НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО”

Факультет Інформатики та обчислювальної техніки
(повне найменування інституту, факультету

Обчислювальної техніки
(повна назва кафедри)

До захисту допущено
Завідувач кафедри
_____ Стіренко С.Г
(підпис) (ініціали, прізвище)
“____”_____2019р

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки **6.050103 « Програмна інженерія »** _____
(код і назва)

на тему: Побудова найкоротшого маршруту на карті з урахуванням областей видимості __ проміжних пунктів

Виконав (-ла): студент (-ка) 4 курсу, групи ІП-53

Зубрич Євгенія Сергіївна _____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асист. каф. ОТ Подрубайло Олександр Олександрович____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант нормо контроль д.т.н.,проф. Сімоненко В.П. _____ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО"

Факультет Інформатики та обчислювальної техніки
(повне найменування інституту, факультету)

Обчислювальної техніки

Освітньо-кваліфікаційний рівень **бакалавр**

Напрям підготовки **6.050103 « Програмна інженерія »**

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Стіренко С.Г.
(підпис) (прізвище ініціали)

"___"_____ 2019 р.

ЗАВДАННЯ

на бакалаврський дипломний проект студентки

Зубрич Євгенії Сергіївни
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) «*Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів»*
керівник проекту (роботи) асистент Подрубайло Олександр Олександрович,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "__" _____ 2019 року №_____
1. Термін здачі студентом закінченого проекту (роботи) *20 червня 2019* р.
3. Вихідні дані до проекту (роботи) *Технічне завдання*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються):
Обґрунтування теми та аналіз існуючих рішень.
Розробка моделі вирішення задачі та алгоритму.
Реалізація вирішення задачі.
Аналіз результатів.
5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень):
1) Схема структурна класів програмного забезпечення

2) Блок-схема алгоритму роботи програми

3) Схема структурна послідовностей

6. Консультанти проекту (роботи), з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В.П.		

7. Дата видачі завдання «____» _____ *2019 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Найменування етапів дипломного проекту (роботи)	Строк виконання етапів проекту(роботи)	Примітки
1.	Вивчення предметної області	15.12.2018	Виконано
2.	Аналіз існуючих методів розв’язання задачі	07.03.2019	Виконано
3.	Постановка та формалізація задачі	21.03.2019	Виконано
4.	Аналіз вимог до програмного забезпечення	28.03.2019	Виконано
5.	Моделювання програмного забезпечення	11.04.2019	Виконано
6.	Оформлення пояснювальної записки	25.05.2019	Виконано
7.	Подання ДП на попередній захист	29.05.2019	Виконано
8.	Подання ДП рецензенту	05.06.2019	Виконано
9.	Подання ДП на основний захист	20.06.2019	

Студент _____ Зубрич Є.С.
(підпис)

Керівник проекту _____ Подрубайло О.О.
(підпис)

Анотація

В бакалаврській дипломній роботі було запропоновано та реалізовано алгоритм побудови найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів.

Програма дозволяє розрахувати оптимальний маршрут на основі загальнодоступних геопросторових даних від проекту OpenStreetMap та параметрів, отриманих від користувача, оцінюючи не лише координати пунктів, але і їх межі по відношенню до інших об’єктів на карті. Програмний продукт було створено на мові Java.

Аннотация

В бакалаврской дипломной работе было предложено и реализовано алгоритм построения кратчайшего маршрута на карте с учетом областей видимости промежуточных пунктов.

Программа позволяет рассчитать оптимальный маршрут на основе общедоступных геопространственных данных от проекта OpenStreetMap и параметров, полученных от пользователя, оценивая не только координаты пунктов, но и их границы по отношению к другим объектам на карте. Программный продукт был создан на языке Java.

Annotation

In the bachelor thesis work, an algorithm for constructing the shortest route on the map considering the areas of visibility of intermediate points was proposed and implemented.

The program allows user to calculate the optimal route based on publicly available geospatial data from the OpenStreetMap project and the parameters received from the user, evaluating not only the coordinates of the points, but also their bounds and relations with other objects on the map. The software product was implemented in Java.

Справки	Формат	*Позначення*	*Найменування*	Кільк. аркушів	№ екземпляра	*Примітки*
			Документація загальна			
			Розроблена заново			
	А4	***ІАЛЦ.466538.002 ТЗ***	Побудова найкоротшого	4		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			Технічне завдання			
	А4	***ІАЛЦ.466538.003 ПЗ***	Побудова найкоротшого	62		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			Пояснювальна записка			
	А4	***ІАЛЦ.466538.004 Д1***	Побудова найкоротшого	12		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			Текст програми			
	А3	***ІАЛЦ.466538.005 Д2***	Побудова найкоротшого	1		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			UML діаграма класів			

					ІАЛЦ.466538.001 ОА			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Зубрич Є.С.			Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів. Опис альбому	Літ.	Арк.	Аркушів
Перев.		Подрубайло					1	2
						НТУУ "КПІ" ім. Ігоря Сікорського, ФІОТ, кафедра ОТ, гр. ІП-53		
Н. контр.		Сімоненко						
Затв.		Стіренко С.Г.						

Справки	Формат	*Позначення*	*Найменування*	Кільк. аркушів	№ екземпляра	*Примітки*
	А3	***ІАЛЦ.466538.006 Д3***	Побудова найкоротшого	1		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			Схема послідовності			
			алгоритму			
	А4	***ІАЛЦ.466538.007 Д4***	Побудова найкоротшого	1		
			маршруту на карті з			
			урахуванням областей			
			видимості проміжних пунктів			
			Діаграма послідовності			

ЗМІСТ

					ІАЛЦ.466358.002 ТЗ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розробн.		Зубрич Є.С.			Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів Технічне завдання	Літ.	Арк. 1 / Аркушів 3
Перевір.		Подрубайло О.О.					
Н. Контр.		Сімоненко В.П.				НТУУ «КПІ», ФІОТ, ІП-53	
Затверд.		Стіренко С.Г.					

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТІОСУВАННЯ

Дане технічне завдання поширюється на розробку програми за темою «Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів». Область застосування: використання як стороннього сервісу для розробки оптимальних туристичних маршрутів.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи освітньо-кваліфікаційного рівня «бакалавр програмної інженерії», затверджене кафедрою спеціалізованих комп'ютерних систем Національного технічного Університету України «Київський політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка моделі інструменту побудови найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів, а також закріплення навичок програмування та проектування програмного забезпечення, набутого впродовж навчання за напрямом «Інженерія програмного забезпечення».

4. ТЕХНІЧНІ ВИМОГИ

4.1. Вимоги до продукту, що розробляється

1. Програма повинна складатись із окремих модулів, елементи яких є слабозв’язними.
2. Незалежно від обраних методів програмування, структури даних, що розробляються, повинні забезпечувати достатню ефективність операцій.

3. Чистота написання коду та дотримання конвенцій обраної мови програмування.

4.2. Вимоги до програмного забезпечення

- Операційна система MS Windows 98, MS Windows XP, MS Windows 7, MS Windows 8, MS Windows 10, Unix OS.
- Java JRE версії 8 та вище
- Gradle

4.3. Вимоги до апаратного забезпечення

- Комп’ютер на базі процесора Intel або AMD від 256 МГц
- Оперативної пам'яті не менше 512 Мбайт
- Вільний простір на диску не менше ніж 100 Мбайт

5. ЕТАПИ РОЗРОБКИ

Вивчення предметної області	15.12.2018
Аналіз існуючих методів розв’язання задачі	07.03.2019
Постановка та формалізація задачі	21.03.2019
Аналіз вимог до програмного забезпечення	28.03.2019
Моделювання програмного забезпечення	11.04.2019
Оформлення пояснювальної записки	25.04.2019
Подання ДП на попередній захист	29.05.2019
Подання ДП рецензенту	05.06.2019
Подання ДП на основний захист	20.06.2019

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

OSM	OpenStreetMap
LP	Linear Programming
ALT	A* search, Landmarks and Triangle inequality
IDDFS	Iterative deepening depth-first search
XML	Extensible Markup Language
JSON	JavaScript Object Notation
API	Application Programming Interface
MVC	Model-View-Controller

ВСТУП

Сьогодні автоматизовані системи побудови та візуалізації маршрутів використовуються у багатьох галузях, а планувальники маршрутів стали невід'ємною частиною повсякденного життя. Найбільш актуальним напрямом наразі є туризм. Позитивний вплив інформаційних технологій на динаміку вітчизняного та міжнародного туристичного потоку привів до трансформування туристичної галузі з такої, що орієнтована на обслуговування організованих туристів, на багатогалузеву сферу діяльності, спрямовану на задоволення різноманітних потреб мільйонів індивідуальних туристів.

Особливо затребуваними стали в наші дні і продовжують активно розвиватися системи планування маршрутів, що працюють в режимі реального часу. Проте існуючі алгоритми здебільшого направлені на пошук найшвидшого та найкоротшого маршруту. Використовуючи системи, що генерують лише найкоротший маршрут турист ризикує не побачити частину пам'яток даної місцевості.

На даний момент недослідженими та нереалізованими є системи які б при побудові найкоротшого маршруту враховували типи об'єктів, що є на мапі, та класифікували їх за тематикою, на основі чого можна було модифікувати маршрут таким чином, щоб він був, з одного боку, коротким, з іншого боку, охоплював якомога більше туристичних об'єктів.

Метою цієї роботи є побудова системи, що при побудові маршруту буде враховувати його довжину та кількість охоплених об'єктів заданого типу. Для цього розробляється алгоритм розрахунку областей видимості об'єктів, обчислення величини їх пріоритету та модель системи, яка на основі цих значень буде обчислювати вагу ребер графу маршрутів та будувати найкоротший шлях.

РОЗДІЛ 1

ОСНОВНІ ВИЗНАЧЕННЯ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ПРОБЛЕМИ ПОБУДОВИ НАЙКОРОТШИХ МАРШРУТІВ

1.1 Актуальність теми дослідження

За останні кілька років планувальники маршрутів стали невід'ємною частиною повсякденного життя. Багато типів планувальників існують в режимі онлайн або окремо у формі пристроїв, пропонуючи маршрути для автомобілів, велосипедів або пішоходів.

Тим не менш, пристрій або веб-служба, що підтримувала би українські міста є досі недоступною. Існуючі планувальники маршрутів, які пропонують можливість планувати автомобільні чи пішохідні маршрути, будують лише найкоротший маршрут і не враховують туристичні пам'ятки. Метою цього проекту є розробка програми яка могла би в режимі реального часу побудувати оптимальний маршрут за заданими координатами початкової та кінцевої точок ураховуючи місцезнаходження, межі та область видимості проміжних пунктів.

Незважаючи на простий та зрозумілий інтерфейс планувальника, технології побудови маршрутів є доволі складними. Пошук найкоротшого шляху з однієї точки карти в іншу може бути вирішено шляхом вирішення проблеми найкоротшого шляху між вузлами на графі, оскільки картографічні дані можна представити у вигляді графу координат. Однак кількість пам'яті та часу, що потрібні для вирішення цієї задачі, дещо ускладнюють проблему.

1.2 Аналіз останніх досліджень і публікацій

1.2.1 Проблема пошуку найкоротшого шляху

Проблема пошуку маршруту на карті може бути представлена у простій формі графа, як зображено на рисунку 1.1. Так вершини графа є відображенням таких об'єктів карти, як будівлі, монументи, тощо, а ребра – вулиць та доріг між

ними. Кожне ребро має власну вагу, значенням якої може бути відстань або час, що потрібно витратити на подолання цієї відстані.

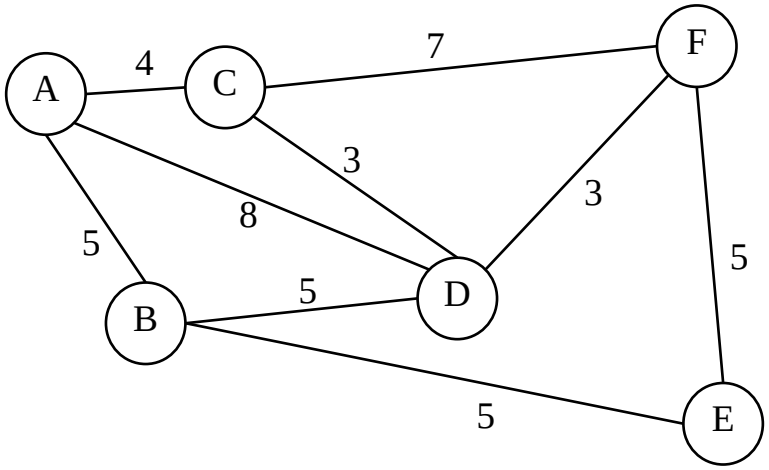


Рис. 1.1. Зважений граф

Щоб знайти найкоротший шлях з, наприклад, вершини A до F, потрібно знайти колекцію ребер, що з'єднують між собою вузли A та F, для якої сума вагів ребер буде найменшою. У цій роботі довжина найкоротшого шляху від деякого вузла s до деякого вузла t буде називатися відстанню від s до t.

Розглянемо орієнтований граф $G = (V, E)$, де V - скінченний набір вузлів і E - набір ребер між цими вузлами. Число вузлів $|V|$ будемо позначати n, а число ребер $|E|$ будемо позначати m. Кожне ребро e має вагу $w\ (e)$. Шлях визначається послідовністю вузлів $(v_1, \dots, v_k)$, для яких $(v_1, v_{i+1}) \in E$ для $1 < i < k$. Вузол, з'єднаний з певним вузлом v деяким ребром, називається сусідом v. Вузол, який передує деякому іншому вузлу шляху, називається його батьком на цьому шляху. Аналогічно, наступник вузла на певному шляху називається його дитиною на цьому шляху.

Якщо початковим вузлом є вершина $s \in V$ і кінцевим вузлом $t \in V$, то найкоротший шлях визначається як шлях $(s, \dots, t)$, який має мінімальну суму ваг всіх ребер на шляху. Довжина найкоротшого шляху від s до вузла v визначається як $g(v)$ і також називається відстанню від s до v.

1.2.2 Метод лінійної оптимізації

Одним із способів вирішення проблеми найкоротшого шляху є використання моделі лінійного програмування (LP Model), описаної в [1]. Для того, щоб сформулювати цю модель, необхідно визначити наступні змінні.

$x_e = \begin{cases} 1 & \text{якщо ребро } e \text{ входить до оптимального шляху} \\ 0 & \text{інакше} \end{cases}$

$\delta_i^+ :=$ множина ребер, що входять у вершину u_i

$\delta_i^- :=$ множина ребер, що виходять з вершини u_i

Модель методом лінійного програмування може бути визначена наступним чином:

Необхідно мінімізувати

$$\sum_{e \in E}^{n} x_e \, w(e) \tag{1.1}$$

при наступних обмеженнях:

$\sum_{e \in \delta_i^-} x_e - \sum_{e \in \delta_i^+} x_e = 0 \quad \forall i: v_i \in V\{s,t\}$

$\sum_{e \in \delta_s^-} x_e - \sum_{e \in \delta_s^+} x_e = 1$

$\sum_{e \in \delta_t^-} x_e - \sum_{e \in \delta_t^+} x_e = -1$

$x_e \geq 0 \ \ \forall e \in E$

Перше обмеження описує вимогу, що кожен вузол, який не є s або t, повинен мати однакову кількість вхідних та вихідних ребер. Друге обмеження описує, що s повинен мати на одне вихідне ребро більше, аніж кількість вхідних. Третє обмеження описує, що вузол t повинен мати на одне вхідне ребро більше, аніж кількість вихідних. Четверте обмеження описує, що змінні повинні бути більшими або рівними 0. Сама модель гарантує, що всі змінні приймають або значення 0, або значення 1, оскільки будь-яке інше значення призведе до неоптимального рішення. Планувальники маршрутів повинні вирішувати проблеми найкоротшого шляху значного розміру. Для цього випадку лінійна оптимізація не є найбільш підходящим методом пошуку рішення.

Незалежно від якості моделі та ефективності коду, час обчислення буде занадто великим, якщо не буде застосовано конкретний підхід до проблеми. Наступні розділи описують ряд алгоритмів, які пропонують більш ефективний спосіб пошуку найкоротших шляхів.

1.2.3 Алгоритм Дейкстри

У роботі [2] Дейкстра описує алгоритм, який вирішує задачу найкоротшого шляху з невід'ємними вагами набагато ефективніше, ніж LP. Цей алгоритм тепер відомий як алгоритм Dijkstra і був ретельно документований. Для кожного вузла v зберігаються дві властивості. Перша - це довжина найкоротшого шляху від s до v, знайденого до цих пір, друга - вузол, що є предком v на цьому шляху. Алгоритм побудує оптимальні шляхи ітераційним способом, поліпшуючи рішення на кожному кроці. Після завершення буде отримано найкоротший шлях від вихідного вузла до всіх інших вузлів графу.

Алгоритм

Для опису пояснення алгоритму необхідні наступні визначення:

- A: = набір вузлів v, для яких знайдено найкоротший шлях від s до v (множина опрацьованих вершин).
- X: = набір вузлів v, для яких шлях від s до v ще невизначений (множина вершин, що в черзі на опрацювання).
- $\hat{g}(v)$: = довжина найкоротшого шляху від s до v знайдена на момент поточної ітерації (це можна розглядати як оцінку або верхню межу для найкоротшого шляху від s до v).
- $\hat{g}(s) := 0$

Ініціалізація алгоритму відбувається наступними значеннями:

- $A = \emptyset$
- $X = \{s\}$

Повторювати наступні кроки алгоритму, доки X не дорівнює пустій множині:

1. Знайти вершину v з множини X таку, що

$$\hat{g}(v) := \min_{u \in X} \hat{g}(u), \tag{1.2}$$

видалити v з X та додати до A.

2. Для кожної вершини u, для якої виконується $(v, u) \in E$ (кожна сусідня вершина v):
   a. Якщо A містить u, то не виконувати жодних дій.
   b. Якщо X містить u, то
      i. якщо $\hat{g}(v) + w(v, u) < \hat{g}(u)$, то встановити $\hat{g}(u) = \hat{g}(v) + w(v, u)$ та $parent(u) = v$.
   c. Якщо u не міститься ані у A, ані у X, додати u до X та встановити $\hat{g}(u) = \hat{g}(v) + w(v, u)$ та $parent(u) = v$.

Коли алгоритм закінчується, всі вузли, які можуть бути досягнуті з s, будуть у A (включаючи t), і задачу буде вирішено. Щоб зберегти деякий час обчислення, алгоритм може бути зупинений, як тільки t буде додано до A, оскільки в цей момент буде знайдено найкоротший шлях до t.

Ефективність алгоритму

Алгоритм Дейкстри завжди знаходить оптимальний шлях. На рис. 1.2 показано як на прикладі даного графу розширюється множина A навколо вихідного вузла s, при чому кожен вузол всередині кордону ближче до s, ніж будь-який вузол за межами кордону. А продовжує розширюватися, поки не містить цільовий вузол.

Визначимо v_k як вузол, який додається до A на кроці k. За допомогою методу індукції [?] доведемо, що v_k - це вузол, для якого довжина найкоротшого шляху $g(v_k)$ є найменшою з усіх вузлів, що не входять в A.

Це тривіально для кроку 1, оскільки s додається до A з $g(s) = 0$.

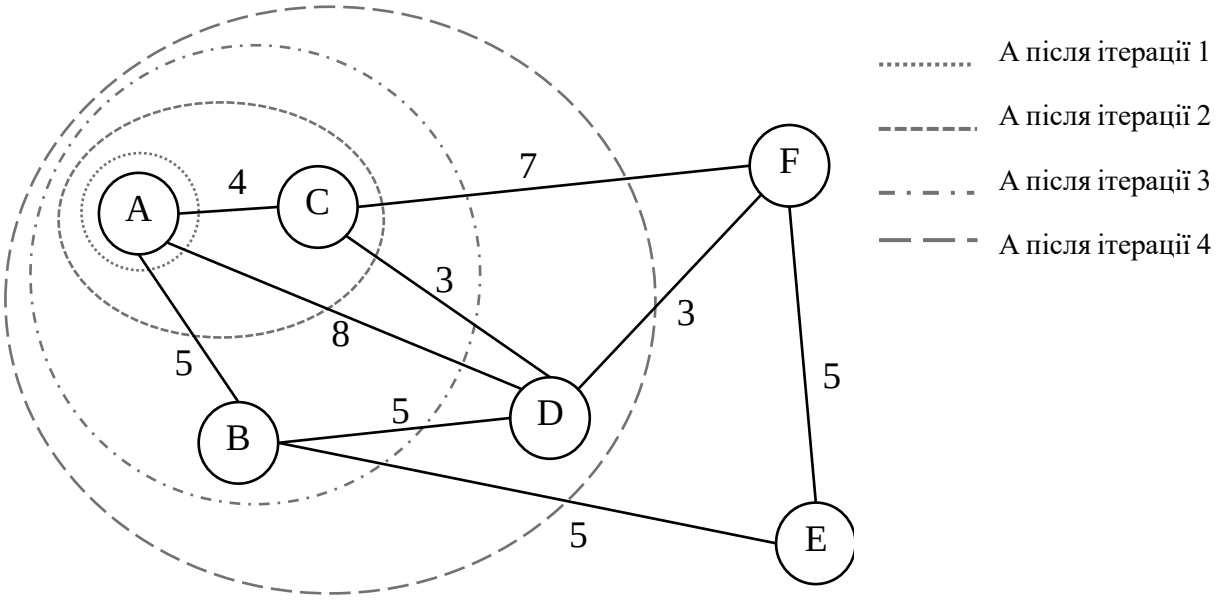


Рис. 1.2. Розширення множини A в алгоритмі Дейкстри

Скажімо, що ця умова дотримується до деякого кроку k, після чого будь-який вузол, що не в A, буде мати більшу відстань від s, ніж будь-який вузол в A. Обов'язково, всі сусіди всіх вузлів з множини A будуть в множині X; будь-який вузол, який не знаходиться ані в A, ані в X, матиме більшу відстань від s, ніж принаймні один вузол у X, оскільки доведеться пройти через вузол з X, щоб дістатися s. Це означає, що вузол v_{k+1}, який необхідно додати до A на кроці $k + 1$, буде знаходитися в X і тому буде знайдений алгоритмом.

Оскільки це доречно для k = 1, це доводить, що в це також доречно для будь-якого $k \geq 1$, або для будь-якого кроку алгоритму. Оскільки найкоротший шлях до v_k коротший, ніж до будь-якого вузла за межами A, цей шлях не містить жодних вузлів, що не входять в A, то найкоротший знайдений шлях - це найкоротший шлях, або $\hat{g}(v_k) = g(v_k)$, що означає, що v_k можна додати до A. Формальний доказ для алгоритму Дейкстри приводиться далі.

Коли вузол додано до A, найкоротший шлях знайдено, і алгоритм більше не буде розглядати цей вузол. Щоб довести, що шлях Дейкстри знаходить для кожного вузла дійсно найкоротший шлях, нам потрібно довести для кожної

ітерації k, що не існує шляху до v_k, коротше ніж $\hat{g}(v_k)$, або $\hat{g}(v_k) = g(v_k)$. Для цього використовується доказ леми 1, наведений у [3].

Лема 1

Для будь-якого оптимального шляху P від s до деякого вузла $u \notin A$, існує вузол $v \in X$ на шляху P з $\hat{g}(v) = g(v_k)$.

Доказ

Якщо $s \in X$, це тривіально істинно, оскільки $\hat{g}(s) = g(s) = 0$. В іншому випадку, нехай v^* - це вузол, який був доданий до А останнім. Це не може бути той же вузол, що і v, оскільки $v \notin A$. Нехай v є нащадком v^* на шляху P. За визначенням $\hat{g}$, має місце вираз (1.3)

$$\hat{g}(v) \leq \hat{g}(v^*) + w(v^*, v). \tag{1.3}$$

Оскільки $v^* \in A$, $\hat{g}(v^*) = g(v^*)$. і тому що P - оптимальний шлях, $\hat{g}(v) \leq \hat{g}(v^*) + w(v^*, v)$. Об'єднання цих результатів призводить до $\hat{g}(v) \leq \hat{g}(v)$, але взагалі $\hat{g}(v) \geq \hat{g}(v)$. З цього випливає, що $\hat{g}(v) = \hat{g}(v)$.

Лема 1 показує, що для кожної ітерації існує вузол $v_k \in X$, де $\hat{g}(v_k) = g(v_k)$, доки існує вузол $u \notin A$. Коли цього вузла u не існує, X дорівнює порожній множині і алгоритм зупиняється. Це доводить, що для кожного вузла, який додається до А, знайдено найкоротший шлях, тому для кожного вузла $v \in A$ алгоритм Дейкстри знайшов найкоротший шлях від s до v.

1.2.4 Алгоритм A*

Під час подорожі до певного пункту призначення, як правило, немає сенсу шукати шлях в протилежному напрямі. Тому з'явився алгоритм, який віддає перевагу вершинам, що знаходяться у напрямку до пункту призначення, і спершу відвідує їх, на відміну від алгоритму Дейкстри, який здійснює пошук у всіх напрямках простору. Харт, Нільссон і Рафаель [3] вводять алгоритм A*, який додає евристику до алгоритму Дейкстри, роблячи його більш направленим до кінцевої вершини. Замість ваги вузла v використовується оцінка накоротшого

шляху $\hat{g}(v)$ від початкової вершини до кінцевої, який пролягає через вузол v. Для цього будо введено функцію (1), що відображає значення найкоротшого шляху від s до t, що проходить через v, при чому $g(v)$ – це відстань від s до v, а $h(v)$ – відстань від v до t.

$$f(v) = g(v) + h(v) \tag{1.4}$$

Також було введено оцінки значень функцій:

$$\hat{f}(v) = \hat{g}(v) + \hat{h}(v) \tag{1.5}$$

Оцінка відстані від s до v, $\hat{g}(v)$ визначається так само, як і в алгоритмі Дейкстри, найкоротший шлях від s до v, знайдений на момент поточної ітерації. Оцінка відстані від v до t, $\hat{h}(v)$ визначається евристично. Ця функція муже будь-якою функцією і часто її визначення є окремою задачею.

Для задач планування маршруту найчастіше використовується евклідова відстань від v до t. Використана евристика має бути припустимою, тобто не має переоцінювати вартість маршруту, оцінка шляху має знаходитись в проміжку $[0; k]$ де k дорівнює фактичній відстані, і монотонною, тобто для кожної вершини v і сусідньої для неї вершини $v`$ має виконуватися нерівність (1.6), де $c(v, v`)$ – фактична відстань між v та $v`$:

$$h(v) \leq c(v, v`) + h(v`) \tag{1.6}$$

Слід зазначити, що A* оптимально ефективний для будь-якої заданої евристичної функції. Це означає, що жоден інший оптимальний алгоритм, що використовує ті ж самі знання, гарантовано не розширюється менше, ніж A*. Стисле пояснення цього наведено в [4]: «будь-який алгоритм, який не розширює всі вузли в межах між кореневим вузлом і ціллю, ризикує пропустити оптимальне рішення». Формальне доведення наведено в [3]. Як показано в наступних розділах, шляхом поліпшення евристики або шляхом виконання певної форми попередньої обробки, час розрахунку може бути значно зменшено.

Одним з недоліків як Дейкстри, так і A* є те, що коли обсяг пошуку збільшується, використання пам'яті швидко зростає для обох методів. Простір пошуку може містити мільйони вузлів і для кожного вузла, який відвідується алгоритмом, адже повинні зберігатися в пам’яті одне ціле число $\hat{g}(v)$ і один покажчик на попередній вузол. Через ці вимоги до пам'яті були розроблені методи, які можуть конкурувати з A* з точки зору продуктивності, але використовують менше пам'яті. Це алгоритми IDA*, MA* і SMA*.

1.2.5 IDA*

Iterative Deepening A* (IDA*) це варіант алгоритму пошуку в глибину з ітераційним заглибленням (IDDFS). На кожній ітерації IDA* встановлюється поріг для функції оцінки $\hat{f}(v)$. Після цього алгоритм шукає шлях, поки функція оцінки для цього шляху не перевищить порогове значення. Коли поріг перевищено, шлях вертається, поки не буде знайдений вузол з сусідньою вершиною, яка ще не була оброблена, і алгоритм продовжує шлях від цього сусіда. Ітерація закінчується, коли всі можливі шляхи до порога були оброблені.

Після завершення ітерації IDA* встановлює поріг для наступної ітерації значенням мінімуму всіх знайдених значень, які перевищували поточний поріг. Цей процес триває, поки не буде знайдено цільовий вузол.

Хоча здається, що IDA* виконує багато додаткових обчислень, розширюючи вузли кілька разів, фактичні накладні витрати це достатньо малі. Причина цього полягає в тому, що зовнішній шар розширених вузлів, розширений в останній ітерації, містить набагато більшу кількість вузлів. Той факт, що інші шари розширені кілька разів, не робить великої різниці. IDA* повертає оптимальне рішення з тими ж обмеженнями евристики, що і A*, проте є кращим варіантом, коли проблема полягає в обмежених ресурсах пам’яті. Більш докладний опис IDA* можна знайти в [5].

1.2.6 SMA*

Недолік IDA* полягає в тому, що він викидає багато інформації після кожної ітерації. Метод, який намагається зберегти якомога більше інформації, називається Memory Bounded A* або MA* [6]. Рассел [7] вводить SMA*, який покращує MA*, дозволяючи втратити ще менше інформації і роблячи алгоритм трохи легшим для реалізації.

SMA* працює приблизно так само, як A*, з тією відмінністю, що він має обмеження на кількість вузлів, які можуть зберігатися в пам'яті. Як тільки ця межа буде досягнута, вона буде видаляти з пам’яті листовий вузол тільки тоді, коли простір необхідний для кращого вузла. Коли вузол повинен бути видалений, він видалить найменший вузол з найвищим значенням $\hat{f}(v)$.

Щоб запобігти зайвій втраті інформації, інформація від нащадків зберігається у попередніх вузлах. Після розширення всіх нащадків вузла v, значення $\hat{f}(v)$ замінюється найменшим значенням всіх нащадків. Ця інформація оновлюється для всіх предків v.

Теорема 4 в [7] говорить, що, крім своєї здатності генерувати поодинокі послідовності, SMA* поводиться однаково з A*, коли кількість вузлів, що генеруються A*, не перевищує межу. SMA* повертає оптимальний шлях, коли достатньо пам'яті для зберігання найменшого оптимального рішення.

1.2.7 Двонаправлений пошук

Двонаправлений пошук застосовує довільний алгоритм пошуку вперед від вихідного вузла s і назад від цільового вузла t. Це зменшує область пошуку, а отже, кількість вузлів, які обробляються, як показано на малюнку 1.3. Площа одного еліпса ліворуч більше, ніж сума двох маленьких праворуч. Для алгоритму Дейкстри, застосування двонаправленого пошуку досить просте.

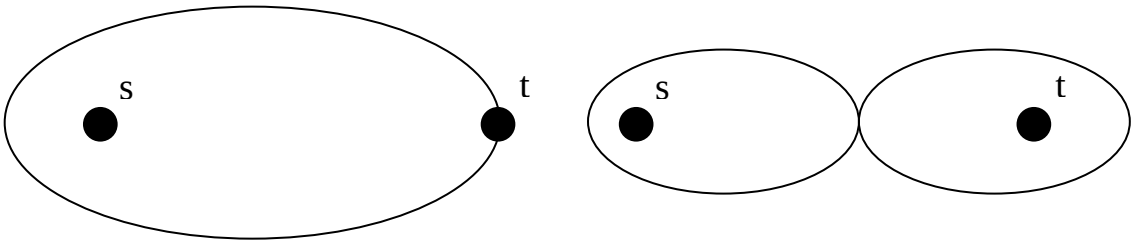


Рис. 1.3. Області пошуку при звичайному пошуку та двонаправленому

Алгоритм починає пошук з обох вузлів, і ітерація для і s, і для t виконується одночасно. Як тільки алгоритм в одному напрямку додає вузол до А, який вже знаходиться в А для іншого напрямку, знайдено найкоротший шлях.

Однак для А* метод не такий простий. Оскільки А* мінімізує оціночну функцію $\hat{f}(v)$ замість відстані від вихідного вузла, не існує гарантії, що мінімальний шлях був знайдений, коли зустрічаються дві закриті набори. Кейнд і Кенз згадують у [8], що попередні дослідження по двонаправленому евристичному пошуку вказують, що накладні витрати методу переважають вигоду. Їхній власний підхід досягає деяких перспективних результатів, зменшуючи час роботи на 30% у найбільш позитивних сценаріях. Однак цей результат є мінімальним у порівнянні з іншими оптимізаціями, які представлені в даній роботі.

Загалом, комбінація А* з двостороннім пошуком не призводить до вражаючих результатів.

1.2.8 Метод геометричної обрізки

Кожна дорога будується лише для того, щоб дістатися різних місць. Якщо інформація про довжину доріг відома під час планування маршруту, дороги, які, не знаходяться в складі найкоротшого шляху, можна легко ігнорувати. На жаль, зберігання цієї інформації займає занадто велику кількість пам’яті. Вагнер і Віллькхальм [9] вводять метод, що називається геометричною обрізкою.

Ідея геометричного методу обрізання полягає в тому, щоб зберігати набір вузлів $S(e)$ для кожного ребра e під час фази попередньої обробки, що містить

всі вузли, які можуть бути досягнуті найкоротшим шляхом, починаючи з e. Для зберігання цих наборів для кожного ребра потрібно $O(nm)$ простору.

Як уже згадувалося вище, це неможливо для великих пошукових просторів, тому Вагнер та ін. зберігають геометричну область для кожного ребра e, що містить щонайменше $S(e)$. На малюнку 4 ребро (C, D) є першим ребром в найкоротшому шляху від C до D, E і F. Отже, $S(C, D)$ містить принаймні ці три вузла.

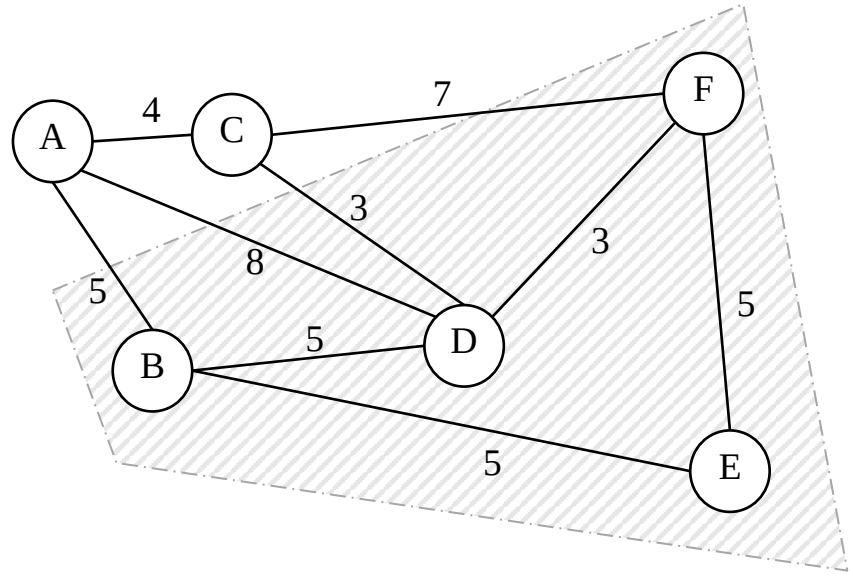


Рис. 4. Обмежуюча область для ребра (C, D)

Ці області містять набір вузлів $C(e)$, званих контейнером. Обмежуюча область називається послідовною, якщо $S(e) \subseteq C(e)$. Якщо область $C(e)$ узгоджується для всіх ребер, то алгоритм Дейкстри з обрізкою знайде найкоротший шлях від s до t. Метод вимагає ще деякі зміни алгоритму Дейкстри. Коли вузол додається до A на кроці 1 алгоритму, розглядатимуться лише вузли u, для яких $t \in C(v, u)$.

Запропонований спосіб потребує значного обсягу попередніх обробок, але може істотно зменшити кількість відвідуваних вузлів для пошуку.

Одним з найбільш ефективних, але простих геометричних форм є так звана обмежувальна коробка, що означає найменшу квадратну коробку, що містить всі вузли в $S(e)$.

При використанні обмежувальних коробок, кількість відвідуваних вузлів зменшується на 90-95% на дорожніх мережах порівняно з алгоритмом Дейкстри, зменшуючи час роботи приблизно на 10 разів.

Хоча Вагнер та інші перевіряли цей метод за допомогою алгоритму Дейкстри, вони в своєму висновку згадують, що їхній метод можна легко комбінувати з іншими методами. Одним із згаданих методів є алгоритм A*

1.2.9 ALT алгоритм

Гольдберг і Харрелсон [10] описують метод поліпшення виконання запитів найкоротшого шляху, який вони називають алгоритмом ALT (A* search, Landmarks and Triangle inequality), оскільки він базується на A*, орієнтирах і нерівності трикутника для найкоротших відстаней (не евклідових відстаней). Метою методу є знайти кращу нижню межу для відстані $h(v)$, що використовується алгоритмом A*. Більш низькі межі призводять до меншої кількості вузлів, які необхідно обробити при пошуку.

Алгоритм ALT вибирає ряд орієнтирів під час попередньої обробки. Для кожного вузла обчислюється відстань до цих наземних орієнтирів. Розглянемо орієнтир L і нехай $d(u, v)$ - це відстань від будь-якого вузла u до будь-якого вузла v. Тоді за нерівністю трикутника, $d(u, L) - d(v, L) \leq d(v, u)$. Аналогічно, $d(L, v) - d(L, u) \leq d(v, u)$.

Оскільки $d(w, L)$ і $d(L, w)$ були обчислені під час попередньої обробки, цей факт може бути використаний для обчислення нижньої межі відстані між двома вузлами. Для досягнення найнижчої нижньої межі використовується максимум по всіх орієнтирах. Як тільки нижня межа знайдена, вона може бути використана як евристика для $\hat{h}(v)$ в алгоритмі A*.

Пошук найкращих орієнтирів є ключовою задачею для загальної продуктивності алгоритмів ALT. Гольдберг та ін. описали три способи вибору орієнтирів. Якщо k - кількість наземних орієнтирів, які потрібно вибрати,

найпростішим описаним методом є вибір k вузлів випадковим чином. Однак є й кращі методи.

Другий описаний спосіб називається вибором найбільш віддаленого орієнтиру. Алгоритм розпочинає роботу, вибираючи один випадковий вузол і вузол, що знаходиться якомога далі від нього, у якості першого орієнтиру. Потім для кожного нового орієнтиру, знаходиться найвіддаленіший вузол від поточного набору орієнтирів.

Для дорожніх карт, наявність орієнтира, що лежить за пунктом призначення, допомагає отримати непогані межі. У цьому випадку метод, який називається планарним вибором орієнтирів, є гарним вибором. Згідно цього методу, необхідно знайти вузол v поблизу центру графа і поділити граф на секції k сегментів, для яких v буде центром, так щоб кожен сектор містив у собі приблизно однакову кількість вершини. Потім для кожного сектора виберіть найвіддаленіший вузол від c, він і буде орієнтиром.

Щоб уникнути ситуації, коли два орієнтири знаходитимуться занадто близько один від одного, якщо орієнтир у секторі A був обраний поблизу межі наступного сектора B, необхідно пропустити вузли B, що знаходяться поблизу межі A.

1.2.10 Багаторівневий підхід

При плануванні маршруту від одного міста до іншого виявляється, що більша частина найкоротшого маршруту не залежить від початкової та кінцевої точок. Деякі частини шляху входять до множини найкоротших шляхів, у багаторівневому підході використовується цей факт.

Сандерс і Шульц [11] вводять варіант цього методу, який вони називають ієрархіями магістралей. Їхній метод будує кілька рівнів для початкового графа під час попередньої обробки, де кожен вищий рівень є абстракцією нижнього. Оригінальний граф є найнижчим рівнем. При створенні нового рівня деякі вузли і ребра можуть бути видалені, а деякі ребра можуть бути додані.

При багаторівневому підході програма намагається знайти на графу так звані магістралі та винести їх на вищий рівень. Спочатку визначається околиця навколо кожного вузла. Магістралі складаються з ребер, які знаходяться на найкоротшому шляху від деякого вузла s до деякого кінцевого вузла t і мають початок та кінець за межами обох околиць s і t, які або покидають околицю s або входять в околицю t, проте не в обидві. Після цього виконуються кроки по очищенню нового рівня. Шляхи, що складаються лише з вузлів з двома сусідами, повність видаляються і замінюються одним ребром. Крім того, всі ізольовані вузли також видаляються. Коли потрібна кількість рівнів створена, всі вузли на кожному рівні є пов'язаними з відповідним рівнем нижче, щоб сформувати результуючий граф.

Побудова маршруту відбувається з використанням модифікованої версії алгоритму A*. Згідно досліджень багаторівневий підхід на реальних дорожніх мережах працює у 500 – 2000 разів швидше, аніж алгоритм Дейкстри.

1.2.11 Субоптимальні алгоритми

Деякі автори, наприклад, Ботеа, Мюллер та Шеффер ввели субоптимальні алгоритми, які значно знижують час обчислення. Хоча це може бути цікавим, Сандерс і Шультс описують в [12], що алгоритми найкоротшого шляху покращилися настільки з точки зору швидкості в останні роки, що це призводить до того, що накладні витрати такі як відображення маршрутів та передача їх по мережі стають єдиним вузьким місцем алгоритмів.

1.2.12 Порівняння алгоритмів

У таблиці 1.1 наведено порівняння алгоритмів, які обговорювалися в цьому розділі. Прискорення порівнюється з алгоритмом Дейкстри, якщо не вказано інше.

Таблиця 1.1. Порівняння алгоритмів пошуку найкоротшого шляху

Назва алгоритму	Попередня обробка даних	Використання пам’яті	Прискорення
LP	Немає	Незначне	Повільніший
A*	Немає	Менше використання пам’яті зумовлене кількістю вершин, що обробляються	Швидший приблизно в 2 рази та в 10 разів при позитивних сценаріях
IDA*	Немає	Незначне, у пам’яті зберігається лише один шлях	Приблизно вдвічі повільніший за A*
SMA*	Немає	Залежить від максимальної кількості вершин	Сповільнення відносно A* у випадку нестачі пам’яті
Двонаправлений пошук	Немає	Менше використання пам’яті зумовлене кількістю вершин, що обробляються	Приблизно вдвічі швидший
Геометрична обрізка	Всі пари найкоротших шляхів	Зберігаються геометричні форми для кожного ребра	У 10 – 20 разів швидший
ALT	Близько 16 запитів на найкоротший шлях	Використовується багато пам’яті, зберігаються усі відстані для кожного ребра	Приблизно у 10 разів швидший
Багаторівневий підхід	Для формування кожного рівня необхідний локальний пошук найкоротшого шляху навколо кожного вузла нижнього рівня	Для кожного рівня необхідно зберігати менший, але окремий граф	Прискорення до 2000 разів у найбільш позитивних сценаріях

1.3 Аналіз відомих технічних рішень

З ростом попиту на навігаційні системи та планувальників маршрутів, збільшилась і кількість досліджень у цій галузі. Насамперед з’явилось чимало систем, що будують найкоротший шлях між двома точками. Більшість з них можна поділити на два основних типи: вбудовані та онлайн-планувальники. З-поміж чисельних програм обох типів нижче наведено найпопулярніші рішення географічної маршрутизації.

1.3.1 TomTom

TomTom - велика міжнародна компанія, яка пропонує автономні навігаційні пристрої. Їх пристрої є одними з найпопулярніших, головним чином завдяки інтуїтивно зрозумілому інтерфейсу, швидкості та точності обчислення маршрутів. Пристрої можуть розраховувати маршрути для подорожі на машині, велосипеді або пішки. На жаль, їх алгоритм маршрутизації та джерела даних не є відкритими для розробників. Нещодавно TomTom також випустила онлайнову версію свого планувальника маршрутів, проте ця версія не має багатьох функцій, які пропонує вбудована версія [13]. Серед іншого, їй бракує можливості планувати маршрути на велосипеді або пішки. Також ані онлайн-версія, ані вбудована не мають української локалізації.

1.3.2 GoogleMaps

Google запустив свою власну службу маршрутизації, що називається Google Maps [14]. Це дуже швидкий, безкоштовний сервіс. Очевидно, що Google виконує певну попередню обробку або кешування, щоб зробити їхню службу маршрутизації такою швидкою, проте деталі та алгоритм маршрутизації залишаються в секреті. Карти Google доступні українською мовою, але Google Maps API не дозволяє модифікувати алгоритм або враховувати пріоритети об’єктів при побудові маршруту.

1.3.3 Via Michelin

Via Michelin - це служба онлайн-маршрутизації, заснована на картах відомого видавця дорожніх карт Michelin [15]. Служба є безкоштовною і швидко працює, але знову ж таки, інформація про алгоритми є конфіденційною. Via Michelin не доступна українською мовою, будує лише найкоротші маршрути і також не завжди точно працює для веломаршрутів.

1.3.4 YourNavigation.org

YourNavigation.org - це демонстраційний веб-сайт для проекту YOURS. Метою проекту є створення веб-сайту маршрутизації на основі даних OpenStreetMap (OSM) з використанням інших відкритих програм. Він використовує механізм маршрутизації з відкритим кодом, який називається Gosmore [16]. Сервіс не дуже швидко працює, і на їхньому веб-сайті згадується, що Gosmore не призначений для генерації маршрутів довжиною більше 200 км. Планувальних надає можливість обрати один з двох типів маршруту: найкоротший або найшвидший, а також один з дев’яти видів транспорту, проте доступний лише англійською мовою.

1.3.5 OpenRouteService.org

OpenRouteService.org - це інший онлайн-сервіс, який використовує дані OSM [17]. Як і YourNavigation.org, це некомерційний сервіс. Служба використовує алгоритм A*, і повільніша на довгих маршрутах за інші аналоги. OpenRouteService.org доступна українською мовою, має можливість налаштування типів доріг, типу маршруту (найшвидший/найкоротший), типу транспортного засобу та його виду палива. Проте так само, як і інші аналогічні сервіси, не враховує типи проміжних пунктів маршруту та не дозволяє конфігурувати їх пріоритетність при побудові маршруту відносно вподобань користувача.

1.3.6 Порівняння сервісів

Результати порівняння п’яти сервісів маршрутизації наведено у таблиці 1.2. Для додатків, що працюють найшвидше, алгоритми маршрутизації виявилися конференційними, інші два додатки з відкритими алгоритмом використовують A*. Згідно результатів видно, що з п’яти досліджених сервісів лише два мають україномовну версію та жоден не підтримує побудову туристичних маршрутів.

Таблиця 1.2. Порівняння існуючих сервісів

Назва	Алгоритм	Джерело даних	Метод вибору точок та напряму	Підтримка української мови	Підтримка пріоритетів об’єктів мапи
TomTom	*конфіденційний*	Власні дані	Адреса	Немає	Немає
Google Maps	*конфіденційний*	Власні дані	Адреса або точка на карті	Є	Немає
Via Michelin	*конфіденційний*	Michelin maps	Адреса або точка на карті	Немає	Немає
YourNavigation.org	A*	OSM	Адреса або точка на карті	Немає	Немає
OpenRouteService.org	A*	OSM	Адреса або точка на карті	Є	Немає

ВИСНОВКИ ДО РОЗДІЛУ 1

У даному розділі було проведене порівняння алгоритмів побудови найкоротшого шляху таких як, A*, багаторівневий підхід, двонаправлений пошук, SMA* та інших, розглянуто їх переваги та недоліки. На сьогодні вже існують такі алгоритми побудови маршрутів, для яких час роботи буде значно меншим за час передачі даних мережею та графічного відображення маршруту. Це означає, що при виборі алгоритму буде доцільними оцінювати також обсяги пам’яті, що буде використовуватися. Таким чином для реалізації задачі за основу було обрано алгоритм A*, завдяки тому, що його використання надає можливість витримувати баланс між швидкодією та обсягами пам’яті, що використовується.

Також було проаналізовано існуючі рішення в області туристичної маршрутизації. За результатами порівняння виявлено, що жоден з додатків не підтримує побудову туристичних маршрутів. Таким чином, актуальною є розробка системи, яка б шукала шлях між точками, враховуючи області видимості проміжних пунктів. Цей підхід дозволить на основі алгоритмів побудови найкоротших маршрутів, виконати класифікацію об’єктів мапи та побудувати туристичний маршрут в автоматичному режимі.

РОЗДІЛ 2
ВИБІР ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ ТА РОЗРОБКА МОДЕЛІ

2.1 Вибір інструментів розробки програми

У наш час існує достатньо широкий вибір технологій та інструментів розробки програмного забезпечення.

У ґалузі розробки клієнт-серверних додатків найпопулярнішими мовами є Java, PHP, Python, Ruby, C#. Проте з них лише Java залишається актуальною протягом 20 років [18]. Саме цю мову і було обрано у якості основної мови розробки цього проекту за такі її переваги, як гнучкість, адже Java-додатки не залежать від платформи ані на рівні вихідного коду, ані на двійковому рівні, їх можна запускати у різних системах; також Java – мова об’єктно-орієнтовного програмування, що дозволяє створювати модульні додатки, вихідний код яких може використовуватися багаторазово, що значно економить ресурси розробника.

Ще одним інструментом розробки програми, з яким потрібно було визначитися. є інтегроване середовище розробки або IDE (Integrated Development Environment). Середовище розробки повинне включати в себе текстовий редактор, компілятор та\або інтерпретатор, засоби автоматизації зборки та відлагодження програм. Також у більшості сучасних середовищ наявна функція автодоповнення коду. Найпопулярнішими IDE для Java-розробки сьогодні є NetBeans, Intellij Idea, Eclipse.

NetBeans є інтегрованим середовищем розробки для Java, розробниками якої є Oracle та Apache Software Foundation, для мов програмування Java, JavaFX, C/C++, PHP, JavaScript та ін. За якістю і можливостям останні версії NetBeans IDE змагається з Intellij Idea та Eclipse, підтримуючи рефакторинг, профілювання, виділення синтаксичних конструкцій кольором, автодоповнення мовних конструкцій на льоту, шаблони коду та інше.

Intellij Idea – комерційне інтегроване середовище розробки на різних мовах програмування від компанії JetBrains. Окрім платної «Ultimate Edition», було випущено і безкоштовну версію під назвою «Community Edition». Community версія підтримує такі інструменти тестування як JUnit, системи контролю версій CSV, Subversion, Mercurial та Git, засоби зборки Maven, Ant, Gradle, мови програмування Java, Scala, Groovy та інші. Доступні засоби інтеграції з системами відстеження помилок Jira, Redmine, Pivotal Tracker та іншими.

Eclipse є інтегрованим середовищем розробки (IDE), що використовується в комп'ютерному програмуванні, і є одним з найбільш широко використовуваних Java IDE. Eclipse містить базове робоче місце і розширювану систему плагінів для налаштування середовища. Eclipse написано в основному на Java, і його основне призначення - розробка додатків Java.

Проаналізувавши переваги та недоліки трьох найпоширеніших середовищ, було вирішено обрати Intellij Idea, здебільшого за такі можливості як автоматичне виправлення та поліпшення якості коду, сучасному графічному інтерфейсу, стабільність плагінів, функції «гарячих клавіш», що економить час розробника.

2.2 Основні рішення з реалізації системи

Завдяки підходу об’єктно-орієнтованого програмування вдасться досягти незалежність реалізації від платформи та навіть архітектури системи. Основний функціонал системи буде надаватися одним або декількома класами, які по своїй ролі є сервісам і можуть використовуватися як сервіси і в консольному додатку, і в веб-додатках. Тобто пропонується створити деякий клас RouteService, який матиме відкрите API [19] побудови маршрутів та буде надавати можливість конфігурувати метод побудови за такими параметрами як тип транспорту, мова відображення інструкцій. Надалі цей сервіс можна буде підключити до контролера.

Зважаючи на розвиток та популярність веб-технологій, критично важливим є надати доступ до API користувачам у мережі Інтернет. Таким чином система має базуватися на клієнт-серверній архітектурі [20].

Клієнтом у даному випадку може виступати як веб-переглядач, так і інший сервіс.

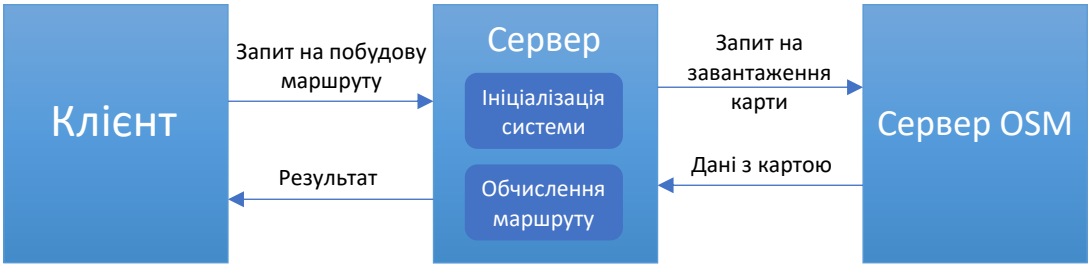


Рис. 2.1. Діаграма клієнт-серверної архітектури

Як показано на рисунку 2.1, клієнт надсилає запит на сервер, на якому запущена система. Система формує і надсилає запит на сервер OSM для завантаження карт. Сервер OSM надсилає карти серверу. Сервер обробляє карти та на їх основі та основі параметрів у запиті від клієнта формує маршрут та надсилає його клієнту у JSON форматі. Варто також зазначити, що запит до OSM сервера надсилається лише раз – при ініціалізації системи та завантаженні мапи.

Архітектура власне додатка також має поділятися на шари. Оптимальним та найбільш популярним підходом на сьогодні є архітектурний шаблон MVC [21] або Модель-представлення-контролер. Цей шаблон передбачає поділ системи на три взаємопов’язані частини: модель даних, вигляд (інтерфейс користувача) та модуль керування. Основною перевагою такого підходу є те, що зміна інтерфейсу користувача мінімально впливатиме на роботу з даними. Це особливо важливо у контексті даної системи, враховуючи, що, інтерфейсом можуть бути абсолютно різні по своїй сутності системи: веб-переглядач, сторонній сервіс, консоль.

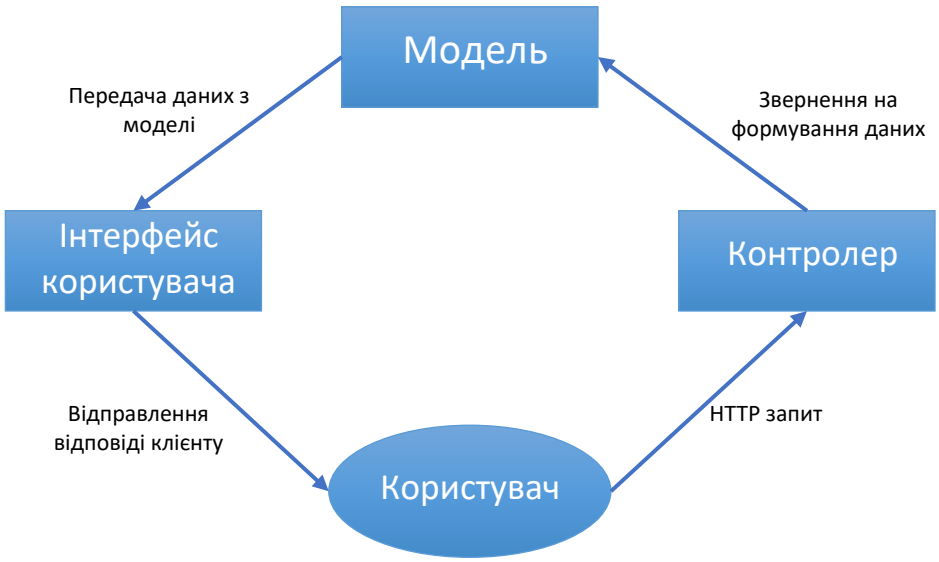


Рис. 2.2. Діаграма MVC архітектури

На практиці зазвичай дані від моделі вертаються у Controller перед відправленням на View для формування ModelAndView, проте у разі якщо це буде контролер, який лише повертатиме сформований JSON об’єкт, то пост-обробка даних буде не потрібна. Також варто зазначити, що контролер зазвичай ще виконує попередню перевірку даних, що надійшли з інтерфейсу користувача, і у разі якщо ці дані не валідні, повертає відповідну відповідь на View без звернення до моделі.

2.3 Модель даних

2.3.1 Формат вхідних даних

Основними перевагами XML формату є те, що він може бути відносно легко прочитаний людиною завдяки своїй структурі, не залежить від пристрою чи системи, якою буде оброблятися, широко поширений і, відповідно, до нього існує багато парсерів, які можна використати для перетворення файлу цього формату в об’єкт графу, проте вирішальним фактором став той факт, що саме у цьому форматі є карти у відкритому доступі – це карти від проекту

OpenStreetMap. Саме тому у якості вхідних даних у системі було вирішено використовувати OSM карти у XML форматі.

Всі об’єкти мапи у форматі OSM XML поділяються на 2 типи : <*node*>, які складаються з одної точки і описуються такими параметрами як довгота (latitude), широта (longitude) та ідентифікатор (node id), а також шляхи (<*way*>), які можуть складатися з 2 та більше точок (максимум 2000). Шлях може бути відкритим або замкненим. Ці типи об’єктів поєднуються між собою за допомогою відносин (<*relation*>). Будівлі, а точніше їх межі зазвичай описуються за допомогою замкнених шляхів. Це означає, що з OSM XML мапи можливо отримати координати меж будівель, що знаходяться в деякому радіусі від точки $x(lat, lon)$.

Спрощена структура OSM XML файлів наведена на рисунку 2.3. Як видно з діаграми, кожен з елементів way, node, relation містить мітки (<tag>), які мають ключ (<key>) та значення (<value>). Саме ці мітки містять у собі інформацію щодо типу об’єкта. Мітки описують особливості елементів карти. Обидва елементи є текстовими полями вільного формату, але часто являють собою числові або інші структуровані елементи.

Мітки представлені як пара ключ = значення. Ключі використовуються для опису теми, категорії або типу об'єкта (наприклад, магістралі чи імені). Ключі можна кваліфікувати за допомогою префіксів, інфіксів або суфіксів (зазвичай, розділених двокрапкою), утворюючи супер- або підкатегорії або простору імен. Загальними просторами імен є специфікація мови та специфікація простору імен дат для ключів імен.

Значення деталізує специфічну форму призначеної для ключа функції. Як правило, значення є текстом вільної форми (наприклад, $name = "Jeff\ Memorial\ Highway"$), одним з набору різних значень (перелік; наприклад, $highway = motorway$), кілька значень з переліку (розділених крапкою з комою), або число (ціле або десяткове), наприклад відстань.

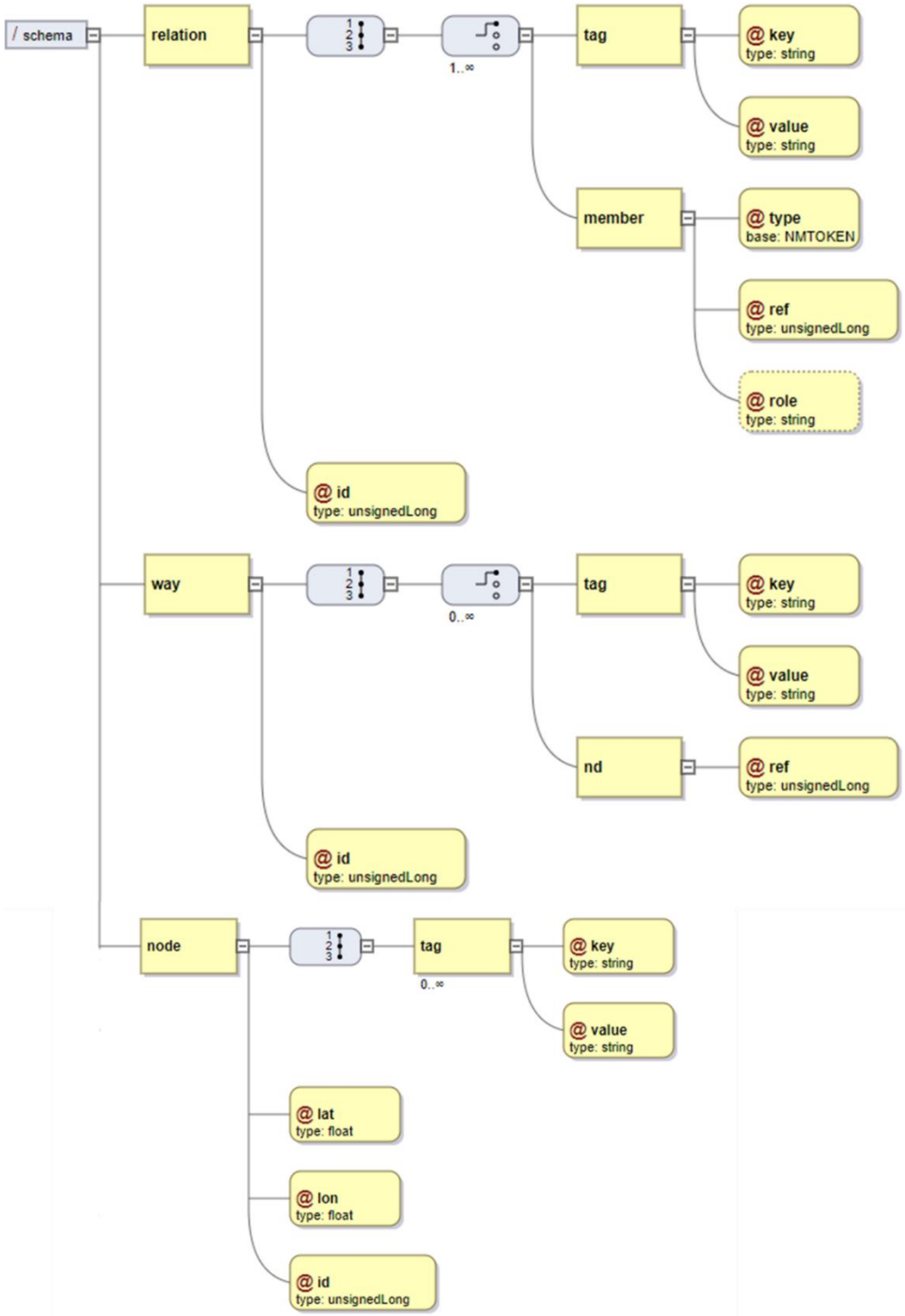


Рис. 2.3. Спрощена структура файлів типу OSM XML

Окрім власне карти на вхід до алгоритму мають прийти географічні координати початкової та кінцевої точок. Визначення географічного розташування зазвичай означає надання широти та довготи розташування. Числові значення широти і довготи можуть зустрічатися у вигляді різних одиниць або форматів: [22]

- шістдесяткова система числення: градуси, хвилини та секунди: 40° 26′ 46″ N 79° 58′ 56″ W
- градуси та десяткові хвилини: 40 ° 26,767 79 N 79 ° 58,933. W
- десяткові градуси: 40.446° N 79.982° W

При цьому постає проблема зберігання таких форматів координат, як десяткові градуси, у пам’яті та вибору відповідного типу даних. Радіус великої пів осі Землі на екваторі становить 6 378 137,0 метрів [23], тобто довжина окружності екватору – 40 075 161,2 метрів. Екватор ділиться на 360 градусів довготи, тому кожен градус на екваторі становить 111 319,9 метра або приблизно 111,32 км. У таблиці 2.1 наведено ступені точності в порівнянні з довжиною.

Таблиця 2.1. Ступені точності у порівняні з довжиною

Кількість знаків після коми	Десяткові градуси	Градуси, хвилини та секунди	Об'єкт, який можна однозначно визнати в такому масштабі	Довжина на місцевості на екваторі
0	1.0	1° 00′ 0″	Країна або великий регіон	111.32 км
1	0.1	0° 06′ 0″	Велике місто або район	11.132 км
2	0.001	0° 00′ 36″	Селище	1.1132 км
3	0.0001	0° 00′ 3.6″	Околиці, вулиця	111.32 м
4	0.00001	0° 00′ 0.36″	Окрема вулиця, земельна ділянка	11.132 м
5	0.000001	0° 00′ 0.036″	Окремі дерева	1.1132 м
6	0.0000001	0° 00′ 0.036″	Окремі люди	111.32 мм
7	0.00000001	0° 00′ 0.0036″	Практичні межі комерційних зйомок	11.132 мм
8	0.000000001	0° 00′ 0.000036″	Спеціалізована зйомка (наприклад, відображення тектонічних плит)	11.132 мм

Таким чином у рамках поставленої задачі необхідно забезпечити точність 6-7 знаків після коми. У мові Java для таких цілей найбільше підходять такі типи даних Double та BigDecimal.

2.3.2 Формат вихідних даних

Маршрут може бути представлений як набір вершин, через які він має проходити. Кожна вершина, як мінімум, має дві координати – довготу та широту. Тобто вихідними даними має бути масив пар координат. Проте окрім самого маршруту користувачеві буде цікаво також отримати інформацію про об’єкти, які охоплює маршрут. Тому найбільш зручним форматом для відображення та передачі цих даних між пристроями буде JSON – JavaScript Object Notation. JSON базується на тексті, може бути прочитаним людиною. Формат дозволяє описувати об'єкти та інші структури даних. Цей формат головним чином використовується для передачі структурованої інформації через мережу.

Приклад отриманого шляху у форматі JSON наведено нижче:

```
{
   "from":[
      30.4936954,
      50.4531868
   ],
   "to":[
      30.5182655,
      50.4450093
   ],
   "nodes":[
      {
         "name":"Парк Шевченка",
         "coordinates":[
            30.5146203,
            50.442955
         ]
      },
      {
         "name":"Володимирський собор",
         "coordinates":[
            30.5168836,
            50.4536958
         ]
      },
      {
         "name":"Театр російської драми ім. Лесі Українки",
         "coordinates":[
            30.5182655,
            50.4450093
         ]
      }
   ]
}
```

У прикладі кожен вузол шляху містить такі дані як назва та координати. Проте окрім назви можна передавати короткий опис об’єкту або посилання на сторінку в Вікіпедії чи офіційний сайт.

2.4 Розробка математичної моделі

2.4.1 Розробка алгоритму побудови найкоротшого шляху

Для вирішення поставленої задачі пропонується ввести наступне визначення функції $\dot{w}(e)$:

$$\dot{w}(e) = k(e) * w(e), \tag{2.1}$$

де $k(e)$ – функція, що визначає величину коефіцієнту k для кожного ребра e.

Значення k повинно вираховуватися відносно типу пункту на мапі, крізь область видимості якого проходить ребро e. При чому $k(e) \in (0,1]$, тоді $\dot{w}(e) \in (0, w(e)]$. Таким чином вдасться досягти врахування типу проміжного пункту при побудові маршруту і збільшення ймовірності входження пріоритетного об'єкта до результуючого маршруту, адже при однакових відстанях, вершина, ребро до якої проходить крізь область видимості пріоритетного об'єкту буде мати меншу величину $\dot{w}(e)$. Це означає, що при виборі вершини v з множини X такої, що $\hat{g}(v) := \min_{u \in X} \hat{g}(u)$ [24], де X: $=$ набір вузлів v, для яких шлях від s до v ще невизначений, буде обрана вершина з найменшим $\dot{w}(e)$.

Для кожного ребра e_i, що проходить через область видимості вершини v, $\dot{w}(e_i)$ буде дорівнювати вазі ребра, помноженій на коефіцієнт $k(v)$: $\dot{w}(e_i) = w(e_i) * k(v)$, а для інших ребер $\dot{w}(e_i) = w(e_i)$, оскільки для них $k = 1$. Вирахуване значення $\dot{w}(e_i)$ далі можна використати в алгоритмах побудови маршруту замість відстані між вершинами, які з’єднує ребро e_i, тобто його довжини.

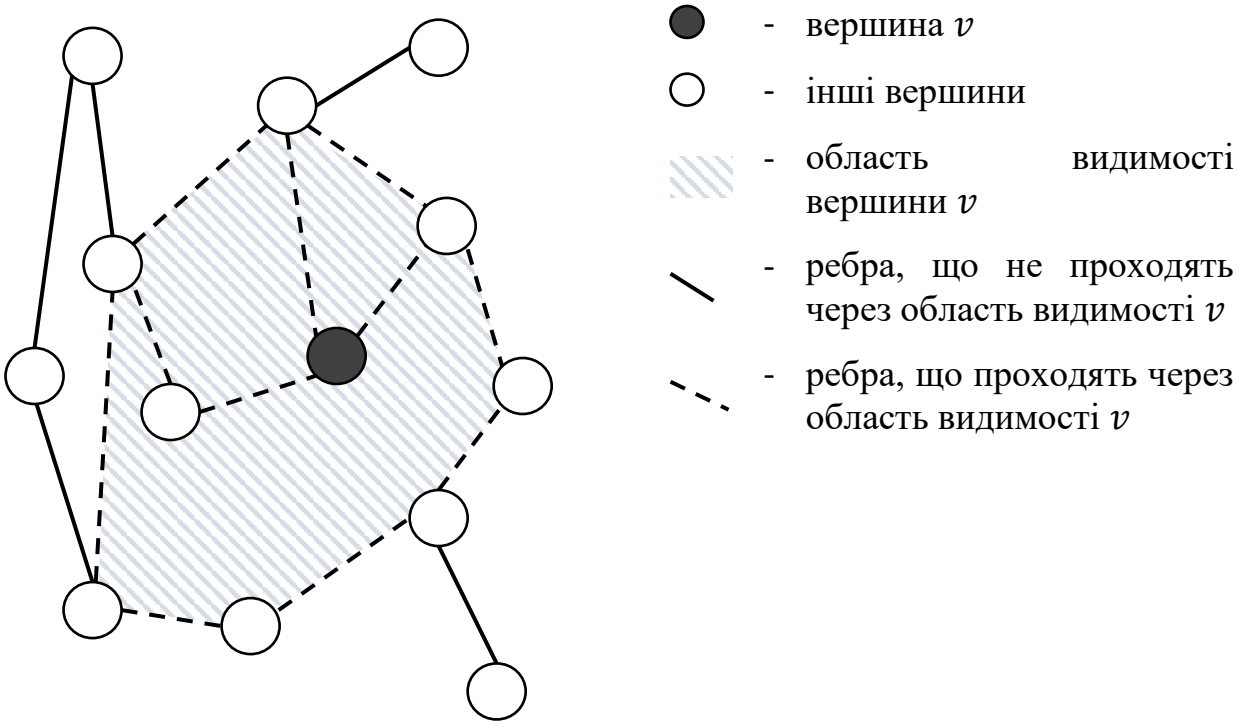


Рис. 2.4. Область видимості точки

2.4.2 Розрахунок області видимості об’єкта на мапі

При розрахунку маршруту знадобиться визначити область видимості кожного з пріоритетних об’єктів, тобто множину точок, з яких буде видно цей об’єкт на місцевості. Це особливо важливо в умовах щільної забудови міста, адже сусідні будівлі можуть перекривати вид на об’єкт.

Методом трасування променів [25] можна побудувати полігон, який і буде визначати область видимості об'єкта. Точкою початку всіх променів може бути центроїд об'єкта, але для більшої точності необхідно брати ще дві найвіддаленіші одна від одної точки, що знаходяться в межі об'єкта, відносно якого вираховується полігон, адже деякі будівлі можуть бути занадто протяжними.

На рисунку 2.5 наведено приклад побудови полігону області видимості. Для побудови полігону обирається точка x початку променів (на малюнку – центр об’єкта). Будується обмежуюче коло радіусом r, який є максимальною відстанню, з якої може бути видно об’єкт. З точки x пускається n променів в усіх напрямках.

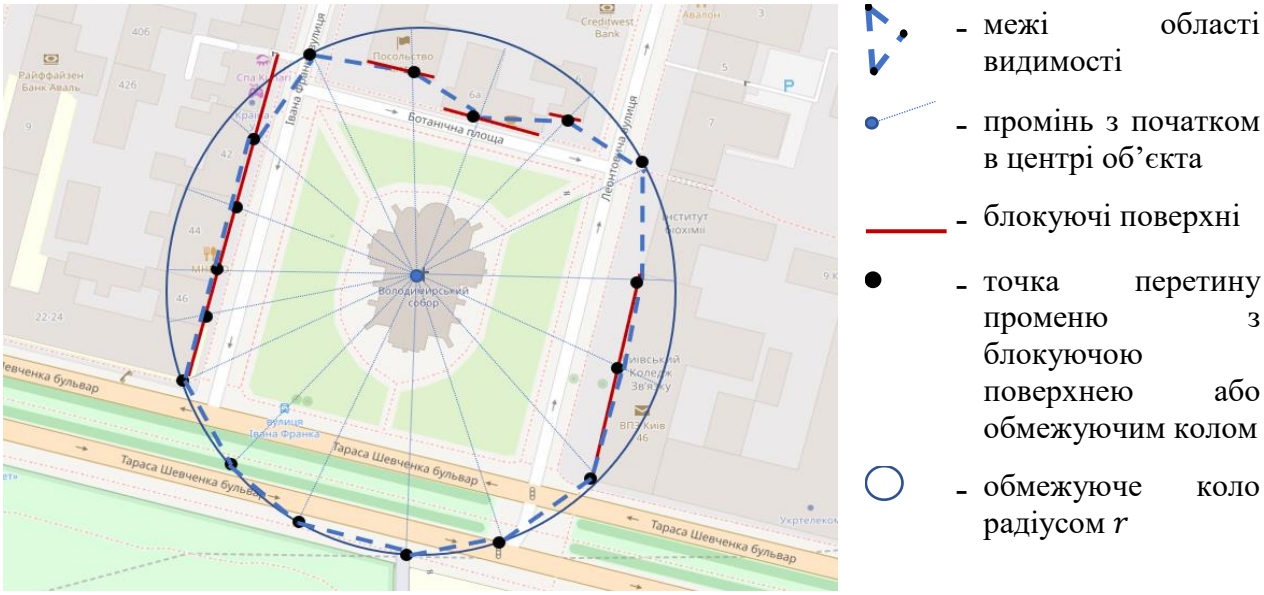


Рис. 2.5. Приклад побудови області видимості

Кожен з променів перетинає або обмежуюче коло, або блокуючу поверхню. Блокуючою поверхнею називається перша межа будівлі, яка зустрічається на шляху променю. Множина точок перетину утворює полігон, який і буде областю видимості.

2.4.3 Розрахунок коефіцієнта k

Більшість об'єктів, що є на OSM XML мапах, мають метадані окрім координат та ідентифікатора. Наприклад, одним з найкорисніших для визначення типу об'єкта може стати тег <*amenity*>. Даний тег використовується для позначення загальнодоступних об'єктів інфраструктури: банки, школи, лікарні, кінотеатри, театри, фонтани, тощо. Також, якщо орієнтуватися на побудову саме туристичних маршрутів, то стануть у нагоді такі теги як <*tourism*>, <*building*> зі значенням *"architecture"*. Якщо потрібно, щоб маршрут пролягав через парки та зелені зони, необхідно знайти всі об'єкти, що позначені тегом <*leisure*> зі значенням *"park"*. Після того, як всі пріоритетні об'єкти будуть знайдені, необхідно визначити рівень їх важливості, адже пам'ятки бувають різні, і мають різною мірою впливати на маршрут. Для прикладу, за відповідність кожній з умов можна нараховувати бали: +3 за наявність тегу <*historic*>, +4 за <*leisure*>

зі значенням *“park”,* +2 за посилання на Wikipedia зі статтею про цей об’єкт, +1 за кожен додатковий тег <*name*>. Суму балів необхідно конвертувати в коефіцієнт зі значенням в межах $(0; 1]$ і поставити у відповідність кожному полігону видимості значення коефіцієнта об’єкта, навколо якого цей полігон було побудовано.

2.5 Опис алгоритму побудови найкоротшого маршруту з урахуванням областей видимості

Алгоритм побудови найкоротшого маршруту з урахуванням областей видимості було організовано як позначено на рис 2.6.

Для запуску програми потрібно одноразово завантажити OSM мапу з OSM сервера. У якості альтернативи можна не завантажувати карту, а зберігати на жорсткому диску копію та використовувати її. Проте при цьому необхідно буде забезпечити регулярне оновлення цього файлу, адже правки можуть вноситися доволі часто і файл швидко втратить свою актуальність. У заголовку запита у якості параметрів необхідно вказати межі регіону, який буде завантажено. Зазвичай це прямокутник, що обмежує адміністративні кордони міста. Проте у разі, якщо це має бути інший регіон, має існувати можливість конфігурувати ці параметри без перезапуску сервера.

Після того, як карта була завантажена, виконується обчислення полігонів областей видимості, детальний алгоритм наведено на рисунку 2.5. Під час цього обчислення також відбувається фільтрація об’єктів мапи. Ті, що є більш пріоритетними за всі інші та будуть надалі брати участь у розрахунках, відбираються наступним чином: зчитуються теги об’єкта; на основі вмісту тегів обраховується $score$ об’єкта, якщо це значення більше, ніж деякий поріг, то об’єкт додається до переліку пріоритетних.

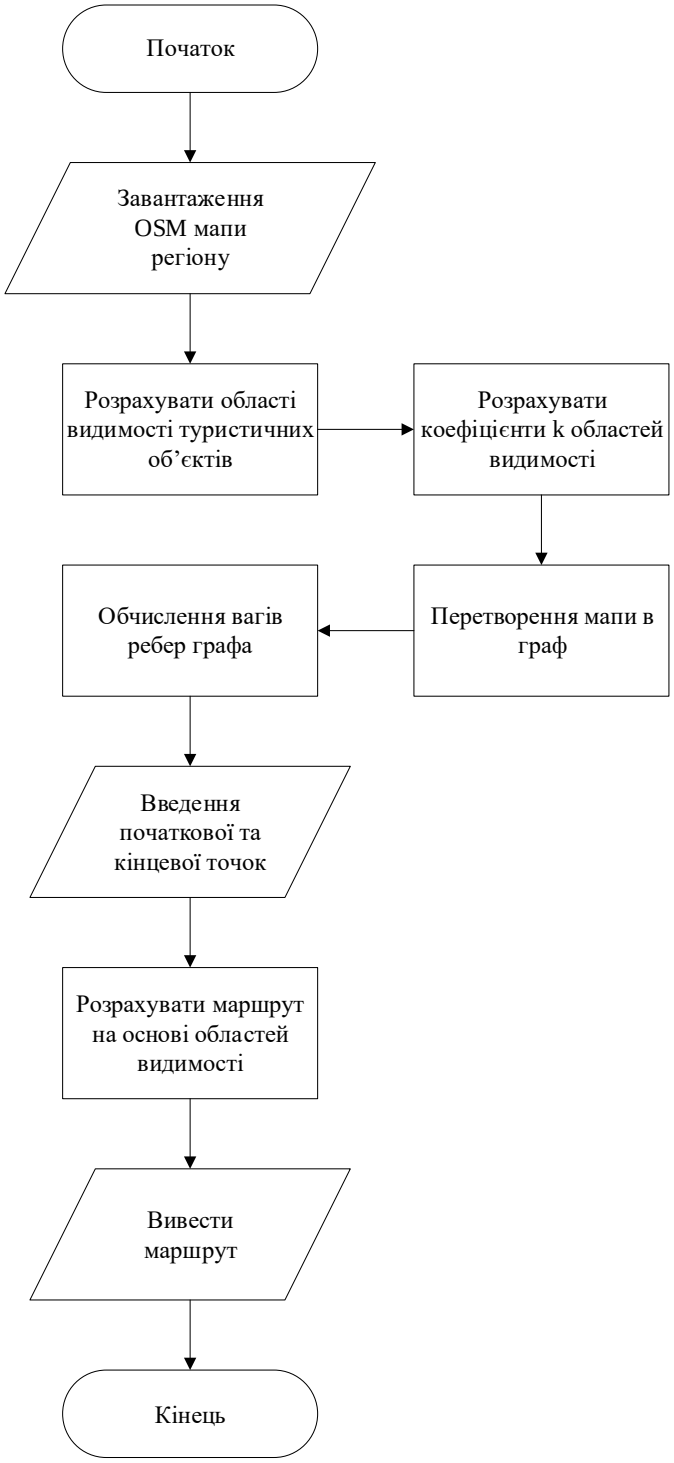


Рисунок 2.6. Схема алгоритму роботи програми

Одним з пунктів алгоритму є визначення множини блокуючих поверхонь. Це відбувається один раз протягом роботи всієї програми, і далі ця множин поверхонь зберігається в пам’яті, доки не будуть обчислені всі області видимості.

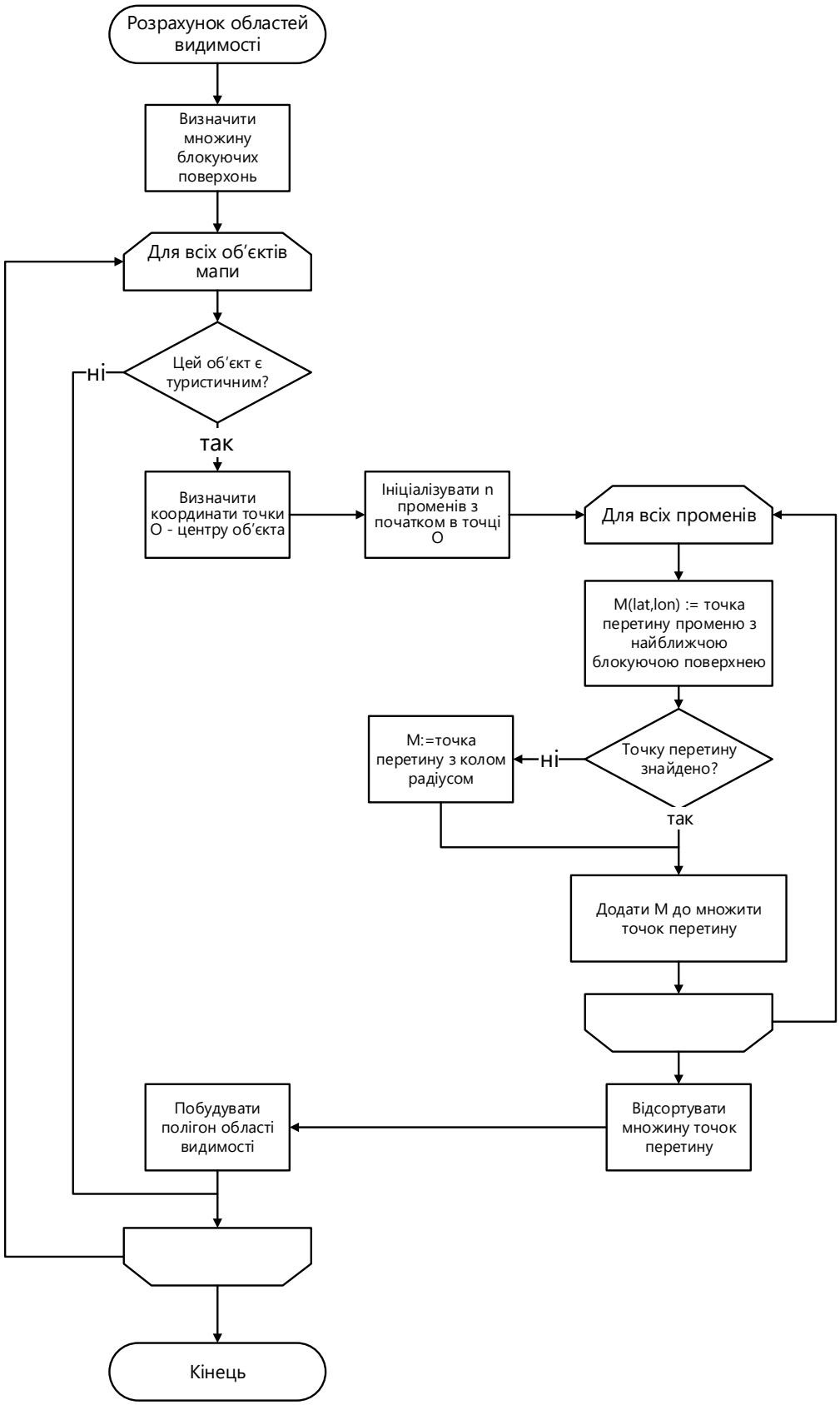


Рисунок 2.5. Схема алгоритму розрахунку областей видимості

Як видно з малюнку, після того, як точки перетину було знайдено, їх необхідно відсортувати. Це потрібно для того, щоб утворений з них полігон був простим, тобто будь-які його дві несуміжні сторони не мають мати спільних точок, ламана, що обмежує полігон не має перетинатися.

Для побудови графу необхідно також обчислити ваги ребер, при цьому використовується множина полігонів областей видимості, отримана на попередньому кроці. Для знаходження точки перетину використовується формула знаходження точки перетину відрізка і прямої.

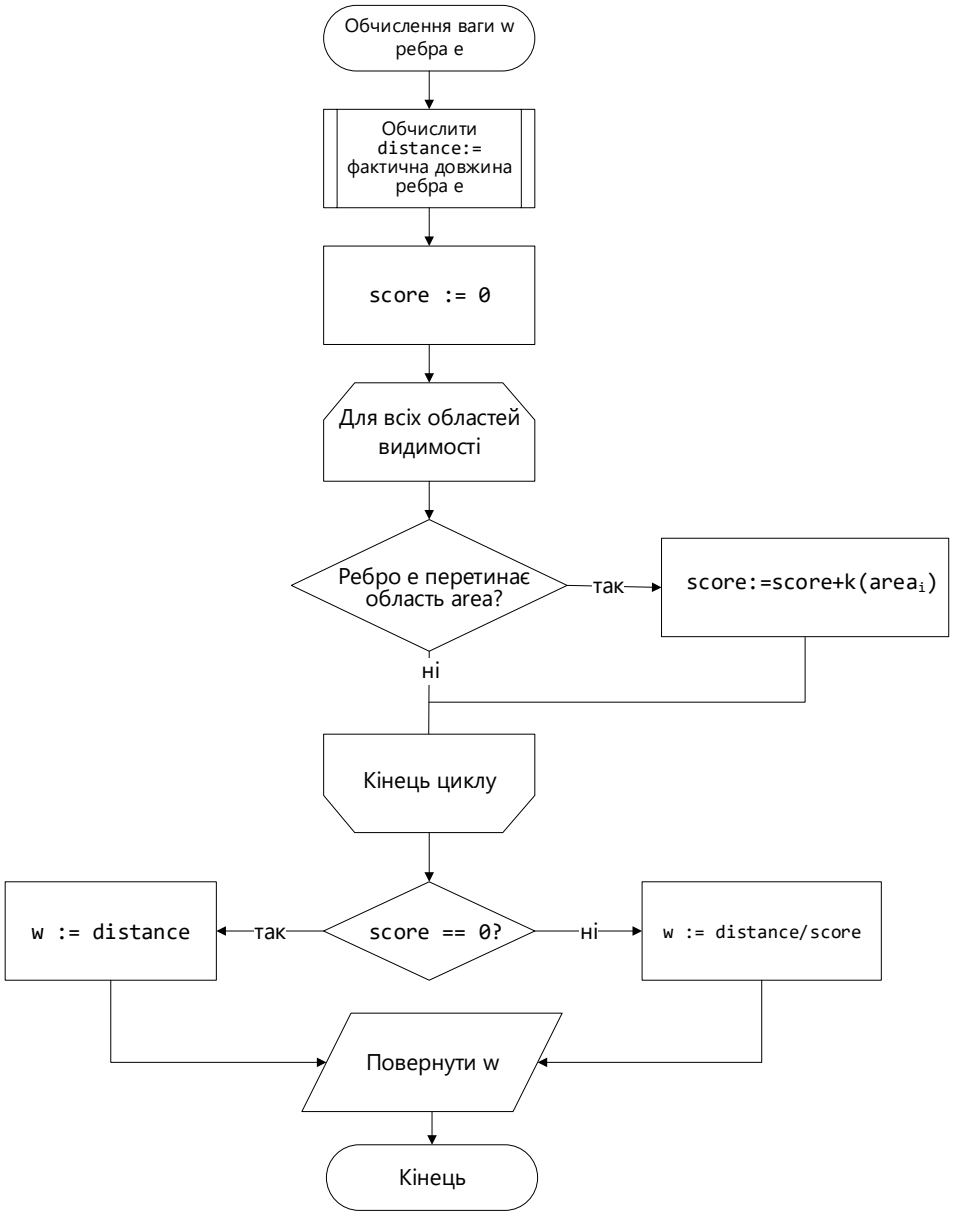


Рисунок 2.6. Схема алгоритму розрахунку ваги ребра

Цей алгоритм застосовується для кожного ребра у графі. Також варто зазначити, що перевірку входження чи перетину ребром області можна замінити перевіркою входження 2-3 контрольних точок ребра в цю область. Контрольними точками ребра можуть 2 крайні та точка середини. Для кожної точки додаються бали за кожну область, в яку вона входять, сума балів після закінчення циклу ділиться на кількість точок. Така заміна може мати місце саме тому, що через особливості представлення даних на OSM картах, довжина ребра у середньому буде у кілька разів меншою за діаметр полігону області видимості. Тому цих декількох контрольних точок буде достатньо, щоб перевірити, чи проходить ребро крізь полігон.

2.6 Аналіз методів рішення

2.6.1 Розрахунок обчислювальної складності алгоритму

Зважаючи на велику кількість об’єктів мапи, важливо розрахувати та проаналізувати обчислювальну складність алгоритму.

Нехай на мапі є n об’єктів та m ребер між ними у тому регіоні, який буде оброблятися алгоритмом. Тоді для виконання розрахунку областей видимості необхідно виконати n циклів для кожного з k променів та m ребер. Проте ця кількість може бути зменшена, якщо попередньо відфільтрувати блокуючі поверхні, лишаючи лише ті, що знаходяться в межах обмежуючого кола, зменшивши порядок цієї операції з $O(n^2)$ до $O(n)$. При цьому ще попередньо потрібно зробити n циклів, щоб зібрати в один масив усі блокуючі поверхні. Також при обчисленні вагів ребер для кожного ребра потрібно перевірити його входження у кожну з областей, тобто ще $m * n$ областей.

Якщо за основу побудови алгоритму найкоротшого шляху взяти алгоритм Дейкстри або A*, то складність цього алгоритму можна оцінити як $O(n * log(n) + m + (1 + k + m) * n)$ або, враховуючи, що k-константа, $O(n * (log(n) + m) + m)$.

Зважаючи на таку обчислювальну складність є сенс розглянути варіант обробки деяких кроків алгоритму у паралельних потоках, що могло б допомогти менше навантажувати систему. Позитивним моментом є те, що виконувати операції, складність яких $O(n * m)$ потрібно тільки один раз при ініціалізації програми. Далі, під час власне роботи та обробки запитів від користувачів буде лише використовуватися алгоритм побудови маршрутів, складність якого можна звести до $O(n * log(n) + m)$.

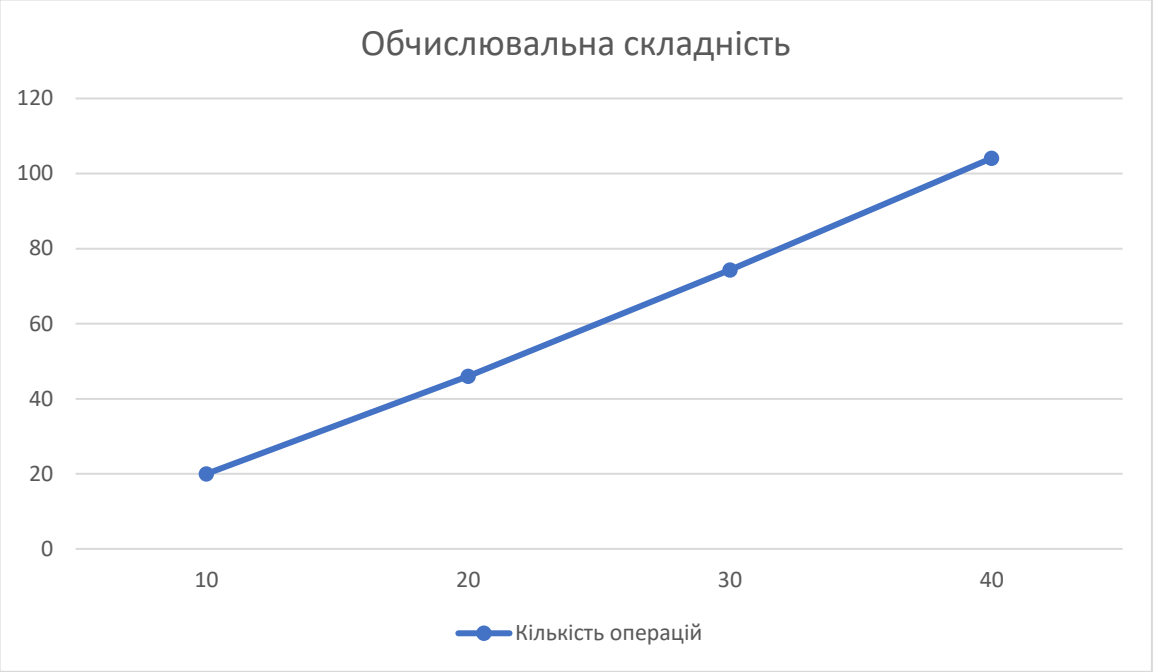


Рис. 2.8. Графік функції обчислювальної складності алгоритму

ВИСНОВКИ ДО РОЗДІЛУ 2

У даному розділі було обрано засоби вирішення задачі та розроблено модель її математичну модель. Система буде побудована на клієнт-серверній архітектурі та матиме відкритий прикладний інтерфейс за допомогою якого користувачі або сторонні додатки зможуть обраховувати туристичні маршрути. Джерелом даних для додатку було обрано OSM карти.

Також було розроблено алгоритм побудови найкоротших маршрутів, що при обчисленні класифікує об’єкти мапи за типом на основі властивостей, описаних в OSM XML, будує навігаційний граф та на основі алгоритмів пошуку найкоротшого шляху та обчислює туристичний маршрут.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ВИРІШЕННЯ ЗАДАЧІ ТА РОЗРОБКА ПРОГРАМИ

3.1 Модель інструменту побудови маршрутів

3.1.1 Модель даних

Враховуючи багатошарову структуру системи було вирішено визначити DTO для передачі даних між шарами системи. Результатом роботи системи є маршрут, тому було створено тип даних Route (рис. 3.1), яким оперує контролер. Route містить усю основну інформацію про маршрут, а саме:

- початкова (from) та кінцева (to) точки
- тривалість маршруту (time)
- довжина маршруту (distance)
- координати точок, з яких складається маршрут (coordinates)
- пам’ятки, які охоплює маршрут (points)
- інструкції (instructions)

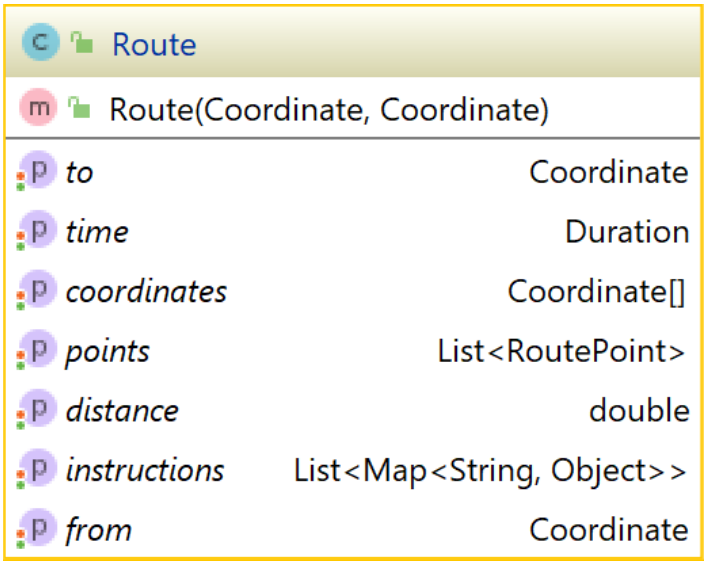


Рис. 3.1. Діаграма класу Route

Типом даних, що описує об’єкт на карті, є PathPoint (рис. 3.2) . Екземпляри цього класу містять дані про назву об'єкта, координати, теги та бали, які нараховуються кожному об'єкту при ініціалізації системи.

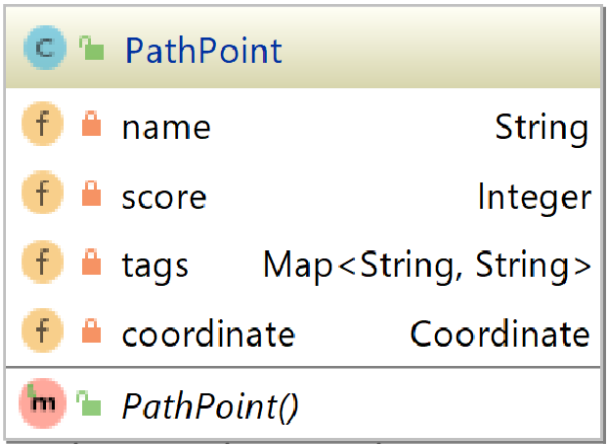


Рис. 3.2. Діаграма класу PathPoint

Окрім даних, що будуть відображені користувачеві, потрібно ще зберігати інформацію про об'єкти, яка буде використовуватися під час обрахунку маршруту. Саме для цього було створено клас ParsingGeometry (рис. 3.3), який є нащадком PathPoint та окрім перерахованих вище властивостей, у ньому також міститься геометрія об’єкту (geometry) та полігон видимості (scopePolygon).

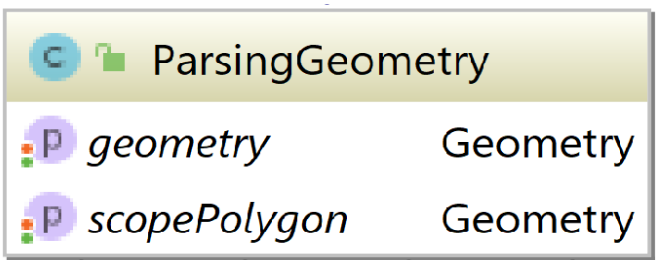


Рис. 3.3. Діаграма класу ParsingGeometry

Також для зручності зберігання та використання ребра графу у контексті створеної системи було визначено клас Edge, що складається з двох координат, які є його початковою та кінцевою точками. На рисунку 3.4 наведено діаграму класу. Для зберігання координат точки у програмі використовується клас Coordinate з пакету com.vividsolutions.jts.geom.

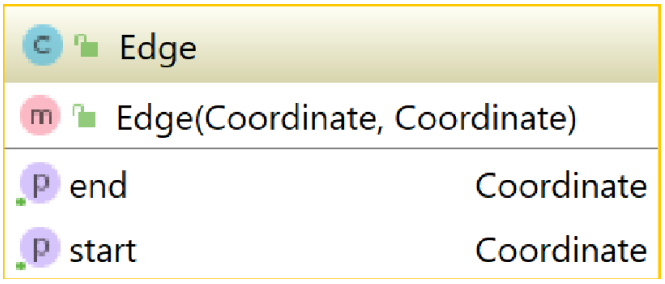


Рис. 3.4. Діаграма класу Edge

3.2 Реалізація інструменту побудови маршрутів

3.2.1 Реалізація сервісу побудови маршрутів

Сервіс побудови маршрутів RouteService реалізовано на основі шаблону проектування «Фасад». В основі цього шаблону лежить ідея приховати складну структуру системи. Фасад може бути спрощеним відображенням системи та надає клієнту швидкий доступ саме до тих функцій, які потрібні користувачеві та переадресовує виклики на ті сервіси, які безпосередньо відповідають за цей функціонал.

Основним методом цього сервісу, який викликається у контролері, є метод buildRoute (), що приймає у якості параметрів координати кінцевої та початкової точок “from”, “to” та повертає маршрут. Даний клас містить у собі посилання на об’єкт типу GraphHopper, у методі buildRoute() відбувається формування запиту GHRequest та його конфігурація такими параметрами, як тип транспорту, метод нарахування ваги ребер графа, локаль. Після чого цей запит надсилається у GraphHopper, який і обчислює шлях та повертає його у вигляді GHResponse, з якого можливо отримати лише перелік координат, з якого складається маршрут, довжину та тривалість маршруту.

Проте для формування відповіді необхідно більше даних. Через те, що основну роботу по обробці мапи виконує MapParser, то і масив елементів ParsingGeometry зберігається в ньому. Отже підключивши MapParser, можна отримати інформацію про відвідані точки, їх назви, теги, тощо.

Також даний сервіс виконує роботу по обробці помилок. Якщо при побудові маршруту виникне exception, то вона буде оброблена: в консоль сервера виведеться повідомлення про помилку та сервіс поверне пустий маршрут.

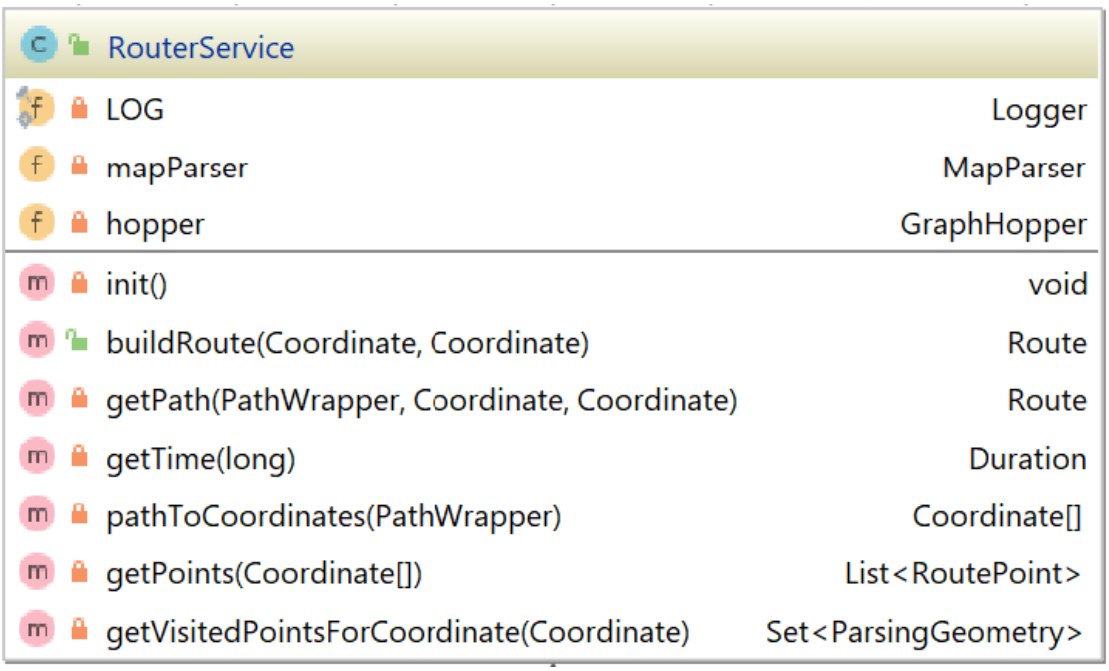


Рис. 3.5. Діаграма класу RouterService

3.2.2 Реалізація інструменту перетворення карти в граф

У якості інструменту перетворення OSM XML мап в об'єкт графу координат було обрано GraphHopper насамперед за можливість перевизначити метод обчислення ваги ребер. GraphHopper - це швидка бібліотека маршрутизації з відкритим вихідним кодом і сервер, написаний на Java, призначена для серверів, настільних ПК, а також для мобільних пристроїв з Android та iOS. Для побудови маршрутів можуть використовуватися такі алгоритми як Дейкстри, A* та Контракційні ієрархії [26].

GraphHopper надає можливість підключити їхню бібліотеку до проекту та перевизначити класи за власним призначенням. Саме тому цей інструмент підходить для вирішення поставленої задачі.

Таким чином було створено клас CustomGraphHopper у якому перевизначено метод обчислення ваги ребер. CustomGraphHopper за умовчанням

обраховує вагу за допомогою класу CustomWeighting. На рисунку 3.6 наведено діаграму класів CustomGraphHopper та CustomWeighting та їхню взаємодію.

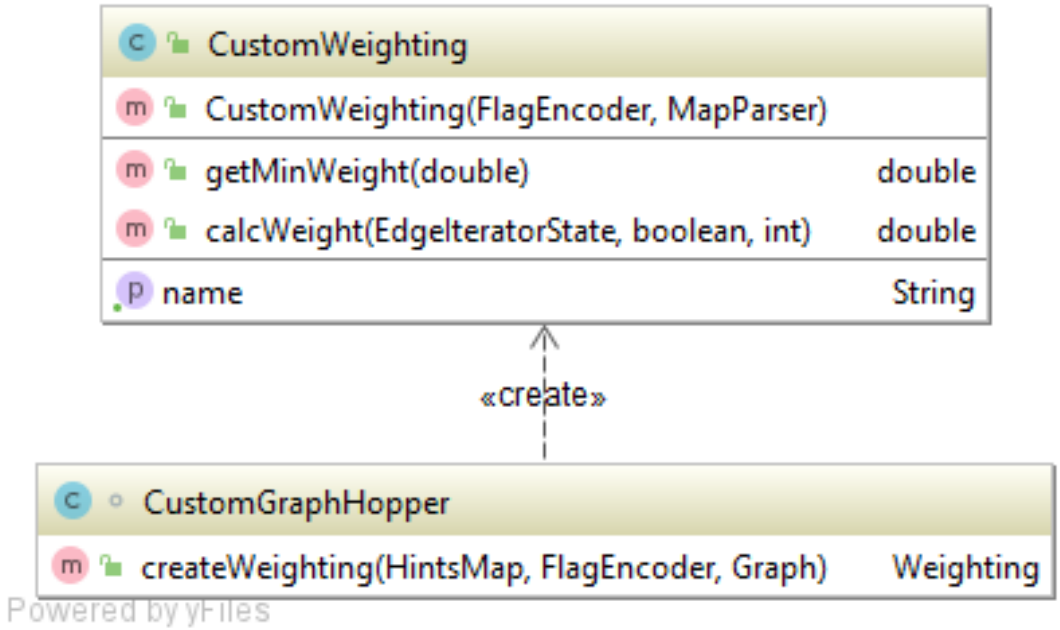


Рис 3.6. Діаграма класів CustomGraphHopper та CustomWeighting.

3.2.3 Реалізація методу нарахування ваги ребра

Однією з ключових сутностей даного проекту є клас CustomWeighting, що є нащадком класу AbstractWeighting з пакету com.graphhopper.routing.weighting. У цьому класі перевизначається логіка нарахування ваги ребра при побудові географічного графу.

У якості параметру у метод calcWeight() передається ребро типу EdgeIteratorState. Це ребро конвертується у набір точок, що йому належать. Зазвичай це 2-3 точки, кількість залежить від методу конвертації. Після цього для кожної з цих точок нараховуються бали, що фактично є сумою коефіцієнтів областей видимості до яких ця точка належить, та вираховується середній бал ребра як середнє арифметичне всіх балів точок ребра. Якщо середній бал не дорівнює нулю, то результатом є фактична довжина ребра, поділена на середній бал, інакше – довжина ребра.

3.2.4 Реалізація способу обчислення областей видимості

Реалізацію способу обробки карти містить у собі клас MapParser. Основний процес обрахунку областей видимості відбувається при завантаженні екземпляра класу. Під час ініціалізації класу формується та надсилається запит до серверу $www.overpass-api.de$ зі вказанням меж мапи, які потрібно вивантажити у пам'ять. Після цього формується перелік об'єктів, що будуть більш пріоритетними за інші при побудові маршруту. Фільтрація відбувається у методі validateEntityAndAddToGeometryList(). За допомогою ScoreCounter сервісу вираховується $score$ на основі тегів об'єкту. Якщо це значення не більше за MIN_SCORE, для об'єкту будується область видимості і результат записується у ParsingGeometry, яка додається до списку parsingGeometries.

Реалізація побудови знаходиться у методі getScopePolygon(). У цьому методі з усіх блокуючих поверхонь вибираються ті, що знаходяться в межах RADIUS від центру об'єкту. Після цього обраховуються координати перетину променів з центру об'єкта з блокуючими поверхнями або колом з радіусом RADIUS. Реалізація цієї дії знаходиться у статичному методі CoordinateService.getCoordinatesOfIntersection(). Масив координат точок перетину сортується у напрямі руху годинникової стрілки. З цього масиву утворюється полігон, який у об'єднанні з полігоном об'єкту утворює полігон області видимості.

Масив parsingGeometries надалі використовується при обчисленні ваги ребра. У метод overlaps() надходить координата. Для всіх ParsingGeometry з parsingGeometries перевіряється входження координати у полігон області видимості. Якщо так, то значення $score$ цієї геометрії додається до загальної суми $score$ ребра. Діаграма класу MapParser наведена на рисунку 3.7.

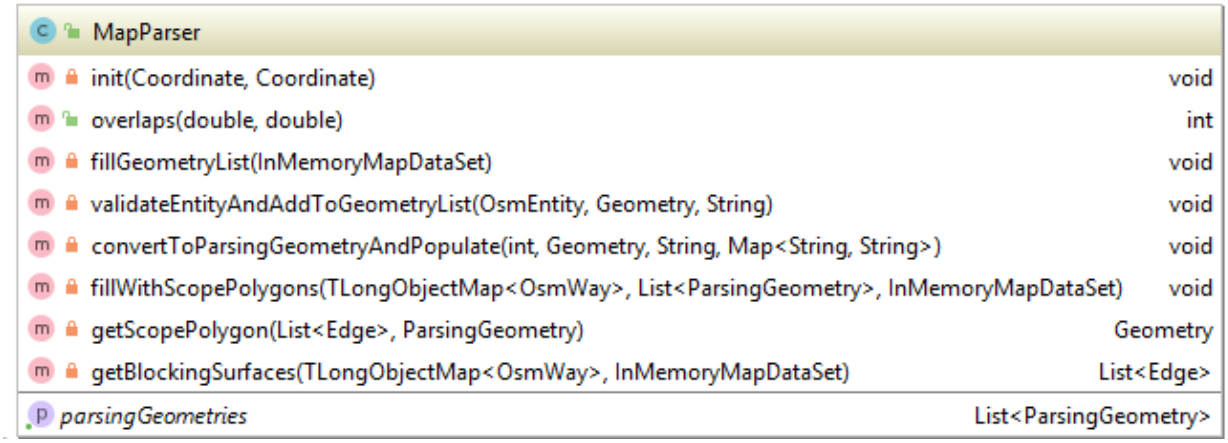


Рис. 3.7. Діаграма класу MapParser.

3.2.5 Реалізація алгоритму сортування вершин

Для коректної побудови полігону з набору точок їх необхідно відсортувати для того, щоб утворений з них багатокутник був простим. Зважаючи, що всі точки мають географічні, а не декартові координати, це є окремою підзадачею. Її реалізація знаходиться у методі filterAndSortCoordinates(), що в CoordinateService. Діаграма класу CoordinateService наведена на малюнку 3.8.

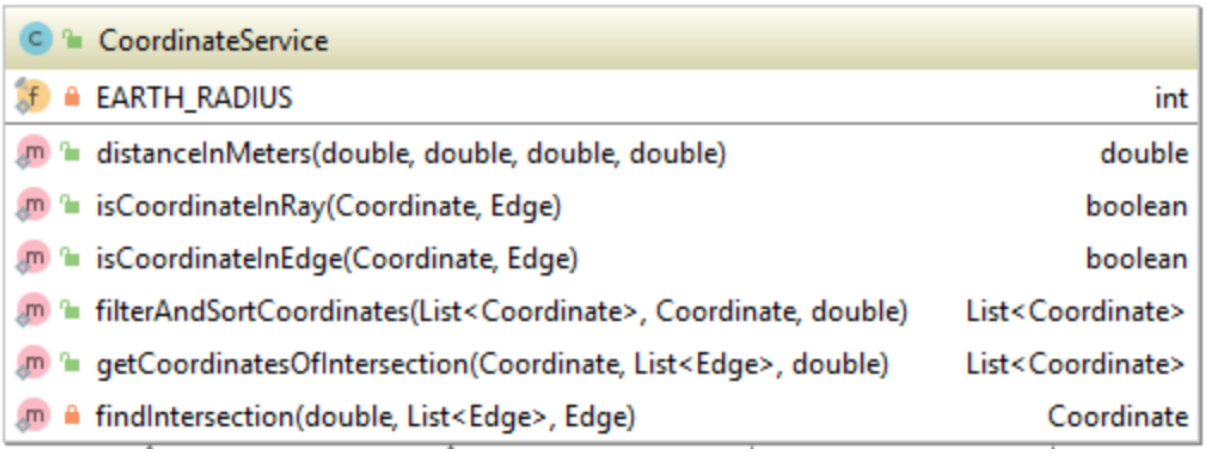


Рис. 3.8. Діаграма класу CoordinateService.

Сортування точок відбувається за напрямком руху годинникової стрілки. В метод передаються такі параметри як масив координат, та координати центру кола в межах якого знаходяться всі точки і відносно якого буде відбуватися сортування. Блок схема алгоритму сортування наведена на рисунку 3.9.

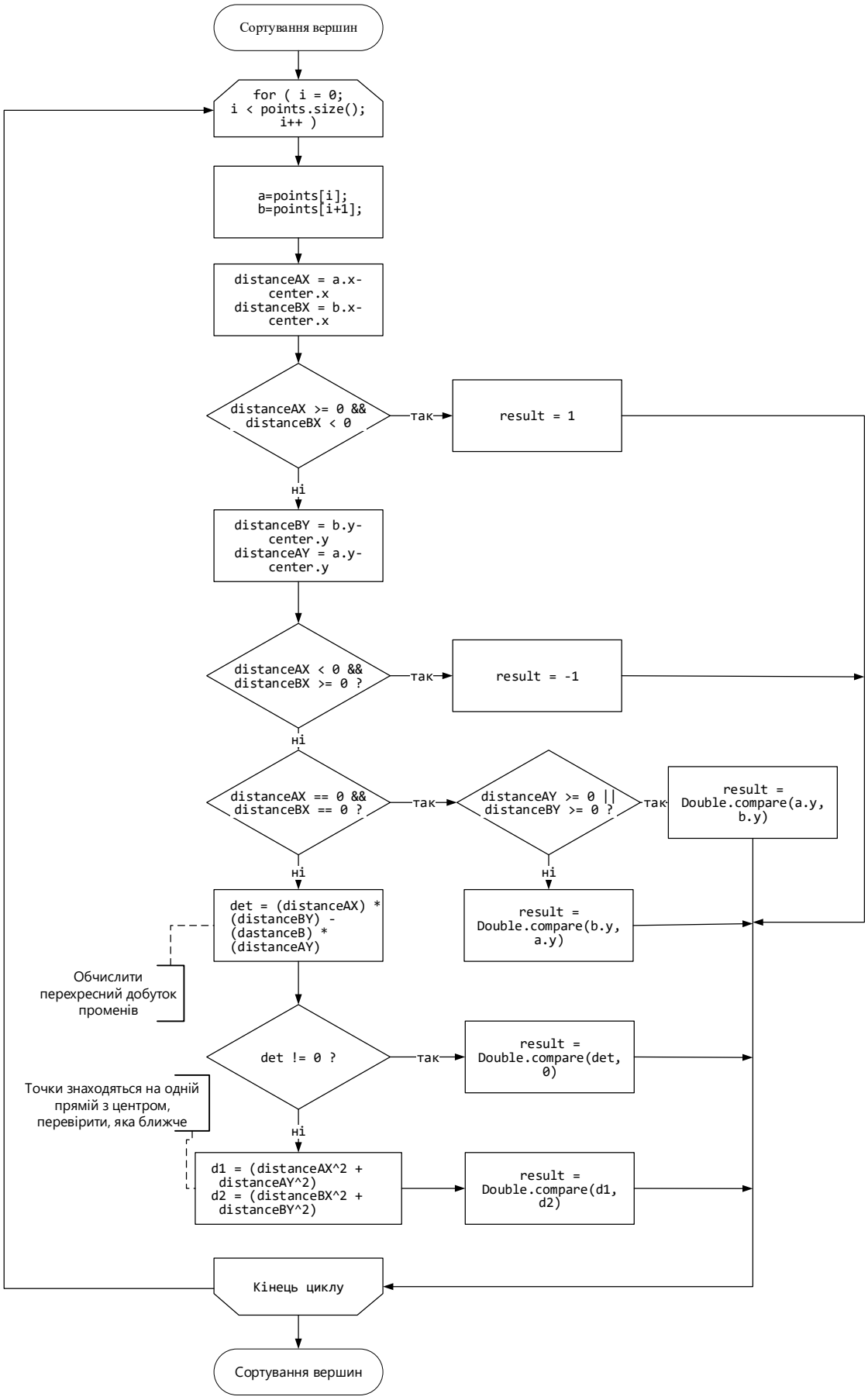


Рис. 3.9. Схема алгоритму сортування точок

При сортуванні попарно порівнюються точки a_i та a_{i+1}. Перш за все обчислюється різниця між точкою та центром по координаті x. Якщо для точки a_i ця різниця невід'ємна, при цьому для a_{i+1} від’ємна, то результатом буде $a_i > a_{i+1}$, якщо навпаки, то $a_i < a_{i+1}$. Якщо жодна з умов не була виконана, то обчислюється різниця між точкою та центром по координаті y. Якщо обидві з цих різниць дорівнюють 0, тоді результатом буде різниця між a_i.y та a_{i+1}.y. Інакше необхідно обчислити добуток перетину векторів від центру до кожної з двох точок. Якщо цей добуток не дорівнює 0, то він і буде результатом. Інакше точки лежать на одній прямій з центом і потрібну обрати ту, що буде найближчою до центру. Отриманий результат використовується у компараторі при сортування масиву.

3.3 Застосування інструменту побудови маршрутів

Задля демонстрації можливостей програми та тестування алгоритму було створено клас RouteController, що надає доступ користувачеві до відкритого API системи. Це клас позначено анотацією @Controller, що означає, що його буде завантажено у Spring Application Context як контролер, тобто клас, що обробляє запити користувача, викликаючи відповідні методи. У цей клас будуть перенаправлятися усі запити, що стосуються URL-адреси «/route». Це сконфігуровано за допомогою анотації @RequestMapping. Також у RouteController містяться методи з такими адресами як "/build", "/restart", "/configure". Тобто, за посиланням “route/build” буде доступним метод buildRoute() у тілі якого відбувається виклик RouterService.buildRoute(). З моделі користувачеві повернеться побудований маршрут за координатами з параметрів запиту. Важливо зазначити, що метод викликається за GET-запитом, а не POST. Це зроблено для того, щоб запит містив координати початкової та кінцевої точок у заголовку. Таким чином користувач зможе поділитися посиланням на побудований маршрут.

У якості прикладу було створено односторінковий сайт, що наглядно демонструє можливості роботи програми, приклад наведено на рисунку 3.10. Під час завантаження серверу та ініціалізації мапи для визначеного регіону масив об’єктів типу ParsingGeometry записується до файлу у форматі OSM Json. Це дозволяє не обчислювати туристичні об’єкти повторно при кожному завантаженні сторінки, адже ця дія могла б значно збільшити час очікування від сервера. Отже, використовується попередньо підготовлений файл з даними, що потрібні для відображення областей видимості та інформації про об’єкт на сторінці.

При переході на сторінку відображається мапа та налаштування координат. Після того, як користувач обрав початкову та кінцеву точки на карті та натиснув кнопку «Побудувати маршрут», значення координат відправляються на контролер. У методі контролера відбувається валідація параметрів, і, якщо вони відповідають встановленому формату десяткових координат, викликається метод побудови маршруту сервісу RouteService. Результат обчислень повертається у контролер і додається у модель, що далі відправляється на Web-сторінку. На веб сторінці дані обробляються та маршрут додається до мапи як окремий шар. Також окремим шаром на карті відображаються області видимості туристичних об’єктів.

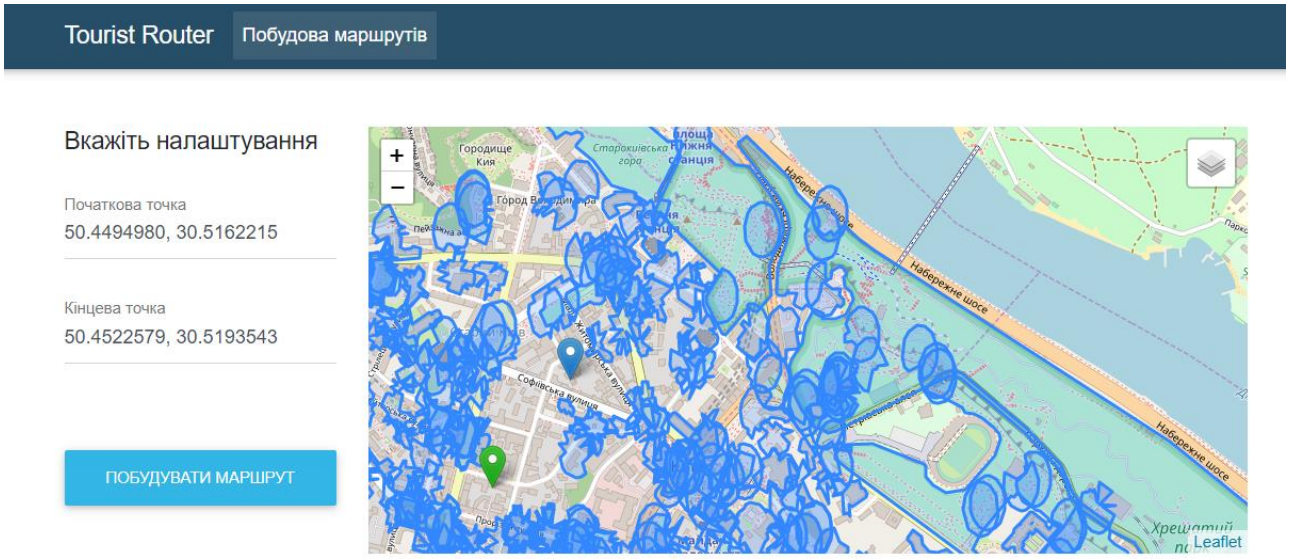


Рис. 3.10. Приклад відображення туристичних пам’яток

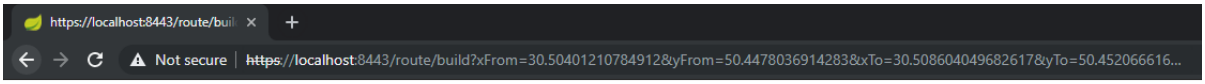
Для кожного з цих об’єктів визначена функція, яка викликається на клік по об’єкту правою кнопкою миші та генерує підказку з інформацією про пам’ятку. Також при виборі точки додається маркер з координатами у місці кліку миші.

3.4 Інструкція користувачеві

Розроблена програма надає можливість будувати маршрути, базуючись на областях видимості туристичних об’єктів регіону та координат початкової та кінцевої точок. При першому запиті до програми у пам’яті вже підготовлений граф маршрутів. Для побудови маршруту необхідно перейти за посиланням «/route/build». Початкова та кінцева точки маршруту мають передаватися у заголовку запиту у якості параметрів. Таким чином сформований запит має відповідати наступному формату: “{site_name}/route/build?xFrom={from.x}&yFrom={from.y}&xTo={to.x}&yTo={to.y}”, де {site_name} – адреса серверу, на якому розгорнуто систему, {from.x} – широта початкової точки, {from.y} – довгота початкової точки, {to.x} - широта кінцевої точки, {to.y} – довгота кінцевої точки. Координати мають передаватися у форматі десяткових градусів.

Результатом роботи програми буде маршрут у форматі JSON, приклад якого наведено на рисунку 3.11.

Окрім координат цей об’єкт містить інформацію щодо довжини маршруту, часу, що потрібен на його подолання, перелік пам’яток, які він охоплює, а також інструкції з навігації. Мову інструкцій можна встановити за допомогою параметру locale. Локаль необхідно передавати у наступному форматі: мова _КРАЇНА. Наприклад, de_GE, де de – код німецької мови, GE – код Німеччини. У такому випадку запит на сервер матиме такий вигляд: https://localhost:8080/route/configure?locale=de_GE за умови, що сервер запущено на локальній машині на порту 8080.




```
{"from":{"x":30.50401210784912,"y":50.4478036914283,"z":"NaN"},"to":{"x":30.508604049682617,"y":50.45206661617709,"z":"NaN"},"distance"
{"standardDays":0,"standardHours":0,"standardMinutes":11,"standardSeconds":703,"millis":703672},"points":[{"name":"Яновський Юрій","sco
{"historic":"memorial","name":"Яновський Юрій","memorial":"plaque","wikipedia":"uk:Яновський Юрій Іванович","wikidata":"Q928940"},"coor
{"x":30.5037701,"y":50.4477964,"z":"NaN"}},{"name":"Усенко Павло","score":7,"tags":{"historic":"memorial","name":"Усенко
Павло","memorial":"plaque","wikipedia":"uk:Усенко Павло Матвійович","wikidata":"Q4477673"},"coordinate":{"x":30.5037625,"y":50.4477776,
Микола","score":7,"tags":{"historic":"memorial","name":"Шпак Микола","memorial":"plaque","wikipedia":"uk:Шпак Микола
Іванович","wikidata":"Q12172266"},"coordinate":{"x":30.5037665,"y":50.4477874,"z":"NaN"}},{"name":"Дубов Микола","score":7,"tags":
{"historic":"memorial","name":"Дубов Микола","memorial":"plaque","wikipedia":"uk:Дубов Микола Іванович","wikidata":"Q4169354"},"coordin
{"x":30.5036553,"y":50.4477601,"z":"NaN"}},{"name":"Кулик Іван","score":7,"tags":{"historic":"memorial","name":"Кулик
Іван","memorial":"plaque","wikipedia":"uk:Кулик Іван Юліанович","wikidata":"Q3917403"},"coordinate":{"x":30.5036928,"y":50.4477541,"z":
Микола","score":7,"tags":{"historic":"memorial","name":"Руденко Микола","memorial":"plaque","wikipedia":"uk:Руденко Микола
Данилович","wikidata":"Q3847375"},"coordinate":{"x":30.5037566,"y":50.447763,"z":"NaN"}},{"name":"Гончар Олесь","score":7,"tags":
{"historic":"memorial","name":"Гончар Олесь","memorial":"plaque","wikipedia":"uk:Гончар Олесь Терентійович","wikidata":"Q2388302"},"coo
{"x":30.5036378,"y":50.447763,"z":"NaN"}},{"name":"Рибак Натан","score":7,"tags":{"historic":"memorial","name":"Рибак
Натан","memorial":"plaque","wikipedia":"uk:Рибак Натан Самійлович","wikidata":"Q4401390"},"coordinate":{"x":30.5037527,"y":50.4477534,"
Іван","score":7,"tags":{"historic":"memorial","name":"Кочерга Іван","memorial":"plaque","wikipedia":"uk:Кочерга Іван
Антонович","wikidata":"Q4236694"},"coordinate":{"x":30.5036845,"y":50.4477554,"z":"NaN"}},{"name":"Первомайський Леонід","score":7,"tag
{"historic":"memorial","name":"Первомайський Леонід","memorial":"plaque","wikipedia":"uk:Первомайський Леонід Соломонович","wikidata":"
{"x":30.5037406,"y":50.4477464,"z":"NaN"}},{"name":"Чабанівський Михайло","score":7,"tags":{"historic":"memorial","name":"Чабанівський
Михайло","memorial":"plaque","wikipedia":"uk:Чабанівський Михайло Іванович","wikidata":"Q12169028"},"coordinate":{"x":30.5037643,"y":50
{"name":"Малишко Андрій","score":7,"tags":{"historic":"memorial","name":"Малишко Андрій","memorial":"plaque","wikipedia":"uk:Малишко Ан
Самійлович","wikidata":"Q2037667"},"coordinate":{"x":30.5037142,"y":50.4477507,"z":"NaN"}},{"name":"Шумило Микита","score":7,"tags":
{"historic":"memorial","name":"Шумило Микита","memorial":"plaque","wikipedia":"uk:Шумило Микита Михейович","wikidata":"Q12172554"},"coo
{"x":30.5037682,"y":50.4477917,"z":"NaN"}},{"name":"Микитенко Іван","score":7,"tags":{"historic":"memorial","name":"Микитенко
Іван","memorial":"plaque","wikipedia":"uk:Микитенко Іван Кіндратович","wikidata":"Q4293022"},"coordinate":{"x":30.5037227,"y":50.447749
{"name":"Рильський Максим","score":7,"tags":{"historic":"memorial","name":"Рильський Максим","memorial":"plaque","wikipedia":"uk:Рильсь
Тадейович","wikidata":"Q2638198"},"coordinate":{"x":30.5037547,"y":50.4477584,"z":"NaN"}},{"name":"Панч Петро","score":7,"tags":
{"historic":"memorial","name":"Панч Петро","memorial":"plaque","wikipedia":"uk:Панч Петро Йосипович","wikidata":"Q2617530"},"coordinate
{"x":30.5037305,"y":50.447748,"z":"NaN"}},{"name":"Кундзіч Олексій","score":7,"tags":{"historic":"memorial","name":"Кундзіч
Олексій","memorial":"plaque","wikipedia":"uk:Кундзіч Олексій Леонардович","wikidata":"Q3386224"},"coordinate":{"x":30.5036998,"y":50.44
```


Рис. 3.11. Приклад JSON файлу з результатами обчислень

Для переходу на UI системи необхідно надіслати запит за такою адресою: “{site_name}/ui/home”. Після цього на сторінці буде відображено мапу на поля для введення параметрів запиту.

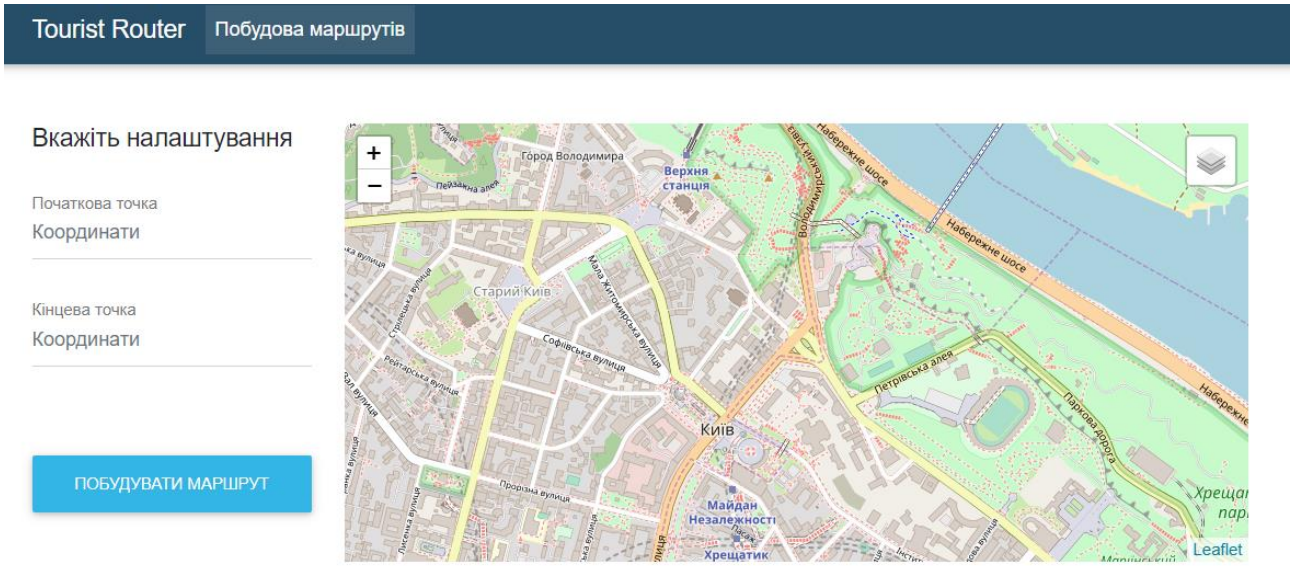


Рис. 3.12 Головна сторінка сайту.

За замовчанням шар мапи з зображенням туристичних пам’яток приховано, проте його відображення можна включити. Для цього необхідно натиснути на

позначку шарів у верхньому правому куті мапи та поставити галочку навпроти позначення «Туристичні об’єкти» (рис. 3.12).

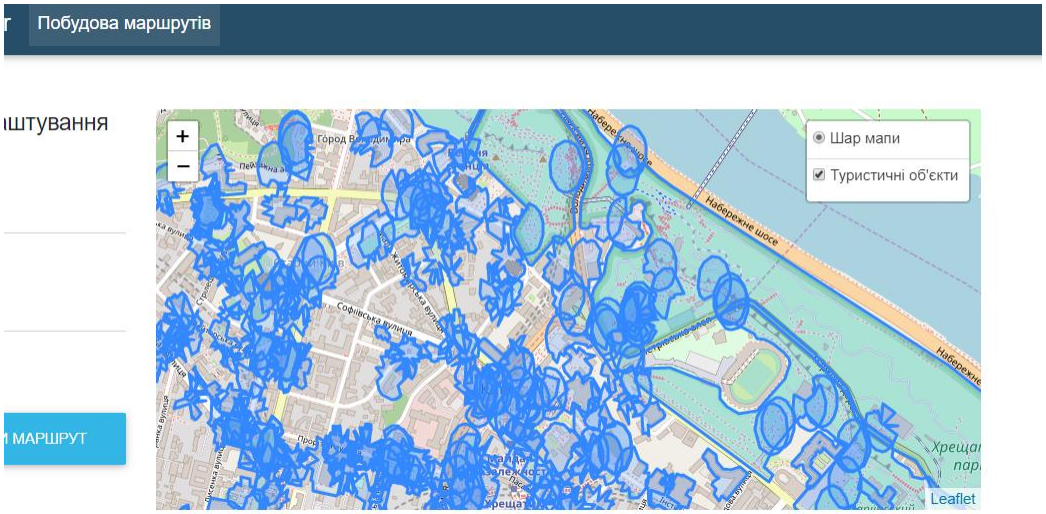


Рис. 3.12 Приклад включення шару з туристичними об’єктами

У блоці «Вкажіть налаштування» знаходяться поля для ручного введення значень координат початкової та кінцевої точок. Для того, щоб вони були опрацьовані коректним чином, необхідно вказати довготу та широту точки у десятинному форматі, розділених комою. При цьому ціла частина у значення має відділятися точкою. Задля зручності також було додано функцію визначення координат за допомогою вибору точки на карті, як показано на рисунку 3.13.

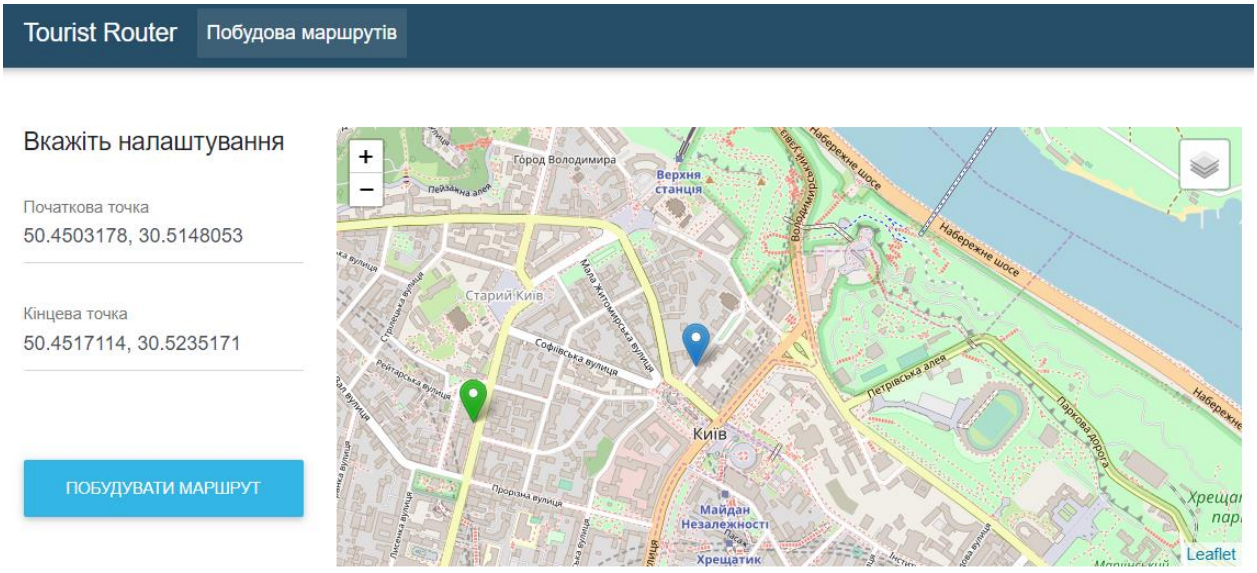


Рисунок 3.13 Приклад вибору точок

Для цього необхідно натиснути правою клавішею миші на місце на карті. У цьому місці має відобразитися маркер – зелений для початкової точки, синій для кінцевої. Точки можна виставляти безліч разів. При повторному виборі початкової точки, попередні маркери будуть видалені.

Після того координати було введено, користувач має натиснути кнопку «Побудувати маршрут». Результат обчислень буде відображено на карті у вигляді лінії з двома маркерами на кінцях.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі було описано реалізацію алгоритму побудови найкоротшого маршруту з урахуванням областей видимості проміжних пунктів. Також описано структуру класів та модулів системи.

У якості інструменту перетворення OSM XML мап в об’єкт графу координат було обрано GraphHopper насамперед за можливість перевизначити метод обчислення ваги ребер. Цей клас використовується також для перевизначення логіки нарахування ваги ребра при побудові географічного графу. Основний процес обрахунку областей видимості відбувається при завантаженні екземпляра класу MapParser, у якому також за допомогою ScoreCounter сервісу вираховується пріоритет на основі тегів об’єкту. Задля демонстрації можливостей програми та тестування алгоритму було створено клас RouteController, що надає доступ користувачеві до відкритого API системи та працює на основі REST-протоколу.

Також у розділі описано інструкцію роботи з користувацьким інтерфейсом, що демонструє коректність роботи алгоритму.

ВИСНОВКИ

У ході роботи було запропоновано алгоритм та описано модель системи, що при побудові маршруту враховує його довжину та кількість охоплених об’єктів заданого типу. Для цього розроблено алгоритм розрахунку областей видимості об’єктів, обчислення величини їх пріоритету та модель системи, яка на основі цих значень обчислює вагу ребер графу маршрутів та будує найкоротший шлях.

На даний момент недослідженими та нереалізованими є системи, які б при побудові найкоротшого маршруту враховували типи об’єктів, що є на мапі, та класифікували їх за тематикою. Це дозволило б модифікувати маршрут таким чином, щоб він був, з одного боку коротким, з іншого боку охоплював якомога більше туристичних об’єктів.

Проаналізовано існуючі рішення в області туристичної маршрутизації. За результатами порівняння виявлено, що жоден з додатків не враховує області видимості точок маршруту. Таким чином, актуальною є розробка системи, яка б шукала шлях між точками, враховуючи області видимості проміжних пунктів. Цей підхід дозволить, на основі алгоритмів побудови найкоротших маршрутів, виконати класифікацію об’єктів мапи та побудувати туристичний маршрут.

У роботі було обрано засоби вирішення задачі та розроблено ЇЇ математичну модель. Побудовано систему на клієнт-серверній архітектурі, що має відкритий прикладний інтерфейс за допомогою якого користувачі або сторонні додатки зможуть обраховувати туристичні маршрути. Джерелом даних для додатку було обрано OSM карти.

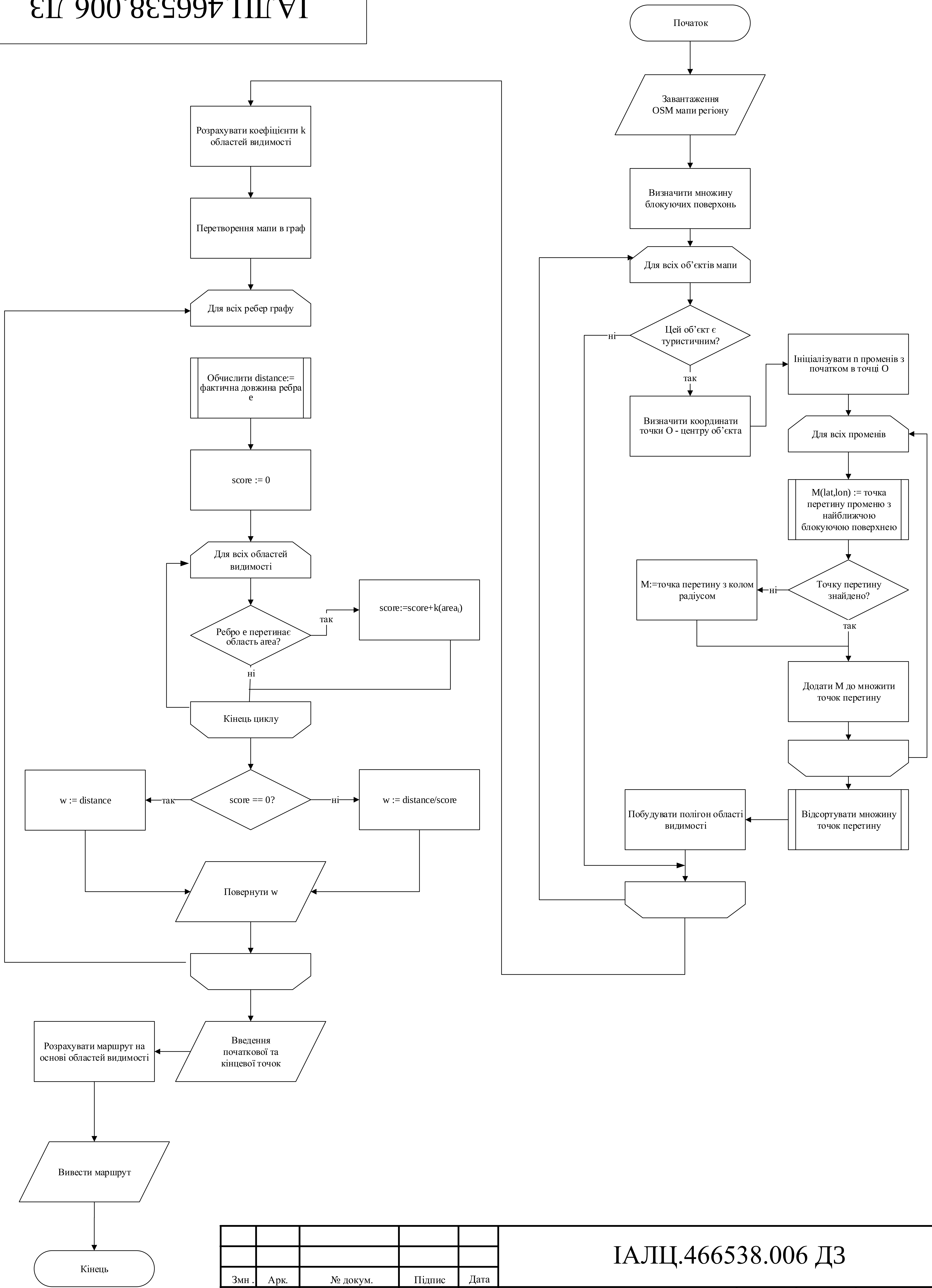
Доступ до інтерфейсу системи здійснюється за допомогою REST-протоколу. Також описано інструкцію користувача, яка демонструє правильність роботи алгоритму.

ПЕРЕЛІК ПОСИЛАНЬ

1. M. J. de Smith, M. F. Goodchild, and P. A. Longley, “Geospatial Analysis: A comprehensive guide to principles, techniques, and software tools”, 2nd ed. London: Troubador, 2007.
2. E. Dijkstra, “A note on two problems in connexion with graphs,” in Numerische Mathematik, vol. 1. 1959, P. 269–271.
3. P. E. Hart, N. J. Nilsson and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” in IEEE Transactions on Systems Science and Cybernetics SSC4, vol. 2. 1968, P. 100–107.
4. S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall, 2003, P. 55–121.
5. R. E. Korf, “Depth-first iterative-deepening: an optimal admissible tree search,” in Artificial Intelligence, vol. 27, 1985, P. 97-109.
6. P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar, “Heuristic search in restricted memory (research note),” in Artificial Intelligence, vol. 41, 1989, P. 197-222.
7. S. Russell, “Efficient memory-bounded search methods,” in Proceedings of the 10th European Conference on Artificial intelligence, 1992, P. 1-5.
8. H. Kaindl and G. Kainz, “Bidirectional heuristic search reconsidered,” in Journal of Artificial Intelligence Research, vol. 7, 1997, P. 283-317.
9. D. Wagner, T. Willhalm, “Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs,” in Proceedings of the 11th European Symposium on Algorithms, 2003, P. 776-787.
10. A. V. Goldberg and C. Harrelson, “Computing the shortest path: A search meets graph theory,” in Proceedings of the Sixteenth Annual ACM-IAM Symposium on Discrete Algorithms, 2005, P. 156-165.
11. P. Sanders and D. Schultes, “Highway hierarchies hasten exact shortest path queries,” in Proceedings of the 17th European Symposium on Algorithms, 2005, P. 568–579.
12. P. Sanders and D. Schultes, “Engineering Fast Route Planning Algorithms,” in 6th Workshop on Experimental Algorithms, 2007, P. 23-36.
13. Marcin Wojnarski «TomTom Traffic Prediction for Intelligent GPS Navigation» // M. Wojnarski, IEEE International Conference on Data Mining Workshops – 2010 – С.20-21.
14. Gabriel Svennerberg “Google Maps API 3” // G. Svennerberg, Apress – 2010 – С.157-160.
15. ViaMichelin Navigation. User Manual [Електронний ресурс] // ViaMichelin, URL:

http://enav.download.viamichelin.com/nav/tel/manuels/gbr/User_Manual_GBR_VMN_New_Edition_v7.pdf (дата звернення 01.05.2019).
16. YourNavigation.org About [Електронний ресурс] // YOURS, URL: https://wiki.openstreetmap.org/wiki/YOURS/ (дата звернення 01.05.2019).
17. Sebastian Schmitz «New Applications based on collaborative geodata – the case of Routing» /, Sebastian Schmitz, Proceedings of XXVIII INCA international congress on collaborative mapping and space technology – 2008 – С.1-7.
18. The Java History Timeline [Електронний ресурс] // Oracle, URL: http://www.java.com/en/javahistory/timeline.jsp (дата звернення 07.05.2019).
19. Benatallah, B., Casati, F., Toumani, F. “Web service conversation modeling: A cornerstone for e-business automation” in IEEE Internet Computing. 8, 2004, P. 46–54.
20. Dustdar, S., Schreiner, W. “A survey on web services composition", International Journal of Web and Grid Services, 2005, P.5-10.
21. Krasner, G.E. and S.T. Pope, “A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80”, Journal of Object-Oriented Programming, 1(3), P. 26-49, 1988.
22. Roger Foster, Dan Mullaney. “Basic Geodesy Article 018: Conversions and Transformations”. National Geospatial Intelligence Agency, 2014.
23. EPSG.io From MapTiler team [Електронний ресурс] // MapTiler, URL: http://epsg.io/7030-ellipsoid (дата звернення 10.05.2019).
24. Knuth, Donald E. «A generalization of Dijkstra's algorithm.» // Knuth, Donald E, Information Processing Letters 6.1 – 1977 – С.1-5.
25. Spencer G.H., M.V.R.K. Murty «General ray-tracing procedure» // Spencer G.H., M.V.R.K. Murty, JOSA 52.6 – 1962 – С.672-678.
26. Robert Geisberger. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. Institut für Theoretische Informatik Universität Karlsruhe, 2010.

ІАЛЦ.466538.006 ДЗ



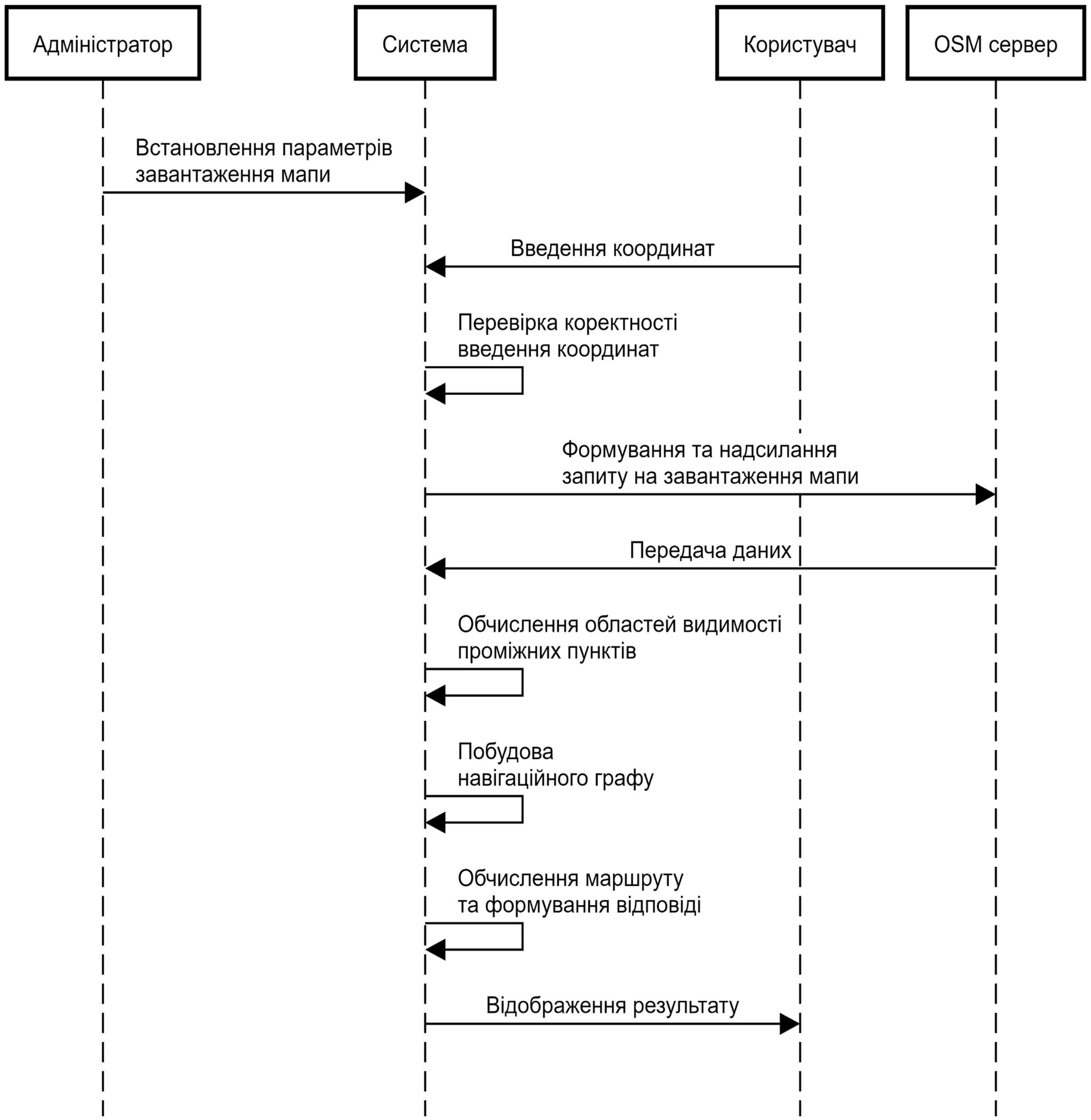
Змн.	Арк.	№ докум.	Підпис	Дата
Розроб.		Зубрич Є.С.		
Перевір.		Подрубайло О.О.		
Н. Контр.		Сімоненко В.П.		
Затверд.		Стіренко С.Г.		

ІАЛЦ.466538.006 ДЗ

Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів. Схема послідовності алгоритму

Літ.	*Арк.*	*Аркушів*
	1	1

НТУУ „КПІ”, ФІОТ, ІП-53



Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.466538.005 Д4		
Розроб.		Зубрич Є.С.			Побудова найкоротшого маршруту на карті з урахуванням областей видимості проміжних пунктів Діаграма послідовності	Літ.	Арк. 1 / Аркушів 1
Перевір.		Подрубайло О.О.					
Н. Контр.		Сімоненко В.П.				НТУУ „КПІ”, ФІОТ, ІП-53	
Затверд.		Стіренко С.Г.					

						ІАЛЦ.466538.005 Д2
Зм.	Арк.	№ докум.	Підпис	Дата		
Розроб.		Зубрич Є.С.			Побудова найкоротшого маршруту на карті з урахуванням обмежень видимості проміжних пунктів UML діаграма класів	Літ.	Арк.	Аркушів
Перевір.		Подрубайло О.О.					1	1
Н. Контр.		Симоненко В.П.				НТУУ „КПІ", ФІОТ, ІП-53
Затверд.		Стрєнко С.Г.