Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К курсовой работе на тему:

«Поиск кратчайшего расстояния между городами»

Руководитель курсового проекта _____ Кузнецова О.В.

(Подпись, дата) (И.О.Фамилия)

Студент _____ Квасников А.В.

(Подпись, дата) (И.О.Фамилия)

Москва, 2019

Введение

Основной целью данной работы является исследование графовых баз данных (Graph DB) и их интеграции внутрь программ.

В результате было сделано веб-приложение, позволяющее заполнить базу данных узлами (nodes) и их связями, после чего найти кратчайшее расстояние между двумя любыми узлами. Для наглядности, данный функционал был представлен в виде поиска кратчайшего расстояния между городами.

1. Аналитический раздел

В данной работе был очень важен выбор технологий, обеспечивающих работу приложения, так как проект представляет собой связь 3 компонент:

- серверная часть;
- клиентская часть;
- база данных.

1.1 Особенности выбора БД

Главным требованием к программе является корректный поиск кратчайшего расстояния между узлами. «Сеть» городов, объединенных связями друг с другом, проще всего представить в виде графа: существует множество алгоритмов для работы над графами. Также, графовая модель имеет преимущество в наглядности представления. Вследствие этого было принято решение использовать графовую базу данных, ведь при работе с «естественными» графовыми структурами, графовые БД обладают производительностью, значительно превосходящей производительность реляционных БД. Кроме того, как было сказано выше, значимую роль играет наглядность представления данных - без сомнения, любую «сеть» из соединенных друг с другом объектов намного легче воспринимать при графическом отображении в виде графа, а не, например, в виде таблиц.

1.2 Структура БД

Данные, необходимые для работы приложения, хранятся в БД в виде графа.

На *рисунке 1* показана схема БД. В нашем случае требуется только две «сущности»: однотипные узлы (города) и однотипные отношения между ними

(дороги). Кроме того, для полноты графа необходимо создавать по два отношения между городами - ведь neo4j использует модель ориентированных графов, то есть рёбра имеют направление.

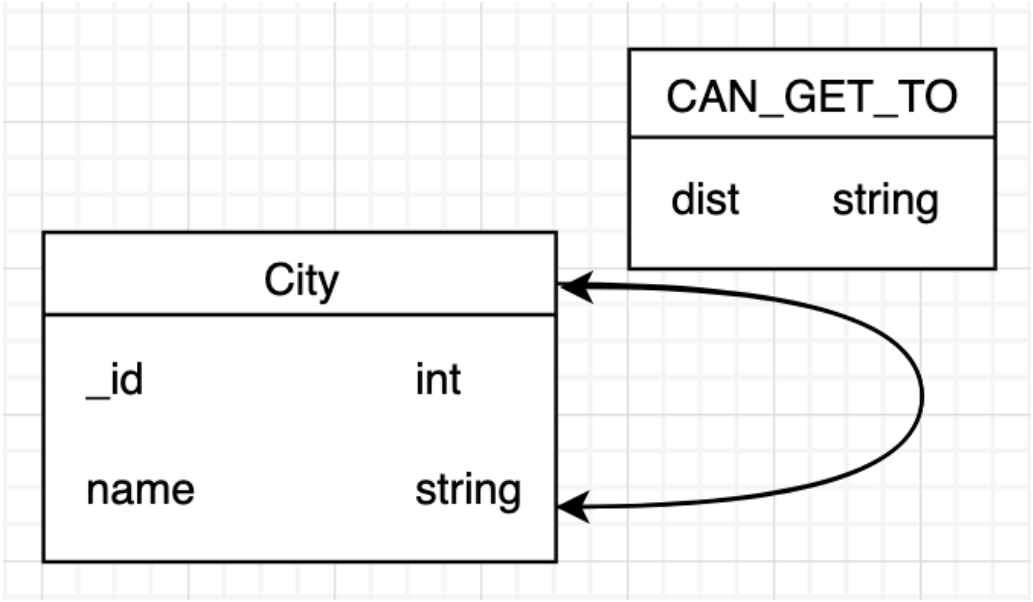

Рисунок 1: схема БД

Города - узлы БД (node). Дороги - отношения (relations). Это показано на *рисунке 2*.

На *рисунке 2* видно, что БД формирует граф из полученных данных. Это позволяет использовать алгоритмы из теории графов максимально эффективно. Ведь поиск пути является сложным запросом в sql: нам необходимо многократно обращаться к таблицам. Графовые БД, напротив, хранят все узлы отдельно. Таким образом, они эффективны для работы с данными, имеющими множество связей между собой. Чем больше связей необходимо пройти, тем эффективнее и рациональнее использование графовой БД в проекте.

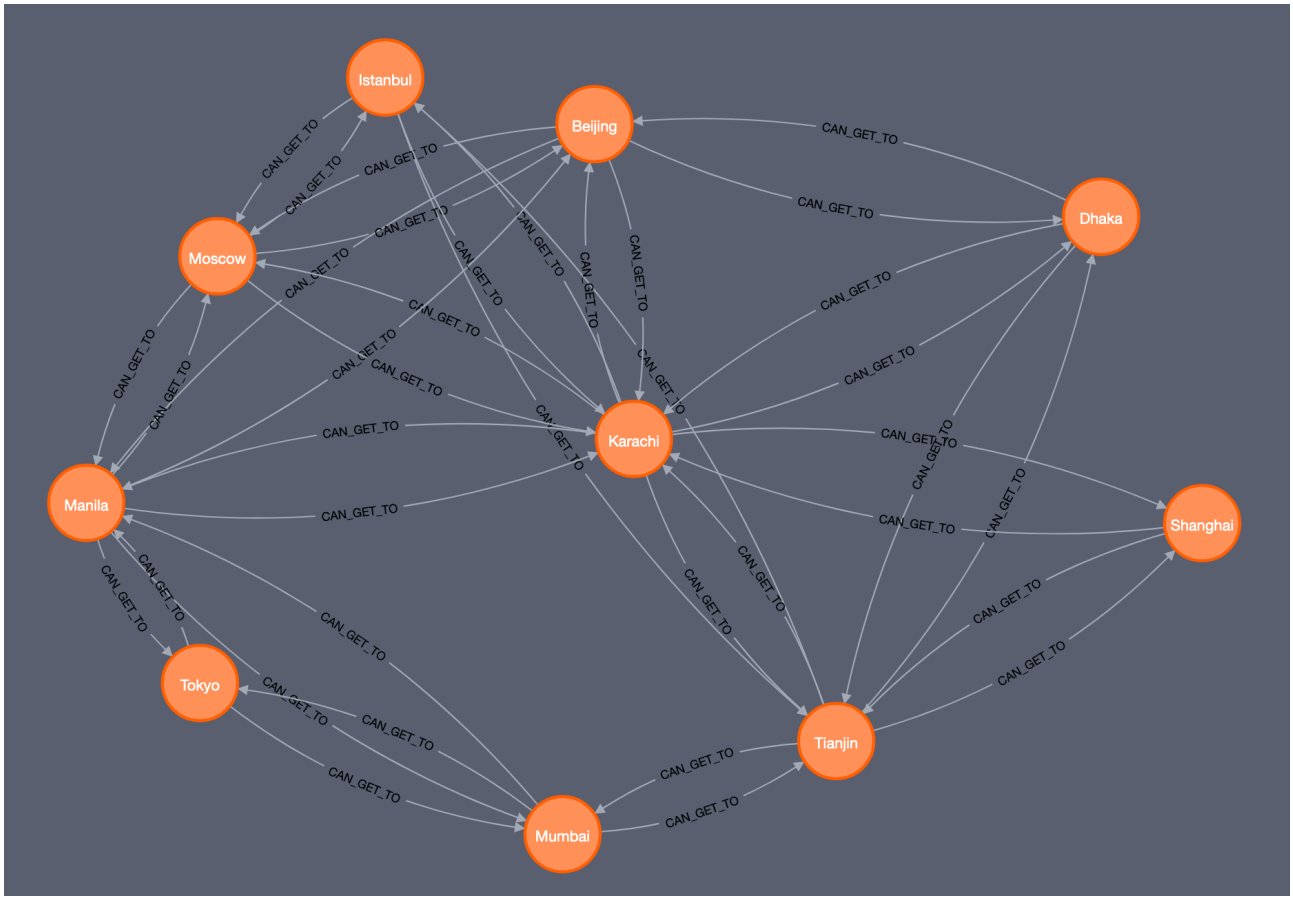

Рисунок 2: представление данных в БД

Данные в БД загружаются из заранее сгенерированного csv-файла. Для подобных задач в neo4j используются специальные библиотеки, такие, как apoc. Данная библиотека, в частности, имеет большое количество функций, реализующих алгоритмы теории графов.

1.3 Особенности выбора технологий, обеспечивающих работу приложения

Веб - одно из самых распространенных направлений в современном IT мире. Из наиболее популярных фреймворков и библиотек, подходящий для поставленной задачи, можно выделить следующие:

1. Ruby on Rails - фреймворк, написанный на языке программирования Ruby, один из основных принципов которого - максимальное использование механизмов повторного использования - не является необходимым в рамках небольшого приложения, не требующего большого объема кода.

2. Flask - микро-фреймворк, написанный на языке Python. Он идеально подходит для разработки небольших приложений, так как обладает небольшим размером «базы». Также, Neo4j поддерживает работу с python. Однако, создание front-end части потребует изучения других технологий и языков программирования. Наиболее популярным решением для реализации клиентской части являются js фреймворки и библиотеки.

3. Node.js - программная платформа, превращающая JavaScript из узкоспециализированного языка в язык общего назначения, позволяет подключать внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Neo4j поддерживает Node.js, существует neo4j-driver, обеспечивающий связь между сервером, написанном JavaScript, и базой данных Neo4j. Особенности работы и настройки neo4j-driver описаны на официальном сайте. Кроме того, язык JavaScript можно использовать как на серверной, так и на клиентской части приложения. Одной из самых распространенных js-библиотек для реализации фронтенда является React.js. Он идеально подходит для связки с Node.js.

Было решено использовать Node.js, так как он:

- Идеально интегрируется с Neo4j;

- Обладает огромным сообществом, множеством статей, посвященных тем или иным деталям разработки;

- Крайне перспективен - его изучение может пригодиться для дальнейших работ.

1.4 Задача о кратчайшем пути

Задача о кратчайшем пути - задача поиска самого короткого пути между двумя вершинами на графе, в которой минимизируется сумма весов рёбер, составляющих путь. Она является одной из важнейших задач теории графов. В частности, это объясняется тем, что ее решение имеет практическую пользу. На примере нашего приложения: есть города (вершины, или узлы графа), соединенные друг с другом дорогами (ребра графа). Необходимо найти кратчайшее расстояние между двумя городами. Решение данного примера может повлиять, например, на маршрут доставки товаров по оптимальному маршруту с целью сэкономить на дорожных расходах, таких, как топливо.

Задача о кратчайшем пути может быть определена для неориентированного, ориентированного или смешанного графа. Так как дороги, как правило, имеют двустороннее движение, мы будем рассматривать задачу на неориентированном графе, то есть мы будем считать, что если из города A можно попасть в город B по прямой связи (то есть без посредников в виде других городов) или по сложному маршруту, включающему другие города, то и из города B можно попасть в город A по такому же пути. Также нужно отметить, что в роли длины ребра может выступать не только расстояние, но и время, расходы на топливо и множество других величин. Например, в качестве длины ребра между городами можно использовать приближенное время перемещения между этими городами по определенному маршруту.

1.5 Выбор алгоритма для решения задачи о кратчайшем пути

Существует множество алгоритмов для решения задачи о кратчайшем пути. Наиболее распространенными на практике оказываются следующие алгоритмы:

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

2. Алгоритм Беллмана-Форда - алгоритм поиска кратчайшего пути в взвешенном графе. Алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана — Форда допускает рёбра с отрицательным весом.

3. Алгоритм Флойда-Уоршелла - динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Разработан в 1962 году Робертом Флойдом и Стивеном Уоршеллом.

По краткому описанию алгоритмов мы можем сразу отбросить 1 из них: алгоритм Флойда-Уоршелла находит кратчайшие расстояние между всем вершинами, а в решении нашей задачи такой необходимости нету. Нас интересует расстояние из одного фиксированного города.

Оставшиеся два алгоритма отличаются только тем, что алгоритм Беллмана-Форда способен работать с отрицательными весами рёбер. Так как расстояние между городами не может быть отрицательным, отрицательных рёбер в нашем графе нет. Следовательно, нет необходимости использовать алгоритм с подобным дополнением.

Таким образом, для решения задачи о кратчайшем пути в рамках поставленных условий (кратчайшее расстояние между городами) был выбран алгоритм Дейкстры.

1.6 Алгоритм Дейкстры

Рассмотрим граф, изображенный на *рисунке 3*. Он представляет собой узлы, соединенные направленными связями (можно использовать и ненаправленные связи, но рассмотрим пример с направленными связями).

Необходимо найти кратчайшее расстояние от одного из узлов до другого.

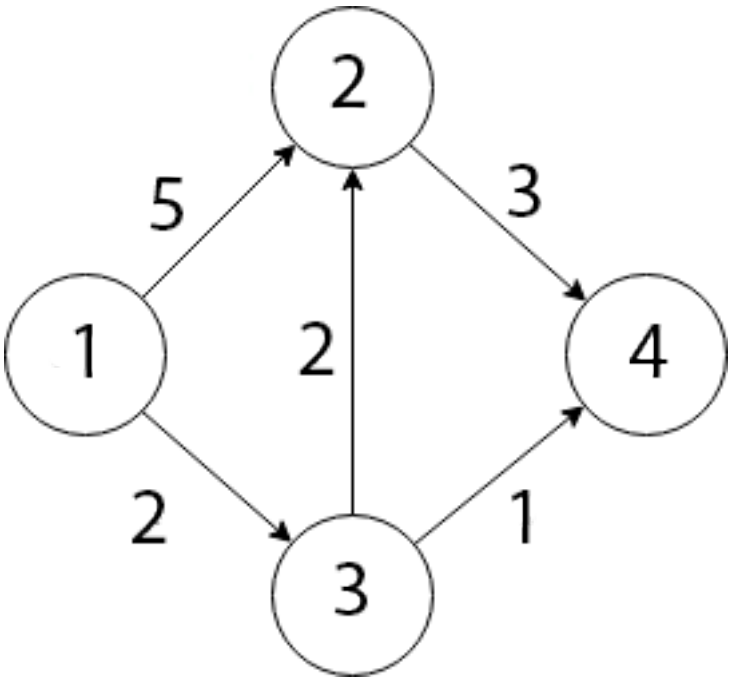

Рисунок 3: исходный граф

Допустим, нам необходимо найти кратчайший путь из вершины 1 в вершину 2. Значения узлов изначально равно прямой связи между первой вершиной и рассматриваемым узлом (без узлов-посредников). В случае, если прямой связи между первой вершиной с каким-либо из узлов не существует, метка равняется бесконечности. Когда все метки расставлены, нам необходимо удалить первую вершину из списка не посещенных вершин, то есть в дальнейшем мы не будем ее рассматривать.

Алгоритм пошагово перебирает все узлы графа и изменяет значения меток на кратчайшее известное расстояние с учетом возможности изменения маршрута (пути).

Расставленные на 1 шаге метки показаны на *рисунке 4* зеленым цветом.

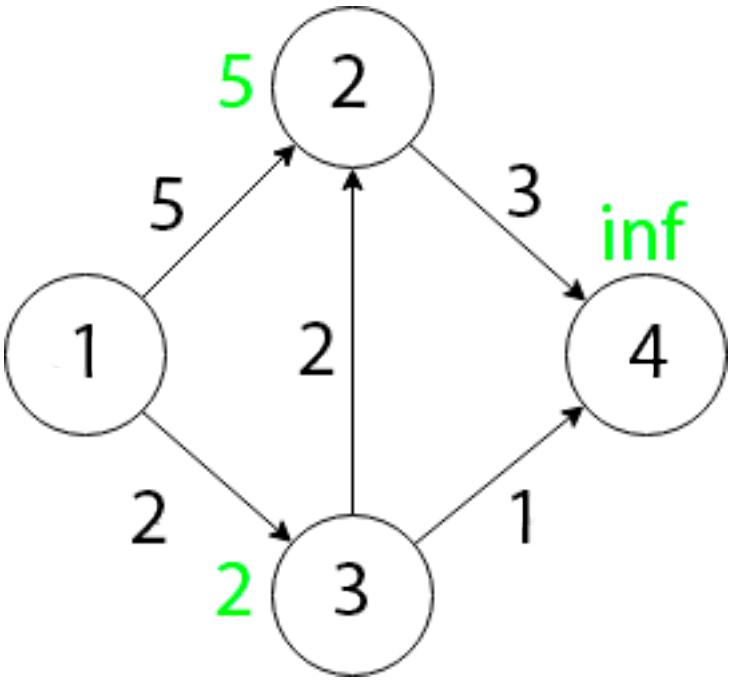

Рисунок 4: 1 шаг алгоритма Дейкстры

Далее выбираем вершину с минимальной меткой. В нашем случае, это вершина 3. Повторяем первый шаг, но уже от вершины под номером 3: каждая из вершин получит метку, равную сумме метки выбранной на данном шаге алгоритма вершины (в нашем случае - вершины 3) и связи между рассматриваемыми вершинами. В случае, если какая-либо из новых меток будет меньше текущей метки вершины, новая метка сохраняется вместо старой. Этот шаг показан на *рисунке 5*, новые метки нарисованы синим цветом. При завершении данного шага, мы вычеркиваем вершину под номером 3 из списка не посещенных вершин, то есть в дальнейшем мы не будем ее рассматривать.

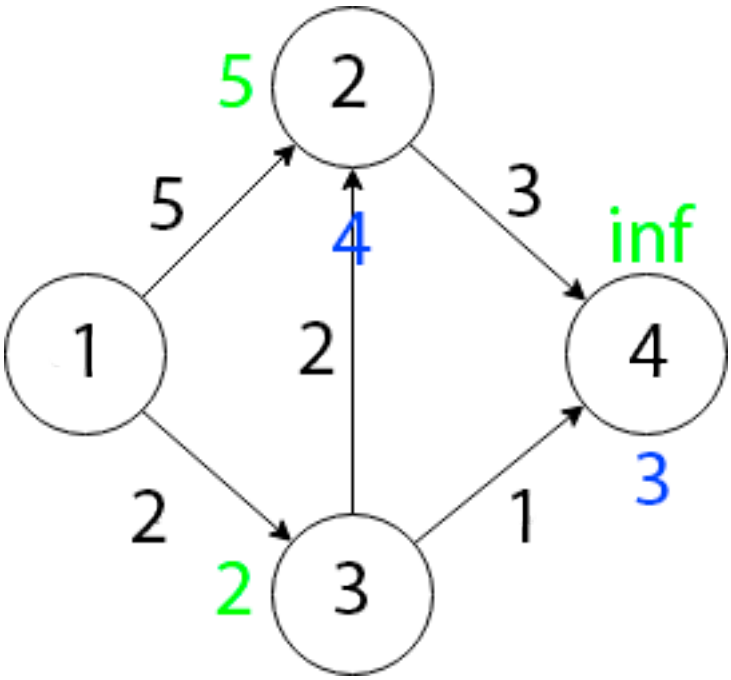

Рисунок 5: 2 шаг алгоритма Дейкстры

Алгоритм продолжает свою работу до тех пор, пока список не посещенных вершин не останется пустым или пока не будут пройдены все связи (тупик). Значение метки необходимой вершины (в нашем случае, вершины 2) по окончании работы алгоритма и будет кратчайшим расстоянием из начальной точки.

2. Конструкторский раздел

Рабочий проект представляет собой веб-приложение, состоящее из:

- Клиентской части - графического интерфейса, обеспечивающего взаимодействие пользователя с приложением через браузер;

- Серверной части - программы, обеспечивающей обработку запросов пользователя;

- Базы данных - программного обеспечения на сервере, отвечающего за хранение и выдачу данных в нужный момент.

Общая схема приложения показана на *рисунке 6*.

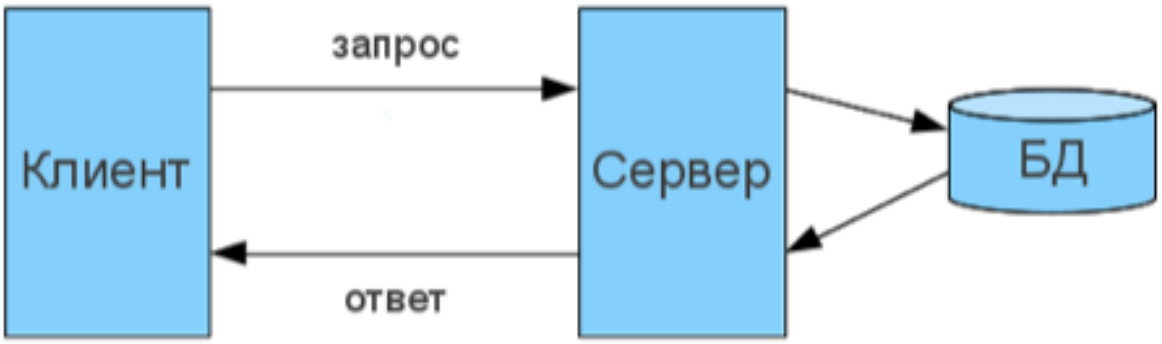

Рисунок 6: схема приложения

2.1 Серверная часть

Сервер в проекте создается с помощью Node.js фреймворка Express. Express основан на базовом модуле Node.js http, однако, в отличии от http, express обладает рядом готовых абстракций, упрощающих создание сервера.

Для использования фреймворков и модулей в Node.js необходимо создать соответствующий объект.

После подключения соответствующего фреймворка (создания объекта), мы можем использовать встроенные функции Express. Одой из таких функций является обработка запросов get и post. В качестве первого параметра

передается маршрут, в качестве второго - функция обработки запроса, или callback.

Для запуска сервера используется функция Express listen, которая получает на вход номер порта.

2.2 Клиентская часть

Клиентская часть приложения разрабатывается на js-библиотеке React.js. При создании frontend части модуль npm автоматически генерирует все необходимые файлы, главными из которых являются app.js и index.html.

App.js содержит в себе логику frontend части приложения: он отвечает за рендер страницы и отправление запросов на серверную часть. Для этого будет использоваться Fetch API - интерфейс получения ресурсов, возвращающий promise, который будет либо выполнен, либо отклонен. Для получения данных из promise используются callback метод then. Но для реализации использовалась обертка вызова в асинхронную функцию, вследствие чего callback метод then был заменен на await. Это сделано для того, чтобы код останавливал свое выполнение на слове await и ждал возвращения promise.

2.3 Взаимодействие клиентской и серверной частей приложения

Все, что необходимо делать клиентской части приложения, это отрисовывать необходимый пользователю интерфейс и реагировать на его взаимодействие с пользователем. Одной из возможных реакций является отправление запроса серверной части с целью получить необходимый пользователю ответ (результат). В случае с нашем приложением, клиентская часть принимает от пользователя названия двух городов и нажатие кнопки поиска.

При нажатии кнопки поиска, клиентская часть формирует запрос, содержащий в себе значения из полей ввода - названия городов. Далее, этот запрос отправляется на серверную часть. Асинхронная «обертка» вызова заставляет программу ждать promise - ответа от сервера. Когда promise выполнился, клиентская часть «парсит» данные с помощью метода Fetch API json(). Результат сохраняется в переменную как строка, к нему добавляется необходимый текст (например, «Расстояние = » + result), таким образом формируется готовая строка ответа. Когда строка сформирована, объект frontend-программы App вызывает встроенный метод forceUpdate(), отрисовывающий ответ в отведенный ему html блок (обновляющий значение html блока).

Серверная часть получает запрос от клиентской части с помощью функций обработки запросов get или post. Одним из аргументов get и post является маршрут. Именно по нему Node.js определяет, какую функцию обработки запросов запустить исходя из полученного запроса. Также аргументами является callback-функция, получающая объекты req (request) и res (result), хранящие тело запроса и ответа соответственно. Далее идет тело функции-обработчика, в нашем случае это запрос в БД (об этом будет написано в пункте 2.5). Завершается функция обработки запроса отправлением результата обратно в клиентскую часть. Для этого используется метод send объекта res, принимающий на вход объект ответа сервера на запрос, обернутый в формат json с помощью функции JSON.stringify().

2.4 Взаимодействие серверной части приложения с БД

Как было сказано в аналитической части, взаимодействие БД с серверной частью обеспечивает Node.js модуль neo4j_driver, получающий на вход bolt БД и данные для входа (логин и пароль).

Для того, чтобы была возможность отправлять запросы, необходимо создать сессию (объект Session), являющийся частью neo4j_driver. В качество

входного параметра Session может принимать режимы read или write для осуществления чтения или записи соответственно, но, так как нас интересует не только чтение данных из БД, но и их запись (например, создание нового города или новой связи). Поэтому мы можем оставить стандартное значение session() без параметра. Это сделает доступным оба режима работы с БД.

Объект Session имеет метод run, позволяющий запустить cypher запрос. Используя promise и метод then, мы получаем возможность дождаться выполнения cypher запроса, получить данные, обработать их и сформировать ответ.

3. Технологический раздел

Так как программа состоит из 3 основных частей (клиентская часть, серверная часть и БД), главной и необходимой задачей является объединение эти частей в единое приложение.

3.1 Особенности выбора СУБД

В настоящий момент существует множество различных графовых СУБД, ведь графовые БД активно развиваются и становятся все более востребованными: например, они используются для моделирования социальных графов (социальных сетей).

Главные критерии выбора:

- Наличие обучающих материалов и полной документации;
- Высокая производительность как с большими, так и с малыми объемами данных;
- Удобство интеграции БД в приложение;
- СУБД должна быть бесплатной.

Также большим преимуществом является встроенная возможность представления данных в виде графа и интерфейс для работы с БД.

По приведенным критериям было подобрано несколько СУБД:

1. OrientDB
2. InfiniteGraph
3. Neo4j

Все приведенные выше СУБД являются распространенными, обладают большим комьюнити. Однако, OrientDB и InfiniteGraph, в основном, используются в крупных проектах. Neo4j, в свою очередь, обладает исчерпывающим руководством и документацией, более ориентированной на еще не знакомых с графовыми БД программистов. Также, Neo4j имеет крайне удобный интерфейс, включающий в себя графическое представление данных в виде графа.

Кроме того, Neo4j использует cypher - язык, который является не только языком запросов, но и языком манипулирования данными (предоставляет функции CRUD).

3.2 Особенности выбора языка программирования

В нашем случае выбор языка программирования является побочным, так как он напрямую зависит от выбранной технологии. Серверная и клиентская части реализуются с помощью Node.js и React.js соответственно. Обе технологии подразумевают использование языка программирования JavaScript.

Язык запросов БД зависит от выбранной СУБД. Neo4j использует язык cypher.

3.3 Отрисовка интерфейса

Отрисовка интерфейса с помощью React.js требует не только знания языка JavaScript, но и знания html и css. Непосредственно html и css отвечают за внешний вид приложения, JavaScript же является некой «надстройкой» - он реализует функционал форм (кнопок, полей ввода), заполняет html блоки нужной информацией и получает из этих блоков информацию, если это необходимо.

3.4 Графическое представления данных

Для графического представления данных используется js-библиотека neovis.js. Эта библиотека создана специально для работы с neo4j: мы создаем асинхронную функцию, внутри которой создается конфиг с данными, необходимыми для подключения к БД и с полями, которые нам понадобится взять из БД. Далее, происходит подключение к БД и рендер полученного графа в html-блок.

3.5 Работа с pdf

В приложение была добавлена функция просмотра РПЗ курсового проекта непосредственно в браузере. Для этого использовалась библиотека react-pdf. Данная библиотека создана для создания и отображения pdf-документов в браузере. React-pdf работает с приложениями, созданными на фреймворке reactjs.

3.6 Использование программы

У пользователя приложения есть возможность найти кратчайшее расстояние между двумя городами, добавить город или связь и посмотреть граф или РПЗ. Для этого нам понадобится 10 html форм для взаимодействия: 2 поля для выбора городов, кнопка для поиска расстояния, поле для ввода названия нового города, кнопка для создания нового города, поле для ввода расстояния между двумя городами и кнопка для его сохранения в БД. Также, нам потребуется поле <p> для вывода результата расчета расстояния между городами и две кнопки для переключения между РПЗ и графом. Интерфейс приложения показан на *рисунке 7*.

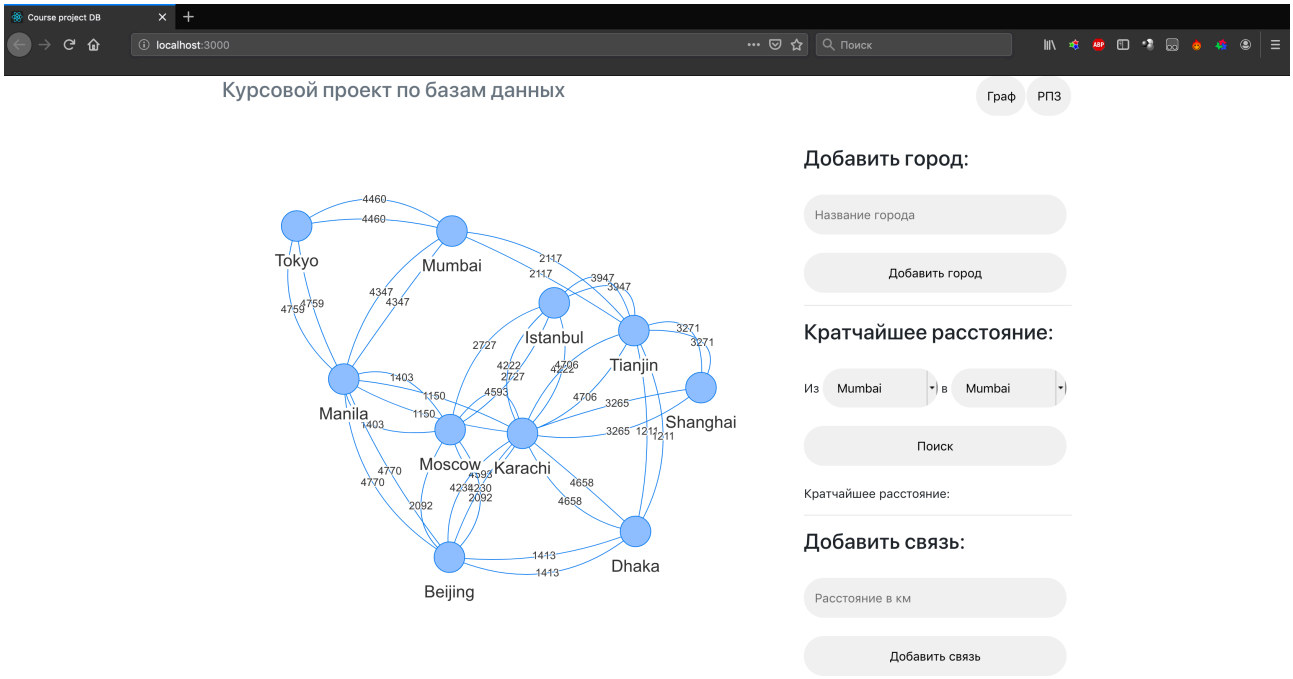

Рисунок 7: интерфейс приложения

При загрузке страницы вызывается init - запрос в БД, возвращающий массив всех городов в БД. Это необходимо для того, чтобы заполнить поля options, обеспечивающие удобный выбор городов для пользователя. Создание связей также использует эти два поля select. Демонстрация полей select на *рисунке 8*.

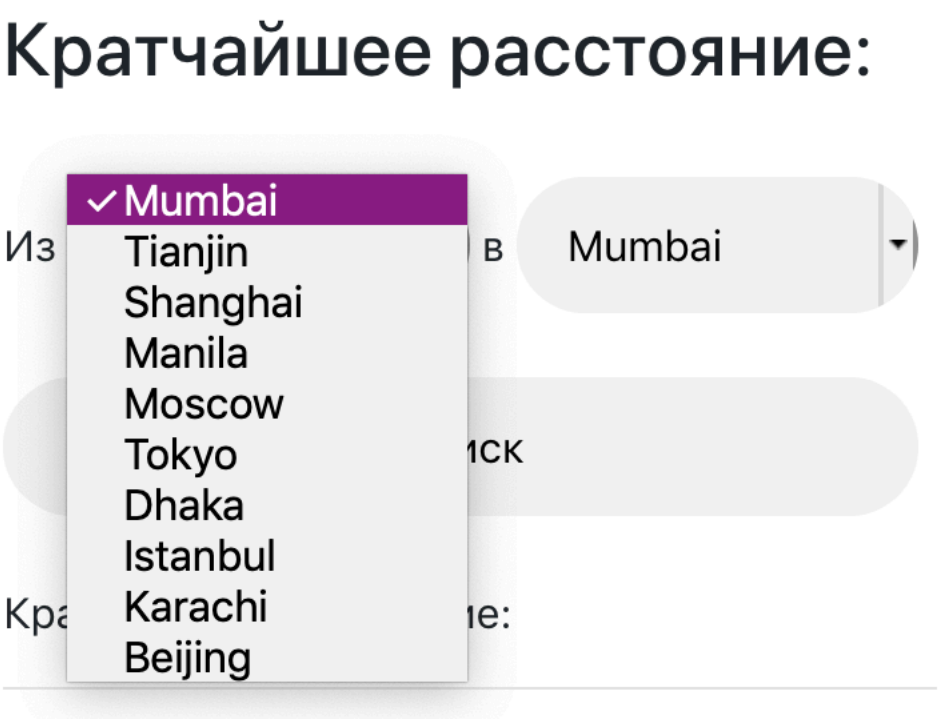

Рисунок 8: select поле с возможностью выбора города из списка

В случае, если пользователь захочет посмотреть РПЗ, ему будет достаточно нажать на соответствующую кнопку навигации. Html-блок, содержащий в себе граф, «заменит» его на pdf-документ с РПЗ. Замещение происходит следующим образом: есть общий html-элемент div, содержащий в себе два других html-блока div. Один из этих блоков содержит в себе граф, другой - РПЗ. Один из блоков всегда спрятан, то есть имеет свойство display:none. Соответственно, нажимая кнопки в разделе навигации (граф или РПЗ), пользователь, фактически, выбирает, какой из этих двух блоков будет показан в браузере, а какой будет спрятан. Такое переключение сделано, в частности, для того, чтобы при переходе от графа к pdf и обратно, текущий прогресс чтения РПЗ не сбрасывался. Пользователь в любой момент может переключиться обратно с графа на pdf-документ и продолжить его читать с той же страницы, на которой он остановился.

В случае, если отображается блок с pdf-файлом, пользователю необходимо предоставить интерфейс для комфортного чтения документа. Для

этого были реализованы две кнопки: перелистывание страниц документа вперед и назад.

Демонстрация рендера pdf-файла на *рисунке 9*.

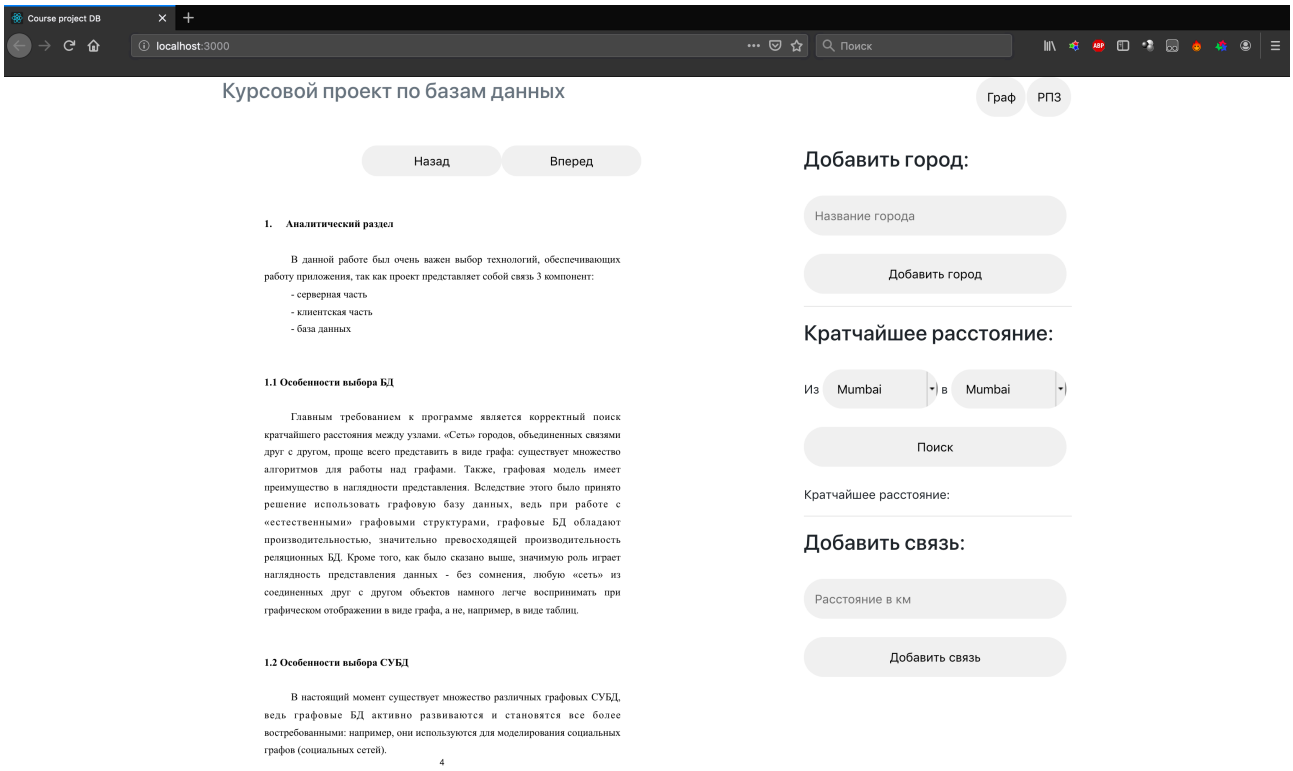

Рисунок 9: рендер pdf-файла

Поля и кнопки интерфейса имеют единый стиль оформления, включающий в себя:

- размер элемента;
- цвет элемента;
- цвет шрифта.

Поля также имеют «шаблонный текст», то есть надпись на поле, подсказывающую пользователю, какие данные он может передать программе при помощи это поля. Данный текст стирается, как только пользователь начинает вводить свои данные. Кроме того, поле выбранное пользователем в текущий момент, подсвечивается при помощи изменения границы элемента. Это помогает пользователю лучше ориентироваться на веб-странице. Пример изображен на *рисунке 10*.

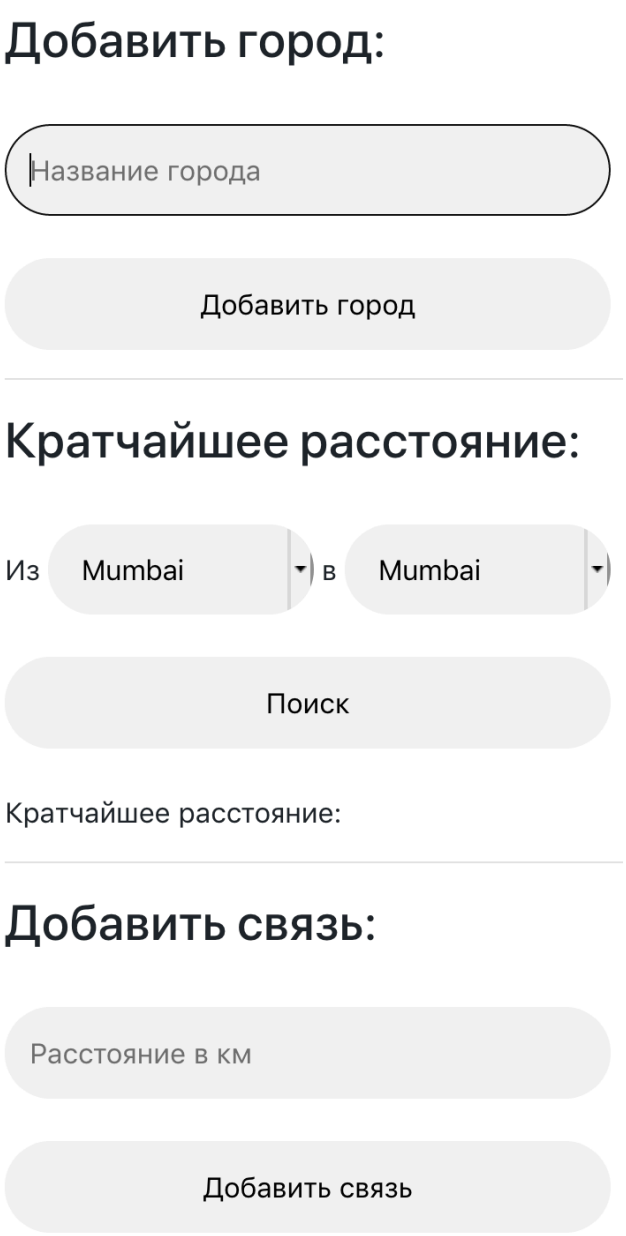

Рисунок 10: шаблонный текст и эффект выбранного пользователем поля

Некоторые из функций, описанных выше, были созданы при помощи библиотеки bootstrap.

3.7 Запуск приложения на сервере

Приложение были загружено на VDS-сервер. Для этого на сервере были установлены:

- Node.js - обеспечивает работу серверной части;

- npm - менеджер библиотек, необходим для добавления в проект ключевых библиотек;

- pm2 - обеспечивает запуск процессов в виде «демонов». Это нужно для того, чтобы серверная и клиентская части приложения работали на сервере в фоновом режиме.

На VDS-сервере загружена Ubuntu 18.04.

4. Исследовательский раздел

Выгода в использовании графовых БД проявляется при увеличении объема данных.

4.1 Рациональность использования графовой БД

Само по себе хранение и манипуляция с данными в Neo4j является более затратным в плане ресурсов. Однако, если граф является достаточно большим, как, например, в случае с социальными сетями, имеющими огромную базу пользователей, их взаимодействий (таких, как лайки, комментарии, follow и тд.). Наш проект имеет относительно небольшой граф, так как с увеличением количества узлов и их связей становится невозможно воспринимать данные графически. Это необходимо для того, чтобы можно было вручную отследить возможные пути между узлами.

4.2 Данные о системе

Разработанное веб-приложение не является ресурсоемким в плане характеристик «железа», поэтому эти данные не будут приведены. Для корректной работы программы гораздо важнее программное обеспечение, обеспечивающее работу каждой из частей приложения. Версии основных библиотек и фреймворков, а также информация о системе, приведены ниже.

ОС: MacOS Mojave 10.14.4.

Версия Node.js: 11.1.0.

Версия npm: 6.9.0.

Разработка велась в текстовом редакторе SublimeText2. Запуск частей приложения, включая БД, производился через приложение «Терминал», являющееся часть операционной системы Mac.

Заключение

В работе было рассмотрено применение графовых БД и их интеграция в web приложения.

Был выбран алгоритм для решения поставленной задачи. Реализация алгоритма Дейкстры осуществлялась на языке JavaScript.

Также, было разобрано строение современных web-приложений, взаимодействие трех ключевых составляющих web-приложений: клиентской части (frontend), серверной части (backend) и базы данных.

Список литературы

[1]. Официальная документация языка cypher. Синтаксис, основные функции. — https://neo4j.com/docs/cypher-manual/current/

[2]. Официальная документация СУБД neo4j. Информация по установке и настройке. Концепты графовых БД. Базовый синтаксис языка cypher. — https://neo4j.com/docs/getting-started/current/

[3]. Алгоритм Дейкстры. Пример работы алгоритма Дейкстры. — https://habr.com/ru/post/111361/

[4]. Текст научной статьи по специальности «Общие и комплексные проблемы естественных и точных наук»: «Обзор алгоритмов поиска кратчайшего пути в графе». — https://cyberleninka.ru/article/v/obzor-algoritmov-poiska-kratchayshego-puti-v-grafe

[5]. Туториал по React.js. Основы технологии, описание базовых конструкций. Описание работы библиотеки. — https://reactjs.org/tutorial/tutorial.html

[6]. «Node.js в действии». Алекс Янг, Брэдли Мек, Майк Кантелон. 2-е издание

[7]. «Graph Databases for Beginners». Bryce Merkl Sasaki, Joy Chao & Rachel Howard. — https://go.neo4j.com/rs/710-RRC-335/images/Graph_Databases_for_Beginners.pdf