Супер крутая работа Олежки!!1

«Тепловая карта предложений жилья»

**Оглавление**

**Введение**

**Введение**

В жизни сегодняшнего человека существует множество занятий, когда ему приходится иметь дело с большим объемом данных. Негласным стандартом для такой работы стало использование различный фильтров. Выбор жилья не стал исключением. Но вместо того что бы с помощью фильтров исключать предложения, но можно изменить способ их представления на более удобный и легкий в восприятие, способный отображать намного больший объем информации на экране и благодаря этому пользователь сможет принимать решение на основание намного большего числа факторов, что сделает решение более взвешенным. Кроме информации о самой квартире можно отобразить информацию об её окружение. О транспортной доступности и об инфраструктуре. Выбор методов отображение и их реализация будет описана ниже.

**Существующие альтернативы**

**...**

**Основные принципы отображения**

1. Предложения:

Для предложений я собираюсь использовать вместо классических маркеров заливку зданий определенным цветом. Цвет будет определятся на основание отличия цены, от заданой пользователем.



Рисунок 1.

Для этого отлично подойдет HSB градиент (Рисунок 1), ограниченный снизу на синем цвете. То есть градиент с параметром hue(оттенок) от 0 до 240. Также он должен быть симметричен, то есть где P – цена.

Благодаря интуитивной ассоциации красного цвета как с чем-то выше норма, а синего наоборот с чем-то ниже нормы, пользователю будет легко привыкнуть к значениям цветов. Но при построении градиента на всем диапазоне цен, представленных на карте может возникнуть проблема. Она заключается в том, что градиент может стать слишком широким, из-за чего он перестанет отображать разницу в ценах.

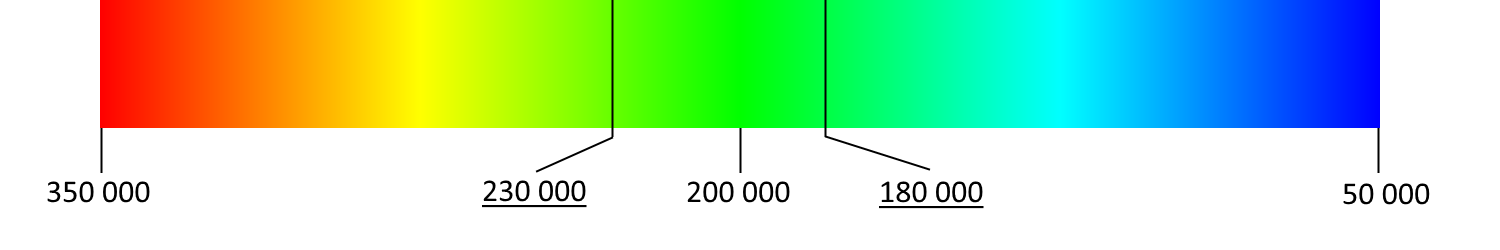


Рисунок 2.

На рисунке 2 показан градиент (все цены в рублях за метр квадратный), отражающий диапазон цен от 50000 до 350000, с указанной ценой в 200000. Также отмечены цены 230000 и 180000. Видно что не смотря на отличие цены на 30000 и 20000 соответственно, их цвет едва различимы. Для решения этой проблемы я решил использовать настраиваемый диапазон цен. Пользователь будет выбирать не только целевую цену, но и диапазон цен.

Используя полученный градиент, я буду использовать для отрисовки домов, причем отрисовывать их я буду на прозрачном фоне, что бы их можно было наложить на карту. Пример представлен на Рисунке 3.

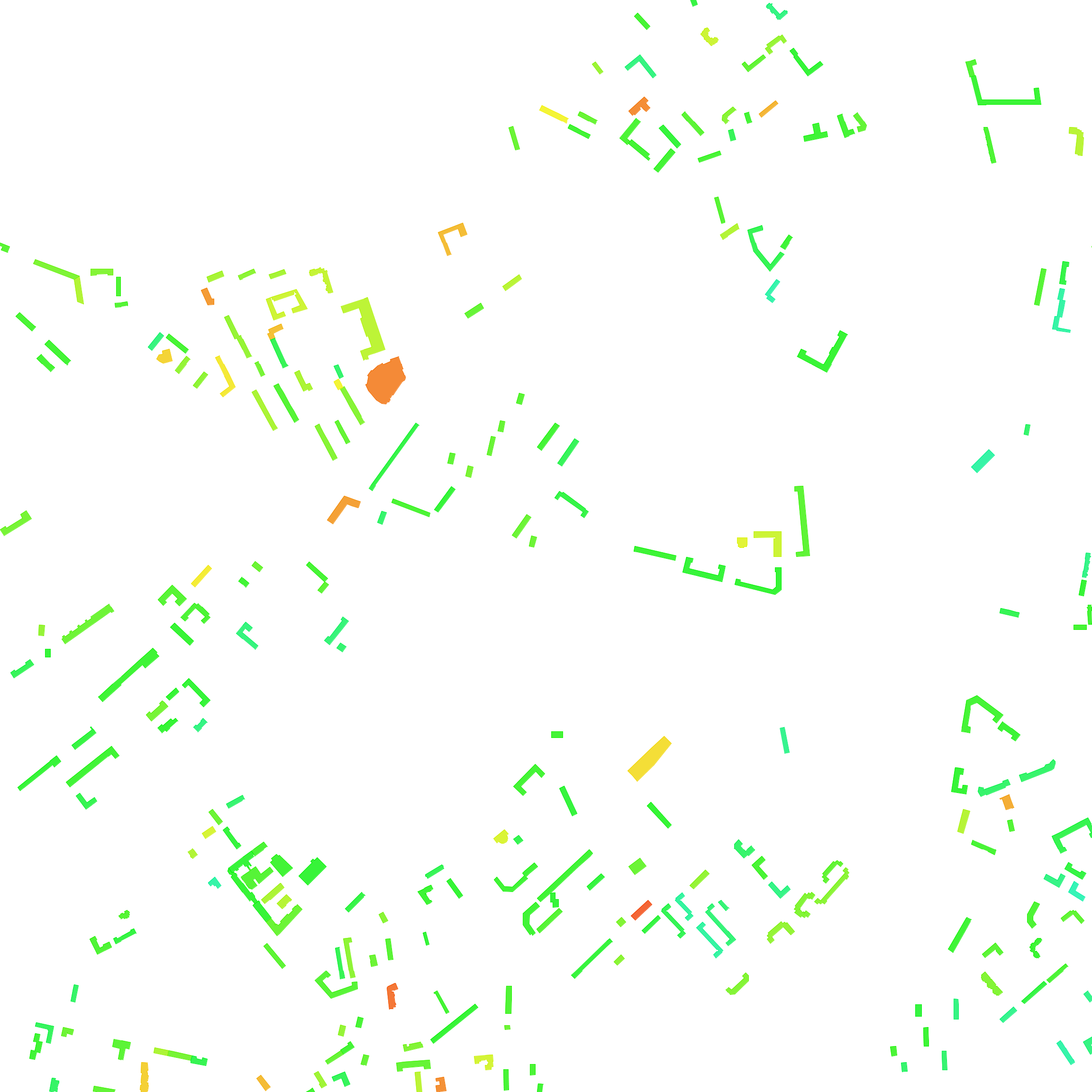


Рисунок 3.

1. Транспортная доступность:

Для отображения транспортной доступности я буду раскрашивать дороги в цвет зависящий от того, за сколько времени можно до той или иной её части. Для этого имея дорожный граф и начальную точку, я найду значение для каждой вершины, которое будет являться минимально возможным временем, за которое можно добраться из начальной вершины в данную. Дальше на основание этих значений и максимального времени, граф раскрашивается с помощью градиента приведенного на рисунке 4.



Рисунок 4.

Соответственно не доступные части графа будут окрашены в темно-красный цвет(находится за отметкой макс. время). Также кроме автомобильного режима, использующего обычные автомобильные дороги, я добавлю пешеходный режим, учитывающий пешеходные улицы и общественный транспорт. Этот граф также должен быть отрисован на прозрачном фоне, причем и его самого я сделаю полупрозрачным, что бы после наложения на карту можно было видеть названия улиц.

1. Доступность инфраструктуры.

При выборе того или иного дома я собираюсь показывать список объектов инфраструктуры поблизости. Магазины, кафе, гос. учреждения.

**Реализация**

1. Структура проекта и инструменты:

Проект состоит из двух крупных частей. Сервер, занимающийся обработкой информации о жилье и рендерящий информацию о нем и отдающий её по запросу. Клиент, отображающий карту с отрисованной на ней информацией и дающий интерфейс пользователю, с помощью которого он может формировать запросы на сервер.

Сервер же состоит из двух частей, сервера раздающего статический контент, тайлы карты, и раздающий динамический контент, тайлы с информацией о жилье/транспорте.

В качестве сервера для статического контента был выбран nginx, за свою производительность и простоту настройки.

Сервер для динамического контента будет написан на java, так как java является классическим языком программирования для написания серверов. Она заслужила такое звание за свою безопасность, производительность и огромное количество библиотек, созданных для неё. Раздача уже готового контента будет сделана с использованием библиотеки netty.

Клиент же будет написан на связке Python и PyQt/wxPython. Эта связка была выбрана из-за того, что я уже был знаком с разработкой десктопных приложений на Python.

1. Сервер:

Все данные карты, цен и методы работы с ними я решил объединить в классе PropertyMap. Соответственно при обработке запросов к серверу, обработчик обращается только к объекту PropertyMap, а все обращения к внутренним объектам производятся уже самим PropertyMap’ом. Такое разделение позволит потом при необходимости держать несколько карт, в одной jvm, с минимальными изменениями.

* 1. Загрузка данных:

Для начала я начал искать источник геоданных, к сожелению открытых и достаточно полных источников очень мало. Единственный, который я нашел – это openstreetmap.org. Их данные были достаточно полны и совершенно открыты, за что выбор и пал на них. После необходимо было написать парсер для этих данных. Они распостронялись в формате osm, который по сути является xml. Структура этого формата очень проста. Есть три основных элемента Node, Way и Relation.

Node – это обычная точка, она в качестве своих xml аттрибутов имеет id, географические координаты и информацию о пользователе что её добавил/изменил. Также в него вложены элементы типа tag, которые содержат в себе информацию о точке в формате ключ-значение. Например автобусная остановка описывается как public\_transport-platform.

Way – это тип описывающий полигоны и линии на карте. Он содержит в себе те же атрибуты что и Node. В него вложены элементы nd и tag, элементы tag также содержат информацию о Way, nd же содержат в себе id Node. Соответственно соединив все данные Node мы получим то, как выглядит Way.

Relation – это элемент группирующий другие элементы. Node, Way и Relation. Обычно эти элементы являются маршрутами общественного транспорта или какими либо сетями.

Для работы над этими данными надо создать классы представляющие гео данные. Они показаны в Приложение 1.

Выбор именно такой схемы классов будет объяснен ниже.

Так как osm файлы обладают внушительным размером(карта Москвы занимает ~1.5GB), то обычные xml парсеры не подходят. Для парсинга больших xml файлов используются stream parsers(потоковые парсеры), которые не строят DOM дерево в памяти, а всего лишь сообщают о типе элемента и его атрибутах. С такими парсерами вся забота о правильной иерархии элементов ложится на плечи разработчика. Поскольку иерархия элементов у osm формата очень проста, то этот парсер становится отличным вариантом. Java включает в себя StAX парсер.

Загрузка происходит по схеме, представленной в Приложение 2. Далее карта – это фрагмент данных osm прямоугольником. В самом начале читаются границы карты и записываются в соответствующие поля, это необходимо для того что при дальнейшей конвертации географических координат в координаты на плоскости, можно было получать не абсолютные их значения, а относительно верхнего левого угла карты(точки с минимальной долготой и максимальной широтой). Сделано это для простоты передачи координат от клиента к серверу.

После в методе load() происходит чтение файла, и создание Node, Way и Relation, которые сохраняются внутри загрузчика, а потом забираются из него с помощью методов getNodes(), getWays(), getRelations(), getSimpleNodes(). Благодарю тому что все реализовано через интерфейс MapLoader можно в дальнейшем написать новый загрузчик, реализующий этот интерфейс и читающий данные из другого формата, тем самым получив поддержку новоги формата. Также этот метод создаёт дорожный граф, подробнее об этом в разделе о дорожном графе.

* 1. Квадродерево:

Для отрисовки домов, их выбора, поиска близкой к ним инфраструктуры и составления дорожного графа мне необходим быстрый поиск по данным карты. Для этого существует множество методов организации данных на плоскости. Из них я выбрал квадродерево(Quadtree), за его простоту в реализации.

Реализация квадродерева находится в классе QuadTreeNode, а методы для поиска в нем и его корень в классе QuadTree. Оно может в себе содержать как точки, так и полигоны с линиями. У меня оно не сбалансированное.

В квадродереве, у каждого узла может быть по 4 потомка, которые делят своего родителя на 4 равные части. Поиск необходимого узла происходит за время O() (где n количество элементов в дереве, при условие что дерево сбалансированное), но так как O() пренебрежительно мало, то сложность поиска в лучшем случае O(), в худшем O() (где - минимальное кол-во элементов в узле, а - максимальное кол-во элементов). Каждый узел при создание считается конечным.

Добавление точки в квадродереве очень тривиально. Когда с линиями и полигонами уже сложнее, так как необходимо их обрезать, что бы они не выходили за границы квадрата. Для обрезки линий был сделан самописный алгоритм, реализация которого в методе addRoad() класса QuadTreeNode. Для обрезки же полигонов я реализовал алгоритм Уайлера-Атертона (Weiler–Atherton). Его реализация находится в методе addPoly\_double(). Тут при разработке мною была совершена ошибка. Из-за того, что MapPoint хранит в себе только целочисленные координаты, то и все вычисления точек пересечения округлялись до целых чисел. Но из-за этого появлялись ошибки округления и алгоритм работал некорректно. Я попытался решить эту проблему с помощью класса SuperSampledMapPoint, который умножал координаты точек на некоторую константу, тем самым увеличивал «разрешение» и уменьшал вероятность ошибки округления, но поняв, что я делаю по сути то же самое что делают числа с плавающей запятой, я принял решения переписать это алгоритм с использованием DMapPoint, который переводил координаты точек в double, которые я после выполнения алгоритмы преобразовывал обратно к MapPoint с помощью округления. А старую реализацию метода под названием addPoly() оставил как напоминание о своей ошибке.

Так как дерево несбалансированное, то узлы делятся после того, как количество элементов в нем превосходит определенный порог. После этого текущей узел делится на 4 новых узла, которые назначаются потомками текущего. И все элементы из текущего перемещаются в его потомков, согласно алгоритму добавления элементов в дерево. При этом текущей элемент перестаёт быть конечным.

Все алгоритмы поиска в квадродереве сводятся к одной последовательности действий. 1. Поиск всех конечных узлов, которые пересекаются или содержатся в фигуре поиска или содержат фигуру поиска. 2. Обход всех найденных узлов и проверка их элементов на вхождение в фигуру поиска или вхождение фигуры поиска в них. 3. Массив элементов что входят в фигуру и будет результатом. Здесь есть пространство для оптимизаций, например, проведя простую проверку на вхождение узла в фигуру поиска, при его полном вхождение мы можем добавить все элементы узла без проверки каждого из них.

Мною реализовано несколько видом поиска в дереве. Поиск по точке, окружности и прямоугольнику. Также они отличаются тем, что они ищут: Way, Node или RoadGraphNode.

* 1. Дорожный граф:

Что бы показать транспортную доступность дома, мне необходимо знать минимальное время, за которое можно добраться до фрагментов дорог. Для расчета этого времени необходимо объединить все возможные пути передвижения в единый граф(то есть объединять не только автомобильные дороги, но и пешеходные и маршруты общественного транспорта), и далее имея подобный граф, провести расчет.

Вершинами графа являются объекты класса RoadGraphNode. Они хранят ссылки на другие вершины, с которыми они связаны и время, за которое можно до них добраться. Хранится 2 разных времени: для пешеходного режима и для автомобильного. Причем время хранится не в минутах, а в сотых долях секунд (0,01 сек). Такая малая единица времени была взята из-за того что на карте встречаются очень короткие участки дорог, которые например машиной со скоростью в 60 км/ч преодолеваются за секунды. И если взять за минимальную единицу времени минуту, то из-за округления, многие участки будут преодолеваться мгновенно и следовательно, результаты будут некорректны.

Так как экземпляров RoadGraphNode достаточно много(при парсинге карты Москвы – 1716727), то очень важно что бы они занимали минимум памяти. Это послужило причиной для отказа от динамических массивов ArrayList в пользу статических массивов java’ы, т.к. согласно библиотеки jol(Java Object Layout) размер ArrayList’a – 24 байта(зависит от конфигурации jvm), а размер обычного Array’я – 16 байт(зависит от конфигурации jvm), но поскольку ArrayList реализован как обертка к Array’ю, то в итоге его размер 40 байт. Соответственно переход от ArrayList к Array сохраняет 24 байта, а если учесть количество RoadGraphNode и количество массивов в них то сохраняется минимум 4\*24\*1716727=164805792 байт 157 Мбайт. Но т.к. в RoadGraphNode есть ещё и вложенные массивы, то реальный выигрыш больше.

Благодаря использованию Array я получил выигрыш по памяти, но из-за того что Array не может автоматически менять свой размер, код сильно усложнился. Для его упрощения мною были написаны Builder’ы для RoadGraphNode и самого дорожного графа – классы RoadGraphNodeBuilder и RoadGraphBuilder. В которых реализована вся логика создания вершин и их соединения. После добавления всех нод в RoadGraphBuilder вызывается его метод getRoadGraph(), который возвращает HashMap<Long, RoadGraphNode>, где ключ это id ноды, а значение сама нода. Все ноды в этом HashMap связаны друг с другом и готовы к использованию.

* 1. Расчет графа:

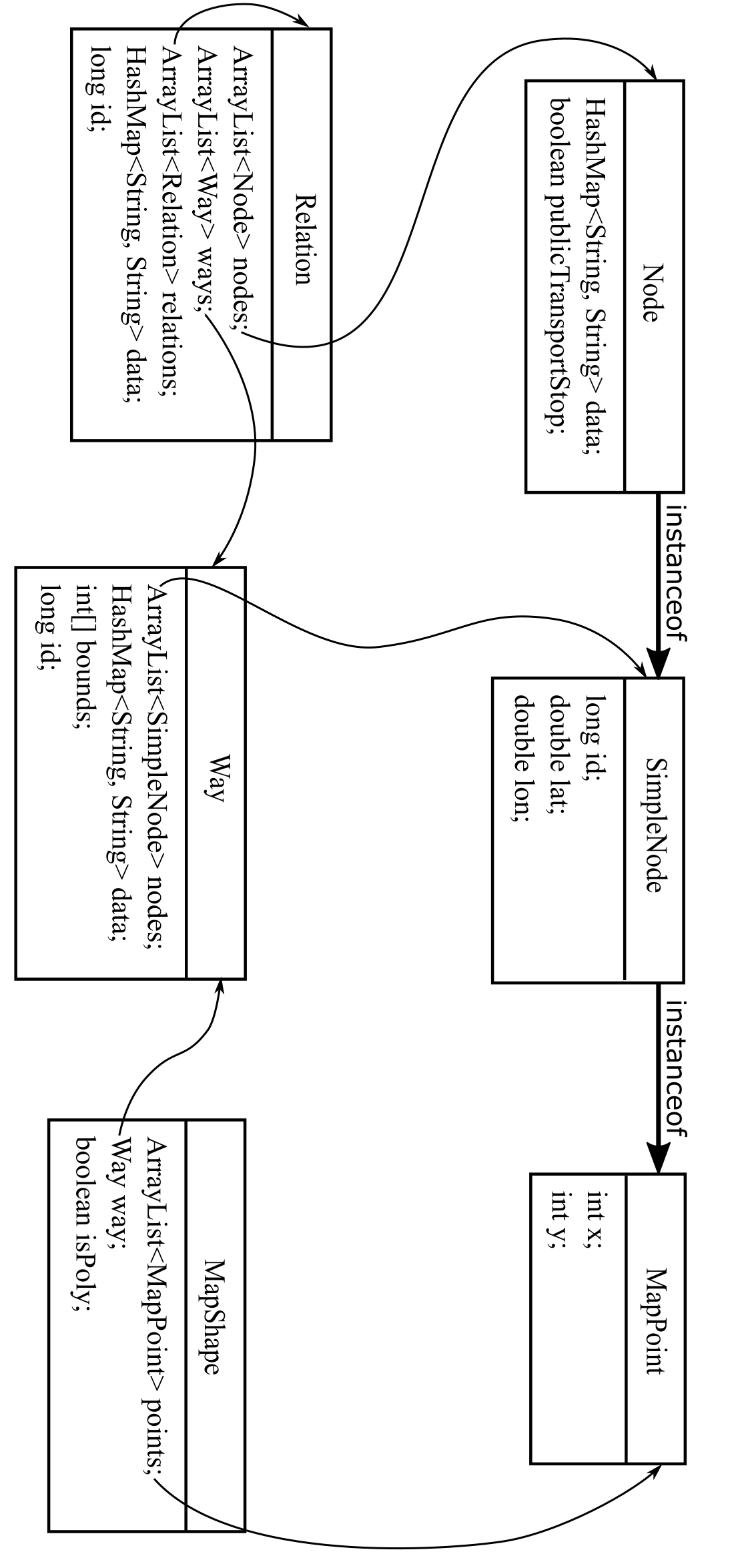
Расчет графа – нахождение минимально возможного времени пути из начальной точки до вершины для вершин графа, если это время меньше максимально допустимого.

При написание алгоритма для расчета графа по началу мною был написан рекурсивный алгоритм(PropertyMap.recCalculateDistances()), очень похожий на поиск в глубину, но вместо сравнения вершины с искомой, я записываю в вершину сумму весов уже пройденных ребер, если она меньше уже записанной в вершине. Перед этим во все вершины записывается максимально возможное число(INT\_MAX). Но при его использование я понял что время его работы слишком велико. Тогда я решил попробовать реализовать расчет графа на основание поиска в ширину(PropertyMap.widthRecCalculateDistance()). Изменения скорости работы показаны в Приложении 4 и Приложении 5.

Даже с ускорением после использования расчета в ширину расчет графа занимает 500 мс. из-за чего нельзя пересчитывать граф на каждый запрос нового тайла. Также при попытки генерировать тайлы параллельно из-за необходимости синхронизировать расчет графа, выигрыш от параллельного исполнению сходит на нет. Нужно временно сохранять результат расчетов для дальнейшей генерации тайлов и для того что бы можно было рассчитывать граф параллельно. Кэш реализован в виде массива в каждой RoadGraphNode, в котором хранится её дальность. Также создан класс следящий за тем какие места в этом массиве уже заняты, а какие свободны и могут быть использованы для новой записи.

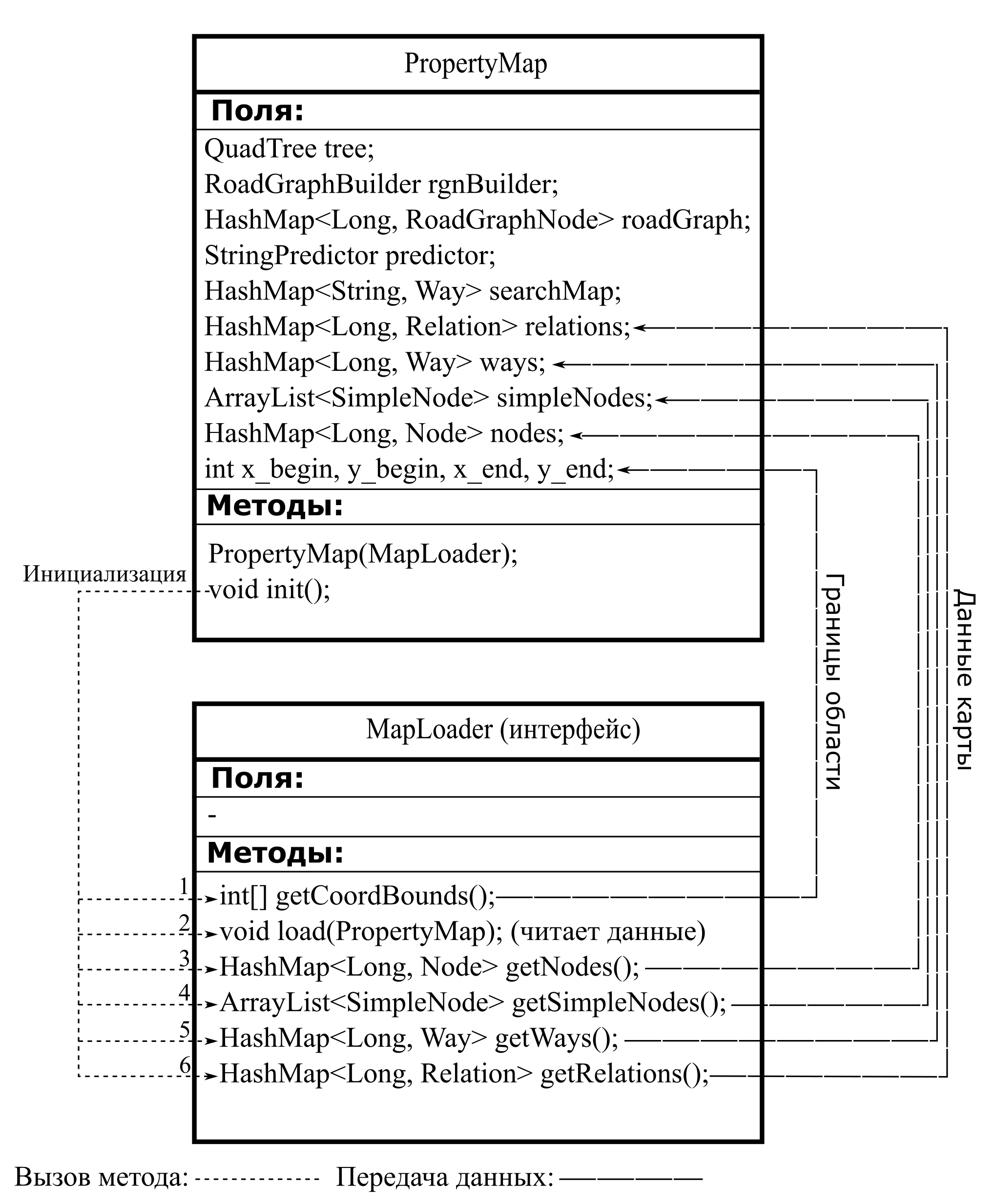
**Приложения**

Классы представляющие геоданные.



Приложение 1.

Процесс загрузки гео данных из файла.



Приложение 2.

Зависимость времени расчета от максимальной дальности.

*Приложение 4.*

График выше, без расчета в глубину. Приложение 5.