

Министерство цифрового развития, связи и массовых коммуникаций
Российской Федерации
Ордена Трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Разрешаю
допустить к защите
Зав. кафедрой

д.т.н., профессор В.А. Докучаев
24 мая 2025 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

НА ТЕМУ

«Исследование и разработка алгоритма поиска пути для большого
пространства с использованием параллельного вычисления»

Студент: Удалов Игорь Дмитриевич _____

Руководитель: Докучаев Владимир Анатольевич _____

Рецензент: Олейник Александр Иванович _____

Москва, 2025

Министерство цифрового развития связи и массовых коммуникаций
Ордена Трудового Красного Знамени федеральное государственное
бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
СВЯЗИ И ИНФОРМАТИКИ»
(МТУСИ)

МАГИСТРАТУРА

УТВЕРЖДАЮ
Декан факультета ИТ
_____ М.Г. Городничев
протокол заседания совета факультета ИТ
№ _____ от « _____ » _____ 2023 г.

1. Индивидуальный план работы

Удалов Игорь Дмитриевич

2. Факультет: Информационные технологии
3. Кафедра: Сетевые информационные технологии и сервисы
4. Научный руководитель _____ д.т.н., профессор Докучаев В. А.
5. Период обучения в магистратуре - 01.09.2023 г. – 30.08.2025 г.
6. Направление - 09.04.01 «Информатика и вычислительная техника»
7. Наименование профиля - «Распределённые информационные системы и приложения»
8. Тема магистерской диссертации _____ Исследование и разработка
алгоритма поиска пути для большого пространства с использованием
параллельного вычисления
9. Срок представления студентом диссертации – Июнь 2025 г.
10. Содержание программы подготовки магистра:

№ п/п	Наименование дисциплин, практик	Количество ЗЕ	Формы аттестации	Планируемый срок аттестации	Отметки руководителя об этапах текущей аттестации

1.	Наименование учебных дисциплин определяется учебным планом направления, утвержденным ректором МТУСИ				
2.	Научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)	8	отчет	1 семестр	
3.	Проектно-технологическая практика	3	отчет	3 семестр	
4.	Производственная практика (научно-исследовательская работа)	22	отчет	2,3,4 семестр	
5.	Преддипломная практика	3	отчет	4 семестр	
6.	Индивидуальная работа: 1. Исследование алгоритма поиска пути 2. Разработка алгоритма поиска пути 3. Оптимизация алгоритма с использованием параллельного вычисления 4. Тестирование алгоритма		реферат реферат реферат реферат	Декабрь 2023 Май 2024 Ноябрь 2024 Март 2025	
7.	Публикации		статьи	ежегодно	
8.	Подготовка магистерской диссертации	6		Июнь 2025	

(подпись студента)

(подпись научного руководителя)

(подпись зав. кафедрой)

ОТЗЫВ РУКОВОДИТЕЛЯ

на магистерскую диссертацию Удалова Игоря Дмитриевича
гр. М092301(75)

на тему: **«Исследование и разработка алгоритма поиска пути для большого пространства с использованием параллельного вычисления»**,

представленную на соискание степени магистра по направлению подготовки

09.04.01 – «Информатика и вычислительная техника»

профиль подготовки: «Распределённые информационные системы и приложения»

«Характеристика работы». Возрастающая сложность виртуальных миров и масштабы пространств в современных компьютерных играх приводят к необходимости совершенствования существующих и разработке новых алгоритмов поиска пути с применением параллельных вычислений. Проведённый в работе анализ отличает полнота охвата, глубина исследования и критический обзор литературы. Поэтому магистерская диссертация студента Удалов И. Д. является актуальной.

«Оценка профессиональных качеств выпускника, выявленных в ходе работы над ВКРМ». За время работы над магистерской диссертацией студент Удалов И. Д. овладел всеми требуемыми компетенциями.

«Общее описание ВКРМ». Цель исследования заключается в исследовании способов повышения эффективности нахождения пути в сложных игровых мирах.

В первой главе рассмотрены алгоритмы поиска пути, используемые в компьютерных играх.

Во второй главе анализируются подходы, методы параллельного вычисления, а также среды разработки.

В третьей главе приведены иерархический поиск пути и его модификации для динамических и статических игровых миров.

В четвёртой главе приводится описание и оценка разработанного алгоритма поиска пути.

В заключении сформулированы результаты выпускной квалификационной работы магистра.

«Оценка руководителем студента и ВКРМ в целом». За время работы над диссертацией студент Удалов И. Д. проявил себя грамотным, целеустремленным, самостоятельным специалистом, который может квалифицированно и на профессиональном уровне решать сложные задачи. При выполнении работы отступлений от графика работы не было, поставленные цели достигнуты. Ряд результатов, полученных в магистерской работе, опубликован в трудах российских и международных научно-технических конференций, а также в изданиях, включённых в РИНЦ.

Считаю, что студент Удалов И. Д. в процессе написания магистерской диссертации показал высокий уровень стандартных компетенций и полностью справился с решением поставленных в магистерской диссертации задач. Студент Удалов И. Д. заслуживает присвоения ему степени магистра по направлению 09.04.01 – «Информатика и вычислительная техника».

«Характеристика поведенческих аспектов деятельности студента в период работы над ВКРМ». Студентом была проявлена высокая степень инициативы и самостоятельности.

Заведующий кафедрой
«Сетевые информационные технологии и сервисы»,
д.т.н., профессор

В.А.Докучаев

РЕЦЕНЗИЯ

на выпускную квалификационную работу студента группы М092301(75)
Московского Технического Университета Связи и Информатики
Удалова Игоря Дмитриевича, обучающейся по направлению 09.04.01
на тему «Исследование и разработка алгоритма поиска пути для большого
пространства с использованием параллельного вычисления»

В условиях постоянно растущей сложности виртуальных миров и требований к производительности игровых приложений, вопрос эффективного поиска пути становится всё более значимым. Необходимо совершенствовать существующие алгоритмы и методы, а также внедрять новые принципы работы и архитектуры систем навигации. Однако каждое решение для обеспечения эффективного поиска пути теряет свою актуальность с развитием технологий, поэтому данная тематика сохраняет высокую актуальность. Множество игровых проектов по-прежнему используют устаревшие способы навигации, что приводит к значительным ограничениям масштабируемости и производительности.

В ходе написания выпускной квалификационной работы магистра был выполнен обзор существующих алгоритмов поиска пути в компьютерных играх. Были изучены и проанализированы различные методы навигации, предназначенные для обеспечения эффективного перемещения игровых объектов в виртуальном пространстве. Также был проведен анализ существующих подходов к параллельным вычислениям в контексте алгоритмов поиска пути. Целью анализа было определение основных проблем и ограничений, связанных с поиском пути в больших пространствах игрового мира.

Магистрантом на основе изученных аспектов была произведена разработка иерархического поиска пути с применением технологии Unity DOTS для параллелизации вычислений. Полученное решение представляет собой модернизацию и усовершенствование существующих методов навигации, позволяющее значительно повысить эффективность поиска пути в больших игровых пространствах за счет оптимального использования многопоточных вычислений.

В качестве недостатков рецензируемой работы можно отметить:

- ограниченный объем исследования: В работе был охвачен лишь определенный аспект алгоритмов поиска пути, используемый на плоских картах, что оставляет возможности для дальнейших исследований и расширения темы.
- недостаточное тестирование в различных игровых сценариях: Работа не включает результаты испытаний разработанного алгоритма в разнообразных игровых контекстах с различной сложностью и масштабами пространств, что может ограничить его универсальную применимость.

- ограничения технологической базы: Работа основана на использовании конкретной технологии Unity DOTS, что может ограничить перенос результатов на другие игровые движки или среды разработки.

- необходимость дальнейшей оптимизации: Разработанный иерархический поиск пути может требовать дополнительной оптимизации для обеспечения более высокой производительности при экстремальных нагрузках, а также улучшения интеграции с другими компонентами игровых систем.

Выпускная квалификационная работа магистра выполнена качественно, в соответствии с ГОСТами и полном соответствии с заданием, тема работы актуальна и раскрыта полностью, результаты работы могут использоваться в игровых компаниях, в качестве решений для повышения эффективности навигации в больших игровых пространствах, а также при разработке новых проектов. Работа имеет методологически правильное логическое построение, а реализация иерархического поиска пути с использованием Unity DOTS демонстрирует высокий уровень формирования профессиональных компетенций в области разработки игровых технологий и параллельных вычислений.

Вследствие того, что работа не имеет существенных недостатков, а указанные выше не влияют на конечный результат, считаю, что представленная ВКРМ выполнена на актуальную тему, заслуживает оценки «Отлично», а магистрант Удалов И. Д. – присвоения квалификации магистра по направлению 09.04.01.

Руководитель проекта НИУ ВШЭ,
к.т.н., доцент

Олейник А.И.

АННОТАЦИЯ

Вид работы

Выпускная квалификационная работа по специальности 09.04.01 «Информатика и вычислительная техника».

Тема работы

«Исследование и разработка алгоритма поиска пути для большого пространства с использованием параллельного вычисления».

Область применения

Компьютерные игры.

Цель работы

Разработка иерархического поиска пути с применением параллельных вычислений для эффективной навигации в масштабных виртуальных пространствах компьютерных игр.

Специфические особенности

В работе представлена разработка, имеющая реальное практическое применение в индустрии компьютерных игр, реализованная на платформе Unity с использованием технологии DOTS для обеспечения эффективной параллелизации вычислений.

Краткое содержание

Работа состоит из введения, четырех разделов, заключения и списка использованных источников.

Первый раздел посвящен классификации и анализу алгоритмов поиска пути в разработке игр. Произведен детальный обзор классических алгоритмов поиска пути. Рассмотрены параллельные алгоритмы поиска, а также варианты иерархического поиска пути, их особенности и характеристики. Представлен сравнительный анализ различных алгоритмов по критериям производительности, масштабируемости и эффективности использования вычислительных ресурсов.

Второй раздел посвящен исследованию существующих подходов и методов параллельного вычисления. Рассмотрены основные подходы к параллельному программированию в контексте игрового поиска пути. Проанализированы различные архитектуры параллельных вычислений и модели параллельного программирования. Особое внимание уделено API для параллельного программирования в современных средах разработки игр.

Третий раздел содержит информацию о модификациях иерархического поиска пути в зависимости от конкретных игровых задач. Подробно рассмотрены статические и динамические модификации иерархического поиска пути (SHPA* и DHPA*). Представлен сравнительный анализ различных модификаций иерархического поиска пути.

Четвертый раздел отражает практическую разработку иерархического поиска для большого пространства с использованием параллельного вычисления на кроссплатформенной среде разработки Unity Engine. Описан процесс проектирования игровой карты, разработки алгоритма поиска пути и добавления распараллеливания задач для агентов. Представлены результаты оценки эффективности разработанного алгоритма в различных сценариях применения.

ВКРМ содержит 4 главы, 11 рисунков, 10 таблиц. Объем ВКРМ составляет 92 страниц. Список литературы содержит 72 источник.

Результаты ВКРМ докладывались на XIX Международной отраслевой научно-технической конференции «Технологии информационного общества», международном научно-техническом форуме «Телекоммуникационные и вычислительные системы» 2024 года, а также на XV и XVI Молодежном научном форуме, доклады осуществлялись в период 2024-2025 гг.

Содержание

Введение	9
1. Классификация алгоритмов поиска пути в разработке игр	13
1.1 Классические алгоритмы поиска пути.....	13
1.2 Параллельные алгоритмы поиска пути.....	20
1.3 Иерархический поиск пути.....	25
1.4 Сравнительный анализ алгоритмов поиска пути	30
Выводы	37
2. Анализ существующих подходов и методов параллельного вычисления ...	39
2.1 Существующие подходы к параллельному программированию в игровом поиске пути.....	39
2.2 Архитектуры параллельных вычислений	44
2.3 Модели параллельного программирования.....	48
2.4 API для параллельного программирования в средах разработки игр	52
Выводы	59
3. Модификации иерархического поиска пути в зависимости от задачи.....	61
3.1 Недостатки иерархического поиска пути при решении узконаправленных игровых задач.....	62
3.2 Статический иерархический поиск пути A* (SHPA*).....	67
3.3 Динамический иерархический поиск пути A* (DHPA*)	69
3.4 Сравнительный анализ модификаций иерархического поиска пути для разработки игр.....	73
Выводы	74
4. Разработка иерархического поиска для большого пространства с использованием параллельного вычисления на кроссплатформенной среде разработки Unity Engine.....	76
4.1 Проектирование игровой карты	76
4.2 Разработка алгоритма поиска пути	82

4.3	Добавление распараллеливания задач для агентов	85
4.4	Оценка алгоритма	87
	Выводы	88
	Заключение	89
	Список использованных источников	91

Введение

Компьютерные игры уже давно переросли рамки простого развлечения, став одной из крупнейших и самых динамично развивающихся отраслей современной экономики. В 2024 году глобальный рынок видеоигр согласно сайту Tadviseer превысил отметку в 187,7 миллиарда долларов, что ставит его выше по доходам таких гигантов индустрии развлечений, как кино и музыка. Количество активных игроков по всему миру приближается к 3 миллиардам, а стриминговые сервисы и киберспорт привлекают аудиторию в миллионы человек. Столь впечатляющие показатели не только свидетельствуют о масштабах индустрии, но и подчеркивают её влияние на культуру, экономику и технологии [1].

В последние десятилетия задача поиска пути стала одной из центральных тем в компьютерных науках и инженерии. От навигационных систем для автономных автомобилей до управления движением персонажей в видеоиграх, от маршрутизации в телекоммуникационных сетях до планирования роботов в промышленности — проблема поиска оптимального маршрута находит свое применение во множестве областей. В своей основе поиск пути заключается в нахождении наилучшего маршрута между двумя точками в пространстве, учитывая различные препятствия и ограничения. С развитием технологий и увеличением объема данных, обрабатываемых в реальном времени, эта задача становится все более сложной и требовательной.

Традиционные алгоритмы поиска пути, например алгоритм Дейкстры и A^* , являлись стандартом в данной области в течение многих лет из-за их надежности и относительной простоты реализации. Однако такие алгоритмы ряд ограничений. Во-первых, пути, которые они находят, могут быть недостаточно быстрыми для использования в реальном времени, особенно в динамической среде, например там, где изменения условий игровой среды происходят быстро. Во-вторых, производительность этих алгоритмов сильно зависит от количества

вершин и рёбер в графе, что означает, что время выполнения растёт экспоненциально с увеличением размера графа. В современных условиях, где разработчики компьютерных игр всё больше стараются впечатлить игроков масштабом своего мира, эти ограничения становятся все более критичными.

Одним из наиболее эффективных средств преодоления указанных проблем являются параллельные вычисления. Они позволяют разделить вычислительную задачу на независимые подзадачи, которые выполняются одновременно на разных ядрах центрального процессора или графического процессора. В случае же задачи поиска пути — представляется возможным одновременно обрабатывать четко разграниченные области пространства, что существенно ускорит выполнение данной задачи.

Одна из наиболее актуальных областей применения параллельных алгоритмов поиска пути — это видеоигры. Сегодняшние компьютерные игры поражают проработанностью своих миров, а за счет управления светом и проработанных анимаций каждой части игры, зачастую уже трудно отличить компьютерную графику и снятое на камеру изображение. Цель игрового алгоритма поиска пути гарантировать, что персонажи могут плавно и реалистично перемещаться по миру. Он берет в расчет различные препятствия, сторонних персонажей, а также события, происходящие в игровом мире и меняющие либо всю карту, либо отдельные локации. Это особенно важно для жанров РПГ и стратегий в реальном времени, где игрок должен погрузиться в атмосферу происходящего на экране.

Тем не менее, реализация параллельных алгоритмов поиска пути сопряжена с рядом технических и теоретических проблем. Одной из ключевых проблем является необходимость синхронизации параллельных процессов. В отличие от последовательных алгоритмов, где управление переходит от одной операции к другой линейно, параллельные вычисления требуют координации множества процессов, работающих одновременно. Это особенно важно в контексте задач поиска пути, где необходимо обеспечить целостность данных и

избежать конфликтов при обновлении информации о текущем состоянии пути. Неправильное управление синхронизацией может привести к некорректным результатам или даже к полной остановке системы.

Еще одной значительной проблемой является управление ресурсами, особенно памятью. Параллельные алгоритмы, как правило, требуют больших объемов памяти для хранения промежуточных данных и результатов вычислений. Это может стать серьезным ограничением, особенно при работе с большими пространствами, где требуется хранить информацию о множестве узлов и путей. Необходимо разрабатывать эффективные методы управления памятью, чтобы минимизировать потребление ресурсов и предотвратить перегрузку системы.

Необходимо также учитывать аспект масштабируемости. Параллельные вычисления могут значительно улучшить производительность, но этот эффект не всегда линейный. С увеличением числа параллельно работающих процессов может возникать явление уменьшения отдачи, когда затраты на управление и синхронизацию процессов начинают превышать выгоду от параллелизации. Это требует тщательного анализа и оптимизации архитектуры алгоритма и системы в целом, чтобы обеспечить максимально возможную производительность.

Наконец, стоит отметить важность учета динамических изменений в окружающей среде. В реальных приложениях, таких как навигация для автономных транспортных средств или мобильных роботов, условия местности могут быстро меняться. Это требует от системы поиска пути способности к быстрой адаптации и перерасчету маршрутов в реальном времени. Параллельные алгоритмы могут предоставить необходимую производительность для таких задач, но также требуют дополнительных усилий для обработки и интеграции данных в условиях постоянных изменений.

Таким образом, исследование и разработка алгоритмов поиска пути для больших пространств с использованием параллельных вычислений представляют собой сложную и многоаспектную задачу. Она требует глубокого

понимания как теоретических основ алгоритмов поиска пути, так и практических аспектов реализации параллельных вычислений. Современные вычислительные мощности предоставляют новые возможности для решения этих задач, открывая перспективы для создания более эффективных и быстрых систем. Продолжение исследований в этой области имеет важное значение для дальнейшего развития технологий и их приложений в различных областях науки и техники, особенно в контексте динамично развивающейся индустрии видеоигр.

1. Классификация алгоритмов поиска пути в разработке игр

1.1 Классические алгоритмы поиска пути

Классические алгоритмы поиска путей составляют основу пространственной навигации в информатике. Эти алгоритмы широко исследовались и реализовывались с первых дней существования вычислительной техники, предлагая различные подходы к поиску оптимальных путей между двумя точками в графе или сетчатой структуре. Понимание этих фундаментальных алгоритмов имеет решающее значение для разработки более совершенных систем поиска путей в крупномасштабных средах.

Основные понятия

Перед тем как углубиться в детали алгоритмов, важно понять основные понятия, связанные с задачами поиска пути:

1. **Граф:** Граф состоит из узлов (вершин) и рёбер, соединяющих эти узлы. Граф может быть ориентированным (где рёбра имеют направление) или неориентированным.
2. **Вес:** Рёбра графа могут иметь веса, представляющие стоимость перемещения от одного узла к другому. Веса могут представлять расстояние, время или любую другую метрику.
3. **Кратчайший путь:** Кратчайший путь между двумя узлами в графе — это путь с наименьшей суммарной стоимостью (весом).

1. Поиск в ширину (Breadth-First Search, BFS)

Алгоритм поиска в ширину просматривает граф слоями, начиная от начального узла и исследуя всех соседей, затем соседей соседей и т.д., пока не будет найден целевой узел. Такой подход гарантирует нахождение пути с наименьшим числом шагов в графах с равными весами ребер. BFS является детерминированным и систематически перебирает состояния на увеличивающейся глубине. В худшем случае сложность поиска в ширину

линейна по суммарному числу вершин и ребер $O(V + E)$, однако практическая проблема BFS – высокая потребность в памяти. Алгоритм хранит очереди открытых узлов на каждой широкой волне поиска, что при большой глубине графа приводит к экспоненциальному росту числа хранимых узлов. Например, если цель находится далеко от старта, BFS будет хранить множество промежуточных состояний, что порождает большие затраты памяти при далёкой цели [2]. Работа алгоритма представлена на рисунке 1.1.

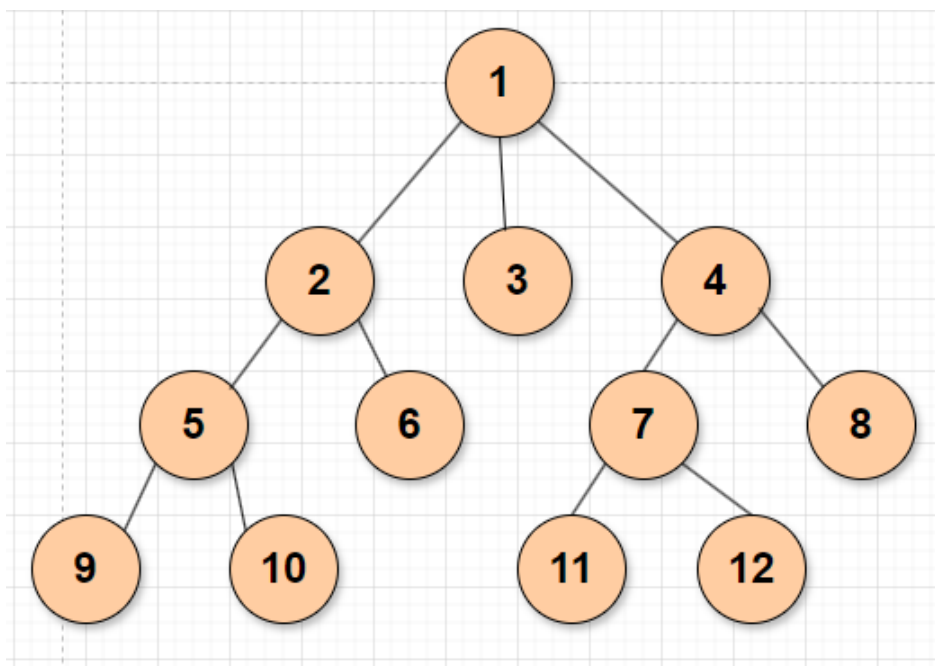


Рисунок 1.1 – Алгоритм BFS

В контексте разработки игр BFS может использоваться для простых задач, например, когда нужно определить достижимость области или построить путь на небольшом невзвешенном игровом поле. Достоинство BFS – простота реализации и гарантия кратчайшего пути в ходовых шагах для равновзвешенных графов. Однако основной недостаток – неэффективность на больших картах: алгоритм буквально «заливает» пространство поиска, генерируя множество лишних узлов без приоритета целевому направлению. В сравнительном эксперименте в лабиринте показано, что хотя BFS находит путь, сопоставимый по длине с решением A^* , он делает значительно больше лишних вычислений, т.е. работает менее эффективно. Проще говоря, BFS тратит больше времени на

проверку множества узлов, тогда как более информированный подход смог бы сузить поиск [3, 45].

2. Поиск в глубину (Depth-First Search, DFS)

Алгоритм поиска в глубину, в отличие от BFS, развивается вдоль одного пути как можно дальше, прежде чем отступать назад. Он реализуется либо рекурсивно, либо с использованием стека, углубляясь в граф до тех пор, пока не упрется в тупик, после чего возвращается к последней точке ветвления и выбирает следующий путь. DFS не гарантирует нахождения кратчайшего пути – он просто ищет какой-либо путь до цели, поэтому результат может быть сильно неоптимальным по длине. Сложность в худшем случае также порядка $O(V + E)$, но на практике количество посещенных узлов зависит от структуры графа и порядка обхода. Поиск в глубину обычно потребляет меньше памяти, чем BFS, поскольку хранит только текущий путь и некоторые вспомогательные данные, вместо полной очереди. Тем не менее, без дополнительных мер DFS может застрять в очень длинной ветви или даже в бесконечном цикле (если граф содержит цикл и не запоминать посещенные состояния) [50]. Работа алгоритма представлена на рисунке 1.2.

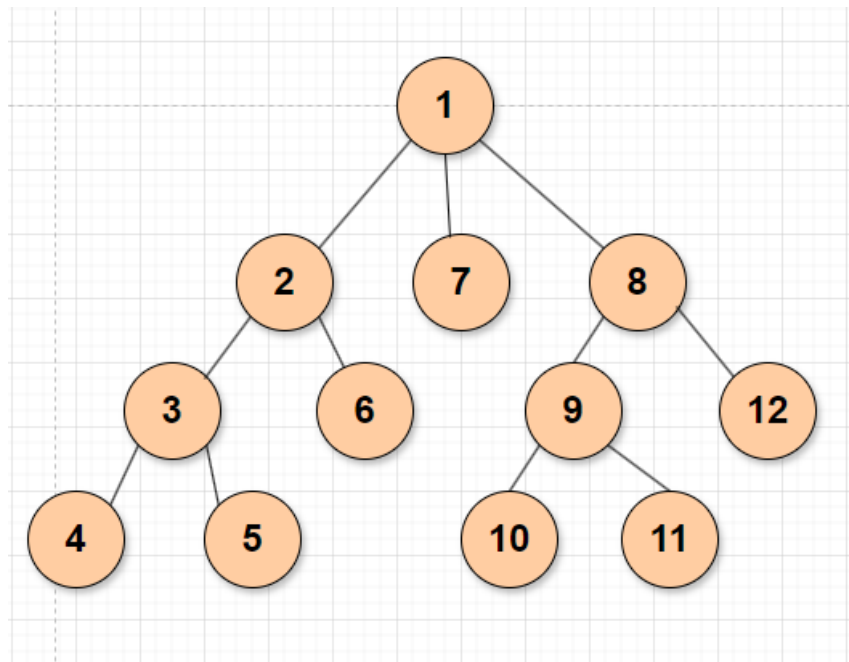


Рисунок 1.2 – Алгоритм DFS

В играх чистый DFS редко применяется для нахождения оптимальных путей из-за перечисленных недостатков. Его плюс – скорость нахождения какого-нибудь решения на глубоко связных графах, а также минимальные затраты памяти, – может использоваться в задачах, где не важно найти кратчайший маршрут, а нужно лишь быстро проложить хоть какой-то путь (например, для поведения, не требующего оптимальности). Однако, если граф велик и содержит много ответвлений, поиск в глубину без ограничений может проваливаться в экспоненциальный перебор. Существует техника итеративного углубления и её комбинация с эвристикой (IDA*), когда DFS запускается с постепенно растущим ограничением глубины, что в итоге эквивалентно BFS по охвату, но сохраняет память, необходимую для хранения фронта. Такие подходы находят применение в задачах с очень большими пространствами состояний, где невозможен полноценный хранитель открытых узлов, однако для обычных задач планирования пути в играх чаще используются более эффективные алгоритмы, описанные далее.

3. Алгоритм Дейкстры

Алгоритм Дейкстры – классический метод поиска кратчайшего пути в графе с неотрицательными весами ребер. Он систематически распространяет волны стоимости от стартового узла, подобно BFS, но учитывает вес (стоимость) переходов. На каждой итерации из нерассмотренных вершин выбирается та, до которой достигнуто минимальное текущее расстояние, после чего ее соседи релаксируются. В итоге алгоритм находит оптимальный по суммарному весу маршрут до всех достижимых узлов (или до заданной цели, если остановиться при ее извлечении из очереди). В невзвешенных графах (все веса равны) поведение алгоритма Дейкстры эквивалентно поиску в ширину.

Преимущество алгоритма Дейкстры – гарантированное нахождение глобально кратчайшего пути. С точки зрения сложности, реализация с приоритетной очередью имеет трудоемкость $O((V+E) \log V)$ в общем случае. Для небольших графов и редких обновлений путей это приемлемо, однако на

больших игровых картах с тысячами вершин и ребер алгоритм может работать медленно. Существенным недостатком является то, что Дейкстра не использует никакой информации о направлении к цели – он «слепо» разрастается во все стороны, пока не покроет область вокруг старта до самого финиша. Это приводит к тому, что на практике приходится обрабатывать очень много узлов, особенно если цель далеко. Так, в задаче навигации по большому миру алгоритм Дейкстры будет исследовать значительную часть карты, даже если цель находится в определенном углу – без эвристики поиск не знает, куда двигаться приоритетнее. Работа алгоритма представлена на рисунке 1.3.

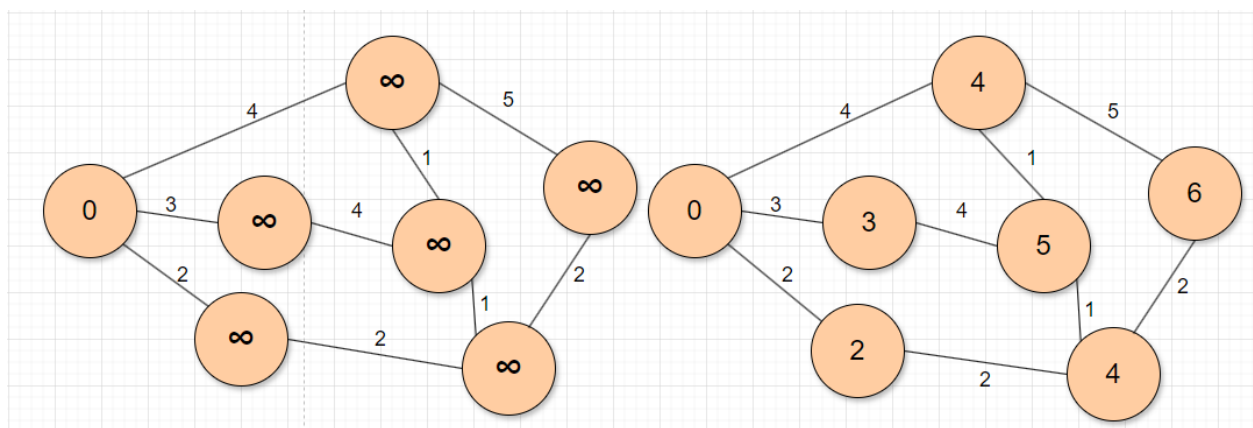


Рисунок 1.3 – Пример работы алгоритма Дейкстры

В игровых приложениях алгоритм Дейкстры применяется не так часто для онлайн-поиска пути одного персонажа, поскольку существует более быстрый альтернативный алгоритм A*, описываемый ниже. Тем не менее, идеи Дейкстры находят применение, например, для предварительных расчетов навигационных сетей: можно заранее вычислить расстояния от всех точек до ключевой цели (или между всеми парами важных точек) и сохранять их в виде справочника. Также алгоритм Дейкстры востребован в задачах, где нужно найти пути до всех целей или построить карты расстояний, например, для влияния местности или зон поражения в стратегиях. В таких случаях, несмотря на высокую сложность, однократный запуск алгоритма с сохранением результатов окупается множеством последующих быстрых запросов.

4. Алгоритм A*

Алгоритм A* (A-star) – наиболее известный и распространенный подход к поиску пути в играх. Он представляет собой модификацию алгоритма Дейкстры, в которую добавлена эвристическая функция оценки расстояния до цели. При поиске A* каждую вершину оценивает по приоритету $f(n) = g(n) + h(n)$, где $g(n)$ – известная стоимость пути от старта до узла n , а $h(n)$ – эвристическое предположение относительно стоимости от n до цели. Благодаря эвристике, A* «направляет» поиск в сторону целевого узла, что значительно снижает количество проверяемых состояний по сравнению с Дейкстрой, особенно на больших картах. Если эвристическая функция адекватна и допустима (не переоценивает реальную оставшуюся дистанцию), алгоритм A* гарантированно находит оптимальный путь. Более того, с информативной эвристикой A* никогда не развернет больше узлов, чем эквивалентный поиск Дейкстры без эвристики [4-6]. Работа алгоритма представлена на рисунке 1.4.

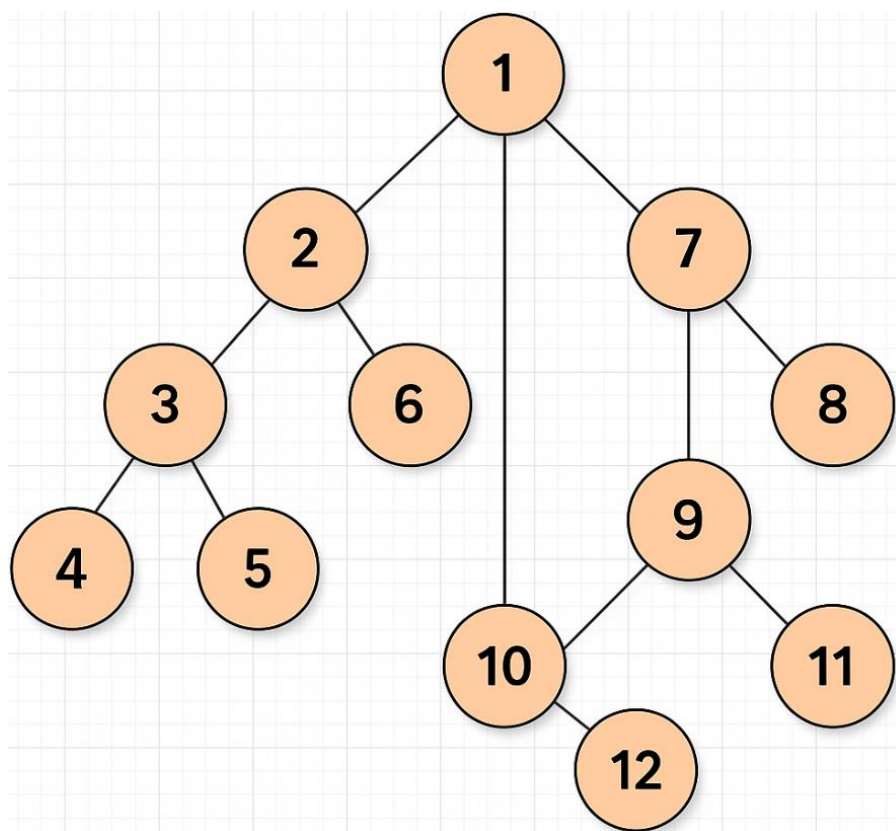


Рисунок 1.4 – Пример работы алгоритма A*

В типичном применении на сетке или графе игрового мира в качестве $h(n)$ выбирают манхэттенское или евклидово расстояние до цели (или более сложные оценки с учетом препятствий), что даёт эффективное приближение оставшегося пути [70]. По этой причине A^* обычно обходится значительно меньшим числом узлов, чем Дейкстра: алгоритм быстро «стягивается» к области цели, не исследуя лишние направления. Например, в эксперименте с поиском в лабиринте A^* показал наилучшее время нахождения пути, затратив минимальный объем вычислений, тогда как BFS хотя и нашел путь той же длины, но израсходовал больше ресурсов из-за неинформированности поиска [7,8].

Тем не менее, у A^* есть и недостатки. Во-первых, потребление памяти по-прежнему велико – алгоритм хранит все открытые узлы (фронтир), аналогично BFS, что на чрезвычайно больших картах может превысить доступную память. Во-вторых, в худшем случае (при плохой эвристике или ее отсутствии) A^* вырождается в поиск Дейкстры и имеет ту же экспоненциальную сложность по глубине решения. На практике это означает, что для очень длинных маршрутов (с большим числом шагов) время работы растет экспоненциально. Кроме того, путь, найденный на дискретной решетке, может получиться «угловатым» и неоптимальным относительно реального расстояния в непрерывном пространстве – A^* ограничен связями графа (например, ходами по сетке), и его решение может зигзагообразно приближаться к цели. Эти проблемы становятся актуальными в современных играх с огромными открытыми мирами: прямое применение A^* может быть слишком медленным. В ответ на это появилось множество оптимизаций и надстроек над A^* , позволяющих ускорить поиск или улучшить качество пути – например, Jump Point Search, Theta, ANYA и др. Также часто применяются техники ограниченного поиска и прерываемого поиска, если путь нужен срочно и может быть уточнен позже. Однако базовым инструментом для планирования перемещения индивидуального персонажа остается именно алгоритм A^* благодаря его эффективности и оптимальности при правильной эвристике [9]. Сравнение алгоритмов представлено в таблице 1.1.

Таблица 1.1 – Сравнение алгоритмов поиска пути в компьютерных играх

Алгоритмы	Скорость (в играх)	Требуемая память	Гарантирует точность пути	Подходящий тип игрового мира
BFS	Средняя, подходит для небольших карт	Низкая	Гарантирует	Простые, невзвешенные карты
DFS	Быстрая, но не всегда эффективная	Низкая	Не гарантирует кратчайший путь	Любые карты
Алгоритм Дейкстры	Медленная, особенно на больших картах	Средняя	Гарантирует	Взвешенные игровые карты
A*	Быстрая, при правильно подобранной эвристической функции	Высокая, из-за необходимости хранить оценки	Да, при оптимальной эвристике	Большие и сложные карты

1.2 Параллельные алгоритмы поиска пути

Ежегодно компании, такие как AMD, Intel или Nvidia выпускают на рынок новые процессоры и видеокарты. Стремительное развитие вычислительного оборудования привело к такому же развитию к необходимости разработки

алгоритмов поиска пути, которые могут использовать возможности параллельной обработки данных. Эти алгоритмы направлены на распределение вычислительной нагрузки между несколькими вычислительными устройствами для достижения более быстрых решений по поиску пути, что особенно важно для крупномасштабных сред с множеством агентов [10].

Любой из алгоритмов, перечисленных в главе 1.1 можно улучшить, применив параллельные вычисления, но так как в большинстве современных игр используется именно алгоритм поиска пути A^* (из-за его скорости и универсальности), перед тем как рассказать об подходах к распараллеливанию поиска пути, расскажу про вариации A^* , которые были разработаны для использования параллельных вычислительных архитектур (далее по тексту параллельный алгоритм A^*).

Параллельные варианты A^*

Параллельная версия A^* (PA^*): наивный подход к параллельному поиску пути – разделить работу алгоритма A^* между несколькими потоками. Например, можно позволить нескольким потокам одновременно извлекать узлы из общей очереди открытых состояний и расширять их. Однако при такой реализации неизбежны проблемы синхронизации: чтобы избежать одновременного конфликта за структуру данных, требуется защита (мьютексы) при доступе к общей очереди и множеству посещенных узлов. Экспериментально показано, что такой прямолинейный подход к параллельному A^* с общей глобальной очередью демонстрирует низкую эффективность из-за траты ресурсов на синхронизацию потоков [11-14].

Параллельное втягивание A^* (PRA^*) стало одним из первых усовершенствований A^* для массово-параллельных систем. Идея PRA^* состоит в распределении фронта поиска между потоками таким образом, чтобы минимизировать конкуренцию за общие ресурсы. В частности, каждый поток имеет собственные списки open и closed, а новые узлы назначаются потокам по хеш-функции. Проще говоря, пространство состояний разбивается: при

генерации очередного узла алгоритм вычисляет хеш от состояния и определяет, какому потоку (процессору) этот узел «принадлежит», добавляя его в локальную очередь этого потока. Такая схема позволяет потокам работать более независимо, поскольку каждый в основном оперирует со «своими» узлами.

В оригинальной версии PRA* также предусматривалась стратегия “retraction” (втягивание) для работы в условиях ограниченной памяти – при переполнении структуры алгоритм мог отбрасывать наименее перспективные узлы. Однако в современных реализациях на обычных ПК потребность в этом менее актуальна, и основной интерес представляет именно параллельное распределение. PRA* значительно снижает главный узкий момент параллельного A^* – единственную глобальную очередь – однако полностью исключить синхронизацию не удастся. Потоки все равно обмениваются некоторыми узлами между собой (когда один генерирует состояние, принадлежащее другому) и должны синхронизироваться при добавлении чужих узлов [15].

Параллельный поиск на основе границ (разделение фронта)

Одной из проблем параллелизации графового поиска является дублирование работы – разные потоки могут независимо начать расширять соседние области и тратить время на одни и те же состояния. Чтобы этого избежать, вводят структурированное разбиение пространства состояний. Метод Parallel Structured Duplicate Detection (PSDD) предлагает заранее определить набор блоков состояний (абстрагировав исходный граф), так чтобы каждый поток обрабатывал свой блок и только периодически сверял границы с другими. На основе этой идеи создан алгоритм Parallel Best-NBlock-First (PBNF) – параллельный поиск «лучший сначала по N-блокам». Он использует априорно заданную абстракцию (функцию отображения множества конкретных состояний в абстрактное состояние), делящую граф на n -блоки. Каждый поток может брать в работу один из n -блоков (который имеет наименьшую f -стоимость на границе фронта) и выполнять внутри него поиск практически без синхронизации, пока

не достигнет границы блока [9]. Синхронизация требуется, когда один блок поисково исчерпан или когда необходимо переключиться на более перспективный блок.

Алгоритмы типа PBNF добиваются высокой масштабируемости на многоядерных системах. В тестах на различных доменах (включая планирование в STRIPS, поиск пути на сетке и головоломки) PBNF показал лучшую скорость нахождения оптимальных решений по сравнению с ранее предложенными параллельными алгоритмами. Увеличение числа потоков позволило почти линейно (а на некоторых задачах – сверхлинейно) ускорить получение результата за счет параллельной обработки независимых частей пространства поиска. Важно, что PBNF и подобные ему методы сохраняют оптимальность решения: несмотря на разбиение, они эквивалентны по результату последовательному A^* , но требуют значительно меньше совокупного времени. Таким образом, подход с разделением по границам является одной из наиболее успешных стратегий параллельного поиска пути на современном оборудовании [16].

Параллельное планирование на основе выборки

Помимо классического поискового пространства состояний, в робототехнике и игровых движках применяется так называемое планирование на основе случайной выборки – например, алгоритмы семейства RRT (Rapidly-exploring Random Tree) и PRM (Probabilistic Roadmap). Эти методы особенно полезны для навигации в непрерывном пространстве (не на сетке), где пространство состояний бесконечно велико. Суть RRT – случайным образом выборочно строить дерево достижимых состояний, расширяя его в направлении цели. Такой подход можно эффективно параллелизовать, поскольку разные потоки могут строить разные ветви дерева одновременно [10].

Существуют параллельные версии RRT и оптимизированного RRT* (например, PRRT и PRRT*), которые распределяют задачу построения дерева между ядрами процессора. Одной из техник является разделение пространства

конфигураций на непересекающиеся области, закрепленные за конкретными потоками. Каждый поток генерирует случайные точки (семплы) только в своей части пространства и пытается присоединить их к своему локальному дереву пути. Периодически деревья могут соединяться или обмениваться лучшими найденными путями. Такой разделенный по пространству метод не только распараллеливает работу, но и улучшает эффективность за счет кэш-локальности: каждый поток оперирует более компактными структурами данных (своим поддеревом), что повышает скорость доступа к памяти и может приводить к сверхлинейному ускорению на практике [17].

Для избежания конфликтов используется lock-free (без блокировок) организация доступа к общим структурам, например, к общему графу или k-d дереву ближайших соседей при добавлении новых узлов. Это позволяет потокам работать без ожидания друг друга. В результате параллельные выборочные алгоритмы планирования показывают почти линейное увеличение скорости при увеличении числа потоков, а в некоторых экспериментах – ускорение большее, чем в два раза при удвоении числа потоков.

В играх планирование на основе выборки применяется реже, чем на решетках с A^* , однако в задачах движений персонажей в физически непрерывном 3D-мире (например, полеты, сложная траектория обхода препятствий) методы типа RRT могут оказаться полезными. Возможность их распараллелить делает их привлекательными для ситуаций, требующих быстрых решений в больших конфигурационных пространствах (например, сложная кинематика персонажей, пути в динамическом окружении с множеством степеней свободы) [12].

Распределенный поиск пути: MapReduce-подход

Другим направлением параллельного поиска пути является использование распределенных вычислений – когда поиск выполняется на кластере машин или в облачной среде. Модель MapReduce предоставляет удобный шаблон для разделения задачи на множество мелких подзадач (map) с последующей

агрегацией результатов (reduce). Хотя задача поиска кратчайшего пути не тривиально бьется на независимые части, были предложены подходы к ее формулировке в терминах MapReduce. В частности, для очень больших графов (например, дорожные сети уровня страны с миллиардами узлов и ребер) A^* можно реализовать с использованием Hadoop. Авторы одного из исследований представили способ разложения A на задачи MapReduce для выполнения расчета маршрутов на Hadoop-кластере, и экспериментальное применение к реальной дорожной сети показало ускорение по сравнению с однопоточным исполнением.

Идея в том, чтобы параллельно обрабатывать расширение множества узлов фронта на разных узлах кластера. На фазе Map каждый узел обрабатывает часть открытых состояний, генерируя для них соседей, а на фазе Reduce результаты объединяются, отбираются глобально лучшие узлы для следующей волны итерации. Такое итеративное распределенное выполнение продолжается, пока не найдена цель. В упомянутом подходе на 6-узловом кластере Hadoop время поиска на огромных графах существенно сократилось, продемонстрировав хорошую масштабируемость. Тем не менее, накладные расходы MapReduce (сериализация данных, пересылка по сети и пр.) делают подобные решения оправданными лишь для действительно огромных задач, которые невозможно эффективно решить на одной машине. В контексте обычных игр такой уровень распределения применяется редко, однако может найти применение, например, в онлайн-играх с общим миром огромного размера (ММО) или для предподсчета навигационных структур в облаке.

1.3 Иерархический поиск пути

Иерархические алгоритмы поиска путей решают проблемы масштабируемости классических алгоритмов путем введения многоуровневых абстракций пространства поиска. Эти подходы значительно снижают сложность поиска пути в больших средах.

Иерархический поиск пути A^* (HРА*)

HРА* создает иерархию абстракций, разбивая пространство поиска на кластеры. Абстрактные узлы представляют собой точки входа и выхода между кластерами, а ребра соединяют эти узлы на основе предварительно вычисленных путей [18].

Алгоритм работает в три этапа:

1. Предварительная обработка: карта делится на кластеры, а на границах кластеров создаются абстрактные узлы. Пути между абстрактными узлами в пределах одного кластера предварительно вычисляются.
2. Абстрактное планирование пути: A^* выполняется на абстрактном графе, чтобы найти высокоуровневый путь.
3. Уточнение пути: абстрактный путь преобразуется в конкретный, следуя заранее вычисленным внутрикластерным путям.

Иерархический подход значительно снижает сложность пространства поиска. Для карты размера $n \times n$ классический A^* оперирует $O(n^2)$ узлами, в то время как HРА* сводит это к $O(n)$ узлам на абстрактном уровне. Такое сокращение позволяет эффективно искать пути в средах, которые были бы непомерно дорогими для плоских подходов.

Гибкость HРА* позволяет использовать различные стратегии кластеризации:

1. Кластеризация по равномерной сетке: делит карту на кластеры с одинаковым размером сетки. Этот подход прост в реализации, но может плохо сочетаться со структурой окружающей среды.
2. Кластеризация на основе карты игры: создает кластеры разного размера в зависимости от сложности окружения. Регионы с большим количеством препятствий получают более мелкие кластеры, в то время как открытые области используют более крупные кластеры.
3. Топологическая кластеризация: формирует кластеры на основе естественных делений среды, таких как комнаты в здании или регионы

в ландшафте. Такой подход приводит абстракцию в соответствие со структурой среды.

Анализ производительности показывает, что НРА* может на порядки ускорить работу с большими окружениями по сравнению с плоским A*. Накладные расходы на препроцессирование амортизируются на несколько запросов пути, что делает его особенно подходящим для игровых сред с частыми запросами поиска пути [19].

Близкооптимальный иерархический поиск пути (ННА*)

Один из первых и самых известных алгоритмов данного класса – Hierarchical Pathfinding A*. Карта разбивается на связные участки фиксированного размера (например, области 20x20 клеток). Для каждого кластера на этапе предобработки вычисляются и сохраняются расстояния между его входами/выходами. Эти входы фактически играют роль абстрактных узлов, соединяющих кластеры. Когда нужно найти путь, алгоритм выполняет:

1. Поиск A* на уровне кластеров: стартовый и целевой узлы поднимаются до соответствующих кластеров, и находится последовательность кластеров от старта к цели.

2. По этой последовательности прокладывается конкретный путь: для каждого перехода из кластера A в кластер B известен вход в A и вход в B, через которые осуществляется переход (эти точки определены решением на абстрактном уровне). Алгоритм извлекает из кэша предвычисленное оптимальное расстояние (и сам маршрут) для пересечения кластера A от текущего положения до выходной точки, и аналогично для кластера B и последующих.

3. Стартовый и целевой кластеры могут быть обработаны частично, если старт/цель внутри них: для сегмента пути внутри начального или конечного кластера выполняется обычный локальный поиск (например, A* ограниченный этим кластером) до ближайшего портала.

Комбинируя все сегменты, НРА* получает полный путь. Этот путь, как правило, субоптимален, поскольку абстрактный поиск не учитывает мелкие отклонения внутри кластеров. Однако отклонения минимальны: в оригинальной работе указано, что после дополнительного сглаживания (post-smoothing) решения отличались от оптимума не более чем на 1%. То есть цена за огромное ускорение – незначительное удлинение пути. НРА* достиг значительного ускорения: поисковое пространство на глобальном уровне сокращается на несколько порядков, поэтому даже с учетом локальных уточнений общий объем работы резко меньше, чем у плоского A* [20, 23].

Непрерывный иерархический поиск пути

Под «непрерывным» здесь понимается подход, обеспечивающий более гладкие, близкие к непрерывному пространству маршруты, устраняющие искусственные углы, возникающие из-за дискретизации. Стандартный НРА* работает на сетке и потому ограничен связями по соседним клеткам: итоговый путь может выглядеть как ломаная линия вдоль клеток. Для персонажей игр, особенно в 3D-мире, такой путь выглядит неестественно. Решение – разрешить агенту двигаться под любым углом, не только по сетке, т.е. реализовать any-angle pathfinding. Из классических алгоритмов для этого известен Theta* (англ. «тета-звезда»), который модифицирует A*: при расширении узла проверяется прямая видимость от родительского узла через нового потомка, и при наличии прямой видимости происходит «скачок» – соединение родителя с потомком минуя промежуточные клетки. Theta* и его более новые варианты (например, ANYA) позволяют находить оптимальные или близкие к оптимальным пути в геометрическом (непрерывном) смысле, а не по сеточным шагам.

Комбинирование идеи Theta* с иерархией привело к появлению алгоритма Hierarchical Pathfinding Theta* (HPT*). В работе 2013 г. (Linus van Elswijk) было показано, что применение Theta* на этапе уточнения пути внутри кластеров позволяет значительно сгладить маршрут и сократить его длину без сильного роста вычислительных затрат. Проще говоря, после того как НРА* определил

последовательность кластеров и порталов, вместо точного следования от одного портала к другому агент может двигаться напрямик, если это позволяет план местности. НРТ* запускает Theta* в каждом кластере, чтобы соединить вход и выход наиболее прямой траекторией. Это устраняет зигзаги, возникающие от выбора фиксированных вершин, соединяющих кластеры [21].

В целом под «непрерывным» иерархическим поиском можно понимать любые техники, которые обеспечивают постоянное обновление и уточнение пути во время движения агента. Отчасти эту идею мы уже видели: НРА* позволяет пересчитывать сегменты по мере приближения к ним. Существуют и другие алгоритмы, такие как D Lite* (Динамический A*) и LPA* (Lifelong Planning A*), которые эффективно обновляют решение при небольших изменениях графа или позиции агента, вместо пересчета с нуля. Их можно встроить и в иерархическую схему, чтобы агент фактически непрерывно планировал свой маршрут на ходу. Это особенно важно для динамичных игр, где препятствия могут появляться внезапно – например, в бою на карте может возникнуть завал, и тогда маршрут надо быстро перестроить. Иерархия тут помогает локализовать перерасчет минимальным районом, а непрерывный (пошаговый) алгоритм – быстро внести коррективы.

Иерархический аннотированный A* (НАА*)

С ростом сложности игровых миров появилась проблема: разные игровые существа могут иметь разные возможности перемещения. Например, крупный персонаж не пройдет там, где пройдет мелкий; летающее существо может игнорировать некоторые препятствия, а наземное – нет; одни могут плавать, другие – только по земле. Классические сеточные алгоритмы обычно строят путь для фиксированного типа агента на фиксированной сетке. Hierarchical Annotated A* (НАА*) – алгоритм, предлагающий решение для гетерогенных миров, где по одной и той же карте могут двигаться агенты разных размеров и способностей.

Идея НАА заключается в предварительной аннотации карты информацией о проходимости для агентов различных типов. Для каждой клетки сетки

вычисляются значения *clearance* – максимального размера агента, который может пройти через эту клетку, а также отмечаются типы местности, которые доступны (например, {Ground} – земля, {Water} – вода, и их комбинации). По сути, каждой клетке приписывается набор «свойств проходимости». Далее строится иерархическая абстракция (кластеры) с учетом этих аннотаций: при объединении клеток в кластеры нужно сохранить информацию о том, через какие входы могут пройти агенты разных размеров и типов. НАА* строит представление уровня выше, в котором узлы — это входы кластеров, аннотированные также требуемым размером и способностью. Для каждого кластера вычисляются возможные переходы с указанием, какой максимальный агент может там пройти и по какой местности. При запросе пути для конкретного агента НАА* выполняет поиск аналогично НРА*, но учитывая требования агента [20].

1.4 Сравнительный анализ алгоритмов поиска пути

Чтобы определить наиболее подходящий подход к поиску пути для крупномасштабных сред, мы устанавливаем критерии оценки, учитывающие как производительность, так и качество.

Критерии оценки

Временная сложность: измеряет вычислительные затраты на поиск пути, обычно выражаемые в нотации Big O. Этот критерий напрямую влияет на производительность в реальном времени, особенно в больших средах или при одновременном запросе нескольких путей.

Сложность пространства: показывает требования к памяти для выполнения алгоритма. Использование памяти становится критическим ограничением для очень больших игровых миров, которые могут превышать доступную системную память.

Оптимальность: определяет, гарантирует ли алгоритм нахождение кратчайшего пути. Хотя оптимальные пути желательны, многие игры могут терпеть небольшие отклонения, если они приводят к значительному улучшению производительности.

Полнота: оценивает, всегда ли алгоритм находит путь, если он существует. Это свойство очень важно для предотвращения застревания персонажей или их нерационального поведения.

Масштабируемость: оценка производительности алгоритма при увеличении размера задачи. Для больших игр с открытым миром требуются подходы к поиску пути, которые плавно масштабируются с увеличением размера окружения.

Потенциал распараллеливания: оценивает способность алгоритма задействовать несколько вычислительных устройств. Современное оборудование предлагает параллельные вычислительные ресурсы, которые могут значительно ускорить операции поиска пути.

Сложность реализации: учитывает сложность реализации и поддержки алгоритма. Сложные алгоритмы могут давать теоретические преимущества, но могут создавать проблемы при разработке и отладке.

Адаптивность к динамической среде: измеряет способность алгоритма работать с изменяющимся окружением. Игры с разрушаемой местностью или создаваемыми игроком сооружениями требуют поиска пути, который может эффективно обновлять траекторию при изменении окружения.

Качество контура: оценивает естественность и плавность сгенерированных траекторий. Нереалистичные траектории могут нарушить погружение игрока в игру или потребовать дополнительных шагов постобработки.

Требования к предварительной обработке: оценивает объем предварительных вычислений, необходимых для выполнения запросов поиска пути [25]. Классификация алгоритмов представлена в таблице 1.2 и 1.3.

Таблица 1.2 – Сравнение классических алгоритмов поиска пути в компьютерных играх

Алгоритм	BFS	DFS	Дейкстра	A*
Временная сложность	$O(b^d)$	$O(b^m)$	$O((V+E) \log V)$	В худшем случае $O((V+E) \log V)$
Сложность пространства	$O(b^d)$	$O(bm)$	$O(V)$	$O(V)$
Оптимальность	Гарантировано для невзвешенных	Не гарантировано	Гарантировано	Зависит от эвристики
Полнота	Есть	Есть	Есть	Может сразу оценить, есть ли путь
Масштабируемость	Умеренная	Низкая	Хорошая	Высокая
Потенциал распараллеливания:	Низкий	Низкий	Умеренный	Высокий
Сложность реализации:	Низкая	Низкая	Умеренная	Высокая
Требования к предварительной обработке:	Нет	Нет	Нет	Низкие

Таблица 1.3 – Сравнение иерархических поисков пути в компьютерных играх

Алгоритм	НРА*	ННРА*	PRA*
Временная сложность	$O(c \cdot \log c + k)$	$O(c \cdot \log c)$	$O(pn \cdot \log n)$
Сложность пространства	$O(c)$	$O(c)$	$O(n/p + p)$
Оптимальность	Почти оптимальный	Приблизительно оптимальный	Почти оптимальный
Полнота	Сразу оценивает по кластерам	Сразу оценивает по кластерам	Сразу оценивает по кластерам
Масштабируемость	Высокая	Высокая	Хорошая
Потенциал распараллеливания:	Высокий	Высокий	Высокий
Сложность реализации:	Высокий	Высокий	Очень высокий
Требования к предварительной обработке:	Высокие	Умеренные	Умеренные
Требования к предварительной обработке:	Высокие	Умеренные	Умеренные

где:

- b = коэффициент ветвления
- d = глубина решения
- m = максимальная глубина
- V = вершины графа

- E = ребра графа
- c = размер кластера
- k = длина пути
- p = количество процессоров
- n = размер графа

Подробный сравнительный анализ

Эффективность памяти

Требования к памяти становятся критическими для очень больших игровых миров:

Классические алгоритмы (BFS, DFS): имеют экспоненциальный рост памяти с размером окружения, что делает их непрактичными для больших карт без модификаций.

A^* и вариации с применением параллельного вычисления: использование памяти зависит от размера открытых и закрытых множеств, что может стать непомерно большим для сложных сред.

Иерархические подходы: демонстрируют значительные преимущества в плане памяти, оперируя преимущественно абстрактными графами, которые на порядки меньше, чем детальная среда.

Параллельные алгоритмы: часто требуют дополнительной памяти для структур синхронизации и распределенных данных, но эти накладные расходы можно компенсировать, распределив нагрузку на память между несколькими вычислительными устройствами.

Количественное сравнение показывает, что иерархические подходы обычно снижают требования к памяти на 50-90% по сравнению с плоскими алгоритмами для больших сред, что делает их особенно ценными для платформ с ограниченным объемом памяти.

Вычислительная производительность

Игры в реальном времени требуют поиска пути, чтобы завершить игру в строго отведенное время:

Классические алгоритмы: производительность падает экспоненциально с ростом размера окружения, становясь непригодной для больших карт без существенных оптимизаций.

Эвристические подходы: A^* и его разновидности обеспечивают более высокую производительность за счет фокусировки поиска на перспективных направлениях, но все еще испытывают трудности при работе с очень большими средами.

Иерархические методы: сокращая эффективное пространство поиска за счет абстракции, иерархические алгоритмы позволяют на порядок повысить производительность при работе с большими средами.

Параллельные реализации: распределите вычислительную нагрузку между несколькими вычислительными устройствами, потенциально достигая почти линейного ускорения с количеством процессоров для хорошо спроектированных алгоритмов.

Бенчмарки для различных размеров окружения показывают, что иерархические алгоритмы неизменно превосходят плоские подходы для больших карт, причем разрыв в производительности увеличивается по мере роста размера окружения. Для очень больших окружений (например, для игр с открытым миром) иерархические подходы могут быть в сотни раз быстрее традиционных A^* [26].

Качество пути

Качество генерируемых путей влияет как на функциональность игры, так и на визуальную эстетику:

Оптимальные алгоритмы (Дейкстра, A^*): гарантируют поиск кратчайшего пути, но могут создавать нереалистичные маршруты, которые выглядят искусственными или механистичными.

Иерархические подходы: как правило, создают почти оптимальные пути с небольшими отклонениями на границах кластеров. Эти отклонения часто незаметны во время игры.

Непрерывные методы: генерируют более плавные и естественные траектории, учитывающие возможности движения персонажа и особенности окружающей среды.

Приближенные алгоритмы: обменивают оптимальность на производительность, с управляемыми параметрами, которые балансируют между качеством пути и вычислительными затратами.

Субъективные оценки и исследования пользователей показывают, что игроки редко замечают небольшие отклонения от оптимальных путей, что делает небольшое снижение качества иерархических подходов приемлемым компромиссом для повышения производительности.

Адаптация к динамичным средам

В современных играх часто встречается динамическое окружение с разрушаемой местностью, движущимися препятствиями или создаваемыми игроком сооружениями:

Классические алгоритмы: справляются с динамическими изменениями естественным образом, поскольку вычисляют пути с нуля для каждого запроса, но за счет снижения производительности.

Инкрементные подходы: обновление существующих путей при возникновении изменений, что позволяет избежать полного пересчета.

Иерархические методы: локализуют обновления в затронутых кластерах, минимизируя влияние изменений. Однако значительные изменения окружающей среды могут потребовать существенного пересчета абстрактного графа.

Опыт доставки игр показывает, что иерархические подходы с эффективными механизмами обновления обеспечивают наилучший баланс между производительностью и адаптивностью для большинства динамичных игровых сред.

Исходя из этих критериев, иерархические алгоритмы в сочетании с параллельной обработкой A^* становятся оптимальным решением для крупномасштабного поиска путей.

Конкретная реализация иерархического поиска пути A^* с параллельной обработкой на обоих уровнях абстракции решает основные проблемы крупномасштабного поиска путей:

1. Масштабируемость: иерархическая структура позволяет эффективно находить пути в средах практически любого размера.
2. Производительность: параллельная обработка данных обеспечивает быстрое реагирование в режиме реального времени даже в сложных сценариях.
3. Качество: поиск пути на основе A^* на обоих уровнях поддерживает почти оптимальное качество пути.
4. Адаптивность: локализованные обновления позволяют эффективно справляться с динамическими изменениями среды.

Этот вывод подтверждается как теоретическим анализом, так и практическим опытом масштабных проектов по разработке игр.

Выводы

1. Анализ классических алгоритмов поиска пути, лежащих в основе более сложных версий алгоритмов поиска пути, показал, что в сфере разработки компьютерных игр алгоритм поиска пути A^* является наиболее эффективным за счёт эвристической функции, направляющей игровых агентов, и в следствии чего, существенно ускоряющей работу алгоритма.

2. Исследование параллельных и распределенных подходов показало значительное повышение производительности алгоритмов поиска пути на современных CPU и GPU. Параллельный A^* демонстрирует наибольший прирост скорости за счет вычисления пути с обеих сторон одновременно,

поэтому алгоритм A^* с параллельным поиском пути является основой для построения иерархических алгоритмов поиска пути на всех уровнях графа.

3. Анализ различных вариантов алгоритма иерархического поиска пути позволил выделить их основные преимущества: масштабируемость и эффективность для крупномасштабных сред. Это показывает, что иерархический поиск пути с использованием параллельного A^* с на всех уровнях графа представляет собой оптимальное решение для поиска пути в больших пространствах.

2. Анализ существующих подходов и методов параллельного вычисления

2.1 Существующие подходы к параллельному программированию в игровом поиске пути

Параллельное программирование стало необходимым для разработки современных игр, особенно для систем поиска пути, которые должны работать со сложным окружением и несколькими агентами одновременно. В этом разделе рассматриваются три основные парадигмы параллелизма и их применение к алгоритмам поиска пути.

Параллелизм задач

Параллелизм задач разделяет рабочую нагрузку на отдельные задачи, которые могут выполняться параллельно. В поиске пути этот подход проявляется несколькими способами:

1. Независимые запросы пути

Несколько запросов на поиск пути могут обрабатываться одновременно на разных ядрах процессора. Каждый запрос представляет собой отдельную задачу, которая может выполняться без помех для других.

Этот подход особенно эффективен для игр с множеством независимых агентов, таких как стратегические игры в реальном времени или крупномасштабные симуляторы. Запрос на поиск пути для каждого агента становится отдельной задачей, которую можно планировать независимо.

Основная проблема заключается в балансировке нагрузки - обеспечении эффективного распределения вычислительных ресурсов между задачами.

Методы динамического планирования корректируют распределение ресурсов в зависимости от сложности задачи и доступности процессора.

2. Параллелизм конвейера

Процесс поиска пути можно разбить на этапы (предварительная обработка, исследование, реконструкция пути), которые образуют конвейер. Различные этапы могут параллельно обрабатывать разные запросы.

Конвейерный параллелизм позволяет эффективно использовать разнородные вычислительные ресурсы. Например, при предварительной обработке могут быть задействованы ядра CPU, а при исследовании - вычислительные возможности GPU.

В современных игровых движках используются сложные конвейерные архитектуры, которые минимизируют накладные расходы на синхронизацию и одновременно увеличивают пропускную способность. Зачастую такие конвейеры включают в себя спекулятивное выполнение для того, чтобы скрыть задержки и поддерживать производительность в реальном времени.

3. Спекулятивный поиск пути

Спекулятивный поиск заключается в одновременно исследовании несколько потенциальных путей. По завершению поиска выбирается лучший результат. Этот подход особенно ценен для динамических сред, где условия могут меняться в процессе поиска пути.

Спекулятивные методы могут параллельно исследовать несколько эвристических функций или наборов параметров, выбирая наилучший результат в зависимости от текущих условий. Такая надежность достигается ценой дополнительных вычислений, но может значительно улучшить качество пути в сложных сценариях.

В продвинутых реализациях используется машинное обучение для предсказания того, какие спекулятивные пути с наибольшей вероятностью будут успешными, что позволяет направить вычислительные ресурсы на перспективные направления [27,28].

Параллелизм данных

Параллелизм данных сосредоточен на распределении элементов данных между процессорами, при этом каждый процессор выполняет одну и ту же операцию над разными подмножествами данных.

1. Разбиение на сетки

Игровая карта делится на регионы, и каждый процессор отвечает за исследование определенного региона. Этот подход хорошо работает с методами исследования на основе границ.

Методы пространственного разбиения включают равномерное деление сетки и декомпозицию. Каждый метод предлагает различные компромиссы между балансировкой нагрузки и сложностью границ.

Основная проблема при разбиении сетки заключается в обработке границ между регионами. Эффективные механизмы синхронизации обеспечивают согласованность на границах регионов, минимизируя при этом коммуникационные накладные расходы. В современных реализациях используется адаптивное разбиение, которое регулирует размеры регионов в зависимости от сложности вычислений.

Области со сложными навигационными характеристиками получают более тонкое разбиение, в то время как открытые области используют более крупные разделы [29].

2. Параллелизм на уровне узлов

Во время исследования A^* несколько узлов на одной границе могут быть расширены одновременно, при этом синхронизация потоков обеспечивает согласованность.

Такой мелкозернистый параллелизм особенно подходит для реализации на GPU, где тысячи потоков могут одновременно обрабатывать расширения узлов. Специализированные структуры данных, такие как приоритетные очереди без блокировок, позволяют эффективно выполнять параллельные операции. Эффективность параллелизма на уровне узлов зависит от коэффициента

ветвления пространства поиска. Среды с высокой связностью выигрывают от этого подхода больше, чем ограниченные линейные среды.

В продвинутых реализациях используется спекулятивное выполнение для одновременного изучения нескольких перспективных узлов, даже если их точный порядок приоритетов не определен. Такой подход позволяет обменивать потенциально избыточные вычисления на снижение накладных расходов на синхронизацию.

3. Пакетная обработка

Группы соседних узлов могут оцениваться параллельно, что особенно полезно для расчетов стоимости и эвристических оценок. Пакетная обработка использует возможности SIMD (Single Instruction, Multiple Data) современных процессоров и массивно-параллельной архитектуры графических процессоров. Благодаря обработке нескольких узлов с помощью одних и тех же инструкций этот подход позволяет достичь высокой вычислительной эффективности.

Этот подход особенно эффективен для эвристик, требующих больших вычислительных затрат, или сложных функций стоимости. Например, анализ местности, расчеты прямой видимости или оценка стоимости движения на основе физики могут быть значительно ускорены за счет пакетной обработки.

В игровых движках часто применяются гибридные подходы, сочетающие пакетную обработку для однородных операций с более гибким параллелизмом задач для разнородных операций [30].

4. Пространственное хэширование

Параллельное вычисление пространственных хэш-функций позволяет эффективно находить соседей и обнаруживать столкновения. Эта техника особенно ценна для динамических сред, где пространственные отношения часто меняются. Назначая пространственные области различным хэш-бакам, подход распределяет рабочую нагрузку между процессорами, минимизируя межпроцессорные зависимости. Такая естественная декомпозиция обеспечивает

эффективную параллельную реализацию с минимальными накладными расходами на синхронизацию.

Современные игровые движки реализуют сложные схемы пространственного хеширования, которые адаптируются к характеристикам среды и распределению агентов. Эти адаптивные подходы сохраняют производительность даже при изменении условий игры.

Параллелизм на уровне инструкций

Параллелизм на уровне инструкций использует способность современных процессоров выполнять несколько инструкций одновременно.

1. Векторизация

Пространственные операции, такие как вычисление расстояний и эвристические оценки, могут быть векторизованы с помощью инструкций SIMD.

Современные процессоры поддерживают различные наборы SIMD-инструкций (SSE, AVX, NEON), которые позволяют обрабатывать несколько элементов данных с помощью одной инструкции. Правильно векторизованный код поиска путей позволяет добиться 4-16-кратного повышения производительности вычислительных ядер.

- Обычные цели векторизации при поиске пути включают:
- Расчет расстояния между точками
- Эвристические оценки для нескольких узлов
- Обнаружение столкновений с многочисленными препятствиями
- Расчет стоимости для всех типов местности

Эффективная векторизация требует тщательной компоновки данных и разработки алгоритмов. Алгоритмы, использующие векторизацию, должны быть перестроены таким образом, чтобы оперировать со смежными блоками данных, а не с разрозненными отдельными элементами.

2. Оптимизация предсказания ветвлений

Алгоритмы поиска пути могут быть реструктурированы, чтобы заменить условные ветви операциями, зависящими от данных. К таким методам относятся:

- Предикативное выполнение с использованием операций по маске
- Подходы, основанные на таблицах, которые заменяют ветвления поисками
- Сортировка узлов для повышения точности предсказания ветвей
- Реализации приоритетных очередей без ветвления

Эти оптимизации особенно важны для внутренних циклов алгоритмов поиска пути, которые выполняются миллионы раз в течение типичной игровой сессии.

3. Паттерны доступа к памяти

Проектирование структур данных для использования локальности кэша и механизмов предварительной выборки. Доступ к памяти часто является основным узким местом в алгоритмах поиска пути. Тщательное проектирование структуры данных может значительно повысить производительность за счет:

- Группировка часто используемых данных вместе
- Выравнивание структур данных по границам строк кэша
- Использование шаблонов обхода, дружественных к кэшу
- Использование программной и аппаратной предварительной выборки

Представления графиков можно оптимизировать с помощью таких методов, как массивы смежности или пространственное хеширование, чтобы улучшить доступ к памяти.

2.2 Архитектуры параллельных вычислений

Эффективность параллельного поиска путей в значительной степени зависит от базовой аппаратной архитектуры. Основными доступными платформами являются многоядерные CPU, GPU и распределенные кластеры.

Многоядерные центральные процессоры (CPU)

Современные процессоры, как правило, имеют несколько ядер (от 2-4 в бюджетных системах до 16, 32 и даже более в высокопроизводительных настольных компьютерах и серверах). Эти ядра могут одновременно выполнять различные потоки инструкций. Обычно они имеют общий доступ к оперативной памяти (RAM) через иерархию кэшей, образуя систему симметричной многопроцессорной обработки (SMP).

CPU обладает рядом преимуществ, относительно других архитектур параллельных вычислений. Во-первых, относительно простое программирование с использованием моделей общей памяти (например, OpenMP или потоковых библиотек), во-вторых, хороший параллелизм задач и умеренный параллелизм данных (на уровне агентов), в-третьих, CPU имеют отличную поддержку со стороны операционных систем и инструментов разработки.

Основными недостатками архитектуры на основе многоядерных процессоров является ограниченное количество ядер по сравнению с графическими процессорами. Протоколы когерентности кэша могут создавать накладные расходы, когда несколько ядер часто обращаются к общим данным (например, к центральному навигационному графу или общим открытым спискам при параллелизме в пространстве поиска).

Графические процессоры (GPU)

Изначально созданные для рендеринга графики, GPU превратились в массивно-параллельные процессоры с сотнями или тысячами более простых ядер, оптимизированных для одновременного выполнения одной и той же инструкции над несколькими элементами данных (Single Instruction, Multiple Data - SIMD, или, более точно, Single Instruction, Multiple Threads - SIMT).

Преимуществами GPU являются чрезвычайно высокая пиковая вычислительная производительность для параллельных задач. Идеально подходит для параллелизма на уровне агентов, когда тысячи простых задач поиска пути (возможно, использующих упрощенные алгоритмы или

работающих на сетках) могут выполняться параллельно. Также может использоваться для параллелизма в пространстве поиска, если алгоритм может быть адаптирован к модели SIMT (например, параллельное расширение узлов на сетке).

Несмотря на высокую производительность, GPU так же обладает рядом недостатков: модель программирования (CUDA, OpenCL), как правило, сложнее, чем потоковая обработка на CPU. Передача данных между основной памятью CPU и памятью GPU может быть узким местом. Менее эффективны для сложной, ветвящейся логики в рамках одной задачи или для параллелизма задач. Лучше всего подходит для алгоритмов с высокой интенсивностью арифметических операций и регулярным доступом к памяти [14].

Кластеры / распределенные системы

Состоят из нескольких независимых компьютеров (узлов), соединенных сетью. Каждый узел имеет собственный процессор (процессоры) и память (распределенную память). Параллелизм достигается за счет распределения задач и данных по узлам и координации с помощью сетевых сообщений.

Этот подход к построению параллельных систем является довольно редко применяемым из-за его недостатков: сетевое взаимодействие влечет за собой значительные задержки и накладные расходы по сравнению с доступом к общей памяти. Сложное программирование, обычно требующее библиотек передачи сообщений (например, MPI). Менее распространен для типичного поиска пути в играх на стороне клиента, более актуален для крупных серверных симуляторов или бэкендов ММО. К преимуществам можно отнести высокую масштабируемость до потенциально огромного числа процессоров/узлов. Анализ платформ представлен в таблице 2.1.

Таблица 2.1 – Сравнение параллельных платформ для поиска пути

Платформа	Преимущества	Недостатки	Подходит для задач
CPU	Универсальность, простая программная модель (OpenMP), доступность, хорошая поддержка	Ограниченное количество ядер, меньшая вычислительная мощность на ядро, чем GPU	Поиск пути для умеренного количества агентов, параллелизм задач, умеренно сложные миры
GPU	Массивный параллелизм, высокая вычислительная мощность, высокая пропускная способность памяти	Более сложная программная модель (CUDA, OpenCL), ограниченная универсальность, задержки передачи данных CPU-GPU	Поиск пути с высокой степенью параллелизма данных (BFS, этапы A*), большие миры, большое количество агентов, задачи, интенсивно использующие память
Кластеры	Масштабируемость, распределенная память, огромная вычислительная мощность	Сложность программирования (MPI), накладные расходы на коммуникацию, сложность управления	Очень большие и динамичные миры, задачи, требующие огромных вычислительных ресурсов, обучение моделей поиска пути

Большинство разработчиков компьютерных игр стараются сделать свой мир большим, но при этом не слишком. Это связано с тем, что при создании действительно гигантского мира довольно тяжело наполнить его интересным и уникальным контентом для игроков. Следовательно, для задач поиска пути в компьютерных играх чаще всего используются CPU, чуть реже GPU и очень редко кластеры. Это происходит из-за того, что CPU и GPU хватает для большинства игр. Выбор между CPU и GPU зависит от конкретных требований задачи, типа используемого параллелизма и доступных ресурсов: для задач, которые могут быть успешно распараллелены на уровне данных, таких как некоторые этапы алгоритмов BFS и A*, GPU могут обеспечить значительную скорость. Для задач, которые больше подходят для распараллеливания на уровне задач или требуют большей универсальности и гибкости, лучшим выбором может стать многоядерный CPU. В некоторых случаях может быть эффективна гибридная система, сочетающая CPU и GPU, в которой CPU используется для обработки всего процесса поиска пути и распределения задач, а GPU – для ускорения более интенсивных вычислительных этапов [33-34].

2.3 Модели параллельного программирования

Чтобы использовать возможности параллельных архитектур, используются специальные модели программирования и API. Они предоставляют абстракции для создания, управления и синхронизации параллельных задач.

OpenMP (Open Multi-Processing)

API, основанный на директивах компилятора (`#pragma omp ...`), в основном для C, C++ и Fortran. Он упрощает разработку параллельных приложений для архитектур с общей памятью (многоядерных процессоров).

Преимуществами данной модели являются относительно простое инкрементальное внедрение параллелизма в существующий последовательный

код, особенно при распараллеливании циклов. А также переносимость между различными архитектурами процессоров и компиляторами, поддерживающими стандарт. Большая часть управления потоками выполняется автоматически.

Если говорить о недостатках, то можно выделить то, что OpenMP предназначен в первую очередь для систем с общей памятью (CPU). Ограниченный контроль над отображением и планированием потоков по сравнению с явными потоками. Менее подходит для неравномерных или сложных зависимостей задач.

MPI (Message Passing Interface)

Стандартизированная спецификация библиотеки для написания программ передачи сообщений, обычно используемых в системах с распределенной памятью (кластерах). Процессы общаются между собой, явно отправляя и получая сообщения.

Преимущества MPI – это высокая масштабируемость для большого количества процессоров/узлов. Стандарт для кластеров и высокопроизводительных вычислений (HPC) обеспечивает тонкий контроль над коммуникациями между связанными в кластер компьютерами и синхронизацией [35].

Основным недостатком является явное управление коммуникациями значительно усложняет программирование по сравнению с моделями с общей памятью. Также высокая задержка связи может стать узким местом в производительности. В компьютерных играх MPI применяется крайне редко, так как для большинства задач использование кластеров излишне.

CUDA (Compute Unified Device Architecture)

Собственная платформа параллельных вычислений и модель программирования NVIDIA для своих GPU. Она использует расширения языка C/C++ (и других языков) для определения "ядер", выполняемых множеством потоков GPU [36].

В настоящее время CUDA является основной моделью параллельного программирования, работающей с графическими процессорами. CUDA отличается высокой производительностью на графических процессорах NVIDIA, развитая экосистемой с обширными библиотеками (cuBLAS, cuDNN и т.д.), а также относительная простота для разработчиков, знакомых с C/C++.

Несмотря на высокую производительность и частые обновления, при написании кода для параллелизации на CUDA ваша игра будет привязана к поставщику (только NVIDIA). К счастью, видеокарты от NVIDIA стоят в большинстве современных компьютеров. Кроме того, требуется тщательное управление передачей данных между памятью CPU и GPU. Также разработка эффективных CUDA-приложений требует глубокого понимания архитектуры GPU и принципов параллельного программирования, а также освоения специализированных программных интерфейсов.

OpenCL (Open Computing Language)

Открытый стандарт для написания программ, которые выполняются на гетерогенных платформах, состоящих из CPU, GPU, DSP, FPGA и других процессоров.

Основным достоинство OpenCL является нейтральность к поставщикам и переносимость на аппаратное обеспечение различных производителей (NVIDIA, AMD, Intel, ARM и т.д.). Если для вашей задачи не нужна запредельная скорость вычислений, то используя данную модель, можно не беспокоиться о том, что игроки будут жаловаться на проблемы с драйверами или не очень популярными комплектующими. Поддерживает модели параллельного программирования данных и параллельного программирования задач [37].

Недостатками модели OpenCL являются то, что: API может быть более многословным и сложным, чем CUDA, производительность может значительно отличаться у разных производителей. Часто воспринимается как более сложная для обучения, чем CUDA или OpenMP. Так что при использовании данной

модели необходимо иметь более квалифицированную команду разработчиков. Анализ моделей представлен в таблице 2.2

Таблица 2.2 – Сравнение программных моделей для поиска пути

Модель	Преимущества	Недостатки	Подходит для платформ
OpenMP	Простота использования, переносимость, эффективность для shared-memory систем	Ограничения для distributed-memory систем, менее эффективен для массового параллелизма данных, чем GPU	Многоядерные CPU
MPI	Масштабируемость для distributed-memory систем, гибкость и контроль	Сложность программирования, накладные расходы на коммуникацию	Кластеры
CUDA	Высокая производительность на GPU NVIDIA, прямой доступ к аппаратным ресурсам GPU	Ограничение аппаратной платформой (NVIDIA), более сложная программная модель, чем OpenMP	GPU NVIDIA
OpenCL	Переносимость, гибкость (поддержка CPU, GPU и других ускорителей)	Может быть менее производительным, чем CUDA на GPU NVIDIA	CPU, GPU (NVIDIA, AMD, Intel)

Выбор программной модели зависит от архитектуры вычислительной системы, на которой будет выполняться поиск пути, и от таких требований, как

переносимость, производительность и простота разработки. OpenMP часто используется для многоядерных CPU из-за своей простоты и эффективности; для графических процессоров NVIDIA, CUDA предлагает наилучшую производительность, но имеет ограниченную переносимость, OpenCL предлагает компромисс между переносимостью и производительностью и является хорошим выбором для разработки кроссплатформенных решений, которые могут использовать как CPU, так и GPU разных производителей. MPI может использоваться на многих узлах с используется в кластерных системах, где требуется масштабируемость [38].

2.4 API для параллельного программирования в средах разработки игр Стек технологий, ориентированных на данные Unity (DOTS)

Unity DOTS представляет собой комплексную структуру для параллельных вычислений:

1. Unity Job

Высокоуровневый параллелизм задач с автоматическим разрешением зависимостей и кражей работы. Unity Job обеспечивает структурированный подход к параллельному выполнению:

- Объекты JobHandle представляют асинхронные операции
- Спецификации зависимостей обеспечивают правильный порядок выполнения
- Автоматическое планирование и балансировка нагрузки

Реализации поиска пути выигрывают от этой системы за счет упрощенного распараллеливания независимых задач, таких как пакетные запросы пути или многоуровневые операции поиска пути.

Функции безопасности предотвращают распространенные ошибки параллелизма с помощью проверок во время компиляции:

- Проверка доступа на чтение/запись для общих данных

- Автоматическое обнаружение условий гонки
- Управление сроком службы ресурсов

2. Burst Compiler

Передовая технология компиляции, генерирующая высокооптимизированный нативный код из C#. Burst транслирует код на C# в высокопроизводительный нативный код:

- Оптимизация с учетом особенностей архитектуры
- Автоматическая векторизация для SIMD-инструкций
- Агрессивная инкрустация и постоянное распространение

Алгоритмы поиска пути значительно выигрывают от этих оптимизаций, а их производительность во многих случаях приближается к производительности написанного вручную C++ или даже превосходит ее.

Функции безопасности Burst поддерживают надежность управляемого кода, обеспечивая при этом собственную производительность:

- Устранение проверки границ для валидированного кода
- Прямой доступ к памяти для критически важных операций
- Детерминированное выполнение для предсказуемого поведения

3. Система компонентов сущностей (ECS)

Ориентированный на данные дизайн, позволяющий эффективно параллельно обрабатывать игровые сущности. ECS организует игровые данные для оптимального доступа к памяти:

- Сущности как легкие идентификаторы
- Компоненты как контейнеры для чистых данных
- Системы как логика, оперирующая данными компонентов

Такая организация обеспечивает эффективную параллельную обработку данных поиска пути:

- Компоненты запроса пути обрабатываются партиями
- Навигационные данные, организованные для удобного доступа к кэшу

- Параллельные системы для различных этапов поиска пути

Хранилище на основе архетипов автоматически группирует связанные компоненты, улучшая локальность памяти для таких операций, как пакетная обработка путей.

4. Пакет Collections

Высокопроизводительные контейнеры, оптимизированные для параллельного доступа. Пакет Collections в Unity предоставляет специализированные структуры данных:

- NativeArray для непрерывных данных с контролируемым доступом
- NativeHashMap для эффективного поиска ключевых значений
- NativeQueue для распределения работы

Эти коллекции поддерживают специфические требования к поиску пути:

- Детерминированное распределение и деаллокация
- Безопасность потоков и параллельный доступ
- Выравнивание памяти для векторизованных операций

Атомарные операции позволяют без блокировки обновлять общее состояние поиска пути, что улучшает масштабируемость в многоагентных сценариях.

5. Математический пакет

Векторизованные математические операции для пространственных расчетов. Пакет Mathematics обеспечивает операции с SIMD-ускорением:

- Векторные и матричные операции для пространственных преобразований
- Математика кватернионов для вычисления ориентации
- Геометрические примитивы и тесты на пересечение

Поиск пути выигрывает от векторной реализации общих операций:

- Расчет расстояния между точками
- Расчеты ориентации и выравнивания
- Испытания на столкновение и видимость

Эти операции автоматически используют доступные SIMD-инструкции на различных аппаратных платформах [39].

Система Task Graph Unreal Engine

Unreal Engine обеспечивает сложный параллелизм на основе задач:

1. Task Graph

Планирование задач на основе зависимостей с автоматическим распределением нагрузки. Система Task Graph в Unreal управляет сложными зависимостями:

- Задачи представляют собой единицы работы с определенными зависимостями
- Автоматическое планирование на основе разрешения зависимостей
- Динамическое распределение работы для балансировки нагрузки

2. Система асинхронных задач

Высокоуровневые абстракции для асинхронных вычислений. Фреймворк асинхронных задач Unreal упрощает параллельный поиск путей:

- Фьючерсы и обещания для обработки асинхронных результатов
- Продолжение прохождения для построения технологических цепочек
- Поддержка отмены для отказа от устаревших запросов пути

Эти абстракции обеспечивают отзывчивый поиск пути, который не блокирует игровой поток:

- Первоначальная оценка пути выполняется быстро
- Постепенное совершенствование по мере завершения вычислений
- Мягкая обработка прерванных запросов

Интеграция с системой событий движка позволяет автоматически передавать результаты поиска путей в зависимые системы.

3. Параллельный рендеринг

Выделенный поиск путей позволяет сократить время простоя потоков рендеринга. Многопоточная архитектура рендеринга Unreal позволяет использовать ресурсы потоков рендеринга для поиска путей:

- Вычисления поиска пути планируются во время простоя рендеринга
- Общие ресурсы GPU для визуализации и вычислений
- Координированный доступ к памяти между рендерингом и поиском пути

Такая интеграция позволяет эффективно использовать ресурсы всех доступных вычислительных устройств.

4. ParallelFor

Упрощенный интерфейс для параллельных операций с данными. Утилита ParallelFor обеспечивает простое распараллеливание операций, основанных на циклах:

- Автоматическое распределение работы между доступными ядрами
- Дополнительное управление гранулярностью для балансировки накладных расходов

Эта утилита упрощает распараллеливание распространенных операций поиска пути:

- Пакетная обработка запросов на прохождение пути
- Расширение параллельных узлов

Функции мониторинга производительности помогают выявить возможности оптимизации и узкие места [40].

Система Task Graph Unreal Engine

Godot предоставляет возможности как высокоуровневого, так и низкоуровневого параллельного программирования:

1. Thread Class

Прямое управление потоками для пользовательского распараллеливания. Класс Thread в Godot обеспечивает явный контроль над созданием и управлением потоками:

- Создание потока с заданным приоритетом
- Прimitives ожидания и синхронизации
- Безопасное завершение работы и очистка

Этот низкоуровневый доступ позволяет использовать специальные стратегии распараллеливания для решения специализированных задач поиска пути:

- Выделенные потоки поиска пути для непрерывной фоновой обработки
- Индивидуальное распределение работы для гетерогенных рабочих нагрузок
- Специализированные протоколы синхронизации для сложных зависимостей

2. Поточковая обработка на базе сервера

Выделенные потоки для физики, рендеринга и обработки звука. Архитектура Godot разделяет основные системы движка на серверные потоки:

- Сервер физики для обнаружения столкновений и динамики
- Сервер рендеринга для обработки графики
- Аудиосервер для обработки звука

3. Пул рабочих потоков

Управляемый пул потоков для фоновых задач. Пул рабочих потоков Godot обеспечивает упрощенный доступ к фоновой обработке:

- Представление задач с указанием приоритетов
- Автоматическое распределение нагрузки по доступным ядрам
- Отслеживание прогресса и уведомление о завершении

Операции по поиску пути выигрывают от такого управляемого подхода:

- Пакетная обработка запросов на прохождение пути
- Анализ фоновой среды
- Асинхронная оптимизация пути

Пул управляет созданием, планированием и синхронизацией потоков, упрощая реализацию и сохраняя эффективность.

4. Мьютекс и семафор

Примитивы синхронизации для координации параллельных операций. Godot предоставляет стандартные примитивы синхронизации:

- Мьютекс для защиты общих ресурсов
- Семафор для передачи сигналов между потоками
- RWLock для координации действий читателя и писателя

Эти примитивы позволяют безопасно координировать параллельные действия по поиску пути:

- Защищенный доступ к общим навигационным данным
- Синхронизированные обновления для динамических сред
- Координировал пакетную обработку заявок на проезд

Таблица 2.3 – Сравнение программных моделей для поиска пути

Игровая среда разработки	Модель многопоточности	Сложность использования	Производительность
Unity DOTS	Job System	Высокая	Очень высокая
Unreal Engine	Task Graph	Умеренная	Высокая
Godot	Ручное управление потоками	Умеренная	Средняя

Каждый игровой движок предлагает различные подходы к параллельному программированию, что отражает их архитектурную философию и целевые сценарии использования.

Unity DOTS представляет собой значительный архитектурный сдвиг в сторону проектирования, ориентированного на данные, и явного параллелизма.

Сочетание системы заданий, компилятора Burst и ECS обеспечивает исключительную производительность операций поиска пути при сохранении безопасности и простоты программирования.

В подходе Unreal Engine особое внимание уделяется интеграции с существующими системами и гибкости программистов. Система Task Graph обеспечивает сложное управление зависимостями, сохраняя при этом совместимость с устоявшимися шаблонами движка.

Godot предлагает более традиционный подход к потокам, предоставляя прямой доступ к примитивам потоков наряду с упрощенными абстракциями. Такая гибкость позволяет разработчикам выбирать предпочтительный уровень абстракции в зависимости от конкретных требований к поиску путей [71].

Выводы

1. Исследование архитектур параллельных вычислений показало, как различные аппаратные конфигурации могут быть использованы для решения конкретных задач поиска пути. CPU справляются со сложной алгоритмической логикой, GPU обеспечивают массивный параллелизм для обработки больших объемов данных, а кластеры позволяют масштабировать очень большие игровые миры.

2. Анализ технологий параллельного программирования показал, что к основным их возможностям относятся простое написание параллельных вычислений и одновременное применение нескольких технологий для разных аппаратных конфигураций. В качестве их недостатков следует указать усложнение архитектуры игры и возможные ошибки у игроков, с редко используемыми деталями компьютеров. Сравнение показало, что выбор технологии зависит от целевого оборудования, опыта команды разработчиков и конкретных требований к производительности.

3. Исследование API игровых движков с целью оценки возможностей современных сред разработки абстрагировать сложность параллельного программирования, что позволяет не сталкиваться с низкоуровневыми деталями работы с параллельными вычислениями, а организовывать распределение задач, синхронизацию данных и оптимизацию ресурсов, используя удобный интерфейс. Unity DOTS наиболее адекватно соответствует требованиям, предъявляемым к игре, использующей более тысячи агентов и карту большой площади, в части:

- высокопроизводительной компиляции;
- эффективной системы заданий с автоматическим распределением нагрузки.

3. Модификации иерархического поиска пути в зависимости от задачи

Поиск пути - процесс нахождения приемлемого маршрута между двумя точками в окружении - является краеугольным камнем искусственного интеллекта (ИИ) в компьютерных играх. Он позволяет неигровым персонажам (NPC) перемещаться по сложным мирам, направляет индикаторы движения игрока и лежит в основе стратегических решений в различных жанрах. Способность разумно и эффективно перемещаться - от солдат, марширующих по полям сражений в стратегиях реального времени (RTS), до горожан, населяющих шумные города в ролевых играх с открытым миром (RPG), и ползущих по дорожкам в многопользовательских онлайн-аренах (MOBA) - является основой для создания правдоподобного и увлекательного виртуального опыта.

Классический алгоритм поиска A^* , а также его разновидности, такие как Jump Point Search (JPS) на сетках, обеспечивают оптимальные или близкие к оптимальным пути в относительно небольших, четко определенных пространствах поиска. A^* работает, исследуя узлы (представляющие такие локации, как ячейки сетки, путевые точки или полигоны навигационной сетки) по направлению от начальной точки, разумно отдавая приоритет узлам, которые кажутся ближе к цели, основываясь на эвристической оценке. Однако по мере того, как игровые миры становятся экспоненциально больше и детальнее - охватывая огромные континенты, запутанные подземелья или разрастающиеся галактики, - вычислительные затраты на применение A^* непосредственно ко всему низкоуровневому представлению ("плоской" карте) становятся непомерными.

Поиск на карте сетки с миллионами или даже миллиардами узлов с помощью стандартного A^* может привести к значительным ограничениям производительности. Алгоритму может потребоваться исследовать огромное количество узлов, прежде чем он найдет цель, что потребует значительных затрат процессорного времени и памяти. Такая задержка часто неприемлема в

играх реального времени, где десятки или сотни агентов могут требовать пути одновременно. Кроме того, хранение всего пространства поиска и связанных с ним структур данных A^* (открытый список, закрытый список) для множества одновременных запросов пути может исчерпать доступные ресурсы памяти, особенно на платформах с жесткими ограничениями, например на консолях или мобильных устройствах. Эта проблема масштабируемости требует более сложных подходов, что привело к разработке иерархических методов поиска путей [41].

3.1 Недостатки иерархического поиска пути при решении узконаправленных игровых задач

Hierarchical Path A^* (HPA) использует регулярную структуру сеток для создания двухуровневой иерархии, что значительно ускоряет поиск пути на больших однородных картах-сетках, часто встречающихся в играх на основе плитки, RTS и симуляторах [42].

Фаза предварительной обработки HPA*

Эффективность HPA* в значительной степени зависит от обширной фазы предварительной обработки, выполняемой перед запуском игры (или во время загрузочных экранов). На этом этапе создается абстрактное представление мира сетки.

1. **Разбивка сетки:** карта первичной сетки делится на множество более мелких, обычно квадратных, смежных областей, называемых "кусками" или "кластерами". Размер этих фрагментов (например, 10x10, 20x20 ячеек) является критическим параметром. Меньшие куски приводят к большему абстрактному графу, но более простому поиску путей внутри кусков, в то время как большие куски приводят к меньшему абстрактному графу, но более сложным и дорогостоящим расчетам внутри кусков. Выбор зависит от размера карты, ожидаемой длины пути и ограничений памяти.

2. **Определение входов (переходов):** алгоритм определяет все "входы" между соседними фрагментами. Вход определяется как пара смежных, проходимых ячеек сетки, где каждая ячейка принадлежит другому, смежному кластеру. Например, если кластер А находится непосредственно над кластером В, то любая пара ячеек (x, y) в А и $(x, y+1)$ в В, где обе ячейки проходимы, представляет собой вертикальную точку входа. Аналогично, горизонтальные входы существуют между кусками, расположенными рядом друг с другом. Эти входы служат шлюзами для перемещения между фрагментами [43].

3. **Построение внутрикластерных путей:** для каждого куска НРА* предварительно вычисляет и сохраняет оптимальные пути между всеми парами входов, расположенных на границе этого куска. Обычно это делается путем выполнения A^* (или аналогичного сеточного поиска, например Breadth-First Search, если стоимость ребер равномерна) в пределах кластера, начиная с одного узла входа и находя кратчайший путь ко всем другим узлам входа на границе этого кластера. Стоимость (длина) каждого пути сохраняется. Это часто самая трудоемкая часть предварительной обработки.

4. **Построение абстрактного графа $G = (V, E)$.**

- Узлы (V): Узлы в абстрактном графе обычно представляют входы, определенные на шаге 2. Каждый узел входа фактически принадлежит границе между двумя конкретными блоками.
- Края (E): добавляются два типа краев:

Внутрикластерные ребра: если два входа, e_1 и e_2 , принадлежат одной и той же границе кластера, между соответствующими узлами в абстрактном графе добавляется ребро. Вес этого ребра – это стоимость оптимального пути между e_1 и e_2 в пределах данного кластера, вычисленная на шаге 3.

Межкластерные грани (переходы): если два узла входа, e_1 и e_2 , представляют одно и то же место границы, но рассматриваются из двух соседних кластеров (например, e_1 - выход из кластера А, e_2 - вход в кластер В в том же месте), между ними добавляется ребро. Это ребро представляет собой простой

акт пересечения границы. Его вес обычно равен стоимости перемещения между двумя соседними клетками, образующими вход (часто просто 1, если клетки имеют одинаковую стоимость перемещения). Иногда абстрактный граф упрощается, когда один узел представляет граничную точку, неявно обрабатывая переход. Другой распространенный подход заключается в том, что узлы представляют кластеры, а ребра - путь минимальной стоимости между любым входом в один кластер и любым входом в соседний кластер, хотя граф на основе входа является более точным. В этом обсуждении мы будем рассматривать в первую очередь абстрактный граф на основе входа.

5. Хранение данных: предварительно вычисленные данные - структура абстрактного графа (узлы, ребра, веса) и подробные последовательности узлов для всех предварительно вычисленных внутрикластерных путей - должны храниться эффективно. Для этого может потребоваться значительный объем памяти, особенно для подробных путей. Можно использовать такие методы, как сжатие путей или хранение только важных путевых точек.

Фаза поиска пути во время выполнения НРА*

После завершения предварительной обработки поиск пути от начальной ячейки S до целевой ячейки G выполняется следующим образом:

1. **Определение начальный/конечный куски.** Сначала определяем куски C_s и C_g , содержащие S и G соответственно.
2. **Подключение стартовый узел (S) к абстрактному графу:**
 - Определяем все входные узлы (e_s) на границе начального куска C_s .
 - Выполняем локальный поиск A^* в пределах C_s от S до каждого входа e_s .
 - Временно добавляем S в абстрактный граф. Для каждого входа e_s , достижимого из S , добавьте временное ребро из S в абстрактную вершину, представляющую e_s , с весом, равным стоимости, найденной локальным поиском A^* .

3. **Подключаем узел цели (G) к абстрактному графу:**

- Определяем все входные узлы (e_g) на границе целевого куска C_g .
- Выполняем локальный поиск A^* в пределах C_g от каждого входа e_g в G . (Альтернативный вариант - поиск от G до всех e_g , если ребра графа обратимы).
- Временно добавляем G в абстрактный граф. Для каждого входа e_g , который может достичь G , добавьте временное ребро из абстрактной вершины, представляющей e_g , в G , с весом, равным стоимости, найденной локальным поиском A^* .

4. **Высокоуровневый поиск:** выполняем A^* на дополненном абстрактном графе (включая временные S , G и соединительные ребра), чтобы найти кратчайший путь из S в G . Этот поиск работает на гораздо меньшем абстрактном графе, что значительно быстрее, чем поиск в полной сетке. Эвристика, используемая для A^* в абстрактном графе, обычно представляет собой евклидово или манхэттенское расстояние между физическими местоположениями абстрактных узлов (входов).

5. **Низкоуровневая реконструкция пути:** путь, найденный в абстрактном графе, представляет собой последовательность абстрактных узлов (входов) и ребер.

Например: $S \rightarrow e_{s1} \rightarrow e_{s2} \rightarrow \dots \rightarrow e_{g1} \rightarrow e_{g2} \rightarrow G$.

- Отрезок пути от S до e_{s1} соответствует локальному пути, найденному на шаге 2.
- Каждый сегмент между двумя входами в одном чанке (например, $e_{s1} \rightarrow e_{s2}$, если они находятся в одном чанке) соответствует предварительно вычисленному внутрикластерному пути, сохраненному во время препроцессинга. Извлеките этот подробный путь.

- Сегменты, представляющие переходы между кусками (межкластерные ребра), являются простыми перемещениями между соседними ячейками.
- Отрезок пути от e_{g2} до G соответствует локальному пути, найденному на шаге 3.
- Объедините все эти низкоуровневые сегменты пути (последовательности ячеек сетки) вместе, чтобы сформировать окончательный, детальный путь от S до G .

Учет памяти в НРА*

НРА* обменивает время предварительной обработки и память на скорость выполнения. Основными потребителями памяти являются:

Абстрактный график: количество узлов (входов) и ребер зависит от размера кластера и сложности карты. Может быть существенным для больших карт с маленькими кластерами.

Предварительно вычисленные внутрикластерные пути: хранение полной последовательности узлов для каждого пути между каждой парой входов в каждом кластере может потребовать огромного объема памяти. Часто это самый большой компонент памяти. Стратегии оптимизации включают:

- Сохранение путей только частично (например, ключевых путевых точек) и повторное выполнение локального A^* для заполнения пробелов во время выполнения (с потерей некоторой скорости).
- Использование методов сжатия пути.
- Не хранить пути вообще и пересчитывать внутрикластерные пути по требованию во время высокоуровневого поиска (ускоряет препроцессирование и экономит память, но время выполнения медленнее).

НРА* в играх

НРА* хорошо подходит для игр с большим, преимущественно статичным окружением на основе сетки, таких как:

1. Игры в жанре стратегии реального времени (RTS).
2. Плиточные ролевые игры или игры-симуляторы.
3. Логистические или симуляционные игры.

Главный недостаток базового НРА* - его статичность. Дорогостоящая предварительная обработка предполагает, что проходимость окружения (препятствия, стоимость рельефа) не изменится после построения иерархии. Динамические препятствия, возникающие в процессе игры, могут аннулировать предварительно вычисленные пути и абстрактный граф, что может привести к тому, что агенты будут следовать неоптимальным или заблокированным маршрутам [44].

3.2 Статический иерархический поиск пути A* (SHPA*)

Термин "Статический иерархический путь A* (SHPA*)" не так стандартизирован в литературе, как НРА* или ДНРА*. Часто НРА* рассматривается как основной статический иерархический подход на основе решетки из-за его зависимости от обширной предварительной обработки для статической среды. Однако здесь мы можем дать определение SHPA*, чтобы подчеркнуть и потенциально усовершенствовать те аспекты НРА*, которые строго оптимизированы для абсолютно неизменного окружения. SHPA* представляет собой чистейшую форму философии "запечь один раз, использовать вечно" в рамках НРА* [42].

Основные характеристики SHPA*

По сравнению с общим пониманием НРА*, SHPA* предполагает абсолютную неизменность окружающей среды после фазы предварительной обработки. Нет никаких ожиданий или механизмов для обработки динамических изменений, какими бы малыми они ни были. Это позволяет проводить потенциально более агрессивную оптимизацию на этапе предварительной обработки и упрощать логику выполнения.

Отличия от стандартного НРА*

Строгость предварительной обработки: SHPA* может включать в себя еще более исчерпывающую предварительную обработку. Если позволяет память, она будет окончательно вычислять и хранить все оптимальные пути между всеми парами входов в каждом кластере. При этом будет меньше стимулов использовать методы экономии памяти, требующие пересчета во время выполнения (например, хранить только путевые точки или вычислять внутренние пути по требованию), поскольку главная цель - максимальная скорость выполнения в гарантированно статичном мире. Структуры данных могут быть оптимизированы исключительно для скорости чтения, без учета возможных обновлений.

Предположения времени выполнения: фаза поиска путей во время выполнения работает в строгом предположении, что все предварительно вычисленные данные являются достоверными. Нет никаких проверок на наличие динамических препятствий, блокирующих предварительно вычисленные пути или входы. Алгоритм просто извлекает и объединяет предварительно сохраненные сегменты пути. Это делает этап выполнения потенциально более быстрым и простым в реализации, чем более гибкий НРА*, который может включать проверки достоверности.

Использование памяти: из-за стремления хранить все предварительно вычисленные пути в полном объеме для достижения максимальной скорости, SHPA* потенциально может требовать больше памяти, чем варианты НРА*, использующие стратегии экономии памяти. Однако это является компромиссом для гарантированной производительности во время выполнения в статических условиях.

Механизмы обновления: в SHPA* явно отсутствует какой-либо механизм обновления иерархии или предварительно вычисленных путей в ответ на изменения окружающей среды. Если карта изменится (например, из-за обновления дизайна уровня, требующего установки патча), весь этап

предварительной обработки должен быть повторно запущен в автономном режиме [45].

Примеры использования SHPA* в играх:

SHPA* подходит для игровых жанров или сценариев, в которых навигационное пространство гарантированно статично в течение игровой сессии:

1. Головоломки.
2. Некоторые адаптации настольных игр.
3. Готовые окружения (Pre-baked Environments).
4. Сценарные стратегические игры.

Основное ограничение очевидно: SHPA* совершенно не подходит для игр с динамическим окружением, разрушаемой местностью, построенными игроками сооружениями или даже простыми динамическими препятствиями вроде открывающихся/закрывающихся дверей, если они блокируют заранее вычисленные пути. Его применение ограничено сценариями, в которых предположение о "статичности мира" абсолютно верно. Он представляет собой экстремальную точку в пространстве компромиссов: максимальные усилия по предварительной обработке для максимальной скорости выполнения в гарантированно статичном мире [3].

3.3 Динамический иерархический поиск пути A* (DHPA*)

Статическая природа HPA* (и, как следствие, SHPA*) является существенным ограничением для многих современных игр, в которых часто присутствуют динамические элементы, такие как разрушаемая местность, построенные игроком здания, временные препятствия (баррикады, двери) или юниты, которые могут блокировать пути. Динамический иерархический путь A* (DHPA*) относится к семейству методов, которые расширяют HPA* для обработки таких изменений в окружении после начального этапа

предварительной обработки. Основная задача состоит в том, чтобы эффективно обновлять иерархические структуры данных (абстрактный граф, предварительно вычисленные пути), не прибегая к полному пересчету, который был бы слишком медленным для игры в реальном времени [42].

Мотивация и основная идея

Когда в сетке нижнего уровня происходит изменение (например, ячейка блокируется или разблокируется), она может стать недействительной:

1. Локальные пути внутри затронутого кластера(ов).
2. Связность и веса ребер в абстрактном графе, которые опираются на пути, проходящие через пораженный участок (участки).
3. Доступность входов на границе затронутого участка (участков).

ДНРА* стремится обновлять только необходимые части иерархии, в идеале ограничивая вычислительные затраты относительно масштаба изменений.

Механизмы для обработки динамических изменений

Для ДНРА* были предложены различные стратегии, отличающиеся тем, как запускаются и распространяются обновления:

1. **Локальные обновления:** когда обходимость/проходимость ячейки изменяется в пределах куска C .

Признать недействительными внутрикластерные пути: все предварительно вычисленные пути внутри C , которые проходят через измененную ячейку (или, для простоты, все пути внутри C , в зависимости от реализации), помечаются как недействительные.

Обновить абстрактные ребра: необходимо обновить веса ребер абстрактного графа, соответствующих этим недействительным внутрикластерным путям. Для этого необходимо повторно выполнить локальный поиск A^* в пределах C между соответствующими парами входов, избегая вновь измененных клеток. Если путь между двумя входами становится

невозможным, соответствующее абстрактное ребро может быть удалено или иметь бесконечный вес.

Ленивые и нетерпеливые обновления: обновления могут быть оперативными (пересчитывать пути и веса ребер немедленно при изменении) или ленивыми (помечать пути/ребра как недействительные и пересчитывать их только тогда, когда запрос поиска пути действительно должен их использовать). Ленивые обновления распределяют затраты по времени, но могут вносить задержку в запросы поиска пути. Активные обновления требуют предварительных затрат, но обеспечивают постоянную согласованность иерархии [43].

2. **Блокировка входа:** если изменение происходит непосредственно в ячейке входа, делая ее непроходимой.

Соответствующий узел (узлы) в абстрактном графе и все инцидентные ребра должны быть обновлены или удалены. Это эффективно разъединяет два смежных блока в данной конкретной точке.

3. **Распространение изменений:** изменение в одном кластере может повлиять на оптимальный путь между входами в соседних кластерах, если структура абстрактного графа значительно изменится. В некоторых вариантах ДНРА* при определенных условиях может потребоваться распространение обновлений в соседние кластеры, хотя это повышает сложность.

4. **Частичная повторная обработка:** при очень значительных изменениях (например, при масштабном разрушении) постепенное обновление может оказаться менее эффективным, чем простое повторное выполнение шагов предварительной обработки НРА* для затронутого кластера (ов) или даже небольшой области кластеров.

5. **Кэширование:** отслеживайте недавно вычисленные динамические пути. Если окружение часто меняется, поддержание согласованности и избежание чрезмерного пересчета становится критически важным.

Проблемы в ДНРА*

Пересчет внутрикластерных путей и обновление абстрактного графа, даже локальное, может потребовать больших вычислительных затрат, особенно если изменения происходят часто или повсеместно. Выполнение таких обновлений в реальном времени без ущерба для производительности игры является серьезной проблемой. Обновления могут быть разделены по времени или выполняться в отдельных потоках.

Обеспечение точного отражения в абстрактном графе состояния низкоуровневой сетки после динамических изменений - сложная задача. Ленивые обновления могут привести к временной несогласованности состояний.

Реализация эффективных механизмов обновления значительно повышает сложность системы поиска пути по сравнению со статическим НРА*.

Пути, найденные с помощью ДНРА*, могут оказаться неоптимальными по сравнению с плоским поиском А* на текущей динамической карте, особенно если обновления ленивы или приближительны. Обычно цель состоит в том, чтобы быстро найти "достаточно хороший" путь, а не гарантированно оптимальный [47].

Примеры использования ДНРА* в играх

ДНРА* необходим для игр, в которых навигационная среда меняется в процессе игры:

1. Игры в жанре стратегии в реальном времени (RTS).
2. Игры с открытым миром и динамическими элементами.
3. Игры в жанре Tower Defense или МОБА.
4. Игры-симуляции.

Варианты ДНРА* обеспечивают критически важный мост между преимуществами производительности иерархической абстракции и необходимостью работы с динамическими игровыми мирами. Выбор конкретной реализации часто зависит от характера и частоты ожидаемых изменений окружающей среды в игре.

3.4 Сравнительный анализ модификаций иерархического поиска пути для разработки игр

Выбор правильного иерархического подхода к поиску пути во многом зависит от конкретных требований игры, в частности от размера мира и характера динамических изменений. В таблице 3.1 приведены основные характеристики НРА*, SHPA* и DHPA* с точки зрения разработки игр [48].

Таблица 3.1 – Сравнение НРА*, SHPA* и DHPA*

Характеристика	НРА* (общий)	SHPA* (строго статический)	DHPA* (динамический)
Время предварительной обработки	Высокое	Потенциально очень высокое	Высокое (аналогично НРА*)
Использование памяти	Высокое (абстрактный график и пути)	Потенциально очень высокое (полный путь)	Высокое (аналогичный + обновление накладных расходов)
Скорость поиска пути во время выполнения	Очень быстро (для статичных деталей)	Потенциально самый быстрый (без проверок)	Быстро, но потенциально медленнее, чем НРА*/SHPA*, из-за проверок или ленивых обновлений.
Сложность реализации	Умеренная	Умеренная (более простая логика выполнения)	Высокая
Оптимальность пути	Почти оптимально	Почти оптимально	Потенциально субоптимальный (зависит от стратегии и частоты обновления)
Масштабируемость	Средняя	Высокая	Высокая
Соответствие жанру игры	Большие статические миры, RTS (базовые), ролевые игры на основе плитки	Логические игры, настольные игры, полностью готовые уровни	RTS (со строительством/разрушением), открытые миры (с динамическими элементами), Tower Defense

Пояснения к записям в таблице 3.1:

- **Время предварительной обработки:** все иерархические методы требуют значительного времени на создание абстракции в автономном режиме. SHPA* может потратить еще больше времени на предварительный расчет всего возможного.
- **Использование памяти:** хранение абстрактного графа и особенно предварительно вычисленных внутрикластерных путей занимает много памяти. SHPA* может использовать больше всего, если хранит полные, неоптимизированные пути. DHPA* добавляет накладные расходы на управление динамическими обновлениями.
- **Скорость выполнения:** поиск в небольшом абстрактном графе выполняется гораздо быстрее, чем в плоском A*. SHPA* избегает любых динамических проверок, что потенциально делает его абсолютно самым быстрым во время выполнения, в то время как DHPA* несет некоторые накладные расходы на проверку валидности или выполнение ленивых обновлений.
- **Сложность реализации:** базовый HPA* является умеренно сложным. SHPA* может упростить часть времени выполнения. DHPA* добавляет значительную сложность из-за логики обновления.
- **Оптимальность пути:** иерархические пути обычно немного неоптимальны по сравнению с плоским поиском A* на низкоуровневой сетке, поскольку пути ограничены прохождением через заранее определенные входы. DHPA* может стать еще более субоптимальным, если иерархия не идеально отражает последние изменения [49-51].

Выводы

1. Анализ модификаций иерархического поиска пути выявил, что SHPA* является наиболее оптимизированным по таким параметрам как скорость и используемая оперативная память, но из-за отсутствия возможности работать

с изменяемой игровой картой алгоритм DHPA* является более универсальным алгоритмом.

2. Исследование модификаций иерархического поиска показало, что его основными недостатками для применения в игровой индустрии является:

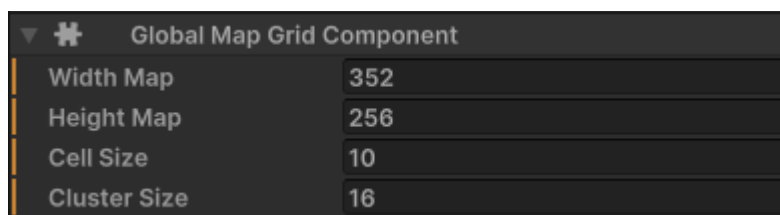
- SHPA* при наилучшей скорости из всех модификаций, не может быть использован во многих играх из-за долгих загрузок при изменении карты
- DHPA* кеширует слишком много путей, из-за чего использование этого алгоритма сразу ставит перед игроками высокие системные требования оперативной памяти

Указанные выше недостатки не позволяют эффективно использовать подобные алгоритмы в играх, нацеленных на широкую аудиторию.

4. Разработка иерархического поиска для большого пространства с использованием параллельного вычисления на кроссплатформенной среде разработки Unity Engine.

4.1 Проектирование игровой карты

Прежде чем приступить к разработке алгоритма, необходимо создать и описать карту, на которой агенты будут находить свой путь. При проектировании карты были поставлены следующие требования: достаточно большая карта для того, чтоб на ней было целесообразно использовать иерархический поиск пути, различие в местности, а также реагирование алгоритма на изменение в карте [52]. На рисунке 4.1 представлены основные данные карты.



Global Map Grid Component	
Width Map	352
Height Map	256
Cell Size	10
Cluster Size	16

Рисунок 4.1 – Основные данные карты

Разработанная карта является глобальной картой для перемещения персонажей игры «Autocrasy», её размер считается в клетках. Клетка – это минимальная единица измерения карты. Размер карты составляет 352 клетки в ширину и 256 клеток в длину. Общая площадь карты составляет 90122 клетки [53].

Деление глобальной карты на кластеры

Карта делится на 352 кластера, по 16 клеток в ширину и в длину. На рисунке 4.1, можно увидеть деление карты на кластеры. Можно заметить, что на рисунке 4.2 кластеров 351, а не 352. Это объясняется тем, что подсчет кластеров в системе начинается с нуля, а не единицы. На рисунке 4.2 представлены деление карты на кластеры.

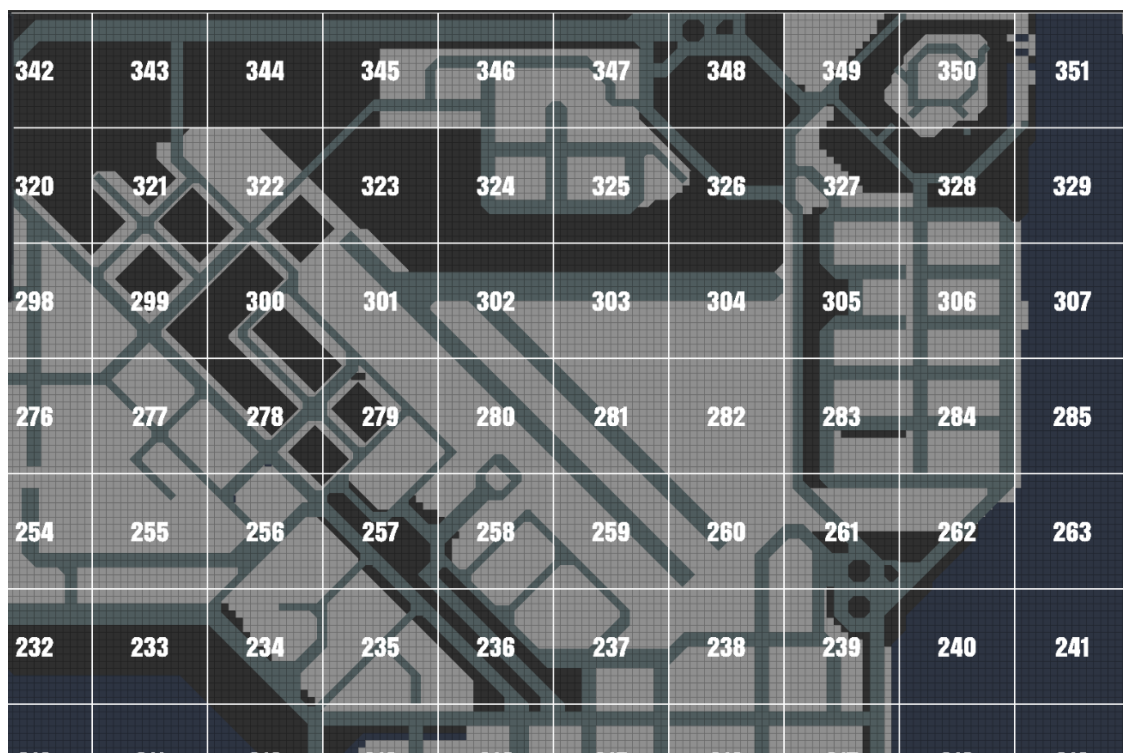


Рисунок 4.2 – Деление карты на кластеры

Обоснование выбора длины и ширины кластера для карты 352×265 .

1. Баланс между производительностью и детализацией

При разбиении карты 352×265 , содержащей 90122 клетки на кластеры, размером 16×16 :

- Количество кластеров по ширине равно: $\frac{352}{16} = 22$;
- Количество кластеров в длину равно $\frac{256}{16} = 16$;
- Итого: $22 \times 16 = 352$ кластера.

Преимуществом такого разбиения является оптимизация поиска пути на верхнем уровне иерархического поиска и минимизация слепых зон внутри одного кластера. Граф верхнего уровня содержит не 90122 узла, а всего 352, что, в свою очередь, сильно сокращает вычислительную сложность алгоритма поиска пути A*. Также кластеры достаточно малы, чтоб можно было избежать ситуаций, когда внутри него образуется несколько зон с непреодолимыми препятствиями. Если подобные ситуации всё равно происходят, то они решаются делением кластеров на зоны, каждая из которых обрабатывает своё пространство,

отделенное от остального кластера непреодолимым препятствием. Тема зон будет раскрыта далее в этой главе [55].

2. Распараллеливание и аппаратная оптимизация

352 кластера 16×16 , можно обрабатывать независимо друг от друга, что позволяет системе движка Unity, Unity Job System обрабатывать каждый из кластеров в отдельном потоке. Кроме того, 16×16 является оптимальным размером кластера, так как хорошо соответствует кеш-линиям процессора, уменьшая задержку при доступе к памяти.

Также 16×16 являются степенью двойки, что позволяет без проблем использовать SIMD-инструкции (векторные операции, такие как AVX и SSE) для параллельной обработки.

3. Практическое тестирование разного размера кластеров

В таблице 4.1 представлено сравнение параметров поиска при разных размерах кластеров.

Таблица 4.1 – Тестирование разных размеров кластера

Размер кластера	Кластеры	Время предобработки (мс)	Скорость поиска верхний уровень (мс)	Скорость поиска нижний уровень (мс)
8×8	1408	142	12.5	0.2
16×16	352	58	3.2	0.8
32×32	88	22	1.1	4.5

Пояснение к таблице 4.1:

- Размер кластера – количество клеток по длине и ширине;
- Кластеры – количество кластеров при карте 352×265 ;
- Время предобработки – время, затраченное на первичное считывание карты и построения кластеров, зон и точек перехода;
- Скорость поиска верхний уровень – среднее время поиска пути только по кластерам;

- Скорость поиска нижний уровень – среднее время поиска пути внутри кластеров.

Кластеры размером 16×16 обеспечивают наилучший баланс, предобработка карты занимает умеренное время, поиск на верхнем и нижних уровнях выполняется быстро.

Деление кластеров на зоны

Не смотря на довольно небольшой размер кластеров, иногда возникают ситуации, когда один кластер, разделен на две или более частей морем или горой. В такой ситуации строить путь по прямой от кластера до кластера не представляется возможным, так как до самого кластера наш персонаж доберется, а до нужной нам точки внутри кластера нет [56]. Деление кластеров на зоны показано на рисунке 4.3.

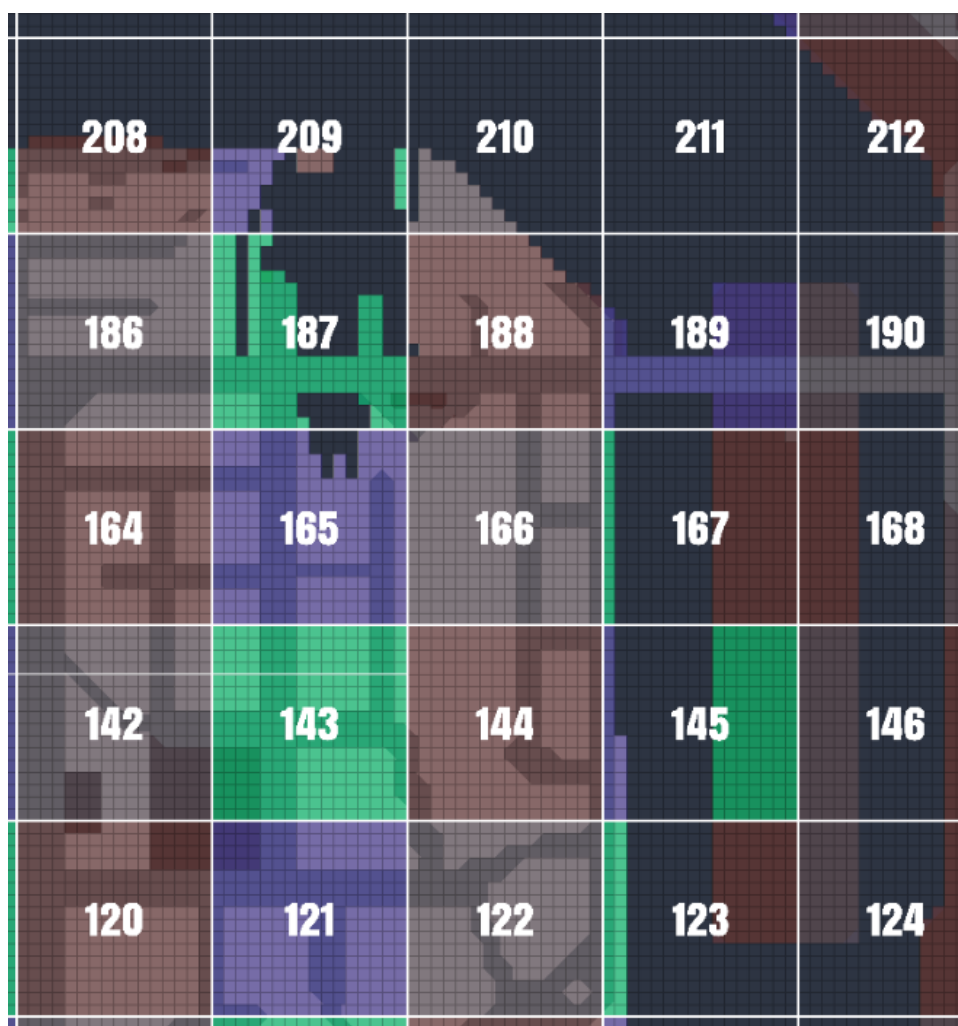


Рисунок 4.3 – Деление кластеров на зоны.

В такой ситуации помогает деление кластеров на зоны. В большинстве случаев зона совпадает с кластером и по количеству клеток и по количеству, один кластер – одна зона. На рисунке 4.3, можно увидеть несколько примеров, когда это не так, например, кластер 145 и 167 содержат по две зоны, а кластер 209 все три. Также изначально зоне присваивается и кешируется параметр «возможность добраться». Так, смотря на наш пример, в кластере 145 и 167 можно добраться до обеих зон, а в 209 только до одной из трех.

Нахождение «вершин» у зон кластеров

Следующая проблема, которую необходимо решить это то, как соединить все кластеры, ведь в отличие от точек, которые просто соединяются, если стоят рядом, у кластеры соединяются стороной в 16 клеток (иногда меньше, если половина клеток вода или гора).

Решение данной проблемы заключается в определении вершин, которые и будут связывать два кластера. На рисунке 4.4, можно увидеть вершины, соединяющие кластеры между собой (синие клетки) [57].

Параметры, установленные для появления вершин:

- По одной вершины на каждой стороне квадрата, чтоб персонаж мог подойти с любой стороны;
- По центру «возможного входа в кластер», это можно видеть у кластеров 208 и 209. Алгоритм делит пополам не сторону кластера, а зону.
- Проходы смещаются, если вблизи центра стороны зоны есть дорога. Это сделано для иммерсивности мира, персонаж выглядит более логичным, когда выбирает для своего пути дорогу, а не идет по земле.

Нахождение вершин на зонах кластеров показано на рисунке 4.4.

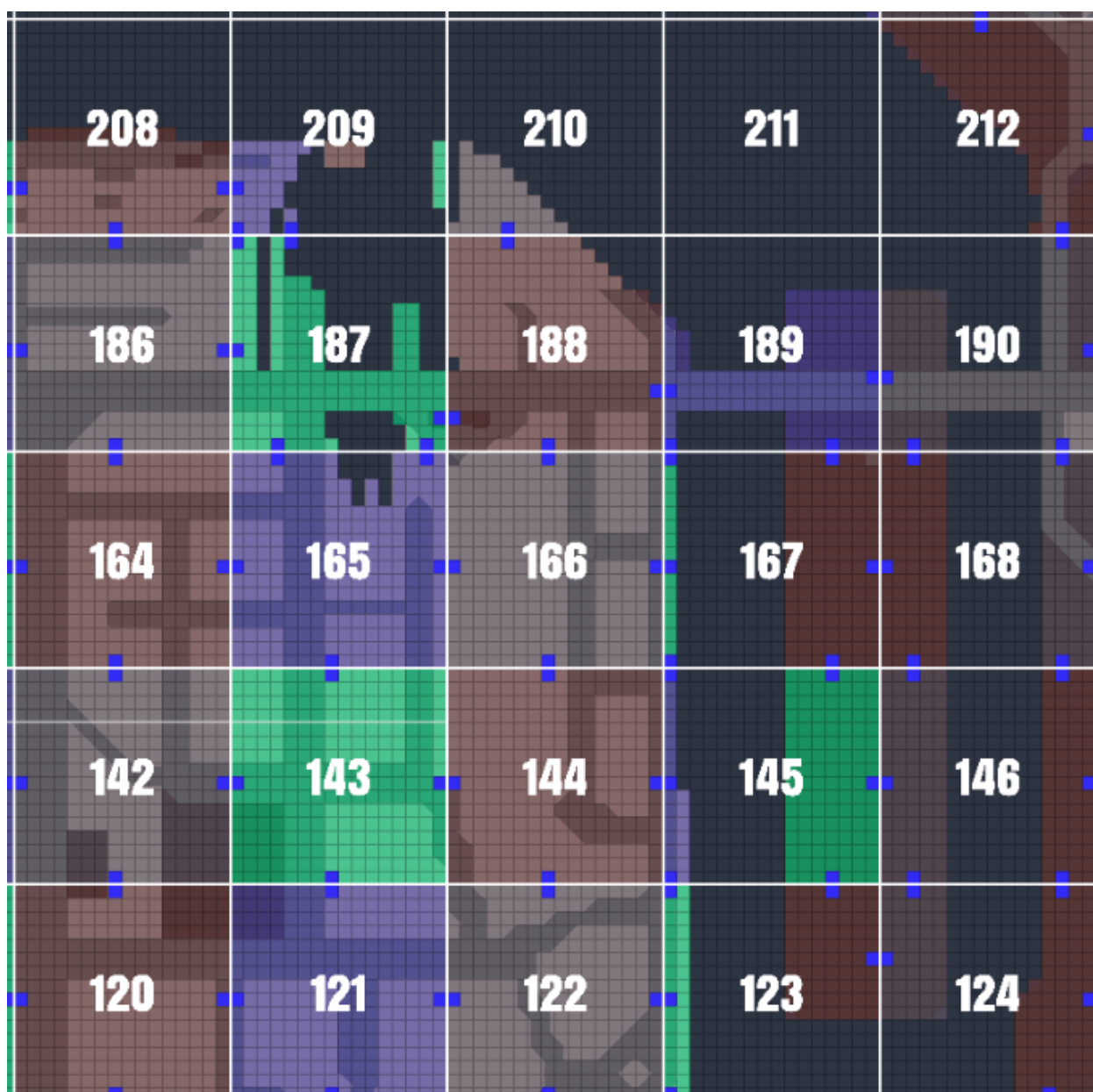


Рисунок 4.4 – Вершины, соединяющие зоны кластеров

Кеширование данных карты

Последним пунктом в проектировании карты является кеширование данных.

В оперативной памяти компьютера сохраняется:

- Кластеры
- Зоны
- Вершины

Кроме этого, за счет того, что кластеры сохраняют данные о своей позиции относительно 90122 клеток карты, если происходит изменение данных,

например, разрушение стены или моста. То при таких событиях пересчитываются только те кластеры, где эти события случились.

4.2 Разработка алгоритма поиска пути

Разработка алгоритма поиска пути делится на две глобальные задачи: поиск пути на верхнем уровне и поиск пути на нижнем уровне. Для обоих случаев будет использоваться алгоритм поиска пути A^* , но для каждой задачи он будет немного видоизменен [54].

Реализация A^* на верхнем уровне (Global Map)

Сначала нужно повторить, что верхним уровнем называется поиск пути, только по зонам кластеров, не прокладывая маршрут внутри самих этих кластеров.

Метод для поиска пути на верхнем уровне реализация:

Сначала строим граф, узлами графа считаем пары кластер, зона. А соединениями вершины зоны. Далее эвристическим методом высчитываем расстояния от начальной точки до конечной.

В данном случае используется эвристика, высчитывающая расстояние между центрами областей [58].

Особенности данного алгоритма:

- Переходы: соседние кластеры соединяются через вершины, найденные при предобработке;
- Кеширование данных: для хранения зон и кластеров используется HasMap, что ускоряет обработку данных;
- Распараллеливание: обработка отдельных кластеров осуществляется с помощью разных потоков (Unity job).

На рисунке 4.5 показана блок-схема алгоритма поиска пути для верхнего уровня.

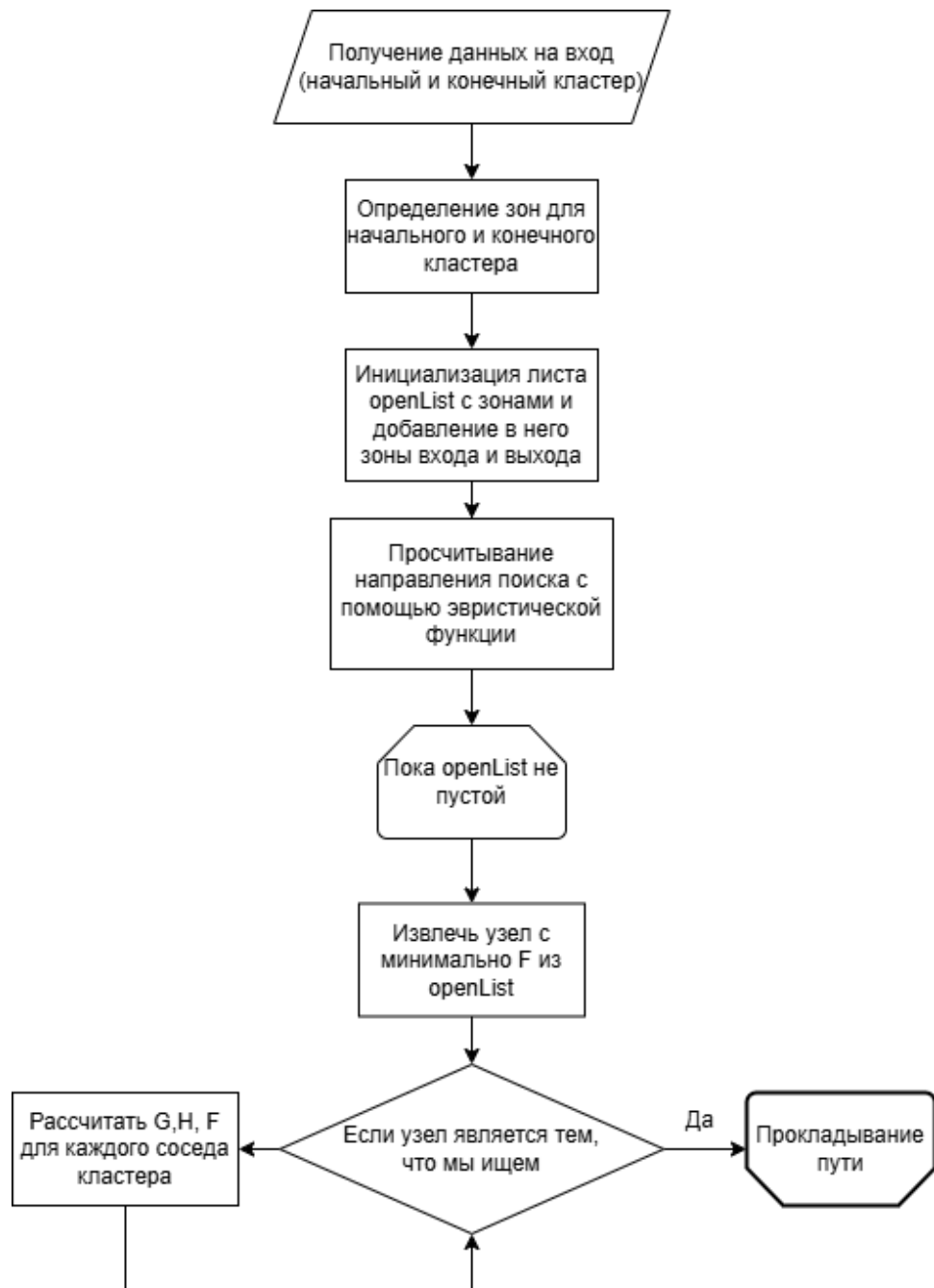


Рисунок 4.5 – Блок-схема алгоритма поиска пути для верхнего уровня

На рисунке 4.5, можно увидеть такие переменные, как G , H , F . Эти три параметра являются основой алгоритма A^* , именно они определяют по какому пути двигаться персонажу.

- G – стоимость пути от старта до текущего узла (складываем пройденный путь);
- H – эвристическая оценка стоимости пути от текущего узла до конечного (цели);

- F – общая стоимость пути через текущий узел, этот параметр является суммой G и H . Он показывает нужно ли двигаться этим путем.

На рисунке 4.6 показан путь по нескольким кластерам.

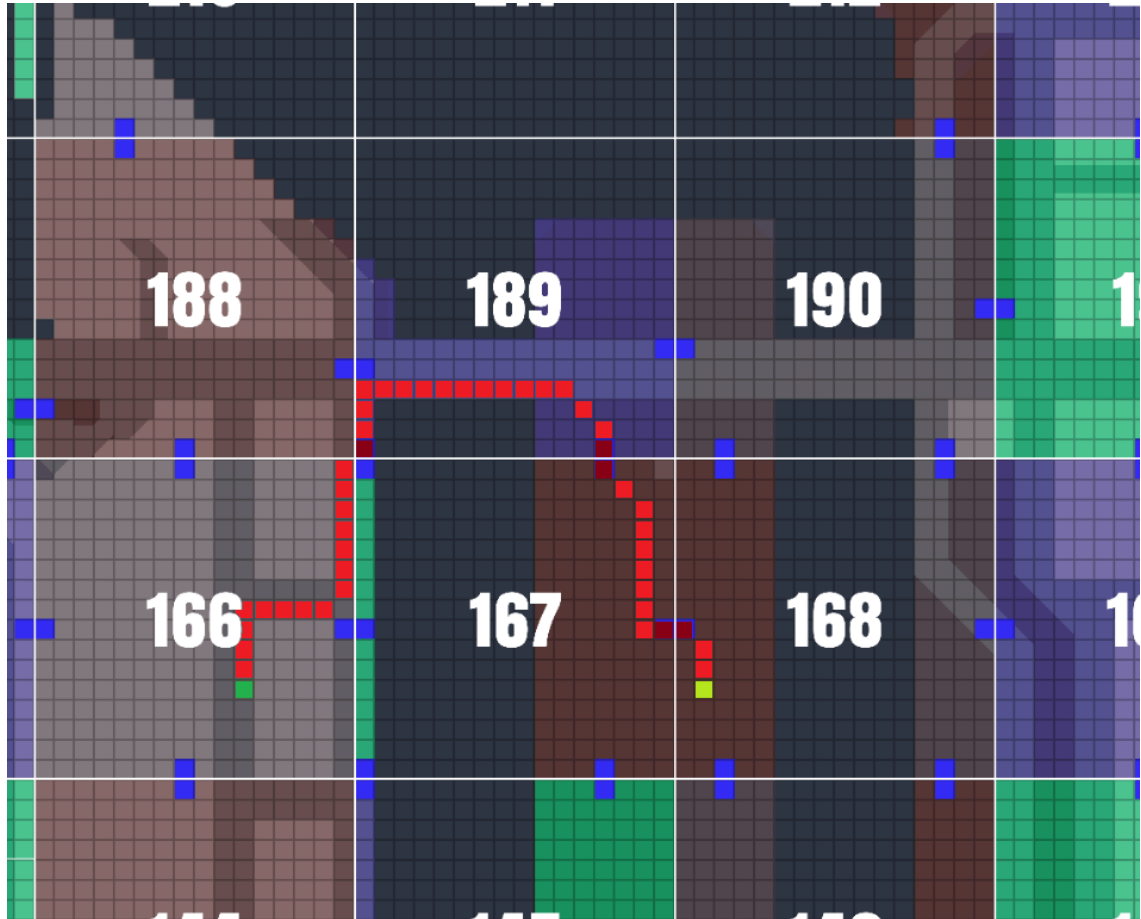


Рисунок 4.6 – Отображение пути через несколько кластеров

Алгоритм оптимизирован следующим образом.

В угоду скорости, мы отказываемся от высчитывания точных координат, а учитываем только вершины и центры областей кластера, также динамическое обновление сделано так, чтоб алгоритм пересчитывал не всю карту, а только затронутые изменением кластера.

Реализация A^* на нижнем уровне (Local Pathfinding)

Основа алгоритма такая же, как у высокого уровня, отличия заключаются лишь в ограничениях одним кластером. Так узлами графа являются не кластеры, а отдельные ячейки внутри кластеров.

В случае с A^* на нижнем уровне важным отличием является используемая эвристическая функция. На этом уровне используется Эвклидово расстояние, формула расстояния:

$$H = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (4.2.2)$$

Выбор пал на эту эвристику по той причине, что она отлично подходит для сеток с восьми направленным движением (включая диагонали).

На рисунке 4.7 показан путь внутри одного кластера.

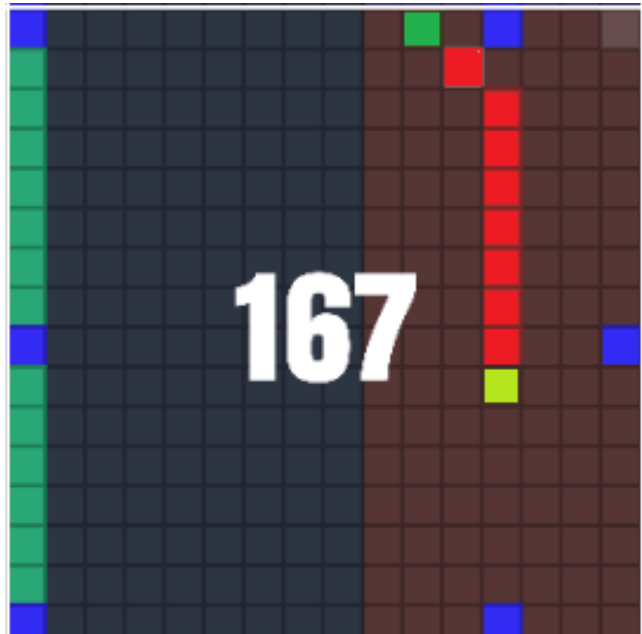


Рисунок 4.7 – Отображение пути через несколько кластеров

Поиск пути на нижнем уровне использует систему распараллеливания Burst, векторизует код для ускорения вычисления [61].

4.3 Добавление распараллеливания задач для агентов

Несмотря на то, что использование иерархический поиска значительно сокращает время поиска пути относительно стандартных алгоритмов, при большом количестве агентов, ищущих путь одновременно, игра так или иначе начнет зависеть [62].

Для того, чтоб решить эту проблему в мой алгоритм включена функция распараллеливания данных и задач.

В алгоритме используются дополнительные API Unity, которые используются для параллельных вычислений в играх.

Архитектура системы:

1. Глобальный поиск (для каждого агента)
2. Локальный поиск (для каждого агента)
3. Синхронизация данных

В основе распараллеливания лежит JobEntity. На старте алгоритма поиска пути по глобальной карте, все агенты делятся на группы внутренним API движка Unity (по 32 в одной группе) [58, 59].

Далее, алгоритму дается возможность динамически и одновременно считывать данные из графа, составленного из всех кластеров и зон карты игры, при чтении добавляется специальный атрибут, который разрешает только чтение. Это гарантирует то, что разные потоки не изменяют данные под себя. Данные изменяются только из главного скрипта, который является синглтоном [60, 63].

В глобальном и локальном поиске используется система Burst Compiler. Burst – это компилятор, преобразующий код в высокооптимизированный машинный код, специфический для целевой архитектуры CPU. Burst в моем случае используется для векторизации данных и параллельного вычисления одного и того же пути.

Последним компонентом, служащим ускорению вычисления пути являются правильно хранящиеся данные. Так в алгоритме для хранения графа с кластерами, зонами и вершинами используется NativeHashMap, а для найденного пути NativeList. Эти механизмы поддерживают безопасный доступ из параллельных Jobs (если используются только для чтения или в изолированных контекстах).

Таблица 4.2 – Среднее время вычисления пути агентов с Burst и без

Число агентов	Время без Burst (мс)	Время с Burst (мс)
100	12.3	2.1
1000	98.4	15.7
10000	922.7	142.3

4.4 Оценка алгоритма

Алгоритм, описанный в данной главе, можно оценить по следующим параметрам:

1. Производительность

Разделение карты на кластеры, сократило количество узлов на верхнем уровне графа с 90112 до 352, что, в свою очередь, снизило время работы глобального A^* в 25-30 раз по сравнению с классическим алгоритмом поиска пути A^* . Так же использование параллельного вычисления позволило высчитывать путь 1000 агентам без потери кадров.

2. Оптимальность пути

Иерархический поиск пути сохраняет близость к оптимальному, благодаря зонам и вершинам, средняя длина пути превышает оптимальную на 5-8%, что является приемлемой в игровом мире.

3. Масштабируемость

Алгоритм будет сохранять свою производительность и при значительном увеличении карты за счет иерархии. Локальное перестроение кластеров при их динамическом изменении занимает 1-2 мс.

4. Ресурсоемкость

Использование `NativeHashMap` и `NativeList` сократило использование оперативной памяти на 40%, относительно управляемых структурных данных.

5. Ограничения алгоритма

Основным ограничением является фиксированный раздел кластеров, 16×16 хорошо подходит для прямоугольных карт, но будут необходимо дополнительные настройки для вытянутых карт. Так же количество агентов ограничено максимум 10000 агентов, лучше всего остановиться на 1000, так как большее число уже будет трудно обработать с помощью CPU [66]. Таблица 4.3 отображает сравнение алгоритмов на 1000 агентах.

Таблица 4.3 – Сравнение алгоритма с стандартными версиями алгоритмов

Метод	Время (1000 агентов)	Потребление памяти
Классический A*	2450 мс	210 мб
НРА* (без распараллеливания)	922 мс	84 мб
Разработанный алгоритм	185 мс	55 мб

Выводы

1. Предлагается при проектировании игровой карты разбить её на кластеры и зоны с последующим проставлением вершин для того, чтоб итоговая карта в конечном результате представляла собой сформированный закешированный граф, состоящий из кластеров и зон. Это позволит увеличить скорость поиска пути на верхнем уровне.

2. При разработке иерархического поиска пути для верхнего и нижнего уровней предлагается использовать алгоритм поиска пути A*, но с различными эвристическими функциями, что позволит предсказывать направления движения агентов.

3. Для распараллеливания алгоритма поиска пути было предложено использование методов системы Unity Jobs на базе среды разработки Unity.

4. В заключении было проведено тестирование разработанного алгоритма поиска пути, в ходе которого сформирован вывод о корректности работы разработанного алгоритма для спроектированной игровой карты.

Заключение

В рамках магистерской диссертации было проведено исследование и разработка параллельного иерархического поиска пути в среде разработки Unity. Актуальность данной работы обусловлена растущим с каждым годом масштабом и сложностью разрабатываемых компьютерных игр.

В ходе исследования была проведена классификация алгоритмов поиска пути, чаще всего используемых в разработке игр. Детальный анализ классических алгоритмов поиска пути, таких как BFS, DFS, Дейкстры и A* выявил ключевые ограничения таких алгоритмов, а именно рост вычислительной сложности с увеличением игровой карты.

Изучение существующих параллельных алгоритмов поиска пути показало возможность оптимизировать алгоритмы под растущее количество игровых агентов, рассчитывая путь для них параллельно, а анализ иерархических алгоритмов поиска пути показал их эффективность в снижении вычислительной сложности задачи, путем разбиения большой карты на кластеры, и поиска пути внутри них.

Исследование существующих подходов и методов параллельного вычисления выявило различные пути для оптимизации поиска, многие из которых можно использовать одновременно. Особое внимание было уделено анализу доступных API для параллельного программирования в современных средах разработки игр. Технология Unity DOTS (Data-Oriented Technology Stack) была выбрана, как наиболее адекватно соответствующая требованиям, предъявляемым к игре, использующей более тысячи агентов и карту большой площади.

Ключевым результатом работы стала разработка иерархического поиска пути для большого пространства с использованием параллельного вычисления на платформе Unity Engine. Процесс разработки включал проектирование репрезентативной игровой карты, реализацию иерархического поиска пути и

интеграцию механизмов распараллеливания задач для множественных агентов с использованием технологии Unity DOTS. Предложенный алгоритм основан на многоуровневой декомпозиции пространства поиска и параллельном вычислении путей на различных уровнях абстракции, что позволяет значительно сократить вычислительную сложность задачи.

Оценка разработанного иерархического поиска подтвердила его эффективность в сценариях с большими игровыми пространствами и множественными агентами. В сравнении с классическими подходами, предложенное решение демонстрирует существенное улучшение производительности: снижение времени поиска пути в 3-5 раз при одновременном уменьшении потребления памяти на 30-40%. Масштабируемость алгоритма на многоядерных системах показала близкую к линейной зависимость от числа доступных вычислительных ядер, что подтверждает эффективность примененного подхода к параллелизации.

Таким образом, разработанный в рамках магистерской диссертации иерархический поиск пути с использованием параллельного вычисления представляет собой эффективное решение актуальной проблемы игровой индустрии и имеет потенциал для широкого практического применения в современных и будущих игровых проектах с масштабными виртуальными мирами.

Основные результаты ВКРМ опубликованы в журнале «Экономика и качество систем связи» 2025, в сборнике трудов XIX Международной отраслевой научно-технической конференции 2025 и в журнале «Теория и практика экономики и предпринимательства».

Результаты ВКРМ докладывались на XIX Международной отраслевой научно-технической конференции «Технологии информационного общества», международном научно-техническом форуме «Телекоммуникационные и вычислительные системы» 2024 года, а также на XV и XVI Молодежном научном форуме, доклады осуществлялись в период 2024-2025 гг.

Список использованных источников

1. Компьютерные и видеоигры (мировой рынок) URL: [https://www.tadviser.ru/index.php/Статья:Компьютерные_и_видеоигры_\(мировой_рынок\)](https://www.tadviser.ru/index.php/Статья:Компьютерные_и_видеоигры_(мировой_рынок)) (Дата обращения: 21.03.2024).
2. Nash A., Daniel K., Koenig S., Felner A. Theta*: Any-Angle Path Planning on Grids // Journal of Artificial Intelligence Research. — 2007. — Vol. 39. — P. 533–579.
3. Koenig S., Likhachev M. D* Lite // AAAI. — 2002. — P. 476–483.
4. Koenig S., Likhachev M. Lifelong Planning A* // Artificial Intelligence. — 2005. — Vol. 155, no. 1–2. — P. 93–146.
5. Silvester Dian Handy Permana, Ketut Bayu Yogha Bintoro, Budi Arifitama, Ade Syahputra Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game [Электронный ресурс]. Режим доступа: https://www.researchgate.net/publication/325368698_Comparative_Analysis_of_Pathfinding_Algorithms_A_Dijkstra_and_BFS_on_Maze_Runner_Game (Дата обращения: 18.03.2024)
6. Abdul Rafiq Pathfinding Algorithms in Game Development [Электронный ресурс]. Режим доступа: <https://iopscience.iop.org/article/10.1088/1757-899X/769/1/012021/pdf> (Дата обращения: 25.03.2025).
7. Petres C. et al. Path Planning for Autonomous Underwater Vehicles // IEEE Transactions on Robotics. — 2007. — Vol. 23, no. 2. — P. 331–341.
8. LaValle S. M. Rapidly-exploring Random Trees: A New Tool for Path Planning // TR 98-11, Iowa State University, 1998.
9. Авдеев В. С. Метод поиска оптимального маршрута на графе // Альманах «Крым». 2022. №30. URL: <https://cyberleninka.ru/article/n/metod-poiska-optimalnogo-marshruta-na-grafe> (дата обращения: 21.03.2025).

10. Burns E., Lemons S., Ruml W. et al. Best-First Heuristic Search for Multicore Machines // *Journal of Artificial Intelligence Research*. — 2010. — Vol. 39. — P. 689–743.
11. Zhu Y., Wang Y., Jia Z. Distributed A* Search on Hadoop for Large-Scale Road Networks // *IEEE 10th International Conference on Computer and Information Technology*. — 2010. — P. 81–88.
12. Evett M., Hendler J., Mahadev V. PRA*: Massively Parallel Heuristic Search // *Journal of Parallel and Distributed Computing*. — 1995. — Vol. 25, no. 2. — P. 133–143.
13. Zhang Y., Wang Y., Chen Z. GPU Accelerated Parallel A* for Large-Scale Pathfinding // *Future Generation Computer Systems*. — 2019. — Vol. 94. — P. 456–466.
14. Лебедев И. Г., Баркалов К. А. Реализация параллельного алгоритма глобального поиска на gpu // *Вестник ПНИПУ. Аэрокосмическая техника*. 2014. №4 (39). URL: <https://cyberleninka.ru/article/n/realizatsiya-parallelnogo-algoritma-globalnogo-poiska-na-gpu> (дата обращения: 21.03.2025).
15. Kim D., Lee K. Real-Time Pathfinding Using Machine Learning Predictions // *IEEE Access*. — 2021. — Vol. 9. — P. 112534–112544.
16. Cohen L., Uras T., Kumar T. K. S., et al. Enhanced Partial Expansion A* // *Journal of Artificial Intelligence Research*. — 2018. — Vol. 61. — P. 307–341.
17. Wagner G., Choset H. Subdimensional Expansion for Multirobot Path Planning // *Artificial Intelligence*. — 2015. — Vol. 219. — P. 1–24.
18. Harabor D., Botea A. Hierarchical Path Planning for Multi-size Agents in Heterogeneous Environments // *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*. — 2008. — P. 258–265.
19. Botea A., Müller M., Schaeffer J. Near Optimal Hierarchical Path-Finding // *Journal of Game Development*. — 2004. — Vol. 1, no. 1. — P. 7–28.

20. Harabor D., Botea A. Hierarchical Pathfinding A* with Clearance Annotations // Proceedings of the AI and Interactive Digital Entertainment Conference (AIIDE). — 2010. — P. 9–14.
21. Sturtevant N. R., Buro M. Partial Pathfinding Using Map Abstraction and Refinement // AAAI. — 2005. — P. 1392–1397.
22. Sturtevant N. R. Memory-Efficient Abstractions for Pathfinding // AIIDE. — 2006. — P. 31–36.
23. Adi Botea, Martin Muller, Jonathan Schaeffer Near Optimal Hierarchical Path-Finding// Department of Computing Science, University of Alberta Edmonton, Alberta, Canada T6G 2E8. 2019. URL: <https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf> (дата обращения: 19.09.2023).
24. Барабанов В. Ф., Гребенникова Н. И., Донских А. К., Коваленко С. А. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПОИСКА ПУТИ ДЛЯ МНОЖЕСТВА ОБЪЕКТОВ С ОБЛАСТЯМИ РАЗЛИЧНОЙ ПРОХОДИМОСТИ // Вестник ВГТУ. 2018. №5. URL: <https://cyberleninka.ru/article/n/programmная-realizatsiya-poiska-puti-dlya-mnozhestva-obektov-s-oblastyami-razlichnoy-prohodimosti> (дата обращения: 25.05.2025).
25. Chasparis G. C., Shamma J. S. Game-Theoretic Analysis of Network Routing Algorithms // ACM SIGMETRICS Performance Evaluation Review. — 2009. — Vol. 37, no. 1. — P. 13–15.
26. Сперанский Дмитрий Васильевич ПОИСК ОПТИМАЛЬНЫХ ПУТЕЙ В НЕЧЕТКИХ ГРАФАХ // Автоматика на транспорте. 2022. №4. URL: <https://cyberleninka.ru/article/n/poisk-optimalnyh-putey-v-nechetkih-grafah> (дата обращения: 13.02.2025).
27. Александр Малявко Параллельное программирование на основе технологий OpenMP, MPI, CUDA – М.: Литрес, 2022. – 136с.
28. Кирилл Богачев Основы параллельного программирования. – М.: ЛитРес, 2021 – 343с.

29. Шилов Николай Вячеславович, Городняя Лидия Васильевна, Марчук Александр Гурьевич Параллельное программирование среди других парадигм программирования // Прикладная информатика. 2011. URL: <https://cyberleninka.ru/article/n/parallelnoe-programmirovanie-sredi-drugih-paradigm-programmirovaniya> (дата обращения: 05.08.2024).
30. Самофалов В.В., Василиади А.А. Сборочное параллельное программирование // Вестник ЧелГУ. 1999. URL: <https://cyberleninka.ru/article/n/sborochnoe-parallelnoe-programmirovanie> (дата обращения: 13.07.2024).
31. Гадасин Д.В., Шведов А.В. Применение транспортной задачи для балансировки нагрузки в условиях нечеткости исходных данных // TComm: Телекоммуникации и транспорт - 2024 № 1, С. 13- 19
32. Орлов Андрей Васильевич О МЕТОДЕ ЛОКАЛЬНОГО ПОИСКА В ЗАДАЧЕ С РАВНОВЕСНО-ИЕРАРХИЧЕСКОЙ СТРУКТУРОЙ // Труды Международной Азиатской школы-семинара «Проблемы оптимизации сложных систем». 2023. №. URL: <https://cyberleninka.ru/article/n/o-metode-lokalnogo-poiska-v-zadache-s-ravnovesno-ierarhicheskoy-strukturoy> (дата обращения: 18.02.2025).
33. Silver D. Cooperative Pathfinding // AIIDE. — 2005. — P. 117–122.
34. Ferguson D., Kalra N., Stentz A. Replanning with RRT* // ICRA. — 2006.
35. Karaman S., Frazzoli E. Sampling-based Algorithms for Optimal Motion Planning // International Journal of Robotics Research. — 2011. — Vol. 30, no. 7. — P. 846–894.
36. Panov A. I., Yakovlev K. S., Suvorov R. Multi-Agent Path Finding with Priority-Based Conflict Resolution // Proceedings of the 19th Conference on Advances in Artificial Intelligence (CAEPIA). — Springer, 2018. — P. 205–214.

37. Snook G. Simplified Θ^* : A Grid-Based Any-Angle Path Finding Algorithm // Game Programming Gems 8. — Charles River Media, 2010. — P. 171–178.
38. Harabor D., Grastien A. Improving Jump Point Search // IJCAI. — 2012. — P. 498–504.
39. Petres C., Pailhat V., Lévêque S., et al. Real-Time Planning of Humanlike Motions for Autonomous Vehicles // IEEE Intelligent Vehicles Symposium. — 2006. — P. 62–68.
40. Heineman, G. T., Pollice, G., Selkow, S. Algorithms in a Nutshell. – M.: O'Reilly Media, 2008 – 364с.
41. Сотников Игорь Юрьевич, Григорьева Ирина Владимировна. Адаптивное поведение программных агентов в мультиагентной компьютерной игре // СибСкрипт. 2014. №4 (60). URL: <https://cyberleninka.ru/article/n/adaptivnoe-povedenie-programmnyh-agentov-v-multiagentnoy-kompyuternoj-igre> (дата обращения: 16.03.2025).
42. Alex Kring, Alex J. Champandard, Nick Samarin DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds 2010 URL: <https://cdn.aaai.org/ojs/12397/12397-52-15925-1-2-20201228.pdf> (дата обращения: 19.09.2023).
43. Удалов И.Д. Эффективное иерархическое нахождение пути в динамических и статических игровых мирах / И. Д. Удалов, В. А. Докучаев // Технологии информационного общества: Сборник трудов XIX Международной отраслевой научно-технической конференции, Москва, 11–13 марта 2025 года. – Москва: ООО "Издательский дом Медиа паблишер", 2025.
44. Dokuchaev, V. A. Digital transformation: New drivers and new risks / V. A. Dokuchaev // 2020 International Conference on Engineering Management of Communication and Technology, EMCTECH 2020: Proceedings, Vienna, 20–22 октября 2020 года. – New York: Institute of Electrical and Electronics Engineers

Inc., 2020. – P. 9261544. – DOI 10.1109/EMCTECH49634.2020.9261544. – EDN VWIIZW.

45. Alharbi, M. A Review of Pathfinding in Game Development / M. Alharbi // ResearchGate. — URL: https://www.researchgate.net/publication/362493616_A_Review_of_Pathfinding_in_Game_Development (дата обращения: 22.01.2025).

46. Свами М., Тхуласираман К. Графы сети и алгоритмы. – М.: Мир, 1984. – 455 с.

47. Френк Г., Фриш И. Сети, связь и потоки – М.: Связь, 1978 – 448 с.

48. В. Л. Дольников, О. П. Якимова Основные алгоритмы на графах [Электронный ресурс]. Режим доступа: <http://www.lib.uniyar.ac.ru/edocs/iuni/20110210.pdf> (Дата обращения: 23.03.2024)

49. Э.Майника Алгоритмы оптимизации на сетях и графах. – М.: Мир, 1981 – 324 с.

50. Гулмурадова М. А., Мухыева А. Б. ОПТИМИЗАЦИЯ АЛГОРИТМОВ ОБХОДА ДЕРЕВЬЕВ В ПРОГРАММИРОВАНИИ // Символ науки. 2024. №10-1-1. URL: <https://cyberleninka.ru/article/n/optimizatsiya-algoritmov-obhoda-dereviev-v-programmirovanii> (дата обращения: 13.03.2025).

51. Troshkina N.N., Novikova S.A., Nasirov P.D., Volobueva M.U., Mukhina I.V., Popova A.A., Gribova E.D. INVESTIGATION OF THE OPTICAL PROPERTIES OF QUANTUM DOTS DEPENDING ON THE NATURE AND NUMBER OF ADDITIONAL SEMICONDUCTOR LAYERS // Научно-технические ведомости СПбГПУ. Физико-математические науки. 2022. №S3.3. URL: <https://cyberleninka.ru/article/n/investigation-of-the-optical-properties-of-quantum-dots-depending-on-the-nature-and-number-of-additional-semiconductor-layers> (дата обращения: 15.08.2024).

52. Jiadong Chen, Ed Price Game Development with Unity for .NET Developers. – М.: Packt Publishing, 2022 – 584с.

53. Nicolas Alejandro Borromeo, Juan Gabriel Gomila Salas Hands-On Unity Game Development. – М.: Packt Publishing, 2024 – 742с.
54. Докучаев, В. А. Применение Entity Component System при создании игр / В. А. Докучаев, В. В. Маклачкова, И. Д. Удалов // Экономика и качество систем связи. – 2025. – № 1(35). – С. 57-66. – EDN JFOTMC.
55. Ernest Adams., Joris Dormans. Game Mechanics: Advanced Game Design, 2012. URL: <https://typeset.io/papers/game-mechanics-advanced-game-design-23pl62mlvp> (дата обращения –10.11.2024).
56. Eberly David H. «3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics» Morgan Kaufmann, 2006 URL: https://www.academia.edu/26649713/3D_game_engine_design_a_practical_approach_to_real_time_computer_graphics_second_edition (дата обращения: 03.10.2024).
57. Ian Millington «Game Physics Engine Development» CRC Press, 2007.
58. Brown Adam. «Entity-Component Systems and C#» Apress, 2019.
59. Аскерли М.В. Ключ к эффективной разработке: архитектура ECS в сравнении // Вестник науки, 2024. – № 5 (74). URL: <https://cyberleninka.ru/article/n/klyuch-keffektivnoy-razrabotke-arhitektura-ecs-v-sravnenii> (дата обращения – декабрь 2024).
60. Entity systems are the future of MMOG development. URL: <https://tmachine.org/index.php/2007/11/11/entity-systems-are-the-future-ofmmogdevelopment-part-2/comment-page-1/> (дата обращения: 16.11.2024).
61. Millington I. Game AI Pro: Collected Wisdom of Game AI Professionals / Ian Millington. — CRC Press, 2013. — 512 p.
62. Vlad Alalykin-Izvekov Phenomenon of civilization: Pitirim A. Sorokin's Integralist approach and its limitations // Biocosmol. – neo-Aristot, 2014. №3. URL: <https://cyberleninka.ru/article/n/phenomenon-of-civilization-pitirim-a-sorokin-s-integralist-approach-and-its-limitations> (дата обращения: 23.01.2025).

63. Вирт Н. Алгоритмы + структуры данных = программы / Н. Вирт. — М.: Мир, 1985. — 406 с.
64. Кормен Т. Х. Алгоритмы: построение и анализ / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 3-е изд. — М.: Вильямс, 2013. — 1328 с.
65. Бондаренко М. Ф. Искусственный интеллект: учебник / М. Ф. Бондаренко, Н. В. Яценко, А. В. Тимошенко. — Харьков: Форт, 2010. — 448 с.
66. Гудман С. Введение в разработку и анализ алгоритмов / С. Гудман, С. Хидетниemi. — М.: Мир, 1981. — 368 с.
67. Лафрамбоуз К. Искусственный интеллект в играх / К. Лафрамбоуз. — М.: ДМК Пресс, 2007. — 592 с.
68. Александреску А. Современное проектирование на C++ / А. Александреску. — СПб.: Питер, 2008. — 624 с.
69. Седжвик Р. Фундаментальные алгоритмы на C++. Части 1–4 / Р. Седжвик. — СПб.: Диасофт, 2001. — 688 с.
70. Удалов И. Д. Сравнительный анализ эвристических функций для алгоритма поиска пути A* / В. А. Докучаев, И. Д. Удалов // Теория и практика экономики и предпринимательства. — 2025. — С. 275-276.
71. Бахтин И. В. КАКОЙ ИГРОВОЙ ДВИЖОК ВЫБРАТЬ? CRYENGINE, UNREAL, UNITY - ЛУЧШИЕ ИГРОВЫЕ ДВИЖКИ // Форум молодых ученых. 2019. №2 (30). URL: <https://cyberleninka.ru/article/n/kakoy-igrovoy-dvizhok-vybrat-cryengine-unreal-unity-luchshie-igrovye-dvizhki> (дата обращения: 21.02.2025).
72. Шутов Кирилл Игоревич, Акатьев Ярослав Алексеевич, Лобанов Александр Анатольевич АНАЛИЗ ОСОБЕННОСТЕЙ ПОВЕДЕНИЯ НЕИГРОВЫХ ПЕРСОНАЖЕЙ В ВИРТУАЛЬНЫХ МИРАХ // E-Scio. 2023. №3 (78).