Министерство образования и науки Российской Федерации

федеральное государственное автономное образовательное учреждение

высшего образования

«Санкт-Петербургский политехнический университет Петра Великого»

**Институт компьютерных наук и технологий**

**Кафедра «Компьютерные интеллектуальные технологии»**

Проект допущен к защите

Директор ИКНТ, проф., д.т.н.

\_\_\_\_\_\_\_\_\_ Заборовский В.С.

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 201\_ г.

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**Исследование методов моделирования принятия решений на примере ботов в компьютерной игре**

направление: 02.04.03 «Математическое обеспечение и администрирование информационных систем»

программа подготовки магистров «Проектирование и разработка информационных систем»

Выполнил(а):

Приходько Евгений Владимирович

Подпись\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель:

доцент, к.т.н., Пак Вадим Геннадьевич

Подпись\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Санкт-Петербург 2017

# РЕФЕРАТ

Работа содержит: 108 с., 61 рис., 7 табл., XX источников, XX прил.

Тема: Исследование методов моделирования принятия решений на примере ботов в компьютерной игре.

Ключевые слова: принятие решений, бот, конечные автоматы, деревья поведения, нечеткая логика, нейронные сети.

Объектом исследования являются алгоритмы и методы принятия решений.

Цель работы – исследование методов моделирования принятия решений на примере ботов в компьютерной игре.

В процессе работы были реализованы боты для компьютерной игры, использующие следующие методы принятия решений: конечные автоматы, деревья поведения, нечеткая логика, нейронные сети. Было произведено сравнение данных методов путем проведения игр реализованных ботов друг против друга.

# Введение

Принятие решений – это выбор одной из альтернатив. Оно может быть автоматизировано с помощью компьютерных программ. Интеллектуальные агенты, принимающие решения, опираются на различные математические модели и алгоритмы такие как алгоритмы поиска оптимального пути, нечеткая логика, конечные автоматы, теория вероятности, нейронные сети и другие.

Системы искусственно интеллекта разделяют на два вида: сильный ИИ и слабый ИИ. Система, обладающая сильным ИИ может мыслить, осознавать себя и реальность, обучаться и принимать решения, основанные на собственном понимании задачи. Системы со слабыми ИИ – это компьютерные программы, созданные для решения конкретной заранее известной задачи. Такие системы окружают нас повсюду: рекомендательные сервисы, системы распознавания голоса, изображений и видео, экспертные системы и базы знаний, роботы, системы «умный дом» и другие.

В данной работе производится обзор различных методов и алгоритмов принятия решений, анализируются их сильные и слабые стороны. На примере разработки бота для компьютерной игры показано, как рассмотренные алгоритмы могут быть применены для создания системы, принимающей решения. В общем случае бот – это компьютерная программа, выполняющая какие-либо действия через интерфейсы, предназначенные для людей. В частном случае компьютерных игр бот – это программа, имитирующая поведение живых игроков в многопользовательских играх. Это агент, который анализирует окружающую обстановку, принимает решения и выполняет действия, которые дают ему игровые преимущества и приводят к выигрышу в соответствии с заданными правилами игры.

Разработка ботов для видеоигр имеет ряд нюансов. Так как основное назначение игр – это развлечение игроков, нет смысла делать компьютерных игроков, которые в любых условиях играют лучше игроков живых, так как в этом случае игра быстро потеряет интерес. Кроме того, все решения бот должен производить в реальном времени, быстро реагируя на действия других игроков. Оба эти фактора приводят к тому, что зачастую при разработке ботов на первый план выходит способность имитировать реальных игроков и вычислительная эффективность алгоритмов, а не максимизация качества самих решений.

# Постановка задачи и обоснование актуальности

## Принятие решений

Принятие решений – процесс построения множества альтернативных возможностей и выбор одного или нескольких элементов из этого множества в соответствии с определенной системой ценностей и предпочтений. Этот процесс может быть автоматизирован с помощью вычислительной техники. Вычислительная система, принимающая решение и действующая в соответствии с ним, называется агентом. Если при этом агент стремится достичь наилучшего результата, такой агент называется рациональным [Рассел и Норвиг].



Рисунок 1.1 – Модель агента

На рисунке представлена модель агента. Агент получает информацию о внешней среде и обновляет свое внутреннее состояние. Затем на основании имеющейся информации и определенной системы правил он производит выбор решения и выполняет связанное с ним воздействие на среду.

В простейшем случае внутреннее состояние может отсутствовать. Такой агент называется простым рефлексным агентом. В этом случае действия, производимые агентом, являются простыми реакциями на внешнюю среду и не зависят от прошлого состояния среды и прошлых действий агента. Такая стратегия оправдана, если датчики всегда предоставляют информацию, которой достаточно для принятия решения.

В случае частичной наблюдаемости среды агент должен поддерживать внутреннее состояние, которое зависит от истории наблюдений и отражает некоторые из ненаблюдаемых аспектов текущего состояния среды. Для корректного обновления внутреннего состояния нужно обладать дополнительной информацией о том, как среда изменяется независимо от агента, и о том, как те или иные действия агента изменяют среду. Эта информация, заложенная в агенте, называется моделью мира, а такой агент – рефлексным агентом, основанным на модели.

Процесс принятия решений можно представить в виде отображения [Шампандар]:

,

где C – контекст, множество возможных состояний мира, S – множество стратегий принятия решений, D – множество существующих решений. Оно показывает, что в соответствии с выбранной стратегией из множества S, реакцией агента на окружение является определенное действие из множества D.

Возможно альтернативное отображение, моделирующее принятие решений:

,

где R – множество вещественных чисел. Каждой комбинации контекста, стратегии и возможного решения ставится в соответствие значение пригодности решения для представленных условий. Оптимальным решением при этом является то, которое при равных состояниях мира и выбранных стратегиях имеет максимальную пригодность.

## Сильный и слабый искусственный интеллект

Философы и инженеры разделяют два подхода к определению искусственного интеллекта: сильный ИИ и слабый ИИ. Впервые такое разделение ввел Джон Сирл в [Сирл]. Сильный искусственный интеллект имеет черты, свойственные человеческому. Система, обладающая сильным ИИ может мыслить, осознавать себя и реальность, обучаться и принимать решения, основанные на собственном понимании задачи. В настоящее время таких систем не существует, и ведутся споры о возможности их существования.

Слабый искусственный интеллект – это программно-аппаратная платформа, разработанная для решения конкретной заранее известной задачи. Такие системы широко используются в наше время: распознавание изображений, видео и голоса, экспертные системы и базы знаний, системы «умный дом», промышленные контроллеры, системы управления роботами и другие системы, основная задача которых – принятие решений.

## Компьютерные игры

В данной работе рассматриваются методы и алгоритмы принятия решений на примере ботов в компьютерной игре. Компьютерные игры или видеоигры – это компьютерные программы для организации игрового процесса. Назначение видеоигр схоже с назначением произведений литературы и кинематографа: они могут использоваться для развлечения, образования или передачи авторской идеи.

Игровая индустрия или индустрия интерактивных развлечений – это один активно развивающихся секторов экономики. Согласно прогнозам аналитиков, оборот игровой индустрии к 2018 году достигнет 96 миллиардов долларов [Video Game Market Overview. DFC Intelligence. Апрель 2016]. Бюджеты крупных игровых проектов сравнялись с бюджетами фильмов-блокбастеров [http://www.scotsman.com/lifestyle/gadgets-gaming/new-gta-v-release-tipped-to-rake-in-1bn-in-sales-1-3081943].

Разработка видеоигр – это технологически сложный процесс. Развитие видеоигр также способствует развитию многих компьютерных наук, таких как компьютерная графика, моделирование физических процессов, алгоритмы и структуры данных, искусственный интеллект и другие.

## Боты в компьютерных играх

В общем случае бот – это компьютерная программа, выполняющая какие-либо действия через интерфейсы, предназначенные для людей. В частном случае компьютерных игр бот – это программа, имитирующая поведение живых игроков в многопользовательских играх. Это агент, который анализирует окружающую обстановку, принимает решения и выполняет действия, которые дают ему игровые преимущества и приводят к выигрышу в соответствии с заданными правилами игры.

Благодаря большой вычислительной мощности современных компьютеров боты могут принимать более эффективные решения чем игроки и с более высокой скоростью. Однако, так как основной задачей видеоигр является развлечение игрока, боты должны имитировать поведение живого человека, играющего с ним на равных. В противном случае игра против оппонента, которого практически невозможно победить, быстро приведет к потере интереса.

## Правила игры

В данной работе рассматривается пример игры в жанре «шутер с видом сверху» (top-down shooter). В играх этого жанра игроки управляют вооруженными персонажами, сражаются с другими игроками или ботами и наблюдают за полем боя с высоты птичьего полета. Графика может быть как двухмерной, так и трехмерной, но игровая логика как правило рассчитывается в двухмерной координатной системе.

{добавить скриншоты из разрабатываемой игры, когда она появится}

Правила игры:

* Игроки обладают ограниченной дальностью и углом обзора.
* В игре есть стены и укрытия. За стеной не видно других игроков. Укрытия не мешают обзору, но все выстрелы могут с вероятностью 50% быть заблокированными укрытием, если стреляющий игрок стоит от него дальше, чем на 1 метр.
* Игроки могут делиться на команды.
* Игра ведется до поражения всех оппонентов. Выигрывает команда, в которой остались игроки.

В терминах теории игр рассматриваемая игра:

* симметричная;
* с нулевой суммой;
* непрерывная;
* параллельная;
* с неполной информацией.

Существует большое количество игр этого жанра, но во всех этих играх в качестве врагов игрока используются слабые монстры, которые нападают со всех сторон и пытаются одолеть игрока количеством. В них в качестве развлекательного фактора используется постоянное движение и стрельба по большому количеству целей. В данной работе предлагается рассмотреть симметричную многопользовательскую игру, в которой все участники, как живые, так и компьютерные, находятся в равных условиях. В этом случае победитель определяется навыками: быстротой реакции, качеством тактических решений, меткостью стрельбы.

## Цель и задачи работы

Целью данной работы является исследование методов моделирования принятия решений на примере ботов в компьютерной игре.

Задачи, решаемы в ходе работы:

* рассмотреть существующие методы и алгоритмы принятия решений;
* реализовать игровых ботов, использующих рассмотренные алгоритмы;
* сравнить полученные реализации по различным критериям;
* сделать выводы об эффективности различных методов и алгоритмов для моделирования процесса принятия решений.

# Обзор методов и алгоритмов принятия решений

## Нечеткая логика

Нечеткая логика (fuzzy logic) является расширением классической математической логики [Zadeh]. В нечеткой логике каждая переменная может принимать вещественное значение в диапазоне [0; 1]: 0 соответствует значению «ложь», 1 – значению «истина», промежуточные значения – частичной истинности. Область значений переменных в классической логике является подмножеством области значений переменных в нечеткой логике.

Нечеткая логика применяется в условиях неопределенности, вызванных отсутствием четкого знания о среде. Такая неопределенность может возникнуть из-за аппаратурной погрешности датчиков, влияния шума, отсутствия теоретической или практической возможности вычислить нужную величину или из-за природы самой величины.

### Операции над элементами

Операции в нечеткой логике могут быть определены разными способами в зависимости от решаемой задачи. При этом они должны быть применимы к переменным классической логики, т.е. при подстановке значений 0 и 1 они должны давать результаты, соответствующие таблицам истинности операций из классической логики. Рассмотрим операции и варианты их определений.

#### Отрицание

Отрицание нечеткой логической переменной x можно получить путем вычитания ее значения из единицы [Zadeh]:

¬x = 1 – x.

#### Конъюнкция и дизъюнкция

Конъюнкция и дизъюнкция в нечеткой логике задаются с помощью треугольных нормы T и конормы S соответственно [Батыршин].

Функции T, S: [0; 1] × [0; 1] → [0; 1] называются треугольной нормой (t-нормой) и треугольной конормой (t-конормой), если они:

* 1. монотонны: T(x, y) ≤ T(y, z), S(x, y) ≤ S(y, z), если x ≤ y ≤ z;
  2. ассоциативны: T(T(x, y), z) = T(x, T(y, z)), S(S(x, y), z) = S(x, S(y, z));
  3. коммутативны: T(x, y) = T(y, x), S(x, y) = S(y, x);
  4. связаны соотношениями де Моргана: ¬T(x, y) = S(¬x, ¬y) и ¬S(x, y) = T(¬x, ¬y);
  5. удовлетворяют граничным условиям: T(0, 0) = T(0, 1) = T(1, 0) = 0, T(1, 1) = 1, S(0, 0) = 0, S(0, 1) = S(1, 0) = S(1, 1) = 1.

Примеры определений конъюнкции и дизъюнкции, удовлетворяющих перечисленным условиям [Батыршин, Штовба]:

Пересечение и объединение по Заде [Zadeh]:

x ∧ y = min(x, y);

x ∨ y = max(x, y).

Троичная логика Лысенко:

x ∧ y = max(a + b – 1, 0);

x ∨ y = min(a + b, 1).

Вероятностные пересечение и объединение:

x ∧ y = xy;

x ∨ y = x + y – xy.

#### Импликация

Импликация определяет причинно-следственное отношение между посылками и заключениями логических правил. Перечислим несколько вариантов определения импликации [Тенетко].

Импликация Геделя (Godel):

x → y = .

Импликация Гогена (Goguen):

x → y = .

Импликация Клини-Дайнса (Kleene-Dienes):

x → y = max(1 - x, y).

### Нечеткие множества

На нечеткой логике строится теория нечетких множеств. Ее основная идея заключается в том, что в отличие от классической теории множеств, где элемент либо принадлежит множеству, либо нет, здесь каждый элемент имеет меру вхождения с областью значений [0; 1].

Пусть 𝔛 = {x} – множество всех рассматриваемых элементов, A ⊆ 𝔛 – нечеткое множество. Для A определена функция принадлежности (membership function):

μA: 𝔛 → [0; 1].

Значение μA(x) = 0 означает, что элемент x не входит в множество A, 1 – что входит полностью, промежуточные значения означают частичное вхождение.

Запись нечетких множеств обычно имеет вид [Bai]:

A = .

При этом знак + не является суммированием, а горизонтальная черта не является делением – такая запись принята в литературе и означает перечисление элементов вместе с их мерами вхождения.

### Операции над множествами

Нечеткие множества являются расширением обычных множеств, поэтому операции над ними должны быть применимы так же и к обычным множествам.

Операции над нечеткими множествами определены следующим образом [Zadeh]:

A = B ⇔ ∀x μA(x) = μB(x),

A ⊂ B ⇔ ∀x μA(x) ≤ μB(x),

A ⋂ B: μA⋂B(x) = μA(x) ∧ μB(x) = T(μA(x), μB(x)),

A ⋃ B: μA⋃B(x) = μA(x) ∨ μB(x) = S(μA(x), μB(x)).

На рисунке Рисунок 2.1 показаны два примера нечетких множеств: A – множество чисел, примерно равных 2, B – множество чисел, примерно равных 3.

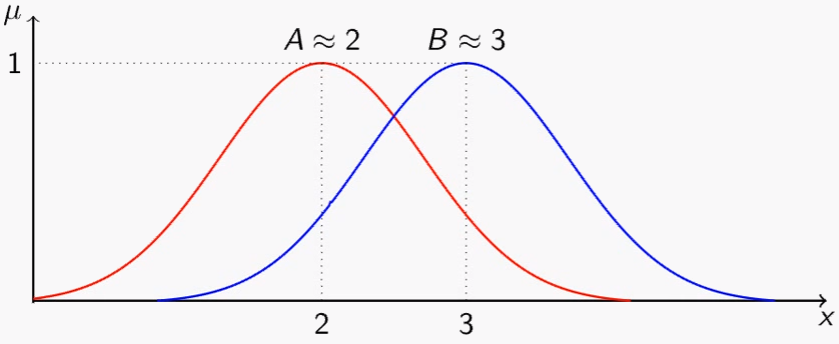


Рисунок 2.1 – Примеры нечетких множеств

На рисунках Рисунок 2.2-Рисунок 2.5 показаны соответственно объединения и пересечения этих множеств с использованием разных определений дизъюнкции и конъюнкции. Выбор конкретного определения зависит от решаемых задач. На примере представленных множеств объединение с использованием произведения функций принадлежности является более подходящим, так как дает более плоское плато у объединенной функции. В этом случае число 2.5 в большей степени принадлежит множеству чисел примерно равных 2 и примерно равных 3, чем 1.5 или 3.5, что более логично.

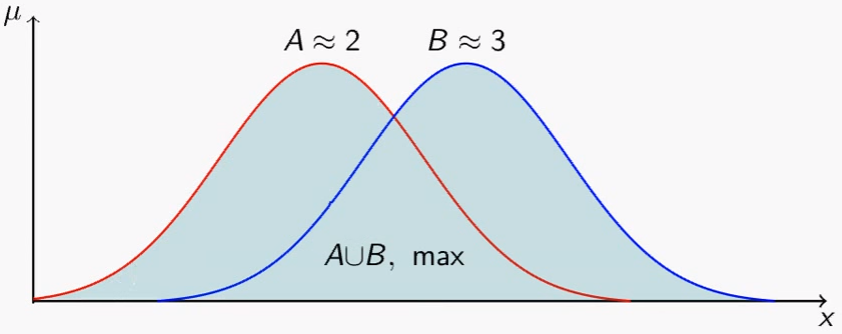


Рисунок 2.2 – Объединение множеств с использованием нормы, определенной через максимум функций принадлежности

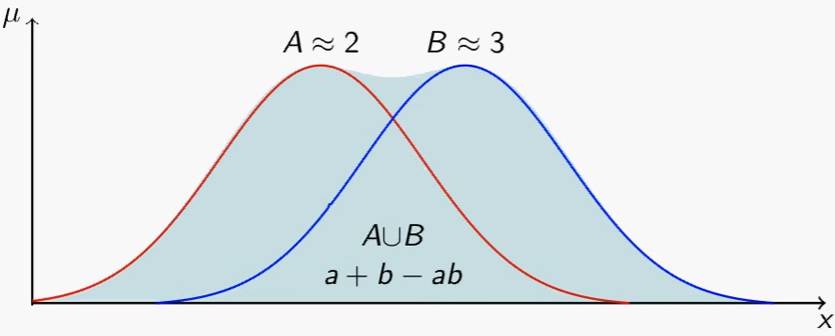


Рисунок 2.3 – Объединение множеств с использованием нормы, определенной через сумму функций принадлежности

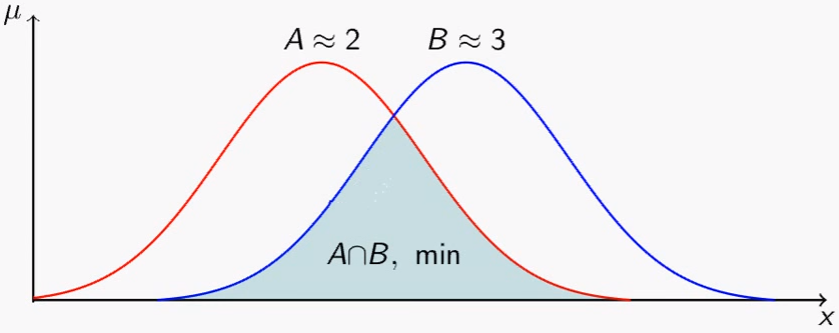


Рисунок 2.4 – Пересечение множеств с использованием конормы, определенной через минимум функций принадлежности

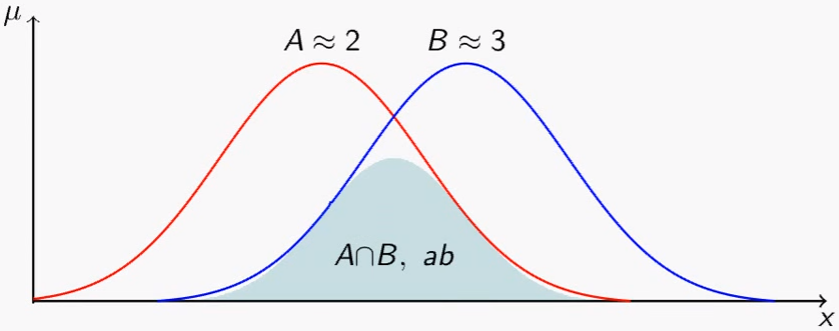


Рисунок 2.5 – Пересечение множеств с использованием конормы, определенной через произведение функций принадлежности

### Лингвистические переменные

Лингвистической переменной называется переменная, значениями которой являются слова или предложения естественного или искусственного языка [Zadeh 1976]. Например, «возраст» - лингвистическая переменная, если она принимает лингвистические, а не числовые значения, т.е. значения «молодой», «не молодой», «очень молодой», «старый», «не очень старый» и другие.

Нечеткая лингвистическая переменная (или просто лингвистическая переменная) – это набор нечетких множеств с одинаковой областью определения, каждое из которых определяет одно лингвистическое значение [Buckland]. Такие лингвистические значения также называются термами [Штовба]. Примеры лингвистических переменных:

Возраст = { молодой, не очень молодой, не очень старый, старый };

Температура = { холодно, прохладно, нормально, тепло, жарко };

Угловое отклонение = { сильно влево, влево, немного влево, нет, немного вправо, вправо, сильно вправо },

На рисунке Рисунок 2.6 изображено графическое представление переменной «угловое отклонение».

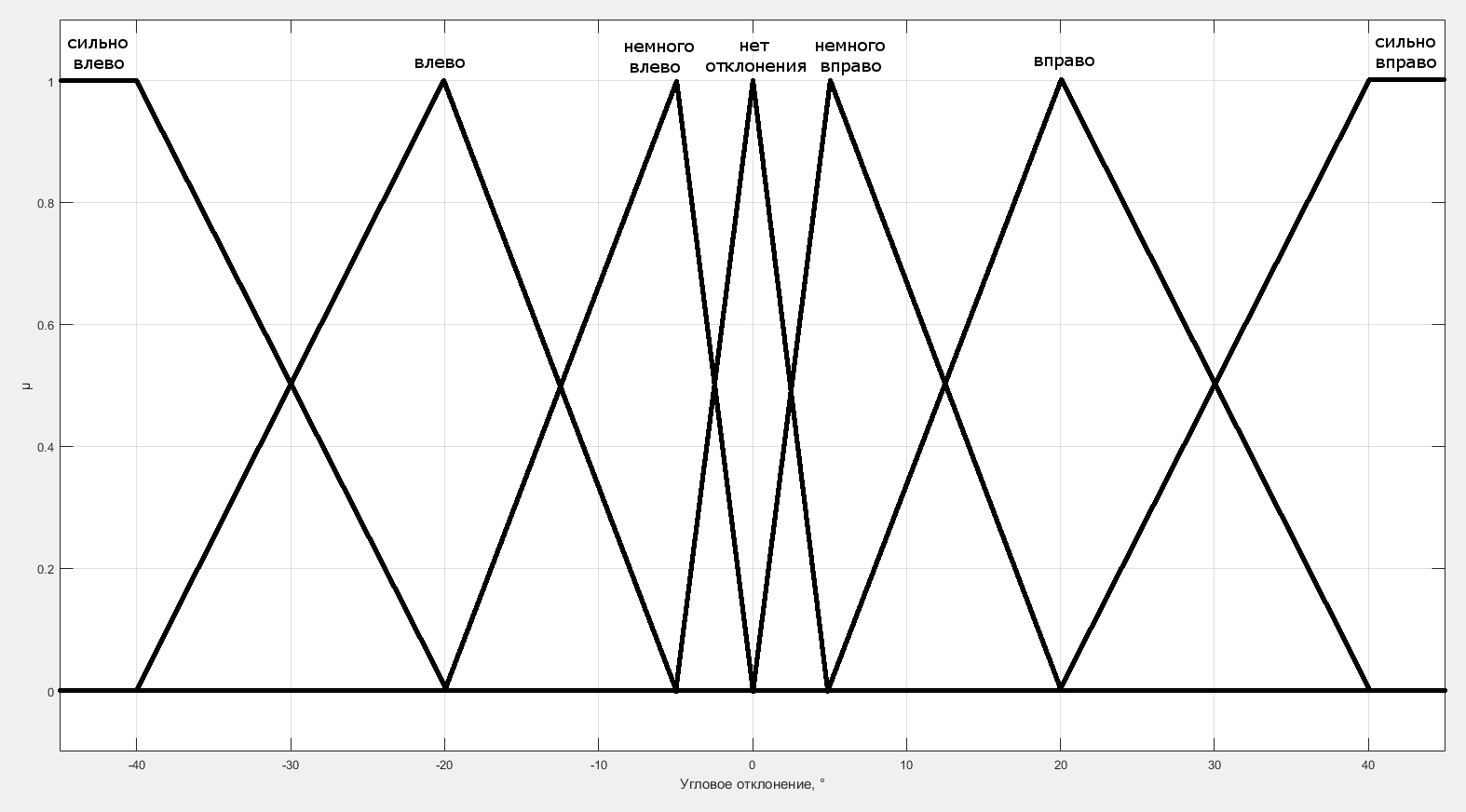


Рисунок 2.6 – Графическое представление лингвистической переменной «угловое отклонение»

Строгих правил для определения лингвистических переменных и их термов нет, но существуют рекомендации, позволяющие упростить разработку, улучшить качество и скорость вычислений [Buckland].

* Функции принадлежности термов должны в совокупности иметь ненулевые значения на всей области определения переменной.
* Сумма значений функций принадлежности всех термов для каждой точки области определения переменной должна быть примерно равна 1 для получения результата без разрывов.
* Форма функции принадлежности в общем случае может быть любой, но рекомендуется выбирать кусочно-линейные функции (треугольные и трапециевидные) для простоты расчетов.
* Для каждой точки области определения должно быть не более двух термов с ненулевой мерой для простоты расчетов.

### Применение нечеткой логики

Работа с нечеткой логикой в практических задачах состоит из трех этапов [Bai]:

* 1. фаззификация (fuzzification) – преобразование четкого значения в нечеткое;
  2. нечеткий вывод (fuzzy inference) с использованием управляющих правил;
  3. дефаззификация (defuzzification) – преобразование нечеткого значения в четкое.

#### Фаззификация

Четкие переменные, отражающие некоторые параметры окружающей среды, должны быть фаззифицированы, т.е. преобразованы к нечеткой лингвистической переменной. Для этого значение переменной нужно подставить в функцию принадлежности каждого терма и взять те термы, для которых значение функции не равно нулю.

#### Нечеткие управляющие правила и нечеткий вывод

Управляющие правила в нечеткой логике строятся по шаблону:

ЕСЛИ условие ТО следствие.

При проектировании нечетких систем управления правила строятся в большей степени на основе экспертного мнения, а не на теории, как в системах четкого управления. Полученные правила обычно похожи на предложения естественного языка:

ЕСЛИ прохладно ТО немного увеличить мощность обогревателя;

ЕСЛИ отклонение немного влево ТО немного повернуть ручку управления вправо;

ЕСЛИ давление резко увеличивается И температура стабильна ТО открыть клапан.

В общем виде правило имеет вид:

ЕСЛИ И И … И

ИЛИ И И … И

…

ИЛИ И И … И

ТО y = dj, для всех j=1..m,

где x1, x2, …, xn – входные лингвистические переменные;

y – выходная лингвистическая переменная;

– терм, которым оценивается переменная xi в строчке с номером jp (p=kj);

kj – количество строчек-конъюнкций, в которых выход оценивается нечетким термом dj, j=1..m;

m – количество термов, используемых в выходной лингвистической переменной [Штовба].

Правила должны покрывать все комбинации входных переменных. Общее количество строчек-конъюнкций равно:

N = , где mi – количество термов в переменной xi.

Полный набор правил называется нечеткой базой данных о влиянии факторов x1, x2, …, xn на значение параметра y.

Нечетким логическим выводом называется аппроксимация зависимости y=f(x1, x2, …, xn) с помощью нечеткой базы данных и операций над нечеткими множествами.

#### Дефаззификация

Результатом нечеткого вывода является набор термов – нечетких множеств. Для получения четкого значения, которое затем можно использовать в качестве управляющего воздействия, необходимо произвести дефаззификацию.

Рассмотрим основные методы дефаззификации [Bai, Buckland]:

Центр максимумов (mean of maximum):

x\* = ,

где a, b – самая левая и самая правая точки, в которых функция принимает максимальное значение. На рисунке Рисунок 2.7 изображен пример дефаззификации с помощью центра максимумов. Данный метод позволяет быстро рассчитать результат, но он не учитывает влияние множеств со степенью достоверности меньше максимальной.

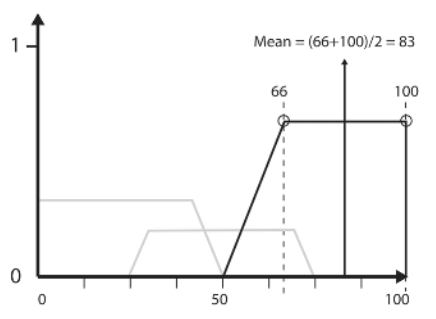


Рисунок 2.7 – Пример дефаззификации с помощью центра максимумов

Центр масс (center of gravity, centroid):

x\* = для непрерывных величин,

x\* = для дискретных величин.

Один из самых распространенных методов. Дает точный результат, но сложно рассчитывается. Пример результата изображен на рисунке Рисунок 2.8. Для ускорения расчетов можно увеличить шаг суммирования (рисунок Рисунок 2.9).

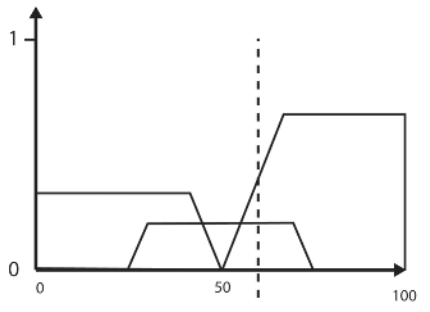


Рисунок 2.8 – Пример дефаззификации с помощью центра масс

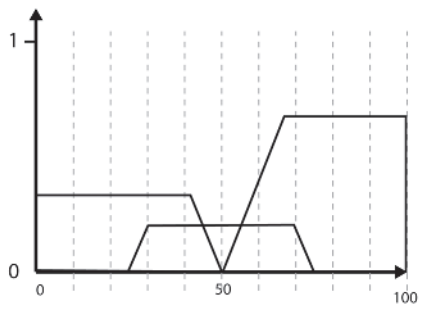


Рисунок 2.9 – Пример разбиения области определения на дискретные интервалы

Взвешенная сумма (weighted average):

x\* = ,

где di – i-й терм выходной лингвистической переменной, mom(di) – средний максимум i-го терма выходной переменной.

Выбор метода дефаззификации, как и выбор нормы и конормы, зависит от решаемой задачи и обычно подбирается экспериментально.

### Метод Комбса

Как уже было сказано ранее, количество правил в конъюнктивной форме равно:

N = ,

где n – количество входных лингвистических переменных, mi – количество термов в переменной xi.

Увеличение количества входных переменных приводит к комбинаторному взрыву – резкому увеличению количества правил, необходимого для описания работы системы. Это приводит как к увеличению времени разработки, так и к увеличению времени расчета результата нечеткого вывода. Одним из решений проблемы комбинаторного взрыва применительно к нечеткой логике является метод Комбса [Buckland, Combs]. В основе метода лежит логическое тождество:

((p ∧ q) → r) ⇔ ((p → r) ∨ (q → r)).

Оно позволяет преобразовать правила нечеткого логического вывода вида

ЕСЛИ условие1 И условие2 ТО следствие

к виду

ЕСЛИ условие1 ТО следствие

ИЛИ

ЕСЛИ условие2 ТО следствие.

При этом правила с одинаковыми условиями объединяются, и итоговое количество правил становится:

N = .

Если количество термов в каждой переменной равно m, то получаем N(n) = m⋅n вместо N(n) = mn, т.е. линейная скорость роста вместо показательной.

Для нечеткой логики такое преобразование равнозначно, только если функция y=f(x1, x2, …, xn) является аддитивно разделяемой (additively separable) [Ross], т.е. f(x1, x2, …, xn) = f1(x1) + f2(x2) + … fn(xn), что в общем случае не так. Таким образом, метод Комбса не позволяет без потерь преобразовать большое количество правил нечеткого вывода к меньшему количеству, но он предоставляет альтернативный подход к изначальному определению нечеткой базы данных как набора правил с одной переменной в условии.

### Преимущества и недостатки нечеткой логики

Нечеткая логика применяется, когда нет четкого знания об окружающей среде. Это может быть вызвано недостатками датчиков, отсутствием модели окружающего мира или сложностью ее применения. Нечеткая логика позволяет свести сложный математический аппарат предметной области к набору эмпирических правил вида «если X, то Y», основываясь на экспертном мнении и экспериментах.

Основным недостатком нечеткой логики является отсутствие математической строгости и неоднозначность определений. Например, разными способами могут быть определены: конъюнкция, дизъюнкция, импликация, набор и форма правил вывода, набор входных и выходных переменных, термы переменных, функции принадлежности термов, методы дефаззификация и другое. Нет возможности заранее рассчитать, как то или иное определение повлияет на результат, поэтому зачастую разработка нечеткой системы управления сводится к перебору параметров и методов, пока результат не будет удовлетворять требованиям. При этом увеличение количества входных переменных и их термов приводит к комбинаторному взрыву количества правил вывода, требуемых для описания системы. Существуют методы для уменьшения количества правил вывода, но они также влияют на результат и требуют экспериментов для определения их применимости.

## Конечные автоматы

Автомат – математическая абстракция, модель дискретного устройства с одним входом, одним выходом и внутренним состоянием. Если количество внутренних состояние конечно, то такой автомат называется конечным (finite-state machine, FSM, finite-state automaton, FSA). В общем случае конечный автомат представляет собой множество состояний и множество переходов между ними и может быть представлен в виде графа (рисунок Рисунок 2.10).

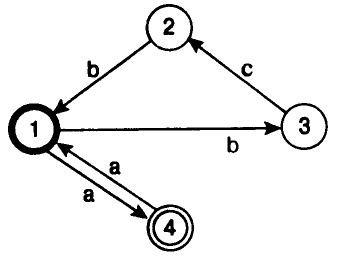


Рисунок 2.10 – Графическое представление конечного автомата

Рассмотрим формальные определения конечных автоматов [Шампандар].

#### Распознающий конечный автомат

В распознающих конечных автоматах в качестве результата работы автомата является его внутренне состояние, в котором он окажется после некоторого набора входных символов. Они используются для распознавания паттернов, дешифрации последовательностей символов и реакций на продолжительное наблюдение за целью. Формально распознающий конечный автомат можно представить в виде множества:

FSM = { I, S, s0, T, δ },

где I – входной алфавит (множество символов, из которых составляются слова, формирующие входные воздействия), S – множество внутренних состояний, s0 ∈ S – начальное состояние, T ⊆ S – множество терминальных состояний, δ: S×I → S – функция перехода.

Входной алфавит – это набор атомарных событий, на которые должен реагировать автомат. Это не обязательно символы, считываемые с ленты. В случае компьютерных игр в качестве входного алфавита используются события: действия оппонентов, изменения обстановки, изменение параметров агента. Распознающий автомат может быть использован для определения маневра, выполняемого оппонентом. Если за одним действием, он выполняет другое, то такая последовательность действий может требовать отдельной реакции, отличающейся от отдельных реакций на каждое атомарное событие. Ответная реакция производится при достижении какого-либо из терминальных состояний.

Функция перехода δ – это отображение, которое текущему состоянию и пришедшему символу ставит в соответствие новое состояние. Если после получения символов, находясь в определенном состоянии, автомат остается в том же состоянии, такой переход называют рекуррентным.

#### Порождающий автомат Мили

В порождающих автоматах в отличие от распознающих результатом работы является поток выходных символов. Формально порождающий конечный автомат можно представить в виде множества:

FSM = { I, O, S, s0, T, δ, λ },

где O – выходной алфавит, λ – функция выходов.

В автоматах Мили выходное символы определяются пройденными переходами между состояниями. Функция выходов λ: S×I → O – это отображение текущего состояния и входного символа на выходной символ.

#### Порождающий автомат Мура

В автоматах Мура выходные символы определяются только текущим состоянием. В отличие от автоматов Мили данные автоматы не учитывают, как именно автомат перешел в текущее состояние. Функция выходов λ: S → O – это отображение текущего состояния на выходной символ.

### Преимущества и недостатки классических конечных автоматов

Порождающие конечные автоматы используются для изменения поведения агентов при изменениях в окружающей среде. Например, поведение агента может изменяться в зависимости от того, есть ли в поле зрения другие агенты. В этом случае он будет переключаться между режимом патрулирования и режимом атаки/защиты. Какие порождающие автоматы использовать – Мили или Мура – зависит от поставленных задач. Как правило, не важно, каким образом и из какого состояния автомат перешел в текущее. Чаще используется автомат Мура. Обычно важно то, в каком именно состоянии сейчас находится агент, так как это определяет его поведение в текущий момент.

Распознающие автоматы могут использоваться для наблюдения за другими объектами, чтобы определить паттерны их поведения. С помощью такого автомата можно понять, что оппонент готовится к некоторому опасному действию, и выполнить ответное действие для его предотвращения.

К преимуществам конечных автоматов можно отнести простоту проектирования при малом количестве состояний. Достаточно просто можно составить список состояний агента и условий перехода между ними.

При росте количества состояний разработка агента становится сложнее, так как при добавлении нового состояния нужно пересматривать все существующие переходы, изменять их и добавлять новые. В худшем случае для N состояний может быть N(N-1) переходов.

Конечные автоматы не являются полными по Тьюрингу [10 Reasons the Age of Finite State Machines is Over]. Это значит, что некоторые задачи не могут быть решены с их помощью. Например, нельзя распознать циклические действия, т.е. определить, что некоторое действие выполняется несколько раз подряд, не используя внешние счетчики.

Автомат может находиться только в одном состоянии в один момент времени. Это упрощенная модель агента. В реальности не всегда можно точно сказать, что агент полностью находится в состоянии защиты или в состоянии атаки. Такое упрощение модели ограничивает область применения автоматов.

Конечные автоматы могут использоваться в тех случаях, когда количество состояний мало, они слабо меняются в ходе разработки и агент не может находится в нескольких состояния одновременно. Например, они могут использоваться для выбора глобальной стратегии поведения.

Конечные автоматы используются в игре Batman: Arkham Asylum [Hanagan] для моделирования поведения охранников. В спокойном состоянии они патрулируют местность. При появлении подозрительных объектов, у них меняется состояние на подозрительное. Они исследуют объект и переходят в состояние активного поиска.

### Иерархические конечные автоматы

В иерархических автоматах каждое состояние может быть либо обычным состоянием, либо конечным автоматом. Иерархический автомат может быть преобразован к классическому виду и обратно за счет изменения количества переходов (рисунок Рисунок 2.11).

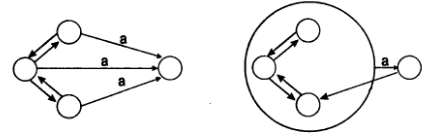


Рисунок 2.11 – Графическое представление классического конечного автомата и иерархического, соответствующих одинаковому поведению

Наличие перехода от вложенного конечного автомата в другое состояние эквивалентно наличию переходов их каждого состояния вложенного автомата. При переходе во вложенный конечный автомат агент переходит в его начальное состояние.

Иерархические конечные автоматы предоставляют альтернативную форму записи и представления в программном коде, но не дают выигрыша по производительности. Они позволяют уменьшить количество связей и увеличить количество состояний, при котором комфортно вести разработку. При дальнейшем роста количества состояний эти преимущества теряются.

### Конечные автоматы с магазинной памятью

Конечные автоматы с магазинной памятью (pushdown automata, stack based FSM) – это конечный автомат, который использует стек для хранения состояний [Белоусов]. Формально его можно представить в виде множества:

FSM = { I, M, S, s0, T, δ },

где M – алфавит памяти (магазина). Конечный автомат может быть порождающим. В этом случае добавляется выходной алфавит и функция выхода. Функция перехода учитывает память: δ: S×I×M → S×M.

Конечные автоматы с памятью позволяют производить более сложные операции, недоступные классическим автоматам. При моделировании принятий решений такие автоматы используются для сохранения истории переходов. Если условие, из-за которого автомат перешел в текущее состояние, больше не выполняется, берется состояние из стека, и агент возвращается к тому, что делал до этого.

### Вероятностные конечные автоматы

Вероятностные автоматы (probabilistic finite-state machine, PFSM) – это конечные автоматы, у которых каждый переход из одного состояния в другое имеет определенную вероятность. Они могут рассматриваться как частный случай марковских цепей первого порядка. Формально их можно представить в виде множества:

FSM = { S, O, s0, T, δ, λ },

где δ: S×[0;1] → S. Такой порождающий конечный автомат может использоваться для генерирования случайных последовательностей действий из выходного алфавита O и добавления реализма и разнообразия в поведение агента.

К вероятностным конечным автоматам так же можно отнести недетерминированные автоматы, когда одной и той же паре текущего состояния и входного символа соответствует несколько следующих состояний. В этом случае δ является не функцией, а отношением перехода. Один из подходов разрешения неопределенности – поддерживать параллельно все возможные состояния и объединять их выходы, но такой подход может привести к экспоненциальному взрыву возможных одновременных состояний. Другой подход – присвоить каждому из возможных переходов вероятность. В этом случае автомат можно представить так:

FSM = { S, I, s0, T, δ },

где δ: S×I×[0;1] → S - отношение перехода. Сумма вероятностей переходов при одинаковых входных символах и текущих состояниях должна быть равна 1. Если переход определен однозначно, то его вероятность равна 1, что соответствует детерминированному автомату.

### Нечеткие конечные автоматы

Нечеткие конечные автоматы (fuzzy finite state machine, FFSM, FuSM) являются расширением классических конечных автоматов [Шампандар]. Такой автомат может находиться в нескольких состояниях одновременно. Формально его можно представить в виде множества:

FFSM = { I, S, s0, μS, μI, T, δ },

где I – множество нечетких входных переменных, S – множество состояний, s­0­­ ∈ S – начальное состояние, μS – функция принадлежности автомата некоторому состоянию (вес состояния), μI – функция истинности входной переменной, T ⊆ S – терминальные состояния, δ – функция переходов, основанная на нечетком выводе. Нечеткий конечный автомат может быть порождающим. При каждом пересчете состояний новый вес вычисляется через текущие веса и значения нечетких входных переменных.

Так как автомат может находиться в нескольких состояниях одновременно, необходимо пересчитывать все веса, а не только проверять текущее состояние, как в классических автоматах. При этом проще считать суммарный эффект входящих переходов, а не проверять все исходящие. Выражение для вычисления весов состояний и условия их корректности:

где – новый вес состояния sj, – текущий вес состояния si, S – треугольная конорма, T – треугольная норма, tij – переход от состояния si к состоянию sj, – мера истинности условия перехода tij, – множество всех переходов, приводящих в состояние sj, – множество всех переходов, исходящих из состояния si. В зависимости от выбора нормы и конормы первое условие корректности может не выполняться после расчета весов. В этом случае необходимо произвести нормализацию:

Нечеткие конечные автоматы позволяют агенту находится в нескольких состояниях одновременно, что делает его поведение более реалистичным. Остальные недостатки классических конечных автоматов остаются. Увеличивается сложность проектирования. Если в обычных автоматах можно проектировать только те переходы, которые нужны, в нечетких автоматах требуется соблюдать условия о полноте правил вывода, чтобы они покрывали все возможные комбинации входных переменных. Также добавляются другие проблемы нечеткой логики: отсутствие строгих определений операторов и большое количество вычислений.

## Деревья поведения

Деревья поведения (behavior trees) – это деревья, в которых каждый листовой узел определяет поведение агента, а промежуточные узлы отвечают за выбор действия или организацию последовательностей действий [Millington]. Листья не имеют связей друг с другом и инкапсулируют некоторое поведение. За счет этого деревья могут содержать сотни узлов и реализовывать сложные паттерны.

Когда узел-родитель обращается к дочернему элементу, тот может вернуть один из трех результатов:

* Успех (success) – действие выполнено.
* Провал (failure) – действие не удалось выполнить.
* Выполнение (running) – действие все еще находится в процессе выполнения, конечный результат еще не получен.

Промежуточные узлы могут быть нескольких видов [Colledanchise]:

* Селектор (selector) – узел, в котором происходит выбор, какой из дочерних элементов должен выполняться. Селектор задает цель, а его дочерние элементы – это варианты поведения, которые позволяют достичь ее. В селекторе располагается логика, отвечающая за выбор наиболее подходящего решения. Примеры селекторов:
  + Вероятностный селектор (probability selector) – селектор, который выбирает решение случайным образом.
  + Селектор с приоритетами (priority selector) – селектор, дочерние элементы которого расположены в порядке уменьшения приоритета. Решения перебираются по очереди. Если решение с бо́льшим приоритетом не подходит (возвращает «Провал»), то селектор переходит к следующему.
* Последовательность (sequence) – узел, дочерние элементы которого должны выполняться по очереди.
* Параллельное выполнение (parallel) – узел, дочерние элементы которого запускаются одновременно и выполняются параллельно.
* Декоратор (decorator) – узел, который имеет только один дочерний элемент и используется для корректировки его вызова или обработки результата. Примеры декораторов:
  + Таймер – блокирует новый вызов дочернего элемента, если с прошлого вызова прошло недостаточно времени. Используется, чтобы некоторые поведения не использовались слишком часто.
  + Повторитель – вызывает дочерний элемент несколько раз подряд.
  + Счетчик – ограничивает количество вызовов дочернего элемента.
  + Выключатель – по внешним сигналам блокирует и разблокирует дочерний элемент.
  + Модификаторы результата – изменяют результат, возвращаемый дочерним элементом и передают его родительским узлам.
  + Отладочные узлы – останавливают выполнение, если сработал дочерний элемент, или используются для журналирования.

На рисунке Рисунок 2.12 изображен пример простого дерева, реализующего проход агента через дверь.



Рисунок 2.12 – Пример дерева поведения

Это отдельное дерево, которое может быть использовано в других деревьях как модуль. В основе его лежит последовательность: подойти, открыть, пройти и закрыть. Но при открытии могут возникнуть различные исходы, а значит нужно с помощью селектора предусмотреть различные реакции. В данном примере используется селектор с приоритетом. Сначала агент пытается просто открыть дверь. Если это не удается и дверь закрыта, то агент пытается открыть замок ключом. Если ключа нет или это действие не удается по другой причине, агенту остается вариант выбить дверь. Если все действия селектора проваливаются (возвращают результат «Провал»), то сам селектор возвращает отрицательный результат и основная последовательность действий завершается. Отрицательный результат будет возвращаться дальше родительским узлам, пока не будет встречен узел, который сможет его обработать. Например, селектор, который вместо прохода через дверь будет искать обходной путь.

### Преимущества и недостатки деревьев поведения

Деревья поведения появились и активно развивались в индустрии компьютерных игр как инструмент, который позволяет упростить разработку поведения агентов за счет модульности [Isla 2005, Isla 2008, Lim 2010, Gillberg 2016]. По сравнению с конечными автоматами у модулей в деревьях меньше связность, что позволяет уменьшить количество сущностей и упростить разработку сложных моделей поведения, состоящих из сотен узлов [Understanding Behavior Trees]. Инкапсуляция атомарных поведений в листьях дерева и более сложных паттернов в поддеревьях позволяет переиспользовать модули в других ветвях одного дерева, разных деревьях и даже в разных проектах. Существуют отдельные редакторы деревьев поведения, с помощью которых геймдизайнеры могут редактировать логику агентов без помощи программистов.

Деревья могут быть очень большими, и постоянный проход, начиная с корня, может привести к излишней нагрузке на процессор. Разбиение всего дерева на поддеревья и проверка только текущего может исправить эту проблему, но в этом случае не будут выполняться проверки в других поддеревьях, а значит может быть пропущено какое-то изменение среды и агент не прореагирует на него должным образом.

Деревья поведения – это способ организации атомарных действий и объединения их в последовательности и ветвления. Они не дают ответ на вопрос, какое решение будет наиболее эффективным в данный момент времени. Эту задачу должен решать разработчик при реализации селекторов.

Агент, основанный на деревьях поведения, реализует жесткие последовательности действий, заданные дизайнерами. Можно внести разнообразие, используя вероятностные селекторы, но в этом случае могут появиться проблемы с отладкой больших деревьев. Такие агенты практически не проявляют эмерджентного поведения. Эмерджентность – это наличие у некоторой системы свойств, не присущих ее компонентам по отдельности. Это значит, что такой агент всегда будет использовать одни и те же средства для решения одних и тех же задач, заранее определенных разработчиком. Агент сможет изменить свое поведение, подстраиваясь под стиль игры оппонентов, только в том случае, если эти изменения учтены на этапе проектирования дерева.

Деревья поведения – это одна из самых популярных моделей принятия решений в компьютерных играх, но его основное применение – моделирование поведения персонажей, у которых есть жестко заданная роль, которой нужно придерживаться. Отклониться от этой роли, используя одни только деревья, они не смогут.

## Нейронные сети

Искусственные нейронные сети (artificial neural networks, ANNs) – это программные структуры, которые могут обучаться и используются для аппроксимации (обобщения) функции по известным значениям исходной функции (обучающее множество). Основное назначение нейронных сетей – это задача классификации, т.е. решение о принадлежности набора параметров на входе определенному классу. Если в качестве входных параметров использовать знания агента об окружающей среде, а в качестве классов – реакции на различные состояния среды, то нейронные сети могут быть использованы для моделирования принятия решений.

### Модель нейрона

Нейронные сети состоят из большого количества однотипных узлов, называемых искусственными нейронами, основанных на упрощенной модели нервной клетки [Шампандар]. При объединении в сети выходы одних нейронов являются входами других. Модель нейрона состоит из нескольких входов, взвешенного сумматора, функции активации и одного выхода (рисунок Рисунок 2.13). Она аппроксимирует функцию , где – вектор входных значений. Каждому входу xi соответствует некоторый коэффициент wi – вес, характеризующий вклад этого входа в итоговый результат. Один из входов отвечает за смещение нуля. Его можно представить либо как отдельный вход, отвечающий только за смещение, либо как один из обычных входов, на который всегда подается единица. Обучение нейрона заключается в подборе весовых коэффициентов, обеспечивающих ожидаемое состояние. При корректировании весов используются численные методы.

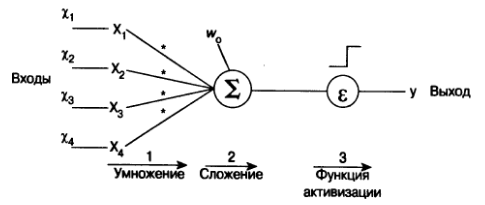


Рисунок 2.13 – Модель нейрона

Процесс расчета состояния нейрона – его выходного значения – состоит из двух этапов: объединение входов с помощью комбинаторной функции и применения функции активизации. Рассмотрим их подробнее.

#### Комбинаторная функция

Входные значения умножаются на соответствующие им весовые коэффициенты и подаются на вход комбинаторной функции, объединяющей входы в одно значение. В большинстве случаев в качестве комбинаторной функции используется суммирование:

.

#### Функция активизации

Функция активизации преобразует объединенное входное значение к выходному. Ее можно считать решающим устройством, определяющим состояние нейрона: активный или не активный. Они могут иметь разный вид, в зависимости от решаемых задач (рисунок Рисунок 2.14).

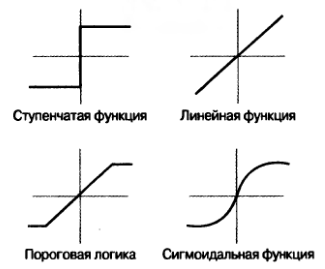


Рисунок 2.14 – Примеры функций активизации

При выборе функции следует учитывать следующие свойства:

* Дифференцируемость, непрерывность и монотонность. Чтобы можно было использовать некоторые численные методы при корректировании весов, функция активизации должна иметь известную легко вычислимую производную, которая определена и больше нуля во всех точках.
* Сложность. Комбинирование нескольких линейных функций приводит к получению другой линейной функции. Поэтому чтобы можно было решать сложные задачи с помощью нейронных сетей, функция активизации должна быть нелинейной.
* Ограниченность. Такое свойство гарантирует, что результаты вычисления функции и ее производной будут конечными.
* Область значений. Функции обычно принимают значения в интервале [0;1] или [-1;1]. Функции могут быть бинарными, т.е. принимать только два дискретных значения, но в этом случае теряется чувствительность к изменению весовых коэффициентов и уменьшается качество обучения.

Как правило используется сигмоидальная (логистическая) функция, так как она удовлетворяет всем перечисленным выше свойствам:

,

где β – коэффициент, определяющий плавность кривой. При больших значениях β функция стремится к ступенчатой, при малых – к пороговой.

### Перцептрон

Перцептрон (персептрон, perceptron) – это структура, состоящая из одного или нескольких нейронов. Одиночный нейрон является частным случаем перцептрона с одним выходом. В общем случае перцептрон – это группа параллельных нейронов, у которых общий набор входов, но разный набор весовых коэффициентов и свой отдельный выход. Нейроны не зависят друг от друга, их состояния не используются при вычислении состояний соседей, поэтому работа перцептрона не отличается от работы нескольких отдельных нейронов, использующих одинаковые входы.

### Многослойный перцептрон

В многослойных перцептронах используются более сложные топологии, состоящие из нескольких слоев нейронов. По своей структуре многослойный перцептрон фактически является несколькими однослойными перцептронами соединенными последовательно: выходы предыдущего являются входами следующего. Как правило они состоят из одного входного слоя, одного выходного и нескольких скрытых (рисунок Рисунок 2.15).

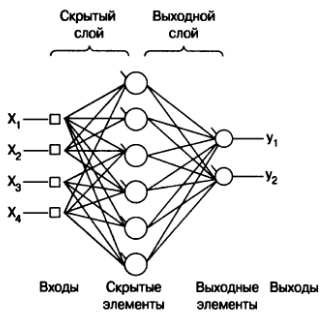


Рисунок 2.15 – Многослойный перцептрон

Многослойные перцептроны позволяют аппроксимировать функции более сложных форм. Количество скрытых слоев может быть сколь угодно большим. Добавление новых слоев может упростить обучение и получить аппроксимацию более сложных функций, однако увеличение количества слоев приводит к увеличению количества вычислений и потребляемой памяти. Теоретически один скрытый слой позволяет получить модель любой непрерывной функции [Шампандар]. Перцептроны с двумя скрытыми слоями позволяют аппроксимировать любую функцию, в том числе не являющейся непрерывной.

Количество нейронов во входном слое равно количеству входов, в выходном – количеству выходов. Веса входов нейронов в первом слое всегда равны единице. Количество нейронов в скрытых слоях нельзя рассчитать по каким-то характеристикам задачи и как правило подбирается экспериментально. В многослойных перцептронах скрытые элементы используются для внутреннего представления информации о задаче. Увеличение количества нейронов может понадобиться, если аппроксимируемая функция имеет сложную форму: зубчатую, острую, с большим количеством разрывов, выбросов и др.

В классическом многослойном перцептроне все выходы предыдущего слоя соединены со всеми входами следующего. В зависимости от задач некоторые связи могут отсутствовать или могут присутствовать связи пропускающие слои. Выходы некоторых нейронов могут быть соединены со входами предыдущих слоев. Сети с такими соединениями называются рекуррентными. Они позволяют решать некоторые сложные задачи, но являются менее стабильными – результат их обучения сложнее предсказать и контролировать.

При расчете выходных значений многослойного перцептрона слои проходятся по очереди. Сначала входные значения передаются без изменений во входной слой, так как его веса равны единице. Их значения подаются на вход первого скрытого слоя и рассчитываются его выходы. Затем аналогично рассчитываются все следующие слои включая выходной. Такой расчет называется прямой передачей (feedforward).

### Обучение перцептрона

Обучение перцептрона является обучением с учителем (supervised learning). Подготавливается обучающее множество: набор входных данных и известный выходной результат для этих данных. Задачей обучения является получение обобщающей функции, которая сможет дать результат для наборов входных данных, не входящих в исходное множество [Millingon].

Записи из обучающего множества подаются на вход сети. Выходной результат сравнивается с ожидаемым, и веса входов нейронов пересчитываются с использованием численных методов, например, градиентного спуска. Рассмотрим упрощенный вариант пересчета весов:

,

где j – индекс нейрона, oi – выход предыдущего нейрона на i-ом входе j-ого нейрона, wij – вес входа между нейронами i и j, η – параметр, характеризующий скорость обучения, δj – функция ошибки. Согласно этому выражению веса входов текущего рассматриваемого нейрона пересчитываются, если ожидаемый выход отличается от полученного. Для выходных нейронов функция ошибки имеет вид:

,

где tj – ожидаемый выход согласно записи из обучающего множества. Для скрытых нейронов ожидаемое выходное значение не известно, поэтому оно должно рассчитываться из взвешенной суммы ошибок нейронов следующего слоя:

,

где k – индекс следующего нейрона, который использует выход этого нейрона. Таким образом оценивается вклад этого нейрона в ошибки следующих.

Используя эти выражения можно оценить влияние каждого нейрона в итоговую ошибку и пересчитать его вес. Такой метод называется методом обратного распространения ошибки (backpropagation). Если вес увеличивается, то связи между нейронами укрепляются – эти входы начинают сильнее влиять на выход. Иначе связи ослабевают, их вклад в выходное значение уменьшается.

η – это параметр, характеризующий скорость обучения. Если он мал, то веса при обучении меняются слабо и требуется большое количество обучающих выборок, чтобы веса значительно изменились. Если велик, то скорость обучения увеличивается, но каждая новая выборка может сильно изменить веса, нивелировав результаты предыдущих этапов обучения. В зависимости от задачи этот параметр может регулироваться. Например, его можно уменьшать с ходом обучения, чтобы в большей степени учитывать историю обучения и в меньшей – новые корректирующие записи.

### Обучение с ослабленным контролем

Обучение с ослабленным контролем (weakly supervised learning) – это альтернативный вариант обучения. В данном случае нет явного обучающего множества с известными ожидаемыми результатами работы. Вместо этого задается функция, оценивающая результаты работы нейронной сети. Например, в играх в качестве оценивающей функции может использоваться количество получаемых очков или численная мера игрового преимущества над противником. Если действия сети оцениваются положительно, то связи, которые привели к этому решению укрепляются.

Данный подход позволяет получить агента, решающего поставленную задачу, без использования обучающего множества. Одна полученное решение зачастую не пригодно для использования в видеоиграх, так как в результате получается агент, использующий всегда одну и ту же тактику, которая дает максимальное преимущество. Такое поведение часто неестественно и не реалистично.

### Применение нейронных сетей в компьютерных играх

Нейронные сети редко используются в компьютерных играх для решения задач искусственного интеллекта, так как зачастую либо существуют более простые способы для решения тех же задач, либо сама задача достаточна сложна и использование нейронных сетей приводит к дополнительным трудностям. Один из минусов нейронных сетей – это трудность их обучения. Необходимо подобрать качественное обучающее множество, что требует большого количества данных, которые не всегда доступны. После обучения полученное поведение сети сложно скорректировать, если оно отличается от желаемого. Для этого либо нужно знать, как веса скрытых нейронов влияют на выход и вручную их изменить, либо изменить обучающее множество и произвести переобучение сети. В первом случае поиск нужных связей может занять много времени. Во втором случае из-за переобучения теряются уже достигнутые результаты, поэтому это решение сложно назвать корректированием сети.

Тем не менее, существуют примеры использования нейронных сетей в компьютерных играх.

#### DeepMind и Starcraft 2

4 ноября 2016 года Google и Blizzard объявили о сотрудничестве: использовании платформы DeepMind для игры в Starcraft 2 [DeepMind and Blizzard to release StarCraft II as an AI research environment]. Данная платформа использует только ту информацию, которую может получить обычный игрок. Она использует распознавание образов, чтобы анализировать положение объектов и принимать решения. Для того, чтобы полноценно играть в Starcraft 2, искусственный интеллект на основе нейронных сетей, наблюдая за игрой живых игроков, должен научиться пользоваться мышью, камерой, управлять игровыми сущностями и ресурсами, принимать тактические решения. Данная работа является исследованием возможностей машинного обучения и не может рассматриваться, как разработка интеллектуальных агентов для коммерческого использования в видеоиграх, так как требует больших вычислительных мощностей, большого количества времени и не имеет предсказуемого результата.

#### BiCNet и Starcraft

Нейронная сеть BiCNet (multiagent bidirectionally-coordinated network), разработанная Alibaba Group совместно с группой исследователей Университетского колледжа Лондона, используется для управления группой агентов [Alibaba Group 2017]. В качестве тестового сценария они использовали управление отрядом в игре Starcraft. Для обучения сети использовалось обучение без учителя. Сеть сначала научилась базовому управлению отрядами, а затем продвинутым тактическим маневрам.

#### Борис и Блицкриг 3

23 февраля 2017 года Nival анонсировали нейросетевой ИИ для стратегической игры Блицкриг 3 [Nival анонсирует первый в мире нейросетевой искусственный интеллект для RTS в игре Блицкриг 3]. Созданный ИИ получил имя Борис. Он использует только открытую информацию об игре, каждые несколько секунд анализирует игровую сессию и на основе полученных данных предсказывает поведение противника и принимает соответствующие решения.

Борис является гибридным ИИ [в Nival рассказали DTF о создании Бориса]. Он использует нейронную сеть для получения информации, ее анализа и для предсказания положения войск оппонента. Для управления игровыми сущностями применяются другие методы. Для каждой игровой карты обучается отдельная нейронная сеть. Все вычисления во время игры производятся на серверах компании.

#### Supreme Commander 2

В стратегической игре Supreme Commander 2 нейронные сети используются для принятия тактических решений при управлении боевыми отрядами [Neural Networks in Supreme Commander 2]. Используется 4 сети: для наземных отрядов, для морских отрядов, для истребителей и для бомбардировщиков. Все сети имеют общую структуру, но обучаются отдельно.

Сеть содержит один скрытый слой. Входной слой содержит 34 нейрона, скрытый – 98 нейронов, выходной – 15 нейронов. К входным параметрам относятся свойства отрядов: суммарное количество очков здоровья, общее количество очков защиты энергетическими щитами, скорость восстановления и ремонта отрядов, среднее значение повреждений в секунду на ближней, средней и дальней дистанции, типы отрядов и другие. К выходным относятся возможные решения: атаковать ближайший отряд противника, слабейший отряд, отряд с максимальной ценностью, отряд с максимальной дальностью атаки, инженерный отряд, атака по ресурсам, побег и другие.

Во время обучения для оценки качества решения используется функция приспособленности (fitness function). Она определяет, получил ли агент игровое преимущество. Под преимуществом имеется в виду не только уничтожение произвольных отрядов (снижение суммарного количества очков здоровья), но и другие изменения параметров, характеризующих снижение угрозы, исходящей от противника (уменьшение количества урона в секунду, повреждение щитов и другое).

Полученный искусственный интеллект решал поставленную задачу, но при его разработке появлялись проблемы, на отладку и исправление которых уходили недели. Как правило, исправления заключались в изменении функции приспособленности. Каждое переобучение нейронной сети занимало один час времени на само обучение и некоторое время на тестирование результата.

### Преимущества и недостатки нейронных сетей

Нейронные сети позволяют получить обобщенные классификаторы по некоторым обучающим наборам данных. Т.е. по известным наборам входных параметров (обычно это знания агента об окружающей среде) с известным решением можно получить систему, которая сможет аппроксимировать решения на остальные не рассмотренные наборы входных данных. Описание всех возможных ситуаций с использованием других методов принятия решения, например, конечных автоматов, деревьев поведения или правил нечеткой логики могло бы потребовать большого количества времени и памяти.

За счет аппроксимации агенты, основанные на нейронных сетях, могут проявлять поведение, которое в них изначально не закладывалось. В некоторых случаях это добавляет реалистичности и разнообразия, в некоторых – затрудняет разработку, так как агент ведет себя не так, как от него ожидается.

Нейронные сети могут быть использованы для аппроксимации функций, вычисление которых в реальном времени могло бы потребовать много времени. В этом случае результат вычисления этой функции для определенных аргументов может быть использован как обучающая запись. Полученный классификатор даст приближенные значения, которые можно будет вычислить в реальном времени во время обработки игрового кадра.

К недостаткам нейронных сетей, критичным для разработки компьютерных игр, можно отнести трудность их настройки и корректирования поведения после обучения. Чтобы изменить что-то после того, как закончено обучение, нужно добавлять обучающие записи и проводить переобучение или дополнительное обучение. В этом случае новые данные могут изменить связи между нейронами так, что будет потерян уже достигнутый результат. В коммерческой разработке довольно часто требуется вносить изменения в правила игры или в поведение ИИ, поэтому нейронные сети и другие методы машинного обучения применяются довольно редко. Хотя уже появляются решения, позволяющие сохранить при переобучении связи, которые оказались полезны в прошлом [Overcoming catastrophic forgetting in neural networks], эти решения еще не распространены широко, поэтому проблема до конца не решена.

Классификаторы на основе нейронных сетей всегда дают какое-то решение, но оно не всегда верное, и при этом сеть не объясняет, как было получено это решение. Из-за этого такие системы сложно отлаживать.

# Общие компоненты, не зависящие от метода принятия решений

Согласно модели агента, изображенной на рисунке Рисунок 1.1, помимо решающего устройства он содержит датчики для получения информации об окружающей среде и исполнительные механизмы (актуаторы) для воздействия на среду. В случае ботов в компьютерных играх нет необходимости в оцифровке окружающего мира с помощью датчиков, т.к. вся необходимая информация уже хранится в оперативной памяти. Однако, эту информацию необходимо преобразовать к виду, удобному для использования в алгоритмах принятия решений и выделить из нее данные, релевантные с точки зрения правил рассматриваемой игры. Для игры, правила которой описаны в пункте 1.5, к таким данным можно отнести количество очков здоровья всех персонажей, их координаты, и планировку уровня. В рамках игры персонажи могут совершать определенные простые действия под управлением игроков: двигаться в заданном направлении, смотреть в заданную точку (целиться) и стрелять. Т.к. бот по определению – это компьютерная программа, использующая интерфейс, предназначенный для человека, персонажи-агенты должны использовать эти же действия.

Рассматриваемые методы и алгоритмы не зависят от используемых технологий и языка программирования. В данной работе используется игровой движок Unity. Это один из самых популярных игровых движков {добавить ссылку на статистику}, позволяющий быстро создавать прототипы и полноценные игры. В качестве языка игровой логики в нем используются C# и javascript. В данной работе будет использоваться C#, так как он обладает более удобной объектной моделью и поддерживает сильную типизацию, которая позволяет уменьшить количество ошибок в ходе разработки.

## Интерфейс управления персонажем

Для управления персонажем используется следующий интерфейс:



Компонент, реализующий этот интерфейс обеспечивает правильное изменение позиции и ориентации трехмерной модели персонажа в пространстве. Детали реализации зависят от используемого игрового движка и не связаны с алгоритмами принятия решений, поэтому в данной работе не рассматриваются. Компонент может принимать команды как от компонента, отвечающего за управление с помощью устройств ввода, так и от компонента, отвечающего за искусственный интеллект.

## Движение

Для управления движением агента используется отдельный компонент. Он принимает на вход точку, в которую должен идти агент, ищет оптимальный путь до нее, используя алгоритмы поиска на графах, и вызывает метод Controllable.Move(Vector3 direction), чтобы направить персонажа к следующей точке найденного пути. Блок-схема алгоритма работы компонента показана на рисунке Рисунок 3.1.

Для поиска пути используется представление местности в виде навигационной сети (NavMesh). Построение сетки и поиск пути осуществляются с помощью инструментов игрового движка и в данной работе не рассматриваются. Пример навигационной сетки, построенной с помощью этих инструментов, показан на рисунке Рисунок 3.2.



Рисунок 3.1 – Блок-схема алгоритма работы компонента Navigation

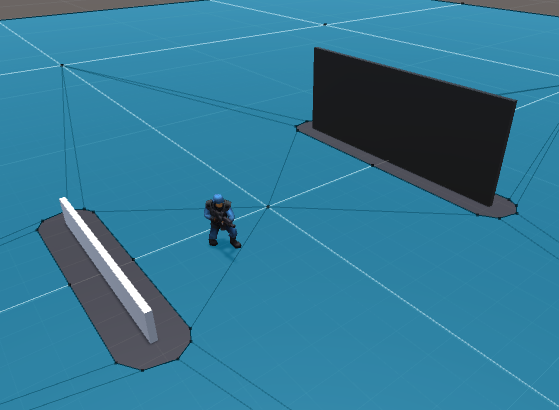


Рисунок 3.2 – Пример навигационной сетки для поиска пути и обхода препятствий

## Прицеливание и стрельба

Все персонажи используют одно и то же оружие. Оно обладает довольно большим разбросом, поэтому точность прицеливания влияет на общее количество попаданий меньше, чем расстояние до цели. Простейшая реализация прицеливания – это наведение на цель с упреждением, зависящим от скорости и направление движения цели, а также скорости полета снаряда. При расчете движение цели принимается за прямолинейное. Блок-схема алгоритма работы компонента показана на рисунке Рисунок 3.3.



Рисунок 3.3 – Блок-схема алгоритма работы компонента Shooting

На рисунке Рисунок 3.4 показана схема для расчета упреждения. – скорость цели, – скорость снараяда, d – расстояние между стрелком и целью, α – угол между вектором движения цели и вектором от стрелка до цели. Параметры цели, стрелка и снаряда известны. Нужно найти время t, за которое цели и снаряд проходят расстояния до точки встречи. По теореме косинусов:

,

.

Получаем квадратное уравнение. Если цель неподвижна, то оно имеет одно решение: t = 0. Если цель движется, получаем два решения. Берем то, что больше нуля, умножаем на скорость цели и получаем требуемое упреждение относительно исходной позиции.



Рисунок 3.4 – Расчет упреждения

## Состояние персонажа

Каждый игровой персонаж обладает рядом параметров, которые в рамках рассматриваемой игры являются открытой информацией и могут быть использованы для принятия решений. Значения параметров обновляются на каждом шаге игровой симуляции. Описание параметров представлено в таблице Таблица 3.1.

Таблица 3.1

Описание параметров персонажей

|  |  |  |
| --- | --- | --- |
| Тип данных | Название | Описание |
| Side | Side | Перечисление из двух элементов: Blue и Red. Используется в качестве индекса персонажа, чтобы различать их при обновлении графического интерфейса. Сами персонажи имеют соответствующие цвета: синий и красный. |
| float | health | Количество оставшихся очков здоровья. Уменьшается при каждом попадании. |
| bool | isDead | Если количество очков здоровья падает до нуля или ниже, персонаж считается проигравшим. |
| float | speed | Максимальная скорость движения. |
| Vector3 | position | Координаты персонажа в пространстве. |
| Vector3 | movementDirection | Вектор направления движения. |
| bool | isInCover | Флаг, указывающий, находится ли персонаж в укрытии. |
| bool | isNearCover | Флаг, указывающий, находится ли персонаж рядом с укрытием. |
| bool | isEnemyVisible | Флаг, указывающий, находится ли персонаж-противник в поле зрения. |
| float | distanceToEnemy | Расстояние до противника. |
| Vector3 | lastEnemyPosition | Координаты, в которых противник был замечен последний раз. Если противник находится в поле зрения, то этот параметр равен его текущим координатам. |
| CharacterState | enemyState | Ссылка на аналогичную информацию о противнике. |

## Анализ местности

Помимо информации о текущем состоянии персонажей, агентам для принятия решений требуется информация об окружающей местности (terrain reasoning). К этой информации можно отнести архитектуру игрового уровня, тактически выгодные позиции, укрытия, области, закрытые от обзора противника и другое. Для этого вся местность разбивается на сетку точек (waypoints) с некоторым фиксированным шагом [Terrain Reasoning GDC 2001, AI For Generated Worlds GDC 2016, Ming-Lun "Allen" Chou 2016]. Шаг выбирается так, чтобы получить достаточно высокую точность и достаточно высокую скорость расчетов.

В [Terrain Reasoning GDC 2001] рассматривается пример с выбором оптимальной позиции для снайпера (рисунок Рисунок 3.5). Для этого каждая точка оценивается по ряду критериев: обзор из данной точки, защищенность, освещенность, может ли противник незаметно подобраться к этой точке и другие. Все оценки взвешиваются и суммируются.

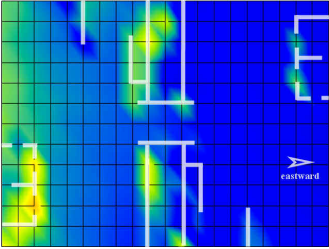


Рисунок 3.5 – Карта оценок позиций для снайпера (чем ярче, тем лучше)

В [AI For Generated Worlds GDC 2016] предлагается использовать несколько карт, в каждой из которых хранятся определенные параметры точек (рисунки Рисунок 3.6, Рисунок 3.7 и Рисунок 3.8). В качестве информации используются расстояния от точки до ключевых объектов: выхода, стен, воды или определенной территории. Для расчета используется алгоритм Дейкстры – осуществляется поиск пути от точки до ближайшего объекта заданного типа. Результатом является длина найденного пути.



Рисунок 3.6 – Карта расстояний до стен (чем ярче, тем ближе)



Рисунок 3.7 – Карта расстояний до выхода (чем ярче, тем ближе)



Рисунок 3.8 – Карта расстояний до воды (чем ярче, тем ближе)

В [Ming-Lun "Allen" Chou 2016] точки расставляются по сетке и могут добавляться и корректироваться вручную, чтобы дополнительно обозначить точки интереса для ботов (рисунок Рисунок 3.9).

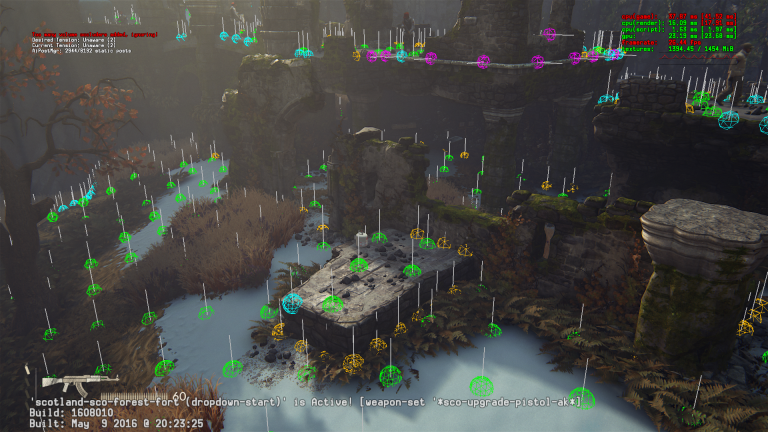


Рисунок 3.9. Точки интереса для ботов, расставленные по сетке и вручную.

Информация, которую нужно обрабатывать для принятия решений, зависит от правил игры. На рисунке Рисунок 3.10 изображена арена из игры, рассматриваемой в данной работе. Черным цветом окрашены стены, они блокируют обзор и все выстрелы. Белым цветом окрашены укрытия, они блокируют половину выстрелов, если стрелок стоит дальше определенного расстояния от него. Если стрелок стоит близко к укрытию, он получает полное преимущество – блокируются только выстрелы противника, собственные выстрелы игнорируют укрытие. Игроки изначально расположены в противоположных углах арены. В других углах расположены аптечки – объекты, восстанавливающие очки здоровья персонажей.

В таких условиях наиболее полезной информацией являются данные об укрытиях и расстояниях до них. Описание параметров, рассчитываемых для каждой точки, представлено в таблице Таблица 3.2. Алгоритм работы компонента, обновляющего значения параметров, изображен на рисунке Рисунок 3.11.

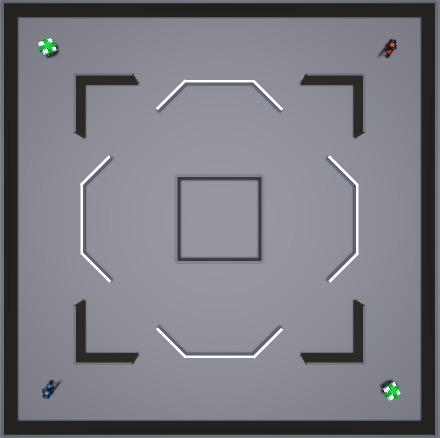


Рисунок 3.10 – Арена, на которой происходят действия игры

Таблица 3.2

Описание параметров, используемых для анализа местности

|  |  |  |
| --- | --- | --- |
| Тип данных | Название | Описание |
| Vector3 | position | Координаты точки |
| float | movementDistanceToAgent | Длина кратчайшего пути от агента до точки |
| float | movementDistanceToEnemy | Длина кратчайшего пути от последней известной позиции противника до точки |
| float | directDistanceToAgent | Эвклидово расстояние от точки до агента |
| float | directDistanceToEnemy | Эвклидово расстояние от точки до последней известной позиции противника |
| bool | isBehindCover | Есть ли укрытие между точкой и последней известной позицией противника. Учитываются только те укрытия, которые будут блокировать выстрелы от противника, т.е. он стоит на некотором расстоянии от них. |
| bool | isInCover | Находится ли точка рядом с укрытием так, что агент, находясь в этой точке будет получать полное преимущество от укрытия. |
| bool | isBehindWall | Есть ли стена между точкой и последней известной позицией противника. |
| bool | isHealthPack | Есть ли в данной точке аптечка. |

Для принятия решения параметрам нужно присвоить веса и просуммировать их. В зависимости от стратегии агента перечисленные параметры могут иметь разную ценность для него. Например, если агент отступает, точки, закрытые степной от противника, будут иметь большую ценность, т.к. позволят скрыться от него. В случае атаки наоборот – агенту нужно держать противника в поле зрения.



Рисунок 3.11 – Блок-схема алгоритма работы компонента TerrainReasoning

На рисунках Рисунок 3.12 и Рисунок 3.13 изображены карты оценок позиций красного игрока для двух разных положений синего игрока при определенной стратегии. Веса отмечены шкалой цвета: от зеленого (минимум) до красного (максимум). Зеленым цветом отмечены точки с прямой видимостью и без укрытий. Они имеют минимальный вес, т.к. находиться в них наиболее опасно для агента. Желтым цветом отмечены точки, в которых видимость заблокирована стенами. Точки отмечены оранжевым цветом, если между ними и противником есть укрытия. Максимальный вес имеют точки в тех местах, где агент получает полное преимущество от укрытий. На рисунке Рисунок 3.14 изображена карта расстояний от агента до точек.

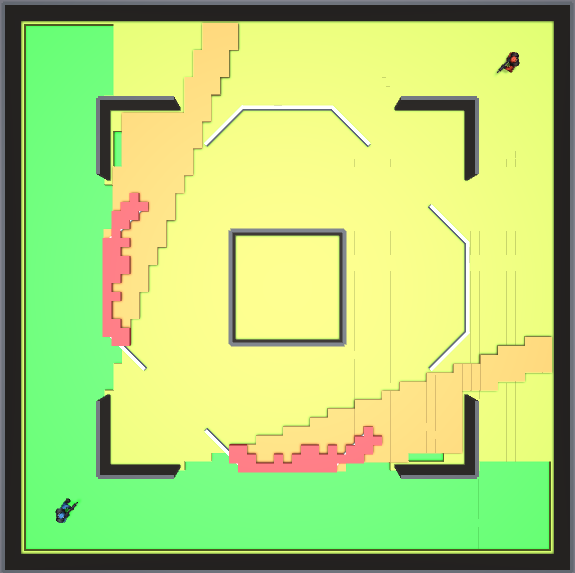


Рисунок 3.12 – Пример карты оценки позиций для исходного положения игроков

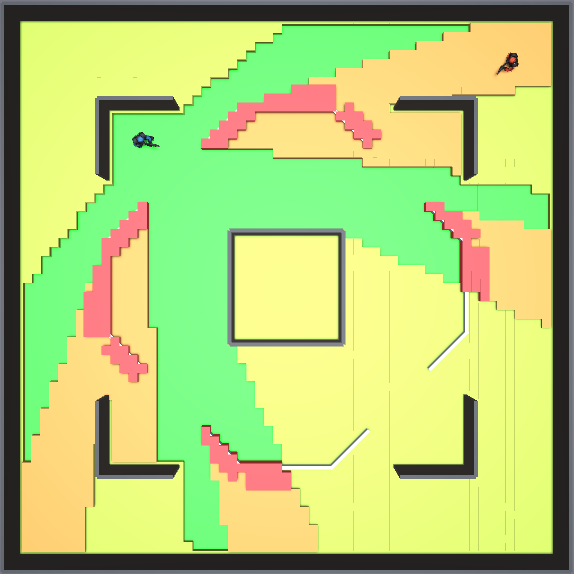


Рисунок 3.13 – Пример карты оценки позиций

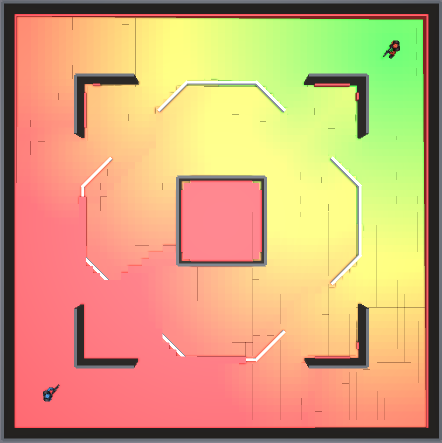


Рисунок 3.14 – Карта расстояний от агента до точек

# Конечные автоматы

Конечные автоматы – это один из простых и популярных методов принятия решений. При проектировании системы, принимающей решения, необходимо определить конечное множество состояний, в которых может находиться система, и множество переходов между состояниями. В рассматриваемой игре агент должен переключаться между состояниями в зависимости от изменения окружающей среды, своего состояния или состояния противника. К таким состояниям можно отнести поиск противника, состояния атаки, состояние защиты и отступление.

Конечные автоматы могут быть реализованы разными способами. Один из популярных вариантов – шаблон проектирования «Состояние» (State). С его помощью можно создать конечный автомат, объединяющий в себе свойства распознающих конечных автоматов, порождающих автоматов Мили и порождающих автоматов Мура. Рассмотрим его реализацию, адаптированную для использования в игровом цикле. Каждое состояние – это реализация отдельного класса, наследника абстрактного класса State.



Объект конечного автомата хранит все состояния, проверяет условия переходов для текущего и осуществляет переходы, если условия выполнены. Блок-схема алгоритма работы автомата изображена на рисунке Рисунок 4.1.



Рисунок 4.1 – Блок-схема работы алгоритма работы класса, реализующего переключение состояний

## Реализация поведения агента

### Поиск противника

В начале игроки расположены в противоположных углах арены и не видят друг друга. Поэтому первой стратегией является поиск противника. В данном состоянии активируется случайное блуждание по карте, пока не будет встречен противник. Если противник обнаружен, автомат переходит в состояние атаки.

### Атака

В этом состоянии агент агрессивно атакует противника, стараясь поддерживать ближнюю или среднюю дистанцию и жертвуя защитой, чтобы нанести больше урона. Для анализа местности выбирается весовая функция, позволяющая выбрать позицию для атаки. Точки имеют больший вес, если из них видно противника, если они позволяют получить полное преимущество от укрытия и если они находятся на определенном расстоянии от противника. На рисунке Рисунок 4.2 показан пример карты весов в состоянии атаки. Агент выбирает точки в определенном радиусе от себя с весом больше порогового. Затем из этих точек берется случайная и устанавливается в качестве точки следования. При таком подходе агент ведет себя более хаотично и постоянно находится в движении, по сравнению с случаем, когда выбирается точка с максимальным весом.

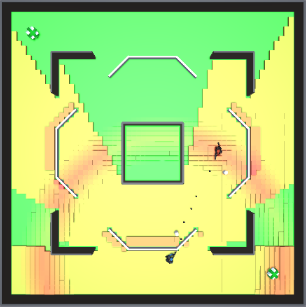


Рисунок 4.2 – Карта весов точек для состояния атаки (зеленый – минимум, красный – максимум)

Если количество очков здоровья агента уменьшается меньше порогового значения, он переходит в состояние защиты. Если цель пропала из вида, агент возвращается в состояние поиска противника.

### Защита

В этом состоянии агент сражается менее агрессивно и выбирает позицию так, чтобы минимизировать получаемый урон. Для этого весовая функция выбирается таким образом, чтобы повысить вес точек за укрытиями и на большом расстоянии от противника. На рисунке Рисунок 4.3 показан пример карты весов в состоянии защиты. Точка для следования выбирается аналогично с состоянием атаки, чтобы агент всегда находился в движении и по нему было сложнее попасть.

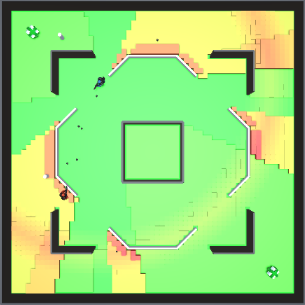


Рисунок 4.3 – Карта весов точек для состояния защиты (зеленый – минимум, красный – максимум)

Если количество очков здоровья агента уменьшается ниже порогового значения, агент начинает отступать и искать аптечки. Это же происходит, если противник пропал из поля зрения. Вместо поиска противника в этом случае лучше попытаться восстановиться. Других переходов из состояния защиты нет, т.к. либо агент переходит в состояние отступления, либо побеждает и игра заканчивается.

### Отступление и поиск аптечек

В этом состоянии весовая функция выбирается так, чтобы больший вес имели точки за стенами. Наибольший вес имеют точки, в которых расположены аптечки. На вес этих точек также влияет расстояние от агента и расстояние от противника, чтобы аптечки имели больший вес, если они ближе к агенту, и меньший вес, если на пути к ним находится противник. На рисунке Рисунок 4.4 показан пример карты весов в состоянии защиты.



Рисунок 4.4 – Карта весов точек для состояния отступления (зеленый – минимум, красный – максимум)

В качестве точки следования выбирается точка с максимальным весом. Каждую секунду проверка производится заново, т.к. ситуация может измениться. Например, противник переместится так, что первая цель станет менее выгодной, либо аптечки в точке следования уже не будет. Если на карте больше нет аптечек или очки здоровья были восстановлены, то агент переходит в состояние поиска противника.

## Вывод

На рисунке Рисунок 4.5 в виде графа изображен итоговый конечный автомат для интеллектуального агента в рассматриваемой игре.



Рисунок 4.5 – Представление конечного автомата в виде графа

В рассмотренных состояниях и переходах не учитывается состояние противника. Например, если у него мало очков здоровья, то вместо защиты или отступления можно было бы продолжить атаку, чтобы его добить. Так же не учитывается, находится ли противник в укрытии. В этом случае он имеет тактическое преимущество, и агрессивная атака против него оказывается менее выгодной.

Если учитывать дополнительные факторы или изменять набор состояний, потребуется вносить изменения в уже существующие компоненты. Например, если добавить новую переменную, влияющую на принятие решений, то нужно будет проверять все существующие состояния и переходы, и изменить их, если эта переменная на них влияет. Добавление нового состояния так же требует проверки уже существующих объектов. При этом если новое состояние не сильно отличается от уже существующих, это может привести к дублированию кода и трудностям в сопровождении. Например, если несколько состояний отвечают за различные стили атаки в зависимости от разных условий среды, они могут иметь большое количество общих черт и отличаться деталями. Это можно сделать за счет настройки поведения в рамках одного состояния, но в этом случае уже будет использоваться смешанная архитектура, т.к. в рамках одной только модели конечных автоматов эта проблема решается сложно.

Иерархические конечные автоматы позволяют упростить проектирование, но не решают перечисленные выше проблемы. В нечетких конечных автоматах общее состояние системы является взвешенной суммой всех возможных состояний. С помощью нечеткой логики можно регулировать степень включения того или иного состояния. Например, таким образом можно получить разнообразные тактики за счет комбинирования состояний атаки и защиты. Однако при большом количестве состояний и переменных нечеткие конечные автоматы обладают теми же проблемами, что и классические. Вероятностные конечные автоматы решают другую задачу и не применимы для рассматриваемых ботов.

# Деревья поведения

Деревья поведения в отличие от конечных автоматов не учитывают, в каком состоянии агент находится в настоящий момент времени, а вместо этого каждый раз проверяют условия среды для того, чтобы выбрать подходящую ветвь. Они представляют собой форму записи сценариев поведения агента в разных условиях.

## Описание элементов деревьев поведения

Среди узлов дерева можно выделить внутренние узлы, которые как правило одинаковы для всех деревьев и не зависят от решаемой задачи (последовательности, селекторы и декораторы), и терминальные узлы, которые проектируются специально под конкретного агента или некоторую его абстракцию. Каждый из узлов после выполнения возвращает одно из трех значений: Success, Failure или Running. Среди терминальных узлов можно выделить узлы, которые используются для проверки условий и узлы-действия. Первые используют коды результата Success и Failure в качестве аналогов логических true и false, а вторые – чтобы показать, удалось ли выполнить какое-либо действие.

С помощью селекторов и последовательностей используя эти элементы, можно составлять нелинейные сценарии, поддерживающие ветвление и логические функции. Рассмотрим их подробнее.

#### Ветвление

На рисунке Рисунок 5.1 показано дерево поведения, имитирующее инструкцию ветвления «если … то … иначе». Если условие возвращает Success, то выполняется следующее действие в последовательности. Если Failure, то вся последовательность возвращает Failure и селектор переходит к следующему дочернему элементу



Рисунок 5.1 – Дерево поведения, имитирующее инструкцию ветвления «если … то … иначе»

#### Логическое И

На рисунке Рисунок 5.2 показано дерево поведения, имитирующее логическое И. Последовательность дойдет до выполнения действия только если все условия вернут Success.



Рисунок 5.2 – Дерево поведения, имитирующее логическое И

#### Логическое ИЛИ

На рисунке Рисунок 5.3 показано дерево поведения, имитирующее логическое ИЛИ. Если все условия в селекторе вернут Failure, то селектор также вернет Failure и последовательность не дойдет до выполнения действия.



Рисунок 5.3 – Дерево поведения, имитирующее логическое ИЛИ

#### Логическое НЕ

На рисунке Рисунок 5.4 показано дерево поведения, имитирующее логическое НЕ. Стандартный декоратор «Инвертор» меняет Success на Failure и наоборот. Если дочерний элемент возвращает Running, то инвертор так же возвращает Running.



Рисунок 5.4 – Дерево поведения, имитирующее логическое НЕ

## Реализация поведения агента

Используя рассмотренные логические конструкции построим дерево, реализующее поведение агента аналогичное реализованному ранее с использованием конечных автоматов. Поведение в каждом состоянии можно определить одной отдельной ветвью: в корне поддерева находится узел «последовательность», первые элементы которой определяют условия нахождения в данном состоянии, а остальные – действия, выполняемые в этом состоянии. На рисунке Рисунок 5.5 показан общий вид такого дерева без детализации отдельных ветвей. Селектор последовательно проверяет все дочерние элементы, пока один из них не вернет Success, т.е. не будет выбрана ветвь, подходящая для текущих условий окружающей среды. На рисунках Рисунок 5.6-Рисунок 5.9 отдельные ветви изображены более подробно. Ветвь для отступления выбирается в последнюю очередь, если условия для всех остальных ветвей не были выполнены, поэтому в ней самой не нужно проверять эти условия.



Рисунок 5.5 – Общий вид дерева поведения для рассматриваемого агента



Рисунок 5.6 – Поддерево, определяющее поведение в состоянии атаки



Рисунок 5.7 – Поддерево, определяющее поведение в состоянии защиты



Рисунок 5.8 – Поддерево, определяющее поведение в состоянии патрулирования и поиска противника



Рисунок 5.9 – Поддерево, определяющее поведение в состоянии отступления и поиска аптечек

Из рисунков видно, что ветви похожи друг на друга. Можно изменить дерево, чтобы уменьшить дублирование элементов. Можно выделить три ветви: стрельба, установка весовой функции и параметром для анализа местности и движение (рисунок Рисунок 5.10).



Рисунок 5.10 – Общий вид измененного дерева поведения для рассматриваемого агента

Теперь в корне дерева расположена последовательность, а не селектор, так как работа дерева свелась в трем параллельным деятельностям агента вместо выбора одной модели поведения. Рассмотрим подробнее ветви дерева (рисунки Рисунок 5.11, Рисунок 5.12 и Рисунок 5.13).



Рисунок 5.11 – Ветвь дерева, отвечающая за управление стрельбой



Рисунок 5.12 – Ветвь дерева, отвечающая за установку параметров анализа местности



Рисунок 5.13 – Ветвь дерева, отвечающая за движение

Управление стрельбой сводится к простому ветвлению: если противник виден, то стрелять, иначе нет. Вторая ветвь содержит проверки условий, которые ранее отвечали за выбор состояния. В зависимости от параметров агента и окружающей среды устанавливаются разные весовые функции для анализа местности. Третья ветвь проверяет несколько условий и если какое-либо из них выполняется, то выбирается новая точка для движения.

## Вывод

Деревья поведения предоставляют инструмент для создания нелинейных сценариев поведения. Каждый терминальный узел – это либо получение информации с датчиков, либо передача действия исполнительному механизму. Структура дерева и его внутренние элементы (последовательности, селекторы и декораторы) формируют решающее устройство.

Основным преимуществом деревьев поведения является модульность. Каждый элемент не зависит от других, из элементов можно собирать поддеревья, которые можно повторно использовать в разных местах. Благодаря этому в с помощью деревьев поведения можно определить более сложное поведение агента, чем при использовании конечных автоматов. Однако, по сравнению с конечными автоматами деревья обладают более низкой производительностью. На каждом шаге симуляции нужно обойти практически все дерево, начиная с корня, в то время как в конечных автоматах нужно проверить только исходящие переходы для текущего состояния.

# Нечеткая логика

В нечеткой логике переменные имеют значения от 0 до 1, соответствующие разной степени истинности. В отличие от конечных автоматов и деревьев поведения при использовании нечеткой логики и нечетких множеств агент может находиться в нескольких состояниях одновременно с разной степенью вхождения. Так же нечеткая логика может быть использована для аппроксимации функций, вычисляющих выходные управляющие воздействия на основе информации об окружающей среде.

Алгоритм работы агента, использующего нечеткую логику, представлен на рисунке Рисунок 6.1.



Рисунок 6.1 – Алгоритм работы агента, основанного на нечеткой логике

## Реализация поведения агента

### Входные переменные

От количества входных лингвистических переменных и количества термов в них зависит количество правил вывода, которые необходимо определить, чтобы полностью описать реакции на все возможные состояния окружающей среды. Количество правил вывода определяется по формуле:

N = , где mi – количество термов в переменной xi,

где n – количество переменных, mi – количество термов в i-ой переменной. Поэтому нужно выделить небольшое количество переменных, важных для принятия решений, чтобы избежать комбинаторного взрыва количества правил вывода. Рассмотрим подробнее входные переменные.

#### Количество очков здоровья агента

HPа = { Мало, Средне, Много }

Определяет выбор стратегии: атака, защита или отступление.

#### Количество очков здоровья противника

HPп = { Мало, Средне, Много }

Так же влияет на выбор стратегии, так как иногда предпочтительным решением является продолжение атаки вместо отступления или наоборот, прекращение атаки, если противник имеет преимущество. Термы HPа и HPп аналогичны и изображены на рисунке Рисунок 6.2.



Рисунок 6.2 – Графическое представление термов переменных HPа и HPп

#### Видимость противника

Видимость противника = { Есть, Нет }

Переменная содержит два дискретных терма и имитирует обычную логическую переменную. Термы представлены двумя синглтонами – функциями принадлежности, которые принимают значение 1 в одной точке и 0 в остальных (рисунок Рисунок 6.3).



Рисунок 6.3 – Графическое представление термов переменной «Видимость противника»

### Выходные переменные

Количество выходных переменных не влияет на количество правил вывода, поэтому его можно не ограничивать. Выходные переменные определяют параметры, влияющие на управляющее воздействие, и их численное значение. В рассматриваемой игре в качестве таких параметров используются коэффициенты весовой функции при анализе местности и параметры выбора точки движения из множества оценок. Рассмотрим эти переменные.

#### Дистанция до противника

Дистанция до противника = { Близкая, Средняя, Дальняя }

Определяет стратегию агента. При агрессивной стратегии агент старается сблизиться с противником, чтобы нанести больше урона, но при этом и увеличивается риск получить урон самому. При защитной стратегии, наоборот, агент стремиться увеличить дистанцию. Графическое представление термов изображено на рисунке Рисунок 6.4.



Рисунок 6.4 – Графическое представление термов переменной «Дистанция до противника»

#### Вес дистанции до противника

Вес дистанции до противника = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, находящихся на оптимальной дистанции от противника.

#### Вес точек в укрытии

#### Вес точек в укрытии = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, в которых агент получает полное преимущество от укрытия, т.е. получает защиту, но укрытие не блокирует его собственные выстрелы.

#### Вес точек за укрытием

#### Вес точек за укрытием = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, которые находятся за укрытием относительно противника, при этом укрытие одинаково блокирует как выстрелы противника, так и агента, если он будет находиться в этой точке.

#### Вес точек за стенами

#### Вес точек за стенами = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, которые находятся за стеной относительно противника, т.е. из этой точки нет прямой видимости противника.

#### Вес точек, в которых находятся аптечки

#### Вес точек, в которых находятся аптечки = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, в которых расположены аптечки, восстанавливающие очки здоровья.

#### Вес точек с прямой видимостью

#### Вес точек с прямой видимостью = { Низкий, Средний, Высокий }

При анализе местности определяет вес точек, таких что между ними и противником нет укрытий и стен. Термы для всех параметров, использующихся в весовой функции, определены одинаково и изображены на рисунке Рисунок 6.5.

#### Радиус поиска точки для движения

#### Радиус поиска точки для движения = { Малый, Средний, Большой }

Определяет радиус, в пределах которого выбирается новая точка для движения. В боевых ситуациях радиус уменьшается, чтобы агент чаще менял направление движения и вел себя менее предсказуемо. В случае отступления или поиска противника радиус увеличивается. Графическое представление термов переменной изображено на рисунке Рисунок 6.6.



Рисунок 6.5 – Графическое представление термов параметров для расчета весов



Рисунок 6.6 – Графическое представление термов переменной «Радиус поиска точки для движения»

### Правила вывода

Для полного описания всех реакций для определенных ранее входных переменных требуется 18 правил выбора. Это число можно уменьшить, если использовать метод Комбса или объединять правила, в которых разные значения одной переменной приводят к одинаковым результатам. Правила вывода в рассматриваемом случае не являются аддитивно разделяемыми, поэтому метод Комбса приведет к ухудшению принятых решений из-за потери информации о взаимном влиянии переменных.

Все правила вывода имеют одинаковый вид:

ЕСЛИ HPа = … И HPп = … И Видимость противника = …

#### ТО Дистанция до противника = …, Вес дистанции до противника = …, …

Поэтому удобнее представить их в виде таблицы Таблица 6.1. Прочерком отмечены переменные, которые не влияют на выходные переменные. Такая запись эквивалентна дизъюнкции всех термов этой переменной. Если переменная определена в соответствии с рекомендациями, дизъюнкция всегда в результате будет давать 1, поэтому такое условие можно не проверять.

Таблица 6.1

Правила вывода

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Входные переменные | | | Выходные переменные | | | | | | | |
| HPa | HPп | Видимость | Дистанция | Вес дистанции | Вес в укрытии | Вес за укрытием | Вес за стеной | Вес аптечек | Вес прямой видимости | Радиус поиска |
| Много | Много | Есть | Средняя | Средний | Низкий | Низкий | Низкий | Низкий | Высокий | Малый |
| Много | Средне | Есть | Близкая | Высокий | Низкий | Низкий | Низкий | Низкий | Высокий | Малый |
| Много | Мало | Есть | Близкая | Высокий | Низкий | Низкий | Низкий | Низкий | Высокий | Малый |
| Средне | Много | Есть | Дальняя | Средний | Высокий | Средний | Средний | Средний | Средний | Малый |
| Средне | Средне | Есть | Средняя | Средний | Средний | Средний | Низкий | Средний | Средний | Малый |
| Средне | Мало | Есть | Близкая | Высокий | Средний | Средний | Низкий | Средний | Низкий | Малый |
| Мало | Много | Есть | Дальняя | Высокий | Высокий | Высокий | Высокий | Высокий | Низкий | Большой |
| Мало | Средне | Есть | Дальняя | Средний | Средний | Средний | Высокий | Высокий | Низкий | Большой |
| Мало | Мало | Есть | Средняя | Средний | Высокий | Высокий | Высокий | Высокий | Низкий | Большой |
| Много | - | Нет | Близкая | Низкий | Низкий | Низкий | Низкий | Низкий | Низкий | Большой |
| Средне | - | Нет | Средняя | Низкий | Низкий | Низкий | Низкий | Средний | Низкий | Большой |
| Мало | - | Нет | Дальняя | Низкий | Низкий | Низкий | Низкий | Высокий | Низкий | Большой |

Описанные в таблице правила вывода были сформулированы на основе наблюдений над игрой человека и уже реализованных ботов. После реализации часть правил были изменены, чтобы скорректировать поведение агента.

## Вывод

Нечеткая логика позволяет реализовать агента с плавными изменениями управляющих воздействий и отсутствием фиксированных состояний. Все правила вывода определяются на основе опытов, наблюдений и экспертного мнения. Главным недостатком этого метода является большое количество правил вывода. Для трех рассмотренных входных переменных получилось 12 правил после оптимизации. Если добавить еще одну переменную с тремя термами, то правил будет 36. Если еще одну – 108. Так как правила определяются вручную, существует большая вероятность допустить ошибку при их определении: либо неправильно определить реакцию агента, либо ошибиться при написании самого правила. Значительно уменьшить количество правил можно с помощью метода Комбса, но он приводит к ухудшению качества принятия решений, так как в большинстве случае правила вывода не являются аддитивно разделяемыми.

При использовании нечеткой логики требуется большое количество ручной работы: выбор входных и выходных переменных, подбор функций принадлежности для их термов, выбор реализации операций на множествами, формулирование правил вывода и другое. Многие из этих вещей необходимо итерационно менять, корректируя поведения агента. При этом зачастую сложно предсказать, как именно изменение повлияет на результат.

# Нейронные сети

Искусственная нейронная сеть – это обучаемый классификатор. Для каждой группы входных значений она определяет класс, к которому они принадлежат. Для рассматриваемой игры входными параметрами являются переменные, характеризующие окружающую среду, а результирующим классом – принятое решение. Принятие решения в данном случае – это выбор стратегии, наиболее подходящей для текущей ситуации.

## Реализация поведения агента

Набор входных переменных аналогичен случаю агента, использующего нечеткую логику: очки здоровья агента, очки здоровья противника и видимость противника. Все переменные определены на подмножестве [0; 1] чисел с плавающей точкой: очки здоровья нормированы на 100, а видимость принимает значение 1, если противник виден, и 0, если не виден. Набор стратегий аналогичен случаю агента, использующего конечный автомат: атака, защита, поиск противника и поиск аптечек.

Нейронные сети редко используются в видеоиграх, но из тех, что используются, лучше всего себя показывают многослойные перцептроны [Millington, Robbins GDC 2012], поэтому рассмотрим эту архитектуру для реализации агента. Входной слой содержит три нейрона (по количеству входов), выходной слой содержит четыре нейрона (по количеству выходов). Остальные параметры могут варьироваться и влиять на качество обучения. Рассмотрим их подробнее.

#### Количество слоев

Рассмотрим двухслойный перцептрон (без скрытого слоя) и трехслойный перцептрон (со скрытым слоем). Перцептроны с большим количеством слоев рассматривать нецелесообразно, учитывая количество переменных и сложность задачи.

#### Количество нейронов в скрытом слое

Так как нет однозначного способа рассчитать необходимый размер скрытого слоя [Шампандар], данный параметр необходимо определить экспериментально.

#### Коэффициент крутизны функции активации

В качестве функции активации используется сигмоида:

,

где β – коэффициент, определяющий ее крутизну.

#### Коэффициент, определяющий скорость обучения

Параметр η в выражении , определяет скорость обучения, т.е. как сильно изменяются веса при обратном распространении ошибки. Большие значения приводят к ускорению обучения, но при этом новые записи переписывают уже накопленную информацию.

#### Количество повторов обучения

Для укрепления связей между нейронами обучающая выборка может быть использована несколько раз подряд.

Обучающее множество представляет собой таблицу с данными (приложение X). Она содержит 200 строк: для всех комбинаций очков здоровья агента и противника от 10 до 100 с шагом 10 и двух значений видимости заданы ожидаемые выходные значения. Среди этих 200 записей 40 соответствуют стратегии атаки, 13 – защиты, 30 – поиску противника и 117 – отступлению и поиску аптечек. Все множество разделяется на два подмножества одинакового размера. Одно используется непосредственно для обучения, а второе для проверки полученного результата. Для проверки качества обучения используется процент записей в проверочном множестве, для которых ожидаемый результат совпал с фактическим.

Обучение нейронной сети можно представить в виде функции y = f(x1, x2, .., xn), где x1, x2, .., xn – это параметры, влияющие на качество обучения, а y – это параметр, определяющий качество обучения. Используя численные методы, можно найти максимум этой функции и оптимальный набор параметров, при котором качество обучения максимально. Рассмотрим отдельно двухслойный и трехслойный перцептроны, так как у них функции имеют разное количество параметров.

#### Двухслойный перцептрон

Для двухслойного перцептрона функция имеет три аргумента: коэффициент функции активации β, коэффициент скорости обучения η и количество повторов обучения. Предварительные исследования показали, что при изменении количества повторов обучения при прочих равных параметрах качество обучения увеличивается, пока не достигнет насыщения. Поэтому имеет смысл зафиксировать набор значений этого параметра и использовать численные методы для поиска двух других аргументов.

Так как нет никакой информации о форме функции, не применимы методы, использующие производные, для ускорения поиска. Одним из методов, не требующих информации о функции, является метод координатного спуска. При его использовании все аргументы кроме одного фиксируются на некоторых значениях, и производится поиск максимума функции от одной переменной. После этого фиксируется полученное значения аргумента, выбирается следующий аргумент и ищется максимум для него. Так повторяется для всех аргументов, пока прирост значения функции в найденной точке оптимума не станет ниже порогового. Пример координатного спуска для функции от двух переменных изображен на рисунке Рисунок 7.1. Для поиска максимума функции от одного аргумента используется метод золотого сечения. Он так же не требует информации о функции и сходится быстрее, чем метод бисекций. На отрезке поиска выбираются две точки, делящие отрезок в пропорциях золотого сечения. В зависимости от значений функции ф этих точках, отрезок поиска отсекается слева или справа. Поиск заканчивается, когда расстояние между точками станет меньше порогового. Пример метода золотого сечения изображен на рисунке Рисунок 7.2.

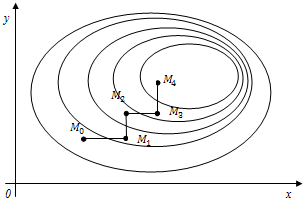


Рисунок 7.1 – Метод координатного спуска



Рисунок 7.2 – Метод золотого сечения

Методом ручного перебора была найдена начальная точка: β = 1.2, η = 0.2. Границы поиска: β ∊ [0.01; 2], η ∊ [0.01; 2]. На этих интервалах функция гладкая и предположительно имеет один экстремум. Найдем оптимальные значения параметров обучения для разных значений количества повторов обучения. Результаты представлены в таблице Таблица 7.1.

Таблица 7.1

Оптимальные параметры обучения двухслойного перцептрона

|  |  |  |  |
| --- | --- | --- | --- |
| Кол-во повторов обучения | β | η | Качество обучения |
| 1 | 1.999995 | 1.999995 | 0.19 |
| 2 | 1.999995 | 1.999995 | 0.59 |
| 3 | 1.38102 | 1.999995 | 0.34 |
| 4 | 1.999995 | 1.999995 | 0.59 |
| 5 | 1.127771 | 0.2175936 | 0.84 |
| 7 | 1.607472 | 0.1269975 | 0.87 |
| 11 | 1.346763 | 1.1647742 | 0.88 |
| 15 | 1.200001 | 0.2072002 | 0.89 |
| 21 | 0.4977484 | 0.342717 | 0.9 |
| 51 | 1.200001 | 0.2087743 | 0.92 |
| 101 | 1.200006 | 0.3711202 | 0.93 |
| 201 | 1.200001 | 1.076998 | 0.93 |
| 501 | 1.999995 | 1.999995 | 0.94 |
| 1001 | 1.999995 | 1.999995 | 0.94 |
| 2001 | 1.999995 | 1.999995 | 0.95 |

#### Трехслойный перцептрон

В отличие от двухслойного перцептрона у трехслойного появляется еще один параметр N – количество нейронов в скрытом слое. Начальное значение N выбрано равным 5. Поиск производится в интервале N ∊ [3; 20], т.к. скрытый слой должен быть по крайней мере не меньше входного слоя и может в несколько раз его превышать его по размеру. Методы поиска экстремума аналогичны прошлому случаю. Результаты представлены в таблице Таблица 7.2.

Таблица 7.2

Оптимальные параметры обучения трехслойного перцептрона

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во повторов обучения | β | η | N | Качество обучения |
| 1 | 1.999995 | 1.999995 | 20 | 0.19 |
| 2 | 1.999995 | 1.999995 | 20 | 0.59 |
| 3 | 1.999995 | 1.999995 | 20 | 0.19 |
| 4 | 1.999995 | 1.999995 | 20 | 0.59 |
| 5 | 1.983816 | 1.998192 | 20 | 0.19 |
| 7 | 1.949064 | 1.993158 | 20 | 0.19 |
| 11 | 1.954211 | 1.818467 | 20 | 0.19 |
| 15 | 1.792787 | 1.865935 | 20 | 0.19 |
| 21 | 1.999995 | 1.891457 | 20 | 0.59 |
| 51 | 1.9688 | 1.530543 | 20 | 0.59 |
| 101 | 1.200001 | 0.3649505 | 5 | 0.78 |
| 201 | 1.002577 | 0.3448411 | 5 | 0.81 |
| 501 | 1.200013 | 0.4668832 | 5 | 0.83 |
| 1001 | 0.6843097 | 0.7512037 | 13 | 0.83 |
| 2001 | 1.545125 | 0.4927048 | 8 | 0.84 |

На рисунке Рисунок 7.3 показаны графики зависимости максимального качества обучения от количества повторов обучения для двух- и трехслойного перцептронов. Значения качества обучения 0.19, 0.34 и 0.59 соответствуют случаям, когда нейронная сеть для всех входов возвращала один и тот же результат, который совпадает с соответствующим количеством записей в проверочном множестве, а значит достоверно не показывают качество обучения. По этой же причине в некоторых случаях найденные значения параметров равны максимуму интервала, на котором производился поиск – из-за того, что функция возвращала всегда одно и то же значение, у нее нет экстремума и метод золотого сечения просто перебирал точки слева на право, пока не доходил до конца интервала. Для двухслойного перцептрона значения качества обучения, отличные от перечисленных ранее, появляются начиная с 5 повторов обучения, а для трехслойного – со 101. При этом для любого количества повторов качество обучения двухслойного перцептрона выше.

Таким образом, исследование показало, что двухслойный перцептрон лучше подходит для решения рассматриваемой задачи: он имеет более высокое качество обучения при меньших количествах повторов обучения. Для дальнейшей работы взят двухслойный перцептрон со следующими параметрами обучения: кол-во повторов = 201, β = 1.200001, η = 1.076998. Дальнейшее увеличение количества повторов незначительно увеличивает качество обучения, но при этом увеличивается время обучения.

Рисунок 7.3 – Графики качества обучения для двух- и трехслойного перцептронов

## Вывод

Нейронные сети – это универсальный аппроксиматор. По некоторому набору известных значений входов и выходов можно получить сеть, которая аппроксимирует все промежуточные значения. В отличие от конечных автоматов и деревьев поведения, где все поведение агента задается разработчиком, с помощью методов, основанных на машинном обучении, можно получить поведение агента, которое изначально в него не закладывалось. С одной стороны, это делает его более реалистичным, а с другой – затрудняет отладку и проектирование из-за того, что не всегда можно предсказать реакцию агента на некоторые события.

Нейронные сети не позволяют вносить ручные корректировки поведения. Для того, чтобы что-то добавить или изменить, необходимо сформировать новое обучающее множество и заново провести обучение. При этом агент может потерять уже отлаженные свойства.

С помощью нейронных сетей сложно получить агента, имеющего определенные реакции на какие-то частные ситуации и редкие наборы входных переменных. Такие ситуации представлены небольшим количеством записей в обучающем множестве, поэтому оказывают меньшее влияние на обучение.

# Сравнение методов и алгоритмов принятия решений

Полученные реализации агентов необходимо сравнить по нескольким критериям. Для этого агенты поочередно сражаются друг с другом на арене, изображенной на рисунке Рисунок 3.10. Ограничение на одну игру – 1 минута. Если к этому времени никто из агентов не одержал победу, игра считается закончившейся в ничью. Во время игр собираются метрики, которые затем будут использованы для анализа и сравнения агентов. Метрики собираются отдельно для каждого агента и в конце игры записываются в базу данных, т.е. получается две записи на одну игру. Рассмотрим их подробнее.

#### Идентификатор игры

Используется для того, чтобы однозначно определить игру, к которой относится запись.

#### Тип контроллера агента

Информация о реализации агента. Может принимать значения: FiniteStateMachine, BehaviorTree, FuzzyLogic и NeuralNetwork.

#### Проиграл ли агент в этой игре

Агент считается проигравшим, если количество очков здоровья достигло нуля. Возможны ситуации, когда оба агента оказываются проигравшими (из-за особенностей баллистики снарядов) или никто из агентов не проиграл (вышло время).

#### Количество нанесенного урона

#### Количество полученного урона

Параметры, косвенно характеризующие поведение агента. По ним можно определить, предпочитает ли агент агрессивное или защитное поведение.

#### Взята ли аптечка

Параметр, косвенно характеризующий, насколько успешно агент реализовал стратегию отступления и поиска аптечек.

#### Среднее время принятия решения

Значение среднего времени, которое требуется агенту, чтобы произвести все расчеты для принятия решения.

Для хранения и обработки записей используется СУБД SQLite, т.к. это легковесная СУБД, не требующая установки и разворачивания сервиса на рабочей машине. SQL-запросы, использующиеся для работы с данными приведены в приложении X. В таблицах Таблица 8.1 и Таблица 8.2 показаны обработанные результаты для 4800 записей (2400 игр, 1200 игр для каждого контроллера). На рисунках Рисунок 8.1-Рисунок 8.4 изображены диаграммы этих данных.

Таблица 8.1

Основные метрики для оценки качества агентов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип контроллера | Победы, % | Поражения, % | Ничьи (время вышло, оба живы), % | Ничьи (оба проиграли), % |
| Конечные автоматы | 41.42±4.27 | 40.75±4.26 | 0.67±0.71 | 17.17±3.27 |
| Деревья поведения | 43.75±4.3 | 37.83±4.2 | 0.67±0.71 | 17.75±3.31 |
| Нечеткая логика | 40.5±4.25 | 42.33±4.28 | 0.17±0.35 | 17±3.25 |
| Нейронные сети | 38.58±4.22 | 43.33±4.29 | 0.67±0.71 | 17.42±3.29 |

Таблица 8.2

Дополнительные метрики для оценки качества агентов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип контроллера | Среднее время принятия решения, мкс | Среднее количество нанесенного урона | Среднее количество полученного урона | Доля игр, в которых была взята аптечка, % |
| Конечные автоматы | 6.47±0.34 | 95.21±2.87 | 87.92±2.4 | 10.5±2.66 |
| Деревья поведения | 16.01±0.6 | 95.77±2.75 | 94.4±3.09 | 24.7±3.73 |
| Нечеткая логика | 670.57±3.11 | 90.18±2.7 | 94.82±2.59 | 18.9±3.39 |
| Нейронные сети | 14.34±0.67 | 92.61±2.68 | 96.63±2.85 | 20.3±3.49 |

Рисунок 8.1 – Диаграмма долей побед, поражений и ничьих

Рисунок 8.2 – Диаграмма времени принятия решения

Рисунок 8.3 – Диаграмма нанесенного и полученного урона

Рисунок 8.4 – Диаграмма доли игр, в которых была взята аптечка

Из таблиц и рисунков видно, что агент, использующий контроллер на деревьях поведения, имеет максимум побед и минимум поражений. Отсюда следует, что деревья поведения – это наиболее подходящий метод принятия решения для рассматриваемой задачи, если в качестве критерия сравнения использовать соревновательные свойства агента.

Из диаграммы времени принятия решения видно, что конечные автоматы, деревья поведения и нейронные сети имеют один порядок значений, а контроллер, основанный на нечеткой логике, на два порядка медленней. Это вязано с тем, что для расчетов в нечеткой логике требуется большое количество вычислений: фаззификация значений, нечеткий вывод и дефаззификация. Самым вычислительно сложным процессом является деффазификация с использованием метода центра масс, т.к. она использует численное интегрирование функция принадлежности для каждой выходной переменной.

Из диаграмм, изображенных на рисунках Рисунок 8.3-Рисунок 8.4, можно сделать выводы о качестве принимаемых решений с точки зрения правил игры. Конечные автоматы и деревья поведения имеют большее количество нанесенного урона, чем нечеткая логика и нейронные сети. Конечные автоматы имеют наименьшее количество полученного урона. Из этого можно сделать вывод, что атака более эффективно реализована у агентов на конечных автоматах и деревьях поведения, а агенты на нечеткой логике и нейронных сетях принимали менее эффективные решения и не использовали подходящие возможности для нанесения максимального урона и не достаточно эффективно использовали укрытия. При этом агент, основанный на конечных автоматах, имеет меньше игр, в которых он взял аптечку. Это может быть объяснено либо более агрессивным поведением в целом (агент реже переходил в режим отступления), либо тем, что агент получал меньше урона и меньше было необходимость в поиске аптечек.

# Заключение

В ходе данной работы был произведен обзор следующих методов и алгоритмов принятия решений: конечные автоматы, деревья поведения, нечеткая логика и нейронные сети. Были реализованы агенты, использующие данные методы и алгоритмы.

#### 1 Конечные автоматы

Было установлено, что с ростом количества состояний, переходов и учитываемых переменных увеличиваются трудозатраты, т.к. требуется поддерживать большое количество связей между компонентами. В связи с этим автомат содержит 4 состояния, реализующих различные стратегии агента: поиск противника, атака, защита и отступление, 7 переходов между ними и проверяет 2 переменных окружающей среды: количество очков здоровья агента и видимость противника.

#### 2 Деревья поведения

Были рассмотрены две реализации агента. В основе первой – дерево, ветви которого реализуют отдельные стратегии агента, а само дерево используется для выбора подходящей ветви. В основе второй – дерево, ветви которого реализуют отдельные компоненты агента, а само дерево последовательно их обходит. Для дальнейшего исследования была выбрана вторая реализация в связи с тем, что для нее требуется меньше элементов. Полученное дерево содержит три ветви: управление стрельбой, управление движение, анализ местности для выбора выгодной позиции.

3 Нечеткая логика

В отличие от конечных автоматов и деревьев поведения, где агент осуществлял выбор стратегии, в данном случае нет стратегий, а процесс принятия решений используется для определения степени воздействия на среду. Для того, чтобы реализовать требуемое поведение агента, были определены 3 входных и 8 выходных переменных. При этом для того, чтобы полностью описать все состояния среды, требуется 18 правил вывода, но за счет объединения правил, не зависящих от некоторых входных переменных, удалось сократить их количество до 12.

4 Нейронные сети

Для реализации агента была выбрана архитектура «многослойный перцептрон», т.к. она широко используется в рассматриваемой предметной области и является универсальным классификатором. В качестве переменных входного слоя используются параметры окружающей среды, в качестве выходных – веса возможных стратегий агента. Для обучения агента использовалось множество из 200 записей, из них 100 – непосредственно для обучения, 100 – для проверки качества обучения. При этом под качеством подразумевается процент записей проверочного множества, ожидаемый выход которых совпал с фактическим. С помощью численных методов координатного спуска и золотого сечения была найдена оптимальная конфигурация сети, обеспечивающая максимальное качество обучения при фиксированном количестве повторов обучения: количество повторов = 201, 2 слоя, β = 1.200001, η = 1.076998.

Было произведено сравнение данных методов путем проведения игр реализованных ботов друг против друга. Всего было проведено 2400 игр, в ходе которых снимались и записывались в базу данных различные метрики. Анализ данных метрик производился с помощью SQL и MATLAB. Было установлено, что для рассматриваемой задачи наиболее эффективным алгоритмом являются деревья поведения. Данный алгоритм имеет максимальный процент побед (43.75%), минимальный процент поражений (37.83%), максимальное значение среднего наносимого урона (95.8 HP) и максимальный процент игр, в которых была взята аптечка (24.7%).

# Список использованных источников

Рассел С., Норвиг П. Искусственный интеллект: современный подход, 2-е изд. : Пер. с англ. – М.: Издательский дом «Вильямс», 2006.

Шампандар Алекс Дж. Искусственный интеллект в компьютерных играх. Как обучить виртуальные персонажи реагировать на внешние воздействия. : Пер. с англ. – М.: Издательский дом «Вильямс», 2007.

Сирл Д. Разум мозга — компьютерная программа? – В МИРЕ НАУКИ. (Scientific American. Издание на русском языке). 1990. № 3

Кормен T., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 3-е издание. –М.: ООО «И.Д. Вильямс», 2013. – 1328 с. : ил.

Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы; Пер. с англ. под ред. А. Шеня. –М.: МЦНМО, 2014. – 320 с.

Amit Patel. Amit’s A\* Pages / Stanford University – URL: <http://theory.stanford.edu/~amitp/GameProgramming/index.html> – (дата обращения: 10.01.2016).

Buckland M. Programming game AI by example. – Wordware Publiching Inc. 2005.

Hagelbäck J. Johansson S. J. Using Multi-agent Potential Fields in Real-time Strategy Games. International Conference on Autonomous Agents and Multi-agent Systems (AAMAS), 2008.

Zadeh L. A. Fuzzy sets. / Information and control 8. 1965.

Батыршин И. З., Недосекин А. О. Нечеткие гибридные системы. Теория и практика. / Под ред. Н.Г. Ярушкиной. – М.: ФИЗМАТЛИТ, 2007.

Штовба С. Д. Введение в теорию нечетких множеств и нечеткую логику [Электронный ресурс] / http://matlab.exponenta.ru/fuzzylogic/book1/1.php, дата обращения: 20.08.2016.

Тенетко М. И., Пескова О. Ю. Проведение эксперимента по обоснованию выбора нечёткой импликации, пригодной для решения задач классификации рисков и выработки наилучших рекомендаций по рискам. – Известия Южного федерального университета. Технические науки, выпуск № 12 (137) / том 137 / 2012.

Bai Y., Wang D. Advanced Fuzzy Logic Technologies in Industrial Applications (Advances in Industrial Control). – Springer, 2006.

Заде Л. Понятие лингвистической переменной и его применение к принятию приближенных решений. – М.: Мир, 1976.

Combs W. E. The Combs Method for rapid inference. 1997.

Ross T.J. Fuzzy logic with engineering applications. Second edition. University of New Mexico, USA. – John Wiley & Sons, Ltd, 2004.

Champandar A. J. 10 Reasons the Age of Finite State Machines is Over. December 28, 2007. [Электронный ресурс] / http://aigamedev.com/open/article/fsm-age-is-over/, дата обращения: 11.03.2017.

Hanagan T. How to Have Fun in an Asylum – AI Game Conference, Vienna. 2012.

Белоусов А. И., Ткачев С. Б. Дискретная математика. — М.: МГТУ, 2006. — 743 с.

Colledanchise M., Ögren P. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. In IEEE Transactions on Robotics vol.PP, no.99, pp.1-18. 2016.

Millington I., Funge J. Artificial intelligence for games. – CRC Press, 2009.

Isla D. Handling Complexity in the Halo 2 AI. Game Developers Conference, 2005.

Isla D. Halo 3 – building a better battle. Game Developers Conference, 2008.

Lim C. U., Baumgarten R., Colton, S. Evolving behaviour trees for the commercial game DEFCON. In Applications of Evolutionary Computation, pp. 100-110. Springer Berlin Heidelberg, 2010.

Gillberg J. AI Behavior Editing and Debugging in 'Tom Clancy's The Division'. Game Developers Conference, 2016.

Champandar A. J. Understanding Behavior Trees. September 6, 2007. [Электронный ресурс] / http://aigamedev.com/open/article/bt-overview/, дата обращения: 12.03.2017.

Orkin J. Three States and a Plan: The A.I. of F.E.A.R. Game Developers Conference, 2006.

Conway C., Higley P., Jacopin E. Goal-Oriented Action Planning: Ten Years Old and No Fear! Game Developers Conference, 2015.

Ивановский Р. И. Теория вероятностей и математическая статистика. Основы, прикладные аспекты с примерами и задачами в среде Mathcad. – СПб.: БХВ-Петербург, 2008.

Васин В. А., Калмыков В. В. Радиосистемы передачи информации: Учебное пособие для вузов. – М.: Горячая линия – Телеком, 2005.

Mark D. Behavioral Mathematics for Game AI. – Course Technology, a part of Cengage Learning, 2009.

Palacios J. Unity 5.x Game AI Programming Cookbook. – Packt Publishing, 2016.

Reynolds, C. W. Steering Behaviors For Autonomous Characters. In the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California. Pages 763-782.

William van der Sterren. Terrain Reasoning for 3D Action Games. Game Developers Conference, 2001.

Isla D., Abercrombie J. AI For Generated Worlds. Game Developers Conference, 2016.

Ming-Lun "Allen" Chou. A Brain Dump of What I Worked on for Uncharted 4 [Электронный ресурс] / <https://www.gamedev.net/resources/_/technical/game-programming/a-brain-dump-of-what-i-worked-on-for-uncharted-4-r4390/>, дата обращения: 28.05.2017.

DeepMind and Blizzard to release StarCraft II as an AI research environment [Электронный ресурс] / <https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment/>, дата обращения: 16.03.2017.

Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, Jun Wang. Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games. Alibaba Group, University College London, 2017.

Nival анонсирует первый в мире нейросетевой искусственный интеллект для RTS в игре Блицкриг 3 [Электронный ресурс] / <http://ru.nival.com/news/newsline/2017/neural-network-ai-for-rts>, дата обращения: 16.03.2017.

«Я ожидаю увидеть бота, с которым будет чрезвычайно тяжело играть»: в Nival рассказали DTF о создании Бориса [Электронный ресурс] / <https://dtf.ru/4883-ya-ozhidayu-uvidet-bota-s-kotorym-budet-chrezvychayno-tyazhelo-igrat-v-nival-rasskazali-dtf-o-sozdanii-borisa>, дата обращения: 16.03.2017.

Robbins M. Neural Networks in Supreme Commander 2. Off the Beaten Path: Non-Traditional Uses of AI. Game Developers Conference, 2012.

Kirkpatrick J., Pascanu R., Rabinowitz N. Overcoming catastrophic forgetting in neural networks. Proceedings of the National Academy of Science of the United States of America. 2017.

Holland J. H. Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, 1975.

Van Waveren J. P. The Quake III Arena Bot. – Master of Science Thesis, Delft University of Technology, 2001.

ПРИЛОЖЕНИЕ X

Множество записей для обучения нейронной сети.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| HPa | HPe | Visibility | Action | Attack | Defence | SearchEnemy | SearchHealth |
| 100 | 100 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 90 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 80 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 70 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 60 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 50 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 40 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 30 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 100 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 100 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 90 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 80 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 70 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 60 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 50 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 40 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 30 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 90 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 100 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 90 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 80 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 70 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 60 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 50 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 40 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 30 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 80 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 70 | 100 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 90 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 80 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 70 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 60 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 50 | 1 | Defence | 0 | 1 | 0 | 0 |
| 70 | 40 | 1 | Attack | 1 | 0 | 0 | 0 |
| 70 | 30 | 1 | Attack | 1 | 0 | 0 | 0 |
| 70 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 70 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 60 | 100 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 90 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 80 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 70 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 60 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 50 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 40 | 1 | Defence | 0 | 1 | 0 | 0 |
| 60 | 30 | 1 | Attack | 1 | 0 | 0 | 0 |
| 60 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 60 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 50 | 100 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 90 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 80 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 70 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 60 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 50 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 40 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 30 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 20 | 1 | Attack | 1 | 0 | 0 | 0 |
| 50 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 40 | 100 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 90 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 80 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 70 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 60 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 50 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 40 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 30 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 20 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 10 | 1 | Attack | 1 | 0 | 0 | 0 |
| 30 | 100 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 90 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 80 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 70 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 60 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 50 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 40 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 30 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 20 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 10 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 100 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 90 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 80 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 70 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 60 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 50 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 40 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 30 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 20 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 10 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 100 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 90 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 80 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 70 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 60 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 50 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 40 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 30 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 20 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 10 | 1 | SearchHealth | 0 | 0 | 0 | 1 |
| 100 | 100 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 90 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 80 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 70 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 60 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 50 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 40 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 30 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 20 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 100 | 10 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 100 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 90 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 80 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 70 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 60 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 50 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 40 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 30 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 20 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 90 | 10 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 100 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 90 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 80 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 70 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 60 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 50 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 40 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 30 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 20 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 80 | 10 | 0 | SearchEnemy | 0 | 0 | 1 | 0 |
| 70 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 70 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 60 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 50 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 40 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 30 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 20 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 100 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 90 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 80 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 70 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 60 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 50 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 40 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 30 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 20 | 0 | SearchHealth | 0 | 0 | 0 | 1 |
| 10 | 10 | 0 | SearchHealth | 0 | 0 | 0 | 1 |

ПРИЛОЖЕНИЕ X

SQL-запросы, использованные для обработки данных

#### Создание таблицы

CREATE TABLE `stats` (

`id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

`gameIndex` INTEGER NOT NULL,

`timestamp` INTEGER,

`controllerType` TEXT NOT NULL,

`dead` INTEGER NOT NULL,

`damageGiven` NUMERIC NOT NULL,

`damageTaken` NUMERIC NOT NULL,

`healthPackCollected` INTEGER NOT NULL,

`framesTime` NUMERIC NOT NULL,

`framesCount` INTEGER NOT NULL,

`averageFrameTime` NUMERIC NOT NULL )

#### Получение информации о статистике побед, поражений и ничьих

SELECT

first.controllerType,

first.dead,

second.dead,

COUNT(\*),

COUNT(\*) / (0.0 + (SELECT COUNT(\*) FROM stats this

WHERE this.controllerType = first.controllerType)) \* 100.0

FROM stats first JOIN stats second

ON first.gameIndex = second.gameIndex AND first.controllerType != second.controllerType

GROUP BY first.controllerType, first.dead, second.dead

#### Получение среднего времени

SELECT controllerType, AVG(averageframeTime) FROM stats WHERE averageframeTime > 0

GROUP BY controllerType

#### Получение среднего количества нанесенного урона, среднего количества полученного урона и доли игр, в которых была взята аптечка

SELECT

controllerType,

AVG(damageGiven),

AVG(damageTaken),

AVG(healthPackCollected) \* 100.0

FROM stats

GROUP BY controllerType