**Министерство образования и науки Российской Федерации**
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**
**(ФГБОУ ВО «АмГУ»)**

Факультет математики и информатики
Кафедра информационных и управляющих систем
Направление подготовки 09.04.01 – Информатика и вычислительная техника
Магистерская программа Компьютерное моделирование

ДОПУСТИТЬ К ЗАЩИТЕ
Зав. кафедрой
_____ А.В.Бушманов
«___»_____ 2017г.

# МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему: Приложение для мобильных устройств реализующее навигацию внутри зданий

Исполнитель
студент группы 553 ом _____ В.В. Фурсов

Руководитель
доцент, канд. техн. наук _____ С.Г. Самохвалова

Руководитель магистерской
программы
профессор, докт. техн. наук _____ Е.Л. Ерёмин

Нормоконтроль
доцент, канд. физ.-мат. наук _____ В.В. Ерёмина

Рецензент
доцент, канд. физ.-мат. наук _____ Д.В. Фомин

Рецензент
доцент, канд. техн. наук _____ Т.В. Труфанова

Благовещенск 2017

# РЕФЕРАТ

Магистерская диссертация содержит 67 с., 23 рисунка, 3 приложения, 38 источников.

ПОИСК ПУТИ, ГРАФ ВИДИМОСТИ, СЕТКА НАВИГАЦИИ, ПУТЕВЫЕ ТОЧКИ, МОБИЛЬНОЕ ПРИЛОЖЕНИЕ, КРОССПЛАТФОРМЕННОСТЬ

Объектом исследования магистерской работы является поиск пути внутри зданий на мобильном устройстве.

Предметом исследования являются алгоритмы поиска путей, их работоспособность в условиях недостатка ресурсов оперативной памяти на мобильном устройстве.

Целью работы является исследование существующих алгоритмов поиска путей и создание программного комплекса, позволяющего выполнять навигацию внутри здания с помощью мобильного устройства.

Для выполнения поставленной цели в работе решаются следующие задачи:

1. анализ существующих кроссплатформенных решений для создания мобильных приложений и выбор подходящего под заданную задачу;

2. анализ существующих алгоритмов поиска путей для оценки ситуации в предметной области, а также выявление путей повышения эффективности алгоритмов поиска пути;

3. тестирование производительности алгоритмов поиска путей;

4. разработка программного комплекса, выполняющая предложенные алгоритмы.

При решении поставленных задач использовались методы системного анализа, поиска пути, проектирования информационных систем и языки программирования.

Научная новизна основных результатов работы состоит в разработанном способе создания карт для навигации по ним и реализации алгоритмов, в условиях недостатка оперативной памяти.

Результаты магистерской работы могут найти применение в областях навигации внутри зданий, робототехнике, играх. Разработанный программный комплекс может быть непосредственно применён для навигации в любом существующем здании на мобильном устройстве.

СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Большое количество работ посвящено решению проблем локальной и гео навигации, обзор которых приведен в [1], но проблемы локальной навигации — навигации внутри помещений и зданий остается актуальной. Стоит отметить значительную заинтересованность в сервисных услугах, предоставляемых на основе местоположения клиента и его предпочтений. Структура зданий с каждым днём усложняется, они становятся более объёмными, в них вносятся изменения планировки, которые не всегда оперативно учитываются. В сооружениях такого типа только постоянные посетители могут уверенно ориентироваться. Очевидно, что в такой ситуации на освоение в незнакомом месте придётся потратить огромное количество времени. Поэтому, возникает потребность в таком сервисе, который поможет пользователю максимально просто и без траты лишнего времени добраться до нужного ему места в здании. Такие системы, как: ГЛОНАСС, GPS, Galileo и др., обеспечивающие работу таких сервисов, как 2GIS, Google Maps, NAVIMIND и др., ориентированы на решение задач геонавигации и не решают проблемы локальной навигации. Так же стоит отметить, что решения проблемы локальной навигации часто являются актуальными не только внутри, но и вне зданий – в условиях плотной застройки часто неэффективны даже системы, предназначенные специально для навигации на открытой местности.

Здания становятся все более громоздкими и классические методы навигации сильно теряют в эффективности. Решение в виде настенных планов уже не являются наглядными, особенно если размеры здания весьма велики. Зачастую конфигурация этажей отличается, что вносит еще больше путаницы в попытку сориентироваться и определить свое местоположение в здании. Вариант использования указателей также неэффективен, так как они используются лишь для обозначения самых важных помещений. Если попытаться установить в здании указатели для всех помещений, то посетитель окажется просто переполнен количеством информации, в которой ему будет необходимо разобраться.

Для решения проблемы локальной навигации, существует не так много готовых решений, а открытых и свободно распространяемых решений не существует вовсе. Поэтому передо мной стояла задача реализации открытого и свободно распространяемого решения, с открытым исходным кодом, которое работало бы на мобильных устройствах.

Первая глава диссертации содержит анализ предметной области, в частности информация о мобильных приложениях; краткий обзор и анализ программных продуктов, предназначенных для кроссплатформенной разработки под мобильные устройства; обзор и анализ существующих алгоритмах поиска пути.

Во второй главе диссертационной работы представлено проектирование информационной системы, общие сведения о ней, назначение и цели создания системы, требования к проектируемой системе, состав и содержание работ по созданию системы.

В третьей главе представлен анализ информационной системы, обоснование необходимости создания программного комплекса, анализ существующих решений для навигации внутри зданий на мобильных устройствах, протестированный различные алгоритмы для поиска пути.

В четвертой главе диссертационной работы выполнена программная реализация приложения, рассмотрены решения для создания карт здания.

# 1 Анализ предметной области

## 1.1 Мобильное приложение

Мобильное приложение – это специально разработанное приложение под конкретную мобильную платформу (iOS, Android, Windows Phone). Обычно приложение разрабатывается на языке высокого уровня и компилируется в нативный код ОС, дающий максимальную производительность.

Мобильные приложения не так зависимы от наличия выхода в интернет в конкретный момент времени, и, в отличие от веб-сайтов, многие операции можно осуществлять в оффлайн режиме.

## 1.2 Кроссплатформенная разработка мобильных приложений

Разработка приложений для персональных компьютеров возможна на большом количестве языков программирования и относительно проста в тестировании, пока речь не заходит о кроссплатформенной разработке приложений под разные операционные системы, включая такие, как мобильные операционные системы. Рассмотрим обзор кроссплатформенных фреймворков для мобильных операционных систем. В настоящее время популярны следующие мобильные операционные системы: iOS, Android и Windows Phone.

Большинство мобильных приложений разработано на языках программирования, стандартных для используемой мобильной операционной системы, называемые «нативными» языками программирования. iOS приложения создаются в среде разработки XCode на языках Swift, Objective-C, C и C++. Для создания приложений под Android используется среда Android Studio и язык Java. Для Windows Phone среда Visual Studio и язык C#. Но иногда существует необходимость создать приложение, которое бы работало на нескольких, отличных друг от друга операционных системах. Для этого используются специальные, кроссплатформенные решения, называемые фреймворками.

Кратко рассмотрим отличия нативной разработки (с использование родного языка программирования), от использования кроссплатформенных фраймворков.

От нативной разработки всегда получаем возможность, использовать все возможности мобильного устройства и операционной системы, а так же, обширное количество документации. Но к сожалению, если появится необходимость в запуске разработанного приложения на другой мобильной операционной системе, придётся заново, с нуля создать приложение под другую мобильную операционную систему. В данной ситуации, может помочь, разработка мобильного приложения с использованием кроссплатформенного Фреймворка.

С использованием кроссплатформенного фреймворка есть несколько решений проблемы, работы приложения на разных мобильных операционных системах. Рассмотрим следующие популярные решения для кроссплатформенной разработки:

Appcelerator Titanium;

PhoneGap;

Xamarin;

RAD Studio.

С Appcelerator Titanium приложения пишутся с помощью языка программирования JavaScript, но используется трансляция в интерфейс платформы. Существует возможность разработки нативного пользовательского интерфейса для каждой мобильной платформы. Titanium позволяет расширять свои возможности с помощью плагинов написанных на нативном языке каждой платформы.

PhoneGap позволяет расширять свои возможности с помощью плагинов написанных на нативном языке каждой платформы. К сожалению, в PhoneGap, в отличии от Titanium, не существует возможности использовать нативные элементы интерфейса, а вместо этого внутри приложения создается веб-браузер, внутри которого располагается обычная HTML-разметка – это означает, что на всех платформах приложение будет выглядеть практически одинаково, что является минусом, так как интерфейс не будет соответствовать интерфейсу платформы.

Xamarin позволяет создавать приложения с помощью языка C#. В отличии от Titanium, код компилируется сразу в нативный код, а не интерпретируется на стадии выполнения. Поэтому производительность и пользовательский интерфейс та-

кой же, как и у родных приложений. Пользовательский интерфейс создается для каждой платформы с помощью стандартных инструментов для платформ. Возможность импорта уже написанного нативного кода.

В RAD Studio приложения создаются с помощью языка Delphi и так же, как и в Xamarin, компилируется сразу в нативный код.

К сожалению, Titanium и PhoneGap имеют очень низкую производительность, так как не в полной мере используют аппаратные возможности устройства. Данные фреймворки пригодны для прототипов и простых приложений, без множества логики.

В Xamarin и RAD Studio используется компиляция, что очень хорошо сказывается на производительности.

После оценки плюсов и минусов, был выбран кроссплатформенный фреймворк Xamarin. Он позволяет в полной мере использовать все возможности конкретной мобильной операционной системы, а при помощи компиляции достигается высокая производительность работы приложения. Так же, несомненным плюсом, является возможность создания своего пользовательского интерфейса для каждой мобильной платформы и написания платформозависимого кода, при необходимости, для реализации высокоэффективного кода или использования инструментария, доступного только в ограниченном числе платформ, не прибегая к использованию нативных языков. Хотя при использовании нативного пользовательского интерфейса или платформозависимого кода снижается объём переиспользуемого кода, однако, ядро приложения со всей бизнес-логикой, остаётся одинаковым для любой платформы, в соответствии с рисунком 1.

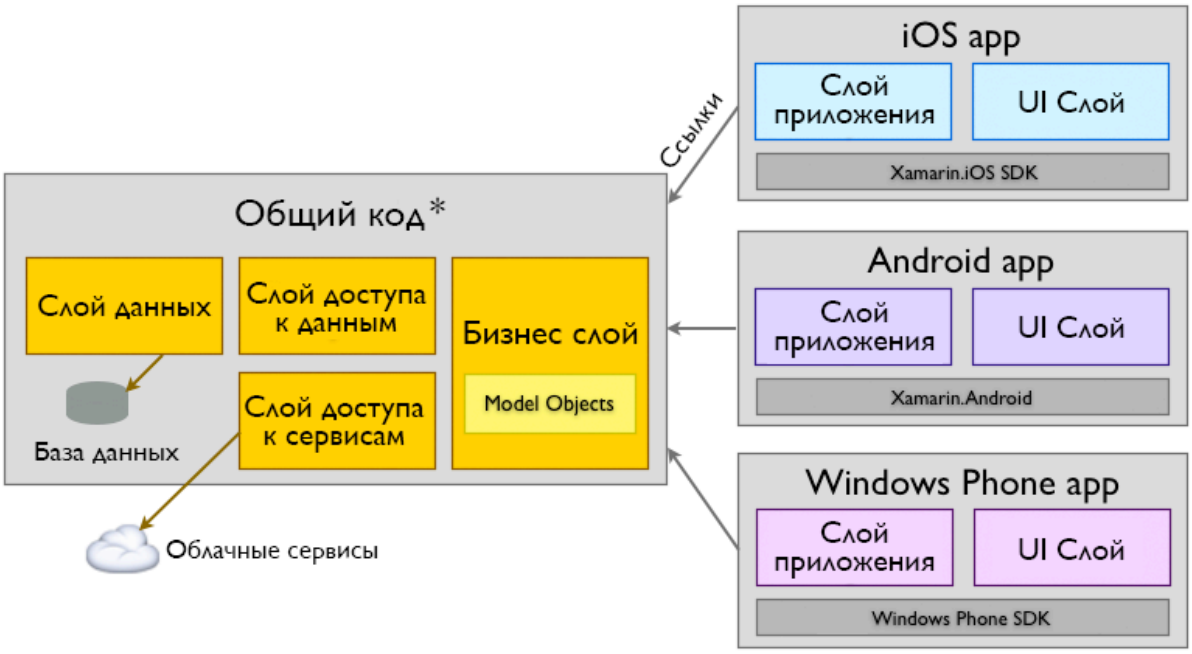



Рисунок 1 – Диаграмма связи переиспользуемого кода и интерфейса

В результате данных, полученных после проведённого исследования, был выбран кроссплатформенный фреймворк Xamarin и язык программирования C#, которые в дальнейшем будут использованы для создания мобильных приложений для студентов АмГУ, которые помогут первокурсникам следить за своей успеваемостью по бально-рейтинговой системе, а также, в навигации внутри зданий университета.

### 1.3 Поиск пути

Поиск пути является основой для применения в области GPS [6], видео-играх [7], робототехнике [8], логистики и моделирования толпы [9,10]. Так же, поиск пути, может быть реализован в статических, динамических и в средах реального времени. В течении последних двух десятилетий удалось повысить точность и эффективность методов поиска путей, но задача по прежнему привлекает большое количество исследований. В настоящее время, наиболее важным является высокая производительность и реалистичные пути для пользователей. Существуют различные варианты задачи поиска пути, такие как:

а) С использованием единственного или множества агентов.

б) С меньшей стоимостью.

в) В окружающей среде с динамическими изменениями.

г) В неоднородной местности.

д) На мобильных устройствах.

е) С неполной информацией.

Каждая из этих проблем имеет различные решения в различных областях. Как правило, поиск пути состоит из двух основных этапов: генерации графов и алгоритма поиска пути.

Задача формирования графов с топологией местности считается основой в робототехнике и видео-играх. В этой задаче поиска пути, навигация производится в различных сплошных средах, таких как известные 2D или 3D среды и в неизвестных 2D средах. Несколько различных методов были предложены для представления навигационной среды для трёх данных сценариев. Каждый из представительных графов относится к одному из двух методов, скелетирование или декомпозиция ячеек. Методы скелетизации извлекают каркас из сплошной среды. Этот скелет захватывает характерную топологию проходимого пространства, определяя граф

$$G = (V, E), \tag{1}$$

где V представляет собой набор вершин, чтобы сопоставить координаты в непрерывной среде и E множество ребер, соединяющих вершины, которые находятся на линии видимости друг от друга.

Как правило, методы скелетирования могут производить два типа неправильных сеток, а именно, граф видимости или путевые точки. Методы декомпозиции ячеек разбивают проходимое пространство в сплошной среде на ячейки. Каждая ячейка, как правило, задаёт круг или выпуклый многоугольник и представляет собой область пространства с проходимостью без каких-либо препятствий. Поскольку клетки представляют собой окружности или выпуклые многоугольники, которые не содержат препятствия, агенты могут перемещаться по прямой линии между любыми двумя координатами в пределах одной ячейки (без планирования

пути). Есть целый ряд гипотез о свойствах карты местности, производимых скелетизацией и декомпозицией ячеек следующим образом:

1. Масштабирование карты местности создаст ненужное пространство и изменения в исходной карте.

2. Существует значительная разница между свойствами сетки карт местности (правильной и неправильной), в зависимости от того, используется ли они для игр или робототехники.

3. Проектирование искусственных карт местности для тестирования, должны учитывать свойства реальных карт местности.

Вторым шагом в процессе поиска пути является сам алгоритм поиска. Здесь задача в том, чтобы вернуть оптимальный путь для пользователей, эффективным образом. «А звёздочка» [11], является одним из наиболее известных алгоритмов поиска пути. Это был первый алгоритм, который использовал эвристическую функцию, чтобы путешествовать по графу с наименьшей стоимостью от начальной вершины до выбранной конечной.

Алгоритм «А звёздочка» вдохновил многие модифицированные и усовершенствованные алгоритмы. Для того, чтобы оценить эффективность таких алгоритмов, надо принимать во внимание время выполнения, накладные расходы памяти и является ли среда поисковой системы статической, динамической, или реального времени "детерминированной".

Отношения между различными типами топологий местности показаны на рисунке 2.

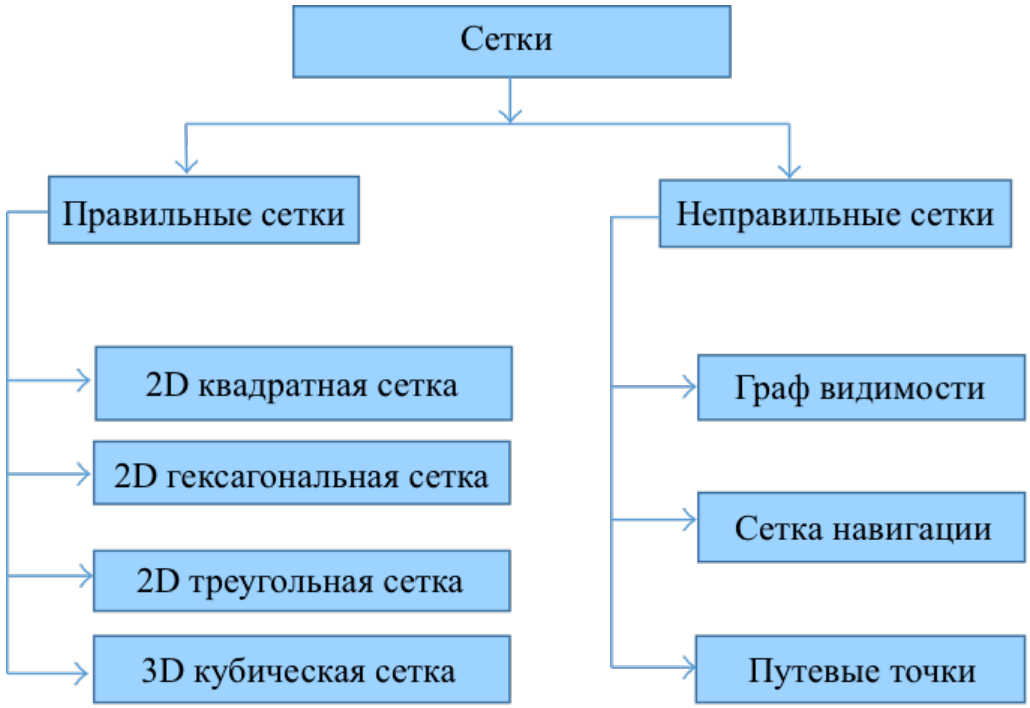


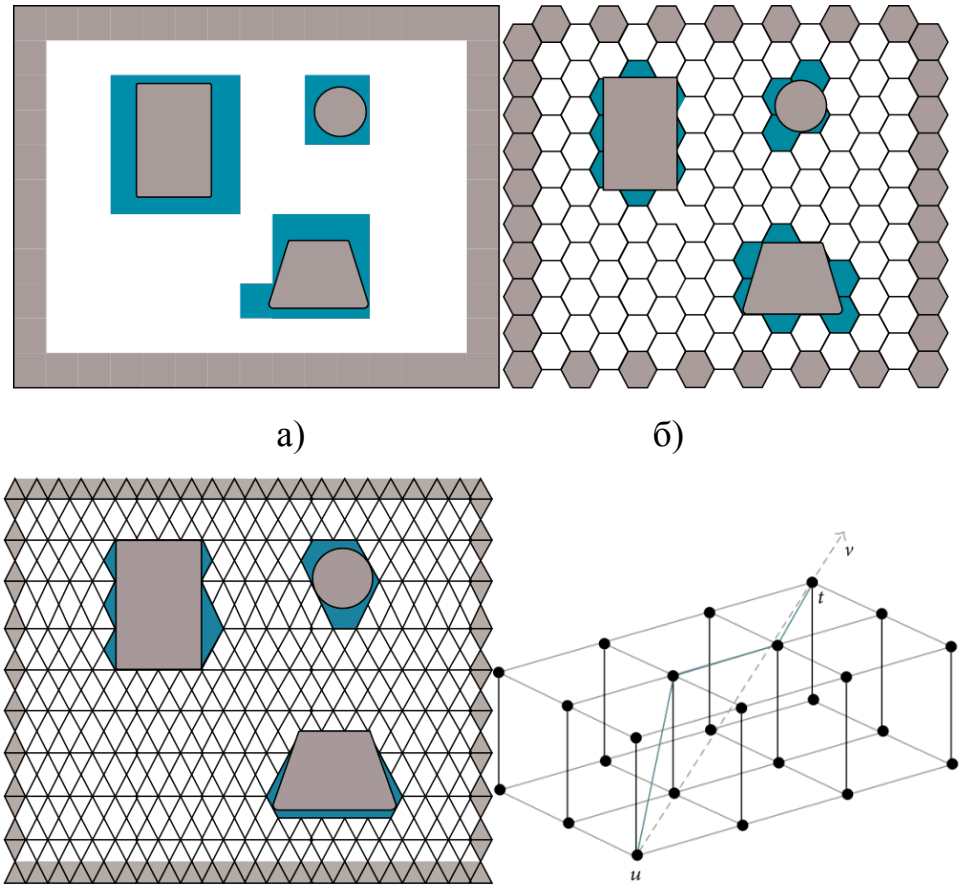

Рисунок 2 – Топология местности

### 1.3.1 Методы на основе сетки

Сетка состоит из вершин или точек, которые соединены между собой ребрами и представляют из себя граф. В большинстве алгоритмов поиска пути, производительность навигации — это основной атрибут представления графа. Фундаментальной концепцией является два популярных подхода, основанных на: правильных и неправильных сетках.

#### 1.3.1.1 Правильные сетки

Правильные сетки являются одним из самых известных типов графов и широко используются. В 2D и 3D средах, регулярные сетки описывают мозаику правильных многоугольников (т.е. равносторонних и равноугольных полигонов). Шестиугольники, квадраты и треугольники являются единственными правильными многоугольниками, которые могут быть использованы как непрерывная мозаика в 2D, в соответствии с рисунком 3 (а, б, в) и кубические решетки, в соответствии с рисунком 3 (г).




а) б)

в) г)

Рисунок 3 – Правильные сетки (а – квадратные сетки с тремя препятствиями; б – гексагональные сетки с тремя препятствиями; в – триангуляционные сетки с тремя препятствиями; г – кубическая решётка без препятствий)

а) 2D квадратная сетка

Квадратные сетки являются одними из наиболее популярных графов в компьютерных играх и навигации по карте, а также многочисленные алгоритмы были предложены для решения задачи поиска пути для этого типа сетки, в соответствии с рисунком 3 (а). Харабор и Грастен [12] предложили алгоритм «прыжковых точек» (jump point search) с одним агентом для решения общей задачи в играх и навигации, а именно, хорошо известной «единичной стоимость сетки в статической среде». Jump point search , в 10 раз быстрее, чем алгоритм «А звёздочка» и имеет небольшие накладные расходы на память. Они оценивали свою работу на основе стандартного набора карт для поиска пути, предложенного Стертевант [13].

Урас и др. [14] опубликовали метод для ускорения поиска пути, путем создания графа подцели. Основная идея этого документа состоит в определении после-

довательности клеток таким образом, что агент всегда движется в простой подцели. Вариации этого метода используют двухуровневые графы подцелей и двухуровневые графы подцелей с попарными расстояниями. В квадратной сетке, существует предел из восьми соседей.

Харабор и Грастен [15] улучшили алгоритм «прыжковых точек» (jump point search), используя три стратегии.

Во-первых, они рассматривают узлы, как базовые блоки операций, а затем использовали систему предварительной обработки в автономном режиме. Третья стратегия представляет собой усовершенствованный набор действующих правил обрезки деревьев. Полученные алгоритмы сравнивались с методом Ураса.

Бная и др. [16] предложили исчерпывающие итерационные алгоритмы (Eita / MC-ITA) для мультиагентной задачи поиска пути. Эти алгоритмы предполагают, что агенты могут обнаружить конфликты во время навигации, назначить приоритет одному агенту и использовать его путь, в то время как другой добавляет к его общей стоимости штраф. Основным недостатком этих методов является то, что приоритеты случайным образом распределяются одному или другому агенту.

В отношении изменяющихся графов в присутствии "динамических препятствий", Андерсон [17] представил отличное дополнение эвристики для поиска с единственным агентом на четырёх-направленной квадратной сетке. Этот метод эффективно сокращает время поиска в неявных графах и единственным недостатком является то, что они реализовали его с алгоритмом «A звёздочка», который излишне расширяет число узлов во время поиска.

Для мультиагентной задачи, Джин и др. [18] использовали алгоритм «A звёздочка» в XNA игре, для того чтобы решить проблему поиска пути. Они использовали алгоритм «A звёздочка» для обработки динамических препятствий с 2D квадратной картой сетки, несмотря на 3D реализацию. Недостатком их работы является то, что «A звёздочка», не всегда является оптимальным для статических карт сетки.

б) 2D гексагональная сетка

Бьорнссон и др. [19] показали, что гексагональные сетки, в соответствии с рисунком 3 (б), имеют много свойств квадратных сеток. Кроме того, шестиугольные сетки имеют меньшее время поиска и потребление памяти, чем сложные сетки графов, построенных из квадратов. Кихано и Гарридо [20] использовали шесть алгоритмов гексагональной сетки для моделирования разведки робота. Они доказали, что алгоритмы на гексагональных сетках, превосходят квадратные сетки для одно- и мультиагентных задач.

Осман и др. [21] обсудили проблемы, с которыми сталкиваются исследователи, которые используют алгоритмы поиска пути в робототехнике. Одной из главных проблем в мобильной робототехники является избегание реальных объектов. Алгоритм D* был применен к методам моделирования 2D-карты с помощью квадратной плитки, шестиугольников и усовершенствованных шестиугольников. Результаты показывают, что усовершенствованные шестиугольники дают лучшие пути, но требуют больше времени для поиске в больших картах. Тем не менее, приемлемые результаты достигаются для малых и средних по размеру карт. Алгоритмы поиска пути в основном были реализованы на квадратных сетках, а мультиагентная задача на шестиугольной сетке еще не была исследована в динамических средах или в средах реального времени.

в) 2D треугольная сетка

Треугольные сетки, в соответствии с рисунком 3 (в) не так широко используется как квадратные и гексагональные сетки, но у них есть некоторые хорошие свойства. Димайн и Буро [22] был предложен способ, который уменьшает затраты на поиск с использованием ограничений Триангуляции Делоне. Изменяя размеры объектов, авторы исследовали влияние окружающей среды на движение во время навигационной задачи. Используя их алгоритмы TA* и TRA*, было обнаружено, что они безупречно работают на больших картах.

Наги [23] получил новую систему координат, состоящую из двух гексагональных и треугольных сеток графов. Эта новая сетка может использоваться в различных областях, например таких, как компьютерная графика и обработка изоб-

ражений. Считается, что эта новая топология может быть использована с детерминированными методами поиска пути, такими как A* и его вариантами.

г) 3D кубическая сетка

В отличие от сеточных графов, рассмотренных выше, кубическая решетка, в соответствии с рисунком 3 (г) является привычным графом для 3D среды. Кылыч и Ялчин использовали простой алгоритм для вычисления движения трех типов волн в 3D-среде и использовали его, чтобы решить проблему поиска пути для робота. Позже авторы улучшили алгоритм, для выполнения в среде 3D. В предложенной модели, робот имеет всего шесть соседских клеток, которые могут быть использованы для навигации к цели. Это является явным недостатком этой модели и число соседних клеток должно быть увеличено, чтобы включать в себя все 26 соседних ячеек для улучшения получившегося пути. Не так давно, Нэш и Кениг [24] опубликовал отличную статью под названием "Any-Angle Path Planning". Они ввели три различных алгоритма, A*, Theta* и Ленивая Theta*, для квадратных, шестигранных, треугольных и кубических сеток. Целью этих алгоритмов является нахождение плавного реального пути для любой топологии местности без учета размера агента или формы препятствий. На основании наблюдаемой производительности, они определили, кратчайшие пути, а не самых коротких путь на сетке. К сожалению, алгоритм "Any-Angle Path Planning" потребляет больше времени, чем обычные алгоритмы поиска пути.

1.3.1.2 Неправильные сетки

Аномальные сетки используются во многих различных задачах и областях.

а) Граф видимости

Граф видимости, в соответствии с рисунком 4 (а) считается фундаментальной структурой в различных областях геометрической теории графов и вычислительной геометрии, и привлекает к изучению различных задач. Граф видимости был недавно был использован при вычислении евклидовых кратчайших путей при наличии препятствий. [25]

Для того, чтобы решить проблему с одним агентом, Надэан-Тахан и Манзури-Шалмани [26] предложили новый генетический алгоритм для поиска эффек-

тивного пути через ряд расширяющихся препятствий. Кроме того, они использовали стратегию для ускорения скорости сходимости путем создания подходящей начальной популяции. Авторы использовали общую вычислительную геометрию на основе подхода суммы Минковского, а не на основе методов ячеек, таких как сетки. Тем не менее, этот метод показал, что он быстрее, чем другие подходы вычислительной геометрии, но производит не оптимальные решения.

б) Сетка навигации

В соответствии с рисунком 4 (б), проходимые участки карты представляют собой навигационную сетку. Эта сетка может быть представлена разными способами, например, с помощью треугольников или многоугольников. Ведь, сетчатые графы и графы видимости очень похожи, но это на самом деле не так. В практическом плане графы видимости являются более сложными, чем сеточные графы. Чаще всего сетки графов применяются в видео-играх.

Сислак и др. [27] представили алгоритм, вдохновленный A*. Их алгоритм AA* (Accelerated(ускоренный) A*) позволяет планировать траекторию полёта на основе задачи поиска пути с единственным агентом. Алгоритм AA* был использован в динамичной среде для моделирования движения самолета. Основными преимуществами является его способность рассматривать самолет не нулевого размера и планировать путь для направленного вектора вверх от горизонтальной плоскости.

Для мультиагентного поиска пути, Кападия и др. [28] предложили структуру режима реального времени для мультиагентной навигации, которая использует несколько неоднородных проблемных областей в больших, сложных и динамических виртуальных средах.

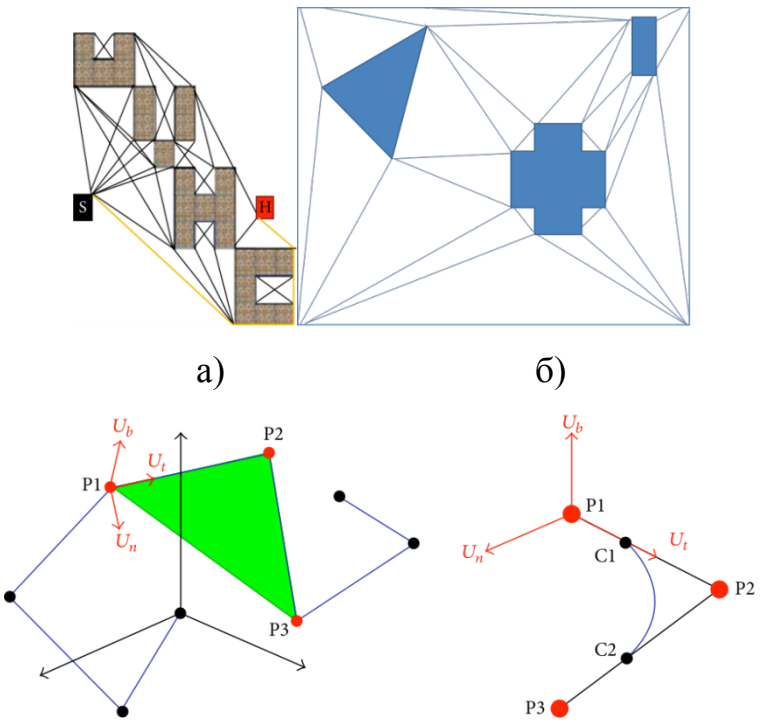
в) Путевые точки

Путевые точки, в соответствии с рисунком 4 (в), широко используются в компьютерных играх и робототехники. Нидерберг и др. [29] предложили алгоритм на основе A* для поиска пути в статических местностях с полигональными препятствиями. Алгоритм генерирует путь в соответствии с минимальным числом то-

чек пути, но страдает от высокого времени вычислений и накладных расходов памяти.

Фергюсон и Стентз [30] разработали алгоритм Field D* с интерполяцией на основе планирования, и описали алгоритм перепланировки для генерации малозатратных путей через однородные и неоднородные сетки. Есть два недостатка в поиске пути на основе сетки. Во-первых, планирование на основе сетки имеет ограниченную способность в нахождении оптимального пути, потому что путь должен проходить через соседние точки. Во-вторых, требования к памяти на основе планирования пути часто неприемлемо высоко.

Бурчард и Саломон [31] предложили скорректированный генетический алгоритм для поиска пути на основе роботов малого размера RoboCup. Из-за природы динамичной среды, робот продолжает вычислять путь, пока он не достигнет цели.




а) б)

в)

Рисунок 4 – Неправильные сетки (а – Граф видимости (коричневые представляют препятствия); б – сеточная система навигации с треугольной сеткой (синие области представляют собой препятствия); в – 3D пути сглаживания)

### 1.3.2 Иерархические методы

Одним из недостатков методов, которые используют правильные и неправильные сетки является то, что они требуют значительного объема памяти. Иерархические методы уменьшают проблему с памятью, позволяя сплошной среде быть дискретизированной. Применяя более мелкую зернистость в некоторых регионах, может быть получено более точное представление, особенно вблизи препятствий. Более грубая зернистость применяется в тех регионах, где в деталях нет необходимости, в таких как широкие и открытые пространства.

### 1.3.3 Вероятностные дорожные карты

Коултер [33] предложил алгоритм отслеживания "pure pursuit algorithm"("чистый алгоритм слежения") для робототехники и наземных навигационных задач. Алгоритм вычисляет искривление, по которому будет двигаться робот "транспортное средство" от текущей позиции к целевой. Автором пренебрегаются два важных параметра в статье: динамические препятствия и природные особенности, такие как горы, долины, ручьи, и так далее. В том же контексте, Кай и Гое [34] обсудили и разработали математический метод расчета траектории для различных типов транспортных средств большой длины. Кроме того, они разработали 3D-модель для моделирования городов, который включает в себя поворотные дороги в окружении домов и других объектов.

Чои и др. [35] представили новую схему для поиска пути естественного, вида фигуры двуногого существа, для облегчения быстрого прототипирования движения и генерации движения на уровне задач. В случае единственного агента, начало и цель определены в виртуальной среде и предложенная схема выводит последовательность движений, чтобы перейти от начальной точки до точки цели, используя захват движения. Эта схема состоит из трех частей: генерация движения, поиск дорожной карты и постройка дорожной карты. К сожалению, эта система применяется только в случае единственного агента и может обрабатывать только статические препятствия. Динамические препятствия и в режиме реального времени, делают планирование движения более сложным.

# 2 ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ

## 2.1 Общие сведения

Наименование разрабатываемой системы: Навигация внутри зданий (Мобильное приложение).

Разработчик: студент второго курса факультета математики и информатики, направление «Информатика и вычислительная техника» Амурского государственного университета, Фурсов Владислав Валерьевич.

Основанием для проведения разработки является выполнение выпускной квалификационной работы.

Плановые сроки начала и окончания работ по созданию системы:

Срок начала работ: 1.02.2016 г.

Срок окончания работ: 1.06.2017 г.

## 2.2 Назначение и цели создания системы

### 2.2.1 Назначение системы

Система предназначена для локальной навигации внутри зданий с помощью мобильного приложения в соответствии с требованиями, зафиксированными в данном Техническом задании.

Информационная система необходима для:

– быстрого поиска необходимого кабинета в здании;

– прокладки маршрута между кабинетами внутри зданий;

– поиска кратчайшего пути в здании.

Так же к функциям можно отнести:

– ввод и хранение избранных кабинетов и маршрутов;

– поиск кабинета по названию;

– загрузка карт и обновлений с сервера через интернет;

Система будет представлять собой мобильное приложение.

### 2.2.2 Цели создания системы

Целями разработки информационной ИС являются:

– быстрый поиск необходимых кабинетов внутри здания;

– построение кратчайшего и достоверного маршрута между двумя точками;

– удобство пользователя.

### 2.3 Требования к системе

Система должна работать под управлением операционных систем iOS и Android, производить прокладку маршрута по загруженным картам зданий и обладать функционалом выбора разных карт и местоположений, между которыми будет строиться навигационный маршрут. Время постройки пути для навигации должно быть не заметно для пользователя.

### 2.4 Состав и содержание работ по созданию системы

Этапы, которые необходимо выполнить для создания информационной подсистемы:

1. Исследовать предметную область, провести анализ существующих решений.

2. Составить техническое задание: выяснить и уточнить функции системы, определить технические и программные средства, необходимые для реализации ИС.

3. Непосредственно проектирование информационной системы: разработка эскизного и технического проектов. На этапе эскизного проекта выполняются следующие работы: инфологическое проектирования подсистемы, логическое проектирование, физическое проектирование.

4. Программная реализация информационной подсистемы.

5. Согласовать созданную информационную подсистему с требованиями.

6. Внедрить созданную систему: установить и настроить программно-аппаратные средства, выявить и устранить ошибки и неполадки.

7. Составить документацию.

### 2.5 Требование к документированию

Состав и содержание документации должны соответствовать требованиям ГОСТ 34.201-89 и нормативно-технических документов (комплекса стандартов и

руководящих документов на автоматизированные системы и единой системы программной документации).

Документация на проектируемую систему должна включать:

– рабочую документацию (на систему в целом, достаточную для ввода в действие, функционирования и обеспечения работоспособности системы);

– эксплуатационную документацию, предназначенную для использования при эксплуатации системы;

– документацию на программные средства вычислительной техники;

– техническое задание;

– эскизный проект;

– технический проект;

– сведения о тестировании системы (включая тестовые данные).

Перечень документов подлежащих разработке на систему: схема функциональной структуры; описание организации информационной базы; руководство по организации сопровождения; программа и методика испытаний; описание применения; технологическая инструкция.

Перечень документов подлежащих разработке по каждому комплексу задач, входящих в разрабатываемую систему: описание постановки комплекса задач с перечнем выходных данных (документов); описание технологического процесса обработки данных; руководство пользователя.

# 3 Анализ информационной системы

## 3.1 Обоснование необходимости создания программного комплекса

В настоящее время обозначилась проблема навигации внутри помещений различных зданий, а также предоставления посетителям услуг, основанных на их местоположении и предпочтениях. Здания становятся всё более объёмными и имеющими сложную структуру. В сооружениях подобного типа уверенно могут ориентироваться лишь те, кто в них побывал много раз, однако и они, чаще всего, ориентируются в зданиях лишь частично, в пределах своих нужд. Первоначальное же освоение может быть довольно затруднительным, а также есть немалое количество людей, у которых вообще нет нужды посещения определённых мест более, чем несколько раз. Очевидно, что, например, тратить час времени на поиск кабинета врача будет нецелесообразным и грозить опозданием на приём, не говоря уже об опозданиях на рабочее или учебное место. Поэтому возникает необходимость в инструменте, который поможет пользователю максимально быстро и без лишних усилий добраться до нужного ему пункта назначения.

Решения, применяемые в навигации внутри помещений, помогают и в ориентировании вне зданий, на улице – там, где в условиях плотной застройки использование систем спутниковой навигации затруднено (нет спутников в прямой видимости, присутствует только отражённый/ослабленный/зашумленный сигнал GPS/Глонасс и т.д.).

В связи с ростом указанных выше характеристик здания (объём и сложность структуры), некоторые методы уже не так эффективны, как раньше. Например, настенные планы теряют наглядность, если этаж здания имеет большую площадь и включает в себя большое количество помещений. Трудно взглянуть сразу на всё изображение и соотнести его с действительностью. Ситуация также может быть усугублена тем, что этажи могут иметь различную структуру. Тогда для каждого из них придётся составлять свой план, и объём информации, которой необходимо оперировать мысленно, возрастёт до неприемлемых величин. Есть и другой подход к ориентации внутри зданий: настенные указатели. Однако, они зачастую не

могут указать путь к конкретному пункту назначения. Чаще они используются для указания местоположения типовых пунктов, среди которых: справочная, магазин, пункт оказания первой помощи и пр. При попытке создать универсальную систему настенных указателей, возникнет проблема, аналогичная проблеме настенных планов, т.е. пользователю предоставляется не минимум необходимой информации, а полная информация о здании, которую необходимо самостоятельно анализировать.

В МГТУ им. Н.Э. Баумана, есть аудитория 501-Ю. Проход к этой аудитории напрямую через центральную часть здания невозможен, так как все переходы закрыты. Попасть в аудиторию 501-Ю можно только поднявшись снизу и только по одной — единственной лестнице, в соответствии с рисунком 5.

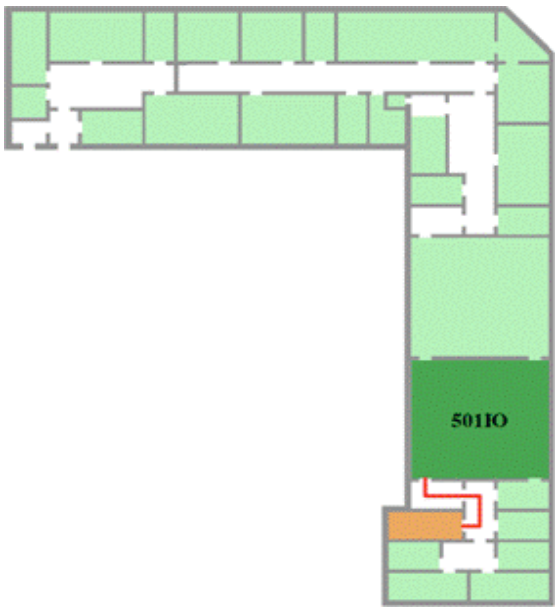



Рисунок 5 – План 5 этажа здания в МГТУ им. Н.Э. Баумана

Таким образом, студенты сталкиваются с другой проблемой: как найти эту самую лестницу? Попасть на нее так же довольно сложно: гарантированный проход есть только на 3 этаже. 3D карты с легкостью решают эту проблему, наглядно показывая весь путь до необходимой аудитории, в соответствии с рисунком 6.

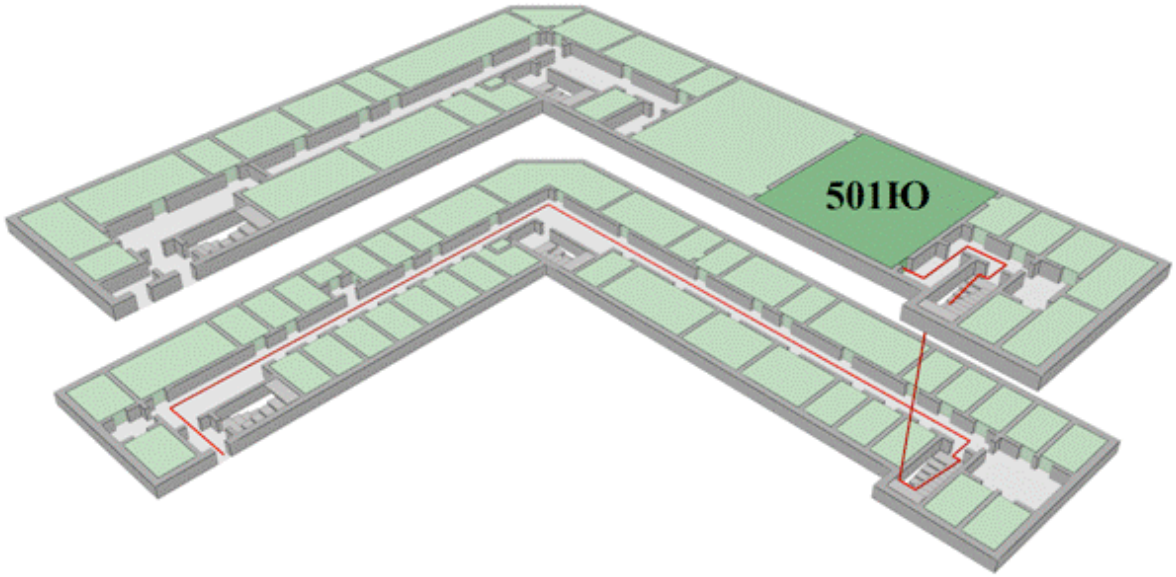



Рисунок 6 – 3D карта с проложенным навигационным путём

Построенный 3D путь наглядно показывает кратчайший (а часто и единственный) маршрут, по которому можно быстрее всего дойти до необходимой аудитории, аналогичные проблемы свойственны большинству зданий старой постройки, а также современным торговым комплексам.

### 3.2 Анализ существующих программ поиска пути внутри зданий

**Navigine**, помогает компаниям отслеживать перемещение клиентов по принадлежащим ей помещениям и разрабатывать систему навигации для них. Нужно установить оборудование (или использовать уже имеющуюся Wi-Fi или Bluetooth-инфраструктуру), подготовить карту помещения и интегрировать Navigine SDK в своё приложение.

Платное. Не работает без интернета, так как использует подключение к серверам для навигации.

**indoo.rs NavigationSDK** — инструмент позволяющий с лёгкостью использовать технологию позиционирования внутри помещений в своих мобильных приложения. Так же платное и не работает без интернета.

**NAV-IN –** Сервис навигации для мобильных устройств Nav-In работает на территории главного кампуса Владивостокского государственного университета

экономики и сервиса. Сервис позволяет открыть план университетского кампуса, найти точку назначения (помещение, объект или субъект) несколькими способами, автоматически сформировать маршрут, определить свое местоположение, получить подробную информацию об интересующем объекте. Например, сервис использует справочник сотрудников, телефонный справочник, расписание занятий преподавателей. Объекты мест интереса разделены на категории, каждая из которых представлена отдельным слоем на карте. Данное приложение только для Владивостокского государственного университета экономики и сервиса и нет возможности добавления своих карт.

**ИНФОРМАЦИОННО – НАВИГАЦИОННАЯ СИСТЕМА** – используется в МГТУ им. Н.Э. Баумана.

### 3.3 Обоснование выбранного алгоритма поиска пути

После проведённого обзора алгоритмов и модификаций, было решено протестировать их производительность и потребление памяти на мобильном устройстве. Для этого, было написано соответствующее программное обеспечение с использованием кроссплатформенного фреймворка Xamarin.

Для теста, были выбраны следующие алгоритмы:

Поиск в ширину (Breadth-first search);

A*;

Алгоритм прыжковых точек (Jump Point Search).

Рассмотрим данные алгоритмы подробнее:

#### 3.3.1 Алгоритм поиска в ширину

Поиск в ширину (Breadth-first search, BFS) – метод обхода графа и поиска пути в графе. Поиск в ширину является одним из неинформированных алгоритмов поиска.

В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер. Алгоритм работает за:

$$O(n+m), \quad (2)$$

где n — число вершин, m — число рёбер.

На вход алгоритма подаётся заданный граф (невзвешенный), и номер стартовой вершины s. Граф может быть как ориентированным, так и неориентированным, для алгоритма это не важно.

Сам алгоритм можно понимать как процесс "поджигания" графа: на нулевом шаге поджигаем только вершину s. На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня" в ширину на единицу (отсюда и название алгоритма).

Более строго это можно представить следующим образом. Создадим очередь q, в которую будут помещаться горящие вершины, а также заведём булевский массив used[], в котором для каждой вершины будем отмечать, горит она уже или нет (или иными словами, была ли она посещена).

Изначально в очередь помещается только вершина s, и used[s] = true, а для всех остальных вершин used[] = false. Затем алгоритм представляет собой цикл: пока очередь не пуста, достать из её головы одну вершину, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из просмотренных вершин ещё не горят, то поджечь их и поместить в конец очереди.

В итоге, когда очередь опустеет, обход в ширину обойдёт все достижимые из s вершины, причём до каждой дойдёт кратчайшим путём. Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей d[]), и компактно сохранить информацию, достаточную для восстановления всех этих кратчайших путей (для этого надо завести массив "предков" p[], в котором для каждой вершины хранить номер вершины, по которой мы попали в эту вершину).

Приложения алгоритма:

Поиск кратчайшего пути в невзвешенном графе.

Поиск компонент связности в графе за O(n+m).

Для этого мы просто запускаем обход в ширину от каждой вершины, за исключением вершин, оставшихся посещёнными (used=true) после предыдущих запусков. Таким образом, мы выполняем обычный запуск в ширину от каждой вершины, но не обнуляем каждый раз массив used[], за счёт чего мы каждый раз будем обходить новую компоненту связности, а суммарное время работы алгоритма составит по-прежнему O(n+m) (такие несколько запусков обхода на графе без обнуления массива used называются серией обходов в ширину).

Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа.

Классический пример — игра, где робот двигается по полю, при этом он может передвигать ящики, находящиеся на этом же поле, и требуется за наименьшее число ходов передвинуть ящики в требуемые позиции. Решается это обходом в ширину по графу, где состоянием (вершиной) является набор координат: координаты робота, и координаты всех коробок.

Нахождение кратчайшего пути в 0-1-графе (т.е. графе взвешенном, но с весами равными только 0 либо 1): достаточно немного модифицировать поиск в ширину: если текущее ребро нулевого веса, и происходит улучшение расстояния до какой-то вершины, то эту вершину добавляем не в конец, а в начало очереди.

Нахождение кратчайшего цикла в ориентированном невзвешенном графе: производим поиск в ширину из каждой вершины; как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину; среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.

Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин (a,b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через d_a[] массив кратчайших расстояний, полученный в результате первого обхода, а через d_b[] — в результате второго обхода. Теперь для любого реб-

ра (u,v) легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие

$$d\_a[u] + 1 + d\_b[v] = d\_a[b] \quad (3)$$

Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин (a,b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через d_a[] массив кратчайших расстояний, полученный в результате первого обхода, а через d_b[] — в результате второго обхода. Теперь для любой вершины v легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие:

$$d\_a[v] + d\_b[v] = d\_a[b] \quad (4)$$

Найти кратчайший чётный путь в графе (т.е. путь чётной длины). Для этого надо построить вспомогательный граф, вершинами которого будут состояния (v,c), где v — номер текущей вершины, c = 0 * 1 — текущая чётность. Любое ребро (a,b) исходного графа в этом новом графе превратится в два ребра ((u,0),(v,1)) и ((u,1),(v,0)). После этого на этом графе надо обходом в ширину найти кратчайший путь из стартовой вершины в конечную, с чётностью, равной 0.

### 3.3.2 Алгоритм A*

A* — алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

В данном алгоритме, эвристическая функция «расстояние + стоимость» (обычно обозначаемой как f(x)) определяет порядок обхода вершин. Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической,

так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Данный алгоритм был впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Это по сути было расширение алгоритма Дейкстры, созданного в 1959 году. Новый алгоритм достигал более высокой производительности с помощью эвристики. В их работе он упоминается как «алгоритм A». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A*.

A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы алгоритм просматривает узлы, смежные с начальным, выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению:

$$f(x) = g(x) + h(x) \quad (5)$$

Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё де-

рево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Чем меньше эвристика h(x), тем больше приоритет, поэтому для реализации очереди можно использовать сортирующие деревья.

Множество просмотренных вершин хранится в closed, а требующие рассмотрения пути — в очереди с приоритетом open. Приоритет пути вычисляется с помощью функции f(x) внутри реализации очереди с приоритетом.

Как и алгоритм поиска в ширину, A* является полным в том смысле, что он всегда находит решение, если таковое существует.

Если эвристическая функция h допустима, то есть никогда не переоценивает действительную минимальную стоимость достижения цели, то A* сам является допустимым (или оптимальным), также при условии, что мы не отсекаем пройденные вершины. Если же мы это делаем, то для оптимальности алгоритма требуется, чтобы h(x) была ещё и монотонной, или преемственной эвристикой. Свойство монотонности означает, что если существуют пути A—B—C и A—C (не обязательно через B), то оценка стоимости пути от A до C должна быть меньше либо равна сумме оценок путей A—B и B—C. (Монотонность также известна как неравенство треугольника: одна сторона треугольника не может быть длиннее, чем сумма двух других сторон.) Математически, для всех путей x, y (где y — потомок x) выполняется:

$$g(x) + h(x) \leq g(y) + h(y) \qquad (6)$$

A* также оптимально эффективен для заданной эвристики h. Это значит, что любой другой алгоритм исследует не меньше узлов, чем A* (за исключением случаев, когда существует несколько частных решений с одинаковой эвристикой, точно соответствующей стоимости оптимального пути).

В то время как A* оптимален для «случайно» заданных графов, нет гарантии, что он сделает свою работу лучше, чем более простые, но и более информированные относительно проблемной области алгоритмы. Например, в неком лабиринте

может потребоваться сначала идти по направлению от выхода, и только потом повернуть назад. В этом случае обследование вначале тех вершин, которые расположены ближе к выходу (по прямой дистанции), будет потерей времени.

Существует несколько особенностей реализации и приёмов, которые могут значительно повлиять на эффективность алгоритма. Первое, на что не мешает обратить внимание — это то, как очередь с приоритетом обрабатывает связи между вершинами. Если вершины добавляются в неё так, что очередь работает по принципу LIFO, то в случае вершин с одинаковой оценкой A* «пойдёт» в глубину. Если же при добавлении вершин реализуется принцип FIFO, то для вершин с одинаковой оценкой алгоритм, напротив, будет реализовывать поиск в ширину. В некоторых случаях это обстоятельство может оказывать существенное значение на производительность.

В случае, если по окончании работы от алгоритма требуется маршрут, вместе с каждой вершиной обычно хранят ссылку на родительский узел. Эти ссылки позволяют реконструировать оптимальный маршрут. Если так, тогда важно, чтобы одна и та же вершина не встречалась в очереди дважды (имея при этом свой маршрут и свою оценку стоимости). Обычно для решения этой проблемы при добавлении вершины проверяют, нет ли записи о ней в очереди. Если она есть, то запись обновляют так, чтобы она соответствовала минимальной стоимости из двух. Для поиска вершины в сортирующем дереве многие стандартные алгоритмы требуют времени

$$O(n) \tag{7}$$

Если усовершенствовать дерево с помощью хеш-таблицы, то можно уменьшить это время.

Алгоритм A* и допустим, и обходит при этом минимальное количество вершин, благодаря тому, что он работает с «оптимистичной» оценкой пути через вершину. Оптимистичной в том смысле, что, если он пойдёт через эту вершину, у алгоритма «есть шанс», что реальная стоимость результата будет равна этой оценке,

но никак не меньше. Но, поскольку A* является информированным алгоритмом, такое равенство может быть вполне возможным.

Когда A* завершает поиск, он, согласно определению, нашёл путь, истинная стоимость которого меньше, чем оценка стоимости любого пути через любой открытый узел. Но поскольку эти оценки являются оптимистичными, соответствующие узлы можно без сомнений отбросить. Иначе говоря, A* никогда не упустит возможности минимизировать длину пути, и потому является допустимым.

Предположим теперь, что некий алгоритм B вернул в качестве результата путь, длина которого больше оценки стоимости пути через некоторую вершину. На основании эвристической информации, для алгоритма B нельзя исключить возможность, что этот путь имел и меньшую реальную длину, чем результат. Соответственно, пока алгоритм B просмотрел меньше вершин, чем A*, он не будет допустимым. Итак, A* проходит наименьшее количество вершин графа среди допустимых алгоритмов, использующих такую же точную (или менее точную) эвристику.

### 3.3.3 Алгоритм прыжковых точек

Алгоритм прыжковых точек (Jump Point Search) – этот алгоритм является улучшенным алгоритмом поиска пути A*. Алгоритм прыжковых точек ускоряет поиск пути, «перепрыгивая» места, которые должны быть просмотрены. В отличие от подобных алгоритмов, алгоритм прыжковых точек не требует предварительной обработки карты и имеет малый расход памяти.

Алгоритм работает на неориентированном графе единичной стоимости. Каждое поле карты имеет менее или 8 соседей, которые могут быть препятствием или нет. Каждый шаг по направлению стоит 1; шаг по диагонали стоит √2. Движения через препятствия не разрешены.

Запись:

$$y = x + kd \tag{8}$$

означает, что точка y может быть достигнута через k шагов из x в направлении d. Когда d – движение по диагонали, перемещение делится на два перемещения по прямой d1 и d2.

Путь:

$$p = (n0, n1, \ldots, nk) \quad (9)$$

упорядоченное перемещение по точкам без циклов из точки n0 до точки nk.

Обозначение p \ x означает, что точка x не встречается на пути p.

Обозначение len(p) означает длину или стоимость пути p.

Обозначение dist(x, y) означает длину или стоимость пути между точками x и y.

«Прыжковые точки» позволяют ускорить алгоритм поиска пути, рассматривая только «необходимые» точки. Такие точки могут быть описаны двумя простыми правилами выбора соседей при рекурсивном поиске: одно правило для прямолинейного движения и другое – для диагонального. В обоих случаях необходимо доказать, что исключая из набора ближайших соседей вокруг точки, найдётся оптимальный путь из предка текущей точки до каждого из соседей, и этот путь не будет содержать в себе посещенную точку. Рассмотрим случай 1, который отражает основную идею:

Случай 1: Отсечённый сосед, в соответствии с рисунком 7.

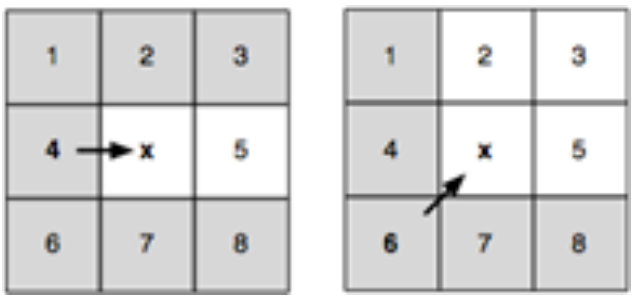



Рисунок 7

X – текущая рассматриваемая точка. Стрелка указывает направление движения. И там и там можно сразу отсечь соседей, выделенных серым, т.к. туда можно попасть по оптимальному пути из p(x), никогда не проходя через x.

Будем ссылаться на множество точек, которые остаются после отсечения настоящих соседей текущей точки. Они отмечены белыми на рисунке. В идеале, мы хотим учитывать только настоящих соседей во время просмотра. Тем не менее, в некоторых случаях, наличие препятствий может означать, что мы должны также рассмотреть небольшой набор до K дополнительных точек

$$(0 \leq K \leq 2) \tag{10}$$

Мы говорим, что это точки вынужденных соседей текущей позиции.

Случай 2: Принуждённый сосед, в соответствии с рисунком 8.

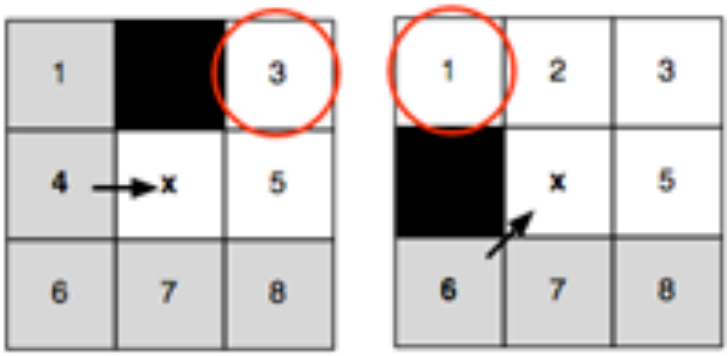



Рисунок 8

X – текущая рассматриваемая точка. Стрелка указывает направление движения. Обратите внимание, что, когда x находится рядом с препятствием, выделенные соседи не могут быть отсечены, любой альтернативный оптимальный путь от p(x) в каждом из этих узлов, блокируется.

Эти случаи применяются следующим образом: вместо создания «вынужденных» и «естественных» соседей мы рекурсивно отсекаем список соседей вокруг каждой точки. Таким образом наша цель заключается в ликвидации «симметрии», рекурсивно «перепрыгивая» через все точки, в которые можно попасть по оптимальному пути, который не проходил через текущую позицию. Рекурсия останав-

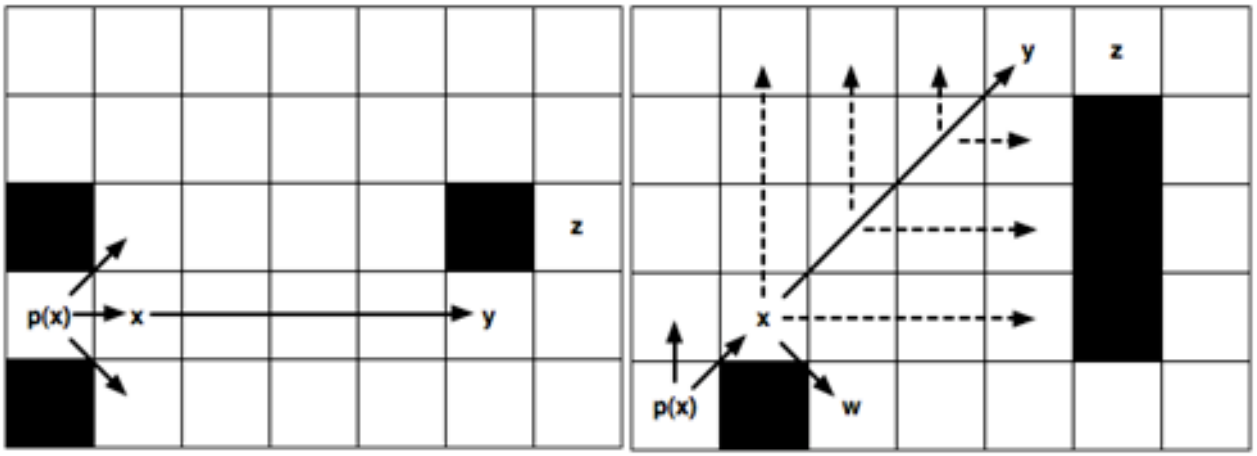
ливается при попадании на препятствие или нашли так называемую «прыжковую точку-преемник» (jump point successor). Прыжковые точки интересны тем, что они имеют соседей, которые не могут быть достигнуты альтернативным путём: оптимальный путь должен идти через текущую точку. Таким образом:

$$g(y) = g(x) + dist(x; y) \quad (11)$$

– стоимость перемещения.

Для обеспечения оптимальности необходимо только определиться как выбирать соседей (сначала линейные, затем диагональные).

Рассмотрим пример, в соответствии с рисунком 9:




а) б)

Рисунок 9

Здесь добавляется точка x для рассмотрения, предком которой является p(x); направление движение от p(x) к x является прямолинейное перемещение вправо.

В соответствии с рисунком 9 (а), рекурсивно применяем правило отсечки и получаем y в качестве преемника прыжковой точки x. Эта точка интересна тем, что есть сосед z, в который можно попасть по оптимальному пути только через y. Промежуточные точки не генерируются и не рассматриваются.

В соответствии с рисунком 9 (б), рекурсивно принимаем диагональные правила отсечки. Обратите внимание, что перед каждым следующим диагональным шагом необходимо рекурсивно пройтись по прямым линиям (выделены пункти-

ром). Только если обе «прямые» рекурсии не могут определить точку следующего прыжка, то перемещаемся дальше по диагонали. Точка w – вынужденный сосед x, создаётся как обычный.

Правила отсечения соседа

Далее опишем, каким образом отсекать множество точек, непосредственно примыкающих к некоторой точке x. Цель заключается в нахождении таких соседей, т.е. neighbours(x), до любых n точек которых нельзя достичь цель оптимально. Мы добиваемся этого путём сравнения двух путей: p, который начинается точкой p(x), посещает x и заканчивается с n и другим путём p’, который так же начинается с p(x), посещает x и заканчивается n, но не содержат x. Кроме того, каждая точка, содержащаяся в p или p’, должна относиться к neighbours(x).

Есть два случая, в зависимости от того, какой переход к x происходит из p(x): прямой ход или диагональный. Стоит учесть, что если x является началом p(x), то p(x) пусто и отсечение не происходит.

Отсекаются любые точки

$$n \in neighbours(x) \tag{12}$$

которые удовлетворяют следующему утверждению:

$$len((p(x), \ldots, n) \setminus x) \leq len(p(x), x, n) \tag{13}$$

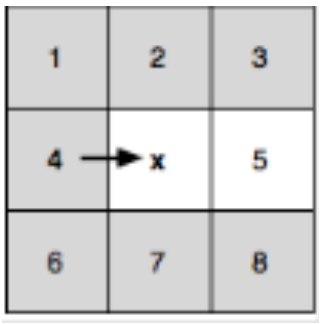



Рисунок 10

Здесь p(x) = 4 и мы отсекаем всех соседей кроме n = 5

Диагональные переходы:

Здесь отличия в том, что путь, который исключает x, должен быть строго доминирующим

$$len((p(x), \ldots, n) \setminus x) \leq len(p(x), x, n) \quad (14)$$

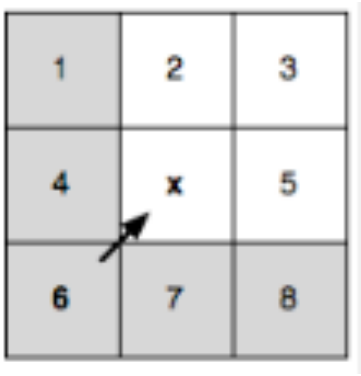



Рисунок 11

Здесь p(x) = 6 и отсекаются все соседи, кроме n = 2, n = 3 и n = 5.

Предполагая, что neighbours(x) не содержат препятствий, будем ссылаться на точки, которые остаются после прямой или диагональной отсечки (при необходимости), как естественные соседи x. Они соответствуют не серым точкам на а и б рисунках. Когда neighbours(x) содержат препятствия, нельзя отсечь всех не естественных соседей. В этом случае такой сосед считается принуждённым (искусственным).

Определение 1.

Точка $n \in neighbours(x)$ является принуждённой, если

1. $n$ не естественный сосед $x$

2. $len((p(x), \ldots, n) \setminus x) \leq len(p(x), x, n)$

В соответствии с рисунком 12, показан прямой переход, где n = 3 — принуждённый.

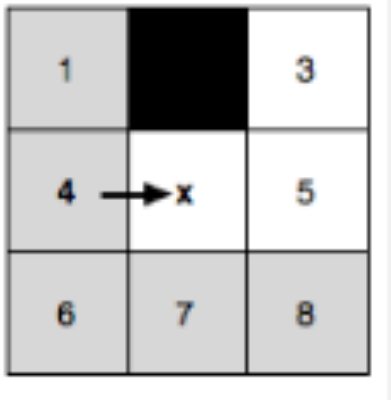



Рисунок 12

В соответствии с рисунком 13, показан пример диагонального перемещения; здесь n=1 – принуждённый сосед.

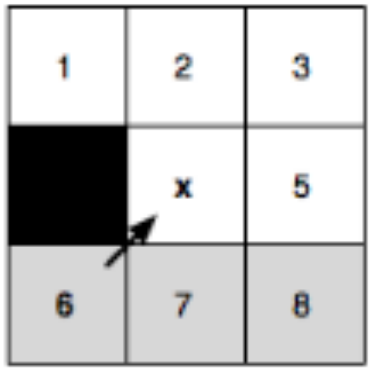



Рисунок 13

Введём определение:

Определение 2.

Точка y является точкой прыжка точки x, в направлении d, если y минимизирует значение k так, что

$$y = x + kd, \tag{15}$$

и выполняется одно из следующих условий:

1. Точка y – точка назначения.

2. У точки y есть хотя бы один сосед, который является принуждённым по определению 1.

3. d – движение по диагонали и существует точка:

$$z = y + kidi, \tag{16}$$

которая лежит в ki шагах в направлении

$$di \in \{d1, d2\}, \tag{17}$$

таких что z – точка прыжка из y при условии 1 или 2.

В соответствии с рисунком 14, показан пример точки прыжка, который определен условием 3. Здесь мы начинаем в точке x и заканчиваем движение по диагонали, пока не наткнёмся на точку y. Из y в точку z можно попасть с ki шагами по горизонтали. Таким образом, z является преемником точки для прыжка x (по условию 2), а это в свою очередь определяет y как преемник для прыжка точки x.

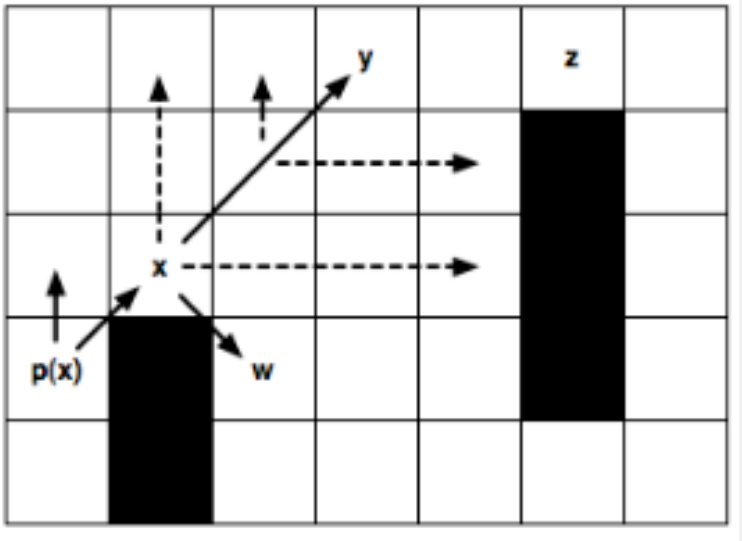



Рисунок 14

Алгоритм 1. Определение преемника.

Зададим: x – текущая точка, s – начало, g – цель

```
1.  successors(x) ← ∅
2.  neighbours(x) ← prune(x, neighbours(x))
3.  for all n ∈ neighbours(x) do
4.      n ← jump(x, direction(x, n), s, g)
5.      add n to successors(x)
6.  return successors(x)
```

Рисунок 15 – Алгоримт определение преемника

Алгоритм, в соответствии с рисунком 15, показывает как искать преемника для текущей точки. Сначала обрезается множество соседей, непосредственно примыкающих к текущей точке x (строка 2). Тогда вместо добавления каждого соседа n в множество successors (преемников) для x, попробуем «перепрыгнуть» к точке, которая находится дальше, но которая лежит относительно направлению x к n (строки 3-5). Например, если ребро (x; n) представляет собой движение по прямой вправо от x, то смотрим точку прыжка непосредственно справа от x. Если находится такая точка, то она добавляется в набор преемников вместо n. Если до точки прыжка дойти не получается, то ничего не добавляется. Процесс продолжается до тех пор, пока все соседи не закончатся, и затем алгоритм вернёт список всех преемников для x (строка 6).

Алгоритм 2. Функция прыжка.

Зададим: x – точка отчёта, d – направление, s – начало, g – цель

```
1.  n ← step(x, d)
2.  if n – препятствие или находится вне графа then
3.      return null
4.  if n = g then
5.      return n
6.  if ∃ n' ∈ neighbours(n) такие что n' – принуждённый then
7.      return n
8.  if d – диагональное then
9.      for all i ∈ {1,2} do
10.         if jump(n, di, s, g) не является null then
11.             return n
12. return jump(n, d, s, g)
```

Рисунок 16 – Алгоритм функция прыжка

Для того, чтобы найти отдельных преемников для точки прыжка, воспользуемся алгоритмом, в соответствии с рисунком 16. Он требует точки отчёта x, направление движения d, а так же начальную точку s и целевую точку g. Алгоритм пытается установить, имеет ли x точку для прыжка среди преемников, перемещаясь по направлению d (строка 1) и проверяет, удовлетворяет ли точка n Определению 2. В этом случае, n обозначается точкой прыжка и возвращается (строки 5, 7 и 11). Если n не является точкой прыжка, алгоритм рекурсивно повторяется и двигается снова в направлении d, но в этот раз n – новая точка отчёта (строка 12). Рекурсия прекращается, когда встречается препятствие и никакие дальнейшие действия не могут быть предприняты (строка 3). Стоит обратить внимание, что перед каждым диагональным шагом алгоритм должен обнаружить точки прыжка по прямым направлениям (строки 9-11). Эта проверка соответствует третьему условию Определения 2 и имеет важное значение для сохранения оптимальности алгоритма.

### 3.4 Тестирование алгоритмов поиска пути для навигации по зданию в мобильном приложении

Для теста использовалась карта, созданная по плану третьего этажа, главного корпуса АмГУ, в соответствии с рисунком 17.

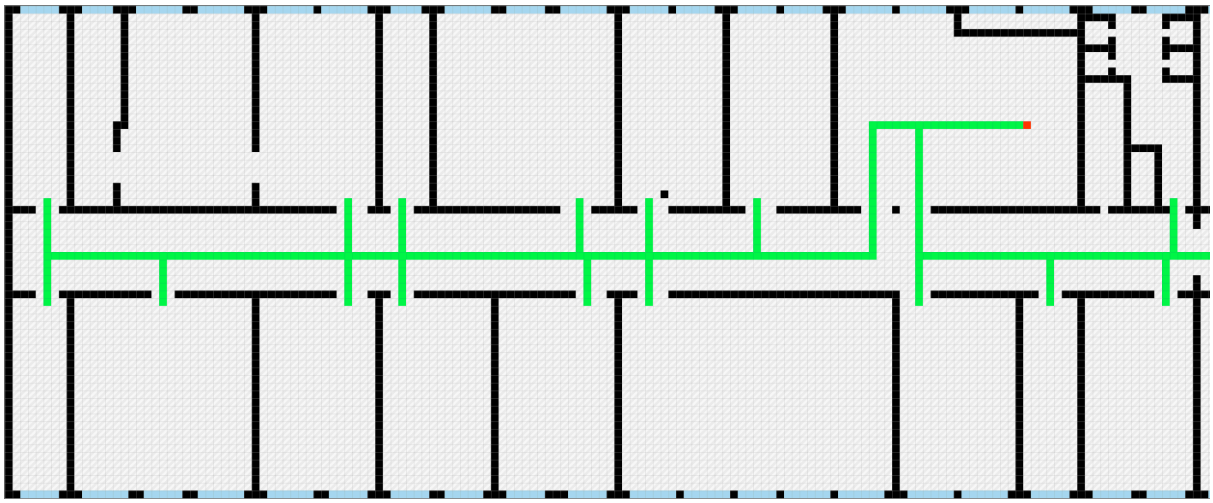


Рисунок 17 – Часть карты третьего этажа, главного корпуса АмГУ

Для поиска пути по зданию, было решено использовать в качестве карты, правильную 2D квадратную сетку, как более подходящую под текущую задачу.

При тестировании учитывались такие параметры, как:

1. Количество итераций.
2. Время расчёта пути.
3. Потребление памяти.

Все замеры были произведены на устройстве iPhone 5 с двух-ядерным процессором на 1,3 ГГц и оперативной памятью в 1 Гигабайт.

### 3.5 Результаты тестов

Замеры для не оптимизированной карты наглядно показаны на графике, в соответствии с рисунком 18 и в таблице 1.

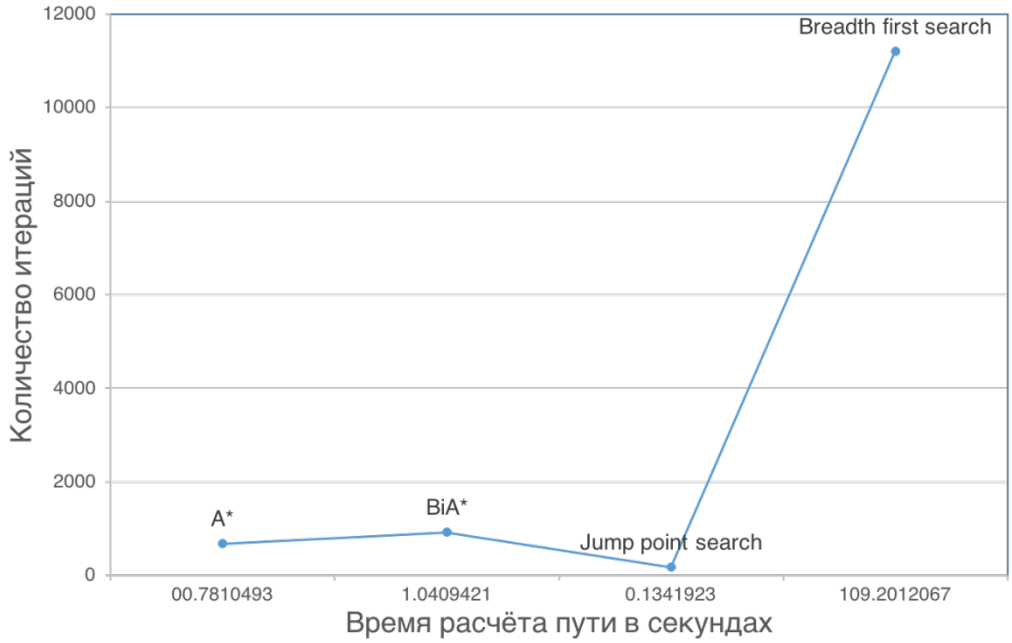



Рисунок 18 – График замеров для не оптимизированной карты

Таблица 1 – Результаты замеров для не оптимизированной карты

| Алгоритм | Время в секундах | Количество итераций |
|---|---|---|
| A* | 00.7810493 | 686 |
| BiA* | 01.0409421 | 927 |
| Jump point search | 00.1341923 | 182 |
| Breadth first search | 109.2012067 | 11203 |

Алгоритм поиска в ширину (Breadth first search) показал себя самым худшим, так как отработал за 109 секунд и выполнил 11203 итерации, что не позволительно для поиска пути в реальном времени, в свою очередь, другие алгоритмы показали себя с лучшей стороны. Алгоритм A* отработал за 0.78 секунды за 686 иттераций, двунаправленный A* отработал за 1.04 секунды, за 927 итераций. Но самым быстрым алгоритмом оказался Jump point search, он отработал за 0.13 сек, всего за 182 итерации.

Замеры для оптимизированной карты наглядно показаны на графике, в соответствии с рисунком 19 и в таблице 2.

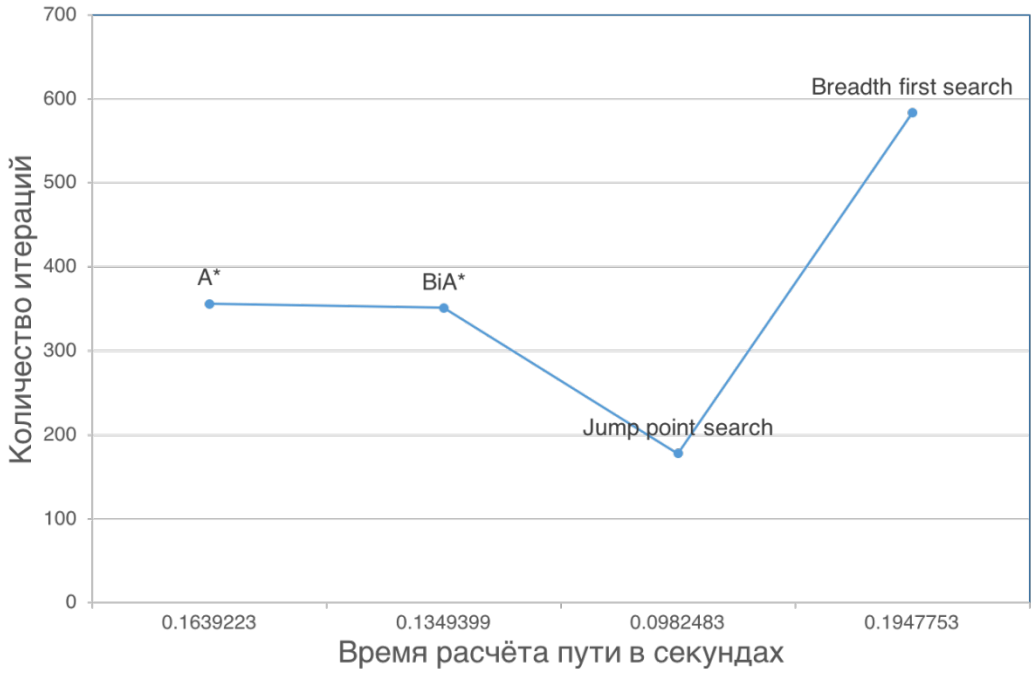



Рисунок 19 – График замеров для оптимизированной карты

Таблица 2 – Результаты замеров для оптимизированной карты

| Алгоритм | Время в секундах | Количество итераций |
|---|---|---|
| A* | 00.1639223 | 356 |
| BiA* | 00.1349399 | 351 |
| Jump point search | 00.0982483 | 178 |
| Breadth first search | 00.1947753 | 583 |

Алгоритм поиска в ширину (Breadth first search) показал себя самым лучше, чем в прошлый раз, переть он отработал за 0.19 секунд и выполнил 583 итерации. Алгоритм A* отработал за 0.16 секунды за 356 итераций, двунаправленный A* от-

работал за 0.13 секунды, за 351 итерацию. Но самым быстрым алгоритмом снова оказался Jump point search, он отработал за 0.1 сек, всего за 178 итерации.

Таким образом, самым лучшим алгоритмом для поиска путей на правильной 2D квадратной сетке для мобильного устройства, с минимальным временем расчёта пути, стал алгоритм «прыжковых точек» (jump point search).

### 3.6 Обоснование выбора среды разработки

Была выбрана среда разработки Xamarin Studio, так как она является кросс-платформенной средой для разработки на языке C# и в неё встроена полная поддержка Xamarin Framework, которая необходима для разработки кросс-платформенных мобильных приложений.

# 4 Программная реализация поиска пути внутри зданий для мобильных устройств

Для возможности создания карт зданий, мной были изучены существующие решения, которые не смогли подойти к требованиям и было решено реализовать свой редактор на основе сетки пикселей, в соответствии с рисунком 20. Но так как каждый пиксель карты будет занимать в памяти место, то такой метод не годится для мобильных устройств, из за малого объёма оперативной памяти, поэтому было найдено другое решение.

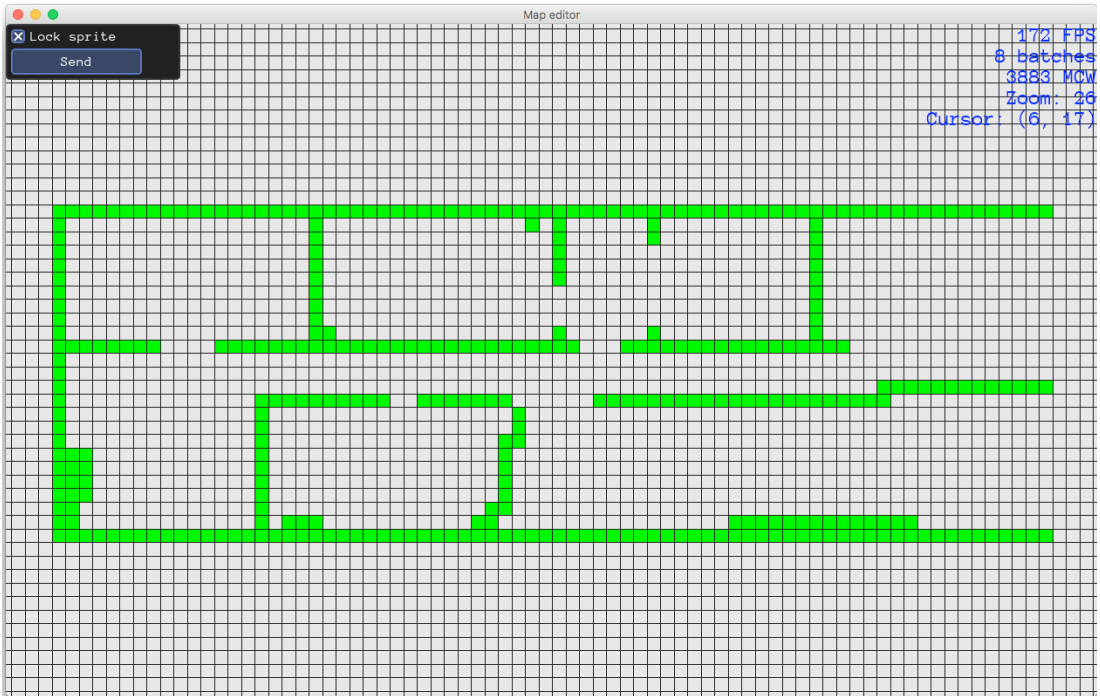



Рисунок 20 – Редактор карт на основе квадратной сетки пикселей

В данный момент, для создания карт зданий, используется векторная графика, в соответствии с рисунком 21, так как данные векторов это точки начала и конца вектора, то они занимают меньше места в памяти устройства и с ними можно легко производить расчёты пути. Так же, вектора занимают мало оперативной памяти, что очень важно на мобильных устройствах, а преобразования и расчёты не сильно нагружают процессор.

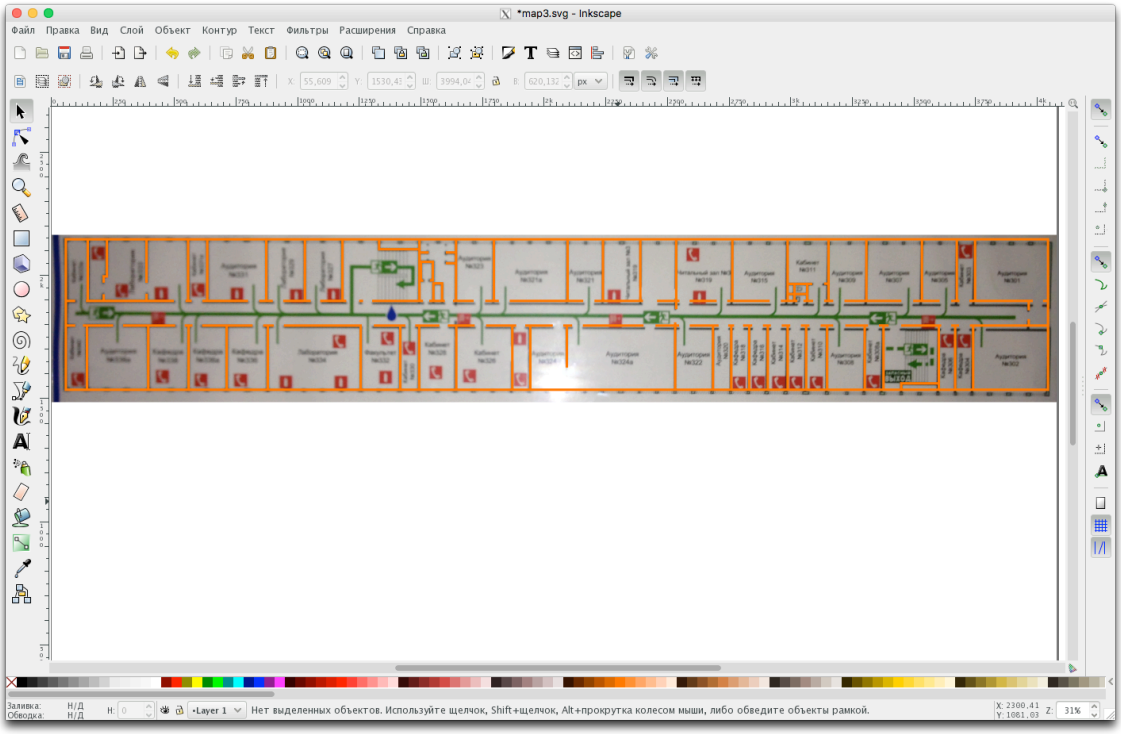



Рисунок 21 – Векторный редактор Inkscape

Мобильное приложение работает согласно алгоритму, в соответствии с рисунком 22. После запуска приложения, происходит считывание загруженных карт в памяти устройства. Пользователь выбирает необходимую ему карту и приложение строит визуализацию карты. Далее пользователь может выбрать точки для прокладываемого маршрута, который будет построен и визуализирован на карте.

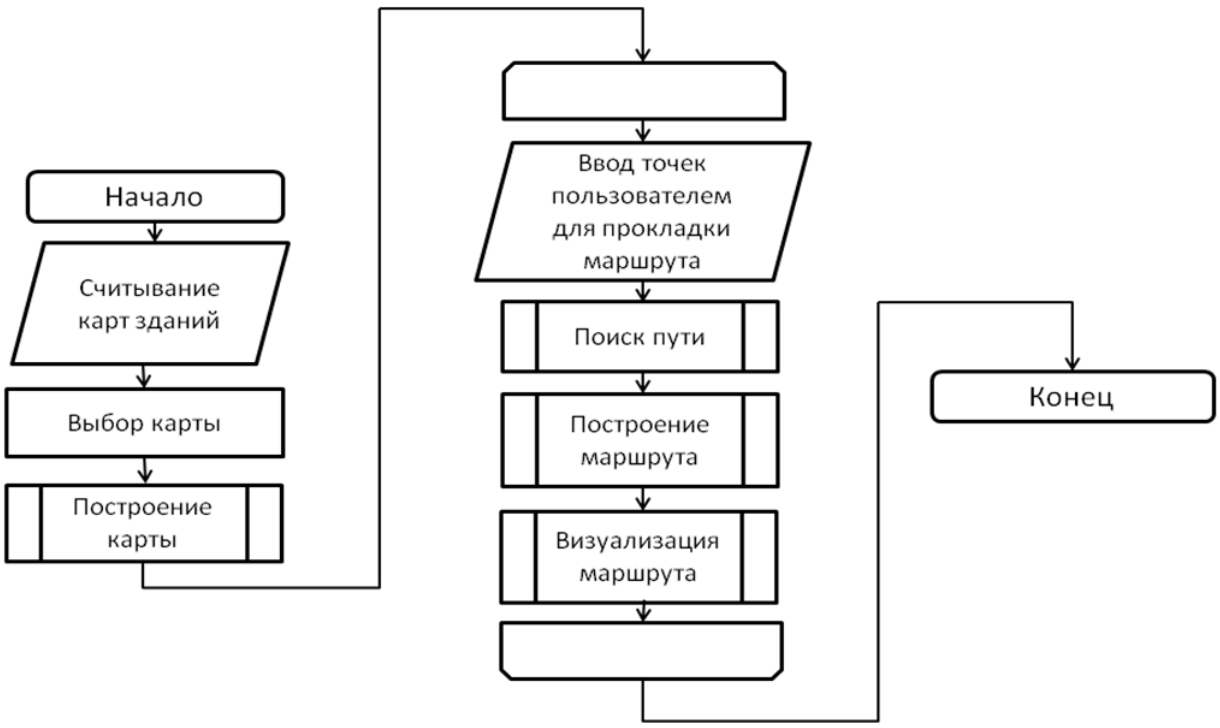



Рисунок 22 – Алгоритм работы программы

Навигация происходит при помощи алгоритма прыжковых точек (Jump point search), строя путь, между указанными пользователем точками в здании, в соответствии с рисунком 23.


Рисунок 23 – Скриншоты мобильного приложения

В приложении, можно выбрать кабинет из списка кабинетов или воспользоваться поиском, который поможет найти кабинет, не зная его точного названия и местоположения.

Для переключения между этажами используются кнопки сбоку экрана, в соответствии с рисунком 23.

В памяти устройства не хранится вся карта целиком, хранятся только граничные точки векторов. Это необходимо для того, чтобы приложение нормально функционировало в условиях недостатка оперативной памяти.

# ЗАКЛЮЧЕНИЕ

Были проведены аналитические работы по изучению и обзору кроссплатформенных решений для разработки мобильных приложений, исследованию различных алгоритмов поиска путей, анализа существующих решений и их востребованности, разработки своего решения для навигации внутри зданий для мобильных устройств.

Использование кроссплатформенного фреймворка Xamarin помогло создать мобильное приложение сразу под несколько различных мобильных операционных систем.

В текущий момент, решений для навигации внутри зданий очень мало, но проблема актуальна, в основном это коммерческие продукты, которые очень дорого стоят и не имеют открытого исходного кода, поэтому было решено, сделать открытый программный продукт.

Программное решение разработано с учётом работы на мобильных устройствах разной мощности и учитывает их малую мощность и малое количество оперативной памяти, что позволяет ему работать в среде с ограниченными аппаратными ресурсами.

Разработанное решение, поможет легко добавлять собственные карты зданий и при необходимости изменять функционал программного обеспечения, под нужды навигации в конкретном помещении.

Данный проект будет распространяться свободно и с открытым исходным кодом.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. Шепель, В. И. Сравнительный анализ глобальных навигационных спутниковых систем / В. И. Шепель, Д. С. Ергалиев, А. Д. Тулегулов // Труды Международного симпозиума «Надежность и качество». – 2012. – 1 т.

2. Официальный сайт Xamarin [Электрон. ресурс]. - Режим доступа: https://xamarin.com - 09.03.2016

3. Сайт разработчиков для устройств Apple [Электрон. ресурс]. - Режим доступа: https://developer.apple.com/ - 09.03.2016.

4. Сайт разработчиков для устройств Android [Электрон. ресурс]. - Режим доступа: http://developer.android.com/ - 09.03.2016.

5. Хабрахабр : Обзор 7 самых популярных кроссплатформенных мобильных фреймворков [Электронный ресурс]. - Режим доступа: https://habrahabr.ru/post/229559/. - 09.03.2016.

6. Стертевант, Н. Р. Сравнение на высоком уровне подходов для ускорения поиска пути / Н. Р. Стертевант, Р. Гайсбергэ. // Труды 6-го AAAI конференции по искусственному интеллекту и интерактивным цифровым развлечениям (AIIDE '10). – 2010. – 76 с.

7. Коливанд, Х. Обзор алгоритмов теневого объема в компьютерной графике / Х. Коливанд, М. С. Сунар. // IETE Technical Review. – 2010. – 30 т – № 1. – С. 38-46.

8. Ван Ден Берг, Й. ANA *: всегда непараметрический А / Й. Ван Ден Берг [и др.]. // Труды 25-й AAAI конференции по искусственному интеллекту (AAAI '11) – 2011. – С. 105-111.

9. Бонет, Б. Планирование в виде эвристического поиска / Б. Бонет, Х. Гефнер. // Искусственный интеллект. – 2001. – 129 т. – № 1-2. – С. 5-33.

10. Гельмерт, М. Понимание задач планирования: домен, сложность и эвристическая декомпозиция // Т. 4929 конспектов лекций по информатике, Берлин, Германия. – 2008.

11. Харт, П. Е. Формальное основание для эвристического определения минимальной стоимости пути // IEEE Transactions по системам математики и кибернетики. – 1968. – 4 т. – № 2. – С. 100-107.

12. Харабор, Д. Обрезка графа для поиска пути на сетке // Труды 25-й Национальной конференции по искусственному интеллекту (AAAI '11), Сан-Франциско, штат Калифорния, США. – 2011.

13. Стертевант, Н. Р. Критерии для поиска пути на основе сетки // IEEE Transactions по вычислительному интеллекту и искусственному интеллекту в играх. – 2012. – 4 т. – № 2. – С. 144-148.

14. Урас, Т. Подцель графа для оптимального поиска пути с восемью соседями на сетке // Труды 23-й Международной конференции по автоматизированному планирования и составлению расписаний (ICAPS '13), Рим, Италия. – 2013.

15. Харабор, Д. Улучшение jump point search // Труды 24-й Международной конференции по вопросам автоматизированного планирования и составления расписаний. – 2014.

16. Бная, З. Многоагентный поиск пути для самостоятельного заинтересованных агентов // Труды 6-го ежегодного симпозиума по комбинаторному поиску. – 2013. – 38-46 с.

17. Андерсон, К. Добавление эвристики для соединенных четырех сеток // Труды 3-го ежегодного симпозиума по комбинаторному поиску. – 2010.

18. Джин, Х. Многоагентная система поиска пути реализованная на XNA // Труды 4-й Международной конференции по вычислительному интеллекту и коммуникационным сетям (CICN '12), Матхура, Индия. – 2011. – 651-655 с.

19. Бьорнссон, Ю. Сравнение различных абстрактных сеток для поиска пути на картах // Труды 18-й Международной совместной конференции по искусственному интеллекту (IJCAI '03), Акапулько, Мексика. – 2003. – 1511-1512 с.

20. Кихано, Х. Ж. Совершенствование объединения робота с использованием шестиугольного мирового представления // В трудах конференции по электронике, робототехнике и автомеханике (CERMA '07). – 2007. – 450-455 с.

21. Осман, М. Ф. Моделирование динамического планирования пути в режиме реального времени, базовое зрение роботов // Интеллектуальные робототехнические системы: Вдохновляя в БУДУЩЕЕ. – 2013.– 1-10 с.

22. Димайн, Д. Эффективный поиск пути на базе триангуляции // 21-я национальная конференция по искусственному интеллекту и 18 конференция по новаторскому использованию искусственного интеллекта. – 2006. – 942-947 с.

23. Наги, Б. Клеточная Топология и топологические системы координат на гексагональных и на треугольных сетках // Ежегодник по математике и искусственному интеллекту.– 2014. – 1-18 с.

24. Кылыч, В. Активная волна вычислений на основе поиска пути, подход к 3-D среде // IEEE Международный симпозиум схем и систем. – 2011. – 2165-2168 с.

25. Нэш, А. Алгоритм «Any-angle path planning» // Журнал по искусственному интеллекту. – 2013. – 34 т. – № 4. – 9 с.

26. Лозано-Переса, Т. Алгоритм для планирования пути без столкновений между многогранными препятствиями // ACM, – 1979. – 22 т. – 560-570 с.

27. Надэан-Тахан, М. Эффективное и безопасное планирование пути для мобильного робота с использованием генетического алгоритма // IEEE конгресс по эволюционным вычислениям (CEC '09). – 2009. – 2091-2097 с.

28. Сислак, Д. Планирование траектории полета // Семинар планирования и применения планирования, – 2009. – 76-83 с.

29. Кападиа, М. Мульти-доменное планирование в режиме реального времени в динамических средах // 12 ACM SIGGRAPH Еврографический симпозиум по компьютерной анимации (SCA '13). – 2013. – 115-124 с.

30. Нидерберг, С. Универсальное планирования пути для приложений реального времени // Интернациональная компьютерная графика (CGI '04). IEEE, Крит, Греция. – 2004. – 299-306 с.

31. Фергюсон, Д. Использование интерполяции для улучшения планирования маршрутов: алгоритм Field D* // Журнал полевой робототехники. –2006. – 23 т. – 79-101 с.

32. Бурчард, Х. Реализация планирования пути с использованием генетических алгоритмов на мобильных роботах // IEEE конгресс по эволюционным вычислениям (CEC '06). – 2006. – 1831-1836 с.

33. Коултер, Р. С. Реализация «pure pursuit path» алгоритма // DTIC документ. – 1992.

34. Кай, Гое, С. Моделирование, серьезные игры и приложения. – 2014.

35. Чой, М. Г Планирование двуногого путешествия с использованием данных захвата движения и вероятностных дорожных карт // ACM труды на графах. – 2003. – 22 т. – № 2. – 182-203 с.

36. Фурсов, В.В. Обзор инструментов кроссплатформенной разработки для мобильных устройств / В.В. Фурсов, С.Г. Самохвалова // Молодёжь XXI века: шаг в будущее : материалы XVII региональной научно-практической конференции (24 мая 2016 года) в 4 т. – Благовещенск: Изд-во БГПУ – 2016.

37. Фурсов, В.В. Обзор алгоритмов поиска пути для мобильного приложения / В.В. Фурсов, С.Г. Самохвалова // Приоритетные научные направления: от теории к практике : сборник материалов XXXVI Международной научно-практической конференции – Новосибирск, 28 февраля 2017 г.

38. Фурсов, В.В. Использование алгоритма поиска пути для навигации по зданию в мобильном приложении / В.В. Фурсов, С.Г. Самохвалова // Вестник Амурского государственного университета: научно-теоретический журнал – Благовещенск – 2017г. – N 77.

ПРИЛОЖЕНИЕ А

Листинг реализации алгоритма поиска в ширину

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace PathFinders.Finders {
    public class BreadthFirstFinder : IFinder {
        private FinderOptions _options;
        private List<Node> _closedSet;

        public List<Node> ClosedSet {
            get {
                return _closedSet;
            }
        }

        public BreadthFirstFinder(FinderOptions opt = null) {
            if (opt == null) {
                opt = new FinderOptions();
            }

            _options = opt;

            if (_options.DiagonalMovement != Grid.DiagonalMovement.Always) {
                if (!_options.AllowDiagonal) {
                    _options.DiagonalMovement = Grid.DiagonalMovement.Never;
                } else {
                    if (_options.DontCrossCorners) {
                        _options.DiagonalMovement = Grid.DiagonalMovement.OnlyWhenNoO
bstacles;
                    } else {
                        _options.DiagonalMovement = Grid.DiagonalMovement.IfAtMostOneOb
stacle;
                    }
                }
            }

            if (_options.DiagonalMovement == Grid.DiagonalMovement.Never) {
                _options.Heuristic = opt.Heuristic != Heuristic.Type.Null ? opt.Heuristic : Heu
ristic.Type.Manhattan;
            } else {
                _options.Heuristic = opt.Heuristic != Heuristic.Type.Null ? opt.Heuristic : Heu
ristic.Type.Octile;
```

```
        }
    }

    public List<Node> FindPath(int startX, int startY, int endX, int endY, Grid grid) {
        _closedSet = new List<Node>();
        List<Node> openSet = new List<Node>();
        Node startNode = grid.GetNodeAt(startX, startY);
        Node endNode = grid.GetNodeAt(endX, endY);

        openSet.Add(startNode);

        while (openSet.Count > 0) {
            // shift
            var currentNode = openSet.First();
            openSet.Remove(currentNode);
            //
            _closedSet.Add(currentNode);

            if (currentNode == endNode) {
                return Util.Backtrace(currentNode);
            }

            var neighbours = grid.GetNeighbours(currentNode, _options.DiagonalMoveme
nt);
            foreach (var neighbourNode in neighbours) {

                if (_closedSet.Count(node => node == neighbourNode) > 0 || openSet.Count
(node => node == neighbourNode) > 0) {
                    continue;
                }

                neighbourNode.Parent = currentNode;
                openSet.Add(neighbourNode);
            }
        }
        return new List<Node>();
    }
  }
}
```

ПРИЛОЖЕНИЕ Б

Листинг реализации алгоритма A*

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace PathFinders.Finders {
    public class AStarFinder : IFinder {
        private FinderOptions _options;

        private List<Node> _closedSet;

        public List<Node> ClosedSet {
            get {
                return _closedSet;
            }
        }

        public AStarFinder(FinderOptions opt = null) {
            if (opt == null) {
                opt = new FinderOptions();
            }

            _options = opt;

            if (_options.DiagonalMovement != Grid.DiagonalMovement.Always) {
                if (!_options.AllowDiagonal) {
                    _options.DiagonalMovement = Grid.DiagonalMovement.Never;
                } else {
                    if (_options.DontCrossCorners) {
                        _options.DiagonalMovement = Grid.DiagonalMovement.OnlyWhenNoO
bstacles;
                    } else {
                        _options.DiagonalMovement = Grid.DiagonalMovement.IfAtMostOneOb
stacle;
                    }
                }
            }

            if (_options.DiagonalMovement == Grid.DiagonalMovement.Never) {
                _options.Heuristic = opt.Heuristic != Heuristic.Type.Null ? opt.Heuristic : Heu
ristic.Type.Manhattan;
            } else {
```

```
            _options.Heuristic = opt.Heuristic != Heuristic.Type.Null ? opt.Heuristic : Heu
ristic.Type.Octile;
        }
    }

    public List<Node> FindPath(int startX, int startY, int endX, int endY, Grid grid) {
        _closedSet = new List<Node>();
        List<Node> openSet = new List<Node>();
        Node startNode = grid.GetNodeAt(startX, startY);
        Node endNode = grid.GetNodeAt(endX, endY);

        openSet.Add(startNode);

        while (openSet.Count > 0) {
            var currentNode = openSet.OrderBy(node =>
              node.EstimateFullPathLength).First();
            if (currentNode == endNode)
                return Util.Backtrace(currentNode);
            openSet.Remove(currentNode);
            _closedSet.Add(currentNode);
            var neighbours = grid.GetNeighbours(currentNode, _options.DiagonalMoveme
nt);
            foreach (var neighbourNode in neighbours) {
                var ng = currentNode.PathLengthFromStart + ((neighbourNode.X -
 currentNode.X == 0 || neighbourNode.Y - currentNode.Y == 0) ? 1 : Math.Sqrt(2));
                var h = _options.Weight * Heuristic.Exec(_options.Heuristic, Math.Abs(nei
ghbourNode.X - endX), Math.Abs(neighbourNode.Y - endY));

                var newNode = new Node() {
                    X = neighbourNode.X,
                    Y = neighbourNode.Y,
                    Parent = currentNode,
                    PathLengthFromStart = (float)ng,
                    HeuristicEstimatePathLength = h
                };

                if (_closedSet.Count(node => node == newNode) > 0) {
                    continue;
                }

                var openNode = openSet.FirstOrDefault(node =>
                  node == newNode);

                if (openNode == null) {
```

```
                openSet.Add(newNode);
            } else {
                if (openNode.PathLengthFromStart > newNode.PathLengthFromStart) {
                    openNode.Parent = currentNode;
                    openNode.PathLengthFromStart = newNode.PathLengthFromStart;
                }
            }
        }
    }

    return new List<Node>();
  }
 }
}
```

ПРИЛОЖЕНИЕ В

Листинг реализации алгоритма Jump point search

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace PathFinders.Finders {
    public class JumpPointFinderBase : IFinder {
        protected List<Node> _closedSet;
        protected List<Node> _openList;

        FinderOptions _options;
        protected Grid _grid;
        protected Node _startNode;
        protected Node _endNode;
        protected bool _trackJumpRecursion = true;

        public List<Node> ClosedSet {
            get {
                return _closedSet;
            }
        }

        public List<Node> OpenedSet
        {
            get
            {
                return _openList;
            }
        }

        public List<Node> TestedList = new List<Node>();

        public JumpPointFinderBase(FinderOptions opt) {
            if (opt == null) {
                opt = new FinderOptions();
            }

            _options = opt;
            _options.Heuristic = opt.Heuristic != Heuristic.Type.Null ? opt.Heuristic : Heuris
tic.Type.Manhattan;
        }

        public List<Node> FindPath(int startX, int startY, int endX, int endY, Grid grid) {
```

```
            _closedSet = new List<Node>();
            _openList = new List<Node>();
            _startNode = grid.GetNodeAt(startX, startY);
            _endNode = grid.GetNodeAt(endX, endY);
            _grid = grid;

            _openList.Add(_startNode);

            while (_openList.Count > 0) {
                var currentNode = _openList.OrderBy(node =>
                  node.EstimateFullPathLength).First();

                if (currentNode == _endNode)
                    return Util.ExpandPath(Util.Backtrace(currentNode));

                _openList.Remove(currentNode);
                _closedSet.Add(currentNode);

                IdentifySuccessors(currentNode);
            }
            return new List<Node>();
        }

        protected void IdentifySuccessors(Node node) {
            var neighbors = FindNeighbors(node);


            foreach (var neighbor in neighbors) {
                var jumpNode = Jump(neighbor.X, neighbor.Y, node.X, node.Y);
                if (jumpNode != null) {
                    var d = Heuristic.Octile(Math.Abs(jumpNode.X -
 node.X), Math.Abs(jumpNode.Y - node.Y));
                    var ng = node.PathLengthFromStart + d; // next `g` value
                    var h = _options.Weight * Heuristic.Exec(_options.Heuristic, Math.Abs(jum
pNode.X - _endNode.X), Math.Abs(jumpNode.Y - _endNode.Y));


                    var newNode = new Node() {
                        X = jumpNode.X,
                        Y = jumpNode.Y,
                        Parent = node,
                        PathLengthFromStart = (float)ng,
                        HeuristicEstimatePathLength = h
                    };
```

```
                if (_closedSet.Count(n => n == newNode) > 0) {
                    continue;
                }

                var openNode = _openList.FirstOrDefault(n =>
                  n == newNode);

                if (openNode == null) {
                    _openList.Add(newNode);
                } else {
                    if (openNode.PathLengthFromStart > newNode.PathLengthFromStart) {
                        openNode.Parent = node;
                        openNode.PathLengthFromStart = newNode.PathLengthFromStart;
                    }
                }
            }
        }
    }

    public virtual Node Jump(int x, int y, int px, int py) {
        throw new NotImplementedException();
    }

    public virtual List<Node> FindNeighbors(Node node) {
        throw new NotImplementedException();
    }
  }
}


using System;
using System.Collections.Generic;

namespace PathFinders.Finders {
  public class JPFNeverMoveDiagonally : JumpPointFinderBase, IFinder {
    public JPFNeverMoveDiagonally(FinderOptions opt) : base(opt) {
    }


    public override Node Jump(int x, int y, int px, int py) {
        var dx = x - px;
        var dy = y - py;
```

```
        Grid grid = _grid;

        if (!grid.IsWalkableAt(x, y)) {
            return null;
        }

        if (_trackJumpRecursion) {
            TestedList.Add(_grid.GetNodeAt(x, y));
        }

        if (grid.GetNodeAt(x, y) == _endNode) {
            return _endNode;
        }

        if (dx != 0) {
            if ((grid.IsWalkableAt(x, y - 1) && !grid.IsWalkableAt(x - dx, y - 1)) ||
                (grid.IsWalkableAt(x, y + 1) && !grid.IsWalkableAt(x - dx, y + 1))) {
                return new Node(x, y);
            }
        } else if (dy != 0) {
            if ((grid.IsWalkableAt(x - 1, y) && !grid.IsWalkableAt(x - 1, y - dy)) ||
                (grid.IsWalkableAt(x + 1, y) && !grid.IsWalkableAt(x + 1, y - dy))) {
                return new Node(x, y);
            }
            if (Jump(x + 1, y, x, y) != null || Jump(x - 1, y, x, y) != null) {
                return new Node(x, y);
            }
        } else {
            throw new Exception("Only horizontal and vertical movements are allowed");
        }

        return Jump(x + dx, y + dy, x, y);
    }

    public override List<Node> FindNeighbors(Node node) {
        var parent = node.Parent;
        Grid grid = _grid;
        var x = node.X;
        var y = node.Y;

        var neighbors = new List<Node>();

        if (parent != null) {
            var px = parent.X;
```

```
            var py = parent.Y;
            var dx = (x - px) / Math.Max(Math.Abs(x - px), 1);
            var dy = (y - py) / Math.Max(Math.Abs(y - py), 1);

            if (dx != 0) {
                if (grid.IsWalkableAt(x, y - 1)) {
                    neighbors.Add(grid.GetNodeAt(x, y - 1));
                }
                if (grid.IsWalkableAt(x, y + 1)) {
                    neighbors.Add(grid.GetNodeAt(x, y + 1));
                }
                if (grid.IsWalkableAt(x + dx, y)) {
                    neighbors.Add(grid.GetNodeAt(x + dx, y));
                }
            } else if (dy != 0) {
                if (grid.IsWalkableAt(x - 1, y)) {
                    neighbors.Add(grid.GetNodeAt(x - 1, y));
                }
                if (grid.IsWalkableAt(x + 1, y)) {
                    neighbors.Add(grid.GetNodeAt(x + 1, y));
                }
                if (grid.IsWalkableAt(x, y + dy)) {
                    neighbors.Add(grid.GetNodeAt(x, y + dy));
                }
            }
        }
        else {
            var neighborNodes = grid.GetNeighbours(node, Grid.DiagonalMovement.Nev
er);
            foreach (var neighborNode in neighborNodes) {
                neighbors.Add(grid.GetNodeAt(neighborNode.X, neighborNode.Y));
            }
        }

        return neighbors;

    }

  }
}
```

```
using System;
using System.Collections.Generic;

namespace PathFinders.Finders {
    public class JumpPointFinder : IFinder {
        private FinderOptions _options;
        JumpPointFinderBase currentFinder;

        public List<Node> TestedList;

        private List<Node> _closedSet;
        public List<Node> ClosedSet {
            get {
                return _closedSet;
            }
        }

        public JumpPointFinder(FinderOptions opt = null) {
            if (opt == null) {
                opt = new FinderOptions();
            }

            _options = opt;
            if (_options.DiagonalMovement == Grid.DiagonalMovement.Never) {
                currentFinder = new JPFNeverMoveDiagonally(_options);
            } else if (_options.DiagonalMovement == Grid.DiagonalMovement.Always) {
                throw new NotImplementedException();
            } else if (_options.DiagonalMovement == Grid.DiagonalMovement.OnlyWhenN
oObstacles) {
                throw new NotImplementedException();
            } else {
                throw new NotImplementedException();
            }
        }

        public List<Node> FindPath(int startX, int startY, int endX, int endY, Grid grid) {
            var find = currentFinder.FindPath(startX, startY, endX, endY, grid);
            _closedSet = currentFinder.ClosedSet;
            TestedList = currentFinder.TestedList;
            return find;
        }
    }
}
```