МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Южно-Уральский государственный университет (национальный исследовательский университет)»

Высшая школа экономики и управления

Кафедра «Экономическая безопасность»

Искусственный интеллект в играх

РЕФЕРАТ

ЮУрГУ – ХХХХХХ. 20ХХ. ХХХХХ. Р

(код направления/специальности, год, номер студенческого )

Руководитель, должность

/ И.О. Фамилия /

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 г.

Автор

студент группы ЭУ – ХХХ

/ И.О. Фамилия /

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 г.

Реферат защищен с оценкой

(цифрой, прописью)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 г.

Челябинск 2020

Искусственный интеллект - симуляция работы мозга определяющая, как в видео игре двигаются и действуют NPC, как атакуют враги, а также то, как хорошо у них это получается. Чтобы понять, как работает ИИ в видеоиграх, изначально нужно дать определение искусственному интеллекту. Интеллект определяется как мыслительная способность получать и применять знания и навыки, и ИИ способен на это, однако он не может делать это сам по себе. ИИ – это определенный набор алгоритмов и компьютер, который их запустил, не становится способным мыслить. Его задача - быстро оценивать ситуацию, но вместо того чтобы реагировать на неё, он может лишь выполнить действия, прописанные в алгоритме для данной ситуации, поэтому результат действия выглядит словно результат мышления. Обычно создается множество условий: если вот это, то вот это, но иногда разработчики могут совсем разойтись и написать условия если вот это и вот это, но не вот это, то вот это. Чем сложнее условия, тем более разумным кажется противник, тем на большее количество обстоятельств он может реагировать, и так как компьютер, как мы считаем, реагирует благодаря искусственному интеллекту, в видеоиграх это может показаться проявлением инстинкта или даже разумом. Однако даже лучший искусственный интеллект не обязательно способен обучаться. Он может начать отслеживать, сколько раз встретилось определенное условие и через некоторое количество раз поменять свое действие после него. Например, если игрок энное количество раз атаковал противника определенным образом, разработчики вполне могли запрограммировать его так, что, к примеру, через пять таких действий игрока, противник начнет вести себя агрессивнее, и по-другому на это реагировать. Добавление значений в базу данных для отслеживания различных вещей в ходе игры можно считать обучением, но весьма примитивным и неполноценным. Под настоящим обучением ученые имеют в виду изменение поведения самостоятельно, наличие мыслей и идей, выполнении новых, изначально не заложенных извне действий. Существует искусственный интеллект, способный на это, но он не используется в видеоиграх, однако все же модель мозга, стимулирующая обучение, уже создана.

Искусственный интеллект может быть представлен разными уровнями.

ИИ, использующий метод резиновой нити по английский роберт band. Использование метода резиновой нити позволяет ИИ не отставать по уровню от игрока. Это происходит благодаря тому, что ИИ отмечает некоторые действия игрока, и на их основании определяет, что он сам может сделать. Например, в гонках из-за этого NPC начинают вести себя агрессивнее, более умело, и в некоторых случаях, они могут даже становиться быстрее. В спортивных видеоиграх команда соперников благодаря этому методу может сильно усилить свою защиту или перейти в куда более агрессивное нападение. Это относится к адаптивной сложности игры. То есть игра вычисляет, насколько хорошо играет пользователь, и пытается подстроиться индивидуально под его уровень игры. Для этого практически на протяжении всей игры должны присутствовать различные уровни искусственного интеллекта. Он должен постоянно наблюдать за действиями игрока, находить определённые триггеры, к примеру, использует ли игрок более сложные комбинации или хаотично нажимает на все кнопки. ИИ должен сортировать такие данные и заносить их в таблицу, чтобы получить сведения о том, какой сложности должна быть игра для этого игрока.

Персонажей, которыми управляет искусственный интеллект, подразделяют на:

* NPC – это нейтральные или дружественные персонажи в игре.
* Боты – самые сложные в программировании, представляют собой врагов, с которыми сражается игрок. Обладают возможностями, схожими с теми, которые имеются у игрока. В любой временной период игры их не может быть много.
* Мобы – в отличие от ботов, обладают низким игровым интеллектом, цель игрока – убивать их для накопления очков опыта, денег или для открытия закрытой территории, так называемого «тумана войны».

Системы игрового ИИ можно условно разделить на два типа: локальный и глобальный.

Локальный - это ИИ отдельной единицы, как например солдат или автоматический механизм. Такие системы состоят из основных четырех элементов:

1. Ввод информации
2. Память
3. Модуль принятия решений
4. Система реагирования на запросы

Ввод информации отвечает за то, чтобы объект воспринял все факты окружающего мира: местонахождение цели, наличие препятствий, звуки и т.д. Принимая во внимание внешние факторы объект, благодаря вычислениям, решает, что ему делать в следующий момент времени. NPC, патрулируя местность должен перемещаться по заранее указанным линиям, проверяя наличие объекта. Если он заметил врага, то алгоритмы заданной системы подают новый сигнал, говорящий о том что игрок попался в его поле обзора, и теперь нужно принимать заложенные программой действия. ИИ также должен правильно оценивать обстановку и принимать решение об отступлении или атаки. В роли ИИ иногда может выступать даже локация.

Глобальный тип применяется в стратегиях, где компьютер, например, управляет армией NPC, но все боты, находящиеся в группе, могут выполнять отдельные действия заданные алгоритмами, но в рамках глобального ИИ. То есть если пользователь ввел войска на определенную территорию и приказал им атаковать, то некоторые солдаты могут впасть бегство, так как состояние здоровья находится на минимуме.

Первый бот в видео игре был введен в 1951 году. Он создавался для игры в шахматы и представлял собой сравнительно простой список алгоритмов. Первые компьютерные игры шестидесятых и семидесятых в большинстве своем не имели интелекта и были созданы как игры для 2 игроков. Охота на вампуса - текстовый квест родом из 1972 года, созданный Грегори Йобом. В игре присутствовали враги с шаблонным списком действий, далеким от современных игр.

В 1980 году выходит всемирно известный pakman суть которого - съесть все точки на карте обходя активно перемещающихся привидений. Все привидения движется по определенному шаблону. В игре присутствует несколько видов привидений красные гонятся за игроком, повторяя его траекторию, розовые строят стратегические засады, голубые пугают игрока, заставляя сделать неправильный ход и оранжевый, стандартный бот, обычно не создающий много трудностей, в отличие от своих собратьев. Именно в этой игре ИИ был наиболее развитым, в сравнении с теми проектами которые выходили с ней в одно время. В девяностые годы начали массово выходить файтинги, начиная от mortal kombat и street fighter, заканчивая talking & soul caliber. Главная инновация - модернизация алгоритмов в реальном времени. NPC начинают искать альтернативные пути выхода из ситуации, анализируя ваши действия, что зачастую усложняет игру, хотя файтинги того времени, как и сейчас, были рассчитаны в основном на противостояние двух реальных игроков.

В 1998 году вышла игра golden eye 007 преподносящая геймеру довольно продвинутый искусственный интеллект. NPC активно реагировали на движение и стрельбу, прячась в укрытии, а так же враги отлично подбирали момент для метания гранат под ноги, но большой минус их разума был в том, что они точно знали местоположение игрока. Позднее вышедший far cry был очень хорош, так как NPC адаптировались под стиль игры пользователя. При обнаружении враги выслеживали вас и прочесывали местность, где видели главного героя в последний раз, в случае, если они его потеряли.

FEAR, вышедший в 2005 году, очень порадовал игроков и игровых аналитиков своим искусственным интеллектом. Отличало FEAR от других игр похожего типа превосходно продуманное поведение врагов. Главное преимущество над NPC других проектов заключалось в том, что они действовали сообща, тактично распределяясь между укрытиями, ставя игрока в неудобное положение. Это и является примером глобального ИИ, когда команда работает в одной связке в рамках указанных действий. ИИ справедливо назвать не искусственным интеллектом, а имитацией интеллекта, так как его поведение максимально условно и ограничивается рамками, заданными разработчиками.

Компания unity показала систему которая обучает персонажей с помощью двух установок: поощрение и наказание. С помощью unity email and gents toolkit они создали сцену в которою курица пытается пересечь оживленную дорогу, собирая поощрение в виде подарков, где удар машины - это наказание. Сначала ИИ проигрывал, но через шесть часов собирал 100 подарков без получения урона

Unity тестируют машинное обучение с помощью obstacle tower – игры, в которую могут играть только персонажи с искусственным интеллектом. Глава подразделения по изучению искусственного интеллекта unity Дэниел Ланге сравнивает это с эволюционным процессом: «Так работают дети. Работаем мы, и работают животные. В процессе обучения (игры) вы переходите от незнания к реальному пониманию».

Человек учится двумя способами: быстро на примерах и медленно, глубоко когда изучает теорию и абстракции. В комбинации оба способа помогают понять принцип задачи, чтобы потом решать новые. Это называется мета-обучением. Но у человеческого интеллекта этот процесс быстрее за счет такого гормона, как дофамин. Компания Deepmind пытается создать систему, которая будет совершать то или иное действие за вознаграждение, то есть за математический аналог дофамина. Так же они создали бота alpha star, который победил в starcraft 2 одного из сильнейших игроков мира. Для обучения alpha star устроили турнир среди нескольких нейросетей, обученных разным стратегиям и, побеждая каждую из них, alpha star улучшалась. В итоге с людьми нейросети уже справляются легко, то есть перспективы развития ИИ в том, чтобы в игры играл он, а не мы.

Играм, которые должны затянуть на многие часы, нужно много контента. Делать его вручную дорого и долго, а играя ИИ понимает устройство игр. Затем он сможет размещать объекты на уровне, автоматически анимировать героев, создавать фото реалистичные текстуры или вылавливать баги лучше тестировщиков.

исследователи из эдинбургского университета совместно с Method Studios создали нейросеть, которую «кормят» записями различных видов движения, благодаря чему персонаж привыкает к различной местности. Он буквально предсказывает координаты пространства. Подобная технология пригодится не только для анимации но и для освещения, цветокоррекции, движения камеры.

Обычно в играх два типа освещения: первый источник привязан персонажу, чтобы окружающая среда выглядела хорошо освещенной. А компания Naughty Dog расставляет фиксированное освещение всего уровня вручную, за счет чего их игры даже спустя несколько лет выглядят фактурно и кинематографично несмотря на устаревание графики. В итоге нейросеть можно будет просто накормить подходящими референсами из фильмов и она сама расставит освещение и покрасит все как надо.

Другой тип - генеративные состязательные сети то есть это 2 нейросети которые конкурируют друг с другом, например в увеличении качества текстур. Одна из них – генеративный алгоритм который пытается создать новое лучшего качества. А другая - дискриминационный алгоритм, который разбирает изображения на детали и раскладывает по ячейкам, к каждой из которых генеративный алгоритм создает новую картинку.

Исследователи из технологического института Джорджии пытаются создать ИИ, который сам делает игры. Они записали футажи первых уровней в super mario bros, Kirby’s Adventure и Mega Man. «Накормили» ими свой алгоритм, а потом заставили его реконструировать уровни и затем завершить игры. После этого они поручили ему создать свою игру.

Компания Spirit AI разрабатывает Character Engine - движок для создания персонажей, которые помнят и реагируют согласно своему характеру, тем самым продвигая сюжет. Движок анализирует не только смысл, но и грамматику фраз которые ввёл игрок. Подобно генеративным алгоритмам он подбирает подходящие по наполнению и настроению результаты. В итоге получаются непредсказуемые диалоги, близкие к настоящим. dos перед ой нарративные дизайнеры имели short работала над игрой blood in la rose на движке аверсу это интерактивный рассказ персонажи в игре имеют межличностные связи если нагрубить одному-то мнение о вашем герое изменится из-за того что тот персонаж на сплетничают другие на этапе тестирования бывало такое что другие персонажи чересчур грубили а сюжет оказывался непроходимым в итоге разработчики скале логические ошибки среди сотен возможных комбинаций вручную ведь система не построено на машинном обучении и не исправлял ошибки сама она картриджи на мели short выпустила пока одну игру rest was или скорее дымку возможности движка в это визуальный новелле ты оказываешь психологическую помощь призраком сценарий игры 17 тысяч слов которые складываются более чем в 200 миллионов комбинаций корректор н джон понимает контекст и по мере диалога сначала может дурачиться но постепенно подбирает более значительные фразы и плавно выводит игрока на серьезные эмоции согласно сюжету в социальных сетях чатах на форумах и стримах ошиваются мошенники боты для рассылки спама и рекламы кого-то хотят дискриминирует и домогаются при этом модератор человек целый день изучает самое плохое в сообществе это рутина которая к тому же может травмировать его психику на основе корректор н джонс перед и я сделали систему модерации по имени союзника на отбирается сортирует комментарии по группам в том числе позитивные во-первых модератор получает классификацию комментов и даже цепочек которые к ним привели это важно для контекста проблемы во вторых так модератору проще отобрать позитивный фидбэк чтобы понять тенденции вкус и сообщество очень красиво звучит ну вот этот пункт самый проблемный

Что такое ИИ?

Игровой ИИ сосредоточен на том, какие действия должен выполнять объект, исходя из условий, в которых находится. Обычно это называют управлением «интеллектуальными агентами», где агент является игровым персонажем, транспортным средством, ботом, а иногда и чем-то более абстрактным: целой группой сущностей или даже цивилизацией. В каждом случае это вещь, которая должна видеть свое окружение, принимать на его основе решения и действовать в соответствии с ними. Это называется циклом Sense/Think/Act (Чувствовать/Мыслить/Действовать):

* Sense: агент находит или получает информацию о вещах в своей среде, которые могут повлиять на его поведение (угрозы поблизости, предметы для сбора, интересные места для исследования).
* Think: агент решает, как реагировать (рассматривает, достаточно ли безопасно собирать предметы или сначала он должен сражаться/скрываться).
* Act: агент выполняет действия для реализации предыдущего решения (начинает движение к противнику или предмету).
* …теперь ситуация изменилась из-за действий персонажей, поэтому цикл повторяется с новыми данными.

Играм не нужна сложная система для извлечения информации, так как большая часть данных уже является ее неотъемлемой частью. Нет необходимости запускать алгоритмы распознавания изображений, чтобы определить, есть ли враг впереди — игра уже знает и передает сведения прямо в процессе принятия решений. Поэтому часть цикла Sense часто намного проще, чем Think и Act.

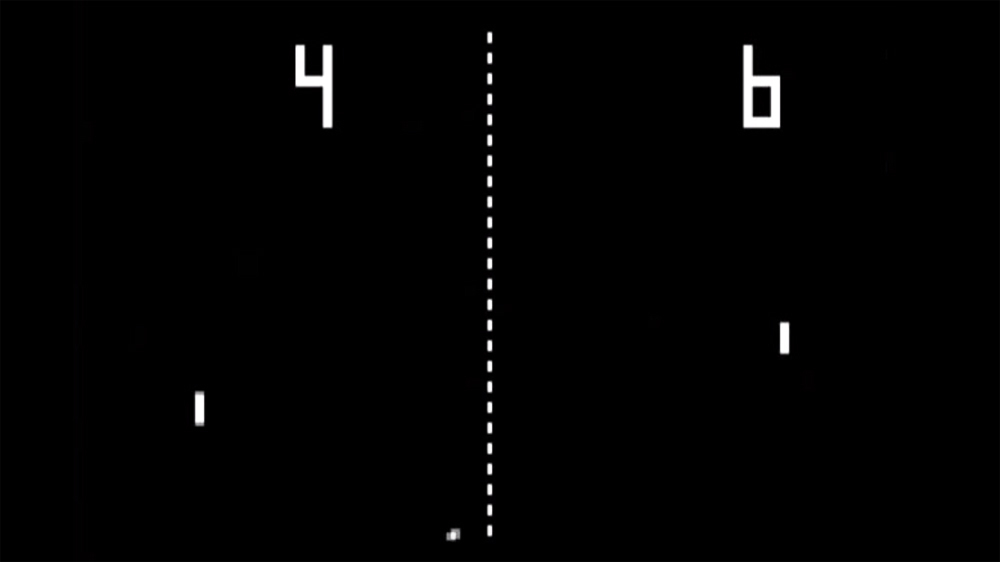
Ограничения игрового ИИ

У ИИ есть ряд ограничений, которые необходимо соблюдать:

* ИИ не нужно заранее тренировать, будто это алгоритм машинного обучения. Бессмысленно писать нейросеть во время разработки, чтобы наблюдать за десятками тысяч игроков и изучать лучший способ игры против них. Почему? Потому что игра не выпущена, а игроков нет.
* Игра должна развлекать и бросать вызов, поэтому агенты не должны находить лучший подход против людей.
* Агентам нужно выглядеть реалистичными, чтобы игроки чувствовали будто играют против настоящих людей. Программа AlphaGo превзошла человека, но выбранные шаги были сильно далеки от традиционного понимания игры. Если игра имитирует противника-человека, такого чувства не должно быть. Алгоритм нужно изменить, чтобы он принимал правдоподобные решения, а не идеальные.
* ИИ должен работать в реальном времени. Это значит, что алгоритм не может монополизировать использование процессора в течение длительного времени для принятия решений. Даже 10 миллисекунд на это — слишком долго, потому что большинству игр достаточно от 16 до 33 миллисекунд, чтобы выполнить всю обработку и перейти к следующему кадру графики.
* Идеально, если хотя бы часть системы управляется данными, чтобы «некодеры» могли вносить изменения, и чтобы корректировки происходили быстрее.

Подходы ИИ, охватывающие цикл Sense/Think/Act

Принятие базовых решений

Начнем с простейшей игры — Pong. Цель: переместить платформу (paddle) так, чтобы мяч отскакивал от нее, а не пролетал мимо. Это как теннис, в котором вы проигрываете, если не отбиваете мяч. Здесь у ИИ относительно легкая задача — решить, в каком направлении перемещать платформу.  
  


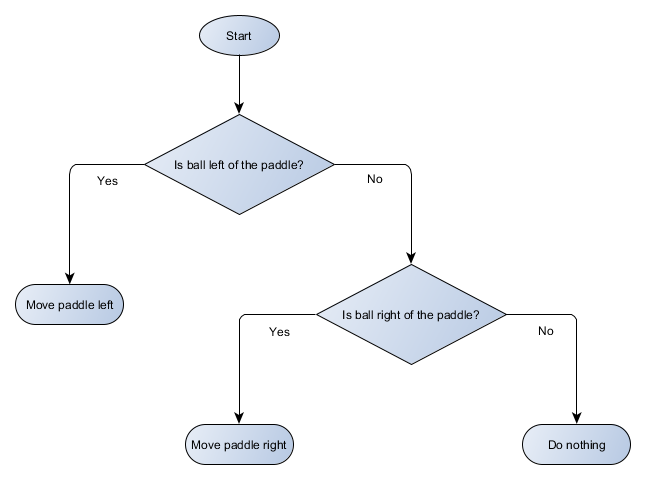
Условные операторы

Для ИИ в Pong есть самое очевидное решение — всегда стараться расположить платформу под мячом.  
  
Простой алгоритм для этого, написанный в псевдокоде:  
  
*every frame/update while the game is running:  
if the ball is to the left of the paddle:  
move paddle left  
else if the ball is to the right of the paddle:  
move paddle right*  
  
Если платформа двигается со скоростью мяча, то это идеальный алгоритм для ИИ в Pong. Не нужно ничего усложнять, если данных и возможных действий для агента не так уж и много.  
  
Этот подход настолько прост, что весь цикл Sense/Think/Act едва заметен. Но он есть:

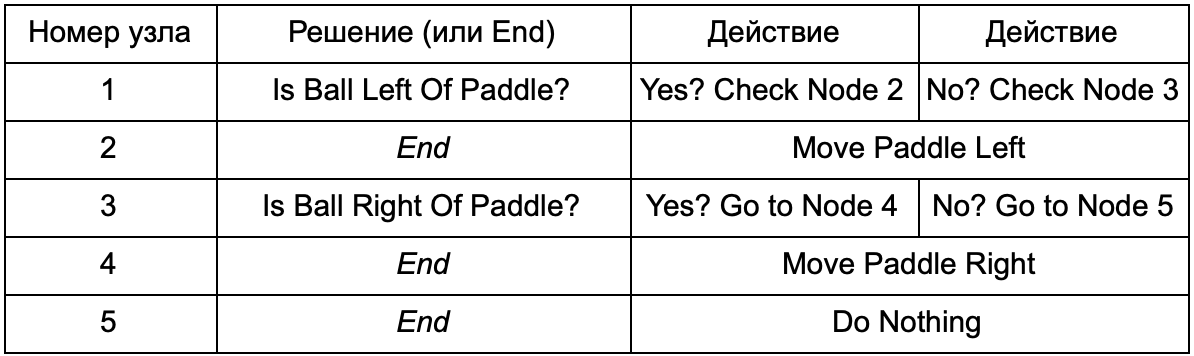
* Часть Sense находится в двух операторах if. Игра знает где мяч и где платформа, поэтому ИИ обращается к ней за этой информацией.
* Часть Think тоже входит в два оператора if. Они воплощают в себе два решения, которые в данном случае являются взаимоисключающими. В результате выбирается одно из трех действий — переместить платформу влево, переместить вправо, или ничего не делать, если она уже правильно расположена.
* Часть Act находится в операторах Move Paddle Left и Move Paddle Right. В зависимости от дизайна игры, они могут перемещать платформу мгновенно или с определенной скоростью.

Такие подходы называют реагирующими — есть простой набор правил (в данном случае операторы if в коде), которые реагируют на текущее состояние мира и действуют.

Дерево решений

Пример с игрой Pong фактически равен формальной концепции ИИ, называемой деревом решений. Алгоритм проходит его, чтобы достичь «листа» — решения о том, какое действие предпринять.  
  
Сделаем блок-схему дерева решений для алгоритма нашей платформы:  
  
  
  
Каждая часть дерева называется node (узел) — ИИ использует теорию графов для описания подобных структур. Есть два типа узлов:

* Узлы принятия решений: выбор между двумя альтернативами на основе проверки некоторого условия, где каждая альтернатива представлена в виде отдельного узла.
* Конечные узлы: действие для выполнения, представляющее окончательное решение.

Алгоритм начинается с первого узла («корня» дерева). Он либо принимает решение о том, к какому дочернему узлу перейти, либо выполняет действие, хранящееся в узле, и завершается.  
  
В чем же преимущество, если дерево решений, выполняет ту же работу, что и if-операторы в предыдущем разделе? Здесь есть общая система, где каждое решение имеет только одно условие и два возможных результата. Это позволяет разработчику создавать ИИ из данных, представляющих решения в дереве, избежав его хардкодинга. Представим в виде таблицы:  
  
  
  
На стороне кода вы получите систему для чтения строк. Создайте узел для каждой из них, подключите логику принятия решений на основе второго столбца и дочерние узлы на основе третьего и четвертого столбцов. Вам все еще нужно запрограммировать условия и действия, но теперь структура игры будет сложнее. В ней вы добавляете дополнительные решения и действия, а затем настраиваете весь ИИ, просто отредактировав текстовый файл с определением дерева. Далее передаете файл геймдизайнеру, который сможет изменить поведение без перекомпилирования игры и изменения кода.  
  
Деревья решений весьма полезны, когда они строятся автоматически на основе большого набора примеров (например, с использованием алгоритма ID3). Это делает их эффективным и высокопроизводительным инструментом для классификации ситуаций на основе получаемых данных. Однако мы выходим за рамки простой системы для выбора действий агентами.

Сценарии

Мы разобрали систему дерева решений, которая использовала заранее созданные условия и действия. Человек, проектирующий ИИ, может организовать дерево так, как хочет, но он все еще должен полагаться на кодера, который это все запрограммировал. Что если мы могли бы дать дизайнеру инструменты для создания собственных условий или действий?  
  
Чтобы программисту не писать код для условий Is Ball Left Of Paddle и Is Ball Right Of Paddle, он может сделать систему, в которой дизайнер будет записывать условия для проверки этих значений. Тогда данные дерева решений будут выглядеть так:  
  
  
  
По сути это то же самое, что и в первой таблице, но решения внутри себя имеют свой собственный код, немного похожий на условную часть if-оператора. На стороне кода это считывалось бы во втором столбце для узлов принятия решений, но вместо поиска конкретного условия для выполнения (Is Ball Left Of Paddle), оно оценивает условное выражение и возвращает true или false соответственно. Это делается с помощью скриптового языка Lua или Angelscript. С помощью них разработчик может принимать объекты в своей игре (ball и paddle) и создавать переменные, которые будут доступны в сценарии (ball.position). Кроме того, язык сценариев проще, чем C++. Он не требует полной стадии компиляции, поэтому идеально подходит для быстрой корректировки игровой логики и позволяет «некодерам» самим создавать нужные функции.  
  
В приведенном примере язык сценариев используется только для оценки условного выражения, но его также можно использовать и для действий. Например, данные Move Paddle Right, могут стать оператором сценария (ball.position.x += 10). Так, чтобы действие также определялось в скрипте, без необходимости программирования Move Paddle Right.  
  
Можно пойти еще дальше и полностью написать дерево решений на языке сценариев. Это будет код в виде жестко запрограммированных (hardcoded) условных операторов, но они будут находится во внешних файлах скрипта, то есть могут быть изменены без перекомпиляции всей программы. Зачастую можно изменить файл сценария прямо во время игры, чтобы быстро протестировать разные реакции ИИ.

Реагирование на события

Примеры выше идеально подходят к Pong. Они непрерывно запускают цикл Sense/Think/Act и действуют на основе последнего состояния мира. Но в более сложных играх нужно реагировать на отдельные события, а не оценивать все и сразу. Pong в таком случае уже неудачный пример. Выберем другой.  
  
Представьте шутер, где враги неподвижны пока не обнаружат игрока, после чего действуют в зависимости от своей «специализации»: кто-то побежит «рашить», кто-то будет атаковать издалека. Это все еще базовая реагирующая система — «если игрок замечен, то сделай что-нибудь», — но ее можно логически разделить на событие Player Seen (игрок замечен) и реакцию (выберите ответ и выполните его).  
  
Это возвращает нас к циклу Sense/Think/Act. Мы можем накодить Sense-часть, которая каждый кадр будет проверять — видит ли ИИ игрока. Если нет — ничего не происходит, но если видит, то создается событие Player Seen. У кода будет отдельный раздел, в котором говорится: «когда происходит событие Player Seen, сделай », где — ответ, который вам нужен для обращения к частям Think и Act. Таким образом, вы настроите реакции к событию Player Seen: на «рашущего» персонажа — ChargeAndAttack, а на снайпера — HideAndSnipe. Эти связи можно создать в файле данных для быстрого редактирования без необходимости заново компилировать. И здесь тоже можно использовать язык сценариев.

Принятие сложных решений

Хоть простые системы реакций очень действенны, бывает много ситуаций, когда их недостаточно. Иногда нужно принимать различные решения, основанные на том, что агент делает в настоящий момент, но представлять это за условие тяжело. Иногда существует слишком много условий, чтобы эффективно представить их в дереве решений или скрипте. Иногда нужно заранее оценивать, как изменится ситуация, прежде чем принимать решение о следующем шаге. Для решения этих проблем нужны более сложные подходы.

Finite state machine

Finite state machine или FSM (конечный автомат) — это способ сказать, что наш агент в настоящее время находится в одном из нескольких возможных состояний, и что он может переходить из одного состояния в другое. Таких состояний определённое количество — отсюда и название. Лучший пример из жизни — дорожный светофор. В разных местах разные последовательности огней, но принцип тот же — каждое состояние представляет что-то (стой, иди и т.д.). Светофор находится только в одном состоянии в любой момент времени, и переходит от одного к другому на основе простых правил.  
  
С NPC в играх похожая история. Для примера возьмем стража с такими состояниями:

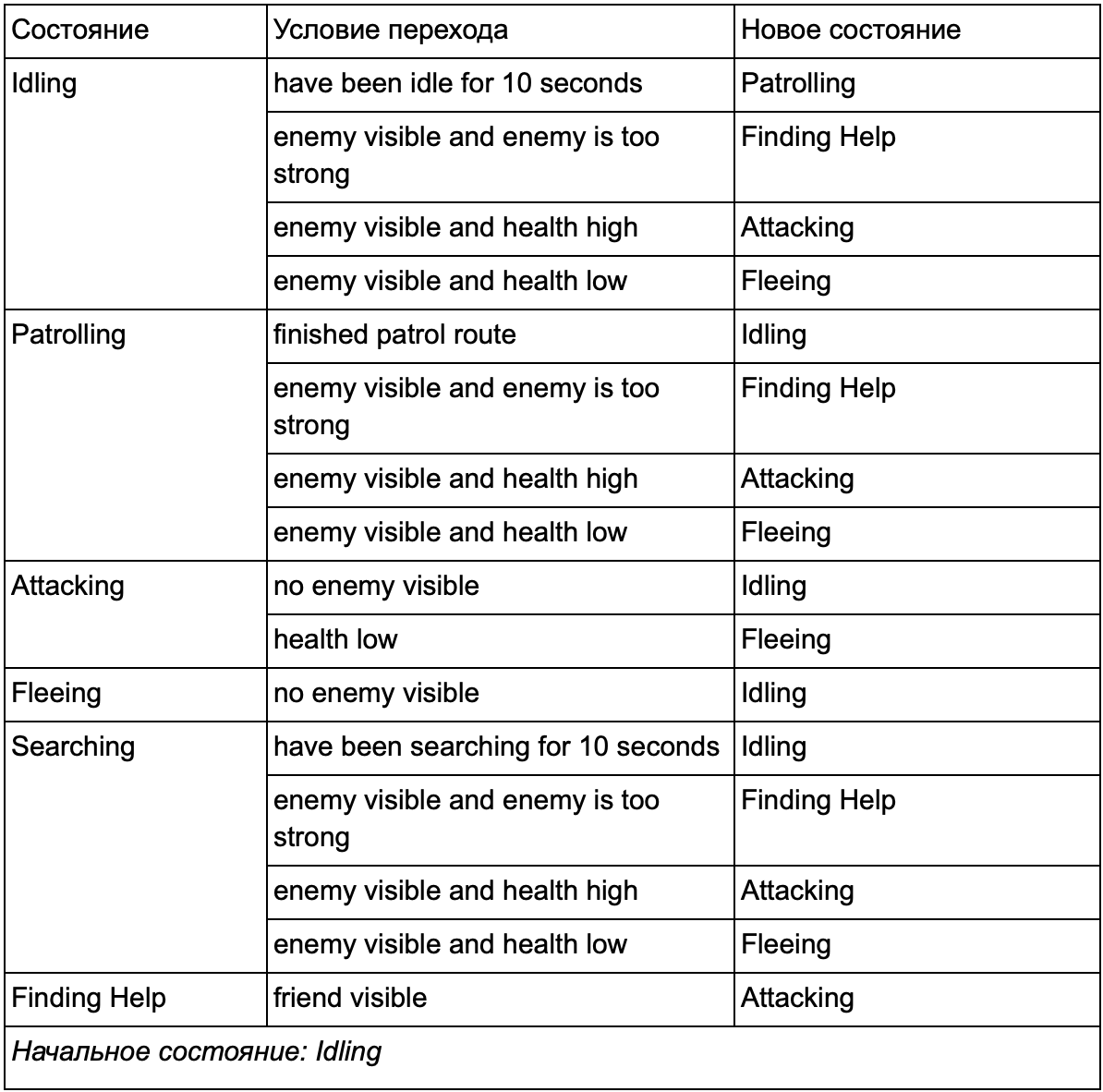
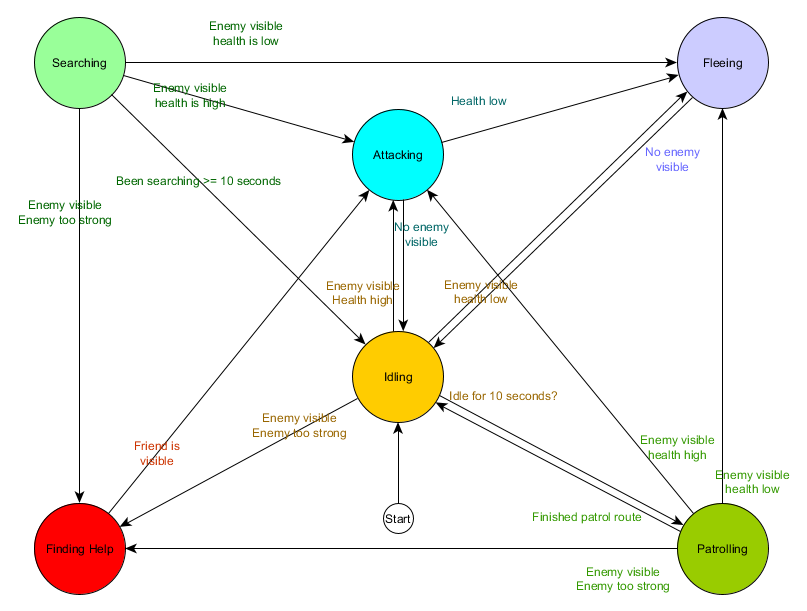
* Патрулирующий (Patrolling).
* Атакующий (Attacking).
* Убегающий (Fleeing).

И такими условиями для изменения его состояния:

* Если страж видит противника, он атакует.
* Если страж атакует, но больше не видит противника, он возвращается к патрулированию.
* Если страж атакует, но сильно ранен, он убегает.

Также можно написать if-операторы с переменной-состоянием стража и различные проверки: есть ли поблизости враг, какой уровень здоровья NPC и т. д. Добавим еще несколько состояний:

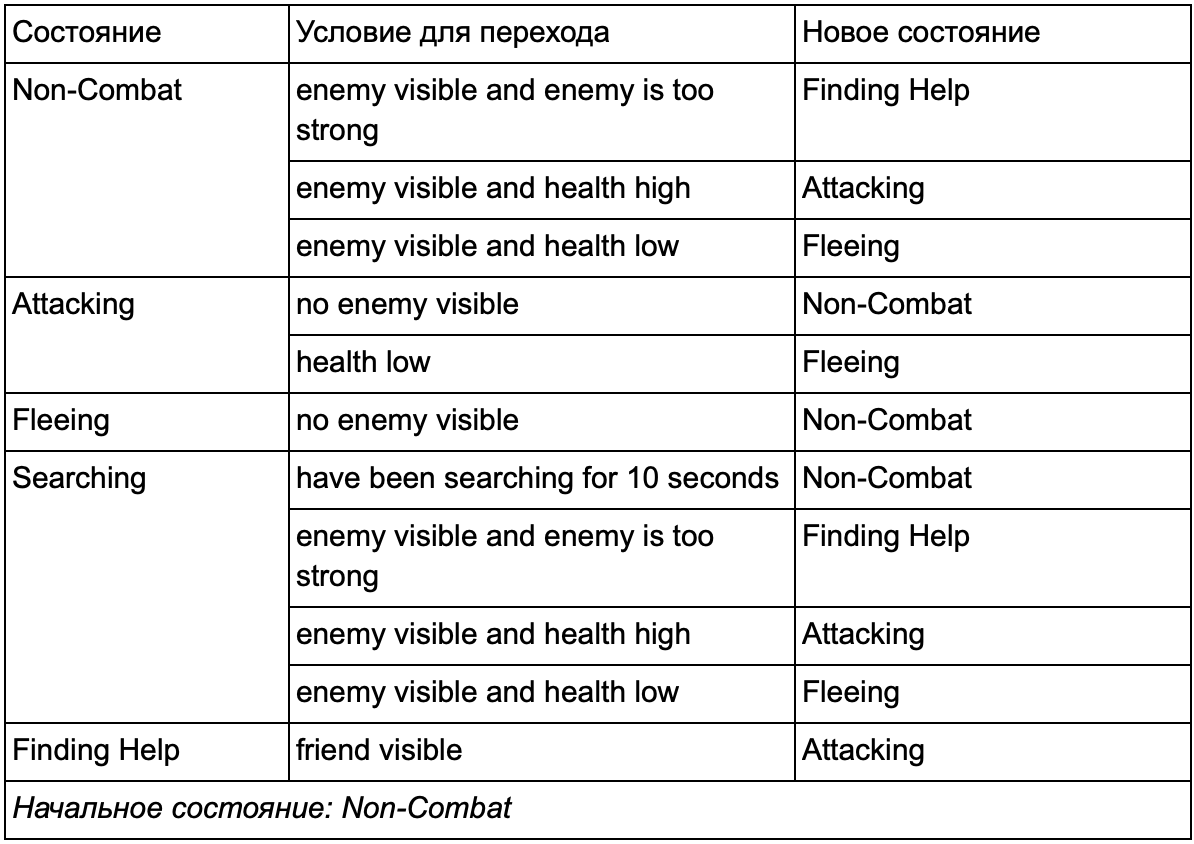
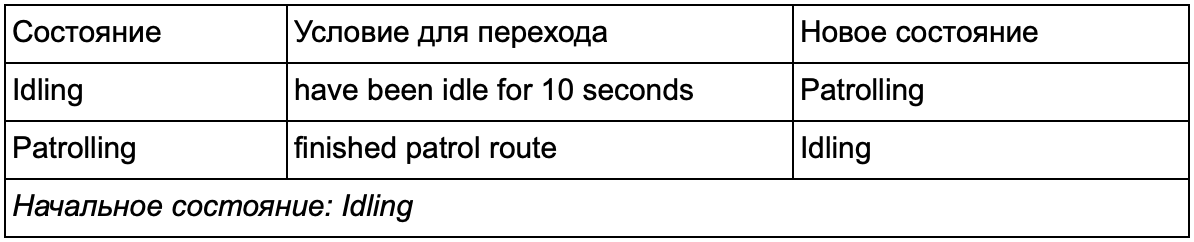
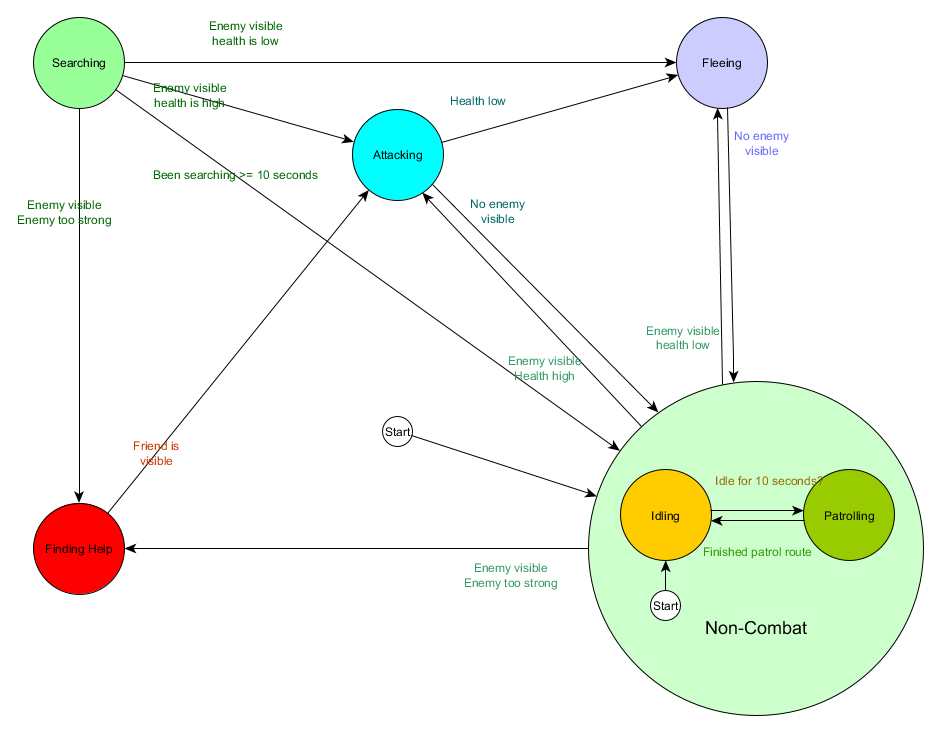
* Бездействие (Idling) — между патрулями.
* Поиск (Searching) — когда замеченный враг скрылся.
* Просить о помощи (Finding Help) — когда враг замечен, но слишком силен, чтобы сражаться с ним в одиночку.

Выбор для каждого из них ограничен — например, страж не пойдет искать скрывшегося врага, если у него низкое здоровье.  
  
В конце концов огромный список «если <x и y, но не z>, то <p>», может стать слишком громоздким, поэтому следует формализовать метод, который позволит нам держать в уме состояния и переходы между состояниями. Чтобы это сделать, примем во внимание все состояния, и под каждым состоянием запишем в список все переходы в другие состояния, вместе с необходимыми для них условиями.  
  
  
  
Это таблица переходов состояний — комплексный способ представления FSM. Нарисуем диаграмму и получим полный обзор того, как меняется поведение NPC.  
  
  
  
Диаграмма отражает суть принятия решений для этого агента на основе текущей ситуации. Причем каждая стрелка показывает переход между состояниями, если условие рядом с ней истинно.  
  
Каждое обновление мы проверяем текущее состояние агента, просматриваем список переходов, и если условия для перехода выполнены, он принимает новое состояние. Например, каждый кадр проверяется истек ли 10-секундный таймер, и если да, то из состояния Idling страж переходит в Patrolling. Таким же образом, состояние Attacking проверяет здоровье агента — если оно низкое, то он переходит в состояние Fleeing.  
  
Это обработка переходов между состояниями, но как насчет поведения, связанного с самими состояниями? Что касается реализации фактического поведения для конкретного состояния, обычно существует два типа «крюка», где мы присваиваем действия к FSM:

* Действия, которые мы периодически выполняем для текущего состояния.
* Действия, которые мы предпринимаем при переходе из одного состояния в другое.

Примеры для первого типа. Состояние Patrolling каждый кадр будет перемещать агента по маршруту патрулирования. Состояние Attacking каждый кадр будет пытаться начать атаку или перейти в состояние, когда это возможно.  
  
Для второго типа рассмотрим переход «если враг виден и враг слишком силен, то перейти в состояние Finding Help. Агент должен выбрать, куда пойти за помощью, и сохранить эту информацию, чтобы состояние Finding Help знало куда обратиться. Как только помощь найдена, агент переходит обратно в состояние Attacking. В этот момент он захочет рассказать союзнику об угрозе, поэтому может возникнуть действие NotifyFriendOfThreat.  
  
И снова мы можем посмотреть на эту систему через призму цикла Sense/Think/Act. Sense воплощается в данных, используемых логикой перехода. Think — переходами, доступными в каждом состоянии. А Act осуществляется действиями, совершаемыми периодически в пределах состояния или на переходах между состояниями.  
  
Иногда непрерывный опрос условий перехода может быть дорогостоящим. Например, если каждый агент будет выполнять сложные вычисления каждый кадр, чтобы определить видит ли он врагов и понять, можно ли переходить от состояния Patrolling к Attacking — это займет много времени процессора.  
  
Важные изменения в состоянии мира можно рассматривать как события, которые будут обрабатываться по мере их появления. Вместо того, чтобы FSM каждый кадр проверял условие перехода «может ли мой агент видеть игрока?», можно настроить отдельную систему, чтобы выполнять проверки реже (например, 5 раз в секунду). А результатом выдавать Player Seen, когда проверка проходит.  
  
Это передается в FSM, который теперь должен перейти в условие Player Seen event received и соответствующе отреагировать. Итоговое поведение одинаково за исключением почти незаметной задержки перед ответом. Зато производительность стала лучше в результате отделения части Sense в отдельную часть программы.

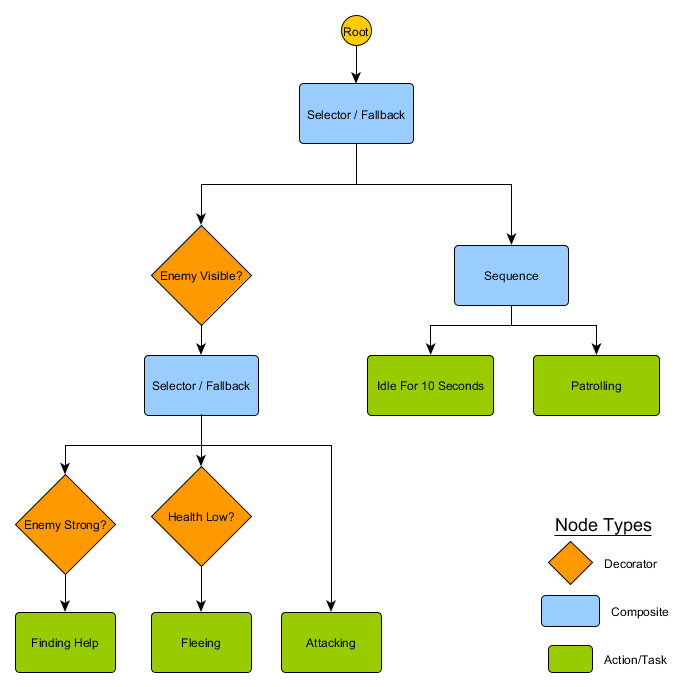
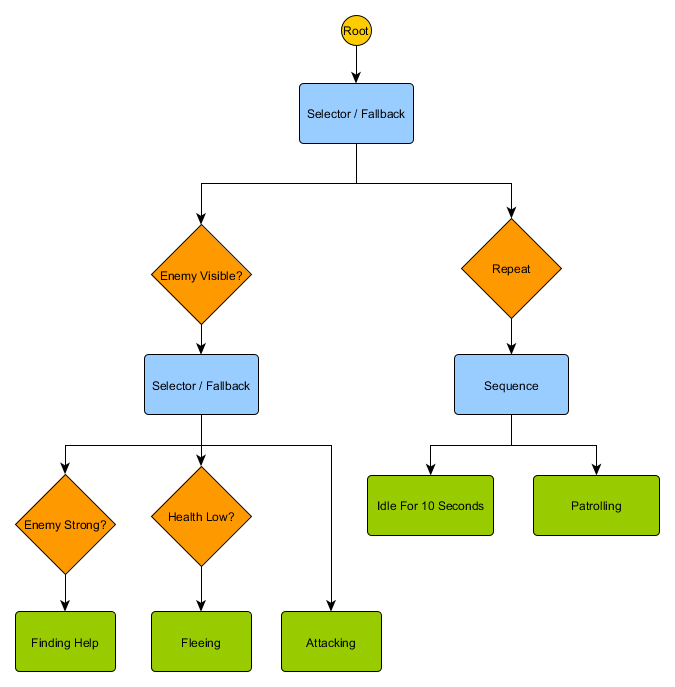
Hierarchical finite state machine

Однако работать с большими FSM не всегда удобно. Если мы захотим расширить состояние атаки, заменив его отдельными MeleeAttacking (ближний бой) и RangedAttacking (дальний бой), нам придется изменить переходы из всех других состояний, которые ведут в состояние Attacking (текущие и будущие).  
  
Наверняка вы заметили, что в нашем примере много дублированных переходов. Большинство переходов в состоянии Idling идентичны переходам в состоянии Patrolling. Хорошо бы не повторяться, особенно если мы добавим больше похожих состояний. Имеет смысл сгруппировать Idling и Patrolling под общим ярлыком «небоевые», где есть только один общий набор переходов в боевые состояния. Если мы представим этот ярлык как состояние, то Idling и Patrolling станут подсостояниями. Пример использования отдельной таблицы переходов для нового небоевого подсостояния:  
  
*Основные состояния:*  
  
  
*Состояние вне боя:*  
  
  
И в форме диаграммы:  
  
  
  
Это та же самая система, но с новым небоевым состоянием, которое включает в себя Idling и Patrolling. С каждым состоянием, содержащим FSM с подсостояниями (а эти подсостояния, в свою очередь, содержат собственные FSM — и так далее сколько вам нужно), мы получаем Hierarchical Finite State Machine или HFSM (иерархический конечный автомат). Сгруппировав небоевое состояние, мы вырезали кучу избыточных переходов. То же самое мы можем сделать для любых новых состояний с общими переходами. Например, если в будущем мы расширим состояние Attacking до состояний MeleeAttacking and MissileAttacking, они будут подсостояниями, переходящими между друг другом на основе расстояния до врага и наличия боеприпасов. В итоге сложные модели поведения и подмодели поведения можно представить с минимумом дублированных переходов.

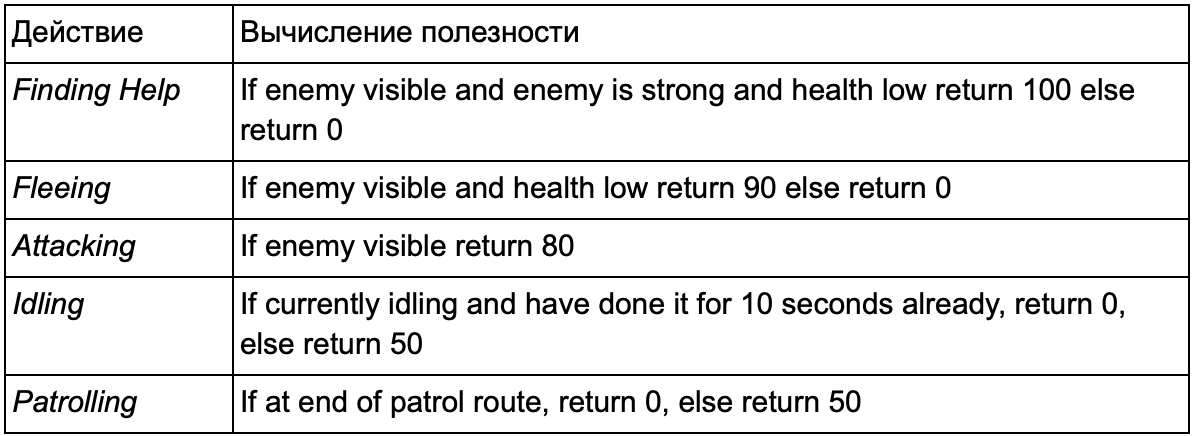
Дерево поведений

С HFSM создаются сложные комбинации поведений простым способом. Тем не менее, есть небольшая трудность, что принятие решений в виде правил перехода тесно связано с текущим состоянием. И во многих играх это как раз то, что нужно. А тщательное использование иерархии состояний может уменьшить количество повторов при переходе. Но иногда нужны правила, работающие независимо от того, в каком состоянии вы находитесь или которые применяются почти в любых состояниях. Например, если здоровье агента упало до 25%, вы захотите, чтобы он убегал независимо от того, был ли он в бою, бездельничал или разговаривал — вам придется добавлять это условие в каждое состояние. А если ваш дизайнер позже захочет изменить порог низкого здоровья с 25% до 10%, то этим снова придется заниматься.  
  
В идеале для этой ситуации нужна система, в которой решения «в каком состоянии пребывать», находятся за пределами самих состояний, чтобы вносить изменения только в одном месте и не трогать условия перехода. Здесь появляются деревья поведений.  
  
Существует несколько способов их реализации, но суть для всех примерно одинакова и похожа на дерево решений: алгоритм начинается с «корневого» узла, а в дереве находятся узлы, представляющие либо решения, либо действия. Правда есть несколько ключевых отличий:

* Теперь узлы возвращают одно из трех значений: Succeeded (если работа выполнена), Failed (если нельзя запустить) или Running (если она все еще запущена и нет конечного результата).
* Больше нет узлов решений для выбора между двумя альтернативами. Вместо них узлы Decorator, у которых есть один дочерний узел. Если они Succeed, то выполняют свой единственный дочерний узел.
* Узлы, выполняющие действия, возвращают значение Running для представления выполняемых действий.

Этот небольшой набор узлов можно объединить для создания большого количества сложных моделей поведения. Представим HFSM стража из предыдущего примера в виде дерева поведения:  
  
  
  
С этой структурой не должно быть явного перехода от состояний Idling/Patrolling к состоянию Attacking или любым другим. Если враг виден, а здоровье персонажа низкое, выполнение остановится на узле Fleeing, независимо от того, какой узел он ранее выполнял — Patrolling, Idling, Attacking или любой другой.  
  
  
  
Деревья поведений сложны — есть много способов их составить, да и найти правильное сочетание декораторов и составных узлов может быть проблематично. Есть также вопросы о том, как часто проверять дерево — мы хотим проходить его каждую часть или только тогда, когда одно из условий изменилось? Как хранить состояние, относящееся к узлам — как узнать, когда мы были в состоянии Idling в течение 10 секунд или как узнать, какие узлы выполнялись в прошлый раз, чтобы правильно обработать последовательность?  
  
Именно поэтому существует множество реализаций. Например, в некоторых системах узлы декоратора заменили встроенными декораторами. Они повторно оценивают дерево при изменении условий декоратора, помогают присоединиться к узлам и обеспечивают периодические обновления.

Utility-based system

У некоторых игр есть множество различных механик. Желательно, чтобы они получили все преимущества простых и общих правил перехода, но не обязательно в виде полного дерева поведения. Вместо того, чтобы иметь чёткий набор выборов или дерево возможных действий, проще изучить все действия и выбрать самое подходящее в данный момент.  
  
Utility-based system (система, основанная на полезности) как раз в этом поможет. Это система, где у агента есть множество действий, и он сам выбирает какое выполнить, основываясь на относительной полезности каждого. Где полезность — произвольная мера того, насколько важно или желательно выполнение этого действия для агента.  
  
Рассчитанную полезность действия на основе текущего состояния и среды, агент может проверить и выбрать наиболее подходящее другое состояние в любое время. Это похоже на FSM, за исключением того, где переходы определяются оценкой для каждого потенциального состояния, включая текущее. Обратите внимание, что мы выбираем самое полезное действие для перехода (или остаемся, если уже выполнили его). Для большего разнообразия это может быть взвешенный, но случайный выбор из небольшого списка.  
  
Система назначает произвольный диапазон значений полезности — например, от 0 (совершенно нежелательно) до 100 (полностью желательно). У каждого действия есть ряд параметров, влияющих на вычисление этого значения. Возвращаясь к нашему примеру со стражем:  
  
  
  
Переходы между действиями неоднозначны — любое состояние может следовать за любым другим. Приоритеты действий находятся в возвращаемых значениях полезности. Если враг виден, и этот враг силен, а здоровье персонажа низкое, то и Fleeing, и FindingHelp вернут высокие ненулевые значения. При этом FindingHelp всегда будет выше. Аналогичным образом, небоевые действия никогда не возвращают больше 50, поэтому они всегда будут ниже боевых. Нужно учитывать это при создании действий и вычислении их полезности.  
  
В нашем примере действия возвращают либо фиксированное постоянное значение, либо одно из двух фиксированных значений. Более реалистичная система предполагает возврат оценки из непрерывного диапазона значений. Например, действие Fleeing возвращает более высокие значения полезности, если здоровье агента низкое, а действие Attacking возвращает более низкие, если враг слишком силен. Из-за этого действие Fleeing имеет приоритет над Attacking в любой ситуации, когда агент чувствует, что у него недостаточно здоровья для победы над противником. Это позволяет изменять приоритеты действий на основе любого числа критериев, что делает такой подход более гибким и вариативным, чем дерево поведения или FSM.  
  
Каждое действие имеет много условий для расчета программы. Их можно написать на языке сценариев или в виде серии математических формул. В The Sims, которая моделирует распорядок дня персонажа, добавляют дополнительный уровень вычислений — агент получает ряд «мотиваций», влияющих на оценки полезности. Если персонаж голоден, то со временем он будет голодать еще сильнее, и результат полезности действия EatFood будет расти до тех пор, пока персонаж не выполнит его, снизив уровень голода, и вернув значение EatFood равным нулю.  
  
Идея выбора действий на основе системы оценок довольно проста, поэтому Utility-based system можно использовать как часть процессов принятия решений ИИ, а не как их полную замену. Дерево решений может запросить оценку полезности двух дочерних узлов и выбрать более высокую. Аналогичным образом, дерево поведения может иметь составной узел Utility для оценки полезности действий, чтобы решить, какой дочерний элемент выполнить.

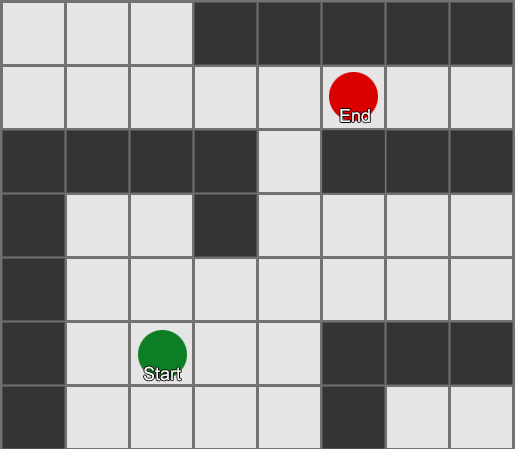
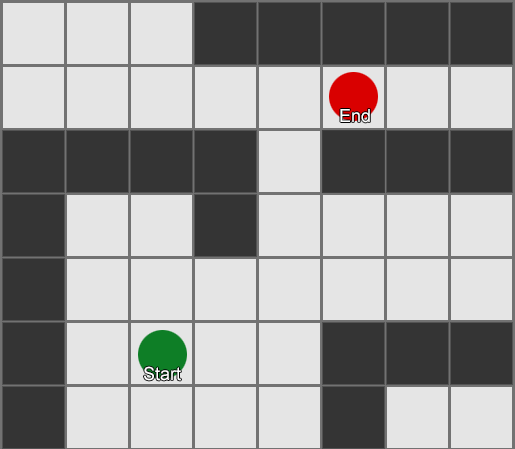
Движение и навигация

В предыдущих примерах у нас была платформа, которую мы перемещали влево или вправо, и страж, который патрулировал или атаковал. Но как именно мы обрабатываем перемещение агента в течение определенного периода времени? Как мы устанавливаем скорость, как мы избегаем препятствий, и как мы планируем маршрут, если добраться до места назначения сложнее, чем просто двигаться по прямой? Давайте это рассмотрим.

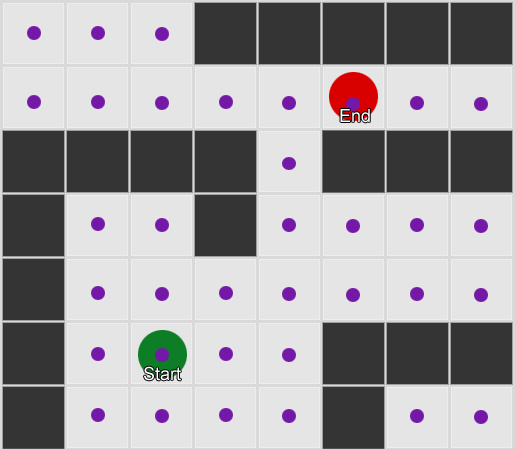
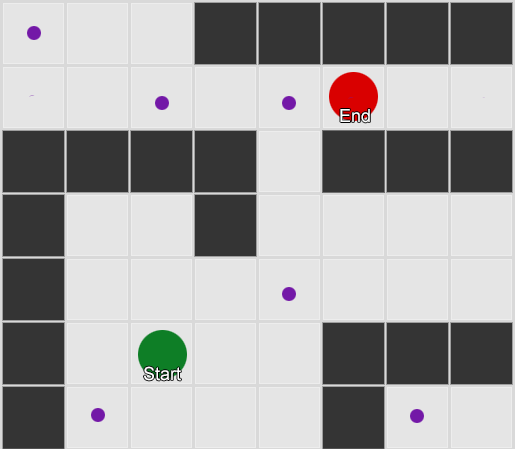
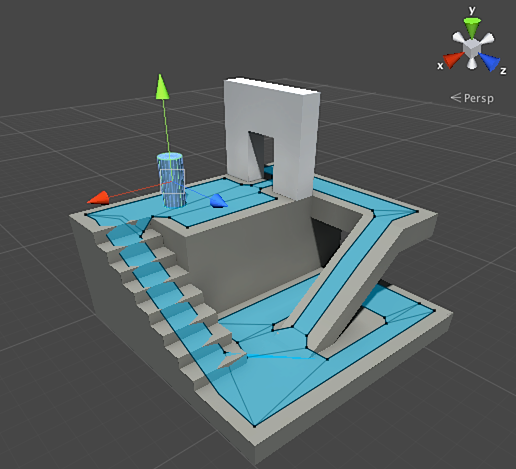
Управление

На начальном этапе будем считать, что каждый агент имеет значение скорости, которое включает в себя, как быстро он двигается и в каком направлении. Она может быть измерена в метрах в секунду, километрах в час, пикселей в секунду и т. д. Вспоминая цикл Sense/Think/Act, мы можем представить, что часть Think выбирает скорость, а часть Act применяет эту скорость к агенту. Обычно в играх есть физическая система, которая выполняет эту задачу за вас, изучая значение скорости каждого объекта и регулируя ее. Поэтому можно оставить ИИ с одной задачей — решить, какую скорость должен иметь агент. Если известно, где агент должен быть, то нужно переместить его в правильном направлении с установленной скоростью. Очень тривиальное уравнение:  
  
*desired\_travel = destination\_position – agent\_position*  
  
Представьте 2D-мир. Агент находится в точке (-2,-2), пункт назначения где-то на северо-востоке в точке (30, 20), а необходимый путь для агента, чтобы оказаться там — (32, 22). Допустим, эти позиции измеряются в метрах — если принять скорость агента за 5 метров в секунду, то мы будем масштабировать наш вектор перемещения и получим скорость примерно (4.12, 2.83). С этими параметрами агент прибыл бы к месту назначения почти через 8 секунд.  
  
Пересчитать значения можно в любое время. Если агент был на полпути к цели, перемещение было бы половиной длины, но так как максимальная скорость агента 5 м/с (мы это решили выше), скорость будет одинаковой. Это также работает для движущихся целей, позволяя агенту вносить небольшие изменения по мере их перемещения.  
  
Но мы хотим больше вариативности — например, медленно нарастить скорость, чтобы симулировать персонажа, движущегося из стоячего состояния и переходящего к бегу. Тоже самое можно сделать в конце перед остановкой. Эти фичи известны как steering behaviours, каждое из которых имеет конкретные имена: Seek (поиск), Flee (бегство), Arrival (прибытие) и т. д. Идея заключается в том, что силы ускорения могут быть применены к скорости агента, на основе сравнения положения агента и текущей скорости с пунктом назначения, чтобы использовать различные способы перемещения к цели.  
  
Каждое поведение имеет немного другую цель. Seek и Arrival — это способы перемещения агента к точке назначения. Obstacle Avoidance (избегание препятствий) и Separation (разделение) корректируют движение агента, чтобы обходить препятствия на пути к цели. Alignment (согласование) и Cohesion (связь) держат агентов при перемещении вместе. Любое число различных steering behaviours может быть суммировано для получения одного вектора пути с учётом всех факторов. Агент, использующий поведения Arrival, Separation и Obstacle Avoidance, чтобы держаться подальше от стен и других агентов. Этот подход хорошо работает в открытых локациях без лишних деталей.  
  
В более трудных условиях, сложение разных поведений работает хуже — к примеру, агент может застрять в стене из-за конфликта Arrival и Obstacle Avoidance. Поэтому нужно рассматривать варианты, которые сложнее, чем просто сложение всех значений. Способ такой: вместо сложения результатов каждого поведения, можно рассмотреть движение в разных направлениях и выбрать лучший вариант.  
  
Однако в сложной среде с тупиками и выбором, в какую сторону идти, нам понадобится что-то ещё более продвинутое.

Поиск пути

Steering behaviours отлично подходит для простого движения на открытой местности (футбольное поле или арена), где добраться от А до Б — это прямой путь с небольшими отклонениями мимо препятствий. Для сложных маршрутов нам нужен pathfinding (поиск пути), который является способом изучения мира и принятия решения о маршруте через него.  
  
Самый простой — наложить сетку на каждый квадрат рядом с агентом и оценить в каких из них разрешено двигаться. Если какой-то из них является пунктом назначения, то следуйте из него по маршруту от каждого квадрата к предыдущему, пока не дойдете до начала. Это и есть маршрут. В противном случае повторяйте процесс с ближайшими другими квадратами, пока не найдете место назначения или не закончатся квадраты (это означает, что нет никакого возможного маршрута). Это то, что формально известно как Breadth-First Search или BFS (алгоритм поиска в ширину). На каждом шаге он смотрит во всех направлениях (поэтому breadth, «ширина»). Пространство поиска похоже на волновой фронт, который перемещается, пока не достигнет искомое место — область поиска расширяется на каждом шаге до тех пор, пока в нее не попадет конечная точка, после чего можно отследить путь к началу.  
  
  
  
В результате вы получите список квадратов, по которым составляется нужный маршрут. Это и есть путь (отсюда, pathfinding) — список мест, которые агент посетит, следуя к пункту назначения.  
  
Учитывая, что мы знаем положение каждого квадрата в мире, можно использовать steering behaviours, чтобы двигаться по пути — от узла 1 к узлу 2, затем от узла 2 к узлу 3 и так далее. Простейший вариант — направиться к центру следующего квадрата, но еще лучше — остановиться на середине грани между текущим квадратом и следующим. Из-за этого агент сможет срезать углы на крутых поворотах.  
  
У алгоритма BFS есть и минусы — он исследует столько же квадратов в «неправильном» направлении, сколько в «правильном». Здесь появляется более сложный алгоритм под названием A\* (A star). Он работает также, но вместо слепого изучения квадратов-соседей (затем соседей соседей, затем соседей соседей соседей и так далее), он собирает узлы в список и сортирует их так, что следующий исследуемый узел всегда тот, который приведет к кратчайшему маршруту. Узлы сортируются на основе эвристики, которая учитывает две вещи — «стоимость» гипотетического маршрута к нужному квадрату (включая любые затраты на перемещение) и оценку того, насколько далеко этот квадрат от места назначения (смещая поиск в правильном направлении).  
  
  
  
В этом примере показано, что агент исследует по одному квадрату за раз, каждый раз выбирая соседний, который является самым перспективным. Полученный путь такой же, как и при BFS, но в процессе было рассмотрено меньше квадратов — а это имеет большое значение для производительности игры.

Движение без сетки

Но большинство игр не выложены на сетке, и зачастую ее невозможно сделать без ущерба реалистичности. Нужны компромиссы. Каких размеров должны быть квадраты? Слишком большими — и они не смогут правильно представить небольшие коридоры или повороты, слишком маленькими — будет чрезмерно много квадратов для поиска, который в итоге займет кучу времени.  
  
Первое, что нужно понять — сетка дает нам граф связанных узлов. Алгоритмы A\* и BFS фактически работают на графиках и вообще не заботятся о нашей сетке. Мы могли бы поставить узлы в любых местах игрового мира: при наличии связи между любыми двумя соединенными узлами, а также между начальной и конечной точкой и по крайней мере одним из узлов — алгоритм будет работать также хорошо, как и раньше. Часто это называют системой путевых точек (waypoint), так как каждый узел представляет собой значимую позицию в мире, которая может быть частью любого количества гипотетических путей.  
  
  
*Пример 1: узел в каждом квадрате. Поиск начинается из узла, в котором находится агент, и заканчивается в узле нужного квадрата.*  
  
  
*Пример 2: меньший набор узлов (путевых точек). Поиск начинается в квадрате с агентом, проходит через необходимое количество узлов, и затем продолжается до пункта назначения.*  
  
Это вполне гибкая и мощная система. Но нужна некоторая осторожность в решениях, где и как поместить waypoint, иначе агенты могут просто не увидеть ближайшую точку и не смогут начать путь. Было бы проще, если мы могли автоматически расставить путевые точки на основе геометрии мира.  
  
Тут появляется navigation mesh или navmesh (навигационная сетка). Это обычно 2D-сетка треугольников, которая накладывается на геометрию мира — везде, где агенту разрешено ходить. Каждый из треугольников в сетке становится узлом в графе и имеет до трех смежных треугольников, которые становятся соседними узлами в графе.  
  
Эта картина является примером из движка Unity — он проанализировал геометрию в мире и создал navmesh (на скриншоте светло-голубым цветом). Каждый полигон в navmesh — это область, на которой агент может стоять или перемещаться из одного полигона в другой полигон. В данном примере полигоны меньше этажей, на которых они расположены — сделано для того, чтобы учесть размеры агента, которые будут выходить за пределы его номинального положения.  
  
  
  
Мы можем искать маршрут через эту сетку, снова используя алгоритм A\*. Это даст нам практически идеальный маршрут в мире, который учитывает всю геометрию и при этом не требует лишних узлов и создания путевых точек.  
  
Pathfinding — слишком обширная тема, о которой мало одного раздела статьи. Если хотите изучить ее более подробно, то в этом поможет [сайт Амита Пателя](https://www.redblobgames.com/pathfinding/a-star/introduction.html).

Планирование

Мы убедились с pathfinding, что иногда недостаточно просто выбрать направление и двигаться — мы должны выбрать маршрут и сделать несколько поворотов, чтобы добраться до нужного места назначения. Мы можем обобщить эту идею: достижение цели это не просто следующий шаг, а целая последовательность, где иной раз требуется заглянуть вперед на несколько шагов, чтобы узнать, каким должен быть первый. Это называется планированием. Pathfinding можно рассматривать как одно из нескольких дополнений планирования. С точки зрения нашего цикла Sense/Think/Act, это то, где часть Think планирует несколько частей Act на будущее.  
  
Разберем на примере настольной игры Magic: The Gathering. Мы ходим первыми с таким набором карт на руках:

* Swamp — дает 1 черную ману (карта земли).
* Forest — дает 1 зеленую ману (карта земли).
* Fugitive Wizard — требует 1 синию ману для призыва.
* Elvish Mystic — требует 1 зеленую ману для призыва.

Оставшиеся три карты игнорируем, чтобы было проще. По правилам игроку разрешено играть 1 карту земли за ход, он может «тапнуть» эту карту, чтобы извлечь из нее ману, а затем использовать заклинания (включая вызов существа) по количеству маны. В этой ситуации игрок-человек знает, что нужно играть Forest, «тапнуть» 1 зеленую ману, а затем вызвать Elvish Mystic. Но как об этом догадаться игровому ИИ?

Простое планирование

Тривиальный подход — пробовать каждое действие по очереди, пока не останется подходящих. Глядя на карты, ИИ видит, что может сыграть Swamp. И играет его. Остались ли другие действия на этом ходу? Он не может вызвать ни Elvish Mystic, ни Fugitive Wizard, поскольку для их призыва требуется соответственно зеленая и синяя мана, а Swamp дает только черную ману. И он уже не сможет играть Forest, потому что уже сыграл Swamp. Таким образом, игровой ИИ сходил по правилам, но сделал это плохо. Можно улучшить.  
  
Планирование может найти список действий, которые приводят игру в желаемое состояние. Также, как каждая квадрат на пути имел соседей (в pathfinding), каждое действие в плане тоже имеет соседей или преемников. Мы можем искать эти действия и последующие действия, пока не достигнем желаемого состояния.  
  
В нашем примере, желаемый результат «вызвать существо, если это возможно». В начале хода мы видим только два возможных действия, разрешенных правилами игры:  
  
*1. Сыграть Swamp (результат: Swamp в игре)  
2. Сыграть Forest (результат: Forest в игре)*  
  
Каждое принятое действие может привести к дальнейшим действиям и закрыть другие, опять же в зависимости от правил игры. Представьте, что мы сыграли Swamp — это удалит Swamp в качестве следующего шага (мы его уже сыграли), также это удалит и Forest (потому что по правилам можно сыграть одну карту земли за ход). После этого ИИ добавляет в качестве следующего шага — получение 1 черной маны, потому что других вариантов нет. Если он пойдет дальше и выберет Tap the Swamp, то получит 1 единицу черной маны и ничего с ней не сможет сделать.  
  
*1. Сыграть Swamp (результат: Swamp в игре)  
1.1 «Тапнуть» Swamp (результат: Swamp «тапнута», +1 единица черной маны)  
Нет доступных действий – КОНЕЦ  
2. Сыграть Forest (результат: Forest в игре)*  
  
Список действий вышел коротким, мы зашли в тупик. Повторяем процесс для следующего действия. Мы играем Forest, открываем действие «получить 1 зеленую ману», которая в свою очередь откроет третье действие — призыв Elvish Mystic.  
  
*1. Сыграть Swamp (результат: Swamp в игре)  
1.1 «Тапнуть» Swamp (результат: Swamp «тапнута», +1 единица черной маны)  
Нет доступных действий – КОНЕЦ  
2. Сыграть Forest (результат: Forest в игре)  
2.1 «Тапнуть» Forest (результат: Forest «тапнута», +1 единица зеленой маны)  
2.1.1 Призвать Elvish Mystic (результат: Elvish Mystic в игре, -1 единица зеленой маны)  
Нет доступных действий – КОНЕЦ*  
  
Наконец, мы изучили все возможные действия и нашли план, призывающий существо.  
  
Это очень упрощенный пример. Желательно выбирать лучший возможный план, а не любой, который соответствует каким-то критериям. Как правило, можно оценить потенциальные планы на основе конечного результата или совокупной выгоды от их выполнения. Можно начислить себе 1 очко за игру карты земли и 3 очка за вызов существа. Играть Swamp было бы планом, дающим 1 очко. А сыграть Forest → Tap the Forest → призвать Elvish Mystic — сразу даст 4 очка.  
  
Вот так работает планирование в Magic: The Gathering, но по той же логике это применяется и в других ситуациях. Например, переместить пешку, чтобы освободить место для хода слона в шахматах. Или укрыться за стеной, чтобы безопасно стрелять в XCOM так. В общем, вы поняли суть.

Улучшенное планирование

Иногда бывает слишком много потенциальных действий, чтобы рассматривать каждый возможный вариант. Возвращаясь к примеру с Magic: The Gathering: допустим, что в игре и на у вас на руках по несколько карт земли и существ — количество возможных комбинаций ходов может исчисляться десятками. Есть несколько решений проблемы.  
  
Первый способ — backwards chaining (обратное формирование цепи). Вместо перебора всех комбинаций, лучше начать с итогового результата и попробовать найти прямой маршрут. Вместо пути от корня дерева к определенному листу, мы двигаемся в обратном направлении — от листа к корню. Этот способ проще и быстрее.  
  
Если у противника 1 единица здоровья, можно найти план «нанести 1 или более единиц урона». Чтобы добиться этого нужно выполнить ряд условий:  
  
1. Урон может нанести заклинание — оно должно быть в руке.  
2. Чтобы разыграть заклинание — нужна мана.  
3. Чтобы получить ману — нужно разыграть карту земли.  
4. Чтобы разыграть карту земли — нужно иметь ее в руке.  
  
Другой способ — best-first search (наилучший первый поиск). Вместо перебора всех путей, мы выбираем наиболее подходящий. Чаще всего этот способ даёт оптимальный план без лишних затрат на поиски. A\* — это форма наилучшего первого поиска — исследуя наиболее перспективные маршруты с самого начала, он уже может найти наилучший путь без необходимости проверять остальные варианты.  
  
Интересным и все более популярным вариантом best-first search является Monte Carlo Tree Search. Вместо угадывания, какие планы лучше других при выборе каждого последующего действия, алгоритм выбирает случайных преемников на каждом шаге, пока не достигнет конца (когда план привел к победе или поражению). Затем итоговый результат используется для повышения или понижения оценки «веса» предыдущих вариантов. Повторяя этот процесс несколько раз подряд, алгоритм дает хорошую оценку того, какой следующий шаг лучше, даже если ситуация изменится (если противник примет меры, чтобы помешать игроку).  
  
В рассказе о планировании в играх не обойдется без Goal-Oriented Action Planning или GOAP (целенаправленное планирование действий). Это широко используемый и обсуждаемый метод, но помимо нескольких отличительных деталей это, по сути, метод backwards chaining, о котором мы говорили ранее. Если задача была «уничтожить игрока», и игрок находится за укрытием, план может быть таким: уничтожь гранатой → достань ее → брось.  
  
Обычно существует несколько целей, каждая со своим приоритетом. Если цель с наивысшим приоритетом не может быть выполнена (ни одна комбинация действий не создает план «уничтожить игрока», потому что игрок не виден), ИИ вернется к целям с более низким приоритетом.

Обучение и адаптация

Мы уже говорили, что игровой ИИ обычно не использует машинное обучение, потому что это не подходит для управления агентами в реальном времени. Но это не значит, что нельзя что-нибудь позаимствовать из этой области. Мы хотим такого противника в шутере, у которого можно чему-нибудь научиться. Например, узнать о лучших позициях на карте. Или противника в файтинге, который блокировал бы часто используемые игроком комбо-приемы, мотивируя использовать другие. Так что машинное обучение в таких ситуациях может быть весьма полезно.

Статистика и вероятности

Прежде чем мы перейдем к сложным примерам, прикинем, как далеко мы можем зайти, взяв несколько простых измерений и используя их для принятия решений. К примеру, стратегия в реальном времени — как нам определить сможет ли игрок начать атаку в первые несколько минут игры и какую оборону против этого приготовить? Мы можем изучить прошлый опыт игрока, чтобы понять, какой может быть будущая реакция. Начнем с того, что у нас нет таких исходных данных, но мы их можем собрать — каждый раз, когда ИИ играет против человека, он может записывать время первой атаки. Спустя несколько сессий мы получим среднее значение времени, через которое игрок будет атаковать в будущем.  
  
У средних значений есть и проблема: если игрок 20 раз «рашил», а 20 раз играл медленно, то нужные значения будут где-то в середине, а это ничего полезного нам не даст. Одним из решений является ограничение входных данных — можно учитывать последние 20 штук.  
  
Аналогичный подход используется при оценке вероятности определенных действий, предполагая, что прошлые предпочтения игрока будут такими же в будущем. Если игрок атакует нас пять раз фаерболом, два раза молнией и один раз врукопашную, очевидно, что он предпочитает фаербол. Экстраполируем и увидим вероятность использования различного оружия: фаербол=62,5%, молния=25% и рукопашная=12,5%. Нашему игровому ИИ нужно подготовиться к защите от огня.  
  
Еще один интересный метод — использовать Naive Bayes Classifier (наивный байесовский классификатор) для изучения больших объемов входных данных и классифицировать ситуацию, чтобы ИИ реагировал нужным образом. Байесовские классификаторы наиболее известны за использование в фильтрах спама электронной почты. Там они исследуют слова, сравнивают их с тем, где появлялись эти слова ранее (в спаме или нет), и делают выводы о входящих письмах. Мы можем сделать то же самое даже с меньшим количеством входных данных. На основе всей полезной информации, которую видит ИИ (например, какие вражеские юниты созданы, или какие заклинания они используют, или какие технологии они исследовали), и итогового результата (война или мир, «рашить» или обороняться и т. д.) — мы выберем нужное поведение ИИ.  
  
Всех этих способов обучения достаточно, но желательно использовать их на основе данных из тестирования. ИИ научится адаптироваться к различным стратегиям, которые использовали ваши плэйтестеры. ИИ, который адаптируется к игроку после релиза, может стать слишком предсказуемым или наоборот слишком сложным для победы.

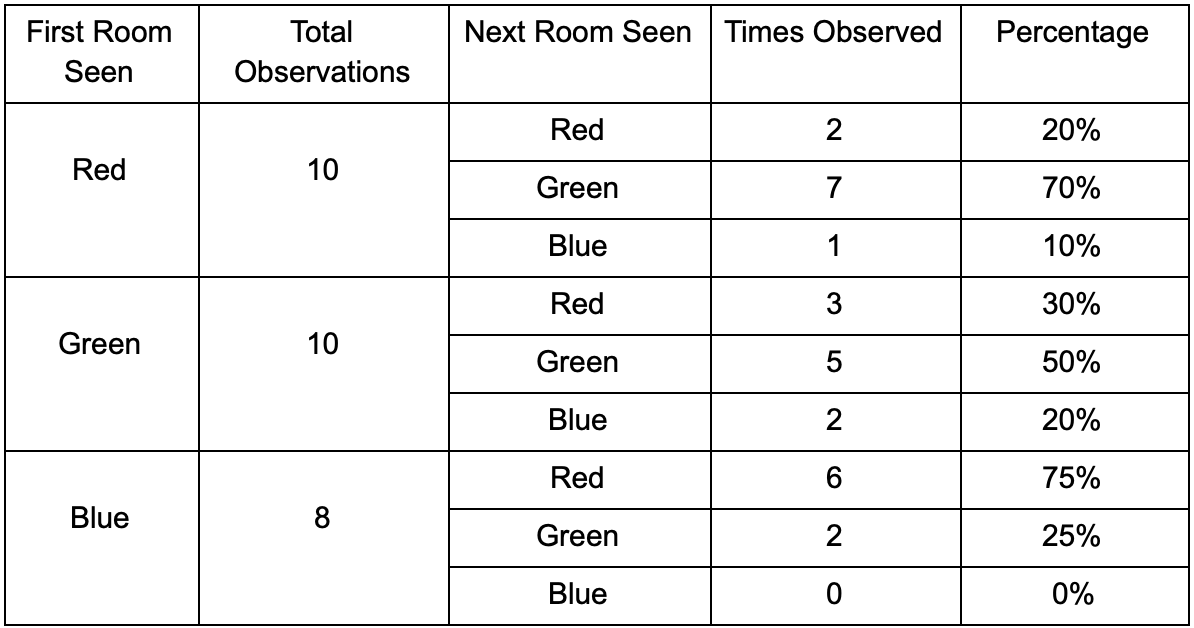
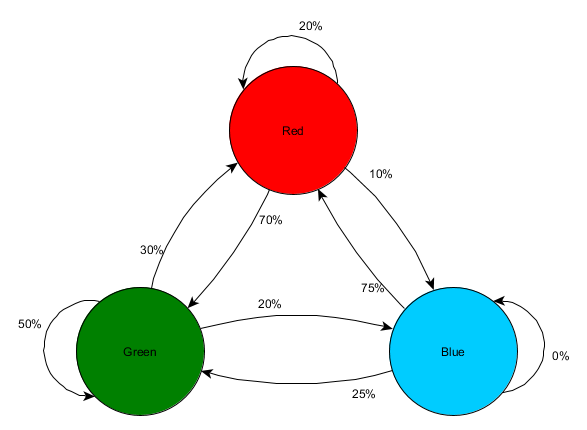
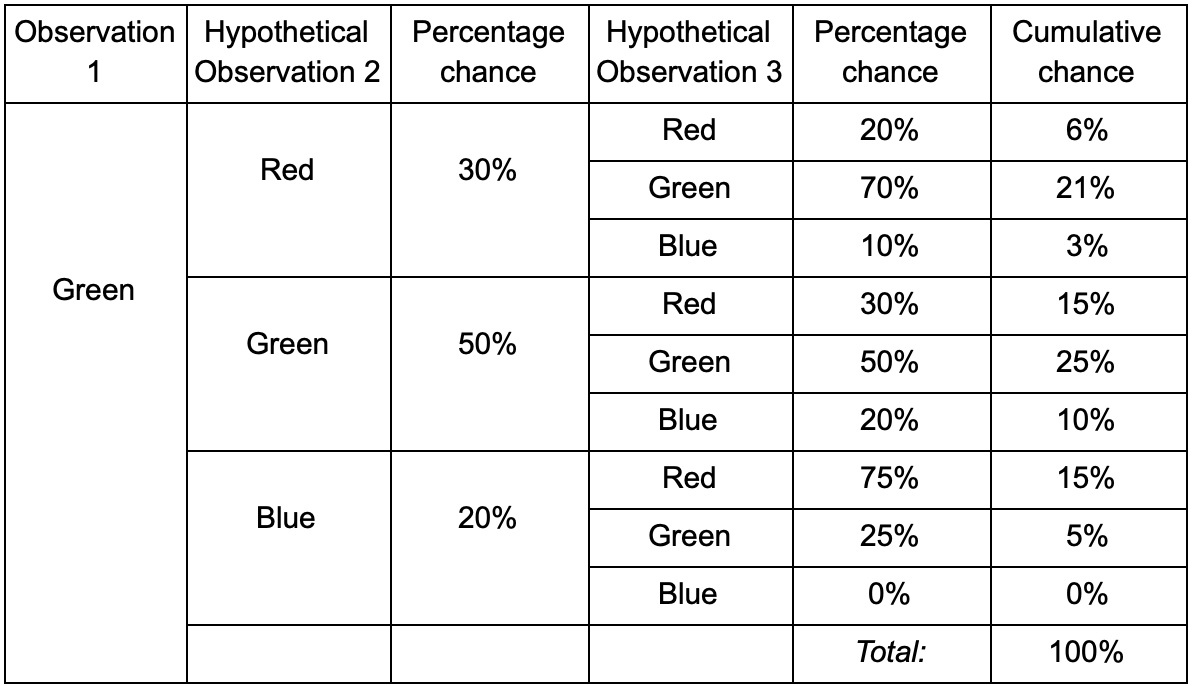
Адаптация на основе значений

Учитывая наполнение нашего игрового мира и правил, мы можем изменить набор значений, которые влияют на принятие решений, а не просто использовать входные данные. Делаем так:

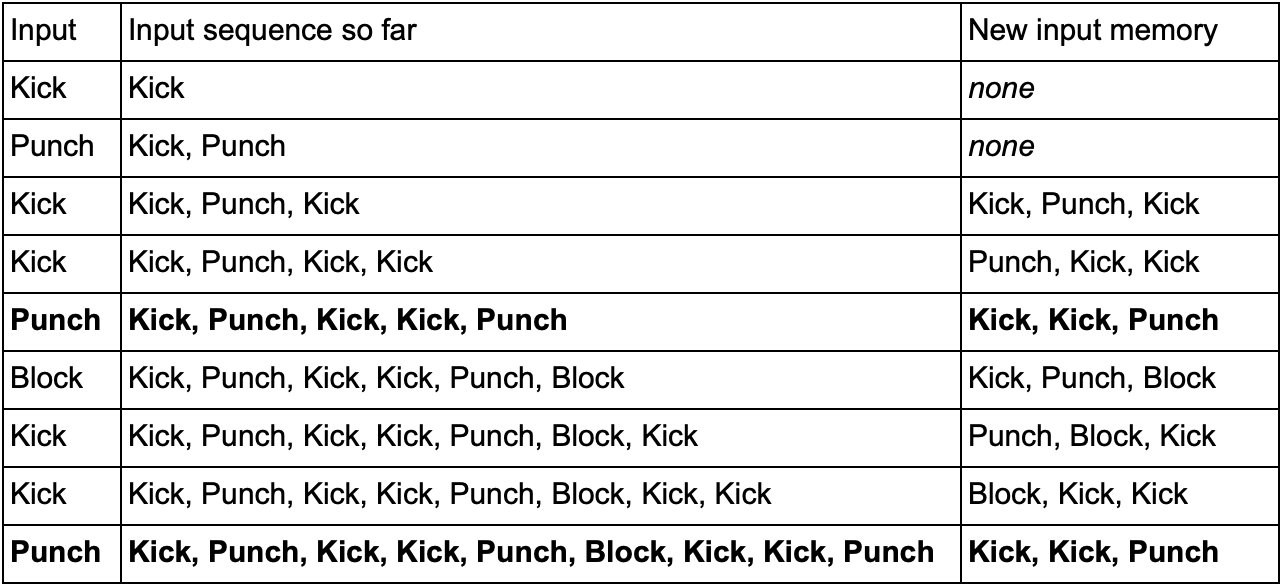
* Пусть ИИ собирает данные о состоянии мира и ключевых событиях во время игры (как указано выше).
* Изменим несколько важных значений (value) на основе этих данных.
* Реализуем свои решения, основанные на обработке или оценке этих значений.

К примеру, у агента есть несколько комнат для выбора на карте шутера от первого лица. Каждая комната имеет свое value, которое определяет насколько она желательна для посещения. ИИ случайно выбирает в какую комнату идти, основываясь на значении value. Затем агент запоминает, в какой комнате его убили, и уменьшает ее value (вероятность, что он туда вернется). Аналогично для обратной ситуации — если агент уничтожает много противников, то value комнаты увеличивается.

Марковская модель

Что если мы используем собранные данные для прогнозирования? Если запомнить каждую комнату, в которой видим игрока в течение определенного периода времени, мы будем предугадывать в какую комнату игрок может перейти. Отследив и записав перемещения игрока по комнатам (values), мы можем прогнозировать их.  
  
Возьмем три комнаты: красную, зеленую и синюю. А также наблюдения, которые мы записали при просмотре игровой сессии:  
  
  
  
Количество наблюдений за каждой комнатой почти равное — где сделать хорошее место для засады мы до сих пор не знаем. Сбор статистики также осложняется респауном игроков, которые появляются равномерно по всей карте. Но данные о следующей комнате, в которую они входят после появления на карте — уже полезны.  
  
Видно, что зеленая комната устраивает игроков — большинство людей из красной переходят в нее, 50% которых остается там и дальше. Синяя комната наоборот не пользуется популярность, в нее почти не ходят, а если ходят, то не задерживаются.  
  
Но данные говорят нам кое-что более важное — когда игрок находится в синей комнате, то следующая комната, в которой мы его скорее всего увидим будет красной, а не зеленой. Несмотря на то, что зеленая комната популярнее красной, ситуация меняется, если игрок находится в синей. Следующее состояние (то есть комната, в которую игрок перейдет) зависит от предыдущего состояния (то есть комнаты, в которой игрок находится сейчас). Из-за исследования зависимостей мы будем делать прогнозы точнее, чем если бы мы просто подсчитывали наблюдения независимо друг от друга.  
  
Предугадывание будущего состояния на основе данных прошлого состояния называется марковской моделью (Markov model), а такие примеры (с комнатами) называют марковскими цепями. Поскольку модели представляют собой вероятность изменений между последовательными состояниями, визуально они отображаются в виде FSM с вероятностью около каждого перехода. Ранее мы использовали FSM для представления поведенческого состояния, в котором находился агент, но эта концепция распространяется на любое состояние, независимо от того, связано это с агентом или нет. В этом случае состояния представляют комнату, которую занимает агент:  
  
  
  
Это простой вариант представления относительной вероятности изменений состояний, дающий ИИ некую возможность предсказывать следующее состояние. Можно предугадывать несколько шагов вперед.  
  
Если игрок в зеленой комнате, то есть 50% шанс, что он там и останется при следующем наблюдении. Но какова вероятность, что он все еще будет там даже после? Есть не только шанс, что игрок остался в зеленой комнате после двух наблюдений, но и шанс, что он ушел и вернулся. Вот новая таблица с учетом новых данных:  
  
  
  
Из нее видно, что шанс увидеть игрока в зеленой комнате после двух наблюдений будет равен 51% — 21%, что он придется из красной комнаты, 5% из них, что игрок посетит синюю комнату между ними, и 25%, что игрок вообще не уйдет из зеленой комнаты.  
  
Таблица — просто наглядный инструмент — процедура требует только умножения вероятностей на каждом шаге. Это означает, что вы можете заглянуть далеко в будущее с одной поправкой: мы предполагаем, что шанс войти в комнату полностью зависит от текущей комнаты. Это называется марковским свойством (Markov Property) — будущее состояние зависит только от настоящего. Но это не стопроцентно точно. Игроки могут менять решения в зависимости от других факторов: уровень здоровья или количество боеприпасов. Так как мы не фиксируем эти значения, наши прогнозы будут менее точными.

N-Grams

А что насчет примера с файтингом и предсказанием комбо-приемов игрока? То же самое! Но вместо одного состояния или события, мы будем исследовать целые последовательности, из которых состоит комбо-удар.  
  
Один из способов сделать это — сохранить каждый ввод (например, Kick, Punch или Block) в буфере и записать весь буфер в виде события. Итак, игрок неоднократно нажимает Kick, Kick, Punch, чтобы использовать атаку SuperDeathFist, система ИИ хранит все вводы в буфере и запоминает последние три, используемые на каждом шаге.  
  
  
(Жирным выделены строки, когда игрок запускает атаку SuperDeathFist.)  
  
ИИ увидит все варианты, когда игрок выбрал Kick, следом за другим Kick, а после заметить, что следующий ввод всегда Punch. Это позволит агенту спрогнозировать комбо-прием SuperDeathFist и заблокировать его, если это возможно.  
  
Эти последовательности событий называются N-граммами (N-grams), где N — количество хранимых элементов. В предыдущем примере это была 3-грамма (триграмма), что означает: первые две записи используются для прогнозирования третьей. Соответственно в 5-грамме первые четыре записи предсказывают пятую и так далее.  
  
Разработчику нужно тщательно выбирать размер N-грамм. Меньшее число N требует меньше памяти, но и хранит меньшую историю. Например, 2-грамма (биграмма) будет записывать Kick, Kick или Kick, Punch, но не сможет хранить Kick, Kick, Punch, поэтому ИИ не отреагирует на комбо SuperDeathFist.  
  
С другой стороны, большие числа требуют больше памяти и ИИ будет сложнее обучиться, так как появится гораздо больше возможных вариантов. Если у вас было три возможных ввода Kick, Punch или Block, а мы использовали 10-грамму, то получится около 60 тысяч различных вариантов.  
  
Модель биграммы это простая марковская цепь — каждая пара «прошлое состояние/текущее состояние» является биграммой, и вы можете предсказать второе состояние на основе первого. 3-грамма и более крупные N-граммы также можно рассматривать как марковские цепи, где все элементы (кроме последнего в N-грамме) вместе образуют первое состояние, а последний элемент — второе. Пример с файтингом показывает шанс перехода от состояния Kick и Kick к состоянию Kick и Punch. Рассматривая несколько записей входной истории как одну единицу, мы, по сути, преобразуем входную последовательность в часть целого состояния. Это дает нам марковское свойство, позволяющее использовать марковские цепи для прогнозирования следующего ввода и угадать, какой комбо-ход будет следующим.

Заключение

Мы поговорили о наиболее распространенных инструментах и подходах в разработке искусственного интеллекта. А также разобрали ситуации, в которых их нужно применять и где они особенно полезны.  
  
Этого должно быть достаточно для понимания базовых вещей в игровом ИИ. Но, конечно же, это далеко не все методы. К менее популярным, но не менее эффективным относятся:

* алгоритмы по оптимизации, включая восхождение по холмам, градиентный спуск и генетические алгоритмы
* состязательные алгоритмы поиска/планирования (minimax и alpha-beta pruning)
* методы классификации (перцептроны, нейронные сети и машины опорных векторов)
* системы для обработки восприятия и памяти агентов
* архитектурные подходы к ИИ (гибридные системы, подмножество архитектур и другие способы наложения систем ИИ)
* инструменты анимации (планирование и согласование движения)
* факторы производительности (уровень детализации, алгоритмы anytime, и timeslicing)