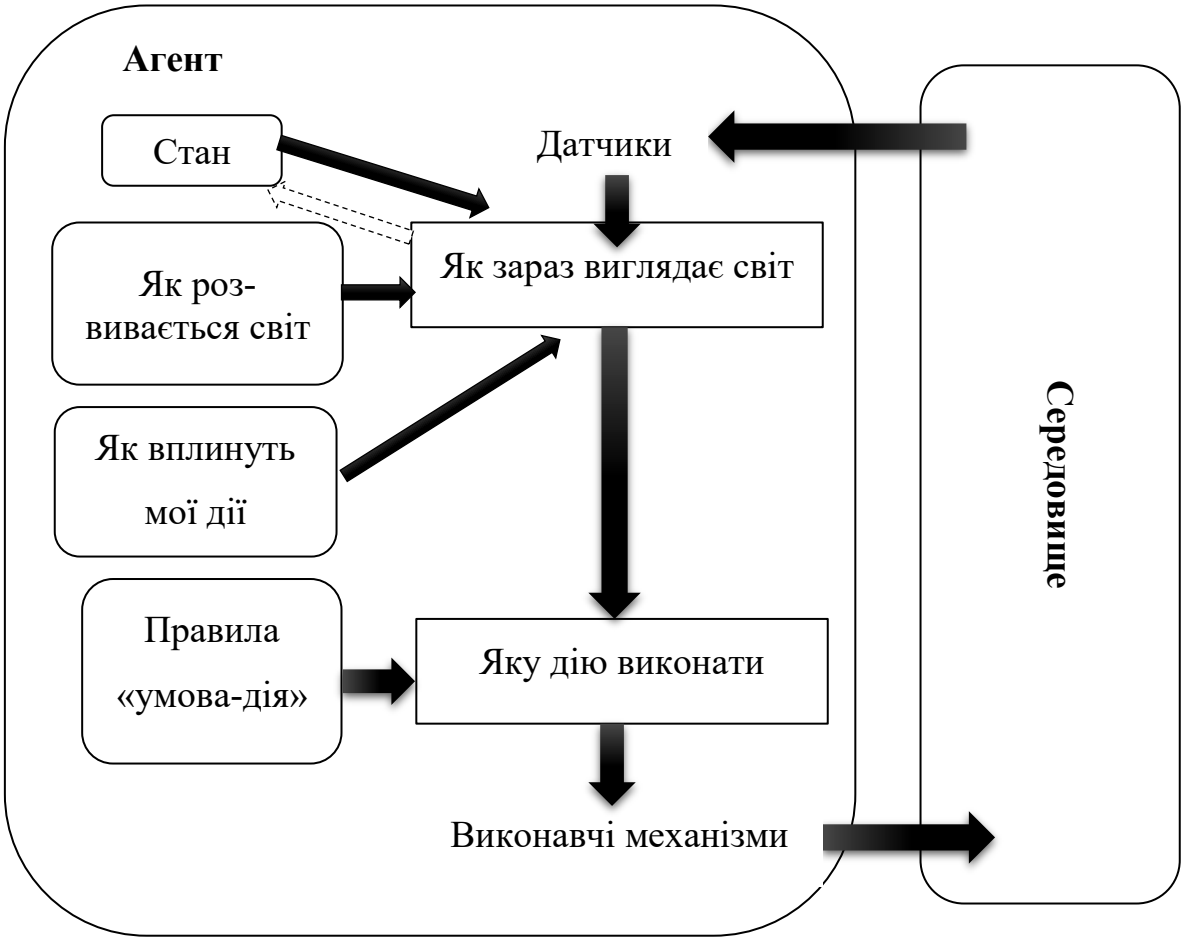


Рисунок 2.2 - Агент, що діє з врахуванням внутрішнього стану

В особливий клас виділяють агентів, що навчаються. Навчання має важливу перевагу: воно дозволяє агенту функціонувати в спочатку невідомих йому варіантах середовища і ставати більш компетентним у порівнянні з тим, що могли б дозволити тільки його початкові знання.

У структурі агенту, що навчаються виділяють чотири концептуальних компоненти. Продуктивним компонентом може бути будь-яка з розглянутих раніше структур агентних програм. Навчальний компонент використовує

інформацію зворотного зв'язку від критика з оцінкою того, як діє агент, і визначає, яким чином має бути модифікований продуктивний компонент для того, щоб він успішніше діяв в майбутньому.

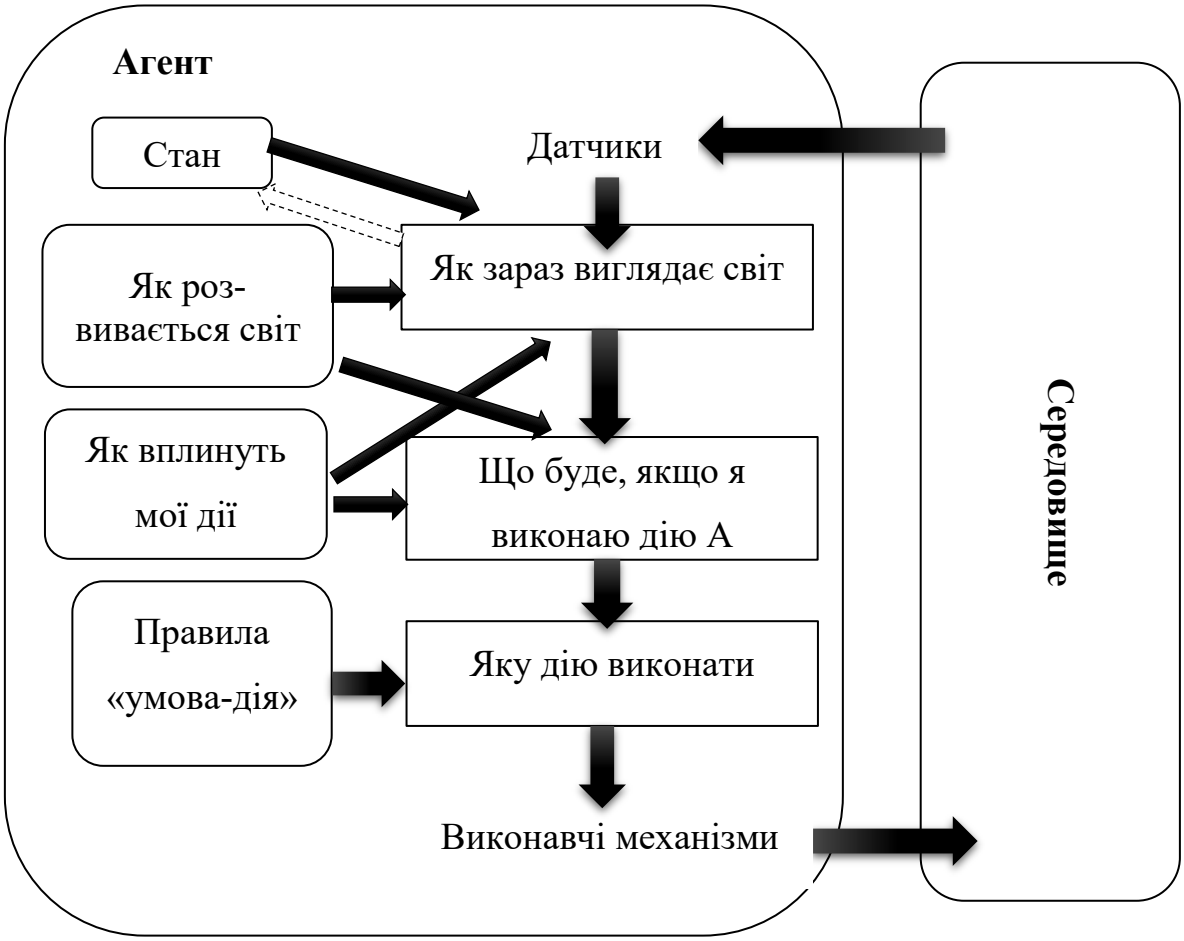


Рисунок 2.3 - Агент, що діє на основі цілей

Критик повідомляє навчальному компоненту, наскільки добре діє агент з урахуванням постійного стандарту продуктивності, оскільки самі результати сприйняття не дають ніяких вказівок на те, чи успішно діє агент. Цей стандарт слід розглядати як повністю зовнішній по відношенню до агента, оскільки агент не повинен мати можливості його модифікувати. Наприклад, шахова програма може отримати результати сприйняття, що вказують на те, що вона поставила мат своєму противнику, але їй потрібно стандарт продуктивності,

який дозволив би визначити, що це - хороший результат, так як самі дані сприйняття нічого про це не говорять.

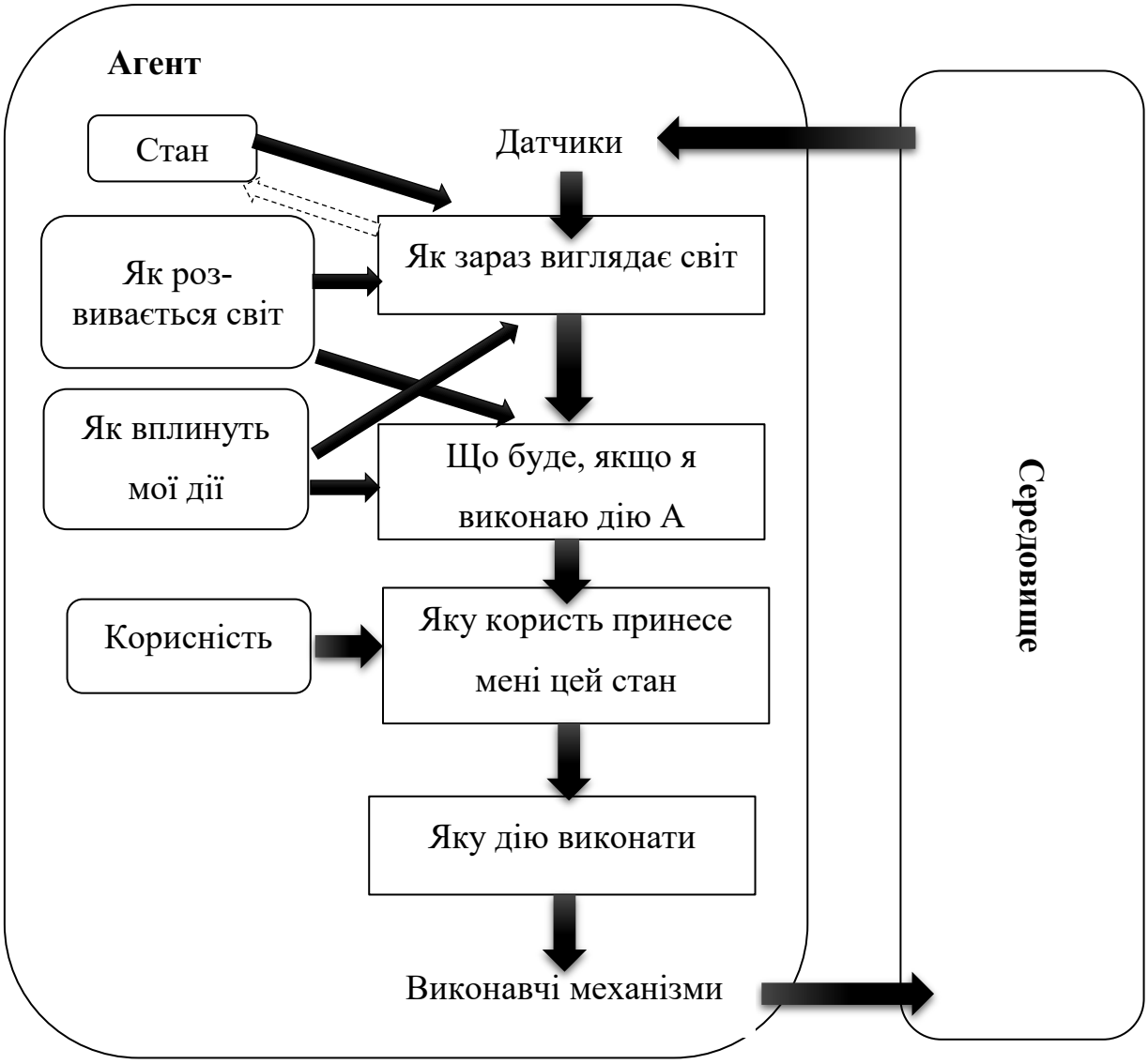


Рисунок 2.4 - Структура агенту заснованого
на моделі і корисності

Останнім компонентом агенту, о навчається є генератор проблем. Його завдання полягає в тому, щоб пропонувати дії, які повинні привести до отримання нового інформативного досвіду. Справа в тому, що якщо продуктивний компонент наданий самому собі, то він буде продовжувати

виконувати дії, які є найкращими з точки зору того, що він знає. Але якщо агент готовий до того, щоб трохи поекспериментувати і в короткостроковій перспективі виконувати дії, які, можливо, виявляться не зовсім оптимальними, то він може виявити кращі дії в майбутньому [17].

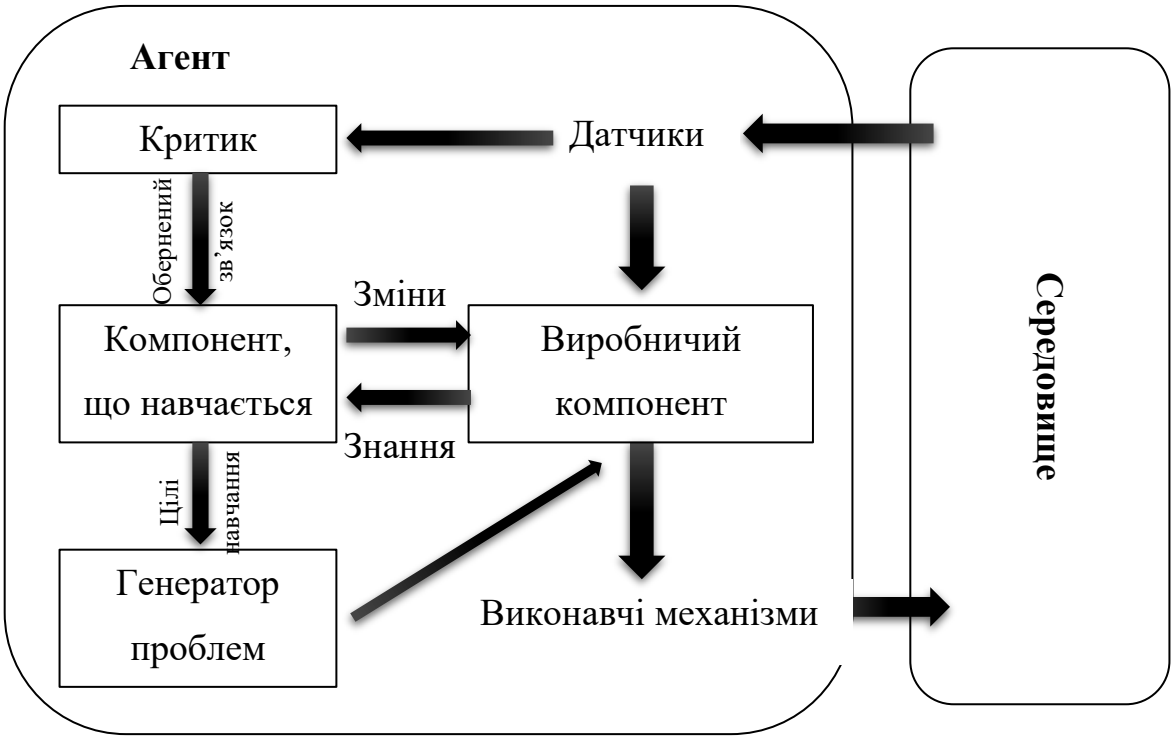


Рисунок 2.5 - Структура агенту, що навчається

Взаємодія між агентами - головна риса МАС, що відрізняє їх від інших інтелектуальних систем. Головними характеристиками будь-якої взаємодії є спрямованість, вибірковість, інтенсивність і динамічність. В контексті МАС ці поняття можна інтерпретувати в такий спосіб:

- спрямованість - позитивна чи негативна; кооперація або конкуренція; співробітництво або конфронтація; координація або субординація;
- вибірковість - взаємодія відбувається між агентами, які будь-яким чином відповідають один одному і поставленому завданню. При цьому агенти можуть бути пов'язані в одному відношенні і незалежні в іншому;

- інтенсивність - взаємодія між агентами не зводиться до наявності або відсутності, а характеризується певною силою;
- динамічність - наявність, сила і спрямованість взаємодій можуть змінюватися з плином часу.

До базових видів взаємодії між агентами відносяться:

- кооперація (співробітництво);
- конкуренція (конфронтація, конфлікт);
- компроміс (врахування інтересів інших агентів);
- конформізм (відмова від своїх інтересів на користь інших);
- ухилення від взаємодії.

Взаємодія агентів обумовлено рядом причин, найважливішими серед яких є такі.

Сумісність цілей (загальна мета). Ця причина зазвичай породжує взаємодія по типу кооперації або співпраці. При цьому слід з'ясувати, чи не веде взаємодія до зниження життєздатності окремих агентів. Несумісність цілей або переконань зазвичай породжує конфлікти, позитивна роль яких полягає в стимулюванні процесів розвитку. Відома модель хижак-жертва є прикладом одночасної взаємодії за двома типами кооперація-конфронтація.

Загальні ресурси. Ресурсами будемо називати будь-які засоби, що використовуються для досягнення агентами своїх цілей. Обмеженість ресурсів, які використовуються багатьма агентами, зазвичай породжує конфлікти. Одним з найбільш простих і ефективних способів вирішення подібних конфліктів є право сильного: сильний агент відбирає ресурси у слабких. Більш тонкі способи вирішення конфліктів забезпечують переговори, спрямовані на досягнення компромісів, в яких враховуються інтереси всіх агентів. Завдання розподілу часток ринку, витрат і прибутків спільних підприємств можна розглядати як приклади взаємодії, обумовленого загальними ресурсами.

Необхідність залучення відсутнього досвіду. Кожен агент має обмежен1 набір знань, необхідних йому для реалізації власних і спільних цілей. У зв'язку з цим йому доводиться взаємодіяти з іншими агентами. При цьому можливі різні ситуації:

- агент здатний виконати завдання самостійно;
- агент може обійтися без сторонньої допомоги, але кооперація дозволить вирішити задачу більш ефективним способом;
- агент не здатний вирішити завдання поодинці;
- залежно від ситуації агенти вибирають тип взаємодії і можуть проявляти різну ступінь зацікавленості в співробітництві.

Взаємні зобов'язання. Зобов'язання є одним з інструментів, що дозволяють упорядкувати хаотичні взаємодії агентів. Вони дозволяють передбачити поведінку інших агентів, прогнозувати майбутнє і планувати власні дії. Можна виділити наступні групи зобов'язань:

- зобов'язання перед іншими агентами;
- зобов'язання агента перед групою;
- зобов'язання групи перед агентом;
- зобов'язання агента перед самим собою.

Формальне представлення цілей, зобов'язань, бажань і намірів, а також всіх інших характеристик становить основу ментальної моделі інтелектуального агента, яка забезпечує його мотивовану поведінку в автономному режимі.

Перераховані причини в різних поєднаннях можуть призводити до різних форм взаємодії між агентами, наприклад:

Проста співпраця, яка передбачає інтеграцію досвіду окремих агентів (розподіл завдань, обмін знаннями). Без спеціальних заходів по координації їх дій;

Координована співпраця, коли агенти змушені узгоджувати свої дії (іноді залучаючи спеціального агенту-координатора) для того, щоб ефективно використовувати ресурси і власний досвід;

Непродуктивна співпраця, коли агенти спільно використовують ресурси або вирішують загальну проблему, не обмінюючись досвідом і заважаючи один одному (як лебідь, рак і щука в байці І.А. Крилова) [18].

2.1.2 Координація поведінки агентів в мультиагентній системі

В процесі моделювання колективної роботи агентів виникає безліч проблем:

- розпізнавання необхідності кооперації;
- вибір відповідних партнерів;
- можливість врахування інтересів партнерів;
- організація переговорів про спільні дії;
- формування планів спільних дій;
- синхронізація спільних дій;
- декомпозиція завдань і поділ обов'язків;
- виявлення конфліктуючих цілей;
- конкуренція за спільні ресурси;
- формування правил поведінки в колективі;
- навчання поведінці в колективі.

Особливістю колективної поведінки агентів є те, що їх взаємодія в процесі розв'язання окремих завдань (або однієї загальної) породжує нову якість вирішення цих завдань. При цьому в моделях координації поведінки агентів вживаються такі основні ідеї:

- відмова від пошуку найкращого рішення на користь «хорошого», що призводить до переходу від процедури строгої оптимізації до пошуку прийнятного компромісу, що реалізує той чи інший принцип координації;
- використання самоорганізації як стійкого механізму формування колективної поведінки;
- застосування рандомізації (випадково-ймовірнісного способу вибору рішень) в механізмах координації для вирішення конфліктів;
- реалізація рефлексивного управління, сутність якого полягає в тому, щоб змусити суб'єкта усвідомлено підкорятися впливу ззовні. Тобто сформувати у нього такі бажання і наміри, які збігаються з вимогами оточення.

Найбільш відомими моделями координації поведінки агентів є: теоретико-ігрові моделі, моделі колективної поведінки автоматів, моделі планування колективної поведінки, моделі на основі BDI-архітектур (Belief - Desire - Intention), моделі координації поведінки на основі конкуренції.

Теоретико-ігрові моделі. Предметом теорії ігор є завдання вибору рішень в умовах невизначеності і конфлікту. Наявність конфлікту передбачає існування як мінімум двох учасників, яких називають гравцями. Множина рішень, можливих для вибору кожним гравцем, називається стратегією. Точками рівноваги гри (оптимальними рішеннями) називають такі стани, коли жодному з гравців невигідно міняти свою позицію. Поняття рівноваги виявилося дуже корисним в теорії МАС, оскільки механізм пошуку положень рівноваги може використовуватися як засіб самоорганізації колективної поведінки агентів. Наслідком подібної інтерпретації є підхід, в якому необхідні атрибути колективної поведінки агентів забезпечуються шляхом конструювання правил гри. Крім того, на основі розвитку теорії ігор в області МАС робляться спроби побудови ефективних, стійких, повністю розподілених протоколів переговорів, спрямованих на координацію колективної поведінки агентів.

Моделі колективної поведінки автоматів**.** Вони засновані на ідеях рандомізації, самоорганізації і повної розподіленості. Моделі цього типу підходять для побудови протоколів переговорів в задачах, які характеризуються великою кількістю дуже простих взаємодій з невідомими характеристиками.

Моделі планування колективної поведінки. Планування може бути централізованим, частково централізованим або розподіленим. В останньому випадку агенти самі приймають рішення про вибір своїх дій в процесі координації часткових планів, в зв'язку з чим виникають питання про раціональної децентралізації, про можливість зміни цілей при виникненні конфліктів, а також проблеми обчислювальної складності.

Моделі на основі BDI-архітектури. У моделях цього класу застосовуються аксіоматичні методи теорії ігор і логічної парадигми штучного інтелекту. Акцент робиться на описі змістовних понять, таких, як переконання (belief), бажання (desire) і наміри (intention). Завдання координації поведінки агентів вирішується шляхом узгодження результатів логічного висновку в базах знань окремих агентів, отриманих для поточного стану зовнішнього середовища, в якому діють агенти. Логічний висновок здійснюється безпосередньо в процесі функціонування агентів, що призводить до високої складності моделей, обчислювальних труднощів і до проблем, пов'язаних з аксіоматичним описом нетривіальних ситуацій, наприклад, коли перед агентом постає вибір між рішенням власної завдання і виконанням зобов'язань по відношенню до партнерів.

Моделі на основі конкуренції. В моделях даного класу використовується поняття аукціон як механізм координації поведінки агентів. Використання механізму аукціону засноване на припущенні про можливість явної передачі «корисності» від одного агента до іншого або до агента-аукціонера, причому ця корисність зазвичай має сенс грошей.

Аукціони прийнято розділяти на відкриті і закриті. У першому випадку запропоновані ціни оголошуються всім учасникам. У закритому аукціоні про пропоновані цінах знає тільки аукціонер. Відкриті аукціони відрізняються за способом проведення. У так званих англійських аукціонах зазвичай задається стартова ціна, яка може збільшуватися учасниками в ході торгів. Перемагає той, хто дасть максимальну ціну. Голландський аукціон починається з верхньої ціни, яка поступово знижується. Переможцем вважається той, хто дав найбільшу поточну ціну. Закриті аукціони поділяють на аукціони першої та другої ціни. В аукціонах першої ціни перемагає той, хто запропонував найвищу ціну, відому тільки аукціонерам. В аукціонах другий ціни переможець визначається таким же способом, але платить за товар не свою ціну, а другу за величиною. Сам по собі механізм аукціону не зачіпає способів прийняття рішень учасниками. Рішення можуть прийматися на основі деякої моделі міркувань, яка може використовувати різні типи знань, доступних агентам, і різноманітні способи їх обробки. Аукціон завжди повинен закінчуватися. Для цього в стратегії його проведення повинні бути закладені кошти для вирішення можливих конфліктів (наприклад, при наявності декількох переможців). Одним з найпростіших способів вирішення конфліктів є рандомізація, коли застосовується випадковий механізм вибору [5].

2.1.3 Приклади мультиагентних систем

Розглянемо практичні приклади організації взаємодії в мультиагентних системах з використанням різних механізмів координації поведінки.

Електронний магазин. Розглянемо типову задачу електронної комерції, в якій беруть участь агенти-продавці і агенти-покупці. Торгівля здійснюється в електронному магазині, який являє собою програму, розміщену на сервері. Її

основним призначенням є організація взаємодії агентів, інтереси яких збігаються. Агенти діють за дорученням своїх персональних користувачів. При цьому агенти-продавці прагнуть продати свій товар за максимально можливою ціною, а агенти-покупці прагнуть купити потрібний товар за мінімальною ціною. Обидва види агентів діють автономно і не мають цілей кооперації. Електронний магазин реєструє появу і зникнення агентів і організовує контакти між ними, роблячи їх «видимими» один для одного.

Поведінка агента-продавця характеризується наступними параметрами:

- бажана дата, до настання якої необхідно продати товар;
- бажана ціна, по якій користувач хоче продати товар;
- найнижча допустима ціна, нижче якої товар не продається;
- функція зниження ціни в часі (лінійна, квадратична і ін.);
- опис товару, що продається.

Агент-покупець має «симетричні» параметри:

- останній термін покупки товару;
- бажана ціна покупки;
- найвища прийнятна ціна;
- функція зростання ціни в часі;
- опис товару, що купується.

Торги ведуться за схемою закритого аукціону першої ціни. Поведінка агентів описується простою моделлю, в якій не використовуються знання і міркування. Агент-продавець, отримавши від електронного магазину інформацію про потенційних покупців свого товару, послідовно опитує їх усіх з метою прийняти рішення про можливість здійснення угоди. Угода укладається з першим агентом-покупцем, який готовий дати за товар запитувану ціну. Продавець не може вдруге вступити в контакт з будь-яким покупцем до тих пір, поки не опитає всіх потенційних покупців. При кожному контакті агент-продавець веде переговори, пропонуючи початкову ціну або

знижуючи її. Агент-покупець діє аналогічним чином, відшукуючи продавців потрібного товару і пропонуючи їм свою ціну покупки, яку він може збільшити в процесі переговорів. Будь-яка угода завершується тільки в разі її схвалення користувачем агента.

Дана схема переговорів являє собою найпростіший випадок взаємодії автономних агентів, що діють реактивно. Проте підсумкова поведінка системи цілком адекватна реальності.

Віртуальне підприємство. Створення віртуальних підприємств є одним із сучасних напрямків бізнесу, яке в значній мірі стимулюється швидким зростанням інформаційних ресурсів і послуг, що надаються в мережі Інтернет. Крім того, поява віртуальних підприємств сприяє скорочення часу життєвого циклу створення виробів і підвищення рівня їх складності, оскільки при цьому виникає необхідність оперативного об'єднання виробничих, технологічних та інтелектуальних ресурсів. Ще одна важлива причина - посилення конкуренції на товарних ринках, що стимулює об'єднання підприємств з метою виживання.

Віртуальне підприємство можна визначити як кооперацію юридично незалежних підприємств, організацій та індивідуумів, які виробляють продукцію або послуги в загальному бізнес-процесі. У зовнішньому світі віртуальне підприємство виступає як єдина організація, в якій використовуються методи управління і адміністрування, засновані на застосуванні інформаційних і телекомунікаційних технологій. Метою створення віртуального підприємства є об'єднання виробничих, технологічних, інтелектуальних та інвестиційних ресурсів для просування на ринок нових товарів і послуг.

Оскільки кожне реальне підприємство в рамках віртуального виконує тільки частину робіт із загального технологічного ланцюжка, то при його створенні вирішуються два головні завдання. Перша - це декомпозиція загального бізнес-процесу на компоненти (підпроцеси). Друге завдання полягає у виборі оптимального складу реальних підприємств-партнерів, які

будуть здійснювати технологічний процес. Перше завдання вирішується із застосуванням методів системного аналізу, а для вирішення другого можуть застосовуватися засоби мультиагентних технологій.

Завдання оптимального розподілу множини робіт (підпроцесів) серед множини працівників (реальних підприємств) в дослідженні операцій формулюється як задача про призначення. Її рішення починається з формування множин підпроцесів і потенційних підприємств-учасників. Потім будуються можливі відображення з множини учасників на множину підпроцесів і робиться вибір найбільш прийнятного відображення, яке відповідає конкретним призначенням підприємств на бізнес-процеси. Для цього можна використовувати механізм аукціону. На рис. 2.6 приведена схема аукціону по створенню віртуального підприємства, в якому виділені бізнес-процеси A, B, C, D, E і беруть участь чотири підприємства: P1, P2, P5, P4, які претендують на їх реалізацію. Кожне з підприємств представлено інтелектуальним агентом, при цьому одне з них (Px) виступає в ролі ініціатора (аукціонера).

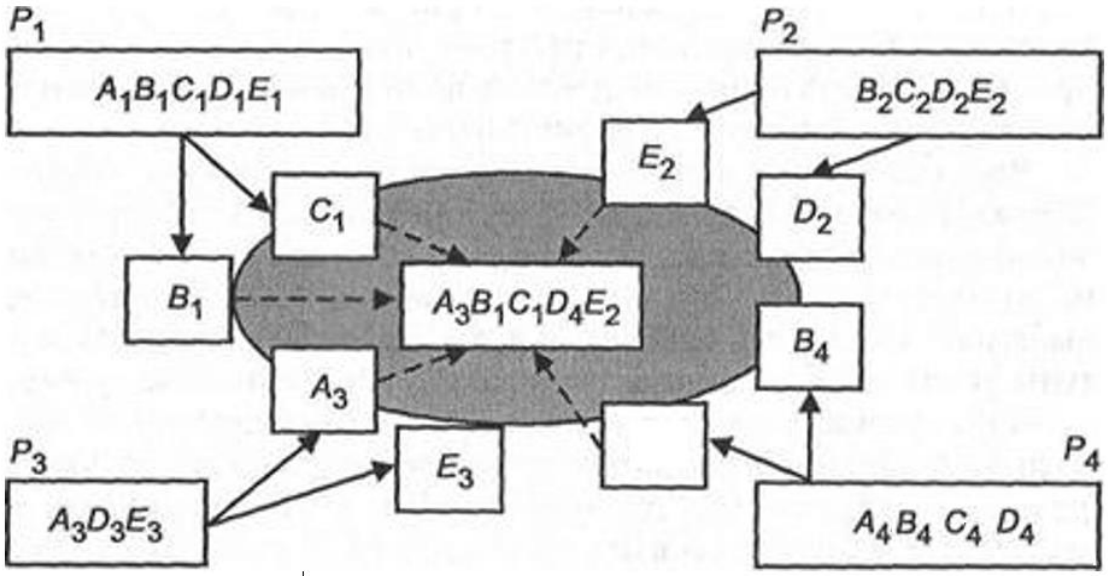


Рисунок 2.6 - Схема створення віртуального підприємства

Перед початком аукціону аукціонер (менеджер) формує базу даних і базу знань про учасників аукціону. Потім він виставляє на продаж окремі бізнес-процеси, інформація про яких представлена стартовою ціною і вимогами по заданому набору показників. Кожен претендент висуває свої пропозиції за параметрами, які він в змозі забезпечити, і свою ціну. Зібравши і обробивши ці пропозиції, аукціонер за допомогою деякої моделі міркування впорядковує потенційних претендентів з урахуванням власної інформації про них. Після цього він приймає рішення про вибір призначень або відкидає їх і висуває нові пропозиції.

Слід зазначити, що завдання створення віртуального підприємства можна віднести до завдань структурного синтезу складних систем, які відповідають заданим вимогам [16].

2.2 Запропонована модель

В даній роботі пропонується мультиагентний підхід до реалізації багатозадачного додатку, що надає користувачеві можливості навігації, пошуку інших користувачів на карті з використанням GPS сигналу мобільного пристрою та побудову колективного маршруту для групи користувачів згідно з їхніми побажаннями.

Далі опишемо агенти, що реалізовані, їхнє призначення і функціонал, та додаткові агенти, реалізація яких передбачається, але не імплементується в межах даної роботи, для розширення функціоналу додатку.

Агент обробки даних OSM. В програмному продукті використовується картографічна інформація, що надається відкритим сервісом OpenStreetMap. Дані, які ми можемо отримати зберігаються у вигляді XML документу. Такий формат є зручним для маніпуляцій, але не є зручним для обробки під час

пошуку маршрутів. Тому виникає необхідність обробити документ та створити дорожній граф в базі даних з яким буде зручно працювати.

Робота агенту полягатиме в завантаженні необхідної частини карти на сервер та парсинг в базу даних. На виході отримуємо заповнені таблиці з вершинами графа доріг та ребрами – дорогами, що з’єднують вершини.

Агент пошуку маршруту. Агент пошуку працює з базою даних, яку згенерував попередній агент і обирає алгоритм, що на думку агенту найкраще діх/дходить до конкретної задачі пошуку шляху.

Агент передобробки графу. Агент реалізовує алгоритм Contraction Hierarchies (CH). Він приступає до роботи перед агентом пошуку маршруту і оптимізує граф доріг – зменшує кількість ребер, замінюючи кілька ребер на одне. Його робота не є обов’язковою, але після обробки графу доріг пошук маршруту стає на багато швидшим.

Додаткові агенти. Для розширення функціоналу можуть бути реалізовані інші агенти, наприклад, агент моніторингу трафіку на дорогах, що надаватиме агенту побудови маршрутів інформацію про затори і, таким чином, вноситиме зміни в знайдений маршрут.

Висновки за розділом 2

В розділі було розглянуто загальну концепцію агентного підходу до розробки програмних продуктів. Розглянуто їх класифікацію та наведено приклади мультиагентних систем.

В результаті запропоновано модель мультиагентної системи для створення і обробки до графу доріг, роутингу, в тому числі колективного роутингу – побудови маршрутів для групи людей.

Запропонована модель передбачає можливість збільшення кількості агентів, що в неї входять для розширення функціоналу програмного продукту.

РОЗДІЛ 3. ОСОБЛИВОСТІ ВИКОРИСТАНИХ АЛГОРИТМІВ ПРОГРАМНОГО ПРОДУКТУ. АРХІТЕКТУРА СИСТЕМИ

В даному розділі запинимося на особливостях реалізації алгоритмів пошуку маршруту в конкретному випадку реалізації мультиагентної системи маршрутизації.

Зокрема детальніше розглянемо двобічні модифікації алгоритмів Дейкстри та A* та детально розглянемо етапи алгоритму Contraction hierarchies та особливості побудови архітектури додатка для реалізації даного алгоритму.

Двосторонні алгоритми відрізняються від односторонніх тим, що запускається дві процедури пошуку: з початкової і кінцевої точки. При чому кроки обох процедур виконуються почергово і у випадку алгоритмів Дейкстри і A* зупинка настає тоді коли в закриті списки обох процедур потрапляє одна і та ж вершина. Після цього шлях прямої процедури об’єднується зі оберненим шляхом зворотної процедури [17].

3.1 Contraction hierarchies

Основною ідеєю алгоритму є почергове усунення вершин з початкового графу в певному порядку. Коли вершина видаляється, найкоротший шлях між точками буде втрачено, якщо він проходить через видалену вершину. В такому випадку нам необхідно додати ребра скорочення (shortcuts) зі збереженням відстаней. Тобто відстань між будь-якими двома точками залишатиметься такою ж, як і в вихідному графі. Таким чином для кожної вершини може бути додано кілька нових ребер.

Врешті-решт ми отримаємо граф, що містить такий самий набір вершин, що і вихідний граф, всі ребра вихідного графа і додані шляхи скорочення. Це і буде результуючий граф [24].

Розглянемо алгоритм детальніше.

3.1.1 Передобробка

Розглянемо, що відбувається при обробці вершини. На рисунку 3.1 зображено приклад елементарного графа рис 3.1.

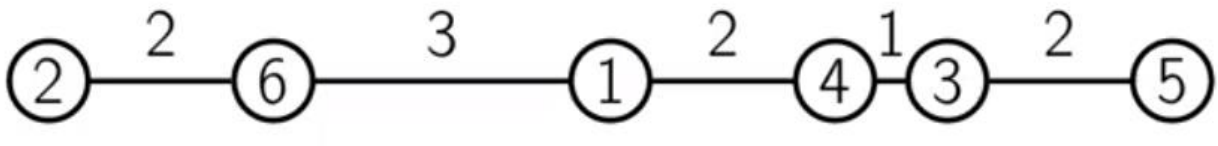


Рисунок 3.1 - Приклад графу

Числа в вершинах порядок в якому вони будуть оброблятися. Про вибір порядку обробки говоритиметься далі. Числа над ребрами вказують на довжину ребра.

При обробці вершини під номером 1 ми вважаємо, що вершина зникає з графа, а з нею і всі ребра, що в неї входять і виходять. Тобто ми втрачаємо зв'язок між вершинами 6 і 4, який є єдиним, а значить і найкоротшим. Ми повинні відновити цей зв'язок додавши ребро скорочення між ними довжиною 5. Результат обробки вершини 1 зображено на рисунку 3.2.

При обробці вершини 2 ніяких зв’язків порушено не буде, а, отже, і ребра додавати не потрібно.

Розглянемо обробку вершини 3. Як і у випадку з вершиною 1 при обробці третьої вершини буде видалено єдиний зв’язок між вершинами 4 і 5. Його теж необхідно відновити. Результат можемо побачити на рисунку 3.3

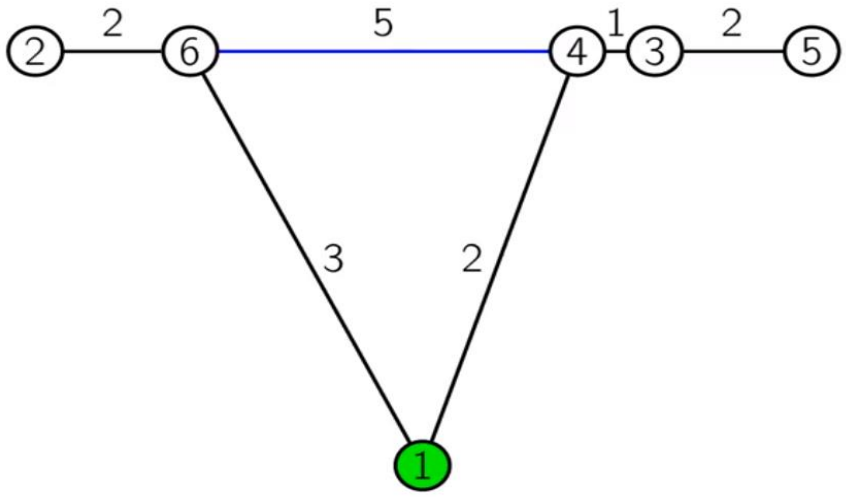


Рисунок 3.2 - Граф після обробки вершини 1

Відмітимо, що при обробці нас цікавлять лише шляхи, що з’єднують вершини, які ще не оброблялися. Так, при обробці вершини номер 4 ми не звертаємо уваги на те, що між 1 та 3 зв’язок зникне. Розглядаємо ми лише вершин 6 та 5 і додаємо між ними ребро скорочення довжини 5+3=8.

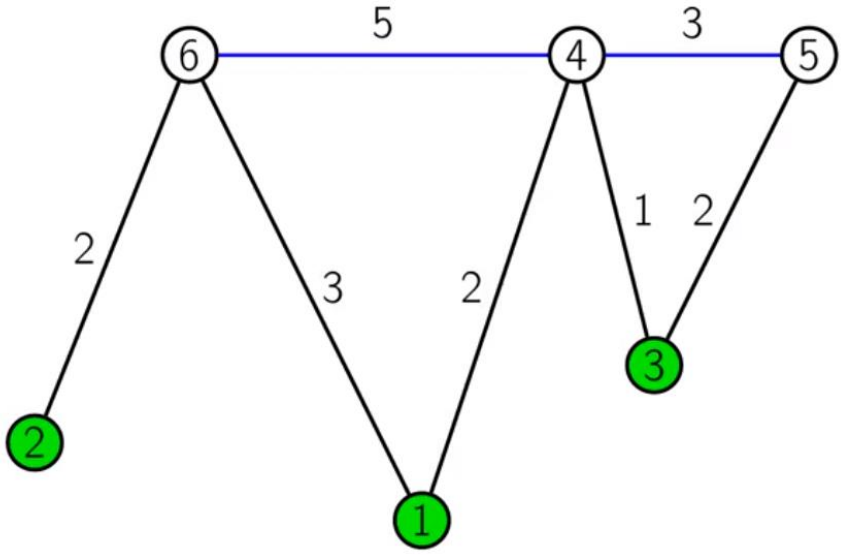


Рисунок 3.3 - Граф після обробки вершини 3

Повний результат обробки графу можемо побачити на рисунку 3.4.

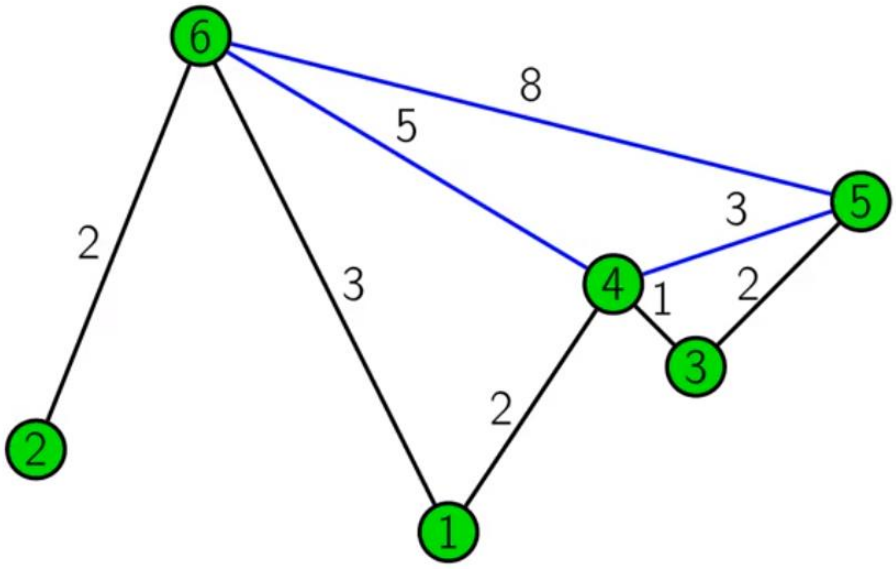


Рисунок 3.4 - Граф після повної обробки

Бачимо, що після обробки вершини 4 між 6 та 5 з’являється лише одне ребро довжини 8, насправді ж це ребро приховує в собі ланцюжок вершин 6 – 1 – 4 – 3 – 5.

Можемо бачити, що вершини на рисунках знаходяться на різній висоті. Ця висота відповідає порядку обробки – чим вище вершина тим пізніше вона оброблена. І чим вище знаходиться вершина тим вона важливіша. Тобто спочатку ми хочемо обробляти менш важливі вершини.

Важливим моментом є те, що додавати ребра скорочення потрібно далеко не завжди, а тільки в тому випадку, коли це ребро буде найкоротшим шляхом між вершинами. Тобто ми будемо додавати ребро тільки тоді, коли в графі не знайдеться жодного шляху між вершинами довжиною меншою або рівною за довжину ребра. Для того, щоб з'ясувати, існує такий шлях, чи ні проводиться процедура Witness search [19].

3.1.2 Witness search

Нехай ми обробляємо вершину v. Тоді для кожної пари ребер (u, v) і (v, w) ми хочемо перевірити, чи існує коротший шлях від u до w, такий, що $l(u, \dots, w) \leq l(u, v) + l(v, w)$. В такому випадку в додаванні ребра скорочення немає сенсу.

При пошуку такого шляху використовується алгоритм Дейкстри з невеликими модифікаціями.

По-перше, цілком очевидно, що ми шукаємо шлях між u і w, що не буде проходити через вершину v. Тому в усіх ітераціях алгоритму цю вершину ми будемо пропускати ніби її немає.

По-друге, запуск даної процедури буде дуже частою операцією тому важливою є її оптимізація.

Нас цікавить, чи існує шлях меншої довжини ніж певна відома величина. Коли в відкритий список ми будемо додавати лише ті вершини відстань до яких буде меншою від довжини ребра скорочення для якого

запускався пошук. Врешті-решт, якщо шуканого шляху не існує то в відкритому списку не залишиться жодного елемента і до ребра *w* ми так і не дійдем. Тоді можна з впевненістю вважати, що шлях скорочення є найкоротшим в графі між вершинами, що розглядаються і його необхідно додавати.

Ще одним варіантом оптимізації є обмеженні кількості ребер в шуканому шляху. Наприклад, ми можемо встановити, що якщо кількість ребер досягла п’яти, а вершини *w* ми так і не досягли то, незважаючи на те, що у відкритому списку ще є елементи процедуру ми зупинимо і вважатимемо, що коротшого шляху не існує.

Варто також відмітити, що хоч і дана процедура буде запускатися досить часто, проте пошук вестиметься між досить близькими вершинами, а, отже, триватиме він недовго [20].

3.1.3 Пошук маршруту в обробленому графі

При пошуку маршруту в обробленому графі необхідно використовувати двобічний алгоритм пошуку. В нашому випадку це буде двобічний алгоритм Дейкстри.

Як вже було сказано, при обробці ми зберігаємо порядок в якому вершини графа були оброблені. Таким чином ми отримуємо ієрархії вершин за їх значимістю. При пошуку маршруту ми будемо йти лише «вгору», тобто від вершини з меншим порядковим номером до вершини з більшим.

Знайдений найкоротший шлях спочатку підійматиметься до якоїсь вершини і потім почне опускатися до кінцевої. Обґрунтування чому таки буде завжди наведено нижче.

Прямий пошук алгоритму запускається з початкової вершини, обернений з кінцевої. На рисунку 3.5 зображено шлях, що буде знайдено між вершинами 2 та 3 в графі, що був розглянутий в попередньому підрозділі.

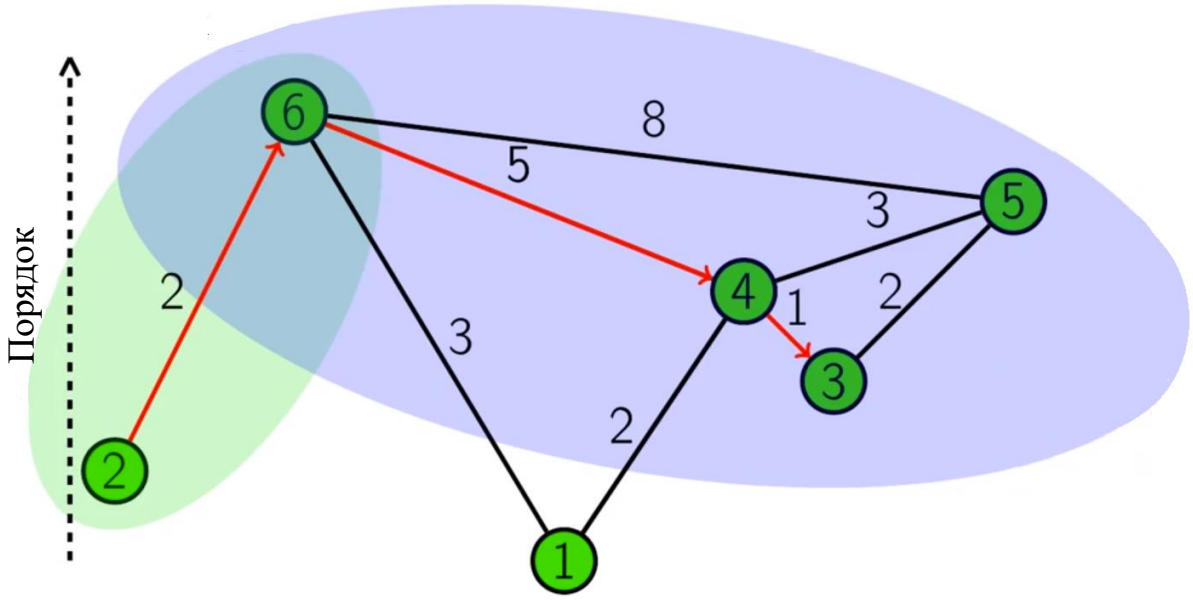


Рисунок 3.5 - Шлях від вершини 2 до вершини 3

В нашому випадку двобічний алгоритм Дейкстри має певні особливості і обмеження.

- використовуються тільки ребра, що йдуть вгору;
- на відміну від звичайного двобічного алгоритму, в нашому випадку робота не зупиняється одразу коли в закритих списках обох пошуків, прямого і зворотнього, опиняється одна і та ж вершина. Протягом роботи алгоритму, одразу ж після виникнення ситуації, коли одна вершина опиняється в обох списках рахується довжина знайденого шляху і зберігається;
- в нашому випадку алгоритм продовжуватиме роботу поки існують вершини, які ще можна обробити. Завдяки тому, що ми будемо йти лише вгору таких вершин буде не так багато, як може здатися на перший погляд;

- розглядаються тільки ті вершини, відстань до яких менша за вже знайдену відстань між шуканими точками. Таким чином ми ще зменшуємо кількість вершин, що будуть розглянуті [21].

Наведемо псевдокод алгоритму:

$estimate = +\infty$

$Fill\ dist, dist^R\ with + \infty\ for\ each\ node$

$dist[s] = 0, dist^R[t] = 0$

$proc = empty, proc^R = empty$

$while\ there\ are\ nodes\ to\ process$

$v = ExtractMin(dist)$

$if\ dist[v] \leq estimate$:

$Process(v)$

$if\ (v\ in\ proc^R\ and\ dist[v] + dist^R[v] < estimate)$

$estimate = dist[v] + dist^R[v]$

$Repeat\ symmetrically\ for\ v^R$

$return\ estimate$

3.1.4 Коректність

Спочатку дамо необхідні визначення.

Розширений граф $G^+ = (V, E^+)$ графу $G = (V, E)$ це граф, що містить ті ж вершини V, що й граф G, ребра E та ребра скорочення додані на стадії передобробки.

Відстанню $d(s, t)$ називатимемо довжину найкоротшого шляху між вершинами $s, t \in V$ графу $G = (V, E)$.

Порядком вершини $r(v)$ назвемо порядковий номер під яким ця вершина була оброблена на стадії переобробки.

Шлях $P: v_1 \to v_2 \to \cdots \to v_k$ розширеного графу G^+ називається *зростаючим*, якщо $\forall\, i < j \in N\ r(v_i) < r(v_j)$, і *спадним*, якщо $\forall\, i < j \in N\ r(v_i) > r(v_j)$.

Лема. $\forall\, s, t \in V$, відстань $d^+(s,t)$ в графі G^+ дорівнює відстані $d(s,t)$ в графі G.

Доведення. По-перше. Ребра тільки додаються до графа $G = (V, E)$. Тому $d^+(s,t) \leq d(s,t)$.

По-друге, для будь-якого доданого ребра скорочення (u, w), існує шлях $u \to v \to w$, довжини $l(u,w) = l(u,v) + l(v,w)$, отже, $d^+(s,t)$ не може бути меншою від $d(s,t)$.

Таким чином ми довели, що $d^+(s,t) = d(s,t)$.

Обґрунтування двобічного пошуку Дейкстри.

Лема. $\forall\, s, t \in V$, якщо існує найкоротший маршрут P_{st} в розширеному графі $G^+(V, E^+)$ то існує вершина v така, що $P_{st} = P_{sv} + P_{vt}$ і P_{sv} – зростаючий, а P_{vt} – спадний.

Доведення. Нехай існує найкоротший шлях між s, t P_{st}. $P_{st}: s \to u_1 \to u_2 \to \cdots \to u_k \to \cdots \to t$.

Припустимо, що існує така вершина u_k, що $r(u_{k-1}) > r(u_k) < r(u_{k+1})$ – назвемо її локальним мінімумом. Можемо побачити на рисунку 3.6.

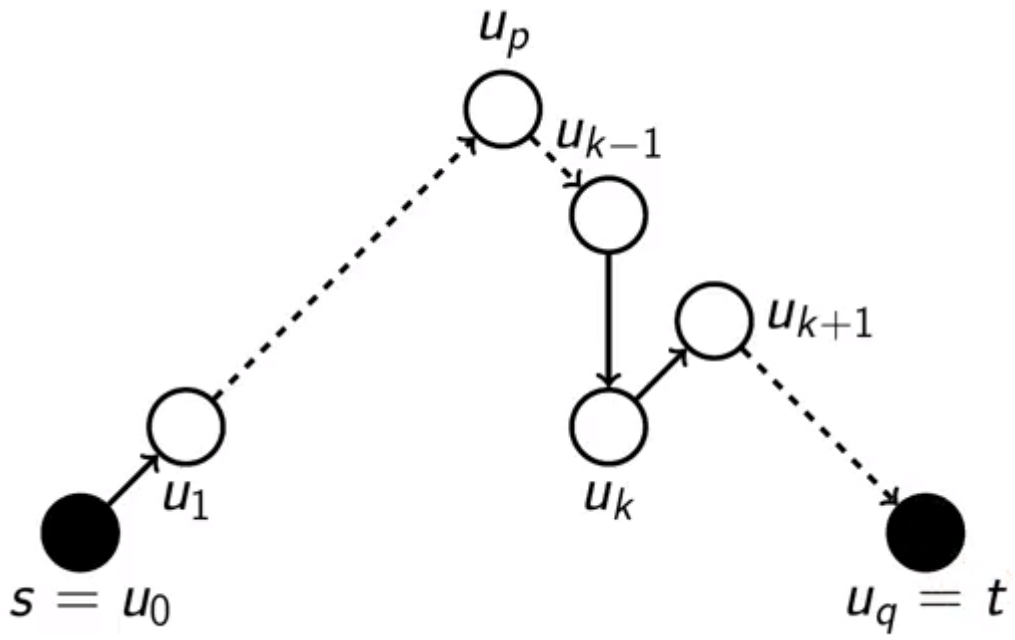


Рисунок 3.6 - $u_k, r(u_{k-1}) > r(u_k) < r(u_{k+1})$

Тоді u_k було оброблено раніше ніж u_{k-1} і u_{k+1} і було б знайдено ребро скорочення (u_{k-1}, u_{k+1}).

В такому випадку існує два варіанти.

- Ребро скорочення (u_{k-1}, u_{k+1}) не додається, оскільки існує коротший шлях між u_{k-1}, u_{k+1}. Тоді наш шлях $P_{st}: s \to \cdots u_k \to \cdots t$ не є найкоротшим. Ми дійшли до протиріччя рис. 3.7;
- Коротшого шляху між u_{k-1}, u_{k+1} не існує. Тоді додається ребро скорочення. В цьому випадку шлях P_{st} буде проходити через додане ребро скорочення і локальний мінімум u_k зникне зі знайденого маршруту рис. 3.8.

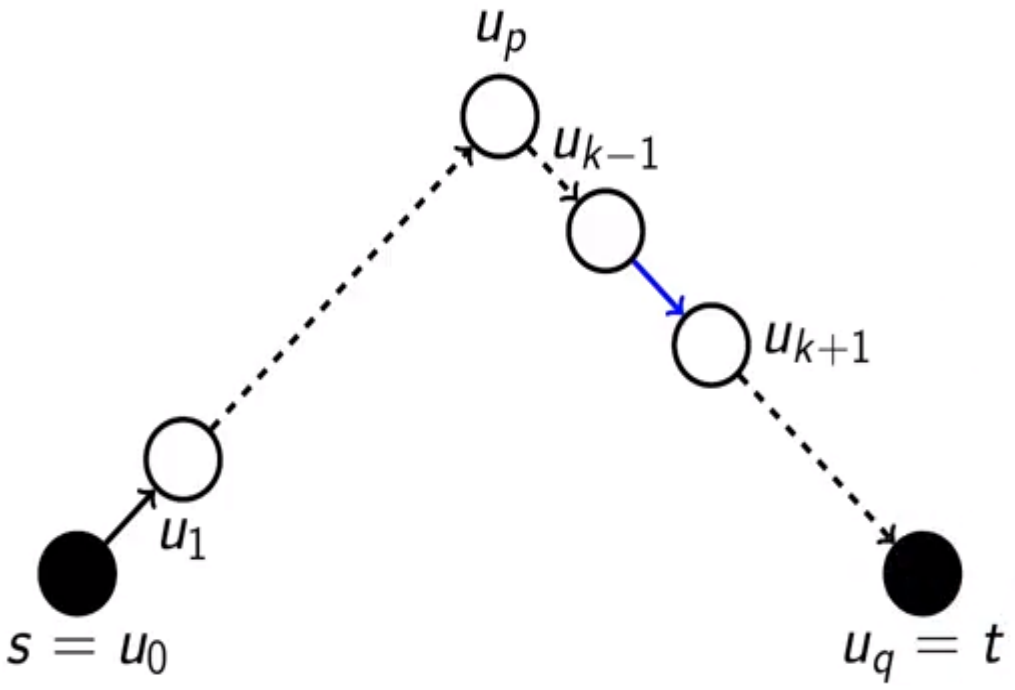


Рисунок 3.7 - Між (u_{k-1}, u_{k+1}) знайдено коротший шлях

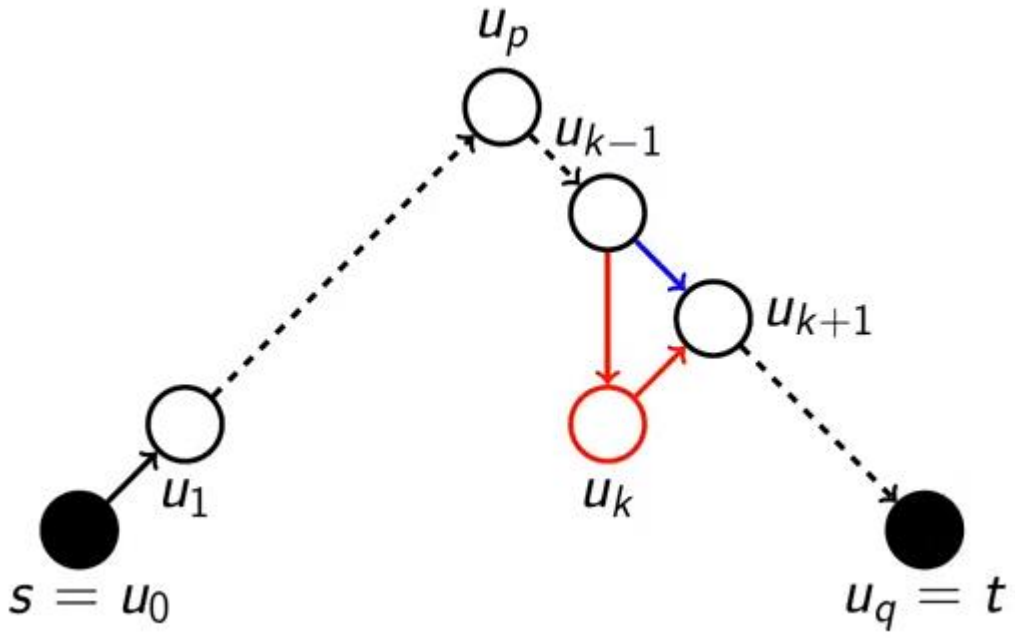


Рисунок 3.8 - Використання знайденого ребра скорочення

Отже, ми довели, що якщо існує найкоротший маршрут то він спочатку зростатиме до якоїсь вершини, а потім спадатиме до кінцевої [22].

3.1.5 Порядок обробки вершин

Незалежно від порядку обробки вершин алгоритм переобробки відпрацює правильно і буде отримано коректний результуючий граф. Але, очевидно, від порядку залежить вигляд графа, а, отже, і швидкість роботи алгоритму пошуку на ньому. Тому на рівні з реалізацією процедури пошуку і додавання ребер скорочення і двобічного алгоритму Дейкстри необхідно реалізувати правильне сортування вершин перед обробкою.

Нашою метою є виконати таке сортування за якого:

- буде додана мінімальна кількість ребер;
- важливі вершини будуть поширені по всій площині графу;
- кількість ребер в найкоротших шляхах буде мінімальною.

Сортування відбуватиметься важливістю вершини в графі. Тобто спочатку буду оброблені менш важливі вершини, потім більш важливі. В результаті сортування визначить порядок вершини в графі, що впливає на роботу алгоритму Дейкстри, оскільки, очевидно, через вершини з меншим порядком проходитиме менша кількість знайдених шляхів (див. Пошук маршруту в обробленому графі).

Для цього нам необхідно ввести формальну міру важливості вершини. В нашому випадку дана міра складається з чотирьох компонентів:

- різниця ребер;
- кількість оброблених сусідів;
- покриття ребрами скорочення;
- рівень вершини.

Різниця ребер

Метою даного критерію є мінімізація кількості ребер в розширеному графі. Розраховується за формулою:

$$ed(v) = s(v) - in(v) - out(v),$$

де *ed(v)* – різниця ребер для вершини *v*,

s(v) – кількість доданих ребер скорочення після обробки вершини *v*,

in(v) – кількість ребер, що входять в вершину *v*,

out(v) – кількість ребер, що виходять з вершини *v*.

В першу чергу необхідно обробляти вершини з малим показником різниці ребер, оскільки саме такі вершини додають найменше ребер скорочення, отже мають найменший вплив на граф загалом, оскільки найкоротших шляхів, що проходитимуть через них буде мало, або взагалі не буде.

Проблемою при розрахунку даного критерію є обрахунок компоненти *c(v)*. Ця компонента відповідає за кількість ребер скорочення, що будуть додані після обробки вершини. Це передбачає запуск процедуру пошуку ребер скорочення, яка є дороговартісною через необхідність запуску процедур witness search. Для малого графа це не складає проблеми, але для реальних графів з кількістю вершин і ребер більша 100 000 це має суттєвий вплив на тривалість роботи алгоритму. Наприклад, для графа Києва, що містить близько 160 000 вершин і близько 175 000 ребер в многопоточному режимі процедура сортування працює більше 1,5 год. В однопоточному – близько чотирьох годин. Саме компонента різниці ребер має найбільший вплив на час роботи процедури сортування.

Кількість оброблених сусідів.

Суть даного критерію – поширити важливі вершини рівномірно. Назва говорить сама за себе, значенням критерію буде кількість вже оброблених

ребер в які, або з яких можна потрапити до вершини *v*. Очевидно, спочатку для всіх вершин значення критерію буде рівним нулю.

Так як і для різниці ребер, чим менше значення має критерій тим раніше повинна бути оброблена вершина. І навпаки, чим більше сусідів вже оброблено тим пізніше повинна бути оброблена вершина, що розглядається.

Покриття ребрами скорочення.

Суть даного критерію – обробляти вершини, від яких «залежить» багато інших вершин пізніше, тобто зробити їх більш важливими. Критерій позначається як *sc(v)* для вершини *v*. І розраховуватиметься як кількість вершин *w*, таких, що при обробці *v* буде додано ребро скороченні до, або від *w*. Якщо критерій має велике значення це означає, що багато вершин «залежать» від *v*.

Так як і з попередніми критеріями, чим менше значення *cv(v)* тим раніше необхідно обробляти вершину.

Рівень вершини.

Рівень вершини *v* означає верхню межу кількості ребер в найкоротшому шляху від будь-якої вершини *s* в розширеному графі $G^+ = (V, E^+)$.

Спочатку рівень усіх вершин прирівнюється до 0. При обробці вершини *v* до сусідньої вершини *u* ми можемо потрапити або пройшовши вершину *v*, або не проходячи її. Тому рівень вершини *u* визначається як максимум між рівнем *u* і рівнем вершини *v + 1*. В результаті після обробки вершини перераховується рівень сусідніх вершин наступним чином:

$$L(u) = max(L(u), L(v) + 1)$$

Міра важливості.

Після обрахунку всіх чотирьох критеріїв можемо формально визначити міру важливості вершини *v:*

$$I(v) = ed(v) + cn(v) + sc(v) + L(v)$$

Дана міра є евристичною і ми можемо змінювати коефіцієнти при характеристиках і аналізувати як змінюється час передобробки і час пошуку маршрутів. Але в роботі реалізовано саме таку евристику, зважаючи на те, що важливість всіх критеріїв є рівною [23].

3.3 Архітектура системи

Програмний продукт реалізований на платформі .NET, мові програмування C# та технології WPF. В якості СУБД використовується Microsoft SQL Server. Реалізований алгоритми можуть використовуватись в якості веб сервера, а графічна оболонка зроблена для візуалізації роботи та для тестування.

Розглянемо детально архітектуру бази даних та серверного застосунку.

3.3.1 Архітектура бази даних

Схема бази даних зображена на рисунку 3.10.

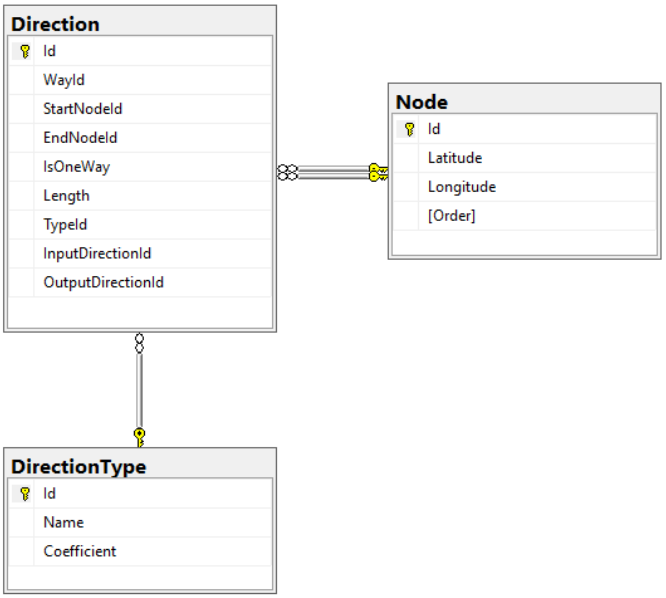


Рисунок 3.10 - Схема бази даних

Node – таблиця, що містить вершини графа.

Атрибути таблиці:

Id – унікальний ідентифікатор вершини. Значення збігається зі значенням ідентифікатора вершини в даних OpenStreetMap. Зроблено це з метою подальшого покращення та розширення функціоналу додатку за рахунок даних, які можна отримати від OpenStreetMap і які зараз не використовуються, наприклад, ми можемо дізнатися де знаходяться світлофори чи пішохідні переходи;

Latitude – географічна широта точки на карті якій відповідає дана вершина графа;

Longitude - географічна довгота точки на карті якій відповідає дана вершина графа. Обидва атрибути необхідні для відображення маршрутів на карті.

Order – атрибут порядку обробки вершини в процесі передобробки методу Contraction hierarchies. Без передобробки значення завжди рівне нулю. Поле має значення і використовується лише в алгоритмі Contraction hierarchies.

Direction – таблиця, що містить ребра графа.

Атрибути таблиці:

Id – унікальний ідентифікатор ребра графа. Автоінкрементне поле;

WayId – атрибут, що містить ідентифікатор дороги в даних OpenStreetMap. Може використовуватися для пошуку адрес на карті. Наразі поле не використовується;

StartNodeId – зовнішній ключ на атрибут [Node].[Id]. Містить посилання на вершину – початок ребра;

EndNodeId – зовнішній ключ на атрибут [Node].[Id]. Містить посилання на вершину –кінець ребра графа;

IsOneWay – булевий атрибут. Містить інформацію про направленість вершини. Значення 1 відповідає односторонньому руху вулицею. Значення 0 – двосторонній рух. Реалізовано з матою оптимізації бази даних за рахунок зменшення кількості ребер в графі. Оптимізація суттєва, оскільки після створення бази даних виявилося, що близько 80% ребер відповідають двостороннім вулицям;

Length – довжина ребра в кілометрах;

TypeId – зовнішній ключ на атрибут [DirectionType].[Id];

InputDirectionId, OutputDirectionId – атрибути, що містять посилання на ідентифікатор таблиці Direction. Заповнюються при обробці Contraction hierarchies в разі створення ребра скорочення. Необхідні для відновлення початкового маршруту. Одному ребру скорочення завжди відповідає 2 інших

ребра. Причому вони теж можуть бути ребрами скорочення. Значення використовуються для відновлення маршруту отриманого методом Contraction hierarchies.

DirectionType – таблиця, що містить типи ребер графа. Містить 13 значень – таблиця 3.1.

Таблиця 3.1 - Значення DirectionType

Id	Name	Coefficient
1	motorway	12
2	Trunk	11
3	primary	10
4	secondary	9
5	tertiary	8
6	unclassified	7
7	residential	6
8	Service	5
9	motorway_link	4
10	trunk_link	3
11	primary_link	2
12	secondary_link	1
13	Shortcut	100

Id – унікальний ідентифікатор;

Name – назва типу;

Coefficient – «важливість» ребра. Чим вище значення – тим важливішій вулиці відповідає дане ребро. Очевидно, при навігації краще обирати великі дороги з типом магістраль – motorway, а не допоміжні – service. Дані значення використовуються в алгоритмі A* при виборі напрямку руху хвилі: враховується відстань до кінцевої вершини і тип дороги.

3.3.2 Архітектура серверного застосунку

На рисунку 3.9 зображено загальну схему розробленої системи.

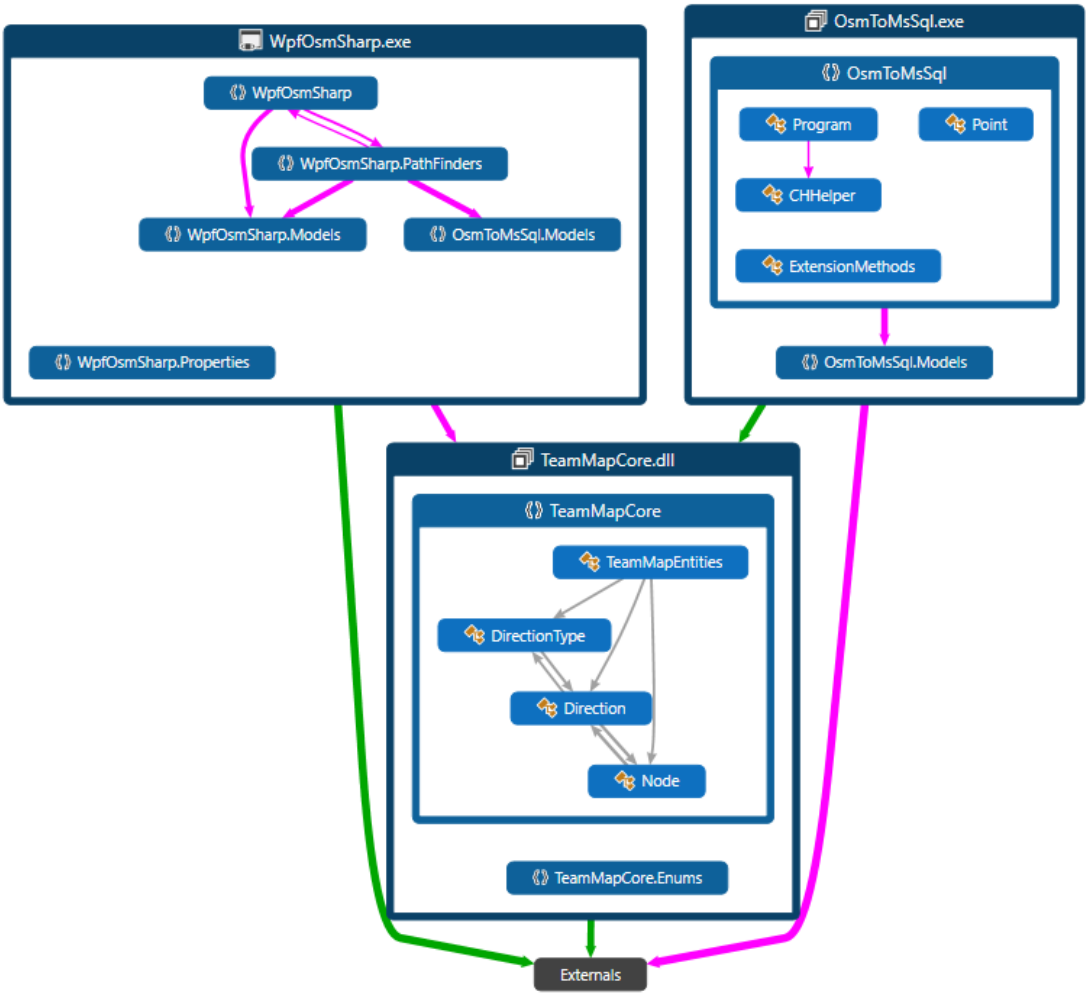


Рисунок 3.9 - Архітектурна схема системи

Рішення складається з трьох компонентів: WpfOsmSharp – агент роутингу реалізовний на фреймворку Windows presentation foundation [25], OsmToMsSql – агент генерування бази даних та обробки отриманого графа доріг. TeamMapCore – компонент зв’язку з базою даних, містить контекст бази даних і надає інтерфейс для отримання, зміни та додавання даних в базу даних.

Розглянемо детальніше кожен компонент.

TeamMapCore

UML-діаграма класів компонента зображена на рисунку 3.10.

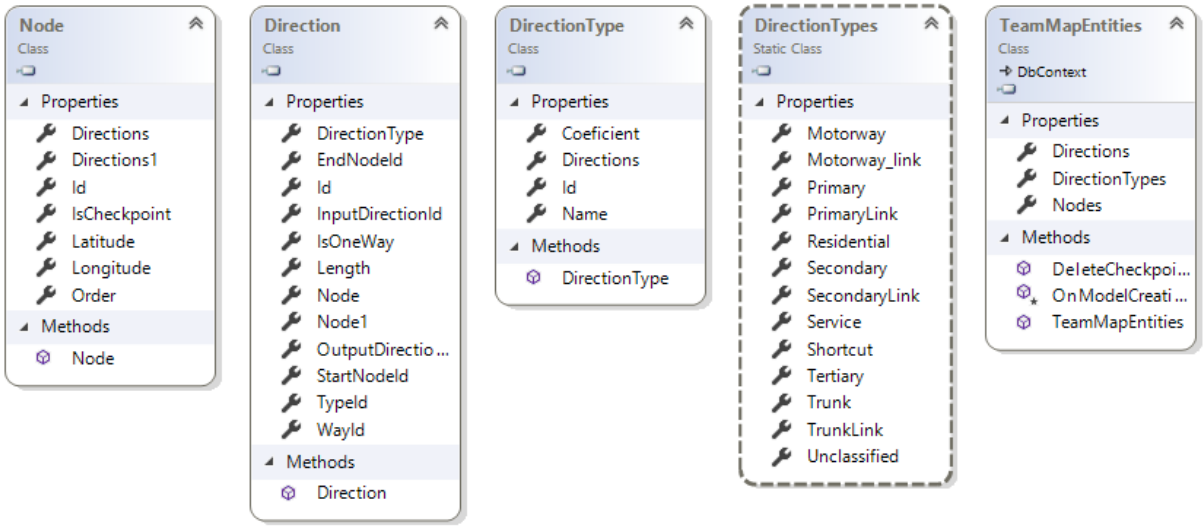


Рисунок 3.10 - UML-діаграма класів TeamMapCore

Компонент містить лише моделі, контекст бази даних TeamMapEntities та допоміжний статичний клас зі значеннями таблиці DirectionType.

OsmToMsSql

Компонент обробки даних від OpenStreetMap та передбробки Contraction hierarchies.

UML-діаграма класів компонента зображена на рисунку 3.11.

Основними класами компонента є Program та CHHelper.

Клас Program виконує генерацію бази даних з XML файлу OpenStreetMap.

Структура картографічного XML файлу OpenStreetMap та його обробка.

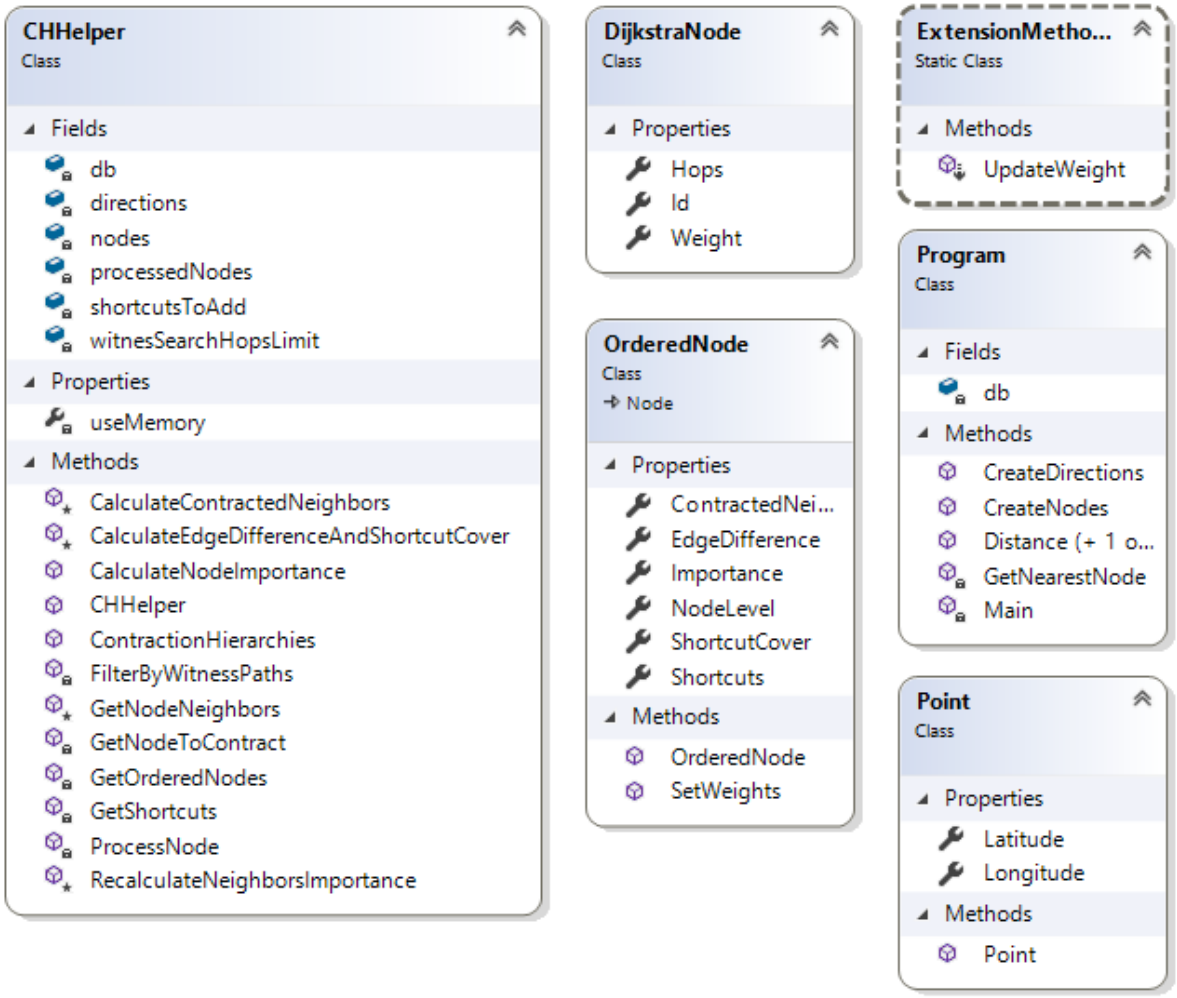


Рисунок 3.11 - UML-діаграма класів OsmToMsSql

Кореневим тегом є тег osm. В нього входять вся інформація, яку несе документ. Для нас важливими є теги node та way. Крім них в документ входять також теги relation, але наразі вони не використовуються.

Тег node містить ряд атрибутів серед яких є id, latitude, longitude. Саме їх ми використовуємо для створення вершини графу. Крім атрибутів node може містити масив тегів tag, які містять додаткову інформація про точку карти, наприклад, тип дороги на якій вона лежить, чи інформацію про будівлю якій вона відповідає, якщо це не є точкою дорожньої мережі. Однією з проблем при генерації бази даних є те, що документ є перенасиченим інформацією і обробити її всю досить складно. Основною ж задачею є профільтрувати дані,

які нам необхідні. Так, наприклад, нам необхідно додавати до нашого графу доріг лише вершини, що є частиною дорожньої мережі, і навпаки, необхідно пропускати вершини які їй не належать. Саме цьому і слугують теги tag. Крім того з них можна отримати додаткову дорожню інформацію яка могла б бути корисною, наприклад, дорожні знаки чи світлофори. Але така обробка не є ціллю роботи, тому вона залишена на майбутнє.

Тег way представляє собою вулицю. Містить в собі масив тегів nd з єдиним атрибутом ref, що посилається на id тегу node. Очевидно, може складатися з двох і більше вершин на карті. Нам необхідно з тегів way створити записи в таблиці Direction нашої бази даних. В даному випадку нічого фільтрувати не потрібно, за винятком тих вершин, які не потрапили в завантажений документ і не були попередньо додані до бази даних. Таке може трапитися, оскільки ми не завантажуємо карту всього світу, вона не є повною і в якийсь момент обривається. Тому інформація в XML файлі не є цілісною. Теж way також містить масив тегів tag, що несуть додаткову інформацію, таку як тип дороги, який ми зберігаємо в базі даних у вигляді зовнішнього ключа на таблицю DirectionType, назва вулиці і т.д.

Теги relation містять масив тегів member, що має атрибут type, який може набувати значення way або node. Таким чином тег relation об’єднують у собі шляхи і вершини. Він може описувати різноманітні об’єкти, наприклад, будинки. В нашій роботі дані тегів relation не використовуються.

Обробка картографічного файлу є нетривіальною задачею, оскільки мова йде про величезні об’єми даних. Так, наприклад, файл з картою Києва містить більше 3,5 млн. рядків. А згенерована база даних містить близько 160 000 вершин і 175 000 ребер.

Клас CHHelper виконує передобробку бази даних в методі Contraction hierarchies.

Складність реалізації обробки бази даних пов’язана з великими об’ємами даних.

По-перше, при такій кількості даних звернення до бази даних займає велику кількість часу. Тому варто працювати з даними в пам’яті. Звісно. їх максимальна кількість значно більше обмежена ніж можлива кількість даних в базі даних, і при збільшенні площі карти, що оброблюється, максимальна кількість буде досягнута. В зв’язку з цим, карти великих розмірів доведеться обробляти частинами, що породжує нові проблеми пов’язані з алгоритмом обробки, а саме з порядком вершин і з подальшим пошуком. Дана проблема може бути предметом подальшого дослідження.

По-друге, багато обчислень в алгоритмі є незалежними один від одного. Наприклад, для розрахунку початкового порядку вершин ми повинні обрахувати 4 характеристики для всіх вершин. Найважчою в плані обчислень буде характеристика «різниця ребер», адже для її розрахунку нам доведеться знайти всі ребра скорочення, що будуть додані в разі її обробки, а це передбачає багаторазовий запуск «дорогої» процедури Witness search – пошуку можливих коротших маршрутів. В процесі обробки вершин, як вже було відмічено раніше, порядок обробки може змінюватись, але при початковому сортуванні вершини незалежні між собою і можуть розраховуватись паралельно. Цим необхідно користуватись, оскільки це може скоротити час сортування до чотирьох разів.

Саму процедуру обробки запустити в паралельно режимі не можна, оскільки в цьому разі ми отримаємо некоректний порядок вершин, що значно погіршить навігацію по графу. «Найдорожчими» операціями в процедурі обробки є вибір вершини, яку слід обробляти, їй відповідає метод GetNodeToContract в UML-діаграмі, і фільтрування знайдених ребер скорочення в методі FilterByWitnessPath.

Обидві ці процедури можна оптимізувати, запускаючи, в першому випадку, перерахунок порядку одразу для n вершин в черзі, де n відповідає кількості ядер процесора машини на якій виконується обробка. В другому випадку можна паралельно запускати n процедур Wintess search.

Процедуру Witness search можна також покращити, запускаючи алгоритм Дейкстри не для всіх знайдених ребер скорочення окремо, а об’єднуючи їх в групи за вершиною початку і зупиняти алгоритм при досягненні найбільш віддаленої вершини, або при досягненні іншої умови зупинки. Таким чином ми зменшимо кількість запусків процедури в 4-25 разів.

По-третє, очевидно, складність алгоритму обробки росте зі збільшенням степені вершини графа. Тобто, якщо лише 2 ребра інцидентні вершині то може бути додано лише 2 ребра скорочення (в обох напрямках у випадку двонаправлених ребер) і тільки для них необхідно запускати Witness search. Така ситуація цілком можлива і, навіть поширена, на початку обробки графа. Тому на початку обробка проходить досить швидко. Але при додаванні ребер скорочення, степені вершин графа ростуть і кількість запусків Witness search теж росте. Саме тому важливе значення має значення максимальної кількості вершин від початкової і кінцевою в процедурі Witness search. Якщо на початку обробки навіть значення 5 не є критичним і суттєво не уповільнює роботу алгоритму, то в кінці, коли кількість ребер інцидентних вершині може досягати 1000, цей показник є дуже важливим. Звісно, зменшивши його ми погіршуємо якість результуючого графа, але значно прискорюємо алгоритм обробки. Тому в процесі обробки можна змінювати його значення від 5 на початку до 2 в кінці.

Для розуміння важливості оптимального використання ресурсів комп’ютера наведемо отримані результати часу обробки. У випадку повної карти Києва маємо близько 160 000 вершин. Якщо використовувати послідовну обробку при сортуванні вершин, сам алгоритм сортування працює близько 13 годин. При використанні паралельної обробки час роботи зменшився до 3 годин.

Висновки за розділом 3

В розділі було детально розглянуто алгоритм Contraction hierarchies. Складові його реалізації – сортування вершин для передобробки, witness search, механізм обробки вершини, модифікований двонаправлений алгоритм Дейкстри та доведено коректність алгоритму.

В розділі було розглянуто архітектуру розробленого програмного продукту: архітектуру бази даних та трьох складових компонент серверного додатку. Також розглянуто схему картографічного файлу OpenStreetMap та запропонований алгоритм його обробки для фільтрації надлишкової інформації, вибору необхідних даних для створення графу доріг і оптимізації його. Розглянуто UML-діаграми класів розробленого програмного продукту та діаграму залежностей.

РОЗДІЛ 4. РЕЗУЛЬТАТИ РОБОТИ

Умовно роботу можна розділити на дві частини: реалізація алгоритмів пошуку маршруту і створення і обробка бази даних.

Проаналізуємо результати роботи обох частин.

4.1 Обробка бази даних

Створення бази даних з XML файлу OpenStreetMap – процедура, що потребує небагато часу. Наведемо результати створення графу для карти Києва та приміських територій – рисунок 4.1.

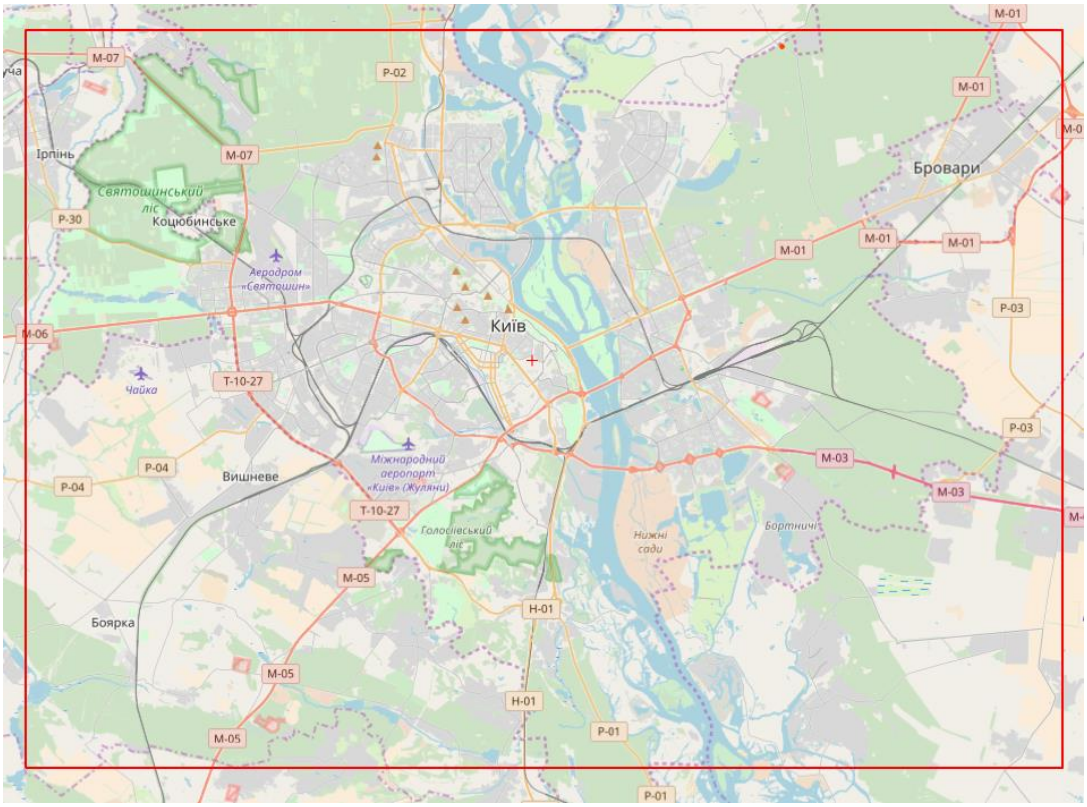


Рисунок 4.1 - Оброблена область Києва та приміських територій

Заповнення таблиці Node всіма значеннями з файлу (1 077 306 значень) триває 7 хвилин 43 секунди. Фільтрація ребер, заповнення таблиці Direction та видалення зайвих вершин триває 25 хвилин 20 секунд. Тобто разом процедура генерації графу доріг триває 33 хвилини 3 секунди. В результаті отримали 161 059 вершин та 175 167 ребер. Можна відмітити, що тривалість роботи лінійно залежить від розміру файлу, що обробляється, тобто від розмірів території. Це достатньо добрий результат, оскільки дана операція виконується лише один раз.

Обробка бази даних для методу Contraction hierarchies – операція значно складніша і потребує багато часу. В межах даної роботи не було необхідності обробляти повну карту Києва, для тестування та порівняння роботи алгоритмів цілком достатньо і її частини, тому було оброблено лише частину Києва – рисунок 4.2.

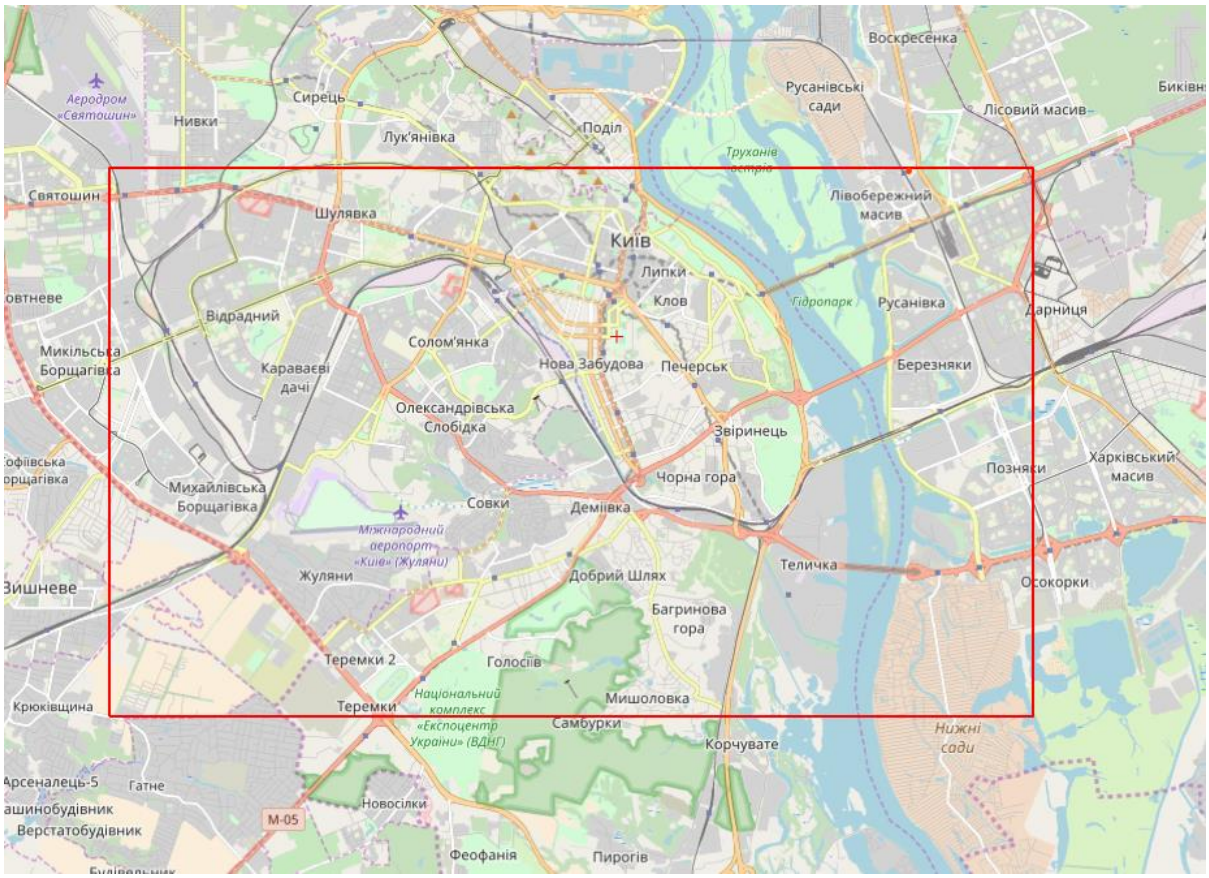


Рисунок 4.2 - Область, оброблена алгоритмом Contraction hierarchies

В відмічену область входить 64 526 вершин та 69 536. Обробка тривала близько 35 годин. В результаті отримали 190 793 ребра скорочення, з яких дублікатами виявилися 67 077 ребер.

Під дублікатами маються на увазі ребра скорочення, що з'єднують однакові вершини. Їхнє утворення можливе у випадку коли процедура Witness search виконується не повністю, а припиняється при досягненні ланцюжків довжини n. В нашому випадку спочатку $n=5$, поступово зменшувалося до 2, а для останніх 100 вершин дорівнювало 1, тобто відсіювалися тільки ті ребра, які вже існують і довжина яких більша за існуючі. Причому, навіть така фільтрація була дуже результативною. Наприклад, якщо можливих ребер скорочення для вершини було порядку 4000, то після фільтрації їх залишалося близько 100.

Звісно, ми отримаємо більш якісний граф при великому значення n або взагалі при відсутності такої умови зупинки процедури Witness search ($n = +\infty$), але це значно впливає на час обробки. Тому, в дослідницьких цілях достатньо згенерувати граф посередньої якості, що і було зроблено.

Дублікати були видалені з бази даних зі збереженням цілісності зв'язків. Таким чином в результуючій базі даних міститься 64 526 вершин та 193 252 ребер.

Відмітимо, що тривалість обробки залежить від розміру карти нелінійно і тривалість роботи алгоритму на карті всього Києва може сягати кількох тижнів.

4.2 Алгоритми пошуку

Для порівняння роботи алгоритмів пошуку будемо запускати процедури пошуку на різних маршрутах. Причому маршрути в вибірці повинні бути як короткими так і довгими, оскільки реалізовані алгоритми можуть по-різному показати себе на маршрутах різної довжини. Одразу можна зробити припущення, що алгоритм Contraction hierarchies буде найкращим і його перевага буде найбільш очевидною саме на довгий маршрутах, а A* добре покаже себе на маршрутах з короткою і прямою траєкторією.

Загалом проаналізуємо по 3 короткі і довгі маршрути.

Маршрут №1. Прямий маршрут по проспекту Перемоги – від цирку до Бесарабською площі. Довжина 2,19 км.

Contraction hierarchies - рис. 4.3:

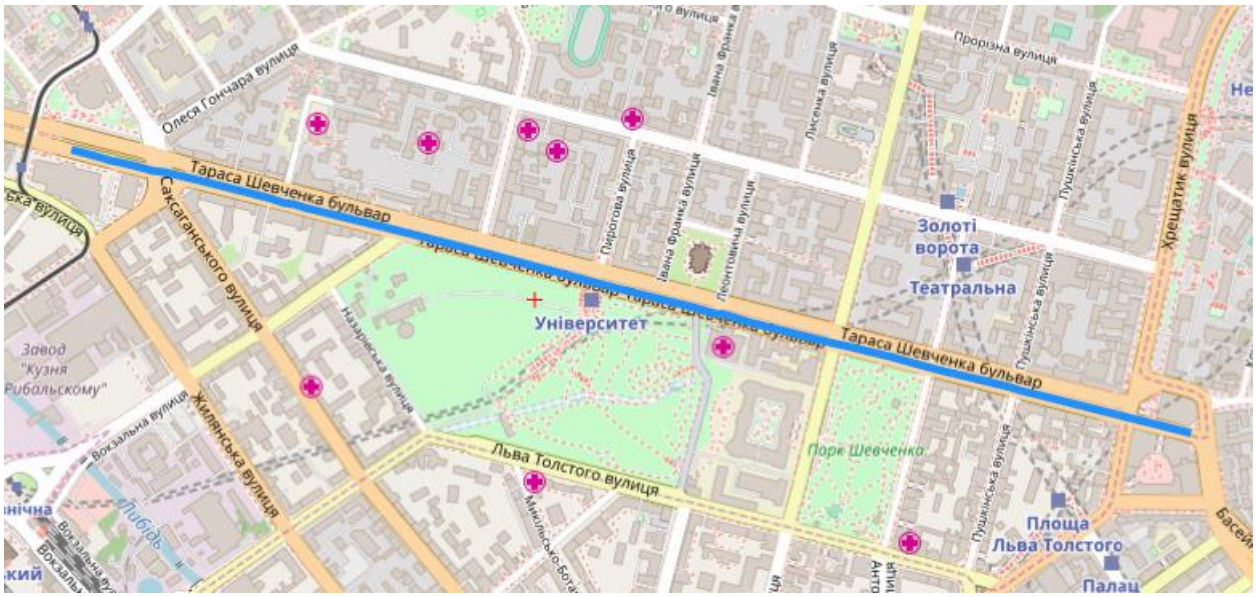


Рисунок 4.3 - Маршрут №1 – Contraction hierarchies

Час пошуку – 1,23 с.

Дистанція – 2,19 км.

A:*

Знайдений маршрут ідентичний попередньому.

Час пошуку – 0,25 с.

Дистанція 2,19 км.

A* двонаправлений:

Знайдений маршрут ідентичний попередньому.

Час пошуку – 0,31 с.

Дистанція 2,19 км.

Результат Google maps – рис.4.4.

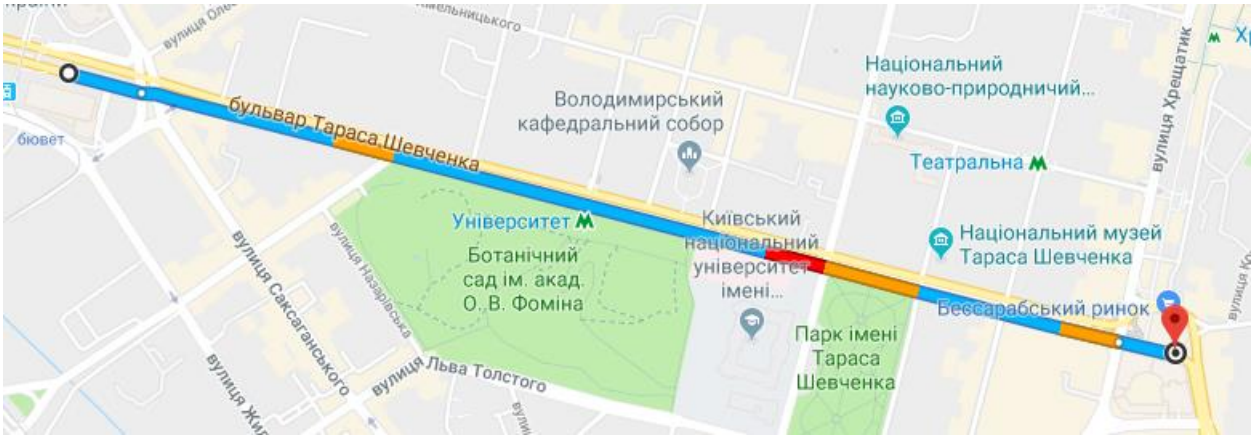


Рисунок 4.4 - Маршрут №1 – Google maps

Дистанція – 2,2 км.

Перший маршрут найпростіший і всі алгоритми показали однаковий результат пошуку. Найшвидшим виявився алгоритм A*, оскільки йому в даному випадку довелося перебрати меншу кількість вершин ніж алгоритму CH (Contraction hierarchies) , а також не довелося витрачати час на переключення між потоками виконання, на відміну від двонаправленого алгоритму A*. Усі маршрути аналогічні маршруту Google maps.

Маршрут №2. Маршрут від цирку до станції метро Олімпійська. Довжина 3,33 км.

Contraction hierarchies – рис.4.5:

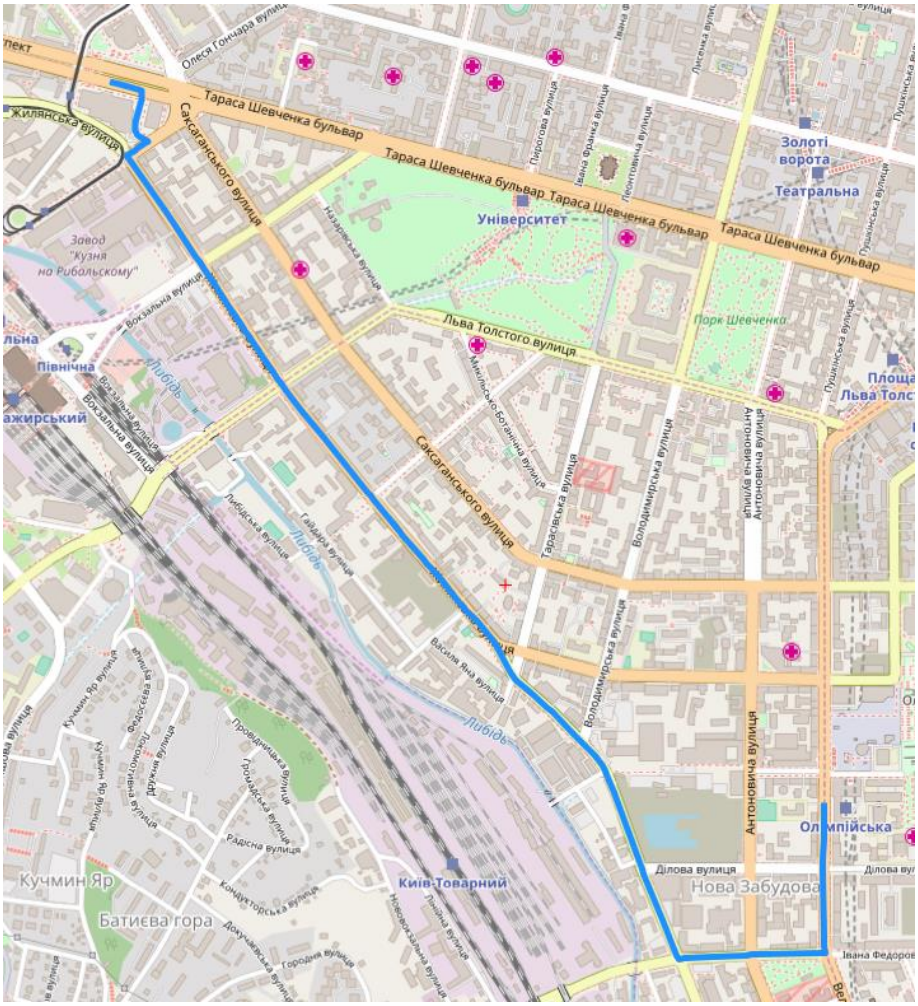


Рисунок 4.5 - Маршрут №2 – Contraction hierarchies

Час пошуку – 1,14 с.

Дистанція – 3,33 км.

A* - рис.4.6:

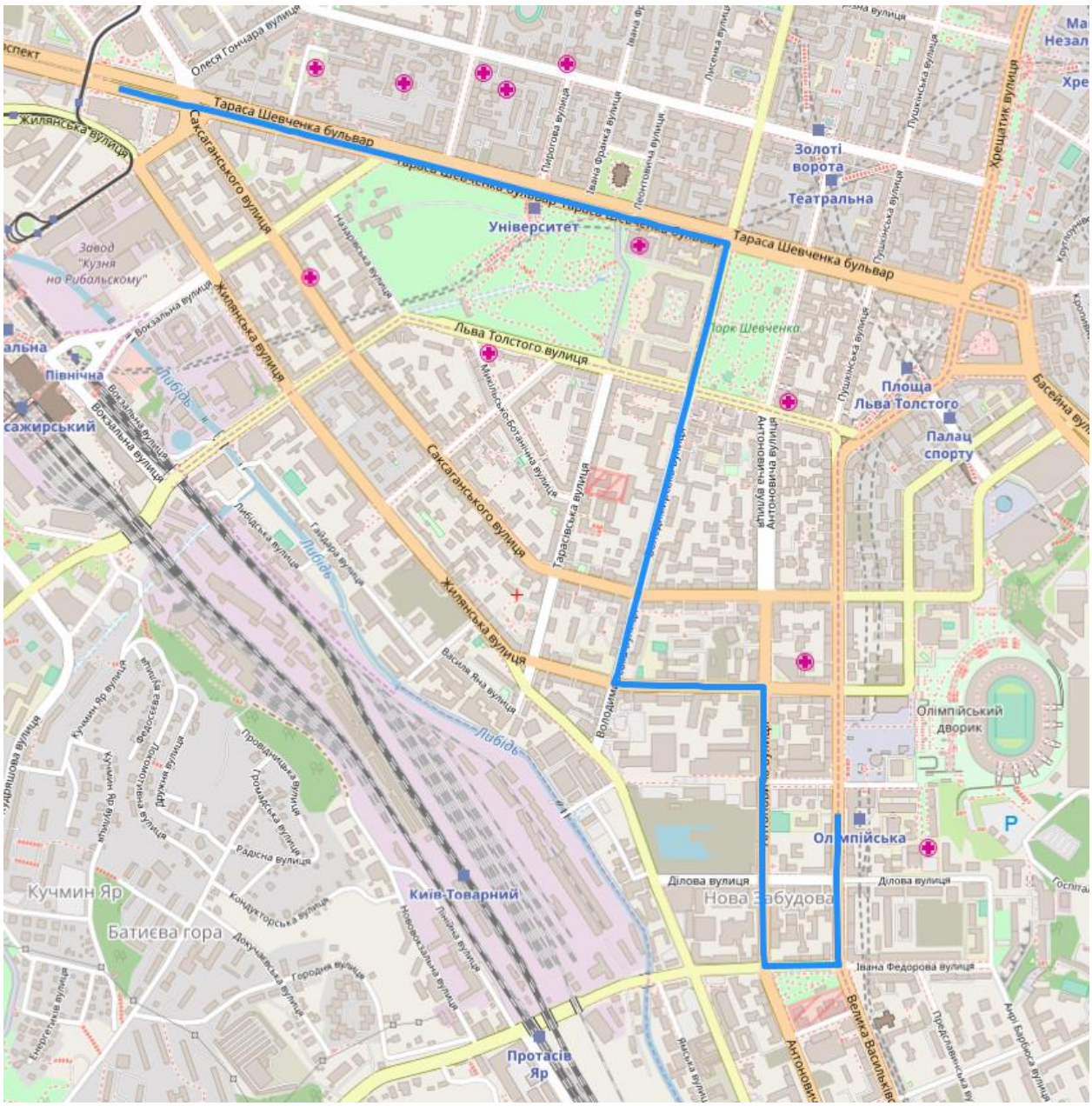


Рисунок 4.6 - Маршрут №2 – A*

Час пошуку – 0,98 с.

Дистанція – 4,12 км.

А* двонаправлений – рис.4.7:

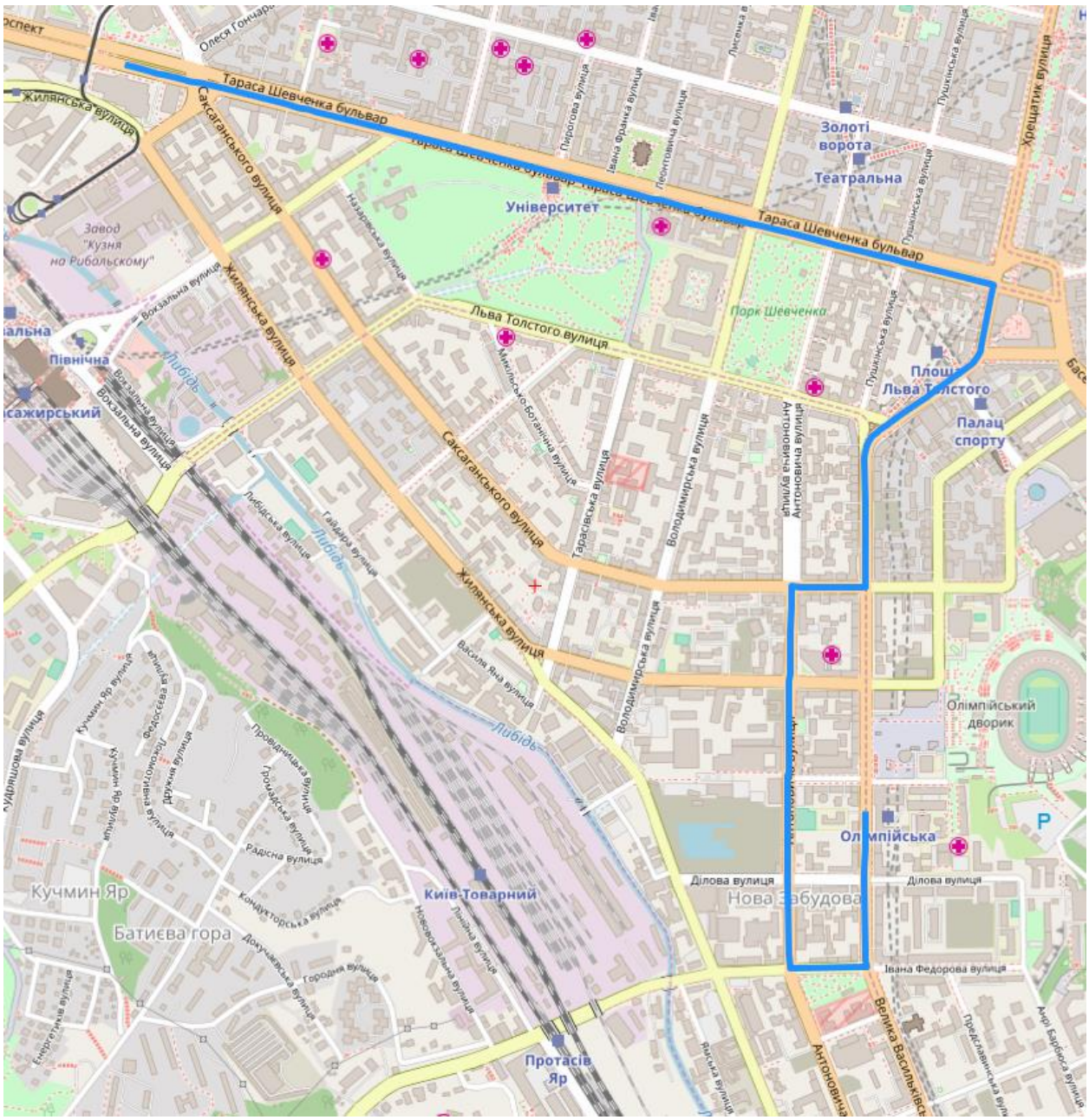


Рисунок 4.7 - Маршрут №2 – А* двонаправлений

Час пошуку – 1,04 с.

Дистанція 4,48 км.

Результат Google maps – рис. 4.8.

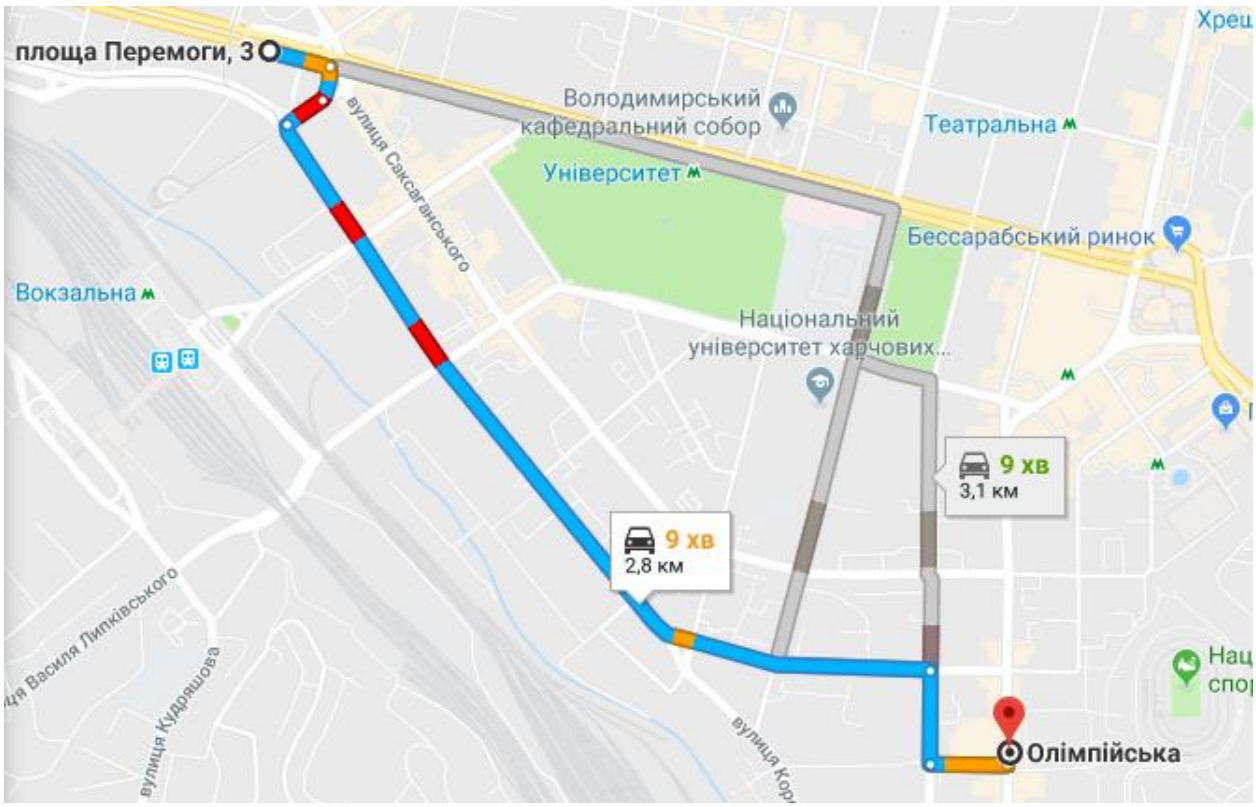


Рисунок 4.8 - Маршрут №2 – Google maps

Довжина – 2,8 км.

В даному випадку всі три маршрути виявилися різними. Це пояснюється тим, що в алгоритмах A* та A* двонаправлений враховується тип дороги, причому враховується по-різному. Алгоритм CH завжди шукає найкоротший шлях незважаючи на тим дороги. Найкраще по часу знову показав себе алгоритм A*, але довжина його маршруту значно більша від результату CH. Якісь доріг в даному випадку співвимірна, а час роботи в CH незначно більший. Тому можна вважати, що в другому випробуванні найкращим виявився алгоритм CH. Найгірше себе проявив A* двонаправлений, показавши середній час та значно більшу відстань.

Результат Google maps дещо кращий за результат CH. Як альтернативу Google maps пропонує маршрут схожий на побудований алгоритмом A*.

Маршрут №3. Маршрут від перетину вулиць Лютеранської та Шовковичної до станції метро Кловська. Довжина – 2,23 км.

Contraction hierarchies – рис.4.9:

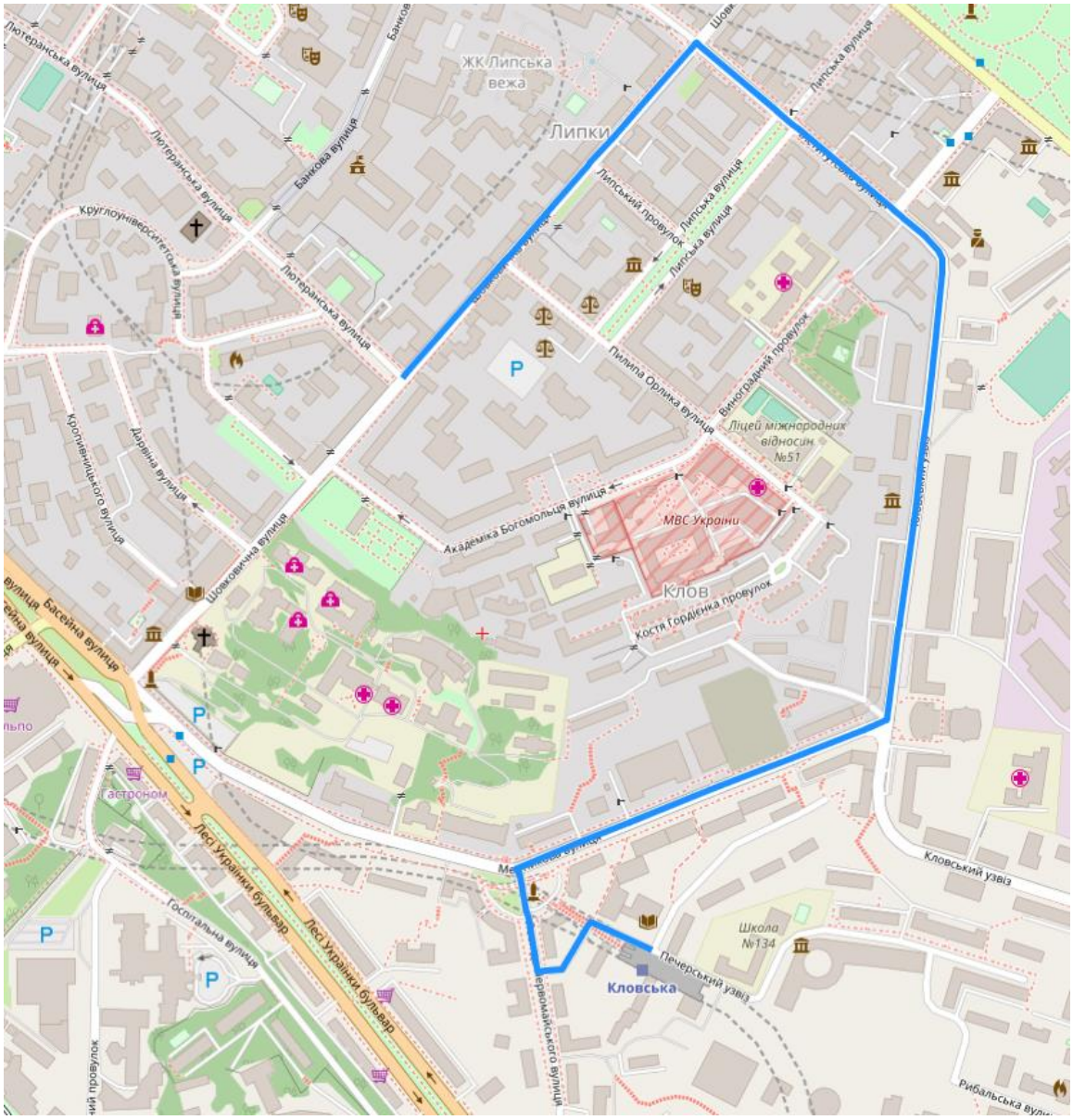


Рисунок 4.9 - Маршрут №3 – Contraction hierarchies

Час пошуку – 0,96 с.

Дистанція – 2,23 км.

А* - рис.4.10:

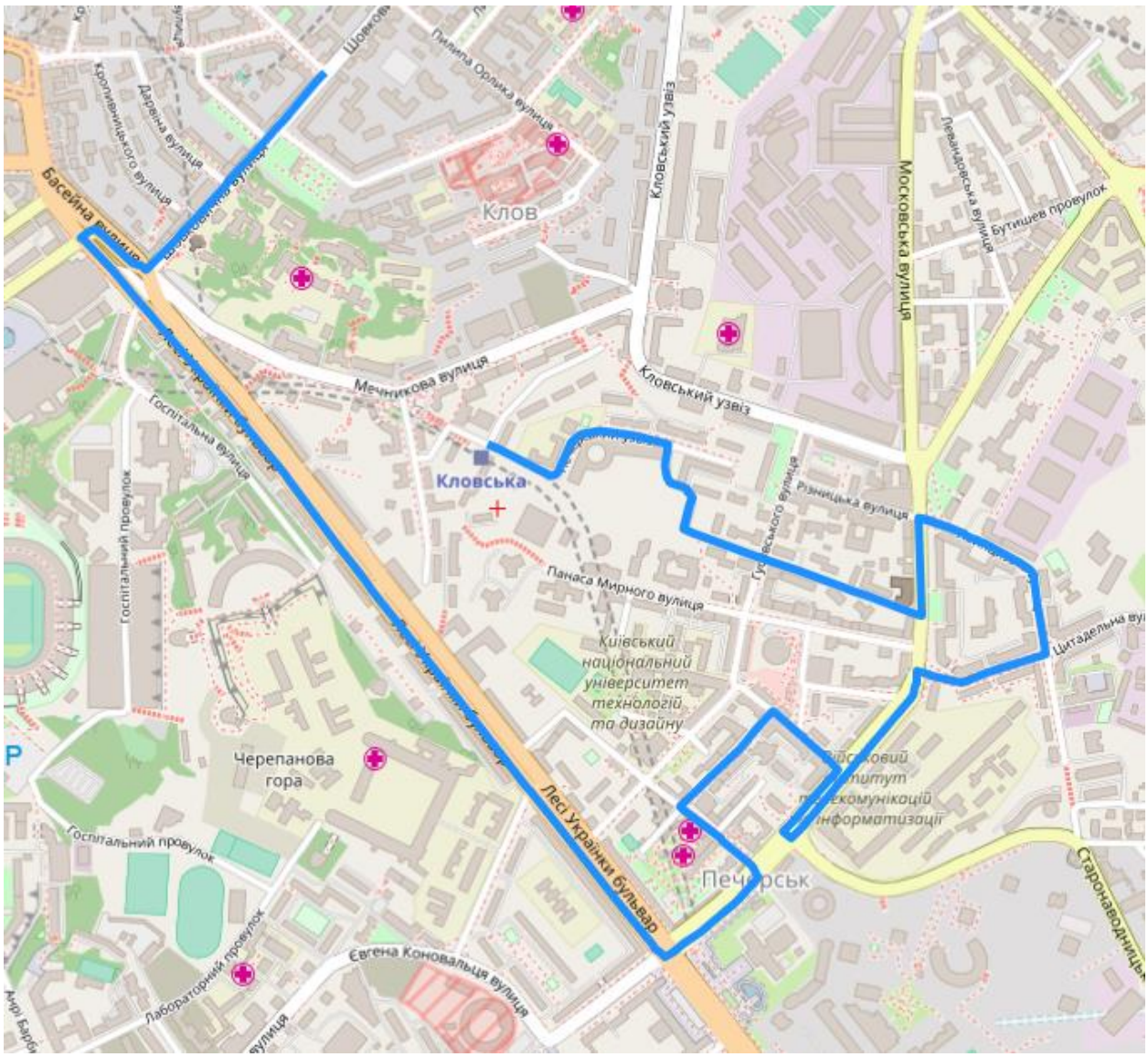


Рисунок 4.10 - Маршрут №3 – А*

Час пошуку – 6,18 с.

Дистанція – 5,51 км.

A* двонаправлений – рис. 4.11:

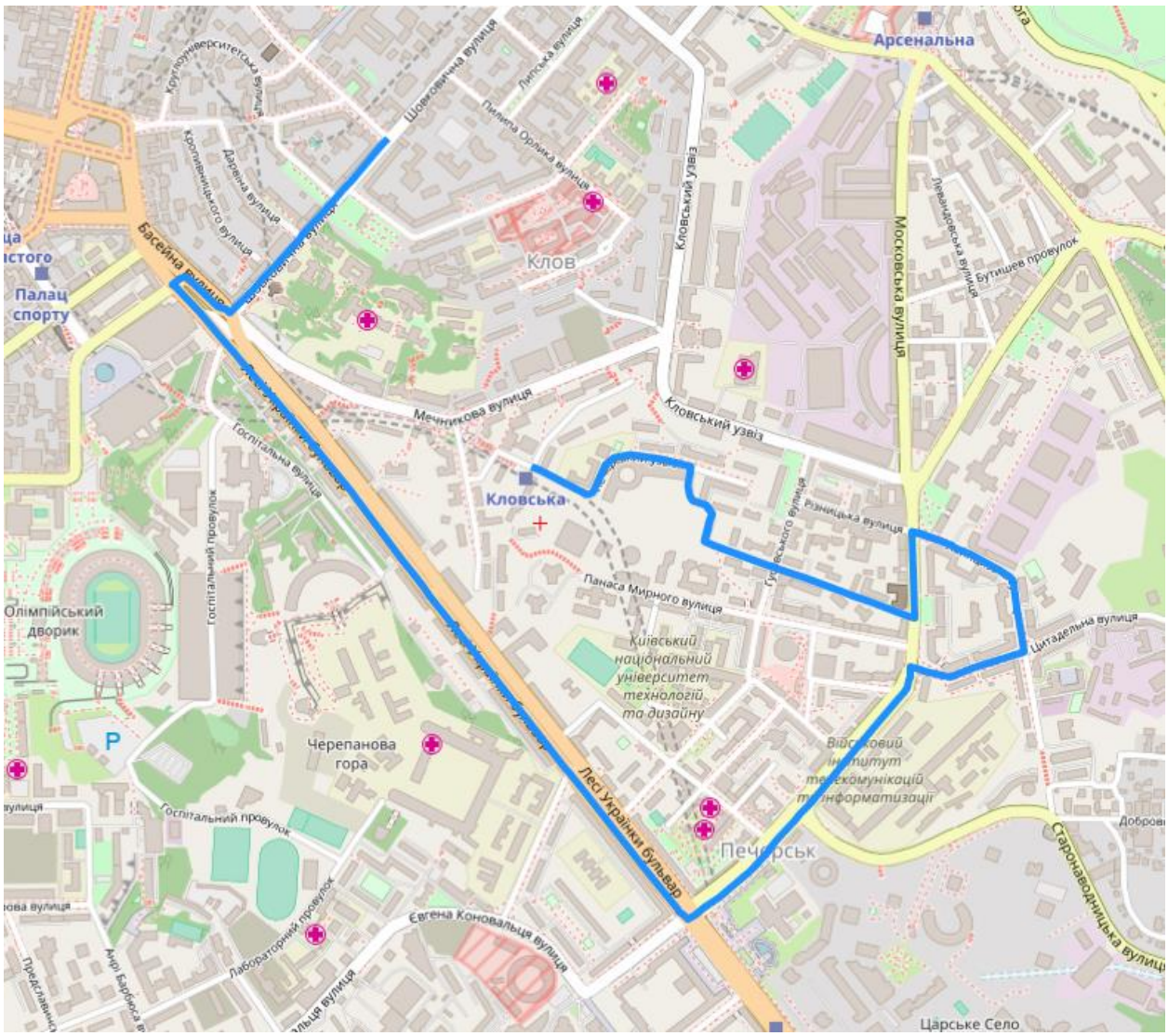


Рисунок 4.11 - Маршрут №3 – A* двонаправлений

Час пошуку – 16,52 с.

Дистанція 4,84 км.

Результат Google maps – рис.4.12.

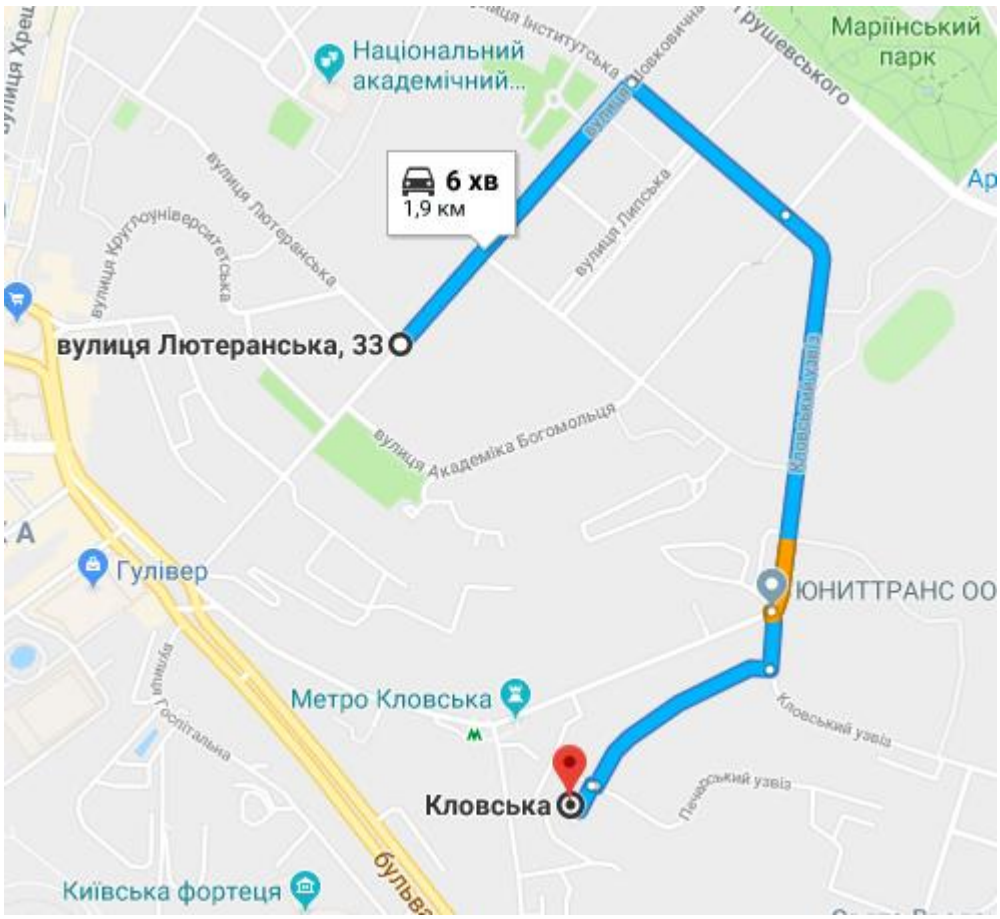


Рисунок 4.12 - Маршрут №3 – Google maps

Дистанція – 1,9 км.

В даному випробуванні алгоритми A* показали дуже поганий результат, як по часу, дистанції, так і по якості маршруту при візуальній оцінці. Пояснити це можна тим, що ці алгоритми прагнуть прокласти маршрут по великим дорогам, на відміну від CH, який не вибирає тип дороги а будує маршрут по вже створеній ієрархії вершин.

В порівнянні з Google maps усі алгоритми проявили себе гірше. Відставання CH незначне.

Маршрут №4. від станції метро Шулявська до станції метро Деміївська. Довжина – 8,78 км.

Contraction hierarchies – рис. 4.13:

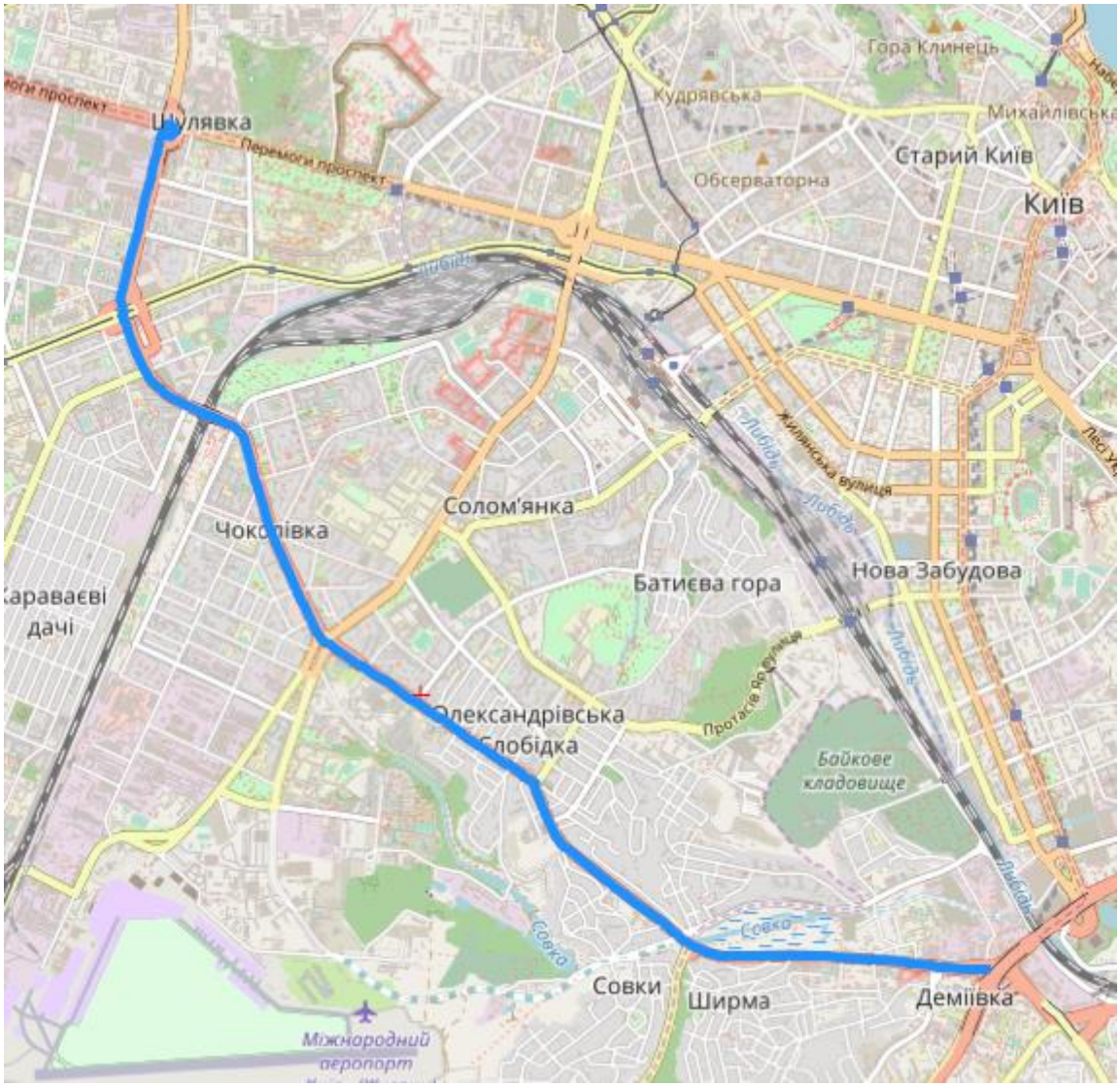


Рисунок 4.13 - Маршрут №4 – Contraction hierarchies

Час пошуку – 1,48 с.

Дистанція – 8,78 км.

A* - рис. 4.14:

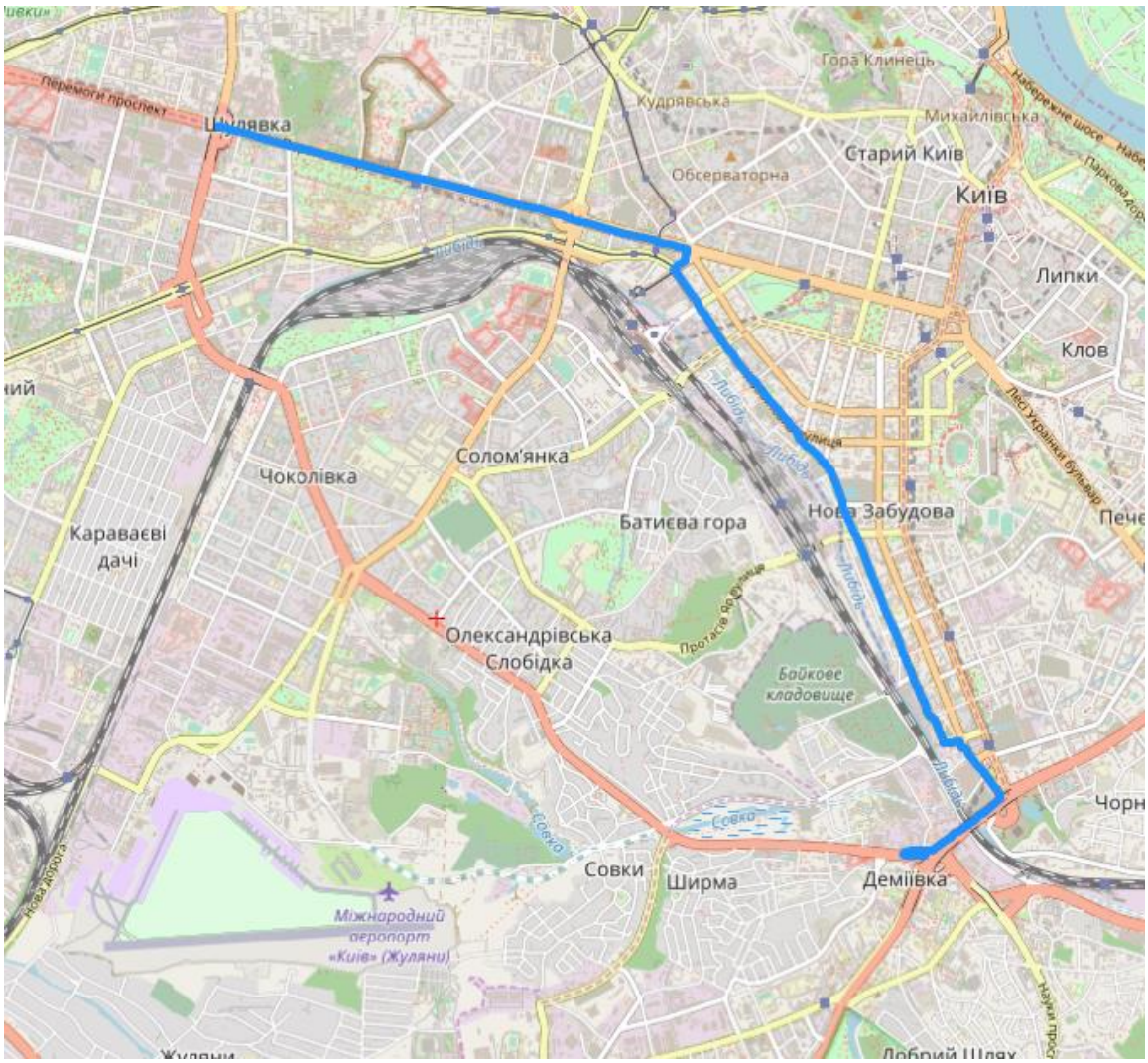


Рисунок 4.14 - Маршрут №4 – A*

Час пошуку – 2,37 с.

Дистанція – 9,4 км.

A* двонаправлений – рис. 4.15:

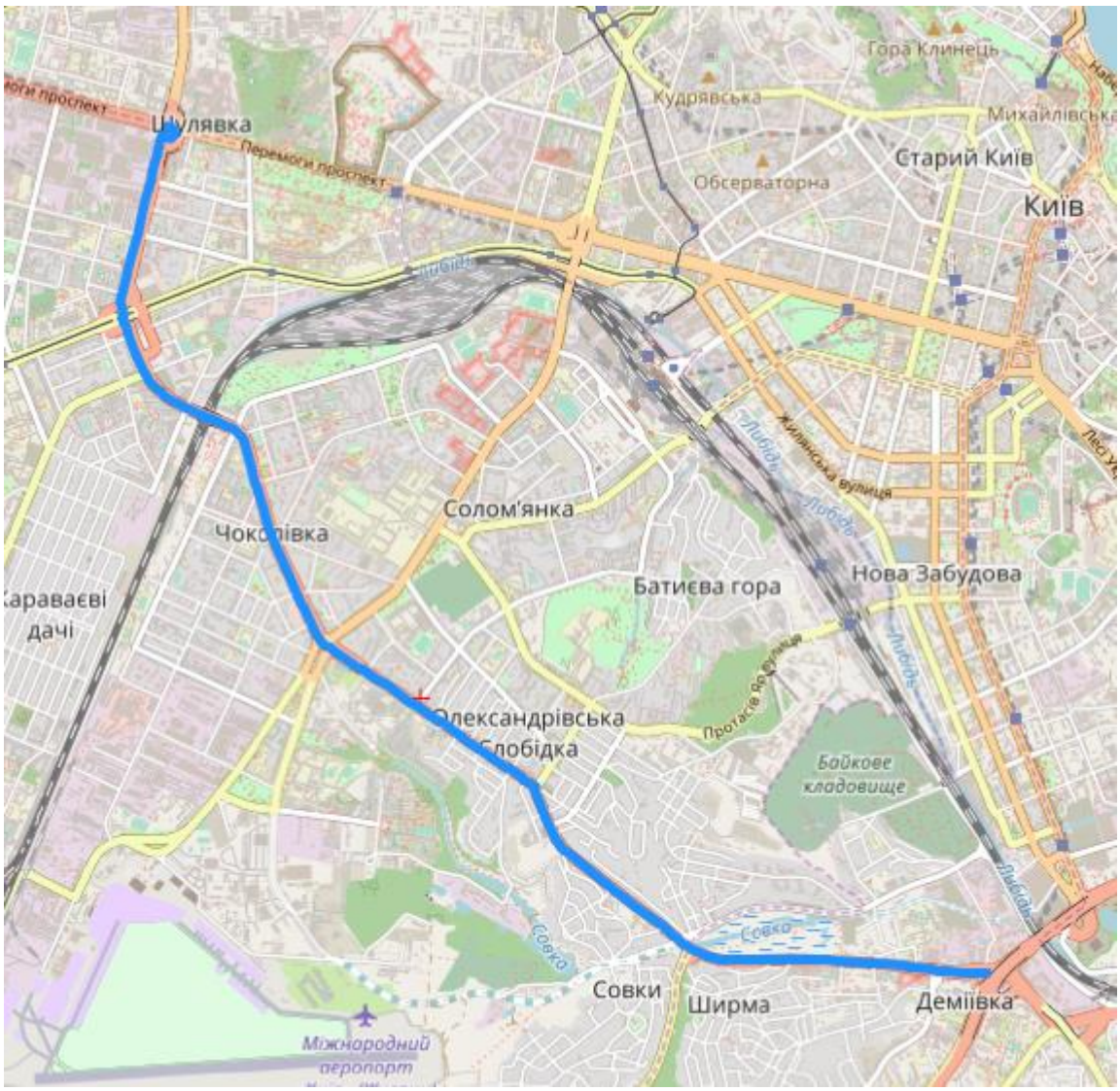


Рисунок 4.12 - Маршрут №4 – A* двонаправлений

Час пошуку – 1,17 с.

Дистанція 8,78 км.

Результат Google maps – рис. 4.16.

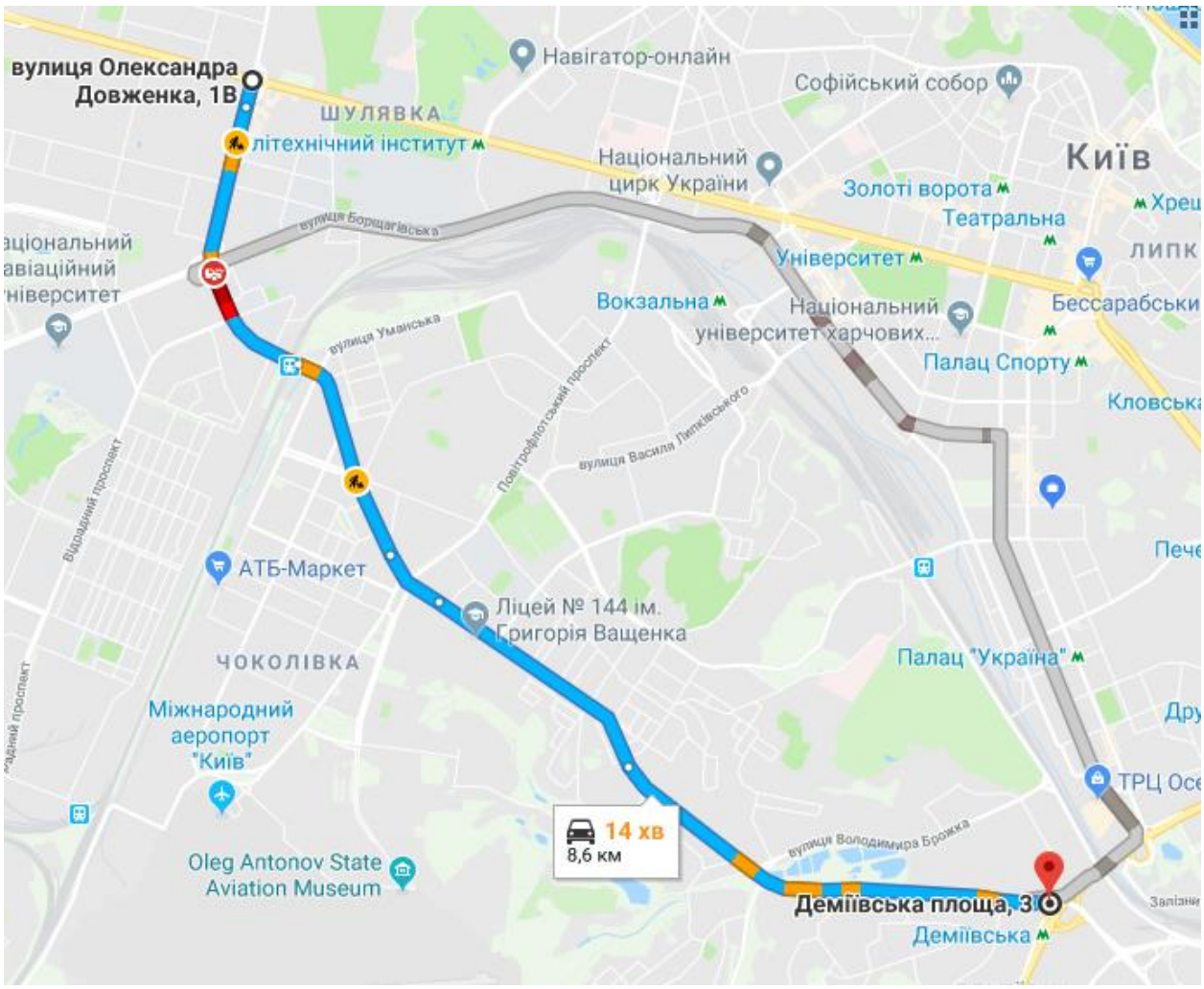


Рисунок 4.16 - Маршрут №4 – Google maps

Довжина – 8,6 км.

Найкращим виявився алгоритм A* двонапрвлений, показавши оптимальним маршрут за найменший час. CH знайшов цей маршрут за трохи більший час. A* проклав довший маршрут за більший час.

Результати CH та A* практично співпадають з Google maps. Різницю можна пояснити похибкою вибору початку і кінця маршруту.

Маршрут №5. Маршрут від Майдану Незалежності до Виставкового центру. Довжина - 9,96 км.

Contraction hierarchies – рис 4.17:

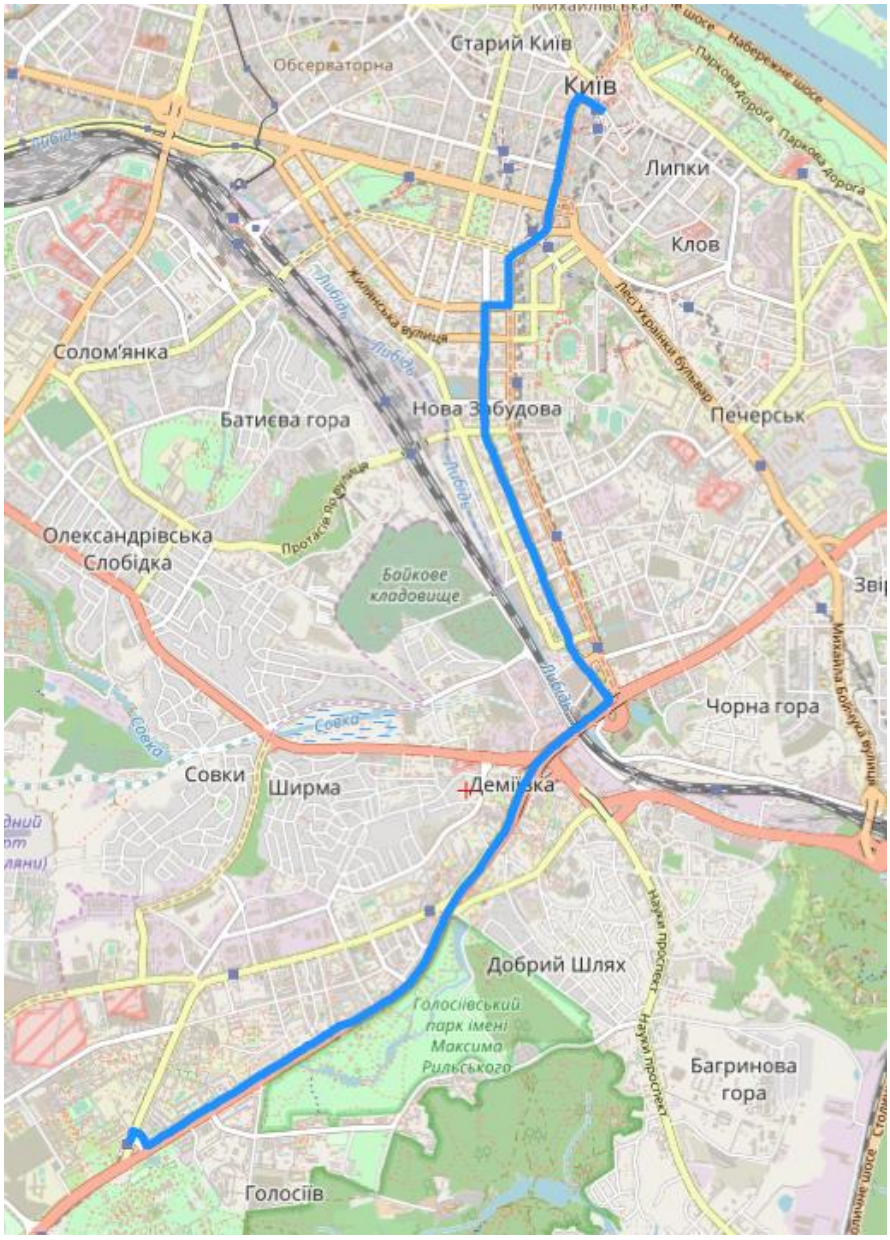


Рисунок 4.17 - Маршрут №5 – Contraction hierarchies

Час пошуку – 2,62 с.

Дистанція – 9,96 км.

A* - рис. 4.18:

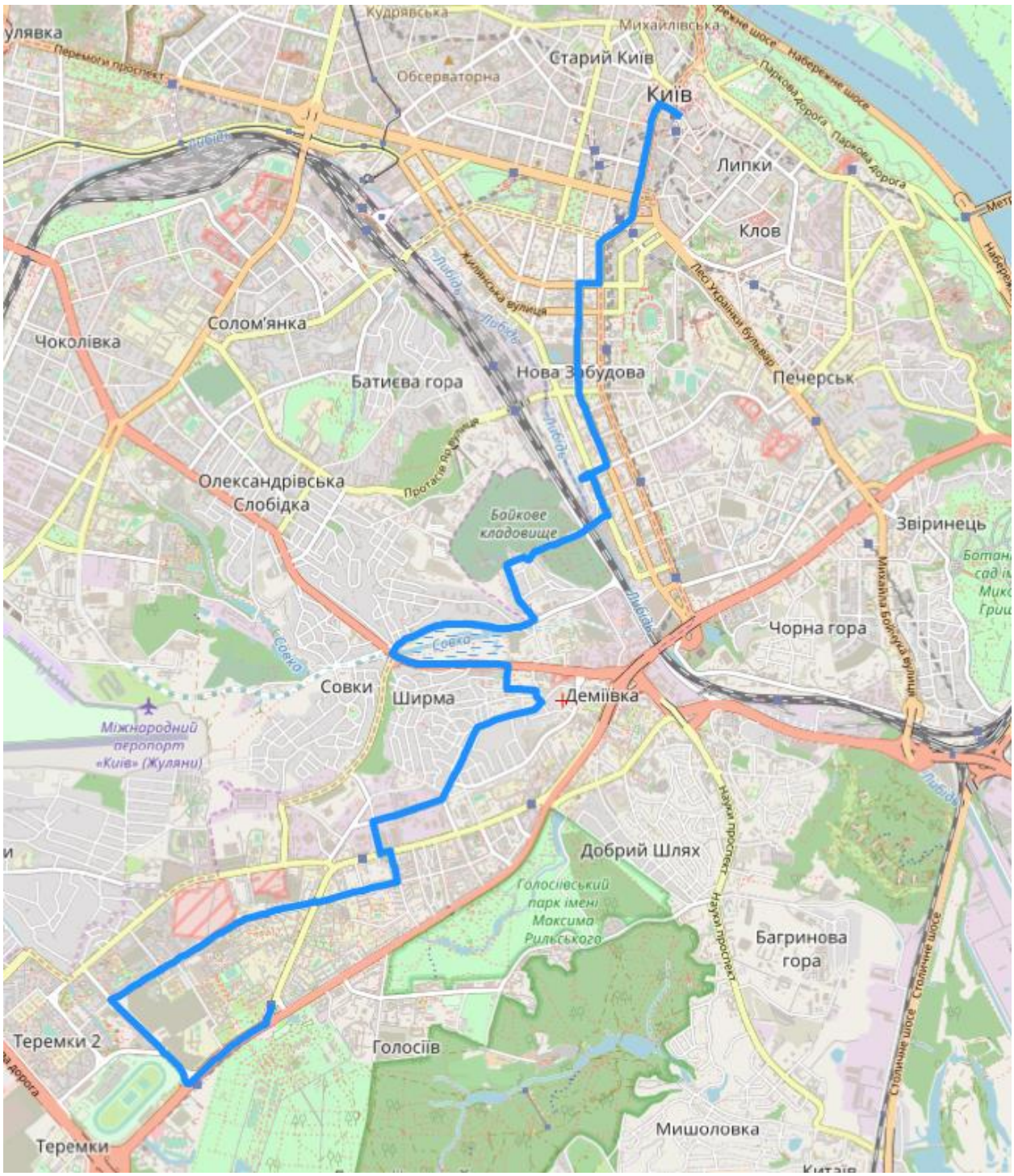


Рисунок 4.18 - Маршрут №5 – A*

Час пошуку – 11,78 с.

Дистанція – 15,62 км.

А* двонаправлений – рис. 4.19:

Рисунок 4.19 - Маршрут №5 – А* двонаправлений

Час пошуку – 82,82 с.

Дистанція 23,07 км.

Результати Google maps – рис. 4.20:

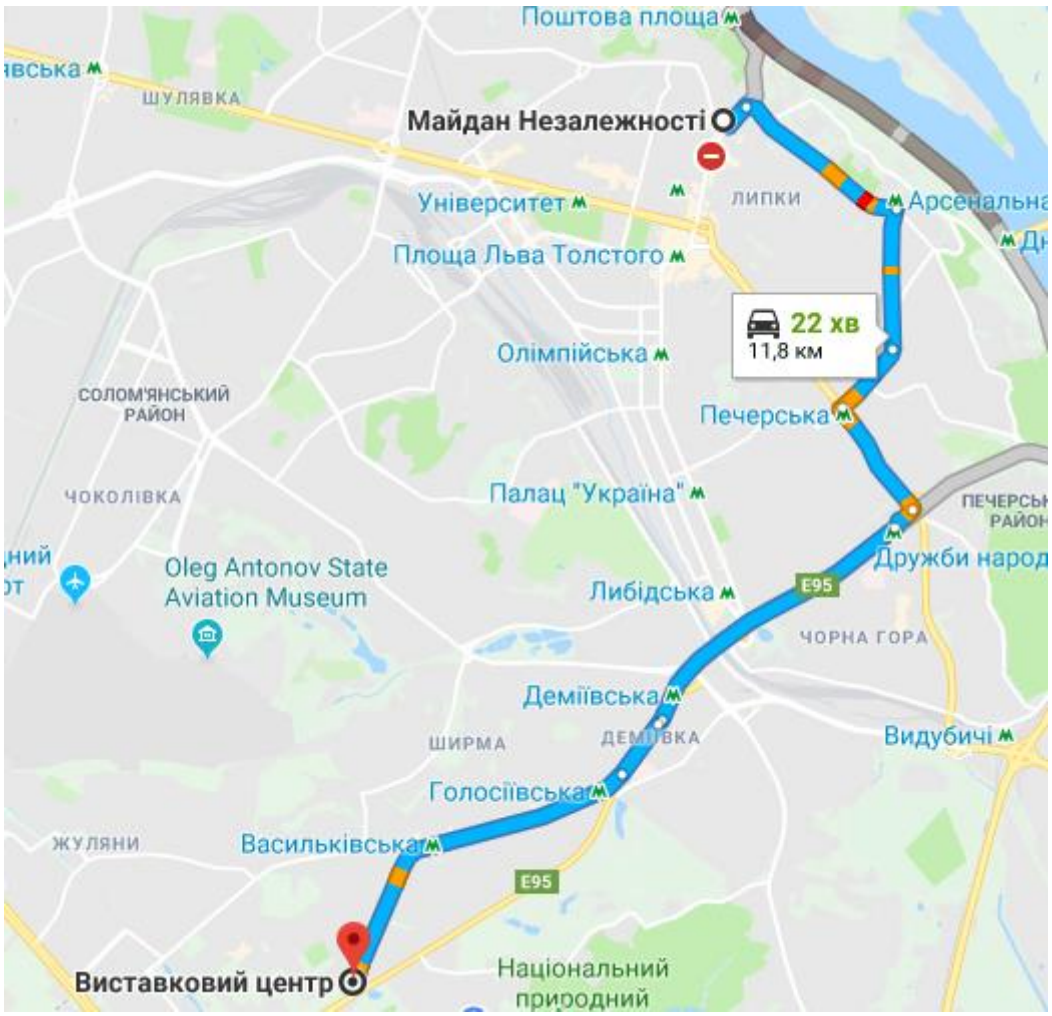


Рисунок 4.20 - Маршрут №5 – Google maps

Дистанція – 11,8 км.

Даний маршрут є досить складним, оскільки і починається, і закінчується не на основних дорогах. Результат роботи алгоритмів на ньому може вважатися показовим. Ми бачимо, що алгоритми A* показують дуже погані результати, а CH показує стабільний час і дуже добру відстань.

В порівнянні з Google maps, CH показав навіть кращий результат – майже на 2 км. коротший маршрут.

Маршрут №6. Маршрут від станції метро Берестейська до станції метро Позняки. Довжина 18,58 км.

Contraction hierarchies – рис 4.21:

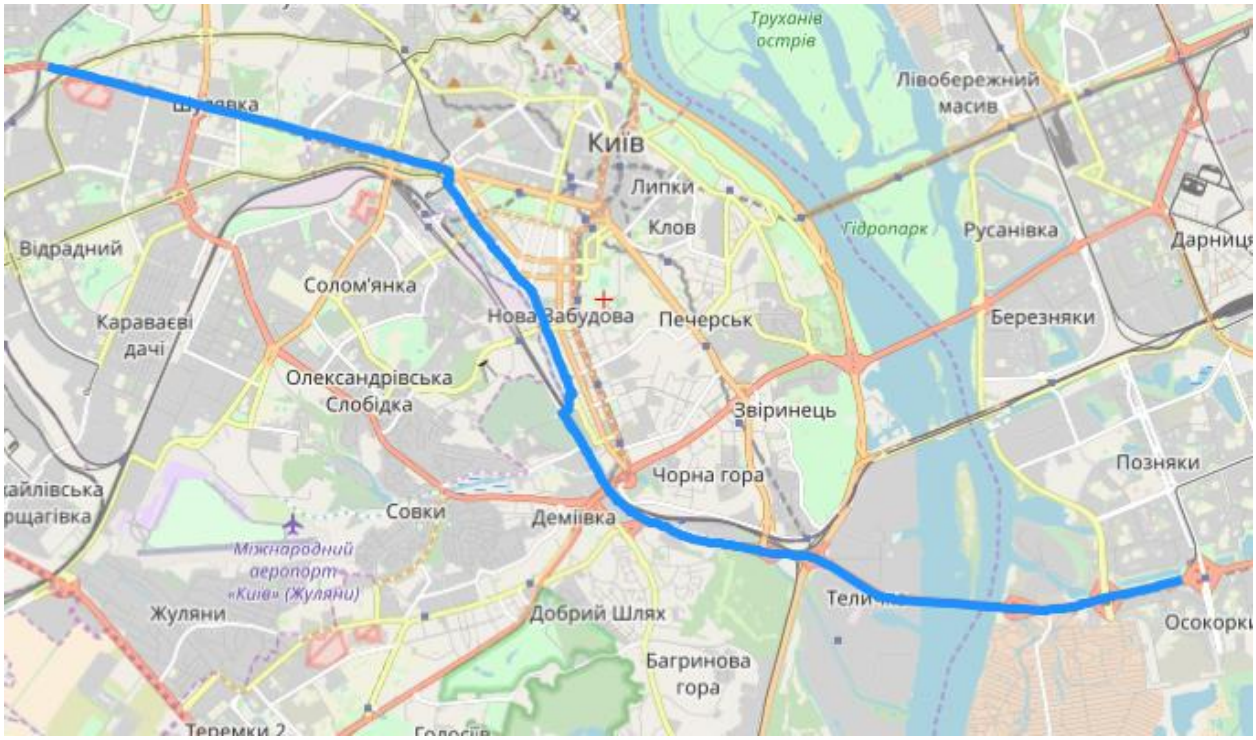


Рисунок 4.21 - Маршрут №6 – Contraction hierarchies

Час пошуку – 2,33 с.

Дистанція – 18,58 км.

A* - рис. 4.22:

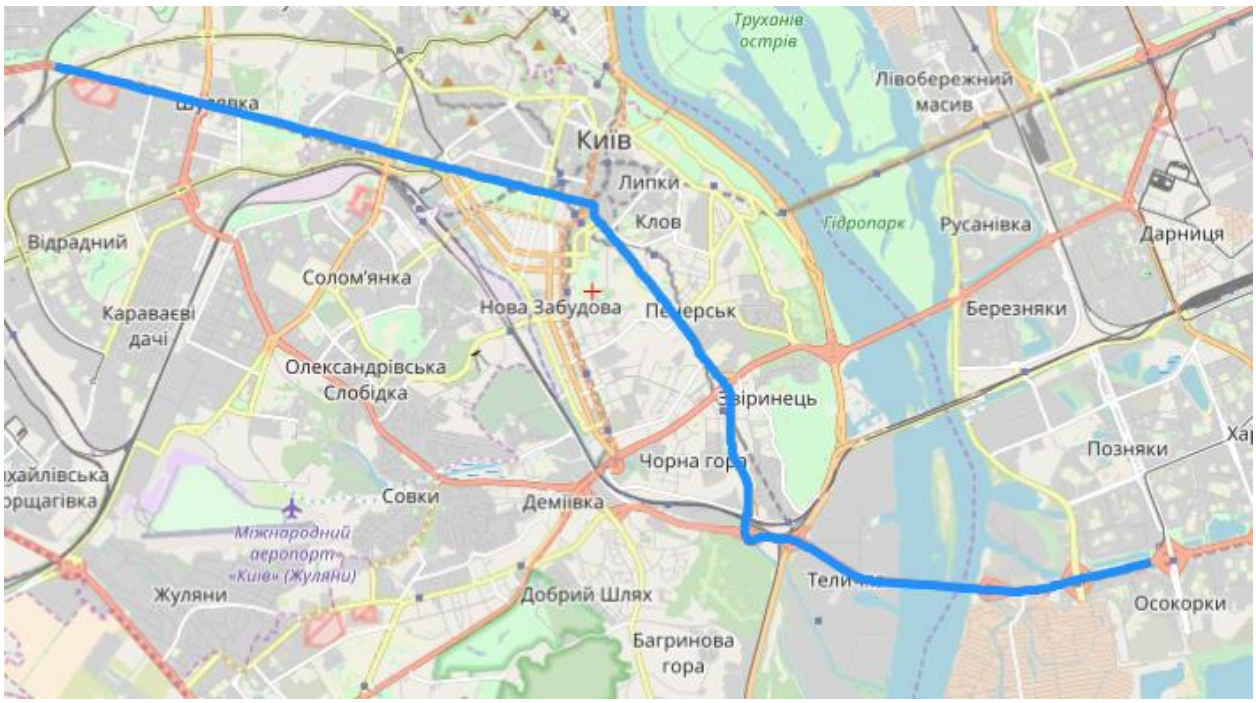


Рисунок 4.22 - Маршрут №6 – A*

Час пошуку – 3,36 с.

Дистанція – 18,71 км.

A* двонаправлений – рис. 4.23:

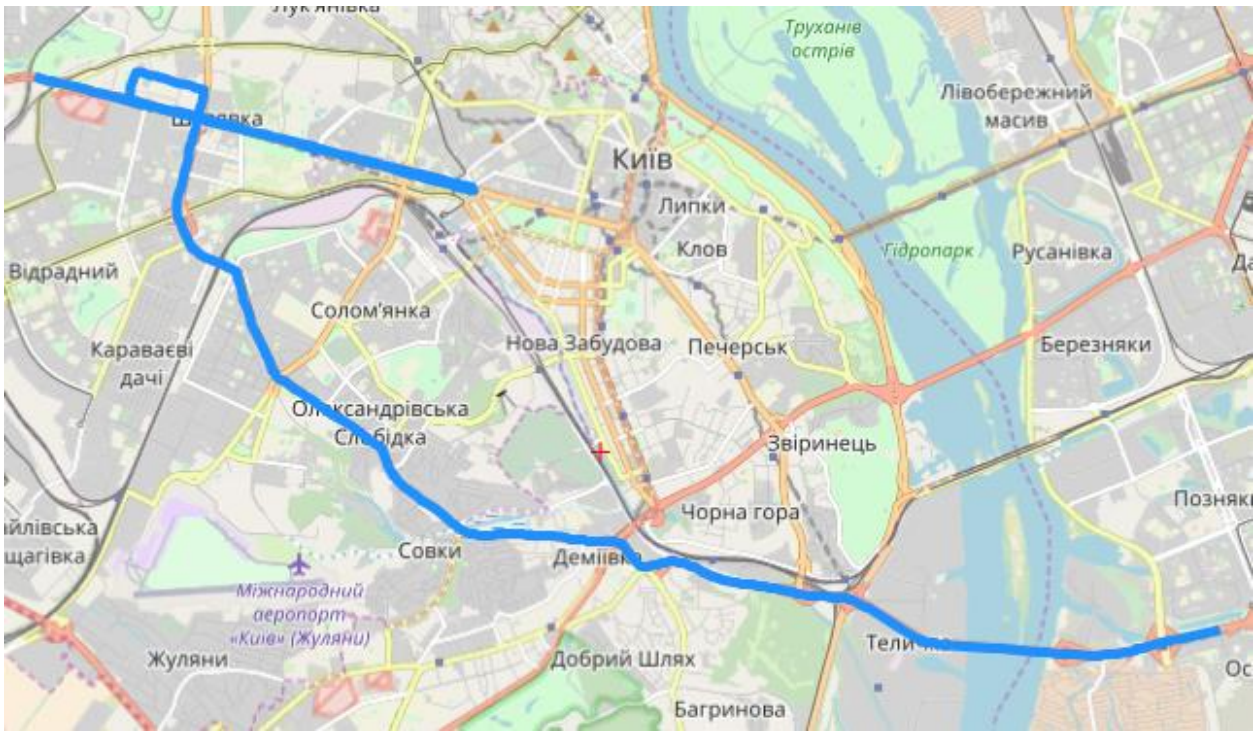


Рисунок 4.23 - Маршрут №6 – A* двонаправлений

Час пошуку – 36,89 с.

Дистанція – 28,48 км.

Результати Google maps – рис. 4.24:

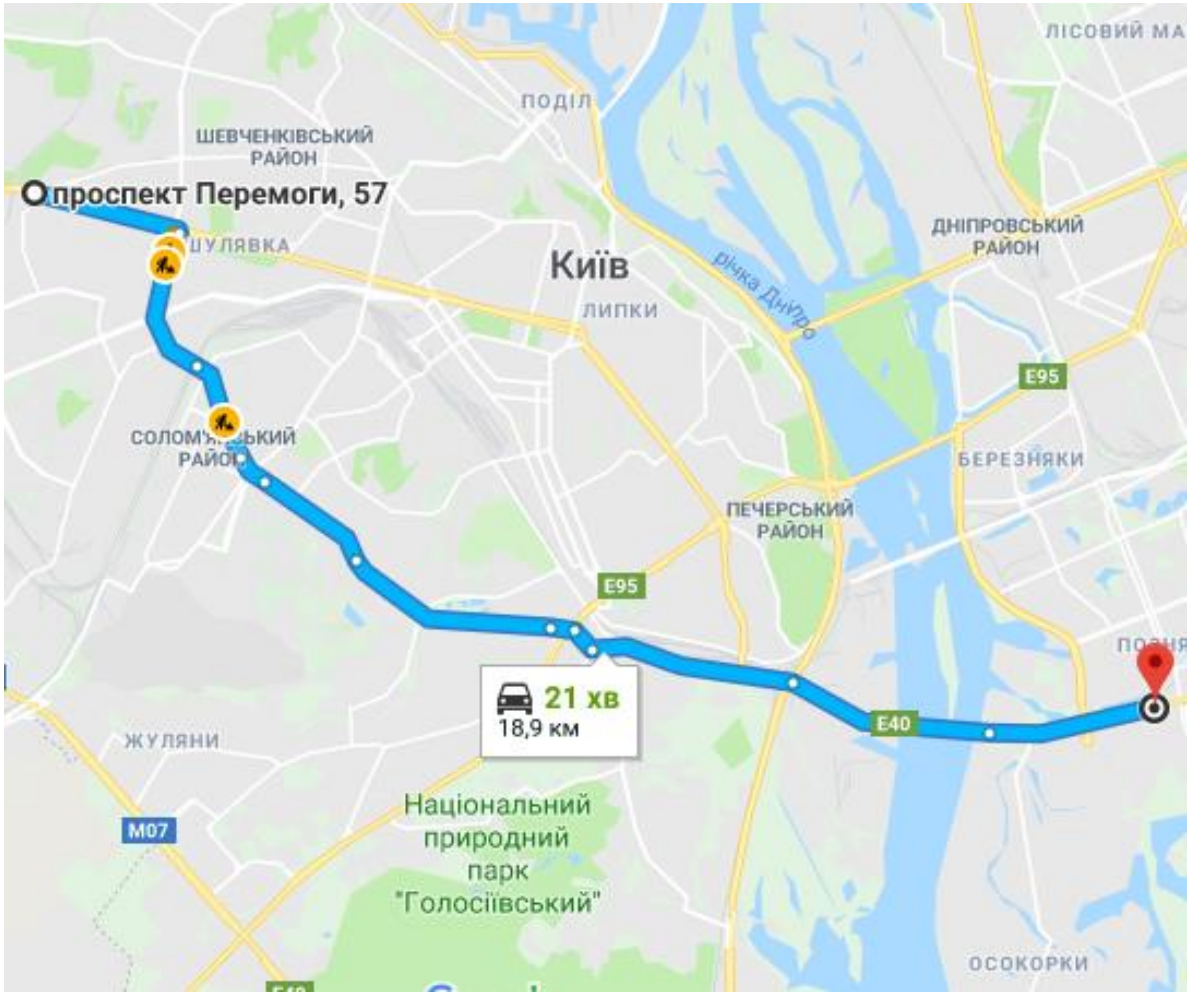


Рисунок 4.24 - Маршрут №6 – Google maps

Дистанція – 20,4 км.

Маршрут є найбільшим з розглянутих. CH показує стабільний хороший результат з дистанцією навіть меншою ніж Google maps і з хорошим часом пошуку. A* в даному випадку теж добре себе проявив, показавши нормальний час і припустимий маршрут.

Для зручності аналізу наведемо порівняльну таблицю результатів – таблиця 4.1.

Таблиця 4.1 Результати обробки маршрутів

№	Алгоритм	Час пошуку, с	Довжина, км	Кращий результат
1	Contraction hierarchies	1.23	2.19	A*
	A*	0.25	2.19	
	A* двонаправлений	0.31	2.19	
	Google maps	-	2.19	
2	Contraction hierarchies	1.14	3.33	CH
	A*	0.98	4.12	
	A* двонаправлений	1.04	4.48	
	Google maps	-	2.8	
3	Contraction hierarchies	0.96	2.23	CH
	A*	6.18	5.51	
	A* двонаправлений	16.52	4.84	
	Google maps	-	1.9	
4	Contraction hierarchies	1.48	8.78	A* двонаправлений
	A*	2.37	9.4	
	A* двонаправлений	1.17	8.78	
	Google maps	-	8.6	
5	Contraction hierarchies	2.62	9.96	CH
	A*	11.78	15.62	
	A* двонаправлений	82.82	23.07	
	Google maps	-	11.8	
6	Contraction hierarchies	2.33	18.58	CH
	A*	3.36	18.71	
	A* двонаправлений	36.89	28.48	
	Google maps	-	20.4	

Висновки за розділом 4

В розділі було розглянуто результати обробки бази даних та роботи реалізованих алгоритмів пошуку.

Тривалість створення графу доріг для Києва є прийнятною – 33 хвилини. Це операція, яка виконується лише раз, тому цей час можна вважати невеликим.

Тривалість обробки графу в алгоритмі Contraction hierarchies є надзвичайно складною в плані обчислень операцією і триває дуже довго. Для неповної карти Києва обробка виконувалася більше 35 годин і це дуже багато,

враховуючи те, що параметри обробки були налаштовані більше на швидкість обробки, а не на якість отриманого графу. Проте, навіть цієї якості виявилося достатньо для нормальної роботи пошукового алгоритму. У зв’язку з такою триваліcтю роботи алгоритму вкрай важливою є його технічна оптимізація. В неї входить як правильне написання коду, так і використання всієї потужності обчислювальної машини. В розробленому продукті ще є відкриті можливості для оптимізації процедури обробки.

В розділі також було розглянуто роботу пошукових алгоритмів: A*, A* двонаправлений та Contraction hierarchies. З шести розглянутих маршрутів по одному разу алгоритми A* та A* двонаправлений виявилися кращими, в інших чотирьох випадках алгоритм Contraction hierarchies був кращим.

Очевидно, вибірка результатів є не досить великою, проте вже з неї ми можемо зробити певні висновки.

По-перше, у випадках коли Contraction hierarchies виявлявся гіршим, різниця була малою, і результати отримані при його роботі були майже такими ж якісними як і в алгоритму, що був кращим.

По-друге, час роботи алгоритмів A* та A* двонаправлений росте зі збільшенням довжини маршруту, а в алгоритму Contraction hierarchies час роботи майже константний, оскільки згідно з ним завжди обробляються всі можливі вершини для вершин початку і кінця. Це є великою перевагою алгоритму.

Технічна реалізація алгоритму Contraction hierarchies також не є оптимальною і є можливості для його прискорення на відміну від A* та A* двонаправленого, які є значно простішими в реалізації.

Відмітимо також, що швидкість пошуку маршруту залежить від якості обробленої бази даних, а в нашому випадку якість її обробки, як вже було сказано, не є дуже високою. Тому витративши додатковий час на обробку ми зможемо отримати алгоритм, який працює ще швидше.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

5.1 Опис ідеї проекту (товару, послуги, технології)

Опис ідеї стартап-проекту представлено в таблиці 5.1.

Таблиця 5.1 - Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
В подорожах туристам часто бракує мобільного зв'язку та інтернету до якого всі звикли. Однією з головних незручностей є складність комунікації на відстані. У випадку коли одна людина з групи загубилася в незнайомому місті і може стати в нагоді даний програмний продукт. Суть його полягає у відображенні на карті мобільного пристрою групи людей у реальному часі. Завдяки цьому кожен з групи зможе орієнтуватись в розташуванні решти.	1. Навігація	Швидкий пошук маршруту
	2. Орієнтування в незнайомому місті	Можливість знайти групу туристів в незнайомому місті
	3. Пошук друзів на карті	

Визначення сильних, слабких та нейтральних характеристик ідеї проекту представлено в таблиці 5.2.

Таблиця 5.2 - Сильні, слабкі та нейтральні характеристики ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(Потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтр. стор.)	S (сильна сторона)
		TeamMap	Google	Maps.me			
1.	Навігація	+	+	+		+	
2.	Можливість бачити інших користувачів на карті	+	-	-			+
3.	Відображення ситуації на дорогах	-	+	-	+		
4.	Відомість бренду	-	+	+	+		
5.	Офлайн навігація	+	-	+			+

5.2 Технологічний аудит ідеї проекту

Технологічна здійсненність ідеї проекту – таблиця 5.3.

Таблиця 5.3 - Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Навігація	OpenStreetMap; Xamarin	Наявні. Необхідна розробка з використанням відповідних бібліотек.	Доступні
2	Відображення кількох користувачів	Xamarin; OsmSharp	-/-	Доступні
3	Офлайн навігація	Xamarin;	-/-	Доступні
Обрана технологія реалізації ідеї проекту: Проект буде реалізований на мові програмування C# з використанням технології мобільної розробки Xamarin, відкритого картографічного ресурсу OpenStreetMap та відповідних бібліотек для його використання.				

5.2 Аналіз ринкових можливостей запуску стартап-проекту

Попередня характеристика потенційного ринку стартап-проекту представлена в таблиці 5.4.

Таблиця 5.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	3
2	Загальний обсяг продаж, грн/ум.од	100 000 (1000 од./ 100грн)
3	Динаміка ринку (якісна оцінка)	Зростає

Продовження таблиці 5.4

№ п/п	Показники стану ринку (найменування)	Характеристика
4	Наявність обмежень для входу (вказати характер обмежень)	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі (або по ринку), %	20%

Характеристика потенційних клієнтів стартап-проекту представлена в таблиці 5.5.

Таблиця 5.5 - Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Побудова маршрутів для групи людей.	Туристи, жителі великих міст	Туристами можуть бути люди будь-якого віку. Необхідно враховувати цей фактор при проектуванні інтерфейсу, щоб програма була зрозумілою для всіх. Загалом поведінка користувачів у контексті використання програми практично не буде відрізнятися.	User-friendly інтерфейс; Швидкодія; Можливість працювати без інтернет з’єднання.

Фактори загроз представлені в таблиці 5.6.

Таблиця 5.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Подібний функціонал з’явиться у більш відомих конкурентів	Люди будуть схильні користуватися вже відомою програмою, а не пробувати щось нове.	Активніша маркетингова кампанія

Фактори можливостей представлені в таблиці 5.7.

Таблиця 5.7 - Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Переклад на додаткові мови	Збільшення можливої кількості споживачів	-
2	Реалізація додатку на інших мобільних платформах	Збільшення можливої кількості споживачів	-

Ступеневий аналіз конкуренції на ринку представлений в таблиці 5.8.

Таблиця 5.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - монополія/олігополія/ монополістична/чиста	Олігополія. Є невелика кількість популярних програм з подібним функціоналом.	Необхідно, щоб у нас був особливий функціонал, який зможе зацікавити споживачів.
2. За рівнем конкурентної боротьби - локальний/національний/…	Глобальний. Люди по всьому світу можуть користуватися будь-якими мобільними додатками.	Позитивно вплинути можна лише перекладом додатка на різні мови. Але на початкових стадіях це не передбачається
3. За галузевою ознакою - міжгалузева/ внутрішньогалузева	Внутрішньогалузева.	-

Продовження таблиці 5.8

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
4. Конкуренція за видами товарів: - товарно-родова - товарно-видова - між бажаннями	Між бажаннями.	Необхідно випереджувати конкурентів у реалізації бажань клієнтів. Постійно моніторити відгуки користувачів та робити оновлення функціоналу.
5. За характером конкурентних переваг - цінова / нецінова	Нецінова. Перевагою в кожного Виробника є особливості функціоналу.	-
6. За інтенсивністю - марочна/не марочна	Не марочна	Торгова марка слабко впливає на позицію на ринку

Аналіз конкуренції в галузі за М. Портером – таблиці 5.9.

Таблиця 5.9 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Google maps Maps.me Yandex maps	Інформація відсутня	-	-	-
Висновки:	Інтенсивність боротьби з існуючими конкурентами достатньо висока. Але ніяких перешкод виходу на ринок немає	Виграти конкуренцію можна тільки за рахунок переваг у функціоналі	-	-	-

Обґрунтування факторів конкурентоспроможності наведено в таблиці 5.10.

Таблиця 5.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Інновації	Програмний продукт матиме функціонал, якого на даний момент немає в жодного з конкурентів
2	Цінова політика	На початкових етапах, а, можливо, і надалі функціонал буде безкоштовним.
3	Підтримка і оновлення	Додаток буде постійно оновлюватись для додавання нового функціоналу та згідно з побажаннями користувачів.

Порівняльний аналіз сильних та слабких сторін наведено в таблиці 5.11.

Таблиця 5.11 - Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з Google						
			–3	–2	–1	0	+1	+2	+3
1	Інновації	15					+		
2	Цінова політика	19				+			
3	Підтримка і оновлення	18					+		

SWOT-аналіз стартап-проекту – таблиця 5.12.

Таблиця 5.12 - SWOT-аналіз стартап-проекту

Сильні сторони: новий функціонал, цінова політика	Слабкі сторони: мала відомість
Можливості: постійне розширення функціоналу	Загрози: повільна робота продукту, поява схожого функціоналу на ринку

Альтернативи ринкового впровадження стартап-проекту розглянуто в таблиці 5.13.

Таблиця 5.13 - Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Переклад на додаткові мови	Низька	3-6 місяці
2	Реалізація додатка на різних мобільних платформах	Висока	6-12 місяців

5.4 Розроблення ринкової стратегії проекту

Вибір цільових груп потенційних споживачів наведено в таблиці 5.14.

Таблиця 5.14 - Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Жителі міст	Для даної цільової групи продукт буде схожим на продукти конкурентів, тому перехід на використання нашого додатку ускладнений.	Попит низький.	Висока інтенсивність конкуренції.	Вхід в сегмент повільний і достатньо складний.
2	Молоді туристи	Для цієї групи наш додаток матиме необхідний функціонал відмінний від конкурентів. До того ж молодь легко сприймає щось нове. Тому цей сегмент готовий до використання продукту	Високий попит завдяки особливостям функціоналу та його унікальності.	Інтенсивність конкуренції помірна.	Вхід в сегмент помірно складний

Продовження таблиці 5.14.

3	Люди старшого віку.	Дана група специфічна тим, що старші люди більш консервативні і, в контексті туризму, рідше подорожують великими групами, тому зникає необхідність використання особливого функціоналу, а консерватизм ще більше ускладнює ситуацію.	Низький попит	Інтенсивність конкуренції помірна	Вхід в сегмент складний
4	Мисливці, рибалки, грибники	Для людей цього сегменту програма буде допомагати не розгубитися. Ситуація схожа з туристичними групами.	Високий попит, проте група специфічна і кількість людей в групі невелика.	Конкуренція низька.	Вхід в сегмент простий.
Які цільові групи обрано: 1 – наймасовіша група, 2, 4					

Визначення базової стратегії розвитку наведено в таблиці 5.15.

Таблиця 5.15 - Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
1	Розроблення нових функціональних можливостей	За рахунок функціоналу, який не можуть надати конкуренти	Групова навігація	

Визначення базової стратегії конкурентної поведінки наведено в таблиці 5.16.

Таблиця 5.16 - Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1	Ні. Але є унікальний функціонал.	Обидва варіанти.	Буде. Базова навігація теж буде доступна, як і в продуктах конкурентів.	

Визначення стратегії позиціонування наведено в таблиці 5.17.

Таблиця 5.17 - Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкуренто спроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Навігація, робота офлайн, навігація для групи людей	Реалізація поставлених цілей. Внесення змін згідно побажань користувачів, отримуючи фідбек	Робота офлайн, навігація для групи осіб	Навігатор, туризм карти

5.5 Розроблення маркетингової програми стартап-проекту

Визначення ключових переваг концепції потенційного товару – таблиці 5.18.

Таблиця 5.18 - Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Навігація	Забезпечує потребу	Переваг в даному контексті немає.
2	Робота офлайн	Забезпечує потребу	Перевага перед тими конкурентами, які не мають такого функціоналу
3	Навігація для групи людей	Забезпечує потребу	Перевага перед усіма конкурентами

Опис трьох рівнів моделі товару приведено в таблиці 5.19.

Таблиця 5.19 - Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>I. Товар за задумом</td><td colspan="3">Навігація, навігація для групи людей, робота офлайн, поради місць для відвідування</td></tr>
<tr><td rowspan="6">II. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>1. Швидкодія
2. Користувацький інтерфейс
3. Відмовостійкість
4. Підтримка</td><td>Нм
Нм
Нм
Нм</td><td>Тх/Тл/Е
Е
Тх/Тл
Тх/Е</td></tr>
<tr><td colspan="3">Якість: стандарти відсутні. Проект покрито автоматизованими тестами.</td></tr>
<tr><td colspan="3">Пакування: Play market, Apple store</td></tr>
<tr><td colspan="3">Марка: ТеаММаР</td></tr>
<tr><td rowspan="2">III. Товар із підкріпленням</td><td colspan="3">До продажу: Додаток який надає функціонал навігатора з особливими функціями, такими як: навігація для групи людей та рекомендації місць для відвідування.</td></tr>
<tr><td colspan="3">Після продажу: Швидкодія, зручний користувацький інтерфейс.</td></tr>
<tr><td colspan="4">Продукт буде з закритим вихідним кодом. Тобто готовий функціонал скопіювати не буде можливості, а захистити ідеї не вдасться.</td></tr>
</table>

Визначення меж встановлення ціни – таблиця 5.20.

Таблиця 5.20 - Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	-	-	-	Продукт безкоштовний. Дохід отримується з реклами.

Формування системи збуту наведено в таблиці 5.21.

Таблиця 5.21 - Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Клієнт завантажує безкоштовний додаток з Play market/Apple store	-	-	-

Концепція маркетингових комунікацій приведена в таблиці 5.22.

Таблиця 5.22 - Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	1 група – жителі міст	Реклама в соц. мережах, додатках пов’язаних з погодою	Карти офлайн.	Розповісти про існування додатку.	Навігація офлайн
2	2 група – молоді туристи. Молодь користуються великою кількістю інших мобільних додатків.	Рекламу можна додавати в інших додатках пов’язаних з туризмом, соц. мережах.	Карти офлайн, навігація для групи	Розповісти про існування додатку.	Навігатор для туристів; Навігація офлайн.

Продовження таблиці 5.22

3	3 група – мисливці, рибалки, грибники	Релама на форумах, соц мережах.	Навігація для групи людей. Карти офлайн.	Розповісти про існування додатку.	Навігація офлайн.

Висновки за розділом 5

Можемо зробити висновок про те, що проект цілком може бути комерційно успішним, оскільки ми знайшли досить великі потенційні групи клієнтів. Конкуренція на ринку існує, а, отже і існують додаткові складнощі при виході на ринок, але завдяки особливостям продукту їх можна подолати. Робимо висновок про те, що подальша імплементація проекту є доцільною. І в майбутньому необхідно обрати стратегію по додаванню особливого функціоналу, який відсутній на ринку для збільшення бази клієнтів та популяризації власного продукту.

ВИСНОВКИ ПО РОБОТІ

В даній роботі було розглянуто підходи до реалізації мультиагентних систем на прикладі реалізації системи маршрутизації за допомогою алгоритмів пошуку маршрутів в графах.

Розроблена система складається з двох агентів: агент обробки бази даних і агент маршрутизації та може бути розширена іншими агентами для збільшення функціональних можливостей програмного продукту.

Агент обробки бази даних відповідає за створення та обробку графу доріг, використовуючи картографічні дані отримані з сервісу OpenStreetMap.

Агент маршрутизації включає в себе реалізацію алгоритмів пошуку маршрутів: A*, A* двонаправлений та Contraction hierarchies.

Було проведено тестування розробленої системи та порівняння отриманих результатів з результатами сервісу Google maps. Найкращим виявився алгоритм Contraction hierarchies, показавши стабільні результати по часу та по якості побудованих маршрутів, як на малих дистанціях, так і на великих. Маршрути побудовані ним були близьким до отриманих в Google maps, а в деяких випадках навіть кращими.

Запропонований стартап-проект передбачає подальший розвиток продукту. На даному етапі створене ядро системи. Розвиток згідно зі стартап-проектом залежить від наявності необхідних ресурсів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Берж К. Теория графов и ее приложения / К. Берж. — М.: ИЛ, 1962. — 320 с.

2. Алексеев В.Е. Нахождения кратчайших путей в графе. / Алексеев В.Е., Таланов В.А. // В сб.: Графы. Модели вычислений. Структуры данных. — Нижний Новгород: Издательство Нижегородского гос. университета, - 2005. — с. 236-237.

3. Татт У. Теория графов / У. Татт — М.: Мир, 1988. — 424 с.

4. Свами М. Графы, сети и алгоритмы / Свами М., Тхуласираман К. — М: Мир, 1984. — 455 с.

5. Labeling Algorithm for Shortest Paths on Road Networks. / [Abraham I., Delling D., Goldberg A., Werneck R.]. - Philadelphia. - Symposium on Experimental Algorithms, 2011. — pp. 230-241.

6. Dijkstra E. A note on two problems in connexion with graphs. // In.: Numerische Mathematik. - 1959. - V. 1.— pp. 269-271.

7. Ford L. Flows in Networks / Ford L., Fulkerson D. — Princeton: Princeton University Press, 1962. — 253 p.

8. YouTube - Алгоритм поиска A* [Електронний ресурс]. – Режим доступу: https://www.youtube.com/watch?v=AsEC2TJZ3JY

9. Coursera - A* Algorithm [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-graphs/lecture/xXi6V/a-algorithm.

10. E-maxx - Алгоритм Флойда-Уоршелла [Електронний ресурс]. – Режим доступу http://e-maxx.ru/algo/floyd_warshall_algorithm.

11. Ананий В. Алгоритмы: введение в разработку и анализ. / Ананий В., Левитин А. // В сб.: Introduction to The Design and Analysis of Aigorithms. — М.: «Вильямс», 2006. — с. 212-215.

12. Moore E. The shortest path through a maze. / E. Moore. - In.: Proceedings of the International Symposium on the Theory of Switching. — Harvard University Press, 1959. — pp. 285–292.

13. MJT - Contraction hierarchies path finding algorithm [Електронний ресурс]. – Режим доступу: https://www.mjt.me.uk/posts/contraction-hierarchies/.

14. aivanoff.blogspot - Агенты и мультиагентные системы [Електронний ресурс]. – Режим доступу: http://aivanoff.blogspot.com/2007/12/blog-post_18.html.

15. Мультиагентные системы [Електронний ресурс]. – Режим доступу: https://intellect.ml/11-multiagentnye-sistemy-5354.

16. Intellect - Мультиагентные системы [Електронний ресурс]. – Режим доступу: https://intellect.ml/11-multiagentnye-sistemy-5354.

17. Wooldridge M. An Introduction to MultiAgent Systems. / M. Wooldridge — Liverpool: John Wiley & Sons, 2002. — p. 366

18. Livejournal - Построение маршрута с помощью алгоритма Дейкстры [Електронний ресурс]. – Режим доступу: https://efimmanevich.livejournal.com/732.html#comments.

19. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. / [R. Geisberger, P. Sanders, D. Schultes, D. Delling, C. C. McGeoch]. // Springer, 2008. - Vol. 5038. – pp. 319–333.

20. Coursera - Contraction hierarchies – preprocessing [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-graphs/lecture/AWFGa/preprocessing.

21. Coursera - Contraction hierarchies – Witness search [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-graphs/lecture/WuGFB/witness-search.

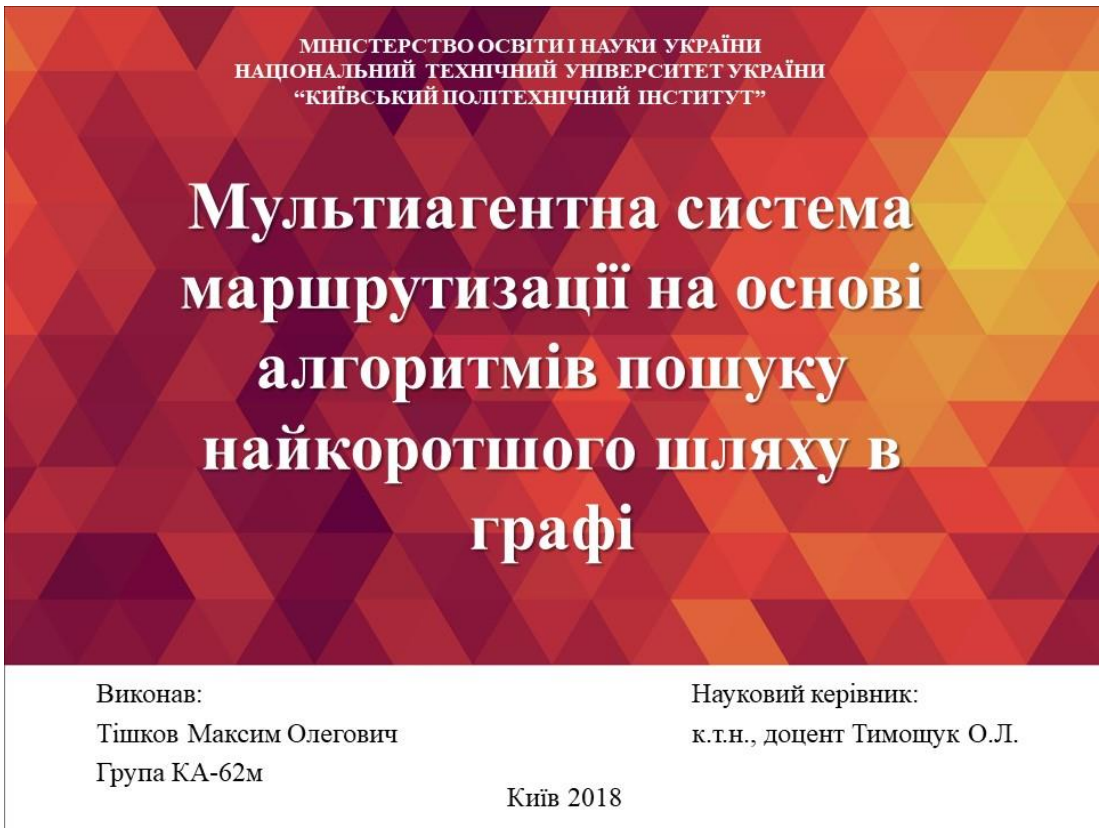
22. Coursera - Contraction hierarchies Query [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-raphs/lecture/XafED/query.

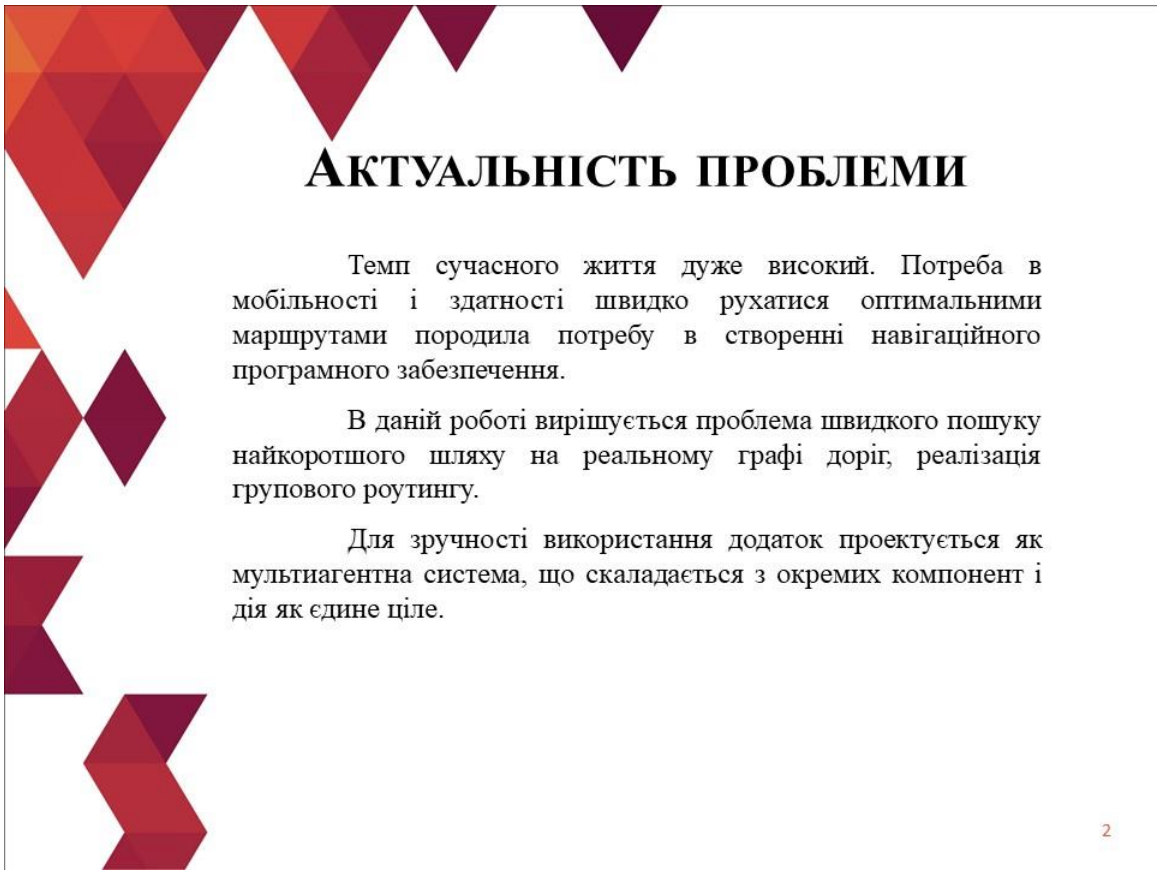
23. Coursera - Contraction hierarchies – Proof of correctness [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-graphs/lecture/uYeBL/proof-of-correctness.

24. Coursera - Contraction hierarchies – Node ordering [Електронний ресурс]. – Режим доступу: https://www.coursera.org/learn/algorithms-on-graphs/lecture/Abyzw/node-ordering.

25. Microsoft Development Network – Windows Presentation Foundation [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx.

ДОДАТОК А ІЛЮСТРАЦІЙНА МАТЕРІАЛИ





Об’єкт, предмет та мета дослідження

- Мета дослідження: розробка мультиагентної системи для пошуку найкоротших маршрутів в графі між двома точками та від кількох точок до центру;
- Об’єкт дослідження: побудова мультиагентної системи маршрутизації на основі алгоритмів пошуку маршруту;
- Предмет дослідження: алгоритми пошуку маршрутів в графі.

3

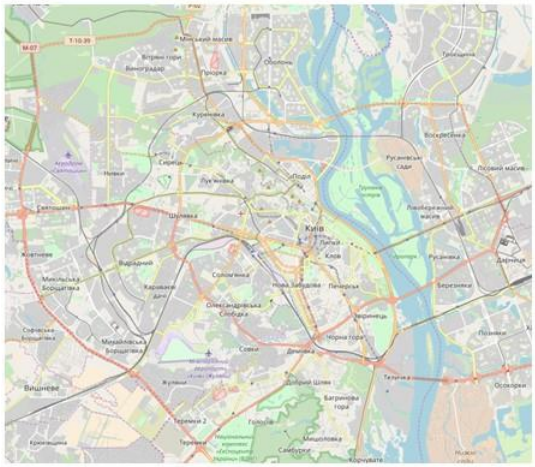
Постановка задачі

- Виконати огляд методів пошуку найкоротшого маршруту в графі;
- Спроектувати мультиагентну систему маршрутизації, що складатиметься з компонента створення графу доріг, компонента передобробки графу та компонента пошуку маршруту;
- Виконати порівняння швидкодії різних алгоритмів та ресурсів необхідних для їхнього функціонування;
- Виконати аналіз отриманих результатів.

4

Створення графу доріг

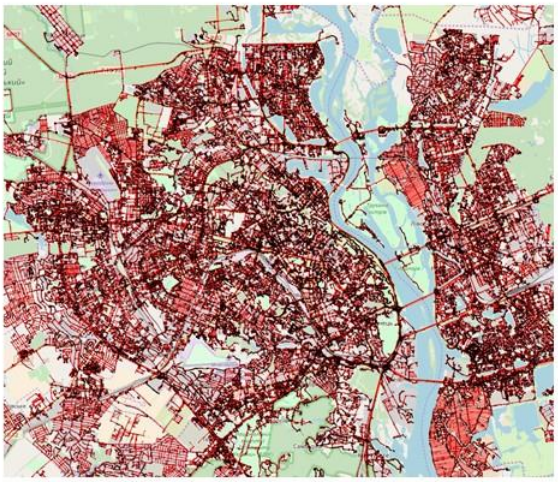
В межах даної роботи використовуються відкриті картографічні дані з сервісу OpenStreetMap. Тестовий регіон – місто Київ.



5

Створення графу доріг

Сервіс OpenStreetMap дозволяє завантажити картографічні дані про будь-який регіон світу. Проте в вихідному вигляді ці дані непридатні для використання, адже вони містять велику кількість надлишкової інформації для роутинга. Тому перед використанням їх необхідно обробити і зберегти у вигляді графу доріг.



В нашому випадку це виконується збереження вершин, що містять географічні координати та ребер, що мають вагу до бази даних MS SQL. В результаті ми отримуємо Зважений, орієнтований граф.

6

Алгоритми пошуку маршруту. Алгоритм Дейкстри

Алгоритм голландського вченого Едсгер Дейкстри знаходить всі найкоротші шляхи з однієї наперед заданій вершини графа до всіх інших.

Для програмної реалізації алгоритму знадобитися два масиви:відкритий і закритий. Відкритий містить вершини графа до яких ми вже знаємо як дістатися, закритий вершини, які вже оброблені і найкоротша відстань до них вже відома.

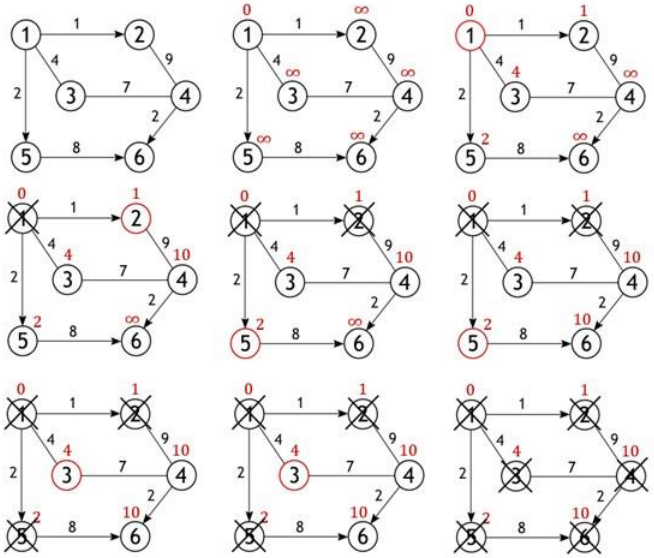
В масивах зберігається поточне значення відстані до вершини і посилання на вершину з якої ми прийшли. Якщо при обробці іншої вершини відстань виявляється меншою ніж вже відома, дані переписуються.

Вершини обробляються по черзі для поточної вершини. Спочатку обробюється вершина v_1 і переноситься у закритй список. Потім обирається наступна активна вершина з відкритого списку і обробюється. Спочатку обробляться всі вершини до яких ми можемо потрапити з вершини v_1.

Даний алгоритм є простим в реалізації і завжди знаходить найкоротшу відстань. Але для реальних графів доріг, що містять мільйони вершин працює він занадто повільно, оскільки перебирає дуже багато зайвих точок.

7

Алгоритми пошуку маршруту. Алгоритм Дейкстри



8

Алгоритми пошуку маршруту. Алгоритм A*

Алгоритм A* є модифікацією алгоритма Дейкстри. Модифікація полягає у виборі вершини, що буде оброблюватись.

Порядок обходу вершин визначається евристичної функцією «відстань + вартість». Ця функція - сума двох інших: функції вартості досягнення розглянутої вершини (x) з початкової вершини (зазвичай позначається як g(x)) і евристичної оцінки відстані від розглянутої вершини до кінцевої (позначається як h (x)).

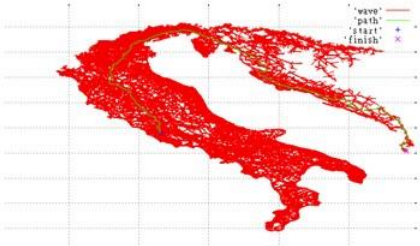
Функція h (x) повинна бути припустимою евристичної оцінкою, тобто не повинна переоцінювати відстані до цільової вершині. Прикладом евристики може слугувати географічна відстань між точками.

9

Алгоритми пошуку маршруту. Порівняння A* і Дейкстри

Обидва алгоритми оброблюють велику кількість даних, тому працюють недостатньо швидко. На рисунках можемо побачити кількість вершин і ребер оброблених алгоритмами при пошку маршруту з Італії в Албанію.

Алгоритм Дейкстри



Алгоритм A*

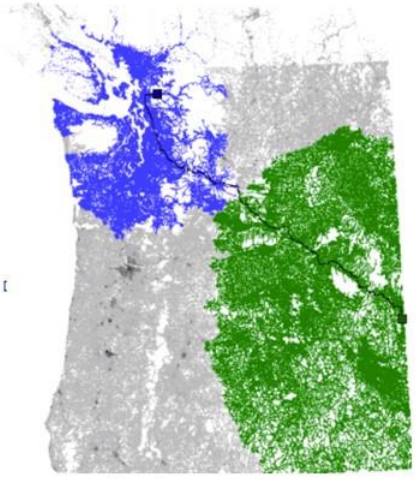
Даний випадок є незручним для A* і звичаної евристики відстані, оскільки маршрут не проходить по прямій і точки спочатку віддаляються від кінцевої точки.

Але незважаючи на це, алгоритм A* все ж обробляє суттєво менше вершин ніж Дейкстри

10

Алгоритми пошуку маршруту. Двонаправлені алгоритми

Двонаправлені алгоритми відрізняються від однонаправлених тим, що запускається дві процедури пошуку: з початкової і з кінцевої точок. При чому кроки обох процедур виконуються почергово і у випадку алгоритмів Дейкстри і A* зупинка настає тоді коли в закриті списки обох процедур потрапляє одна і та ж вершина. Після цього шлях прямої процедури об'єднується зі оберненим шляхом зворотної процедури



Хвилі двонапрвленого алгоритму Дейкстри

11

Алгоритми пошуку маршруту. Contraction hierarchies

Contraction hierarchies - алгоритмом переобробки графа. Тобто завдяки цьому алгоритму ми отримаємо оптимізований граф для пошуку маршруту.

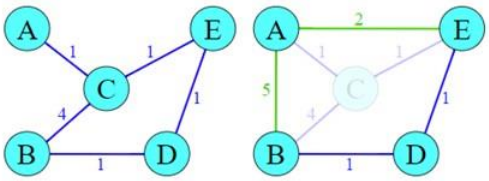
Реалізація алгоритму складається з 3 трьої етапів:

- Сортування вершин для обробки
- Обробка вершин
  - Перевірка пордяку вершини
  - Визначення всіх можливих шляхів скорочення
  - Фільтрація за допомогою Witness search
- Реалізація двонаправленого алгоритму Дейкстри

12

Contraction hierarchies. Обробка вершини

1. Обробка вершини полягає в пошуку ребер скорочення – додаткових ребер, що з'єднані через поточну вершину, і мають довжину, що дорівнює сумі довжин ребер з яких воно утворюється.



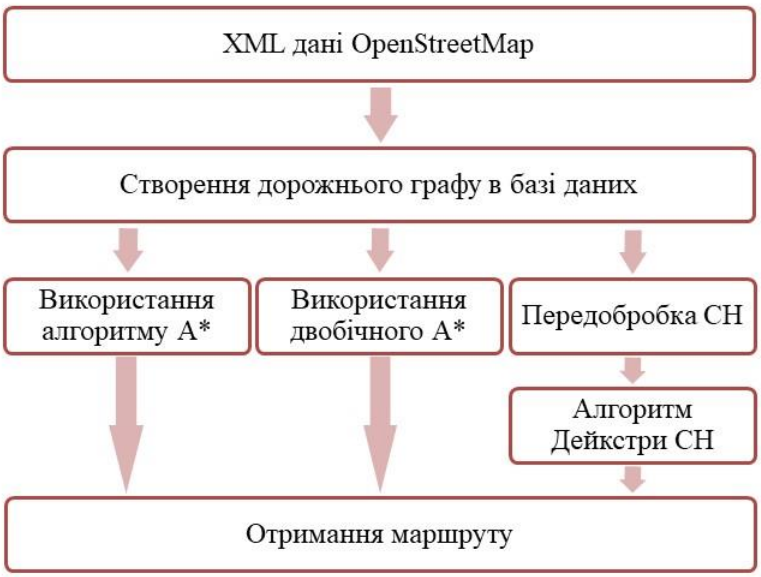
Додавання ребер скорочення

2. Вершину для обробки ми вибираємо з черги. Але після обробки однієї вершини, позиція іншої в черзі може змінитися. Тому для вибору вирішини для обробки спочатку проводиться перерахунок порядку.

3. Знайдені ребра скорочення слід додавати лише в тому випадку, якщо шлях між знайденими вершинами, що проходить через додане ребро буде мінімальним. Для перевірки цього для кожного ребра скорочення запускається процедура Witness search – звичайний алгоритм Дейкстри, оптимізований під дану задачу.

13

Функціональна схема програми

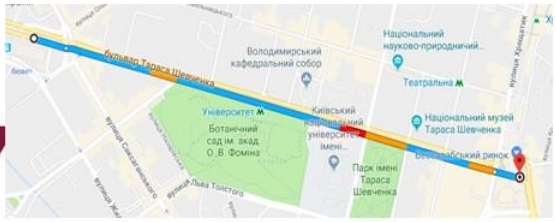


14

Результати роботи

Маршрут №1. Прямий маршрут по проспекту Перемоги – від цирку до Бесарабською площі. Довжина 2,19 км.

A*, A* довнаправлений, Contraction hierarchies



Google maps

CH:

Час пошуку – 1,23 с.

Дистанція – 2,19 км.

A*:

Час пошуку – 0,25 с.

Дистанція 2,19 км.

A* двонаправлений:

Час пошуку – 0,31 с.

Дистанція 2,19 км.

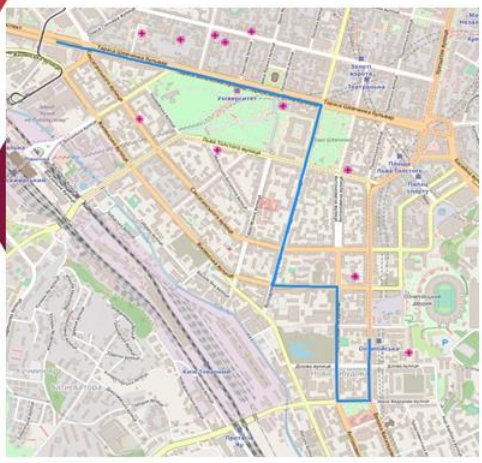
Google maps:

Дистанція – 2,2 км.

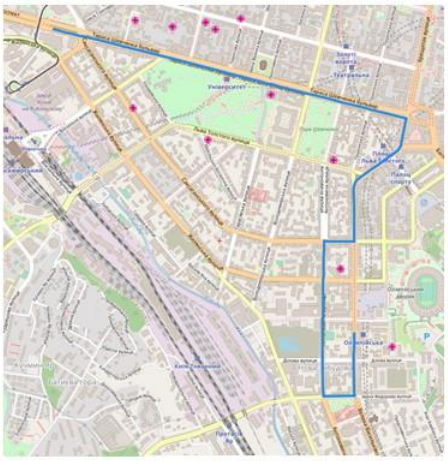
15

Результати роботи

Маршрут №2. Маршрут від цирку до станції метро Олімпійська. Довжина 3,33 км.



A*

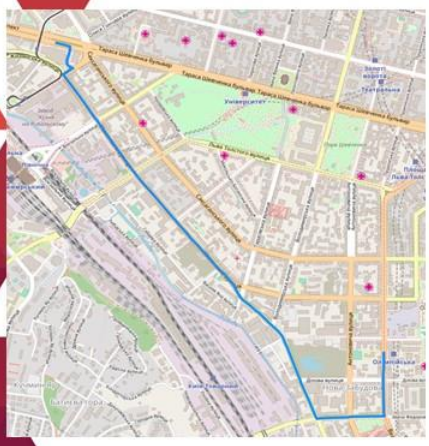


A* Двонаправлений

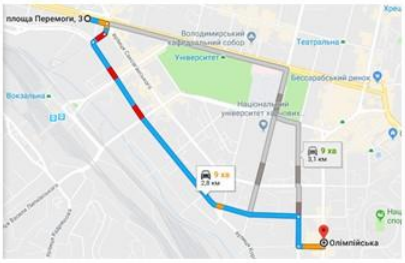
16

Результати роботи

Маршрут №2. Маршрут від цирку до станції метро Олімпійська. Довжина 3,33 км.



Contraction hierarchies



Google maps

CH:

Час пошуку – 1,14 с.

Дистанція – 3,33 км.

A*:

Час пошуку – 0,98 с.

Дистанція – 4,12 км.

A* двонаправлений:

Час пошуку – 1,04 с.

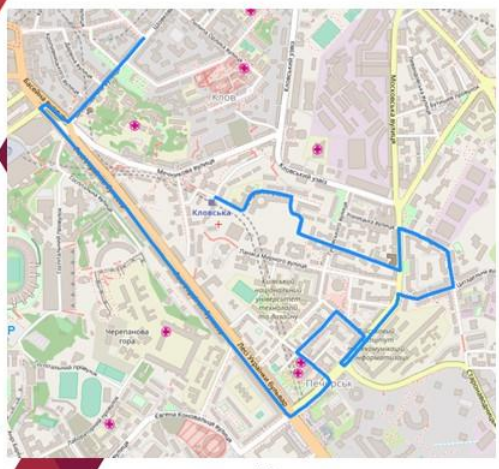
Дистанція 4,48 км.

Google maps:

Дистанція – 2,8 км.

Результати роботи

Маршрут №3. Маршрут від перетину вулиць Лютеранської та Шовковичної до станції метро Кловська. Довжина – 2,23 км.

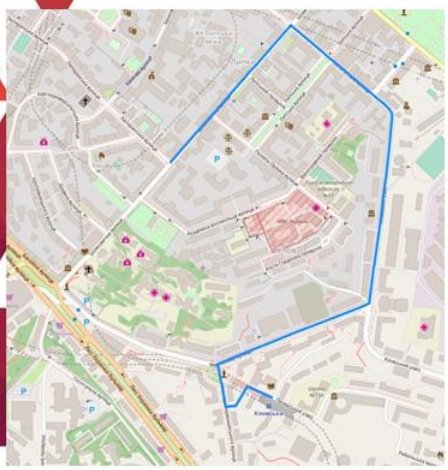


A*

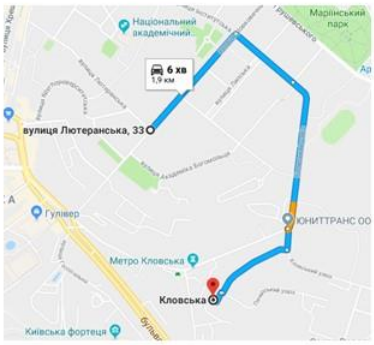
A* Двонаправлений

Результати роботи

Маршрут №3. Маршрут від перетину вулиць Лютеранської та Шовковичної до станції метро Кловська. Довжина – 2,23 км.



Contraction hierarchies



Google maps

CH:

Час пошуку – 0,96 с.

Дистанція – 2,23 км.

A*:

Час пошуку – 6,18 с.

Дистанція – 5,51 км.

A* двонаправлений:

Час пошуку – 16,52 с.

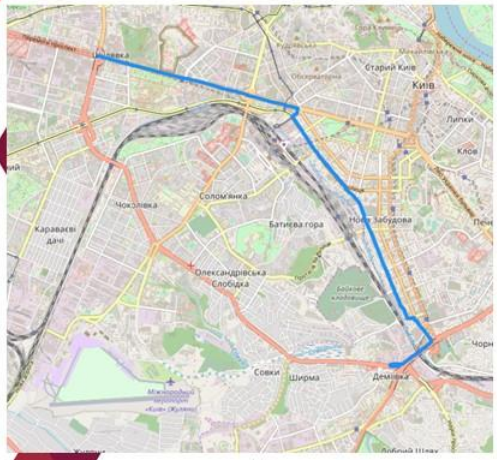
Дистанція 4,84 км.

Google maps:

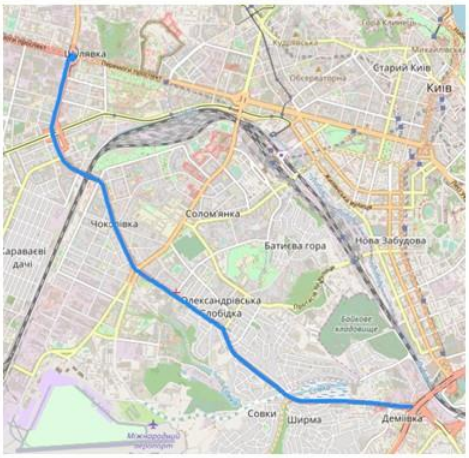
Дистанція – 1.9 км.

Результати роботи

Маршрут №4. від станції метро Шулявська до станції метро Деміївська. Довжина – 8,6 км.



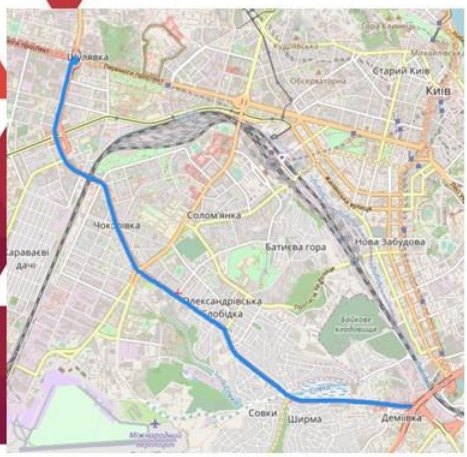
A*



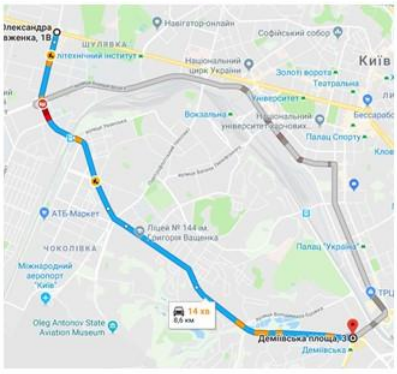
A* Двонаправлений

Результати роботи

Маршрут №4. від станції метро Шулявська до станції метро Деміївська. Довжина – 8,6 км.



Contraction hierarchies



Google maps

CH:

Час пошуку – 1,48 с.

Дистанція – 8,78 км.

A*:

Час пошуку – 2,37 с.

Дистанція – 9,4 км.

A* двонаправлений:

Час пошуку – 1,17 с.

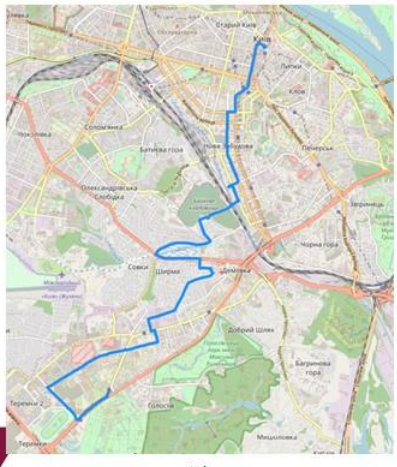
Дистанція 8,78 км.

Google maps:

Дистанція - 8,6 км.

Результати роботи

Маршрут №5. Маршрут від Майдану Незалежності до Виставкового центру. Довжина - 9,96 км.

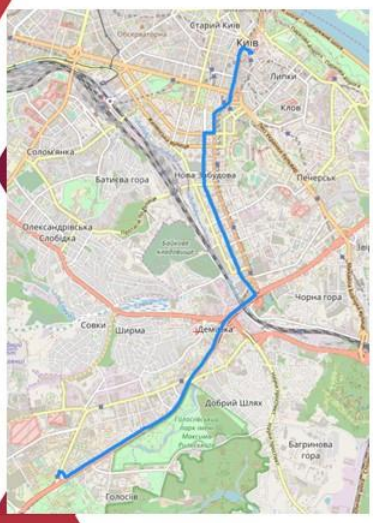


A*

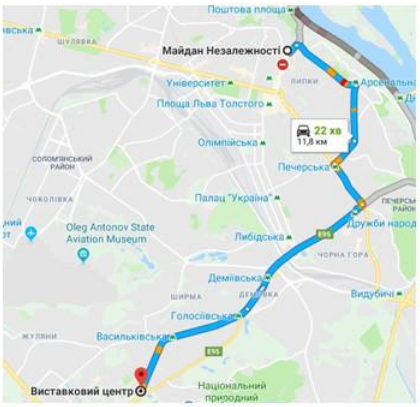
A* Двонаправлений

Результати роботи

Маршрут №5. Маршрут від Майдану Незалежності до Виставкового центру. Довжина - 9,96 км.



Contraction hierarchies



Google maps

CH:

Час пошуку – 2,62 с.

Дистанція – 9,96 км.

A*:

Час пошуку – 11,78 с.

Дистанція – 15,62 км.

A* двонаправлений:

Час пошуку – 82,82 с.

Дистанція 23,07 км.

Google maps:

Дистанція – 11.8 км.

23

Результати роботи

Порівняльна таблиця результатів роботи:

№	Алгоритм	Час пошуку, с	Довжина, км	Кращий результат
1	Contraction hierarchies	1.23	2.19	A*
	A*	0.25	2.19	
	A* двонаправлений	0.31	2.19	
	Google maps	-	2.19	
2	Contraction hierarchies	1.14	3.33	CH
	A*	0.98	4.12	
	A* двонаправлений	1.04	4.48	
	Google maps	-	2.8	
3	Contraction hierarchies	0.96	2.23	CH
	A*	6.18	5.51	
	A* двонаправлений	16.52	4.84	
	Google maps	-	1.9	
4	Contraction hierarchies	1.48	8.78	A* двонаправлений
	A*	2.37	9.4	
	A* двонаправлений	1.17	8.78	
	Google maps	-	8.6	
5	Contraction hierarchies	2.62	9.96	CH
	A*	11.78	15.62	
	A* двонаправлений	82.82	23.07	
	Google maps	-	11.8	

24

Висновки по роботі

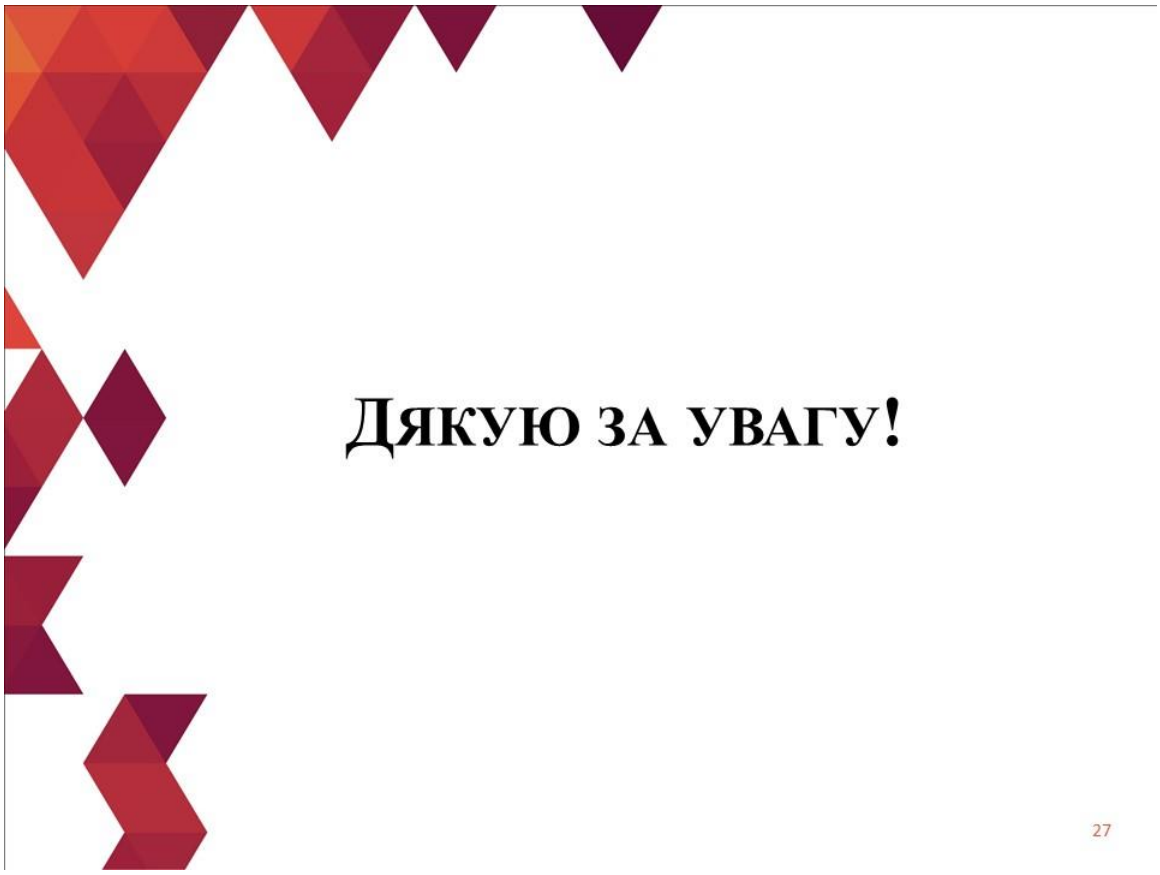
- Розглянуто та реалізовано класичні алгоритми пошуку маршрутів та високоефективний алгоритм Contraction Hierarchies
- Побудовано мультиагентну систему маршрутизації та створення графу доріг придатного для побудови маршрутів на реальних дорогах
- Порівняно ефективність алгоритмів пошуку маршруту

25

Практична цінність

- Побудована система може бути застосована як база для створення мобільних додатків. Може бути портована на мобільні платформи, чи виступати веб сервером для мобільних пристроїв;
- Реалізовані алгоритми з прийнятною швидкістю знаходять найкоротші маршрути на карті Києва, що дозволяє застосовувати їх в повсякденному житті

26


ДЯКУЮ ЗА УВАГУ!
27

ДОДАТОК Б ЛІСТИНГ ПРОГРАМИ

OsmToMsSql

CHHelper.cs:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Diagnostics;
using System.Linq;
using System.Linq.Expressions;
using System.Text;
using System.Threading.Tasks;
using OsmToMsSql.Models;
using TeamMapCore;

namespace OsmToMsSql
{
    public class CHHelper
    {
        private readonly TeamMapEntities db;
        private List<OrderedNode> nodes;
        private List<OrderedNode> processedNodes = new List<OrderedNode>();
        private List<Direction> directions = new List<Direction>();
        private List<Direction> shortcutsToAdd= new List<Direction>();
        private bool useMemory => directions.Count > 0;
        private int witnesSearchHopsLimit = 4;
        public CHHelper(TeamMapEntities db)
        {
            this.db = db;
        }
        public void ContractionHierarchies()
        {
            directions = db.Directions.ToList();
            //directions.ForEach(x =>
            //{
            //    x.Node1.Order = 0;
            //    x.Node.Order = 0;
            //});
            nodes = GetOrderedNodes();
            //test - uncomment
            foreach (var node in nodes)
            {
                db.Nodes.FirstOrDefault(x => x.Id == node.Id).Order = node.Importance;
            }
            db.SaveChanges();

            Stopwatch watch = Stopwatch.StartNew();
            int order = 1;
            while (nodes.Count > 0)
            {
                OrderedNode node = GetNodeToContract();
                ProcessNode(node, order);
                processedNodes.Add(node);
                //RecalculateNeighborsImportance(node.Id);
                order++;
            }
            watch.Stop();
            try
            {
                foreach (var node in db.Nodes.ToList())
                {
```

```
                            try
                            {
                                node.Order = processedNodes.FirstOrDefault(x => x.Id == node.Id)?.Order
?? 0;
                            }
                            catch (Exception e)
                            {
                                string error = e.Message;
                            }
                        }
                        if (useMemory)
                        {
                            shortcutsToAdd = shortcutsToAdd.Select(x => new Direction
                            {
                                Id = x.Id,
                                TypeId = x.TypeId,
                                StartNodeId = x.StartNodeId,
                                EndNodeId = x.EndNodeId,
                                IsOneWay = x.IsOneWay,
                                WayId = x.WayId,
                                Length = x.Length,
                                InputDirectionId = x.InputDirectionId,
                                OutputDirectionId = x.OutputDirectionId
                            }).ToList();

                            while (shortcutsToAdd.Count > 0)
                            {
                                int n = Math.Min(100000, shortcutsToAdd.Count);
                                db.Directions.AddRange(shortcutsToAdd.GetRange(0, n));
                                db.SaveChanges();
                                shortcutsToAdd.RemoveRange(0, n);
                            }
                        }
                    }
                    catch(Exception e)
                    {
                        string error = e.Message;
                    }

                    if (useMemory)
                    {
                        db.SaveChanges();
                    }
                }

                private List<OrderedNode> GetOrderedNodes()
                {
                    Stopwatch watch = Stopwatch.StartNew();
                    int i = 0;
                    nodes = db.Nodes.ToList().Select(x => new OrderedNode(x)).ToList();
                    Parallel.ForEach(nodes, node =>
                    {
                        CalculateNodeImportance(node);
                        i++;
                    });
                    watch.Stop();

                    foreach (var dbNode in db.Nodes)
                    {
                        dbNode.Order = nodes.FirstOrDefault(x => x.Id == dbNode.Id).Importance;
                    }
                    db.SaveChanges();
                    return nodes.OrderBy(x => x.Importance).ToList();

                    //test
                    //List<OrderedNode>      nodes      =      db.Nodes.ToList().Select(x      =>      new
OrderedNode(x)).ToList();
                    //foreach (var node in nodes)
                    //{
                    //    node.EdgeDifference = node.Order;
                    //    node.Order = 0;
                    //}
                    //return nodes.OrderBy(x => x.Importance).ToList();
                }
```

```
        private OrderedNode GetNodeToContract()
        {
            nodes = nodes.OrderBy(x => x.Importance).ToList();

            OrderedNode node;
            while (true)
            {
                node = nodes[0];
                nodes.RemoveAt(0);
                CalculateNodeImportance(node);
                if (nodes.Count == 0 || node.Importance <= nodes.Min(x => x.Importance))
                {
                    break;
                }

                int i = 0;
                while (i < nodes.Count && nodes[i].Importance < node.Importance)
                {
                    i++;
                }

                nodes.Insert(i, node);
            }

            return node;

            //OrderedNode node = nodes[0];
            //nodes.RemoveAt(0);
            //return node;
        }

        private void ProcessNode(OrderedNode node, int order)
        {
            List<Direction> shortcuts = node.Shortcuts;//GetShortcuts(node.Id);
            node.Order = order;
            //db.Nodes.Single(x => x.Id == node.Id).Order = order;
            if (shortcuts.Count > 0)
            {
                long maxId = directions.Max(x => x.Id);
                int i = 1;
                foreach (var shortcut in shortcuts)
                {
                    shortcut.Id = maxId + i++;
                }
                if (useMemory)
                {
                    shortcutsToAdd.AddRange(shortcuts);
                    directions.AddRange(shortcuts);
                }
                else
                {
                    db.Directions.AddRange(shortcuts);
                }
            }
            List<long> neighborsIds = GetNodeNeighbors(node.Id).Select(x => x.Id).ToList();
            List<OrderedNode> neighbors = nodes.Where(x => neighborsIds.Contains(x.Id)).ToList();
            foreach (var n in neighbors)
            {
                n.NodeLevel = Math.Max(n.NodeLevel, node.NodeLevel + 1);
            }
            if (!useMemory)
            {
                db.SaveChanges();
            }

        }

        private List<Direction> GetShortcuts(long nodeId)
        {
            IEnumerable<long> nodesIds = nodes.Select(x => x.Id);
            List<Direction> edges = (useMemory ? directions.AsQueryable() : db.Directions).Where(x
=>
```

```
                            ((x.StartNodeId == nodeId && nodesIds.Contains(x.EndNodeId)) || (x.EndNodeId
== nodeId && nodesIds.Contains(x.StartNodeId)))).ToList();
                        List<Direction> inputEdges = edges.Where(x => x.EndNodeId == nodeId || (x.StartNodeId
== nodeId && !x.IsOneWay)).ToList();
                        List<Direction> outputEdges = edges.Where(x => x.StartNodeId == nodeId || (x.EndNodeId
== nodeId && !x.IsOneWay)).ToList();
                        List<Direction> shortcuts = new List<Direction>();

                        foreach (var inputEdge in inputEdges)
                        {
                            foreach (var outputEdge in outputEdges.Where(x => x.Id != inputEdge.Id))
                            {
                                shortcuts.Add(new Direction()
                                {
                                    TypeId = 13,
                                    StartNodeId = inputEdge.StartNodeId == nodeId ? inputEdge.EndNodeId :
inputEdge.StartNodeId,
                                    EndNodeId = outputEdge.EndNodeId == nodeId ? outputEdge.StartNodeId :
outputEdge.EndNodeId,
                                    IsOneWay = true,
                                    WayId = 0,
                                    Length = inputEdge.Length + outputEdge.Length,
                                    InputDirectionId = inputEdge.Id,
                                    OutputDirectionId = outputEdge.Id
                                });
                            }
                        }
                        shortcuts = shortcuts.Where(x => x.StartNodeId != x.EndNodeId).ToList();
                        shortcuts = FilterByWitnessPaths(nodeId, shortcuts);
                        foreach (var shortcut in shortcuts)
                        {
                            shortcut.Node1 = nodes.FirstOrDefault(x => x.Id == shortcut.StartNodeId);
                            shortcut.Node = nodes.FirstOrDefault(x => x.Id == shortcut.EndNodeId);
                        }
                        return shortcuts;
                    }


                    /// <summary>
                    /// return List of end node ids
                    /// </summary>
                    /// <param name="proceeingNodeId"></param>
                    /// <param name="startNodeId"></param>
                    /// <param name="endNodeIds"></param>
                    /// <param name="shortcutLength"></param>
                    /// <returns></returns>
                    private        List<Direction>        FilterByWitnessPaths(long        processingNodeId,
IEnumerable<Direction> shortcuts)
                    {
                        List<Direction> resultShortcuts = new List<Direction>();
                        IEnumerable<long> startNodeIds = shortcuts.Select(x => x.StartNodeId).Distinct();
                        foreach (var startNodeId in startNodeIds)
                        {
                            IEnumerable<Direction> currentShortcuts =
                                shortcuts.Where(x => x.StartNodeId == startNodeId);
                            double maxWeight = currentShortcuts.Max(x => x.Length);
                            IEnumerable<long> endNodeIds = currentShortcuts.Select(x => x.EndNodeId);

                            List<DijkstraNode> openList = new List<DijkstraNode>(){ new DijkstraNode() { Id =
startNodeId, Weight = 0 } };
                            List<DijkstraNode> closeList = new List<DijkstraNode>();
                            while (openList.Count > 0)
                            {
                                DijkstraNode      currentNode      =      openList.OrderByDescending(x      =>
x.Weight).FirstOrDefault();
                                List<DijkstraNode> visibleFromCurrent;
                                try
                                {
                                    List<Direction> tempDirections = (useMemory ? directions.AsQueryable() :
db.Directions).Where(
                                        x =>
                                            //nodes.Count == 0 || nodes.Count(y => y.Id == x.StartNodeId ||
y.Id == x.EndNodeId) == 2 &&
```

```
                                    (x.StartNodeId == currentNode.Id || (x.EndNodeId == currentNode.Id
&& !x.IsOneWay)) &&
                                    x.EndNodeId    !=    processingNodeId    &&    x.StartNodeId    !=
processingNodeId).ToList();


                                visibleFromCurrent   =   currentNode.Hops   <   witnesSearchHopsLimit   ?
tempDirections
                                    .Select(x => new DijkstraNode()
                                    {
                                        Id = x.EndNodeId == currentNode.Id ? x.StartNodeId : x.EndNodeId,
                                        Weight = currentNode.Weight + x.Length,
                                        Hops = currentNode.Hops + 1
                                    })
                                    .Where(x => !closeList.Select(y => y.Id).Contains(x.Id))
                                    .OrderBy(x => x.Weight).ToList() : new List<DijkstraNode>();
                            }
                            catch (Exception e)
                            {
                                List<Direction> temp = directions.Where(x => x.Node == null || x.Node1 ==
null).ToList();
                                openList.RemoveAt(0);
                                continue;
                            }
                            foreach (var node in visibleFromCurrent.Where(x => x.Weight <= maxWeight))
                            {
                                IEnumerable<long> openIds = openList.Select(x => x.Id);
                                if (!openIds.Contains(node.Id))
                                {
                                    openList.Add(node);
                                } else
                                {
                                    DijkstraNode  openedNode  =  openList.FirstOrDefault(x  =>  x.Id  ==
node.Id);
                                    if (openedNode.Weight > node.Weight)
                                    {
                                        openedNode.Weight = node.Weight;
                                    }
                                }
                            }
                            closeList.Add(currentNode);
                            openList.Remove(currentNode);

                            if (endNodeIds.All(x => closeList.Select(y => y.Id).Contains(x)))
                            {
                                //List<long> allIds = closeList.Select(x => x.Id).ToList();
                                //allIds.Add(processingNodeId);
                                //string allIdsString = string.Join(", ", allIds),
                                //    endIdsString = string.Join(", ", endNodeIds);
                                break;
                            }
                        }
                        resultShortcuts.AddRange(currentShortcuts.Where(x =>
                            closeList.All(y => y.Id != x.EndNodeId) ||
                            closeList.FirstOrDefault(y => y.Id == x.EndNodeId).Weight > x.Length));
                        if (resultShortcuts.Count != 0)
                        {
                            //List < long > allIds = closeList.Select(x => x.Id).ToList();
                            //allIds.Add(processingNodeId);
                            //string allIdsString = string.Join(", ", allIds),
                            //    endIdsString = string.Join(", ", endNodeIds);
                            var a = 0;
                        }
                    }

                    return resultShortcuts;
                }

                public void CalculateNodeImportance(OrderedNode node)
                {
                    CalculateEdgeDifferenceAndShortcutCover(node,  out  var  edgeDifference,  out  var
shortcutCover);
                    node.EdgeDifference = edgeDifference;
                    node.ShortcutCover = shortcutCover;
```

```
                    node.ContractedNeighbors = CalculateContractedNeighbors(node.Id);
                    if (node.NodeLevel > 0)
                    {
                        var a = 0;
                    }
                    //Node level calculated after contracting the node
                }

                protected void CalculateEdgeDifferenceAndShortcutCover(OrderedNode node, out int
edgeDifference, out int shortcutCover)
                {
                    IEnumerable<Direction> connectedEdges = (useMemory ? directions.AsQueryable():
db.Directions).Where(x => x.StartNodeId == node.Id || x.EndNodeId == node.Id);
                    int inputOutputEdges = connectedEdges.Sum(x => x.IsOneWay ? 1 : 2);
                    node.Shortcuts = GetShortcuts(node.Id);
                    edgeDifference = node.Shortcuts.Count - inputOutputEdges;
                    List<long> shortcutedNodeIds = node.Shortcuts.Select(x => x.StartNodeId).ToList();
                    shortcutedNodeIds.AddRange(node.Shortcuts.Select(x => x.EndNodeId).ToList());
                    shortcutCover = shortcutedNodeIds.Distinct().Count();
                }

                protected void RecalculateNeighborsImportance(long nodeId)
                {
                    List<OrderedNode> neighbors = nodes.Where(x => GetNodeNeighbors(nodeId).Select(y =>
y.Id).ToList().Contains(x.Id)).ToList();
                    foreach (var neighbor in neighbors)
                    {
                        CalculateNodeImportance(neighbor);
                    }
                }

                protected int CalculateContractedNeighbors(long nodeId)
                {
                    int cn = GetNodeNeighbors(nodeId).Count(x => x.Order != 0);
                    return cn;
                }

                protected IQueryable<Node> GetNodeNeighbors(long nodeId)
                {
                    return (useMemory ? directions.AsQueryable() : db.Directions).Where(x => x.StartNodeId
== nodeId || x.EndNodeId == nodeId)
                        .Select(x => x.Node.Id == nodeId ? x.Node1 : x.Node);
                }
            }
        }
```

TeamMapService

IPathFinder.cs

```
using System;
using System.Collections.Generic;
using GMap.NET;
using TeamMapCore;

namespace WpfOsmSharp.PathFinders
{
    public interface IPathFinder
    {
        Tuple<List<PointLatLng>, double> RouteFinding(Node start, Node end);

        List<PointLatLng> GetLastUsedPoints();
    }
}
```

BasePathFinder.cs

```
using GMap.NET;
using System.Collections.Generic;
using System.Linq;
```

```
using OsmToMsSql.Models;
using TeamMapCore;
using WpfOsmSharp.Models;

namespace WpfOsmSharp.PathFinders
{
    public class BasePathFinder
    {
        protected TeamMapEntities db;
        protected IList<Node> dbNodes;
        protected IList<Direction> dbDirections;
        protected List<PointLatLng> LastUsedPoints { get; set; } = new List<PointLatLng>();

        public BasePathFinder(TeamMapEntities db)
        {
            this.db = db;
            dbNodes = db.Nodes.ToList();
        }

        protected virtual List<PointLatLng> GetPath(AStarNode node, List<PointLatLng> path = null)
        {
            if (path == null)
            {
                path = new List<PointLatLng>();
            }
            path.Add(new PointLatLng(node.Latitude, node.Longitude));
            if (node.PreviousNode == null)
            {
                path.Reverse();
                return path;
            }
            return GetPath(node.PreviousNode, path);
        }

        protected List<DijkstraNode> GetPath(DijkstraNode node, List<DijkstraNode> path = null)
        {
            if (path == null)
            {
                path = new List<DijkstraNode> { node };
            }
            else
            {
                path.Add(node);
            }
            if (node.ParentNode != null)
            {
                return GetPath(node.ParentNode, path);
            }
            path.Reverse();
            return path;
        }

        public virtual List<PointLatLng> GetLastUsedPoints()
        {
            return LastUsedPoints;
        }
    }
}
```

AStarPathFidner.cs:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using GMap.NET;
using TeamMapCore;
using WpfOsmSharp.Models;

namespace WpfOsmSharp.PathFinders
{
    public class AStarPathFinder: BasePathFinder, IPathFinder
    {
```

```
        public AStarPathFinder(TeamMapEntities db, List<Direction> directions) : base(db)
        {
            dbDirections = directions;
        }

        public AStarPathFinder(TeamMapEntities db) : base(db)
        {
            dbDirections = db.Directions.Where(x => x.TypeId != 13).ToList();
        }


        public Tuple<List<PointLatLng>, double> RouteFinding(Node startNode, Node endNode)
        {
            Stopwatch watch = Stopwatch.StartNew();
            AStarNode watchingNode = new AStarNode(startNode),
                endWatchingNode = new AStarNode(endNode);
            List<AStarNode> openList = new List<AStarNode>
                {
                    watchingNode
                },
                closedList = new List<AStarNode>();
            watchingNode.CalculateH(endWatchingNode);
            while (watchingNode.H > 0)
            {
                List<Direction> directions = dbDirections
                    .Where(x => x.StartNodeId == watchingNode.Id || (x.EndNodeId == watchingNode.Id &&
!x.IsOneWay)).ToList().Where(x => closedList.All(y => y.Id != x.Node.Id)).ToList();
                if (directions.Count != 0)
                {
                    List<AStarNode> neighbours = directions
                        .Select(x => new AStarNode(watchingNode, x, endWatchingNode)).ToList();


                    foreach (var neighbour in neighbours)
                    {
                        AStarNode existNode = openList.FirstOrDefault(x => x.Id == neighbour.Id);
                        if (existNode != null)
                        {
                            if (existNode.ShortestPathLength > neighbour.ShortestPathLength)
                            {
                                existNode.ShortestPathLength = neighbour.ShortestPathLength;
                                existNode.PreviousNode = watchingNode;
                            }
                        }
                        else
                        {
                            openList.Add(neighbour);
                        }
                    }
                }


                openList.Remove(watchingNode);
                closedList.Add(watchingNode);
                if (openList.Count == 0)
                {
                    break;
                }
                AStarNode newWatchingNode = openList.Aggregate((node1, node2) =>
                    //((0.3 * node1.ShortestPathLength + node1.H) / node1.DirectionCoeficient <
                    // (0.3 * node2.ShortestPathLength + node2.H) / node2.DirectionCoeficient)
                    node1.H < node2.H
                        ? node1
                        : node2);

                watchingNode = newWatchingNode;
            }
            LastUsedPoints = closedList.Select(x => x.ToPointLatLng()).ToList();
            List<PointLatLng> path = GetPath(watchingNode);
            watch.Stop();
            return new Tuple<List<PointLatLng>, double>(path, watchingNode.ShortestPathLength);
        }
    }
```

```
        }
```

BiAStarPathFinder.cs:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using GMap.NET;
using WpfOsmSharp.Models;
using TeamMapCore;

namespace WpfOsmSharp.PathFinders
{
    public class BiAStarPathFinder : BasePathFinder, IPathFinder
    {
        readonly object synch = new object();

        protected List<AStarNode> ForwardCloseList { get; set; }
        protected List<AStarNode> BackCloseList { get; set; }

        public BiAStarPathFinder(TeamMapEntities db, List<Direction> directions) : base(db)
        {
            dbDirections = directions;
        }

        public BiAStarPathFinder(TeamMapEntities db) : base(db)
        {
            dbDirections = db.Directions.Where(x => x.TypeId != 13).ToList();
        }

        public Tuple<List<PointLatLng>, double> RouteFinding(Node startNode, Node endNode)
        {
            Stopwatch watch = Stopwatch.StartNew();
            ForwardCloseList = new List<AStarNode>();
            BackCloseList = new List<AStarNode>();
            LastUsedPoints = new List<PointLatLng>();
            CancellationTokenSource source = new CancellationTokenSource();
            CancellationToken cancellationToken = source.Token;
            Task[] tasks =
            {
                Task.Factory.StartNew(() => BackSearch(startNode, endNode, source), cancellationToken),
                Task.Factory.StartNew(() => ForwardSearch(startNode, endNode, source), cancellationToken)
            };

            Task.WaitAll(tasks);

            AStarNode forwardIntersection, backIntersection;
            if (ForwardCloseList.Intersect(BackCloseList, out forwardIntersection, out backIntersection))
            {
                List<PointLatLng> path = new List<PointLatLng>();
                path.AddRange(GetPath(forwardIntersection));
                var backPath = GetPath(backIntersection);
                backPath.Reverse();
                backPath.RemoveAt(0);
                path.AddRange(backPath);
                LastUsedPoints.AddRange(ForwardCloseList.Select(x => x.ToPointLatLng()));
                LastUsedPoints.AddRange(BackCloseList.Select(x => x.ToPointLatLng()));
                watch.Stop();
                return new Tuple<List<PointLatLng>, double>(path, forwardIntersection.ShortestPathLength +
backIntersection.ShortestPathLength);
            }

            return null;
        }
```

```
        protected void ForwardSearch(Node startNode, Node endNode, CancellationTokenSource source)
        {
            Search(startNode, endNode, true, source);
        }

        protected void BackSearch(Node startNode, Node endNode, CancellationTokenSource source)
        {
            Search(endNode, startNode, false, source);
        }

        protected void Search(Node startNode, Node endNode, bool isForward, CancellationTokenSource
source)
        {
            AStarNode watchingNode = new AStarNode(startNode),
                endWatchingNode = new AStarNode(endNode);
            List<AStarNode> openList = new List<AStarNode>
            {
                watchingNode
            };
            watchingNode.CalculateH(endWatchingNode);

            while (watchingNode.H > 0)
            {
                if (!isForward)
                {
                    var a = 0;
                }
                else
                {
                    var a = 0;
                }
                if (source.IsCancellationRequested)
                {
                    break;
                }
                //using (var db1 = new TeamMapEntities())
                //{
                    List<Direction> directions;
                    lock (synch)
                    {
                        directions = isForward
                            ? dbDirections
                                .Where(x => x.StartNodeId == watchingNode.Id ||
                                            (x.EndNodeId == watchingNode.Id && !x.IsOneWay))
                                .ToList().Where(x => ForwardCloseList.ToList().All(y => y.Id !=
x.Node.Id)).ToList()
                            : dbDirections
                                .Where(x => x.EndNodeId == watchingNode.Id ||
                                            (x.StartNodeId == watchingNode.Id && !x.IsOneWay))
                                .ToList().Where(x => BackCloseList.ToList().All(y => y.Id !=
x.Node.Id)).ToList();
                    }

                    if (directions.Count != 0)
                    {
                        List<AStarNode> neighbours = isForward
                            ? directions
                                .Select(x => new AStarNode(watchingNode, x, endWatchingNode)).ToList()
                            : directions
                                .Select(x => new AStarNode(watchingNode, x, endWatchingNode,
true)).ToList();


                        foreach (var neighbour in neighbours)
                        {
                            AStarNode existNode = openList.FirstOrDefault(x => x.Id == neighbour.Id);
                            if (existNode != null)
                            {
                                if (existNode.ShortestPathLength > neighbour.ShortestPathLength)
                                {
                                    existNode.ShortestPathLength = neighbour.ShortestPathLength;
                                    existNode.PreviousNode = watchingNode;
                                }
                            }
```

```
                        else
                        {
                            openList.Add(neighbour);
                        }
                    }
                }
            //}


            openList.Remove(watchingNode);
            lock (synch)
            {
                if (isForward)
                {
                    ForwardCloseList.Add(watchingNode);
                }
                else
                {
                    BackCloseList.Add(watchingNode);
                }
            }
            lock (synch)
            {
                if (ForwardCloseList.HasIntersection(BackCloseList))
                {
                    source.Cancel();
                }
            }
            if (openList.Count == 0)
            {
                source.Cancel();
                break;
            }
            AStarNode newWatchingNode = openList.Aggregate((node1, node2) =>
                ((0.3 * node1.ShortestPathLength + node1.H) / node1.DirectionCoeficient <
                 (0.3 * node2.ShortestPathLength + node2.H) / node2.DirectionCoeficient)
                    ? node1
                    : node2);
            watchingNode = newWatchingNode;
        }
    }
}
}
```

BiDijkstraPthFinder.cs:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Media;
using GMap.NET;
using OsmToMsSql.Models;
using TeamMapCore;
using TeamMapCore.Enums;
using WpfOsmSharp.Models;

namespace WpfOsmSharp.PathFinders
{
```

```
public class BiDijkstraPathFinder: BasePathFinder, IPathFinder
{
   protected bool isCH = false;
   private List<Direction> tempDirections;


   public BiDijkstraPathFinder(TeamMapEntities db) : base(db)
   {
      dbDirections = db.Directions.ToList();
      isCH = db.Nodes.Any(x => x.Order > 0);
   }


   public Tuple<List<PointLatLng>, double> RouteFinding(Node start, Node end)
   {
      //GetNodeIdsByShortcutId(591460);

      tempDirections = dbDirections.Where(x => Math.Min(x.Node.Order, x.Node1.Order) >= Math.Min(start.Order,
end.Order)).ToList();
      Stopwatch watch = Stopwatch.StartNew();
      List<DijkstraNode> forwardOpenList = new List<DijkstraNode>()
      {
         new DijkstraNode() {Id = start.Id, Weight = 0, Order = start.Order}
      };
      List<DijkstraNode> forwardCloseList = new List<DijkstraNode>();
      List<DijkstraNode> backOpenList =
         new List<DijkstraNode>() {new DijkstraNode() {Id = end.Id, Weight = 0, Order = end.Order}};
      List<DijkstraNode> backCloseList = new List<DijkstraNode>();
      long intersectionId = 0;
      List<Task> tasks = new List<Task>();

      Stopwatch watch0 = Stopwatch.StartNew();
      Parallel.Invoke(() =>
      {
         while (forwardOpenList.Count > 0)
         {
            DijkstraStep(forwardOpenList, forwardCloseList, true);
         }
      },
      () =>
      {
         while (backOpenList.Count > 0)
         {
            DijkstraStep(backOpenList, backCloseList, false);
         }
      });
      watch0.Stop();
      try
      {
         Stopwatch watch1 = Stopwatch.StartNew();
```

```
                intersectionId = GetBestIntersection(forwardCloseList, backCloseList);
                List<PointLatLng> openPath = new List<PointLatLng>();
                double distance = 0;
                if (intersectionId != 0)
                {
                    List<DijkstraNode> closePath = new List<DijkstraNode>();
                    List<Node> forwardNodes = new List<Node>();
                    List<DijkstraNode> backClosePath = new List<DijkstraNode>();
                    List<Node> backNodes = new List<Node>();
                    Parallel.Invoke(() =>
                        {
                            closePath = GetPath(forwardCloseList.FirstOrDefault(x => x.Id == intersectionId));
                            forwardNodes = GetNodesByClosePath(closePath);
                        },
                        () =>
                        {
                            backClosePath = GetPath(backCloseList.FirstOrDefault(x => x.Id == intersectionId));
                            backNodes = GetNodesByClosePath(backClosePath, true);
                            backNodes.Reverse();
                            backNodes.RemoveAt(0);
                        });
                    distance = closePath.Last().Weight + backClosePath.Last().Weight;
                    List<Node> nodes = new List<Node>();
                    nodes.AddRange(forwardNodes);
                    nodes.AddRange(backNodes);

                    openPath = nodes.Select(x => new PointLatLng(x.Latitude, x.Longitude)).ToList();
                }
                watch1.Stop();
                watch.Stop();
                return new Tuple<List<PointLatLng>, double>(openPath, distance);
            }
            catch (Exception e)
            {
                var openPath = new List<PointLatLng>();
                openPath.AddRange(forwardCloseList.Select(x => dbNodes.FirstOrDefault(y => y.Id == x.Id)).Select(x => new
PointLatLng(x.Latitude, x.Longitude)));
                openPath.AddRange(backCloseList.Select(x => dbNodes.FirstOrDefault(y => y.Id == x.Id)).Select(x => new
PointLatLng(x.Latitude, x.Longitude)));
                return new Tuple<List<PointLatLng>, double>(openPath, 0);
            }
        }

        protected void DijkstraStep(List<DijkstraNode> openList, List<DijkstraNode> closeList, bool isForward)
        {
            if (openList.Count == 0)
            {
                return;
            }
            DijkstraNode currentNode = openList.OrderBy(x => x.Weight).FirstOrDefault();
```

```
                IEnumerable<Direction> query = tempDirections;
                if (isForward)
                {
                    query = query.Where(x =>
                        x.StartNodeId == currentNode.Id || (x.EndNodeId == currentNode.Id && !x.IsOneWay));
                }
                else
                {
                    query = query.Where(x =>
                        x.EndNodeId == currentNode.Id || (x.StartNodeId == currentNode.Id && !x.IsOneWay));
                }
                query = query.Where(x => x.Node.Id == currentNode.Id ? x.Node1.Order > currentNode.Order : x.Node.Order > currentNode.Order);
                List<DijkstraNode> visibleFromCurrent = query.ToList()
                    .Select(x =>
                    {
                        Node node = x.StartNodeId == currentNode.Id
                            ? x.Node
                            : x.Node1;
                        return new DijkstraNode()
                        {
                            Id = node.Id,
                            Weight = currentNode.Weight + x.Length,
                            ShortcutId = x.TypeId == DirectionTypes.Shortcut ? (long?)x.Id : null,
                            Order = node.Order,
                            ParentNode = currentNode
                        };
                    })
                    .Where(x => !closeList.Select(y => y.Id).Contains(x.Id))
                    .OrderBy(x => x.Weight).ToList();
                foreach (var node in visibleFromCurrent)
                {
                    if (!openList.Select(x => x.Id).Contains(node.Id))
                    {
                        openList.Add(node);
                    }
                    else
                    {
                        DijkstraNode openedNode = openList.FirstOrDefault(x => x.Id == node.Id);
                        if (openedNode.Weight > node.Weight)
                        {
                            openedNode.Weight = node.Weight;
                            openedNode.ShortcutId = node.ShortcutId;
                            openedNode.ParentNode = currentNode;
                        }
                    }
                }
                closeList.Add(currentNode);
                openList.Remove(currentNode);
            }
```

```
protected long GetBestIntersection(List<DijkstraNode> list1, List<DijkstraNode> list2)
{
   IEnumerable<long> intersections = list1.Select(x => x.Id).Intersect(list2.Select(x => x.Id));
   if (!intersections.Any())
   {
      return 0;
   }

   long bestIntersectionId = 0;
   double bestWeight = Double.MaxValue;
   foreach (var intersection in intersections)
   {
      double weight1 = list1.FirstOrDefault(x => x.Id == intersection).Weight,
         weight2 = list2.FirstOrDefault(x => x.Id == intersection).Weight,
         weight = weight1 + weight2;
      if (weight < bestWeight)
      {
         bestWeight = weight;
         bestIntersectionId = intersection;
      }
   }

   return bestIntersectionId;
}

protected List<Node> GetNodesByClosePath(List<DijkstraNode> closePath, bool isReverse = false)
{
   List<Node> resultNodes = new List<Node>();
   for (int i = 0; i < closePath.Count; i++)
   {
      DijkstraNode dNode = closePath[i];
      if (dNode.ShortcutId.HasValue)
      {
         List<Node> nodesToAdd = GetNodeIdsByShortcutId(dNode.ShortcutId.Value);
         if (isReverse)
         {
            nodesToAdd.Reverse();
         }
         nodesToAdd.RemoveAt(0);
         resultNodes.AddRange(nodesToAdd);
      }
      else
      {
         resultNodes.Add(dbNodes.FirstOrDefault(x => x.Id == dNode.Id));
      }
   }

   return resultNodes;
}
```

```
protected List<Node> GetNodeIdsByShortcutId(long shortcutId)
{
   //Direction dir = dbDirections.FirstOrDefault(x => x.Id == shortcutId);
   //List<Node> nodes = new List<Node> { dir.Node1 };
   //Stopwatch watch1 = Stopwatch.StartNew();
   //GetNodeIdsByShortcut(dir, nodes);
   //watch1.Stop();

   //Stopwatch watch2 = Stopwatch.StartNew();
   //List<Node> nodes1 = GetNodesByShortcut(dir);
   //watch2.Stop();

   //Stopwatch watch3 = Stopwatch.StartNew();

   List<Node> nodes2 = new List<Node>();
   using (SqlConnection conn = new SqlConnection(db.Database.Connection.ConnectionString))
   {
      using (SqlDataAdapter da = new SqlDataAdapter())
      {
         da.SelectCommand = new SqlCommand("GetNodesByShortcutId", conn);
         da.SelectCommand.Parameters.AddWithValue("shortcutId", shortcutId);
         da.SelectCommand.CommandType = CommandType.StoredProcedure;

         DataSet ds = new DataSet();
         da.Fill(ds);

         foreach (DataRow row in ds.Tables["Table"].Rows)
         {
            var itemArray = row.ItemArray;
            nodes2.Add(new Node
            {
               Id = (long)itemArray[0],
               Latitude = (double)itemArray[1],
               Longitude = (double)itemArray[2],
               Order = (int)itemArray[4]
            });
         }
      }
   }

   //var temp = db.Database.ExecuteSqlCommand("GetNodesByShortcutId @shortcutId",
   //    new SqlParameter("shortcutId", shortcutId));
   //watch3.Stop();
   return nodes2;
}

protected void GetNodeIdsByShortcut(Direction direction, List<Node> nodes)
{
   if (direction.TypeId == DirectionTypes.Shortcut)
```

```
            {
                IEnumerable<Direction> foundDirections = dbDirections.Where(x =>
                    x.Id == direction.InputDirectionId.Value || x.Id == direction.OutputDirectionId.Value);
                GetNodeIdsByShortcut(foundDirections.FirstOrDefault(x => x.Id == direction.InputDirectionId.Value), nodes);
                GetNodeIdsByShortcut(foundDirections.FirstOrDefault(x => x.Id == direction.OutputDirectionId.Value), nodes);
            }
            else
            {
                nodes.Add(nodes.Select(x => x.Id).Contains(direction.EndNodeId) ? direction.Node1 : direction.Node);
            }
        }

        protected List<Node> GetNodesByShortcut(Direction shortcut)
        {
            List<Direction> directions = new List<Direction>();
            GetDirectionsByShortcut(shortcut, directions);
            List<Node> nodes = new List<Node>();

            directions.ForEach(x =>
            {
                nodes.Add(nodes.Any(y => y.Id == x.Node1.Id) ? x.Node : x.Node1);
            });
            nodes.Add(nodes.Any(y => y.Id == shortcut.Node1.Id) ? shortcut.Node : shortcut.Node1);
            return nodes;
        }

        private void GetDirectionsByShortcut(Direction direction, List<Direction> directions)
        {
            if (direction.TypeId == DirectionTypes.Shortcut)
            {
                GetDirectionsByShortcut(dbDirections.FirstOrDefault(x => x.Id == direction.InputDirectionId),
                    directions);
                GetDirectionsByShortcut(dbDirections.FirstOrDefault(x => x.Id == direction.OutputDirectionId),
                    directions);
            }
            else
            {
                directions.Add(direction);
            }
        }


    }
}
```