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

# 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, генетические алгоритмы 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. Модель агента.

На рисунке 1 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;
* кооперативная;
* 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;
* сформулировать задачи, которые должен решать бот;
* для каждой задачи выбрать методы и алгоритмы, подходящие для ее решения, обосновать выбор;
* реализовать выбранные алгоритмы;
* протестировать полученные реализации;
* сделать выводы об эффективности различных методов и алгоритмов для моделирования процесса принятия решений.

# Обзор методов и алгоритмов принятия решений

## Рулевое управление (steering behavior)

В [Reynolds] была предложена модель, описывающая поведение автономных персонажей и их взаимодействие друг с другом посредством векторов сил. Автономные персонажи – это агенты, использующиеся в анимации и в интерактивных системах, таких как видеоигры или симуляции. Эта модель дает описание параметров агентов и набор взаимодействий, которые можно комбинировать друг с другом для получения различных стратегий поведения. Различные комбинации стратегий позволяют моделировать движение транспортных средств, поведение охотника и жертвы, толпы людей, стаи птиц или косяка рыб.

Агент моделируется инерционной материальной точкой, которая обладает следующими параметрами:

* масса;
* позиция в пространстве;
* ориентация в пространстве;
* скорость;
* максимальная сила;
* максимальная скорость.

На каждом шаге симуляции параметры обновляются в соответствии с физическими законами:

сила = min(рулевое воздействие, максимальная сила);

ускорение = сила / масса;

скорость = min(скорость + ускорение, максимальная скорость);

позиция = позиция + скорость;

ориентация пересчитывается через вектор скорости.

Рулевое воздействие определяется стратегией движения. Рассмотрим эти стратегии.

#### Движение к точке (seek)

Стратегия, согласно которой агент старается максимально сблизиться с целью. Для того, чтобы агент двигался к точке максимально эффективно, необходимо придать следующее воздействие (рисунок 2):

воздействие = текущая скорость – желаемая скорость.



Рисунок 2. Движение к точке.

#### Бегство (flee)

Стратегия, согласно которой агент старается максимально отдалиться от цели. Идентично движению к точке, но с обратным направлением вектора желаемой скорости.

#### Прибытие (arrive)

Аналогично движению к точке, но агент начинает тормозить на некотором расстоянии от цели, чтобы при ее достижении скорость была равна нулю. Для этого задаются торможение и порог – расстояние до цели, начиная с которого скорость должна падать. Если расстояние больше порогового, то скорость рассчитывается как обычно. Если меньше, то:

скорость = min(желаемая скорость \* расстояние до цели / порог, максимальная скорость).

Коэффициент «расстояние до цели / порог» дает линейное торможение при приближении к цели.

#### Преследование (pursuit)

Стратегия, согласно которой агент старается максимально сблизиться с движущейся целью. Воздействие рассчитывается аналогично движению к точке, но целью движения является не сам преследуемый объект, а его прогнозируемая позиция в будущем с предположением, что объект движется прямо и не поворачивает. Так как в большинстве случаев предположение о прямолинейном движении не верно, появится ошибка, но за счет того, что воздействие пересчитывается на каждом шаге симуляции, она будет мала.

В качестве цели обычно используется позиция объекта через время T. Идеальным значением T является время до встречи с объектом. Однако, из-за непредсказуемого поведения объекта, эту величину сложно предсказать, поэтому используются приближенные оценки T, полученные на основании расстояния между объектами и из взаимном расположении (движутся навстречу друг другу или догоняют).

#### Преследование со смещением (offset pursuit)

Аналогично стратегии преследования, но агент старается находиться на фиксированном смещении от цели. Используется для следования за лидером и поддержания строя.

#### Уклонение (avoid)

Стратегия, согласно которой агент старается максимально отдалиться от движущейся цели. Стратегия аналогична стратегии преследования, но с обратным направлением вектора желаемой скорости. На рисунке 3 показаны маршруты преследования и уклонения.



Рисунок 3. Преследование и уклонение.

#### Скитание (wander)

Стратегия, согласно которой агент движется по местности, имитируя движение без цели. На каждом шаге симуляции к вектору скорости добавляется случайное смещение в сторону, которое плавно отклоняет агента от прежнего курса (рисунок 4).



Рисунок 4. Скитание.

#### Избегание препятствий (obstacle avoidance)

Стратегия, согласно которой агент старается избегать столкновений со статическими объектами, находящимися у него на пути. Перед агентом в направлении движения моделируется прямоугольник, ширина которого равна габаритам агента, а длина пропорциональна скорости движения. Если этот прямоугольник пересекается с препятствием, то к агенту прикладывается воздействие, отклоняющее его от этого препятствия (рисунок 5).



Рисунок 5. Избегание препятствий.

#### Разделение (separation)

Групповая стратегия, согласно которой агенты стараются держаться на расстоянии друг от друга. Если расстояние между двумя агентами превышает заданное пороговое значение, то к ним прикладывается воздействие, отводящее их друг от друга. Группа агентов, следующих такой стратегии будет распределяться по местности, пока расстояние между ними не станет больше порогового.

#### Выравнивание (alignment)

Групповая стратегия, согласно которой агенты стараются сохранять одинаковую ориентацию в пространстве. Для этого на каждом шаге симуляции вычисляется средний вектор направления агентов и принимается за желаемое направление. Вычитая из него текущее направление агента, получаем воздействие, которое нужно приложить, чтобы его повернуть.

#### Сплочение (cohesion)

Групповая стратегия, согласно которой агенты стараются быть ближе друг к другу. Для этого вычисляется воздействие, которое двигает агентов к центру масс группы.

## Алгоритмы на графах

Алгоритм поиска пути в графе широко применяется в протоколах маршрутизации при обменах данными по сети, для решения логистических задач и для ориентирования систем искусственного интеллекта в пространстве. Графы часто используются для построения модели местности, в которой агент должен принимать решения. В этом случае карта местности представляется в виде графа, часто называемого навигационной сеткой (navigation mesh, navmesh) (рисунок 6).



Рисунок 6. Пример навигационной сетки, используемой в видеоиграх.

### Алгоритм Дейкстры

Алгоритм Дейкстры – алгоритм на графах, изобретенный нидерландским ученым Эдсгером Дейкстрой в 1959 году. В общем случае находит кратчайшие пути от выбранного узла до всех остальных, результатом работы является дерево кратчайших путей [Кормен].

Рассмотрим пример работы алгоритма на графе, изображенном на рисунке 7. Найдем кратчайшие пути от вершины 1 до всех остальных. Каждой вершине сопоставляется метка со значением, равным длине пути от начальной вершины до нее. Перед началом работы алгоритма метке исходной вершины устанавливается значение ноль, а всем остальным – бесконечность или другое особое значение, обозначающее отсутствие пути (рисунок 8). В ходе работы алгоритма эти значения могут уменьшаться, если был найден более короткий путь до вершины.

На каждой итерации алгоритма вершина с минимальной меткой устанавливается в качестве текущей и помечается звездочкой, что означает, что до данной вершины найдено кратчайшее расстояние и уменьшаться оно больше не будет. В данном случае выбирается вершина 1, так как ее метка равно нулю, а остальные бесконечности. Для всех вершин, смежных с текущей, пересчитывается значение метки: новое значение равно сумме значения текущей вершины и веса ребра между этими вершинами. Если новое значение меньше текущего, то метка обновляется, а в отдельную структуру сохраняется текущая вершина.

dist[2] = min{∞, 0 + 2} = 2, prev[2] = 1

dist[7] = min{∞, 0 + 2.83} = 2.83, prev[7] = 1

Результат первой итерации представлен на рисунке 9.

|  |  |
| --- | --- |
| Граф для примера.png  Рисунок 7. Исходный взвешенный граф. | Граф для примера.png  Рисунок 8. Начальное состояние меток. |
| Граф для примера.png  Рисунок 9. Результат первой итерации. | Граф для примера.png  Рисунок 10. Результат второй итерации. |

Выбирается новая вершина с минимальным значением из тех, что еще не были посещены (т.е. не отмечены звездочкой), и пересчет меток повторяется для вершин смежных с ней. В данном примере такой вершиной является вершина 2.

dist[3] = min{∞, 2 + 2} = 4, prev[3] = 2

Результат второй итерации представлен на рисунке 10.

Обход вершин продолжается либо пока не будут посещены все вершины, либо пока не останутся вершины только со значением равным бесконечности. Второй вариант означает, что есть вершины, не достижимые из исходной. Результат обхода вершин из примера представлен на рисунке 11.

Для того, чтобы получить кратчайший путь из исходной вершины к другой, необходимо, используя структуру prev, восстановить путь, начиная с целевой вершины.



Рисунок 11. Итоговые значения меток.

Рассмотрим получение пути от вершины 1 до вершины 5:

prev[5] = 4;

prev[4] = 7;

prev[7] = 1.

Оптимальный путь из вершины 1 в вершину 5: 1-7-4-5.

Если требуется определить путь только до одной вершины, то в качестве критерия останова можно использовать достижение алгоритмом целевой вершины. При этом изменяется инициализация: нет смысла изначально помечать все вершины специальным символом или бесконечностью, так как некоторые из них могут быть не рассмотрены вовсе. Вместо этого, при пересчете вершины необходимо проверять, рассчитывается ли метка в первый раз, и инициализировать ее.

Рассмотрим псевдокод алгоритма Дейкстры [Дасгупта]:

procedure Dijkstra(graph, start, goal) {

processed = {} // просмотренные вершины

dist[start] = 0 // рассматриваемые вершины и их метки

prev[start] = null // предыдущие вершины

// обход вершин

while (dist не пусто) {

v = «извлечь из dist вершину с минимальным расстоянием от исходной вершины»

«добавить v в processed»

if (v == goal)

break // целевая вершина достигнута

for (e ∈ дуги, исходящие из v) {

v2 = e.getDestination()

if (processed содержит v2)

continue // вершина уже была посещена

d = dist[v] + e.getWeight()

if (dist не содержит v2 OR dist[v2] > d) {

dist[v2] = d

prev[v2] = v

}

}

}

}

Сложность алгоритма зависит от используемых реализаций структур данных [Кормен]. Проанализируем сложность выполнения псевдокода из раздела. Пусть V – это количество вершин графа, а E – количество ребер. Алгоритм посещает каждую вершину максимум один раз (ни разу, если она не достижима из исходной), при этом для каждой вершины алгоритм проверяет все исходящие дуги (ребра) и пересчитывает известное расстояние для смежных вершин. Можно сформулировать выражение для оценки сложности для неизвестных АТД:

TДейкстра(V, E) ~ V·(Tdist.getMin(V) + Tprocessed.set(V)) +

+ E·(2·Tdist.get(V) + Tdist.set(V) + Tprev.set(V))

Если выбрать для их реализации сбалансированные деревья, получим:

TДейкстра(V, E) ~ V·(O(log V) + O(log V)) + E·(O(log V) + O(log V) + O(log V))

~ O((V + E) · log V)

Если какой-либо из операторов имеет степень роста O(n), то по правилу умножения получаем:

TДейкстра(V, E) ~ O(V2)

Таким образом, степень роста алгоритма Дейкстры определяется самым медленным оператором используемых в нем структур данных.

### Алгоритм A\*

Алгоритм A\* (произносится «А звезда» или «А стар», от англ. A star) – алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины к другой. В 1964 году Нильс Нильсон изобрел эвристический подход к увеличению скорости алгоритма Дейкстры. Этот алгоритм был назван А1. В 1967 году Бертрам Рафаэль сделал значительные улучшения по этому алгоритму, но ему не удалось достичь оптимальности. Он назвал этот алгоритм A2. Тогда в 1968 году Питер Э. Харт представил аргументы, которые доказывали, что A2 был оптимальным при использовании последовательной эвристики лишь с незначительными изменениями. Таким образом, он обозначил новый алгоритм в синтаксисе звездочкой, он начинается на А и включает в себя все возможные номера версий.

Аналогично алгоритму Дейкстры A\* поочередно просматривает вершины и обновляет минимальное известное расстояние от исходной вершины до текущей. Отличие заключается в получении следующей вершины из очереди. Помимо информации о расстоянии от исходной вершины, A\* также использует информацию о положении целевой вершины. Каждая следующая вершина выбирается по минимуму суммы f(v) = g(v) + h(v), где g(v) – минимальное известное расстояние от исходной вершины до вершины v, h(v) – значение эвристической функции для вершины v.

Эвристика позволяет сократить количество вершин, просмотренных алгоритмом, предлагая направление для поиска, которое позволит приблизиться к цели, но при этом не гарантирует, что полученное приближение будет верным. Эвристика может использоваться для настройки и управления алгоритмом [Amit’s A\* Pages]:

* Если эвристическая функция всегда возвращает ноль, то алгоритм превращается в алгоритм Дейкстры, т.е. для выбора вершины используется только расстояние от исходной вершины.
* Если оценка всегда меньше либо равна реальной дальности от вершины до цели, то A\* гарантированно вернет оптимальный путь.
* Если эвристическая функция возвращает точное расстояние от вершины до цели, то алгоритм посетит только вершины, входящие в оптимальный путь.
* Если оценка завышена, т.е. ее значение для некоторых вершин превышает реальную длину пути, алгоритм не гарантирует, что найденный путь будет оптимальным. Однако, завышенная оценка отсеивает большое количество вершин, что позволяет увеличить быстродействие за счет потери качества.
* Если оценка много больше расстояния от исходной вершины, то поведение алгоритма близко к жадному поиску по первому наилучшему совпадению.

Эвристическая функция называется допустимой эвристической оценкой, если она не переоценивает расстояние между узлами.

Как правило, A\* используются на графах, основанных на физическом представлении, например, в прямоугольной системе координат. Используя геометрические свойства вершин графа, можно оценить минимально возможное расстояние между ними. Рассмотрим эвристические функции, которые обычно применяются с A\* [Buckland, Amit’s A\* Pages]:

* Манхэттенское расстояние. Метрика, введённая Германом Минковским. Согласно этой метрике, расстояние между двумя точками равно сумме модулей разностей их координат (рисунок 12). Также имеет название «расстояние городских кварталов». Используется, если перемещение по клеткам допускается только в четырех направлениях.

.



Рисунок 12. Манхэттенское расстояние между двумя узлами.

* Диагональное расстояние. Используется, если разрешено движение между диагональными клетками. Одна из разностей координат проходится по диагонали, другая – по прямой аналогично манхэттенскому расстоянию (рисунок 13).

где .



Рисунок 13. Диагональное расстояние между двумя узлами.

* Эвклидово расстояние. Представляет собой геометрическое расстояние между двумя точками в пространстве (рисунок 14). Гарантированно дает минимальную оценку расстояния.

.



Рисунок 14. Эвклидово расстояние между двумя узлами.

Рассмотрим псевдокод алгоритма A\*:

procedure AStar(graph, start, goal, heuristic) {

closed = {} // посещенные вершины

open = {start} // рассматриваемые вершины и их метки

prev[start] = null // структура для хранения предшествующих вершин

while (open не пусто) {

v = «извлечь из open вершину с минимальным значением f = g + h»

«добавить v в closed»

if (v == goal)

break // целевая вершина достигнута

for (e ∈ дуги, исходящие из v) {

v2 = e.getDestination()

if (closed содержит v2)

continue // вершина уже была посещена

newG = g[v] + e.getWeight()

if (open не содержит v2) {

h[v2] = heuristic(v2, goal)

g[v2] = newG

prev[v2] = v

} else {

if (g[v2] > newG) {

g[v2] = newG

prev[v2] = v

}

}

}

}

}

Временная сложность алгоритма зависит от эвристики. В худшем случае число вершин, исследуемых алгоритмом растет экспоненциально по сравнению с длиной оптимального пути: O(bd), где b – коэффициент ветвления (средняя степень вершины), d – длина оптимального пути. Сложность становится полиномиальной, если эвристическая функция удовлетворяет следующему условию:

, где – это оптимальная эвристика, то есть точное расстояние от x до цели [Рассел и Норвиг].

### Иерархический A\*

### IDA\*

### A\*mbush

### Jump Point Search

### Потенциальные поля

{добавить литературу}

Потенциальные поля – это метод, используемый для поиска пути. В этой модели различные объекты, влияющие на маршрут агентов, создают притягивающие или отталкивающие поля, идентичные электрическим. Целевая точка создает притягивающее поле, препятствия и другие агенты – отталкивающее {добавить рисунок}. При перемещении агент выбирает среди ближайших точек графа ту, где заряд максимальный, и переходит в нее {добавить рисунок}. Таким образом, данный алгоритм поиска пути является жадным и не учитывает более далекие точки и саму цель при принятии решения.

Из-за того, что данный алгоритм является жадным, поиск пути является быстрым и может быть использован, когда требуется организовать движение большого количества агентов. Построение полей в свою очередь, может оказаться вычислительно трудной задачей, так как на одной карте может быть большое количество объектов, оказывающих влияние на итоговое поле. Требуется либо использовать оптимизации, добавляя кэширование и ленивые вычисления, либо ограничивать область применения статическими полями.

Данный поиск пути подвержен проблеме локального оптимума. Если на пути движения агента находится объект с вогнутой поверхностью, может возникнуть ситуация, когда точка с большим потенциалом находится внутри этого объекта {добавить рисунок}. В этом случае агент упрется в стену и не будет иметь возможности покинуть это место. Для предотвращения таких ситуаций нужно либо избегать объектов с вогнутыми поверхностями, либо учитывать их в расчетах полей. Например, при расчете полей считать их вклад не напрямую за объектом, а с учетом огибания его поверхности. Такое решение приводит к усложнению требуемых вычислений, поэтому не всегда пригодно.

С помощью формы создаваемых полей можно управлять поведением агентов и частично объединить решения задач поиска пути и принятия тактических решений. Например, если использовать нелинейное распространение полей, можно воссоздать поведение, когда агенты подходят на расстояние стрельбы и держат дистанцию.

Разные агенты могут использовать разные поля. Например, если форма поля используется для управления поведением агента, или если из-за различной проходимости разные объекты могут либо считаться препятствиями, либо игнорироваться при поиске пути. В этом случае требуется рассчитывать и хранить несколько полей, что негативно сказывается на производительности и потребляемой памяти.

## Деревья решений

[Millington, Шампандар]

## 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 15 показаны два примера нечетких множеств: A – множество чисел, примерно равных 2, B – множество чисел, примерно равных 3.



Рисунок 15. Примеры нечетких множеств.

На рисунках 16, 17 и 18, 19 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 16. Объединение множеств с использованием нормы, определенной через максимум функций принадлежности.



Рисунок 17. Объединение множеств с использованием нормы, определенной через сумму функций принадлежности.



Рисунок 18. Пересечение множеств с использованием конормы, определенной через минимум функций принадлежности.



Рисунок 19. 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 20 изображено графическое представление переменной «угловое отклонение».



Рисунок 20. 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 21 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 21. 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 22 Для ускорения расчетов можно увеличить шаг суммирования (рисунок 23).



Рисунок 22. Пример дефаззификации с помощью центра масс.



Рисунок 23. 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.

### 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 24).



Рисунок 24. 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 25).



Рисунок 25. 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:

Рассмотрим пример нечеткого конечного автомата и вычисления весов состояний. На рисунке 26 изображен граф состояний и переходов для агента-стражника.

S = { Патрулирование, Атака, Защита };

I = { Очки здоровья (hit points, HP), Есть враги поблизости }.

Графические представления входных переменных представлены на рисунках 27 и 28. Переменная «Есть враги поблизости» представлена двумя синглтонами – функциями принадлежности, которые принимают значение 1 в одной точке и 0 во всех остальных. Нечеткая переменная, представленная двумя синглтонами идентична обычной логической переменной. Использование обычных термов и синглтонов позволяет комбинировать в одном автомате элементы нечеткой и классической логик.



Рисунок 26. Граф состояний и переходов нечеткого конечного автомата.



Рисунок 27. Лингвистическая переменная «Очки здоровья».



Рисунок 28. Лингвистическая переменная «Есть враги поблизости».

Переходы между состояниями описываются следующими правилами вывода:

ЕСЛИ Патрулирование И Нет врагов ТО Патрулирование;

ЕСЛИ Патрулирование И Обнаружен враг ТО Атака;

ЕСЛИ Атака И Обнаружен враг И Много HP ТО Атака;

ЕСЛИ Атака И Мало HP ТО Защита;

ЕСЛИ Атака И Нет врагов ТО Патрулирование;

ЕСЛИ Защита И Обнаружен враг И Мало HP ТО Защита;

ЕСЛИ Защита И Много HP ТО Атака;

ЕСЛИ Защита И Нет врагов ТО Патрулирование.

Как было сказано в разделе 2.4.5, правила должны покрывать все комбинации входных переменных. Если один из термов переменной используется в условии выхода из состояния, то и все остальные термы так же должны учитываться в этом или других переходах (включая рекуррентные). Если это условие не соблюдается, возможны ситуации, когда сумма весов всех состояний не равна 1. В худшем случае может возникнуть ситуация, когда веса всех состояний равны 0, тогда следующее вычисление любого веса так же даст 0.

Рассмотрим для примера некоторые изменения входных переменных. В качестве нормы T возьмем минимум, в качестве конормы S – максимум. В начальный момент времени:

s0 = Патрулирование; S =

HP = 100, Есть враги поблизости = Нет.

В поле зрения появился противник:

(Патрулирование) = (Патрулирование ∧ Нет врагов) ∨ (Защита ∧ Нет врагов) ∨ (Атака ∧ Нет врагов) = Нет врагов ∧ (Патрулирование ∨ Атака ∨ Защита) = 0 ∧ (1 ∨ 0 ∨ 0) = 0;

(Атака) = (Патрулирование ∧ Обнаружен враг) ∨ (Атака ∧ Обнаружен враг ∧ Много HP) ∨ (Защита ∧ Много HP) = (1 ∧ 1) ∨ (0 ∧ 1 ∧ 1) ∨ (0 ∧ 1) = 1;

(Защита) = (Атака ∧ Мало HP) ∨ (Защита ∧ Обнаружен враг ∧ Мало HP) = (0 ∧ 0) ∨ (0 ∧ 1 ∧ 0) = 0;

S =

Значение очков здоровья опустилось до 50:

(Патрулирование) = 0 ∧ (0 ∨ 1 ∨ 0) = 0;

(Атака) = (0 ∧ 1) ∨ (1 ∧ 1 ∧ 0.5) ∨ (0 ∧ 0.5) = 0.5;

(Защита) = (1 ∧ 0.5) ∨ (0 ∧ 1 ∧ 0.5) = 0.5;

S =

Значение очков здоровья опустилось до 20:

(Патрулирование) = 0 ∧ (0 ∨ 0.5 ∨ 0.5) = 0;

(Атака) = (0 ∧ 1) ∨ (0.5 ∧ 1 ∧ 0) ∨ (0.5 ∧ 0) = 0;

(Защита) = (0.5 ∧ 1) ∨ (0.5 ∧ 1 ∧ 1) = 0.5;

после нормализации: S =

Противник исчез из области видимости:

(Патрулирование) = 1 ∧ (0 ∨ 0 ∨ 1) = 1;

(Атака) = (0 ∧ 0) ∨ (0 ∧ 0 ∧ 0) ∨ (1 ∧ 0) = 0;

(Защита) = (0 ∧ 1) ∨ (1 ∧ 0 ∧ 1) = 0;

S =

В рассмотренном примере агент сначала патрулирует местность, затем обнаруживает врага и переходит в состояние атаки. Этот переход идентичен переходу в классическом конечном автомате благодаря синглтонам. Затем в ходе действий агент теряет очки здоровья и постепенно переходит в состояние защиты. Благодаря нечеткой логике агент может комбинировать состояния атаки и защиты и более тонко планировать тактические маневры. После потери противника из вида, он снова переходит к патрулированию.

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 29 изображен пример простого дерева, реализующего проход агента через дверь.



Рисунок 29. 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.

## Теория полезности

Использование элементов экономической теории полезности (utility theory) в разработке ИИ для компьютерных игр активно развивает Дейв Марк [Dave Mark]. Основная идея заключается в том, что у любого товара или услуги разделяют понятия ценность (value) и полезность (utility). Подобное разделение можно применить и к задаче приятия решений.

Ценность – это объективное свойство решения, обладающего ценой. Полезность – субъективное свойство решения, характеризующее, насколько хорошо оно может решить задачу агента, удовлетворить его потребности, помочь достичь цели. Например, один и тот же игровой ресурс в стратегической игре имеет одну и ту же цену для всех агентов, но если одному из них он позволяет получить игровое преимущество, а второму нет, то его полезность для первого будет выше.

### Предельная полезность

Предельная полезность (marginal utility) – это полезность, которую агент получает от каждого нового объекта или решения одного типа. Она отражает тот факт, что полезность решения, принятого в первый раз, отличается от полезности того же решения, если его принимать второй, третий, n-ый раз. Например, полезность каждого нового элемента одно ресурса будет уменьшаться, если у агента уже избыток этого ресурса.

Предельная полезность рассчитывается как производная от полной полезности по количеству {добавить рисунок}:

,

где MU – предельная полезность, TU – полная полезность (сумма полезности одиночных решений), n – количество объектов или решений одного типа.

Предельная полезность может убывать (вторая производная полной полезности меньше нуля) или возрастать (больше нуля). В рассмотренном ранее примере убывающая предельная полезность, так как каждый новый элемент ресурса в меньшей степени помогает получить игровое преимущество. Возрастающей предельной полезностью обладают синергичные объекты, т.е. те объекты, совместный эффект которых больше суммы их отдельных эффектов. В стратегической игре такими объектами могут быть боевые единицы, формирующие эффективные боевые построения и вместе выполняющие свои задачи лучше, чем по отдельности.

### Распределения, веса, функции, вот это все

### Принятие решений на основе полезности

Любая игровая сущность, влияющая на принятие решения, обладает полезностью. Термин «полезность» в данном случае может сбивать с толку, но он является устоявшимся переводом. Оригинальный термин «utility» имеет и другие значения: практичность, утилитарность, применимость, выгодность.

Рассмотрим упрощенный пример. Есть два агента, у каждого есть какое-то оружие и шкала, характеризующая количество очков здоровья. У первого агента есть два возможных решения: атаковать или защищаться (отступать и искать укрытие). Есть 4 фактора, влияющих на принятие решения.

* Количество очков здоровья агента. Условный параметр, характеризующий боевую способность персонажа. Если значение падает до нуля, он погибает. Данный параметр обладает убывающей предельной полезностью: если очков здоровья мало, они являются более ценными, так как любая ошибка может привести к поражению. {добавить рисунок}
* Количество очков здоровья противника. Параметр аналогичен прошлому, но для агента он имеет обратный смысл – чем этот параметр больше, тем хуже. {добавить рисунок}
* Качество оружия агента. Упрощенный параметр, характеризующий точность, урон, скорострельность, дальнобойность и другие параметры оружия. В реальных примерах все эти свойства желательно рассматривать по отдельности. {добавить рисунок}
* Качество оружия оппонента. Параметр аналогичен прошлому, но для агента он имеет обратный смысл – чем этот параметр больше, тем хуже. {добавить рисунок}

На основе рассмотренных параметров рассчитывается полезность каждого отдельного решения и выбирается то, полезность которого максимальна.

{добавить расчет}

### Преимущества и недостатки теории полезности

Теория полезности позволяет агенту проанализировать и оценить все возможные решения с точки зрения их применимости к текущей игровой ситуации и выбрать наиболее подходящее. В отличии от подходов, где поведение агента жестко задается разработчиком (конечные автоматы и деревья поведения), т.е. вручную описываются реакции на предполагаемые ситуации, в данном случае агент сам выбирает, что ему предпринять. Поведение агента становится более разнообразным, более реалистичным и менее предсказуемым.

Принятие решений на основе теории полезности легко отлаживать, так как у любого решения можно узнать рассчитанное значение полезности и определить, как оно было получено.

Для численной оценки решений требуется проанализировать игровые сущности, влияющие на решения, и их вклад. При этом многие сущности напрямую нельзя сравнивать, так как они имеют разные физические величины, например, расстояние до цели и запас патронов. Для этого требуется преобразовывать их полезность к одной общей нормализованной шкале, что не всегда является простой и очевидной задачей. Качество расчетов полезности решений зависит от того, насколько адекватно все факторы были сведены к одной шкале и насколько правильно подобраны кривые. Данные задачи не имеют общего решения и должны рассматриваться индивидуально в каждой задаче.

## Планировщики

[Buckland]

## Вероятностный подход и проверка статистических гипотез

Когда агент не обладает полными знаниями об окружающей среде, он должен строить модель мира и прогнозировать его изменения. Так как изменение среды в большинстве случаев является случайным процессом, наличие модели позволяет строить предположения о текущем состоянии окружающего мира только с определенной степенью уверенности. Степень уверенности может быть описана с помощью случайных переменных, для которых можно применить теорию вероятности и математическую статистику [Рассел и Норвиг, Ивановский].

Каждый отдельный параметр окружающей среды представляет собой случайную переменную. В зависимости от области определения выделяют три вида переменных:

* Булевы переменные. Могут принимать значения «истина» или «ложь».
* Дискретные переменные. Могут принимать значения из конечного множества значений.
* Непрерывные переменные. Могут принимать значения из бесконечного множества действительных чисел или его подмножества.

Для каждого возможного значения ai переменной A задается вероятность P(A=ai) того, что переменная имеет это значение. Запись P(ai) аналогична записи P(A=ai). Так же возможна векторная запись:

P(A) = <P(a1), P(a2), …, P(an)>.

Выражение P(x) часто читается как «вероятность события x».

Для непрерывных переменных задается плотность вероятности f(a), а вероятность того, что переменная будет иметь некоторое значение в окрестности ai, рассчитывается как интеграл плотности:

.

Вероятность P(A) в отсутствии любой другой информации называется безусловной или априорной. При наличии информации о других переменных априорная вероятность более не применима, так как переменные могут влиять друг на друга. В этом случае используется вероятность P(A|B) – вероятность события A при условии того, что произошло событие B. Такая вероятность называется условной или апостериорной.

Апостериорная вероятность рассчитывается по правилу (теореме) Байеса:

Оно позволяет пересчитать вероятность A, взяв в расчет как ранее известную информацию об A, так и данные о новых наблюдениях – событиях B. Если априорная вероятность P(A) не известна, ее заменяют некоторой оценкой, обычно априорное распределение принимают равномерным.

Выражение P(B|A) в теореме Байеса называется функцией правдоподобия. Это функция от параметра A, которая показывает, насколько были бы правдоподобны наблюдения B при условии, что имело место событие A.

Принятие решений с использованием вероятностных методов сводится к задаче проверки статистических гипотез – одного из классов задач математической статистики. Статистическая гипотеза – предположение о виде распределения и свойствах случайной величины, которое можно подтвердить или опровергнуть с помощью методов математической статистики [Ивановский, Васин].

Рассмотрим в качестве примера агента, участвующего в некоторой карточной стратегической игре с неполной информацией. Случайной величиной является набор карт в руках оппонента, а также в некоторых случаях выбранная им стратегия. Рациональный агент должен принимать решения, которые увеличивают его шансы на выигрыш. Гипотезой является предположение о конкретном значении случайной величины. Выдвигается множество гипотез H={hi}n, по одной для каждого возможного значения. Если априорная вероятность значений не известна, то берется равномерное распределение P(hi) = const. После того, как оппонент делает ход, мы наблюдаем событие и рассчитываем апостериорные вероятности гипотез по теореме Байеса. Агент выбирает одну гипотезу из множества в соответствии с некоторым критерием, принимает ее за текущее состояние окружающего мира и выполняет ход, который согласно установленным в него правилам является оптимальной реакцией на такое состояние окружающей среды.

При этом всегда существует вероятность принятие ошибочного решения при нехватке информации или недостаточном количестве наблюдений. Для количественной оценки ущерба (потерь), связанного с принятием решений, вводится функция потерь Π(hi, γj), которая задает потери при принятии решения hi, при условии, что правильным решением было γj.

### Критерий минимума среднего риска

Качество решения можно характеризовать математическим ожиданием функции потерь:

,

где – средний риск, – вероятность того, что одновременно принята гипотеза hi и правильным решением является γj.

Правило, при котором выбирается гипотеза с минимальным средний риском, называется байесовским правилом, байесовским критерием или критерием минимума среднего риска.

### Минимаксный критерий

Для использования критерия минимума среднего риска требуется большое количество дополнительной априорной информации о вероятностях правильности различных гипотез. Если такой информации нет, используется минимаксный критерий – выбирается гипотеза, у которой максимальная возможная ошибка минимальна. Он позволяет уменьшить потери в худшем случае, но не учитывает вероятность его возникновения.

### Критерий минимума полной вероятности ошибки

Если все ошибочные решения одинаково нежелательны, можно задать функцию потерь постоянной и не учитывать при принятии решения:

.

Тогда средний риск принимает вид:

Получаем частный случай байесовского критерия, который также называют критерием минимума полной вероятности ошибки или критерием идеального наблюдателя.

### Критерий максимума апостериорной вероятности

При постоянной функции потерь все ошибки имеют одинаковую стоимость, поэтому при принятии решения можно исходить только из свойств самих гипотез (их апостериорных вероятностей) и наблюдаемых событий. В этом случае решение принимается в пользу гипотезы с максимальной апостериорной вероятностью P(hi|E).

По теореме Байеса:

где E – наблюдаемые события и текущее состояние мира, P(E) – априорная вероятность текущего состояния (одинакова для всех гипотез, поэтому информация о ней не требуется при принятии решения).

### Критерий максимального правдоподобия

Если не известна информация об априорных вероятностях гипотез, то их распределение принимают за равномерное, т.е. P(hi)=const. Тогда получаем:

В этом случае критерий максимума апостериорной вероятности сводится к критерию максимальной правдоподобности.

### Преимущества и недостатки вероятностного подхода

Вероятностный подход применим к случаям, когда модель окружающей среды хорошо изучена, т.е. есть информация об априорных вероятностях, функциях правдоподобия, функциях потерь и других. К таким случаям можно отнести пример с карточной игрой, рассмотренный ранее. Зная карты у себя на руках и карты на столе, агент может рассчитать вероятности того или иного набора карт в руке оппонента и принять оптимальное решение в соответствии с заданным критерием. Другим примером является обнаружение и различение радиосигналов цифровым приемником [Васин]. Априорными данными является излученный сигнал – закодированный символ из известного конечного алфавита, апостериорными – принятый сигнал, являющийся суммой излученного сигнала с учетом затухания и шума. В отличие от примера с карточной игрой, в данном случае нельзя рассчитать точные вероятности того или иного события, но можно произвести оценки, основанные на статистической модели канала передачи и предположении о распределении шума.

В отсутствии подробной модели, приходится делать упрощения: априорные параметры принимать за равномерное распределение, потери считать равнозначными, делать предположения о функциях правдоподобия. Это приводит к уменьшению качества принятых решений. Статистические методы основаны на наблюдениях, поэтому чтобы сделать правильный выбор из набора гипотез, необходимо увеличивать количество наблюдений, а это не всегда возможно.

## Нейронные сети

[Шампандар, Millington]

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 {добавить рисунок}. При объединении в сети выходы одних нейронов являются входами других. Модель нейрона состоит из нескольких входов, взвешенного сумматора, функции активации и одного выхода [Шампандар]. 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.

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 {добавить рисунок}. 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 {добавить рисунок}, 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 {проверить в MATLAB}.

### Перцептрон

Перцептрон (персептрон, perceptron) – это структура, состоящая из одного или нескольких нейронов {добавить рисунок}. 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 {добавить рисунок}. Существуют и другие топологии {добавить рисунок}, т.е. способы соединения отдельных нейронов и слоев в единую сеть, но в данной работе будет рассмотрен классический вариант, как один из самых популярных и простых.

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

{поискать и добавить }

### 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.

#### 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

[Off the Beaten Path: Non-Traditional Uses of AI]

### 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.

## Генетические алгоритмы

[Шампандар]

# Выбор методов и алгоритмов для разработки

## Задачи, решаемые ботами, и подходящие для их реализации методы и алгоритмы

### Поиск пути

{Курсовик по алгоритмам и структурам данных}

### Прицеливание и стрельба

[Шампандар, Millington]

### Тактические решения

[Terrain Reasoning, Шампандар]

### Многоагентное взаимодействие

[Millington, coordinated movement]

# Алгоритмизация и реализация

## Выбор и обоснование языка программирования и библиотек

Рассматриваемые методы и алгоритмы принятия решений 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, может принимать команды как от компонента, отвечающего за управление с помощью устройств ввода, так и от компонента, отвечающего за искусственный интеллект.

# 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.

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.

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.

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. GDC 2001.

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.

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.

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.