РЕФЕРАТ

Пояснительная записка содержит 15 страниц, 4 части, 1 приложение, 0 рисунков, 0 таблиц и N источника.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задачей нахождения кратчайшего пути является поиск оптимального и короткого пути между двумя точками. Проблема нахождения кратчайших путей возникает в таких случаях как: оптимизация перевозки грузов и пассажиров, оптимальная маршрутиризация пакетов в сети, навигация искусственного интеллекта и игрока в компьютерных играх, а так же навигация роботов в пространстве. Все компьютерные игры имеют поиск путей в том или ином виде, поэтому скорость и точность алгоритма часто влияет на качество искусственного интеллекта и восприятия игрока.

Существует несколько представлений пространства для проведения поиска путей, одним из них является квадратная сетка, которая проста для создания и используется в стратегиях реального времени и других играх с двумерной картой. Хотя квадратная сетка в большинстве случаев является неоптимальным представлением области поиск, с ней очень просто работать и легко модифицировать, что значительно упрощает программную работу с игровой картой. В следствии неоптимальности представления карты появляется необходимость в оптимизации алгоритмов поиска работающих с ней посредством общей оптимизации логики работы алгоритма, введение некоторых допущений и ограничений на область поиска, а так же проведение низкоуровневых оптимизаций.

Таким образом, задача нахождения кратчайшего пути на области представленной квадратной сеткой является актуальной проблемой, которая будет рассмотрена и исследована в данной работе.

Целью выпускной работы является проведение анализа существующих алгоритмов поиска путей, разработка различных вариантов алгоритмов и их оптимизаций, создание оптимизированной универсальной библиотеки для нахождения оптимальных маршрутов и анализ полученных результатов.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Алгоритмы нахождения кратчайших путей

Задачей нахождения кратчайшего пути является поиск оптимального и короткого пути между двумя точками. Решения этой задачи в большинстве случаев основаны на алгоритме Дейкстры для нахождения кратчайшего пути в взвешенных графах.

Простейшие алгоритмы для обхода графа, такие как поиск в ширину и поиск в глубину могут найти какой-то путь от начальной до конечной вершины, но не учитывают стоимость пути. Одним из первых алгоритмов поиска пути с учётом его стоимости был алгоритм Беллмана – Форда, который проходит по всем возможным маршрутам и находит наиболее оптимальный, вследствие чего имеет временную сложность $O(|V||E|)$, где V - количество вершин, а E - количество рёбер. Однако для нахождения пути близкого к оптимальному не обязательно перебирать все пути, а можно отсекать неперспективные направления на основе некой эвристики, что может дать таким алгоритмам нижнюю оценку $O(|E|\log(|V|))$. Такими алгоритмами являются алгоритм Дейкстры, A* и их модификации.

1.1.1 A*

Алгоритм A* (A звёздочка) - это алгоритм общего назначения, который может быть использован для решения многих задач, например для нахождения путей. A* является вариацией алгоритма Дейкстры и используя эвристическую функция для ускорения работы, при этом гарантируя наиболее эффектное использование онной.

Алгоритм A* поочерёдно рассматривает наиболее перспективные неисследованные точки или точки с неоптимальным маршрутом до них, выбирая пути которые минимизируют $f(n) = g(n) + h(n)$, где n последняя точка в пути, $g(n)$ - стоимость пути от начальной точки до точки n, а $h(n)$ - эвристическая оценка стоимости пути от n до конца пути. Когда точка исследована, алгоритм

останавливается если это конечная точка, иначе все её соседи добавляются в список для дальнейшего исследования.

Для нахождения пути от начальной до конечной точки, кроме стоимости пути до точки, следует записывать и её предка (точку из которой мы пришли в неё).

Свойства алгоритма A*:

- Алгоритм гарантирует нахождение пути между точками, если он существует;

- Если эвристическая функция $h(n)$ не переоценивает действительную минимальную стоимость пути, то алгоритм работает наиболее оптимально;

- A* оптимально эффективен для заданной эвристики $h(n)$.

На оценку сложности A* влияет использованная эвристика, в худшем случае количество точек рассмотренных A* экспоненциально растёт по сравнению с длинной пути, однако при эвристике $h(n)$ удовлетворяющей условию $|h(x) - h^*(x)| \leq O(\log h^*(x))$, где $h^*(n)$ - оптимальная эвристика, алгоритм будет иметь полиномиальную сложность. Так же на временную сложность влияет выбранный способ хранения закрытых и открытых точек.

Оценка памяти

1.1.2 JPS

1.1.3 HPA* и HAA*

HPA* (Hierarchical Path-Finding A*) - добавляет алгоритму A* иерархическую абстракцию, разбивая карту на прилегающие друг к другу кластеры, которые соединены входами. Одной из основных идей алгоритма является то, что расчёт пути в A* каждый раз происходит с нуля, что можно исправить добавив сохранение кратчайших путей между определёнными точками.

Первым этапом алгоритма является препроцессинг карты для построения кластеров и их входов. При этом возможно построение нескольких уровней графа кластеров используя один и тот же алгоритм рекурсивно на созданных во время предыдущего прохода кластерах.

Во время исполнения программы запрос нахождения пути выполняется рекурсивно находя и уточняя путь на графе начиная с самых крупного уровня кластеров. После нахождения пути может применятся его сглаживание.

Алгоритм HPA* работает с такими допущениями:

- Все актёры имеют одинаковый размер, при этом все части навигационной сетки проходимы ими;

- На всех участках карты агенты имеют одинаковую проходимость.

Иерархическая структура карты сильно ускоряет поиск пути, однако алгоритм HPA* работает не учитывая такие важные параметры как размер агентов и проходимость местности.

В итоге размер агентов и проходимость карты должны учитываться при нахождении пути, что в случае с алгоритмом HPA* приводит к тому, что эти параметры должны учитываться при оценки путей между входами.

Алгоритм иерархического поиска является достаточно абстрактным для учёта указанных проблем. Для этого на основе алгоритма HPA* был создан алгоритм HAA* (Hierarchical Annotated A*), который при создании путей между входами кластера учитывает размеры атёров и проходимость местности.

Основная разница с алгоритмом HPA* у алгоритма HAA* состоит в шаге формирования пути между транзитными точками в рамках кластера и дальнейшем шаге их оптимизации. При нахождении пути между транзитными точками в кластере следует найти пути для всех агентов разных размеров и проходимости

В итоге алгоритм HAA* имеет такие же преимущества как HPA*, а так же возможность учёта размера агента и проходимости карты. В зависимости от выбранной тактики устранения похожих путей между транзитными точками кластеров конечный путь будет хуже оптимального до 4-8%.

По сравнению с JPS алгоритмы HPA* и HAA*, работают дольше и являются намного сложнее в реализации, которая может быть несоразмерна с полученной от них выгодой, однако они могут выдавать начальные участки пути намного быстрее чем JPS и A*, что в некоторых случаях оправдывает их написание.

Для реализации в выпускной работе были выбраны алгоритмы A* и JPS.

1.2 Различные представления области поиска

1.3 Эвристические функции

1.4 Постановка задачи

Целью выпускной работы является проведение анализа существующих алгоритмов поиска путей, разработка различных вариантов алгоритмов и их оптимизаций, создание оптимизированной универсальной библиотеки для нахождения оптимальных маршрутов и анализ полученных результатов. Создание библиотеки состоит из заданий:

- анализ алгоритмов нахождения путей;
- разработка алгоритма A*;
- разработка алгоритма JPS;
- разработка препроцессинга GoalBounding;
- интеграция GoalBounding в алгоритм A*;
- интеграция GoalBounding в алгоритм JPS;
- разработка визуализатора для наглядной оценки и проверки алгоритмов;
- создание модуля для сравнения и валидации алгоритмов;

Для написания библиотеки был выбран язык C++. Для визуализации результатов была выбрана библиотека SFML и SFGUI. Для использования параллельности - библиотека threadpool11.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Программное обеспечение

Для написание библиотеки нахождения кратчайших путей был выбран язык программирования C++ стандарта C++11. C++ является современным языком высокого уровня, который предоставляет широкие возможности по оптимизации кода, в отличии от интерпретируемых языков и языков с JIT оптимизациями. Так же C++ широко используется в игровых приложениях. Для создания библиотеки язык C++ был выбран по следующим причинам: кроссплатформенность, быстродействие, низкоуровневые оптимизаций, возможность внедрения библиотеки в существующие программные продукты.

Стандарт C++11 привнёс в язык многие функции, которые позволяют писать более понятный и современный код, уменьшить количество случайных ошибок и повысить читаемость. В следствии чего были устранены многие недостатки в сравнении с другими языками программирования.

В C++11 для измерения времени используется стандартная библиотека chrono. В chrono существует несколько реализаций измерения времени:

- system_clock - общесистемное время;

- steady_clock - монотонное время, которое никогда не подстроено;

- high_resolution_clock - наиболее точное время с наименьшим доступным периодом.

Библиотека SFML (Simple and Fast Multimedia Library) - кроссплатформенная библиотека для создания мультимедийных приложений. Имеет простой платформонезависимый интерфейс для рисования графики.

SFGUI - библиотека работающая совместно с библиотекой SFML и предоставляющая возможности для отрисовывания интерфейсов.

Для использования многопоточности была подключена библиотека threadpool11, которая реализует пул потоков. Пул потоков –

Для написания кода библиотеки была выбрана среда разработки CLion, которая имеет редактор с поддержкой синтаксиса C++11 и его подсветкой,

имеет средства рефактоинга и поддерживает различные CVS, например Git. Так же имеется поддержка CMake - системы кроссплатформенной сборки проектов и управления зависимостями и тестами.

В качестве системы контроля версий был выбран Git. Git - распределённая система контроля версий, которая направлена на скорость работа и целостность данных, имеет гибкую и простую систему создания и объединения веток.

2.2 Архитектура ПО

2.3 UML-моделирование ПО

Унифицированный язык моделирования (UML) - язык общего назначения для визуализации, спецификации, конструирования и документации программных систем [1]. Язык UML объединяет в себе семейство разных графических нотаций с общей метамоделью.

Преимуществами UML являются:

- объектно-ориентированность, что делает его близким к современным объектного-ориентированным языкам;

- расширяем, что позволяет вводить собственные текстовые и графические стереотипы;

- прост для чтения;

- позволяет описать системы со всех точек зрения.

В UML используется три вида диаграмм:

- структурные диаграммы - отражают статическую структуру системы;

- диаграммы поведения - отражают поведение системы в динамике, показывают, что должно происходить в системе;

- диаграммы взаимодействия - подвид диаграмм поведения, которые выражают передачу контроля и данных внутри системы.

Для моделирования системы проектируемой в данной выпускной работе будут использованы структурные диаграммы, а именно диаграмма вариантов использования (Use case diagram) и диаграмма классов (Class diagram).

Диаграмма вариантов использования -

Диаграмма классов -

2.4 Описание интерфейса взаимодействия

3 ОПИСАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ

3.1 Реализации алгоритма A*

3.2 Реализация алгоритма JPS

3.3 Реализация и интеграция GoalBounds

3.4 Реализация визуализатора

4 АНАЛИЗ РЕЗУЛЬТАТОВ

5 ТЕСТИРОВАНИЕ

ВЫВОДЫ

СПИСОК ЛИТЕРАТУРЫ

1. Unified Modeling Language User Guide, The, 2nd Edition / G. Booch, J. Rumbaugh, I. Jacobson - Addison-Wesley Professional, 2005. - 496 с. - ISBN 0-321-26797-4.

Приложение А