

ОГЛАВЛЕНИЕ

| | |
|---|----|
| ВВЕДЕНИЕ..... | 3 |
| 1. Обзор предметной области..... | 4 |
| 1.1. Основные определения..... | 4 |
| 1.2. Виды транспорта и его особенности..... | 5 |
| 1.2.1. Железнодорожный..... | 5 |
| 1.2.2. Воздушный..... | 6 |
| 1.2.3. Автомобильный..... | 6 |
| 1.3. Построение маршрутов..... | 6 |
| 1.3.1. Мультимодальность..... | 6 |
| 1.3.2. Временные интервалы..... | 7 |
| 1.3.3. Инкрементальное построение..... | 7 |
| 1.3.4. Адаптивность по времени..... | 7 |
| 1.4. Построение фильтров к доступным маршрутам..... | 7 |
| 1.4.1. Косвенные признаки..... | 8 |
| 1.4.2. Осуществление фильтрации..... | 8 |
| 1.4.3. Функциональные зависимости..... | 8 |
| 1.5. Сортировка маршрутов..... | 9 |
| 1.5.1. Количество пересадок..... | 9 |
| 1.5.2. Время отправления/прибытия..... | 10 |
| 1.5.3. Время в пути..... | 10 |
| 1.6. Известные алгоритмы..... | 10 |
| 1.6.1. Алгоритм Дейкстры..... | 10 |
| 1.6.2. Алгоритм Йена..... | 12 |
| Выводы по главе 1..... | 13 |
| 2. Алгоритм построения маршрутов..... | 14 |
| 2.1. Модели данных..... | 14 |
| 2.1.1. Статичный граф..... | 14 |
| 2.1.2. Граф расписаний..... | 15 |
| 2.1.3. Граф рейсов..... | 17 |
| 2.2. Построение маршрутов..... | 22 |
| 2.2.1. Дополнение временных интервалов..... | 22 |
| 2.2.2. Фаза инициализации..... | 23 |
| 2.2.3. Фаза обхода..... | 25 |

| | |
|--|----|
| 2.3. Сортировка маршрутов | 28 |
| 2.3.1. Количество пересадок | 28 |
| 2.3.2. Время прибытия..... | 29 |
| 2.3.3. Время отправления..... | 29 |
| 2.3.4. Время в пути | 30 |
| 2.4. Построение фильтров | 30 |
| 2.4.1. Косвенные признаки..... | 31 |
| 2.4.2. Осуществление фильтрации..... | 33 |
| 2.4.3. Функциональные зависимости..... | 33 |
| 2.4.4. "Белые"и "черные"фильтры | 33 |
| Выводы по главе 2 | 33 |
| 3. Детали реализации и тестирование | 34 |
| 3.1. Работа с базой данных | 34 |
| 3.1.1. Особенности базы данных..... | 34 |
| 3.1.2. Персистентные модели данных..... | 34 |
| 3.1.3. Кэши | 34 |
| 3.2. Генерация карт транспортных сетей | 34 |
| 3.2.1. Генерация транспортных узлов..... | 34 |
| 3.2.2. Генерация транспортных рейсов..... | 34 |
| 3.2.3. Генерация центральных узлов..... | 34 |
| 3.3. Результаты тестирования | 35 |
| Выводы по главе 3 | 35 |
| ЗАКЛЮЧЕНИЕ | 36 |

ВВЕДЕНИЕ

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В данной главе описаны основные определения из области поиска путей в теории графов. В первом разделе главы описаны понятия из теории графов. Во втором – из формальной области транспортных сетей. В третьем разделе формализуется задача и список требований по поддерживаемым свойствам для построения маршрутов. Четвертый раздел содержит краткое описание основных алгоритмов теории графов для поиска путей с описанием преимуществ и недостатков данных подходов.

1.1. Основные определения

Определение 1. Теория графов – раздел дискретной математики, изучающий свойства графов.

Определение 2. Граф – это множество вершин (узлов), соединенных ребрами. В строгом определении графом называется такая пара множеств. $G = \langle E, V \rangle$, где V – подмножество любого счетного множества, а E – подмножество $V \times V$.

Определение 3. Маршрут – это конечная последовательность вершин, в которой каждая вершина (кроме последней) соединена ребром со следующей в последовательности вершиной. Цепью называется маршрут без повторяющихся ребер. Простой цепью называется маршрут без повторяющихся вершин (откуда следует, что в простой цепи нет повторяющихся ребер).

Определение 4. Ориентированный маршрут (или путь) – это конечная последовательность вершин и дуг, в которой каждый элемент инцидентен предыдущему и последующему.

Определение 5. Цикл – это цепь, в которой первая и последняя вершины совпадают. При этом длиной пути (или цикла) называют число составляющих его ребер.

Определение 6. Транспортное средство – это совокупность технических систем, предназначенных для перемещений людей и грузов из одного места в другое.

Определение 7. Транспортный узел – это комплекс транспортных устройств в пункте стыка нескольких видов транспорта, совместно выполняющих операции по обслуживанию транзитных, местных и городских перевозок грузов и пассажиров.

Определение 8. Транспортный рейс –

Определение 9. Транспортная сеть – это совокупность всех транспортных рейсов, представленных в течение интервала продажи билетов.

Определение 10. Остановка — специально отведенное общественное место, предназначенное для посадки/высадки пассажиров рейсового транспортного средства.

Определение 11. Расписание —

Определение 12. Мультимодальный маршрут — это конечная последовательность транспортных рейсов, попав на которые в определенные промежутки времени можно добраться от начального транспортного узла до конечного.

Определение 13. Построитель маршрутов — это программный комплекс для обработки внешних поисковых клиентских запросов, имеющий доступ к полному объему данных о расписаниях на всех транспортных узлах и осуществляющий выдачу определенного количества маршрутов в соответствии с поступившими в запросах требованиями. Также в качестве дополнительных возможностей доступно построение фильтров и различной статистики (активные транспортные узлы, активные транспортные рейсы, проходящие через заданный узел).

Определение 14. Клиентское приложение — это любое приложение, которое осуществляет запросы к построителю маршрутов за результатом (маршрутами и фильтрами).

1.2. Виды транспорта и его особенности

В транспортной сети, в которой будут строиться маршруты, будет существовать только транспорт с конкретным расписанием транспортных рейсов. Таким образом, идет допущение о том, что система сети идеальна и весь транспорт гарантировано совершает остановки в назначенное время. Постановка вспомогательных свойств для построителя маршрутов, которые позволяют сгладить последствия этого допущения будут описаны в следующих главах. Далее идет описание рассматриваемого транспорта.

1.2.1. Железнодорожный

В задаче будут рассматриваться 2 вида железнодорожного транспорта. Во-первых, это будут поезда дальнего следования, у которых небольшое количество рейсов (около 10^5 в течение интервала продажи билетов). Во-вторых, это будут электрички, которые уже совершают до 10^6 рейсов за аналогич-

ный промежуток времени. Транспортными узлами являются железнодорожные станции и вокзалы.

1.2.2. Воздушный

Воздушный транспорт будет представлен только самолетами. При этом количество рейсов около 10^3 , поэтому особый интерес этот случай не представляет. Но стоит отметить, что в большинстве случаев мультимодальный маршрут не будет содержать больше одного воздушного сегмента пути. Транспортными узлами являются аэропорты.

1.2.3. Автомобильный

Автомобильный транспорт состоит из автобусных междугородних рейсов. Около 95% таких рейсов совершаются только между соседними городами, что сильно упрощает задачу, но количество все равно большое — 10^6 . Также в эту категорию входит транспорт в пределах города (или любого крупного населенного пункта), например, такси. Стоит отметить, что в этот вид транспорта можно внести любые другие средства передвижения внутри города, так как в конечном счете это не будет влиять на алгоритм. При этом важно, чтобы у нового транспорта в пределах города имелась возможность рассчитать эвристическое времени передвижения между двумя транспортными узлами, которые относятся к одному населенному пункту. Эту задачу следует решать на основе статистики или с помощью сторонних сервисов, которые умеют анализировать дорожную ситуацию, например, такие сервисы, как 2gis или Яндекс.карты, которые могут оценить время движения на основе карты пробок. Транспортными узлами являются автобусные остановки и крупные населенные пункты.

1.3. Построение маршрутов

Основная задача, ставящаяся перед строителем маршрутов — построение маршрутов по данным, доступным в его памяти и внешних базах данных, доступных для чтения в конкретный момент времени. На алгоритм построения маршрутов в транспортной сети накладываются следующие условия и ограничения.

1.3.1. Мультимодальность

Маршруты могут быть мультимодальными, то есть проходить через несколько точек-остановок, содержать пересадки, проходить разными видами

транспорта со своими особенностями и т.д.; Это нужно для того, чтобы была возможность добраться из любой точки в любую, где есть хотя бы какой-нибудь транспорт. Вариант пройти пешком небольшой кусок пути тоже доступен внутри крупного населенного пункта также должен быть доступен.

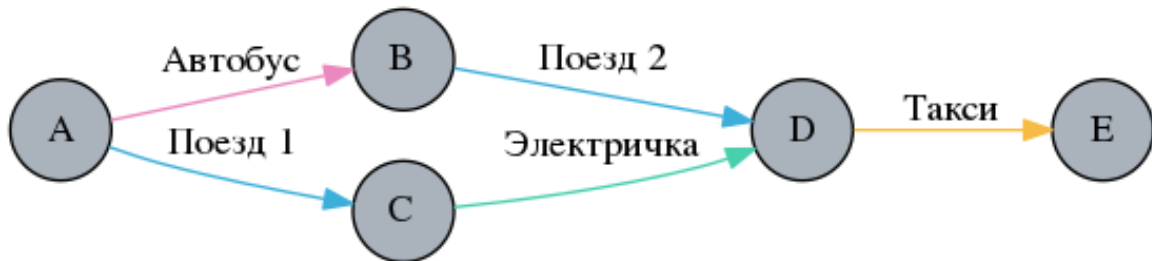


Рисунок 1 – Несколько мультимодальных маршрутов от точки А до точки Е.

1.3.2. Временные интервалы

Маршруты можно строить для определенных интервалов времени. Например, хотим выехать в промежуток с 8-00 до 12-00 утра, а приехать в любой день на следующей неделе, но обязательно после 21-00. Это требуется, чтобы иметь возможность бронировать гостиницу, не отходя от кассы.

1.3.3. Инкрементальное построение

Маршруты требуется строить инкрементально (не все сразу, а только небольшую часть из существующих) из-за того, что возможное количество маршрутов может достигать до 10^9 между парой крупных населенных пунктов с 3 допустимыми пересадками и интервалом времени в пути равным нескольким дням. Это требуется для конечного клиентского приложения, чтобы можно было организовать страничный показ результатов без полного вычисления всех маршрутов на предыдущих страницах.

1.3.4. Адаптивность по времени

Маршруты могут строиться адаптивно по времени из-за того, что важно время отклика алгоритма, то есть в приоритете время выполнения над показом действительно всех требуемых результатов.

1.4. Построение фильтров к доступным маршрутам

Под фильтром в данном случае понимается предикат, который принимает в качестве аргумента построенный маршрут и возвращает ИСТИНА или

ЛОЖЬ в зависимости от того, удовлетворяет ли маршрут критериям поиска, которые задает фильтр. Таким образом, помимо построения самих маршрутов требуется построить фильтры по доступным маршрутам со следующими условиями.

1.4.1. Косвенные признаки

Из-за того, что маршруты строятся не все сразу, то кроме непосредственно найденных маршрутов существует огромное количество потенциально доступных маршрутов. При этом мы хотим получить к ним доступ по косвенным признакам. Например, это может быть тип транспорта, номер поезда или тип места в самолете (у окна/у туалета/в хвосте). Формально любой параметр доступный в модели данных может стать доступным для фильтрации¹.

1.4.2. Осуществление фильтрации

1.4.3. Функциональные зависимости

Не последнюю роль в фильтрах играют функциональные зависимости, потому что в последствии нужно будет их эффективно показывать без противоречий. Например, тип места «у окна» в купе и каюте относятся к разным типам транспорта и их нельзя объединять. Пример «дерева» функциональных зависимостей для поезда:

- Точки отправления и прибытия
- Интервалы отправления и прибытия
- Вид транспорта:
- Поезд:
 - Перевозчик
 - Бренд
 - Номер
 - Тип вагона:
 - * Купе:
 - Верхнее/нижнее
 - Не у туалета
 - * Сидячий:

¹Под моделью данных в данном случае подразумевается любой абстрактный объект, который имеет отражение в реальном мире: поезд, самолет, аэропорт и т.д.

- У окна/у прохода
- * Плацкарт:
 - Верхнее/нижнее
 - Не у туалета
 - Боковое/не боковое

Каждый уровень списка зависит от родительского уровня и строго им определяется для того, чтобы исключить ситуации, когда несколько косвенных признаков совпадают уже разных видов транспорта. Например, признак «Перевозчик» у поезда и самолета. Такое разделение необходимо для корректного с точки зрения логики и удобного вывода результата в клиентском приложении.

1.5. Сортировка маршрутов

Маршруты требуется строить в порядке сортировки. В простейшем варианте можно сортировать только построенные маршруты, что не представляет из себя никакой сложности. В сложном варианте маршруты строятся на основе любого предиката сравнения пары маршрутов и выдаются в результат, гарантируя определенный порядок. В рамках данной работы подходит «средний» вариант. Требуется гарантировать определенный порядок построенных маршрутов без пропусков, но предикаты известны заранее. Всего их 4 основных вида:

1. Количество пересадок
2. Время отправления
3. Время прибытия
4. Время в пути

Далее рассмотрим каждый из них более подробно.

1.5.1. Количество пересадок

Самая простая сортировка в рамках данной работы (или просто сортировка по-умолчанию). Несложно заметить, что количество пересадок будет пропорционально количеству транспорта, который будет включен в мульти-модальный маршрут. И если представить каждый отрезок пути на конкретном транспорте отдельным ребром в абстрактном графе, то сортировка будет происходить относительно количества ребер.

1.5.2. Время отправления/прибытия

У каждой остановки любого транспортного рейса существует время прибытия и время отбытия. У первой и последней остановок оно совпадает. Если задан бесконечный или полу-бесконечный интервал, то нижний предел времени отграничен текущим северным временем, а верхний предел – неким разумным пределом, например, датой продаж. Для выбора верхней границы существуют разные стратегии. Например, при заданном интервале отправления можно устанавливать интервал прибытия зависимым от него. Для этого, нужно знать максимальную продолжительность пути в системе, что не представляет сложностей при ограничении максимального количества пересадок.

1.5.3. Время в пути

1.6. Известные алгоритмы

1.6.1. Алгоритм Дейкстры

Классический алгоритм для вычисления кратчайших путей в статическом графе $G = (V, E)$ с функцией веса² для ребер называется алгоритмом Дейкстры. В частности, алгоритм, разработанный Э. Дейкстрой, решает проблему нахождения кратчайших маршрутов от единственной стартовой вершины s до всех остальных вершин в графе G . Алгоритм поддерживает для каждой вершины u метку $\delta(u)$ с предварительным расстоянием от s до u . Каждая вершина может находиться в одном из следующих состояниях: нерассмотренная, рассмотренная, посещенная.

Изначально только вершина s считается рассмотренной с $\delta(s) = 0$ и все остальные вершины u являются нерассмотренными с $\delta(u) = \infty$.

На каждом шаге вершина u с минимальной $\delta(u)$ извлекается и удаляется из приоритетной очереди Q . Будет говорить, что вершина посещена, так как теперь известно, что $\delta(u)$ – кратчайшее расстояние. Все исходящие ребра (u, v) посещенной вершины релаксируются, то есть сравнивается кратчайшее расстояние от s через u до вершины v с предварительным расстоянием $\delta(v)$. Если оно меньше, то мы обновляет $\delta(v)$ и v в приоритетной очереди Q . Стоит заметить, что такое обновление для нее это либо операции вставки, если вершина рассмотрена, либо операция уменьшения ключа в противном случае.

²Предполагается, что в статическом графе ребра имеют неотрицательные веса. Для общего случая можно использовать алгоритм Беллмана — Форда.

Алгоритм заканчивает работу только тогда, когда очередь становится пустой. Это произойдет после n шагов, где n – это количество вершин в графе G .

В приведенном ниже псевдокоде видно, что иногда мы не посещаем все вершины в графе. Это происходит из-за того, что либо такие вершины недостижимы из s , либо очередь становится пустой раньше, чем мы доходим до такой вершины. Для того, чтобы избежать инициализации алгоритма за $O(n)$ при последовательных вызовах, можно сохранять посещенные вершины и обновлять их значения δ после конца алгоритма. Таким образом, асимптотика алгоритма зависит только от количества посещенных вершин и релаксируемых ребер, а не от n .

Листинг 1 – Алгоритм Дейкстры

```

function DIJKSTRA( $s$ )
     $\delta = \{\infty, \dots, \infty\}$                                  $\triangleright$  предварительные расстояния
     $\delta(s) = 0$                                                  $\triangleright$  поиск начинается из вершины  $s$ 
     $Q.update(0, s)$                                             $\triangleright$  приоритетная очередь
    while  $Q \neq \emptyset$  do
         $(_, u) = Q.deleteMin()$                                  $\triangleright$  посещаем  $u$ 
        for  $e = (u, v) \in E$  do                                 $\triangleright$  релаксируем ребра
            if  $\delta(u) + c(e) < \delta(v)$  then
                 $\delta(v) = \delta(u) + c(e)$ 
                 $Q.update(\delta(v), v)$ 
            end if
        end for
    end while
end function

```

Алгоритм Дейкстры использует максимум n операций вставки в приоритетную очередь, n удалений и m операций уменьшения ключа, обеспечивая таким образом асимптотику $O(m + n \log n)$ при использовании Фибоначчиевой кучи.

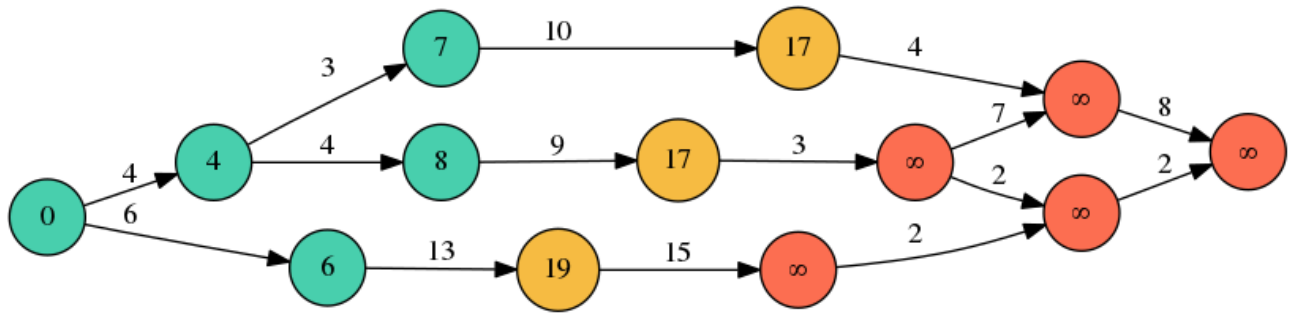


Рисунок 2 – Состояние алгоритма Дейкстры после посещения 5 вершин.
Посещенные - зеленые, рассмотренные - желтые, нерассмотренные - красные.

1.6.2. Алгоритм Йена

Алгоритм Йена находит k путей без циклов от единственной стартовой вершины s до конечной вершины t в статичном графе. Разработанный Йеном алгоритм предполагает, что в его основе будет лежать любой другой алгоритм поиска кратчайшего пути, например, его основой может послужить алгоритм Дейкстры. Идея основана на том, что можно построить изначально кратчайший путь и потом на основе этого пути искать отклонения, чтобы построить следующий. Каждая k итерация будет искать отклонения от кратчайших путей, полученных на $k - 1$ итерациях.

Листинг 2 – Алгоритм Йена

```

function YEN( $s, t, K$ )
     $A[0] = Dijkstra(s, t)$   $\triangleright$  определяем кратчайший путь от  $s$  до  $t$ 
     $B = \emptyset$   $\triangleright$  приоритетная очередь, хранящая кандидатов на  $k$  кратчайший
    путь
    for  $k = 1$  to  $K$  do
        for  $i = 0$  to  $size(A[k - 1]) - 1$  do
             $spurNode = A[k - 1].node(i)$   $\triangleright$  Вершина ответвления
            извлекается из полученного на предыдущей итерации пути
             $rootPath = A[k - 1].nodes(0, i)$   $\triangleright$  Последовательность вершин от
            стартовой до вершины ответвления образуют корневой префикс пути
            for  $p \in A$  do
                if  $rootPath = p.nodes(0, i)$  then
                    удаляем  $p.edge(i, i + 1)$  из графа
                end if
            end for
            for  $rootPathNode \in rootPath$  do
                if  $rootPathNode \neq spurNode$  then
                    удаляем  $rootPathNode$  из графа
                end if
            end for
             $spurPath = Dijkstra(spurNode, t)$   $\triangleright$  вычисляем
            путь-ответвление
             $totalPath = rootPath + spurPath$   $\triangleright$  Полный путь состоит из
            корневого префикса и пути-ответвления
             $B.append(totalPath)$   $\triangleright$  Добавление кандидата на кратчайший  $k$ 
            путь
            восстанавливаем в графе удаленные на текущей итерации ребра
            и вершины
        end for
        if  $B = \emptyset$  then
            break
        end if
         $(\_, A[k]) = B.deleteMin()$ 
    end for
end function

```

Выводы по главе 1

ГЛАВА 2. АЛГОРИТМ ПОСТРОЕНИЯ МАРШРУТОВ

2.1. Модели данных

В этом разделе будут описаны 3 способа представления данных о транспортной системе в виде графа, на котором впоследствии будут применяться алгоритмы для построения маршрутов и доступ к которому будет иметь построитель маршрутов.

2.1.1. Статичный граф

В статичном случае каждое ребро взвешенно функцией $c : E \rightarrow R$, и не имеет параллельных ребер. Для каждого ребра $e = (u, v)$ будем писать иногда $c(u, v)$ вместо $c(e)$. Будем называть такой граф простым взвешенным. Вес ребра можно интерпретировать как среднее время движения, требуемое на преодоление сегмента дороги, или как физическую длину. Длина пути P в таком случае равна $c(P) = \sum_{i=1}^k c(e_i)$. Путь P^* будет являться кратчайшим в том случае, если не существует другого пути P' с такой же стартовой и конечной вершинами, что и у пути P^* , такого, что $c(P') < c(P^*)$.

Можно построить граф транспортных рейсов, который будет соответствовать данному случаю. Для этого возьмем на вершину графа транспортный узел, а движение транспорта от одного транспортного узла до другого – за ребро. За вес ребра будет принята длина сегмента пути, так как она не меняется с течением времени. Далее на таком графе можно применить алгоритм Йена и найти k путей.

К сожалению, такой подход имеет ряд существенных минусов. Во-первых, будет доступна только одна естественная сортировка – по количеству пересадок, так как в данном случае это будет просто количество ребер. Во-вторых, поддержка временных интервалов потребует дополнительных вызовов алгоритма для того, чтобы гарантированно получить те маршруты, которых попадают в определенные временные границы. В-третьих, это размер графа, который зависит от количества остановок каждого транспорта за период даты продаж.

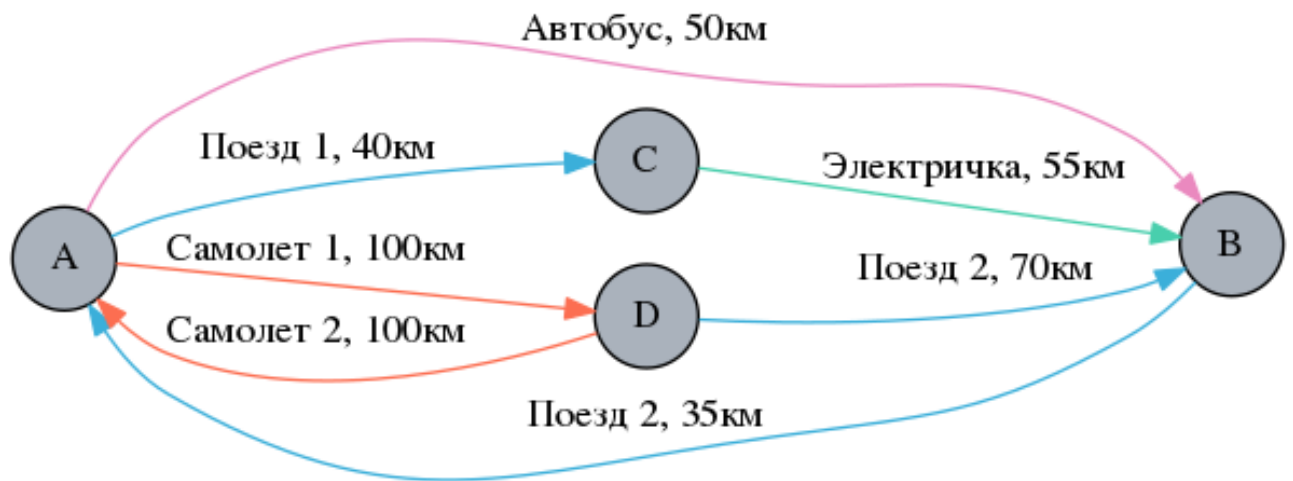


Рисунок 3 – Статичный граф с 4 городами и 6 транспортами

2.1.2. Граф расписаний

Традиционно расписания представляются множеством поездов (автобусов, самолетов и т.д.). Каждый поезд посещает последовательность станций (автобусных остановок, аэропортов и т.д.). Для каждой станции, за исключением последней, расписание включает в себя время отбытия, и для каждой станции, за исключением первой, включает в себя время прибытия, как показано в таблице:

Для того, чтобы была возможность математически определить связи, состоящие из нескольких поездов, мы разделим их в элементарные связи. Более формально, мы имеем множество станций B , множество остановок Z_S на каждой станции $S \in B$ и множество элементарных связей C , чьи элементы c являются кортежами вида $c = \{Z_d, Z_a, S_d, S_a, \tau_d, \tau_a\}$. Такой кортеж интерпретируется как поезд, который отправляется со станции S_d со временем отбытия τ_d после остановки Z_d и затем следующую остановку Z_a на станции S_a с временем прибытия τ_a . Если x обозначает поле кортежа, то $x(c)$ является значением x в элементарной связи c . Событие остановки похоже на идентификатор поезда, но на самом деле является более сложным. Определим событие остановки как последовательное прибытие и отбытие поезда со станции, не осуществляя пересадку. Для соответствующего прибытия элементарной связи c_1 и отбывающей связи c_2 выполняется $Z_a(c_1) = Z_d(c_2)$. Более того, событие остановки является локальным по отношению к каждой станции. Введем дополнительные события остановки для начала транспортного рейса и для его конца.

Определение 15. Длительность элементарной связи c определяется как $d(c) = \tau_a(c) - \tau_d(c)$.

На станции или любом другом транспортном узле $S \in B$ возможно совершить пересадку с одного поезда на другой, если время между прибытием и отбытием на станции S больше и равно минимальному времени пересадки – $\minTransferTime(S)$. Аналогично вводится максимальное время – $\maxTransferTime(S)$. Для простоты дальнейших рассуждений примем $\minTransferTime = const$ и $\maxTransferTime = const$.

Пусть $P = (c_1, \dots, c_k)$ будет последовательностью элементарных связей. Определим $dep_i(P) = \tau_d(c_i)$, $arr_i(P) = \tau_a(c_i)$, $S_d(P) = S_d(c_1)$, $S_a(P) = S_a(c_k)$, $Z_d(P) = Z_d(c_1)$, $Z_a(P) = Z_a(c_k)$, $dep(P) = dep_1(P)$, $arr(P) = arr_k(P)$ и $d(P) = arr(P) - dep(P)$. Таким образом, последовательность P будет называться согласованной связью между станциями $S_d(P)$ и $S_a(P)$, для которой выполняются следующие условия:

1. Станция отправления c_{i+1} является станцией прибытия c_i .
2. Для минимального времени пересадки соблюдается либо $Z_d(c_{i+1}) = Z_a(c_i)$, либо $dep_{i+1}(P) - arr_i(P) \geq \minTransferTime(S_a(c_i))$

Для того, чтобы построить из расписания граф, потребуется получить все маршруты поездов. Граф будет иметь 3 вида вершин, каждая из которых будет содержать время и принадлежать станции. Для каждой элементарной связи $c_1 = (Z_1, Z_2, S_1, S_2, \tau_1, \tau_2)$ из станции S_1 в станцию S_2 на одинаковом поезде мы добавим вершину отправления $S_1d@ \tau_1$ на станции S_1 со временем отправления τ_1 , вершину прибытия $S_2a@ \tau_2$ на станции S_2 со временем прибытия τ_2 и ребро $S_1d@ \tau_1 \rightarrow S_2a@ \tau_2$, обозначающее поездку на транспортном средстве со станции S_1 на S_2 . Если транспортное средство продолжает движение со станции S_2 во время τ_3 , то добавим ребро $S_2a@ \tau_2 \rightarrow S_2d@ \tau_3$, представляющее собой ожидание транспорта на станции S_2 . Это возможно, независимо от того, насколько мала разница $\tau_3 - \tau_2$.

Для каждой вершины отправления $S_2d@ \tau$ добавим вершину пересадки $S_2t@ \tau$ с тем же временем и добавим ребро $S_2t@ \tau \rightarrow S_2d@ \tau$ между ними. Также мы добавим ребро $S_2t@ \tau \rightarrow S_2t@ \tau'$, идущее ко следующей вершине пересадки по возрастаю времени. Такие ребра будут называть ребрами ожидания (или ребрами пересадки). Теперь для того, чтобы позволить совершить пересадку после прибытия в вершину $S_2a@ \tau_2$, добавим ребро к первой вер-

шине пересадки, для которой выполняется $\tau \geq \tau_2 + \minTransferTime(S_2)$ и $\tau \leq \tau_2 + \maxTransferTime(S_2)$. Это даст возможность, совершить пересадку на любой транспортный рейс.

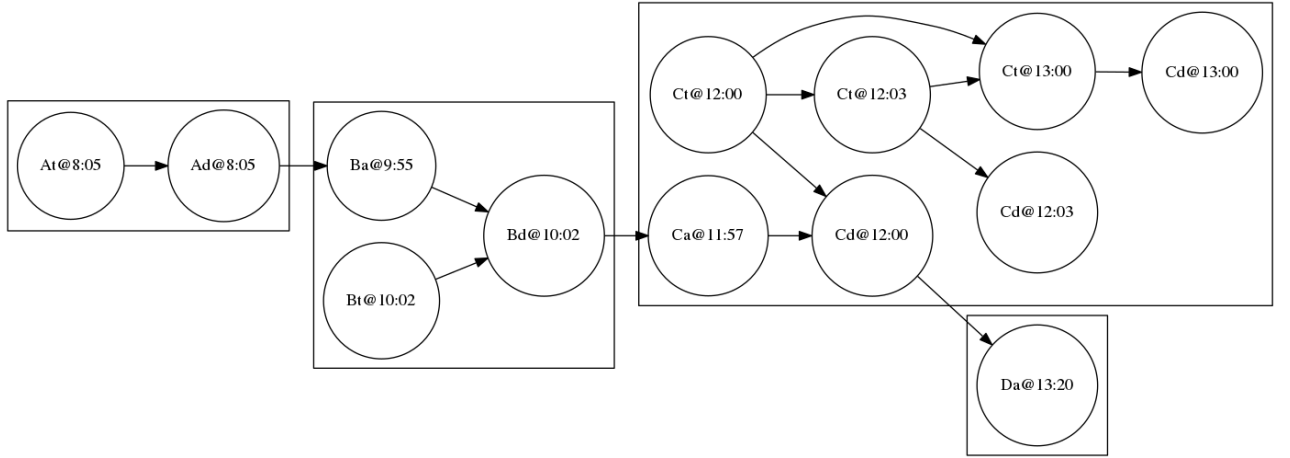


Рисунок 4 – Для каждого события остановки есть соответствующая вершина. Вес ребра неявно задан по определению длительности элементарной связи. В данном примере $\minTransferTime(C)$ равен 10 минутам.

2.1.3. Граф рейсов

Попробуем упростить и в то же время улучшить модель на основе графа расписаний. Заметим, что вершины пересадки St можно удалить из графа без потери информации. Для этого заменим все ребра вида $St@t \rightarrow St@t'$ на ребра $Sd@t \rightarrow Sd@t'$. Это возможно сделать, потому что для каждой вершины пересадки $St@t$ существует парная ей вершина отправления $Sd@t$ с одинаковым временем t . Также поменяем конец ребер вида $Sa@t \rightarrow St@t'$ на вершины отправления. Таким образом, поддерживать упорядоченный порядок по времени отправления нужно в самих вершинах отправления.

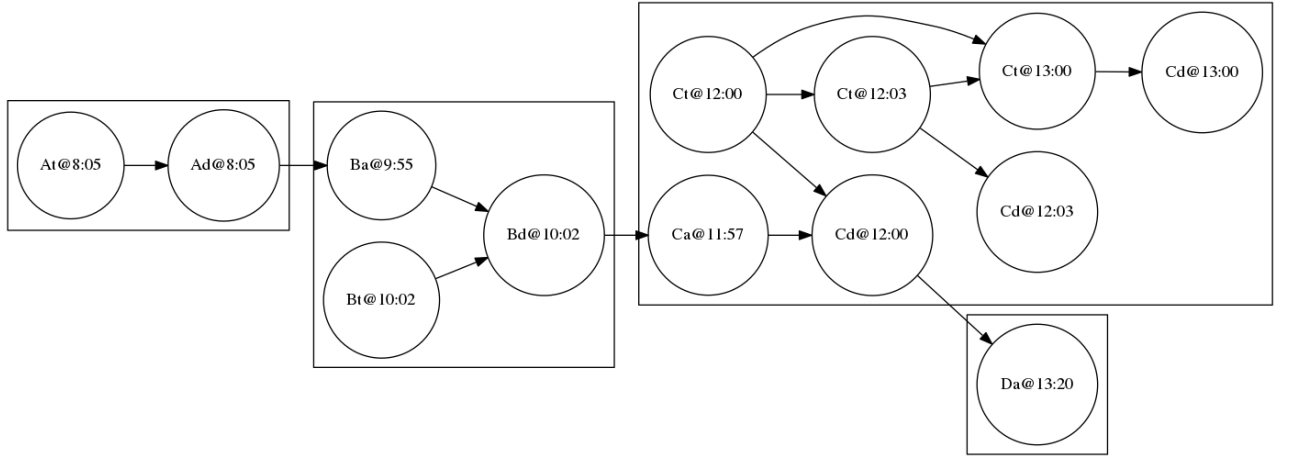


Рисунок 5 – Для каждого события остановки есть соответствующая вершина. Вес ребра неявно задан по определению длительности элементарной связи. В данном примере $\minTransferTime(C)$ равен 10 минутам.

Следующим шагом улучшения модели будет проведение операции под названием транзитивное замыкание.

Определение 16. В теории графов транзитивным замыканием графа будет являться добавление дополнительных ребер между всеми вершинами, между которыми существует путь.

Наивный способ транзитивного замыкания предполагает добавление ребер между всеми парами вершин отправления $S_i d @ \tau_i$ и прибытия $S_j a @ \tau_j$. Такие ребра будут называться кратчайшими. В них можно хранить информацию о K_{max} кратчайших маршрутах между парой транспортных узлов S_i и S_j , где K_{max} – максимальное количество маршрутов, которые могут быть запрошены клиентским приложением у строителя маршрутов.

Для того, чтобы сделать полное замыкание графа, нужно построить кратчайшие маршруты от каждой вершины отправления до каждой вершины прибытия. Для такой задачи существует алгоритм Флойда-Уоршелла, который требует $O(n^3)$ времени, где n – количество вершин¹ и $O(n^3 K_{max})$ памяти.

В итоге, несмотря на всю простоту модели, к сожалению, такой подход крайне неэффективно расходует память и требует огромное время на фазу предварительного расчета.

¹Улучшенная версия алгоритма требует $O(n^2 \log n)$ времени.

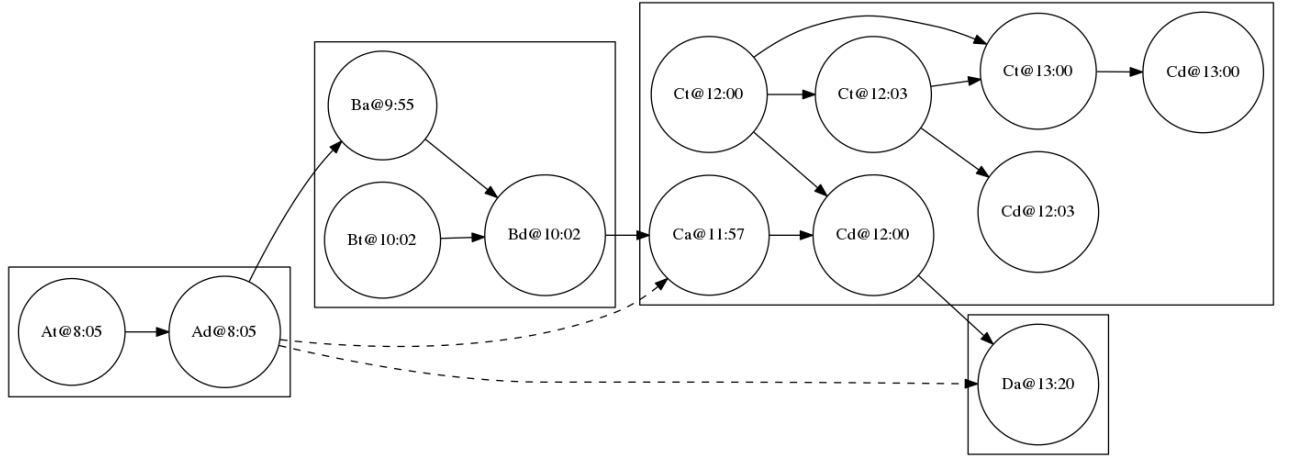


Рисунок 6 – Для каждого события остановки есть соответствующая вершина. Вес ребра неявно задан по определению длительности элементарной связи. В данном примере $\minTransferTime(C)$ равен 10 минутам.

Улучшенный способ транзитивного замыкания работает локально. Вместо того, чтобы добавлять ребра между всеми парами вершин отправления $S_i d @ \tau_i$ и прибытия $S_j a @ \tau_j$ во всем графе, будем добавлять их только между такими вершинами, построенными на основе конкретного транспортного рейса.

Оценим время и память требуемое для построения такой модели. Пусть q – количество остановок в одном транспортном рейсе, n – количество транспортных рейсов. Рассмотрим остановку Z_i , где i – номер остановки в транспортном рейсе. В худшем случае от каждой вершины отправления $S_i d @ \tau$ будет идти $q - i$ ребер до всех оставшихся вершин прибытия $\forall j > i : S_j a @ \tau_j$. Аналогично до каждой вершины $S_i a @ \tau$ будет идти i ребер от всех вершин прибытия $\forall j < i : S_j a @ \tau_j$. Не сложно заметить, что на один транспортный рейс таким образом будет приходиться $\frac{q^2}{2}$ ребер. Требуемое время, которое потребуется для построения модели, равно $O(q^2 n)$, не считая времени на получение минимальной вершины, а суммарная память – $O(q^2 n)$ для ребер и $O(qn)$ для вершин.

Листинг 3 – Алгоритм построения транзитивного замыкания

```

function CLOSURE(runs, stops)
  for run  $\in$  runs do
    for i = 0 to size(stops[run]) do
       $S \leftarrow stops[run][i]$   $\triangleright$  получаем из события остановки станцию
      Создаем вершину  $S_i a @ \tau_i$ 
      for j = 0 to i – 1 do
        создаем ребро  $S_j d @ \tau_j \rightarrow S_i a @ \tau_i$ 
      end for
      Создаем вершину  $S_i d @ \tau_i$ 
      for j = i + 1 to size(stops(run)) do
        создаем ребро  $S_i d @ \tau_i \rightarrow S_j a @ \tau_j$ 
      end for
       $minNode \leftarrow getMinNode(S_i d @ \tau_i)$   $\triangleright$  получаем минимальную
      вершину отправления в порядке сортировки по времени
      Создаем ребро пересадки  $S_i a @ \tau_i \rightarrow minNode$ 
    end for
  end for
end function

```

Попробуем изменить подход и использовать единую вершину остановки. Для этого назовем элементарной остановкой кортеж из 4 элементов $St = (i, r, s, \tau)$, хранящий информацию о номере остановки i , транспортном рейсе r , станции s , а также метки времени τ , которая будет хранить либо время отправления, либо прибытия в зависимости от того, как мы будем её интерпретировать. Задачу о хранении возможных пересадок перенесем из модели графа в модели транспортных рейсов, для которых известны также сами события остановок. Создадим по массив в транспортном рейсе размером с количество его остановок, который будет хранить неупорядоченные списки элементарных остановок.

Также нам потребуется хранить 2 структуры данных с поддержкой быстрого поиска элементов в определенной границе. Такой структурой данных может быть красно-черное дерево, поддерживающее поиск элемента за $O(\log n)$, или аналогичный по асимптотике и возможностям список с пропусками. Обе эти структуры помогут быстро находить элементарные остановки во множестве событий прибытия $T_a(s)$ и отбытия $T_d(s)$ на станцию s .

Рассмотрим, как теперь будет строиться модель. Построение модели можно реализовать инкрементально, то есть добавлять в модель по одному

Листинг 4 – Алгоритм построения пересадок

```

function BUILDTRANSFERS(runs)
  for  $r \in runs$  do
     $transfers[r] := \emptyset$   $\triangleright$  Список пересадок для транспортного рейса  $r$ 
    for  $i = 0$  to  $|W(r)| - 1$  do
       $w := W(r)[i]$ 
       $s := S(w)$   $\triangleright$  по точке маршрута можно получить станцию
      if  $i \neq 0$  then  $\triangleright$  игнорируем  $i = 0$ , так как только сели на поезд и
      пересадку делать не нужно в любом случае
         $st_a := (i, R, s, \tau_a(w))$  Текущая элементарная остановка со вре-
        менем прибытия
         $T_a(s).put(\tau_a(w), st_a)$  Помещаем остановку в расписание прибы-
        тий станции  $s$ 
        for  $(\tau, st') \in T_a(s) : \tau - \tau_a(w) \in [T_{min}(s), T_{max}(s)]$  do
          if  $R(st') \neq r$  then  $\triangleright$  Исключаем пересадку на текущий
          транспорт
             $transfers[i].add(st')$ 
          end if
        end for
      end if
      if  $i \neq |W(r)| - 1$  then  $\triangleright$  не делаем пересадки на поезд, который
      уже находится на своей конечной станции
         $st_d := (i, R, s, \tau_d(w))$ 
         $T_d(s).put(\tau_d(w), st_d)$ 
        for  $(\tau, st') \in T_a(s) : \tau - \tau_d(w) \in [-T_{max}(s), -T_{min}(s)]$  do
          if  $R(st') \neq r$  and  $i \neq 0$  then  $\triangleright$  игнорируем  $i = 0$ , так как
          только сели на поезд и пересадку делать не нужно в любом случае
             $transfers[i].add(st_d)$ 
          end if
        end for
      end if
    end for
  end for
end function

```

транспортному рейсу. Для каждого такого рейса создается массив *transfers*, который хранит на i позиции список элементарных остановок отправления, на которые можно пересест с текущего рейса на i точке маршрута. Пустой список означает, что либо в точке маршрута другой транспорт не останавливается, либо он не подходит по критериям пересадки, то есть не соблюдаются допустимые интервалы времени, связанные с T_{min} и T_{max} . Для каждого рейса

выполняется обход его маршрутных точек с обновляем пересадок в каждой из них.

2.2. Построение маршрутов

Теперь у нас есть эффективная модель в виде графа рейсов для хранения информации о транспортной сети. На каждой станции поддерживается несколько структур данных для определения прибывающих и отбывающих поездов по конкретному интервалу времени. Также для каждого транспортного рейса можно быстро узнать возможные пересадки на пути всего следования поезда. Попробуем придумать эффективный алгоритм поиска маршрутов на заданной модели.

2.2.1. Дополнение временных интервалов

Типичный запрос к построителю маршрутов выглядит как кортеж из 5 элементов $q = (s_d, t_d, s_a, t_a, k)$, где s_d и s_a – станции отправления и прибытия соответственно, $t_d = (\tau_{d1}, \tau_{d2})$ и $t_a = (\tau_{a1}, \tau_{a2})$ – требуемые интервалы времени, k – количество маршрутов. Временные интервалы бывают 3 типов:

1. Фиксированные
2. Полубесконечные
3. Бесконечные

Листинг 5 – Приведение интервалов к фиксированному виду

```
function NORMALIZETIME( $t_a, t_d$ )
  if  $t_{d1} = -\infty$  or  $t_{d1} < T_{now}$  then
     $t_{d1} := T_{now}$ 
  end if
  if  $t_{a2} = \infty$  or  $t_{a2} > T_{end}$  then
     $t_{a2} := T_{end}$ 
  end if
  if  $t_{d2} = \infty$  or  $t_{d2} > T_{end}$  then
     $t_{d2} := t_{a2}$ 
  end if
  if  $t_{a1} = \infty$  or  $t_{a1} < T_{now}$  then
     $t_{a1} := t_{d1}$ 
  end if
end function
```

Фактически бесконечность в данном случае условная, потому что время построителя маршрутов всегда ограничено снизу текущим временем построения.

теля T_{now} , а сверху – временем конца продаж $T_{end} = T_{now} + T_{len}$, где T_{len} – интервал продажи билетов. Для упрощения работы с интервалами имеет смысл привести их к фиксированному типу².

2.2.2. Фаза инициализации

Идея алгоритма состоит в следующем: будем генерировать состояния, в которых можем находиться на пути следования до точки назначения. В процессе генерации будем осуществлять фильтрацию маловероятных и невозможных состояний с точки зрения требований к маршруту.

В качестве допустимого состояния будет выступать кортеж из двух элементов $e = (st, m)$, где st – элементарная остановки, m – количество совершенных пересадок.

В процессе работы нам потребуется множество вспомогательных структур данных, в которые будут записываться промежуточные результаты. Часть из них будет являться пересадочными массивами. Под этим термином подразумеваются массивы, индекс в которых будет указывать на текущее число рассматриваемых пересадок. Например, элемент массива *array*[3] будет хранить данные для состояний, в которых уже совершили 3 пересадки.

- *expanded* – пересадочный массив множеств посещенных состояний;
- *explored* – пересадочный массив множеств рассмотренных состояний
- *queue* – очередь состояний;
- *pathsCount* – пересадочный массив хеш-таблиц для остановок, хранящий число возможных маршрутов;
- *successors* – пересадочный массив хеш-таблиц для остановок, хранящий список прямых сегментов;
- *predecessors* – пересадочный массив хеш-таблиц для остановок, хранящий список обратных сегментов;

2.2.2.1. Дополнительные модели

Помимо структур данных нам потребуются несколько новых моделей, в которых мы будем хранить совершаемые действия, чтобы прийти в некоторое состояние e . Такие модели будем называть сегментами. В общем случае будет существовать 2 вида сегментов:

²Для чего это сделано будет более понятно в следующей главе с конкретной реализацией

1. **Прямой сегмент** ($A - B - C$). Он будет определять действия, совершаемые в процессе обхода графа от событий отправления. Он потребуется нам при сортировке по времени отправления;
2. **Обратный сегмент** ($B - C - D$). Он будет определять действия, совершаемые в процессе обхода графа от событий прибытия. Он потребуется нам для всех остальных видов сортировок;

Каждый сегмент состоит из 3 элементарных остановок. Остановка A является событием отправления 1 поезда, B – его прибытием, C – это отправление 2 поезда, D – прибытие. Можно заметить, что часть $B - C$ совпадает у обоих сегментов. Эта часть является неявным событием пересадки, с помощью которого можно однозначно конвертировать сегменты друг в друга, а также находить пересечения. Как будет показано позже, это поможет реализовать двунаправленный обход графа рейсов.

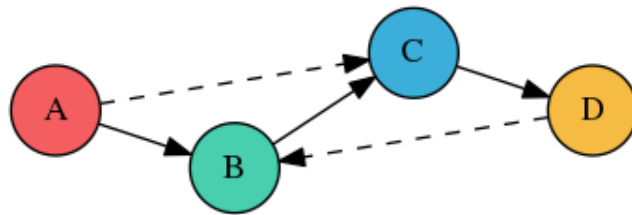


Рисунок 7 – Совмещение прямого и обратного сегментов. Пунктирные линии показывают следующее состояние, которое будет добавлено в очередь.

На самом деле любая сортировка может быть выполнена с наличием только одного вида сегментов, потому что путем их обхода можно из одного вида получить другой.

2.2.2.2. Заполнение вспомогательные структур

Перед фазой обхода необходимо произвести инициализацию необходимых данных. Эта операция будет происходить один раз на каждый запрос q .

Во-первых, должны быть начальные состояния, чьи остановки st удовлетворяют запросу q и приведены к фиксированному типу. Аналогично, требуется создать конечные состояний (если обход графа будет с конца). Во втором случае возникает проблема из-за того, что зная какие конечные остановки st' , мы не знаем, при каком количестве пересадок m они совершаются. В таком случае нам придется создать состояния для всех возможных вариантов,

то есть $\forall m : m \in [0, M]$, где M – максимально допустимое количество пересадок.

Во-вторых для некоторых эвристических оптимизаций будет полезно сохранить список начальных и конечных транспортных узлов, чтобы не проверять каждый раз непосредственно множества состояний.

Листинг 6 – Заполнение вспомогательных структур

```

for  $s \in sources$  do
     $departures.add(P(s))$     ▷ Добавляем транспортный узел в список узлов
                             отправлений
     $pathsCount[0].put(s, 1)$  ▷ Из остановки можно доехать в саму себя за 0
                             пересадок только 1 способом
     $st = (s, 0)$              ▷ Создаем состояние с событием отправления  $s$  и 0
                             совершенных пересадок
     $explored.add(st)$         ▷ Добавляем состояние в список рассмотренных
     $queue.add(st)$           ▷ Добавляем в очередь на обработку
end for
for  $t \in targets$  do
     $arrivals.add(P(t))$      ▷ Добавляем транспортный узел в список узлов
                             прибытия
end for

```

2.2.3. Фаза обхода

В фазе обхода мы начинаем рассматривать в произвольном или упорядоченном порядке все доступные на текущий момент состояния. Соответственно состояния берутся из очереди *queue*, которая может быть обычным связным списком или двоичной кучей. Для простоты будет рассматривать первый вариант, где порядок рассмотрения определяется порядком добавления в очередь. Одним из основных критериев остановки является то, что очередь стала пустой. Это говорит нам о том, что мы рассмотрели все возможные состояния, подходящие под требования запроса q , или не может сгенерировать новые.

Для примера рассмотрим прямой обход графа рейсов. После извлечения очередного состояния $v = (st, m)$ добавим его во множество посещенных состояний *expanded*. Теперь нужно сделать проверку, что не остановка st из состояния v не совершается в одном из конечных транспортных узлов. В противном случае, можно перейти к рассмотрению следующего состояния. Получим по остановке st все возможные прямые сегменты, где $A = st$.

Для каждого прямого сегмента рассмотрим остановку отправления второго поезда C . Следует заметить, что сегмент либо является пересадочным, либо $I(A) < I(C)$. Это следует из того, что $R(A) = R(C)$ и $T(A) \leq T(C)$ для любого сегмента. Если сегмент пересадочный, то новое количество пересадок $m' = m + 1$. Далее следует проверить, что состояние $v' = (C, m')$ отсутствует в списке посещенных состояний. Если это так, то можно добавить в список прямых сегментов $successors[m]$ текущий рассматриваемый сегмент. Аналогично, можно сделать для обратного сегмента и $predessors[m']$ ³. Формально это соответствует следующей динамике:

$$predessors[m'] [C] = \{ \langle A, m \rangle \mid \exists \langle A, m \rangle \rightarrow \langle C, m' \rangle \}$$

Аналогично, можно подсчитать количество путей:

$$pathsCount[m'] [C] = pathsCount[m] [A] + pathsCount[m'] [C];$$

На последней стадии новое состояние v' можно добавить в очередь *queue*, предварительно проверив, что оно отсутствует в списке рассмотренных состояний. Это предотвратит экспоненциальный рост очереди.

Листинг 7 – Фаза обхода графа рейсов

```
function FILLLEVEL(level, k)
  for t ∈ targets do
    incomming := predecessors[k][t];
    for s ∈ incomming do
      level.add(k, s)
    end for
  end for
end function
```

Теперь рассмотрим более подробно получение списка всех возможных прямых сегментов по остановке A . В общем виде остановка $A = (i, r, s, \tau)$, поэтому мы можем получить список маршрутных точек $W(r)$ транспортного рейса r . Далее начнем рассматривать каждую такую точку с номерами больше i , так как префикс маршрута, включающий точки с 0 до i уже пройдены поездом. Для каждой маршрутной точки $w \in W(r)$ известен транспортный

³В целях оптимизации памяти следует заполнять только один вид сегментов для соответствующего прямого или обратного обхода графа рейсов.

узел $S(w)$, поэтому мы можем проверить, чтобы $S(w)$ не входили в списки станций отправления и прибытия. Для этого у нас имеются множества *departures* и *arrivals*, так как там не собираемся делать пересадку. Стоит отметить, что могут существовать случаи, когда маршрут делает петлю, посещает одну и ту же станцию несколько раз, но в рамках этой работы будем считать, что они отсутствуют. Иначе бы было необходимо различать такие маршрутные точки. Сделать это можно было бы по временной метке или номеру. В данный момент следует указать, что прямые и обратные сегменты имеют еще 2 вида:

1. **Конечный сегмент.** Он будет определять действие, когда мы уже находимся на транспорте в точке отправления и не делаем пересадку.
2. **Пересадочный сегмент.** Аналогичен предыдущему виду за исключением того, что пересадка на другой транспорт делается.

Тогда в случае, когда одна из маршрутных точек является конечной станцией, то можно создать конечный сегмент. Учитывая, что сегмент хранит 3 остановки, то прямой конечный сегмент будет выглядеть как $A - C - C$.

Теперь можно из графа рейсов получить через $T_a(S(w))$ и $T_d(S(w))$ списки элементарных остановок прибытия и отбытия и сгенерировать все оставшиеся пересадочные сегменты. Для этого можно воспользоваться массивом *transfers*, который присутствует у каждого транспортного средства. Тогда остановка A известна заранее, B получается из текущей маршрутной точки, а C берется из данного массива. В конце все полученные сегменты следует отфильтровать, что будет описано в данной главе позже.

Листинг 8 – Получение списка прямых сегментов

```

function SUCCESSORS(departures, A, arrivals)
  for  $t \in \text{targets}$  do
    incomming := predecessors[ $k$ ][ $t$ ];
    for  $s \in \text{incoming}$  do
      level.add( $k$ ,  $s$ )
    end for
  end for
end function

```

2.3. Сортировка маршрутов

По постановке задачи маршруты требуется строить инкрементально в порядке сортировки. Виды сортировок были описаны в обзоре данной работы. Все дополнительные структуры данных, полученные на предыдущем этапе, потребуются теперь для реализации фазы построения и сортировки маршрутов. Из-за того, что все предикаты сортировок известны нам заранее, мы можем придумать и реализовать необходимый алгоритм для каждого из них, который будет эффективнее абстрактного аналога. Рассмотрим каждый вид сортировки подробнее.

2.3.1. Количество пересадок

Как было сказано ранее, данный вид сортировки является наиболее простым из представленных, потому что на фазе обхода графа рейсов заполнение списков прямых и обратных сегментов выполнялось в естественном порядке из-за пересадочных массивов. Тогда мы можем инкрементально строить маршруты в таком же порядке.

Для начала на данной фазе нам потребуются дополнительные модели. Будем называть их слоями. В общем случае слои будут содержать концы сегментов (прямых или обратных), по которым можно деструктивно итерироваться, то есть производить обход с удалением не нужных сегментов. Всего 2 вида слоев:

1. **Произвольный слой.** Имеет доступ к сегментам в порядке их добавления.
2. **Упорядоченный слой.** Имеет доступ к сегментам, сортированных по их остановкам или прочим параметрам.

Теперь определим функцию, которая производит заполнение слоя с помощью множества обратных сегментов. Основная идея в том, чтобы обойти все конечные элементарные остановки прибытия и их обратные сегменты для каждого количества совершенных пересадок. Таким образом, сегменты $A - B - C$, где состояния $v = (C, m)$ для остановок C имеют меньшее количество пересадок m , будут в начале слоя.

Для того, чтобы начать восстановление (построение) маршрутов нужно создать и заполнить корневой слой. Для сортировки по количеству пересадок будем оперировать произвольными слоями. Мы хотим поддерживать для каждого вида сортировки следующие направления: восходящую (ASC) и

Листинг 9 – Заполнение переданного слоя

```
function FILLLEVEL(level, k)  
  for t ∈ targets do  
    incomming := predecessors[k][t];  
    for s ∈ incomming do  
      level.add(k, s)  
    end for  
  end for  
end function
```

нисходящую (DESC). В таком случае, для текущего вида сортировки можно просто совершать заполнение корневого слоя по возрастанию количества пересадок (ASC) или по убыванию (DESC).

Листинг 10 – Подготовка сортировки по количеству пересадок

```
function SORTEDBYTRANSFERSCOUNT(level, k)  
  unvisited := ∅ ▷ случайный уровень  
  if direction = ASC then  
    for i = 0 to maxTransfersCount do  
      FILLLEVEL(unvisited, i)  
    end for  
  else  
    for i = maxTransfersCount downto 0 do  
      FILLLEVEL(unvisited, i)  
    end for  
  end if  
  return unvisited  
end function
```

2.3.2. Время прибытия

2.3.3. Время отправления

Листинг 11 – Строим и возвращаем следующий маршрут по множеству исходящих сегментов

```

function FORWARDNEXT(level, k)
  while stack  $\neq \emptyset$  do
    level := stack.peek()           ▷ получаем текущий уровень в стеке
    if level  $\neq \emptyset$  then
      st = level.poll()             ▷ извлекаем состояние из уровня
      b := st.segment.B()
      t := st.transfers
      if arrivals содержит P(b) then           ▷ Маршрут найден, так как
пришли в точку отправления
        path = BUILDFROM(state, true)           ▷ Строим маршрут
        candidates.add(path)                   ▷ Добавляем маршрут в список
кандидатов
      if |path| > 1 then
        stack.poll()
      end if
    end if
    outgoing := successors[t][b];
    if outgoing  $\neq \emptyset$  then
      level' :=  $\emptyset$                      ▷ создаем новый случайный уровень
      for segment  $\in$  outgoing do
        if segment является пересадкой then
          t' = t + 1
        else
          t' = t
        end if
        st' := (st, segment, t')
        level'.add(st')
      end for
      stack.push(level')
    end if
  else
    stack.poll()   ▷ на текущем уровне не осталось состояний, можно
подняться выше
  end if
end while
end function

```

2.3.4. Время в пути

2.4. Построение фильтров

Построение фильтров также, как и поиск маршрутов, состоит из двух фаз: предварительного расчета и основной. В первом случае модели данных,

которые отвечают за транспортные средства переводятся в удобные и компактные модели, которые будут использоваться при непосредственном поиске маршрутов. Во втором случае готовые структуры данных будут позволять на этапе фазы обхода графа рейсов собирать композитные фильтры для каждой остановки с помощью динамического программирования. Также как и случае подсчета количества маршрутов мы построим динамику, которая будет позволять определять доступные параметры фильтрации для пары элементарных остановок.

2.4.1. Косвенные признаки

В фазе предварительного расчета мы будем извлекать косвенные признаки из моделей транспортных средств. Как было описано в главе 1, такими признаками может быть тип транспорта, номер поезда или тип места в самолете и т.д. Для удобной работы с ними мы разделим все признаки по функциональным типам, то есть, например, признак "перевозчик" у поезда и самолета будут отнесены к разным типам признаков. Это позволит нам логически отделять на этапе отдачи результата клиентскому приложению.

В качестве компактной модели свойств $Pr = \{(t, \{(v, c)\})\}$, хранящей признаки, будет выбрана хеш-таблица из типа признака t в хеш-таблицу значений признака v в количество свободных мест c , которые удовлетворяют ему. Например, есть 1 поезд с 3 вагонами (один вагон имеет 20 мест): 2 из них купе, 1 - плацкарт. Из них в плацкарте половина верхних мест, а другая половина - нижних. Тогда будет создан объект свойств для поезда со следующими записями:

- Тип транспорта
 - (Поезд, 60)
- Тип вагона
 - (Купе, 40)
 - (Плацкарт, 20)
- Тип места
 - (Верхнее, 20)
 - (Нижнее, 20)

Как можно заметить на данном примере, свойства не хранятся в виде иерархии функциональных зависимостей. Это сделано для экономного расхода памяти, а также для упрощения операций со свойствами в процессе вы-

полнения динамики по свойствам. Функциональные зависимости известны заранее и задаются вручную в специальном объекте, о котором речь пойдет позже.

Теперь нужно ввести 2 операции в пространстве свойств признаков, которые будут называться минимум и максимум. Обе операции будут являться ассоциативными, то есть:

$$(x \circ y) \circ z = x \circ (y \circ z) \text{ для любых элементов } x, y, z.$$

Минимум на свойствах работает как минимум количества свободных мест по всем парам признаков $\langle t, v \rangle$. При этом работает правило дополнения, то есть если пара присутствует в свойстве A , но отсутствует в B , то берется число свободных мест из A .

Листинг 12 – Минимум из пары свойств

```

function MIN( $Pr_a, Pr_b$ )
  if  $Pr_a = \emptyset$  then
    return  $Pr_b$ 
  end if
  if  $Pr_b = \emptyset$  then
    return  $Pr_a$ 
  end if
   $Pr_r := \emptyset$ 
  for  $\langle t, v \rangle : Pr_a \cup Pr_b$  do
     $Pr_r[t][v] = \min(Pr_a[t][v], Pr_b[t][v])$ 
  end for
  for  $\langle t, v \rangle : Pr_a \setminus Pr_b$  do
     $Pr_r[t][v] = Pr_a[t][v]$ 
  end for
  for  $\langle t, v \rangle : Pr_b \setminus Pr_a$  do
     $Pr_r[t][v] = Pr_b[t][v]$ 
  end for
  return  $Pr_r$ 
end function

```

▷ Результирующее свойство

Максимум на свойствах работает аналогично минимуму, только для количества свободных мест операция меняется на максимум.

2.4.2. Осуществление фильтрации

В фазе обхода при построении маршрутов для каждой элементарной остановки генерировались все возможные сегменты, которые являлись пересадочными и конечными. Таким образом, чтобы избежать попадания в следующую элементарную остановку, нужно фильтровать соответствующий сегмент, поэтому в процессе генерации каждому сегменту должно сопоставляться свойство, хранящее признаки для транспортного средства, отправляющегося из остановки A^4

В итоге, мы должны хранить свойства для каждого отрезка пути, соединяющего смежные маршрутные точки. Так как сегмент является множеством последовательно соединенных отрезков пути следования, то свойство сегмента является минимумом из всех свойств отрезков, входящих в него:

$$Pr_s = \min_{i \in s} Pr_i$$

Для того, чтобы ускорить построение свойства сегмента можно воспользоваться деревом отрезков, позволяющее находить результат операции на интервале за $O(\log q)$ времени и требующее $O(2q)$ памяти, где q – число маршрутных точек транспортного рейса. В качестве операции будет взята операция минимума свойств. Это возможно, так как определенный ранее минимум ассоциативен, так же имеет нейтральный элемент – пустое свойство (пустую хеш-таблицу). Таким образом, множество свойств является моноидом, который можно применять в дереве интервалов.

2.4.3. Функциональные зависимости

2.4.4. "Белые" и "черные" фильтры

Выводы по главе 2

⁴Для упрощения рассматривается прямой сегмент, для обратного будет остановка прибытия D .

ГЛАВА 3. ДЕТАЛИ РЕАЛИЗАЦИИ И ТЕСТИРОВАНИЕ

3.1. Работа с базой данных

3.1.1. Особенности базы данных

Краткий принцип работы VoltDB, что такое in-memory и newSQL

3.1.2. Персистентные модели данных

Почему все модели транспортной сети персистентные (иммутабельные).
Необходимость такого подхода.

3.1.2.1. Транспорт

Иерархия моделей. Разделение на модель и сущность.

3.1.2.2. Остановки

Модель остановки, страницы остановок, нормализация времени в период.

3.1.2.3. Пересадки

Как строим на основе страниц остановок.

3.1.3. Кэши

Зачем нужно? Географическая распределенность системы.

3.1.3.1. Для моделей данных

Время жизни объекта.

3.1.3.2. Для запросов

Хранение дерева сегментов по ключу запроса. Генерация ключей.

3.2. Генерация карт транспортных сетей

3.2.1. Генерация транспортных узлов

Равномерная 2D генерация точек. Генерация признаков узлов.

3.2.2. Генерация транспортных рейсов

Прокладка естественных маршрутов.

3.2.3. Генерация центральных узлов

Генерация точек-городов.

3.3. Результаты тестирования

Время запроса с 10с до 2с, прикрепить 3 графика. Сравнение со старой системой, сравнение с Йеном.

Выводы по главе 3

Все сделано и работает. Скорость работы возросла, новые функции появились. Покрыто интеграционными и авто тестами.

ЗАКЛЮЧЕНИЕ