***Оглавление***

*Оглавление\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2*

*Введение\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3*

*Глава 1. Теоретическое введение в реализацию алгоритмов поиска пути в графе\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4*

*1.1. Описание задачи\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4*

*1.2. Алгоритм Дейкстры\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 5*

*1.3. Алгоритм Беллмана-Форда\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 6*

*1.4. Алгоритм Флойда\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 8*

*1.5. Алгоритм А\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 10*

*1.6. Алгоритм поиска в глубину (DFS)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 12*

*1.7. Сравнение алгоритмов\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 14*

*1.8. Вывод\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 15*

*Глава 2. Практическая реализация алгоритмов поиска пути в*

*графе \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 17*

*2.1. Общие сведения по реализации\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 17*

*2.2. Общие сведения о графе \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 18*

*2.3. Описание методов\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 21*

*2.4. Пример запуска программы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 28*

*Заключение\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 37*

*Список литературы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 39*

*Приложение\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 40*

***Введение***

Глава 1 вводит читателя в тему реализации алгоритмов поиска путей в графе. В данной работе рассматриваются основные алгоритмы, используемые для реализации поиска пути в графе, такие как алгоритм Дейкстры, алгоритм Беллмана-Форда, алгоритм Флойда, алгоритм А\* и алгоритм поиска в глубину (DFS). В конце главы приводится сравнение данных алгоритмов.

В главе 2 представлена практическая реализация алгоритмов поиска пути в графе и описание используемых функций. Также описаны общие сведения о проекте.

Каждый алгоритм стоит применять в зависимости от условий задачи. Если необходимо найти кратчайший путь по определённым параметрам, то необходимы алгоритм Дейкстры, алгоритм Беллмана-Форда(показывают наилучшую скорость выполнения) и алгоритм Флойда. Если необходимо работать с графом, где присутствуют отрицательные веса, то под эти задачи подойдет алгоритм Беллмана-Форда, который способен обрабатывать отрицательные циклы.

**Глава 1. *Теоретическое введение в реализацию алгоритмов поиска пути в графе***

**1.1. Описание задачи**

Алгоритмы поиска пути в графе используются для нахождения оптимального пути между вершинами в графе. Одним из наиболее известных алгоритмов является алгоритм Дейкстры, который находит кратчайший путь от одной вершины к остальным взвешенного графа. Также существуют алгоритмы поиска в глубину и поиска в ширину, которые используются для обхода графа и нахождения пути от начальной вершины к целевой. Реализация этих алгоритмов требует понимания структуры графа и выбора подходящего метода в зависимости от конкретной задачи. В данной задачи, мы будем реализовывать алгоритм Дейкстры, алгоритм Беллмана-Форда, алгоритм Флойда, алгоритм А\* и алгоритм поиска в глубину(DFS).

**1.2. Алгоритм Дейкстры**

**Алгоритм Дейкстры** – это алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Алгоритм находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

**Суть алгоритм Дейкстры** заключается в классическом методе поиска кратчайшего пути во взвешенном графе. Его цель — найти кратчайший путь от начальной вершины к каждой другой вершине в графе. Он основывается на жадном принципе выбора: на каждом шаге алгоритм выбирает вершину с наименьшей известной длиной пути и обновляет информацию о расстоянии до соседних вершин.

**Из ключевых особенностей** алгоритма Дейкстры можно выделить:

* **Жадность:** Основной принцип работы алгоритма - на каждом шаге выбирается вершина с наименьшей известной длиной пути. Этот жадный подход позволяет находить локально оптимальные решения на каждом этапе, что в итоге приводит к глобально оптимальному решению.
* **Неотрицательные веса:** Алгоритм Дейкстры предназначен для графов без рёбер отрицательного веса. Это связано с тем, что в процессе выполнения алгоритма, если бы существовали рёбра с отрицательным весом, то возникли бы проблемы с выбором кратчайшего пути.
* **Посещение вершин один раз:** Алгоритм Дейкстры гарантирует, что каждая вершина будет посещена только один раз. Это достигается путём обновления расстояний только в том случае, если найден более короткий путь.
* **Построение кратчайших путей:** Помимо нахождения длин кратчайших путей, алгоритм Дейкстры может сохранять информацию о предыдущих вершинах, что позволяет восстановить кратчайший путь от начальной вершины до любой другой.

**Алгоритм Дейкстры** является важным инструментом для решения задач маршрутизации, оптимизации планов путешествий и других задач, связанных с нахождением кратчайших путей в сетевых структурах.

**1.3. Алгоритм Беллмана-Форда**

**Алгоритм Беллмана-Форда** – это алгоритм поиска кратчайших путей в графе, который может работать с графами, содержащими рёбра с отрицательным весом. Название алгоритма связано с именами математика и инженера-электрика Ричарда Беллмана и математика Рудольфа Форда, которые впервые описали его в 1958 году.

**Основная цель** алгоритма Беллмана-Форда – нахождение кратчайших путей от одной начальной вершины ко всем остальным. Алгоритм подходит для графов с отрицательными весами, но при этом не допускает наличие отрицательных циклов. Отрицательный цикл – это цикл в графе, в котором сумма весов рёбер отрицательна.

**Принцип работы** алгоритма Беллмана-Форда следующий:

1. **Инициализация:** Устанавливаются начальные расстояния до всех вершин, кроме начальной, в бесконечность, а расстояние до начальной вершины устанавливается в 0.
2. **Релаксация рёбер:** Повторяем **|V| - 1** раз, где **|V|** - количество вершин в графе. На каждом шаге рассматриваем все рёбра графа и пытаемся улучшить расстояние до вершин.
3. **Проверка наличия отрицательных циклов:** После **|V| - 1** итерации проверяем, есть ли отрицательные циклы. Если расстояния до каких-либо вершин уменьшились на этом шаге, то граф содержит отрицательный цикл, и алгоритм не может быть применен.
4. **Восстановление путей:** Если отрицательных циклов нет, то можно восстановить кратчайшие пути до всех вершин.

**Из особенностей** алгоритма Беллмана-Форда можно выделить:

- **Способность работать с отрицательными весами:** Алгоритм позволяет находить кратчайшие пути в графах, где могут присутствовать рёбра с отрицательными весами.

- **Проверка наличия отрицательных циклов:** После выполнения **|V| - 1** итерации, алгоритм проверяет наличие отрицательных циклов в графе.

- **Временная сложность:** Алгоритм имеет временную сложность **O(|V| \* |E|)**, где **|V|** - количество вершин, **|E|** - количество рёбер в графе.

**Алгоритм Беллмана-Форда** широко применяется в сетевых технологиях, транспортной логистике и других областях, где необходимо учитывать возможность существования отрицательных весов рёбер в графах.

**1.4. Алгоритм Флойда**

**Алгоритм Флойда -** это алгоритм для нахождения кратчайших путей между всеми парами вершин в ориентированном графе. Этот алгоритм был предложен в 1962 году американским математиком Робертом Флойдом.

**Основная цель** алгоритма Флойда – найти кратчайшие пути между каждой парой вершин в графе. В отличие от алгоритма Дейкстры и алгоритма Беллмана-Форда, которые решают задачу нахождения кратчайших путей от одной вершины ко всем остальным, алгоритм Флойда решает задачу между всеми парами вершин.

**Принцип работы** алгоритма Флойда:

1. **Инициализация:** Создается матрица расстояний, где элемент **(i, j)** представляет собой длину кратчайшего пути между вершинами **i** и **j**. На этапе инициализации, если существует ребро между вершинами **i** и **j**, то **(i, j)** устанавливается равным весу этого ребра; если ребра нет, то **(i, j)** устанавливается в бесконечность.
2. **Обновление матрицы:** Для каждой вершины **k** от **1** до **N** (где **N** - общее количество вершин), а также для каждой пары вершин **i, j,** проверяется, является ли путь от **i** до **j** короче, если пройти через вершину **k**. Если это так, то матрица обновляется: **(i, j)** = **min((i, j), (i, k) + (k, j))**.
3. **Построение кратчайших путей:** После завершения выполнения алгоритма, матрица расстояний содержит кратчайшие пути между всеми парами вершин.

**Из особенностей** алгоритма Флойда можно выделить:

* **Универсальность:** Алгоритм решает задачу поиска кратчайших путей между всеми парами вершин в графе.
* **Обработка отрицательных весов:** Алгоритм Флойда может работать с графами, содержащими рёбра с отрицательным весом, но не допускает отрицательных циклов, так как они могут привести к неопределенным результатам.
* **Временная сложность:** Временная сложность алгоритма Флойда составляет **O(N^3)**, где **N** - количество вершин в графе. Это делает его менее эффективным для больших графов по сравнению с алгоритмами Дейкстры и Беллмана-Форда, но пригодным для относительно небольших графов.

**Алгоритм Флойда** применяется в сетевых технологиях, где необходимо заранее вычислить кратчайшие пути между всеми парами узлов для оптимизации маршрутизации и других ситуациях, где важна информация о кратчайших путях между всеми вершинами графа.

**1.5. Алгоритм А\***

**Алгоритм A\*** представляет собой эффективный алгоритм поиска пути, используемый для нахождения кратчайшего пути от начальной точки к целевой точке в графе. Этот алгоритм широко применяется в робототехнике, компьютерных играх, геоинформационных системах и других областях, где требуется нахождение оптимального маршрута.

**Принцип работы** алгоритма A\* основан на комбинации эвристической оценки (предположения) и стоимости уже пройденного пути. Алгоритм поддерживает поиск пути в графе с весами и ориентированными рёбрами. Он использует две функции для каждой вершины:

1. **g(n)**: Стоимость пути от начальной вершины до вершины **n**.
2. **h(n)**: Эвристическая оценка стоимости пути от вершины n до целевой вершины (это предполагаемое расстояние).

Для каждой вершины **n**, алгоритм A\* вычисляет общую оценку **f(n)** как сумму **g(n)** и **h(n)**: **f(n) = g(n) + h(n)**.

**Процесс работы** алгоритма A\*:

1. **Инициализация:** Устанавливаются начальная вершина и целевая вершина. Для начальной вершины устанавливается **g**(начальная) = 0, и вычисляется **h**(начальная) как эвристическая оценка расстояния до целевой вершины.
2. **Открытый и закрытый список:** Создаются два списка - открытый и закрытый. Открытый список содержит вершины, которые еще предстоит проверить, а закрытый список содержит те, которые уже были проверены.
3. **Цикл поиска:** Повторяется следующий процесс:

- Из открытого списка выбирается вершина с наименьшей оценкой **f(n)**.

- Проверяется, является ли эта вершина целевой. Если да, то алгоритм завершается, и путь восстанавливается.

- В противном случае вершина перемещается в закрытый список, и рассматриваются все её соседи.

- Для каждого соседа вычисляются значения **g** и **h**, и если суммарная оценка **f** меньше, чем предыдущая, обновляются значения **g** и **h**, а вершина добавляется в открытый список.

1. **Завершение:** Алгоритм завершается, когда целевая вершина достигнута или открытый список пуст, что означает, что путь не существует.

**Из особенностей** алгоритма A\* можно выделить:

* **Эвристическая оценка:** Использование эвристической оценки **h(n)** позволяет алгоритму эффективно направлять поиск в сторону целевой вершины, что ускоряет процесс.
* **Оптимальность:** При определённых условиях и эвристике **h(n),** алгоритм A\* гарантирует нахождение оптимального пути.
* **Подбор эвристики:** Эффективность алгоритма может зависеть от правильного выбора эвристической оценки **h(n).** Она должна быть допустимой (не завышать действительное расстояние) и монотонной.

**Алгоритм A\*** широко используется в реальных приложениях, таких как поиск маршрутов в компьютерных играх, маршрутизация в сетях, планирование движения роботов и другие задачи, где требуется нахождение оптимального пути в графе.

**1.6. Алгоритм поиска в глубину(DFS)**

**Алгоритм поиска в глубину** (Depth-First Search, DFS) представляет собой один из основных методов обхода графов. Он используется для нахождения всех вершин в графе, доступных из заданной начальной вершины. Также алгоритм применяется для проверки связности графа и поиска циклов в нем.

**Принцип работы** алгоритма поиска в глубину:

1. **Инициализация:** Выбирается начальная вершина. Помечается, что эта вершина уже посещена. В случае поиска компонент связности или обхода всего графа, процесс начинается с первой вершины.
2. **Поиск в глубину:** Начиная с выбранной вершины, алгоритм следует по рёбрам графа вглубь, до тех пор, пока не достигнет конечной вершины или вершины, у которой нет непосещенных соседей.
3. **Рекурсивный характер:** Алгоритм часто реализуется рекурсивно. При посещении вершины вызывается функция поиска в глубину для каждого непосещенного соседа этой вершины.
4. **Пометка посещенных вершин:** Посещенные вершины могут быть помечены, чтобы избежать повторного посещения.
5. **Завершение:** Процесс повторяется для всех непосещенных вершин, пока все вершины графа не будут посещены.

**Из особенностей** алгоритма поиска в глубину можно выделить:

* **Глубина перед шириной:** Алгоритм поиска в глубину, в отличие от алгоритма поиска в ширину, идет так глубоко, как это возможно, прежде чем вернуться и исследовать другие направления. Это делает его хорошим выбором для задач, связанных с исследованием структуры графа в глубину.
* **Стек вызовов:** Рекурсивная реализация алгоритма поиска в глубину использует стек вызовов. В некоторых случаях это может привести к переполнению стека при работе с очень большими графами. В таких случаях лучше использовать итеративную реализацию с явным стеком данных.
* **Поиск компонент связности:** Алгоритм поиска в глубину позволяет эффективно находить компоненты связности в неориентированных графах.
* **Временная сложность:** Временная сложность алгоритма поиска в глубину составляет **O(V + E)**, где **V** - количество вершин, **E** - количество рёбер в графе.

**Алгоритм поиска в глубину** находит применение в различных областях, таких как топологическая сортировка, выделение компонент связности, анализ графов, а также в решении задач, связанных с обходом деревьев и графов.

**1.7. Сравнение алгоритмов**

1. **Алгоритм Дейкстры:**

* *Цель*: Находит кратчайший путь от одной вершины к остальным взвешенного графа.
* *Особенности*: Жадность, работает только для графов без рёбер отрицательного веса.
* *Применение*: Задачи маршрутизации и оптимизации путей в сетевых структурах.

1. **Алгоритм Беллмана-Флойда:**

* *Цель*: Находит кратчайшие пути от одной начальной вершины ко всем остальным, допускает рёбра с отрицательным весом.
* *Особенности*: Работает с отрицательными весами, проверка на отрицательные циклы.
* *Применение*: Сетевые технологии, транспортная логистика.

1. **Алгоритм Флойда**:

* *Цель*: Находит кратчайшие пути между всеми парами вершин в ориентированном графе.
* *Особенности*: Универсальность, обработка отрицательных весов.
* *Применение*: Сетевые технологии, где нужно заранее вычислить кратчайшие пути между всеми парами узлов.

1. **Алгоритм А\*:**

* *Цель*: Находит кратчайший путь от начальной точки к целевой точке в графе.
* *Особенности*: Использует эвристическую оценку, эффективен в различных областях.
* *Применение*: Робототехника, компьютерные игры, геоинформационные системы.

1. **Алгоритм поиска в глубину (DFS)**

* *Цель*: Обходит граф, находит все вершины, доступные из заданной начальной вершины, применяется для проверки связности графа и поиска циклов.
* *Особенности*: Глубина перед шириной, рекурсивный характер.
* *Применение*: Топологическая сортировка, выделение компонент связности.

**1.8. Выводы**

* **Выбор алгоритма зависит от конкретной задачи:** Например, для нахождения кратчайших путей в сетевых структурах часто используется алгоритм Дейкстры, а для обхода всего графа - алгоритм поиска в глубину.
* **Условия применимости:** алгоритм Беллмана-Форда стоит рассматривать при наличии рёбер с отрицательным весом, в то время как алгоритм Дейкстры требует неотрицательные веса.
* **Эффективность:** Алгоритмы имеют различную временную сложность. Например, алгоритм поиска в глубину имеет временную сложность **O(V + E)**, а алгоритм Флойда - **O(N^3)**, что делает их более или менее эффективными в зависимости от размера графа.
* **Применение в различных областях:** Каждый из этих алгоритмов имеет свои области применения, и выбор зависит от конкретных требований задачи.

Каждый алгоритм имеет свои сильные и слабые стороны, и выбор определенного зависит от конкретных условий задачи.

Таблица 1. Сравнение алгоритмов в зависимости от задачи

|  |  |  |
| --- | --- | --- |
| Подходящий алгоритм | Условия задачи | Сложность алгоритма |
| Алгоритм поиска в глубину | Найти всевозможные пути между двумя вершинами | **O(V + E)** |
| Алгоритм Дейкстры | Найти кратчайший путь между двумя вершинами | **O((V + E) \* log(V))** |
| Алгоритм Беллмана-Форда | Найти кратчайший путь между двумя вершинами с учётом рёбер, которые имеют отрицательный вес | **O(V \* E)** |
| Алгоритм Флойда | Найти кратчайший путь между всеми парами вершин | **O()** |
| Алгоритм А\*(модификация алгоритма Дейкстры) | Найти путь между двумя вершинами по сбалансированным параметрам | **O(b^d)**  b – макc. кол-во детей у каждой вершины, d - глубина решения |

***Глава 2. Практическая реализация алгоритмов поиска пути в графе***

**2.1. Общие сведения по реализации**

Алгоритмы поиска пути в графе реализованы на языке С++17 с применением парадигмы ООП.

*Проект включает в себя следующие файлы:*

* **destinations.txt:** в данном файле содержится информация о пунктах назначения, которые будут присутствовать в графе.
* **flights.txt:** в данном файле содержится информация о рейсах между пунктами назначения. Например, такая как: стоимость рейса, длительность полёта. Данный файл применяется для графа, в котором отсутствуют рёбра с отрицательным весом.
* **flights1.txt:** в данном файле содержится точно такая же информация, как в файле flights.txt, но за одним исключением: в нём присутствуют рёбра с отрицательным весом. Он необходим для демонстрации работы алгоритма Беллмана-Форда.
* **CourseWork.cpp:** содержит в себе класс TravelSystem, который содержит в себе две структуры, Destination необходима для хранения информации о рейсах, а InefficientPath – для хранение информации о неэффективных путях. Также, там присутствуют 4 вектора для хранения разного рода информации о путях и все необходимые методы для реализации задачи. В коде граф представлен в виде матрицы смежности.

В данном проекте достаточно большое количество методов, поэтому было принято решение описать только ключевые для решения задачи, а об остальных просто упомянуть.

**2.2. Общие сведения о графе**

В данном проекте присутствуют данные о пунктах назначениях и рейсах, которые указаны в файлах. Файл destonation.txt содержит в себе информацию о пунктах назначения, которые присутствуют в файле. Их список:

* Moscow
* New-York
* Minsk
* Berlin
* London
* Warsaw
* Paris
* Lisbon

В последних двух текстовых файлах присутствует информация о рейсах. В файле flights.txt о рейсах содержатся пути **только** с положительным весами рёбер, а в flights1.txt присутствуют рейсы с отрицательным весом ребра.

Информация по файлу flights.txt с рейсами:

1. Moscow New-York 500 8
2. Moscow Minsk 100 1
3. Moscow Berlin 300 4
4. New-York London 700 10
5. Minsk Warsaw 150 2
6. Minsk Paris 400 5
7. Berlin Paris 250 3
8. Berlin Warsaw 200 2
9. London Lisbon 300 2
10. London Paris 150 1
11. Warsaw Lisbon 100 3
12. Paris Lisbon 125 2

Информация по файлу flights1.txt с рейсами:

1. Moscow New-York -500 8
2. Moscow Minsk 100 1
3. Moscow Berlin 300 4
4. New-York London 700 -10
5. Minsk Warsaw 150 -2
6. Minsk Paris 400 5
7. Berlin Paris 250 3
8. Berlin Warsaw 200 2
9. London Lisbon 300 2
10. London Paris 150 1
11. Warsaw Lisbon -100 3
12. Paris Lisbon 125 2

Первый параметр в текстовом файле с рейсами означает пункт отправления, второй параметр – пункт назначения, третий параметр – цена рейса, а последний, четвёртый параметр – длительность рейса.

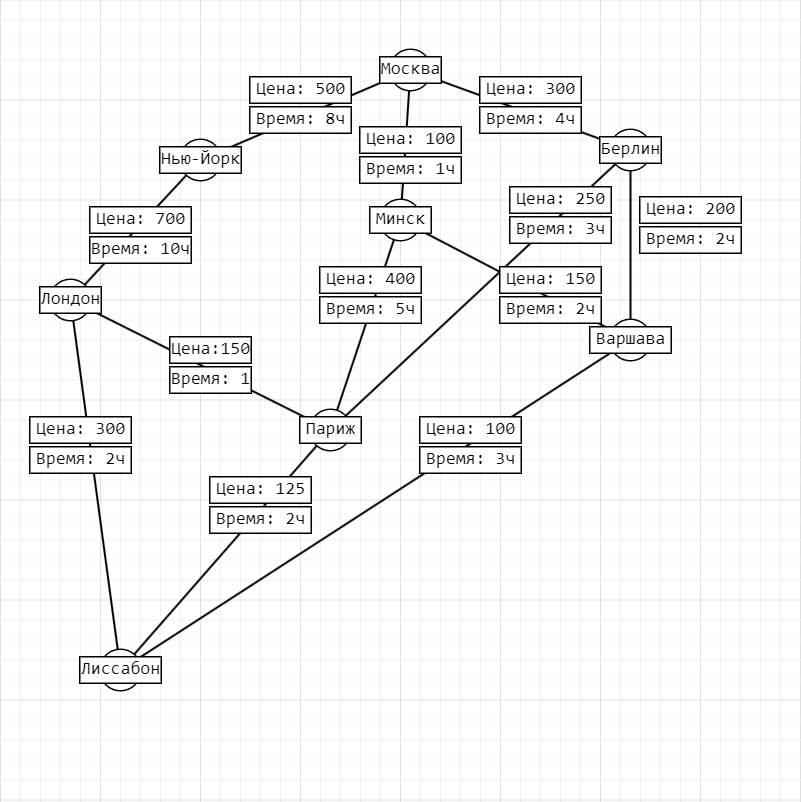


Рисунок 1. Визуализация графа с данными из файла flights.txt

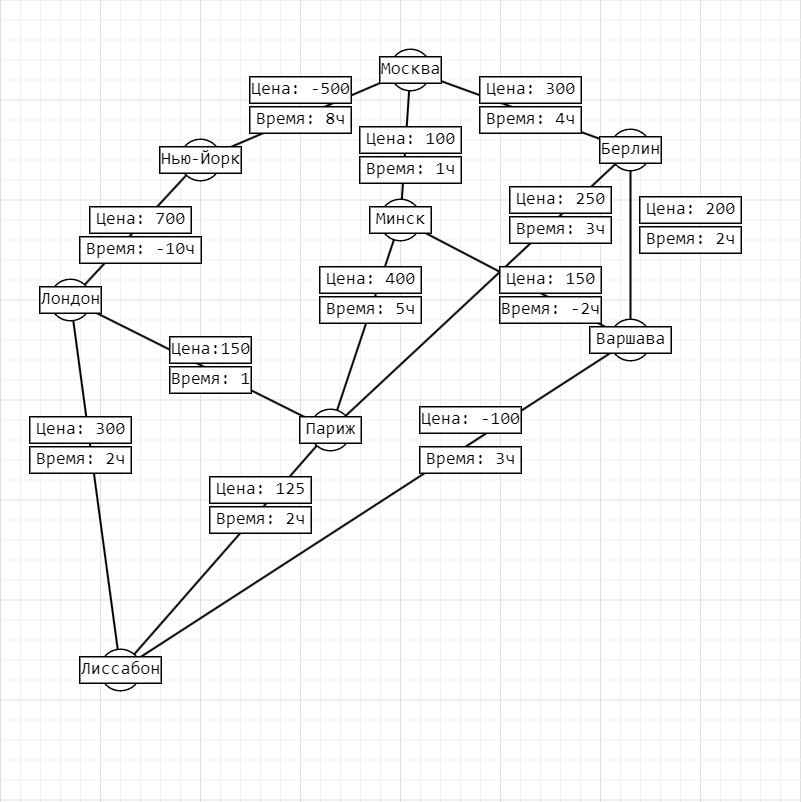


Рисунок 2. Визуализация графа с данными из файла flights1.txt

**2.3. Описание методов**

**В классе TravelSystem** определены следующие методы:

Метод addDestination() – добавляет пункт назначения в граф.

Метод createFlight() - создаёт рейс между двумя пунктами назначения.

Метод displayFlights() - выводит информацию о рейсах в графе.

Метод loadDestinationsFromFile() - загружает информацию о пунктах назначения из файла.

Метод loadFlightsFromFile() – загружает информацию о рейсах их файла.

Метод displayAdjacencyMatrix() – выводит матрицу смежности для отладки(техническая информация).

Метод findRoute() – поиск маршрута между двумя пунктами с использованием алгоритма поиска в глубину.

Рассмотрим данный метод подробнее.

1. **Параметры:**
   * **fromCity** и **toCity**: исходный и конечный пункты назначения, между которыми осуществляется поиск маршрутов.
2. **Работа метода:**
   * Метод инициализирует пустой текущий путь (**currentPath**) и вектор всех найденных маршрутов (**allPaths**).
   * Затем осуществляется проверка наличия указанных городов в списке пунктов назначения. Если города существуют, определяются соответствующие им индексы в векторе **destinations**.
   * Начальный пункт добавляется в текущий путь, и вызывается рекурсивная функция поиска в глубину (**dfs()**), которая обходит граф и формирует все возможные маршруты от начальной до конечной точки.
   * Найденные маршруты добавляются в вектор **allPaths**.
   * Наконец, метод выводит найденные маршруты в консоль.
3. **Алгоритм поиска в глубину (dfs()):**
   * Рекурсивная функция **dfs()** проверяет, достигнут ли конечный пункт. Если да, текущий путь добавляется в вектор всех путей.
   * Для каждой смежной вершины, имеющей рейс между текущей и смежной вершиной, проверяется, посещена ли она в текущем пути. Если нет, она добавляется в текущий путь, и рекурсивно вызывается **dfs()** для следующей вершины.
   * После завершения рекурсии вершина удаляется из текущего пути, что позволяет исследовать другие варианты.
4. **Вывод результатов:**
   * Найденные маршруты выводятся в консоль в удобном формате.
5. **Обработка ошибок:**
   * Предусмотрена обработка ситуации, когда один или оба города не найдены в списке пунктов назначения. В таком случае выводится сообщение об ошибке.
6. **Примечание:**
   * Данный метод может выявлять все возможные маршруты между двумя городами, что может привести к большому числу результатов в случае сложного графа.

Метод findBestRouteDijkstra() - алгоритм Дейкстры, который ищет путь в графе.

Рассмотрим данный алгоритм подробнее.

**Параметры:**

**fromCity** - начальный город

**toCity** - конечный город

**Основные шаги метода**

1. Инициализация переменных:

* **from** и **to** - индексы начального и конечного городов в векторе **destinations**.
* **distance** - вектор текущих расстояний от начального города до всех остальных.
* **parent** - вектор предшественников для восстановления оптимального пути.
* **visited** - вектор, отслеживающий посещение каждого города.

1. Инициализация начального города:

* Расстояние от начального города до самого себя устанавливается в 0.

1. Основной цикл алгоритма:

* Выбор города с наименьшим текущим расстоянием и обновление расстояний до соседних городов, если новый путь короче.

1. Формирование пути:

* Формирование оптимального пути от начального города до конечного с использованием вектора parent.

1. Вывод результата:

* Вывод наилучшего маршрута.

Метод findBestRouteBellmanFord() – алгоритм Беллмана-Форда, который ищет путь в графе c учётом рёбер с отрицательным весом.

Рассмотрим данный метод поподробнее.

**Параметры:**

* **fromCity** - начальный город
* **toCity** - конечный город

**Основные шаги метода**

1. *Инициализация переменных:*
   * **from** и **to** - индексы начального и конечного городов в векторе **destinations**.
   * **distance** - вектор текущих расстояний от начального города до всех остальных. Изначально все расстояния устанавливаются в бесконечность.
   * **parent** - вектор предшественников для восстановления оптимального пути.
2. *Инициализация начального города:*
   * Расстояние от начального города до самого себя устанавливается в 0.
3. *Применение алгоритма Беллмана-Форда:*
   * Для каждого города выполняется релаксация ребер графа. Если найден более короткий путь к городу, обновляется значение расстояния и запоминается предшественник.
4. *Проверка наличия отрицательных циклов:*
   * После завершения основного цикла, производится дополнительная проверка наличия отрицательных циклов. Если обнаружен цикл, выводится сообщение об ошибке, и метод завершает выполнение.
5. *Формирование маршрута:*
   * В случае отсутствия отрицательных циклов, метод строит оптимальный маршрут от начального города до конечного с использованием вектора **parent**.
6. *Вывод результата:*
   * Выводится наилучший маршрут.

Метод FloydWarshall() – алгоритм Флойда, который находит путь в графе между всеми парами вершин в графе.

Рассмотрим данный алгоритм поподробнее.

Данный метод реализует алгоритм Флойда для нахождения кратчайших путей между всеми парами вершин в графе. Граф представлен матрицей смежности, где города представлены строками, а стоимость перемещения между городами задается весами ребер.

Шаги алгоритма:

1. **Инициализация переменных:**
   * **n** - количество вершин в графе (размер вектора **destinations**).
   * Создаются матрицы **durations** и **costs** для хранения кратчайших путей между всеми парами вершин.
2. **Инициализация матриц кратчайших путей:**
   * Инициализация матрицы **durations** текущими значениями длительности перемещения между городами. Если между вершинами существует ребро, то в матрице устанавливается соответствующее значение, если нет - **numeric\_limits<int>::max()**.
   * Инициализация матрицы **costs** текущими значениями стоимости перемещения между городами. Аналогично, если между вершинами существует ребро, то в матрице устанавливается соответствующее значение, если нет - **numeric\_limits<int>::max()**.
3. **Применение алгоритма Флойда для длительности:**
   * Для каждой вершины **k** проверяются все пары вершин **i** и **j**. Если существует путь от **i** до **k** и от **k** до **j**, причем общая длительность этого пути меньше текущей длительности между **i** и **j**, то обновляется значение в матрице **durations**.
4. **Вывод результатов для длительности:**
   * Выводятся кратчайшие длительности между всеми парами вершин. Если значение в матрице **durations[i][j]** не является бесконечностью, выводится информация о длительности маршрута от вершины **i** до **j**.
5. **Применение алгоритма Флойда для стоимости:**
   * Для каждой вершины **k** проверяются все пары вершин **i** и **j**. Если существует путь от **i** до **k** и от **k** до **j**, причем общая стоимость этого пути меньше текущей стоимости между **i** и **j**, то обновляется значение в матрице **costs**.
6. **Вывод результатов для стоимости:**
   * Выводятся кратчайшие стоимости между всеми парами вершин. Если значение в матрице **costs[i][j]** не является бесконечностью, выводится информация о стоимости маршрута от вершины **i** до **j**.

**Примечания:**

1. **Отсутствие прямого пути:**
   * Если значение **durations[i][j]** или **costs[i][j]** осталось равным **numeric\_limits<int>::max()**, это указывает на отсутствие прямого пути между вершинами **i** и **j**.

Метод void findBalancedRouteAStar() – поиска пути между двумя пунктами назначения по сбалансированным параметрам(длительность рейса и цена).

Рассмотрим данный алгоритм поподробнее.

Данный алгоритм реализует поиск оптимального маршрута между двумя городами, учитывая баланс между длительностью и стоимостью перемещения.

**Параметры:**

* **fromCity** - начальный город
* **toCity** - конечный город
* **durationWeight** - вес длительности перемещения
* **costWeight** - вес стоимости перемещения

**Основные шаги метода:**

1. *Инициализация переменных:*
   * **from** и **to** - индексы начального и конечного городов в векторе **destinations**.
   * Определение эвристической функции **heuristic** для оценки расстояния между двумя вершинами.
   * Приоритетная очередь **pq** для хранения вершин с приоритетом, упорядоченных по комбинированной стоимости.
   * Векторы **totalCost**, **parent** и **visited** для отслеживания текущих стоимостей, предшественников и посещенных вершин.
2. *Инициализация начальной вершины:*
   * Устанавливается начальное значение стоимости от начальной вершины (**fromCity**) равным 0.
3. *Основной цикл алгоритма A*\*:
   * В цикле выбирается вершина **u** с наименьшей комбинированной стоимостью из приоритетной очереди.
   * Если достигнута конечная вершина (**u == to**), алгоритм завершается.
   * В противном случае, вершина **u** помечается как посещенная.
   * Для каждой соседней вершины **v**, которая еще не была посещена и имеет положительные значения стоимости и длительности, вычисляются стоимость и длительность перемещения до **v**.
   * Вычисляется эвристическая оценка расстояния до конечной вершины **to**.
   * Рассчитывается комбинированная стоимость, и если она меньше текущей стоимости до **v**, обновляются значения и вершина добавляется в приоритетную очередь.
4. *Обработка неэффективных путей:*
   * Если комбинированная стоимость больше или равна текущей стоимости до вершины **v**, путь считается неэффективным, и выводится сообщение, а также сохраняется информация о неэффективном пути в структуре данных **inefficientPaths**.

**Примечание**

* В алгоритме A\* используется комбинированная стоимость, которая учитывает веса стоимости и длительности, а также эвристическую функцию для оценки расстояния до конечной вершины.

**Стоит отметить** что во всех алгоритмах, кроме алгоритма Флойда(в этом нет необходимости, учитывая его суть), есть возможность отследить процесс отсечения неэффективных путей.

Метод processingThreadFindRoute() – выводит прогресс решений алгоритма findRoute().

Метод processingThreadFindBestRouteDijkstra() – выводит прогресс решений алгоритма findBestRouteDijkstra().

Метод processingThreadFindBesttRouteBellmanFord() – выводит прогресс решений алгоритма findBestRouteBellmanFord().

Метод processingThreadFindBalancedRouteAStar() - выводит прогресс решений алгоритма findBalancedRouteAStar() по шагам.

Метод MeasuringTimeOfAStar() – измеряет время выполнения метода findBalancedRouteAStar().

Метод MeasuringTimeOfDijkstraDuration – измеряет время выполнения метода findBestRouteDijkstra().

Метод MeasuringTimeOfBellmanFord() – измеряет время выполнения метода findBestRouteBellmanFord().

Метод MeasuringTimeOfFloydWarshall() – измеряет время выполнения метода FloydWarshall().

Метод void MeasuringTimeFindRoute() – измеряет время выполнения метода findRoute().

**2.4. Пример запуска программы**

Ещё раз продемонстрируем графы, в одном из которых есть рёбра с отрицательным весом.

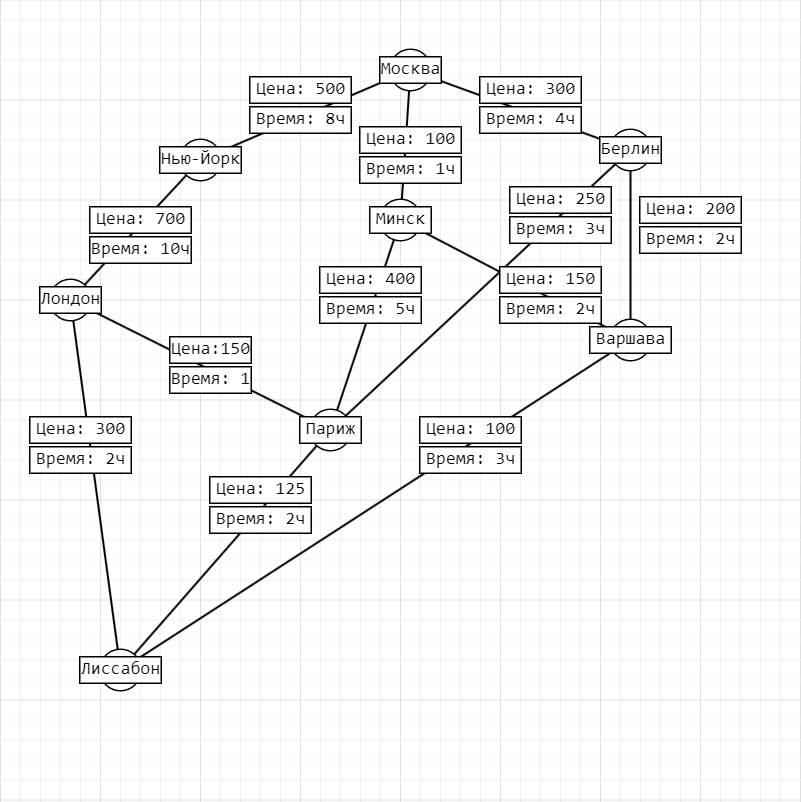


Рисунок 3. Визуализация графа с данными из файла flights.txt



Рисунок 4. Матрица смежности для графа flights со стоимостью рейсов



Рисунок 5. Матрица смежности для графа flights с длительностью рейсов



Рисунок 6. Матрица смежности для графа flights1 со стоимостью рейсов

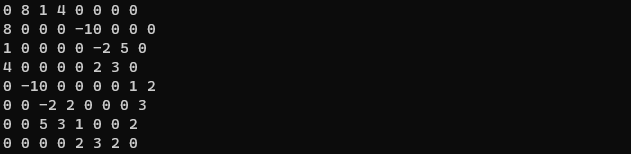


Рисунок 7. Матрица смежности для графа flights1 с длительностью рейсов

Допустим, мы хотим узнать кратчайший путь по цене и длительности от Москвы до Лиссабона. Используем алгоритм Дейкстры и получим следующий результат:

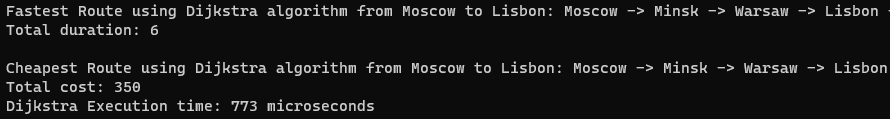


Рисунок 8. Результат работы алгоритма Дейкстры в графе из файла flights.txt

Также, взглянем визуально на путь в графе:

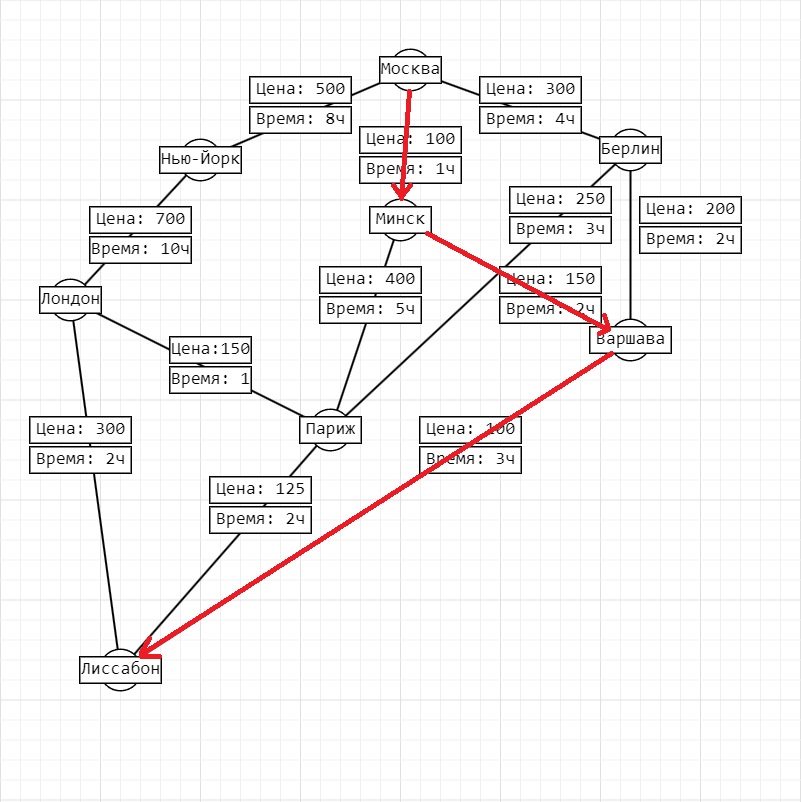


Рисунок 9. Визуальное представление построенного пути в графе

Если проанализировать данный путь, можно понять, что это действительно является наилучшим по цене и по длительности.

Продолжим рассмотрение других алгоритмов.

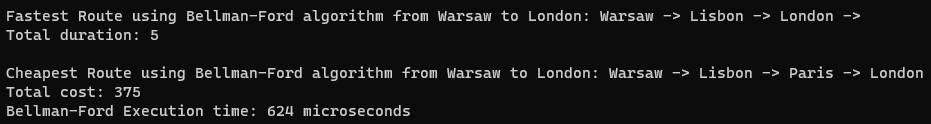


Рисунок 10. Результат работы алгоритма Беллмана-Форда в графе flights

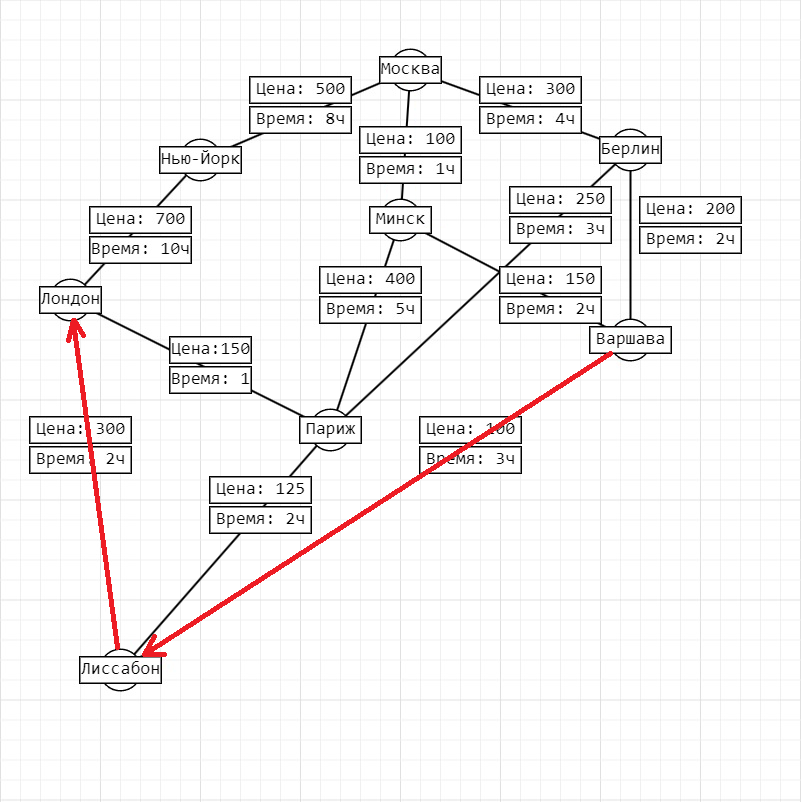


Рисунок 11. Визуальное представление построенного пути в графе по длительности

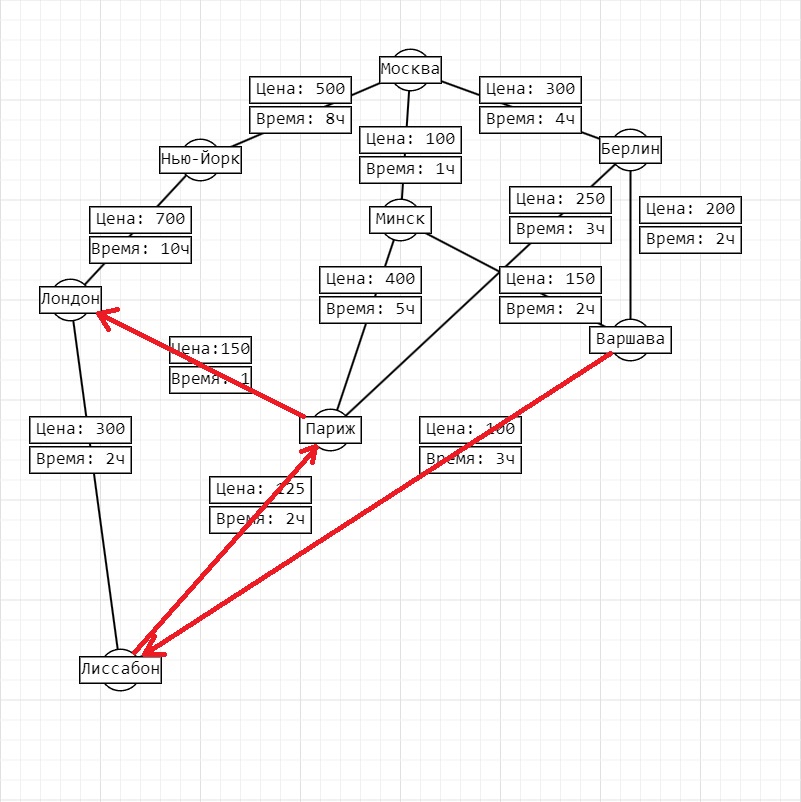


Рисунок 12. Визуальное представление построенного пути в графе по цене

Теперь попробуем использовать алгоритм Беллмана-Форда в графе с отрицательными весами. Попробуем построить путь из Москвы в Лиссабон.



Рисунок 13. Результат работы алгоритма Беллмана-Форда в графе flights1

Как мы видим, при попытке построить кратчайший маршрут в графе с отрицательными весами, алгоритм обнаруживает отрицательный цикл и прекращает свою работу.

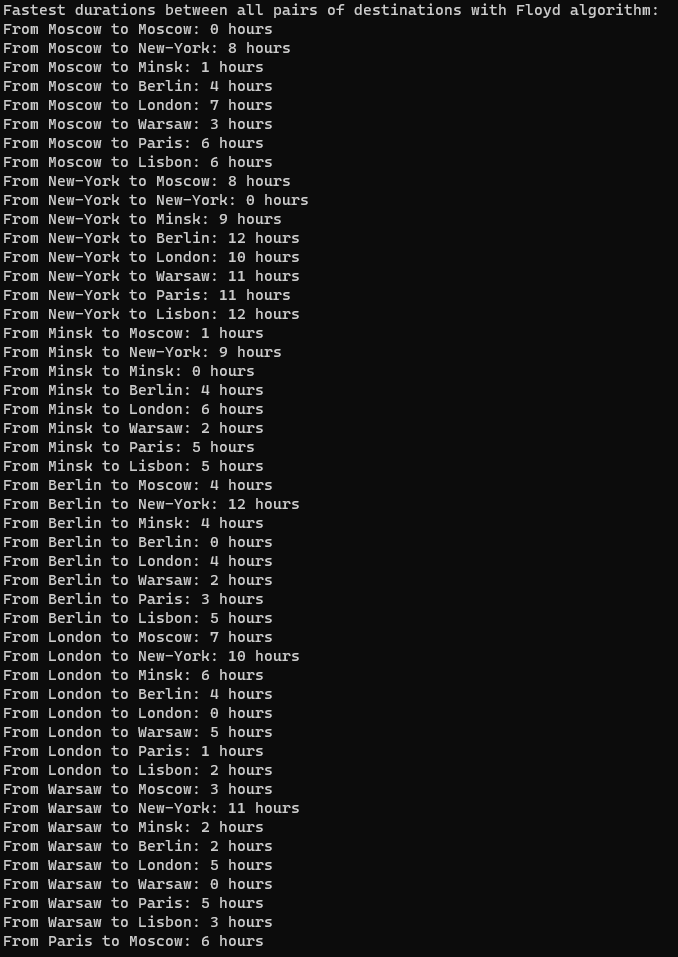


Рисунок 14. Результат работы алгоритма Флойда в графе flights(1/3)

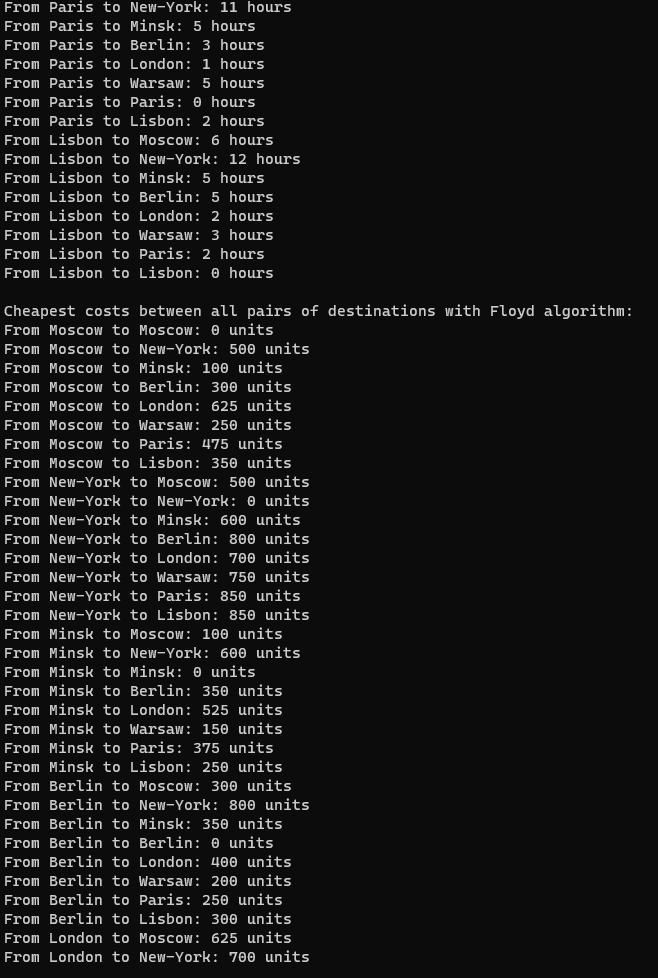


Рисунок 14. Результат работы алгоритма Флойда в графе flights(2/3)



Рисунок 14. Результат работы алгоритма Флойда в графе flights(3/3)

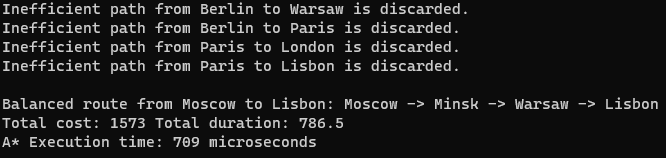


Рисунок 15. Результат работы алгоритма А\* в графе flights

**Примечание:** можно заметить пример отображения отсечённых путей.

Визуальное представление пути можно посмотреть на рис. 5.

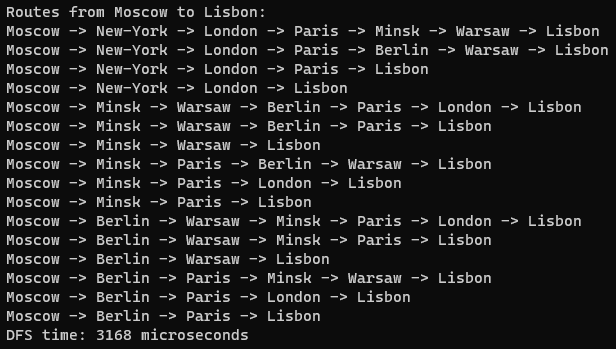


Рисунок 16. Результат работы алгоритма поиска в глубину в графе flights

**Заключение**

В ходе разработки и реализация алгоритмов поиска пути в графе на языке программирования С++ с использование алгоритмов Дейкстры, Беллмана-Форда, Флойда, А\* и поиска в глубину, были получены некоторые результаты.

Взглянем на результаты сравнения скорости выполнения(в микросекундах) алгоритмов в графе flights. Для примера был взят путь от Москвы до Лиссабона.

Таблица 2. Сравнение скорости выполнения алгоритмов

|  |  |
| --- | --- |
| Алгоритм поиска пути | Время выполнения(в мкс) |
| Алгоритм Дейкстры | 772 микросекунды |
| Алгоритм Беллмана-Форда | 630 микросекунды |
| Алгоритм Флойда | 25574 микросекунды |
| Алгоритм поиска в глубину | 3168 микросекунд |
| Алгоритм А\* | 990 микросекунд |

1. **Изучение различных методов поиска пути:**
   * Был проведен анализ и изучение различных алгоритмов поиска пути в графе, позволяющих решать задачи нахождения оптимального пути, учитывая различные критерии, такие как длительность, стоимость и другие.
2. **Разработка и реализация алгоритмов:**
   * Были разработаны и реализованы алгоритмы Дейкстры, Беллмана-Форда, Флойда, A\* и поиска в глубину с учётом теоретического материала на языке программирования C++. Реализации алгоритмов предоставляют возможность находить оптимальные маршруты и выполнять различные задачи анализа графов.
3. **Обработка отрицательных циклов:**
   * В рамках реализации алгоритма Беллмана-Форда был добавлен механизм обнаружения отрицательных циклов и соответствующих предупреждений. Это позволяет предотвращать поиск оптимальных путей в случае наличия циклов отрицательного веса.
4. **Сравнение производительности:**
   * Произведено сравнение производительности различных алгоритмов поиска пути в графе для определения их эффективности. Это помогло оценить скорость работы каждого из них. Наиболее быстрым оказался алгоритм Беллмана-Форда.

**Список литературы**

*1. Кормен Т. Алгоритмы: Построение и анализ / Т. Кормен, Ч. Лейзер-*

*сон, Р. Ривест. [Пер. с англ.: К. Белов и др.] - МОСКВА: МЦНМО,*

*1999 - 955 с. ISBN 5-900916-37-5*

*2. Wans, S-X. The Improved Dijkstra's Shortest Path Algorithm and Its Appli-*

*cation / S-X Wang // Procedia Engineering. – 2012 - №29. – Pp. 1186 –*

*1190 doi: 10.1016/j.proeng.2012.01.110*

*3. Кнут, Д. Искусство программирования, том 3 Сортировка и поиск / Д.*

*Кнут – Москва: «Вильямс», 2013 – 824 с.*

*4. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт; пер. с англ. –*

*Санкт-Петербург, «Невский диалект», 2001 – 352 с.*

*5. Шилдт, Г. Полный справочник по С++, 4-е издание / Г. Шилдт; пер. с*

*англ. – Москва: Вильямс, 2006 – 800 с.*

**Приложение**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <algorithm>

#include <limits>

#include <queue>

#include <cmath>

#include <thread>

#include <chrono>

#include <utility>

using namespace std;

class TravelSystem {

private:

// Хранит информацию о рейсах

struct Destination {

int duration;

double cost;

Destination(int dur, double co) {

duration = dur;

cost = co;

}

};

// Хранит отсечённые пути

struct InefficientPath {

string from;

string to;

};

vector<InefficientPath> inefficientPaths;

vector<string> destinations;

vector<vector<Destination>> adjacencyMatrix;

vector<string> currentPath;

vector<vector<string>> allPaths;

// Алгоритм поиска в глубину, используется в методе findRoute()

void dfs(int current, int destination) {

if (current == destination) {

allPaths.push\_back(currentPath);

return;

}

for (size\_t i = 0; i < destinations.size(); ++i) {

if (adjacencyMatrix[current][i].cost > 0) {

// Проверка на посещенность вершины

if (find(currentPath.begin(), currentPath.end(), destinations[i]) == currentPath.end()) {

// Помечаем вершину как посещенную

currentPath.push\_back(destinations[i]);

dfs(i, destination);

// Отмечаем вершину как непосещенную после завершения DFS

currentPath.pop\_back();

}

}

}

}

public:

// Метод для добавления пункта назначения

void addDestination(const string& dest) {

destinations.push\_back(dest);

// Обновляем матрицу смежности

for (auto& row : adjacencyMatrix) {

row.emplace\_back(0, 0); // Добавляем новый элемент с нулевыми значениями

}

// Добавляем новую строку с нулевыми значениями

vector<Destination> newRow(destinations.size(), Destination(0, 0));

adjacencyMatrix.push\_back(newRow);

}

// Метод для создания рейса между двумя пунктами

void createFlight(const string& fromCity, const string& toCity, double cost, int duration) {

auto fromIndex = find(destinations.begin(), destinations.end(), fromCity);

auto toIndex = find(destinations.begin(), destinations.end(), toCity);

if (fromIndex != destinations.end() && toIndex != destinations.end()) {

int from = distance(destinations.begin(), fromIndex);

int to = distance(destinations.begin(), toIndex);

// Обновляем матрицу смежности с информацией о рейсе

adjacencyMatrix[from][to].cost = cost;

adjacencyMatrix[to][from].cost = cost; // Предполагаем, что рейс двусторонний

adjacencyMatrix[from][to].duration = duration;

adjacencyMatrix[to][from].duration = duration;

}

else {

cerr << "One or both cities not found." << endl;

}

}

// Метод для вывода информации о рейсах

void displayFlights() const {

for (size\_t i = 0; i < destinations.size(); ++i) {

for (size\_t j = 0; j < destinations.size(); ++j) {

if (adjacencyMatrix[i][j].cost > 0) {

cout << "Flight from " << destinations[i] << " to " << destinations[j]

<< " Cost: " << adjacencyMatrix[i][j].cost << " Duration: " << adjacencyMatrix[i][j].duration << endl;

}

}

}

}

// Метод для загрузки информации о пунктах назначения из файла

void loadDestinationsFromFile(const string& filename) {

ifstream file(filename);

if (file.is\_open()) {

string name;

while (file >> name) {

addDestination(name);

}

file.close();

}

else {

cerr << "Unable to open the file: " << filename << endl;

}

}

// Метод для загрузки информации о рейсах из файла

void loadFlightsFromFile(const string& filename) {

ifstream file(filename);

if (file.is\_open()) {

string from, to;

double cost;

int duration;

while (file >> from >> to >> cost >> duration) {

createFlight(from, to, cost, duration);

}

file.close();

}

else {

cerr << "Unable to open the file: " << filename << endl;

}

}

// Допустим, метод для вывода матрицы смежности для отладки

void displayAdjacencyMatrix() {

for (size\_t i = 0; i < destinations.size(); ++i) {

for (size\_t j = 0; j < destinations.size(); ++j) {

cout << adjacencyMatrix[i][j].cost << " ";

}

cout << endl;

}

cout << endl;

for (size\_t i = 0; i < destinations.size(); ++i) {

for (size\_t j = 0; j < destinations.size(); ++j) {

cout << adjacencyMatrix[i][j].duration << " ";

}

cout << endl;

}

}

// Метод для поиска маршрута между двумя пунктами

void findRoute(const string& fromCity, const string& toCity) {

currentPath.clear();

allPaths.clear();

auto fromIndex = find(destinations.begin(), destinations.end(), fromCity);

auto toIndex = find(destinations.begin(), destinations.end(), toCity);

if (fromIndex != destinations.end() && toIndex != destinations.end()) {

int from = distance(destinations.begin(), fromIndex);

int to = distance(destinations.begin(), toIndex);

// Инициализируем текущий путь

currentPath.push\_back(destinations[from]);

dfs(from, to);

// Выводим найденные маршруты

cout << "\nRoutes from " << fromCity << " to " << toCity << ":" << endl;

for (const auto& path : allPaths) {

for (size\_t i = 0; i < path.size(); ++i) {

cout << path[i];

if (i < path.size() - 1) {

cout << " -> ";

}

}

cout << endl;

}

}

else {

cerr << "\nOne or both cities not found." << endl;

}

}

// Метод поиска пути с помощью алгоритма Дейкстры

void findBestRouteDijkstra(const string& fromCity, const string& toCity) {

int from = distance(destinations.begin(), find(destinations.begin(), destinations.end(), fromCity));

int to = distance(destinations.begin(), find(destinations.begin(), destinations.end(), toCity));

vector<vector<int>> distances(destinations.size(), vector<int>(2, numeric\_limits<int>::max()));

vector<vector<int>> parents(destinations.size(), vector<int>(2, -1));

vector<vector<bool>> visited(destinations.size(), vector<bool>(2, false));

distances[from][0] = 0; // 0 для длительности

distances[from][1] = 0; // 1 для цены

for (size\_t count = 0; count < destinations.size() - 1; ++count) {

for (int weightType = 0; weightType < 2; ++weightType) {

int u = -1;

for (size\_t i = 0; i < destinations.size(); ++i) {

if (!visited[i][weightType] && (u == -1 || distances[i][weightType] < distances[u][weightType]))

u = i;

}

visited[u][weightType] = true;

for (size\_t v = 0; v < destinations.size(); ++v) {

double edgeWeight = (weightType == 0) ? adjacencyMatrix[u][v].duration : adjacencyMatrix[u][v].cost;

if (!visited[v][weightType] && edgeWeight > 0 &&

distances[u][weightType] != numeric\_limits<int>::max() &&

distances[u][weightType] + edgeWeight < distances[v][weightType]) {

distances[v][weightType] = distances[u][weightType] + edgeWeight;

parents[v][weightType] = u;

}

//else {

// // Путь считается неэффективным, сохраняем информацию

// cout << "Inefficient path from " << destinations[u] << " to " << destinations[v] << " is discarded.\n";

// inefficientPaths.push\_back({ destinations[u], destinations[v] });

//}

// Если применить в коде, то будет и без того большой объём информации в консоли

// Поэтому, если интересно, то просто необходимо убрать комментарии

}

}

}

// Вывод результатов для длительности

vector<string> fastestRoute;

for (int v = to; v != -1; v = parents[v][0]) {

fastestRoute.push\_back(destinations[v]);

}

reverse(fastestRoute.begin(), fastestRoute.end());

cout << "\nFastest Route using Dijkstra algorithm from " << fromCity << " to " << toCity << ": ";

for (const auto& city : fastestRoute) {

cout << city << " -> ";

}

cout << "\nTotal duration: " << distances[to][0] << endl;

// Вывод результатов для цены

vector<string> cheapestRoute;

for (int v = to; v != -1; v = parents[v][1]) {

cheapestRoute.push\_back(destinations[v]);

}

reverse(cheapestRoute.begin(), cheapestRoute.end());

cout << "\nCheapest Route using Dijkstra algorithm from " << fromCity << " to " << toCity << ": ";

for (const auto& city : cheapestRoute) {

cout << city << " -> ";

}

cout << "\nTotal cost: " << distances[to][1] << endl;

}

// Метод поиска пути с помощью алгоритма Беллмана-Форда

bool findBestRouteBellmanFord(const string& fromCity, const string& toCity) {

int from = distance(destinations.begin(), find(destinations.begin(), destinations.end(), fromCity));

int to = distance(destinations.begin(), find(destinations.begin(), destinations.end(), toCity));

vector<vector<int>> distances(destinations.size(), vector<int>(2, numeric\_limits<int>::max()));

vector<vector<int>> parents(destinations.size(), vector<int>(2, -1));

distances[from][0] = 0; // 0 для длительности

distances[from][1] = 0; // 1 для цены

const double Max\_Negative\_Cost = -5000;

const int Max\_Negative\_Duration = -50;

for (size\_t count = 0; count < destinations.size() - 1; ++count) {

for (size\_t i = 0; i < destinations.size(); ++i) {

for (size\_t j = 0; j < destinations.size(); ++j) {

if (adjacencyMatrix[i][j].cost > 0) {

// Релаксация для продолжительности

if (distances[i][0] != numeric\_limits<int>::max() &&

distances[i][0] + adjacencyMatrix[i][j].duration < distances[j][0]) {

distances[j][0] = distances[i][0] + adjacencyMatrix[i][j].duration;

if (distances[j][0] < Max\_Negative\_Duration) {

cerr << "\nNegative cycle has been detected\n";

return false;

}

parents[j][0] = i;

}

// Релаксация для цены

if (distances[i][1] != numeric\_limits<int>::max() &&

distances[i][1] + adjacencyMatrix[i][j].cost < distances[j][1]) {

distances[j][1] = distances[i][1] + adjacencyMatrix[i][j].cost;

if (distances[j][0] < Max\_Negative\_Cost) {

cerr << "\nNegative cycle has been detected\n";

return false;

}

parents[j][1] = i;

}

}

}

}

}

// Вывод результатов для длительности

vector<string> fastestRoute;

for (int v = to; v != -1; v = parents[v][0]) {

fastestRoute.push\_back(destinations[v]);

}

reverse(fastestRoute.begin(), fastestRoute.end());

cout << "\nFastest Route using Bellman-Ford algorithm from " << fromCity << " to " << toCity << ": ";

for (const auto& city : fastestRoute) {

cout << city << " -> ";

}

cout << "\nTotal duration: " << distances[to][0] << endl;

// Вывод результатов для цены

vector<string> cheapestRoute;

for (int v = to; v != -1; v = parents[v][1]) {

cheapestRoute.push\_back(destinations[v]);

}

reverse(cheapestRoute.begin(), cheapestRoute.end());

cout << "\nCheapest Route using Bellman-Ford algorithm from " << fromCity << " to " << toCity << ": ";

for (const auto& city : cheapestRoute) {

cout << city << " -> ";

}

cout << "\nTotal cost: " << distances[to][1] << endl;

// Отрицательных циклов не обнаружено

return true;

}

// Метод поиска пути с помощью алгоритма Флойда

void FloydWarshall() {

size\_t n = destinations.size();

// Создаем матрицы для хранения кратчайших путей между всеми парами вершин

vector<vector<int>> durations(n, vector<int>(n, numeric\_limits<int>::max()));

vector<vector<int>> costs(n, vector<int>(n, numeric\_limits<int>::max()));

// Инициализируем матрицы кратчайших путей текущими значениями

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (i == j) {

durations[i][j] = 0;

}

else if (adjacencyMatrix[i][j].duration > 0) {

durations[i][j] = adjacencyMatrix[i][j].duration;

}

}

}

// Алгоритм Флойда для длительности

for (size\_t k = 0; k < n; ++k) {

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (durations[i][k] != numeric\_limits<int>::max() &&

durations[k][j] != numeric\_limits<int>::max() &&

durations[i][k] + durations[k][j] < durations[i][j]) {

durations[i][j] = durations[i][k] + durations[k][j];

}

}

}

}

// Выводим результат для длительности

cout << "\nFastest durations between all pairs of destinations with Floyd algorithm:" << endl;

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (durations[i][j] != numeric\_limits<int>::max()) {

cout << "From " << destinations[i] << " to " << destinations[j] << ": " << durations[i][j] << " hours" << endl;

}

}

}

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (i == j) {

costs[i][j] = 0;

}

else if (adjacencyMatrix[i][j].cost > 0) {

costs[i][j] = adjacencyMatrix[i][j].cost;

}

}

}

// Алгоритм Флойда для стоимости

for (size\_t k = 0; k < n; ++k) {

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (costs[i][k] != numeric\_limits<int>::max() &&

costs[k][j] != numeric\_limits<int>::max() &&

costs[i][k] + costs[k][j] < costs[i][j]) {

costs[i][j] = costs[i][k] + costs[k][j];

}

}

}

}

// Выводим результат для стоимости

cout << "\nCheapest costs between all pairs of destinations with Floyd algorithm:" << endl;

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (costs[i][j] != numeric\_limits<int>::max()) {

cout << "From " << destinations[i] << " to " << destinations[j] << ": " << costs[i][j] << " units" << endl;

}

}

}

}

// Метод поиска пути с помощью алгоритма Флойда

void FloydWarshall() {

size\_t n = destinations.size();

// Создаем матрицы для хранения кратчайших путей между всеми парами вершин

vector<vector<int>> durations(n, vector<int>(n, numeric\_limits<int>::max()));

vector<vector<int>> costs(n, vector<int>(n, numeric\_limits<int>::max()));

// Инициализируем матрицы кратчайших путей текущими значениями

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (i == j) {

durations[i][j] = 0;

}

else if (adjacencyMatrix[i][j].duration > 0) {

durations[i][j] = adjacencyMatrix[i][j].duration;

}

}

}

// Алгоритм Флойда для длительности

for (size\_t k = 0; k < n; ++k) {

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (durations[i][k] != numeric\_limits<int>::max() &&

durations[k][j] != numeric\_limits<int>::max() &&

durations[i][k] + durations[k][j] < durations[i][j]) {

durations[i][j] = durations[i][k] + durations[k][j];

}

}

}

}

// Выводим результат для длительности

cout << "\nFastest durations between all pairs of destinations with Floyd algorithm:" << endl;

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (durations[i][j] != numeric\_limits<int>::max()) {

cout << "From " << destinations[i] << " to " << destinations[j] << ": " << durations[i][j] << " hours" << endl;

}

}

}

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (i == j) {

costs[i][j] = 0;

}

else if (adjacencyMatrix[i][j].cost > 0) {

costs[i][j] = adjacencyMatrix[i][j].cost;

}

}

}

// Алгоритм Флойда для стоимости

for (size\_t k = 0; k < n; ++k) {

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (costs[i][k] != numeric\_limits<int>::max() &&

costs[k][j] != numeric\_limits<int>::max() &&

costs[i][k] + costs[k][j] < costs[i][j]) {

costs[i][j] = costs[i][k] + costs[k][j];

}

}

}

}

// Выводим результат для стоимости

cout << "\nCheapest costs between all pairs of destinations with Floyd algorithm:" << endl;

for (size\_t i = 0; i < n; ++i) {

for (size\_t j = 0; j < n; ++j) {

if (costs[i][j] != numeric\_limits<int>::max()) {

cout << "From " << destinations[i] << " to " << destinations[j] << ": " << costs[i][j] << " units" << endl;

}

}

}

}

// Метод для поиска машрута между двумя пунктами по сбалансированным параметрам(алгоритм A\*, модификация алгоритма Дейкстры),

// а также, реализация процесса «жадного» поиска маршрута или отсечения перебираемых путей, которые оцениваются как неэффективные, для ускорения поиска.

void findBalancedRouteAStar(const string& fromCity, const string& toCity, double durationWeight, double costWeight) {

int from = distance(destinations.begin(), find(destinations.begin(), destinations.end(), fromCity));

int to = distance(destinations.begin(), find(destinations.begin(), destinations.end(), toCity));

// Определение эвристической функции для расстояния между двумя вершинами

auto heuristic = [this](int city1, int city2) {

return abs(city1 - city2);

};

priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;

vector<double> totalCost(destinations.size(), numeric\_limits<double>::max());

vector<int> parent(destinations.size(), -1);

vector<bool> visited(destinations.size(), false);

totalCost[from] = 0;

pq.push({ 0, from });

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

if (u == to) {

break; // Достигнута конечная вершина

}

visited[u] = true;

for (size\_t v = 0; v < destinations.size(); ++v) {

if (!visited[v] && adjacencyMatrix[u][v].cost > 0 && adjacencyMatrix[u][v].duration > 0) {

double costToV = totalCost[u] + costWeight \* adjacencyMatrix[u][v].cost;

double durationToV = totalCost[u] + durationWeight \* adjacencyMatrix[u][v].duration;

double heuristicCost = heuristic(v, to);

// здесь баланс между длительностью и ценой

double combinedCost = costWeight \* costToV + durationWeight \* durationToV + heuristicCost;

if (combinedCost < totalCost[v]) {

totalCost[v] = combinedCost;

parent[v] = u;

pq.push({ combinedCost, v });

} else {

// Путь считается неэффективным

cout << "Inefficient path from " << destinations[u] << " to " << destinations[v] << " is discarded.\n";

// сохраняет информацию в структуре данных для последующего анализа

inefficientPaths.push\_back({ destinations[u], destinations[v] });

}

}

}

}

// Строим маршрут

vector<string> balancedRoute;

for (int v = to; v != -1; v = parent[v]) {

balancedRoute.push\_back(destinations[v]);

}

reverse(balancedRoute.begin(), balancedRoute.end());

// Выводим результат

cout << "\nBalanced route from " << fromCity << " to " << toCity << ": ";

for (const auto& city : balancedRoute) {

cout << city << " -> ";

}

cout << "\nTotal cost: " << totalCost[to] << " Total duration: " << totalCost[to] / durationWeight << endl;

}

// Отображает прогресс выполнения алгоритма поиска в глубину

void processingThreadFindRoute(const string& fromCity) {

for (size\_t i = 0; i < destinations.size(); ++i) {

// вызов метода поиска маршрута для каждой вершины

findRoute(fromCity, destinations[i]);

// Периодически выводим текущий прогресс

cout << "\nProgress: " << i + 1 << " out of " << destinations.size() << " vertices processed.\n";

}

}

// Отображает прогресс выполнения алгоритма Дейкстры

void proccesingThreadFindBestRouteDijkstra(const string& fromCity) {

for (size\_t i = 0; i < destinations.size(); ++i) {

findBestRouteDijkstra(fromCity, destinations[i]);

// Выводим текущий прогресс

cout << "Progress: " << i + 1 << " from " << destinations.size() << " destinations processed.\n";

}

cout << "Calculation of the cheapest route is completed.\n";

}

// Отображает прогресс выполнения алгоритма Беллмана-Форда

void proccesingThreadFindBestRouteBellmanFord(const string& fromCity) {

for (size\_t i = 0; i < destinations.size(); ++i) {

findBestRouteBellmanFord(fromCity, destinations[i]);

// Выводим текущий прогресс

cout << "Progress: " << i + 1 << " from " << destinations.size() << " destinations processed.\n";

}

cout << "Calculation of the cheapest route is completed.\n";

}

// Отображает прогресс выполнения алгоритма А\*

void proccesingThreadFindBalancedRouteAStar(const string& fromCity, double durationWeight, double costWeight) {

for (size\_t i = 0; i < destinations.size(); ++i) {

findBalancedRouteAStar(fromCity, destinations[i], durationWeight, costWeight);

// Выводим текущий прогресс

cout << "Progress: " << i + 1 << " from " << destinations.size() << " destinations processed.\n";

}

cout << "Calculation of the fastest route is completed.\n";

}

// Отображает производительность по времени алгоритма А\*(в микросекундах)

void MeasuringTimeOfAStar(const string& fromCity, const string& toCity, double durationWeight, double costWeight) {

auto start = chrono::high\_resolution\_clock::now();

findBalancedRouteAStar(fromCity, toCity, durationWeight, costWeight);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "A\* Execution time: " << duration.count() << " microseconds\n";

}

// Отображает производительность по времени алгоритма Дейкстры(в микросекундах)

void MeasuringTimeDijkstra(const string& fromCity, const string& toCity) {

auto start = chrono::high\_resolution\_clock::now();

findBestRouteDijkstra(fromCity, toCity);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Dijkstra Execution time: " << duration.count() << " microseconds\n";

}

// Отображает производительность по времени алгоритма Беллмана-Форда(в микросекундах)

void MeasuringTimeBellmanFord(const string& fromCity, const string& toCity) {

auto start = chrono::high\_resolution\_clock::now();

findBestRouteBellmanFord(fromCity, toCity);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Bellman-Ford Execution time: " << duration.count() << " microseconds\n";

}

// Отображает производительность по времени алгоритма Флойда(в микросекундах)

void MeasuringTimeFloydWarshall() {

auto start = chrono::high\_resolution\_clock::now();

FloydWarshall();

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Floyd-Warshall Execution time: " << duration.count() << " microseconds\n";

}

void MeasuringTimeFindRoute(const string& fromCity, const string& toCity) {

auto start = chrono::high\_resolution\_clock::now();

findRoute(fromCity, toCity);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "DFS time: " << duration.count() << " microseconds\n";

}

};

int main() {

TravelSystem travelSystem;

// Загружаем информацию о пунктах назначения из файла

travelSystem.loadDestinationsFromFile("destinations.txt");

// Загружаем информацию о рейсах из файла

travelSystem.loadFlightsFromFile("flights.txt");

// Выводим информацию о рейсах

travelSystem.displayFlights();

// Выводим матрицу смежности для отладки

travelSystem.displayAdjacencyMatrix();

travelSystem.MeasuringTimeOfAStar("Moscow", "Lisbon", 2.0, 1.0);

travelSystem.processingThreadFindRoute("Moscow");

cout << "\n-------------Example of Display Dijkstra proccesing---------------------\n";

travelSystem.proccesingThreadFindBestRouteDijkstra("Moscow");

cout << "\n-----------------------------End of example-----------------------------\n";

travelSystem.findBestRouteDijkstra("Moscow", "Lisbon");

travelSystem.findBestRouteDijkstra("Warsaw", "London");

travelSystem.findBestRouteDijkstra("Warsaw", "New-York");

cout << "\n-------------Example of Display Bellman-Ford proccesing---------------------\n";

travelSystem.proccesingThreadFindBestRouteBellmanFord("Moscow");

cout << "\n-----------------------------End of example-----------------------------\n";

travelSystem.findBestRouteBellmanFord("Moscow", "Lisbon");

travelSystem.findBestRouteBellmanFord("Warsaw", "London");

travelSystem.findBestRouteBellmanFord("Warsaw", "New-York");

travelSystem.findBalancedRouteAStar("Moscow", "Lisbon", 2.0, 1.0);

cout << "\n-------------Example of Display A\* proccesing---------------------\n";

travelSystem.proccesingThreadFindBalancedRouteAStar("Moscow", 2.0, 1.0);

cout << "\n-----------------------------End of example-----------------------------\n";

TravelSystem travelSystemWithNegativeWeight;

// Загружаем информацию о пунктах назначения из файла

travelSystemWithNegativeWeight.loadDestinationsFromFile("destinations.txt");

// Загружаем информацию о рейсах из файла

travelSystemWithNegativeWeight.loadFlightsFromFile("flights1.txt");

travelSystemWithNegativeWeight.displayFlights();

travelSystemWithNegativeWeight.displayAdjacencyMatrix();

travelSystemWithNegativeWeight.MeasuringTimeFindRoute("Moscow", "Lisbon");

travelSystemWithNegativeWeight.findBestRouteBellmanFord("Moscow", "Lisbon");

travelSystemWithNegativeWeight.findBestRouteBellmanFord("Warsaw", "London");

travelSystemWithNegativeWeight.findBestRouteBellmanFord("Warsaw", "New-York");

travelSystem.FloydWarshall();

travelSystem.MeasuringTimeOfAStar("Moscow", "Lisbon", 2.0, 1.0);

travelSystem.MeasuringTimeDijkstra("Moscow", "Lisbon");

travelSystem.MeasuringTimeDijkstra("Warsaw", "London");

travelSystem.MeasuringTimeDijkstra("Warsaw", "New-York");

travelSystem.MeasuringTimeBellmanFord("Moscow", "Lisbon");

travelSystem.MeasuringTimeBellmanFord("Warsaw", "London");

travelSystem.MeasuringTimeBellmanFord("Warsaw", "New-York");

travelSystemWithNegativeWeight.MeasuringTimeBellmanFord("Moscow", "Lisbon");

travelSystemWithNegativeWeight.MeasuringTimeBellmanFord("Warsaw", "London");

travelSystemWithNegativeWeight.MeasuringTimeBellmanFord("Warsaw", "New-York");

travelSystem.MeasuringTimeFloydWarshall();

travelSystemWithNegativeWeight.MeasuringTimeFloydWarshall();

return 0;

}

Приложение 1