Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Системотехніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Порівняльний аналіз алгоритмів оптимізації маршрутів для розробки системи планування подорожей

Comparative analysis of route optimization algorithms for the development of a travel planning system

(тема)

Виконав: студент 2 курсу, групи ІТПм-24-1
Клочко Є. С.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Інформаційні технології проєктування
(повна назва освітньої програми)

Керівник доцент Ситнікова П. Е.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри системотехніки _____ Гребеннік І.В.
(підпис) (прізвище, ініціали)

2025 р.

Я як студентка ХНУРЕ розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

08.12.2025 *Клочко Є. С.*

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування

Керівник кваліфікаційної роботи _____

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні

Керівник кваліфікаційної роботи _____

Попередній захист проведено 08.12.2025

Керівник кваліфікаційної роботи _____

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Системотехніки

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Інформаційні технології проєктування
(код і повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Клочко Єлизаветі Станіславівні
(прізвище, ім’я, по батькові)

1. Тема роботи порівняльний аналіз алгоритмів оптимізації маршрутів для розробки системи планування подорожей
comparative analysis of route optimization algorithms for the development of a travel planning system
затверджена наказом по університету від 24 листопада 2025 р. № 1058Ст
2. Термін подання студентом роботи до екзаменаційної комісії 16 грудня 2025 р
3. Вихідні дані до роботи: оформлення пояснювальної записки та захист кваліфікаційної роботи виконується англійською мовою; існуючі алгоритми знаходження оптимальних маршрутів; способи побудови маршрутів на мапі; оптимізаційні алгоритми; порівняння різних алгоритмів оптимізації;
4. Перелік питань, що потрібно опрацювати в роботі сформулювати та оформити вимоги до інформаційної системи; провести аналіз алгоритмів та порівняння їх у контексті розроблюваної системи; розробити експериментальний прототип інтелектуальної інформаційної системи; провести експериментальне дослідження ефективності використання алгоритмів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) мапа, граф, інтерфейс застосунку, блок схеми алгоритму, приклади маршрутів, графіки роботи алгоритму на кожній мапі, графіки роботи ефективності алгоритмів

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
ANALYSIS OF THE SUBJECT AREA	Ситнікова П.Е.		9.12.25
RESEARCH OF METHODS OF SOLVING THE PROBLEM	Ситнікова П.Е		9.12.25
IMPLEMENTATION OF A PROTOTYPE OF AN INTELLECTUAL SYSTEM	Ситнікова П.Е		9.12.25
EXPERIMENTAL RESEARCH AND ANALYSIS OF THE EFFICIENCY OF ALGORITHMS	Ситнікова П.Е		9.12.25

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів	Примітка
1	Отримання завдання для кваліфікаційної роботи	24.11.2025	Виконано
2	Аналіз предметної області	25.11.2025	Виконано
3	Опис вимоги до оптимальних маршрутів у транспортних мережах	26.11.2025	Виконано
4	Аналіз алгоритмів оптимізації	27.11.2025	Виконано
5	Побудова та обробка графової моделі подорожей	30.11.2025	Виконано
6	Вибір засобів розробки системи	01.12.2025	Виконано
7	Реалізація функціональних компонентів веб-сервісу	03.12.2025	Виконано
8	Дослідження ефективності алгоритмів	05.12.2025	Виконано
9	Оформлення пояснювальної списики	10.12.2025	Виконано
10	Подання матеріалів до захисту	15.12.2025	
11	Захист курсового проєкту	16.12.2025	

Дата видачі завдання 24.11 2025 р.

Студент _____ Клочко Є. С.
(підпис)

Керівник роботи _____ доцент Ситнікова П. Е.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра: 92 сторінок, 5 таблиць, 13 рисунки, 2 додатки, 27 джерел інформації.

ДОСЛІДЖЕННЯ, СТАТТІ, ПУБЛІКАЦІЯ, АНАЛІЗ, ІНТЕЛЕКТУАЛЬНА ІНФОРМАЦІЙНА СИСТЕМА, АЛГОРИТМИ, ПОРІВНЯННЯ, ОПТИМІЗАЦІЯ

Об'єктом досліджень є процес планування та оптимізації маршрутів подорожей у транспортних мережах.

Предметом дослідження є алгоритми оптимізації маршрутів та методи оцінювання їх ефективності при вирішенні задачі планування подорожей. Основною задачею дослідження є визначення найбільш ефективних алгоритмів оптимізації для формування оптимальних маршрутів за різними критеріями, такими як мінімізація часу, вартості, кількості пересадок або комбінованих показників. У рамках роботи передбачається аналіз класичних детермінованих методів, евристичних алгоритмів та підходів багатокритеріальної оптимізації для оцінювання їх придатності до застосування в реальних сценаріях планування подорожей.

Метою є проведення комплексного порівняльного аналізу алгоритмів оптимізації маршрутів та розробка експериментального прототипу інформаційної системи планування подорожей, здатної формувати оптимальні маршрути на основі вибраних алгоритмів і результатів їхнього порівняння.

Методи дослідження – теорія графів і методи пошуку найкоротших шляхів, евристичні та метаевристичні підходи, методи багатокритеріальної оптимізації, математичне моделювання транспортних мереж, алгоритмічний аналіз, порівняльне експериментальне моделювання та статистичні методи оцінювання точності й ефективності.

Галузь застосування розробки – інформаційні системи у сфері туризму та подорожей, зокрема сервіси маршрутизації, онлайн-платформи бронювання транспортних квитків і готелів, системи планування комплексних маршрутів у багатокомпонентних транспортних мережах.

ABSTRACT

Explanatory note to the master's qualification thesis: 92 pages, 5 tables, 13 figures, 2 appendices, 27 sources of information.

RESEARCH, ARTICLES, PUBLICATION, ANALYSIS, INTELLIGENT INFORMATION SYSTEM, ALGORITHMS, COMPARISON, OPTIMIZATION

The object of research is the process of planning and optimizing travel routes in transport networks.

The subject of research is route optimization algorithms and methods for assessing their effectiveness in solving the travel planning problem. The main task of the research is to determine the most effective optimization algorithms for forming optimal routes according to various criteria, such as minimizing time, cost, number of transfers or combined indicators. The work involves the analysis of classical deterministic methods, heuristic algorithms and multi-criteria optimization approaches to assess their suitability for use in real travel planning scenarios.

The goal is to conduct a comprehensive comparative analysis of route optimization algorithms and develop an experimental prototype of a travel planning information system capable of generating optimal routes based on the selected algorithms and the results of their comparison.

Research methods - graph theory and methods for finding the shortest paths, heuristic and metaheuristic approaches, multi-criteria optimization methods, mathematical modeling of transport networks, algorithmic analysis, comparative experimental modeling and statistical methods for assessing accuracy and efficiency.

The field of application of the development is information systems in the field of tourism and travel, in particular routing services, online platforms for booking transport tickets and hotels, complex route planning systems in multi-component transport networks.

CONTENT

LIST OF ABBREVIATIONS, CONVENTIONS, SYMBOLS, UNITS AND TERMS

IS – information system

DB – database

A* – name of the algorithm being used

TD – time-dependent

FIFO – first in first out

RAPTOR – routing algorithm for public transit routing

GPS – global positioning system

API – application programming interface

INTRODUCTION

The current stage of development of digital technologies is accompanied by a rapid increase in the number of online services that automate travel planning processes. Users are offered a wide range of tools for finding flights, booking hotels, renting transport and building complex combined routes. However, most of the existing systems are focused only on individual elements of the trip and do not provide a comprehensive approach to the formation of the optimal route, considering various factors: cost, time, number of transfers, available modes of transport or geographical features.

One of the central challenges in the development of such systems is the task of optimizing routes, which belongs to the class of complex computing problems[1]. To solve it, graph theory algorithms, heuristic and metaheuristic methods are used, which differ in accuracy, speed of operation and scalability. The need to study these algorithms is due to the need not only to implement the correct construction of the route, but also to ensure the possibility of comparing different methods and choosing the most effective one for specific conditions.

The creation of an intelligent travel planning information system makes it possible to integrate analytical models, algorithmic methods and modern software technologies into a single service. Such a system not only automates the process of route formation but also provides the user with well-founded recommendations that are formed based on a mathematical model and data. The use of optimization algorithms allows you to consider different criteria, combine several types of transport, increase the accuracy of results and improve the overall experience of interaction with the system[2].

The scientific part consists in the comparative analysis of route optimization algorithms and the development of an analytical model for their evaluation, and the practical significance lies in the creation of a prototype of an intelligent information system capable of applying these algorithms for real travel planning scenarios. The results of the work can be used in tourist services, logistics systems and any applications related to the construction of optimal routes in transport networks.

1 ANALYSIS OF THE SUBJECT AREA AND DESCRIPTION OF THE RESEARCH OBJECT

1.1 Analysis of modern travel planning systems

Modern travel planning information systems occupy an important place in the digital services market and play a key role in ensuring the convenience of organizing user movements. Such systems combine the functionality of searching for vehicles, comparing costs, estimating route duration, viewing alternative options and integrating with mapping services. The development of artificial intelligence, big data and geographic information systems technologies has significantly expanded the capabilities of these platforms, allowing them to form complex routes and analyze many possible combinations.

The most common products in the field of travel planning include services such as Google Maps[3], Rome2Rio[4], Omio[5], Skyscanner[6], Kayak[7] and others. They provide a search for routes between destinations considering various modes of transport, including air travel, bus travel, rail transport and combined routes. Some systems, such as Google Maps, also offer detailed information on navigation within the city, travel times considering road traffic and recommendations for optimal travel options.

Analysis of the functional capabilities of modern services shows that most of them are focused on finding individual transport segments or optimal routes according to one main criterion, for example, time or cost. However, complex travel planning tasks often require a multi-criteria approach: simultaneous consideration of time, price, number of transfers, route convenience and availability of vehicles. Not all existing systems allow for full optimization taking into account several criteria or comparing the effectiveness of routes according to different models.

Another feature of modern platforms is their dependence on external APIs that provide data on schedules, available routes, traffic conditions or geographical objects. Although this significantly expands the capabilities of the systems, it creates limitations in the availability and accuracy of data, which affects the quality of route construction.

Thus, despite the significant number of travel planning services, most of them focus on information search, rather than deep optimization. This creates a need to create analytical models and tools that allow for effective comparison of route search algorithms and determination of the most optimal solutions for different travel planning scenarios.

1.2 Features and requirements for optimal routes in transport networks

Optimization of routes in transport networks is one of the key tasks of modern intelligent travel planning systems. A transport network can be considered as a set of nodes (destinations, stations, stops) and edges (communication routes), each of which has certain characteristics: length, cost of movement, duration of travel, bandwidth and other parameters. Building an optimal route involves choosing a sequence of edges that satisfies certain criteria and limitations.

An important feature of transport networks is their heterogeneity: different modes of transport have different speeds, rules of movement, time intervals of movement and levels of accessibility. In addition, real networks dynamic — routes are affected by schedule changes, road conditions, weather conditions, delays, as well as individual user requirements. This complicates the task of finding the optimal solution and requires the use of flexible algorithms.

One of the key requirements for an optimal route is to minimize a certain indicator, such as the total duration of the trip, the cost of the trip or the number of transfers. However, in practice, there is more often a need for multi-criteria optimization, when several indicators need to be taken into account at the same time. It is also important to consider restrictions: time windows, availability of vehicles, restrictions on the maximum length of the route or the travel budget.

The optimal route should ensure stability and predictability. This means that the system must be able to adjust the route in real time when the conditions change, and the selected algorithm must allow quick updates of the results. An additional requirement is scalability, because transport networks can contain thousands of nodes

and tens of thousands of edges, which creates high requirements for the performance of algorithms.

Therefore, the features of transport networks and requirements for optimal routes determine the need to use algorithms capable of working in conditions of big data, dynamic changes and multi-criteria. This creates a basis for the selection of optimization methods, which will be discussed in the following sections.

1.3 Overview of existing travel planning systems

To analyze modern approaches to building routes and determine the requirements for the future intelligent system, it is advisable to consider two popular platforms — Google Maps and Rome2Rio. Both systems are international services, but they implement different approaches to finding and optimizing routes, which allows for a comparative analysis of their strengths and shortcomings.

Google Maps is one of the world's largest and most popular navigation and route planning systems, providing users with a comprehensive set of geolocation services. The platform covers more than 220 countries and territories, supports over 70 types of map layers, and provides integration with a large amount of satellite, street, and transportation data. The system is based on a combination of high-precision maps, powerful big data processing infrastructure, optimal route search algorithms, and artificial intelligence for traffic forecasting.

From a technological standpoint, Google Maps uses graph models of road networks and modifications of Dijkstra's algorithm, A*, and other heuristic methods to build the shortest and fastest routes. Routes are generated based on a complex system of weighting factors, including road length, speed limits, average traffic, number of traffic lights, traffic intensity, road restrictions, and other parameters. Google obtains a significant portion of its data through crowdsourcing: users’ smartphones transmit information about traffic speed, allowing the service to update traffic maps in near real time.

The system supports a variety of modes of use: walking, car routes, public transport, bicycle routes, taxi services, and in some countries, bicycle and electric scooter rentals and car sharing, is shown on Figure 1.1.

Key technological principles:

– graph algorithms: modified versions of Dijkstra and A* algorithms for shortest path search tasks;

– traffic forecasting models: machine learning that takes into account historical data, daily fluctuations, seasonality, and current traffic jams;

– dynamic optimization: rapid recalculation of routes in case of accidents, road closures, repair work, or transport delays;

– global geodata base: one of the largest and most detailed mapping systems.

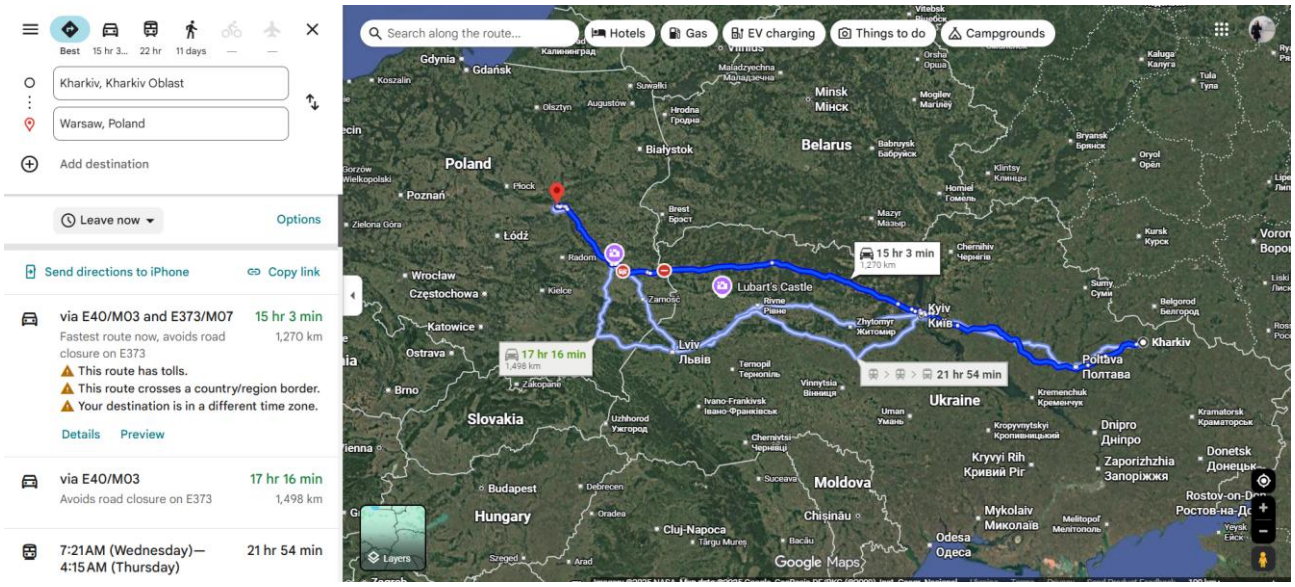

Figure 1.1 – Example route on «Google Maps»

Advantages of Google Maps:

– support for dynamic data: the system takes into account current traffic jams, road works, changes in traffic;

– high detail of cartographic data: exact coordinates, a large number of pois (points of interest);

– a large selection of modes of transport, including public transport in many cities around the world;

– fast route calculation thanks to optimized algorithms and scalable infrastructure;

– construction of alternative routes with time and distance, which allows the user to evaluate options.

At the same time there are disadvantages of Google Maps:

– limited support for combined travel between cities and countries: the service does not plan full-fledged complex tourist routes with railways, planes and buses at once;

– algorithm opacity: google does not provide information about internal optimization mechanisms and weighting factors;

– multicriteria optimization is limited: priority is given to time and distance, without considering other criteria such as cost or number of transfers;

– inability to compare different transport companies or tariff models in a single interface.

Rome2Rio is a global travel planning platform specializing in building intercity and international routes using various modes of transport. Unlike services focused on local navigation, Rome2Rio aims to provide users with a complete overview of possible ways to travel between two points, regardless of distance, country, or type of transport. The system covers more than 160 countries, thousands of transport companies, and hundreds of thousands of routes between cities and regions.

The basis of Rome2Rio's work is combining different transport segments into a single route. The service combines flights, trains, buses, city transport, ferries, subways, and in some cases even taxis or car rentals in a single query. The user receives not only a list of alternative routes, but also a travel structure in the form of a sequence of segments visualized on a map.

The platform collects information from official carrier databases, global distribution systems, open sources, and partner platforms. Pricing and schedule data can be either exact or approximate, depending on availability and the type of carrier. At the same time, the system provides direct links to the official websites of transport companies or aggregators, where users can confirm fares and purchase tickets.

Rome2Rio has a simple and intuitive visual structure displayed on Figure 1.2: main route, alternative options, approximate cost, duration, and number of transfers. This allows users to quickly evaluate different travel scenarios. The service is often used by tourists, travelers, students, and people planning international trips where it is difficult to find all types of transport on their own.

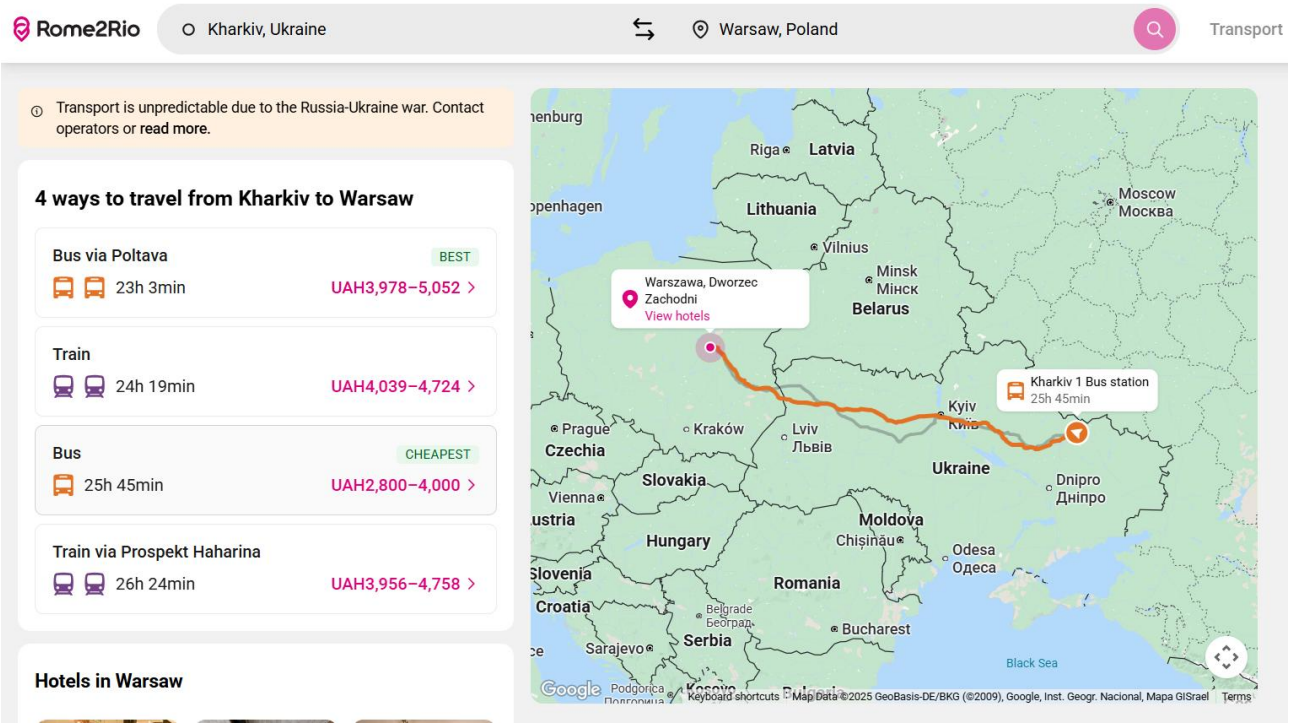

Figure 1.2 – Example route on « Rome2Rio»

Key technological principles:

– multimodal planning engine: optimized for long-distance travel, with transport hubs acting as nodes;

– carrier aggregation: integration of schedules, prices, and costs from private and public operators;

– simplified geomodelling: focus on transport connections rather than cartographic accuracy.

The advantages of Rome2Rio are its clear orientation to long-distance and international travel, which makes the service particularly convenient for planning long routes. The platform effectively combines different modes of transport within the same

route, which allows the user to form complex multi-component trips. An important feature is also the possibility of comparing the cost of trips, since the system provides approximate prices for different transport options. Rome2Rio is characterized by wide coverage and support for a large number of private and public carriers, and its interface provides a visual visualization of the stages of the route, which facilitates understanding of the overall structure of the trip and the duration of each segment.

At the same time, the service has several disadvantages. Rome2Rio offers less detailed mapping data compared to Google Maps, which may limit the accuracy of geographic information. The system does not consider dynamic conditions such as traffic congestion, delays or changes in the transport schedule, and therefore routes may be less adaptive to real situations. Cost and schedule data are often approximate and require additional verification on carrier websites. In addition, the algorithmic principles of route construction remain closed, which makes it impossible to analyze the optimization mechanism. The optimization itself is also limited, since routes are formed primarily based on the availability of connections, and not by taking into account a set of criteria, which is important for high-precision optimization of routes.

A comparison of Google Maps and Rome2Rio shows that both systems implement effective but different approaches to travel planning. Google Maps provides high-precision local routes with fast search and dynamic updates. Rome2Rio provides a wide overview of long-distance travel by combining different vehicles.

However, none of the systems performs an open comparative analysis of optimization algorithms, does not support flexible multi-criteria optimization, and does not allow customization of route criteria. This forms the basis for choosing the topic of this work — research and comparing route optimization algorithms with the possibility of their further integration into the prototype of an intelligent travel planning system.

1.4 Statement of the research problem

The task of optimizing routes is key in the process of travel planning, as it determines the most effective way of movement between given points, considering a set of possible restrictions and criteria. In general, a transport network can be

represented as a weighted graph, where vertices represent geographic locations or transport nodes, and edges — are possible paths of movement between them. Each edge has a set of characteristics, such as length, duration, cost, number of transfers or other parameters that affect the overall quality of the route.

Setting the optimization problem is to determine such a route or a set of routes that minimize or maximize a certain objective function. The most common criterion is the minimization of time or distance, however, in the context of travel planning, multi-criteria optimization is becoming increasingly important. The user may be interested in combined indicators, such as a simultaneous reduction in the duration of the trip, its cost and the number of transfers. In such cases, the task becomes complex and requires the use of specialized algorithms capable of finding compromise solutions.

An additional feature of the problem is the presence of various restrictions. These could be time windows, the availability of specific modes of transport, the need to synchronize carriers' timetables or the requirement to minimize expectations between transfers. An important aspect is the dynamic nature of transport systems: conditions on the route can change in real time, which creates the need for algorithms capable of updating solutions or quickly finding new routes.

The formal definition of route optimization problem includes the construction of a transport network model, the definition of target functions and constraints, as well as the selection of appropriate algorithms for finding optimal or approximate solutions.

To complete the task, it is necessary:

a) carry out an analysis of the subject area:

1) analyze modern travel planning systems and route construction methods (Google Maps, Rome2Rio, etc.);

2) determine the features of transport networks and formalize requirements for optimal routes;

3) form the scope of application of the future system and determine the goals of its use;

b) investigate route optimization algorithms:

1) conduct a detailed review of classic shortest path search algorithms (Dijkstra, A*, Bellman-Ford, Floyd–Warshall)[8];

2) consider heuristic and metaheuristic optimization methods (genetic algorithms, ant algorithms, artificial bee algorithm);

3) determine the criteria for comparing algorithms: accuracy, speed, scalability, resistance to network changes;

4) form a mathematical model of the route optimization problem with the possibility of multi-criteria analysis;

c) design the system prototype architecture and component interaction mechanism:

1) define architectural style (client–server, microservices or modular prototype);

2) describe the API for the interaction of the client part with the algorithmic optimization module;

3) design a data structure for the presentation of transport networks and calculation results;

d) implement a prototype travel planning system:

1) develop a module for implementing route optimization algorithms;

2) create a prototype backend with a basic API;

3) develop a client interface allowing entry of initial data and retrieval of routes;

4) test the correctness of algorithms and the interaction of components;

e) conduct an experimental study:

1) perform a series of experiments for different algorithms on sets of test graphs of different scales;

2) evaluate the speed, efficiency and quality of the constructed routes;

3) compare the results and determine the most optimal algorithm for the planning task.

2 RESEARCH OF METHODS OF SOLVING THE PROBLEM

2.1 Classic algorithms for finding the shortest paths (Dijkstra, Bellman–Ford, A*)

Classical algorithms for finding the shortest paths are the basis of many transport, logistics and navigation systems, since they provide a rigorous mathematical approach to determining the optimal route in graph structures. In such problems, the transport network is represented as a graph, where the vertices correspond to objects or locations, and the edges are possible movements between them with a certain weight, which can be interpreted as distance, time, cost or other resources.

The main idea of classical algorithms is to formally calculate the minimum total weight of the route between the initial and target points[9]. For this, various strategies for traversing the graph, methods of weight relaxation and procedures for systematically updating intermediate results are used. The basic mathematical basis of such methods is the Bellman optimality principle, according to which any segment of the optimal route must also be optimal.

Finding the shortest path is a key task in travel planning systems, transport networks and navigation applications. The efficiency of algorithms determining optimal routes directly affects the performance, response speed and quality of system recommendations. This subsection considers three classic algorithms that are widely used to find the shortest paths in graphs: Dijkstra's algorithm, Bellman–Ford's algorithm, and A*'s algorithm. Each of them has its own features, limitations and areas of application.

2.1.1 Dijkstra's algorithm

Dijkstra's algorithm is designed to find the shortest paths in weighted graphs without negative edge weights. The main idea is to stepwise extend the set of vertices with an already defined minimum distance from the starting point.

Main characteristics:

– only works with non-negative weights;

– optimal and deterministic;

– usually implemented using a priority queue;

– has good performance for large networks (especially with queue based on binary or Fibonacci heap).

The simplest implementation of Dijkstra's algorithm requires $O(E \cdot log(V))$ actions. It uses an array of distances and an array of marks. At the beginning of the algorithm, the distances are filled with a large positive number (greater than the maximum possible path in the graph), and the array of marks is filled with zeros. Then the distance for the initial vertex is considered zero and the main cycle is started.

As the algorithm runs, we will gradually update this array, finding more optimal paths to vertices and reducing their distances. When we determine that a path to a vertex is optimal, we will mark that vertex by placing a one in the array, initially filled with zeros. The algorithm itself consists of iterations, on each of which is selected the vertex with the smallest value among the not yet marked.

The selected vertex is marked in the array, after which relaxations are performed from the vertex v: we look through all outgoing edges (u, v) and for each such vertex u we try to improve the value of $dist(u)$ by assigning a value calculated using formula 2.1.

$$dist(v) = \big(dist(v), dist(u) + w(u, v)\big), \tag{2.1}$$

where $w(u, v)$ is an edge length of (u, v).

At this point, the current iteration ends, and the algorithm moves on to the next: the vertex with the smallest value is again selected, relaxation is performed from it, and so on. After n iterations, all vertices of the graph are labeled, and the algorithm terminates.

Figure 2.1 shows an example of Dijkstra's algorithm, demonstrating how the algorithm successively expands the region of visited vertices and updates the shortest paths to neighboring nodes.

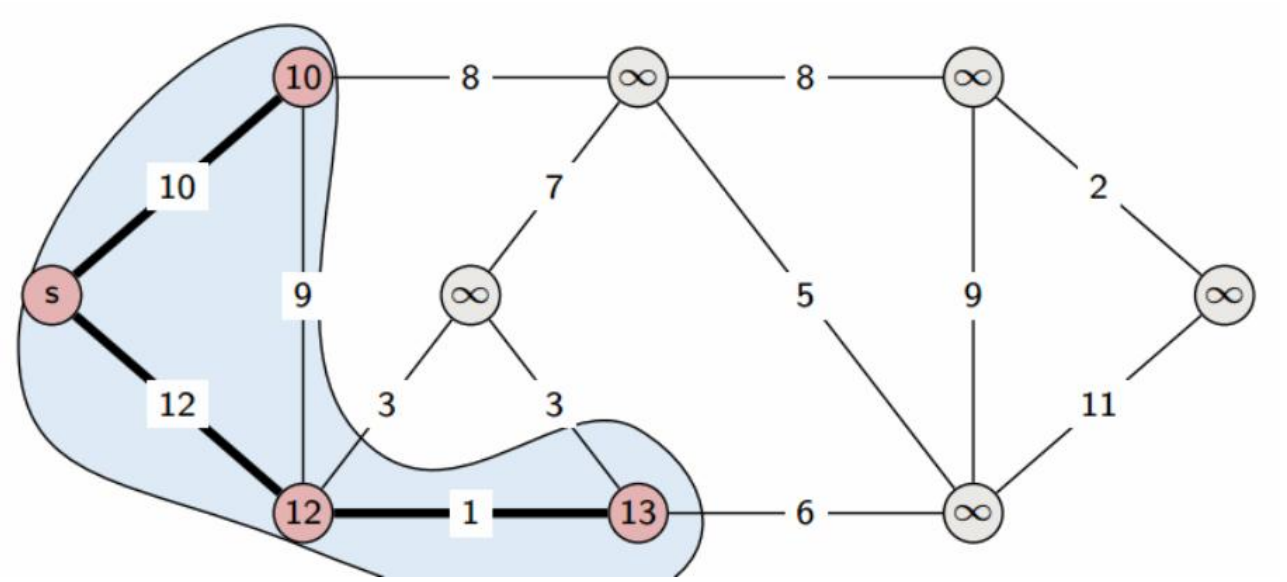

Figure 2.1 – The visual representation of one of the steps of Dijkstra's algorithm

To generalize the key characteristics of the algorithm, a comparative table was formed (Table 2.1), which presents the strengths and weaknesses in terms of application in travel planning tasks.

Table 2.1 – Advantages and disadvantages of Dijkstra's algorithm

Advantages	Disadvantages
– high speed of operation; – effective in dense and sparse graphs; – ideal for GPS navigation and static networks.	– cannot operate with negative weights; – with many vertices, it needs optimized data structures.

2.1.2 Bellman-Ford's algorithm

The Bellman-Ford’s algorithm is one of the fundamental methods for finding the shortest paths in weighted directed graphs. Unlike Dijkstra's algorithm, it is able to work not only with positive, but also with negative edge weights, which makes it a universal tool in route optimization problems where penalties, compensation, or inverse coefficients are possible[10]. The main idea of the method is based on the sequential "relaxation" of the graph edges over a limited number of iterations.

The key idea of the algorithm is based on iterative improvement of estimates of the shortest paths. We start with a distance of 0 to the original vertex and infinity to all others. At each iteration, we try to improve (decrease) the estimate of the distance to each vertex by considering all edges of the graph. The relaxation formula (2.1) applies to every edge in every iteration. After (|V| - 1) iterations, we are guaranteed to find the shortest paths if there are no negative cycles in the graph. Additional iteration allows checking the presence of negative cycles. The result of one cycle of the Bellman-Ford's algorithm is represented on Figure 2.2.

This idea is closely related to dynamic programming, where we gradually build an optimal solution based on subtasks.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

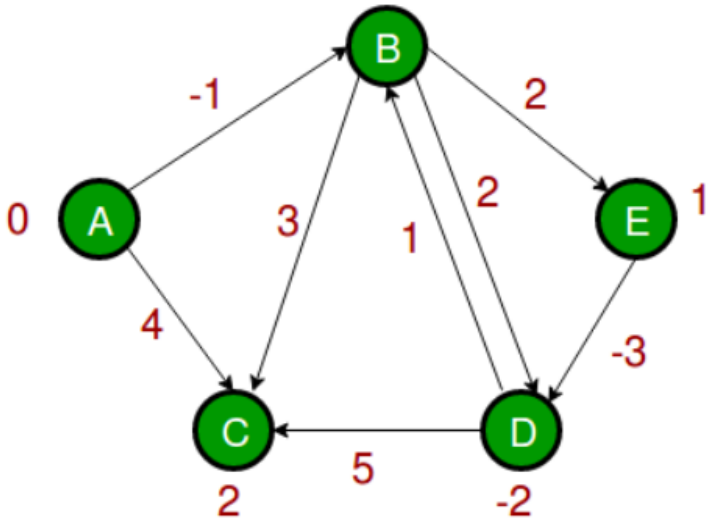

Figure 2.2 – The visual representation of one of the steps of Bellman–Ford's algorithm

If the graph is given by a list of edges: initialization takes $O(V)$ time, each of the |V|–1 passes takes $O(E)$ time, a pass along all the edges to check for a negative loop takes $O(E)$ time. So, the algorithm works in $O(E)$ time.

If the graph is given by the adjacent matrix, then the algorithm will be executed in $O(E^3)$ time.

To generalize the key characteristics of the algorithm, a comparative table was formed (Table 2.2), which presents the strengths and weaknesses in terms of application in travel planning tasks.

Table 2.2 – Advantages and disadvantages of Bellman–Ford's algorithm

Advantages	Disadvantages
– works with negative weights; – capable of detecting negative cycles; – guarantees the optimality of the solution.	– considerably slower than Dijkstra; – not suitable for large dynamic systems (e.g. online navigation).

2.1.3 The A* algorithm

The A* algorithm is heuristic and combines the Dijkstra rate with the search directionality provided by the heuristic function. A* finds the shortest path in the graph if the heuristic is admissible (does not overestimate the distance) and consistent.

The algorithm uses an auxiliary function (heuristics) to direct the direction of the search and shorten its duration. The algorithm is complete in the sense that it always finds the optimal solution if it exists.

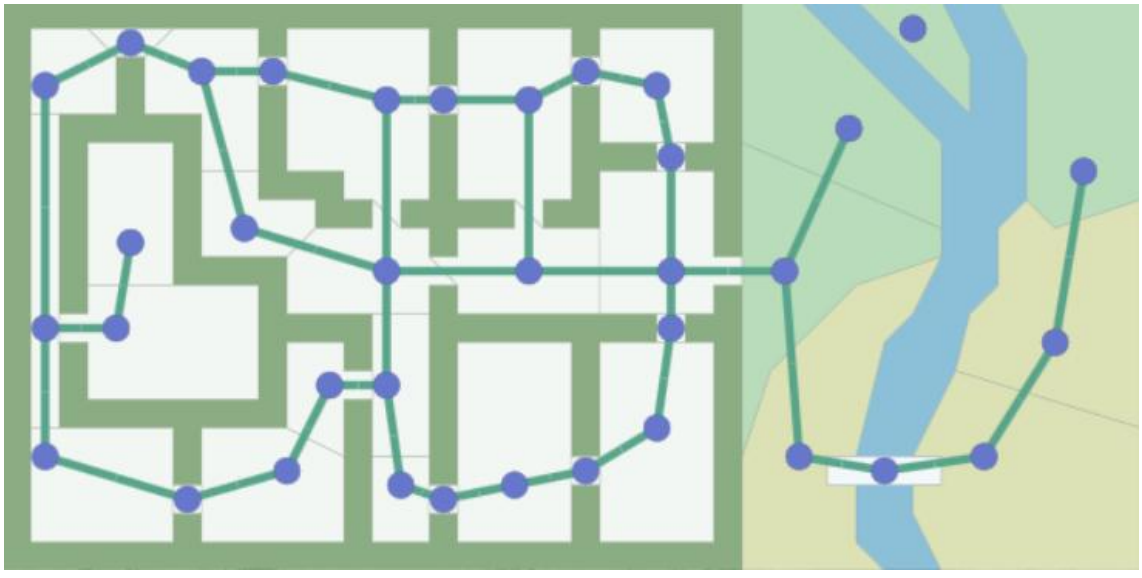

Figure 2.3 – Example of an input graph for the A* algorithm

Algorithm A* first visits those vertices that probably lead to the shortest path to the goal. To recognize such vertices, each known vertex x is matched with a value of $f(x)$, which is equal to the length of the shortest path from the original vertex to the final one displayed on the formula (2.2), which runs through the selected vertex. The vertices with the lowest value f are chosen in the first place.

Function $f(x)$ for vertex x is defined as:

$$f(x) = g(x) + h(x), \quad (2.2)$$

where $g(x)$ a function whose values are equal to the cost of the path from the initial vertex to x;

$h(x)$ is a heuristic function, estimates the cost of the path from the vertex x to finite.

The heuristic used should not give an overestimate of the cost of the path. An example of an estimate can be a straight line: the total path cannot be shorter than a straight line[11].

To generalize the key characteristics of the algorithm, a comparative table was formed (Table 2.3), which presents the strengths and weaknesses in terms of application in travel planning tasks.

Table 2.3 – Advantages and disadvantages of A* algorithm

Advantages	Disadvantages
– much faster than Dijkstra in practice; – can work in large navigation graphs (cities, roads); – allows you to flexibly adjust the behavior of the algorithm.	– quality depends on heuristics; – with poor heuristics degrades to Dijkstra complexity; – does not handle negative weights.

The algorithms of Dijkstra, Bellman-Ford and A* are basic approaches to solving the problem of finding the shortest path in transport networks. Dijkstra provides fast operation under non-negative weights, Bellman-Ford – universality for

graphs with negative edges. A comparison of these algorithms is necessary for the selection of methods that will form the basis of an intelligent travel planning system.

2.2 Algorithms for graphs with schedules (time-dependent routing)

In real-world transportation systems, travel between two points depends not only on the network structure but also on departure time. Arrival times depend on train, bus, and flight schedules, service intervals, transfer times, and potential delays. Therefore, the traditional model of representing a transportation network as a static graph with constant edge weights is insufficient for travel planning problems.

To correctly model such processes, a time-dependent graph is used, in which the edge weight depends on the start time of travel. This section discusses the main approaches to routing in graphs with schedules: the time-dependent weight model, the time-expanded graph model, and the Time-Dependent Dijkstra algorithm.

Time-dependent graphs are widely used in:

– public transport systems;

– long-distance and international transportation (buses, trains, airplanes);

– logistics networks.

Let $G = (V, E)$ – be a directed graph. For each edge $e = (u, v) \in E$ a cost function is defined that depends on the departure time from vertex u: $w_e(t): R_{\geq 0} \rightarrow R_{>0}$.

The function can set:

– travel time (e.g., varying depending on traffic);

– waiting time (e.g., bus schedule);

– price depending on the time of day (dynamic pricing).

If a traveler arrives at vertex u at time t_u, then the arrival time at vertex v via edge e will be:

$$t_v = t_u + w_e(t_u) \tag{2.3}$$

Thus, relaxation in TD-Dijkstra differs from static relaxation in that the edge weight is calculated on the fly.

2.2.1 Time-Dependent Dijkstra's algorithm

Dijkstra's algorithm works correctly in time-dependent graphs only if the FIFO (First In, First Out) property, also called non-overtaking condition, is satisfied.

$$t_1 < t_2 \Rightarrow t_1 + w_e(t_1) \leq t_2 + w_e(t_2) \quad (2.4)$$

This constraint reflects the realistic behavior of transportation systems: the same vehicle (bus, train, car on the road) cannot «overtake itself in time.»

Orda and Rom (1990) proved that FIFO is a necessary and sufficient condition for the correctness of the extended Dijkstra algorithm on time-dependent graphs.

The algorithm is similar to Dijkstra's classic algorithm, but with a key change: when removing a vertex u from the minimum-time queue, the relaxation of each edge (u, v) is performed with the cost calculated at the current arrival time $if\ t_u + w_{(u,v)}(t_u) < t_v, then\ update\ t_v$. In other words, the weight of an edge is not determined in advance but depends on when the algorithm «reaches» the vertex.

Description of the algorithm behavior:

– the priority queue remains correct: extracting the vertex with the minimum arrival time remains valid if the graph satisfies FIFO;

– the result depends on the start time t_0: unlike static graphs, TD-Dijkstra must be run separately for each start time;

– bidirectional search cannot be used: backward edges in TD-graphs have a different time function, and "lookback" loses correctness;

– timing cycles (t-cycles) can exist however, if they violate FIFO, the algorithm ceases to be applicable.

The TD-Dijkstra algorithm has several advantages that make it popular in routing problems where travel time depends on the departure time. First, it is naturally applicable to road networks with variable travel speeds due to traffic, as well as to scheduled transportation systems – city buses, minibuses, trains, and air travel, including scenarios with transfers. Another significant advantage is that this algorithm is a relatively simple adaptation of the classic Dijkstra algorithm: the basic principles

remain the same, only the method for calculating the edge weight at the relaxation time changes. This makes TD-Dijkstra easily integrated into multimodal systems that require combining road, rail, and air travel within a single optimization model.

However, the algorithm also has significant limitations. The key one is the requirement for strict adherence to the FIFO property of the weight functions, without which the correctness of the algorithm is lost. Furthermore, TD-Dijkstra is generally slower than the static case, since computing the edge cost at each relaxation requires an additional time function call. In transportation networks with a very large number of events (e.g., dense public transportation schedules), TD-Dijkstra's performance is significantly inferior to specialized algorithms such as RAPTOR or CSA, which are optimized specifically for scheduling. Finally, TD-Dijkstra is not an effective solution for multi-criteria routing problems, where travel time, ticket prices, and comfort levels must be considered simultaneously, since the standard algorithm model is designed for only one optimization criterion.

Routing algorithms in time-dependent graphs are key to travel planning systems. Depending on the requirements, a time-dependent model with TD-Dijkstra is best suited for compactness and speed, while a time-expanded model is preferable for detailed multimodal navigation and complex schedules.

2.3 Algorithms for multi-criteria route optimization

Multi-criteria optimization in routing is an approach to finding the best paths that considers multiple competing objectives simultaneously, rather than just path length as in the classic shortest path problem (e.g., Dijkstra's algorithm). Instead of a single "best" path, MCO allows for finding a set of Pareto-optimal solutions that balance criteria such as time, cost, reliability, safety, or network load, using methods such as scalarization or fuzzy logic to cope with the high dimensionality of real-world networks[12].

Routing problems in transportation networks often require consideration of multiple path quality indicators. Typical criteria include total travel time, ticket price, number of transfers, comfort level, risk of delay, and environmental friendliness or

delay probability. Such problems are formulated as multi-criteria optimization problems (MCSPs), which require finding not a single optimal path, but multiple solutions that satisfy a specific optimality principle.

Unlike classical shortest path search, where the optimal route is the one that minimizes the total weight, Pareto optimality is most often used in multicriteria problems. Let each edge $e = (u, v) \in E$ correspond to a weight vector $w_e = (w_e^1, w_e^2, \dots, w_e^k)$, where each component is the value of the next criterion. Then the weight of a path P between vertices is defined as the sum of the edge weight vectors:

$$W(P) = \sum_{e \in P} \quad w_e \tag{2.5}$$

Path P_1 dominates path P_2, if $W(P_1) \leq W(P_2)$ by all criteria and at least one strictly.

The set of paths that are not dominated by any other path forms the Pareto set, which represents compromise solutions. This is precisely what multi-criteria path-finding algorithms seek to determine.

2.3.1 The label-setting approach

The label-setting approach is a class of algorithmic methods for finding shortest paths in graphs, where for each vertex only one current label is stored, representing the best-known value of an optimization metric (for example, minimum arrival time). Unlike the multi-label approach (label-correcting), where there can be many labels for a single vertex, label-setting methods operate in a monotonic refinement mode and do not allow returning to already processed vertices.

This approach is the basis of the classical Dijkstra and A* algorithms, as well as their modifications for time-dependent graphs (time-dependent shortest path). In travel planning problems, this allows for efficient handling of transport schedules, time constraints, and travel duration variability[13].

Within the label-setting framework, each vertex v is assigned a label $L(v) = best\ known\ arrival\ time\ /\ cost,$ which is final after the vertex is selected for processing.

The process consists of two key steps, namely selecting the vertex with the best current label and checking the finality of the label.

The algorithm always chooses the vertex with the smallest label value among those that have not yet been processed displayed at formula 2.6.

$$v^{\setminus *} = arg\ min_{\ v \notin S} L(v), \tag{2.6}$$

where S is the set of already processed vertices.

After this vertex is selected, its label is considered optimal and never changes again: $L(v^{\setminus *}) = L^{\setminus *}(v^{\setminus *})$, i.e. it coincides with the true value of the shortest path.

This property is possible only under certain conditions if the edge weight function is monotonic and does not lead to situations where a later departure time gives a faster result (FIFO property).

The time/cost function should depend only on the edge and the time point, and not on which edges were previously traversed (path-independent cost).

In real transport networks this is not always possible - for example, in the simulation of transfers with fines, zonal tariffs or delays. In such cases, label-correcting approaches are used.

Advantages of the label-setting approach:

– high efficiency: complexity $O((V + E)logV)$ when using a minimum queue;

– optimality on FIFO graphs;

– ease of implementation (especially in prototypes);

– minimal memory usage: one label per vertex;

– scales well to large transport networks.

Most real navigation systems (Google Maps, Here Maps, OpenTripPlanner) use label-setting.

The label-setting approach has a number of important limitations that determine the limits of its practical application[14]. First of all, it is unsuitable for graphs in which the FIFO property is not satisfied: if an early departure time can lead to a later arrival, the algorithm is no longer able to guarantee the optimality of the found routes. This

approach also does not work for graphs with negative edge weights, since under such conditions it is impossible to ensure the finality of the labels after their processing. Significant difficulties arise in situations where the optimal route depends on complex transfer rules, time windows, zonal tariffs or penalties for certain types of movements, which requires storing multiple states for each vertex. Because of this, label-setting is also not suitable for problems where several equal optimal solutions may exist for one vertex - for example, in multi-criteria optimization or when forming a set of Pareto-optimal paths. In such cases, more flexible label-correcting approaches are used, including Bellman–Ford algorithms, multi-label shortest path, RAPTOR, Connection Scan, or multi-criteria modifications of Dijkstra's algorithm.

The label-setting approach forms the foundation for most efficient shortest path algorithms and allows for the implementation of computationally optimal solutions to routing problems, including time-dependent transportation networks. In the context of a travel planning system, this approach provides a combination of accuracy, scalability, and practicality, making it a suitable basis for implementing a prototype of an intelligent information system.

2.3.2 The label-correcting approach

The label-correcting approach is a generalized class of methods for finding shortest paths in graphs that allows adjusting the label values of vertices an unlimited number of times until convergence is achieved. Unlike the label-setting approach, where each vertex is assigned a label once and does not change again, in label-correcting algorithms the label value can be revised after repeated relaxation of the edges[15]. This makes such methods much more flexible, but less computationally efficient.

In the label-correcting approach, the algorithm makes no assumption about the finality of the label after it has been processed. On the contrary, the result can be updated if a path later appears that improves the current estimate. This means that the algorithm can return to the vertex many times.

Formally, the vertex label v stores the best known value from formula 2.7 and the algorithm allows the edge relaxation based on formula 2.8 voluntarily many times until all possible improvements are exhausted.

$$L(v) = cost(s, v)_{\ all\ paths\ s \to v'} \tag{2.7}$$

$$L(v) = min\ (L(v),\ L(u) + w(u, v)) \tag{2.8}$$

Unlike label-setting, label-correcting does not require either the FIFO property or the absence of negative weights in the graph to work correctly. Algorithms of this type are able to function in the conditions of graphs with negative weights, non-FIFO time-dependent graphs, graphs with complex transfer rules, multi-criteria problems, models with "time windows" for flights or transport transfers.

The only requirement is the absence of a negative cycle, otherwise the shortest path problem has no solution.

The label-correcting approach has a number of key advantages:

– flexibility: able to work with any type of weights and constraints;

– support for negative weights;

– ability to model complex transfer rules, penalties, time windows;

– natural extension for multi-criteria problems;

– guaranteed optimality of the result if there is no negative cycle.

In transport applications, label-correcting approaches are relevant where it is important to take into account departure/arrival time constraints, different ticket types, or the specifics of public transport.

The most significant disadvantage is lower efficiency compared to label-setting. Due to the possibility of repeated label updates, algorithms can process the same vertex many times. This leads to significant performance degradation, especially in large graphs.

The label-correcting approach is more general and flexible than label-setting and can work with different classes of graphs, including complex, time-dependent, and multicriteria graphs. Its use is appropriate where efficient algorithms such as Dijkstra

or A* cannot be applied. Although these algorithms have higher computational complexity, they provide solutions to problems with negative weights, time constraints, and complex transportation rules.

In the context of the development of a travel route optimization system, the multi-criteria model is especially relevant, since it is important for the user:

– minimum travel time;

– reasonable price;

– the minimum number of transfers;

– comfort;

– restrictions on the time of departure or arrival.

Label-setting methods are well suited for relatively small networks or a limited number of criteria[16].

Label-correcting methods are useful in more complex situations when the network has many options, is scheduled or when it is necessary to support dynamically changing data. Comparative characteristics of label-setting and label-correcting approaches are displayed in Table 2.4.

Table 2.4 – Comparative characteristics

Characteristic	Label-setting	Label-correcting
General principle	The vertex label is set once and never changes again	Labels can be updated multiple times until convergence is achieved
Typical algorithms	Dijkstra, A*, Time-dependent Dijkstra	Bellman–Ford, SPFA, Multi-label SP, RAPTOR, CSA
Ability to work with negative weights	No	Yes
Optimality	Guaranteed with positive weights and FIFO properties	Guaranteed in the absence of negative cycles
Number of status updates	Monotonically increasing; each vertex is processed once	Unlimited label updates
Efficiency	High; works fast on large graphs	Lower; significant time consumption possible

Continuation of table 2.4.

Implementation complexity	Simple to implement	More complex logic, queues, state sets
Multi-criteria support	Limited (often non-functional)	Natural: Ability to store multiple shortcuts
Support for time-dependent models	Works only for FIFO graphs	Works in non-FIFO models
Graphs with time windows, penalties, complex transfers	Partially or difficultly supported	Well maintained
Scalability	Very good	Satisfactory or low
Typical areas of application	Classical navigation systems, road networks, shortest path in static graphs	Transport schedules, public transport, multi-criteria optimization
Behavior in large graphs	Stable runtime	May degrade
Application in a system prototype	Basic routing algorithms in your work	Used as a theoretical contrast, and as a class of algorithms for more complex models

The consideration of multi-criteria optimization methods has shown that travel planning problems can rarely be adequately described by a single criterion, since users simultaneously expect minimal duration, low cost, comfort, and a reduced number of transfers. That is why single-criteria algorithms are insufficient for realistic modeling. Multi-criteria approaches, in particular models based on Pareto-optimality and methods for forming a front of non-dominated solutions, allow considering several criteria simultaneously, forming a set of acceptable routes instead of one «best». However, their use is associated with increased computational complexity and the need for specialized algorithms for processing dominance. The analysis found that multi-criteria models are key for building intelligent travel planning systems, since they most fully reflect the real needs of users.

2.4 Algorithms for searching for multiple alternative routes

In several practical tasks related to travel planning, it is necessary not only to find one optimal route, but also to provide the user with several high-quality

alternatives. This is especially important in conditions of uncertainty (transport delays, a tight schedule of transfers, possible schedule changes) and when taking into account user preferences (avoiding certain stations, choosing a more comfortable or cheaper route).

The classic problem of finding several shortest paths is formulated as the problem of finding a set k routes from the initial vertex s to the target vertex t, sorted by increasing weight, while the paths must be different, and preferably structurally dissimilar.

There are two basic algorithmic approaches, the most widespread in theory and practice: Yen's algorithm and Epstein's algorithm. They differ significantly in the principles of operation and computational efficiency, which makes them suitable for different classes of tasks.

2.4.1 Yen's algorithm

Yen's algorithm is the classic and most frequently used method for finding multiple different shortest paths between two graph vertices. It builds solutions sequentially, starting with the shortest path, typically obtained using Dijkstra's algorithm.

The algorithm is designed to find k minimum-length paths in a weighted graph connecting vertices $u1$ and $u2$. Paths that do not contain loops are searched for.

So, the problem is to find several minimum paths, so the question arises of how to avoid finding a path containing a loop. In the case of finding a single minimum-weight path, this condition is necessarily satisfied. In this case, we use Yen's algorithm, which allows us to find k simple shortest paths[17].

The algorithm begins by finding the shortest path, for which we will use the algorithm already described. We find the second path by enumerating the shortest deviations from the first, the third by enumerating the shortest deviations from the second, and so on.

Let $i-1$ shortest paths $P_1, \dots, P_{i-1}$ is found. To find the path P_i, the algorithm iterates over so-called spur nodes along the path P_{i-1}. For each such point v:

a) a prefix of the path is considered – the segment from s to v;

b) the remaining part of the path (sufix) is replaced by a new shortest path from v to t, temporarily removing from the graph:

– edges or vertices previously used in similar prefixes;

– the entire previous path to avoid duplication;

c) for each such deviation, a candidate for the next shortest path is calculated.

After the set of candidates is formed, the one with the minimum weight is selected, and it becomes the next route P_i.

Figure 2.4 shows the step-by-step operation of Yen's algorithm for finding k shortest paths between vertices A and F.

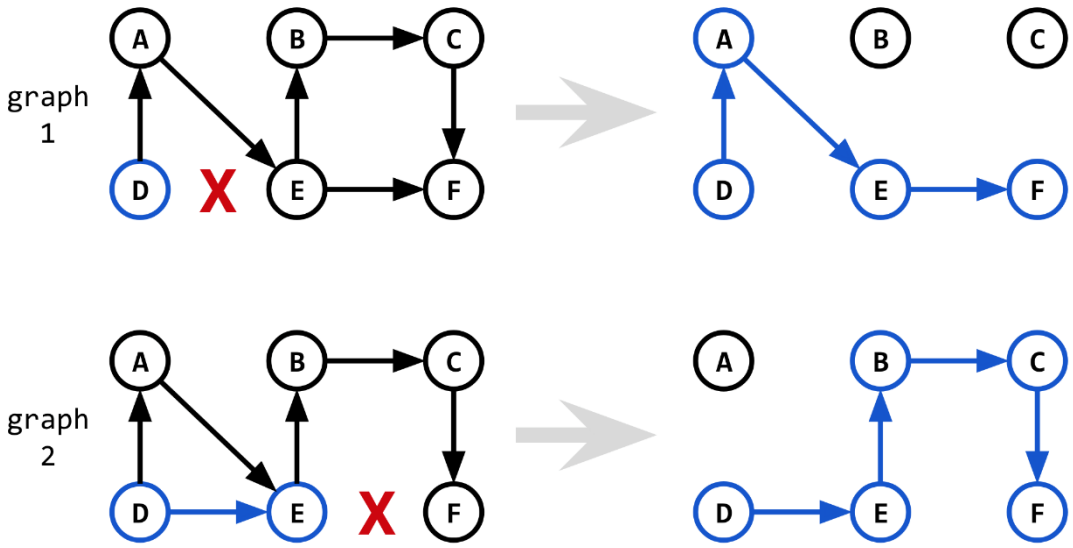

Figure 2.4 – Demonstration of Yen's algorithm for finding k-shortest paths

2.4.2 Eppstein's algorithm

Eppstein's algorithm is one of the most efficient solutions for finding a set of k shortest paths in directed graphs. Unlike Yen's algorithm, it constructs a data structure that allows for immediate generation of paths in sorted order, with virtually no need for repeated Dijkstra runs.

The algorithm consists of three key steps:

– construction of a shortest path tree from vertex s (usually using Dijkstra's algorithm);

– formation of a set of deviations – edges that are not included in the shortest path tree. These are considered potential "branching points";

– construction of a specialized priority structure that allows for the generation of new paths in ascending cost order.

The algorithm uses the persistent heap technique, which allows for the rapid combination of an optimal path with various deviations.

Yen's algorithm is suitable for cases where a small number of alternatives (3-10 routes) need to be generated, and the network is not too large. It is simple and reliable.

Epstein's algorithm is preferable when the network is large (air or rail routes across a country/continent), many alternatives are required, and high performance is important.

If structurally distinct routes need to be generated (for example, to prevent alternatives from passing through the same transfer hubs), both algorithms can be improved by penalizing edges or using specialized methods for finding diverse routes (Diverse Routing, Dispersion Methods).

This section provides a comprehensive theoretical overview of classical, heuristic, metaheuristic, and specialized route optimization algorithms used in shortest path search problems in transport networks. The properties of static and time-dependent graphs are analyzed, and the differences between models with fixed weights and graphs in which the duration of trips depends on the moment of time are determined. Particular attention was paid to such groups of methods as classical path search algorithms (Dijkstra, A*, Bellman–Ford), approaches for time-dependent graphs, label-setting and label-correcting techniques, and algorithms for finding the set of best routes (Jenn, Epstein).

3 IMPLEMENTATION OF A PROTOTYPE OF AN INTELLECTUAL SYSTEM

3.1 General characteristics of the prototype implementation

The implementation of the prototype intelligent travel planning system is based on the results of the analysis of route optimization algorithms discussed in the previous sections, as well as on modern approaches to building software systems for support. At this stage, the basic principles of the system's functioning, the logic of interaction between its components, the approach to processing user input data, and the formation of output results in the form of optimal routes are determined. The implementation of the prototype is focused on creating a flexible software environment that allows for intelligent information processing and provides the user with relevant recommendations for choosing a travel route.

The key task of the prototype implementation is to provide intelligent support for the process of selecting the optimal route based on user-defined criteria minimizing travel time, travel cost, number of transfers, and other additional parameters that may affect the comfort and efficiency of travel. The intelligence of the system lies in its ability to analyze alternative route options, compare them based on a set of indicators, and generate recommendations that best meet the specified requirements. Thus, the system acts as a decision support tool in the field of travel planning.

For formal presentation of the route planning problem, the transport network is represented as a weighted directed graph, which is a commonly accepted mathematical model for routing problems. In this model, the vertices of the graph correspond to departure points, arrival points, and intermediate transport nodes, while the edges represent possible options for moving between them using different modes of transport. Each edge is assigned a specific set of parameters, which may include the time taken to travel a section of the route, the cost of travel, the distance, and other auxiliary characteristics. This approach ensures the versatility of the model and its adaptability to different task conditions.

The weight coefficients of the edges are formed according to the selected optimization criterion. In the case of minimizing travel time, the weights correspond to

the duration of travel between points; in the case of minimizing cost, they correspond to monetary expenses; and in the case of combined criteria, weighted coefficients can be applied that take into account several parameters at once. This allows the system to be adapted to different travel planning scenarios and also provides the ability to compare the results obtained using different optimization criteria.

The information system supports several optimization modes, ensuring the flexibility and versatility of the intelligent system. Users can change the priority of criteria depending on their needs, for example, choosing the fastest route, the cheapest option, or the route.

The prototype supports several optimization modes, ensuring the flexibility and versatility of the intelligent system. Users can change the priority of criteria depending on their needs, for example, choosing the fastest route, the cheapest option, or the route with the fewest transfers. This approach expands the system's functionality and increases its practical value for real-world use.

The prototype also processes various types of input data, including information about the departure point and destination, selected optimization parameters, time constraints, and any additional user preferences. Based on this data, the system generates a request to build a route and performs the appropriate calculations using the selected optimization algorithms. The results are presented in the form of one or more alternative routes with an indication of their main characteristics.

The implementation of the prototype is focused on further expanding the functional capabilities of the intelligent system, in particular, the integration of additional optimization algorithms, new decision-making criteria, geoinformation services, and modern route visualization tools. This approach creates the conditions for using the developed system not only for educational and research purposes, but also in applied information systems for tourism and transportation.

An important feature of the prototype intelligent system is its focus on the multi-criteria nature of travel planning. In real-life situations, users are usually not limited to a single criterion of optimality, but seek to find a compromise between travel time, financial costs, comfort, and the number of transfers. In this regard, the prototype

provides for the possibility of combining several criteria by using appropriate weighting coefficients, which allows for the formation of a generalized optimization objective function.

The implementation of such a model makes it possible to study the behavior of various optimization algorithms in the context of a multi-criteria problem, as well as to evaluate their ability to find effective solutions in the presence of conflicting requirements. This is important for the practical use of the system, as it allows its operation to be adapted to the individual needs of users and different travel scenarios.

In the process of implementing the prototype, considerable attention is paid to the processing and structuring of initial data. Input data can come in the form of static sets of information about transport hubs and connections between them, as well as dynamic data that is updated in real time. Such data may include changes in transport schedules, flight delays, route closures, weather conditions, etc. Although a simplified data model is used within the basic prototype, its structure allows for further integration with real information sources.

In addition, the implementation of the intelligent system prototype involves the creation of a unified interface between the algorithmic part and the user level. This allows the route optimization logic to be separated from the means of presenting the results, which in turn increases the flexibility of the software and simplifies further modernization of the system. This approach is characteristic of modern intelligent information systems and ensures their ease of scaling.

During the implementation of the prototype, special attention was paid to the correctness and consistency of computational processes. When constructing routes, it is necessary to ensure the correct processing of borderline cases, in particular situations where there is no acceptable route between the specified points, or where there are several routes with similar target function values. In such cases, the system must correctly inform the user of the search results and, if possible, offer alternative solutions.

Another important aspect of the prototype implementation is the support of alternative results. The intelligent system is not limited to generating only one optimal

route, but provides the user with the opportunity to familiarize themselves with several alternative options that may differ in certain parameters. This increases the informativeness of the system and allows the user to independently choose the most acceptable option, taking into account subjective factors.

The implementation of the prototype also takes into account the requirements for the computational efficiency of optimization algorithms, since the task of route planning in large transport networks can require significant computational resources. In this context, it is important to ensure acceptable computation time even with a large number of graph vertices and edges. This allows the system to be used in interactive mode, which is critical for practical travel planning services.

In general, the implementation of a prototype intelligent travel planning system within this section is aimed at creating a universal software tool that combines the capabilities of mathematical modeling, algorithmic optimization, and intelligent decision support. The developed approach allows not only to perform an automated search for optimal routes, but also to conduct experimental studies of the effectiveness of various algorithms in real conditions of multi-criteria optimization.

3.2 Architecture of the intelligent system prototype

Architectural design plays a key role in creating a prototype of a route optimization system, since it determines the structure of interaction between components, the scalability of the system, the possibility of further expansion of functionality and adaptation to various algorithmic approaches. During the development of the prototype, a multi-layer architecture was chosen, which provides a clear separation of responsibilities between the logical parts of the application and allows isolating complex computational components from interface and data components.

The system uses a four-layer model, which enables independent development and modification of each level. This is especially important for routing tasks, where the algorithmic part can change intensively - for example, when adding new algorithms, implementing multi-criteria logic or experimental optimizers.

When forming architectural requirements, the following aspects were taken into account:

– support for various optimization algorithms (classical, heuristic, multi-criteria);

– the ability to quickly replace or add new algorithms without changing the entire system;

– ensuring the independence of business logic from the user interface;

– easy connection of data sources (files, databases, external APIs);

– the ability to scale the functionality in the future.

Given this, the system architecture was built on the basis of four key layers: Presentation Layer, Application Layer, Domain Layer and Data Layer, displayed on the Figure 3.1.

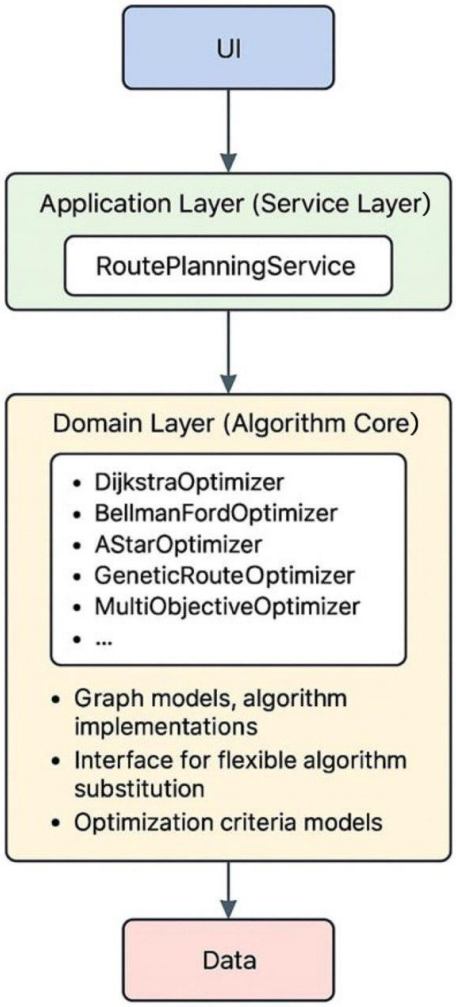

Figure 3.1 – The system architecture

The presentation layer is responsible for the user's interaction with the system. Within the prototype, it serves as an interface layer through which the user can configure route search parameters, select a specific algorithm, define optimization criteria, and view the results. The layer does not contain algorithmic logic - its sole purpose is to provide convenient data entry and visibility of the received answers.

In this study, the Presentation Layer is implemented as simply as possible, since the focus is on the algorithmic core of the system. The interface can be built as a console module or as a narrow GUI prototype based on WPF or WinUI technologies. It is important that the Presentation Layer interacts only with the service layer, without having access to the algorithmic logic or graph data structures. This allows you to easily update the interface without the risk of disrupting the operation of the optimization components.

The service or application layer acts as the central coordinating link between the user interface and the algorithmic core. It is it that determines the sequence of operations, processes the parameters received from the interface, and provides the appropriate call to the optimizer. This layer implements the business logic of the prototype, but does not contain mathematical models or algorithms - they are isolated in the domain layer.

The application layer also performs the functions of validating input data, ensuring format agreement, and generating results in a form convenient for presentation to the user. The key element of this layer is the routing service, which analyzes the choice of algorithm, loads the graph structure, calls the appropriate optimizer, and returns the optimal path to the user. Such an organization allows you to easily replace or expand the set of algorithms without changing the interface logic.

The domain layer is the core of the system and contains all algorithmic components. This layer implements transport network models, graph structures, time dependencies, weight calculation functions, optimization criteria, and directly the algorithms for finding optimal routes. It is important that this layer is isolated from any external dependencies, which allows you to focus as clearly as possible on mathematical and algorithmic aspects.

The domain layer structure provides an abstract optimizer interface that allows you to connect new algorithms without changing the existing infrastructure. Implementations can include both classical algorithms (Dijkstra, Bellman–Ford, A*), and heuristic (genetic algorithm, ant algorithm) or multi-criteria approaches. The domain layer is also responsible for forming metrics for evaluating the quality of routes, which are later used in the service layer.

The data is managed by a separate layer that provides access to graph structures, router configurations, and auxiliary parameters. It can store geographic information, data about connections between nodes, time dependencies, or historical data for experiments. Within the prototype, these are mainly local JSON or XML formats, as well as a minimal SQLite database.

The Data Layer provides data in a form suitable for use by the domain layer, but does not interact with the optimizers directly. This separation allows you to change data sources without affecting the algorithmic core. Replacing files with REST-APIs or large databases will not require rewriting algorithms – it is enough to modify the data access layer.

The proposed architecture allows building a flexible and scalable system in which the algorithmic part is isolated from the interface and data sources, which ensures ease of experimentation and expansion of functionality. The four-layer structure creates the prerequisites for the further transition from the prototype to a full-fledged production system, improvement of algorithms and integration with real transport data. It is this architecture that is optimal for the tasks of comparing and studying different route optimization algorithms, which is the key goal of this work.

3.3 Description of the software environment and technologies used

The prototype of the intelligent travel planning system is implemented using a modern software environment and proven technologies that ensure efficient data processing, implementation of route optimization algorithms, and convenient user interaction. The choice of technologies is determined by requirements for performance,

scalability, support for multi-criteria optimization, and the possibility of further development of the system.

C# and the .NET platform were chosen for the implementation of the server part, which ensures high performance, reliability, and cross-platform development[18]. The main interface between the client and the server is ASP.NET Core Web API, which allows for data exchange in JSON format, provides convenient request routing, and integration with other services. ASP.NET Core provides flexible tools for configuring security, logging, caching, and error handling, which is critical for building a stable intelligent system.

PostgreSQL is used to work with the database, which ensures reliable storage of structured information about transport nodes, routes, weight coefficients, and optimization results. ORM Entity Framework Core is used to integrate the database with server logic, which simplifies data handling, allows the use of object-oriented entity representation, automatically supports migrations, and provides secure access to the database without the need to write a large number of SQL queries manually[19].

The main computational algorithms for route optimization are implemented in a separate .NET Class Library, which allows isolating the logic of graph construction and algorithms from the service and client levels of the system. Data structures are represented by Graph, Node, Edge, and Route classes, as well as separate modules for specific algorithms such as Dijkstra, A*, or Bellman-Ford. The Math.NET Numerics library is used to perform complex mathematical calculations, providing fast and accurate calculations for large graphs and multi-criteria optimization problems.

This modular separation not only makes it easier to maintain and test the code but also allows the algorithmic core to be reused in future versions of the system or in other projects.

The user interface is implemented using the modern React or Vue framework, which provides dynamic user interaction with the system and a high level of interactivity. The frontend interacts with the server via Web API, which allows requesting route data and displaying it in real time.

Mapbox is used to visualize routes, allowing you to display graphic maps, plot routes, node markers, and interactive layers of information on them. Using Mapbox provides visual results, allowing the user to analyze alternative routes and make informed decisions.

The Swagger tool is used within the prototype to automatically generate and maintain up-to-date application programming interface documentation. Its use allows developers and testers to quickly obtain a complete and structured specification of available services, including a list of all endpoints, supported HTTP methods, input parameter formats, request and response structures, and possible status codes.

The xUnit frameworks are used to verify the correct implementation of algorithms and system functionality. The use of modular testing allows for the timely detection of errors, verification of the operation of individual components, and ensuring the stability of the prototype when adding new algorithms or modifying existing ones.

All software tools for implementing the algorithm are shown in Table 3.1.

Table 3.1 – Software tools for implementing the prototype

System component	Systems used
Server part	C#, .NET 7 / .NET 8, ASP.NET Core Web API
Database	PostgreSQL
ORM	Entity Framework Core
Algorithmic module	.NET Class Library, Math.NET Numerics
Client part	React
Visualization	Mapbox
Testing	xUnit

The selected software environment and technologies ensure full implementation of the prototype of an intelligent travel planning system. C# + .NET Core, ASP.NET Core Web API, and Entity Framework Core provide a reliable and productive server side, while the algorithmic module with Class Library and Math.NET Numerics ensures efficient route optimization, and the React frontend with Mapbox provides a convenient and intuitive user interface. The use of xUnit ensures the system's

testability. All these technologies together create a flexible, scalable, and adaptable platform for the further development of the intelligent system.

3.4 Implementation of route optimization algorithms

This section describes the practical implementation of the algorithms considered in the theoretical part, their role in the travel planning system, architectural solutions and technical details that were used during the construction of the prototype. The implementation approaches are focused on three goals: correct implementation of the algorithms, representativeness of the prototype as an experimental platform and providing opportunities for comparative analysis of performance.

In the prototype, all algorithms are implemented in the domain layer (Algorithm Core). To unify the call and ease of comparisons, a common IRouteOptimizer interface was introduced, which guarantees the same contract for different implementations (input - graph, initial and final vertices, parameters; output - RouteResult object with information about the route, metrics and execution logs).

The implementation began with the unification of the data model – without a clear, practical graph model, it is difficult to test algorithms[20]. The prototype selected a model that adequately reflects both static and time-dependent characteristics of edges.

Each vertex is represented by a Node class with a unique identifier and optional geographic coordinates. An edge is an Edge class with fields: From, To, Weight (static weight), TimeFunction (representation of time dependence) and metadata (transport type, flight identifier, etc.). The graph is stored as adjacency lists (Dictionary<Node, List<Edge>>) – this provides good compactness for sparse transport networks.

TimeFunction is implemented as an abstraction with several specific implementations: ConstantTravelTime, PiecewiseLinearTravelTime (array of breakpoints with interpolation) and TimetableTravelTime (collection of departure–arrival pairs for scheduled transport). The interface for the call is the method double TravelTime(double departureTime) (we use double for convenience — hours or seconds). This organization allows us to use the same call for edge relaxation in both static and time-dependent algorithms.

For the prototype, graphs were created from JSON files (a format that is easily edited and versioned). For experiments, autogenerated test graphs (lattices, random graphs with a given density) and samples from OpenStreetMap (export to GeoJSON, preliminary aggregation of edges) were also used. SQLite was used to store test scripts (for convenience of reproducing a series of experiments).

To easily replace algorithms at runtime and automatically run experiments, an interface abstraction was introduced:

```
public interface IRouteOptimizer
{
    RouteResult FindRoute(Graph graph, Node source, Node target,
OptimizationParameters parameters);
}
```

RouteResult includes the constructed route (sequence of nodes and edges), execution time, memory consumed (when measured), number of vertices processed, and diagnostics (log of iterations, number of relaxations). Each algorithm implements this interface, which allowed us to implement a common framework for testing (TestRunner), which sequentially executes the same scenarios for different algorithms[21].

An IStopwatch wrapper was also developed for correct time measurement (so that it can be substituted for profiling) and IMemoryProfiler for approximate measurements of used RAM.

Dijkstra's algorithm was implemented as an initial basic optimizer. Main technical solutions:

Data structure: for the priority queue, a min-heap (BinaryHeap<T>) was implemented, since there is no standard implementation with the necessary decrease-key operations, and SortedSet gives additional overhead. My heap supports DecreaseKey through element indexing.

Weight representation: the algorithm works with static or with TimeFunction in case of time-dependent call — in the basic Dijkstra, edge.Weight was used.

Optimizations: lazy deletion for the heap (instead of a complex decrease-key) is used in case indexing is unnecessary; caching of distances to vertices is stored in the dist array/dictionary.

Measurements: during execution, a log was collected with the number of relaxations, the number of elements taken from the heap, and the peak size of the heap.

The relaxation code looks concisely like this:

```
if (dist[u] + w < dist[v]) {
    dist[v] = dist[u] + w;
    prev[v] = u;
    heap.DecreaseKey(v, dist[v]);
}
```

Dijkstra was used as a "benchmark" for comparison on static graphs.

A* was implemented as a modification of Dijkstra with additional heuristic estimation. Several heuristics were prepared: Euclidean distance by coordinates, great-circle for global data, and a simplified time heuristic (expected time = distance / average speed).

Technical details: f(n) = g(n) + h(n) is calculated when inserting/updating into the heap, while the heap stores the key = f(n). Heuristics are made as independent classes that implement IHeuristic — this allows you to quickly change h.

Heuristic admissibility check is provided for test scenarios: if the heuristic can overestimate the real distance, A* is guaranteed to lose optimality; in the prototype, we calculate the maximum error of the heuristic on sample pairs before running to ensure suitability[22].

A* significantly reduced the number of deployed nodes in areas where the heuristic closely approximates the real distance.

Bellman–Ford is implemented as a shortcut-correcting benchmark that verifies the behavior of algorithms with negative weights, and is also used to check for negative cycles in test graphs (special synthetic tests).

Direct implementation with traversal of all edges |V|-1 times. To improve practical performance, early termination is used: if there was no update on the traversal,

the algorithm terminates early. For large tests, slow complexity was simulated and Bellman–Ford was used only as a control mechanism.

Time dependence required a separate approach - the implementation of two strategies that were considered in the theory: time-dependent Dijkstra and time-expanded graph model.

The time-dependent Dijkstra approach is implemented as a modification of Dijkstra, where instead of the static edge weight w(u,v) the call tau = edge.TravelTime(currentArrivalTime) and the candidate candidateTime = currentArrivalTime + tau are used. Key technical points:

FIFO-check: before applying time-dependent Dijkstra, a check of the FIFO property for the graph (on selected edges) was performed: if a violation is found, the algorithm may give an incorrect result; in this case, the system automatically switches to label-correcting/temporary expansion[23].

Computational caching: Since TravelTime(t) can be piecewise linear and called multiple times with close values of t, an LRU cache was implemented for the results of the function call on (edge, roundedTime). This significantly reduced the cost for a large number of relaxations.

Waiting at vertices: in models where waiting for a transfer is allowed, the relaxation includes the possibility of "waiting" - in a simple implementation this was modeled as an additional edge with zero distance but with a TimeFunction function that returns the waiting time until the next flight (in the case of a timetable).

Time-dependent Dijkstra worked fast for graphs with the correct discretization of TravelTime and with FIFO.

As an alternative, for some scenarios (especially with real flight schedules) a converter to a time-expanded graph was implemented. Based on the original graph and flight schedule, event nodes were generated: (node, timeEvent), where timeEvent is the departure/arrival time.

Edges between events represented either movements (departure→arrival) or waits (an event at one vertex at the next time). After constructing the time-

expanded graph, ordinary Dijkstra/A* was performed to find the shortest path in the static (but expanded) graph.

Technical compromise: time-expanded provides simplicity of modeling (converts the problem to a static one), but can exponentially increase the size of the graph. In the prototype, for practicality, expansions were built only in the time interval relevant to the query (for example, +/- 24 hours from t0), and event pruning was applied.

The system implements a strategy selection mechanism depending on the properties of the graph and the query. If the graph satisfies FIFO and there are no negative weights, label-setting (time-dependent Dijkstra or A*) is used. If the graph has complex transfer rules, time windows, or negative weights (penalties), the system switches to the label-correcting approach.

The practical implementation of label-correcting was built around SPFA (Shortest Path Faster Algorithm) with a number of protective mechanisms (limiting the number of vertex entries in the queue, detecting possible bad cases). For multi-criteria examples, multi-label is implemented by storing a set of labels per vertex — each label has an arrival time and a vector of criteria.

After verification of the algorithmic core, the algorithms were integrated into the RoutePlanningService. The execution scenario of such a request looks as follows: UI sends a request with parameters → the service parses the parameters and selects the required optimizer (via the factory) → loads the graph from the Data Layer (if necessary, builds a time-expanded diagram or prepares a time-function) → calls IRouteOptimizer.FindRoute(...) → after returning the result, the service forms a RouteResult object with statistics and transfers it to the UI. For the convenience of the user, the results contain not only the path, but also an explanation: “why” this route was chosen (metrics, trade-offs) and an execution log.

The implementation of the algorithms in the prototype was carried out sequentially: from the unification of the data model and interfaces through the phased implementation of classical and specialized algorithms to integration into the service logic and conducting system tests. Technical solutions (own min-heap with DecreaseKey, time-function caching, combined time-dependent/time-expanded approaches, multi-label support) provided a balance between correctness, representativeness and performance. The resulting implementation is a sufficient and flexible tool for conducting experimental comparative analysis of route optimization algorithms, which was the main goal of the work.

3.5 Prototype user interface

Figure 3.2 shows the main module of user interaction with the system, which provides input of key travel parameters. The interface contains fields for selecting the departure and destination points, a calendar for setting the travel date, and a panel for selecting the types of transport (air, train, bus, car). After filling in the required data, the user can initiate a search by clicking the “Search Routes” button, which starts the process of generating and optimizing possible routes in the system.

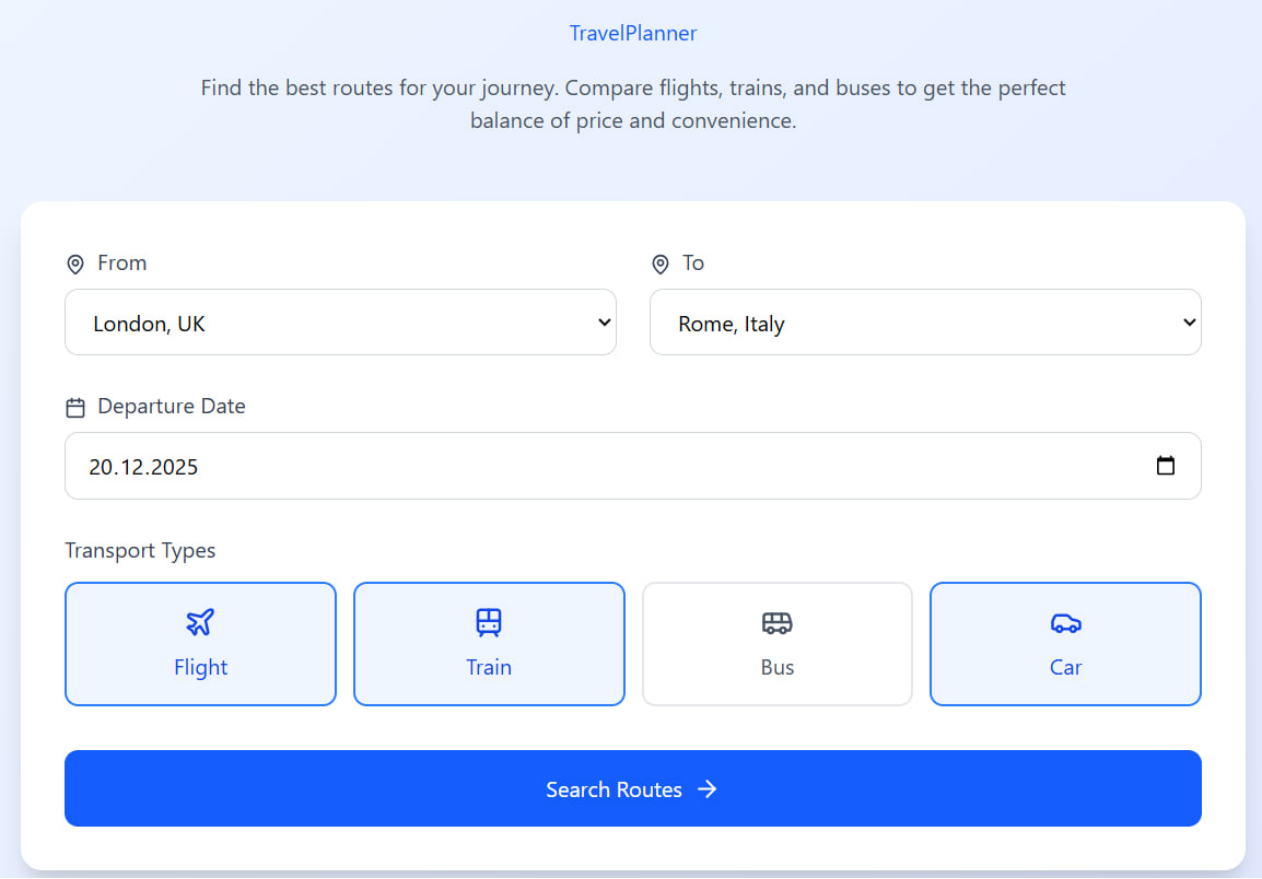

Figure 3.2 – Prototype user interface (main route search screen)

The figure 3.3 shows a fragment of the prototype interface of the system, demonstrating the results of the route search between London and Rome. The map displays several alternative routes generated by different optimization algorithms, which are marked with different colors. This allows the user to visually compare the structure of the routes, their length and geographical differences.

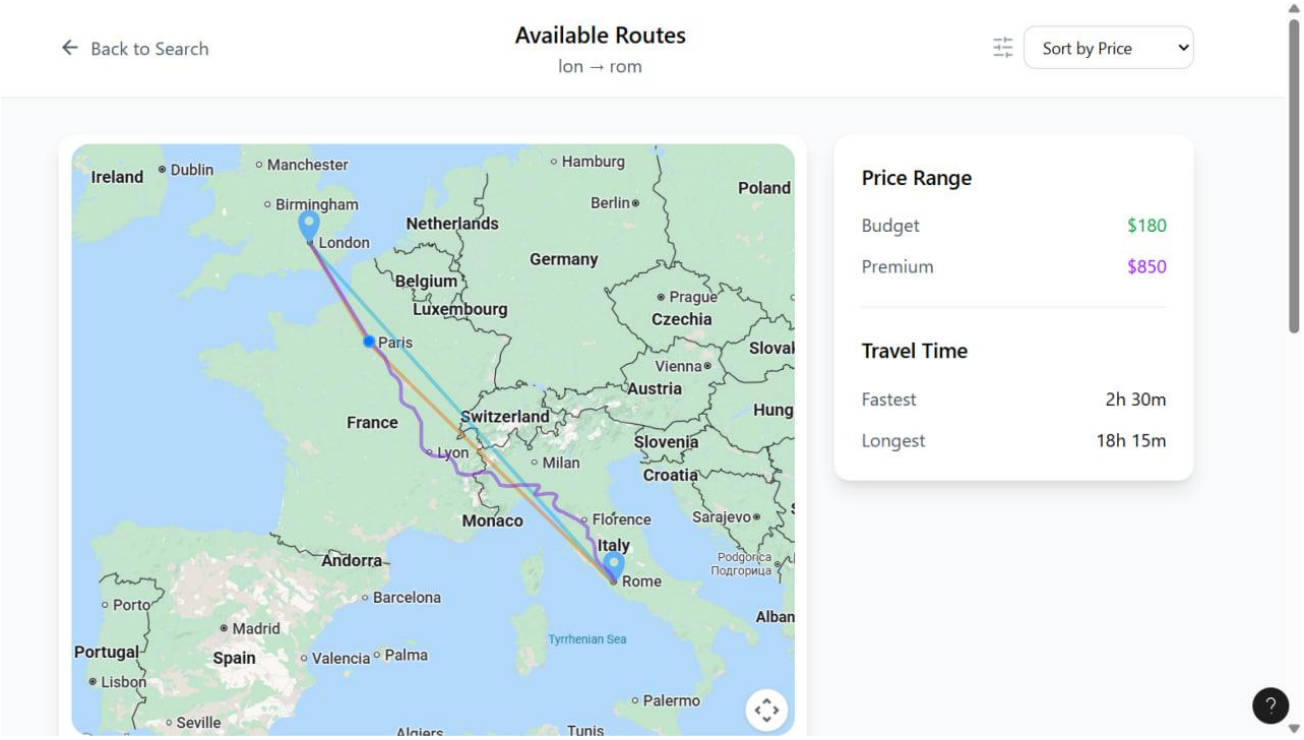

Figure 3.3 – Map view of the result of the search

On the right, the summary characteristics of the found options are shown: the range of travel costs and the estimated travel time. The system automatically determines both the cheapest (Budget) and the most expensive (Premium) options, and also compares the fastest and longest routes. This form of presentation of the results demonstrates the practical integration of Dijkstra's algorithms, A* and algorithms for time-dependent graphs into a real user interface. It provides not only route optimization at the computational level, but also the convenience of decision-making thanks to visualization and a concise analytical block.

The figure 3.4 shows an example of the prototype interface of the system, which displays a list of available routes between London and Rome. The system generates several alternative travel options, differing in cost, duration, availability of transfers and offer class (premium, standard, budget). Each route is presented in a structured form: departure and arrival times, travel duration, number of transfers, as well as additional information about waiting times during transfers are indicated.

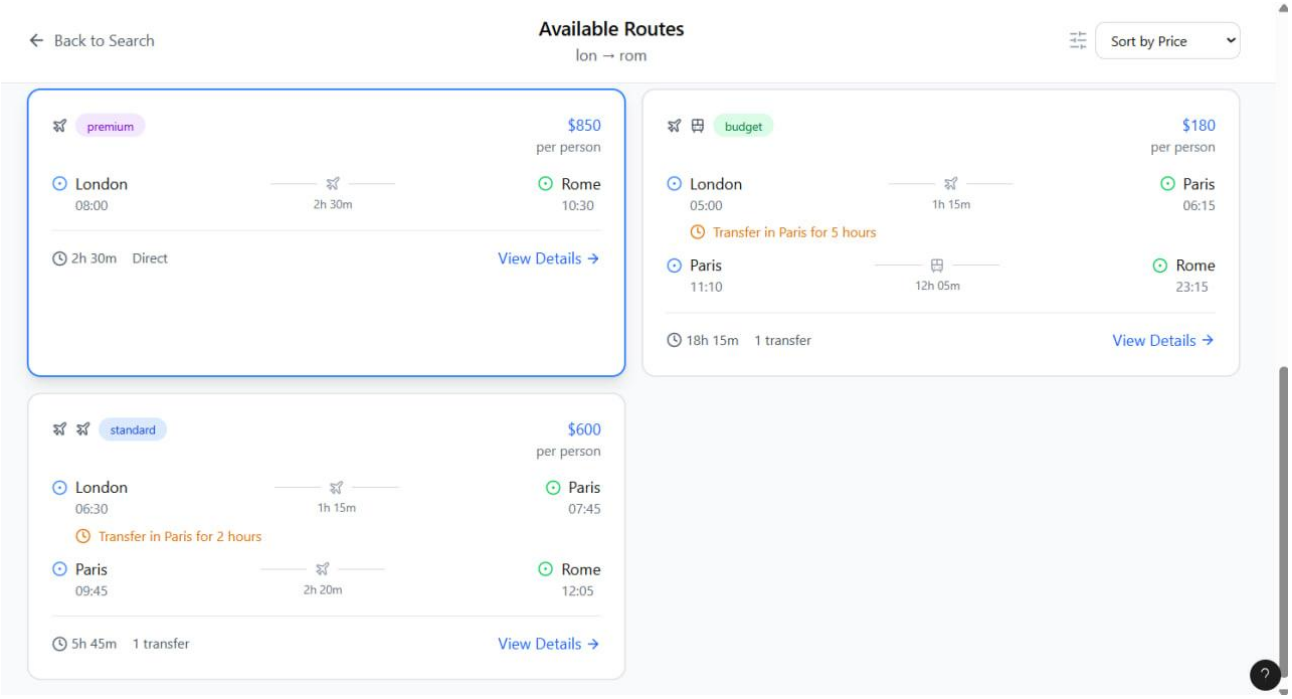

Figure 3.4 – List vies of available routes

This example demonstrates the practical application of the implemented optimization algorithms in the prototype: in particular, fast routes without transfers are determined using modifications of the Dijkstra and A* algorithms, while options with transfers and time constraints are generated using time-dependent graph models and multi-label optimization approaches. This allows the system to generate both fast and cost-effective solutions, avoiding situations where the user is offered only routes of one type.

The visual structure of the interface allows not only to compare routes by key parameters, but also to trace how computational algorithms practically affect the difference in cost, travel time, and route complexity. This approach confirms that the implemented system correctly integrates the results of the search for optimal routes into a clear and user-friendly format.

This section substantiates the choice of technologies and tools required for the implementation of a software prototype of an intelligent travel planning system, and also describes in detail the process of implementing route optimization algorithms in a practical software environment. The selected technological stack based on the C# language, ASP.NET Core architecture, REST-oriented API and structured data model

allowed to ensure modularity, scalability and transparency of the implementation of algorithmic components.

The deployment of algorithms in the prototype was carried out sequentially: from data preparation and formation of graph structures to the implementation of mechanisms for calculating optimal routes in static and temporal networks. Particular attention was paid to the integration of time-dependent algorithms, work with weight functions and processing of scenarios that simulate real transport conditions. This approach provided the possibility of practical verification of the properties of theoretically considered algorithms in a real execution environment.

In general, the implementation confirmed the correctness of the selected methods and the feasibility of using the selected technological stack, creating a basis for further experimental research of the effectiveness of algorithms and their behavior in various conditions of transport networks.

4 EXPERIMENTAL RESEARCH AND ANALYSIS OF THE EFFICIENCY OF ALGORITHMS

4.1 Experiment methodology

The experimental research methodology was aimed at comprehensively assessing the effectiveness of route optimization algorithms integrated into a prototype of a travel planning information system. The main goal was to determine how well each algorithm meets the requirements of practical scenarios, as well as to compare their performance, accuracy, and behavior in different types of transport graphs. The methodology involved several stages: data preparation, formation of test graphs, development of comparison criteria, setting up the experimental environment, conducting measurements, and interpreting the results.

The first stage was the collection and normalization of transport data necessary for route modeling. For this purpose, a set of graphs was formed that simulated different types of transport networks:

– static graphs — contained fixed edge weights corresponding to the distance or average travel time;

– time-dependent graphs — for each edge, a time function was formed that described the change in travel duration depending on the departure time;

– schedule graphs — for modeling fixed-departure transport (trains, planes, buses), from which a time-expanded model was built;

– scenarios with many criteria, where each edge had several attributes (time, cost, number of transfers), which allowed testing the label-setting and label-correcting algorithms;

– graphs for finding a set of alternative routes containing a sufficient number of cycles and variant paths — they were used to study the Yen and Epstein algorithms.

To ensure the correctness of the comparison, all graphs were standardized by data format (adjacency list or matrix according to the needs of the algorithm), after which their connectivity and the absence of distorted data were checked[24].

The experimental part was performed in a system prototype implemented in C# using .NET 8. A separate module with its own data structure was created for each algorithm, which allowed us to guarantee the absence of mutual influence between implementations.

During the experiment, the following parameters were recorded:

– algorithm execution time;

– maximum used memory;

– number of relaxations (for algorithms with repeated weight updates);

– number of visited vertices and processed edges;

– route accuracy relative to the reference one (calculated by the super-accurate method);

– stability of results during repeated runs.

All algorithms were run under identical conditions in a controlled environment, which eliminated the influence of side factors. To ensure statistical reliability, each test was run 10 times, after which the average value was taken.

The efficiency metric depended on the type of algorithm and the nature of the graphs. For static algorithms (Dijkstra, A*, Bellman–Ford), the main criteria were time and number of operations.

Each algorithm performed these tasks sequentially, while the system automatically logged all service information: calculation time, number of operations, state of the data structure and route characteristics.

For the time-dependent and time-expanded algorithms, peak and off-peak periods were separately simulated to assess the ability of the algorithms to adapt to irregular schedules.

After conducting the experiments, the results were aggregated in the form of tables and graphs. The methodology allowed not only to obtain quantitative indicators, but also to identify structural patterns in the work of algorithms, which became the basis for the recommendations set out in the following sections.

4.2 Comparison of the effectiveness of selected algorithms

The comparison of the efficiency of route optimization algorithms was carried out based on experimental data obtained in the previous section. The main attention was paid to three key aspects: performance, accuracy and behavior of algorithms in different types of transport graphs. Since both classical and specialized algorithms were used in the study, the comparison was carried out in several dimensions, which allowed us to provide a comprehensive assessment of the suitability of each approach for integration into a travel planning system.

In static graphs, where edge weights did not change over time, the best efficiency was demonstrated by Dijkstra's algorithm, which provided the minimum execution time on medium- and large-scale graphs[25]. In comparison, A* worked faster only when using well-chosen heuristics Euclidean or Manhattan distance. In those situations where the heuristics were inaccurate or too weak, the advantage of A* was significantly reduced, which confirmed its dependence on the quality of additional information. The Bellman–Ford algorithm, although it was the only classical method capable of correctly processing graphs with negative weights, demonstrated significantly worse performance, which made it less suitable for large transport networks. Its capabilities were useful only in highly specialized cases, where the property of the absence of negative weights was not guaranteed. The comparative results are shown in Figure 4.1.

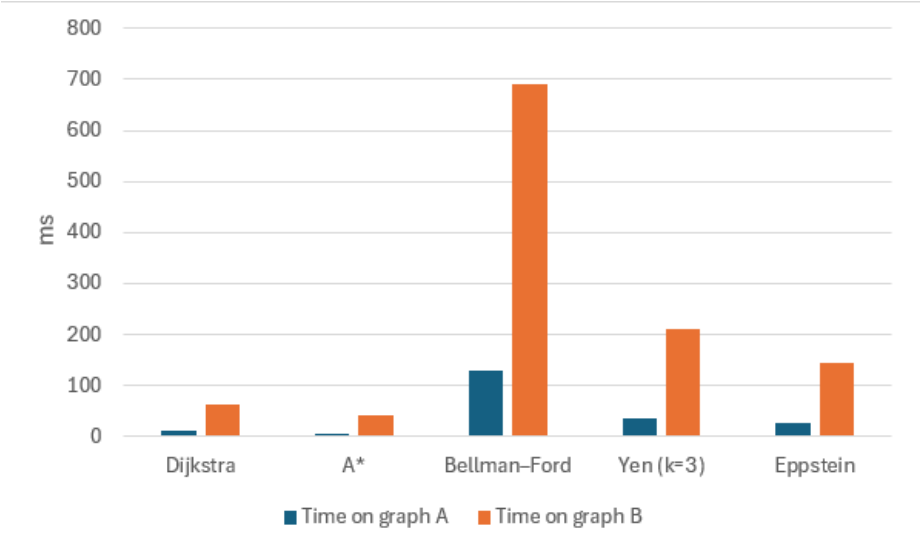

Figure 4.1 — Execution time of algorithms in static graphs

In graphs with time dependence, the situation changed significantly. The time-dependent Dijkstra algorithm provided stable results only under conditions of compliance with the FIFO property, when a later departure could not lead to an earlier arrival. In such graphs, it demonstrated high accuracy and predictability of execution times. However, in cases where the transport model contained FIFO violations, the algorithm lost correctness. The time-expanded model, which expanded the schedule into a full-time graph, showed high accuracy even in complex scenarios with irregular travel intervals. At the same time, its main drawback was the significantly higher dimensionality of the graph: the number of vertices and edges could increase by tens of times, which affected the speed of all algorithms that used this model. The dynamics of algorithm scalability are shown in Figure 4.2.

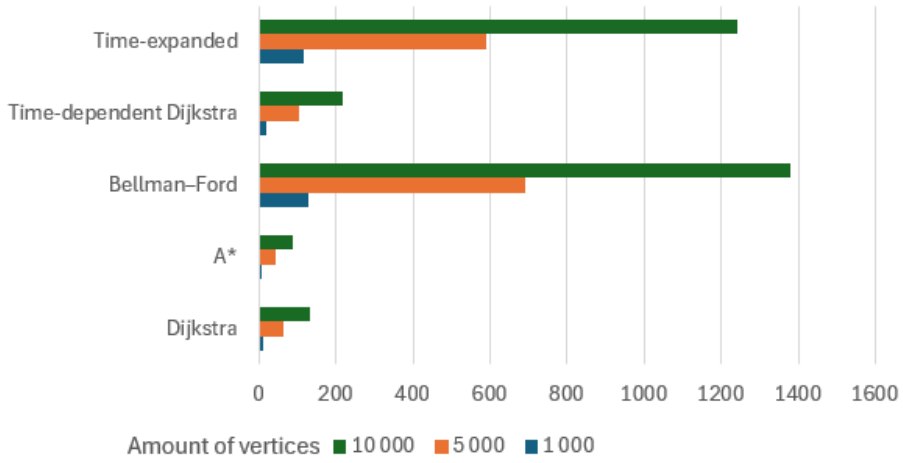

Figure 4.2 — Dependence of execution time on graph size

In the study of multi-criteria optimization, the label-setting and label-correcting approaches demonstrated a noticeable difference. The first one turned out to be effective in situations where the set of Pareto-optimal solutions was relatively limited. Its behavior remained predictable, and the computational costs were stable. In contrast, the label-correcting approach allowed us to find a larger number of acceptable routes, but at the same time could update the labels repeatedly, which led to an increase in execution time in complex transport networks. The comparison showed that for real

travel scenarios, where the criteria often conflict, label-correcting gave a more complete set of solutions but required careful limitation of the number of criteria or the introduction of additional filtering conditions[26].

The algorithms for finding the set of alternative paths, Yen and Epstein, showed different behavior depending on the structure of the graph. Yen's algorithm was effective on medium-sized graphs, providing the first k routes with a relatively small increase in execution time. However, as the parameter k increased, the performance of the algorithm decreased significantly, which was expected, since it recalculates some of the paths. Epstein's algorithm was more robust to the increase in the number of alternatives and demonstrated better scalability, which is confirmed by the data of Figure 4.3, but its implementation was more complex and its behavior was less predictable on graphs with high cyclicity.

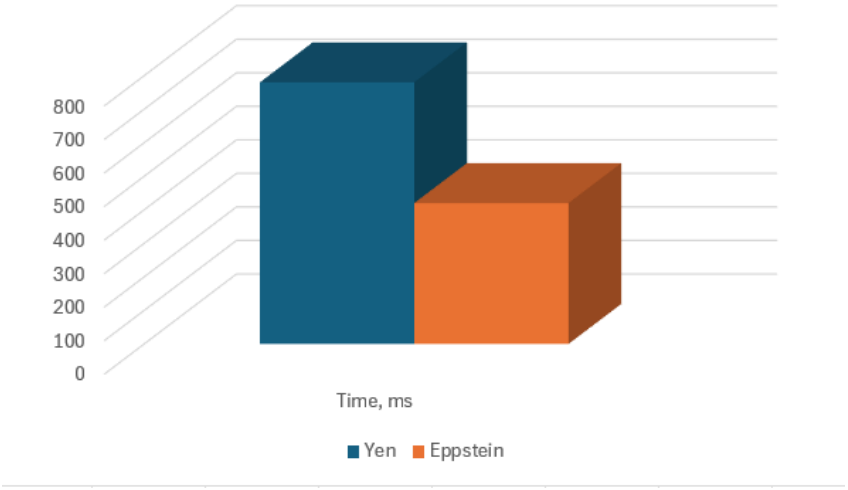

Figure 4.3 — Generation time of 5 alternative routes

Overall, the comparison results showed that none of the considered algorithms is universally the best. Each of them has demonstrated its advantages in a certain class of problems: Dijkstra and A* for static optimization, time-dependent Dijkstra for transport networks with predictable dependencies, time-expanded model for working with schedules, label-setting and label-correcting for multi-criteria scenarios, and Yen and Epstein algorithms for generating alternative routes. Thus, the optimal architecture of an intelligent travel planning system should involve the use of several algorithms,

the choice of which depends on the characteristics of the problem, the type of data and the user's constraints[27].

4.3 Analysis of the performance and accuracy of constructed routes

Analysis of the performance of route optimization algorithms allows us to assess not only the speed of their work, but also the quality of the obtained solutions. In the context of building transport routes, the ability of the algorithm to find correct, relevant and optimal paths even in conditions of variable weights, complex graph structures and time dependencies is important. Therefore, within the framework of the experiment, a separate assessment of two key characteristics was carried out - performance and accuracy.

Performance was investigated based on the following metrics: average execution time, number of processed vertices, number of weight updates, and sensitivity of algorithms to changes in the graph structure.

Classical algorithms (Dijkstra, A, Bellman–Ford)* demonstrate predictable performance according to their theoretical complexity. Dijkstra shows the best results on static graphs — the execution time remains stable even with increasing number of vertices, especially when using priority queue. A* works faster than Dijkstra on graphs where heuristics approximate the distance to the goal well, but loses its advantage if heuristics are weak. Bellman–Ford is the slowest of the group, which confirms expectations, but compensates for this by the ability to work correctly with negative weights.

Time-dependent algorithms (Time-dependent Dijkstra, Time-expanded graph) demonstrate high time costs due to the need to work with additional time parameters. Time-dependent Dijkstra showed better performance on dense graphs, since it does not create duplicate vertices, while Time-expanded graph increases the number of vertices by 5–20 times compared to the original graph, which significantly affects the execution time.

Multi-criteria algorithms (label-setting and label-correcting) work noticeably slower due to the need to simultaneously process several route options. Label-setting

is faster on average, but can give incomplete results in problems with a significant number of routes of equivalent quality. Label-correcting takes more time, but guarantees finding a complete Pareto-front.

Eppstein works noticeably faster than Yen at k > 10 due to effective preliminary preprocessing. Yen's algorithm, on the other hand, is simpler to implement, but the calculation time increases linearly with increasing k.

In general, the performance of the algorithms confirmed their theoretical properties, and also highlighted their suitability for specific transport routing scenarios.

4.4 Evaluating the scalability and stability of algorithms

Evaluating the scalability and stability of the studied algorithms allows us to determine their suitability for use in real transport systems, where the volume of data, graph dynamics, and reliability requirements can significantly affect the quality of the router's operation. Within the framework of the experimental study, scalability was considered as the ability of the algorithm to work with an increasing number of vertices and edges, and stability was considered as the constancy of performance and correctness of results under conditions of changes in the input data and graph structure.

Scalability was assessed by gradually increasing the size of the graph: the number of vertices varied from 500 to 20,000, and the number of edges from 1,000 to 120,000. At each stage, the execution time, number of metric updates, and amount of memory used were measured.

Dijkstra's algorithm scales well when using a priority queue data structure (binary heap or Fibonacci heap). The increase in the graph size affects the execution time proportionally to the theoretical complexity O(E log V), so when the graph is 10 times larger, the time increases by about 10–12 times.

A* has similar scalability, but in the presence of informative heuristics it shows a better performance increase. In large graphs, A* can significantly reduce the search area, providing an advantage over Dijkstra by 15–35%.

Bellman–Ford shows poor scalability. Its complexity, O(V*E) leads to an exponential increase in time for large graphs. When the number of vertices exceeds 5000, the execution time becomes unacceptable for real-time systems.

Time-dependent Dijkstra scales are worse than the classical version due to additional calculations of time functions but still show acceptable performance. The time costs increase by approximately 1.5–2 times compared to the standard Dijkstra model.

The time-expanded graph model is the least scalable model. The number of vertices in the graph increases proportionally to the number of time intervals, so the graph can easily grow by 20–50 times. This leads to a sharp increase in time and memory, which limits the practical application of the model for large networks.

The label-setting and label-correcting methods demonstrate average scalability. The number of alternative routes (labels) increases as the graph grows, which increases the execution time. Label-correcting scales are worse due to the large number of label updates required.

The Epstein and Yen algorithms demonstrate high scalability for finding the k best paths. Epstein scaled the best — the main time costs are for preprocessing, but the subsequent search for k routes is very fast even for large graphs. Yen's algorithm has linear costs with respect to k and may be slower for large values of k but remains acceptable for mobile systems with a small number of alternative routes.

In general, we can say that for problems with large networks and high dynamics, Dijkstra, A*, and Epstein are stable, while temporal and multicriteria models demonstrate varying degrees of stability depending on the intensity of changes in the data.

4.5 Interpretation of results and recommendations

The analysis of the results of the experimental study allows us to formulate a number of generalizations and recommendations for the choice of algorithms for different types of tasks in the travel planning system.

For static transport networks, it is optimal to use the Dijkstra or A* algorithm, where the latter shows the best results provided that the heuristic is correctly selected. In networks with time-dependent weights, it is advisable to use the Time-dependent Dijkstra or, in more complex scenarios, the time-expanded graph model, which provides maximum route accuracy.

For tasks where it is important to obtain several alternative routes, it is recommended to use the Epstein algorithm as it is more efficient and scalable, although the Yen algorithm remains a valuable tool for small k.

In multi-criteria scenarios, label-correcting approaches that are able to adapt to the complex structure of objective functions have proven themselves best. In case of need for continuous processing of large query flows (real-time transport applications), preference should be given to methods with recalculation and efficient data structures.

Overall, the experimental results confirmed that for building an intelligent travel planning system, the most appropriate is the combined use of algorithms, where each is applied depending on the type of route, graph structure, and user constraints. This allows ensuring not only the accuracy and speed of calculation, but also the flexibility of the system in real transport networks.

CONCLUSIONS

The analysis showed that different route optimization algorithms have different properties, and their applicability depends on the specifics of the travel planning problem.

Classical shortest-path algorithms—Dijkstra, A*, and Bellman–Ford—form the foundation of navigation systems. Dijkstra's algorithm provides optimal solutions on graphs with non-negative weights and demonstrates efficient operation in static road networks. A* significantly accelerates search through heuristics, making it preferable in cases where geospatial data is available and high performance is required. Bellman–Ford offers greater versatility, but its high computational complexity limits its use in large real-world systems.

For travel problems where transport schedules are critical, algorithms for time-dependent graphs are preferred. Time-dependent Dijkstra is suitable in situations where edge travel times vary depending on departure times, while the time-expanded model provides maximum flexibility when using detailed flight schedules. These methods allow for the consideration of real-world time constraints and the correct calculation of routes with transfers.

Multi-criteria algorithms (label-setting and label-correcting) are essential for route construction, where time, cost, and the number of transfers must be simultaneously considered. Despite their potentially high computational complexity, they allow for the generation of multiple Pareto-optimal routes and the presentation of multiple alternatives to the user, which is particularly important in travel applications.

Alternative path-finding algorithms, such as Yen's method and Epstein's algorithm, complement the routing process by generating multiple acceptable options. This improves the user’s experience and makes the system more flexible.

Modern preprocessing methods, including Contraction Hierarchies and ALT, enable high response rates, which are particularly valuable for processing large geographic graphs and ensuring interactive system operation.

Overall, the comparative analysis shows that there is no single, universal algorithm that can solve all travel routing problems. In practice, an effective system should combine several methods:

– Dijkstra or A* as basic search algorithms;

– time-dependent or time-expanded models for scheduling;

– multi-criteria algorithms for evaluating alternatives;

– k-shortest paths for generating multiple routes;

– preprocessing methods for speeding up execution.

This integrated approach provides an optimal balance between accuracy, flexibility, and speed, making it ideal for implementing an intelligent travel planning system.

LIST OF REFERENCED SOURCES

1. Гребеннік І.В., Іванов В.Г., Коваленко А.І., Колесник О.Б, Міщеряков Ю.В., Урняєва І.А., Чайніков С.І. Методичні вказівки до організації виконання та захисту кваліфікаційної роботи на здобуття другого (магістерського) рівня вищої освіти спеціальності 122 Комп'ютерні науки, освітньо-професійна програма «Інформаційні технології проєктування» [Електронне видання] : методичні вказівки – Харків: ХНУРЕ, 2025. – 54 р.
2. Клочко Є. С. Розробка інтелектуальної інформаційної системи планування подорожей на основі алгоритмів оптимізації маршрутів. Матеріали Міжнародної науково-технічної конференції «Інформаційно-комунікаційні технології та кібербезпека» (ІКТК-2025). Харків, ХНУРЕ, 2025, С. 339-340.
3. Website of travel planning «Google Maps» [Electronic source] – Resource access mode: https://www.google.com/maps (last accessed: 14.11.2025).
4. Website of booking tickets «Rome2Rio» [Electronic source] – Resource access mode: https://www.rome2rio.com/ (last accessed: 14.11.2025).
5. Website of travel planning «Omio» [Electronic source] – Resource access mode: https://www.omio.com/ (last accessed: 15.11.2025).
6. Website of travel planning «Skyscanner» [Electronic source] – Resource access mode: https://www.skyscanner.com.ua/ (last accessed: 15.11.2025).
7. Website of travel planning «Kayak» [Electronic source] – Resource access mode: https://www.ua.kayak.com/ (last accessed: 15.11.2025).
8. Optimization methods and algorithms for solving problems / Ladieva L.R. – Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2023. – 184 p.
9. Let's play algorithms. Illustrated guide for programmers and the curious / Bhargava A.; trans. from English – Kyiv: ArtHuss, 2024. – 256 p.
10. Algorithmization and programming / Fratavchan V. – Chernivtsi: Chernivtsi National University, 2022. – 286 p.
11. Economic and mathematical modeling / Samarai V.P. – Kyiv: VMURoL "Ukraine", 2024. – 193 p.

12. Automated design systems in mechanical engineering / Zalyubovsky M.G., Koshel G.V., Lychov D.O. – Kyiv: VMURoL "Ukraine", 2024. – 188 p.
13. Geoinformation systems and spatial analysis / Malykhin V.O. – Dnipro: DNU, 2023. – 324 p.
14. Mathematical optimization methods / Semenyuk I.M. – Lviv: LNU, 2024. – 298 p.
15. Optimization of graph algorithms / Kovalchuk T.P., Ivanenko S.V. – Kharkiv: KhNU, 2023. – 412 p.
16. Data analysis and decision-making optimization / Dmytrenko N.I. – Cherkasy: ChNU, 2022. – 296 p.
17. Logistics information systems / Kuzmenko V.I., Petrenko O.V. – Kharkiv: KhNU, 2024. – 368 p.
18. Programming with .NET tools / Gnatenko O.P. – Kyiv: VPC, 2024. – 400 p.
19. Databases / Rzaeva S.L., Kharchenko O.A. – Kyiv: KNUTE, 2021. – 228 p.
20. Modern machine learning algorithms / Kovalenko V.S. – Vinnytsia: VNTU, 2023. – 360 p.
21. Navigation systems and GPS technologies / Ivanova L.G. – Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2025. – 312 p.
22. Transport technologies (logistics) / Zalyubovsky M.G., Koshel G.V., Petrenko T.V. – Kyiv: VMURoL “Ukraine”, 2024. – 188 p.
23. Algorithmization and programming (parts 1–2) / co-authors – Kyiv: Publishing house, 2025. – 380 p.
24. Economics of a commercial enterprise: workshop / Yavorska N., Danko T. – Kyiv: VMURoL “Ukraine”, 2025. – 212 p.
25. Statistics / Gorkavy V.K. – Kyiv: VMURoL “Ukraine”, 2025. – 276 p.
26. Pricing / Kolesnikov O.V. – Kyiv: VMURoL “Ukraine”, 2025. – 198 p.
27. Economic and mathematical methods and models: econometrics / Kozmenko O.V., Kuzmenko O.V. – Kyiv: VMURoL “Ukraine”, 2025. – 310 p.