Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

(Университет ИТМО)

Факультет цифровых трансформаций

Образовательная программа Технологии разработки компьютерных игр
Направление подготовки (специальность) Прикладная информатика

О Т Ч Е Т

о преддипломной практике

Тема задания: *Разработка модифицированного алгоритма целеориентированного планирования*
Обучающийся *Ястребов Никита Кириллович, J4121*

Руководитель практики от университета: Бравичев Кирилл Александрович, преподаватель школы разработки видеоигр

Практика пройдена с
оценкой _____

Дата 25.05.2024

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Классический алгоритм – алгоритм целеориентированного планирования, для которого разрабатывалась модификация.

Модифицированный алгоритм – алгоритм целеориентированного планирования, включающий в себя разработанную модификацию.

NPC – (англ. Non-Player Character – не управляемый игроком персонаж) – виртуальный персонаж, который управляется системой искусственного интеллекта. NPC может выполнять различные функции в игре, такие как диалог с игроком, предоставление квестов, торговля, бои и другие действия.

Агент – сущность, контролируемая системой игрового искусственного интеллекта.

СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ

$\boldsymbol{A}$ – матрица

$\boldsymbol{w}$ – вектор

$\boldsymbol{A}_{n\times m}$ – матрица из n строк и m столбцов

$\boldsymbol{w}_{n\times 1}, \boldsymbol{w}_{1\times n}$ – вектор строка/ столбец из n элементов

$\boldsymbol{A}^T$ – транспонированная матрица

$\boldsymbol{A}^{-1}$ – обратная матрица

$\mathrm{tr}(\boldsymbol{A})$ – след матрицы

$\|\boldsymbol{A}\|$ – норма матрицы

$\frac{\partial}{\partial \boldsymbol{A}}$ – производная по матрице

$\overline{(a, b)}$ – последовательность натуральных чисел из отрезка $[a; b]$

ВВЕДЕНИЕ

Видеоигры за свою более чем полувековую историю превратились из простых развлечений для программистов в масштабную индустрию и важную часть культуры. Как и кино, они прошли путь от увеселительной диковинки до мирового признания. Сегодня видеоигры привлекают самую большую аудиторию среди всех развлечений, как показывают исследования.

Для видеоигр характерна интерактивность, то есть возможность игрока и игры обмениваются информацией. Для обеспечения интерактивности служат множество игровых систем. Одна из таких систем - искусственный интеллект, отслеживающий и реагирующий на действия игрока. Он особенно важен в играх с множеством механик и большим миром, которые наиболее популярны на рынке.

В числе сфер применения искусственного интеллекта в играх – контроль над NPC. Они могут быть простыми животными или полноценными спутниками, взаимодействующими с игроком в зависимости от жанра и масштаба игры.

Однако индустрия видеоигр, как отмечается, отстает в области искусственного интеллекта, но имеет потенциал для их адаптации. Недостаток исследований в данной сфере подчеркивает важность новых разработок в этой области. Таким образом, настоящая работа призвана обеспечить актуальность и востребованность исследований в этой сфере.

Объектом данного исследования является алгоритм целеориентированного планирования (GOAP). GOAP широко применяется в создании систем принятия решений NPC в играх. Он позволяет NPC адаптироваться к текущей цели и среде. В начале своего использования GOAP вызвал большой интерес у разработчиков за его способность создавать умных NPC. Примером успешного применения GOAP стала игра F.E.A.R. [1] от Monolith. Позже GOAP стал популярным инструментом [2], [3], [4], позволяющим NPC адаптироваться к различным ситуациям и выполнять сложные задачи. Однако, несмотря на его популярность, GOAP в настоящее

время не используется повсеместно, уступая таким подходам, как деревья поведения, что вызывает необходимость модернизации для его актуализации и становления стандартом качественного искусственного интеллекта.

Анализ алгоритма позволяет обнаружить несколько его недостатков, в числе которых – высокая чувствительность к эвристическим мерам, а также не учитывание предшествующих итераций планирования при построении очередного плана.

Целью настоящей работы является повышение реалистичности поведения агентов, управляемых GOAP-алгоритмом, за счет снижения чувствительности планов к параметрам эвристик и учета накопленной в результате предшествующего планирования информации.

В процессе достижения поставленной цели возникли и были решены следующие задачи:

1) анализ реализаций алгоритма целеориентированного планирования в публикациях и профильной литературе;
2) проектирование и реализация собственной системы принятия решений на базе GOAP;
3) проектирование и разработка модификации системы для достижения поставленной цели;
4) проектирование системы сбора экспериментальных данных, а также получение уравнений линейно регрессии;
5) интеграция спроектированной системы в существующий игровой движок;
6) тестирование, испытание системы;
7) анализ результатов испытаний.

Для реализации системы принятия решений в качестве языка программирования был выбран C++ в силу его быстроты, удобства, а также простоты использования в связке с движком Unreal Engine 5, который

использовался для создания испытательной среды, тестирования и визуализации.

Результатом работы стала программа-планировщик, интегрированная в приложение-прототип, собранное в Unreal Engine 5. Эта программа представлена в виде заголовков, файлов исходного кода и файлов UnrealEngine Blueprints (.uasset), что позволяет легко интегрировать ее в любой проект Unreal Engine 5.

Для проверки возможности модификации алгоритма так, чтобы он учитывал накопленный опыт агента при принятии решений, было проведено сравнение поведения, получаемого с использованием классического и модифицированного алгоритма.

Для оценки снижения чувствительности алгоритма к колебаниям эвристик проводились как экспериментальные измерения, так и теоретический анализ численных результатов. Это помогло подтвердить, что модифицированный алгоритм действительно обеспечивает более стабильное поведение и устойчивость к изменениям входных данных.

1 Обзор предметной области

1.1 Обзор поведенческих алгоритмов

В основе контроллеров NPC в играх лежит базовый цикл, изображенный на рисунке 1 [5].

Рисунок 1 – Высокоуровневый цикл искусственного интеллекта NPC

Коротко рассмотрим все его этапы:

На этапе «Обновление мотивации» анализируется внутреннее состояние агента и состояние окружающего мира. На основе этого определяются нужды агента, например, восполнить запас патронов, спрятаться атаковать врага – то есть, фиксируется текущая цель.

На этапе «Планирование действия» на основе текущей цели из списка доступных действий выбирается действие, которое приблизит агента к цели. Например, при необходимости пополнить запас патронов, агент может из двух доступных ящиков с патронами выбрать тот, что расположен ближе к нему.

Наконец, на этапе «Совершение действия» агент выполняет команды, закрепленные за выбранным действием.

За планирование текущего действия отвечает поведенческий алгоритм. Среди поведенческих алгоритмов наиболее известными являются rule-based system, конечный автомат [6], [7], [8] behavior tree [9], utility-based system [10], [11] hierarchical task network planning (HTNP) [12], goal-oriented behavior (GOB) [13], [14] и GOAP (целеориентированное планирование) [15]. Также перспективным является обучение с подкреплением (reinforcement learning) [16], [17], [18], [19]. Каждый из них подходит для различных целей. Алгоритм, отлично справляющийся с управлением напарником в RPG в открытом мире, может быть неудачным для определения поведения противника в шутере от первого лица. В этом разделе приведен краткий анализ этих алгоритмов, а также сравнение с алгоритмом GOAP.

Конечный автомат (FSM) — это алгоритм, который описывает процесс с конечным числом заранее заданных дискретных состояний и переходов между ними. Управляющий поток FSM останавливается в одном из состояний, и переходы из этого состояния определяют возможные следующие состояния. Каждый переход связан с определенным событием (например, событием в игровом мире). Когда событие происходит, активируется соответствующий переход из текущего состояния в новое. Таким образом, конечный автомат переводит агента из одного состояния к другому пошагово.

К достоинствам конечного автомата можно отнести следующие пункты:

- простая структура;
- возможность нескольких реализаций;
- возможность использовать FSM в виде подмодуля более обширно системы ИИ;
- элементарная визуализация и отладка.

Rule-based system, имеет структуру, состоящую из двух частей: базы данных, содержащей знания, доступные искусственному интеллекту, и набора

правил вида “if-then”. Правила могут сверяться с базой данных, чтобы определить, выполнено ли их условие “if”. Правила, у которых выполнены условия, активируются. В результате активации правила, срабатывает логика “then”. Отличием между состоянием в FSM и правила в rule-based system является возможность наличия нескольких одновременно активных правил (конечный автомат всегда пребывает в едином состоянии), что позволяет модульно конструировать поведение агента.

К достоинствам этого алгоритма также можно отнести то, что он легко позволяет устанавливать причинно-следственные связи при анализе и деконструкции поведения агента, что облегчает отладку.

Алгоритм behavior tree основывается на понятии *задачи*. Задача представляет собой некую функцию, возвращающую значение «истина» или «ложь» – статус. Задачи объединяются в деревья иерархически, то есть, более сложные задачи строятся из более простых. Это позволяет легко комбинировать поддеревья задач для создания сложного поведения.

Деревья поведения имеют три основных типа задач: условия, действия и композиты. Как и в других алгоритмах условия представляют собой условные выражения, содержащие переменные из данных игрового мира. Действия изменяют значения этих переменных и возвращают статус в зависимости от успеха или неудачи операции. Композиты объединяют другие задачи в структуры для создания более сложного поведения. Статус композита выбирается на основе значений включенных в него задач и зависит от типа композита.

Дерево поведения хранит все возможные поведения агента. Маршрут из корневого узла в лист дерева представляет собой один возможный набор действий. В процессе построения поведения на дереве выполняется обход. При этом, выбор очередного узла в обходе зависит от статусов ранее выполненных задач.

Основное преимущество decision tree в легкости анализа и освоения, а также простоте реализации. Несмотря на эту простоту, зачастую используются

модифицированные и усложненные алгоритмы. Так, например, деревья поведения дополняются средствами машинного обучения [17].

В алгоритме utility-based AI все доступные действия оцениваются с использованием математических формул. Затем агент выбирает курс действия с максимальной совокупной оценкой.

Каждому действию соответствует набор факторов. Оценка действия состоит в аккумулировании оценок по каждому фактору. Получение оценки каждого фактора состоит из двух этапов: получение «сырого» значения и его модификация посредством характеристической кривой (response curve). «Сырое» значение получается на основе данных игрового мира. Оно является нормализованным. Это значение пропускается через преобразователь характеристической кривой, и получается итоговая оценка, снова нормализованная.

С помощью кривой ответа дизайнер может вкладом каждого фактора в итоговую оценку. Поскольку настройка кривой ответа (а также выбор входных данных, подаваемых на нее) полностью зависит от данных (data-driven), модификация и отладка поведения также легко доступны дизайнеру, что является достоинством utility-based AI. Полученные оценки факторов затем умножаются друг на друга, чтобы получить общую оценку действия.

Алгоритм обучения с подкреплением в своей наиболее общей форме состоит из трех компонентов:

- стратегия исследования для опробования различных действий в игре;
- функция подкрепления, которая дает обратную связь о том, насколько благоприятны были последствия совершенного действия;
- правило обучения, связывающее эти два предыдущих компонента.

Для всех известных действиях в том или ином состоянии хранится так называемое Q-значение, показывающее ожидаемую (на основе

предшествующего опыта) выгоду от совершения этого действия в этом состоянии. Стратегия исследования состоит в выборе действия с наибольшим Q-значением.

Для игрового искусственного интеллекта, обучение с подкреплением имеет свои преимущества:

Во-первых, обучение с подкреплением хорошо подходит для сред с множеством взаимодействующих компонентов и большой долей неопределенности, таких как оптимизация поведения группы персонажей или нахождение последовательностей действий в игре с несколькими игроками-людьми.

Во-вторых, методика позволяет упростить представление состояний среды для обучения. Необязательно сообщать алгоритму все детали о состояниях – достаточно лишь использовать те состояния, на которых будет происходить обучение, что снижает вычислительную сложность алгоритма.

Кроме этого, данный алгоритм может быть применен для решения большого числа задач, например, для выбора тактик на основе знаний о действиях противника, или для управления движением персонажа или транспортного средства.

Hierarchical Task Network Planning (HTNP) – это алгоритм игрового искусственного интеллекта, который заключается в построении и выполнении планов для NPC. Он предлагает менее ресурсоемкий и более удобный для дизайнера подход по сравнению с планированием в реальном времени. HTNP состоит из планировщика, домена и состояния мира.

Состояние мира, как и в других алгоритмах, содержит информацию об окружающей среде, и хранит только необходимые данные для принятия решений.

Домен состоит из задач, которые могут быть примитивными или составными. Примитивные задачи – это простые последовательности действий, выполняемых NPC, с условиями и результатами, определяющими возможности и эффекты их выполнения. Операторы, на которые разбиваются

примитивные действия, представляют собой базовые действия NPC, такие как ходьба или атака.

Составные задачи – это более высокоуровневые задачи с несколькими возможными методами достижения. Каждый метод при этом представляет собой цепочку задач (примитивных или составных), образующих иерархию внутри домена. Планировщик формулирует планы, декомпозируя задачи, обходя иерархическую структуру в глубину. При этом он соотносит эффекты и условия задач, отбрасывая и переходя к новой задаче при их несоответствии.

HTNP может реагировать на изменения мира во время планирования с помощью ожидаемых эффектов, позволяя выполнить конкретные задачи без повторной оценки всего плана. Он позволяет дизайнерам контролировать действия NPC и их поведенческие шаблоны и обеспечивает лучшую производительность ЦП в сравнении с планированием в реальном времени, предлагая быстрее и более длительные планы.

Как следует из названия, в основе алгоритма целеориентироанного поведения (GOB) лежат цели. Агент может иметь несколько целей, каждая из которых имеет разную срочность достижения (важность), выраженную численным значением. Цели могут быть самыми разнообразными - от примитивных задач, таких как восстановление здоровья, до более сложных, например, победа над противником. NPC стремится уменьшить их срочность. Считается, что цель с нулевой срочностью полностью выполнена. Таким образом вместо удаления цели из списка можно сделать ее срочность нулевой.

Помимо целей, у персонажей в играх есть набор возможных действий. Доступность действий зависит от текущего состояния игры и окружения персонажа. Совершение действия изменяет значения срочности целей агента. В играх-стрелялках действия являются более атомными и могут удовлетворять конкретные цели. В более сложных моделях одно действие влияет на срочность различных целей как положительно, так и отрицательно. Некоторые действия могут привести к выполнению цели через последовательность шагов.

Алгоритм GOB выбирает действия штучно или в цепочках таким образом, чтобы минимизировать суммарную срочность целей.

1.2 Обзор алгоритма GOAP

Понятия цели и действия являются общими и фундаментальными для всех целеориентированных методов. Среди них и целеориентированное планирование. Точкой отсчета истории GOAP принято считать публикации Jeff Orkin [15], [20] . Данный алгоритм от GOB отличает то, что здесь понятие характеризуется не абстрактным «мотивом», а набором желаемых состояний. Согласно упомянутым выше источникам определяются следующие ключевые понятия.

Цель – некоторый набор условий, которые агент стремится выполнить. В любой момент времени у агента есть активная цель, задающая его поведение. При этом для цели вводится величина приоритета, численно выражающая важность одной цели относительно других и меняющаяся в зависимости от состояния агента и внешних событий.

По Оркину задача выбора текущей цели тривиальна – в каждый момент времени текущей целью является та, что обладает максимальным приоритетом.

План – последовательность действий, совершаемых агентом для достижения некоторого состояния, удовлетворяющего текущей цели (одного из *целевых состояний*) из текущего состояния.

Действие – составляющая плана, выражающая переход между двумя состояниями агента. Визуально представляет собой какое-либо деяние агента.

Для каждого действия определяется, в каких ситуациях его можно выполнить (то есть, вводятся так называемые *условия* данного действия), а также какие изменения выполнение этого действия внесет в состояние агента и окружающего мира (то есть, вводятся так называемые *эффекты* данного действия).

Последовательно устанавливая соответствия между эффектами и условиями действий, можно получить *выполнимую последовательность действий*. Например, действие Attack может быть доступно только при условии, что оружие агента заряжено, следовательно, последовательность Reload->Attack – выполнимая.

Вся вышеприведенная терминология опирается на понятие *состояния*, а GOAP во многом происходит из концепции конечных автоматов [7]. Преимущество идеи Оркина состоит в разделении состояний и переходов между ними.

Действительно, согласно теории конечных автоматов, состояние и есть действие, а переход между ними определялся исключительно самими состояниями. С ростом разнообразия внутриигровых ситуаций такой подход стал крайне неудобным, так как количество переходов, которые дизайнер должен был учесть, росло как квадрат числа состояний. Введения действия как перехода между состояниями, зависящего только от условий и эффектов, способствовало упрощению процесса планирования, так как теперь игровая логика, включающая в себя десятки различных игровых ситуаций, сосредотачивалась не в состояниях, а именно в действиях, количество которых не зависело от числа состояний. Для наглядности на рисунке 2 приведена схема конечного автомата, примененного в игре F.E.A.R. под руководством Оркина.

Как видно, модель искусственного интеллекта упростилась до трех возможных состояний: Animate, GoTo и UseSmartObject. При этом, куда именно перемещается агент в данном состоянии или какая именно анимация проигрывается, устанавливается действием.

Состояние агента задается с помощью атрибутов.

Атрибут – это некоторая характеристика агента. Примеры атрибутов: hpLeft, position, isWeaponLoaded. Атрибуты могут быть представлены различными типами: целочисленный, трехмерный вектор, логический.

Значения всех атрибутов в текущий момент времени задают его текущее состояние. В простейшем случае и условия, и эффекты – это набор пар атрибут-значение. Считают, что если значения атрибутов в условии равны значениям соответствующих атрибутов состояния, то такое состояние удовлетворяет условиям и данное действие можно выполнить.

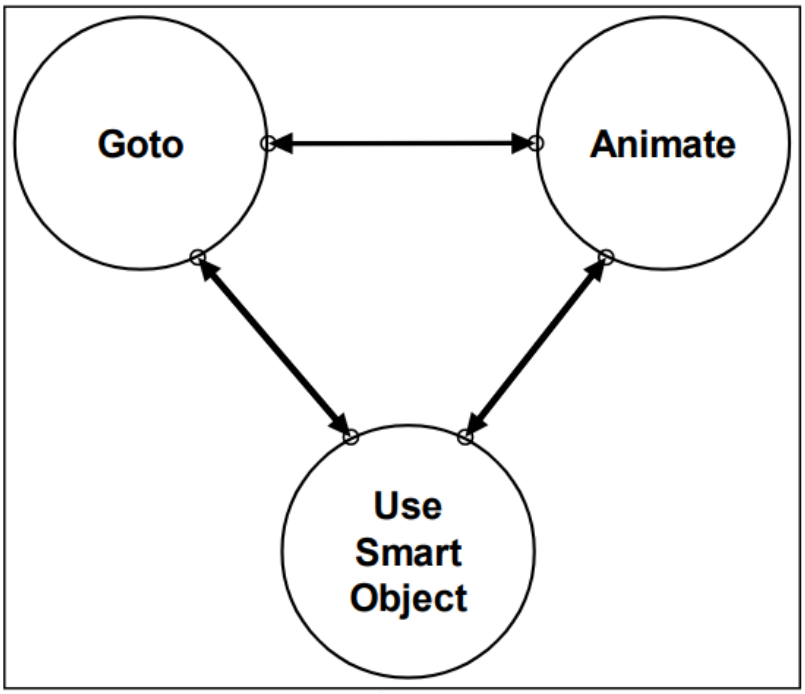

Рисунок 2 – Конечный автомат NPC в игре F.E.A.R.

Еще одно ключевое понятие GOAP – это планирование.

Планирование – процесс построения плана. Задача, которую решает GOAP-алгоритм – задача планирования, поэтому сам алгоритм будет называть *планировщиком*. Планировщик составляет из действий выполнимую последовательность действий, переводящую агента из текущего состояния в целевое - план.

План можно строить из настоящего в будущее, то есть начать выбор действий от начального состояния, или из будущего в настоящее, то есть начать выбор действий от цели. Второй подход кажется более удачным по следующей причине. Если выбирать действие на основе целевых условий, можно на ранних этапах построения отбросить те действия, что не удовлетворяют ни одному из них и, тем самым сократить пространство

решений. С другой стороны, при построении плана от начального состояния, вывод о том, что план бесперспективен, можно сделать только путем перебора всех возможных комбинаций действий.

Оркин предлагает строить план с конца, то есть от условий. Для этого вводится множество текущих условий, которое в начале планирования заполняется условиями цели. Затем, просматриваются все действия и для каждого проверяется, удовлетворяют ли его эффекты хотя бы одному из текущих условий. В случае, если действие прошло проверку, оно включается в план, удовлетворенные условия удаляются из множества текущих условий, а условия включаемого действия в него добавляются. Так повторяется до тех пор, пока начальное состояние агента не будет удовлетворять текущим условиям. Пример построения плана по такому алгоритму представлен на рисунке 3.

В данном примере цели KillEnemy соответствует значение true атрибута kIsTargetDead. Данную цель удовлетворяет действие Attack. Оно включается в план, выполненное условие убирается из множества текущих и заменяется условиями действия – kIsWeaponLoaded==true. Аналогичным образом выбираются действия LoadWeapon и DrawWeapon. Наконец, на последнем этапе, оказывается, что все значения атрибутов из условий совпадают с соответствующими значениями из начального состояния. Таким образом, получен план DrawWeapon->LoadWeapon->Attack.

При этом, если на каком-то шаге окажется, что в план по указанному критерию могут быть добавлены несколько действий (каждое из них удовлетворяет хотя бы одному условию из актуального множества условий), то процесс разветвляется, а по каждой ветке планирования далее пропускается свой план.

Процесс построения множества планов, переводящих агента из начального состояния в одно из целевых, можно интерпретировать как обход графа [21], в котором вершины представлены состояниями, а ребра –

действиями. В таком случае, задача построения плана трансформировалась в задачу о поиске пути в граф.

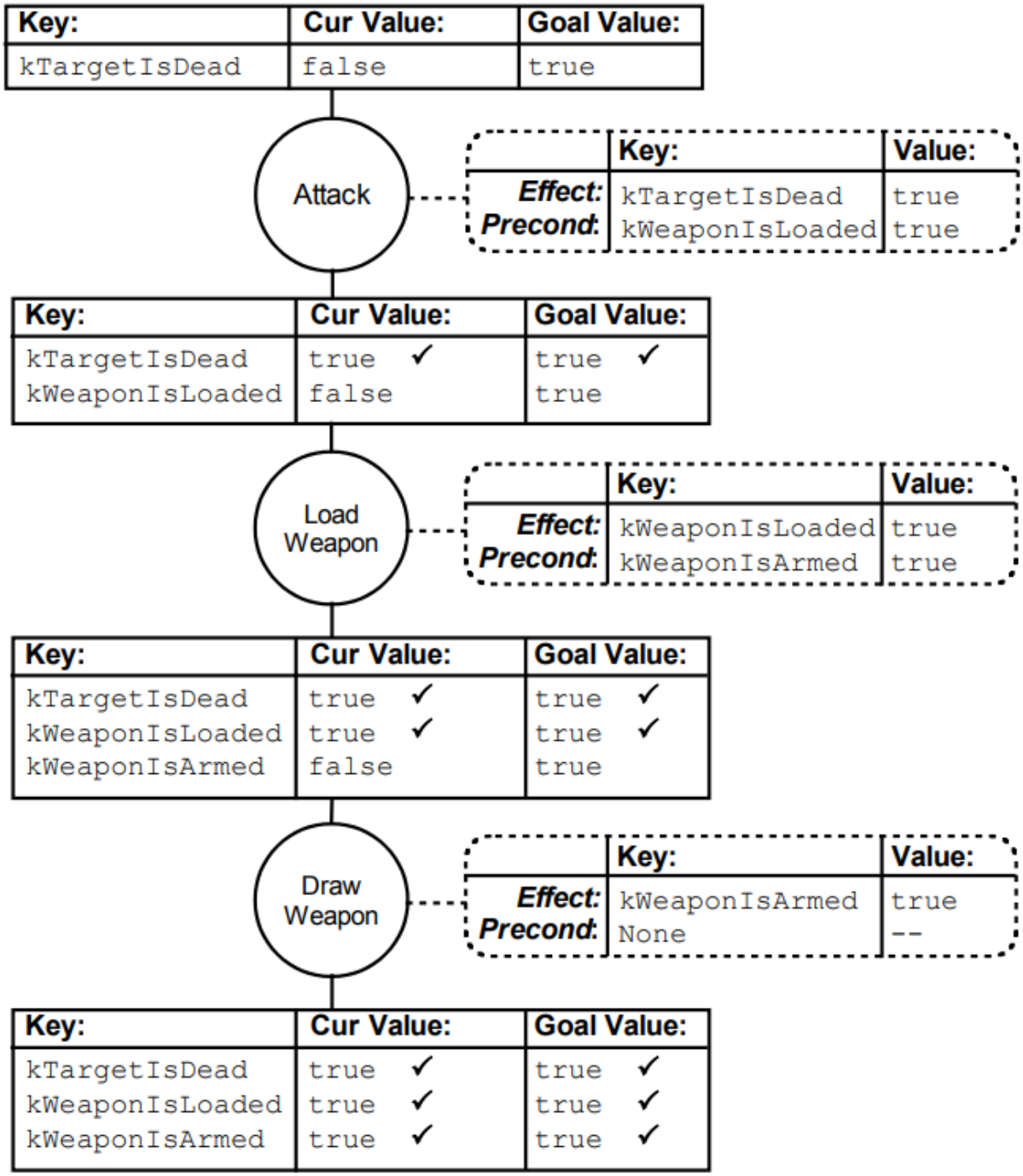

Рисунок 3 – Пример построения плана

Очевидно, встает вопрос: по какому критерию выбирать путь, то есть план. Понятно, что исходя из игровых механик, а также внешних обстоятельств, определенные действия могут быть более или менее выгодным. Мерой «невыгодности» действия служит числовая величина цены.

Цена может отражать различные аспекты действия. Она может быть фиксированной или меняться в зависимости от внешних факторов. Например, для действия GoTo цена может быть пропорциональна длине пути, который предстоит преодолеть, а для действия Attack она может быть пропорциональна ожидаемому шансу успешной атаки, которая зависит от сиюминутной расстановки сил в игре.

Зная цены всех действий в плане, можно рассчитать стоимость самого плана как сумму всех этих цен. Тогда понятно, что наиболее выгодным будет план с минимальной стоимостью – «самый дешевый». Тогда длину ребра в полученном графе планирования можно принять равной стоимости действия. Тогда задача поиска оптимального плана формулируется как задача нахождения кратчайшего пути в графе.

Эта задача имеет множество алгоритмов решения [22], [23]. Стандартом считается эвристический алгоритм A* [24]. В качестве эвристики Оркин предлагает принимать количество условий из множества текущих условий, неудовлетворенных начальным состоянием. Это значение является приемлемым в рамках простых моделей, но позже будет предложена иная, кажущаяся более удачной эвристика.

Если удалось построить план, то агент приступает к его выполнению и следует вплоть до достижения цели. Однако, если до этого текущая цель меняется, то выполнение текущего плана прерывается, строится план для новой цели, и он становится текущим.

1.3 Преимущества алгоритма GOAP

На рисунке 4 [25] изображена динамика развития игрового искусственного интеллекта на протяжении всей истории игровой индустрии. На этом рисунке в хронологическом порядке представлены применяемые сегодня алгоритмы игрового искусственного интеллекта. Наблюдается процесс перехода от безусловного прямого контроля над агентами в пользу автономных систем.

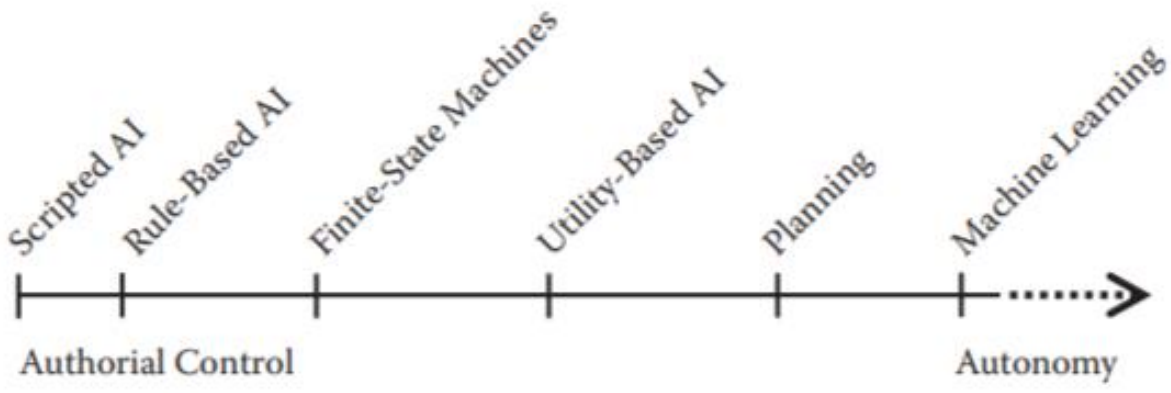

Рисунок 4 – Динамика развития игрового искусственного интеллекта

Рисунок демонстрирует основное преимущество целеориентированного планирования над такими алгоритмами, как rule-based AI, FSM, utility-based AI – возможность создавать непредсказуемое, адаптивное поведение. Модульность GOAP, обеспечиваемая отказом от переходов, тесно связанных с состояниями (что характерно для FSM и rule-based AI), в пользу более гибких действий, доступных из многих состояний, позволяет создавать более «человекоподобных» агентов.

Кроме того, достоинством GOAP является структурированность и масштабируемость. Этот алгоритм можно применять как для сиюминутного выбора действия (построение короткого плана), так и для долгосрочного планирование (построение стратегии). Это в меньшей степени удобно для других подходов.

Алгоритмы машинного обучения, такие как, например, обучение с подкреплением, в перспективе могут обогнать GOAP по качеству и реалистичности получаемого поведения. Тем не менее наиболее явным недостатком алгоритмов обучения в сфере игрового искусственного интеллекта является низкая доступность и популярность, сложность отладки и визуализации, а также высокая вычислительная трудоемкость. В текущий момент алгоритмы машинного обучения являются наименее доступным инструментом для проектирования NPC в промышленном геймдизайне.

1.4 Недостатки алгоритма GOAP

К недостаткам классического алгоритма GOAP можно отнести его следующие особенности:

1) Высокая чувствительность решений к распределению приоритетов

   Как было сказано ранее, приоритеты целей определяются нуждами агента и внешними обстоятельствами. Определение значений приоритетов – целиком задача гейм-дизайнера. Ее решение требует глубокого анализа и необходимости предсказания всевозможных игровых ситуаций, что зачастую невозможно.

   Например, приоритет цели StayHealthy может повышаться при снижении текущего уровня здоровья. В какой-то момент он может превысить приоритет цели KillEnemy, и агент будет искать возможности повысить уровень здоровья. Такое поведение может быть благоприятным не во всех ситуациях. Его реалистичность зависит исключительно от того, как удачно были выбраны функции расчета приоритетов дизайнером. Чем больше геймплейных ситуаций становятся возможными в игре, тем объемнее вычисления. Для того, чтобы иметь возможность регулировать вклад приоритета в процесс выбора текущей цели, требуется другая величина, определяемая единым образом для всех целей.

2) «Жадный» выбор текущей цели

   Выбор текущей цели определенно влечет изменения для планов будущих целей. Понятно, что выбор конкретной цели может увеличить стоимость планов, либо их уменьшить. В качестве примера рассмотрим следующую ситуацию.

Модельный пример изображен на рисунке 5. Предположим, в начальный момент времени агент находится в точке О. Имеются две цели: одна из них требует прибытия агента в точку A (цель A), другая – в точку B (цель B). Таким образом, возможны два варианта пути для агента: $O \to A \to B$ и $O \to B \to A$. При этом, первый вариант длиннее второго. Очевидно, что, если приоритет

цели A больше приоритета цели B, то агенту придется пройти по длинному пути. Поэтому, можно сделать вывод, что, если разность приоритетов A и B не так велика по сравнению с самими приоритетами, стоит выбрать второй вариант.

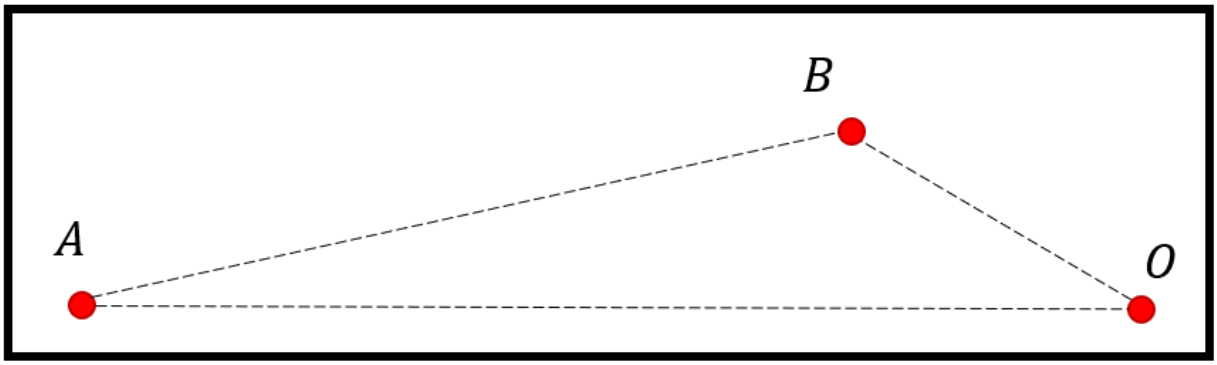

Рисунок 5 – Пример неоптимального «жадного» выбора текущей цели

Итак, возникает необходимость учитывать побочные эффекты достижения конкретной цели раньше других, а значит требуется определить некоторую фитнесс функцию, зависящую от порядка достижения целей. Это и было сделано в процессе исследования.

Выводы по разделу 1

В разделе проведен обзор популярных поведенческих алгоритмов в игровом ИИ, а также их сравнение с алгоритмом целеориентирвоанного планировния, показаны его преимущества. В ходе обзора обозначены основные понятия и принципы модернизируемого алгоритма, по шагам разобран ход его выполнения. Работа алгоритма продемонстрирована на примере.

Анализ существующих реализаций подхода GOAP позволил выделить следующие его недостатки:

– Ограниченность в сложности поведения

  GOAP хорошо подходит для создания простых и структурированных поведенческих моделей, но может оказаться недостаточно гибким для реализации сложных и непредсказуемых поведений. Кроме того,

чем сложнее модель окружающего агента мира, тем сложнее геймдизайнеру предсказать возможные игровые ситуации.

– Трудности с адаптацией к изменяющейся среде

  GOAP может столкнуться с трудностями при адаптации к динамически изменяющейся среде, так как планы действий могут быть заранее построены и не учитывать новые обстоятельства. Тогда при изменении состояния окружающего мира агент будет вынужден перестраивать текущий план, кратно увеличивая количество требуемых вычислений.

– Сложность настройки и зависимость от эвристик

  Настройка GOAP может быть сложной и требовать значительного времени и усилий. Не всегда легко определить правильные условия, эффекты и стоимости действий для получения желаемого поведения. Кроме того, эвристики, такие как приоритет цели, могут вносить нежелательный человеческий фактор, делающим поведение агента менее реалистичным.

– Игнорирование опыта прошлых итераций

  В пределах своего жизненного цикла множество возможных состояний агента, а также множество его целей ограничены. Следовательно, зачастую агенту приходится планировать в знакомых обстоятельствах. Например, может случиться так, что для данных цели и начального состояния ранее уже был построен план. Данные, полученные в результате прошлых итераций планирования, можно использовать для в качестве дополнительного фактора при построении очередного плана.

Обнаруженные недостатки проиллюстрированы примером.

Последние два пункта особенно критичны, если мы вспомним, что в отличие от академического искусственного интеллекта, главная задача

которого – достичь поставленной цели при наименьших затратах, игровой искусственный интеллект в первую очередь должен обладать свойством достоверности и правдоподобно имитировать человеческое поведение. Чувствительность системы планирования к назначаемым геймдизайнером эвристикам, а также не учитывание агентом ранее накопленной им информации, делает поведение агентов менее «человекоподобным».

Недостатки усложняют настройку алгоритма, а также снижают правдоподобность и реалистичность получаемого поведения агентов и, очевидно, требуют исправления.

2 Разработка алгоритма

2.1 Общие принципы разработанной модификации

Ранее было показано, что порядок выполнения целей влияет на трудоемкость достижения отдельных целей. По аналогии с планом, определяющим порядок совершения действий, введем понятие стратегии. *Стратегия* – последовательность целей. Главным отличием модифицированного алгоритма от классического состоит в том, что классический алгоритм лишь осуществлял выбор очередной цели, а модифицированный – построение стратегии, то есть цепочки целей. На рисунке 6 представлена схема разработанного модифицированного алгоритма принятия решений.

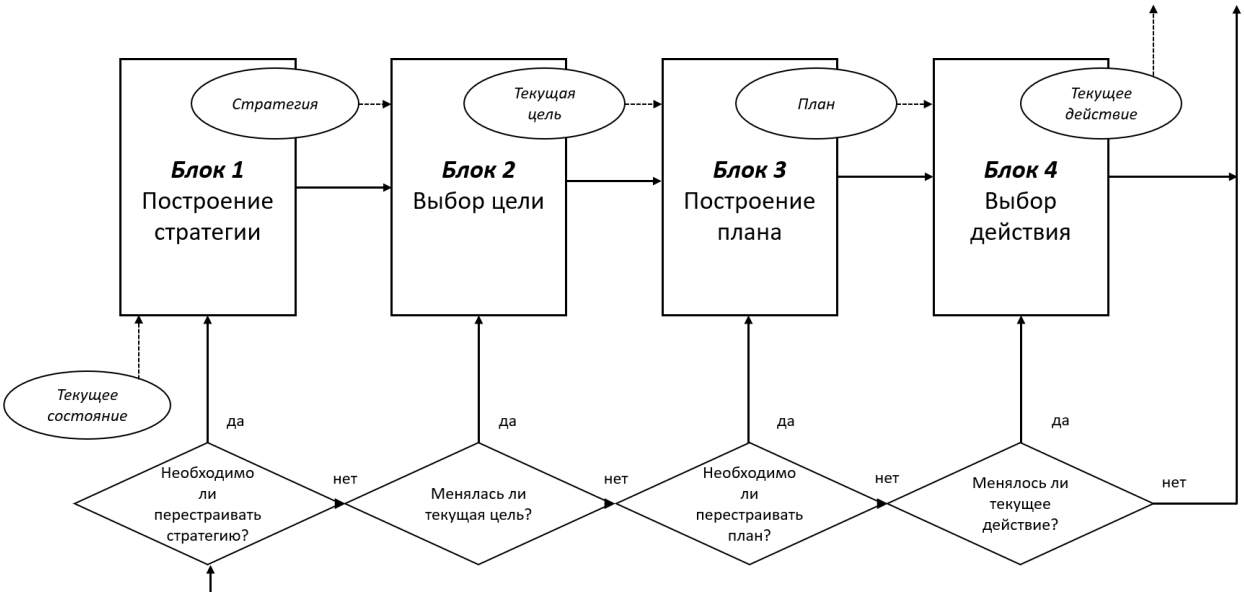

Рисунок 6 – Схема работы разработанной системы принятия решений

Система принятия решений начинает свой цикл с построения стратегии и выбора в ней первой цели, затем для выбранной цели строится план и в нем выбирается первое действие. В будущем, если не возникает необходимости в перестроении стратегии или плана, система просто инкрементирует счетчик текущей цели или действия в случае, если цель была достигнута или действие – выполнено. В эллипсах указаны блоки данных, передаваемые между блоками исполнения (прямоугольники).

В классическом алгоритме блок 1 отсутствует. Блоки 2-4 соответствуют этапам цикла, изображенному на рисунке 1.

Цель разрабатываемой модификации – выбор предпочтительной стратегии.

Для того, чтобы оценить предпочтительность той или иной стратегии, необходимо ввести определенную числовую меру – стоимость стратегии [13]. Чем выше стоимость стратегии, тем больше затрат требуют совокупно планы всех включенных в нее целей. Таким образом, предпочтительная (оптимальная) стратегия обладает минимальной стоимостью.

Введем следующие обозначения:

- множество целей $\Theta = \{G_1, G_2, \dots, G_k\}$;
- приоритет q_i цели G_i;
- стратегия $C(\Theta) = \{G_{i_1}, G_{i_2}, \dots, G_{i_k}\}$ – перестановка множества Θ – и ее стоимость $c(C) = c(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_k})$ с учетом начального состояния $\boldsymbol{s}^0$;
- упорядоченное множество атрибутов $\mathrm{A} = \{a_1, a_2, \dots, a_n\}$;
- вектор-состояние агента $\boldsymbol{s} = \{x_1, x_2, \dots, x_n\}$, где x_i – значение i-того атрибута, соответствующее данному состоянию;
- план перехода из состояния $\boldsymbol{s}^0$ в целевое состояние $\boldsymbol{s}$ цели G по плану $\boldsymbol{s} = P(\boldsymbol{s}^0, G)$ и стоимость этого плана $p(\boldsymbol{s}^0, G)$.

Таким образом, задача поиска оптимальной стратегии сводится к нахождению минимума стоимости стратегии:

$$C^* = \{G_{i_1^*}, G_{i_2^*}, \dots, G_{i_k^*}\} = \arg \min_{G_{i_1}, G_{i_2}, \dots, G_{i_k} \in \Theta} c(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_k}). \quad (1)$$

2.2 Определение стоимости стратегии

Теперь необходимо подобрать такую функцию $c(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_k})$, которая отвечала бы поставленным перед разработкой алгоритма целям. При построении этой функции принимались во внимание следующие наблюдения:

– Затраты на выполнение той или иной цели (стоимость плана) зависят от того, какие цели были выполнены ранее. Действительно, действия, совершаемые агентом, являются переходами между состояниями. Стоимость плана по достижению цели есть затраты на выполнение всех включенных в него действий. План и его стоимость зависят также и от начального состояния, следовательно, на затраты на выполнение очередной цели влияет то, какие планы были построены в прошлом, то есть, какие цели уже были достигнуты на момент построения плана очередной цели.
– На стоимость стратегии влияет приоритет целей. Приоритет отражает эвристику, заданную геймдизайнером и является инструментом прямого контроля над порядком выполнения целей, поэтому модифицированный алгоритм должен учитывать этот параметр при построении стратегии.

Стоимость стратегии можно рассматривать как сумму:

$$c\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_k}\right) = \\ = \alpha\left(\boldsymbol{s}^0, G_{i_1}\right) + \alpha\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}\right) + \cdots + \alpha\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_k}\right), \quad (2)$$

где слагаемое $\alpha\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right)$ отражает траты на достижение расположенной на j-той позиции в стратегии цели G_{i_j} после достижения целей $G_{i_1}, G_{i_2}, \dots, G_{i_{j-1}}$. Это слагаемое должно иметь выражение согласно двум соображениям, изложенным выше.

Будем определять слагаемое в сумме (2) следующим образом:

$$\alpha\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right) = \pi\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right) + \tau_{i_j}(j), \quad (3)$$

где

– π - издержки от включения плана $P\left(\boldsymbol{s}^{\boldsymbol{i_{j-1}}}, G_{i_j}\right)$ в стратегию. Состояние $\boldsymbol{s}^{\boldsymbol{i_{j-1}}}$, получается рекуррентно по формуле:

$$\boldsymbol{s^{i_{j-1}}} = P\left(\boldsymbol{s^{i_{j-2}}}, G_{i_{j-1}}\right), j \in \overline{(2,k)}, \boldsymbol{s^{i_0}} = \boldsymbol{s^0}, \quad (4)$$

Издержки тем выше, чем дороже план и чем менее вероятно, что он на практике приведет к достижению цели.

– $\tau_{i_j}(j)$ – функция задержки цели G_{i_j}, которая показывает издержки от постановки этой цели на j-тую позицию в стратегии. Чем выше значение $\tau_i(s)$, тем менее выгодно ставить цель G_i на s-тую позицию в стратегии. Очевидно, задержка отвечает следующим требованиям:

$$\tau_i(s) > \tau_i(t) \Leftrightarrow s > t\ , i \in \left(\overline{1,k}\right), \quad (5)$$

$$\tau_s(i) > \tau_t(i) \Leftrightarrow q_s > q_t, i \in \left(\overline{1,k}\right). \quad (6)$$

На практике функцию задержки можно рассматривать как эвристику, при условии удовлетворения требований (5) и (6), то есть некоторую возрастающую функцию от приоритета цели и индекса ее позиции в стратегии.

2.3 Определение меры издержек плана

В качестве $\pi\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right)$ можно было бы принять реальную стоимость плана $p\left(s^{i_{j-1}}, G_{i_j}\right)$, однако это не представляется возможным, ибо стоимость плана зависит от непрерывно меняющихся во времени внешних факторов, которые невозможно предсказать заранее, на стадии построения стратегии.

Вместо этого, в силу того, что, издержки от включения плана в стратегию тем выше, чем дороже сам план и чем менее вероятно достижение цели в результате следования данному плану, $\pi\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right)$ вычислялась по следующей формуле:

$$\pi\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \dots, G_{i_j}\right) = \tilde{p}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) \cdot \beta\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right), \quad (7)$$

где $\tilde{p}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$ – оценка стоимости плана, получаемая с помощью метода линейной регрессии, а $\beta\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) \in [1; \beta_{max}]$ – коэффициент недостижимости. Полагалось

$$\beta\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) = (1 - \beta_{max}) \cdot \tilde{b}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) + \beta_{max}, \quad (8)$$

где $\tilde{b}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) \in [0; 1]$ – вероятность достижения цели путем выполнения плана $P\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$ – также определялась путем регрессии. Для фиксированных начального состояния и цели получилась зависимость $\beta(\tilde{b})$, представленная на рисунке 7.

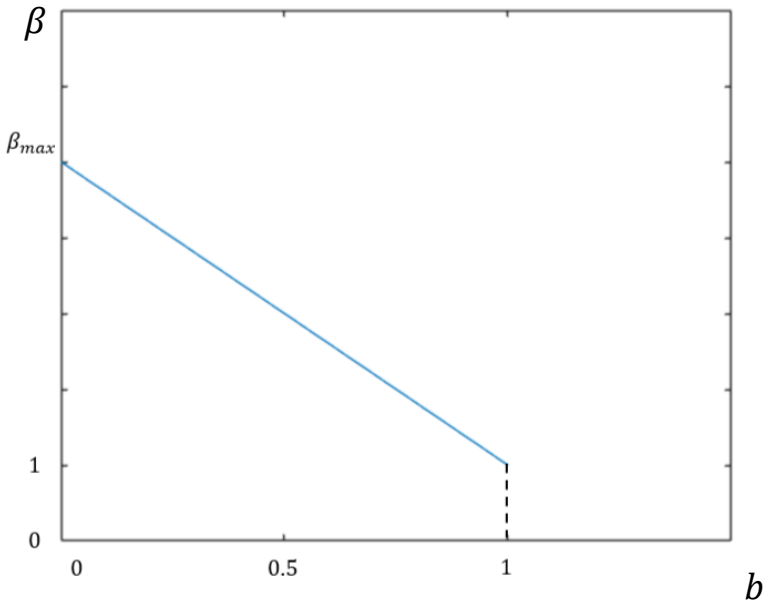

Рисунок 7 – Зависимость коэффициента недостижимости от вероятности достижения цели

β_{max} выбирался так, чтобы, если вероятность успешного достижения цели данным планом мала (т.е. $\beta\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) \approx \beta_{max}$), то соответствующее значение $\pi\left(\boldsymbol{s}^0, G_{i_1}, G_{i_2}, \ldots, G_{i_j}\right)$ было достаточно большим, чтобы план $P\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$ не был бы выбран на стадии построения стратегии.

Наконец, состояние $\boldsymbol{s}^{i_j} = P\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$, необходимое для определения $\tilde{p}\left(\boldsymbol{s}^{i_j}, G_{i_{j+1}}\right), \tilde{b}\left(\boldsymbol{s}^{i_j}, G_{i_{j+1}}\right)$ также определялось с помощью линейной регрессии. То есть, полагалось $\boldsymbol{s}^{i_j}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right) = \tilde{\boldsymbol{s}}^{i_j}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$.

2.4 Применение линейной регрессии

Для интеграции фактора опыта прошлого были применены средства машинного обучения, а именно линейной регрессии.

Линейная регрессия [26] – один из самых простых методов машинного обучения, что делает его легко понятным и интерпретируемым, что позволяет легко интегрировать этот метод в существующий код и структуру игры.

Кроме того, линейная регрессия обладает небольшой вычислительной сложностью, что позволяет использовать ее в реальном времени в игровых средах без значительного увеличения нагрузки на систему.

Наконец, линейная регрессия обладает небольшой вычислительной сложностью, что позволяет использовать ее в реальном времени в игровых средах без значительного увеличения нагрузки на систему.

В процессе вычисления необходимых величин для различных целей $G_1, G_2 \dots G_k$ будут использоваться различные регрессионные модели в силу того, что характер цели в некоторой степени определяет стоимость плана и вероятность ее достижения. В этом разделе будут рассматриваться принципы построения модели для отдельной фиксированной цели, поэтому второй аргумент в регрессантах $\tilde{p}\left(\boldsymbol{s}^{i_j}, G_{i_{j+1}}\right)$, $\tilde{b}\left(\boldsymbol{s}^{i_j}, G_{i_{j+1}}\right)$ и $\boldsymbol{s}^{i_j}\left(\boldsymbol{s}^{i_{j-1}}, G_{i_j}\right)$ будет далее опущен. Кроме того, для краткости обозначим строки $\boldsymbol{s} = \boldsymbol{s}_{1\times(n+1)} = \left\{1, \boldsymbol{s}^{i_{j-1}}\right\}$, $\boldsymbol{s}^*(\boldsymbol{s}) = \boldsymbol{s}^*_{1\times(n+1)}(\boldsymbol{s}) = \left\{1, \boldsymbol{s}^{i_j}(\boldsymbol{s})\right\}$, $\tilde{\boldsymbol{s}}^* = \tilde{\boldsymbol{s}}^*_{1\times(n+1)} = \left\{1, \tilde{\boldsymbol{s}}^{i_j}(\boldsymbol{s})\right\}$. Размерности векторов были повышены на 1, чтобы от решений в виде аффинных функций перейти к чисто линейным.

Для определения скалярных величин $\tilde{p}(\boldsymbol{s})$ и $\tilde{b}(\boldsymbol{s})$ применялись классические формулы (9) и (10) линейной регрессии [27], [28]:

$$\tilde{p}(\boldsymbol{s}) = \boldsymbol{s}\boldsymbol{w}_{\boldsymbol{p}}, \quad (9)$$

$$\tilde{b}(\boldsymbol{s}) = \boldsymbol{s}\boldsymbol{w}_{\boldsymbol{b}}, \quad (10)$$

где $\boldsymbol{w_p} = \boldsymbol{w_p}_{(n+1)\times 1}$ и $\boldsymbol{w_b}_{(n+1)\times 1}$ – векторы-столбцы коэффициентов регрессии, получаемые для данной цели по формулам (11), (12).

$$\boldsymbol{w_p} = (\boldsymbol{S}^T\boldsymbol{S})^{-1}\boldsymbol{S}^T\boldsymbol{p} \quad (11)$$

$$\boldsymbol{w_b} = (\boldsymbol{S}^T\boldsymbol{S})^{-1}\boldsymbol{S}^T\boldsymbol{b} \quad (12)$$

В формулах (10) и (11) $\boldsymbol{p} = \boldsymbol{p}_{m\times 1}$ и $\boldsymbol{b} = \boldsymbol{b}_{m\times 1}$ – векторы входных параметров, собранных в результате m экспериментов, $\boldsymbol{S} = \boldsymbol{S}_{m\times(n+1)}$ – (невырожденная) матрица информации, составленная из m векторов-строк $\boldsymbol{s_1}, \boldsymbol{s_2}, \dots, \boldsymbol{s_m}$, собранных в результате m экспериментов. В этой матрице i-тый столбец содержал значения i-того атрибута (кроме первого столбца, он был заполнен единицами).

Осталось получить формулу для вычисления $\tilde{\boldsymbol{s}}^*(\boldsymbol{s})$. Так как $\tilde{\boldsymbol{s}}^*(\boldsymbol{s})$– векторная величина, требуется переписать классические формулы мультилинейной регрессии [27] для случая векторного регрессанта.

Согласно методу линейной регрессии будем искать решение в виде:

$$\widetilde{\boldsymbol{s}^*} = \boldsymbol{s}\boldsymbol{W}_{\boldsymbol{s}^*}, \quad (13)$$

где $\boldsymbol{W}_{\boldsymbol{s}^*}$ – матрица коэффициентов линейной регрессии размерности $(n+1)\times(n+1)$.

Согласно [27] искомый вектор $\tilde{\boldsymbol{s}}^*(\boldsymbol{s})$ минимизирует квадрат нормы вектора ошибок, т.е.

$$\boldsymbol{W}_{\boldsymbol{s}^*} = \arg\min_{\boldsymbol{\Omega}} \|\boldsymbol{S}^* - \boldsymbol{S}\boldsymbol{\Omega}\|^2. \quad (14)$$

Здесь матрица входных параметров $\boldsymbol{S}^* = \boldsymbol{S}^*_{m\times(n+1)}$ аналогично матрице $\boldsymbol{S}$ составлена из m векторов-строк $\boldsymbol{s_1^*}, \boldsymbol{s_2^*}, \dots, \boldsymbol{s_m^*}$, собранных в результате m экспериментов.

Таким образом искомая матрица $\boldsymbol{W}_{\boldsymbol{s}^*}$ является минимумом функции $\|\boldsymbol{S}^* - \boldsymbol{S}\boldsymbol{\Omega}\|^2$. Кроме того, в качестве матричной нормы используется норма Фробениуса как естественное расширение евклидовой векторной нормы [29],

вследствие чего выражения для минимизируемой функции можно переписать в виде:

$$\begin{gathered}\|\boldsymbol{S}^{*}-\boldsymbol{S\Omega}\|^{2}=\mathrm{tr}\left((\boldsymbol{S}^{*}-\boldsymbol{S\Omega})^{\mathrm{T}}(\boldsymbol{S}^{*}-\boldsymbol{S\Omega})\right)= \\ =\mathrm{tr}\left((\boldsymbol{S}^{*T}-\boldsymbol{\Omega}^{\mathbf{T}}\boldsymbol{S}^{T})(\boldsymbol{S}^{*}-\boldsymbol{S\Omega})\right)= \\ =\mathrm{tr}\left(\boldsymbol{S}^{*T}\boldsymbol{S}^{*}-\boldsymbol{S}^{*T}\boldsymbol{S\Omega}-\boldsymbol{\Omega}^{T}\boldsymbol{S}^{T}\boldsymbol{S}^{*}+\boldsymbol{\Omega}^{T}\boldsymbol{S}^{T}\boldsymbol{S\Omega}\right).\end{gathered} \tag{15}$$

Для нахождения минимума $\boldsymbol{W}_{\boldsymbol{S}^{*}}$ этой функции приравняем ее производную по $\boldsymbol{\Omega}$ к 0. Для нахождения матричных производных воспользуемся формулами, приведенными в [30]

$$\begin{gathered}\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}\left(\boldsymbol{S}^{*T}\boldsymbol{S}^{*}-\boldsymbol{S}^{*T}\boldsymbol{S\Omega}-\boldsymbol{\Omega}^{T}\boldsymbol{S}^{T}\boldsymbol{S}^{*}+\boldsymbol{\Omega}^{T}\boldsymbol{S}^{T}\boldsymbol{S\Omega}\right)\right)= \\ =\frac{\partial}{\partial\boldsymbol{\Omega}}\left(tr\left(\boldsymbol{S}^{*T}\boldsymbol{S}^{*}\right)\right)-\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}\left(\boldsymbol{S}^{*T}\boldsymbol{S\Omega}\right)\right)-\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}(\boldsymbol{\Omega}^{\mathbf{T}}\boldsymbol{S}^{T}\boldsymbol{S}^{*})\right)+ \\ +\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}(\boldsymbol{\Omega}^{\mathbf{T}}\boldsymbol{S}^{T}\boldsymbol{S\Omega})\right)\end{gathered} \tag{16}$$

Определим каждое слагаемое в (16):

$$\frac{\partial}{\partial\boldsymbol{\Omega}}\left(tr\left(\boldsymbol{S}^{*T}\boldsymbol{S}^{*}\right)\right)=0 \tag{17}$$

$$\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}\left(\boldsymbol{S}^{*T}\boldsymbol{S\Omega}^{T}\right)\right)=\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}(\boldsymbol{\Omega S}^{T}\boldsymbol{S}^{*})\right)=\boldsymbol{S}^{*T}\boldsymbol{S} \tag{18}$$

$$\frac{\partial}{\partial\boldsymbol{\Omega}}\left(\mathrm{tr}(\boldsymbol{\Omega S}^{T}\boldsymbol{S}^{*})\right)=(\boldsymbol{S}^{T}\boldsymbol{S}^{*})^{\boldsymbol{T}}=\boldsymbol{S}^{*\boldsymbol{T}}\boldsymbol{S} \tag{19}$$

$$\frac{\partial}{\partial\boldsymbol{\Omega}}\mathrm{tr}\left((\boldsymbol{\Omega}^{\mathbf{T}}\boldsymbol{S}^{T}\boldsymbol{S\Omega})\right)=\boldsymbol{S}^{T}\boldsymbol{S\Omega}+(\boldsymbol{S}^{\boldsymbol{T}}\boldsymbol{S})^{\boldsymbol{T}}\boldsymbol{\Omega}=\mathbf{2}\boldsymbol{S}^{\boldsymbol{T}}\boldsymbol{S\Omega} \tag{20}$$

Подставив формулы (17) – (20) в (16), получим:

$$2\left(-\boldsymbol{S}^{*\boldsymbol{T}}\boldsymbol{S}+\boldsymbol{S}^{\boldsymbol{T}}\boldsymbol{S\Omega}\right)=0. \tag{21}$$

Решив (21) относительно $\boldsymbol{\Omega}$, получим:

$$\boldsymbol{W}_{\boldsymbol{S}^{*}}=(\boldsymbol{S}^{\boldsymbol{T}}\boldsymbol{S})^{-\mathbf{1}}\boldsymbol{S}^{\boldsymbol{T}}\boldsymbol{S}^{*}. \tag{22}$$

2.5 Алгоритм модификации

Итак, после получения формул вычисления всех необходимых величин, можно сформулировать обобщенный алгоритм модификации.

1. Для целей $G_1, G_2, \dots, G_k$ в результате построения соответственно $m_1, m_2, \dots, m_k$ планов сохраняются кортежи данных: $\{\boldsymbol{S}_i, \boldsymbol{p}_i, \boldsymbol{b}_i, \boldsymbol{S}_i^*\}, i \in (\overline{1,k})$.
2. Для целей $G_1, G_2, \dots, G_k$ по формулам (11), (12) и (22) определяются векторы и матрицы коэффициентов регрессии $\left\{\boldsymbol{w}_{p_i}, \boldsymbol{w}_{b_i}, \boldsymbol{W}_{s^*i}\right\}, i \in (\overline{1,k})$.
3. Для выбора оптимальной стратегии перебираются все стратегии (т.е. перестановки множества целей) с поиском минимума по стоимости, определяемой формулой (2). Для последовательного расчета слагаемых в этой формуле используются формулы (3), (4), а также выбранная функция задержки, удовлетворяющая условиям (5) и (6). Мера издержек плана определяется по формуле (7). Для расчета коэффициента недостижимости применяется формула (8), куда подставляется вероятность достижения цели, полученная по формуле (10). Для определения оценки стоимости плана используют формулу (9). Для первого аргумента следующего слагаемого (начального состояния плана) применяют формулу (13).

2.6 Выбор языка программирования

Для реализации системы принятия решений в качестве языка программирования был выбран C++. Этот выбор продиктован несколькими причинами.

Во-первых, C++ является одним из самых быстрых и эффективных языков программирования, что особенно важно для системы искусственного интеллекта, которая должна обрабатывать большие объемы данных в режиме реального времени.

Во-вторых, популярность C++ и его активная поддержка в многочисленных сферах применения позволяет использовать готовые программные решения на различных этапах написания кода, что и было сделано в ходе работы.

Наконец, C++ имеет широкую поддержку в индустрии разработки игр. В частности, написание программы на этом языке позволит легко внедрить ее в проект, выполненный в Unreal Engine. Именно это и послужило основным фактором, повлиявшим на выбор этого языка.

2.7 Реализация алгоритма A*

В связи с тем, что рассмотренные реализации, такие как [31] и [32], пригодны лишь для навигационных задач, в рамках ВКР был разработана программа, пригодная для построения планов с применением алгоритма A*.

В качестве основы классического алгоритма целеориентированного планирования с опорой на реализации [31] [32] был реализован шаблонный класс AStarSolver. Его метод Pathfind реализует алгоритм поиска пути A*. Класс выполнен в виде фреймворка и предназначен для наследования с конкретными реализациями различных операций, таких как расчет эвристического значения, получение соседних вершин и проверка условий завершения поиска. Код метода представлен в приложении 1. Остановимся на его ключевых элементах.

Структура Node представляет собой узел в графе. Она содержит поля VertexId, DistFromStart и Heuristic. Они применяются для расчета приоритета узлов в алгоритме поиска пути.

В качестве шаблонного параметра t_vertex передается тип данных, ассоциируемый с экземпляром Node и содержащий данные, характерные для решаемой задачи. В случае планирования, это могут быть данные об активном множестве условий или ранее включенных в план действиях. Экземпляры t_vertex передаются в перегружаемые в наследнике класса методы в качестве аргументов.

Метод Pathfind реализует алгоритм A* с использованием Фибоначчиевой кучи для эффективных операций с очередью с приоритетом. Он итеративно просматривает узлы, вычисляет длину пути из начальной вершины и обновляет лучший найденный до сих пор путь. Фибоначчева куча discQueue применялась для быстрого определения узла с наименьшей длиной пути, а в контейнерах discovered и expanded сохранялись узлы, требующие будущего просмотра и уже просмотренные узлы соответственно. В качестве реализации Фибоначчевой кучи использовалось решение [33].

Класс предоставляет несколько защищенных виртуальных методов, которые должны быть переопределены в подклассах:

- GetHeuristic вычисляет эвристическое значение для заданной вершины;
- GetNeighbors получает соседние вершины и их расстояния от заданной вершины;
- Satisfies проверяет, удовлетворяет ли вершина целевому условию;
- GetDistanceDenominator и GetHeuristicsDenominator возвращают максимально возможные расстояние и эвристику соответственно, эти значения используются для нормировки.

2.8 Реализация планировщика

За построение планов отвечал метод ConstructPlan класса Planner, наследника AStarSolver. Остановимся на основных аспектах соответствующего кода.

В качестве шаблонного параметра t_vertex использовалась структура Vertex с полями:

- ActiveConditionSet – активное на данном этапе построения множество условий;
- PrevActionId – идентификатор типа предыдущего действия в плане;
- PrevActionInstance хранит условия, эффекты и цену последнего добавленного в план действия;

– ActionCtr – количество ранее добавленных в план действий.

Класс Planner содержит реализации следующих методов интерфейса AStarPlanner:

– GetNeighbors пробегает по всем доступным действиям и выбирает те, что уменьшают активное множество условий, для каждого из выбранных типов действий строятся новые вершины, содержащие модифицированные активные множества условий;
– Satisfies проверяет, удовлетворяет ли активное множество условий вершины начальному состоянию планирования;
– GetHeuristic вычисляет эвристику вершины как сумму расстояний условий из активного множества вершины и значений атрибутов в начальном состоянии, подробнее о расстоянии атрибута рассказано в третьем разделе;
– GetDistanceDenominator и GetHeuristicsDenominator.

Код методов GetNeighbors и GetHeuristics представлен в приложениях Б и В соответственно. На рисунке 8 представлен пример результата корректной работы планировщика.

Достоинством реализованной программы является возможность пользователя определить свои классы, поддерживающие интерфейсы IAttribute, Goal и IAction, инкапсулирующие логику атрибута, цели и действия соответственно.

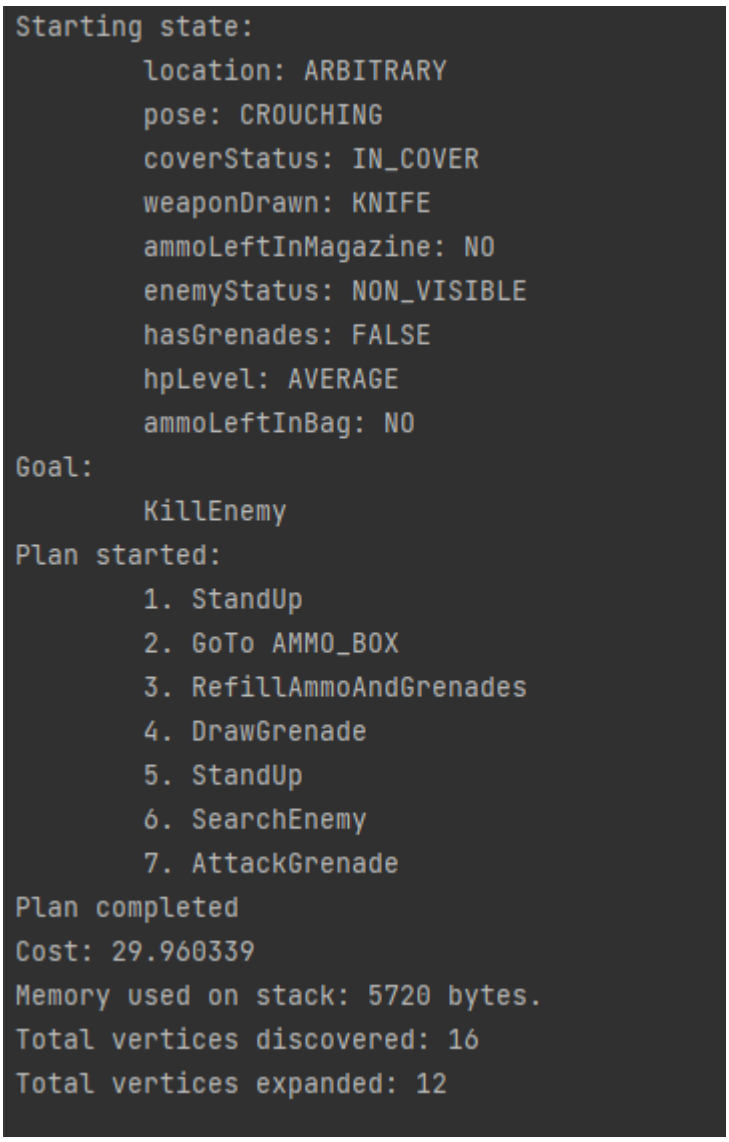

Рисунок 8 – Отчет о корректном построении плана

2.9 Реализация блока построения стратегии

Для реализации непосредственно самой модификации был разработан класс Strategist. Его объявление представлено ниже, а код основных методов - в приложении 4.

```
class Strategist
{
public:
    Strategist(const DataBase& data, bool mustUseRegression = false);
    void ConstructStrategy(const ValueSet& initState, Strategy& strategy) const;
private:
    float GetPlanCostEstimate(const ValueSet& initState, int goalId) const;
```

```
    ValueSet GetResultStateEstimate(const ValueSet& initState, int goalId) const;
    const DataBase& _data;
    std::vector<VectorXf> gwp;
    std::vector<VectorXf> gwb;
    std::vector<MatrixXf> gws;
    const float BETA_MAX = 5.0f;
};
```

Рассмотрим его основные элементы:

- конструктор, если алгоритм запущен в модифицированном режиме, читает собранные данные из текстовых файлов и рассчитывает коэффициенты регрессии по формулам (11), (12) и (22) для каждой цели. Полученные значения хранятся в полях gwp, gwb, gws;
- метод ConstructStrategy путем перебора определяет стратегию с минимальной стоимостью согласно пункту 3 алгоритма, описанного в подразделе 2.4. В числе параметров передается и массив приоритетов целей, полученных симуляции. Используемая в нем функция Goal::GetTardiness возвращает значение функции задержки по приоритету и позиции цели в стратегии, согласно ограничениям (5) и (6);
- метод GetPlanCostEstimate возвращает значение меры издержек плана по формуле (7);
- метод GetResultStateEstimate возвращает вектор результирующего состояния плана по формуле (13).

Для матричных операций использовалась шаблонная библиотека Eigen [34].

2.10 Реализация контроллера агента

Контроллер NPC был спроектирован согласно схеме, изображенной на рисунке 6. Вся логика алгоритма сосредоточена в методе UpdateAi класса UGoapController. Соответствующий код приведен в приложении 5.

Данный класс является наследником стандартного класса Unreal Engine ActorComponent, который прикрепляется к объекту Pawn, соответствующему управляемому NPC.

В каждом кадре в метод UpdateAi передаются значения wasActionComplete и mustBuildStrategy, уведомляющие контроллер соответственно о том, было ли завершено текущее действие и существует ли необходимость перестроить текущую стратегию. Проведя цикл обновления, построив при этом очередной план или стратегию, планировщик передает Pawn имя текущего действия и его эффекты. Имя может использоваться для выбора текущего поведения агента, как это показано на рисунке 9. Эффекты требуются для сложных действий, таких как GoTo, для указания точки, куда агенту надлежит переместиться.

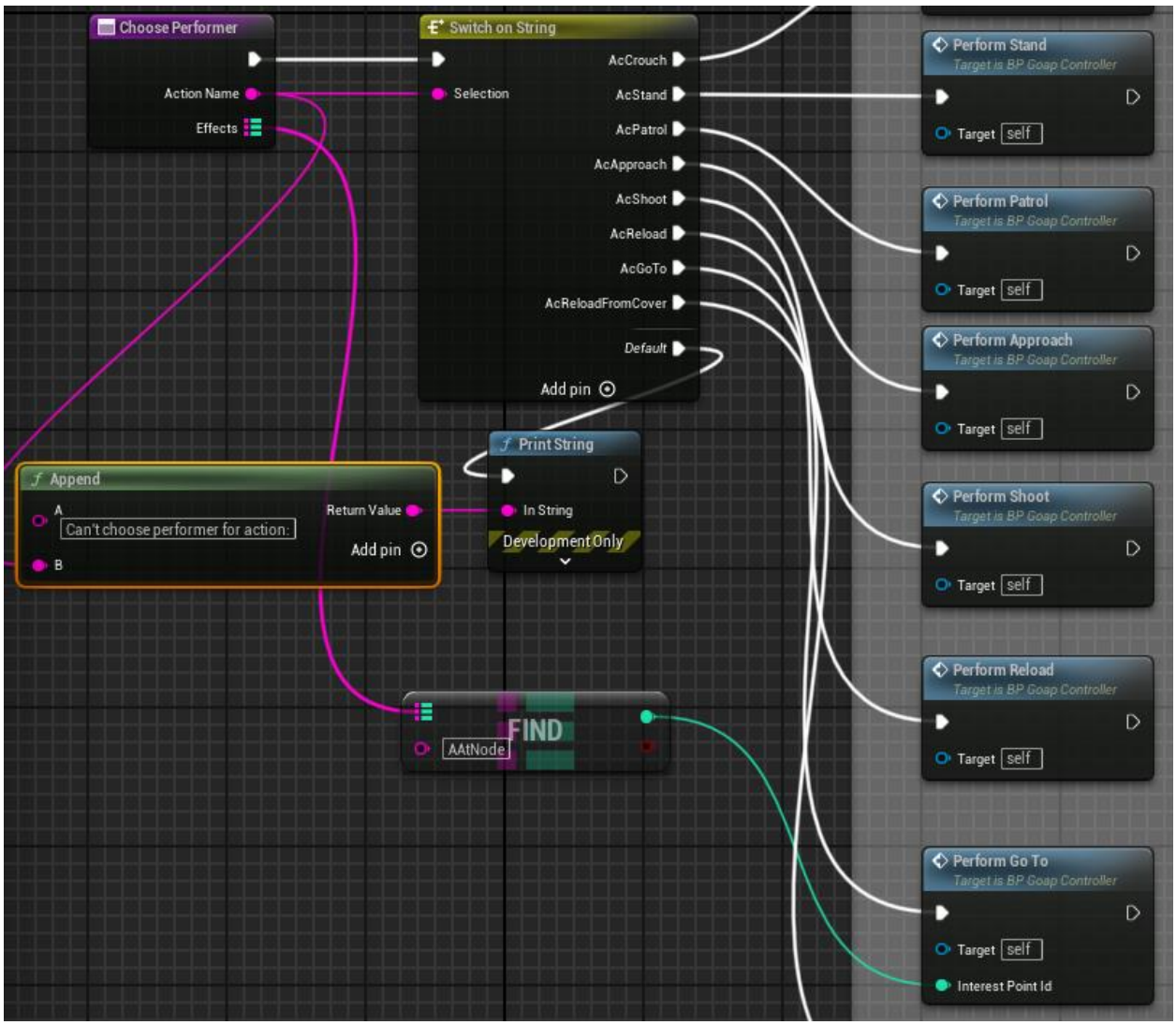

Рисунок 9 – Пример использования результатов планирования для определения текущего поведения агента

Выводы по разделу 2

В разделе описывается модификация алгоритма целеориентированного планирования, призванная устранить недостатки классического алгоритма, обнаруженные при его анализе в предыдущем разделе.

Сначала приводятся теоретические выкладки модификации, вводятся новые термины и понятия, такие как «стратегия», формулируется задача поиска оптимальной стратегии как задача минимизации линейной функции. Кроме того, получены формулы, помощью которых средствами машинного обучения получают значения оптимизируемой функции.

Далее кратко описываются детали реализации модификации в виде программы, рассматриваются ее важнейшие элементы.

3 Тестирование и анализ результатов

3.1 Моделирование испытательной среды в Unreal Engine 5

Для тестирования и настройки алгоритма был разработан проект в движке Unreal Engine 5. Проект реализован в виде прототипа игры в жанре шутера от третьего лица. Он включает в себя игровой уровень, контроллер NPC, а также управляемого игроком персонажа. Прототип позволяет воссоздать сценарии, демонстрирующие работоспособность алгоритма, отладить его, в случае неполадок, и – самое важное – собрать требуемые статистические данные. Прототип предполагает следующий геймплей:

1. Игрок и NPC противостоят друг другу, нанося повреждения попаданиями из автомата. При достижении здоровья игрока или NPC нуля игра завершается победой участника с запасом здоровья выше 0.
2. В случае низкого запаса патронов и игрок, и NPC могут пополнить его из ящика с патронами.
3. В случае низкого запаса здоровья игрок и NPC могут пополнить его у стационарной станции лечения.
4. Игроку и NPC доступны укрытия. Если игрок прячется в укрытии, NPC может потерять его из виду.

Укрытия, станции лечения и ящики с патронами были разработаны в виде т.н. «точек интереса» по аналогии со SmartObjects [15]. На рисунке 10 представлена схема тестового уровня.

Для реализации контроллера NPC на базе GOAP была разработана модель, включающая 3 цели, 9 действий и 6 атрибутов, которые представлены в таблицах 1-3.

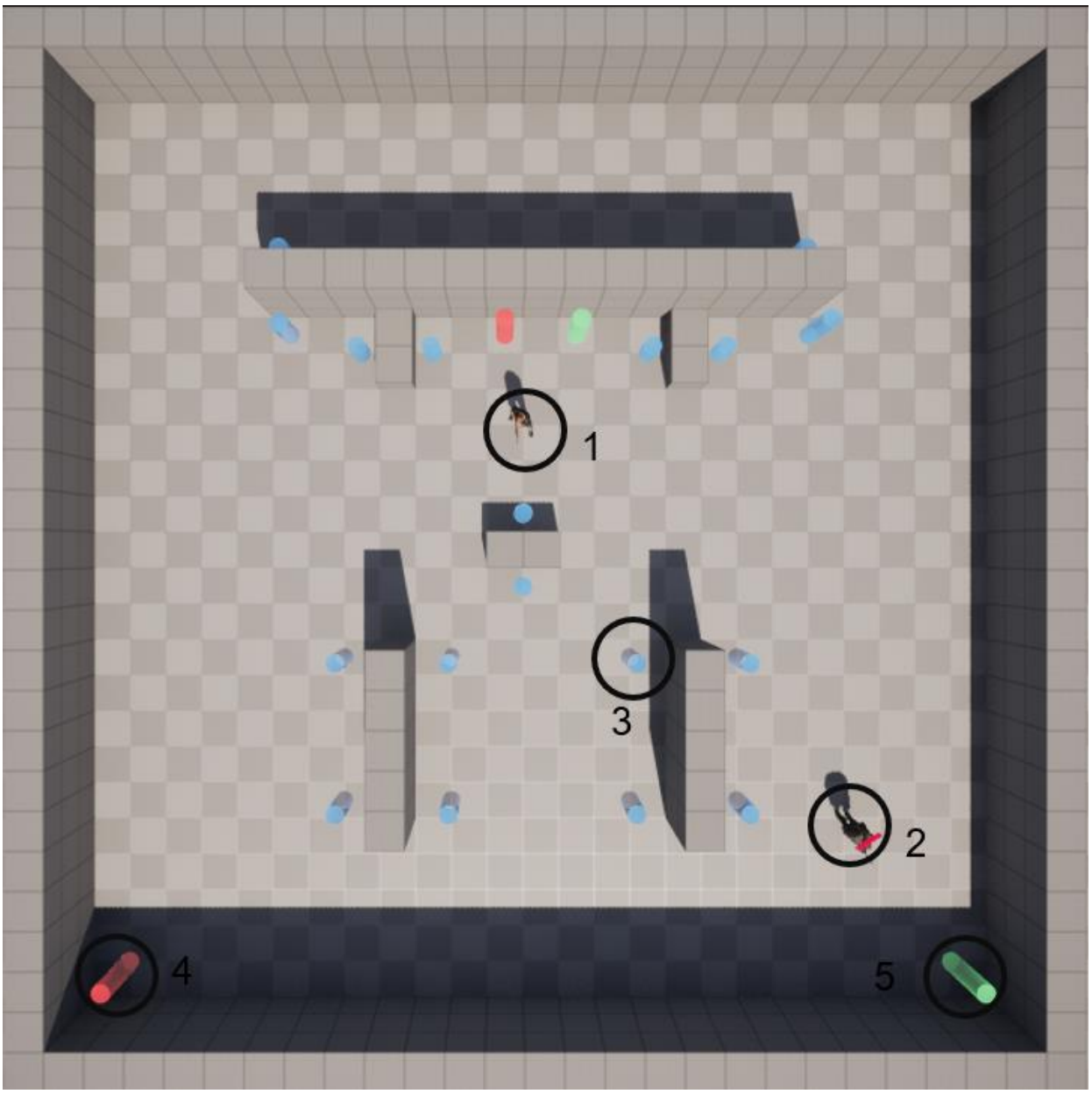

Рисунок 10 – Схема уровня прототипа. Цифрами обозначены: 1 – персонаж игрока, 2 – NPC, 3 – укрытие, 4 – ящик с патронами, 5 – станция лечения

В результате отладки симуляции, значения стоимостей действий и приоритетов целей были подобраны так, чтобы максимизировать продолжительность «жизни» агента и правдоподобность его поведения.

Таблица 1 – Цели и их приоритеты

№ п\п	Имя цели	Ожидаемое событие	Приоритет
1	GAttackEnemy	Агент атакует игрока	Постоянное значение
2	GHeal	Агент пополняет здоровье	Обратно пропорционален оставшемуся количеству очков здоровья
3	GPickupAmmo	Агент пополняет запас патронов	Обратно пропорционален количеству оставшихся магазинов для автомата

Таблица 2 – Атрибуты, передаваемая ими информация и возможные значения

№ п\п	Имя атрибута	Передаваемая информация	Возможные значения
1	AIsCrouching	Находится ли агент в приседе	истина/ ложь
2	AEnemyStatus	Статус противника по отношению к данному агенту	в зоне видимости/ не в зоне видимости/ в зоне атаки/ в бою
3	AAmmoInMag	Количество оставшихся патронов в магазине	0-30
4	AMagsLeft	Количество имеющихся полных магазинов	0-3
5	AAtNode	Идентификатор точки интереса, где сейчас находится агент, 0, если не находится в точке интереса	0-22
6	AHpLeft	Количество оставшихся процентов здоровья	0-100

Для вычисления эвристики требовалось определить функцию расстояния $d_i(a,b), i \in (\overline{1,n})$ для каждого атрибута. Выражение для расстояния представлено в формуле (23):

$$d_i(a,b) = \begin{cases} 1_{a=b}, i = 1,2 \\ |a-b|, i = 3,4,6 \\ \text{dist}(a,b), i = 5 \end{cases}, \quad (23)$$

где $1_{a=b} = \begin{cases} 1, a = b \\ 0, a \neq b \end{cases}$, $\mathrm{dist}(a, b)$ – длина пути между точками с идентификаторами a и b.

Таблица 3 – Действия и их стоимости

№ п\п	Имя действия	Действие	Стоимость
1	AcCrouch	Присесть	Постоянное значение
2	AcStand	Встать	Постоянное значение
3	AcSearch	Искать противника	Прямо пропорциональна времени, прошедшему с последнего контакта с игроком
4	AcApproach	Подойти к противнику	Прямо пропорциональна длине пути до игрока
5	AcShoot	Выстрелить в противника	Постоянное значение; штраф, если агент не в укрытии
6	AcReload	Перезарядить оружие	Постоянное значение; штраф, если агент не в укрытии
7	AcGoTo	Идти в точку интереса	Прямо пропорциональна длине пути до точки интереса
8	AcHeal	Пополнить запас здоровья	Постоянное значение; штраф, если точка лечения в зоне видимости игрока
9	AcTakeMagazines	Пополнить запас магазинов для оружия	Постоянное значение; штраф, если ящик с патронами в зоне видимости игрока

Также в целях отладки был разработан набор виджетов, составляющих пользовательский интерфейс, позволяющий в реальном времени отслеживать текущую стратегию, план и состояние агента. Пример визуализации контроллера NPC представлен на рисунке 11.

Рисунок 11 – Отладочный интерфейс контроллера NPC

3.2 Сбор статистических данных

Как сообщалось в предыдущей главе, для возможности предсказания стоимостей планов при выполнении очередной цели G из состояния S^0 или смерти NPC в кортеж сохранялись следующие данные:

- стоимость плана $p(S, G)$;
- индикатор $b(S, G) \in (0,1)$ того, была ли на самом деле достигнута цель в результате следования построенному плану;
- фактическое состояние $S^*(S, G)$ на момент достижения цели.

Цель считалась невыполненной (т.е. $b(S, G) = 0$), если в процессе ее выполнения агент был уничтожен. Если цель не была выполнена, то $S^*(S, G)$ не фиксировался. Таким образом, не каждому сохраненному начальному состоянию S соответствовало конечное результирующее состояние $S^*(S, G)$. Количество сохраненных S будем обозначать m, количество сохраненных S^* будем обозначать m^*.

Сбор одного блока данных будем называть пробой. При этом, два блока, у которых совпадали цель G и состояние S, усреднялись. Значения атрибутов S^* усреднялись попарно. Для этого для каждого атрибута была выбрана функция усреднения. Для атрибутов 3, 4, 6 в качестве усредненного значения

было выбрано округленное среднее арифметическое; для атрибутов 1, 2 – мода. В качестве среднего среди ряда точек интереса (атрибут 5) выбиралась наиболее близкая к центру масс [35] ряда точка, причем в качестве массы точки принимается число ее появлений в ряду.

$$\tilde{a}_i = \begin{cases} \frac{1}{m^*}\sum_{j=1}^{m} a_{ij}, i \in \{3,4,6\} & (24) \\ \arg\max_{k\in(1,m)} \nu_{ik}, i \in \{1,2\} & (25) \\ \arg\min_{k\in(1,m^*)} \mathrm{dist}\left(\boldsymbol{r_k}, \left(\frac{\sum_{j=1}^{m^*} \nu_{ik}\boldsymbol{r}_j}{\sum_{j=1}^{m^*} \nu_{ik}}\right)\right), i = 5 & (26) \end{cases}$$

В формулах (24), (25) ν_{ik} – количество повторений значения a_{ik} среди i-тых элементов векторов $\boldsymbol{s_1^*} \dots \boldsymbol{s_{m^*}^*}$:

$$\nu_{ik} = \sum_{j=1}^{m^*} 1_{a_{ik}=a_{ij}} \quad (27)$$

$1_{a_{ik}=a_{ij}}$ – индикаторная величина [36]:

$$1_{a_{ik}=a_{ij}} = \begin{cases} 1, a_{ik} = a_{ij} \\ 0, a_{ik} \neq a_{ij} \end{cases}, \quad (28)$$

В формуле (26) r_j – радиус-вектор точки интереса с идентификатором a_j; $\mathrm{dist}(\boldsymbol{r_k}, \boldsymbol{r_j})$ – длина пути между точками интереса с идентификаторами a_k и a_j соответственно.

Таким образом получались усредненные величины $\tilde{p}(G, S^0), \tilde{b}(G, S^0)$ и $\tilde{S}^*(G, S^0)$.

С целями сбора статистических данных было проведено 50 проб для каждой цели (итого 150 проб). Количества различных начальных состояний для каждой цели приведены в таблице 4.

Для нахождения коэффициентов регрессии из собранных значений строились матрицы. Так как $m \geq m^*$, матрица информации (регрессор) $\boldsymbol{S_{n\times m}}$ строилась из всех m собранных векторов-состояний $\boldsymbol{s}$:

$$\boldsymbol{S_{n\times m}} = \begin{pmatrix} | & & | \\ \boldsymbol{s_1} & \cdots & \boldsymbol{s_m} \\ | & & | \end{pmatrix}$$

Оставив из векторов $\boldsymbol{s}$ только те состояния, планы из которых привели к выполнению цели, составим из них матрицу информации (регрессор):

$$\boldsymbol{S_{n\times m^*}} = \begin{pmatrix} | & & | \\ \boldsymbol{s_{i_1}} & \cdots & \boldsymbol{s_{i_{m^*}}} \\ | & & | \end{pmatrix}$$

При этом в силу отсутствия идеально скоррелированных атрибутов, псевдообратные матрицы обеих матриц существует.

Аналогично были получены векторы и матрица регрессантов $\boldsymbol{p}$, $\boldsymbol{b}$ и $\boldsymbol{S}^*$.

Таблица 4 – Количества различных начальных состояний для целей

№ цели п/п	Кол-во различных начальных состояний
1	28
2	10
3	12

Далее, пользуясь решениями (11), (12) и (22) уравнений линейной регрессии, были получены векторы $\boldsymbol{w_G}^{\boldsymbol{p}}$, $\boldsymbol{w_G}^{\boldsymbol{b}}$ и матрица $\boldsymbol{W_G^{S^*}}$ коэффициентов линейной регрессии.

При вызове ConstructStrategy с помощью полученных коэффициентов по формулам (9), (10), (13) получались ожидаемые значения $p(S,G), b(S,G)$ и $\boldsymbol{s}^*(S,G)$. По этим данным строились оптимальные стратегии.

3.3 Сравнение поведений на модельных примерах

Рассматривалось два модельных примера.

Пример 1. Расположение агента, игрока и точек интереса показано на рисунке 12. Известно, что агент имеет 30 процентов здоровья, 15 патронов, 0 полных магазинов, а также он видел противника в точке C несколько секунд назад. Вычислив приоритеты каждой из целей, представленных в таблице 1, получим стратегию, представленную в таблице 5.

Пути агента, соответствующие каждой из стратегий показаны на рисунке 12. Мы видим, что путь, полученный в результате

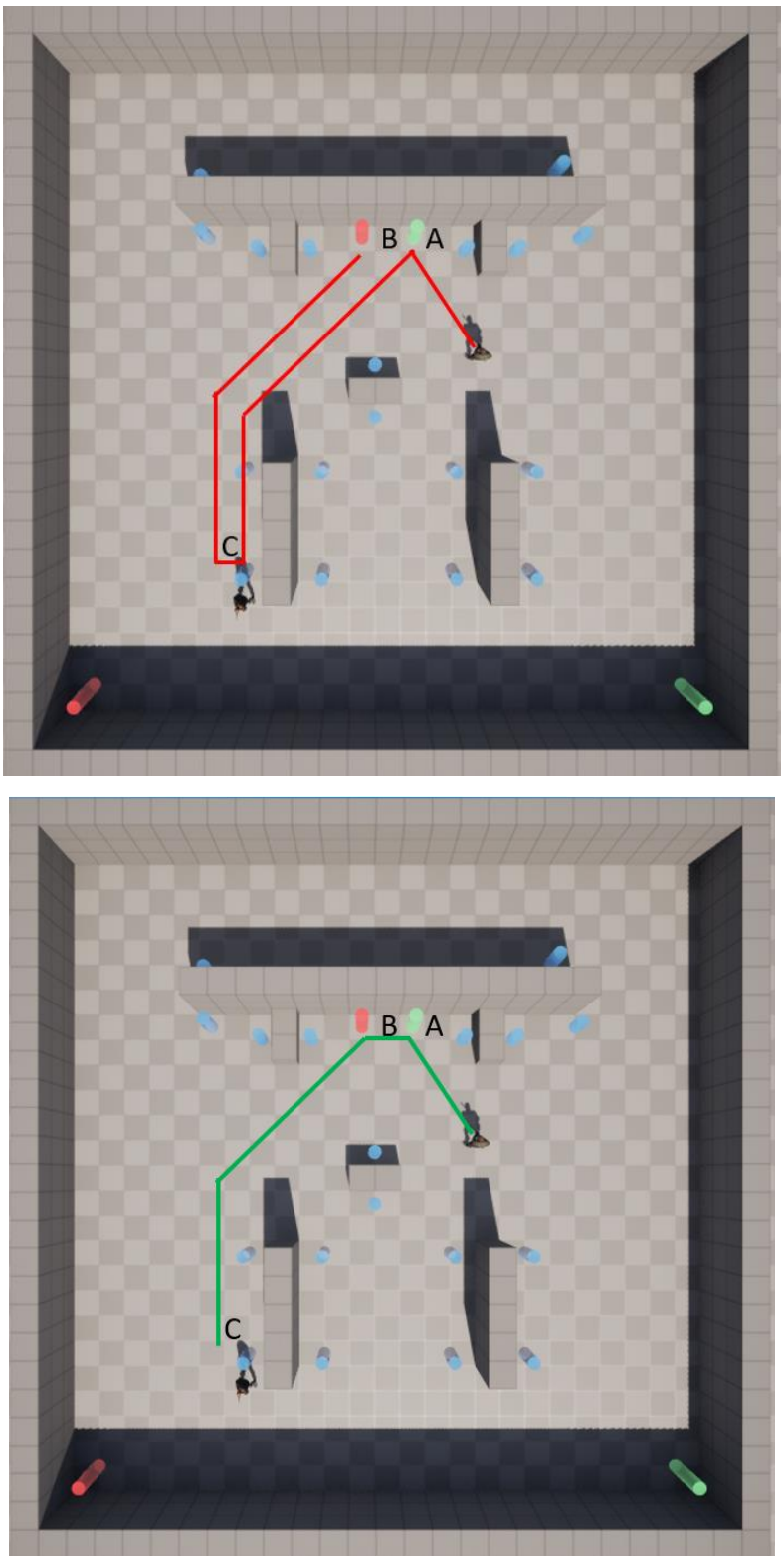

Рисунок 12 – Сравнение поведения агента в модельном примере 1: классический алгоритм (сверху, путь агента показан красными отрезками) и модифицированный алгоритм (снизу, путь агента показан зелеными отрезками). Буквами обозначены точки целей

модифицированного алгоритма короче. Следовательно, сама стратегия более реалистична и сходна той, что ожидалось, когда данный пример рассматривался в первой части работы.

Таблица 5 – Стратегия, полученная классическим алгоритмом в модельном примере 1

Цель	Порядок в стратегии (классический алгоритм)	Порядок в стратегии (модифицированный алгоритм)
Пополнить здоровье (A)	1	1
Атаковать врага (B)	2	3
Пополнить патроны (C)	3	2

Пример 2. Известно, что агент имеет 100 здоровья, 10 патронов, 0 полных магазинов, а также сейчас происходит бой с противником. Расположение агента и игрока показаны на рисунке 13.

Для данного примера получаются стратегии, приведенные в таблице 6.

Как видно, модифицированный алгоритм поместил цель «Пополнить патроны» перед целью «Атаковать врага», в отличие от классического алгоритма. Дело в том, что в среднем атаковать врага с небольшим запасом патронов чаще приводит к гибели агента. Алгоритм сравнил стоимости опрометчивой атаки и преждевременного похода к ящику с патронами и оказалось, что выгоднее заранее пополнить патроны. На практике это продлило жизнь NPC.

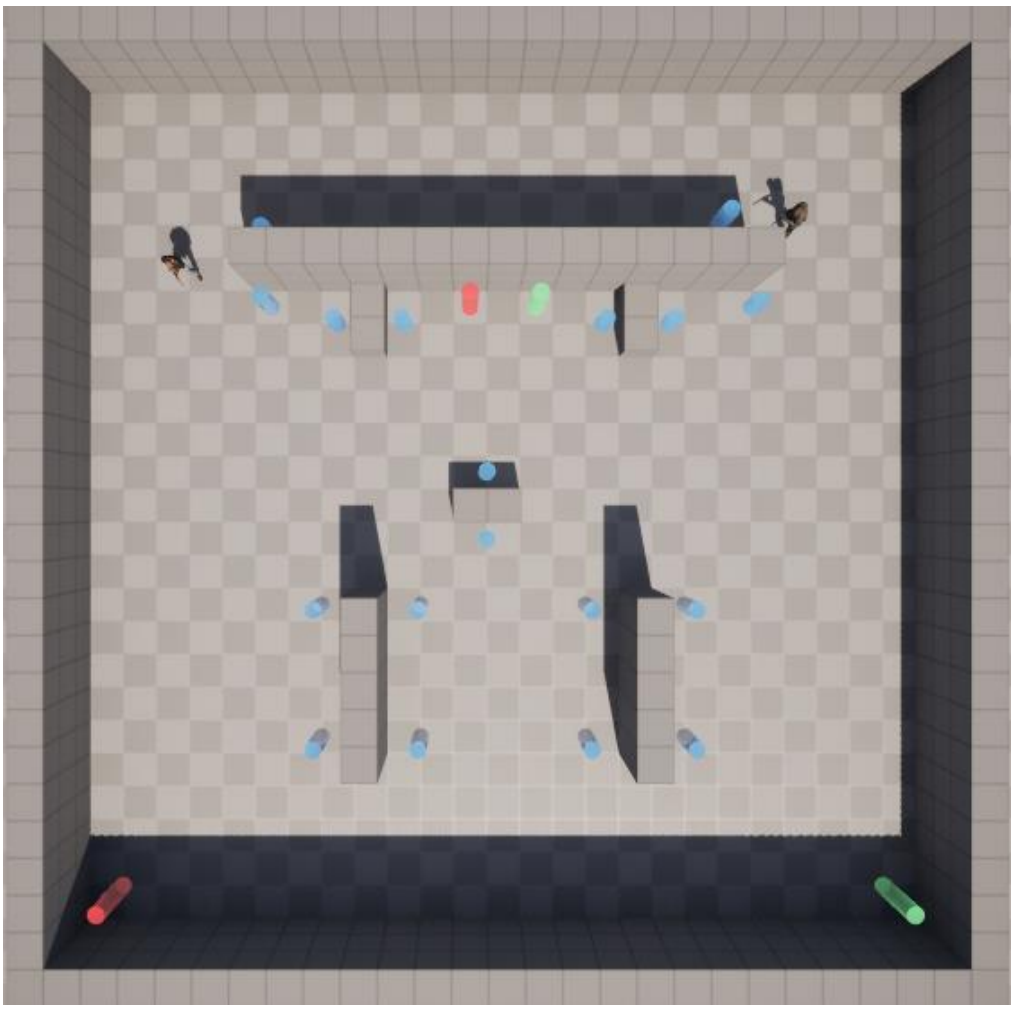

Рисунок 13 – Расположение агента и игрока в модельном примере 2

Таблица 6 – Стратегия, полученная классическим алгоритмом в модельном примере 2.

Цель	Порядок в стратегии (классический алгоритм)	Порядок в стратегии (модифицированный алгоритм)
Атаковать врага	1	2
Пополнить патроны	2	1
Пополнить здоровье	3	3

3.4 Определение радиуса устойчивости

Учитывая, что оценка стоимости плана, соединяющего цели, зависит не только от самих целей, но и от предшествующих в стратегии планов (конкретных целевых состояний), задача минимизации стоимости стратегии является разновидностью задачи коммивояжера [37], [38] с переменной длиной дуг (Time-Dependent Travelling Salesman Problem) [39].

Такой тип задач элементарно приводится к обобщенной задаче коммивояжера [40]. Имея граф с вершинами, помеченными индексами 1-3, соответствующим целям, а также стартовую вершину с индексом 0, соответствующую начальному состоянию, трансформируем его в четырехдольный граф, введя второй индекс, означающий порядок вершины в пути обхода. Будем называть i-той долью независимое множество вершин, соответствующих i-той цели. В нулевой доле находится лишь стартовая вершина 0,0. Имея три цели и заданное начальное состояние, получим обобщенную задачу коммивояжера с $1 + 3^2 = 10$ вершинами. Полученный четырехдольный граф показан на рисунке 14.

Суть задачи состоит в поиске кратчайшего пути в графе, проходящего через каждую долю строго однократно.

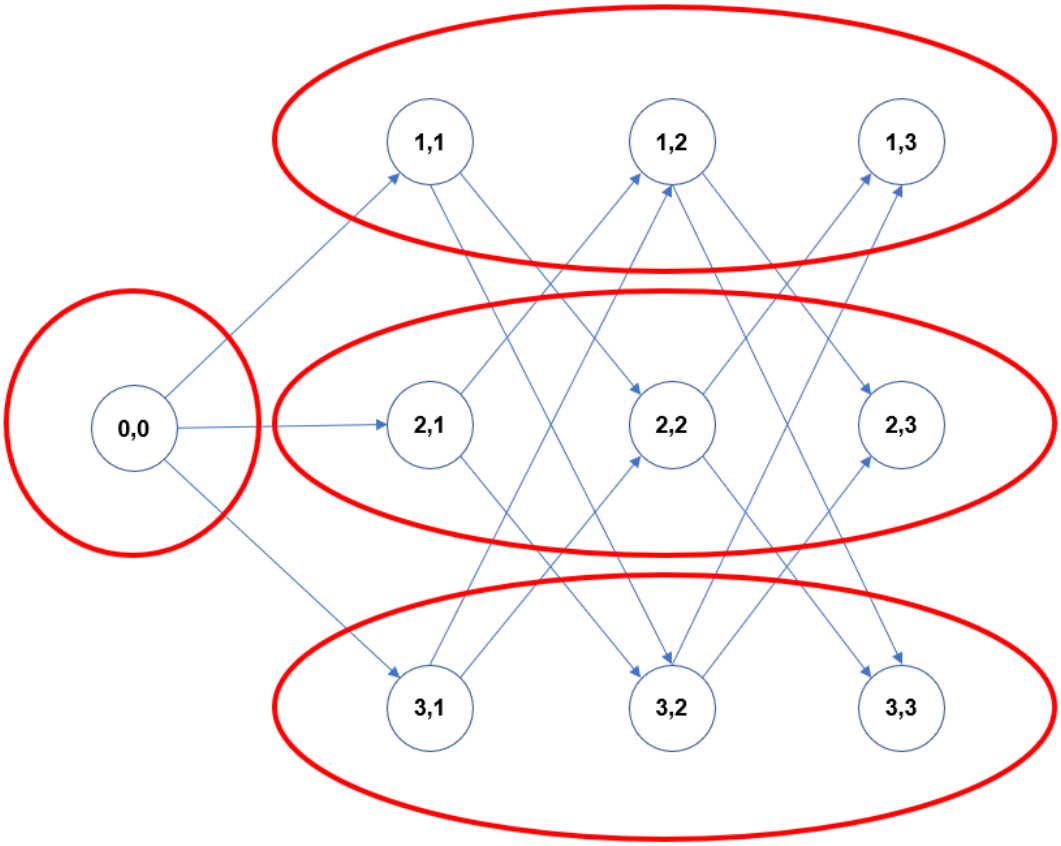

Рисунок 14 – Полученный с целью анализа устойчивости четырехдольный граф. Доли показаны красным. Первый индекс вершины соответствует индексу цели, второй – порядку достижения

Согласно [41] такая задача может быть приведена к классической задаче коммивояжера путем кластеризации графа, причем без повышения порядка задачи. Отсюда можно сделать вывод, что аппарат, применяемый в работе [42] для анализа устойчивости решения задачи коммивояжера, будет пригоден и для соответствующего анализа оптимальной стратегии.

Рассмотрим незамкнутую задачу коммивояжера с фиксированной стартовой вершиной на графе с матрицей расстояний $\boldsymbol{C}_{q\times q}$. Обозначим множество решений такой задачи как $\Phi(\boldsymbol{C})$.

Рассмотрим R_n множество всех квадратных матриц порядка q. Очевидно, что $\boldsymbol{C} \in R_n$. Шаром устойчивости $Q_\rho(\boldsymbol{C})$ матрицы $\boldsymbol{C}$ называется такой шар радиуса ρ с центром в матрице $\boldsymbol{C}$, что $\forall A \in Q_\rho(C)\ \Phi(A) \subseteq \Phi(C)$.

Наконец, радиусом устойчивости ρ_0 матрицы C называется радиус большего шара устойчивости, т.е.

$$\rho_0(\boldsymbol{C}) = \arg\sup_{\rho} Q_\rho(\boldsymbol{C}). \quad (29)$$

Посылка, лежащая в основе проведенного анализа, заключается в том, что алгоритм, поставляющий матрицы с большим радиусом устойчивости, дает более устойчивые решения, то есть менее чувствителен к колебаниям приоритетов целей.

Для вычисления радиуса устойчивости воспользуемся формулой (7) из [42]:

$$\rho_0(\boldsymbol{C}) = \max_{i \neq j} \frac{c(\sigma_j) - c(\sigma_i)}{2\left(q - \left|\sigma_j \cap \sigma_i\right|\right)}, \quad (30)$$

где $\sigma_j = \hat{\sigma}$ – оптимальная стратегия, а $\left|\sigma_j \cap \sigma_i\right|$ – количество общих планов у стратегий σ_j и σ_i.

Итак, для сравнения устойчивости классического и модифицированного алгоритмов, необходимо по транспортным матрицам $\boldsymbol{C'}$ и $\boldsymbol{C''}$ соответственно вычислить радиусы устойчивости. Так как элементы этих матриц могут сильно отличаться по значениям, более пригодным для сравнения является некоторое «нормированное» значение радиуса. Так как радиус устойчивости имеет смысл определенного значения метрики пространства R_n, разделив радиус (30) на норму матрицы $\boldsymbol{C}$ получим нормированное значение радиуса устойчивости:

$$\widetilde{\rho_0}(\boldsymbol{C}) = \frac{\rho_0(\boldsymbol{C})}{\left|\left|\boldsymbol{C}\right|\right|}. \quad (31)$$

Для сравнения устойчивости было проведено 10 экспериментов, в ходе которых определялись значения нормированных радиусов устойчивости по формуле (31) для транспортных матриц $\boldsymbol{C'}$ и $\boldsymbol{C''}$ классического и модифицированного алгоритма соответственно. Средние значения нормированных радиусов представлены в таблице 7.

Как видно, применяя модифицированный алгоритм, можно получать в среднем в три раза более устойчивые решения.

Таблица 7 – Значения усредненного нормированного радиуса устойчивости для классического и модифицированного алгоритмов.

	Усредненный нормированный радиус устойчивости
Классический алгоритм	0.114
Модифицированный алгоритм	0.438

3.5 Экспериментальное измерение устойчивости

Для экспериментальной оценки устойчивости, приоритет цели представлялся в виде

$$\pi(G) = (1 + \delta_G)\pi_0(G), \quad (32)$$

где $\pi_0(G)$ – эвристика приоритета цели, а параметр δ_G принимал случайные значения из нормального распределения в интервале $[0; \delta_{max}]$. δ_{max} варьировался от 0 до 1 с шагом 0.1.

На каждое значение δ_{max} проводилось 10 экспериментов. Фиксировалось количество раз ν, когда в результате варьирования параметра стратегия изменилась. График зависимости $\nu(\delta_{max})$ представлен на рисунке 15.

График на рисунке 15 показывает, что использование модифицированного алгоритма приводит к более редким изменениям стратегии. Вероятно, это связано с тем, что стратегия, построенная на базе опыта прошлых итераций, учитывает возможные изменения приоритетов и адаптируется к ним, что позволяет варьировать приоритеты в определенных границах без необходимости перестроения стратегии. Кроме того, это коррелирует с динамикой радиуса устойчивости, исследованной в предыдущей главе.

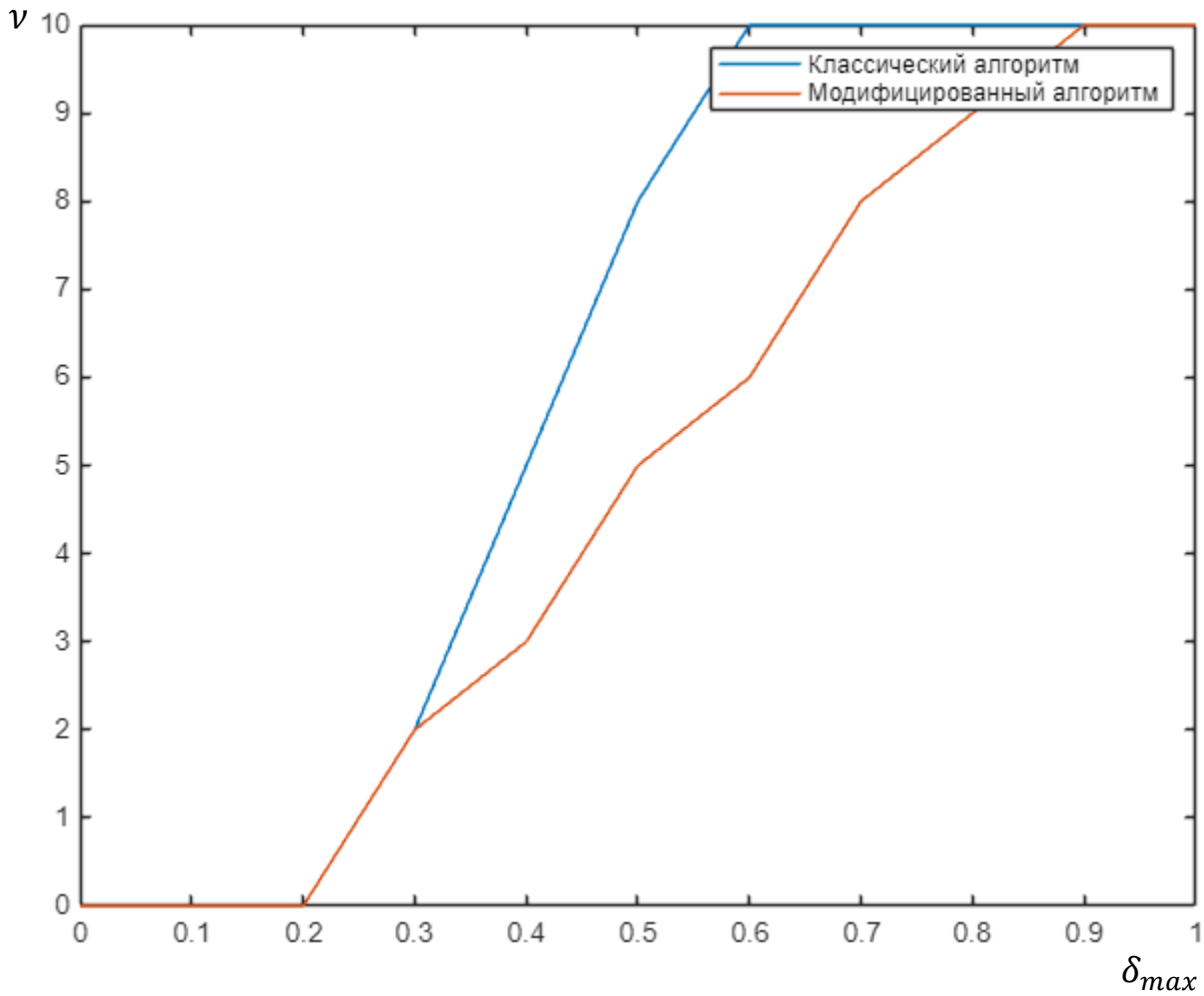

Рисунок 15 – Сравнение зависимостей частоты смены стратегии от относительной амплитуды колебаний приоритетов целей.

3.6 Будущие перспективы

Для улучшения получаемых с помощью модифицированного алгоритма решений будут предприняты следующие шаги:

– Планируется провести тестирование в условиях с большим количеством неопределенности, что включает в себя увеличение числа целей, действий и атрибутов. Это позволит оценить работоспособность и эффективность алгоритма в более сложных и разнообразных сценариях, а также выявить его возможные ограничения и улучшить его адаптацию к различным условиям

– Планируется провести исследование по оценке влияния выбора алгоритмов усреднения и вычисления расстояния между значениями

атрибутов на устойчивость, получаемых с помощью модификации стратегий. Это позволит определить, насколько изменения в этих параметрах влияют на работу алгоритма, его точность и предсказательную способность. Результаты исследования помогут оптимизировать выбор и настройку эвристик для достижения лучших результатов.

– Планируется исследовать возможности применения более точных алгоритмов машинного обучения для предсказания результатов планирования на основе собранной статистики. Это позволит повысить точность прогнозирования и оптимизировать процесс планирования, учитывая более сложные зависимости и взаимодействия между различными атрибутами.

Выводы по разделу 3

В разделе приведено описание испытательной среды, построенной для тестирования разработанной модификации, а также непосредственно сам процесс и результаты тестирования.

Модифицированный алгоритм был протестирован на двух модельных примерах. Результаты показали, что модификация алгоритма приводит к наилучшему поведению по сравнению с немодифицированным в обоих случаях. Это свидетельствует о его эффективности и точности в различных сценариях.

Дополнительно к тестированию на модельных примерах проводился сравнительный анализ устойчивости алгоритма. Результаты показали, что модифицированный алгоритм увеличивает устойчивость решений в среднем в три раза. Экспериментально измерялась частота изменения стратегии от относительной амплитуды колебаний приоритета целей. Результаты показали, что использование модификации позволило сократить количество перестроений стратегии.

ЗАКЛЮЧЕНИЕ

В ходе выполнения ВКР было проведено тщательное теоретическое исследование, в рамках которого изучалась концепция целеориентированного планирования. В результате этого был выявлен ряд недостатков, которые оказывают влияние на реалистичность поведения агентов, управляемых алгоритмом GOAP. К этим недостаткам относятся игнорирование агентами накопленного опыта и чрезмерная чувствительность к эвристике приоритетов целей.

Для преодоления указанных недостатков было предложено модифицировать процесс определения порядка выполнения целей, используя линейную регрессию для предсказания стоимости и реализуемости различных планов. На основе этих предсказаний цели были упорядочены таким образом, чтобы минимизировать прогнозируемую общую стоимость планов.

Далее, на этапе реализации модификации, была разработана программа, состоящая из нескольких блоков. Один из этих блоков отвечает за построение планов с использованием классического алгоритма GOAP, в то время как другой блок отвечает за построение стратегии в соответствии с разработанным модифицированным алгоритмом.

Особенностью созданной программы является модульность, позволяющая пользователям переопределять свои классы, тем самым настраивая работу приложения согласно нуждам своего проекта.

Приложение было успешно интегрировано в проект, созданный на игровом движке Unreal Engine 5. В рамках этого проекта была разработана симуляция, а также отладочный интерфейс для тестирования раработанной программы.

В ходе проведения испытаний была установлена эффективность разработанной модификации алгоритма целеориентированного планирования, что подтвердило успешность нашего подхода к улучшению работы системы GOA

Для этого модифицированный алгоритм был протестирован на двух модельных примерах. Результаты тестирования позволили сравнить поведение модифицированного алгоритма с поведением немодифицированного алгоритма, а также с ожидаемыми результатами в данных модельных условиях. Сравнительный анализ показал, что модификация алгоритма приводит к получению наиболее реалистичного и оптимального поведения в обоих модельных примерах. Это подтверждает эффективность модифицированного подхода и его способность к более точному предсказанию и управлению в различных сценариях.

Кроме того, дополнительно к тестированию на модельных примерах проводился сравнительный анализ устойчивости алгоритма. В ходе серии экспериментов фиксировались значения нормированных радиусов устойчивости для транспортных матриц классического и модифицированного алгоритмов. Сравнение средних значений показало, что при использовании модифицированного алгоритма устойчивость решений в среднем увеличивается в три раза.

Наконец, экспериментально измерялась частота изменения стратегии от относительной амплитуды колебаний приоритета целей. Полученная зависимость показывает, что при использовании модифицированного алгоритма частота изменения стратегии возрастает медленнее. Вероятно, это связано с тем, что стратегия, основанная на опыте предыдущих итераций, способна адаптироваться к изменениям приоритетов без необходимости полного перестроения.

В ходе выполнения ВКР был разработан модуль для интеграции в игровые проекты на игровом движке Unreal Engine. Этот модуль представляет собой важный ресурс для разработчиков, который расширяет их возможности при создании игровых миров. Также были определены направления для будущих исследований, направленных на улучшение модификации и ее удобства использования. Это включает в себя не только технические детали,

но и работу над интерфейсом и поддержкой, чтобы обеспечить более эффективное взаимодействие для разработчиков игр.

Мои исследовательские результаты, включая текст ВКР, исходные коды, а также примеры с инструкциями к запуску, были опубликованы в репозитории на платформе GitHub. Вы можете получить доступ к ним, перейдя по следующей ссылке: https://github.com/SigmaKlim/GOAP_UE.git .

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Monolith Productions, F.E.A.R., Sierra Entertainment, 2005.

2 GSC Game World, S.T.A.L.K.E.R.: Shadow of Chernobyl, THQ, 2007.

3 Monolith Productions, MiddleEarth: Shadow of Mordor.

4 Ubisoft Montreal, Tom Clancy’s Splinter Cell: Chaos Theory, Ubisoft, 2005.

5 M. Dawe и S. Gargolinski, Behavior Selection Algorithms: An Overview, в Game AI Pro: Collected Wisdom of Game AI Professionals, Boca Raton, FL, USA, CRC Press, 2014, 47-60.

6 J. Smed и H. Hakonen, Finite State Machines, в Algorithms and Networking for Computer Games, Chichester, John Wiley & Sons Ltd, 2006, 122-134.

7 I. Millington, State Machines, в AI for Games, Boca Raton, Florida, CRC Press, 2019, 314-337.

8 D. Jagdale, Finite State Machine in Game Development, International Journal of Advanced Research in Science, Communication and Technology, т. 10, № 1, 384-390, 2021.

9 I. Millington, Behavior Trees, в Ai for Games, Boca Raton, FL , CRC Press, 2019, 338-373.

10 D. Graham, An Introduction to Utility Theory, в Game AI Pro: Collected Wisdom of Game AI Professionals, Boca Raton, FL, CRC Press, 2014, 113-126.

11 M. Lewis, Choosing Effective Utility-Based Considerations, в Game AI Pro 3: Collected Wisdom of Game AI Professionals, Boca Raton, FL, CRC Press, 2017, 242-257.

12 A. Kedalo, H. Aslam, A. Zykov и M. Mazzara, Comparing Behaviour Tree and Hierarchical Task Network Planning Methods for their Impact on Player Experience, в IEEE Symposium Series on Computational Intelligence, Mexico City, 2023.

13 I. Millington, Goal-Oriented Behavior, в AI for Games, CRC Press, 2019.

14 M. Buckland, Goal-Driven Agent Behavior, в Programming Game AI by Example, Plano, Texas, Wordware Publishing, 2005, 379-414.

15 J. Orkin, Three States and a Plan: The A.I. of F.E.A.R., в Материалы GDC, 2006.
16 I. Millington, Reinforcement Learning, в AI for Games, Boca Raton, FL, CRC Press, 2019, 627-641.
17 Y. Fu, L. Qin и Q. Yin, A Reinforcement Learning Behavior Tree Framework for Game AI, в 2nd International Conference on Economics, Social Science, Arts, Education and Management Engineering, Hunan, 2016.
18 D. Dwibedi и A. Vemula, Playing Games with Deep Reinforcement Learning, в 31st AAAI Conference on Artificial Intelligence, San Francisco, 2016.
19 T. Simonini, An Introduction to Unity ML Agents, Towards Data Science, URL: https://towardsdatascience.com/an-introduction-to-unity-ml-agents-6238452fcf4c. (дата обращения: 2023-05-17).
20 J. Orkin, Applying Goal-Oriented Action Planning to Games, AI Game Programming Wisdom, 2002.
21 В. Е. Алексеев и В. А. Таланов, Графы. Модели вычислений. Структуры данных, Нижний Новгород: Издательство Нижегородского государственного университета, 2001.
22 I. Abraham и A. P. Goldberg, A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks, в Experimental Algorithms - 10th International Symposium, Kolimpari, 2011.
23 R. Gutman, Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks, в Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, 2004.
24 J. Smed и H. Hakonen, Path Finding, в Algorithms and Networking for Computer Games, Chichester, John Wiley & Sons Ltd, 2006, 108-109.
25 K. Dill, What Is Game AI?, в Game AI Pro: Collected Wisdom of Game AI Professionals, Boca Raton, FL, CRC Press, 2014, 3-9.
26 Высшая Школа Цифровой Культуры, Университет ИТМО, Элементы статистики и Введение в МО (онлайн-курс), 2023. URL:

https://courses.openedu.ru/courses/course-v1:ITMOUniversity+ELSTATINTROML+spring_2023_ITMO_mag/pdfbook/0/. (дата обращения: 2024-03-12).
27 А. Хлевнюк, Основы линейной регрессии, 11 08 2020. URL: https://habr.com/ru/articles/514818/. (дата обращения: 13 04 2024].
28 J. Kun, Regression and Linear Combinations, 29 23 2021. URL: https://www.jeremykun.com/2021/03/29/regression-and-linear-combinations/. (дата обращения: 2024-03-12).
29 R. A. Horn и C. R. Johnson, Norms for Vectors and Matrices, в Matrix Analysis, Cambridge, Cambridge University Press, 2018, 340-370.
30 M. S. P. Kaare Brandt Petersen, The Matrix Cookbook, 15 11 2012. URL: http://matrixcookbook.com/. (дата обращения: 2024-03-14).
31 D. Barczyński, A* algorithm, URL: https://github.com/daancode/a-star.git. (дата обращения: 2023-02-12).
32 S. Petrov, A*, URL: https://github.com/k0dep/astar.git. (дата обращения: 2023-12-02).
33 D. H. Costa, Fibonacci Heap, URL: https://github.com/diogolhc/fibonacci-heap.git. (дата обращения: 2023-11-17).
34 Institut national de recherche en sciences et technologies du numérique, Eigen, TuxFamily, URL: https://eigen.tuxfamily.org/index.php?title=Main_Page. (дата обращения: 2024-04-12).
35 Журавлев В.Ф., Общие теоремы динамики, в Основы теоретической механики, Москва, Физматлит, 2001, p. 66.
36 Т. Кормен, Ч. Лейзерсон, Р. Ривест и К. Штайн, Индикаторная случайная величина, в Алгоритмы: построение и анализ, Москва, Санкт-Петербург, Киев, Вильямс, 2013, 144-145.
37 И.В.Мудров, Задача о коммивояжере, Москва: Знание, 1969.
38 H. Abeledo, A. Pessoa, R. Fukasawa и E. Uchoa, The Time Dependent Traveling Salesman Problem: Polyhedra and Branch Cut-and-Price Algorithm, в

Материалы конференции Mathematical Programming Computation, Naples, 2010.
39 J. J. Miranda-Bront, I. Méndez-Díaz и Z. P., An integer programming approach for the time-dependent TSP, Electron. Notes Discret. Math, № 36, 351-358, 2010.
40 J.-C. Picard и M. Queyranne, Time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling., Ecole Polytechnique de Montréal, Montreal, 1977.
41 C. Noon и B. James, An Efficient Transformation of the Generalized Travelling Salesman Problem, Information Systems and Operationl Research, № 31, 39-45, 1993.
42 В. К. Леонтьев, Устойчивость задачи коммивояжера, Журнал вычислительной математики и математической физики, т. 15, № 5, 1298-1309, 1975.

ПРИЛОЖЕНИЕ А

Метод Pathfind класса AStarSolver

```
bool AStarSolver::Pathfind (Path<t_vertex>& path, t_vertex start, t_vertex finish =
t_vertex()) const
{
	FibonacciHeap<Node> discQueue;
	std::unordered_map<unsigned, Element<Node>*> discovered;
	std::unordered_set<unsigned> expanded;
	std::unordered_map<t_vertex, unsigned,
	VertexKey<t_vertex>, VertexEqual<t_vertex>> vid;
	std::vector<const t_vertex*> idv;
	std::unordered_map <unsigned, unsigned> cameFrom;
	unsigned discoveredHeapSize = 0;
	float normalizedDistance = 0.0f;
	float normalizedHeuristics =
	GetHeuristic(start, finish) / GetHeuristicsDenominator();
	Node currentNode(0, normalizedDistance, normalizedHeuristics);
	idv.push_back(&start);
	cameFrom.insert({ 0, UINT_MAX });
	Element<Node>* currentElement = discQueue.insert(currentNode);
	discovered.insert({ 0, currentElement });
	std::vector <t_vertex> neighborVertices;
	std::vector<float> distances;
	while (true)
	{
		int discoveredHeapSizeDelta = 0;
		if (discQueue.isEmpty() == true)
		{
			path.Vertices.clear();
			return false;
		}
		currentNode = discQueue.extractMin();
		discoveredHeapSizeDelta--;
		unsigned currentVertexId = currentNode.VertexId;
```

```
discovered.erase(currentVertexId);
const t_vertex currentVertex = *idv[currentVertexId];
if (Satisfies(currentVertex, finish) == true)
        break;
expanded.insert(currentVertexId);
GetNeighbors(neighborVertices, distances, currentVertex, finish);
for (size_t i = 0; i < neighborVertices.size(); i++)
{
        normalizedDistance = distances[i] / GetDistanceDenominator();
        normalizedHeuristics = GetHeuristic(neighborVertices[i], finish) /
                GetHeuristicsDenominator();
        auto discOrExpIt = vid.find(neighborVertices[i]);
        if (discOrExpIt == vid.end())
        {
                unsigned neighborId = idv.size();
                idv.push_back(&vid.insert({neighborVertices[i],
                        neighborId}).first->first);
                Node neighborNode(      neighborId,
                currentNode.DistFromStart + normalizedDistance,
                normalizedHeuristics);
                auto* neighborElement =
                        discQueue.insert({ neighborNode });
                discovered.insert({ neighborId, neighborElement });
        }
        else
        {
                unsigned neighborId = discOrExpIt->second;
                Node neighborNode( neighborId, currentNode.DistFromStart
                        + normalizedDistance, normalizedHeuristics);
                auto discIt = discovered.find(neighborId);
                if (discIt != discovered.end())
                {
                        auto* neighborElement = discIt->second;
                        if (neighborElement->getKey().DistFromStart >
                                neighborNode.DistFromStart)
```

```
                                {
                                        discQueue.decreaseKey(neighborElement,
                                                neighborNode);
                                        cameFrom.insert_or_assign(neighborId,
                                                currentNode.VertexId).second;
                                }
                        }
                }
        }
        neighborVertices.clear();
        distances.clear();
        if (discoveredHeapSizeDelta > 0)
                discoveredHeapSize += discoveredHeapSizeDelta;
}
path.Cost = currentNode.DistFromStart * GetDistanceDenominator();
unsigned anotherId = currentNode.VertexId;
while (anotherId != UINT_MAX)
{
        const t_vertex anotherVertex = *idv[anotherId];
        path.Vertices.push_back(anotherVertex);
        anotherId = cameFrom.at(anotherId);
}
for (u_int i = 0; i < path.Vertices.size() / 2; i++)
        std::swap(path.Vertices[i], path.Vertices[path.Vertices.size() - 1 - i]);
return true;
}
```

ПРИЛОЖЕНИЕ Б

Метод GetNeighbors класса Planner

```
void Planner::GetNeighbors(std::vector<Vertex>& neighbors, std::vector<float>&
distances, const Vertex& vertex, const Vertex& finish) const
{
    if (vertex.ActionCtr > MAX_NUM_ACTIONS_PER_PLAN)
        return;
    for (unsigned i = 0; i < _data.ActionCatalogue.Size(); i++)
    {
        std::vector<ActionInstanceData> actions;
        (*_data.ActionCatalogue.GetItem(i))
          ->ConstructActionInstancesPriori(actions, vertex.ActiveConditionSet,
          vertex.PrevActionInstance.UserData);
        for (auto& action : actions)
        {
            ConditionSet reducedConditionSet(vertex.ActiveConditionSet.Size());
            bool isActionUseful = vertex.ActiveConditionSet.Reduce(action.Effects,
                    reducedConditionSet, action.UserData);
            ConditionSet mergedConditionSet(vertex.ActiveConditionSet.Size());
            bool isActionLegit = reducedConditionSet.Merge(action.Conditions,
                    mergedConditionSet);
            if (isActionUseful == true && isActionLegit == true)
            {
                neighbors.push_back({       mergedConditionSet, i,
                                            action,
                                            vertex.ActionCtr + 1,
                                            *_data.ActionCatalogue.GetName(i)});
                                             distances.push_back(action.Cost);
            }
        }
    }
}
```

ПРИЛОЖЕНИЕ В

Метод GetHeuristic класса Planner

```
float Planner::GetHeuristic(const Vertex& vertex/*active cond set*/, const Vertex&
        finish/*start state*/) const
{
   float heuristics = 0.0f;
   for (int i = 0; i < vertex.ActiveConditionSet.Size(); i++)
      if (vertex.ActiveConditionSet.IsAffected(i))
      {
         auto targetValue =
        std::static_pointer_cast<CEqual>(finish.ActiveConditionSet.GetProperty(i))
                ->Value;
         heuristics+= vertex.ActiveConditionSet.GetProperty(i)->
                Evaluate(     targetValue,
                              _data.AttributeCatalogue.GetItem(i)->get(),
                              vertex.PrevActionInstance.UserData);
      }
}
```

ПРИЛОЖЕНИЕ Г

Метод ConstructStrategy класса Strategist

```
void Strategist::ConstructStrategy(const ValueSet& initState, Strategy& strategy)
const
{
    strategy.GoalIds = std::vector<size_t>(_data.GoalCatalogue.Size());
    std::iota(strategy.GoalIds.begin(), strategy.GoalIds.end(), 0);
    std::vector<size_t> bestStartegy(_data.GoalCatalogue.Size());
    float bestCost = MAX_FLT;
    do
    {
        auto nextInitState = initState;
        float strategyCost = 0.0f;
        for (auto& goalId : strategy.GoalIds)
        {
            strategyCost += GetPlanCostEstimate(nextInitState, goalId) +
                    (*_data.GoalCatalogue.GetItem(goalId))
                    ->GetTardiness(priorities[goalId], goalId);
            nextInitState = GetResultStateEstimate(nextInitState, goalId);
        }
        if (strategyCost < bestCost)
        {
            bestCost = strategyCost;
            bestStartegy = strategy.GoalIds;
        }
    } while (std::next_permutation(strategy.GoalIds.begin(),
          strategy.GoalIds.end()));
    strategy.GoalIds = bestStartegy;
    strategy.Cost = bestCost;
}
float Strategist::GetPlanCostEstimate(const ValueSet& initState, int goalId) const
{
    float p = ((RowVectorXf)initState * gwp[goalId])[0];
    float b = ((RowVectorXf)initState * gwb[goalId])[0];
```

```
    float beta = (1 - BETA_MAX) * b + BETA_MAX;
    return p * beta;
}

ValueSet Strategist::GetResultStateEstimate(const ValueSet& initState, int
goalId) const
{
    RowVectorXf sv = (RowVectorXf)initState * gws[goalId];
    return ValueSet(sv);
}
```

ПРИЛОЖЕНИЕ Д

Метод UpdateAi класса UGoapController

```
void UGoapController::UpdateAi(bool wasActionComplete, bool
mustBuildStrategy, FString& actionName, TMap<FString,int32>& effects)
{
    bool mustUpdateOutput = false;
    if (mustBuildStrategy == true)
    {
        StrategistPtr->ConstructStrategy(_goalPriorities, _currentStrategy);
        _currentGoalIndex = -1;
    }
    if (mustBuildStrategy == true || _currentActionIndex >=
_currentPlan.ActionIds.size())
    {
        _isStateKnown = false;
        if (mustBuildStrategy == false)
        {
            _currentState = (*DataPtr ->GoalCatalogue.GetItem(_currentStrategy
                    .GoalIds[_currentGoalIndex]))->OnGoalCompleted(_currentState);
            if (_mustSaveStatistics == true)
                SavePlanData(true);
        }
        _currentGoalIndex = (_currentGoalIndex + 1) % DataPtr
                ->GoalCatalogue.Size();
        _currentState = (*DataPtr->GoalCatalogue.GetItem(_currentStrategy
                .GoalIds[_currentGoalIndex]))->OnGoalTaken(_currentState);
        _currentPlan.Clear();
        _currentPlan.Goal = (*DataPtr->GoalCatalogue.GetItem(_currentStrategy
                .GoalIds[_currentGoalIndex]))->GetConditions();
        _currentPlan.GoalName = (*DataPtr->GoalCatalogue.GetName(
                _currentStrategy.GoalIds[_currentGoalIndex]));
        _currentPlan.StartState = _currentState;
        MY_ASSERT(PlannerPtr->ConstructPlan(_currentPlan,
                GenerateSupData()));
```

```
        if (_currentPlan.ActionIds.size() == 0)
        {
            _currentActionIndex++; //move on to next goal
            UpdateAi(false, false, actionName, effects);
        }
        _currentActionIndex = 0;
        _expectedState = _currentState;
       expectedState.Modify(_currentPlan.ActionInstances[_currentActionIndex]
          .Effects);
        mustUpdateOutput = true;
    }
    else if (wasActionComplete == true)
    {
        _currentState = _expectedState;
        if (++_currentActionIndex < _currentPlan.ActionIds.size())
        _expectedState.Modify(_currentPlan.ActionInstances[_currentActionIndex]
                .Effects);
        else
            UpdateAi(false, false, actionName, effects);
        mustUpdateOutput = true;
}
if (mustUpdateOutput == true)
{
        effects.Empty();
        actionName = FString(DataPtr->ActionCatalogue.GetName(
                _currentPlan.ActionIds[_currentActionIndex])->c_str());
        for (size_t i = 0; i < _currentPlan.ActionInstances[_currentActionIndex].
                Effects.Size(); i++)
            if (_currentPlan.ActionInstances[_currentActionIndex]
                .Effects.IsAffected(i))
              effects.Add(FString(DataPtr->AttributeCatalogue.GetName(i)->c_str()),
              _currentPlan.ActionInstances[_currentActionIndex].Effects.GetValue(i));
    }
}
```