Институт вычислительной математики и математической геофизики СО РАН

На правах рукописи

Конин Максим Васильевич

Проблемы реализации гиперсетевых технологий моделирования сложных систем

Специальность 05.13.18 —
«Математическое моделирование, численные методы и комплексы программ»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
д.т.н., профессор
Родионов Алексей Сергеевич

Новосибирск — 2017

Оглавление

Стр.

Стр.

Стр.

Введение

В диссертации предлагается разработка программного комплекса для хранения и анализа гиперсетей. Понятие ”гиперсеть”было впервые дано в работе Владимира Константиновича Попокова [1]. С тех пор данная теория стремительно развивается и нашла свое применение в следующих направлениях:

1. Оптимизация структурированных кабельных сетей.
2. Оптимизация инженерных инфраструктур на местности.
3. Проверка живучести информационных сетей.
4. Решение транспортных задач.
5. Создание различных геоинформационных систем.

Данные задачи актуальны и востребованы. С ростом инфраструктур, подлежащий оптимизации, растет и объем данных, которые требуется обрабатывать. Кроме того, все эти задачи сугубо прикладные. А для прикладных задач характерно дальнейшая поддержка, модификация и документация. Данное обстоятельство диктует требования системы, которая будет агрегировать данные, получение после выполнения алгоритмов и по требованию выдавать их либо специалисту, либо другим алгоритмам для дальнейшей работы. На данные момент целостной системы и каких-либо стандартов для работы с данными, формализованными на языке теории гиперсетей не существует.

Целью данной работы является построение программного комплекса, который будет выполнять задачи хранения, анализа и представления гиперсетевых данных.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Исследовать существующие задачи и алгоритмы их решения в рамках теории гиперсетей.
2. Исследовать существующие программные комплексы, как в области гиперсетей так и в смежных областях, например в теории графов.
3. Разработать программный комплекс отвечающий поставленной цели.
4. Разработать алгоритмы генерации различных представлений.
5. Разработать программные интерфейсы для взаимодействия с системой.
6. В качестве демонстрации системы, решить задачу оптимизации коллекторной сети и задачу оптимизации общественного транспорта.

Научная новизна:

1. Впервые предложена схема хранения гиперсетей и графов в документо-ориентировных базах данных.
2. Впервые построен целостный программный комплекс обеспечивающий целостное хранение гиперсетей. Приведены алгоритмы для генерации различных представлений гиперсетей. Учтена возможность горизонтального масштабирования при обработки больших объемов данных.
3. Разработаны программные интерфейсы для работы с системой.
4. Предложен генетический алгоритм для решения задачи оптимизации коллекторной сети мегаполиса.
5. Предложен генетический алгоритм для решения задачи оптимизации маршрутов общественного транспорта.

Практическая значимость Результаты работы могут быть применены для построения алгоритмов оптимизации систем сетевой структуры, в качестве математической модели которых используется гиперсеть.

Методология и методы исследования. …

Основные положения, выносимые на защиту:

1. Первое положение
2. Второе положение
3. Третье положение
4. Четвертое положение

Достоверность полученных результатов обеспечивается … Результаты находятся в соответствии с результатами, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на:

1. Одинадцатой международной азиатской школе-семинаре ”Проблемы оптимизации сложных систем”.
2. Двенадцатой международной азиатской школе-семинаре ”Проблемы оптимизации сложных систем”.
3. Конференциях молодых ученых ИВМиМГ СО РАН в 2016, 2015 и 2014 годах.
4. Конференции ”Перспективные информационные и телекоммуникационные технологии”СибГУТИ.

Личный вклад. Автор принимал активное участие во всем

Публикации. Основные результаты по теме диссертации изложены в 0 печатных изданиях, 0 из которых изданы в журналах, рекомендованных ВАК, 0 — в тезисах докладов.

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и двух приложений. Полный объём диссертации составляет 81 страницу, включая 5 рисунков и 0 таблиц. Список литературы содержит 45 наименований.

Глава 1. Задачи теории гиперсетей

1.1 О теории гиперсетей

Понятие ”гиперсеть”было впервые дано в работе [1]. Гиперсеть это математическая модель для описания сложных систем сетевой структуры, таких как: сети связи, транспортные сети, информационные сети и т.д. Ключевой особенностью Гиперсетей является возможность описывания имеющих более 2 структур каждая из которых может быть вложена в другую. Например рассматривая транспортную сеть города мы имеем дело сетью улиц, в которую отображены (вложены) множество маршрутов общественного транспорта.

Традиционно, при решении задач в перечисленных областях использовались графы и гиперграфы в качестве математической модели [2]. Отдельные задачи хорошо изучены и решены в рамках теории графов. Но на практике, когда необходимо построить реальную систему, мы не можем абстрагироваться от всех взаимосвязей и внешних факторов, которые оказывают влияние на итоговую структуру гиперсети. В некоторых задачах, мы имеем дело с нестационарными структурами - структуры, которые изменяются с течением времени. В таких случаях поставить четкую математическую задачу практически невозможно. Даже сама формулировка окажется громоздкой, не говоря уже о решении. Либо придется пренебрегать каким-либо воздействием. и решать частную задачу. Для решения некоторого ряда задач можно использовать другие известные модели, такие как сендвич графы [3] или вложенные граф [4], [5]. Но гиперсети являются более широкой моделью (сэндвич графы и вложенные графы могут быть описаны так же в рамках гиперсетей), которая может, достаточно лаконично описать сложные иерархические, многоуровневые сетевые модели.

В [6] были даны базовые понятия теории гиперсетей, построена классификация, предложены некоторые алгоритмы.

На данном математическом аппарате был решен целый ряд прикладных задач в разных сферах:

1. Оптимизация структурированных кабельных сетей.
2. Проверка живучести информационных сетей.
3. Решение транспортных задач.

4. Создание различных геоинформационных систем.

Далее в работе [7] было дано определение S-Гиперсети (Структурированной Гиперсети). Данная модель позволяет описывать объекты с произвольным уровнем вложенности (в отличии от простой гиперсети, которая описывает только 2 уровня).

1.2 Основные определения

***Определение* 1.2.1.** *Граф, или неориентированный граф G — это упорядоченная пара $G := (V,E)G := (V,E)$, где V — это непустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых рёбрами.*

***Определение* 1.2.2.** *Ориентированный граф (сокращённо орграф) G — это упорядоченная пара $G := (V,A)G := (V,A)$, где V — непустое множество вершин или узлов, и A — множество (упорядоченных) пар различных вершин, называемых дугами или ориентированными рёбрами.*

***Определение* 1.2.3.** *Гиперграф — обобщение графа, в котором каждым ребром могут соединяться не только две вершины, но и любые подмножества вершин.*

***Определение* 1.2.4.** *Абстрактной гиперсетью называется $AS = (X, V, R; P, F, W)$, состоящей из следующих объектов: [6]*

1. $X = (x_1, x_2,..., x_n)$ *— множество вершин;*
2. $V = (v_1, v_2, ..., v_g)$ *— множество ветвей;*
3. $R = (r_1, r_2, ..., r_m)$ *— множество ребер;*
4. $P\colon V \to 2^X$ *— отображение, сопоставляющее каждому элементу $v \in V$ множество $P(v) \subseteq X$ его вершин. Тем самым отображение определяет гиперграф $PS = (X,V; P)$;*
5. $F\colon R \to 2^v_{PS}$ *- отображение, сопоставляющее каждому элементу $r \in R$ множество $F(r) \subseteq V$ его ветвей. Отображение F определяет гиперграф $FS = (V, R; F)$.*
6. $\forall r \in R, W : r \to 2^{P(F(r))}$ *отображение, сопоставляющее каждому элементу $r \in R$ подмножество $W(r) \subseteq P(F(r))$ его вершин, где $P(F(r))$*

— множество вершин в PS, инцидентных ветвям $F(r) \subseteq V$ Таким образом, отображение W определяет гиперграф $WS = (X, R; W)$.

Гиперграф PS назовем первичной сетью гиперсети AS, а гиперграф WS — вторичной сетью гиперсети AS.

***Определение* 1.2.5.** *Абстрактная гиперсеть $S = (X, V, R; P, F, W)$ называется гиперсетью, если: [6]*

1. $\forall v \in V |P(v)| = 2$
2. $\forall r \in R |W(r)| = 2$
3. $\forall r \in R$*множество* $F(r) \subseteq V$ *составляет маршрут в графе* $PS = (X, V)$.

***Определение* 1.2.6.** *Пусть задано множество графов (гиперграфов)* $G_0 = (V^0, E^0), G_0 = (V^0, E^0), ..., G_0 = (V^n, E^n)$.

S-Гиперсетью будем называть дерево $T_0 = (Z, F)$, где $Z = z_0, z_1, ...z_k$, $F = f_1...f_k$, $z_i = G_i (0 \leqslant i \leqslant n)$, а f_i - пара отображений (f^v, f^e) определяет вложение графов G_i в G_{i-1}

$f^e : E^{i-1} \to 2^{E^i}$ *- это маршрут в графе* G_i,

$f^v : V^{i-1} \to V^i$ *- отображение вершин из* G_{i-1} *в* G_i.*[7]*

1.3 Примеры и способы решения прикладных задач

1.3.1 Построение структурированных кабельных систем в здании

Данная здача была подробно рассмотрена в работах: [8], [9], [10].

Структурированная кабельная система (СКС) — законченная совокупность кабелей связи и коммутационного оборудования, отвечающая требованиям соответствующих нормативных документов [11].

В перечисленных работах особое внимание уделено переходу от концептуальной модели структурированной кабельной сети к строгой математической модели.

Концептуальная модель - это абстрактная, без привязки к каким-либо математическим понятиям, модель определяющая состав и структуру сети, элементы

сети и их взаимосвязи. Математическая же модель - это описание структуры сети, ее элементов и взаимосвязей, на языке выбранного математического аппарата, например, теории графов. Очевидно, что выбор математического аппарата влияет на возможность обобщения задачи на более широкий спектр задач. Чем универсальнее был выбран аппарат, тем больше возможностей для формально описание взаимосвязей и ограничений в сетях.

В работе [8] описана проблематика и доказана актуальность данной задачи. Рассмотрены разные подходы к построению математических моделей структурированных кабельных сетей и показано преимущества применения методов теории гиперсетей для моделирования СКС. Как видно из работы, СКС предоставляет из себя сложную систему, полно описать которую в рамках теории графов нет возможности.

В работах [9], [10], [12] построена математическая модель на основе гиперсети, описывающая структурированные кабельные сети в здании. Доказана корректность модели и показаны преимущества перед аналогичными грабовыми моделями. Доказано, что задача построения структурированных кабельных сетей лежит в классе NP. Что в свою очередь диктует необходимость применения неточных методов построения оптимизационных алгоритмов, так как в большинстве случаев задача проектирования кабельных систем имеет огромную размерность.

В рамках этих работ ставится и решается задача оптимальной трассировки кабелей СКС. Данная задача сводится к задаче нахождения оптимального вложения вторичной сети гиперсети в первичную. Для решения задачи построен генетический алгоритм Алгоритм реализован на языке программирования Delphi. По приведенным экспериментальным данным, можно заключить, что данный подход дает отличные показатели на реальных задачах. Тем не менее, алгоритм не использует никакие внешние решения или базы данных, для получения данных о гиперсети. В качестве источника данных используются текстовые файлы. Так же и результатом работы алгоритма является текстовый файл в определенном формате, содержащий данные о найденном решении. Данный факт осложняет дальнейшую поддержку найденного решения. Часто возникает необходимость добавить элемент в существующую сеть и найти оптимальную трассу до него. Имея лишь алгоритм оптимизации, проектировщик заново должен сформировать входные данные, с учетом существующий сети, и запустить его на новых данных.

1.3.2 Построение инженерной инфраструктуры.

Инженерная инфраструктура является совокупностью ряда сложных инженерных систем. Таких например как:

1. системы электроснабжения
2. системы теплоснабжения
3. системы водоснабжения и водоотведения
4. системы вентиляции и кондиционирования воздуха
5. системы газоснабжения
6. внутренние сети связи (телефонная сеть, структурированная кабельная система, система автоматизированного диспетчерского управления, система контроля доступа, система визуализации).

Очевидно предыдущая задача является подзадачей построения инженерной инфраструктуры в целом. Следует учесть, что математический аппарат гиперсетей может использоваться только для какой-либо одной инженерной сети, например, СКС, на практике же все инженерные сети взаимосвязаны.

При проектировании, например, электросети, инженер-проектировщик учитывает расположение системы теплоснабжения и следую определенным ГОСТ-ам и стандартам проектирует электрическую сеть с учетом ограничений, наложенных системой теплоснабжения.

При проектировании инженерных кабельных систем одним из наиболее трудозатратных процессов является трассировка кабелей. Большую роль в осложнении данного процесса играет наличие огромного количества ограничений на прокладку кабелей.

В частности для решения этой задачи, в работе [7] был предложен новая математическая модель - S-Гиперсеть (структурированная гиперсеть). Данный математический аппарат позволяет учитывать влияния разных систем на одном уровне друг на друга.

С использованием этого нового математического аппарата в работе [13] была решена задача построения кабельных трасс в здании. В данной работе построена математическая модель инженерной инфраструктуры здания и обоснована ее корректность. На основе математической модели был предложен и реализован генетический алгоритм нахождения оптимальной структуры. Язык реализации алгоритма - $C\#$.

Данный алгоритм применялся в программном обеспечении в комплексе с платформами AutoCAD и AutoCAD Revit, которые служили источниками данных для алгоритма.

1.3.3 Гиперсети в информационных сетях.

Под информационными сетями понимается сеть предназначенная для обработки, хранения и передачи данных. К информационным сетям можно отнести такие системы как: LAN, WAN, мобильные сети, беспроводные сети передачи данных.

Хотя некоторые из перечисленных систем являются подсистемами структурированных кабельных сетей, данная тематика содержит ряд особенных задач, присущих только ей. Например, крайне важной характеристикой информационной сети является ее живучесть. Каждый владелец информационной сети желает обеспечить бесперебойную работу системы даже при выходе из строя некоторого количества элементов сети.

В целом, вопрос живучести сетей связи был подробно изучен в [14], [15] и [16]. В [15] приведена классификация элементов сетей и их разрушений, построена математичиская модель информационной сети на основе гиперсети, введено понятие устойчивости гиперсети к разным видам удаления компонентов, разработан анализа устойчивости гиперсети и проведено математическое и имитационное моделирование. Данные результаты очень важны для построения информационных сетей, так как в прикладных задача мы сталкиваемся с многокритериальной задачей оптимизации. Когда инженер-проектировщик проектирует информационную сеть он должен не только минимизировать задачи на монтаж, а также и увеличить живучесть сети. И подход, описанный в данных работах, вполне может воплотить это в реальность.

Различные модели, как из теории графов так и теории гиперсетей, для систем передачи данных были рассмотрены в [17].

В работах [18; 19] рассматриваются проблемы надежности мобильных сетей передачи данных. В качестве математической модели предлагается использовать нестационарные гиперсети.

***Определение* 1.3.1.** $HS(\tau) = (X, V, R; P, F, W)$ *- нестационарная гиперсеть со следующими параметрами* $(\tau \in [0, T])$*:*

1. *каждой вершине* $x_i \in X$ *сопоставлена емкость вершины (буфер)* $\beta_i \geqslant 0$*;*
2. *каждой ветви* $v_j \in V$ *сопоставлена функция пропускной способности ветви* $\alpha_j(\tau) \geqslant 0$*;*
3. *каждому ребру* $r_k \in R$ *сопоставлена функция пропускной способности ребра* $\delta_k(\tau) \geqslant 0$ *и* $\tau_r \geqslant 0$ *задержка передачи потока по ребру.*

Мобильные сети опять же имеют свою специфику и свои задачи. Как видно из приведенных выше работ, для решения прикладных задач теории графов не хватает. И именно поэтому в ряде работ про исследования мобильных телекоммуникационных систем применяются гиперсети.

В работе [20] построен и реализован алгоритм решения задачи нахождения максимального потока в гиперсетях. Показаны варианты применения данного алгоритма в решении прикладные задач мобильных сетей.

Большую роль в информационных сетях так же играет и мониторинг. В работах [21; 22] рассмотрен вопрос оптимальной расстановки устройств мониторинга. Для представления сети опять же используется гиперсеть.

В работах [23; 24] рассматривается уже инженерная инфраструктура не в рамках одного здания, общиной местности. В том числе и на местности, имеющей сложный природный рельеф и особые природные условия. В рассмотренных задачах, гиперсеть в качестве математической модели помогла описать все сложные факторы, влиявшие на структуру сети.

1.3.4 Транспортные системы.

Задачи по модернизации или проектированию новых транспортных сетей набирают популярность с ростом мегаполисов и появлением новых видов общественного транспорта. Транспортная система крупного города представляет из себя сложную иерархическую систему, в общем случае нестационарную. В нее включены множество сетей, взаимосвязанных друг с другом и даже принадлежат разным владельцам. Математическая модель для решения транспортных задач

должна объединять все факторы и ограничения, имеющие влияние на функционирование сети.

В работах [7], [25] построена подробная математическая модель транспортной сети мегаполиса. Модель представляет из себя S-Гиперсеть со следующими уровнями:

1. Уличная сеть города. Основа модели
2. Дороги и другие транспортные линейные сооружения
3. Проезжая часть
4. Полоса движения
5. Локусы — локальные участки сети на полосах движения, соответствующие местам, занимаемым транспортными единицами в течение определенного интервала времени.
6. Маршруты движения транспорта
7. Графы аллелей — множество вершин, соответствующих пересечению стоп-линий с полосами движения на перекрестках, разветвлениях и в других узлах дорожной сети. Ребрам сопоставляются аллели — участки полос на проезжей части
8. Граф локусов

Так же в статье приводится ряд задач, которые решаются на основе поставленной модели. Можно выделить следующие группы:

1. Моделирование транспортных потоков
2. Модели передвижения пассажиров и товаров по городу
3. Управление транспортными системами:
4. Математические модели для задач оптимального размещения пунктов обслуживания транспортных средств и населения

В статье [26] представлен подход к построению алгоритма (без программной реализации) для решения задачи размещения пунктов обслуживания в транспортных сетях. Алгоритм базируется на описанной ранее гиперсетевой модели транспортной сети.

Множество задач в сфере транспорта остаются не решенными. Существующая инфраструктура многих мегаполисов требует модернизации из-за увеличившегося количества транспортных средств. Очевидно задача оптимизации сети не является разовой. Транспортную система нуждается в постоянном мониторинге и постоянном пересмотре оптимизационных решений. Очевидно, для удовлетво-

рения этих подробностей так же нужна единая система для хранения и предоставления данных математической модели.

В работе[27] проводились исследования региональное транспортной сети. Математической моделью многоуровневой сети выступает гиперсеть. В ходе исследования был построен алгоритм формирования вариантов для оптимизации транспортной (автомобильной и железнодорожной) сети при прогнозах экономического развития региона.

1.4 Выводы

Очевидно существует уйма актуальных задач для решения которых требуется аппарат гиперсетей. К сожалению, на сегодняшний момент нет какой-либо библиотеки, которая реализует базовый набор операций над гиперсетями. Каждый автор приведенных работ, который реализует свой алгоритм, вынужден сам описывать структуры данных, настраивать обработку входных данных и сохранения результата. К тому же, во многих случаях, конечному пользователю недостаточно только решить одну из приведенных задач. Реальные системы недостаточно только построить, в процессе эксплуатации инженеры сталкиваются с рядом задач по мониторингу, модернизации и поддержке систем. Каждая построена система должна быть хорошо документирована и легко изменяема. Так как нет единого описания форматов входных и выходных данных, нет возможности быстро соединить алгоритмы друг с другом. Что делает невозможным собрать полноценную систему, которая, например, будет и строить сеть оптимальною по стоимости и следить за показателями живучести в реальном времени.

Отсутствие единого источника и хранилища данных делает невозможным дальнейшую поддержку предложенного решения. Да, безусловно инженер, который воспользуется одним из алгоритмов, может реализовать функцию экспорта полученных данных в свою базу. Но делегирование таких задач на плечи конечных пользователей не есть хороший путь.

Эти проблемы, диктуют необходимость реализации решения, которое будет формализовать протоколы работы с гиперсетями. Решение, которое будет являться источником и хранилищем данных для всех алгоритмов, использующих в качестве математической модели гиперсети.

Глава 2. Проектирование системы хранения и анализа гиперсетевых данных

2.1 Постановка задачи

Следуя выводам в предыдущей главе, можно заключить, что на сегодняшний момент не существует универсального решения, для унификации взаимодействий с гиперсетями. Не существует единого стандарта их описания при реализации алгоритмов решения прикладных задач.

Целью данной работы является создание программного комплекса, который:

1. позволит сохранять данные в виде гиперсети на удаленном источнике и переиспользовать их;
2. обеспечит удобные механизмы получения и обновления данных;
3. обеспечит бесперебойную работу с большими данными (решения задач в рамках мегаполиса);
4. обеспечит возможность частичного получения данных и в различных представлениях;

Для реализации поставленной цели, в рамках работы будут решены следующие задачи:

1. Проектирование архитектуры программного комплекса с ориентацией на обработку большого объема данных;
2. Обзор вариантов и выбор оптимальной системы управления базой данных;
3. Проектирование структуры базы данных;
4. Разработка и реализация алгоритмов для генерации часто используемых представлений. Таких как: список смежности, список инциденции;
5. Разработка и реализация для общения пользователя с системой;

2.2 Существующие решения

На данный момент комплексного решения в области гиперсетей не существует. Но так понятие гиперсети тесно связано с понятием графа, то в текущем разделе будут рассмотрены решения для графов.

2.2.1 Графы в реляционных базах

Реляционная модель данных (РМД) — логическая модель данных, прикладная теория построения баз данных, которая является приложением к задачам обработки данных таких разделов математики, как теория множеств и логика первого порядка.

Термин «реляционный» означает, что теория основана на математическом понятии отношение (relation). В качестве неформального синонима термину «отношение» часто встречается слово таблица.

Для лучшего понимания РМД следует отметить три важных обстоятельства[28]:

1. модель является логической, то есть отношения являются логическими (абстрактными), а не физическими (хранимыми) структурами;
2. для реляционных баз данных верен информационный принцип: всё информационное наполнение базы данных представлено одним и только одним способом, а именно — явным заданием значений атрибутов в кортежах отношений; в частности, нет никаких указателей (адресов), связывающих одно значение с другим;
3. наличие реляционной алгебры позволяет реализовать декларативное программирование и декларативное описание ограничений целостности, в дополнение к навигационному (процедурному) программированию и процедурной проверке условий.

Кроме того, для управления и модификации данных в реляционных базах данных применяется специализированный язык запросов SQL (Structured Query Language). Данный язык предоставляет все необходимые средства для работы с реляционными данными при этом являясь абстрактной прослойкой, независи-

щий от конкретной реализации базы данных. Хотя SQL и не является процедурным языком программирования, Тем не менее, существуют процедурные языки надстройки, реализованные в конкретных системах управления базами данных, такие как PL/SQL для OracleDB, PSQL для Interbase и Firebird, PL/pgSQL для PostgreSQL. Данные расширения значительно увеличивают возможности SQL и позволяют решать сложные задачи средствами самой СУБД.

Реляционная модель берет свое начало в 70-ых годах двадцатого века. Опираясь на серьезный математический аппарат - реляционную алгебру, данный подход хорошо зарекомендовал себя. На базе реляционной модели был построено огромное кол-во программно-аппаратных комплексов. В том числе, не остались без внимания и задачи, связанные графами.

В [29] приводится схема хранения графом в реляционной модели, а также некоторые алгоритмы для работы с ней. Граф предлагается хранить в 2 таблицах. Одна служит хранилищем вершин, а другая ребер. Ребра имеют 2 внешних ключа на входящую и сходящую вершины.

Так же, в этой работе приводится ряд алгоритмов из теории графов.

Например алгоритм поиска в ширину будет выглядеть так:

```
PROCEDURE BuildPathEnum
BEGIN
  DECLARE pathlength, oldsize, newsize INTEGER NOT NULL;
  -- начинаемспустойтаблицы   , первичногоключанет
  CREATE TABLE PathEnum
     (source     CHAR(2) NOT NULL,
      target  CHAR(2) NOT NULL,
      pathlength     INTEGER;
  CONSTRAINT non_negative_pathlength
             CHECK(pathlength >= 0);
  -- путьузлаксамомусебе     - необязательныйфрагмент
  BEGIN
    INSERT INTO PathEnum SELECT DISTINCT source, target, 0 FROM
       Edges;

  -- загрузкавтаблицупутисдлиной      = 1
  INSERT INTO PathEnum SELECT source, target, 1 FROM Edges;

  -- строкивводятсятолькововремяростатаблицы
  SET oldsize = 0;
  SET newsize = (SELECT COUNT(*) FROM PathEnum);
  WHILE (oldsize < newsize)
```

```
    LOOP
      INSERT INTO PathEnum
      SELECT P1.source, B1.target, (P1.pathlength + 1)
        FROM PathEnum AS PI, Edges AS B1
      -- перемещениепоимеющимсяпутямнаодинуровень
       WHERE EXISTS (SELECT *
                       FROM PathEnum AS P2
                      WHERE B1.source = P2.target)
      -- вводвтаблицутольконовыхстрок
         AND NOT EXISTS (SELECT *
                           FROM PathEnum AS P3
                          WHERE P1.source = P3.source
                            AND B1.target = P3.target);
      SET oldsize = newsize;
      SET newsize = (SELECT COUNT(*) FROM PathEnum)
    END LOOP;
  END;
END;
```

После этого, можно при помощи 1 запроса к таблице BuildPathEnum получить список всех достижимых из конкретной вершины вершин, вместе с длинной маршрута. Или скажем можно так же за 1 запрос построить список смежности. При этом если мы построим индекс по полю target, то данные запросы будет выполняться за время O(1).

Но у данного подхода есть конечно и свои минусы. Главный минус заключается в том, что таблицу PathEnum необходимо полностью перестраивать после каждой модификации данных о ребрах графа. Хотя поиск в ширину и имеет линейную сложность, тем не менее на больших данных мы получим достаточную задержку. Кроме этого, нужно понимать, что на время пока идет обновление таблицы PathEnum, мы должны блокировать все запросы на получения из нее данных.

Другие минусы вытекают из минусов самих реляционных баз данных, а именно

1. Наличие строгой схемы данных. В задачах, которые перечислены в первой главе, элементы сети обладают совершенно разным набором параметров. Например, в кабельных системах содержаться такие элементы как коммутатор и конечная информационная розетка, свойства которых кардинально разные. Реляционная модель конечно дает решение этой проблемы. Мы можем организовать либо Single Table Inheritance (STI) -

хранение всех данных в одной таблице, сигнатура которой будет являться объединением всех свойств все возможных элементов. Либо хранить все в разных таблицах, но в ребрах прописывать полиморфную связь указывая название таблицы с вершиной. В первом случае мы сталкиваемся с избыточной информацией в таблице вершин, а во втором в усложнении запросов к базе данных. Стоит отметить, что некоторые решения на базе SQL внедряют некоторые решения присущие NoSQL миру, например, PostgreSQL позволяет хранить в поле json объект произвольной структуры и строить по этому полю запросы.

2. Сложности при горизонтальном масштабировании. При работе с задачами, озвученными в первой главе, мы имеет дело с большим объемом данных. Особенно в транспортных задачах. Там размерность графа достигает миллионов вершин. Реляционные системы управления базами данных изначально проектировались без учета того, что база будет масштабирована на несколько машин. Да безусловно существуют проекты, которые обрабатывают колоссальное количество реляционных данных на кластерах из нескольких сотен машин, но настройка такого кластера требует большого количества труда квалифицированных инженеров.
3. SQL крайне плохо работает с древовидными, иерархическими структурами и рекурсивными запросами.
4. Скудные возможности агрегации данных.

2.2.2 Графовые системы управления базами данных

Графовая база данных — разновидность баз данных с реализацией сетевой модели в виде графа и его обобщений. Сетевая модель была одним из первых подходов, использовавшимся при создании баз данных в конце 50-х — начале 60-х годов. В последствии с появлением реляционной модели данных, сетевой подход был вытеснен.

Тем не менее, с возросшей подробности в организации хранения и анализа данных в виде графа, в 2007 был выпущен продукт под названием Neo4j. Который является первой графовой системой управления базами данных.

Графовые базы данных применяются для моделирования социальных графов (социальных сетей) [30], в биоинформатике, а также для семантической паутины[31].

Neo4j очень хорошо справляется с задачей хранения данных и поиска маршрутов, имеет целый ряд встроенных алгоритмов для работы с графами. Там не менее в качестве недостатков к ней можно отнести практически все достоинства реляционных баз. К тому же это решение очень молодо и не успела обзавестись крупной инфраструктурой. Чаще всего данное решение используют как дополнительную базу данных, а не как основную. То есть основные данных хранят в тех же реляционных базах, а связи в neo4j. Так же, хочется отметить, что конечной целью данной работы является построения решения для гиперсетей. Гиперсеть объект намного сложнее чем граф, и реализовать его хранения в базе основанной на графах не представляется возможным.

2.3 Методы представления графов

Представление графа очень важный вопрос. Правильно выбранный способ представления графов может изрядно снизить расход памяти, а иногда и облегчить вычислительную нагрузку при решении некоторых прикладных задач. Например, в целях снижения расхода памяти вместо матрицы смежности можно использовать список смежности, который обладает той же функциональностью, но при этом занимает меньше места в памяти, за счет того, что не хранит избыточной информации о графе.

Одной из задач данной работы является рассмотреть все возможные представления графов и выбрать оптимальный, с точки зрения универсальности и расхода ресурсов метод. Кроме этого, в дальнейшем будут рассмотрены алгоритмы перехода от одного представления к другому. Как говорилось выше, разрабатываемое программное обеспечение должно подходить для решения большинства прикладных задач и давать конечному пользователю выбор, какое представление использовать для решения своей задачи.

Всевозможные представления графов были подробно рассмотрены в работе [6]. Приведем некоторые из них:

1. Матричные

а) Матрица смежности вершин
б) Матрица инцидении ”вершины-ребра
в) Матрица расстояний
г) Матрица смежности ребер
д) Матрица инциденции ”клики-вершины”
е) Хроматические матрицы

2. Списки
   а) Список смежности вершин
   б) Список инцидентных ребер для вершин
   в) Список инцидентных вершин для ребер
   г) Матрица расстояний
   д) Матрица смежности ребер
   е) Матрица инциденции ”клики-вершины”
3. Теоретико множественные
   а) Реберное покрытие
   б) Веерно-вершинное покрытие
   в) Веерно-реберное покрытие
   г) Покрытие цепями

В данной работе, как будет показано ниже, для построения схемы хранения данных графа будут выбраны списки. Списки являются наиболее универсальными и менее ресурсоемким инструментом. Например, список смежности, храня по сути туже информацию, занимает в памяти в разы меньше места чем матрица смежности.

2.4 Методы представления гиперсетей

В работах [6; 21] показывается, что гиперсеть может быть задана двумя матрицам инциденции: ”вершина-ребро”и ”ребро-ветвь”. Следовательно, так же можно перейти к представлению на двух списках инциденции ”вершина-ребро”и ”ребро-ветвь”.

Кроме того, не сложно заметить, что на самом деле. Для того, чтобы представить гиперсеть, нам необходимо:

1. представить граф первичной сети;

2. представить отображение ребер в ветви;

Соответственно, для представления графа можно выбрать любой удобный способ из предыдущей части.

2.5 Представление S-гиперсетей

S-Гиперсети являются еще более сложным объектом, чем гиперсеть. Как видно из определения, S-гиперсеть состоит из двух компонент:

1. Система графов $\{G_i\}$
2. Система отображений $\{F_k^{i,j}s\}$, где каждое отображение F, есть пара отображений вершин в вершины и ребер в ребра.

Рассмотрим s-гиперсеть H построенную на двух графах $G_1(V_1, E_1)$ и $G_2(V_2, E_2)$ и одном отображение $F = \{f_v, f_e\}$, где $f_v = V_1 \mapsto V_2$, а $f_e = E_1 \mapsto E_2$.

Зададим множество $U = \{(v_1, v_2) \mid v_2 = f_v(v_1)\}$.

Очевидно, что множества $V_1 \cup V_2$ и U задают двудольный граф. С другой же стороны, исходя из построения множества U, данный граф является представлением отображения f_v. Следовательно, для представления отображения f_v можно использовать одно из предоставлений графа.

Аналогичным способом можно поступить и с отображением f_e.

2.6 Выбор типа базы данных

В одной из предыдущей части было рассказано о некоторых типах хранилищ данных и показаны методы представления там графов. На сегодняшний день доминирующие позиции занимают реляционные модели данных. Но тем не менее, существует подход NoSQL (Not Only SQL), в рамках которого реализуются системы либо с полным отрицанием SQL, либо с какой то частичное его поддержкой. Данный подход возник в первую очередь из-за необходимости в масштабировании систем и обработки больших данных, то есть для решения тех задач, с которыми реляционная модель справится не может.

Одним из примеров NoSQL баз являются документо-ориентированные базы данных.

Документо-ориентированная система управления базами данных — СУБД, специально предназначенная для хранения иерархических структур данных (документов) и обычно реализуемая с помощью подхода NoSQL. В основе документо-ориентированных СУБД лежат документные хранилища (англ. document store), имеющие структуру дерева (иногда леса). Структура дерева начинается с корневого узла и может содержать несколько внутренних и листовых узлов. Листовые узлы содержат данные, которые при добавлении документа заносятся в индексы, что позволяет даже при достаточно сложной структуре находить место (путь) искомых данных. API для поиска позволяет находить по запросу документы и части документов. В отличие от хранилищ типа ключ-значение, выборка по запросу к документному хранилищу может содержать части большого количества документов без полной загрузки этих документов в оперативную память.

Данные в документо-ориентированных базах хранятся следующим образом:

1. База данных может иметь ноль или более коллекций
2. Коллекции состоят из нуля или более документов
3. Документ состоит из одного или более полей
4. Каждое ”поле”может хранить, либо другой докумен0,т либо данные атомарного типа (строка, число, массив, blob)

При этом нет никакой, заранее определенной схемы данных. В одной коллекции могут содержаться документы совершенно разной структуры.

По сравнению с решениями на базе реляционной модели данных, документо-ориентированно ориентированные системы управления базами данных обладают следующими преимуществами:

1. поддерживают иерархические структуры данных;
2. отсутвие жесткой схемы данных;
3. практически все NoSQL решения изначально проектировались на работу с большими данными. Данные решения прекрасно масштабируются, что позволяет легко настроить высокопроизводительный и отказоустойчивый кластере;
4. очень богатые возможности агрегации данных.;

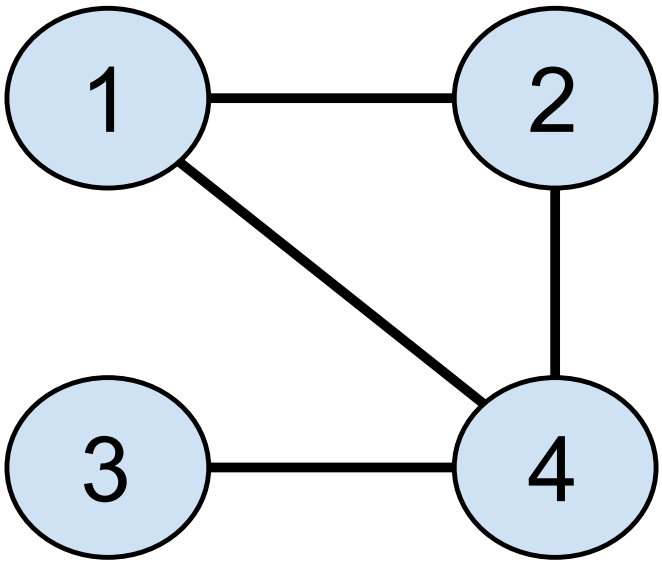


Рисунок 2.1 — Граф

В качестве документо-ориентированной базы рассматривается MongoDB. Данная СУБД управляет наборами JSON-подобных документов, хранимых в двоичном виде в формате BSON.

JSON(англ. JavaScript Object Notation)-текст представляет собой набор пар ключ: значение. Где значение может быть либо экземпляром примитивного типа данных, либо JSON-текстом

BSON (англ. Binary JavaScript Object Notation) — формат электронного обмена цифровыми данными, основанный на JavaScript, бинарная форма представления простых структур данных и ассоциативных массивов (которые в контексте обмена называют объектами или документами). Является надмножеством JSON, включая дополнительно регулярные выражения, двоичные данные и даты.

2.7 Схема хранения графов

Рассмотрим представленный на картинке 2.1 граф . В MongoDB в качестве типа данных для поля ID документа используется ObjectId (12 байтный BSON тип). Для простоты представления данных в листингах в качестве значений полей _id будут взяты номера вершин (ребер), которые указаны на иллюстрации.

Самым простым способом хранения графа в документо-ориентированной базе данных является хранение коллекций вершин, для каждой из которых задан

список смежности. В отличии от большинства реляционных баз, MongoDB допускает использование массива в качестве типа данных для полей документов.

Изображенный на иллюстрации граф будет представлен следующим образом

```
{ _id: 1, adjacent_node_ids: [2,4] }

{ _id: 2, adjacent_node_ids: [1,4] }

{ _id: 3, adjacent_node_ids: [4] }

{ _id: 4, adjacent_node_ids: [1,3] }
```

Достоинства данного подхода:

1. Компактный размер. Отсутствует коллекция ребер
2. Легкость в получении матрицы смежности. Для этого достаточно просто сделать выборку по полю $adjacent\_node\_ids$
3. Легкость в реализации алгоритмов поиска в глубину и в ширину. При выборке вершины мы сразу же получаем список смежности.

Недостатки:

1. Отсутствие возможности задания некоторых свойств для ребер, например, веса, что является критичным в большинстве задач
2. Данным способом невозможно представить граф с кратными ребрами
3. В случае неориентированного графа необходимо контролировать связи между ребрами на обоих концах. Т.е. при добавлении ребра, к примеру, между 3-ей и 1-ой вершиной, необходимо добавить 3-ю вершину в список смежности к 1-ой, а 1-ю к 3-ей.

Теперь рассмотрим ситуацию, когда ребра графа задаются отдельными документами. Данный подход напоминает реляционную модель данных. Для этого пронумеруем все ребра как изображено на 2.2.

Коллекция вершин:

```
{ _id: 1 }

{ _id: 2 }

{ _id: 3 }
```

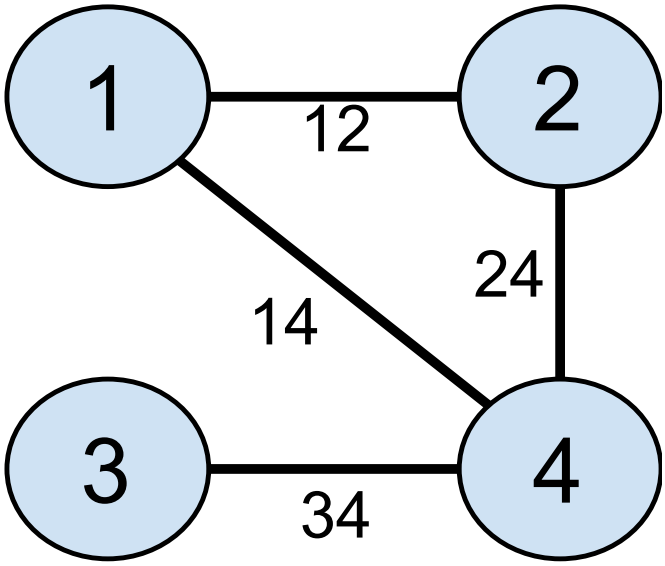


Рисунок 2.2 — Граф с пронумерованными ребрами

```
6
7 { _id: 4 }
```

Коллекция ребер:

```
1 { _id: 12, node_ids: [1, 2] }
2
3 { _id: 24, node_ids: [2, 4] }
4
5 { _id: 14, node_ids: [1, 4] }
6
7 { _id: 34, node_ids: [3, 2] }
```

Достоинства данного подхода:

1. легко добавить любые свойства к ребрам;
2. нет проблем при вставки данных, как вершин, так и ребер;
3. можно представлять графы с кратными ребрами;

Недостатки:

1. при обходе графа в ширину или в глубину, необходимо делать дополнительный запрос на получение смежных вершин;
2. в некоторых случаях представлены избыточные данные;

Второй способ является более общим с точки зрения покрытия всех разновидностей графов. Кроме того, он не привносит критичных проблем с произ-

водительностью. Проблему дополнительных запросов при обходе графа можно решить, построив матрицу или список смежности.

2.8 Схема хранения гиперсетей

Отображения f^e можно представить следующим образом:

```
{ _id: 1, source_id: 1, path: [2,3,4] }
```

Где $source_id$ - ребро в графе G_{i-1}, а $target_ids$ - путь по ребрам в графе G_i
Итого, мы имеем следующую схему базы данных:

1. Коллекция верши - V
2. Коллекция ребер - E
3. Коллекция графов - $Graphs$
4. Коллекция отображений ребер - FE
5. Отображения вершин - FV ($FV \subset E$)

2.9 Общая архитектура программного комплекса.

На иллюстрации представлена схема компонентов системы.

1. Кластер базы данных
2. Веб приложение на elixir/phoenix
3. Балансировщик нагрузок Nginx
4. Node.js фронтед приложение

База данных развернута в кластере из трех серверов. Арбитр, мастер и слейв. Для повышения производительности запросов, можно расширить количество до пяти, семи и т.д.

Веб приложение отвечает за обработку запросов по HTTP. Сохраняет, отдает и модифицирует данные, конвертируя в нужный формат. Основной язык программирования был выбран - Elixir. функциональный, распределённый язык программирования общего назначения, который работает на виртуальной машине Erlang (BEAM). Построен поверх Erlang, что обеспечивает распределённость, от-

казоустойчивость, исполнение в режиме мягкого реального времени. Elixir обладает высокой скоростью работы, богатыми возможностями для распараллеливания программ и приятным синтаксисом. Веб приложение предоставляет JSON-API и GraphQL API для работы с гиперсетями, о них речь пойдет в третьей главе данной работы. Веб приложение так же может быть масштабировано на несколько машин. Балансировку нагрузок между экземплярами серверов обеспечивает веб сервер Nginx.

2.10 Алгоритмы

В данной части будут приведены алгоритмы для получения данных из гиперсети, необходимых для решения прикладных задач, а также доказана их корректность. Например, при работе с графами, часто требуется получить матрицу смежности. Наличие базовых алгоритмов крайне важно для функционирования системы и облегчает работу с пользователей с ней. В большинстве задач исследователю нет необходимости загружать гиперсеть целиком, где-то требуется только один список смежности, где-то нужно получить отображение одного графа в другой. Поэтому система должна уметь отдавать нужные представления для разных типов задач. И должна уметь делать это как можно быстрее. Поэтому в работе была предпринята попытка реализовать большую часть алгоритмов при помощи низкоуровневых запросов в самой базе данных. Данный подход позволит получить максимум производительности от выбранного инструмента. Кроме того, как было описано выше, данный класс систем управления базами данных легко масштабируется, что даст возможность в дальнейшим настроить высоко производительный кластер, для которого не будет проблемой обработать огромный объем данных, скажем, транспортной системы какого-нибудь мегаполиса.

Все алгоритмы будут реализованы с использованием встроенного в БД Aggregation Framework. Каждый алгоритм есть последовательность агрегаций (операторов), которые применяются к выбранной коллекции. Результатом работы является некоторый набор данных из БД.

Aggregation Framework поддерживает следующий набор операторов:

1. match (M) - ограничивает выборку по условиям;
2. addFields (A) - добавляет новые вычислимые поля;

3. project (P) - ограничивает набор полей документов выборки или добавляет новые вычислимые поля;
4. lookup (L) - ”склеивает”документы из разных коллекций;
5. unwind (U) - ”разворачивает”документ по полю, которое является массивом;
6. group (G) - группирует документы по полю;

Приведем формальные определения:

***Определение* 2.10.1.** *Документом будем называть неупорядоченное множество полей и их значений* $\{a_1 :' ...', a_2 :' ...', ...,a_n :' ...'\}$

***Определение* 2.10.2.** *Набором данных будем называть неупорядоченное множество документов.*

***Определение* 2.10.3.** *Оператором фильтрации, match* (M)*, будем называть функцию двух аргументов, действующую следующим образом:*

$$M(S, cond) = \{x | x \textit{ удовлетворяет условию } cond\}$$

Где:

1. S *- набор входных данных*
2. $cond$ *- условия вида:* $\{a_1 : \{o_1 : val_1, ..., o_k : val_k\}, ..., a_n : \{o_1 : val_1, ..., o_l : val_l\}\}$*, где* o_i *операторы запросов.*
3. *элемент* x *удовлетворяет условию* $cond$*, тогда и только тогда, когда каждый атрибут* x *удовлетворяет условию на него из* $cond$

Пример:

```
1   S = [{a: 1, b: "b"}, {a: 2, b: "c"}]
2   M(S, {a: {gt: 1}) = [{a: 2, b: "c"}]
3   M(S, {a: {gt: 0}, b: "b"}) = [{a: 1, b: "b"}]
```

***Определение* 2.10.4.** *Объединением двух документов* $A \cup B$ *будем называть результат объединения множества атрибутов 2 документов. При этом если оба документа имеют одинаковое поле* a*, то в результат попадет только поле последнего документа* (B)*.*

Пример:

$$\{a:1,b:1\} \cup \{b:2,c:3\} = \{a:1,b:2,c:3\}$$

Как не трудно заметить данная операция не является коммутативной.

***Определение* 2.10.5.** *Оператором добавления полей, match (A), будем называть функцию двух аргументов, действующую следующим образом:*

$$A(S,a) = \{x | x = y \cup a, y \in S\}$$

***Определение* 2.10.6.** *Разностью двух документов $A \setminus B$ будем называть документ который содержит все поля документа A , которые не содержатся в документе B.*

***Определение* 2.10.7.** *Оператором ограничения, project (P), будем называть функцию двух аргументов, действующую следующим образом:*

$$P(S,a) = \{x|\}$$

Пример:

```
S = [{a: 2, b: 2}]
P(S, {a: 1, c:3}) = [{a: 2, c: 3 }]
```

***Определение* 2.10.8.** *Оператором развертки, unwind (U), будем называть функцию двух аргументов, которая к каждому документу S применяет следующую функцию $f(x, field)$:*

Если $field \notin x$ или значение поля $field$ в документе x не является массивом, то: $f(x, field) = x$

Если $field \in x$ и $x[field] = [val_1, ..., val_n]$, то: $f(x, field) = [(x \cup \{field : val_1\}), ...(x \cup \{field : val_n\})]$

Пример:

```
S = [\a: 1, b: [1,2]}, {a: 2}]
U(S, "b") = [{a: 1, b: 1]}, {a: 1, b: 2]}, {a: 2}]
```

***Определение* 2.10.9.** *Пусть задано две коллекции документов: C, K. Оператором склейки, lookup (L), будем называть функцию двух аргументов, которая к каждому документу S применяет функцию $f(x, opt) = x \cup y$, где:*

1. *opt документ вида* $\{from : K, localField : l, foreignField : f, as : out\}$
2. *y, документ вида* $\{out : [k_1...k_n]\}$, *где* $k_i \in K$ *такой, что* $k_i[f] == x[l]$

Пример:

```
C = [{a: 1, l: 1}, {a: 2, l:2}]
K = [{b: 1,a : 1}, {b: 2,a :2}]

L(C, {from: K,
      localField: l,
      foreignField: f,
      as: output }
) = [
  {a: 1, l: 1, output: [{b: 1,a : 1}]},
  {a: 2, l:2, outputs: [{b: 2,a :2}]}
]
```

Для краткости будем обозначать:

$$L(X, \{from : K, localField : l, foreignField : f, as : out\}) = L(X, Kasout)$$

Будем обозначать, что оператор F применяется к набору данных S с аргументами $options$ следующим образом - $F(S, options)$.

Так как результатом работы оператора так же является некоторый набор данных, то справедливо понятие композиции операторов. В функциональном программировании принято обозначать композицию функций через специальный pipeline operator ($F \triangleright G$), который означает, что результат, возвращаемый функцией F, передается в качестве первого аргумента в функцию G.

Будем считать, что следующие две записи эквивалентны:

$$F(G(optionsForG), optionsForF)$$

$$G(optionsForG) \triangleright F(optionsForG)$$

Возьмём гиперсеть со структурой, представленной н 2.3

Обозначим:

1. V_H1 - Выборка всех вершин графа $H1$
2. E_H1 - Выборка всех ребер графа $H1$

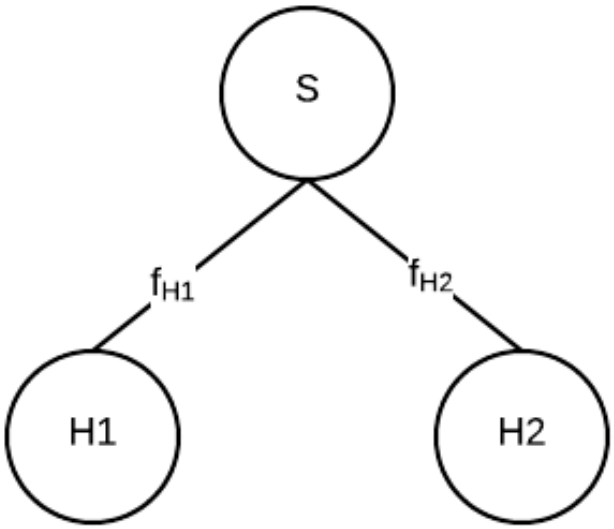


Рисунок 2.3 — Структура гиперсети для алгоритмов

3. FV^S_{H1} - Выборка отображений вершин из графа $H1$ в граф S
4. FE^S_{H1} - Выборка отображений ребер из графа $H1$ в граф S
5. V_H2 - Выборка всех вершин графа $H2$
6. E_H2 - Выборка всех ребер графа $H2$
7. FV^S_{H2} - Выборка отображений вершин из графа $H2$ в граф S
8. FE^S_{H2} - Выборка отображений ребер из графа $H2$ в граф S

Далее, построим алгоритмы получения следующих данных:

1. Списки инцидентности/смежности графов
2. Список смежности объединенного графа
3. Список инцидентности вершин H_1/H_2 к вершинам S
4. Список инцидентности ребер H_1/H_2 к ребрам S
5. Список слабой инцидентности

2.10.1 Список смежности графа

Здесь и далее под списком инцидентности/смежности графа будем иметь ввиду все списки инцидентности/смежности для всех вершин графа

Алгоритм для неориентированного гиперграфа:

$$E_{H_1} \tag{2.1}$$
$$\triangleright A(\{sources : nodes\}) \tag{2.2}$$
$$\triangleright U(\text{"}sources\text{"}) \tag{2.3}$$
$$\triangleright U(\text{"}nodes\text{"}) \tag{2.4}$$
$$\triangleright A(\{cond : node == source\}) \tag{2.5}$$
$$\triangleright M(\{cond : false\}) \tag{2.6}$$
$$\triangleright G(\text{"}source\text{"}) \tag{2.7}$$

Где:

1. $A(\{sources : nodes\})$ - копирование массива $nodes$ в $sources$
2. $U(\text{"}sources\text{"})$ - "развертка"массива $sources$
3. $U(\text{"}nodes\text{"})$ - "развертка"массива $nodes$
4. $A(\{cond : node == source\}) \triangleright M(\{cond : false\})$ - исключаем все записи, где $source$ равен $node$
5. $G(\text{"}source\text{"})$ - группировка по полю $source$

***Теорема* 2.10.1.** *Приведенный алгоритм генерации списка смежности является корректным.*

Доказательство. Как видно из построения алгоритма, он генерирует набор документов вида:

$$\{source : v_1, nodes : [v_2, v_3...]\}$$

Где в поле $source$ находится идентификатор вершины, а в поле $nodes$ - список смежных вершин.

Чтобы доказать корректность алгоритма на необходимо:

1. показать, что выходной результат содержит все документы для всех не изолированных вершин;
2. показать, что список $nodes$ содержит все смежные вершины;
3. показать, что список $nodes$ не содержит не смежной вершины с $source$;

Обозначим множество результатов алгоритма через R.

Предположим вначале что для вершины v не существует документ в R, но существует ребро e которое соединяет ее с вершиной u. Фильтрация происходит только на шаге 2.1 и 2.6, из чего легко сделать вывод, что либо $v = u$, либо $e \notin E_{H_1}$

.

Покажем, что массив $nodes$ содержит все вершины смежные с выбранной вершиной v.

Пусть существует вершина u и ребро e которое соединяет вершины u и v. Предположим, что в R не существует документ с ключом $source : v$ в котором массив $nodes$ не содержит идентификатора вершины u. По аналогии с предыдущими рассуждениями, получаем, что либо $v = u$, либо $e \notin E_{H_1}$, следовательно массив $nodes$ содержит идентификатор вершины u.

Покажем, что массив $nodes$ не содержит вершин несмежных с выбранной вершиной v.

Предположим, что не существует ребра соединяющего вершины v с вершиной u, но при этом массив $nodes$, документа с ключом $source : v$ из R, содержит идентификатор вершины u. Очевидно, что чтобы такое было возможно, E_{H_1} должен содержать хотя бы один элемент в котором массив $nodes$ содержит как минимум идентификатор и вершины v и вершины u. Так как множество E_{H_1} есть множество вершин графа H_1, то приходим к противоречию.

□

2.10.2 Список инцидентности вершин H1/H2 к вершинам графа S

Алгоритм в точности совпадает с представленным выше, только в качестве начальных данных берется $FV^S_{H_1}$:

$$FV^S_{H_1} \tag{2.8}$$

$$\triangleright A(\{sources : nodes\}) \tag{2.9}$$

$$\triangleright U(\text{"}sources\text{"}) \tag{2.10}$$

$$\triangleright U(\text{"}nodes\text{"}) \tag{2.11}$$

$$\triangleright A(\{cond : node == source\}) \tag{2.12}$$

$$\triangleright M(\{cond : false\}) \tag{2.13}$$

$$\triangleright G(\text{"}source\text{"}) \tag{2.14}$$

Так как шаги алгоритма в точности совпадают с алгоритмом нахождения списка смежности, кроме первого шага, то и рассуждения по доказательству его корректности аналогично приведенным выше.

2.10.3 Список смежности объединенного графа

Объединенный граф Q получается путем объединения множеств вершин графов H_1, H_2 и S и множеством ребер $E_Q = E_{H_1} \cup E_{H_2} \cup FV^S_{H_1} \cup FV^S_{H_2}$.

Принцип построения остается прежним:

$$E_Q \tag{2.15}$$
$$\triangleright A(\{sources : nodes\}) \tag{2.16}$$
$$\triangleright U(\text{”}sources\text{”}) \tag{2.17}$$
$$\triangleright U(\text{”}nodes\text{”}) \tag{2.18}$$
$$\triangleright A(\{cond : node == source\}) \tag{2.19}$$
$$\triangleright M(\{cond : false\}) \tag{2.20}$$
$$\triangleright G(\text{”}source\text{”}) \tag{2.21}$$

Очевидно, что таким способом можно получить различные списки смежности для разных графов, необходимо только подменить начальное множество, к которому впоследствии применить уже известную цепочку операторов.

2.10.4 Слабая инциденция

Важным понятием в теории гиперсетей является понятие слабой инциденции элементов гиперсети.

***Определение* 2.10.10.** *Два элемента из различных множеств слабо инцидентны, если найдется элемент из третьего множества, инцидентный им обоим. [1]*

Пример: вершина $x \in X$ слабо инцидентна ребру $r \in R$, если и только если существует элемент $v \in V$, такой что $x \in P(v)$ и $v \in F(r)$, то есть $x \in P(F(r))$. В

рассматриваемой гиперсети, например, для вершины 2 слабо инцидентны ребра 12, 14, 23. Причем, ребра 12 и 23 - инцидентны в классическом понимании.

Алгоритм генерации списка слабой инциденции

$$FE^S_{H_1} \tag{2.22}$$

$$\triangleright L(E_{H1} as edge) \tag{2.23}$$

$$\triangleright U(\text{"}edge\text{"}) \tag{2.24}$$

$$\triangleright U(\text{"}edge.nodes\text{"}) \tag{2.25}$$

$$\triangleright G(\text{"}edge.nodes\text{"}) \tag{2.26}$$

***Теорема* 2.10.2.** *Приведенный алгоритм генерации списка слабой инциденции является корректным.*

Доказательство. Пусть существует вершина $v \in V^{H_1}$, которая слабо инцидентна ребру $e \in E_S$ Предположим, что в результатах работы алгоритма в списке ребер для вершины отсевает ребро e.

Из определения слабой инциденции следует, что существует ребро $u \in E_{H_1}$, который инцидентен вершине v и и ребру e (через отображение $f^e_{S-H_1}$). Следовательно во множестве $FE^S_{H_1}$ присутствует документ, с $source = u$, поле $path$, которого содержит ребро e. На втором, третьем и четвертом шаге выполняется "развертка"ребра по инцидентным ему вершинам в результате чего в выборке появится документ с полем $node = v$ поле $path$, которого содержит e. Шаг группировки, группирует документы по полю $node$ сливая в один массив массивы $path$. В силу определения оператора группировки, в результат алгоритма будет присутствовать документ для ребра v в списке ребер у которого будет содержаться ребро e.

Предположим теперь, что в массиве для вершины $v \in V^{H_1}$ существует $e \in E_S$, которое не слабо инцидентно v. По построению алгоритма и определениям операторов легко прийти к выводу, что множество $FE^S_{H_1}$ должно содержать документ в массив $path$, которого входит e. Следовательно существует $u \in E_{H_1}$, которое инцидентно и v и e. Следовательно, v слабо инцидентно e.

□

2.11 MapReduce алгоритмы

MapReduce — это фреймворк для решения некоторых наборов распределенных задач с использованием большого количества компьютеров (называемых «нодами»), образующих кластер.

Работа MapReduce состоит из двух шагов: Map и Reduce.

На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным злом — master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам — worker node) для предварительной обработки. Название данный шаг получил от одноименной функции высшего порядка.

На Reduce-шаге происходит свёртка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат — решение задачи, которая изначально формулировалась.

Преимущество MapReduce заключается в том, что он позволяет распределено производить операции предварительной обработки и свертки. Операции предварительной обработки выполняются независимо друг от друга и могут производиться параллельно (хотя на практике это ограничено источником входных данных и/или количеством используемых процессоров). Аналогично, множество рабочих узлов могут осуществлять свертку — для этого необходимо только чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени. Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, MapReduce может быть применен к большим объёмам данных, которые могут обрабатываться большим количеством серверов. Так, MapReduce может быть использован для сортировки петабайта данных, что займет всего лишь несколько часов. Параллелизм также дает некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свертки, возникает сбой, то его работа может быть передана другому рабочему узлу (при условии, что входные данные для проводимой операции доступны).

На основе данного подхода можно реализовать ряд алгоритмов над графами.

2.11.1 Поиск в ширину при помощи MapReduce

Рассмотрим задачу поиска в ширину.

Данный подход был предложен Jimmy Lin и Chris Dyer в [32]

Идея: будем применять операцию MapReduce к графу до тех пор, пока не дойдем до искомой вершины. Одна итерация п росматривает все достижимые вершины из множества пройденных и вычисляет кратчайшие расстояния

Представление данных:

Данные представляются парами: (key, value) где:

1. Key - вершина n (идентификатор вершины)
2. Value - d - расстояние от вершины n до начала, adjacent_list - вершины доступные из n

Инициализация для всех вершин, кроме начальной, задать $d = \infty$. Как только для искомой вершины значение d будет меньше ∞, тогда можно останавливать алгоритм.

Map функция:

$$emit(n, d)$$

$$\forall m \in adjacent\_list : emit(m, d + 1)$$

Функция $emit(n, d)$ генерирует значение с ключом n и значением d.

Sort/Shaffle: Сгруппировать расстояния по достижимым вершинам

Reduce: Для каждой вершины выбираем путь с минимальным расстоянием.

Рассмотрим алгоритм поиска маршрута из вершины 1 до вершины 2 для представленного выше графа.

Начальные данные:

```
1 { _id: 1, adjacent_node_ids: [2,3], d: 0 }
2
3 { _id: 2, adjacent_node_ids: [1,3], d: inf }
4
5 { _id: 3, adjacent_node_ids: [1,2], d: inf}
6
7 { _id: 4, adjacent_node_ids: [3],   d: inf }
```

Итерация 1:

Результат выполнения функции Map:

```
// map(1)
{ _id: 1, d: 0 }
{ _id: 2, d: 1 }
{ _id: 1, d: 1 }

// map(2)
{ _id: 2, d: inf }
{ _id: 1, d: inf }
{ _id: 3, d: inf }

// map(3)
{ _id: 2, d: inf }
{ _id: 1, d: inf }
{ _id: 3, d: inf }
{ _id: 4, d: inf }

// map(4)
{ _id: 4, d: inf }
{ _id: 3, d: inf }
```

Результат выполнения функции Shaffle

```
{ _id: 1, d: [0, inf, inf] }
{ _id: 2, d: [1, inf, inf] }
{ _id: 3, d: [1, inf, inf] }
{ _id: 4, d: [inf, inf] }
```

Результат выполнения функции Reduce

```
{ _id: 1, d: 0 }
{ _id: 2, d: 1 }
{ _id: 3, d: 1 }
{ _id: 4, d: inf }
```

Как видно, за одно итерацию мы посетили вершины 2 и 3, тем не менее у вершины 4 значение d осталось равным бесконечности, критерий остановки не выполнен.

Итерация 2: Результат выполнения функции Map:

```
// map(1)
{ _id: 1, d: 0 }
{ _id: 2, d: 1 }
{ _id: 1, d: 1 }

// map(2)
{ _id: 2, d: 1 }
{ _id: 1, d: 2 }
{ _id: 3, d: 2 }

// map(3)
{ _id: 2, d: 2 }
{ _id: 1, d: 2 }
{ _id: 3, d: 1 }
{ _id: 4, d: 2 }

// map(4)
{ _id: 4, d: inf }
{ _id: 3, d: inf }
```

Результат выполнения функции Shaffle

```
{ _id: 1, d: [0, 2, 2] }
{ _id: 2, d: [1, 1, 2] }
{ _id: 3, d: [1, 2, inf] }
{ _id: 4, d: [inf, 2] }
```

Результат выполнения функции Reduce

```
{ _id: 1, d: 0 }
{ _id: 2, d: 1 }
{ _id: 3, d: 1 }
{ _id: 4, d: 2 }
```

На этой итерации алгоритм добрался до четвертой вершины и посчитал расстояние до нее. Критерий остановки выполнен. Алгоритм завершен.

Плюсом такой реализации алгоритма поиска является возможность выполнять функции map и reduce параллельно. С другой стороны, имеются накладные расходы на посещения вершин из множества уже просмотренных.

Замечание: Из функции map можно убрать генерацию значений, если $d = \infty$.

2.11.2 Алгоритм Дейкстры при помощи MapReduce

Задача: Дан взвешенный ориентированный граф $G(V, E)$ без дуг отрицательного веса. Найти кратчайшие пути от некоторой вершины a графа G до всех остальных вершин этого графа.

Для построения алгоритма необходимо немного модифицировать алгоритм поиска в ширину, предложенный выше.

1. для каждой вершины зададим вес;
2. изменим генерацию значений в функции map: $emit(n, d + w_n)$, где w_n - вес вершины n;
3. изменим критерий остановки. Алгоритм останавливается, если итерация не изменяет ни одного расстояния;

2.11.3 Поиск компонент связности

***Определение* 2.11.1.** *Компонента связности графа G - это подграф $G(U)$, порождённый множеством $U \subseteq V(G)$ вершин, в котором для любой пары вершин $u, v \in U$ в графе G существует (u, v) - цепь и для любой пары вершин $u \in U$, $w \notin U$ не существует (u, w) - цепи.*

Задача: Для заданного графа G найти количество компонент связности.

Модифицируем алгоритм поиска в ширину следующим образом:

1. Параметр d будет отвечать не за расстояние от начальной вершины, а за цвет. На момент инициализации зададим для каждой вершины $d = \_id$.

2. Изменим генерацию значений в функции map: $emit(n, d)$. Т.е. Для каждой вершины, окрашиваем все смежные с ней вершины в ее цвет.
3. Функция reduce выбирает минимальный цвет для каждой вершины.
4. Критерий остановки: Алгоритм останавливается, если за итерацию не поменялся цвет ни одной из вершин.

Количество компонент связности - количество различных цветов вершин.

Недостаток данного подхода заключается в том, что каждая итерация алгоритма может окрасить только вершины, непосредственно смежные с компонентной связности. Это приведет к тому, что если на шаге n у нас останется только 2 компоненты связности, которые имеют между собой мост, то нам понадобится еще некоторое кол-во шагов (в худшем случае, равное мощности одной из компонент) что бы завершить алгоритм.

Поэтому имеет смысл при наличии малого числа компонент запускать после итерации алгоритм слияния смежных компонент:

1. Для каждой компоненты связности U находим множество C_U цветов смежных ей вершин.
2. Каждую вершину графа с цветом из множеств C_U окрашиваем в цвет компоненты U

2.12 Численные эксперименты

Численные эксперименты проводились над алгоритмом генерации списка слабой инциденции. Графы S, H_1 имели одинаковую структуру ребра графа H_1 один к одному отображались в ребра графа S

Варианты графов:

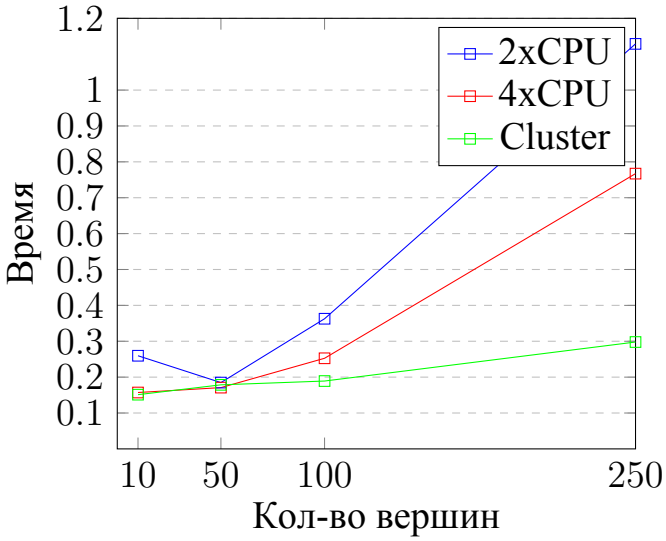
1. $K10$ - полный граф на 10 вершинах;
2. $K100$ - полный граф на 100 вершинах;
3. $K250$ - полный граф на 250 вершинах;
4. $G1000$ - граф-сетка на 1000 вершинах;
5. $G10000$ - граф-сетка на 10000 вершинах;
6. $G100000$ - граф-сетка на 10000 вершинах;

Эксперименты проводились на следующих конфигурациях:

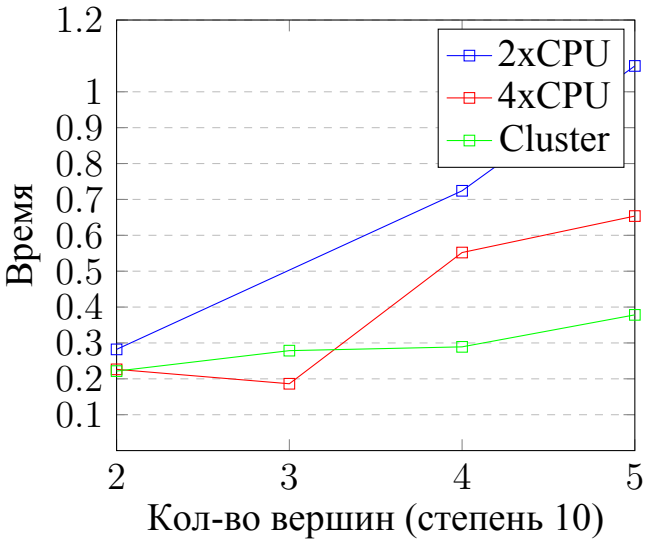
1. 2 ядерный процессор intel i5 2.6GHz;

2. 4 ядерный процессор intel Xeon E3-1231v3
3. Кластер из 5 виртуальных серверов с 2 виртуальными ядрами на каждом

Время работы алгоритма для полных графов



Время работы алгоритма на графах-решотках



2.13 Выводы

В данной главе была спроектирована структура базы данных для хранения гиперсетей. Так как понятие гиперсети тесно связано с графами, были рассмотрены существующие методы представления и хранения графов. Был проведен анализ типов баз данных и выбрал наиболее подходящий под поставленные задачи.

Описана схема хранения данных и построены некоторые алгоритмы, которые позволяют получить различные представления графов и гиперсетей. Данные

алгоритмы показывают, что данная схема позволяет быстро переходить от одного представления к другому. Конечно, существует избыточность данных в таком способе хранения, но для конечного пользователя существует возможность получения только тех данных которые ему необходимы для решения задачи. К тому же объем расходуемой памяти не растет экспоненциально при увеличении количества вершин, ребер графов или правил отображения. Что при нынешнем развитии вычислительной техники делаем избыточность данных не таким серьезным недостатком.

Численный эксперимент показал, во-первых, что данные алгоритмы, действительно, работают быстро даже на довольно больших объемах данных. Во-вторых, что представленная архитектура имеет возможность к горизонтальному масштабированию, что позволит ускорить алгоритмы для действительно больших данных. Даже небольшой кластер способен увеличить в разы производительность алгоритмов, за счет параллельной обработки данных. А при наличии достаточных вычислительных ресурсов есть возможность разворачивать кластера баз данных на десятках машин. Данный факт еще раз доказывает правильность выбора конкретной системы управления базами данных.

Глава 3. Методы взаимодействия с системой

Для успешного взаимодействия и решения прикладных задач на базе построенного программного комплекса, представлено три уровня взаимодействия с системой:

1. JSON-API
2. GraphQL API
3. Hypernet DSL

3.1 JSON-API

3.1.1 Базовые понятия

Приведем некоторые понятия из веб технологий

HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных.

Основой HTTP является технология «клиент-сервер», то есть предполагается существование:

1. Клиентов, которые инициируют соединение и посылают запрос;
2. Серверов, которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом

Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке:

1. Стартовая строка - определяет тип сообщения;
2. Заголовки — характеризуют тело сообщения, параметры передачи, метод, ключи авторизации и другое;
3. Тело сообщения — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой. Тело может отсутствовать

Заголовок запроса всегда должен содержать метод, HTTP поддерживает следующий набор методов:

1. OPTIONS
2. GET
3. HEAD
4. POST
5. PUT
6. PATCH
7. DELETE
8. TRACE
9. CONNECT

Ответ сервера должен содержать код состояния цело, трехзначное число.

URI (англ. Uniform Resource Identifier) — унифицированный идентификатор ресурса. Который соответвено может однозначно идентифицировать некий ресурс.

Единый указатель ресурса (англ. Uniform Resource Locator) — единообразный определитель местонахождения ресурса. Это то, что каждый человек может видеть в адресной строке браузера. Например URL статьи в википедии про URL - https://ru.wikipedia.org/wiki/URL

Одна из частей URL - URL-путь (далее просто ”путь”), это все точно находится после основного домена. В нашем примере это /wiki/URL

3.1.2 О JSON-API

JSON-API - спецификация для построения API (программный интерфейс приложения, интерфейс прикладного программирования) (англ. application programming interface, API) Данная спецификация декларирует то как должны выглядеть ответы и запросы к серверу от сторонних программ по протоколу HTTP. Спецификация также декларирует метод работы со связями объектов, что в нашем случае очень удобно. JSON API предназначен для минимизации количества запросов и количества данных, передаваемых между клиентами и серверами.

Основой JSON-API является ресурс. Ресурс — это некий объект с определенным количеством и связей и который поддерживает один или несколько следующих методов:

1. Create. Создание ресурсы
2. Index. Получения списка ресурсов
3. Show. Возвращает ресурс по уникальному идентификатору
4. Update. Обновляет данные ресурса
5. Destroy. Удаляет ресурс по уникальному идентификатору

Пусть в нашей системе существует ресурс - вершина (node). Тогда json-api декларирует, что сервер должен поддерживать следующие url-ы:

1. /nodes с методом GET, который возвращает список вершин
2. /nodes/:id с методом GET, который возвращает вершину с указным идентификатором
3. /nodes с методом POST, который создает новую вершину
4. /nodes/:id с методом PUT/PATCH, который обновляет данные вершины с указным идентификатором
5. /nodes/:ids с методом DELETE, который удаляет вершину с указным идентификатором

В каждом запросе к JSON-API должен присутствовать заголовок - ”Accept: application/vnd.api+json”

Кроме соглашения об именовании URL, рассматриваемый стандарт ставит четкие требования к формату передачи данных.

3.1.3 Базовые типы данных

JSON документ должен быть в корне каждого запроса и ответа, содержащего данные. Этот объект определяет «верхний уровень» документа.

Документ должен содержать как минимум один из следующих ключей на верхнем уровне:

1. data - основные данные документа;
2. errors - массив ошибок;
3. meta - различная метаинформация;

При этом элементы errors и data не могут содержаться в одном документе.

Верхний уровень документа может так же опционально содержать следующие ключи:

1. jsonapi - описание серверной информации, например, версию стандарта;
2. links - содержит массив документов типа Link (”ссылка”);
3. included - содержит любые связанные ресурсы. Используется чтобы дать возможность получить документ со всеми связями за один запрос:

Документ тип ”ссылка может содержать один из следующих ключей:

1. self - простая ссылка (url), которая сгенерировала данный документ;
2. related - ссылки на другие, связанные с текущим, ресурсы. Содержит ключи: href - сама ссылка, meta - для метаинформации;

В ключе data может содержаться либо один документ типа Resource, либа массив таких документов, либо null.

Документ типа ”ресурс”обязан содержать следующие ключи:

1. id - уникальный идентификатор;
2. type - тип ресурса;

Документ типа ”ресурс”может опционально содержать следующие ключи:

1. attributes - атрибуты ресурса в виде корректного JSON документа;
2. relationships - документ типа Relationship (отношение) описывающий отношения между текущем Документ и любыми другими;
3. links - содержит массив документов типа Link
4. mets - по аналогии с метаинформацией на ”верхнем уровне”

Документ типа Relationship должен содержать один из перечисленных ключей

1. links - содержит массив документов типа Link
2. data - содержит документ типа Resource Linkage (связь ресурсов)
3. meta - метаинформация

Resource Linkage позволяет связать полученные ресурсы без дополнительных запросов к серверу. Документ данного типа, в зависимости от типа связи (к одному, ко многим) может принимать следующие значения:

1. null - для пустого отношения ”к одному”;
2. пустой массив - для пустого отношения ”ко многим”;
3. документ с ключами id и type для непустого отношения ”к одному”;
4. массив документов с ключами id и type для непустого отношения ”ко многим”;

3.1.4 Фильтрация и сортировки

Для сортировки ресурсов, в переменных запроса можно передать ключ sort, в котором могут быть перечислены (через запятую) названия полей документа по которым надо фильтровать результаты. Для сортировки в обратном порядке к названию поля, необходимо добавить знак −.

Для фильтрации результатов, в переменных запроса можно передать ключ filter. JSON-API не декларирует явные требования к виду данных в этом параметре. Это зависит от реализации фильтрации на стороне сервера. Для системы о которой идет речь в этой работе, приведены примеры в приложении.

3.1.5 Реализованные ресурсы

В рамках данной работы были реализованы следующие ресурсы, которые советуют приведенной выше спецификации:

1. Гиперсеть. URL - /hypernets;
2. Граф. URL - /graphs
3. Вершина. URL - /nodes
4. Ребро. URL - /edges
5. Отображение. URL - /mappings

В приложении приведены примеры запросов для каждого из ресурсов.

3.2 GrapQL

GraphQL — это стандарт декларирования структуры данных и способов получения данных с сервера. В отличии от JSON-API, здесь клиент, который взаимодействует с сервером, декларирует те данные которые он хочет получить. Допустим если я хочу получить только идентификаторы всех вершин то через JSON-API я этого сделать не смогу, метод вернет полностью все данные о всех верши-

нах. В graphQL же пользователь в запросе может это указать и сервер должен будет под это подстроится.

В GraphQL есть два полностью отделенных друг от друга понятия: тип запроса (формат данных о нем) и способ его получения.

Допустим у нас есть 2 типа данных, вершина и ребро:

```
type Node {
  id: ID
  name: String
  edges: [Edge]
}

type Edge {
  id: ID
  name: String
  nodes: [Node]
}
```

Это описывает только форматы данных, но это описание абсолютно ничего говорит о там как эти данные могут быть извлечены с сервера.

Что бы действительно получить доступ к списку вершин или ребер, нам в нашей схеме необходимо создать тип Query:

```
  type Query {
    nodes: Node
    edges: Edge
  }
```

Теперь можно послать обычный HTTP запрос на получения данных:

```
GET /graphql?query={ nodes { name, edges { name } } }
```

Сервер примет этот запрос и преобразует в набор запросов к базе данных, извлечет данные и преобразует все это в в нужный формат ответа. В итоге будет получен список инциденции вершин с ребрами в следующем виде:

```
[
  {
    name: "v1",
    edges: [{name: "e1", name: "e2"}]
```

```
  }
]
```

Соответственно, чтобы получить просто список вершин нужно выполнить следующий запрос:

```
GET /graphql?query={ nodes { name } } }
```

Как видно из примеров, в слуае с GraphQL, клиент сам решает в каком формате получить данные. В случае JSON-API, для получения списка смежности нам нужно было бы описывать дополнительный ресурс, тут же мы все сделали в рамках единого метода.

Рассмотрим теперь запросы на изменения данных. Для этого необходимо объявить в схеме тип Mutation:

```
type Mutation {
  addNode(data: AddNodeData): Node
}

input AddNodeData {
  name: String
}
```

Теперь создать вершины мы можем следующим запросом со следующим телом:

```
mutation {
  addNode(input: {
    name: "v1",
  }) {
    id
  }
}
```

Данный запрос создаст вершину и вернет ее идентификатор. Так же можно строить более сложные запросы с вложенными объектами, например, создать 2 вершины и соединить их ребром в одни запрос, или даже целиком описать создание гиперсети.

Как показывают примеры выше, GrapQL предлагает более удобный, в некоторых случаях, способ получения данных. Как было сказано в первой главе, для большинства задач нет необходимости получать все данные гиперсети, часто нам даже не нужно получать все данные вершины графа, так как в основном нам надо знать только ее связи с другими вершинами.

3.3 Hypernet DSL

Кроме низкоуровневых API, в рамках данной работы был построен DSL (Domain-specific language, Предметно-ориентированный язык), данный язык является расширением языка программирования общего назначения ruby. Данный язык разрабатывался главным образом, для того, чтобы дать пользователю человек понятный и выразительный способ описания гиперсети. Так же, одной из целей была инкапсуляция сетевого взаимодействия. Для того, чтобы пользоваться приведенными выше JSON-API и GraphQL, все же необходимы определенные познания в области веб технологий. И пользователь должен будет использовать некоторую библиотеку для взаимодействий по сети. Данный же DSL все делает сам, давая пользователю лишь определенный набор объектов и функций.

Так как предлагаемый язык является расширением языка ruby, то он содержит все базовые конструкции такие как: условия, циклы, обработчики ошибок, и т.д.

Язык базируется на 5 основных классах:

1. Hypernet - отвечает за операции с гиперсетями
2. Graph - отвечает за операции с графами
3. Node - отвечает за операции с вершинами
4. Edge - отвечает за операции с ребрами
5. Mapping - отвечает за операции с отображениями
6. NodeMapping - отвечает за операции с отображениями вершин
7. EdgesMapping - отвечает за операции с отображеними ребер

Каждый класс имеет следующие методы:

Метод создания экземпляра с сохранением на сервере (create). Параметры объекта можно передать как в аргументах в методы так и заполнить в передаваемом блоге кода.

Примеры:

```
# Передачаназваниевершинывкачествеаргумента      :
Node.create name: 'v1'

# Передачаназваниявершинывблокекода      :
Node.create do |n|
  n.name = 'v1'
end

# Созданиеграфана   3 вершинах:
Graph.create(name: 'G1') do |graph|
  3.times do |i|
    graph.nodes << Node.new(name: "v#{i}")
  end
end
```

Метод поиска в базе данных (query). Метод делает запрос к серверу и возвращает все элементы с заданным запросом. Если аргументы не были переданы, то вернет все записи.

Примеры:

```
# Создадим 3 вершинысразнымииименами
3.times { |i| Node.create name: "v#{i}" }

Node.query(name: 'v1').map &:name
# => ['v1']

Node.query.map &:name
# => ['v1', 'v2', 'v3']
```

Метод поиска объекта по уникальному идентификатору (find).

Пример:

```
Node.find(1)
```

Каждый экземпляр перечисленных классов имеет следующие методы.

Метод удаления (destroy)

```
# Создадимобъектисразуегоудалим      :
node = Node.create
node.destroy

# Удалениеобъектапо   id
Node.find(1).destroy
```

```
# Удалениевсехвершинсзаданнымиименем
Node.query(name: 'v1').each { |n| n.destroy }
```

Метод обновление данных (update). Работает аналогичным образом как метод создания, но может быть вызван только у экземпляра класса.

Метод сохранения данных на сервер (save) Примеры обновления данных посредствам методов save и update

```
node = Node.find(1)
node.name = 'v2'
node.save

node.update name: 'v3'
```

Каждый объект перечисленных классов содержит все атрибуты и все связи доступные через JSON-API. Связи используют ”ленивую”загрузки, чтобы получить все связи за один запрос, можно использовать параметр includes. Добавление объекта в связь осуществляется методами: «, create.

3.4 Выводы

В данной главе продемаонстрированы средства работы с построенным сервисом. каждый из них обладает разной степенью абстракции, своими плюсами и минусами.

JSONAPI подойдет в случаях когда нужно быстро поучить набор каких то простых данных по сети или встроить сервис как источник данных в разрабатываемое приложение.

GraphQL стоит использовать, когда нужно получать данные в более сложном виде.

При помощи dsl можно писать программы для импорта/экспорта данных. Нопример можно получить данные из базы и сохранить в файл для дальнейшей передаче алгоритму. Кроме того, можно реализовывать алгоритмы на языке ruby и удобно взаимодействовать с сервисом.

Глава 4. Примеры практических задач

4.1 Прокладка сетей инженерно-технического обеспечения в подземных коллекторах

Под сетями инженерно-технического обеспечения понимается совокупность сооружений и коммуникаций, непосредственно используемых в процессе тепло-, газо-, электро-, водоснабжения и водоотведения.

Коллектор для инженерных коммуникаций – проходной тоннель для прокладки и обслуживания инженерных коммуникаций, не сообщающийся с другими подземными сооружениями и оборудованный внутренними инженерными системами.

Виды коллекторов:

1. коллекторы-трубопроводы — трубы большого диаметра и тоннели, служащие для пропуска различных жидкостей;
2. специальные коллекторы (каналы), в которых размещают один вид подземных сетей;
3. общие, или совмещенные коммуникационные коллекторы для совместной прокладки трубопроводов и кабелей различного назначения;

Данная задача актуальна при проектировании новых жилых районов городов, а также при модернизации старых. Применение оптимального решения по упаковки инженерных сетей позволит не только сэкономить значительные средства на монтаж, а так же значительно редуцировать затраты на поддержку и мониторинг сетей.

4.1.1 Постановка задача

Существует три основных типа коллекторов:

1. Магистральные. Коллекторы между районами
2. Локальные районные коллекторы
3. Подводка коммуникаций к дому

На основе ГОСТ-ов и СНиП-ов выделены следующие ограничения на проектирование коллекторов:

1. Общие правила размещения
    а) Инженерные коммуникации в поперечном сечении могут размещаться с двух сторон или с одной стороны.
    б) Требования, определяющие габариты коллектора:
        1) Ширина прохода должна быть на 0.1м больше диаметра трубопровода, размещаемого в коллекторе, но не менее 0.8м. При двухстороннем размещении электропроводов, ширина прохода должна быть не менее 1м
        2) Высота должна быть не мене 1.8м
2. Теплопроводы
    а) В коллекторе, расположение теплопровода следует предусматривать в 2 яруса. На нижнем - подающий теплопровод, на верхнем - обратный.
    б) Расстояние между изолированными теплопроводами, а также между теплопроводами и кабелями связи и электропроводами по вертикале не должно быть меньше 0.2м.
    в) В коллекторе, компенсацию тепловых удлинение следует предусматривать в соответствии с требованиями сп 124.13330. Для этого по трассе теплопровода необходимо предусмотреть повороты трассы, П-образные компенсаторы, а на прямых участках - установку сильфоновых компенсаторов. Участки компенсации следует разделять неподвижными опорами с креплением на них теплопроводов.
3. Водопроводы
    а) Прокладку водопровода следует предусматривать на бетонных, железобетонных или металлических опорах, которые при необходимости имеют конструктивную связь с отделкой коллектора. Между опорой и трубой, а также между хомутом и трубой необходимо предусмотреть диэлектрическую прокладку.
    б) Расстояние от водопроводных труб, до отделки коллектора или других коммуникаций, следует принимать не менее 200мм

в) Следует предусмотреть теплоизоляцию и антикоррозийную защиту труб

4. Кабели

а) Прокладку электрических кабелей необходимо предусмотреть по потолкам или в лотках, которые опираются на консоли. Кабеля связи непосредственно по консолям.

б) Кабели инженерного оборудования необходимо размещать на полках или лотках в верхней части коллектора, при этом расстоянии от консоли до конструкции коллектора по вертикали должно быть не менее 0,15м. а длину консоли не более 0,6м.

в) Расстояние между консолями кабелей связи по вертикали не менее 0,15, длинна не более 0,63м.

г) Расстояние от верха консоли кабеля связи до полки электрических кабелей не менее 0.2м.

д) Для электрических кабелей с напряжением до 35кВ расстояние между консолями по вертикале не менее 0,25м, длинна консоли не более 0,5м а расстояние до верха конструкции коллектора не менее 0,2м

е) Размещение электрических кабелей более высокого напряжения следует предусматривать на отдельных полках ниже кабелей, более низкого напряжения.

ж) При размещении на одной полке кабелей разных сечений и марок, расстояние между ними не должно быть меньше одного диаметра кабеля.

Остановимся только на магистральных коллекторах и на следующих инженерных сетях:

1. Теплопроводы
2. Водопроводы
3. Электрические кабеля
4. Кабеля связи

Каждой инженерной сети и коллекторам можно сопоставить совокупную цену монтажа и строительных материалов на один метр.

Задача, которую необходимо будет решить формулируется следующим образом:

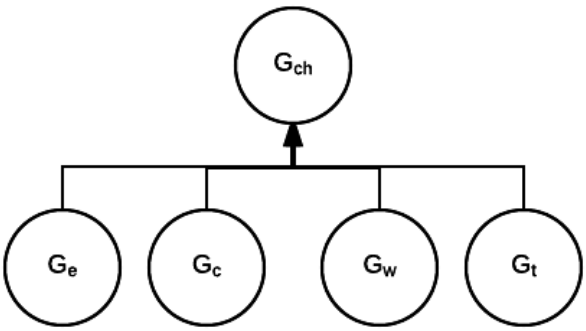


Рисунок 4.1 — Модель коллекторной сети

Найти такую структуру коллекторной сети и проложить все виды инженерных коммуникация таким образом, чтобы затраты на монтаж и на закупку строительных материалов были минимальны.

В качестве ограничений служат выше приведенные выдержки из ГОСТ-ов. Так же необходимо, что бы все точки-клиенты были запитаны от источников.

4.1.2 Математическая модель

Для начала необходимо построить граф возможных каналов прокладки коллекторных систем G_{ch}. Стоит заметить, что данный граф часто совпадает с транспортной системой города, так как в большинстве случаев коллекторы стараются прокладывать вдоль транспортных магистралей.

На следующем этапе необходимо построить графы топологий сетей. Эти графы строятся отдельно для каждой инженерной системы. Поэтому зададим следующие графы:

1. G_t - теплопроводы;
2. G_w - водопроводы;
3. G_e - электрические кабеля;
4. G_c - кабеля связи;

Построим следующую s-гиперсеть4.1:

$$S = (G_{ch}, G_t, G_w, G_e, G_c, F_{t;ch}, F_{t;ch}, F_{w;ch}, F_{e;ch}, F_{c;ch}$$

Где: $F_{i;j}$ - отображение графа G_i в граф G_j.

Необходимо найти все отображения $F_{t;ch}, F_{t;ch}, F_{w;ch}, F_{e;ch}, F_{c;ch}$.

Целевая функция:

$$F = \sum_{e \in E_{ch}} ind_{ch}(e) cost(e) + \sum_{e \in (E_t + E_w, E_e, E_c)} cost(e)$$

4.1.3 Входные данные

Входными данными для алгоритма будет является:

1. список инциденции для графа G_{ch};
2. отображение графа G_{dist} в G_{ch};

4.1.4 Алгоритм

Как нетрудно заметить данная задача NP сложна. Поэтому в качестве алгоритма ее решения выбран генетический алгоритм.

Так как целью данной работы не ставилось нахождения наиболее оптимального алгоритма, то далее будет приведен алгоритм наиболее простой в реализации, но при этом служащий иллюстрацией работы системы в целом. Поэтому будем искать оптимальное вложение сетей не всех вместе, а поочередно. То есть сначала найдем оптимальные маршруты для теплопроводов. потому для водоснабжения и так далее, но будем оставлять за собой право увеличивать размеры коллектора, если достигнем предела вместимости.

Следовательно, алгоритм решения задачи можно представить, как композицию четырёх последовательно выполняемых генетических алгоритмов. Так же стоит заметить, что на каждом следующем шаге изменяются стоимости прокладки коллекторов. То есть, если при нахождении оптимального вложения графа G_t в граф G_{ch}, алгоритм построил маршрут для некоторого ребра $e_t \in E_{G_t}$ через ребро $e_{ch} \in G_{ch}$, то при нахождении вложения графа G_w в граф G_{ch} ребро e_{ch} будет иметь вес равный нулю. Благодаря этому на каждом следующем шаге, алгоритм будет строить сети в уже существующих коллекторах, если это возможно.

Топология всех сетей G_t, G_w, G_e, G_c будет иметь структуру типа ”звезда”. соответственно, чтобы найти вложения каждого графа в граф возможных коллекторных трасс, необходимо найти дерево, которое соединяет источник ресурса со всеми потребителями. Далее, удалив все ненужные вершины и ребра из дерева получим коллекторную сеть необходимую для прокладки инженерной сети.

Легко заметить, что задача отображения одного из графов G_t, G_w, G_e, G_c в G_{ch} сводится к отысканию минимального остовного дерева в подграфе K графа G_{ch}. Задача поиска минимального остовного дерева с ограничениями в графе является NP сложной и хорошо изученной. В частности, в работе [9] предложен генетический алгоритм решения данной задачи на примере структурированной кабельной системы.

В данной работе будет использована модификация данного алгоритма. Разница будет заключаться в способе кодирования деревьев.

Генетический алгоритм - это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе.

Этапы генетического алгоритма

1. задание целевой функции для особей популяции;
2. сгенерировать начальную популяцию;
3. цикл пока не выполнен критерий останова:
    а) размножение (кроссинговер);
    б) мутирование;
    в) вычислить значение целевой функции для всех особей;
    г) селекция (формирование нового поколения);

Перейдем к рассмотрению определенных стадий алгоритма.

Кодирование хромосом

В качестве алгоритма кодирования остовных деревьев будем использовать код Прюфера. Использование кода прюфера в генетический алгоритмах не является новшеством. В частности, в статье [33] было продемонстрировано примене-

ние данного подхода для отыскания остовного дерева с ограничением на степени вершин.

Код Прюфера был предложен впервые Хайнцем Прюфером при доказательстве формулы Кэли в 1918 году [34]. Код однозначно сопоставляет произвольному конечному дереву на n вершинах последовательность длинны $n-2$ чисел от 1 до n с возможными повторениями.

Процедура кодирования:

Пусть задано дерево T с пронумерованными вершинами $\{1, 2, 3, ..., n\}$. Построение кода Прюфера дерева T ведётся путем последовательного удаления вершин из дерева, пока не останутся только две вершины. При этом каждый раз выбирается концевая вершина с наименьшим номером и в код записывается номер единственной вершины, с которой она соединена. В результате получаем последовательность длинны $n-2$ из чисел $\{1, 2, 3, ..., n\}$ с повторениями.

Процедура декодирования:

Пусть задана некоторая последовательность $s = (s_1, s_2, ...s_{n-1})$. Зададим список вершин $v = 1, 2, 3, ..., n$. Выберем первый номер, который не встречается в коде s - i_1. Добавим в дерево ребро (i_1, s_1). После этого удалим s_1 из s и $i_1 v$. Повторим процесс до тек пор, пока s не пусто.

По сравнению с кодированием битовой строкой, где каждый бит отвечает за присутствие или присутствие ребра в дереве, код Прюфера имеет следующие достоинства:

1. код Прюфера имеет меньшую длину;
2. при случайной мутации или кроссинговере невозможно получить не дерево;

Целевая функция

Стоит заметить, что если ниш исходный граф не является полным, то при мутации и кроссинговере есть вероятность получения хромосомы, которая не будет является допустимым решением задача, так как может содержать ребро между вершинами, которого нет в исходном графе. Так же, в задаче присутствует ряд ограничений, из-за которых случайное дерево не может быть допустимым

решением. Для разрешения данной ситуации представим целевую функцию как векторное произведение функций f и g, где:

1. f - функция по дереву возвращает стоимость монтажа коммуникаций;
2. g - возвращает 1 если дерево является допустимым решением, возвращает 0 если не является;

Функция f убирает из остовного дерева все ребра и вершины, не лежащие на маршрутах от источника ресурса до клиента и рассчитывает стоимость возведения коллекторной сети.

Оператор селекции

Пусть заданы две хромосомы. Лучшую из них следует выбирать следующим образом:

1. Если значения функции g обоих хромосом равно нулю или единицы то выберем хромосому с минимальным значением функции f
2. Если значение функции g одной из хромосомы равно единице а другой нулю то выберем хромосому со значением функции g равной единице, несмотря на значения функции f

Таким заданием оператора селекции мы определим преимущество валидных хромосом над невалидными.

Оператор кроссинговера

Определим оператор кроссинговера:

Пусть заданы две хромосомы $a = (a_1, a_2, a_3, ...a_{n-2})$, $b = (b_1, b_2, b_3, ...b_{n-2})$ родителей.

Выполним следящие действия:

1. Выберим две позиции i,$j < n - 2i < j$
2. Получим хромосомы потомков следующего вида: $a = (a_1, ...a_i, b_{i+1}, ...b_j, a_{j+1},$ $b = (b_1, ...b_i, a_{i+1}, ...a_j, b_{j+1}, ...b_{n-2})$

Результаты

Результатом выполнения генетического алгоритма будет являться остновное дерево графа. Но нам необходимо найти только маршруты соединяющие вершины графа инженерной сети. Для этого достаточно просто убрать не нужные вершины и ребра из дерева. Единственность каждого маршрута гарантируется по определению дерева.

На основе полученного дерева возможно построить отображение графа топологии инженерной сети в граф возможных коллекторных трасс (G_{ch}).

4.2 Минимизация расстояний до остановок общественного транспорта

4.2.1 Постановка задача

Необходимо проложить маршруты общественного транспорта таким образом, чтобы сумма расстояний от жилых домов до остановок общественного транспорта была минимальной. При этом вводятся ограничение на длину и количество маршрутов.

Данная задача является попыткой рассмотреть частный случай задачи оптимизации общественного транспорта в мегаполисе. Что в настоящие время актуально в больших городах. В общем случае необходимо учитывать не только точки от куда люди двигаются до остановок, но также и маршруты их передвижения, то есть, начальную и конечную точку.

4.2.2 Математическая модель

В качестве математической модели выступает S-гиперсеть $S = (H, G_{routes}, G_b, F_{G_{routes};H}, F_{G_b;G_{routes}})$ 4.2, где:

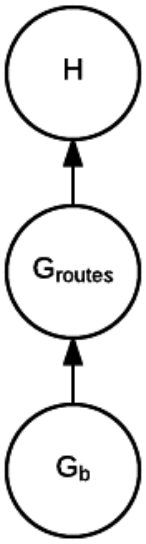


Рисунок 4.2 — Модель транспортной сетиs

1. H - граф дорожно-транспортной сети с узлами на перекрестках и остановках;
2. G_{routes} - граф маршрутов;
3. G_b - граф точек расположения скопления пассажиров (содержит только вершины);

Для решения задачи необходимо и достаточно найти отображения $FG_{routes}; H, FG_b; G_{routes}$ и построить граф G_{routes} таки образом, что бы значение функции $f(S) = \sum_{v \in V^{G_b}} dist(v, F_{G_{routes};H} F_{G_b;G_{routes}}(v))$ было минимальным.

Органичения:

1. N - максимальное количество маршрутов;
2. M - максимальная длина маршрута;

4.2.3 Входные данные

От системы нам понадобится:

1. список инциденции графа H;
2. Список вершин графа G_b;

4.2.4 Алгоритм

Для начала примем тот факт, что водители общественного транспорта двигаются от одной остановки до следующей по кратчайшему маршруту. На основании этого факта, построим полный граф $H^* = (V_{H^*}, E_{H^*})$, где $V_{H^*} \subseteq V_H$ - множество остановок, а любое ребро $e(v_1, v_2) \in E_{H^*}$ имеет вес, равный длине кратчайшего маршрута между вершинам v_1, v_2 в графе H. Данную операцию можно проделать любым известным алгоритмом для поиска кратчайшего расстояния между парами вершин в графе

Легко решить данную задачу перебором. Для этого нам нужно перебрать все возможные совокупности маршрутов в графе H^*, каждый из которых имеет длину не более M;

Каждая совокупность маршрутов определяет вариант графа G_routes. Для каждого варианта мы должны построить отображение $F_{G_b;G_routes}$. Очевидно, что имеет смысл рассматривать только такие отображения, где вершина графа G_b отображается в ближайшую вершину графа G_b

Итого, неформально алгоритм можно записать так:

1. Строим граф H^* из графа H.
2. Находим множество R всех маршрутов в графе H^*.
3. Для каждого подмножества множества R вычисляем значение целевой функции.
4. Выбираем подмножества множества R с минимальным значением целевой функции.

Вычисление целевой функции сводится к вычислению следующего выражения: $\sum_{v \in V^{G_b}} dist(v, T)$, где T - это объединение всех вершин выбранных маршрутов.

Рассмотрим для начала задачу маршрутизации транспорта. Классическая задача маршрутизации транспорта формулируется следующем образом:

Пусть заданы два множества V - клиенты, D - депо. Пусть задан граф $G = (V \cup D, E)$ и каждое ребро из E имеет вес (стоимость проезда). Необходимо построить m маршрутов транспортных средств минимальной суммарной стоимости, которые начинаются и заканчиваются в одном из депо, заданных множеством D, и каждая вершина из V должна быть включена в маршрут одного и только од-

ного транспортного средства. Количество маршрутов m может быть заданным заранее или вычисляться в ходе работы.

Данная задача была впервые озвучена в [35]. И на сегодняшней момент достаточно хорошо изучена. В работах [35—45] предложен ряд алгоритмов, как точных, так и эвристических, в частности алгоритмы муравьиной колонии, генетические алгоритмы и даже нейронные сети, для решения этой задачи. Так как задача принадлежит классу NP, то имеет смысл рассматривать эвристические алгоритмы.

Ключевым отличием задачи маршрутизации транспорта от рассматриваемой задачи прокладки маршрутов общественного транспорта , является то что в первой необходимо покрыть все множество клиентов, а во второй это не обязательно. Остановка может стоять в неудобном никому месте и проводить туда маршрут будет не нужно. Кроме того, стоит заметить, что разные маршруты общественного транспорта могут посещать одну и туже остановку, тогда как в задачах маршрутизации транспорта нет необходимости доставлять что-то клиенту больше одного раза. Но, несмотря на это, эти две задачи весьма похожи. И методы, применяемы для решения задачи маршрутизации транспорта, могут быть использованы и поставленной задаче.

Для построения решения был выбран генетический алгоритм, так как этот тип алгоритмов наиболее простой в реализации и дает неплохие результаты, в частности и для решения маршрутизации транспорта.

Кодирование хромосом

Хромосома в данном случае представляет из себя последовательность чисел из множества $\{0, 1, ..., n\}$ с возможными повторениями, где n - количество вершины в графе. Соответственно каждый ген интерпретируется как номер вершины. 0 служит разделителем маршрутов.

Пример:

Пусть в полном графе из трех вершин заданы два маршрута - $12 \to 3, 2 \to 1$. Тогда хромосома будет выглядеть следующим образом - 1,2,3,0,2,1

Целевая функция

По аналогии с предыдущей задачей, введем целевую функцию как векторное произведение функций f, g. Где функция g - проверяет на условия на длину и кол-во маршрутов, а f вычисляет целевую функцию.

Оператор селекции

Оператор селекции полностью идентичен оператору из предыдущей главы.

Оператор мутации

Операция мутации будет происходить в два этапа:

1. выбор случайной позиции;
2. замена на случайно сгенерированное число;

При этом, в зависимости от выбранной позиции имеем разные законы распределения для выбора нового значения гена.

Пусть случайно выбран ген i. Тогда если ген $i-1, i-2, i+2$ или $i+1$ имеют значения равные 0 то для выбора следующего значения имеем равномерный закон распределения для чисел 1,2,3, ..., n. То есть, в ходе мутации мы не можем получить хромосому с двумя подряд нулями или с путем длинны равной единицы. Что увеличивает вероятность генерации валидных хромосом.

Если же значения генов $i-1, i-2, i+1, i+2$ не равны нулю, то имеем следующий закон распределения:

1. $p_0 = \frac{1}{(n+1)} * \varepsilon$, где $\varepsilon \leqslant 1$ заданное в начале алгоритма;
2. $\forall i > 0, p_i = \frac{1}{(n)-\frac{\varepsilon}{(n+1)}}$;

Оператор кроссинговера

Как не трудно заметить, хромосомы имеют разную длину. Поэтому необходимо по-особенному задать оператор кроссинговера.

Пусть есть две хромосомы $A = (a_1, a_2, ...a_m)$ и $B = (b_1, b_2, ..., b_k)$, $m > k$. Тогда выберем хромосому с минимальной длинной - B. Сгенерируем случайное число $i < k$ b получим следующие хромосомы:

$$(b_1, b_2, ...b_i, a_{i+1}, ..., a_m)$$

$$(a_1, a_2, ...a_i, b_{i+1}, ..., b_k)$$

Результаты

Результатом работы алгоритма будет являться строка, представляющая маршруты общественного транспорта. Декодировав строку получим массив маршрутов и на их основании построим отображение $FG_{routes}; H$.

Далее для каждой вершины $v \in G_b$. Найдем ближайшую, в геометрическом смысле вершину $u \in G_{routes}$. И в заключении построим отображение $FG_{routes}; H$, которое отображает вершины v в ближайшие вершины u.

4.2.5 Выводы

В данной главе были продемонстрированы задачи, которые были решены автором при помощи построенной системы. Данные задачи являются актуальными задачами при планировании и оптимизации городских инфраструктур.

На этих задачах особенно видна необходимость в системе целостного хранения денных. Например, построенная структура инженерных систем не является собственно конечным решением. Инженерные инфраструктуры подвержены изменениям из вне. Кроме того, в этих задачах очень важно дальнейшее хранение и документирование решений.

Стоит так же заметить, что хоть и задачи формулировались в терминах гиперсетей, алгоритмы не используют полные данные, а выбирают из базы только нужные. Что значительно облегчает работу.

Заключение

Основные результаты работы заключаются в следующем.

1. В обзоре приведен ряд задач, решаемых с помощью теории гиперсетей. Рассмотрены основные результаты в этой области. Показана актуальность использования гиперсетевых технологий и широкое поле ее применения. Рассмотрены программные реализации решений на основе гиперсетей. Приведены существующие программные реализации алгоритмов и методов хранения гиперсетевых объектов.
2. В силу того, что понятие гиперсети тесно связано с понятием графа, во второй главе рассмотрены существующие решения для организации хранения графовых данных в разного вида системах управления базами данных. Отмечены плюсы и минусы каждого подхода.
3. Предложена схема хранения данных гиперсети в документно-ориентированной базе данных MongoDB. Формализовано понятия документа и операций над документами. Построены алгоритмы получения различных предоставлений гиперсетей и графов. Доказана их корректность.
4. Предложенная схема реализована в виде веб сервиса на функциональном языке программирования elixir. Построена масштабируемая архитектура сервиса и все компоненты развернуты на облачных виртуальных машинах.
5. Численные эксперименты показывают состоятельность решения даже при больших объёмах данных. Кроме того, эксперименты проводились, в том числе, и над кластером из нескольких экземпляров базы данных. В ходе чего, была показана возможность горизонтального масштабирования системы при увеличении объемов данных
6. Были построены три языка взаимодействия с системой. Два из них (JSONAPI, GraphQL) являются общепринятыми стандартами построения программного интерфейса приложений. Третий является расширением интерпретируемого, объектноориентированного языка общего назначения Ruby.
7. Были построены алгоритмы решения задачи оптимизации коллекторной сети и оптимизации маршрутов общественного транспорта. Программ-

ная реализация алгоритмов взаимодействует с системой, получая оттуда исходные данные и загружая результаты вычислений.

Список литературы

1. *Попков В. К.* Математичиские модели связности. — Новосибирск : ИВМиМГ СО РАН, 2006. — 490 с.
2. *Оре О.* Графы и их применение. — Рипол Классик, 2002.
3. *Golumbic M. C.*, *Kaplan H.*, *Shamir R.* Graph sandwich problems // Journal of Algorithms. — 1995. — Т. 19, № 3. — С. 449—473.
4. *Poulovassilis A.*, *Levene M.* A nested-graph model for the representation and manipulation of complex objects // ACM Transactions on Information Systems (TOIS). — 1994. — Т. 12, № 1. — С. 35—68.
5. *Levene M.*, *Loizou G.* A graph-based data model and its ramifications // IEEE Transactions on Knowledge and Data Engineering. — 1995. — Т. 7, № 5. — С. 809—823.
6. *Попков В. К.* Гиперсети и структурные модели сложных систем // Математические и имитационные модели сложных систем. — Новосибирск, 1981. — С. 26—48.
7. *Попков В. К.* Применение теории S-гиперсетей для моделирования систем сетевой структуры // Проблемы информатики. — Новосибирск, 2010. — № 4. — С. 17—40.
8. *Галямов В. А.* О задаче построения структурированных кабельных систем // Новосиб. Ун-та. — Новосибирск, 2005. — С. 36.
9. *Галямов В. А.*, *Соловей С. С.* Генетический алгоритм задачи размещения структурированной кабельной системы здания // Научное обозрение. - М.:Издательство ”НАУКА”.. — Новосибирск, 2005. — № 5. — С. 23—25.
10. *Галямов В. А.*, *С. С. С.* Генетический алгоритм задачи размещения структурированной кабельной системы здания // Вестник НГУ, Сер. Информационные технологии. — Новосибирск, 2005. — С. 4.
11. ГОСТ Р 53246-2008 Информационные технологии. Системы кабельные структурированные. Проектирование основных узлов системы. Общие требования. — Москва : Стандартинформ, 2006. — 71 с.

12. *Галямов В. А.* Исследование и разработка моделей и методов оптимизации структур телекоммуникационных систем: дис. ... канд. физ. мат. наук. — Новосибирсе, 2006. — 164 с.

13. *Конин М. В.*, *Лепнер Э. Ю.*, *Попков Г. В.* Применение S-гиперсетей для автоматизированного проектирования инженерной инфраструктуры предприятия // Проблемы информатики. — 2013. — Т. 3. — С. 65—72.

14. *Попков В. К.* Математические модели живучести сетей связи. — Новосибирск : ИВМиМГ СО РАН, 1990. — 490 с.

15. *Попков В.*, *Блукке В. П.*, *Дворкин А. Б.* Модели анализа устойчивости и живучести информационных сетей // Проблемы информатики. — Новосибирск, 2009. — № 4. — С. 69—78.

16. *Величко В. В.*, *Попков Г. В.*, *Попков В. К.* Модели и методы повышения живучести современных систем связи. — Москва : Горячая линия–Телеком, 2014. — 270 с.

17. *Соколова О. Д.* Графовые модели для задач функционирования современных сетей передачи данных // Проблемы информатики. — 2014. — № 4. — С. 61—68.

18. *Величко В. В.*, *Юргенсон А. Н.* Модели структурной надежности в мобильных сетях передачи данных // ISSN 0013-57771 Электросвяз. — Новосибирск, 2004. — № 3. — С. 36—37.

19. *Величко В. В.*, *Юргенсон А. Н.* Задача анализа структурной надежности мобильных сетей связи // Труды ИВМиМГ серия Информатика. — Новосибирск, 2005. — С. 58—65.

20. *Юргенсон А. Н.* Вычисление показателей живучести информационных сетей на модели нестационарной гиперсети: дис. ... канд. физ. мат. наук. — Новосибирсе, 2006. — 97 с.

21. On Optimal placement of the monitoring devices on channels of communication network / A. Rodionov [и др.] // Computational Science and Its Applications–ICCSA 2009. — 2009. — С. 465—478.

22. *Соколова О. Д.*, *Юргенсон А. Н.* ЗАДАЧА ОПТИМАЛЬНОГО РАЗМЕЩЕНИЯ УСТРОЙСТВ АНАЛИЗА ИНФОРМАЦИОННЫХ ПОТОКОВ В СЕТЯХ // Проблемы информатики. — 2010. — № 2. — С. 33—41.

23. *Попков В. К.*, *Токтошов Г. Ы.* Гиперсетевая технология оптимизации инженерных сетей в горной или пересеченной местности // Вестник Бурятского государственного университета. — 2010. — № 9.

24. *Токтошов Г. Ы.* Гиперсети в моделировании и оптимизации совмещенной прокладки подземных инженерных коммуникаций // Проблемы информатики. — 2014. — № 1. — С. 15—23.

25. *Попков В.* О моделировании городских транспортных систем гиперсетями // Автомат. и телемех. — 2011. — Т. 6. — С. 179—189.

26. *Попков Г. В.*, *Попков В. К.* Об одном способе размещения пунктов обслуживания в транспортных сетях // Проблемы информатики. — Новосибирск, 2012. — № 3. — С. 33—38.

27. *Казаков А. Л.*, *Петров М. Б.*, *Маслов А. М.* Исследование региональной транспортной системы с использованием гиперсетей // Транспорт Урала. — 2013. — № 4. — С. 22—28.

28. *Codd E. F.* A Relational Model of Data for Large Shared Data Banks // Communications of the ACM. — New York, 1970. — Т. 13, № 3. — С. 111—116.

29. *Д. С.* SQL для профессионалов. — Москва : Лори, 2009. — 464 с.

30. *Alberton L.* Graphs in the database: Sql meets social networks // Techportal, Sep. — 2009. — Т. 7. — С. 1—11.

31. *Rodriguez M. A.* Graph databases and the future of large-scale knowledge management // Invited Lecture for Risk Analysis Symposium, Risk Analysis of Complex Systems for National Security Applications, Decision Applications Division and the Center for Nonlinear Studies, Santa Fe, New Mexico. — 2009.

32. *Lin J.*, *Dyer C.* Data-Intensive Text Processing with MapReduce. — 2010.

33. *Hanr L.*, *Wang Y.* A novel genetic algorithm for degree-constrained minimum spanning tree problem // Int J Comput Sci Netw Secur. — 2006. — Т. 6, 7A. — С. 50—57.

34. *Hirzebruch F.* Neue topologische Methoden in der algebraischen Geometrie. Т. 9. — Springer-Verlag, 2013.

35. *Christofides N.* The vehicle routing problem // Revue française d'automatique, informatique, recherche opérationnelle. Recherche opérationnelle. — 1976. — Т. 10, № V1. — С. 55—70.

36. *Maniezzo A.* Distributed optimization by ant colonies // Toward a practice of autonomous systems: proceedings of the First European Conference on Artificial Life. — Mit Press. 1992. — С. 134.

37. *Ball M.*, *Levy L.* A Vehicle Routing Improvement Algorithm: Comparison of a 'greedy' and a Matching Implementation for Inventory Routing. — Montréal: Université de Montréal, Centre de recherche sur les transports, 1984.

38. *Gendreau M.*, *Hertz A.*, *Laporte G.* A tabu search heuristic for the vehicle routing problem // Management science. — 1994. — Т. 40, № 10. — С. 1276—1290.

39. *H G. el* Solving routing problems by a self-organizing map. — 1991.

40. *Gillett B.*, *Miller L.* A heuristic algorithm for the vehicle-dispatch problem // Operations research. — 1974. — Т. 22, № 2. — С. 340—349.

41. *Golden B.*, *Magnanti T.*, *Nguyen H.* Implementing vehicle routing algorithms // Networks. — 1977. — Т. 7, № 2. — С. 113—148.

42. *Jeon G.*, *Leep H.*, *Shim J.* A vehicle routing problem solved by using a hybrid genetic algorithm // Computers & Industrial Engineering. — 2007. — Т. 53, № 4. — С. 680—692.

43. *Laporte G.*, *Semet F.* Classical heuristics for the vehicle routing problem // Cahiers du GERAD. — 1999.

44. *Schmitt L.* An evaluation of a genetic algorithmic approach to the vehicle routing problem: тех. отч. / Working paper, Department of Information Technology Management, Christian Brothers University, Memphis, TN. — 1995.

45. *Xu J.*, *Kelly J.* A network flow-based tabu search heuristic for the vehicle routing problem // Transportation Science. — 1996. — Т. 30, № 4. — С. 379—393.

Список рисунков

Список таблиц

Приложение А

GraphQL схема

Листинг А.1 GraphQL схема

```
type Hypernet {
  id: ID
  name: String
  node: [Node]
  graphs: [Graph]
  mappings: [Mapping]
}

type Graph {
  id: ID
  name: String
  nodes: [Node]
  edges: [Edge]
  hypernet: Hypernet
  mappings: [Mapping]
}

type Node {
  id: ID
  name: String
  data: Map
  hypernet: Hypernet
  graphs: [Graph]
  edges: [Edge]
}

type Edge {
  id: ID
  name: String
  data: Map
  graph: Graph
  nodes: [Node]
}

type Mapping {
  id: ID
```

```
  hypernet: Hypernet
  source_graph: Graph
  target_graph: Graph
  node_mapping: [NodeMapping]
  edge_mapping: [EdgeMapping]
}

type NodeMapping {
  id: ID
  source_node: Node
  target_node: Node
  hypernet: Hypernet
  mapping: Mapping
  source_graph: Graph
  target_graph: Graph
}

type NodeMapping {
  id: ID
  source_edge: edge
  target_edges: [edges]
  hypernet: [Hypernet]
  mapping: Mapping
  source_graph: Graph
  target_graph: Graph
}

type Paginator {
  page: Integer
  per_age: Integer
}

type Query {
  hypernets(paginator: Paginator): [Hypernet]
  hypernet(id: ID!): Hypernet

  nodes(hypernet_id: ID!, paginator: Paginator): [Node]
  node(id: ID!): Node

  grahps(hypernet_id: ID!, paginator: Paginator): [Graph]
  graph(id: ID!): Graph
```