Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Факультет інформаційних технологій
(факультет)

Кафедра системного аналізу та управління
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня бакалавра

Студента Голоденко Аріадни Вячеславівни
академічної групи 124-20-2
спеціальності 124 Системний аналіз

на тему: «Оптимізація пасажирських перевезень з використанням Deutsche Bahn»

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**Інституційною**	
кваліфікаційної роботи	*к.т.н., доц. Желдак Т.А.*			
розділів:				
Інформаційно-аналітичний	*к.т.н., доц. Желдак Т.А.*			
Спеціальний розділ	*к.т.н., доц. Желдак Т.А.*			
Рецензент	*к.т.н., доц. Приходченко С.Д.*			
Нормоконтролер	*к.ф.-м.н., доц. Хом'як Т.В.*			

Дніпро

2024

ЗАТВЕРДЖЕНО:
завідувач кафедри
Системного аналізу та управління
(повна назва)

_____ *к.т.н., доц. Желдак Т.А.*
(підпис) (прізвище,ініціали)

«____» _____20___року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня *бакалавра*

студенту *Голоденко А. В.* **академічної групи** *124-20-2*
спеціальності: *124 Системний аналіз*
на тему «*Оптимізація пасажирських перевезень з використанням Deutsche Bahn»*
затверджену наказом ректора НТУ «Дніпровська політехніка»
від *№469-с від 23.05.2024 р.*

Розділ	Зміст	Терміни виконання
1. Інформаційно-аналітичний розділ	*Проаналізувати структуру Deutsche Bahn, визначити предметну область дослідження та проблеми, обґрунтувати методи виконання завдань, включаючи аналіз існуючих рішень та технологій.*	08.01.2024 – 08.03.2024
2.Спеціальний розділ	*Розробити алгоритми та створити систему для оптимізації пасажирських перевезень, враховуючи час в дорозі, зручність пересадок, мінімізацію витрат. Включити та протестувати алгоритми Дейкстри та A*.*	08.03.2024 – 15.06.2024

Завдання видано _____ *доц. Желдак Т.А.*
(підпис) (прізвище, ініціали)

Дата видачі: *08.01.2024 р.*

Дата подання до екзаменаційної комісії: *03.06.2024 р.*

Прийнято до виконання _____ Голоденко А. В.

(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 77 с., 23 рис., 3 табл., 6 додатків, 12 джерел, 2 блок-схеми.

Об'єктом дослідження в роботі є процес оптимізації пасажирських перевезень з використанням системи Deutsche Bahn.

Предметом дослідження є алгоритми та програмне забезпечення для оптимізації маршрутів пасажирських перевезень.

Метою даної кваліфікаційної роботи є підвищення зручності та ефективності переміщення пасажирів за рахунок алгоритмів для оптимізації пасажирських перевезень.

Методи дослідження: алгоритми Дейкстри та A*, системний аналіз для моделювання мережі перевезень, об'єктно-орієнтоване програмування на мові C++ для реалізації алгоритмів.

В *інформаційно-аналітичному розділі* наведено аналіз системи пасажирських перевезень Deutsche Bahn, визначено основні проблеми та виклики в оптимізації маршрутів, а також обґрунтовано вибір методів та інструментів для їх вирішення.

У *спеціальному розділі* розроблено та реалізовано алгоритми для знаходження оптимальних маршрутів з урахуванням часу в дорозі, зручності пересадок та мінімізації витрат. Проведено тестування та верифікацію розробленої системи на основі реальних даних.

Практична цінність отриманих результатів полягає в тому, що запропоновані алгоритми та система дозволяють значно підвищити ефективність планування маршрутів, зменшуючи час в дорозі та кількість пересадок для пасажирів.

Ключові слова: ОПТИМІЗАЦІЯ ПЕРЕВЕЗЕНЬ, АЛГОРИТМ ДЕЙКСТРИ, АЛГОРИТМ A*, ПАСАЖИРСЬКІ ПЕРЕВЕЗЕННЯ, DEUTSCHE BAHN, C++, СИСТЕМНИЙ АНАЛІЗ, МАРШРУТИЗАЦІЯ.

ABSTRACT

Explanatory note: 77 pages, 23 figures, 3 tables, 6 appendices, 12 references, 2 diagrams.

The object of the research is the process of optimizing passenger transportation using the Deutsche Bahn system.

The subject of the research is the algorithms and software for optimizing passenger transportation routes.

The purpose of this bachelor's thesis is to enhance the convenience and efficiency of passenger transportation through algorithms designed to optimize passenger transit.

Research methods: Dijkstra's and A* algorithms, system analysis for modeling the transportation network, object-oriented programming in C++ for algorithm implementation.

The informational-analytical section provides an analysis of the Deutsche Bahn passenger transportation system, identifies key problems and challenges in route optimization, and justifies the choice of methods and tools for solving them.

The special section develops and implements algorithms for finding optimal routes considering travel time, transfer convenience, and cost minimization. The developed system has been tested and verified using real data.

The practical value of the results lies in the fact that the proposed algorithms and system significantly improve the efficiency of route planning, reducing travel time and the number of transfers for passengers.

Keywords: TRANSPORTATION OPTIMIZATION, DIJKSTRA'S ALGORITHM, A* ALGORITHM, PASSENGER TRANSPORTATION, DEUTSCHE BAHN, C++, SYSTEM ANALYSIS, ROUTING.

ЗМІСТ

Перелік умовних скорочень

DB – Deutsche Bahn;

IC – InterCity;

ICE – InterCityExpress;

C++ – мова програмування високого рівня;

Hbf – Hauptbahnhof;

SFML – Simple and Fast Multimedia Library.

ВСТУП

Ефективна організація пасажирських перевезень є важливим завданням у сучасних умовах урбанізації та зростання кількості пасажирів. Інтенсифікація транспортних потоків вимагає удосконалення методів планування та управління маршрутами, що дозволяє забезпечити зручність і швидкість переміщення пасажирів. Системний аналіз, як підхід до вирішення складних проблем, надає інструменти для оптимізації маршрутів та графіків руху, що дозволяє значно підвищити ефективність роботи транспортної системи.

Використання методів та технологій, застосовуваних Deutsche Bahn, є актуальним завданням, оскільки ця система відома своєю ефективністю та точністю. DB застосовує передові алгоритми та програмне забезпечення для оптимізації маршрутів і розкладів руху, що може бути корисним для вдосконалення української транспортної системи.

Актуальність роботи полягає в тому, що зростання урбанізації та кількості пасажирів створює потребу в удосконаленні процесів планування пасажирських перевезень для забезпечення зручності та ефективності транспортної системи. Тому об'єктом дослідження було обрано процес оптимізації пасажирських перевезень з використанням системи Deutsche Bahn.

Метою даної кваліфікаційної роботи є підвищення зручності та ефективності переміщення пасажирів шляхом розробки алгоритмів та програмного забезпечення для оптимізації пасажирських перевезень з використанням системного аналізу та методів, які використовуються у DB.

Для досягнення поставленої мети в роботі мають бути вирішені задачі:

- провести аналіз існуючих методів оптимізації маршрутів пасажирських перевезень;
- визначити вимоги до алгоритмів та програмного забезпечення для оптимізації маршрутів;
- розробити та реалізувати алгоритми для знаходження оптимальних маршрутів з урахуванням часу в дорозі, зручності пересадок та мінімізації витрат;
- провести тестування та верифікацію розробленого програмного забезпечення на основі реальних даних.

Об'єктом дослідження є процес оптимізації пасажирських перевезень.

Предметом дослідження є алгоритми та програмне забезпечення для оптимізації маршрутів пасажирських перевезень.

Для розв'язання поставлених задач будуть використані наступні методи:

- системний аналіз для моделювання транспортної мережі;
- алгоритми Дейкстри та A* для пошуку найкоротших шляхів;
- об’єктно-орієнтоване програмування на мові C++ для реалізації алгоритмів.

Розробка та впровадження запропонованих алгоритмів та програмного забезпечення дозволять значно підвищити ефективність планування маршрутів пасажирських перевезень, зменшуючи час у дорозі та кількість пересадок. Це сприятиме підвищенню точності розкладів руху та загальної якості пасажирських перевезень.

Практична цінність отриманих результатів полягає в можливості їх застосування для вдосконалення транспортної системи, що матиме значний соціальний та економічний ефект. Запропоновані алгоритми та система дозволять зменшити витрати на перевезення, покращити зручність та швидкість переміщення пасажирів, що сприятиме підвищенню ефективності роботи транспортної системи в цілому.

1 ІНФОРМАЦІЙНО-АНАЛІТИЧНИЙ РОЗДІЛ

1.1 Теоретичні основи пасажирських перевезень

У сучасному світі пасажирські перевезення не лише забезпечують мобільність населення, а й відіграють ключову роль у розвитку економіки, соціальному зв'язку та сприяють культурній обміну між різними регіонами та країнами. Важливість цього аспекту стає особливо очевидною в умовах постійного зростання населення та містобудівних та інфраструктурних викликів. Розуміння основних понять та термінів у цій галузі є ключовим для подальшого аналізу та оптимізації пасажирських перевезень.

Пасажирські перевезення – це вид транспортних послуг, спрямованих на переміщення пасажирів з одного місця в інше. Цей процес може бути здійснений різними видами транспорту, такими як:

- залізничний транспорт;
- автомобільний транспорт;
- міський транспорт;
- водний транспорт;
- авіаційний транспорт.

Транспортна мережа є основним фундаментом для забезпечення зв'язку між різними місцями та місцевостями для пасажирських перевезень. Ця система включає в себе різноманітні види транспорту, такі як дороги, залізниці, водні шляхи та авіалінії, які працюють спільно для забезпечення ефективного переміщення людей.

Маршрут в пасажирських перевезеннях є вирішальним елементом, що визначає шлях переміщення транспортного засобу та його зупинки, а також є стратегічно розробленою системою, спрямованою на максимальну зручність та ефективність для пасажирів. Планування маршрутів вимагає

ретельного аналізу попиту, географічних особливостей та інфраструктури. Оптимально спроектований маршрут може значно зменшити час подорожі та збільшити комфорт пасажирів, що веде до підвищення загальної якості обслуговування. Визначення місць для зупинок на маршруті відіграє критичну роль у забезпеченні доступності транспорту для мешканців. Це може включати такі фактори, як:

- густота населення;
- наявність ключових об'єктів (наприклад, навчальних закладів, магазинів);
- забезпечення зручних пересадок на інші маршрути.

Стратегічне розміщення зупинок також сприяє використанню громадського транспорту та зменшенню автомобільного трафіку, що може допомогти у покращенні екологічної ситуації в містах. Більш того, моніторинг та аналіз ефективності маршрутів є важливою складовою оптимізації системи пасажирських перевезень. Це дозволяє операторам реагувати на зміни в попиті, впроваджувати інноваційні рішення та покращувати якість обслуговування пасажирів [1].

Залізничні перевезення (англ. *railway transportation*, нім. *Eisenbahntransport m*) – це технологічний процес перевезення пасажирів та вантажів з використанням залізничного транспорту. Залізниця виступає в якості наземної транспортної системи, яка дозволяє переміщати пасажирів і товари з одного пункту до іншого шляхом руху поїздів по залізничних коліях.

Залізничні перевезення, або залізничний транспорт, є важливою складовою транспортної інфраструктури багатьох країн, включаючи Німеччину, де DB відіграє ключову роль у забезпеченні цих послуг.

Однією з головних переваг залізничних перевезень є ефективність та велика потужність перевезення великих обсягів вантажів та пасажирів на великі відстані. Залізниця може бути дуже конкурентоспроможною у

випадках, коли потрібно перевозити великі обсяги товарів або пасажирів на великі відстані.

Багато структур, що є цікавими із точки зору математики, інформатики та їх практичного застосування, можуть бути описані за допомогою *графів* [2].

У загальному значенні *графом* називають непорожню множину вершин (вузлів), з'єднаних ребрами. Для різних галузей застосування графи можуть розрізнятися напрямами, обмеженнями на кількість зв'язків, додатковими відомостями про їх вершини та ребра.

У строгому математичному визначенні *граф* являє собою пару множин: $G = (V, E)$, де V – підмножина будь-якої лічильної множини (вершини), а E – підмножина декартового добутку $V \times V$ (ребра або зв'язки).

Повним називається граф, у якого будь-яка вершина з'єднується (ребрами або дугами) з усіма іншими вершинами. Так, граф, зображений на рис. 1.1, не повний.

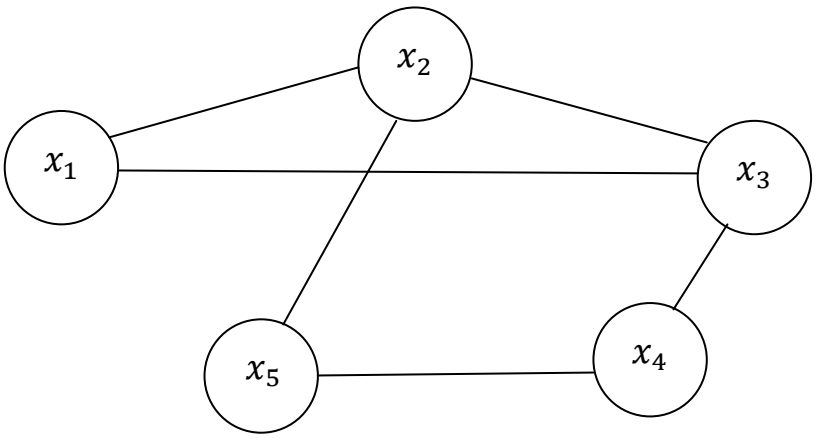


Рис. 1.1. Приклад геометричного зображення повного графа

Підграфом G_A графа $G = (X, \Gamma)$ називається граф, що включає лише ті з вершин графа G, які утворюють множину A, а також дуги, що з'єднують ці вершини. Наприклад, підграфом є окреслена пунктиром область A на рис. 1.2.

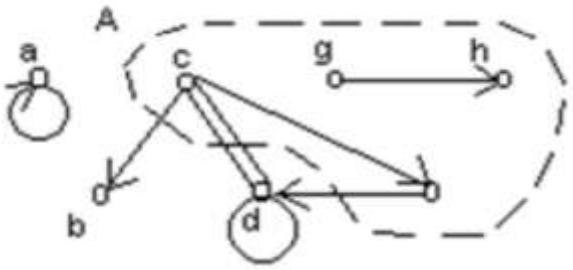


Рис. 1.2. Приклад геометричного зображення підграфа

У математичному записі підграф позначається у такий спосіб:

$$G_A = (A, \Gamma_A), \text{де } A \subseteq X, \Gamma_{AX} = (\Gamma_X) \cap A. \quad (1.1)$$

Частковим графом G_Δ стосовно графа $G = (X, \Gamma)$, є такий, що містить тільки частину дуг графа G, тобто визначений умовою:

$$G\_\Delta = (X, \Delta), \text{де } \Delta x \subseteq \Gamma_X \quad (1.2)$$

Приміром, нехай $G = (X, \Gamma)$ – карта залізничних маршрутів DB. Тоді мапа окремого регіону являє собою підграф, а мапа, яка включає лише певні маршрути або вузли, – це частковий граф.

1.2 Організація технологічного процесу в Deutsche Bahn

1.2.1 Характеристика діяльності та структура підприємства

Deutsche Bahn (з нім. "Німецька залізниця") – один з найбільших та найважливіших залізничних операторів у світі. Воно було створено в 1994 році після реформування залізничної системи Федеративної Республіки Німеччина в результаті об’єднання Deutsche Bundesbahn та Deutsche Reichsbahn, і його головною метою було покращення та ефективніше управління залізничним транспортом країни.

Ядром компанії є федеральні залізниці. DB AG створена як холдингова компанія з операційного управління, яка керує як компаніями залізничної інфраструктури (EIU), так і компаніями залізничного транспорту

(EVU) у Німеччині. За даними з офіційного веб-сайту DB [3] діяльність зосереджена в таких сферах діяльності:

- надання широкого спектру пасажирських перевезень;
- забезпечення вантажних перевезень як на внутрішніх, так і міжнародних маршрутах;
- володіння значною частиною залізничної інфраструктури в Німеччині;
- надання клієнтам різноманітні логістичні послуги.

В публікації «Der DB-Konzern. Daten und Fakten 2022» [4] зазначається, що DB-Konzern є провідним постачальником у секторі мобільності та логістики, і в основному складається з Integrated Rail System (з англ. «Інтегрована залізнична система») і двох великих міжнародних дочірніх компаній DB Schenker і DB Arriva – рис. 1.3. Інтегрована залізнична система включає пасажирські перевезення в Німеччині, залізничні вантажні перевезення, підрозділи операційного обслуговування та компанії залізничної інфраструктури. Основна увага в її діяльності приділяється залізничному транспорту Німеччини [5].

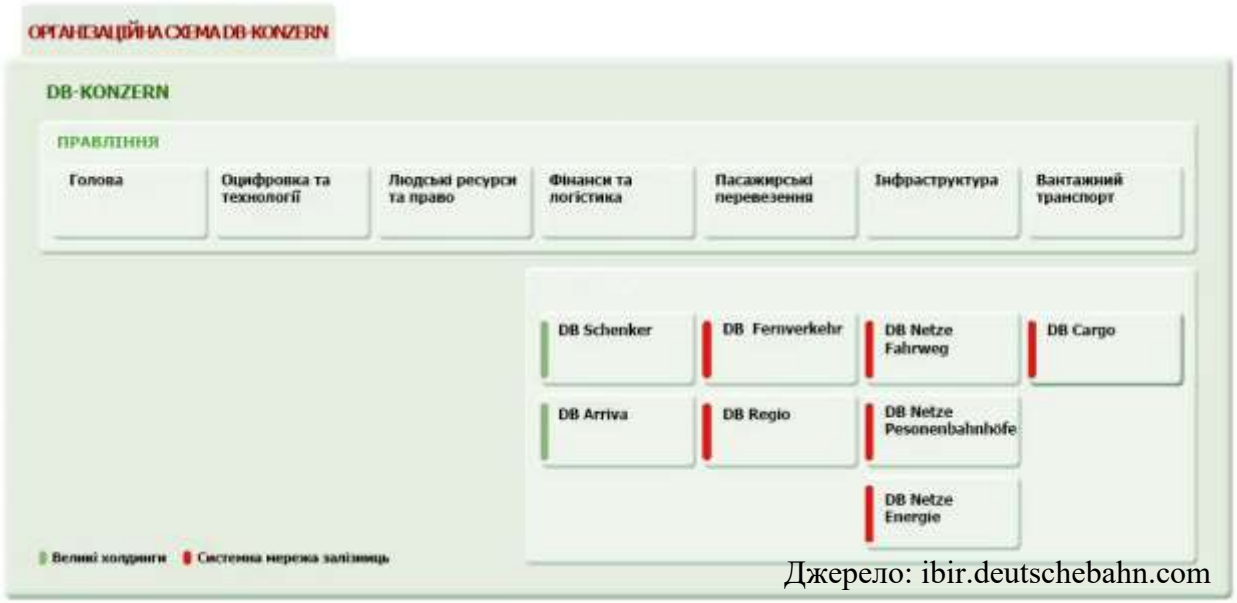


Рис. 1.3. Схема організації DB-Konzern

1.2.2 Мультимодальний транспортний сервіс DB

У DB представлене різноманіття транспортних засобів, які забезпечують пасажирам широкий спектр можливостей для комфортних та ефективних подорожей як всередині країни, так і за її межами – рис. 1.4. З врахуванням різних вимог та потреб пасажирів, DB пропонує різні види транспорту, включаючи:

- високошвидкісні поїзди (Hochgeschwindigkeitszüge);
- міжнародні поїзди (Intercity та Eurocity);
- міські електрички (S-Bahnen);
- автобуси та трамваї (Busse та Straßenbahnen);
- міський транспорт міський (U-Bahn);
- додаткові послуги (Sonstige Züge);
- виклик на транспорт (Anrufpflichtige Verkehre).

Поїзди DB відображають різноманітність варіантів подорожей, які задовольняють різні потреби пасажирів. Від місцевих регіональних поїздів, що забезпечують зв'язок між містами та селищами в межах регіону, до міжміських та міжнародних маршрутів, які покривають великі відстані та забезпечують зв'язок з іншими країнами Європи, DB пропонує широкий спектр можливостей для подорожей.

Крім поїздів, мережа автобусних маршрутів DB є важливою частиною транспортної інфраструктури. Вона допомагає подолати короткі та середні відстані, забезпечуючи пасажирам зручний доступ до різних міст та регіонів. Це особливо корисно для тих місць, які не мають залізничного з'єднання або для яких автобусний транспорт є більш ефективним варіантом.

Рис. 1.4. Сторінка мобільного додатку DB для вибору транспорту

У багатьох містах Німеччини DB також пропонує трамвайні та метрополітенові маршрути. Ці види громадського транспорту дозволяють пасажирам легко та швидко пересуватися по місту, дістатися до різних районів та використовувати їх як зручний засіб для щоденних подорожей.

Навіть за межами Німеччини, DB забезпечує міжнародні маршрути, які з'єднують країну з іншими країнами Європи, такими як Франція, Австрія, Швейцарія та інші. Це відкриває додаткові можливості для пасажирів подорожувати та відкривати нові напрямки.

1.2.3 Планування маршрутів та розкладів

У системі DB, як і в будь-якій великій транспортній компанії, організація технологічного процесу є критично важливою для забезпечення

ефективності, безпеки та якості перевезень. Технологічний процес включає в себе ряд складних систем та процедур:

- управління рухом поїздів;
- планування маршрутів і розкладів;
- технічне обслуговування;
- інноваційні технології;
- системи керування руху та моніторингу;
- технології для пасажирів;
- безпека та стандарти.

Планування маршрутів та розкладів поїздів є критично важливим етапом у роботі залізничних перевізників, таких як DB. Цей процес має на меті оптимізувати рух поїздів для забезпечення максимальної ефективності та комфорту для пасажирів, враховуючи при цьому різноманітні фактори.

Одним з ключових аспектів планування є аналіз пасажиропотоку. Це означає вивчення обсягів пасажирського руху на різних маршрутах та в різні часи доби. Під час аналізу враховуються різні фактори, такі як час року, святкові дні, події та інші події, які можуть впливати на популярність та пасажиропотік на певних напрямках.

Крім того, враховується і популярність конкретних маршрутів. На основі зібраної статистики та інформації про попит планувальники розробляють оптимальні розклади поїздів, щоб задовольнити потреби пасажирів та максимізувати використання ресурсів.

Також враховується час подорожі та швидкість руху поїздів. Планувальники стараються мінімізувати час подорожі для пасажирів, розробляючи оптимальні маршрути та використовуючи швидкісні поїзди на відстанях, де це є доцільним.

Крім того, важливо враховувати і синхронізацію з іншими видами транспорту. Пасажири часто використовують комбіновані маршрути, що включають поїзди, автобуси, літаки тощо. Тому важливо забезпечити зручні та ефективні пересадки між різними видами транспорту.

Загалом, планування маршрутів та розкладів поїздів - це складний процес, який вимагає уваги до різних факторів для забезпечення ефективності, безпеки та зручності для пасажирів. Ретельне планування дозволяє залізничним перевізникам оптимізувати роботу та надавати якісний сервіс своїм клієнтам.

1.3 Проблеми при плануванні маршруту

Успішна організація пасажирських перевезень має низку складнощів, особливо коли маршрути сплановані неправильно. Ці проблеми включають:

- недооцінка часу подорожі;
- недооцінка пасажиропотоку;
- неадекватне обслуговування;
- неефективне використання ресурсів;
- надмірна кількість пересадок;
- неправильно розрахована ціна.

Недооцінка часу подорожі є однією з основних труднощів, що може виникнути в процесі планування маршрутів. Помилкова оцінка часу між зупинками або непередбачені транспортні затори можуть призвести до неочікуваних затримок та незручностей для пасажирів.

Недооцінка пасажиропотоку також є важливою проблемою, яка може виникнути при плануванні маршрутів. Недостатня кількість місць у поїздах

або переповненість вагонів може призвести до невдоволення пасажирів та втрати їх довіри до компанії.

Неадекватне обслуговування також може виникнути внаслідок неправильного планування маршрутів. Недостатнє покриття деяких районів або пунктів призначення може призвести до незадоволення мешканців та пасажирів та негативно вплинути на імідж компанії.

Неефективне використання ресурсів є ще однією проблемою, яка може виникнути при плануванні маршрутів. Неправильне розподілення транспортних засобів та ресурсів може призвести до зайвого споживання палива та фінансових втрат.

Надмірна кількість пересадок також може стати причиною незручностей для пасажирів. Часті пересадки можуть зробити поїздку більш складною та вимагати додаткових зусиль від пасажирів, що може вплинути на загальний комфорт подорожі.

Неправильно розрахована ціна квитків також може стати перешкодою для пасажирів. Висока ціна квитків може призвести до недостатньої доступності перевезень та втрати потенційних клієнтів для DB [6].

1.4 Аналітичний огляд систем визначення раціональних маршрутів Deutsche Bahn

Ефективне функціонування залізничного транспорту є ключовим для забезпечення високої якості пасажирських перевезень. DB стикається з численними викликами в процесі планування і оптимізації маршрутів та графіків руху потягів.

Оптимізація маршрутів та графіків залізничного транспорту є ключовим елементом в покращенні ефективності та якості пасажирських перевезень. Для досягнення цієї мети використовуються різноманітні підходи

та методи, спрямовані на підвищення пунктуальності, зручності та швидкості маршрутів.

Перший етап оптимізації полягає в аналізі поточного стану мережі маршрутів та графіків руху потягів. Це включає в себе оцінку потреб пасажирів, обсягів перевезень на різних напрямках, а також інфраструктурних особливостей мережі. На основі отриманих даних розробляються стратегії розташування маршрутів та графіків, які максимально відповідають потребам користувачів та забезпечують оптимальне використання ресурсів.

Одним із ключових аспектів оптимізації є використання передових алгоритмів та моделей для визначення оптимальних маршрутів та розкладів руху потягів. Це допомагає зменшити час подорожі, уникнути заторів та перевантажень на мережі, а також забезпечити плавну роботу всіх компонентів транспортної системи.

Крім того, важливим елементом оптимізації є впровадження новітніх технологій, таких як системи автоматизованого управління рухом поїздів, системи моніторингу та контролю за рухом потягів, а також інформаційні системи для пасажирів. Ці технології дозволяють покращити точність та ефективність управління рухом, зменшити ризики виникнення аварій та забезпечити безпеку перевезень.

Нарешті, для забезпечення позитивного досвіду користувачів важливо створювати програми та стратегії, спрямовані на підвищення задоволеності клієнтів. Це може включати в себе впровадження програм лояльності, розробку зручних мобільних додатків для покупки квитків та отримання інформації про рейси – рис. 1.5, а також забезпечення високого рівня обслуговування та комфорту пасажирів під час подорожі.

Рис. 1.5. Сторінка мобільного додатку DB для бронювання квитків

Задачі пошуку шляху в контексті оптимізації маршрутів та графіків залізничного транспорту включають у себе визначення найоптимальнішого маршруту між станціями, що відображається у виборі найефективніших маршрутів для забезпечення швидкої та ефективної перевезення пасажирів.

Існує кілька алгоритмів для розв'язання задач пошуку шляху у графах:

1) Алгоритм Дейкстри знаходить найкоротший шлях від однієї вершини графа до всіх інших, при умові, що ваги ребер не можуть бути від'ємними. Він використовує метод "жадібного" підходу, обираючи на кожному кроці найменшу вагу для досягнення наступної вершини.

2) Алгоритм A* також знаходить найкоротший шлях, але він враховує не лише вагу ребер, а й евристичну оцінку відстані до кінцевої вершини. Він комбінує в собі методи жадібного підходу із використанням евристичної оцінки, що дозволяє йому працювати більш ефективно, особливо на великих графах.

3) Алгоритм Беллмана-Форда знаходить найкоротший шлях між двома вершинами, навіть якщо ваги ребер можуть бути від'ємними. Він працює, шукаючи всі можливі шляхи в графі і поступово оновлює відомості про найкоротший шлях до кожної вершини.

4) Алгоритм пошуку в ширину (BFS) шукає найкоротший шлях від однієї вершини до всіх інших у ненапрямленому графі. Він працює шляхом обходу графа у ширину, розглядаючи всі вершини на певній відстані від початкової вершини.

Кожен з цих алгоритмів має свої унікальні переваги та обмеження, і вибір конкретного залежить від властивостей графа та потреб застосування.

1.5 Алгоритм Дейкстри

Ідея алгоритму. Кожній вершині з множини X поставимо у відповідність число $d(x)$ – найменшу відому відстань від цієї вершини до s. Алгоритм включає кілька кроків, зокрема, на кожному з них опрацьовується одна вершина з метою зменшення чисел $d(x)$. Робота алгоритму закінчується, коли всі вершини опрацьовано.

Опишемо цей алгоритм.

Крок 1. *Ініціалізація.*

Задамо, що $d(x) = 0$ і $d(x) = \infty$ для всіх вершин x, відмінних від s (це відображає той факт, що відстані до цих вершин на даний момент не відомі). Жодну вершину ще не опрацьовано (не забарвлено). Через у позначимо останню із забарвлених вершин. Позначимо (пофарбуємо) вершину s і будемо вважати, що $y = s$.

Крок 2. *Основний крок алгоритму.* Для кожної незабарвленої вершини x перерахуємо величину відстані $d(x)$ в такий спосіб:

$$d(x) = \min \{d(x); d(y); +a(y, x)\}. \quad (1.3)$$

Якщо $d(x) = \infty$ для всіх незабарвлених вершин x, то належить закінчити процедуру алгоритму, оскільки у вихідному графі відсутні шляхи з вершини s у незабарвлені.

Коли ні, то слід пофарбувати ту з вершин x, для якої величина $d(x)$ найменша. Крім того, потрібно пофарбувати дугу, що веде до обраної на даному кроці вершини x [саме для цієї дуги досягався мінімум виразу (1.3)]. Вважати, що $y = x$.

Крок 3. Якщо $y = t$, то процедуру закінчити, оскільки найкоротший шлях від вершини s до вершини t знайдений (це єдиний шлях від s до t, котрий складається із забарвлених дуг). А якщо ні, то перейти до *кроку* 2. Алгоритм описано.

Забарвлені дуги утворюють у вихідному графі орієнтоване дерево з коренем у вершині s. Його називають *орієнтованим деревом найкоротших шляхів*. Єдиний шлях від вершини s до будь-якої вершини x, що йому належить, буде найкоротшим між зазначеними вершинами.

Оскільки на всіх етапах алгоритму Дейкстри забарвлені шляхи утворюють у вихідному графі орієнтоване дерево, то алгоритм можна розглядати як процедуру нарощування орієнтованого дерева з коренем у

вершині s. Коли в процедурі нарощування досягнута вершина t, процедуру припиняють.

Для визначення найкоротших шляхів від вершини s s до всіх вершин вихідного графа процедуру нарощування дерева слід продовжити, допоки всі вершини графа не будуть включені в орієнтоване дерево найкоротших шляхів. При цьому для вихідного графа буде отримано *покривне дерево* (звісно, якщо в даному графі існує хоча б одне з таких).

Отже, аби описаний вище алгоритм дозволив одержати дерево найкоротших шляхів від вершини s до всіх інших вершин, його третій крок повинен бути скорегований у такий спосіб:

Крок 3а. Якщо всі вершини виявляються забарвленими, то процедуру закінчують (існує єдиний найкоротший шлях від вершини s до будь-якої вершини x і він складається тільки із забарвлених дуг). А якщо ні, то повертаються до *кроку* 2.

Зауважимо, що алгоритм Дейкстри застосовується при роботі з орієнтованими графами, які не мають петель, а довжини дуг не набувають від'ємних значень. Але, він може бути узагальнений на випадок, коли деякі з дуг мають від'ємні значення довжини. Необхідна модифікація описаного вище алгоритму (Форда) передбачає такі дії:

− На *другому кроці* алгоритму перерахування величини відстані $d(x)$

За допомогою співвідношення (1.1) проводиться для всіх вершин, а не тільки для незабарвлених. Отже, числа $d(x)$ можуть зменшуватися як для незабарвлених, так і для пофарбованих вершин.

− Якщо для деякої пофарбованої вершини x відбувається зменшення величини $d(x)$, то із неї та з інцидентної ній пофарбованої дуги забарвлення знімається.

− Процедура алгоритму закінчується тільки тоді, коли всі вершини пофарбовані й якщо після виконання *кроку* 2 жодне із чисел $d(x)$ не змінюється [1].

1.6 Алгоритм A*

Алгоритм A (A-star)* є одним із найпопулярніших та найефективніших алгоритмів для пошуку найкоротшого шляху у графах. Він поєднує в собі елементи двох інших алгоритмів: алгоритму Дейкстри та жадібного пошуку найкращим першим. A* використовується для знаходження найкоротших шляхів у багатьох застосуваннях, таких як навігація роботів, ігровий штучний інтелект, та маршрутизація в мережах [7].

*Алгоритм A** є методом пошуку шляху в графах, який використовує евристичні функції для оцінки вартості шляху. Він визначається наступною формулою:

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

де $f(u)$ – загальна оцінка вартості шляху через вузол u; $g(u)$ – фактична вартість шляху від початкового вузла до поточного вузла u, $h(u)$ – евристична оцінка вартості найкоротшого шляху від вузла u до цільового вузла.

Функція вартості $g(u)$ фактичну вартість шляху від початкового вузла до поточного вузла u. Це може бути відстань, час або інші ресурси, витрачені на проходження цього шляху.

Евристична функція $h(u)$ оцінює вартість найкоротшого шляху від поточного вузла u до цільового вузла. Евристична функція повинна бути допустимою, тобто вона ніколи не повинна перевищувати фактичну вартість

найкоротшого шляху. В іншому випадку, алгоритм може не знайти оптимальний шлях.

Опишемо цей алгоритм.

Крок 1. *Ініціалізація*

Спочатку встановлюємо значення $g(s) = 0$ для початкової вершини s і $g(x) = \infty$ для всіх інших вершин x, що відображає невідомість відстаней до цих вершин на даний момент. Далі обчислюємо евристичну оцінку $h(x)$ для всіх вершин x, де $h(x)$ — це оцінка найкоротшої відстані від вершини x до цільової вершини t. Після цього встановлюємо (1.4) для всіх вершин x.

Початкову вершину s додаємо до відкритого списку, а також ініціалізуємо порожній закритий список.

Крок 2. На цьому етапі вибираємо вершину y з найменшою вартістю $f(y)$ з відкритого списку. Якщо $y = t$, то найкоротший шлях знайдено, і алгоритм завершується. Видаляємо вершину y з відкритого списку та додаємо її до закритого списку. Потім для кожної сусідньої вершини x від вершини y перевіряємо, чи x вже знаходиться в закритому списку; якщо так, то пропускаємо її. Далі обчислюємо тимчасову g-вартість для вершини x, як суму $g(y)$ та вартості ребра між x. Якщо вершина x не знаходиться у відкритому списку або тимчасова g-вартість менша за поточну $g(x)$, оновлюємо $g(x)$, обчислюємо нову вартість $f(x)$, і додаємо x до відкритого списку, якщо її там ще не було. Повторюємо цей крок, поки не буде знайдено найкоротший шлях або не буде опрацьовано всі можливі вершини.

1.7 Висновки до розділу

Проведений інформаційно-аналітичний розділ дозволяє зробити ряд висновків щодо пасажирських перевезень та процесів, пов'язаних з плануванням маршрутів у DB.

Детальний розгляд теоретичних основ пасажирських перевезень підкреслює важливість ефективного планування маршрутів та графіків для забезпечення надійності та задоволення потреб пасажирів. Правильна організація пасажирських перевезень є ключовим фактором успіху транспортного підприємства. Аналіз структури підприємства та характеристик діяльності DB показує, що складна організаційна структура та велика кількість задіяних елементів вимагають ретельного та зваженого підходу до планування.

Процес планування маршрутів та розкладів є складним завданням, яке стикається з рядом викликів. Основні з них включають необхідність врахування попиту пасажирів, забезпечення стиковок між різними видами транспорту та мінімізацію затримок. Специфічні проблеми планування маршрутів у DB включають великі розриви в покритті, погану інтеграцію між регіональними та національними маршрутами, а також часті технічні та інфраструктурні збої.

Для подолання цих викликів важливо застосовувати методи оптимізації маршрутів та графіків. Використання передових алгоритмів, таких як алгоритм Дейкстри та алгоритм A*, дозволяє значно підвищити ефективність планування. Алгоритм Дейкстри забезпечує знаходження найкоротших шляхів у графі з невід'ємними вагами, що є корисним для базового планування маршрутів. Алгоритм A*, у свою чергу, враховує як

відстань від джерела до вузла, так і оцінку відстані від вузла до цілі, що дозволяє ефективніше знаходити оптимальні маршрути.

Таким чином, оптимізація маршрутів та графіків руху є ключовим аспектом для покращення якості пасажирських перевезень у DB. Впровадження передових технологій та алгоритмів, а також врахування специфічних викликів і проблем, дозволить забезпечити більш надійне та комфортне транспортне обслуговування пасажирів.

2 СПЕЦІАЛЬНИЙ РОЗДІЛ

2.1 Аналіз вхідних даних

Аналіз вхідних даних є критичним етапом в дослідженні оптимізації пасажирських перевезень у контексті DB. Передбачення ефективних та зручних маршрутів для пасажирів вимагає ретельного вивчення різноманітних аспектів мережі залізничного транспорту.

На цьому етапі аналізу ми обмежуємося розглядом мережі ICE та IC станцій DB. Ці типи потягів, зображений на рисунку 2.1, є одними з найшвидших та найбільш комфортних у мережі німецьких залізниць та забезпечують високоякісне пасажирське обслуговування.

Рис. 2.1. Потяг типу ICE

Аналізуючи саме ICE/IC Streckennetz (мережу ICE/IC), увага зосереджується на головних маршрутах та ключових з'єднаннях, які забезпечують швидку та зручну транспортну доступність між містами та регіонами. Ця мережа відіграє стратегічну роль у забезпеченні ефективних та

затишних перевезень для пасажирів, що подорожують як в межах країни, так і за її межам – ДОДАТОК Г. Обмежуючись аналізом лише ICE/IC Streckennetz, можна детальніше дослідити мережу головних маршрутів та їх взаємозв'язок з ключовими станціями, що сприяє розробці ефективних стратегій оптимізації пасажирських перевезень та підвищенню рівня сервісу для користувачів.

Для моделювання графів з вершинами, що мають різний розклад і ребрами, що з'єднують ці вершини, було використано структури C++.

Структура *Time* призначена для зберігання часу у форматі годин і хвилин – рисунок 2.2. Вона має дві змінні: *hours*, яка представляє години, і *minutes*, яка представляє хвилини. Конструктори *Time()* та *Time(int h, int m)* використовуються для ініціалізації об'єкту *Time*. Перший конструктор створює об'єкт часу зі значеннями за замовчуванням, тобто 0 годин і 0 хвилин. Другий конструктор дозволяє встановити значення годин та хвилин за допомогою переданих аргументів. Такий підхід забезпечує гнучкість використання структури *Time* для представлення різних моментів у часі.

```
struct Time {
    int hours;
    int minutes;

    Time() : hours(0), minutes(0) {}
    Time(int h, int m) : hours(h), minutes(m) {}
};
```

Рис. 2.2. Структура Time

Структура *Vertex*, зображена на рисунку 2.3, є ключовим елементом для моделювання вершин у графічних структурах даних. Вона призначена для зберігання інформації про вершину графа, що включає в себе ім'я вершини, її координати на площині та часи, пов'язані з цією вершиною.

У полях структури *Vertex* зберігається наступна інформація:

- *name*: Рядок, що представляє ім'я вершини.
- *x, y*: Величини типу *float*, що відповідають координатам вершини на площині.
- *times*: Вектор об'єктів *Time*, який містить часи, пов'язані з даною вершиною.

Конструктори *Vertex()* та *Vertex(std::string n, float _x, float _y, std::vector<Time> t)* відповідають за ініціалізацію об'єкту вершини. Перший конструктор ініціалізує вершину з порожніми значеннями для усіх полів, тоді як другий конструктор приймає параметри:

- *n*: Ім'я вершини.
- *_x, _y*: Координати вершини на площині.
- *t*: Вектор часів, що пов'язаний з даною вершиною.

Таким чином, за допомогою цієї структури можна зручно представляти вершини графа разом з інформацією про їхнє ім'я, положення та часові характеристики.

```
struct Vertex {
    std::string name;
    float x, y;
    std::vector<Time> times;
    Vertex() : name(""), x(0.0f), y(0.0f) {}
    Vertex(std::string n, float _x, float _y, std::vector<Time> t) :
        name(n), x(_x), y(_y), times(t) {}
};
```

Рис. 2.3. Структура Vertex

Дані, що стосуються міст та часу проходження потягів через вказані станції мають ключове значення для планування подорожей, визначення оптимальних маршрутів та забезпечення пунктуальності руху [8].

Для кращого розуміння цих даних, їх було систематизовано та представлено у вигляді фрагменту коду *vertexCoordinates* з ДОДАТКУ Д з відповідними містами та часом проходження потягом кожної станції.

Структура *Edge* відіграє ключову роль у моделюванні ребер графа – рисунок 2.4. У полях структури зберігається наступна інформація:

- *from*: Рядок, що містить ім'я початкової вершини ребра.
- *to*: Рядок, що містить ім'я кінцевої вершини ребра.
- *cost*: Величина, що відображає вартість переміщення між цими містами.

```
struct Edge {
    std::string from, to;
    float cost;
};
```

Рис. 2.4. Структура Edge

Кожен елемент у векторі *edges*, наведений у ДОДАТКУ Д, є об'єктом типу *Edge*, що представляє зв'язок між двома вершинами графа.

Важливим аспектом кваліфікаційної роботи є візуалізація залізничної мережі та найкоротшого шляху між двома станціями. Для цього було використано бібліотеку SFML, яка дозволяє створювати графічні інтерфейси та працювати з графікою. Використовуючи вище вказані структури було побудовано граф для моделювання залізничної мережі – рисунок 2.5. Залізнична мережа моделюється як граф, де вершини представляють залізничні станції, а ребра – залізничні маршрути між станціями з відповідними вартостями проїзду.

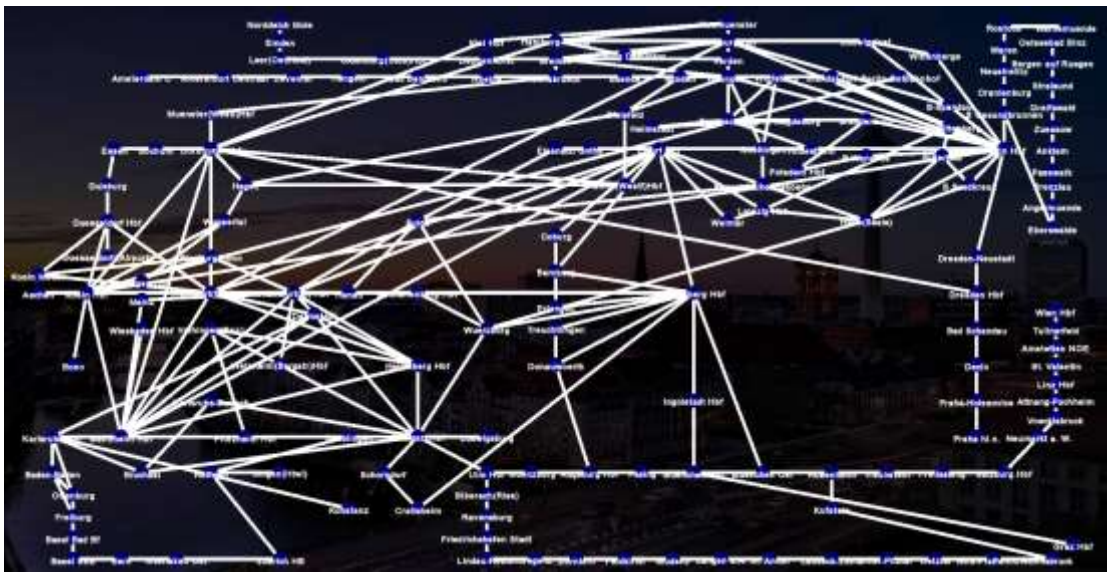


Рис. 2.5. Модель залізничної мережі

2.2 Пошук за методом Дейкстри

У цьому розділі алгоритм Дейкстри використовується для пошуку найкоротших шляхів між станціями в залізничній мережі DB, включаючи високошвидкісні поїзди ICE та IC.

Алгоритм аналізує залізничну мережу, починаючи з пошуку найкоротшого маршруту між двома визначеними станціями. У цьому конкретному випадку знайдемо оптимальний маршрут від станції Berlin Hbf до станції Karlsruhe Hbf. Це стратегічно важливе завдання, оскільки обидві станції є ключовими вузлами у залізничній системі Німеччини.

Початкова точка маршруту, Berlin Hbf, є однією з найбільш зайнятих та важливих станцій у Німеччині, з'єднуючи місто Берлін з різними регіонами країни та зовнішніми напрямками. У той час як Karlsruhe Hbf служить ключовим вузлом на півдні країни, забезпечуючи зв'язок з іншими важливими містами та регіонами.

Процес пошуку найкоротшого маршруту між цими двома станціями передбачає аналіз та оцінку різних можливих шляхів, враховуючи час подорожі та кількість пересадок.

З метою наочного представлення етапів та процесу використання алгоритму Дейкстри, розроблено блок-схему алгоритму, яка надається у ДОДАТКУ Е, що сприяє кращому розумінню ілюстрованого алгоритму.

У функції *dijkstra_time_multiple*, наданій у ДОДАТКУ Д, реалізовано варіацію алгоритму Дейкстри, яка знаходить кілька найкоротших шляхів між двома вузлами у графі. Опишемо цей алгоритм.

Крок 1. *Ініціалізація.* Починаємо з ініціалізації порожнього вектора *shortest_paths*, який буде містити знайдені найкоротші шляхи. Копіюємо вектор ребер *original_edges* до змінної *edges*.

Крок 2. Цикл пошуку шляхів. Поки не знайдено достатню кількість найкоротших шляхів (*shortest_paths.size() < num_paths_to_find*), виконуємо *кроки 3 − 7*.

Крок 3. Ініціалізація даних. Створюємо *shortest_time* – словник для зберігання часу найкоротших шляхів до кожної вершини, і *previous_station* - словник для зберігання попередніх вершин у найкоротших шляхах. Створюємо пріоритетну чергу *pq*, де ключем є час до вершини, що ще не відвідана.

Крок 4. Ініціалізація часу. Для кожної вершини у *vertices* встановлюємо початковий час як *infinity*, за винятком початкової вершини, до якої встановлюємо час як 0. Додаємо початкову вершину до черги.

Крок 5. Пошук найкоротших шляхів. Поки черга не порожня, вибираємо вершину з найменшим часом *current_station* і розглядаємо всі ребра, що виходять з цієї вершини за формулою 1.3. Для кожного ребра оновлюємо час до сусідньої вершини, якщо новий час менший за поточний.

Крок 6. Формування шляху. Якщо досягнута кінцева вершина, будуємо найкоротший шлях, відштовхуючись від *previous_station*. Перевіряємо, чи цей шлях унікальний, і додаємо його до *shortest_paths*.

Крок 7. Оновлення даних. Очищаємо дані, що використовувались для пошуку поточного набору шляхів, і оновлюємо список ребер, видаляючи ті, які входять у знайдені шляхи.

Крок 8. Повернення результату. Повертаємо знайдені найкоротші шляхи.

Функція *filter_edges*, зображена на рисунку 2.6, виконує фільтрацію ребер графа шляхом вилучення тих ребер, які вже були використані у знайдених найкоротших шляхах під час виконання алгоритму Дейкстри, як вказано у *кроці 7*. Основна мета цієї функції полягає у видаленні ребер, що належать знайденим найкоротшим шляхам, з загального списку ребер. Це необхідно для того, щоб уникнути використання тих самих ребер у подальших ітераціях алгоритму та забезпечити знаходження нових найкоротших шляхів.

```
std::vector<Edge> filter_edges(const std::vector<Edge>& edges,
                               const std::unordered_set<std::string>& visited,
                               const std::vector<std::pair<std::vector<std::string>,
                               Time>>& shortest_paths) {
    std::set<std::pair<std::string, std::string>> used_edges;

    for (const auto& path : shortest_paths) {
        for (size_t i = 0; i < path.first.size() - 1; ++i) {
            used_edges.emplace(path.first[i], path.first[i + 1]);
        }
    }

    std::vector<Edge> filtered_edges;
    for (const auto& edge : edges) {
        if (used_edges.find(std::make_pair(edge.from, edge.to)) == used_edges.end()) {
            filtered_edges.push_back(edge);
        }
    }

    return filtered_edges;
}
```

Рис. 2.6. Функція filter_edges

Після обчислення 3 найкоротших шляхів між двома станціями, Berlin Hbf та Karlsruhe Hbf, за алгоритмом Дейкстри, та поїздка, яка триває найменше всього часу, візуалізується на карті – рисунок 2.7. Це забезпечує наочне представлення результатів роботи алгоритму Дейкстри та дозволяє користувачам легко візуалізувати найкоротший маршрут.

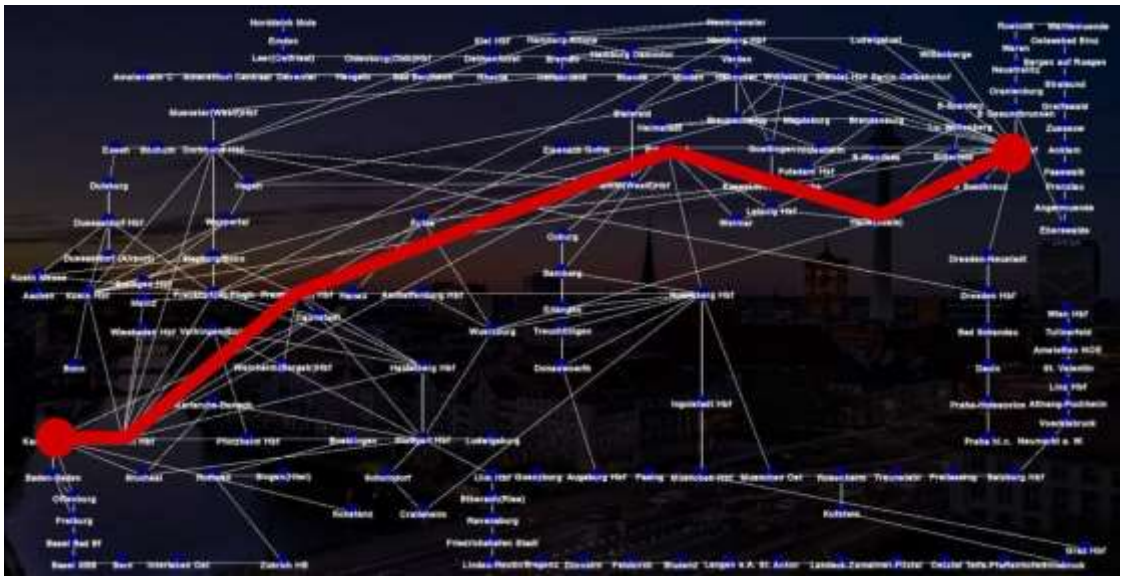


Рис. 2.7. Найкоротший шлях за алгоритмом Дейкстри

На рисунку 2.8 наведено частину коду, яка відповідає за виведення мапи з найкоротшим шляхом на екран. В циклі перевіряється, чи існують знайдені найкоротші шляхи, і якщо так, то обирається перший з них для візуалізації.

```
for (const auto& vertex : vertexCoordinates) {
    sf::CircleShape circle(10);
    circle.setFillColor(sf::Color(0, 0, 128));
    circle.setPosition(vertex.second.x - 10, vertex.second.y - 10);
    window.draw(circle);

    sf::Text text(vertex.second.name, font, 12);
    text.setFillColor(sf::Color(255, 255, 255));
    text.setStyle(sf::Text::Bold);
    float textX = vertex.second.x - text.getLocalBounds().width / 2;
    float textY = vertex.second.y - text.getLocalBounds().height / 2;
    text.setPosition(textX, textY);
    window.draw(text);
}

if (!shortest_paths.empty()) {
    const auto& first_path = shortest_paths.front();

    for (size_t i = 0; i < first_path.first.size() - 1; ++i) {
        auto from = vertexCoordinates[first_path.first[i]];
        auto to = vertexCoordinates[first_path.first[i + 1]];

        sf::Vertex line[] = {
            sf::Vertex(sf::Vector2f(from.x, from.y), sf::Color::Red),
            sf::Vertex(sf::Vector2f(to.x, to.y), sf::Color::Red)
        };

        window.draw(line, 2, sf::Lines);
    }
}
```

Рис. 2.8. Цикл для відображення найкоротшого шляху

У процесі виконання програми, окрім візуалізації на графічному інтерфейсі, результати роботи алгоритму Дейкстри також виводяться на консоль. Це дозволяє користувачеві отримати детальну інформацію про знайдені маршрути, включаючи список станцій, кількість пересадок, загальний час у дорозі та вартість поїздки. Такий вивід допомагає користувачам зручно аналізувати різні варіанти маршрутів, обирати найоптимальніший з них залежно від часу, кількості пересадок або вартості поїздки, і є додатковим підтвердженням правильності виконання алгоритму. На рисунку 2.9 показані результати виконання алгоритму Дейкстри, де початковою станцією є Berlin Hbf, а кінцевою – Karlsruhe Hbf.

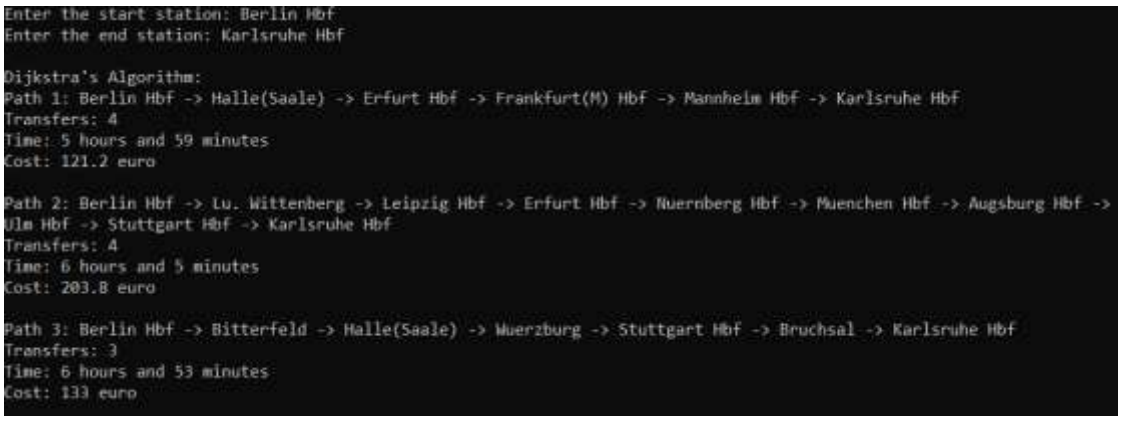


Рис. 2.9. Вивід результатів на консоль

Таким чином, реалізація алгоритму Дейкстри в контексті залізничної мережі дозволяє ефективно планувати маршрути, знаходячи найкоротші шляхи з врахуванням різних параметрів, таких як час і вартість поїздки. Це демонструє можливість застосування алгоритмічних методів для вирішення практичних завдань у реальних системах.

2.3 Пошук за методом A*

Розглянемо практичне застосування алгоритму A* для знаходження найкоротшого шляху у залізничній мережі та визначимо як вибирати та реалізовувати евристичні функції для цього конкретного контексту.

У даному розділі для демонстрації алгоритму обрано станції Berlin Hbf та Karlsruhe Hbf, аналогічно до розділу 2.2. Це забезпечить можливість порівняння результатів пошуку найкоротшого шляху, отриманих за допомогою методу A*, з результатами, здобутими за алгоритмом Дейкстри.

У методі A* реалізується оцінка оптимального маршруту за допомогою двох параметрів: часу подорожі та вартості квитка.

Використання функції оцінювання з двома параметрами дозволяє алгоритму A* ефективніше керувати процесом пошуку найкоротшого шляху в графі. По-перше, час подорожі дозволяє оцінити загальну витрату часу на досягнення кожної вершини від початкової, враховуючи швидкість руху та інші параметри, що впливають на тривалість подорожі між вершинами.

З іншого боку, евристична оцінка залишкової вартості (ціна квитка) надає приблизну інформацію про вартість досягнення кінцевої вершини з поточної позиції. Ця оцінка дозволяє алгоритму оцінювати складність досягнення кінцевої вершини та враховувати її при прийнятті рішення про напрямок пошуку.

Поєднання цих двох параметрів у функції оцінювання дозволяє алгоритму A* знаходити оптимальний шлях, враховуючи як час подорожі, так і вартість квитків. Такий підхід допомагає забезпечити ефективний та швидкий пошук найкращого рішення у задачах маршрутизації.

Основною особливістю цього алгоритму є використання евристичної функції, яка оцінює відстань від поточної вершини до кінцевої мети.

З метою наочного представлення етапів та процесу використання евристичної функції, розроблено блок-схему алгоритму, яка надається у ДОДАТКУ Е.

Функція *calculate_heuristic*, показана на рисунку 2.10, призначена для обчислення евристичної оцінки вартості подорожі від поточної вершини (*current_vertex*) до кінцевої вершини (*end_vertex*). Ця оцінка використовується для прийняття рішень в алгоритмі A*, дозволяючи оптимізувати пошук найкоротшого шляху.

```
Time calculate_heuristic(const Vertex& current_vertex,
                         const Vertex& end_vertex,
                         const std::vector<Edge>& edges) {
    float total_cost = std::numeric_limits<float>::max();

    for (const auto& edge : edges) {
        if (edge.from == current_vertex.name && edge.to == end_vertex.name) {
            total_cost = std::min(total_cost, edge.cost);
        }
    }

    if (total_cost == std::numeric_limits<float>::max()) {
        return Time(0, 0);
    }


    return Time(0, static_cast<int>(total_cost)) + current_vertex.times[0];
}
```

Рис. 2.10. Функція calculate_heuristic

Опишемо цей алгоритм.

Крок 1. Ініціалізація змінної *total_cost* як максимального значення.

```
float total_cost = std::numeric_limits<float>::max();
```

Крок 2. Перебір всіх ребер графу та оновлення *total_cost* з мінімальною вартістю.

```
for (const auto& edge : edges) {
    if (edge.from == current_vertex.name && edge.to == end_vertex.name) {
        total_cost = std::min(total_cost, edge.cost);
    }
}
```

Крок 3. Повернення нульового часу, якщо прямого шляху між вершинами немає, або обчислення часу подорожі як суми *total_cost* і часу поточної вершини.

```
if (total_cost == std::numeric_limits<float>::max()) {
    return Time(0, 0);
}


return Time(0, static_cast<int>(total_cost)) + current_vertex.times[0];
```

Функція *a_star_time_multiple* у ДОДАТКУ Д реалізує алгоритм A* для пошуку найкоротших шляхів у напрямлених графах мережі з урахуванням часу подорожі та вартості квитків. Вона використовує евристичний підхід, що дозволяє здійснювати ефективний пошук шляху, враховуючи як часові обмеження, так і вартість квитків. Опишемо цей алгоритм.

Крок 1. Ініціалізація змінних. Спочатку функція ініціалізує необхідні змінні, такі як список найкоротших шляхів, список ребер, мапи для відстеження найкоротшого часу та попередніх вершин, а також пріоритетна черга для вибору наступної вершини для обробки.

Крок 2. Основний цикл. Функція працює у циклі, доки не буде знайдено потрібну кількість найкоротших шляхів. У кожній ітерації циклу шукається наступний найкоротший шлях від початкової до кінцевої вершини.

Крок 3. Пошук найкоротшого шляху. За допомогою пріоритетної черги обирається наступна вершина для розгляду. Для кожної вершини перевіряються всі вихідні ребра, і обчислюється час подорожі до кожної з них. Якщо новий шлях є коротшим, ніж попередній, оновлюється найкоротший час і зберігається інформація про попередню вершину.

Крок 4. Побудова найкоротшого шляху. Якщо шлях від початкової до кінцевої вершини знайдено, він зберігається, і процес повторюється для знаходження інших шляхів. В іншому випадку, якщо кінцева вершина недосяжна, процес завершується.

Крок 5. Повернення результатів. Найкоротші шляхи повертаються як вектор пар, де перший елемент пари - це шлях від початкової до кінцевої вершини, а другий - час подорожі цього шляху [9].

Після обчислення найкоротших шляхів за алгоритмом A*, результати відображаються на карті, де шляхи між станціями підсвічуються зеленим

кольором – рисунок 2.11. Це дозволяє візуалізувати знайдені шляхи та зрозуміти їх маршрут на мапі.

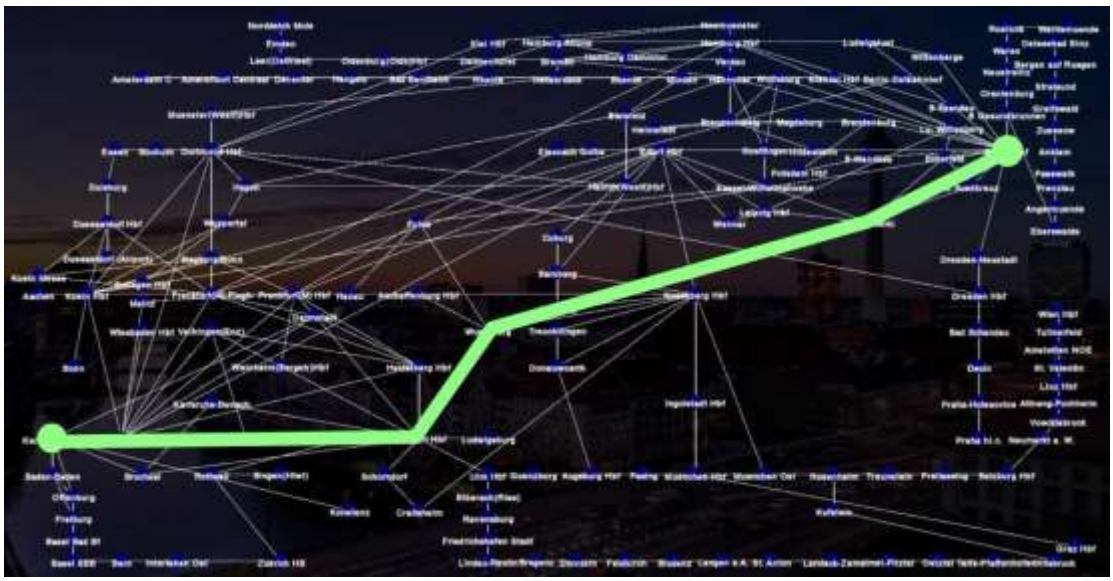


Рис. 2.11. Найкоротший шлях за алгоритмом A*

На рисунку 2.12 наведено частину коду, яка відповідає за виведення мапи з найкоротшим шляхом на екран. В циклі перевіряється, чи існують знайдені найкоротші шляхи, і якщо так, то обирається перший з них для візуалізації.

```
if (!a_star_shortest_paths.empty()) {
    const auto& first_path_a_star = a_star_shortest_paths.front();

    for (size_t i = 0; i < first_path_a_star.first.size() - 1; ++i) {
        auto from = vertexCoordinates[first_path_a_star.first[i]];
        auto to = vertexCoordinates[first_path_a_star.first[i + 1]];

        sf::Vertex line[] = {
            sf::Vertex(sf::Vector2f(from.x, from.y), sf::Color::Green),
            sf::Vertex(sf::Vector2f(to.x, to.y), sf::Color::Green)
        };

        window.draw(line, 2, sf::Lines);
    }
}
```

Рис. 2.12. Цикл для відображення найкоротшого шляху

Програма дозволяє отримувати детальну інформацію про знайдені маршрути алгоритмом Дейкстри, виводячи результати на консоль. Результати виводяться у вигляді списків трьох маршрутів з початкової станції Berlin Hbf до кінцевої Karlsruhe Hbf, надаючи користувачам повну інформацію для вибору оптимального маршруту, що показано на рисунку 2.13.

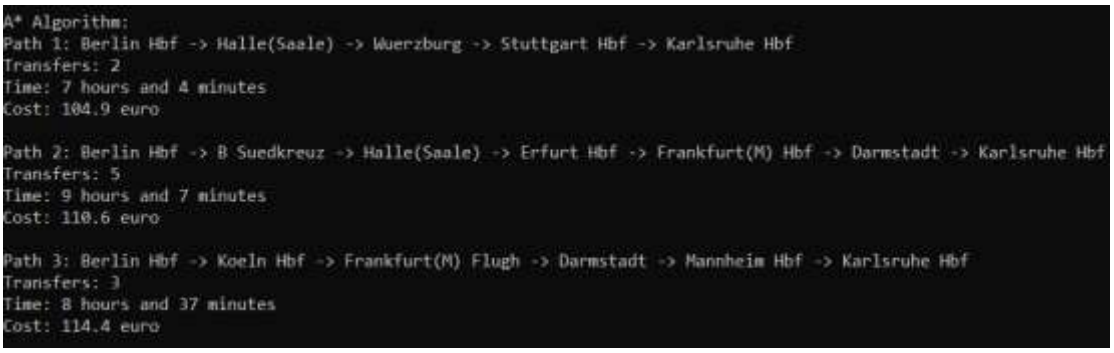


Рис. 2.13. Вивід результатів на консоль

Реалізація алгоритму A* для залізничної мережі відкриває можливість ефективного планування маршрутів, у яких одночасно враховуються час подорожі та вартість квитків. Це демонструє, як алгоритмічні методи можуть бути успішно використані для розв'язання реальних завдань. Використання алгоритму A* дає можливість оптимізувати маршрути, враховуючи обидва параметри одночасно, що дозволяє знаходити оптимальні рішення відповідно до поставлених вимог та обмежень.

2.4 Порівняльний аналіз алгоритмів

У цьому розділі буде проведено порівняльний аналіз двох алгоритмів пошуку найкоротших шляхів у залізничній мережі: алгоритму Дейкстри та алгоритму A*. Обидва алгоритми використовуються для знаходження оптимальних маршрутів, але мають різні підходи та критерії для оцінювання

шляхів. У процесі аналізу буде розглянуто ефективність, точність та особливості застосування кожного з алгоритмів.

Дослідження спрямоване на оцінку та порівняння ефективності алгоритмів Дейкстри та A* у пошуку найкоротших шляхів у графі. В процесі аналізу застосовуються кілька етапів, які зображені на блок-схемі на рисунку 2.14. Основною метою є визначення, який з цих алгоритмів краще справляється з ефективним пошуком найкоротшого шляху в заданому графі [10].

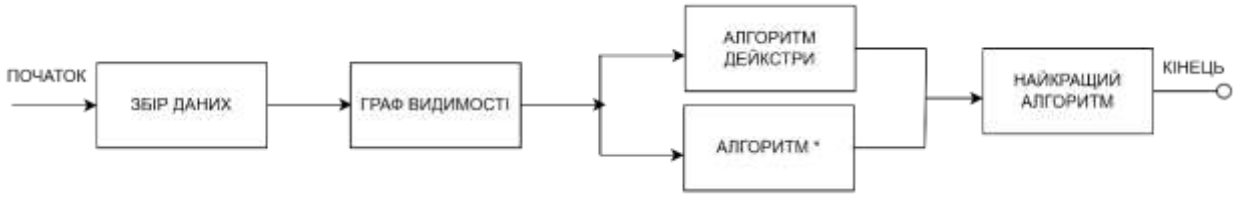


Рис. 2.14. Діаграма методології дослідження

Алгоритм Дейкстри є класичним методом пошуку найкоротших шляхів, який використовує єдиний критерій оцінювання – час подорожі. Основні етапи алгоритму включають:

- ініціалізація всіх вершин з нескінченно великою відстанню, крім початкової вершини, яка має нульову відстань;
- вибір вершини з найменшою поточною відстанню та перевірка всіх її сусідів;
- оновлення відстаней до сусідніх вершин, якщо через поточну вершину шлях коротший;
- повторення процесу до обробки всіх вершин або до знаходження кінцевої вершини.

Алгоритм A* є вдосконаленою версією алгоритму Дейкстри, яка включає додаткову евристику для оцінки залишкового шляху. Алгоритм використовує дві складові для оцінювання: час подорожі та вартість квитка. Основні етапи алгоритму A*:

- ініціалізація всіх вершин з нескінченно великою відстанню та евристичною оцінкою, крім початкової вершини, яка має нульову відстань та евристичну оцінку;
- вибір вершини з найменшою сумою поточної відстані та евристичної оцінки;
- перевірка всіх сусідів вибраної вершини та оновлення їх відстаней з урахуванням евристики;
- повторення процесу до обробки всіх вершин або до знаходження кінцевої вершини [11].

Для кращого розуміння відмінностей між алгоритмом Дейкстри та алгоритмом A* проведено їх порівняльний аналіз за основними критеріями. У таблиці 1.1 наведено порівняння цих двох алгоритмів за критеріями оцінювання, ефективності та складності реалізації.

Таблиця 2.1

Порівняння алгоритмів Дейкстри та A*

Критерій	Алгоритм Дейкстри	Алгоритм A*
Критерії оцінювання	Використовує єдиний критерій – мінімальний час подорожі	Поєднує два критерії – мінімальний час подорожі та вартість квитка, що дозволяє отримати більш оптимальні результати з урахуванням витрат
Ефективність	Гарантує знаходження найкоротшого шляху з точки зору часу	Може бути більш ефективним у випадках, коли важливо враховувати як час, так і вартість, завдяки використанню евристичної оцінки
Складність реалізації	Простий у реалізації та зрозумілий з точки зору теоретичних основ	Потребує додаткового налаштування евристичної функції, що може бути складнішим завданням, але дозволяє досягати кращих результатів у деяких випадках

Використання алгоритмів Дейкстри та A* для пошуку найкоротшого маршруту у залізничних мережах виявляється критичним, де вони забезпечують практично миттєвий результат. Однак на великому масштабі мапи A* забезпечує швидше рішення, ніж Дейкстра. Алгоритм A* сканує область лише у напрямку до цільової точки за допомогою евристичної функції, що враховується під час обчислень. У той час, як Дейкстра досліджує область рівномірно у всіх напрямках і, як правило, досліджує набагато більшу область перед досягненням цільової точки, що робить його повільнішим, ніж A*. Це можна підтвердити шляхом порівняння кількості ітерацій Дейкстри та A*, де більша кількість вузлів призводить до більшої різниці у кількості ітерацій або часу [12].

Порівнюючи результати, надані в таблицях 2.2 та 2.3, для маршруту від Berlin Hbf до Karlsruhe Hbf, можна зазначити, що алгоритм Дейкстри виявився ефективнішим у знаходженні швидших маршрутів з меншою кількістю пересадок. Проте варто відзначити, що алгоритм A* враховує не лише час подорожі, а й вартість квитка. Це дозволяє знайти більш економічні варіанти, хоча зазвичай займає трохи більше часу на пошук оптимального маршруту. Таким чином, алгоритм Дейкстри може бути швидшим у знаходженні шляхів, але алгоритм A* надає більш раціональні та економічні рішення, враховуючи обидва критерії: час подорожі та вартість.

Таблиця 2.2

Результат за алгоритмом Дейкстри

Кількість пересадок	Час	Вартість
4	5:59	121,20 €
4	6:05	203,80 €
3	6:53	133,00 €

Таблиця 2.3

Результат за алгоритмом A*

Кількість пересадок	Час	Вартість
2	7:04	104,90 €
5	9:07	110,60 €
3	8:37	114,40 €

Розглянемо детальніше на рисунках 2.15 та 2.16 показники тривалості та вартості поїздки, залежно від обраного алгоритму.

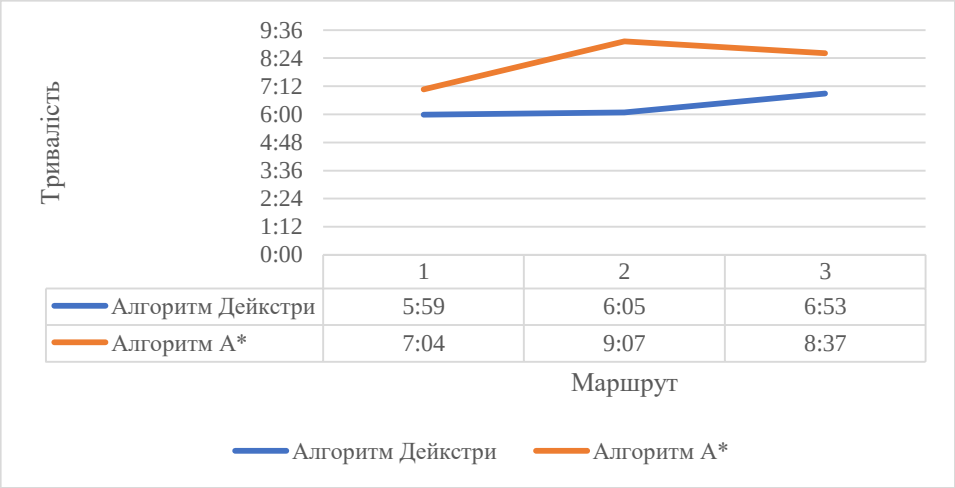


Рис. 2.15. Порівняння тривалості поїздки

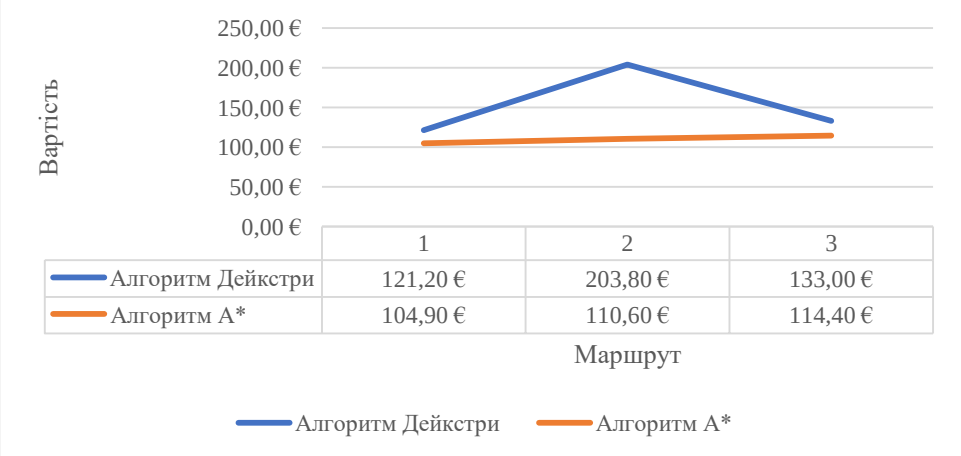


Рис. 2.16. Порівняння вартості поїздки

На рисунку 2.17 представлена карта з найкоротшими шляхами, знайденими кожним з цих алгоритмів. Червоний колір позначає шлях,

знайдений алгоритмом Дейкстри, тоді як зелений — шлях, знайдений алгоритмом A*.

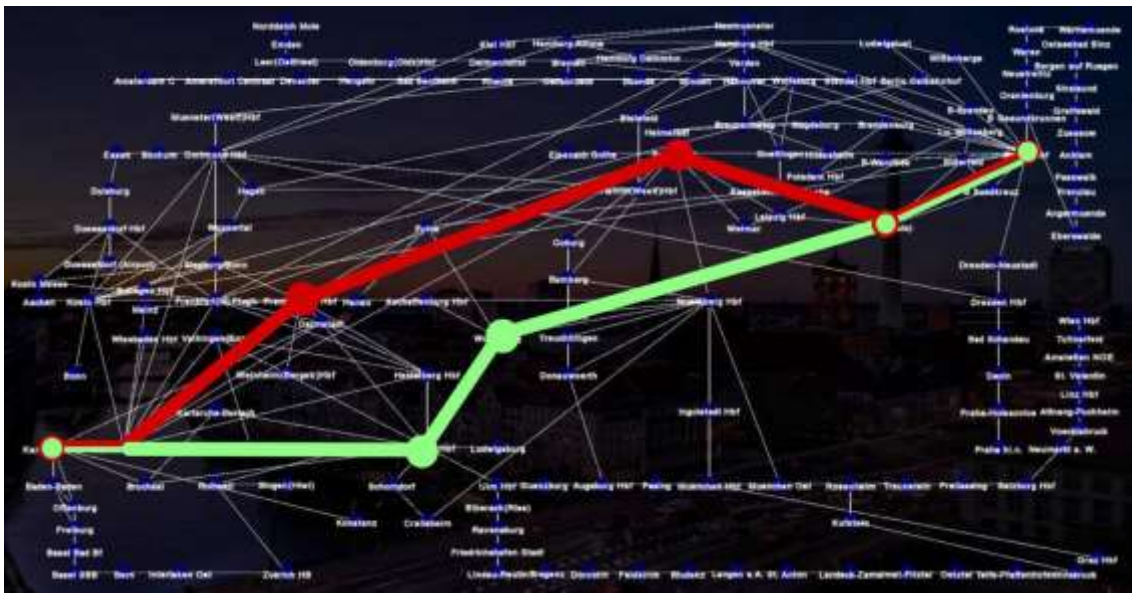


Рис. 2.17. Карта з найкоротшими шляхами

З аналізу видно, що алгоритм Дейкстри часто обирає більш прямолінійний шлях, оскільки він розглядає всі можливі варіанти рівномірно. У той же час, алгоритм A* швидше знаходить шляхи, які спрямовані більш точно до цілі, що дозволяє йому уникати деяких зайвих перевірок і зменшує кількість пройдених вузлів.

На основі наданих даних можна зробити висновок, що алгоритм Дейкстри і алгоритм A* успішно знаходять найкоротші маршрути від Berlin Hbf до Karlsruhe Hbf, проте з різними підходами до оптимізації. Алгоритм Дейкстри ефективніший у виявленні швидших маршрутів з меншою кількістю пересадок, в той час як алгоритм A* враховує і оптимізує і час подорожі, і вартість квитків, надаючи більш економічні варіанти. Вибір між цими алгоритмами залежатиме від конкретних умов і вимог, таких як час подорожі та фінансові витрати, які можуть варіюватися залежно від потреб користувача.

2.5 Висновки до розділу

Під час розробки програмного забезпечення для оптимізації пасажирських перевезень з використанням DB особлива увага приділялась всім аспектам реалізації та функціональності. Початковий етап аналізу вхідних даних виявився критичним для правильного розуміння структури та потоків даних, які необхідно обробляти.

Алгоритм Дейкстри був впроваджений з метою забезпечення швидкого і точного пошуку найкоротших маршрутів між будь-якою парою станцій. Цей метод виявився ефективним для ситуацій, коли основним критерієм є мінімальний час подорожі, оскільки він розглядає всі можливі шляхи з ваговими функціями на ребрах графа.

У випадках, коли користувачів також цікавить економія коштів, було використано алгоритм A*. Цей метод дозволяє покращити ефективність пошуку, використовуючи евристичну оцінку відстані до цільової точки, що дозволяє зменшити кількість вузлів, які потрібно переглянути для досягнення цілі.

Порівняльний аналіз двох алгоритмів показав, що вони можуть бути доповненнями один одного в залежності від специфічних потреб користувачів. Дейкстрів алгоритм виявився переважним у випадках, коли пріоритетом є мінімізація часу, тоді як A* став кращим варіантом для використання в ситуаціях, де необхідно ураховувати і час, і вартість.

Загальною метою цього проекту було створення програмного забезпечення, яке не лише покращує процес планування маршрутів, але й сприяє забезпеченню зручності і задоволеності користувачів пасажирськими перевезеннями DB. Такий підхід дозволяє підтримувати високий стандарт

обслуговування і відповідати різноманітним потребам пасажирів у мобільності та швидкості пересування.

ВИСНОВКИ

В кваліфікаційній роботі було проведено дослідження з оптимізації пасажирських перевезень з використанням системного підходу та технологій Deutsche Bahn. Дослідження охоплювало кілька ключових етапів, включаючи теоретичні аспекти пасажирських перевезень, організацію технологічного процесу в Deutsche Bahn, аналіз існуючих проблем планування маршрутів, а також детальне вивчення і реалізацію алгоритмів Дейкстри і A* для вдосконалення цього процесу.

У розділі 1 було проведено аналіз теоретичних основ пасажирських перевезень, що дозволило зрозуміти основні принципи та вимоги до оптимального планування маршрутів у великих транспортних системах. Особлива увага була приділена організації технологічного процесу в Deutsche Bahn, його структурі та мультимодальному сервісу. Це підкреслило важливість інтеграції різних видів транспорту для забезпечення ефективності маршрутизації. Аналіз структури підприємства та характеристик діяльності DB показав, що складна організаційна структура та велика кількість задіяних елементів вимагають ретельного та зваженого підходу до планування.

Процес планування маршрутів та розкладів виявився складним завданням, яке стикається з рядом викликів. Основні з них включають необхідність врахування попиту пасажирів, забезпечення стиковок між різними видами транспорту та мінімізацію затримок. Специфічні проблеми планування маршрутів у DB включають великі розриви в покритті, погану інтеграцію між регіональними та національними маршрутами, а також часті технічні та інфраструктурні збої. Для подолання цих викликів було визначено важливість застосування методів оптимізації маршрутів та графіків. Використання передових алгоритмів, таких як алгоритм Дейкстри та алгоритм A*, дозволило значно підвищити ефективність планування. Алгоритм Дейкстри забезпечував знаходження найкоротших шляхів у графі з

невід'ємними вагами, що є корисним для базового планування маршрутів. Алгоритм A*, у свою чергу, враховував як відстань від джерела до вузла, так і оцінку відстані від вузла до цілі, що дозволило ефективніше знаходити оптимальні маршрути.

У розділі 2 було проведено детальний аналіз вхідних даних та реалізовано методи Дейкстри і A* для пошуку найкращих маршрутів. Результати цього аналізу показали, що обидва алгоритми мають свої переваги в залежності від конкретних умов і вимог користувачів. Алгоритм Дейкстри, завдяки своїй простоті та ефективності у малопотужних графах, дозволив знаходити швидкі маршрути. З іншого боку, алгоритм A* використовував евристичні оцінки для прискорення пошуку і зменшення обсягу обробки даних, що зробило його корисним для складних транспортних мереж із великою кількістю вузлів.

Порівняльний аналіз обох алгоритмів показав, що вони можуть доповнювати один одного, забезпечуючи комплексне планування маршрутів з урахуванням різних факторів, таких як час подорожі, вартість та кількість пересадок. Впровадження передових технологій та алгоритмів, а також врахування специфічних викликів і проблем, дозволило забезпечити більш надійне та комфортне транспортне обслуговування пасажирів.

Загальною метою цього проекту було створення програмного забезпечення, яке не лише покращує процес планування маршрутів, але й сприяє забезпеченню зручності і задоволеності користувачів пасажирськими перевезеннями DB. Такий підхід дозволив підтримувати високий стандарт обслуговування і відповідати різноманітним потребам пасажирів у мобільності та швидкості пересування.

Отже, проведене дослідження підтвердило, що оптимізація маршрутів та графіків руху є ключовим аспектом для покращення якості пасажирських перевезень у DB. Розроблені алгоритми та система дозволили значно підвищити ефективність планування маршрутів, зменшуючи час у дорозі та

кількість пересадок для пасажирів. Це сприяло підвищенню загальної задоволеності користувачів та ефективності роботи транспортної системи.

Завдяки реалізації запропонованих підходів, було досягнуто значного прогресу у вдосконаленні пасажирських перевезень, що забезпечило покращення якості обслуговування та задоволення потреб пасажирів. В майбутньому подальше вдосконалення системи та інтеграція новітніх технологій можуть ще більше підвищити ефективність транспортних мереж та задоволеність пасажирів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Загальний курс транспорту: навч. посібник / О. О. Соловйова, І. І. Висоцька, І. М. Герасименко. – К. : НАУ, 2019. – 244 с.
2. Дискретна математика: навч. посібник / В.В. Слєсарєв, І.В. Новицький, С.А. Ус. – М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». – Дніпро : НТУ «ДП», 2023. – 183 с.
3. Шевченко, Ю. О. (2022). Обробка і аналіз даних з використанням електронних таблиць. Частина І «Обробка даних». https://ir.nmu.org.ua/handle/123456789/162623

4. Шевченко, Ю. О. (2022). Обробка і аналіз даних з використанням електронних таблиць. Частина II «Аналіз даних та макроси». https://ir.nmu.org.ua/handle/123456789/162624
5. Deutsch Bahn. URL : https://www.deutschebahn.com/de (дата звернення 17.04.2024 р.).
6. Der DB-Konzern. Daten und Fakten 2022. URL – Режим доступу : https://www.deutschebahn.com/ – (дата звернення 20.04.2024 р.).
7. Integrierter Bericht 2023. URL: https://ibir.deutschebahn.com/2023/de/start/ – (дата звернення 01.05.2024 р.).
8. The Driving Factors of Passenger Transport / Gerard de Jong, Van de Riet, O. A. W. T. – European Journal of Transport and Infrastructure Research. – DOI: 10.18757/ejtir.2008.8.3.3348. License: CC BY. Significance b.v., 2008. – 250 p.
9. Pathfinding Algorithms in Graphs and Applications: Treball final de grau / Daniel Monzonís Laparra; Director: Dr. Antoni Benseny. – Barcelona: Facultat de Matemàtiques i Informàtica, Departament de Matemàtiques i Informàtica, 2019. – 60 p.
10. Bahnhofssuche. URL: https://www.bahnhof.de/suche (дата звернення 01.05.2024 р.).
11. Russell, S., & Norvig, P. (2021). "Artificial Intelligence: A Modern Approach." 4th ed. Pearson. Chapter 3: Informed Search and Exploration.
12. Wayahdi, M. R., Ginting, S. H. N., & Syahputra, D. (2021). Greedy, A-Star, and Dijkstra's Algorithms in Finding Shortest Path. International Journal of Advances in Data and Information Systems, 2(1), 45-52. DOI: 10.25008/ijadis.v2i1.1206. [CC BY-SA 4.0].

13. A Comparison of Pathfinding Algorithms URL: https://www.youtube.com/watch?v=GC-nBgi9r0U (дата звернення 15.05.2024 р.).
14. Rachmawati, D., & Gustin, L. (2020). Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem. Journal of Physics Conference Series, 1566(1), 012061. URL : https://doi.org/10.1088/1742-6596/1566/1/012061 (дата звернення 01.05.2024 р.).

ДОДАТОК А

Відомість матеріалів кваліфікаційної роботи

№ з/п	Позначення	Найменування	Кількість аркушів	Примітки
1				
2		Документація		
3				
4	САУ.КР.24.03.ПЗ	Пояснювальна записка	76	Формат А4
5				
6	САУ.КР.24.03.ДМ	Демонстраційний матеріал	15	Презентація на CD-R
7				
8	САУ.КР.24.03.КР	Копія роботи	1	Диск CD-R
9				

10								
11								
12								
13								
14								
15								
16								
17								
18								
					САУ.КР.24.12.ДА.ПЗ.			
Змін.	Аркуш	№ докум.	Підпис	Дата				
Розроб.		Голоденко			**Матеріали кваліфікаційної роботи**	Літ.	Аркуш	Аркушів
К. розд.		Желдак						
Керівн.		Желдак				НТУ «ДП», 12; 124-20-2		
Н.контр.		Хом’як						
Зав. каф.		Желдак						

ДОДАТОК Б

Відгук
на кваліфікаційну роботу бакалавра

студентки групи *124 – 20 – 2*
спеціальності *124 Системний аналіз*

Тема кваліфікаційної роботи: «Оптимізація пасажирських перевезень з використанням Deutsche Bahn»
Обсяг кваліфікаційної роботи: 77 стор.
Мета кваліфікаційної роботи: підвищення зручності та ефективності переміщення пасажирів за рахунок алгоритмів для оптимізації пасажирських перевезень.
Актуальність теми _____
_____
_____

Тема кваліфікаційної роботи безпосередньо пов'язана з об'єктом діяльності бакалавра спеціальності 124 Системний аналіз, оскільки _____
_____
_____

Виконані в кваліфікаційній роботі завдання відповідають вимогам ступеня бакалавра. Оригінальність наукових рішень полягає в _____
_____
_____
_____

Практичне значення результатів кваліфікаційної роботи полягає в _____
_____
_____

Висновки підтверджують можливість використання результатів роботи в _____
_____
_____

Оформлення пояснювальної записки та демонстраційного матеріалу до неї виконано згідно з вимогами. Роботу виконано самостійно, відповідно до завдання та у повному обсязі (*в разі невідповідності – вказати*)
У роботі відзначено такі недоліки: _____
_____
_____

Кваліфікаційна робота в цілому заслуговує оцінки:_____

З урахуванням висловлених зауважень автор (не) заслуговує присвоєння освітньої кваліфікації «бакалавр з системного аналізу».

Керівник кваліфікаційної роботи бакалавра,
науковий ступінь, вчене звання, посада _____ / ПІБ

ДОДАТОК В

Рецензія
на кваліфікаційну роботу бакалавра
студентки групи *124 – 20 – 2*
спеціальності *124 Системний аналіз*

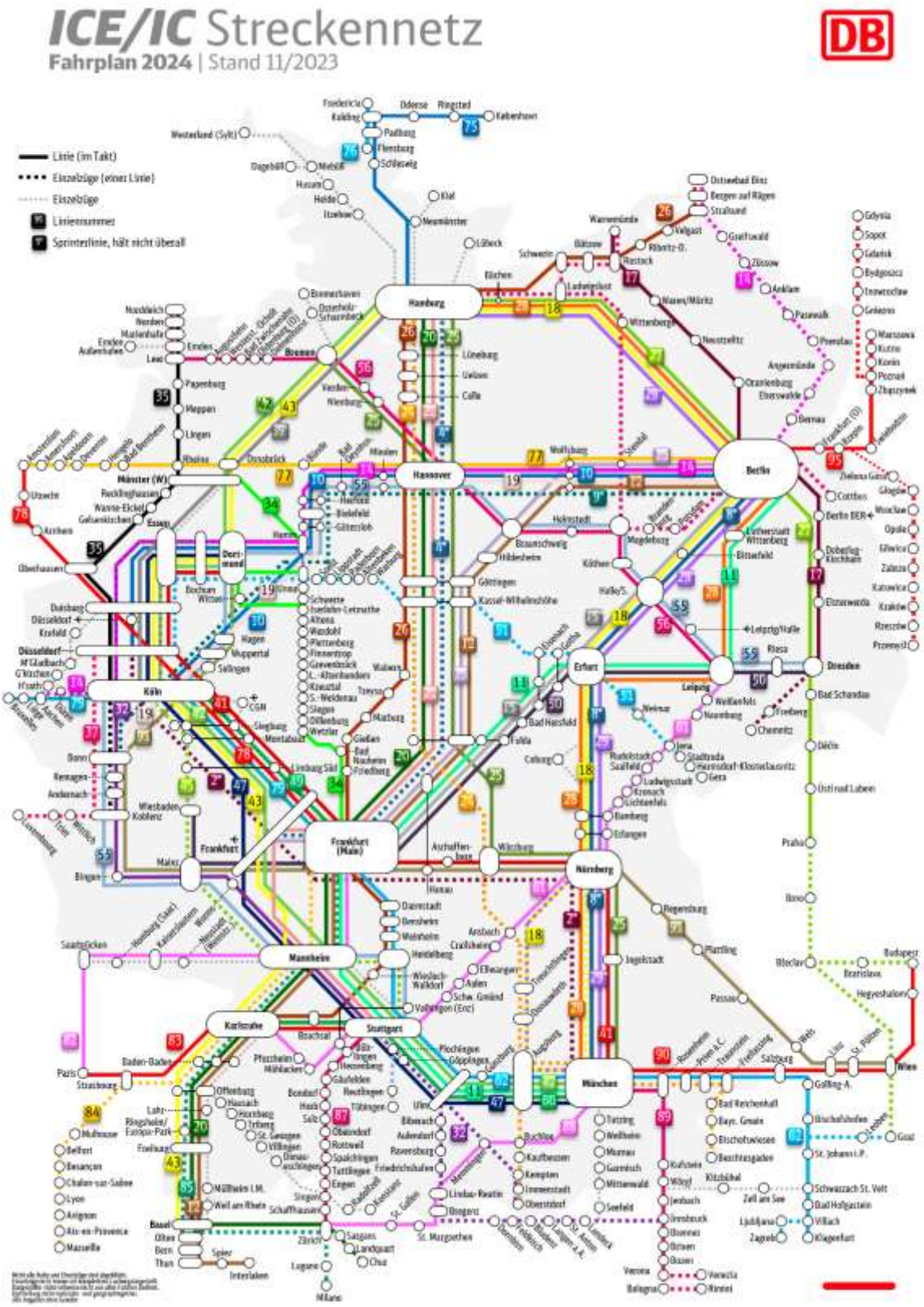
Тема кваліфікаційної роботи:
«Оптимізація пасажирських перевезень з використанням Deutsche Bahn»
Обсяг кваліфікаційної роботи: 77 стор.
Висновок про відповідність кваліфікаційної роботи завданню та освітньо-професійній програмі спеціальності _____
_____
_____
Загальна характеристика кваліфікаційної роботи, ступінь використання нормативно-методичної літератури та передового досвіду
_____
_____
_____
Позитивні сторони кваліфікаційної роботи:
_____
_____
_____

Основні недоліки кваліфікаційної роботи:
_____
_____
_____
_____
Кваліфікаційна робота в цілому заслуговує оцінки: _____

З урахуванням висловлених зауважень автор (не) заслуговує присвоєння освітньої кваліфікації «бакалавр з системного аналізу».

Рецензент,
науковий ступінь, вчене звання, посада _____ /
ПІБ

ДОДАТОК Г. ЗАЛІЗНИЧНА МЕРЕЖА



«Рисунок Г.1 – Залізнична мережа ICE/IC станом на 2024 рік»

ДОДАТОК Д. ТЕКСТ ПРОГРАМНОГО МОДУЛЮ

```
#include <SFML/Graphics.hpp>
#include <SFML/System.hpp>
#include <vector>
#include <string>
#include <map>
#include <iostream>
#include <queue>
#include <limits>
#include <algorithm>
#include <unordered_set>
#include <set>


struct Time {
    int hours;
    int minutes;

    Time() : hours(0), minutes(0) {}
    Time(int h, int m) : hours(h), minutes(m) {}
};

Time operator-(const Time& t1, const Time& t2) {
    int h = t1.hours - t2.hours;
    int m = t1.minutes - t2.minutes;
    if (m < 0) {
        m += 60;
        h--;
    }
    return Time(h, m);
}

Time operator+(const Time& t1, const Time& t2) {
    int total_minutes1 = t1.hours * 60 + t1.minutes;
    int total_minutes2 = t2.hours * 60 + t2.minutes;
    int total_minutes = total_minutes1 + total_minutes2;
    int h = total_minutes / 60;
    int m = total_minutes % 60;
    return Time(h, m);
}

bool operator>(const Time& t1, const Time& t2) {
    if (t1.hours > t2.hours) {
        return true;
    }
    else if (t1.hours == t2.hours) {
        return t1.minutes > t2.minutes;
    }
    return false;
}

bool operator<(const Time& t1, const Time& t2) {
    if (t1.hours < t2.hours) {
        return true;
    }
    else if (t1.hours == t2.hours) {
        return t1.minutes < t2.minutes;
    }
    return false;
}

bool operator>=(const Time& t1, const Time& t2) {
    if (t1.hours > t2.hours) {
        return true;
```

```
    }
    else if (t1.hours == t2.hours) {
        return t1.minutes >= t2.minutes;
    }
    return false;
}

struct Vertex {
    std::string name;
    float x, y;
    std::vector<Time> times;
    Vertex() : name(""), x(0.0f), y(0.0f) {}
    Vertex(std::string n, float _x, float _y, std::vector<Time> t) :
        name(n), x(_x), y(_y), times(t) {}
};

struct Edge {
    std::string from, to;
    float cost;
};

std::vector<Edge> filter_edges(const std::vector<Edge>& edges, const std::unordered_set<std::string>& visited, const
std::vector<std::pair<std::vector<std::string>, Time>>& shortest_paths) {
    std::set<std::pair<std::string, std::string>> used_edges;

    for (const auto& path : shortest_paths) {
        for (size_t i = 0; i < path.first.size() - 1; ++i) {
            used_edges.emplace(path.first[i], path.first[i + 1]);
        }
    }

    std::vector<Edge> filtered_edges;
    for (const auto& edge : edges) {
        if (used_edges.find(std::make_pair(edge.from, edge.to)) == used_edges.end()) {
            filtered_edges.push_back(edge);
        }
    }

    return filtered_edges;
}


struct Time;
std::vector<std::pair<std::vector<std::string>, Time>> dijkstra_time_multiple(const std::map<std::string, Vertex>& vertices,
const std::vector<Edge>& original_edges, const std::string& start, const std::string& end, int num_paths_to_find) {
    std::vector<std::pair<std::vector<std::string>, Time>> shortest_paths;

    std::vector<Edge> edges = original_edges;

    while (shortest_paths.size() < num_paths_to_find) {
        std::map<std::string, Time> shortest_time;
        std::map<std::string, std::string> previous_station;
        std::priority_queue<std::pair<Time, std::string>, std::vector<std::pair<Time, std::string>>, std::greater<std::pair<Time,
std::string>>> pq;

        for (const auto& vertex : vertices) {
            shortest_time[vertex.first] = Time(std::numeric_limits<int>::max(), std::numeric_limits<int>::max());
        }

        shortest_time[start] = Time(0, 0);
        pq.push(std::make_pair(Time(0, 0), start));

        std::unordered_set<std::string> visited;

        while (!pq.empty()) {
            auto current_station = pq.top().second;
            pq.pop();

            if (visited.find(current_station) != visited.end()) {
```

```
            continue;
        }
        visited.insert(current_station);

        for (const auto& edge : edges) {
            if (edge.from == current_station) {
                Time travel_time(0, 0);
                for (const auto& time : vertices.at(current_station).times) {
                    if (time.hours == shortest_time[current_station].hours && time.minutes >=
shortest_time[current_station].minutes) {
                        travel_time = time - shortest_time[current_station];
                        break;
                    }
                    else if (time.hours > shortest_time[current_station].hours) {
                        travel_time = time - shortest_time[current_station];
                        break;
                    }
                }

                if (shortest_time[edge.to] > shortest_time[current_station] + travel_time) {
                    shortest_time[edge.to] = shortest_time[current_station] + travel_time;
                    previous_station[edge.to] = current_station;
                    pq.push(std::make_pair(shortest_time[edge.to], edge.to));
                }
            }
        }
    }


    if (previous_station.find(end) != previous_station.end()) {
        std::vector<std::string> shortest_path;
        std::string current = end;
        while (current != start) {
            shortest_path.push_back(current);
            current = previous_station[current];
        }
        shortest_path.push_back(start);

        std::reverse(shortest_path.begin(), shortest_path.end());


        bool is_unique = true;
        for (const auto& path : shortest_paths) {
            if (path.first == shortest_path) {
                is_unique = false;
                break;
            }
        }

        if (is_unique) {
            shortest_paths.push_back(std::make_pair(shortest_path, shortest_time[end]));
        }


        visited.clear();
        edges = filter_edges(original_edges, visited, shortest_paths);
    }
    else {
        break;
    }
  }

  return shortest_paths;
}

Time calculate_heuristic(const Vertex& current_vertex,
                         const Vertex& end_vertex,
                         const std::vector<Edge>& edges) {
  float total_cost = std::numeric_limits<float>::max();
```

```
    for (const auto& edge : edges) {
        if (edge.from == current_vertex.name && edge.to == end_vertex.name) {
            total_cost = std::min(total_cost, edge.cost);
        }
    }

    if (total_cost == std::numeric_limits<float>::max()) {
        return Time(0, 0);
    }


    return Time(0, static_cast<int>(total_cost)) + current_vertex.times[0];
}



std::vector<std::pair<std::vector<std::string>, Time>> a_star_time_multiple(const std::map<std::string, Vertex>& vertices, const
std::vector<Edge>& original_edges, const std::string& start, const std::string& end, int num_paths_to_find) {
    std::vector<std::pair<std::vector<std::string>, Time>> shortest_paths;

    std::vector<Edge> edges = original_edges;

    while (shortest_paths.size() < num_paths_to_find) {
        std::map<std::string, Time> shortest_time;
        std::map<std::string, std::string> previous_station;
        std::priority_queue<std::pair<Time, std::string>, std::vector<std::pair<Time, std::string>>, std::greater<std::pair<Time,
std::string>>> pq;

        for (const auto& vertex : vertices) {
            shortest_time[vertex.first] = Time(std::numeric_limits<int>::max(), std::numeric_limits<int>::max());
        }

        shortest_time[start] = Time(0, 0);
        pq.push(std::make_pair(Time(0, 0), start));

        std::unordered_set<std::string> visited;

        while (!pq.empty()) {
            auto current_station = pq.top().second;
            pq.pop();

            if (visited.find(current_station) != visited.end()) {
                continue;
            }
            visited.insert(current_station);

            for (const auto& edge : edges) {
                if (edge.from == current_station) {
                    Time travel_time(0, 0);
                    for (const auto& time : vertices.at(current_station).times) {
                        if (time.hours == shortest_time[current_station].hours && time.minutes >=
shortest_time[current_station].minutes) {
                            travel_time = time - shortest_time[current_station];
                            break;
                        }
                        else if (time.hours > shortest_time[current_station].hours) {
                            travel_time = time - shortest_time[current_station];
                            break;
                        }
                    }
                    if (shortest_time[edge.to] > shortest_time[current_station] + Time(0, static_cast<int>(edge.cost))) {
                        shortest_time[edge.to] = shortest_time[current_station] + Time(0, static_cast<int>(edge.cost));
                        previous_station[edge.to] = current_station;
                        pq.push(std::make_pair(shortest_time[edge.to] + calculate_heuristic(vertices.at(edge.to), vertices.at(end), edges),
edge.to));
                    }
                }
```

```
            }
        }

        if (previous_station.find(end) != previous_station.end()) {
            std::vector<std::string> shortest_path;
            std::string current = end;
            while (current != start) {
                shortest_path.push_back(current);
                current = previous_station[current];
            }
            shortest_path.push_back(start);

            std::reverse(shortest_path.begin(), shortest_path.end());

            bool is_unique = true;
            for (const auto& path : shortest_paths) {
                if (path.first == shortest_path) {
                    is_unique = false;
                    break;
                }
            }

            if (is_unique) {
                shortest_paths.push_back(std::make_pair(shortest_path, shortest_time[end]));
            }

            visited.clear();
            edges = filter_edges(original_edges, visited, shortest_paths);
        }
        else {
            break;
        }
    }

    return shortest_paths;
}


int count_transfers(const std::vector<std::string>& path, const std::map<std::string, Vertex>& vertices, const
std::vector<Edge>& edges) {
    if (path.size() <= 1) {
        return 0;
    }

    int num_transfers = 0;
    Time arrival_time = vertices.at(path[0]).times[0];

    for (size_t i = 1; i < path.size(); ++i) {
        const std::string& current_station = path[i];
        const std::string& previous_station = path[i - 1];

        const Edge* edge = nullptr;
        for (const auto& e : edges) {
            if (e.from == previous_station && e.to == current_station) {
                edge = &e;
                break;
            }
        }

        if (edge == nullptr) {
            continue;
        }

        const Vertex& current_vertex = vertices.at(current_station);
        const Time& current_arrival_time = arrival_time;
        bool waiting = false;
        for (size_t j = 0; j < current_vertex.times.size(); ++j) {
            if (current_vertex.times[j] > current_arrival_time && current_vertex.times[j] - current_arrival_time >= Time(0, 10)) {
                waiting = true;
```

```
            break;
        }
    }

    if (waiting) {
        num_transfers++;
    }


    arrival_time = current_vertex.times.back();
}

return num_transfers;
}



int main() {
    std::string start_station, end_station;
    sf::RenderWindow window(sf::VideoMode(1600, 800), "ICE Trains Graph");

    std::map<std::string, Vertex> vertexCoordinates = {
    {"Stuttgart Hbf", {"Stuttgart Hbf", 600, 600, {{0, 11}, {1, 43}, {2, 16}, {2, 22}, {2, 26}, {3, 39}, {5, 02}, {5, 8}, {5, 12}, {5,
38}, {5, 45}, {5, 51}, {5, 52}, {6, 2}, {6, 5}, {6, 16}, {6, 20}, {6, 24}, {6, 26}, {6, 36}, {6, 50}, {7, 4}, {7, 10}, {7, 13}, {7,
15}, {7, 25}, {7, 27}, {7, 28}, {7, 36}, {7, 45}, {7, 51}, {7, 59}, {8, 31}, {8, 44}, {9, 1}, {9, 45}, {9, 52}, {9, 59}, {10, 38},
{11, 8}, {12, 38}, {12, 48}, {13, 8}, {14, 38}}}},
    {"Heidelberg Hbf", {"Heidelberg Hbf", 600, 500, {{1, 23}, {3, 35}, {6, 33}, {6, 54}, {7, 18}, {7, 53}, {9, 11}}}},
    {"Frankfurt(M) Flugh", {"Frankfurt(M) Flugh", 300, 400, {{4, 57}, {4, 59}, {6, 36}, {7, 6}, {7, 50}, {7, 51}, {8, 20}, {8, 36},
{8, 39}, {9, 6}, {9, 18}, {9, 20}, {9, 51}, {10, 20}, {11, 6}, {11, 18}, {11, 20}}}},
    {"Frankfurt(M) Hbf", {"Frankfurt(M) Hbf", 420, 400, {{4, 39}, {6, 40}, {7, 55}, {8, 0}, {8, 4}, {8, 44}, {9, 0}, {9, 4}, {9,
10}, {9, 40}, {9, 44}, {9, 56}, {10, 4}, {10, 8}, {10, 44}, {11, 0}, {11, 4}, {11, 40}, {11, 44}, {12, 44}}}},
    {"Erfurt Hbf", {"Erfurt Hbf", 950, 200, {{3, 55}, {5, 18}, {6, 29}, {6, 42}, {6, 48}, {7, 24}, {7, 28}, {7, 40}, {8, 9}, {8, 24},
{8, 29}, {8, 45}, {8, 48}, {9, 5}, {9, 9}, {9, 24}, {9, 26}, {9, 28}, {10, 24}, {10, 26}, {12, 26}}}},
    {"Berlin Hbf", {"Berlin Hbf", 1450, 200, {{0, 20}, {0, 22}, {3, 53}, {4, 0}, {4, 25}, {4, 30}, {4, 57}, {5, 4}, {5, 7}, {5, 26},
{5, 29}, {5, 30}, {5, 33}, {5, 34}, {5, 54}, {6, 0}, {6, 4}, {6, 7}, {6, 28}, {6, 30}, {6, 36}, {6, 38}, {6, 43}, {6, 46}, {6, 57}, {7,
0}, {7, 4}, {7, 6}, {7, 8}, {7, 16}, {7, 23}, {7, 26}, {7, 32}, {7, 34}, {7, 36}, {7, 38}, {7, 44}, {7, 46}, {7, 55}, {7, 58}, {9, 53},
{10, 22}, {10, 57}, {11, 30}, {11, 42}, {12, 22}, {12, 32}, {14, 32}}}},
    {"Ludwigsburg", {"Ludwigsburg", 700, 600, {{1, 53}, {2, 36}}}},
    {"Mannheim Hbf", {"Mannheim Hbf", 170, 600, {{3, 45}, {3, 55}, {4, 21}, {5, 59}, {6, 28}, {7, 4}, {7, 18}, {7, 28}, {7, 32},
{8, 2}, {8, 6}, {8, 23}, {8, 28}, {8, 37}, {9, 18}, {9, 27}, {9, 28}, {9, 53}, {9, 56}, {10, 27}, {10, 28}, {11, 27}, {11, 53}, {11,
56}, {12, 27}, {13, 27}, {13, 53}}}},
    {"Dortmund Hbf", {"Dortmund Hbf", 300, 200, {{5, 22}, {7, 21}, {7, 22}, {8, 8}, {8, 58}, {9, 10}, {9, 22}, {10, 11}, {10,
48}, {11, 10}, {11, 21}, {12, 5}, {12, 21}, {13, 21}}}},
    {"Koeln Hbf", {"Koeln Hbf", 120, 400, {{6, 03}, {6, 49}, {8, 4}, {9, 15}, {9, 40}, {10, 4}, {10, 9}, {10, 42}, {11, 15}, {12,
4}, {12, 9}, {12, 12}}}},
    {"Kiel Hbf", {"Kiel Hbf", 700, 50, {{11, 23}, {13, 48}, {17, 22}}}},
    {"Ulm Hbf", {"Ulm Hbf", 700, 650, {{1, 14}, {3, 52}, {4, 38}, {4, 45}, {6, 3}, {6, 17}, {6, 36}, {6, 45}, {7, 1}, {7, 8}, {7,
34}, {7, 44}, {7, 56}, {7, 57}, {8, 10}, {8, 45}, {8, 53}, {9, 0}, {10, 54}, {11, 58}, {13, 58}, {14, 3}}}},
    {"Augsburg Hbf", {"Augsburg Hbf", 850, 650, {{0, 29}, {3, 11}, {4, 2}, {5, 29}, {5, 33}, {5, 37}, {6, 2}, {6, 15}, {6, 53}, {6,
58}, {7, 11}, {7, 45}, {7, 57}, {7, 59}, {8, 13}, {8, 17}, {8, 46}, {8, 53}, {9, 40}, {10, 9}, {11, 40}, {12, 8}, {12, 40}, {14,
41}}}},
    {"Muenchen Hbf", {"Muenchen Hbf", 1000, 650, {{0, 1}, {0, 56}, {2, 38}, {3, 32}, {4, 13}, {4, 48}, {5, 0}, {5, 6}, {5, 14},
{5, 32}, {5, 34}, {5, 44}, {5, 53}, {6, 3}, {6, 14}, {6, 26}, {6, 27}, {6, 41}, {6, 47}, {6, 52}, {6, 57}, {7, 13}, {7, 18}, {7, 28},
{7, 46}, {7, 47}, {7, 54}, {7, 56}, {8, 30}, {8, 49}, {9, 10}, {9, 17}, {9, 18}, {9, 28}, {10, 2}, {10, 10}, {10, 42}, {11, 3}, {11,
11}, {12, 10}, {12, 42}, {13, 13}, {15, 13}}}},
    {"Vaihingen(Enz)", {"Vaihingen(Enz)", 300, 450, {{5, 17}, {5, 53}, {6, 23}, {6, 41}, {7, 53}, {8, 14}}}},
    {"Hamburg Hbf", {"Hamburg Hbf", 1050, 50, {{5, 37}, {7, 12}, {7, 25}, {8, 25}, {9, 12}, {9, 24}, {9, 54}, {10, 14}, {10,
36}, {10, 55}, {11, 36}, {11, 51}, {12, 14}, {12, 22}, {12, 36}, {13, 24}, {13, 36}, {13, 51}, {14, 14}, {14, 23}, {16, 14}}}},
    {"Hamburg-Altona", {"Hamburg-Altona", 800, 50, {{7, 26}, {7, 41}, {8, 41}, {9, 24}, {9, 39}, {12, 3}, {12, 30}, {12, 36},
{13, 41}, {14, 6}, {14, 30}, {14, 39}}}},
    {"Wiesbaden Hbf", {"Wiesbaden Hbf", 200, 450, {{8, 33}}}},
    {"Schorndorf", {"Schorndorf", 550, 650, {{6, 27}}}},
    {"Crailsheim", {"Crailsheim", 600, 700, {{7, 25}}}},
    {"Nuernberg Hbf", {"Nuernberg Hbf", 1000, 400, {{5, 22}, {5, 57}, {6, 29}, {6, 37}, {6, 55}, {6, 57}, {7, 25}, {7, 53}, {7,
58}, {8, 0}, {8, 2}, {8, 8}, {8, 18}, {8, 29}, {8, 52}, {8, 54}, {8, 56}, {8, 58}, {8, 59}, {9, 52}, {10, 0}, {10, 37}, {10, 52}}}},
    {"Boeblingen", {"Boeblingen", 500, 600, {{6, 37}, {7, 37}}}},
    {"Rottweil", {"Rottweil", 300, 650, {{7, 41}, {8, 41}}}},
    {"Zuerich HB", {"Zuerich HB", 400, 770, {{9, 26}, {16, 0}}}},
    {"Bruchsal", {"Bruchsal", 200, 650, {{6, 53}, {7, 38}}}},
```

{"Karlsruhe Hbf", {"Karlsruhe Hbf", 70, 600, {{7, 15}, {8, 53}, {8, 58}, {9, 57}, {11, 9}, {11, 57}, {13, 9}, {13, 57}}}},
{"Konstanz", {"Konstanz", 500, 700, {{10, 47}}}},
{"Karlsruhe-Durlach", {"Karlsruhe-Durlach", 300, 550, {{7, 50}}}},
{"Berlin Ostbahnhof", {"Berlin Ostbahnhof", 1300, 100, {{15, 17}}}},
{"Singen(Htwl)", {"Singen(Htwl)", 400, 650, {{9, 23}}}},
{"Weinheim(Bergstr)Hbf", {"Weinheim(Bergstr)Hbf", 400, 500, {{8, 21}}}},
{"Graz Hbf", {"Graz Hbf", 1550, 750, {{16, 14}}}},
{"Pforzheim Hbf", {"Pforzheim Hbf", 350, 600, {{8, 32}}}},
{"Leipzig Hbf", {"Leipzig Hbf", 1100, 285, {{2, 13}, {5, 42}, {6, 42}, {7, 42}, {8, 42}, {10, 10}, {13, 10}}}},
{"Duesseldorf Hbf", {"Duesseldorf Hbf", 150, 300, {{6, 25}, {6, 31}, {8, 32}, {9, 7}, {9, 15}, {9, 36}, {10, 2}, {10, 5}, {10, 40}, {11, 7}, {11, 10}, {11, 36}, {12, 37}, {11, 42}, {12, 7}}}},
{"Dresden Hbf", {"Dresden Hbf", 1410, 400, {{9, 6}, {18, 40}}}},
{"Lu. Wittenberg", {"Lu. Wittenberg", 1370, 170, {{1, 10}, {5, 10}, {6, 9}, {7, 10}, {8, 9}}}},
{"Bitterfeld", {"Bitterfeld", 1360, 210, {{1, 29}, {6, 32}, {8, 32}, {9, 22}, {11, 22}}}},
{"Halle(Saale)", {"Halle(Saale)", 1250, 300, {{1, 48}, {6, 12}, {6, 13}, {6, 16}, {6, 50}, {7, 10}, {7, 45}, {8, 13}, {8, 19}, {8, 31}, {8, 38}, {8, 50}, {9, 4}, {9, 45}, {11, 4}}}},
{"Weimar", {"Weimar", 1050, 300, {{3, 41}}}},
{"Gotha", {"Gotha", 850, 200, {{4, 14}}}},
{"Eisenach", {"Eisenach", 800, 200, {{4, 30}, {7, 54}, {9, 55}, {12, 0}}}},
{"Fulda", {"Fulda", 600, 300, {{5, 35}, {7, 2}, {7, 45}, {7, 46}, {8, 2}, {8, 48}, {9, 2}, {9, 45}, {10, 0}, {10, 2}, {10, 45}, {10, 48}, {11, 7}, {11, 45}}}},
{"Hanau", {"Hanau", 500, 400, {{6, 22}, {8, 25}, {10, 25}, {12, 25}}}},
{"Brandenburg", {"Brandenburg", 1250, 160, {{1, 6}, {8, 19}}}},
{"Magdeburg", {"Magdeburg", 1150, 160, {{1, 46}, {9, 1}}}},
{"Braunschweig", {"Braunschweig", 1050, 160, {{2, 34}, {5, 55}, {7, 55}, {8, 57}, {9, 50}}}},
{"Hannover", {"Hannover", 1050, 100, {{3, 9}, {6, 18}, {6, 37}, {7, 28}, {7, 53}, {8, 28}, {8, 32}, {8, 37}, {8, 53}, {9, 28}, {9, 32}, {10, 17}, {10, 25}, {11, 17}, {11, 33}, {12, 17}}}},
{"Minden", {"Minden", 980, 100, {{3, 59}, {6, 48}, {9, 0}}}},
{"Bielefeld", {"Bielefeld", 900, 150, {{4, 26}, {7, 19}, {7, 31}, {8, 20}, {9, 23}, {9, 31}, {9, 51}, {10, 20}}}},
{"Hamm(Westf)Hbf", {"Hamm(Westf)Hbf", 900, 250, {{5, 2}, {7, 48}, {8, 47}, {9, 50}, {10, 24}, {10, 47}}}},
{"Bochum", {"Bochum", 220, 200, {{5, 47}, {8, 24}, {9, 23}, {10, 24}, {11, 1}, {11, 23}}}},
{"Essen", {"Essen", 160, 200, {{5, 58}, {6, 58}, {8, 35}, {8, 58}, {9, 34}, {10, 3}, {10, 35}, {11, 7}, {11, 12}, {11, 34}, {11, 41}, {12, 3}, {13, 3}}}},
{"Duisburg", {"Duisburg", 150, 250, {{6, 11}, {6, 45}, {8, 46}, {8, 48}, {9, 48}, {9, 50}, {10, 48}, {10, 54}, {11, 27}, {11, 28}, {11, 48}, {11, 50}, {12, 50}}}},
{"Baden-Baden", {"Baden-Baden", 70, 650, {{9, 16}, {11, 26}, {12, 15}, {13, 26}}}},
{"Offenburg", {"Offenburg", 100, 680, {{9, 33}, {10, 31}, {12, 30}, {14, 30}}}},
{"Freiburg", {"Freiburg", 100, 710, {{10, 4}, {11, 1}, {12, 12}, {13, 2}, {14, 12}, {15, 1}}}},
{"Basel Bad Bf", {"Basel Bad Bf", 100, 740, {{10, 38}, {11, 36}, {12, 46}, {13, 10}, {13, 37}, {14, 46}, {15, 36}}}},
{"Basel SBB", {"Basel SBB", 100, 770, {{10, 47}, {11, 47}, {13, 47}, {14, 55}, {15, 47}}}},
{"Darmstadt", {"Darmstadt", 450, 430, {{8, 17}, {9, 7}, {9, 22}, {10, 7}, {11, 7}, {11, 21}, {12, 8}, {13, 7}}}},
{"Wolfsburg", {"Wolfsburg", 1120, 100, {{5, 8}, {5, 37}, {5, 46}, {6, 52}, {7, 8}, {7, 37}, {7, 52}, {8, 38}, {8, 52}, {9, 8}}}},
{"Goettingen", {"Goettingen", 1100, 200, {{6, 23}, {6, 50}, {7, 54}, {8, 23}, {8, 50}, {8, 54}, {9, 35}, {9, 50}, {10, 23}, {10, 37}, {10, 54}, {11, 35}}}},
{"Kassel-Wilhelmshoehe", {"Kassel-Wilhelmshoehe", 1100, 250, {{7, 12}, {7, 14}, {7, 34}, {8, 34}, {9, 12}, {9, 14}, {9, 34}, {10, 12}, {10, 34}, {10, 35}, {11, 14}}}},
{"Bern", {"Bern", 170, 770, {{12, 56}}}},
{"Interlaken Ost", {"Interlaken Ost", 250, 770, {{13, 58}}}},
{"Stendal Hbf", {"Stendal Hbf", 1200, 100, {{5, 14}, {6, 22}, {8, 11}}}},
{"Duesseldorf (Airport)", {"Duesseldorf (Airport)", 150, 350, {{8, 58}, {10, 58}, {11, 58}}}},
{"Hildesheim", {"Hildesheim", 1170, 200, {{6, 18}, {8, 18}, {9, 20}}}},
{"Coburg", {"Coburg", 800, 320, {{7, 3}}}},
{"Bamberg", {"Bamberg", 800, 370, {{7, 25}, {8, 16}, {9, 15}, {10, 15}}}},
{"Erlangen", {"Erlangen", 800, 420, {{7, 51}, {9, 36}}}},
{"Hagen", {"Hagen", 350, 250, {{8, 27}, {9, 21}, {10, 27}, {10, 59}, {11, 21}, {12, 59}}}},
{"Wuppertal", {"Wuppertal", 320, 300, {{8, 44}, {9, 38}, {10, 42}, {10, 44}, {11, 38}, {12, 42}}}},
{"Mainz", {"Mainz", 200, 410, {{9, 40}, {11, 40}}}},
{"Wittenberge", {"Wittenberge", 1350, 70, {{5, 58}, {6, 19}, {8, 0}}}},
{"Ludwigslust", {"Ludwigslust", 1250, 50, {{6, 17}, {6, 39}, {7, 40}, {8, 19}}}},
{"Hamburg Dammtor", {"Hamburg Dammtor", 900, 70, {{7, 31}, {8, 31}, {9, 30}}}},
{"Bonn", {"Bonn", 100, 500, {{10, 15}, {11, 43}}}},
{"Donauwoerth", {"Donauwoerth", 800, 500, {{6, 0}, {8, 6}, {9, 48}, {11, 48}}}},
{"Wuerzburg", {"Wuerzburg", 700, 450, {{6, 25}, {6, 52}, {7, 24}, {7, 53}, {8, 25}, {8, 52}, {9, 25}, {9, 52}, {10, 23}}}},
{"Biberach(Riss)", {"Biberach(Riss)", 700, 680, {{14, 33}}}},
{"Ravensburg", {"Ravensburg", 700, 710, {{15, 3}}}},
{"Friedrichshafen Stadt", {"Friedrichshafen Stadt", 700, 740, {{15, 16}}}},
{"Lindau-Reutin", {"Lindau-Reutin", 700, 770, {{15, 56}}}},
{"Bregenz", {"Bregenz", 770, 770, {{16, 4}}}},

{"Dornbirn", {"Dornbirn", 830, 770, {{16, 15}}}},
{"Feldkirch", {"Feldkirch", 900, 770, {{16, 34}}}},
{"Bludenz", {"Bludenz", 970, 770, {{16, 54}}}},
{"Langen a.A.", {"Langen a.A.", 1040, 770, {{17, 25}}}},
{"St. Anton", {"St. Anton", 1110, 770, {{10, 8}, {16, 7}, {17, 35}}}},
{"Landeck-Zams", {"Landeck-Zams", 1200, 770, {{15, 41}, {18, 2}}}},
{"Imst-Pitztal", {"Imst-Pitztal", 1280, 770, {{15, 6}, {18, 17}}}},
{"Oetztal", {"Oetztal", 1350, 770, {{14, 55}, {18, 30}}}},
{"Telfs-Pfaffenhofen", {"Telfs-Pfaffenhofen", 1435, 770, {{14, 42}, {18, 44}}}},
{"Innsbruck", {"Innsbruck", 1520, 770, {{8, 14}, {14, 15}, {19, 5}}}},
{"Buende", {"Buende", 900, 100, {{8, 41}}}},
{"Osnabrueck", {"Osnabrueck", 800, 100, {{8, 21}, {9, 4}, {10, 21}, {12, 21}, {14, 21}}}},
{"Rheine", {"Rheine", 700, 100, {{9, 33}}}},
{"Bad Bentheim", {"Bad Bentheim", 600, 100, {{9, 48}}}},
{"Hengelo", {"Hengelo", 500, 100, {{10, 7}}}},
{"Deventer", {"Deventer", 420, 100, {{10, 41}}}},
{"Amersfoort Centraal", {"Amersfoort Centraal", 320, 100, {{11, 24}}}},
{"Amsterdam C", {"Amsterdam C", 200, 100, {{12, 0}}}},
{"B Gesundbrunnen", {"B Gesundbrunnen", 1450, 150, {{6, 40}, {7, 42}, {7, 48}}}},
{"Oranienburg", {"Oranienburg", 1450, 120, {{7, 3}, {8, 3}}}},
{"Neustrelitz", {"Neustrelitz", 1450, 90, {{7, 40}, {8, 40}}}},
{"Waren", {"Waren", 1450, 60, {{7, 57}, {8, 57}}}},
{"Rostock", {"Rostock", 1450, 30, {{8, 37}, {9, 37}}}},
{"B Suedkreuz", {"B Suedkreuz", 1400, 250, {{6, 48}, {7, 39}, {9, 46}, {10, 15}, {10, 50}, {11, 22}, {11, 33}, {12, 15}, {14, 25}}}},
{"Muenchen Ost", {"Muenchen Ost", 1100, 650, {{1, 5}, {12, 31}}}},
{"Rosenheim", {"Rosenheim", 1200, 650, {{1, 39}, {7, 4}, {13, 4}}}},
{"Kufstein", {"Kufstein", 1200, 700, {{7, 26}, {13, 26}}}},
{"Dresden-Neustadt", {"Dresden-Neustadt", 1410, 350, {{9, 0}}}},
{"Bad Schandau", {"Bad Schandau", 1410, 450, {{9, 34}}}},
{"Decin", {"Decin", 1410, 500, {{9, 52}}}},
{"Praha-Holesovice", {"Praha-Holesovice", 1410, 550, {{11, 14}}}},
{"Praha hl.n.", {"Praha hl.n.", 1410, 600, {{11, 24}}}},
{"B-Wannsee", {"B-Wannsee", 1250, 210, {{7, 47}}}},
{"Potsdam Hbf", {"Potsdam Hbf", 1150, 230, {{7, 57}}}},
{"Helmstedt", {"Helmstedt", 940, 170, {{9, 27}}}},
{"Verden", {"Verden", 1050, 75, {{11, 28}}}},
{"Bremen", {"Bremen", 800, 75, {{9, 15}, {10, 45}, {11, 15}, {11, 50}, {13, 15}, {15, 15}}}},
{"Delmenhorst", {"Delmenhorst", 700, 75, {{12, 2}}}},
{"Oldenburg(Oldb)Hbf", {"Oldenburg(Oldb)Hbf", 550, 75, {{12, 23}}}},
{"Leer(Ostfriesl)", {"Leer(Ostfriesl)", 400, 75, {{13, 14}}}},
{"Emden", {"Emden", 400, 50, {{13, 38}}}},
{"Norddeich Mole", {"Norddeich Mole", 400, 25, {{14, 19}}}},
{"Warnemuende", {"Warnemuende", 1540, 30, {{10, 1}}}},
{"B-Spandau", {"B-Spandau", 1370, 140, {{7, 46}}}},
{"Eberswalde", {"Eberswalde", 1520, 310, {{8, 19}}}},
{"Angermuende", {"Angermuende", 1520, 280, {{8, 37}}}},
{"Prenzlau", {"Prenzlau", 1520, 250, {{9, 1}}}},
{"Pasewalk", {"Pasewalk", 1520, 230, {{9, 17}}}},
{"Anklam", {"Anklam", 1520, 200, {{9, 43}}}},
{"Zuessow", {"Zuessow", 1520, 170, {{9, 56}}}},
{"Greifswald", {"Greifswald", 1520, 140, {{10, 9}}}},
{"Stralsund", {"Stralsund", 1520, 110, {{10, 31}}}},
{"Bergen auf Ruegen", {"Bergen auf Ruegen", 1520, 80, {{11, 3}}}},
{"Ostseebad Binz", {"Ostseebad Binz", 1520, 50, {{11, 31}}}},
{"Aachen", {"Aachen", 50, 400, {{13, 11}}}},
{"Guenzburg", {"Guenzburg", 765, 650, {{1, 0}, {6, 46}, {8, 46}}}},
{"Siegburg/Bonn", {"Siegburg/Bonn", 300, 350, {{5, 45}, {9, 0}, {10, 0}, {10, 30}, {12, 0}}}},
{"Muenster(Westf)Hbf", {"Muenster(Westf)Hbf", 300, 150, {{7, 54}, {9, 54}, {11, 54}, {13, 54}}}},
{"Neumuenster", {"Neumuenster", 1050, 25, {{11, 4}, {13, 25}, {17, 3}}}},
{"Traunstein", {"Traunstein", 1280, 650, {{2, 18}}}},
{"Freilassing", {"Freilassing", 1360, 650, {{2, 38}}}},
{"Salzburg Hbf", {"Salzburg Hbf", 1450, 650, {{2, 47}}}},
{"Neumarkt a. W.", {"Neumarkt a. W.", 1500, 600, {{3, 28}}}},
{"Voecklabruck", {"Voecklabruck", 1525, 575, {{3, 53}}}},
{"Attnang-Puchheim", {"Attnang-Puchheim", 1525, 550, {{3, 58}}}},
{"Linz Hbf", {"Linz Hbf", 1525, 525, {{4, 30}}}},
{"St. Valentin", {"St. Valentin", 1525, 500, {{4, 44}}}},
{"Amstetten NOE", {"Amstetten NOE", 1525, 475, {{5, 0}}}},

```
{"Tullnerfeld", {"Tullnerfeld", 1525, 450, {{5, 43}}}},
{"Wien Hbf", {"Wien Hbf", 1525, 425, {{6, 5}}}},
{"Pasing", {"Pasing", 925, 650, {{2, 46}}}},
{"Ingolstadt Hbf", {"Ingolstadt Hbf", 1000, 550, {{4, 49}, {5, 26}, {5, 53}, {6, 52}, {7, 56}, {8, 24}}}},
{"Aschaffenburg Hbf", {"Aschaffenburg Hbf", 600, 400, {{7, 32}, {8, 32}, {9, 32}, {10, 32}}}},
{"Koeln Messe", {"Koeln Messe", 50, 375, {{9, 15}, {9, 36}, {10, 15}, {10, 48}, {11, 14}, {12, 15}}}},
{"Treuchtlingen", {"Treuchtlingen", 800, 450, {{6, 20}}}},
{"Solingen Hbf", {"Solingen Hbf", 200, 385, {{10, 28}, {12, 28}}}}
};

std::vector<Edge> edges = {
   {"Stuttgart Hbf", "Heidelberg Hbf", 25.50f},
   {"Heidelberg Hbf", "Frankfurt(M) Flugh", 35.75f},
   {"Frankfurt(M) Flugh", "Erfurt Hbf", 80.20f},
   {"Erfurt Hbf", "Berlin Hbf", 90.50f},
   {"Stuttgart Hbf", "Ludwigsburg", 5.20f},
   {"Ludwigsburg", "Mannheim Hbf", 10.75f},
   {"Mannheim Hbf", "Frankfurt(M) Flugh", 45.80f},
   {"Frankfurt(M) Flugh", "Dortmund Hbf", 80.35f},
   {"Heidelberg Hbf", "Mannheim Hbf", 10.25f},
   {"Frankfurt(M) Flugh", "Koeln Hbf", 75.45f},
   {"Koeln Hbf", "Dortmund Hbf", 45.60f},
   {"Dortmund Hbf", "Kiel Hbf", 90.90f},
   {"Stuttgart Hbf", "Ulm Hbf", 35.10f},
   {"Ulm Hbf", "Augsburg Hbf", 25.80f},
   {"Augsburg Hbf", "Muenchen Hbf", 40.25f},
   {"Stuttgart Hbf", "Vaihingen(Enz)", 10.40f},
   {"Vaihingen(Enz)", "Mannheim Hbf", 20.70f},
   {"Frankfurt(M) Flugh", "Hamburg Hbf", 110.15f},
   {"Vaihingen(Enz)", "Heidelberg Hbf", 15.60f},
   {"Heidelberg Hbf", "Frankfurt(M) Hbf", 40.95f},
   {"Stuttgart Hbf", "Mannheim Hbf", 15.80f},
   {"Dortmund Hbf", "Hamburg-Altona", 75.20f},
   {"Ulm Hbf", "Muenchen Hbf", 45.60f},
   {"Stuttgart Hbf", "Schorndorf", 7.90f},
   {"Schorndorf", "Crailsheim", 18.40f},
   {"Crailsheim", "Nuernberg Hbf", 40.25f},
   {"Stuttgart Hbf", "Boeblingen", 5.50f},
   {"Boeblingen", "Rottweil", 30.75f},
   {"Rottweil", "Zuerich HB", 55.90f},
   {"Stuttgart Hbf", "Bruchsal", 20.30f},
   {"Bruchsal", "Karlsruhe Hbf", 15.50f},
   {"Karlsruhe Hbf", "Konstanz", 60.80f},
   {"Mannheim Hbf", "Wiesbaden Hbf", 25.75f},
   {"Frankfurt(M) Flugh", "Duesseldorf Hbf", 70.90f},
   {"Mannheim Hbf", "Erfurt Hbf", 85.40f},
   {"Bruchsal", "Karlsruhe-Durlach", 12.20f},
   {"Mannheim Hbf", "Koeln Hbf", 55.60f},
   {"Koeln Hbf", "Berlin Ostbahnhof", 100.30f},
   {"Rottweil", "Singen(Htwl)", 35.20f},
   {"Stuttgart Hbf", "Weinheim(Bergstr)Hbf", 20.40f},
   {"Weinheim(Bergstr)Hbf", "Frankfurt(M) Hbf", 30.60f},
   {"Mannheim Hbf", "Dortmund Hbf", 57.80f},
   {"Dortmund Hbf", "Dresden Hbf", 92.40f},
   {"Muenchen Hbf", "Graz Hbf", 74.50f},
   {"Vaihingen(Enz)", "Pforzheim Hbf", 6.20f},
   {"Pforzheim Hbf", "Karlsruhe Hbf", 9.80f},
   {"Berlin Hbf", "Lu. Wittenberg", 31.20f},
   {"Lu. Wittenberg", "Bitterfeld", 12.50f},
   {"Bitterfeld", "Halle(Saale)", 7.80f},
   {"Halle(Saale)", "Leipzig Hbf", 9.40f},
   {"Leipzig Hbf", "Weimar", 17.60f},
   {"Weimar", "Erfurt Hbf", 5.80f},
   {"Erfurt Hbf", "Gotha", 9.20f},
   {"Gotha", "Eisenach", 11.30f},
   {"Eisenach", "Fulda", 27.60f},
   {"Fulda", "Hanau", 34.90f},
   {"Hanau", "Frankfurt(M) Hbf", 12.60f},
   {"Brandenburg", "Magdeburg", 37.80f},
```

```
{"Magdeburg", "Braunschweig", 34.90f},
{"Braunschweig", "Hannover", 25.60f},
{"Hannover", "Minden", 25.20f},
{"Minden", "Bielefeld", 27.80f},
{"Bielefeld", "Hamm(Westf)Hbf", 20.50f},
{"Hamm(Westf)Hbf", "Dortmund Hbf", 14.90f},
{"Dortmund Hbf", "Bochum", 6.80f},
{"Bochum", "Essen", 9.10f},
{"Essen", "Duisburg", 9.20f},
{"Duisburg", "Duesseldorf Hbf", 7.50f},
{"Duesseldorf Hbf", "Koeln Hbf", 8.70f},
{"Koeln Hbf", "Frankfurt(M) Flugh", 50.20f},
{"Frankfurt(M) Flugh", "Mannheim Hbf", 28.90f},
{"Mannheim Hbf", "Karlsruhe Hbf", 15.80f},
{"Karlsruhe Hbf", "Baden-Baden", 8.20f},
{"Baden-Baden", "Offenburg", 10.50f},
{"Offenburg", "Freiburg", 9.40f},
{"Freiburg", "Basel Bad Bf", 18.90f},
{"Basel Bad Bf", "Basel SBB", 4.20f},
{"Frankfurt(M) Flugh", "Darmstadt", 8.70f},
{"Darmstadt", "Heidelberg Hbf", 6.50f},
{"Heidelberg Hbf", "Stuttgart Hbf", 15.50f},
{"Berlin Hbf", "Hamburg Hbf", 69.80f},
{"Wolfsburg", "Goettingen", 22.40f},
{"Goettingen", "Kassel-Wilhelmshoehe", 18.70f},
{"Kassel-Wilhelmshoehe", "Fulda", 22.60f},
{"Frankfurt(M) Hbf", "Mannheim Hbf", 21.50f},
{"Karlsruhe Hbf", "Offenburg", 18.20f},
{"Basel SBB", "Bern", 29.40f},
{"Bern", "Interlaken Ost", 41.80f},
{"Frankfurt(M) Hbf", "Darmstadt", 5.70f},
{"Darmstadt", "Mannheim Hbf", 6.80f},
{"Mannheim Hbf", "Stuttgart Hbf", 13.50f},
{"Stendal Hbf", "Wolfsburg", 24.50f},
{"Wolfsburg", "Hannover", 18.20f},
{"Duisburg", "Duesseldorf (Airport)", 19.80f},
{"Duesseldorf (Airport)", "Duesseldorf Hbf", 6.90f},
{"Wolfsburg", "Braunschweig", 9.60f},
{"Braunschweig", "Hildesheim", 10.80f},
{"Hildesheim", "Goettingen", 12.30f},
{"Lu. Wittenberg", "Leipzig Hbf", 14.70f},
{"Leipzig Hbf", "Erfurt Hbf", 20.50f},
{"Erfurt Hbf", "Coburg", 27.80f},
{"Coburg", "Bamberg", 14.90f},
{"Bamberg", "Erlangen", 8.30f},
{"Erlangen", "Nuernberg Hbf", 12.50f},
{"Nuernberg Hbf", "Muenchen Hbf", 32.40f},
{"Halle(Saale)", "Erfurt Hbf", 29.10f},
{"Erfurt Hbf", "Nuernberg Hbf", 38.90f},
{"Berlin Hbf", "Brandenburg", 5.60f},
{"Berlin Hbf", "Wolfsburg", 31.40f},
{"Berlin Hbf", "Stendal Hbf", 24.70f},
{"Berlin Hbf", "Halle(Saale)", 24.90f},
{"Berlin Hbf", "Hannover", 29.90f},
{"Hannover", "Bielefeld", 12.70f},
{"Bielefeld", "Hagen", 13.90f},
{"Hagen", "Wuppertal", 10.20f},
{"Wuppertal", "Koeln Hbf", 16.70f},
{"Erfurt Hbf", "Frankfurt(M) Hbf", 29.90f},
{"Frankfurt(M) Hbf", "Frankfurt(M) Flugh", 5.80f},
{"Frankfurt(M) Hbf", "Mainz", 15.20f},
{"Mainz", "Mannheim Hbf", 14.30f},
{"Berlin Hbf", "Wittenberge", 19.60f},
{"Wittenberge", "Ludwigslust", 15.10f},
{"Ludwigslust", "Hamburg Hbf", 20.20f},
{"Hamburg Hbf", "Hamburg-Altona", 6.90f},
{"Erfurt Hbf", "Eisenach", 11.10f},
{"Fulda", "Frankfurt(M) Hbf", 17.70f},
{"Darmstadt", "Karlsruhe Hbf", 19.90f},
```

```
{"Baden-Baden", "Freiburg", 10.70f},
{"Hamm(Westf)Hbf", "Hagen", 14.20f},
{"Hamburg Hbf", "Hamburg Dammtor", 5.90f},
{"Berlin Hbf", "Koeln Hbf", 32.90f},
{"Koeln Hbf", "Bonn", 8.90f},
{"Erfurt Hbf", "Bamberg", 24.20f},
{"Bamberg", "Nuernberg Hbf", 16.30f},
{"Nuernberg Hbf","Donauwoerth", 10.10f},
{"Donauwoerth", "Augsburg Hbf", 11.30f},
{"Berlin Hbf", "Bitterfeld", 20.60f},
{"Halle(Saale)", "Wuerzburg", 32.40f},
{"Wuerzburg", "Stuttgart Hbf", 36.40f},
{"Ulm Hbf", "Biberach(Riss)", 10.40f},
{"Biberach(Riss)", "Ravensburg", 15.30f},
{"Ravensburg", "Friedrichshafen Stadt", 7.50f},
{"Friedrichshafen Stadt", "Lindau-Reutin", 8.70f},
{"Lindau-Reutin", "Bregenz", 3.20f},
{"Bregenz", "Dornbirn", 3.50f},
{"Dornbirn", "Feldkirch", 3.80f},
{"Feldkirch", "Bludenz", 3.20f},
{"Bludenz", "Langen a.A.", 3.70f},
{"Langen a.A.", "St. Anton", 4.50f},
{"St. Anton", "Landeck-Zams", 3.40f},
{"Landeck-Zams", "Imst-Pitztal", 5.80f},
{"Imst-Pitztal", "Oetztal", 4.20f},
{"Oetztal", "Telfs-Pfaffenhofen", 4.60f},
{"Telfs-Pfaffenhofen", "Innsbruck", 3.90f},
{"Hannover", "Buende", 14.30f},
{"Buende", "Osnabrueck", 11.40f},
{"Osnabrueck", "Rheine", 13.80f},
{"Rheine", "Bad Bentheim", 10.60f},
{"Bad Bentheim", "Hengelo", 5.20f},
{"Hengelo", "Deventer", 5.10f},
{"Deventer", "Amersfoort Centraal", 7.80f},
{"Amersfoort Centraal", "Amsterdam C", 7.50f},
{"Berlin Hbf", "B Gesundbrunnen", 3.60f},
{"B Gesundbrunnen", "Oranienburg", 3.90f},
{"Oranienburg", "Neustrelitz", 11.20f},
{"Neustrelitz", "Waren", 9.20f},
{"Waren", "Rostock", 9.50f},
{"Berlin Hbf", "Ludwigslust", 19.50f},
{"Hamburg Dammtor", "Hamburg-Altona", 3.40f},
{"Berlin Hbf", "B Suedkreuz", 3.40f},
{"B Suedkreuz", "Halle(Saale)", 22.60f},
{"Nuernberg Hbf", "Muenchen Ost", 30.20f},
{"Muenchen Ost", "Rosenheim", 18.50f},
{"Rosenheim", "Kufstein", 9.90f},
{"Kufstein", "Innsbruck", 10.70f},
{"Innsbruck", "Telfs-Pfaffenhofen", 3.30f},
{"Telfs-Pfaffenhofen", "Oetztal", 4.20f},
{"Oetztal", "Imst-Pitztal", 4.40f},
{"Imst-Pitztal", "Landeck-Zams", 4.60f},
{"Landeck-Zams", "St. Anton", 3.40f},
{"Berlin Hbf", "Dresden-Neustadt", 31.80f},
{"Dresden-Neustadt", "Dresden Hbf", 2.80f},
{"Dresden Hbf", "Bad Schandau", 3.80f},
{"Bad Schandau", "Decin", 5.30f},
{"Decin", "Praha-Holesovice", 6.40f},
{"Praha-Holesovice", "Praha hl.n.", 3.20f},
{"Basel SBB", "Zuerich HB", 25.50f},
{"Berlin Hbf", "B-Wannsee", 3.40f},
{"B-Wannsee", "Potsdam Hbf", 4.30f},
{"Potsdam Hbf", "Brandenburg", 4.40f},
{"Magdeburg", "Helmstedt", 9.80f},
{"Helmstedt", "Braunschweig", 13.60f},
{"Hannover", "Verden", 9.60f},
{"Verden", "Bremen", 12.20f},
{"Bremen", "Delmenhorst", 4.40f},
{"Delmenhorst", "Oldenburg(Oldb)Hbf", 6.90f},
```

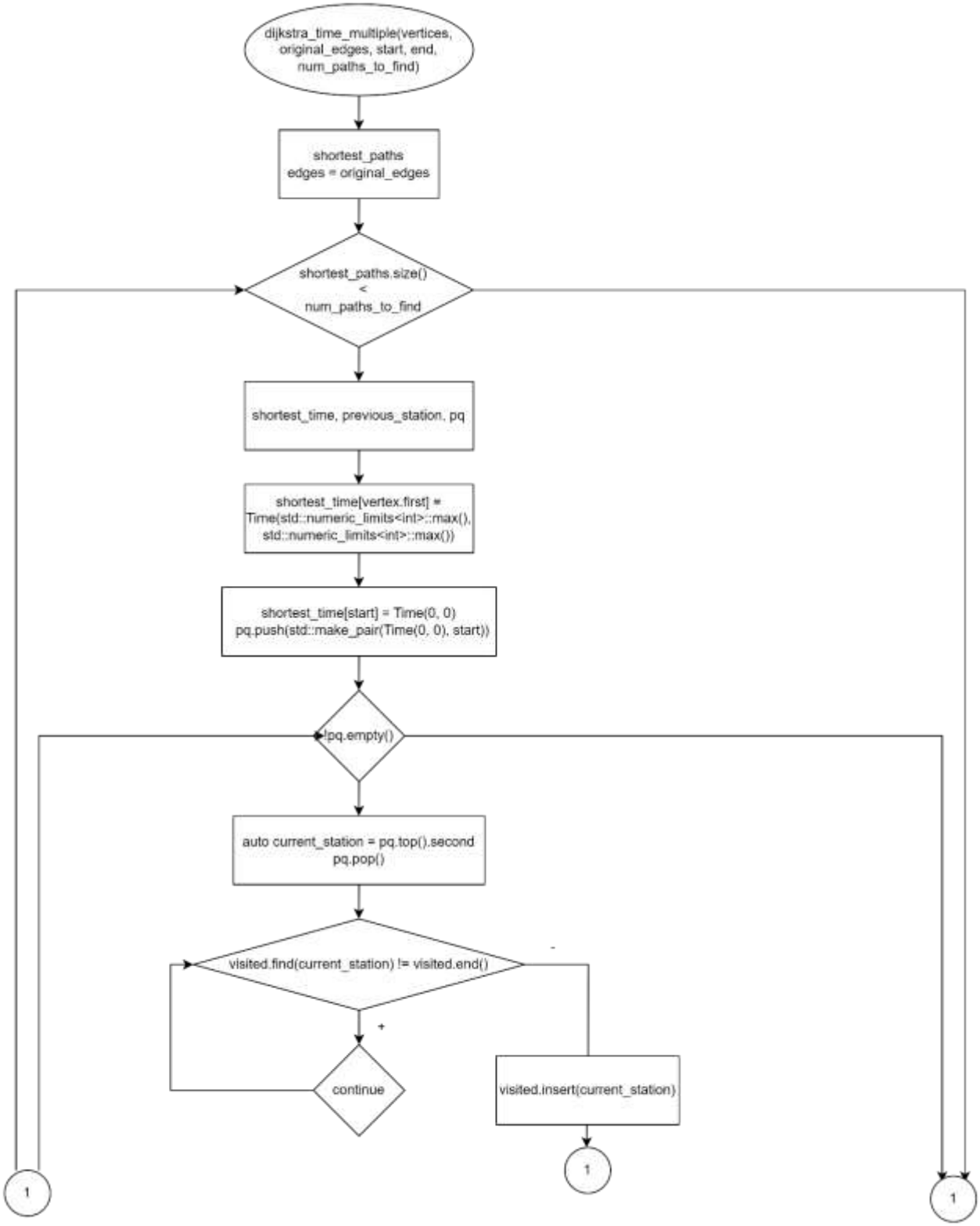
```
{"Oldenburg(Oldb)Hbf", "Leer(Ostfriesl)", 11.40f},
{"Leer(Ostfriesl)", "Emden", 10.50f},
{"Emden", "Norddeich Mole", 7.90f},
{"B Suedkreuz", "Bitterfeld", 17.90f},
{"Rostock", "Warnemuende", 4.30f},
{"Berlin Hbf", "B-Spandau", 3.40f},
{"B-Spandau", "Hamburg Hbf", 27.90f},
{"B Gesundbrunnen", "Eberswalde", 3.10f},
{"Eberswalde", "Angermuende", 3.50f},
{"Angermuende", "Prenzlau", 3.80f},
{"Prenzlau", "Pasewalk", 3.20f},
{"Pasewalk", "Anklam", 3.70f},
{"Anklam", "Zuessow", 4.50f},
{"Zuessow", "Greifswald", 3.40f},
{"Greifswald", "Stralsund", 3.90f},
{"Stralsund", "Bergen auf Ruegen", 4.20f},
{"Bergen auf Ruegen", "Ostseebad Binz", 5.80f},
{"Koeln Hbf", "Aachen", 15.50f},
{"Muenchen Hbf", "Augsburg Hbf", 20.80f},
{"Augsburg Hbf", "Guenzburg", 7.90f},
{"Guenzburg", "Ulm Hbf", 12.20f},
{"Ulm Hbf", "Stuttgart Hbf", 18.30f},
{"Mannheim Hbf", "Frankfurt(M) Hbf", 12.50f},
{"Frankfurt(M) Hbf", "Siegburg/Bonn", 18.60f},
{"Siegburg/Bonn", "Koeln Hbf", 5.70f},
{"Koeln Hbf", "Duesseldorf Hbf", 10.80f},
{"Duesseldorf Hbf", "Duisburg", 7.90f},
{"Duisburg", "Essen", 4.40f},
{"Essen", "Dortmund Hbf", 6.70f},
{"Dortmund Hbf", "Muenster(Westf)Hbf", 13.20f},
{"Muenster(Westf)Hbf", "Osnabrueck", 12.50f},
{"Osnabrueck", "Bremen", 18.20f},
{"Bremen", "Hamburg Hbf", 27.80f},
{"Hamburg Hbf", "Neumuenster", 20.90f},
{"Neumuenster", "Kiel Hbf", 12.40f},
{"Muenchen Hbf", "Muenchen Ost", 3.80f},
{"Rosenheim", "Traunstein", 10.30f},
{"Traunstein", "Freilassing", 7.40f},
{"Freilassing", "Salzburg Hbf", 7.80f},
{"Salzburg Hbf", "Neumarkt a. W.", 12.50f},
{"Neumarkt a. W.", "Voecklabruck", 6.60f},
{"Voecklabruck", "Attnang-Puchheim", 4.70f},
{"Attnang-Puchheim", "Linz Hbf", 6.90f},
{"Linz Hbf", "St. Valentin", 5.20f},
{"St. Valentin", "Amstetten NOE", 7.30f},
{"Amstetten NOE", "Tullnerfeld", 10.40f},
{"Tullnerfeld", "Wien Hbf", 15.60f},
{"Muenchen Hbf", "Pasing", 4.60f},
{"Pasing", "Augsburg Hbf", 4.30f},
{"Augsburg Hbf", "Ulm Hbf", 15.80f},
{"Muenchen Hbf", "Ingolstadt Hbf", 12.90f},
{"Ingolstadt Hbf", "Nuernberg Hbf", 21.30f},
{"Nuernberg Hbf", "Wuerzburg", 18.20f},
{"Wuerzburg", "Fulda", 21.70f},
{"Fulda", "Kassel-Wilhelmshoehe", 21.40f},
{"Kassel-Wilhelmshoehe", "Goettingen", 20.50f},
{"Goettingen", "Hannover", 22.30f},
{"Hannover", "Hamburg Hbf", 27.80f},
{"Wuerzburg", "Aschaffenburg Hbf", 15.90f},
{"Aschaffenburg Hbf", "Frankfurt(M) Hbf", 16.80f},
{"Siegburg/Bonn", "Koeln Messe", 3.40f},
{"Koeln Messe", "Duesseldorf Hbf", 13.80f},
{"Stuttgart Hbf", "Karlsruhe Hbf", 11.20f},
{"Stuttgart Hbf", "Karlsruhe-Durlach", 11.80f},
{"Augsburg Hbf", "Donauwoerth", 12.30f},
{"Donauwoerth", "Treuchtlingen", 11.50f},
{"Treuchtlingen", "Nuernberg Hbf", 16.40f},
{"Nuernberg Hbf", "Erfurt Hbf", 32.10f},
{"Erfurt Hbf", "Halle(Saale)", 21.70f},
```

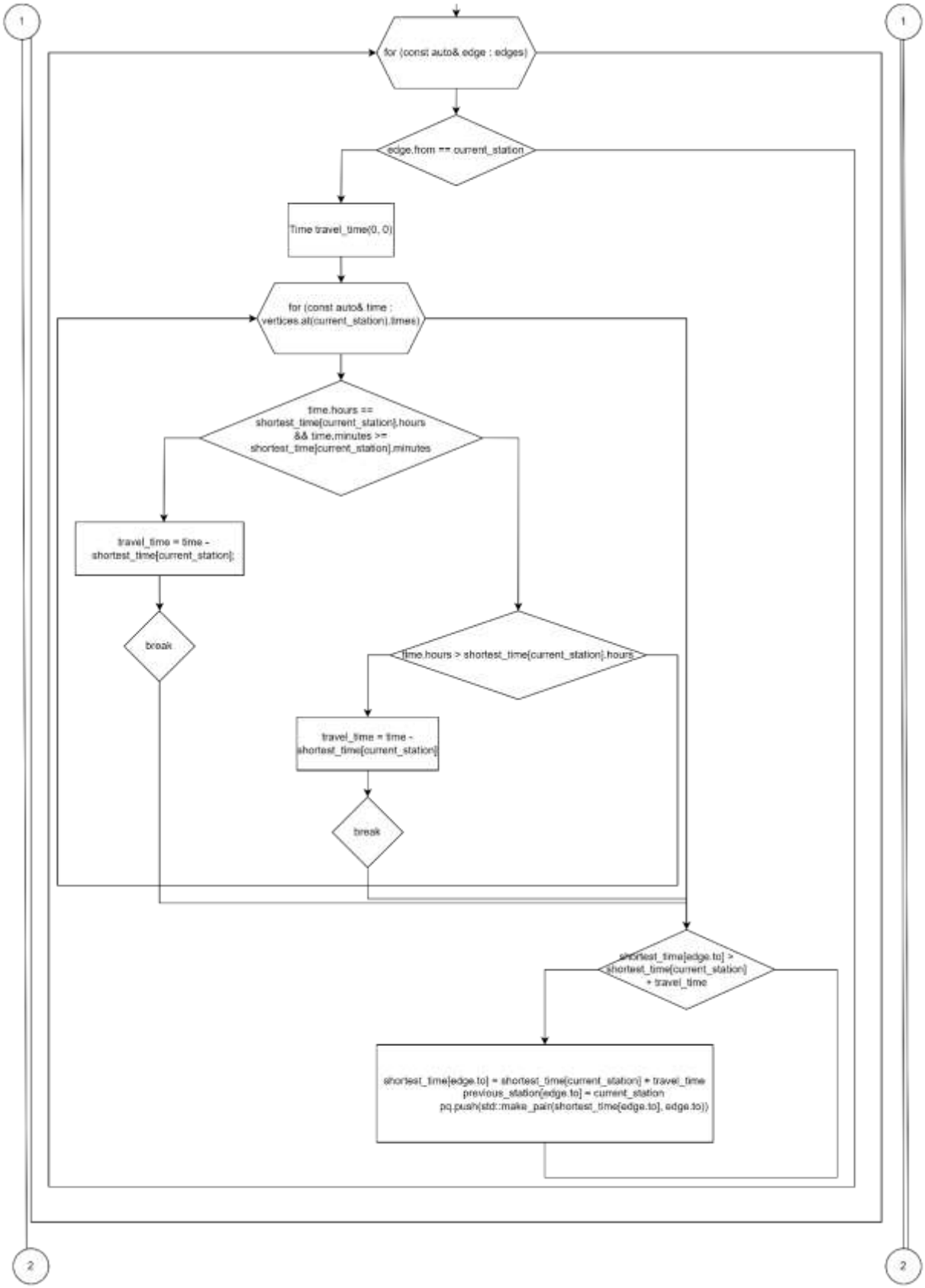
```
        {"Halle(Saale)", "Bitterfeld", 6.30f},
        {"Bitterfeld", "B Suedkreuz", 25.20f},
        {"B Suedkreuz", "Berlin Hbf", 3.90f},
        {"Hamburg-Altona", "Neumuenster", 16.80f},
        {"Hannover", "Bremen", 22.70f},
        {"Koeln Hbf", "Solingen Hbf", 10.90f},
        {"Solingen Hbf", "Wuppertal", 3.70f},
        {"Wuppertal", "Hagen", 8.60f},
        {"Hagen", "Dortmund Hbf", 7.80f},
        {"Muenchen Hbf", "Nuernberg Hbf", 27.40f},
        {"Nuernberg Hbf", "Frankfurt(M) Flugh", 40.50f},
        {"Frankfurt(M) Flugh", "Koeln Messe", 35.20f},
        {"Stuttgart Hbf", "Darmstadt", 14.90f},
        {"Darmstadt", "Frankfurt(M) Hbf", 11.30f},
        {"Halle(Saale)", "B Suedkreuz", 25.20f}
    };

    sf::Font font;
    if (!font.loadFromFile("D:/Desktop/Универ/Диплом/SFML-2.6.1-windows-vc16-32-bit/SFML-2.6.1/Arial.ttf")) {
        return EXIT_FAILURE;
    }

    sf::Texture backgroundTexture;
    if (!backgroundTexture.loadFromFile("C:/photo_2024-02-21_17-54-06.jpg")) {
    }

    sf::Sprite backgroundSprite(backgroundTexture);
    backgroundSprite.setScale(window.getSize().x / backgroundSprite.getLocalBounds().width, window.getSize().y /
backgroundSprite.getLocalBounds().height);

    bool shortestPathPrintedDijkstra = false;
    bool shortestPathPrintedAStar = false;
    std::cout << "Enter the start station: ";
    std::getline(std::cin, start_station);

    std::cout << "Enter the end station: ";
    std::getline(std::cin, end_station);

    std::cout << std::endl;

    float lineThickness = 1.0f;

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(backgroundSprite);

        for (const auto& edge : edges) {
            sf::Vector2f from(vertexCoordinates.at(edge.from).x, vertexCoordinates.at(edge.from).y);
            sf::Vector2f to(vertexCoordinates.at(edge.to).x, vertexCoordinates.at(edge.to).y);
            sf::Vector2f direction = to - from;
            float length = sqrt(direction.x * direction.x + direction.y * direction.y);
            sf::Vector2f unitDirection = direction / length;
            sf::Vector2f unitPerpendicular(-unitDirection.y, unitDirection.x);

            sf::RectangleShape line(sf::Vector2f(length, lineThickness));
            line.setPosition(from);
            line.setFillColor(sf::Color::White);
            line.setOrigin(0, lineThickness / 2.0f);
            line.setRotation(atan2(direction.y, direction.x) * 180 / 3.14159265);

            window.draw(line);
        }
```

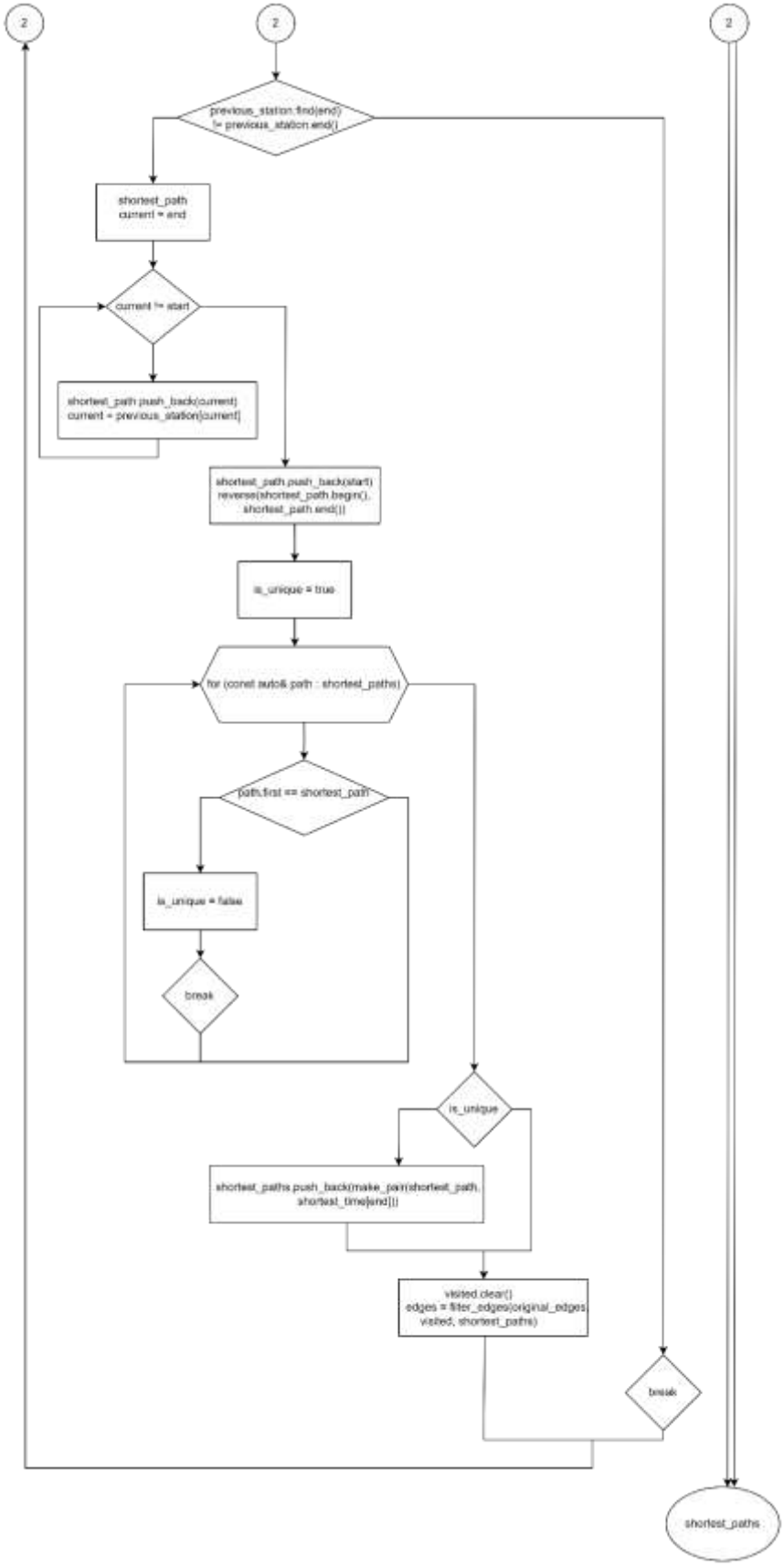
```
        int num_paths_to_find = 3;

        std::vector<std::pair<std::vector<std::string>, Time>> shortest_paths = dijkstra_time_multiple(vertexCoordinates, edges,
start_station, end_station, num_paths_to_find);
        std::vector<Edge> edges_copy = edges;

        std::vector<std::pair<std::vector<std::string>, Time>> a_star_shortest_paths = a_star_time_multiple(vertexCoordinates,
edges_copy, start_station, end_station, num_paths_to_find);

        if (!shortestPathPrintedDijkstra) {
            std::cout << "Dijkstra's Algorithm:" << std::endl;
            for (int i = 0; i < shortest_paths.size(); ++i) {
                std::cout << "Path " << i + 1 << ": ";
                const auto& path = shortest_paths[i].first;
                float total_cost = 0.0f;
                Time total_duration = { 0, 0 };
                for (int j = 0; j < path.size(); ++j) {
                    std::cout << path[j];
                    if (j < path.size() - 1) {
                        std::cout << " -> ";
                        for (const auto& edge : edges) {
                            if (edge.from == path[j] && edge.to == path[j + 1]) {
                                total_cost += edge.cost;
                                break;
                            }
                        }
                    }

                }
                std::cout << std::endl;


                int num_transfers = count_transfers(path, vertexCoordinates, edges);
                std::cout << "Transfers: " << num_transfers << std::endl;

                std::cout << "Time: " << shortest_paths[i].second.hours << " hours and " << shortest_paths[i].second.minutes << "
minutes" << std::endl;
                std::cout << "Cost: " << total_cost << " euro" << std::endl;
                std::cout << std::endl;
            }
            shortestPathPrintedDijkstra = true;
        }


        if (!shortestPathPrintedAStar) {
            std::cout << std::endl;
            std::cout << std::endl;
            std::cout << "A* Algorithm:" << std::endl;
            for (int i = 0; i < a_star_shortest_paths.size(); ++i) {
                std::cout << "Path " << i + 1 << ": ";
                const auto& path = a_star_shortest_paths[i].first;
                float total_cost = 0.0f;
                Time total_duration = { 0, 0 };
                for (int j = 0; j < path.size() - 1; ++j) {
                    std::cout << path[j] << " -> ";
                    const auto& current_station = path[j];
                    const auto& next_station = path[j + 1];
                    for (const auto& edge : edges) {
                        if (edge.from == current_station && edge.to == next_station) {
                            total_cost += edge.cost;
                            total_duration = total_duration + Time(0, static_cast<int>(edge.cost));

                            const auto& current_vertex = vertexCoordinates.at(current_station);
                            const auto& next_vertex = vertexCoordinates.at(next_station);
                            for (const auto& time : current_vertex.times) {
                                if (time > total_duration) {
                                    total_duration = time;
                                    break;
                                }
```

```
                    }
                    break;
                }
            }
        }

        std::cout << path.back() << std::endl;
        int num_transfers = count_transfers(path, vertexCoordinates, edges);
        std::cout << "Transfers: " << num_transfers << std::endl;
        std::cout << "Time: " << total_duration.hours << " hours and " << total_duration.minutes << " minutes" << std::endl;
        std::cout << "Cost: " << total_cost << " euro" << std::endl;
        std::cout << std::endl;
    }
    shortestPathPrintedAStar = true;
}


for (const auto& vertex : vertexCoordinates) {
    sf::CircleShape circle(10);
    circle.setFillColor(sf::Color(0, 0, 128));
    circle.setPosition(vertex.second.x - 10, vertex.second.y - 10);
    window.draw(circle);

    sf::Text text(vertex.second.name, font, 12);
    text.setFillColor(sf::Color(255, 255, 255));
    text.setStyle(sf::Text::Bold);
    float textX = vertex.second.x - text.getLocalBounds().width / 2;
    float textY = vertex.second.y - text.getLocalBounds().height / 2;
    text.setPosition(textX, textY);
    window.draw(text);
}

if (!shortest_paths.empty()) {
    const auto& first_path = shortest_paths.front();

    for (size_t i = 0; i < first_path.first.size() - 1; ++i) {
        auto from = vertexCoordinates[first_path.first[i]];
        auto to = vertexCoordinates[first_path.first[i + 1]];

        sf::Vertex line[] = {
            sf::Vertex(sf::Vector2f(from.x, from.y), sf::Color::Red),
            sf::Vertex(sf::Vector2f(to.x, to.y), sf::Color::Red)
        };

        window.draw(line, 2, sf::Lines);
    }
}

if (!a_star_shortest_paths.empty()) {
    const auto& first_path_a_star = a_star_shortest_paths.front();

    for (size_t i = 0; i < first_path_a_star.first.size() - 1; ++i) {
        auto from = vertexCoordinates[first_path_a_star.first[i]];
        auto to = vertexCoordinates[first_path_a_star.first[i + 1]];

        sf::Vertex line[] = {
            sf::Vertex(sf::Vector2f(from.x, from.y), sf::Color::Green),
            sf::Vertex(sf::Vector2f(to.x, to.y), sf::Color::Green)
        };

        window.draw(line, 2, sf::Lines);
    }
}

window.display();
sf::sleep(sf::milliseconds(1000));
  }
  return 0;
}
```

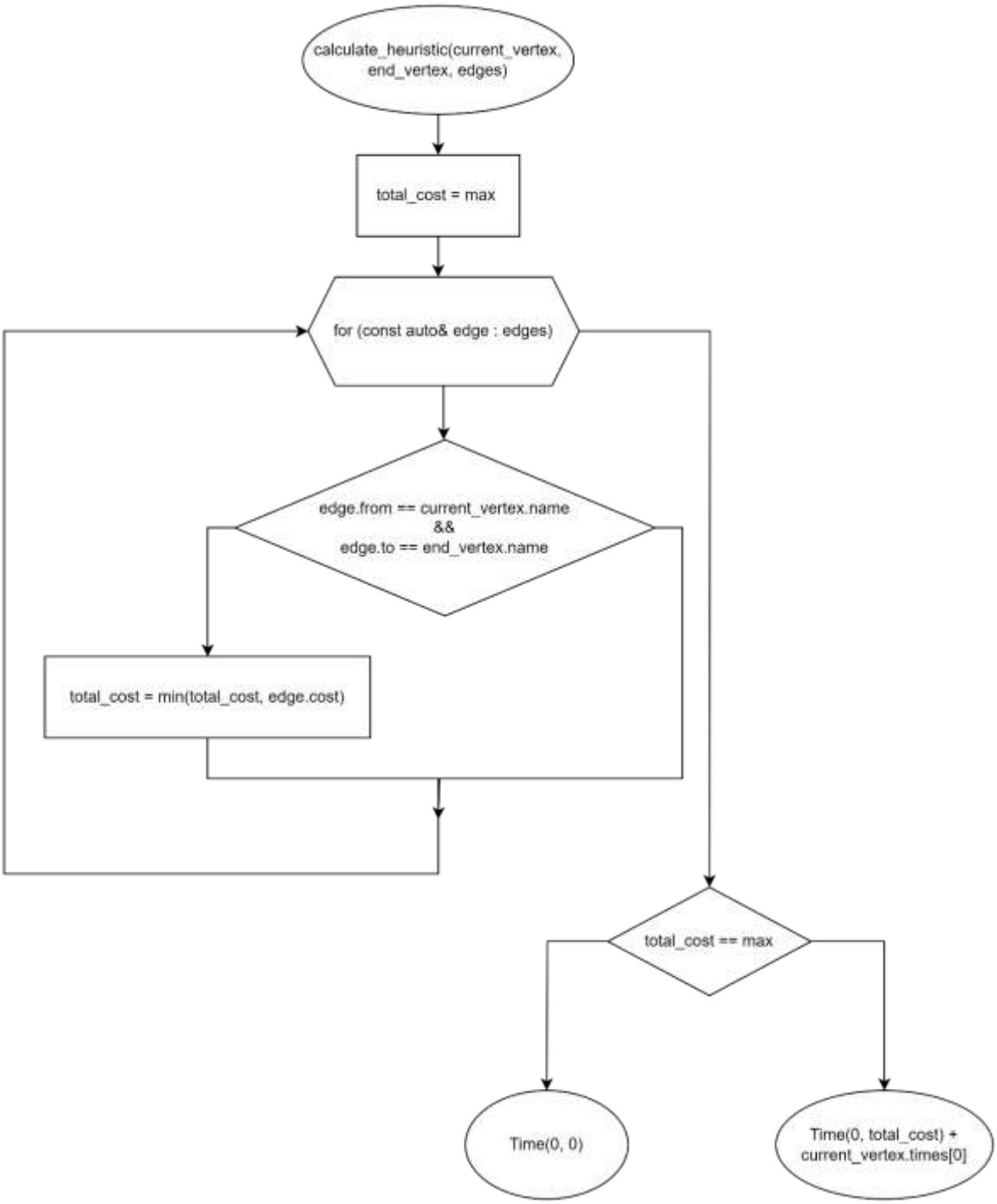
ДОДАТОК Е. БЛОК-СХЕМИ АЛГОРИТМІВ




1
for (const auto& edge : edges)
edge.from == current_station
Time travel_time(0, 0);
for (const auto& time : vertices.at(current_station).times)
time.hours == shortest_time[current_station].hours && time.minutes >= shortest_time[current_station].minutes
travel_time = time - shortest_time[current_station];
break
time.hours > shortest_time[current_station].hours
travel_time = time - shortest_time[current_station];
break
shortest_time[edge.to] > shortest_time[current_station] + travel_time
shortest_time[edge.to] = shortest_time[current_station] + travel_time
previous_station[edge.to] = current_station
pq.push(std::make_pair(shortest_time[edge.to], edge.to))
1
2
2



«Блок-схема Е.1 – Блок-схема алгоритму Дейкстри»



«Блок-схема Е.2 – Блок-схема алгоритму пошуку евристичної оцінки»