# 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

## 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 2Рисунок 2).

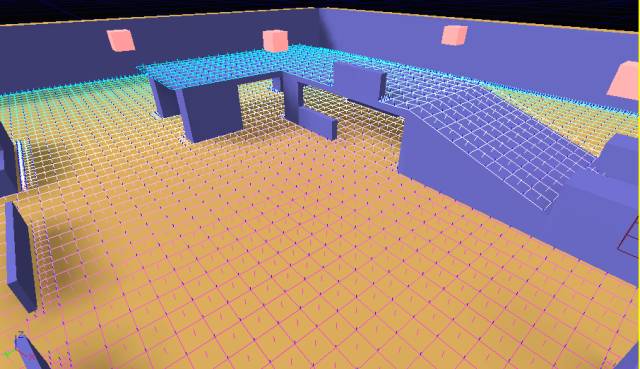


Рисунок 2. Пример навигационной сетки, используемой в видеоиграх.

### 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 3. 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 4). 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 5.

|  |  |
| --- | --- |
| Граф для примера.png  Рисунок 3. Исходный взвешенный граф. | Граф для примера.png  Рисунок 4. Начальное состояние меток. |
| Граф для примера.png  Рисунок 5. Результат первой итерации. | Граф для примера.png  Рисунок 6. 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 6.

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 7.

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

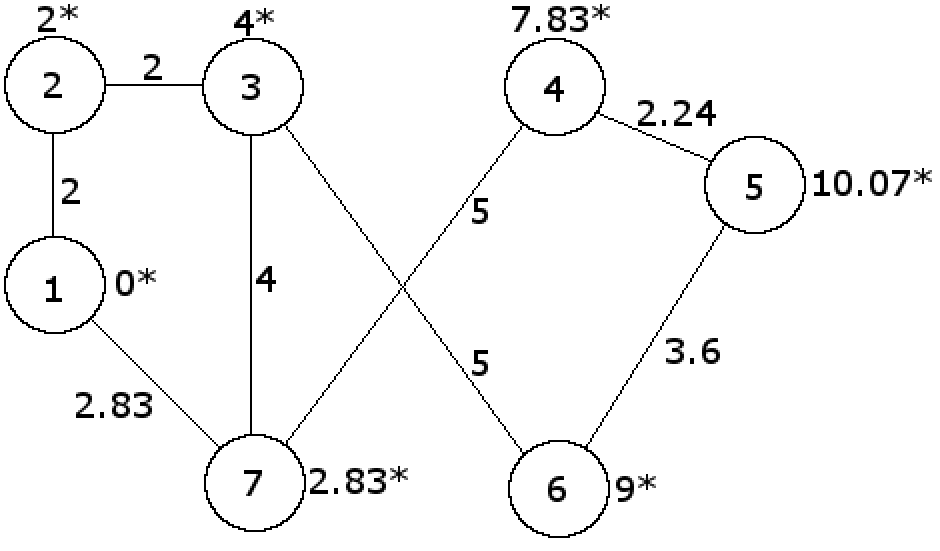


Рисунок 7. Итоговые значения меток.

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 8). Также имеет название «расстояние городских кварталов». Используется, если перемещение по клеткам допускается только в четырех направлениях.

.

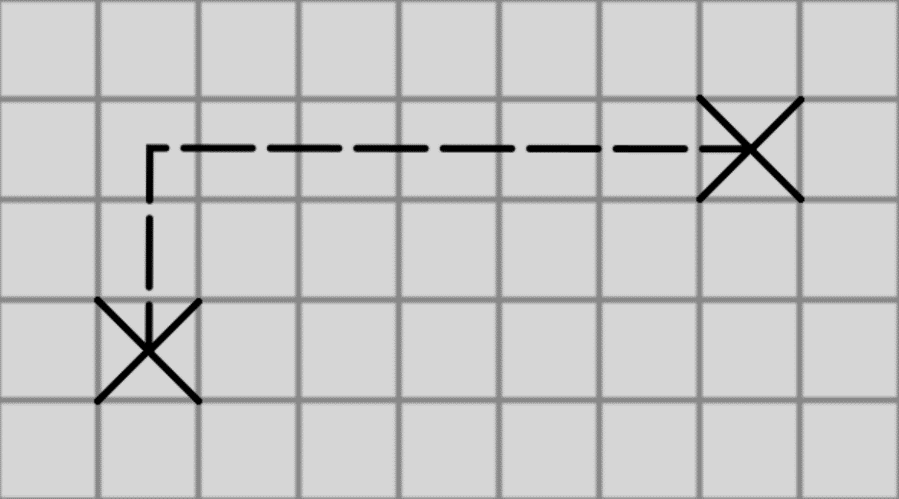


Рисунок 8. Манхэттенское расстояние между двумя узлами.

* Диагональное расстояние. Используется, если разрешено движение между диагональными клетками. Одна из разностей координат проходится по диагонали, другая – по прямой аналогично манхэттенскому расстоянию (рисунок 9).

где .

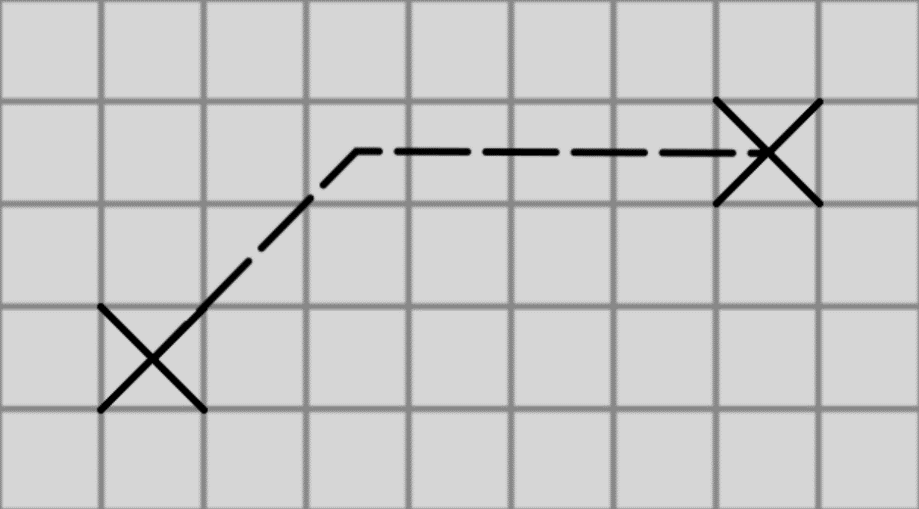


Рисунок 9. Диагональное расстояние между двумя узлами.

* Эвклидово расстояние. Представляет собой геометрическое расстояние между двумя точками в пространстве (рисунок 10). Гарантированно дает минимальную оценку расстояния.

.



Рисунок 10. 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].

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

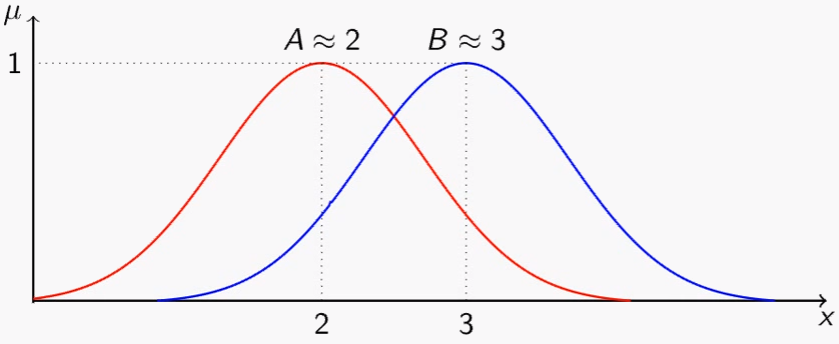


Рисунок 11. Примеры нечетких множеств.

На рисунках 12, 13 и 14, 15 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.

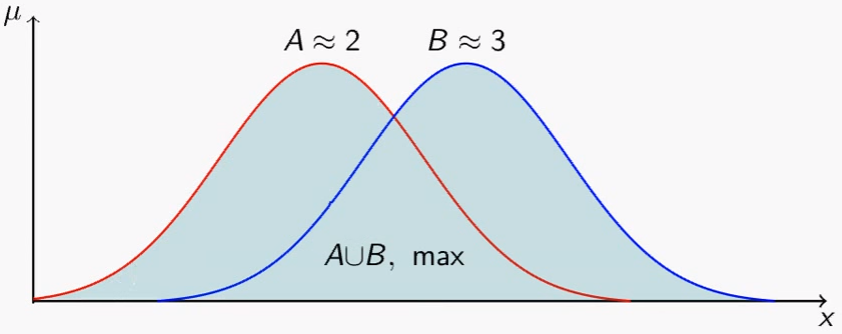


Рисунок 12. Объединение множеств с использованием нормы, определенной через максимум функций принадлежности.

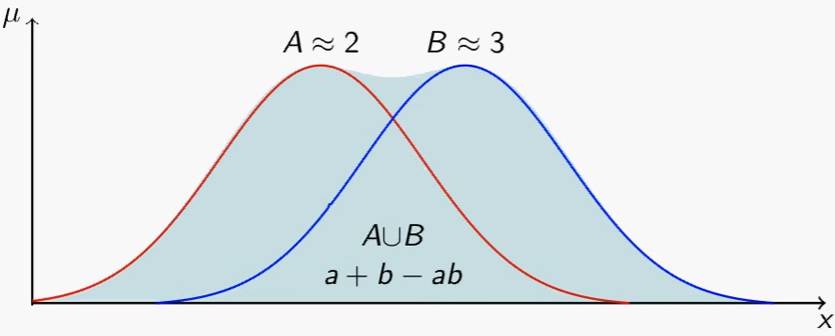


Рисунок 13. Объединение множеств с использованием нормы, определенной через сумму функций принадлежности.

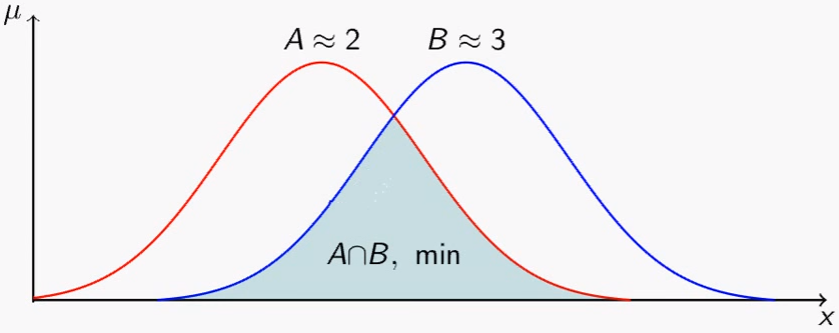


Рисунок 14. Пересечение множеств с использованием конормы, определенной через минимум функций принадлежности.

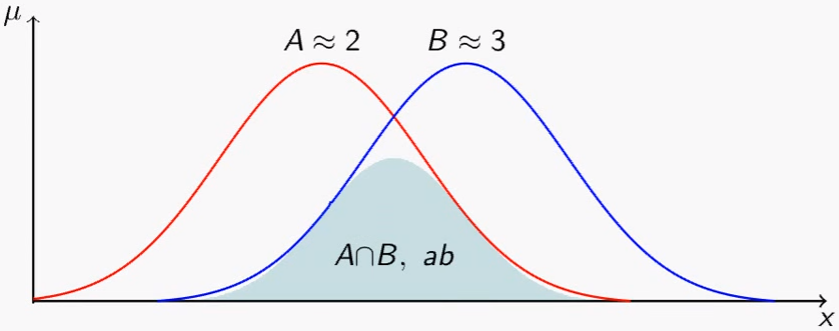


Рисунок 15. Пересечение множеств с использованием конормы, определенной через произведение функций принадлежности.

### 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 16 изображено графическое представление переменной «угловое отклонение».

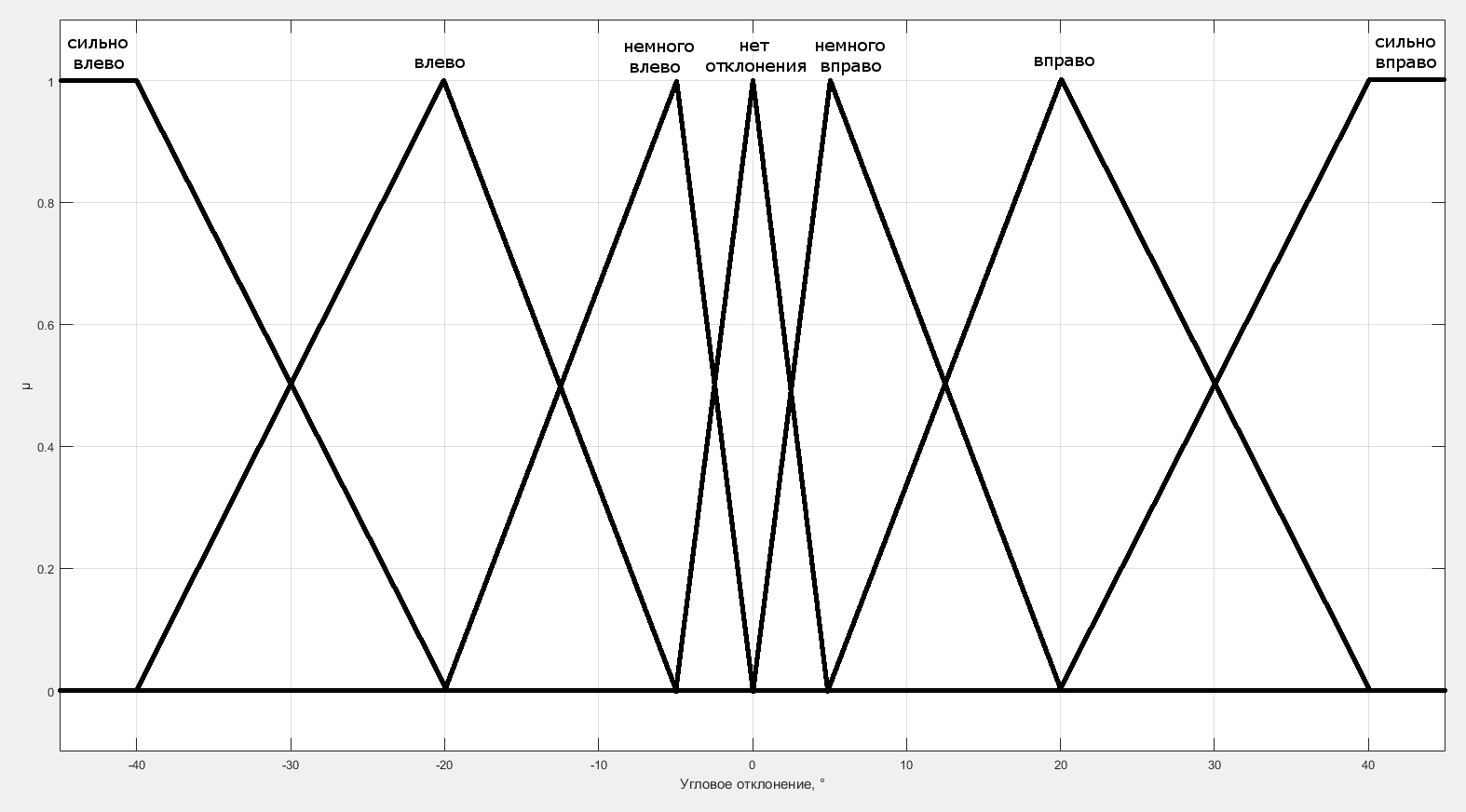


Рисунок 16. Графическое представление лингвистической переменной «угловое отклонение».

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 17 изображен пример дефаззификации с помощью центра максимумов. Данный метод позволяет быстро рассчитать результат, но он не учитывает влияние множеств со степенью достоверности меньше максимальной.

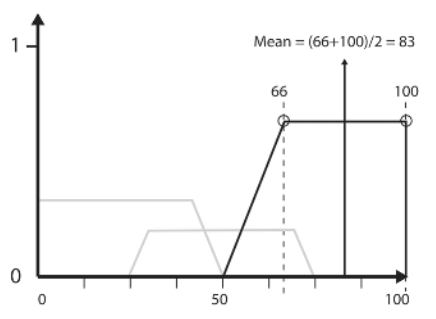


Рисунок 17. Пример дефаззификации с помощью центра максимумов.

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 18 Для ускорения расчетов можно увеличить шаг суммирования (рисунок 19).

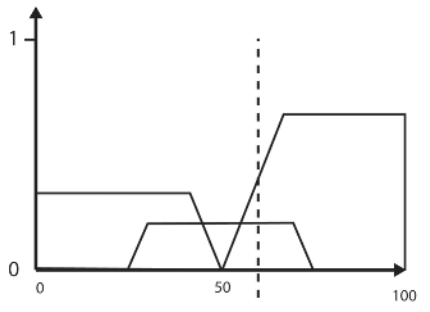


Рисунок 18. Пример дефаззификации с помощью центра масс.

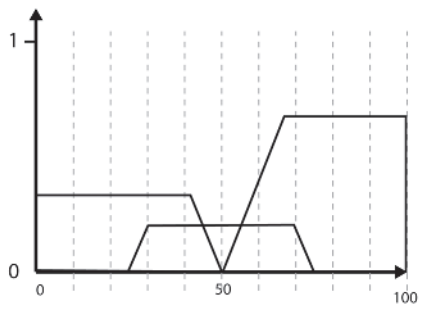


Рисунок 19. Пример разбиения области определения на дискретные интервалы.

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 20).

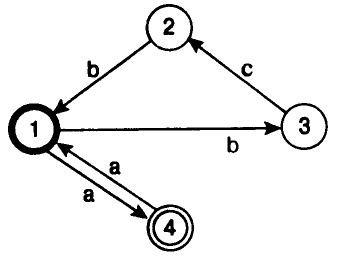


Рисунок 20. Графическое представление конечного автомата.

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 21).

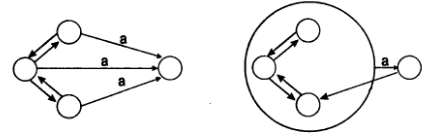


Рисунок 21. Графическое представление классического конечного автомата и иерархического, соответствующих одинаковому поведению.

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 22 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 23 и 24. 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.

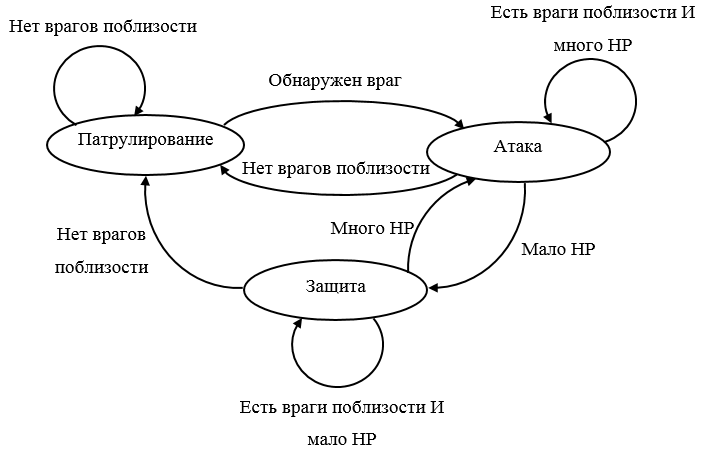


Рисунок 22. Граф состояний и переходов нечеткого конечного автомата.



Рисунок 23. Лингвистическая переменная «Очки здоровья».



Рисунок 24. 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 2.2.5, 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 25 изображен пример простого дерева, реализующего проход агента через дверь.



Рисунок 25. 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.

Любой игровой фактор, влияющий 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.

### 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 (рисунок 26):

,

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

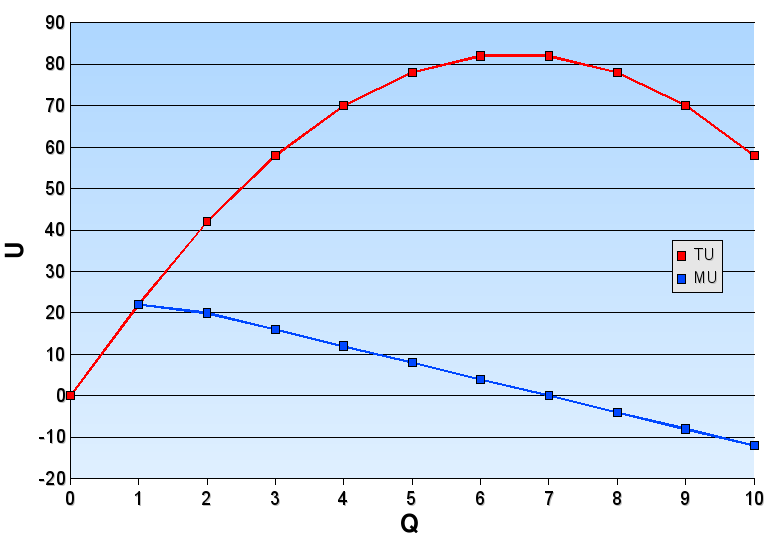


Рисунок 26. Графики полной и предельной полезности.

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.

### Оценка полезности величин и вычисление их влияния на решение

Любой игровой фактор может быть оценен с точки зрения его полезности.

* Факторы, имеющие количественное значение. К ним можно отнести параметры игровых сущностей: количество очков здоровья, количество патронов, урон, наносимый оружием, дистанция до цели, доступные ресурсы и другое. Полная полезность является функцией от этого значения.
* Факторы, не имеющие количественного значения. К ним можно отнести оценки тактического преимущества, наличие укрытий поблизости, уровень тревоги, качество ландшафта. Так же к этому типу можно отнести абстрактные понятия, для которых не существует числовых значений: угроза, безопасность, мораль, агрессивность, смелось, эмоции. Для оценки полезности таких факторов, необходимо искусственно ввести шкалу, имеющую минимум, максимум и шаг. Полная полезность является функцией от этой шкалы. Например, для тактической игры может понадобиться оценить проходимость местности. Для этого можно ввести 10-балльную шкалу, где 10 – полностью проходимая местность, 1 – полностью непроходимая. Тогда: дорога имеет значение 10, равнина – 8, лес – 5, болото – 3, горы – 2.

Полная полезность является функцией от параметра, характеризующего игровой фактор, влияющий на принятие решения. Часто в качестве функции используется линейная, полиномиальная и логистическая функции или их комбинации. Значения нормируются, чтобы можно было сравнивать полезность разных факторов.

Полезность решения является функцией от значений факторов, влияющих на него. В простейшем случае она вычисляется как взвешенное среднее:

,

где Ud – полезность решения, Ufi – полезность i-го фактора, wi – вес i-го фактора. Факторы, влияющие на решение, и их вес определяются разработчиком на основе предметной области, правил игры и желаемого поведения.

Для выбора конкретного решения на основании его полезности могут использоваться разные методы в зависимости от задачи. Например, может быть принято решение, у которого полезность максимальна. Для добавления разнообразия в поведение можно использовать полезность решения в качестве меры его вероятности. Для этого отсекаются решения с полезностью меньше пороговой, а полезность оставшихся нормируется так, чтобы их сумма была равна 1. Получившиеся коэффициенты являются вероятностью принятия этих решений.

### Пример

Рассмотрим упрощенный пример. Есть два агента, у каждого есть какое-то оружие и шкала, характеризующая количество очков здоровья (hit points). У первого агента есть несколько возможных решений: атаковать (играть агрессивно), защищаться (играть осторожно, искать укрытие), искать другое оружие, искать аптечку (игровой объект, восстанавливающий очки здоровья). Есть 4 фактора, влияющих на принятие решения.

* HPa – количество очков здоровья агента. (рисунок 27). 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.



Рисунок 27. График полезности HPa.

* HPe – количество очков здоровья противника. Параметр аналогичен HPa.
* Wa – качество оружия агента. (рисунок 28). Упрощенный параметр, характеризующий точность, урон, скорострельность, дальнобойность и другие параметры оружия. В реальных примерах все эти свойства желательно рассматривать отдельно.



Рисунок 28. График полезности Wa.

* We – качество оружия оппонента. Параметр аналогичен Wa.

Возможные решения:

* Атака: A = (3 \* HPa + 2 \* Wa + 1 \* (1 - We)) / 6.
* Защита: D = (1 \* HPe + 2 \* We + 2 \* (1 - HPa)) / 5.
* Поиск оружия: W = 1 – Wa.
* Поиск аптечек: H = 1 – HPa.

Рассмотрим несколько игровых ситуаций:

1. HPa = 100; Wa = 6; HPe = 80; We = 4;

U(HPa) = 1; U(Wa) = 0.6; U(HPe) = 0.8944; U(We) = 0.4;

A = 0.8; D = 0.3389; W = 0.4; H = 0.

Предпочтительным действием является атака.

1. HPa = 15; Wa = 6; HPe = 80; We = 4;

U(HPa) = 0.3873; U(Wa) = 0.6; U(HPe) = 0.8944, U(We) = 0.4.

A = 0.4936; D = 0.5840; W = 0.4; H = 0.6127.

Предпочтительным действием является отступление и поиск аптечек.

### 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. Качество расчетов полезности решений зависит от того, насколько адекватно все факторы были учтены в выражениях для их расчета и насколько правильно подобраны кривые и вес факторов. Данные задачи не имеют общего решения и должны рассматриваться индивидуально в каждой задаче.

## Планирование действий, основанное на целях

В отличие от конечных автоматов и деревьев поведения, где текущее состояние мира определяет поведение агента, в методе, основанном на целях исходными предпосылками к решению является желаемое состояние мира после его принятия. Планирование действий, основанное на целях (goal-oriented action planning, GOAP) – это метод принятия решений, разработанный Джеффом Оркином для реализации искусственного интеллекта в видеоигре F.E.A.R. [Orkin 2006]. Данный метод был предложен примерно в то же время, что и деревья поведения [Isla 2005], как один из способов решения проблемы возрастающей сложности разработки искусственного интеллекта с использованием конечных автоматов – наиболее популярного на тот момент метода принятия решений в компьютерных играх.

Для выбора действия агента строится граф возможных состояний игрового мира: узлы представляют состояния, а ребра – действия агента, которые изменяют это состояние. У агента есть некоторая цель. Эта цель описывается желаемым состоянием игрового мира. С помощью алгоритмов поиска кратчайшего пути на графе состояний мира ищется оптимальная последовательность действий, которая приводит текущее состояние мира к желаемому.

Рассмотрим основные понятия GOAP.

#### Модель мира (world model)

Модель игрового мира задается набором влияющих на принятие решений параметров. Состояние мира (world state) – это набор конкретных значений параметров. Пример модели приведен в таблице 1.

|  |  |
| --- | --- |
| Параметр | Тип |
| Очки здоровья агента | Float |
| Очки здоровья противника | Float |
| Укрытие | Enum { Далеко, Близко, В укрытии } |
| Противник в поле зрения | Boolean |
| В поле зрения противника | Boolean |
| Расстояние до противника | Float |
| Агент атакован | Boolean |
| Противник атакован | Boolean |
| Рядом с | Узел навигационного графа или координата |

Таблица 1. Пример модели мира.

#### Цель (goal)

Цель определяет, что хочет достичь агент. Для каждой цели задано желаемое состояние мира. Оно содержит часть параметров, которые должны принимать нужное значение. Если какие-то параметры не указаны в желаемом состоянии, их значение может быть произвольным. Примеры целей приведены в таблице 2. При выборе активной цели учитывается текущее состояние мира.

|  |  |
| --- | --- |
| Цель | Желаемое состояние мира |
| Бежать в укрытие | Укрытие = В укрытии |
| Атаковать противника | Противник атакован = true |
| Спасаться бегством | В поле зрения противника = false |

Таблица 2. Примеры целей.

#### Действие (action)

Действия – это то, что может выполнить агент, чтобы изменить состояние мира. Для каждого действия определено предусловие – состояние мира, необходимое для того, чтобы оно было выполнимым, и постусловие – состояние мира после того, как оно будет выполнено. В некоторых случаях действиям присваивается стоимость, чтобы при планировании учесть тот факт, что некоторые действия более предпочтительны, чем другие. У действий может быть переменная стоимость. Например, стоимость движения зависит от расстояния, а стоимость стрельбы от экипированного оружия. Примеры действий приведены в таблице 3.

|  |  |  |
| --- | --- | --- |
| Действие | Предусловие | Постусловие |
| Атаковать оружием ближнего боя | Расстояние до противника < дальность рукопашной атаки | Противник атакован = true |
| Атаковать оружием дальнего боя | Расстояние до противника < дальность дальней атаки | Противник атакован = true |
| Атаковать оружием дальнего боя из укрытия | Укрытие = В укрытии  Расстояние до противника < дальность дальней атаки | Противник атакован = true |
| Идти в точку (точка) |  | Рядом с = точка |
| Укрыться | Укрытие = Рядом | Укрытие = В укрытии |

Таблица 3. Примеры действий.

#### Планирование (planning, plan formulation)

План – это последовательность действий, которые будет выполнять агент. Планирование – это поиск последовательности действий, оптимальной по критерию минимума количества действий или минимума суммарной стоимости действий, если она определена. Для поиска используются алгоритмы поиска пути на графах. Как правило это A\* или одна из его разновидностей. В качестве дистанции между узлами графа используется количество действий (вес ребра равен 1) или суммарная стоимость действий (вес ребра равен стоимости действия). В качестве эвристики используется количество отличающихся параметров между состояниями. В отличии от навигации в пространстве, где дистанция и эвристика имеют одни и те же размерность и физический смысл (расстояние между узлами), при планировании требуется сбалансировать эвристику и дистанцию между состояниями, чтобы A\* давал оптимальный путь без ошибок. Эвристика используется для ускорения поиска, поэтому если сложно ее подобрать соизмеримой со стоимостью действий, можно использовать заниженную эвристику или отказаться от нее совсем, чтобы сохранить качество решений.

На рисунке 29 изображен пример поиска оптимального плана. Целью является «Атаковать противника». Желаемое состояние для этой цели можно получить с помощью трех действий, каждое из которых имеет разные предусловия. Если предусловиях нескольких или всех действий уже выполнены, то они все будут считаться равноценными. Если выбирать действия случайно, то может оказаться так, что агент выберет действие «Атаковать оружием дальнего боя» и выйдет из укрытия. Наличие укрытия не указано в цели, но если агент уже в нем находится, выход из него можно считать тактической ошибкой. Чтобы учесть предпочтение, можно добавить стоимость действий. Например, стоимость «Атаковать оружием ближнего боя» < стоимость «Атаковать оружием дальнего боя из укрытия» < стоимость «Атаковать оружием дальнего боя». В этом случае если агент находится рядом с противником, то он предпочтет атаку врукопашную, если в укрытии на расстоянии стрельбы, то стрельбу из укрытия, иначе просто стрельбу без укрытия.



Рисунок 29. Пример планирования на графе состояний мира.

GOAP используется в различных играх: F.E.A.R. (2005), F.E.A.R. 2: Project Origin (2009), Middle-earth: Shadow of Mordor (2014), Rise of the Tomb Raider (2015) и других [GOAP GDC 2015]. Он используется значительно реже деревьев поведения, появившихся примерно в то же время. Деревья поведения позволяют создавать жестко заданное прогнозируемое поведение с помощью простых инструментов без привлечения программистов, что зачастую предпочтительней в больших коммерческих проектах. С другой стороны, GOAP используется в проектах, где большое внимание уделяется реалистичному и интересному поведению персонажей.

При использовании GOAP агент сам выбирает наиболее подходящие действия и планирует последовательность их выполнения. Это дает более реалистичное и менее предсказуемое поведение по сравнению с методами, где последовательности действий агентов задаются разработчиком (например, деревья поведения). Это свойство так же может являться и недостатком, так как в некоторых случаях с точки зрения игрового дизайна требуется выполнить не то действие, которое агент считает наиболее подходящим. При использовании GOAP у разработчиков меньше возможностей для ручной корректировки поведения агента.

GOAP полностью развязывает цели и действия. Это значит, что можно добавлять в систему новые сущности, не изменяя уже существующие и не добавляя дополнительных связей как, например, в конечных автоматах. Однако, стоит иметь в виду, что добавление новых действий может увеличить нагрузку на процессор.

GOAP решает, как с помощью заданного набора действий эффективно достичь цели, но не как выбрать текущую цель. Для этого нужно использовать другие методы, дающие численную оценку важности цели, например, нечеткую логику, теорию полезности или нейронную сеть.

Данный метод может сильно нагружать процессор, так как для каждого агента требуется выполнять алгоритм поиска пути на графе, которой может иметь большое количество узлов и ребер. Чтобы уменьшить нагрузку, можно уменьшить количество возможных действий, ограничить длину последовательности действий, уменьшить количество переменных, описывающих состояние мира, и ограничить количество агентов, обращающихся к планировщику в один кадр игровой симуляции. Так же часть задач может быть перенесена в отдельные подсистемы искусственного интеллекта.

Если цель не достижима, т.е. не существует пути от желаемого состояния мира до текущего, алгоритм поиска пути будет перебирать все возможные комбинации действий. Это может занять много времени, поэтому рекомендуется искусственно ограничить максимальную глубину поиска, т.е. максимальное количество действий в запланированной последовательности.

## Нейронные сети

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

### Модель нейрона

Нейронные сети состоят из большого количества однотипных узлов, называемых искусственными нейронами, основанных на упрощенной модели нервной клетки [Шампандар]. При объединении в сети выходы одних нейронов являются входами других. Модель нейрона состоит из нескольких входов, взвешенного сумматора, функции активации и одного выхода (рисунок 30). Она аппроксимирует функцию , где – вектор входных значений. Каждому входу xi соответствует некоторый коэффициент wi – вес, характеризующий вклад этого входа в итоговый результат. Один из входов отвечает за смещение нуля. Его можно представить либо как отдельный вход, отвечающий только за смещение, либо как один из обычных входов, на который всегда подается единица. Обучение нейрона заключается в подборе весовых коэффициентов, обеспечивающих ожидаемое состояние. При корректировании весов используются численные методы.

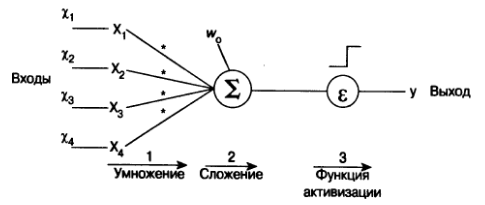


Рисунок 30. Модель нейрона.

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

#### Комбинаторная функция

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

.

#### Функция активизации

Функция активизации преобразует объединенное входное значение к выходному. Ее можно считать решающим устройством, определяющим состояние нейрона: активный или не активный. Они могут иметь разный вид, в зависимости от решаемых задач (рисунок 31).

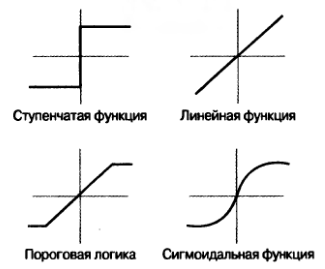


Рисунок 31. Примеры функций активизации.

При выборе функции следует учитывать следующие свойства:

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

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

,

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

### Перцептрон

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

### Многослойный перцептрон

В многослойных перцептронах используются более сложные топологии, состоящие из нескольких слоев нейронов. По своей структуре многослойный перцептрон фактически является несколькими однослойными перцептронами соединенными последовательно: выходы предыдущего являются входами следующего. Как правило они состоят из одного входного слоя, одного выходного и нескольких скрытых (рисунок 32).

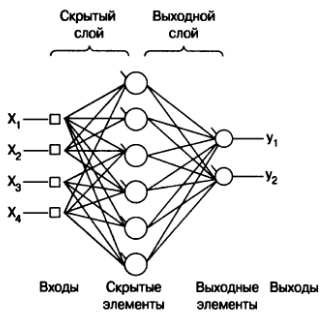


Рисунок 32. Многослойный перцептрон.

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

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

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

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

### Обучение перцептрона

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

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

,

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

,

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

,

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

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

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

### Обучение с ослабленным контролем

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

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

### Применение нейронных сетей в компьютерных играх

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

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

#### DeepMind и Starcraft 2

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

#### Борис и Блицкриг 3

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

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

#### Supreme Commander 2

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

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

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

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

### Преимущества и недостатки нейронных сетей

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

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

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

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

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

# Общие компоненты, не зависящие от метода принятия решений

Согласно модели агента, изображенной на рисунке 1, помимо решающего устройства он содержит датчики для получения информации об окружающей среде и исполнительные механизмы (актуаторы) для воздействия на среду. В случае ботов в компьютерных играх нет необходимости в оцифровке окружающего мира с помощью датчиков, т.к. вся необходимая информация уже хранится в оперативной памяти. Однако, эту информацию необходимо преобразовать к виду, удобному для использования в алгоритмах принятия решений и выделить из нее данные, релевантные с точки зрения правил рассматриваемой игры. Для игры, правила которой описаны в пункте 1.5, к таким данным можно отнести количество очков здоровья всех персонажей, их координаты, и планировку уровня. В рамках игры персонажи могут совершать определенные простые действия под управлением игроков: двигаться в заданном направлении, смотреть в заданную точку (целиться) и стрелять. Т.к. бот по определению – это компьютерная программа, использующая интерфейс, предназначенный для человека, персонажи-агенты должны использовать эти же действия.

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

## Интерфейс управления персонажем

Для управления персонажем используется следующий интерфейс:



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

## Движение

Для управления движением агента используется отдельный компонент. Он принимает на вход точку, в которую должен идти агент, ищет оптимальный путь до нее, используя алгоритмы поиска на графах, и вызывает метод Controllable.Move(Vector3 direction), чтобы направить персонажа к следующей точке найденного пути. Блок-схема алгоритма работы компонента показана на рисунке 33.



Рисунок 33. Блок-схема алгоритма работы компонента Navigation.

Для поиска пути используется представление местности в виде навигационной сети (NavMesh). Построение сетки и поиск пути осуществляются с помощью инструментов игрового движка и в данной работе не рассматриваются. Пример навигационной сетки, построенной с помощью этих инструментов, показан на рисунке 34.

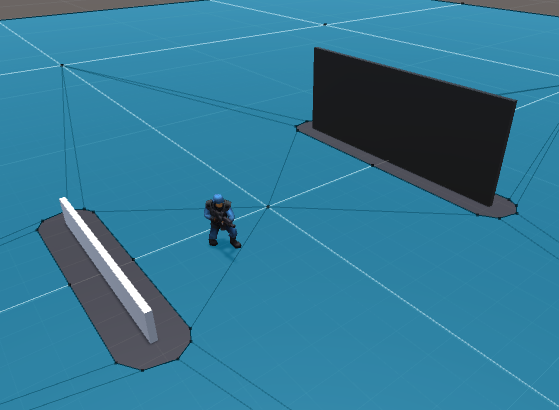


Рисунок 34. Пример навигационной сетки для поиска пути и обхода препятствий.

## Прицеливание и стрельба

Все персонажи используют одно и то же оружие. Оно обладает довольно большим разбросом, поэтому точность прицеливания влияет на общее количество попаданий меньше, чем расстояние до цели. Простейшая реализация прицеливания – это наведение на цель с упреждением, зависящим от скорости и направление движения цели, а также скорости полета снаряда. При расчете движение цели принимается за прямолинейное. Блок-схема алгоритма работы компонента показана на рисунке 35.



Рисунок 35. Блок-схема алгоритма работы компонента Shooting.

На рисунке 36 показана схема для расчета упреждения. – скорость цели, – скорость снараяда, d – расстояние между стрелком и целью, α – угол между вектором движения цели и вектором от стрелка до цели. Параметры цели, стрелка и снаряда известны. Нужно найти время t, за которое цели и снаряд проходят расстояния до точки встречи. По теореме косинусов:

,

.

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



Рисунок 36. Расчет упреждения.

## Состояние персонажа

Каждый игровой персонаж обладает рядом параметров, которые в рамках рассматриваемой игры являются открытой информацией и могут быть использованы для принятия решений. Значения параметров обновляются на каждом шаге игровой симуляции. Описание параметров представлено в таблице 4.

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

Таблица 4. Описание параметров персонажей.

## Анализ местности

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

В [Terrain Reasoning GDC 2001] рассматривается пример с выбором оптимальной позиции для снайпера (рисунок 37). Для этого каждая точка оценивается по ряду критериев: обзор из данной точки, защищенность, освещенность, может ли противник незаметно подобраться к этой точке и другие. Все оценки взвешиваются и суммируются.

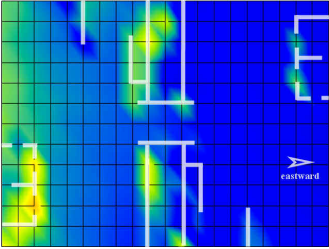


Рисунок 37. Карта оценок позиций для снайпера (чем ярче, тем лучше).

В [AI For Generated Worlds GDC 2016] предлагается использовать несколько карт, в каждой из которых хранятся определенные параметры точек (рисунки 38, 39 и 40). В качестве информации используются расстояния от точки до ключевых объектов: выхода, стен, воды или определенной территории. Для расчета используется алгоритм Дейкстры – осуществляется поиск пути от точки до ближайшего объекта заданного типа. Результатом является длина найденного пути.



Рисунок 38. Карта расстояний до стен (чем ярче, тем ближе).



Рисунок 39. Карта расстояний до выхода (чем ярче, тем ближе).



Рисунок 40. Карта расстояний до воды (чем ярче, тем ближе).

Информация, которую нужно обрабатывать для принятия решений, зависит от правил игры. На рисунке 41 изображена арена из игры, рассматриваемой в данной работе. Черным цветом окрашены стены, они блокируют обзор и все выстрелы. Белым цветом окрашены укрытия, они блокируют половину выстрелов, если стрелок стоит дальше определенного расстояния от него. Если стрелок стоит близко к укрытию, он получает полное преимущество – блокируются только выстрелы противника, собственные выстрелы игнорируют укрытие. Игроки изначально расположены в противоположных углах арены. В других углах расположены аптечки – объекты, восстанавливающие очки здоровья персонажей.

В таких условиях наиболее полезной информацией являются данные об укрытиях и расстояниях до них.

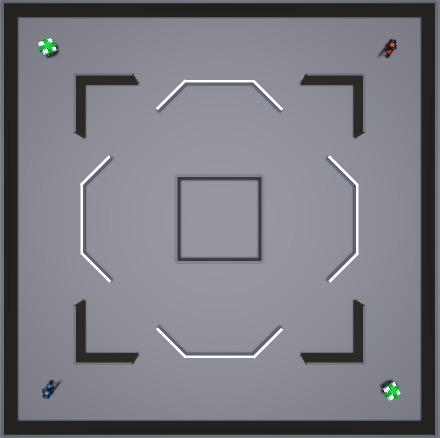


Рисунок 41. Арена, на которой происходят действия игры.

Описание параметров, рассчитываемых для каждой точки, представлено в таблице 5. Алгоритм работы компонента, обновляющего значения параметров, изображен на рисунке 42.

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

Таблица 5. Описание параметров, используемых для анализа местности.

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



Рисунок 42. Блок-схема алгоритма работы компонента TerrainReasoning.

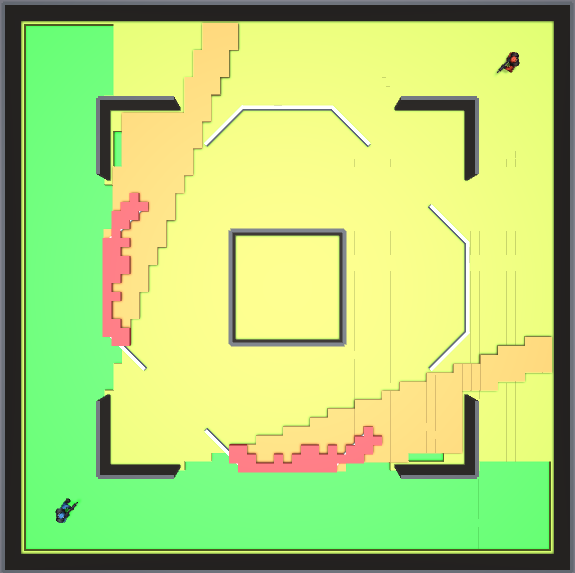


Рисунок 43. Пример карты оценки позиций для исходного положения игроков.

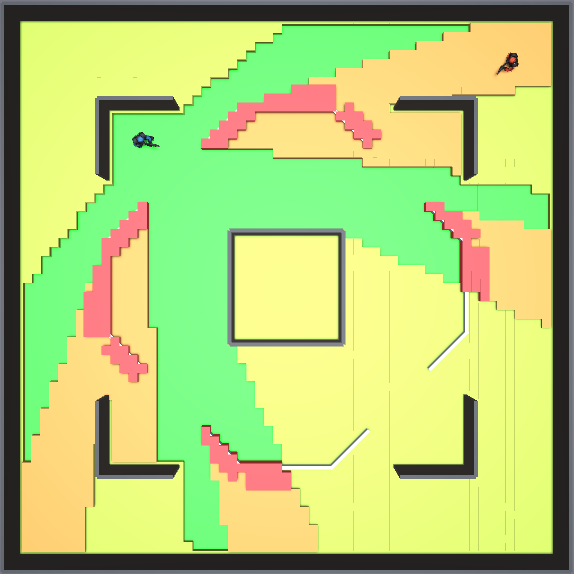


Рисунок 44. Пример карты оценки позиций.

На рисунках 43 и 44 изображены карты оценок позиций красного игрока для двух разных положений синего игрока при определенной стратегии. Веса отмечены шкалой цвета: от зеленого (минимум) до красного (максимум). Зеленым цветом отмечены точки с прямой видимостью и без укрытий. Они имеют минимальный вес, т.к. находиться в них наиболее опасно для агента. Желтым цветом отмечены точки, в которых видимость заблокирована стенами. Точки отмечены оранжевым цветом, если между ними и противником есть укрытия. Максимальный вес имеют точки в тех местах, где агент получает полное преимущество от укрытий.

На рисунке 45 изображена карта расстояний от агента до точек.

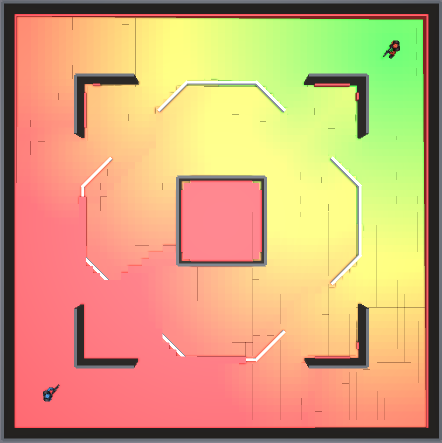


Рисунок 45. Карта расстояний от агента до точек.

# Конечные автоматы

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

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



Объект конечного автомата хранит все состояния, проверяет условия переходов для текущего и осуществляет переходы, если условия выполнены. Блок-схема алгоритма работы автомата изображена на рисунке 46.Рисунок 50



Рисунок 46. Блок-схема работы алгоритма работы класса, реализующего переключение состояний.

## Поиск противника

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

## Атака

В этом состоянии агент агрессивно атакует противника, стараясь поддерживать ближнюю или среднюю дистанцию и жертвуя защитой, чтобы нанести больше урона. Для анализа местности выбирается весовая функция, позволяющая выбрать позицию для атаки. Точки имеют больший вес, если из них видно противника, если они позволяют получить полное преимущество от укрытия и если они находятся на определенном расстоянии от противника. На рисунке 47 показан пример карты весов в состоянии атаки. Агент выбирает точки в определенном радиусе от себя с весом больше порогового. Затем из этих точек берется случайная и устанавливается в качестве точки следования. При таком подходе агент ведет себя более хаотично и постоянно находится в движении, по сравнению с случаем, когда выбирается точка с максимальным весом.

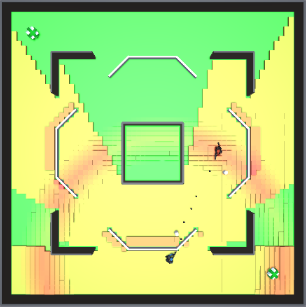


Рисунок 47. Карта весов точек для состояния атаки (зеленый – минимум, красный – максимум).

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

## Защита

В этом состоянии агент сражается менее агрессивно и выбирает позицию так, чтобы минимизировать получаемый урон. Для этого весовая функция выбирается таким образом, чтобы повысить вес точек за укрытиями и на большом расстоянии от противника. На рисунке 48 показан пример карты весов в состоянии защиты. Точка для следования выбирается аналогично с состоянием атаки, чтобы агент всегда находился в движении и по нему было сложнее попасть.

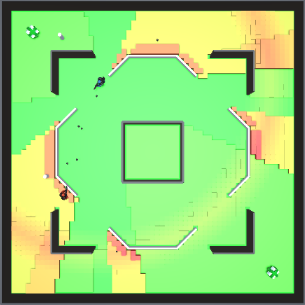


Рисунок 48. Карта весов точек для состояния защиты (зеленый – минимум, красный – максимум).

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

## Отступление и поиск аптечек

В этом состоянии весовая функция выбирается так, чтобы больший вес имели точки за стенами. Наибольший вес имеют точки, в которых расположены аптечки. На вес этих точек также влияет расстояние от агента и расстояние от противника, чтобы аптечки имели больший вес, если они ближе к агенту, и меньший вес, если на пути к ним находится противник. На рисунке 49 показан пример карты весов в состоянии защиты. В качестве точки следования выбирается точка с максимальным весом. Каждую секунду проверка производится заново, т.к. ситуация может измениться. Например, противник переместится так, что первая цель станет менее выгодной, либо аптечки в точке следования уже не будет.



Рисунок 49. Карта весов точек для состояния отступления (зеленый – минимум, красный – максимум).

Если на карте больше нет аптечек или очки здоровья были восстановлены, то агент переходит в состояние поиска противника.

## Вывод

На рисунке 50 в виде графа изображен итоговый конечный автомат для интеллектуального агента в рассматриваемой игре.



Рисунок 50. Представление конечного автомата в виде графа.

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

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

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

# Деревья поведения

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

## Описание элементов деревьев поведения

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

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

#### Ветвление

На рисунке 51 показано дерево поведения, имитирующее инструкцию ветвления «если … то … иначе». Если условие возвращает Success, то выполняется следующее действие в последовательности. Если Failure, то вся последовательность возвращает Failure и селектор переходит к следующему дочернему элементу



Рисунок 51. Дерево поведения, имитирующее инструкцию ветвления «если … то … иначе».

#### Логическое И

На рисунке 52 показано дерево поведения, имитирующее логическое И. Последовательность дойдет до выполнения действия только если все условия вернут Success.



Рисунок 52. Дерево поведения, имитирующее логическое И.

#### Логическое ИЛИ

На рисунке 53 показано дерево поведения, имитирующее логическое ИЛИ. Если все условия в селекторе вернут Failure, то селектор также вернет Failure и последовательность не дойдет до выполнения действия.



Рисунок 53. Дерево поведения, имитирующее логическое ИЛИ.

#### Логическое НЕ

На рисунке Рисунок 54 показано дерево поведения, имитирующее логическое НЕ. Стандартный декоратор «Инвертор» меняет Success на Failure и наоборот. Если дочерний элемент возвращает Running, то инвертор так же возвращает Running.



Рисунок 54. Дерево поведения, имитирующее логическое НЕ.

## Реализация поведения агента

Используя рассмотренные логические конструкции построим дерево, реализующее поведение агента аналогичное реализованному ранее с использованием конечных автоматов. Поведение в каждом состоянии можно определить одной отдельной ветвью: в корне поддерева находится узел «последовательность», первые элементы которой определяют условия нахождения в данном состоянии, а остальные – действия, выполняемые в этом состоянии. На рисунке 55 показан общий вид такого дерева без детализации отдельных ветвей. Селектор последовательно проверяет все дочерние элементы, пока один из них не вернет Success, т.е. не будет выбрана ветвь, подходящая для текущих условий окружающей среды. На рисунках 56-59 отдельные ветви изображены более подробно. Ветвь для отступления выбирается в последнюю очередь, если условия для всех остальных ветвей не были выполнены, поэтому в ней самой не нужно проверять эти условия.



Рисунок 55. Общий вид дерева поведения для рассматриваемого агента.



Рисунок 56. Поддерево, определяющее поведение в состоянии атаки.



Рисунок 57. Поддерево, определяющее поведение в состоянии защиты.



Рисунок 58. Поддерево, определяющее поведение в состоянии патрулирования и поиска противника.



Рисунок 59. Поддерево, определяющее поведение в состоянии отступления и поиска аптечек.

Из рисунков видно, что ветви похожи друг на друга. Можно изменить дерево, чтобы уменьшить дублирование элементов. Можно выделить три ветви: стрельба, установка весовой функции и параметром для анализа местности и движение (рисунок 60).



Рисунок 60. Общий вид измененного дерева поведения для рассматриваемого агента.

Теперь в корне дерева расположена последовательность, а не селектор, так как работа дерева свелась в трем параллельным деятельностям агента вместо выбора одной модели поведения. Рассмотрим подробнее ветви дерева (рисунки 61, 62 и 63).



Рисунок 61. Ветвь дерева, отвечающая за управление стрельбой.



Рисунок 62. Ветвь дерева, отвечающая за установку параметров анализа местности.



Рисунок 63. Ветвь дерева, отвечающая за движение.

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

## Вывод

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

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

# Список использованных источников

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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