# РЕФЕРАТ

Пояснительная записка 87 стр., 21 рис., 5 табл., 15 ист., 1 прил.

MALMO CHALLENGE, МУЛЬТИАГЕНТНАЯ СРЕДА, РАСПОЗНАВАНИЕ ПОВЕДЕНИЯ, ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ, Q-LEARNING, MINECRAFT

Объектом исследования (разработки) является стратегия взаимодействия агентов искусственного интеллекта для достижения общей цели.

Цель работы – разработать модель поведения агентов в мультиагентной среде для распознавания намерений напарника и коллаборации с ним для достижения общей или собственной второстепенной цели.

При выполнении работы была разработана модель поведения агента на основе искусственного интеллекта для навигации в двумерном пространстве на поле с разреженными препятствиями и определения намерений и цели напарника. Были разработаны две программы: одна для обучения агента в упрощенной двумерной среде, вторая для тестирования его способностей в трехмерной среде, приближенной к реальности. В качестве трехмерной среды используется игра Minecraft, требующая непрерывного передвижения агентов (в отличие от дискретного в упрощенной среде) и предоставляющая усложненную динамическую составляющую среды. По результатам тестирования обеих программ полученная точность определения намерений и вероятность достижения заданной цели агентами достигает 90%.

## ABSTRACT

In the course of the graduate work, a model of artificial intelligence agent behavior was developed for navigation in a two-dimensional space in a field with sparse obstacles and identification of partner's intentions and goals. Two programs were developed: the first one to train the agent in a simplified two-dimensional environment, the second one to test its abilities in a three-dimensional environment close to the real world. The game Minecraft is used as a three-dimensional environment, that requires continuous movement of agents (as opposed to discrete steps in the simplified environment) and providing a more complex dynamic component of the environment. According to the results of testing of both programs, the achieved accuracy of partner’s intentions identification and the probability of reaching a given goal by the agents is 90%.

## СОДЕРЖАНИЕ

# ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

ВКР – выпускная квалификационная работа

ГИП – графический интерфейс пользователя

ИИ – искусственный интеллект

# ВВЕДЕНИЕ

Данная работа преследует те же цели, что и соревнования The Malmo Collaborative AI Challenge, на основании которых она выполнялась. В качестве основных целей можно выделить необходимость исследований и разработки агентов на основе искусственного интеллекта, способных объединяться с человеком и другими агентами для решения общей задачи, а также ответить на ряд вопросов: каким образом программные агенты могут распознавать чьи-либо намерения и понимать, какую задачу пытается решить их напарник, какие образцы поведения способствуют решению общей задачи и как группа агентов может общаться и координировать действия всех членов группы. Поскольку исследования в области взаимодействия агентов на основе искусственного интеллекта начались относительно недавно, еще не на все из перечисленных вопросов были найдены точные ответы, что предполагает необходимость дальнейшего развития в данной области и тестирования различных сценариев взаимодействия агентов.

В основе данной работы лежит расширенная версия этапа соревнований Malmo challenge, имеющего название Pig chase. Решения официальной версии задачи Pig chase неприменимы к расширенной, поскольку среда взаимодействия агентов значительно отличается. Благодаря этому появляется возможность экспериментирования с методами машинного обучения, отличными от тех, что уже применялись участниками официальных соревнований.

# 1. СОВРЕМЕННОЕ СОСТОЯНИЕ ВОПРОСА

## 1.1. История вопроса

Существует множество определений искусственного интеллекта (далее ИИ). В качестве примера можно привести определение из статьи «Что такое искусственный интеллект» 2007 года – «Это наука и техника изготовления интеллектуальных машин, в частности компьютерных программ. Эта область науки тесно связана с задачей изучения человеческого интеллекта с использованием компьютеров, однако ИИ не должен ограничиваться применением биологически наблюдаемых методов.» [1] Однако активное изучение ИИ началось еще в 1950х годах, когда Алан Тьюринг задал в своей работе вопрос «Могут ли машины думать?», предложив тест, содержащий набор задач, позволяющих отличить человека от ИИ. С прошествием времени и развитием методов ИИ тест многократно изменялся и дополнялся, оставаясь актуальным и по сегодняшний день под названием «Тест Тьюринга».

Одной из самых влиятельных работ является «Искусственный интеллект: Современный подход», последнее изменение которой производилось в 2022 году. В ней Стюарт Рассел и Питер Норвиг выделяют два подхода разработки интеллектуальных систем: человеческий и идеальный. В первом случае система думает и ведет себя подобно человеку, что соответствует определению Тьюринга, во втором – думает и ведет себя строго рационально.

В самом простом виде ИИ представляет собой способность компьютера самостоятельно решать творческие задачи, выполнением которых обычно занимается человек. ИИ включает в себя методы машинного обучения, одним из подразделов которого, в свою очередь, является глубокое обучение. Эти понятия в настоящее время часто встречаются в научных работах. В частности, их используют в системах, предсказывающих или классифицирующих данные на основе подаваемой в них информации.

Можно выделить два основных типа искусственного интеллекта: слабый (Artificial Narrow Intelligence, ANI) и сильный (Artificial General Intelligence, AGI). Первый тип направлен на решение конкретного узкого круга задач. Однако называть его «слабым» довольно сложно, поскольку в некоторых задачах он достигает уровня человека, а иногда даже превышает его. В качестве самых известных примеров можно привести голосовых помощников, таких как «Алиса», и самоуправляемые транспортные средства. Второй тип на данный момент является лишь теорией и представляет собой ИИ, интеллект которого соответствует интеллекту человека по всем критериям, более того, сильный ИИ имеет самосознание, он способен решать различные задачи, обучаться и планировать будущее. Иногда выделяют третий тип – искусственный суперинтеллект, способный теоретически превзойти человека по вышеупомянутым критериям. Несмотря на то, что в данный момент не существует практической реализации ни второго, ни третьего типов, ученые продолжают исследования в этой области.

Первым и одним из самых успешных узконаправленных ИИ является компьютер Deep Blue, созданный с единственной целью – обучить агента на основе ИИ играть в шахматы. Под ИИ-агентом (интеллектуальным агентом) подразумевается «программа, самостоятельно выполняющая задание, указанное пользователем компьютера, в течение длительных промежутков времени.» [11] Агент способен ориентироваться в окружающей среде (возможно, виртуальной) и совершать определенные действия для получения вознаграждений, заданных самой средой. Уже в 1997 году он стал первым компьютером, выигравшим матч против мирового чемпиона Гарри Каспарова. В этом компьютере использовалось дерево поиска оптимальных ходов вплоть до 20 шагов вперед. Функция оценки качества каждого хода настраивалась вручную, но позже была оптимизирована программными методами и активно применялась в различных областях. Более того, в компьютер была заложена история множества игр гроссмейстеров со всего мира, послужившая основанием оценки качества ходов. Несмотря на успех

Deep Blue, он был далеко не самым производительным компьютером в 1997 году, что оставляло место для дальнейшего улучшения работы программы.

В 2013 организация DeepMind повторила успех Deep Blue, обучив ИИ на играх компании Atari Games до уровня профессиональных игроков. Несмотря на то, что в рассматриваемых играх предоставляется небольшой набор возможных действий, ИИ был натренирован без использования промежуточных награждений системы, а только на пикселях с экрана компьютера и результатах каждой игры. Спустя некоторое время очередной ИИ от компании DeepMind – AI AlphaGo победил чемпиона мира в игре с одноименным названием AlphaGo.

Были сделаны попытки разработки ИИ для игры StarCraft 2, однако эта игра предоставляет гораздо больший выбор возможных действий и имеет более детализированный мир. Также игроку необходимо понимать намерения своего оппонента и составлять свою стратегию в соответствие с ними. На данный момент существующие реализации ИИ не показали таких высоких результатов, как их предки в более простых играх.

Коллективный интеллект касается не только сферы ИИ, но является неотъемлемой частью коммуникации между людьми, начиная с времен пещерных рисунков. С помощью рисунков люди могли делиться своими идеями и планировать групповую охоту, что значительно повышало шансы на успех. Люди, способные грамотно распределять действия между членами команды чрезвычайно важны в работе как с другими людьми, так и с ИИ-агентами.

С появлением сначала письменности, а затем сети Интернет, люди получили доступ к огромному количеству информации, что, безусловно, положительно сказывается на возможности оптимального распределения действий между членами различных групп. С увеличением объема доступной информации начали появляться программы, способные обрабатывать данные гораздо эффективнее человека, в связи с чем появился вопрос о создании

программных агентов, способных объединяться с людьми для достижения общих целей.

С 20 века существуют обучающиеся системы ИИ, не способные объединяться с другими агентами или людьми, а также существуют системы совместной работы, способствующие объединению людей посредством компьютера, без встроенного механизма обучения. Для создания совместного ИИ необходимо эти две системы объединить. Первый раз такое объединение рассматривалось в мультидисциплинарном исследовании «радикальные инновации», в котором были выделены требования к созданию совместного ИИ и показано возможное будущее подобных систем. Однако в этой работе в основном описываются различные подходы к созданию рассматриваемой системы, такие как социальная и развивающая психологии, нейронауки, робототехника, языковые модели, а ИИ является лишь частью статьи. На основе «радикальной инновации» позже вышла статья, посвященная аналогичной теме (Stefik & Price, 2023), но в этот раз в центре внимания был именно ИИ. В статье рассматриваются научные приемы и технологии для создания совместного ИИ, а также предлагается план действий для их реализации. Как отмечает автор статьи «Roots and Requirements for Collaborative AI» в переводе с английского: совместные агенты должны обучаться так же, как люди, в совместной среде, иначе они окажутся хрупкой демонстрационной системой, которая провалится при выполнении реальных практических задач. [2]

Джозеф Ликлайдер в своей работе «Симбиоз человека и компьютера» представил идею внедрения ИИ в команды людей для лучшего понимания запутанных ситуаций и планирования действий. Смысл идеи в том, что компьютер получает конкретную задачу с входными значениями и выдает числовой ответ, в то время как человек ставит цели, формулирует гипотезы и оценивает решения задач. Джозеф Ликлайдер отмечал, что в будущем компьютеры также помогут ставить людям оптимальные цели, и в целом подобный симбиоз будет более производительным, чем решение проблемы

компьютером в одиночку или группой людей. Однако одним из самых больших препятствий к осуществлению планов была невозможность общаться с компьютером на человеческом языке. В связи с этим были созданы совместные системы, позволяющие членам команды обмениваться информацией через компьютер, но не непосредственно с самим компьютером.

### 1.2. Современные проблемы

При разработке агентов совместного ИИ выделяют следующие наиболее актуальные проблемы (задачи) [10]:

1. Каждому агенту необходимо способствовать координации и стремиться к достижению общих целей;
2. Агент должен быть способен моделировать задачи, намерения и поведение других членов команды на основе состояний системы (все ли идет по плану, нужна ли напарнику помощь и т.д.);
3. Все агенты (особенно управляемые людьми) должны вести себя предсказуемо;
4. Агенты должны быть управляемыми;
5. Состояние и намерения агента должны быть как можно более простыми и понятными для остальных членов команды;
6. Агенты также должны быть способны обнаруживать и обрабатывать соответствующие сообщения о статусе и намерениях напарников;
7. У самих агентов должна быть возможность координации общих (командных) целей;
8. Технологии планирования и самоуправления должны иметь совместный характер;
9. Агенты должны уметь управлять своим вниманием;
10. Все члены команды должны участвовать в оптимизации затрат на общую координацию.

При построении совместных систем часто возникают ситуации, когда один из агентов по каким-либо причинам провалил задание и больше не способен играть свою роль. Чтобы система могла продолжать функционировать, такому агенту необходимо подать сигнал о наступившей или приближающейся опасности. В некоторых ситуациях агент может изменить свою предопределенную роль без внешних опасностей, что очень негативно влияет на совместный прогресс команды, поскольку противоречит изначальному плану.

Чтобы оставаться полезным элементом системы в течение всего времени ее работы, каждому агенту необходимо понимать ситуацию, в которой он находится. Например, осознавать положение других агентов в виртуальной среде, придерживаются ли остальные агенты первоначального плана и, если нет, то попытаться понять их стратегию преодоления возникших трудностей. Поскольку агенты пока еще не способны общаться между собой как люди, чаще всего используют небольшой набор определенных действий, которые могут предпринимать участники. Это позволяет превратить проблему обработки поведения в простую задачу классификации. Однако, если часть агентов находится под полным управлением людей, такой подход становится неприменим.

Чем более предсказуемо ведут себя агенты, тем лучше каждый из них сможет смоделировать текущее состояние системы. Если агент часто меняет свои цели или работает неоптимально, понять его стратегию становится сложно даже человеку. Также можно заметить, что чем более адаптивным является агент, то он менее предсказуем. Большинство алгоритмов управления агентами проектируются максимально прозрачными и определенными. Только таким образом удается построить надежную совместную систему.

Полная прозрачность при разработке механизмов управления агентами позволяет сохранить предсказуемость их поведения. Достаточно часто возникают случаи, когда требуется вмешательство в подобные механизмы с

их последующим изменением, чтобы, например, улучшить производительность какого-либо агента. Для соответствия всем этим требованиям используют специальные политики поведения, которые можно редактировать без изменения самого кода агентов. Наличие полного набора возможных действий агента позволяет легче классифицировать отдельные элементы его поведения.

Существует мнение, что работа качественных автоматических систем должна быть незаметна для человека. [10] Но с совместными системами ситуация обратная: чем очевиднее действия каждого агента, тем легче понять его стратегию целиком. Наблюдая за работой системы у человека не должно возникать вопросов на подобии «Что этот агент пытается сделать?» и «Что он планирует делать дальше?». В команде, состоящей полностью из людей, участники склонны ориентироваться на свои собственные мысли при попытке проанализировать поведение остальных членов, однако среди агентов такой подход невозможен, поскольку он часто приводит к ошибкам и недопониманиям, понижая уровень доверия к ИИ.

Передачи информации о поведении и намерениях каждым агентом может оказаться недостаточно для создания качественной системы, если агенты не способны правильно их обрабатывать. На основании полученных сообщений от других участников агенты должны формировать текущее состояние как всех членов команды, так окружающей среды и выполняемой задачи. Основная проблема заключается в неспособности большинства агентов распознавать нюансы поведения участников, если не указывать на них напрямую. Считается, что пока ИИ не достигнет уровня человека, асимметрия в координации не позволит им качественно обрабатывать сообщения, посылаемые человеком. [10]

В случаях, когда ситуация (задача, цель) изменяется, и агенты больше не могут придерживаться первоначального плана, необходимо предусмотреть возможность обсуждения новой стратегии между агентами. Если агенты не могут изменить цель или объяснить это изменение другим участникам, они

будут отталкиваться от первоначального общего плана. Обычно такие ситуации решаются механизмами и алгоритмами, принимающими на вход командные цели агентов и возвращающими готовые стратегии с обработкой всех ситуаций, имеющих возможность возникнуть при ее выполнении. Очевидно, данный подход неприменим, если хоть один из агентов управляется человеком.

При работе в команде люди делят всю задачу на подзадачи и по завершении каждой из них обсуждают результаты и продумывают дальнейшие действия. Для качественной работы системы необходимо добавить подобный метод планирования во взаимодействие агентов. Для его реализации агентам следует

1. поддерживать общую картину плана настолько детальной, насколько это возможно, используя все доступные сообщения других членов команды;
2. обнаруживать возможные опасности, проявляющиеся при продолжении следования текущему плану, и запускать перепланировку при необходимости;
3. оценивать жизнеспособность изменений, внесенных в план другими участниками;
4. управлять механизмом перепланирования, в том числе нанимать дополнительных агентов, если процесс требует больше усилий, чем доступно в данный момент;
5. перераспределить роли между агентами, если план был изменен;
6. изменять виды коммуникации для лучшего понимания информации агентами.

При реализации механизма планирования стоит учитывать, что на этапе перепланирования агенты могут оказаться на разных стадиях выполнения общего первоначального плана.

При передаче сообщений о состоянии и поведении агенты должны специальным образом направлять внимание других участников только на

важные детали, игнорируя незначительные действия и прочий шум. Агентам необходимо понимать, каких сообщений стоит ожидать от каждого члена команды и на что стоит обращать внимание, основываясь на своей внутренней сформированной модели участников и конкретной ситуации. Если этот процесс будет реализован некорректно, могут возникнуть ситуации, при которых автоматическая система начнет незаметно для наблюдателей компенсировать различные неполадки и в какой-то момент дойдет до своего предела. На момент превышения возможностей автоматизированной системы провал может уже оказаться необратим. Настоящий прорыв произойдет, когда агенты смогут общаться так же свободно, как организованная команда людей. Однако на данный момент еще актуальны вопросы «Как определить, что агент не справляется со своей задачей, если сама задача еще не провалилась?» и «Как агент может подать сигнал о скором достижении предела своих возможностей?». Часто для решения данной проблемы используют сигналы о пересечении некоторой границы возможностей агента. Но подобные методы редко применяются на практике, так как не учитывают контекст задачи и сигнализируют о неполадке слишком рано, либо, наоборот, слишком поздно, когда провала уже не избежать.

Координация действий агентов, посылка сообщений и их обработка могут занимать много времени. Следовательно, агентам (как и людям) следует оптимизировать данные методы для уменьшения затрат на их реализацию. Работа в команде требует постоянного вложения ресурсов для поддержания этих процессов, поэтому полностью избавиться от затрат не получится (при отсутствии ресурсов каждый агент будет следовать своей собственной цели и пытаться достичь ее в одиночку). Однако снижение затрат является лишь частью процесса оптимизации стратегии. Помимо этого, агенты должны подстраиваться под своих операторов, а не пытаться подстроить их под себя лишь ради экономии ресурсов. На процесс адаптации может уйти большое количество ресурсов, поскольку вместе с

приспособлением к новой среде агенты должны сохранять предсказуемость своего поведения, как было упомянуто ранее. Основной проблемой является недостаточный уровень приспособления агентов к потребностям и знаниям людей.

### 1.3. Пути решения проблем

Часто системы совместного планирования включают в себя характеристики, не встречающиеся в традиционных системах планирования. Например, разработка плана в таких системах должна быть поэтапной. Другими словами, требующей детального рассмотрения небольших частей плана, составление списка возможных действий и принятия решений по каждой части в отдельности, и только после разбора всех частей формируется полная картина действий. Другой важнейшей составляющей системы планирования является стабильность. Изменения в итоговом плане должны быть минимальны при возникновении новых ограничений на возможные решения. Если каждый раз при появлении новых трудностей система будет составлять весь план заново, итоговый список действий может кардинально измениться. Также при создании плана не должна использоваться фиксированная стратегия поиска, поскольку при формировании плана не существует определенного порядка рассмотрения его подзадач. Из этого критерия вытекает еще одна важная характеристика системы: открытость к новшествам. Под руководством человека система должна быть способна формировать и анализировать планы, которые при других обстоятельствах она бы не создала. [7]

Две основные задачи, возникающие при создании системы совместного планирования между человеком и компьютером, – это определение контекстно-независимого уровня решения поставленной задачи и создание системы гибридного анализа плана. Первая задача представляет собой прослойку между системой коммуникации человека с компьютером и

системой анализа плана. Абстрактная структура такой системы изображена на рисунке 1.1.

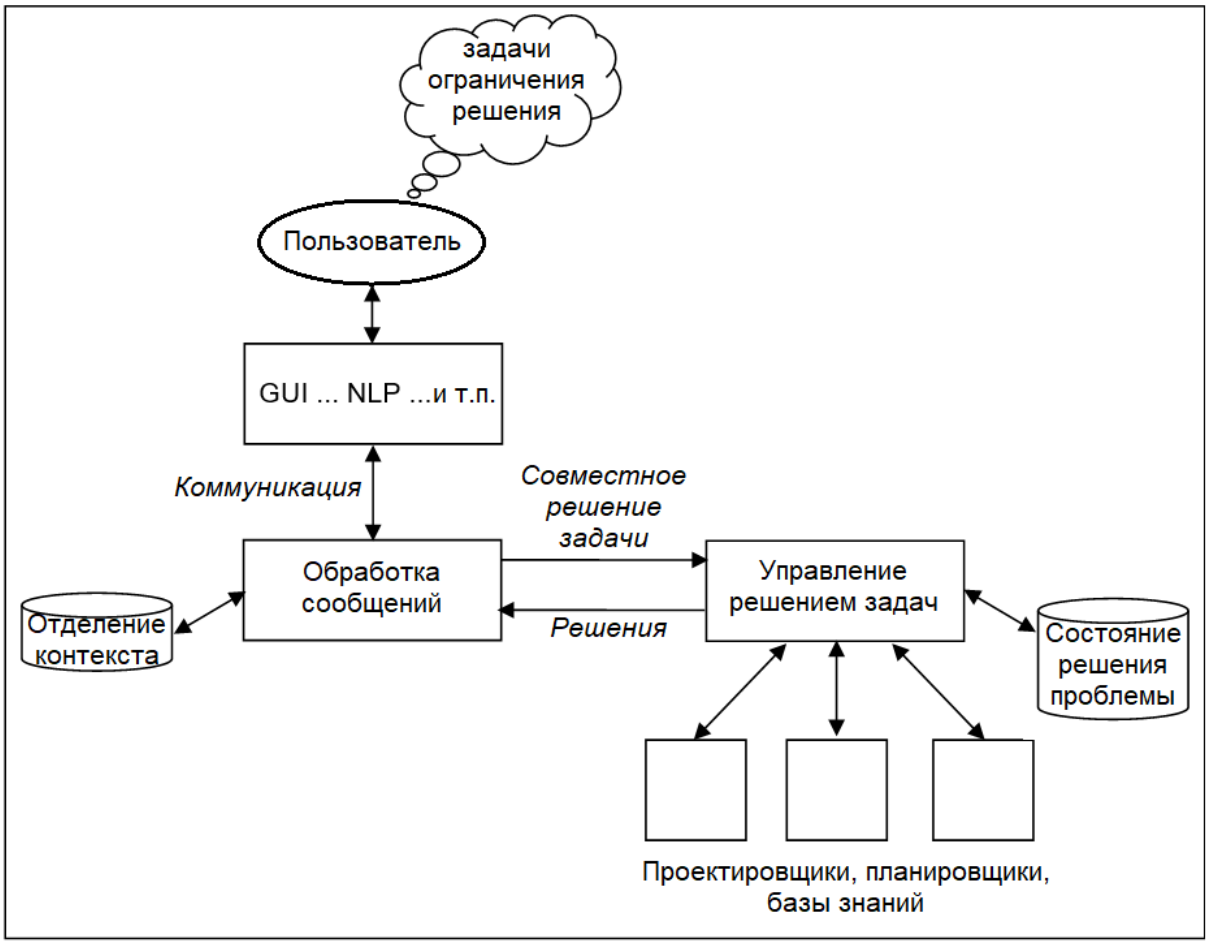


Рисунок 1.1 – Структура системы совместного планирования [7]

На рисунке изображен пользователь со своими целями, которых он пытается достичь, возможные ограничения на решения задачи и, возможно, часть самого решения. Эта информация передается системе посредством GUI (Graphical user interface) – графического интерфейса пользователя и NLP (Natural Language Processing) – обработки естественного языка, которые являются самыми часто используемыми средствами коммуникации человека с компьютером. Самыми известными и доступными на данный момент примерами использования такого интерфейса являются виртуальные ассистенты Алиса, Сири и другие. Далее система передает полученную информацию в центр управления решением задач, который, в свою очередь, создает и сохраняет текущее состояние решения. Как только были интерпретированы намерения пользователя и сгенерировано состояние, блок

управления решением задач вызывает традиционные методы искусственного интеллекта: классификацию, планирование, поиск по базе знаний и т.д., наиболее подходящие в контексте текущей задачи. При такой структуре системы функциональность (компоненты ИИ) может быть добавлена динамически, прямо во время работы, если того требует задача. Однако традиционные методы не всегда подходят для последовательного составления плана, когда могут добавляться и исчезать ограничения, или в ситуациях, когда пользователь предлагает готовую часть решения.

Цель центра управления решением задач – поддерживать совместное планирование и сохранять историю состояний решения проблемы. Для успешного выполнения своих функций необходимо определить следующие требования к структуре системы:

1. Цели и задачи пользователя должны иметь иерархическую структуру: иметь возможность детализироваться и фокусироваться на отдельных частях общей задачи, в которой все уровни связаны ограничениями на возможные решения;
2. Каждый уровень иерархии является абстрактным описанием всех его составляющих, а связь между ними состоит из ограничений и предположений о независимости;
3. Должна быть возможность составить конкретный план действий, основанный на иерархии задач, удовлетворяющий всем ограничениям и учитывающий всевозможные решения и их длительность на каждом этапе;
4. План, описанный в предыдущем пункте, должен быть представлен пользователю с детальным описанием состояний окружающей среды на протяжении выполнения всего плана с учетом возможных внешних факторов и способов их обработки.

Эти 4 требования позволяют применять традиционные методы ИИ напрямую к поиску решений проблемы и поддерживать максимальную

эффективность взаимодействия пользователя с компьютером. Для составления абстрактного плана на основе иерархичной структуры целей используются контекстно-независимые алгоритмы темпоральной логики [7] (это «логика, в высказываниях которой учитывается временной аспект; используется для описания последовательностей явлений и их взаимосвязи по временной шкале» [12]). Именно, опираясь на абстрактный план, традиционные методы ИИ формируют конкретные действия, которые в конечном счете передаются пользователю системы. Обычно пользователь, получив план действий, добавляет ограничения и запрашивает план, учитывающий их. Вместе с самой последовательностью действий система выделяет три состояния окружающей среды: до, во время и после выполнения плана, что позволяет пользователю точнее оценить результат работы ИИ.

Однако несмотря на усилия центра управления решением задач, остается проблема правильной интерпретации задач классическими методами ИИ. Преимуществами планировщика, способного самостоятельно обрабатывать контекст задачи, являются скорость, выразительность (поскольку пользователю необходима наиболее полная информация о действиях, планах, времени и состоянии окружающей среды), полнота (агент не обязан оценивать все существующие планы, но должен поддерживать возможность поэтапного приближения к любому из них) и гибкость (агент должен подстраиваться под изменения, внесенные пользователем).

### 1.4. Вывод по разделу

Таким образом, на данный момент уже существует достаточно много подходов к созданию систем совместного ИИ, большую часть которых составляют системы совместного планирования человека и компьютера. Однако конкретные методы, применяемые для реализации систем, могут сильно различаться и зависеть от самой задачи, поставленной перед системой. Тема создания совместного ИИ, несмотря на первые ее

упоминания в 1950х годах в работах Алана Тьюринга, начала активно изучаться и внедряться в повседневную жизнь только в 2020х годах, что показывает необходимость в продолжении исследования и поиска более эффективных и точных алгоритмов планирования в различных сферах информационных технологий.

Для дальнейшего исследования систем совместного планирования целесообразно разработать подобную систему, например, для соревнований в рамках проекта Malmo, приблизив виртуальную среду к реальности. В таком случае, несмотря на простоту задачи бинарной классификации поведения агентов, понадобятся новые алгоритмы, поскольку методы, примененные в оригинальной версии соревнований, потеряют свою актуальность в расширенной версии.

# 2. ОПИСАНИЕ РАЗРАБОТКИ

Данная работа основывается на проекте Malmo [9], разработанном компанией Microsoft. Цель проекта заключается в исследовании возможностей ИИ и машинного обучения, в частности метода обучения с подкреплением. «Обучение с подкреплением (Reinforcement Learning) – это метод машинного обучения, в котором наша система (агент) обучается методом проб и ошибок. Идея заключается в том, что агент взаимодействует со средой, параллельно обучаясь, и получает вознаграждение за выполнение действий.» [4] На странице проекта организаторы задают следующие вопросы:

1. Как разработать ИИ, способный ориентироваться в комплексных виртуальных средах?
2. Может ли он научиться, в том числе у людей, взаимодействовать с окружающей средой?
3. Может ли такой ИИ сохранять и передавать свои знания, накопленные за все время его существования для решения новых проблем?

Проект предоставляет пользователям платформу для различных экспериментов, состоящую из игры Minecraft, позволяющей игрокам вносить практически любые изменения в виртуальный мир и взаимодействовать внутри него; модификации этой игры для возможности управления внутриигровыми персонажами посредством скриптов, написанных на языке Python; и нескольких исследований, приведенных в качестве примера использования платформы. Помимо платформы для экспериментов с ИИ проект Malmo предоставляет возможность участия в соревнованиях, посвященных различным сферам ИИ.

В данной работе предлагается решение задачи 2017 года под названием The Malmo Collaborative AI Challenge (также известной под названием Pig chase) [8]. Данная задача посвящена взаимодействию человека и программного агента для достижения общей цели. Отдельное внимание уделяется следующим вопросам:

1. Как ИИ-агенты могут распознавать чьи-либо намерения?
2. Как ИИ-агенты могут понимать, какое поведение положительно сказывается на командной работе?
3. Как агенты могут общаться и согласовывать между собой стратегии достижения цели?

Эти вопросы представляют собой особый интерес, поскольку еще ни на один из них не было найдено точного ответа.

Задача представляет собой мини-игру, основанную на теоретической игре «охота на оленя», в которой двум агентам необходимо работать совместно, для получения наибольшей награды, либо действовать поодиночке для возможности получения меньшего вознаграждения. Действия мини-игры происходят на ограниченной по периметру плоскости с препятствиями; помимо двух агентов на плоскости находится еще одна сущность (свинья, исходя из названия этапа). У каждого из агентов есть выбор: поймать сущность (загнать ее в угол либо зажать между препятствиями, ограничив все пути отхода), получив максимальную награду, или сдаться и выйти из игры, переместившись в заранее определенную позицию и получив небольшую награду. [8]

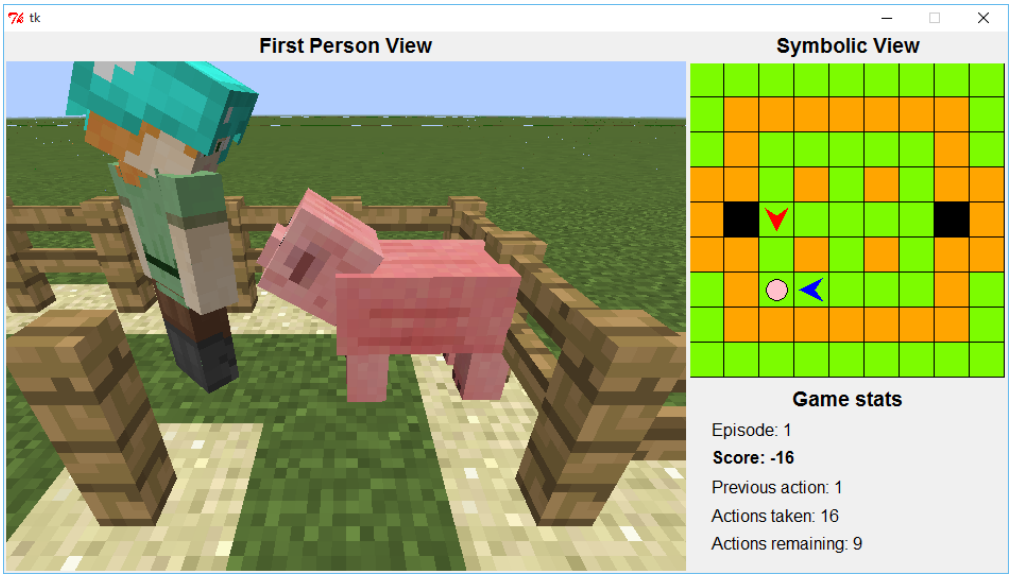


Рисунок 2.1 – Кадр из мини-игры Pig chase [8]

На рисунке 2.1 можно увидеть два изображения, отражающие одно и то же состояние системы: глазами одного из агентов, ориентирующегося в

трехмерной среде, (от первого лица) и упрощенная двумерная символьная схема. На правом верхнем изображении отражено полное состояния системы в текущий момент времени. На нем можно увидеть направленные стрелки, обозначающие положение и ориентацию двух агентов, розовый круг – положение сущности, зеленые квадраты – свободное пространство, оранжевые квадраты – препятствия и черные квадраты – точки выхода из игры. Под символьной схемой находится численная статистика состояния: количество завершенных игр за время работы программы, значение награды для текущего агента (агента, с точки зрения которого был сделан снимок экрана в левой части), числовой идентификатор предыдущего действия агента, количество действий, произведенных агентами в течение игры, и максимально возможное количество предпринятых действий до завершения игры. Если агенты превысят максимально допустимое количество действий перед тем, как поймать сущность или сдаться, игра завершается автоматически, а итоговая награда каждого из агентов равна соответствующему значению в поле «Score».

В данной работе игровой уровень был расширен: увеличены размеры плоскости, добавлены препятствия и перемещено положение точки выхода из игры. Уровень, используемый во всех дальнейших иллюстрациях, выглядит следующим образом (без учета препятствий, ограничивающих уровень по периметру, поскольку цель всегда находится в пределах уровня):

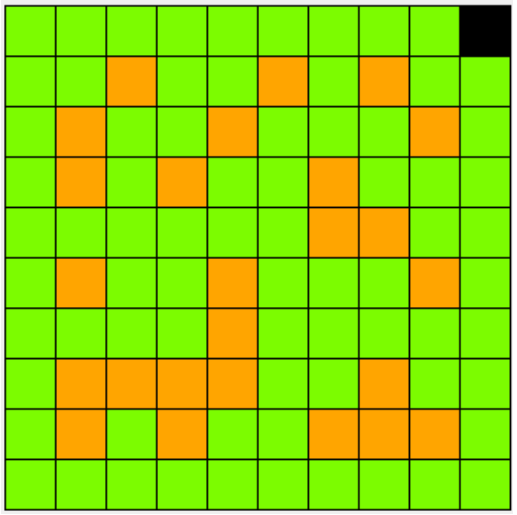

Рисунок 2.2 – Символьная схема уровня, используемого в данной работе

Расширение уровня необходимо для создания и тестирования новых алгоритмов ИИ, поскольку для уровня, определенного в оригинальных соревнованиях, уже было предложено множество решений, основанных на популярных традиционных алгоритмах обучения с подкреплением. В частности, алгоритмах, обучающихся и тестируемых на фиксированных уровнях. Другими словами, для работоспособности таких алгоритмов структура уровня (препятствия) должна постоянно оставаться неизменной. Расширенная же версия предполагает возможность тестирования агентов на уровнях с динамически сгенерированными препятствиями.

### 2.1. Принципы, материалы и методы

Поскольку проект Malmo фокусируется на методах обучения с подкреплением, для мультиагентных сред рекомендуется использовать алгоритм Q-обучения. [13]

Q-обучение (Q-Learning) – это алгоритм безмодельного обучения с подкреплением, позволяющего агенту изучить стоимость каждого возможного действия в каждом состоянии системы. [13] Самая базовая реализация алгоритма включает в себя таблицу (Q-таблица), столбцами которой являются действия агента, а строками – состояния системы. Каждая ячейка отражает стоимость действия в текущем состоянии, которую агент выучил в процессе тренировки модели. Обучение модели происходит посредством получения наград за совершенные действия, предоставляемых виртуальной средой. Модель запоминает, какие действия в каждом состоянии приводят к положительному исходу, а какие к наказанию агента. При запуске программы Q-таблица может заполняться как нулями, так и случайными значениями. Далее, после каждого действия агента обновляется одна ячейка таблицы, находящаяся на пересечении предыдущего действия агента ($a_t$) и состояния системы на момент совершения этого действия ($s_t$), по формуле, называемой уравнением Беллмана или функцией полезности [13]:

$$Q_{(s_t,a_t)} = (1-\alpha) * Q_{(s_t,a_t)} + \alpha(r_t + \gamma * \max Q(s_{t+1}, a))$$

Где Q – Q-таблица, $\alpha$ – коэффициент обучения, определяющий степень доверия модели новой информации и $\gamma$ – коэффициент дисконтирования, позволяющий системе предпочитать краткосрочные или долгосрочные награды. Поиск максимума в выражении $\max Q(s_{t+1}, a)$ происходит относительно доступного набора действий в состоянии $s_{t+1}$ после совершения действия $a_t$. Общего правила для вычисления значений коэффициентов не существует, для каждой ситуации они подбираются отдельно, основываясь на результатах работы системы. Если очередное действие агента привело к окончанию игры, обновление таблицы на этом этапе не происходит. Алгоритм обучения выглядит следующим образом:

1. Агент совершает действие – либо случайное, либо соответствующее самому большому значению (награде) в Q-таблице напротив текущего состояния системы;
2. Агент обновляет значение в ячейке Q-таблицы, фигурирующей в предыдущем пункте (даже если было выбрано случайное действие);
3. Уменьшается коэффициент случайности поведения агента, теперь он чаще полагается на модель, сформированную в виде Q-таблицы;
4. Переход к пункту 1, если не был получен сигнал окончания обучения.

Таким образом, когда случайность исключается из поведения агента, каждое выбранное им действие приносит наибольшее значение награды. Случайность необходима для наиболее полного понимания агентом окружающей среды путем ее исследования. Сама награда, хранящаяся в ячейке Q-таблицы, является взвешенной суммой всех будущих наград, начиная от текущего состояния системы. Поскольку набор всех состояний и переходов между ними представляет собой Марковский процесс принятия решений, при достаточно длительной фазе обучения и статичности окружающей среды значения Q-таблицы должны достигнуть оптимума для всех возможных состояний. [13]

Однако, как упоминается в [13], мультиагентную среду необходимо превратить в среду с одним агентом (к которому применяется Q-обучение).

Это связано с требованием сходимости метода Q-обучения: среда должна быть статична. Другими словами, если одно и то же действие в одном и том же состоянии системы может принести к разным значениям наград, оптимальное состояние Q-таблицы никогда не будет достигнуто. В связи с этим разработка программы была разделена на реализацию двух напрямую независимых алгоритмов: алгоритм перемещения агента по уровню и алгоритм распознавания намерений напарника. Решение обеих подзадач основывается на Q-обучении.

Среди методов, предлагаемых участниками соревнований, встречаются алгоритмы, не использующие никаких дополнительных средств коммуникации между агентами помимо их местоположения на уровне. Такие методы наилучшим образом подчеркивают важность необходимости не только качественного распознавания поведения, но и качественной демонстрации своих намерений. В связи с этим было принято решение создать алгоритм, требующий минимального общения между агентами, такого как передача своих текущих координат на уровне.

В качестве примера можно рассмотреть метод, предложенный участниками, занявшими первое место в соревнованиях 2017 года [5]. Их алгоритм распознавания намерений предполагает наличие двух стратегий, одной из которых придерживается агент-напарник в течение всей игры. Агент может действовать либо полностью случайным образом, либо наиболее оптимально для достижения общей цели. Подразумевается, что напарник не может менять стратегию во время самой игры. Далее они преобразуют мультиагентную среду в среду с одним агентом, объявляя напарника частью виртуальной среды. Небольшие размеры оригинального уровня позволяют построить дерево всевозможных ситуаций на игровом поле во время тренировки модели. На этапе тестирования это позволит достаточно точно и быстро определять стратегию напарника при помощи алгоритма дерева поиска Монте-Карло. Значение вероятности каждой стратегии, полученное на этом этапе, обрабатывается через вывод Байеса. К

недостаткам этого алгоритма можно отнести необходимость небольшого размера уровня, чтобы сохранить все состояния в оперативную память и поддерживать высокую скорость работы.

Поскольку работа состоит из двух частей, начать следует с реализации предсказуемого поведения целенаправленного агента. В связи с увеличением размеров уровня, используемого в данной ВКР, сохранять все возможные состояния в памяти стало невозможно, следовательно, способ, описанный в предыдущем абзаце, не подойдет. Одной из самых известных игр, в которой необходимо достигать цели, обходя препятствия на плоскости, является «Змейка». Ее концепция очень близка к мини-игре Pig chase: змейка существует на двумерной плоскости, разделённой на дискретные ячейки, цель располагается в одной из этих ячеек, а роль препятствий играет само тело змейки, растущее по мере продвижения игрового процесса.

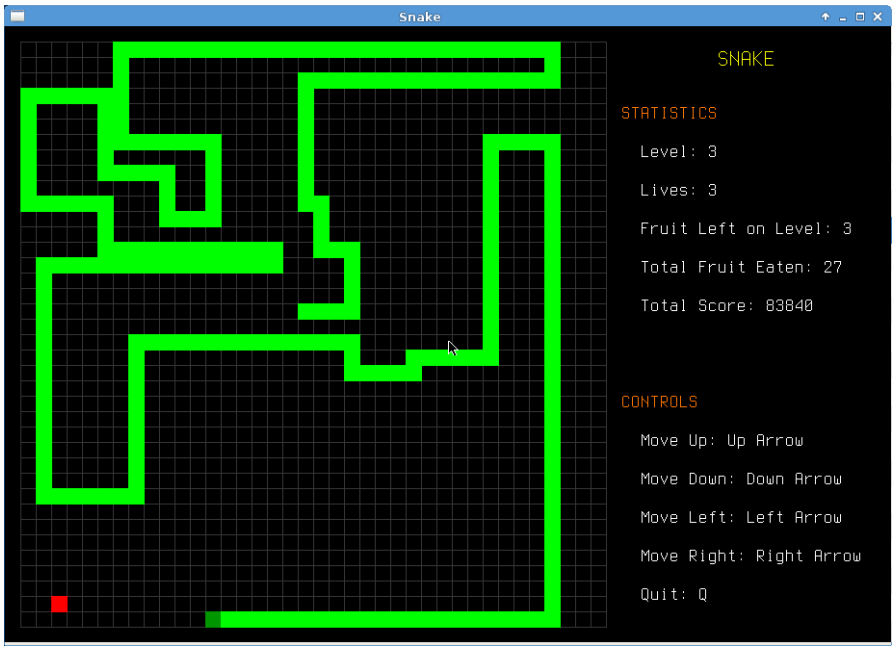


Рисунок 2.3 – Иллюстрация игры "Змейка"

Из рисунка 2.3 можно заметить, что управление в змейке также совпадает с возможными действиями каждого агента в Pig chase: перемещение влево, вправо, вниз и вверх. Самое большое различие – это обработка столкновения персонажа с препятствием или краем уровня. В случае змейки игровой прогресс полностью сбрасывается, а в случае с Pig chase агент не двигается с места, при этом уменьшая счетчик максимального

количества действий до окончания игры. Поскольку столкновение с препятствием глобально не влияет на продвижение агента к цели, этим различием можно пренебречь и применять модель обучения, аналогичную модели обучения змейки.

К счастью, наиболее частые модели машинного обучения, используемые для игры в змейку, основываются на методе Q-обучения. Рассмотрим состояния, входящие в Q-таблицу для игры «Змейка». [3]

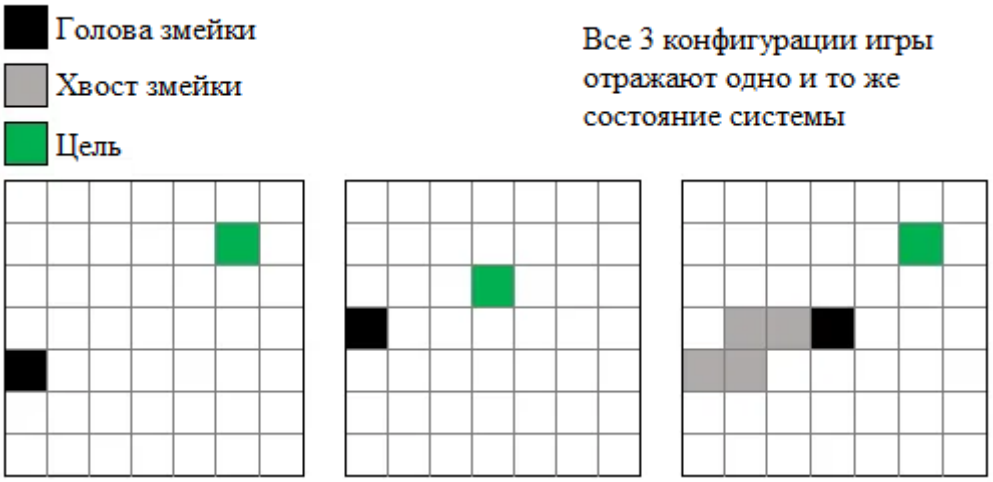


Рисунок 2.4 – Визуализация состояний системы для игры «Змейка» [3]

Как было упомянуто ранее, невозможно сохранить все комбинации положений головы змейки, цели и препятствий (каждого блока хвоста змейки) для поля, представленного на рисунке 2.4 в качестве примера. Таким образом, состояние системы собирается из следующих параметров:

1. Горизонтальное положение головы змейки относительно цели (находится справа, слева или на том же уровне)
2. Вертикальное положение головы змейки относительно цели (аналогично пункту 1)
3. Наличие препятствий в соседних клетках слева, справа, сверху и снизу.

Третий пункт представляет собой комбинацию из четырех бинарных значений, соответствующих каждой стороне. Таким образом, Q-таблица имеет $3 * 3 * 2^4 = 144$ строки и 4 столбца, соответствующих каждому перемещению головы змейки на соседние клетки, что значительно меньше количества всех комбинаций параметров среды. Стоит обратить внимание,

что границы уровня, хотя и не показаны на рисунке 2.2, обрабатываются аналогично препятствиям.

В [3] положительные награды выдаются за продвижение головы змейки в сторону цели и за достижение цели, а отрицательные за столкновение с препятствием (хвостом или краем уровня) и удаление головы от цели. Однако система наград в Pig chase отличается: агента наказывают за каждое выполненное действие, независимо от его влияния на игровой процесс, а поощряют только за достижение цели. В связи с изменением структуры уровня система награждений агента также должна быть отредактирована.

Поскольку для определения намерений и предсказания поведения напарника было решено не вводить никаких дополнительных способов коммуникации между агентами, а делать выводы только на основании его положения, во второй части задания также используется метод Q-обучения. Несмотря на простоту данного метода, он должен показать хорошие результаты, поскольку при рассмотрении оригинального уровня (без расширения и других модификаций) в большинстве ситуаций можно ограничиться фиксированным набором правил классификации поведения, независимо от алгоритма достижения цели агентом. Так, например, в [5] участники предпочли метод планирования методу обучения модели. То есть в их работе на этапе обучения строилось дерево поиска, чтобы в дальнейшем составлять (планировать) по нему самый оптимальный план действий.

### 2.2. Теоретические исследования и расчеты

Для достижения оптимальных значений в ячейках Q-таблицы такого размера необходимо несколько сотен или тысяч эпизодов обучения. Среда игры Minecraft слишком требовательна к ресурсам для тренировки модели, поэтому было принято решение создать упрощенную версию виртуальной среды. Поскольку оба агента и существо перемещаются только горизонтально, трехмерный уровень можно представить в виде двумерного

массива, содержащего свободные ячейки и ячейки с препятствиями. Пример такого массива изображен на рисунке 2.2. Агенты и существа способны дискретно перемещаться в соседние клетки этого массива, достигая своих целей по тем же правилам, что и в трехмерном пространстве.

Коэффициенты обучения и дисконтирования взяты напрямую из [3] в связи со схожестью задач: 0.7 и 0.5 соответственно, однако коэффициент случайности действий, изначально равный 0.9, не фиксирован на протяжении всего периода обучения, а уменьшается в 1.3 раза после каждого эпизода. Значение подобрано таким образом, чтобы к 10000 шагу агента случайности в его действиях почти не осталось. Поскольку на этом этапе проектируется только алгоритм достижения цели одним агентом, необходимо изменить эту цель соответствующим образом: координаты агента и цели должны совпадать, только в таком случае текущая игра (эпизод) завершается.

Качество первой половины процесса обучения может сильно сказаться на результирующей модели. Могут возникнуть ситуации, когда агент, совершая случайные движения, не смог добраться до цели несколько эпизодов подряд. В таком случае модель может перестать уделять внимание направлению к цели, учитываемому при формировании состояния системы. Ситуацию усугубляет отсутствие разницы между наградами при движении агента в сторону цели или в противоположную ей (эта разница присутствует в реализации Q-обучения для игры «Змейка» в [3]). Такое безразличие системы может отражаться в бесконечном зацикленном переходе агента между двумя соседними клетками. Существует 3 решения этой проблемы:

1. Запрет перемещения в клетку предыдущего местоположения агента;
2. Большая по модулю отрицательная награда за перемещение в клетку предыдущего местоположения агента;
3. Добавление небольшой случайности движения на время тестирования агента (чтобы присутствовала вероятность выйти из цикла, перейдя на другую соседнюю клетку);

Первый способ полностью отрезает возможность входить в циклы из двух смежных клеток, однако приводит к появлению другой проблемы. Ее можно обнаружить при детальном рассмотрении следующей ситуации:

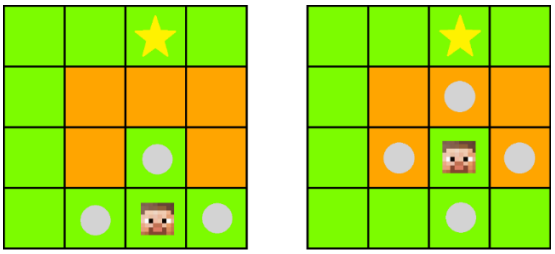

Рисунок 2.5 – Агент не может сделать шаг назад и оказывается в тупике

На рисунке 2.5 представлена ситуация, в которой агент, исходя из текущего состояния системы, не обнаруживает препятствий на пути к цели (звездочке) и делает шаг в ее сторону, оказываясь в ловушке. Поскольку цель может находиться «сверху» на любом расстоянии, эта ситуация часто встречается на практике, следовательно, убирать возможность делать шаг назад нерационально.

Возможность наказания агента является оптимальным вариантом, чтобы заставить его избегать зацикливаний. Таким образом в параметры состояния системы добавляется последнее действие агента (что по смыслу аналогично направлению последней посещенной клетки, но более стандартизировано). Однако данное решение спасает только от самых простых циклов, включающих в себя две соседние клетки. Агент может избежать наказания, используя четыре клетки, что также является ситуацией, часто встречающейся на практике. Составлять массив всех посещенных ячеек нерационально, поскольку цель может изменять свое положение, что приведет к поиску более длинного пути, и сильно увеличится размер данных, описывающих состояние системы.

На случай более комплексных циклов необходимо ввести небольшую вероятность совершения агентом случайного действия. Эта случайность должна присутствовать как в период обучения, так на время тестирования агента, поскольку столкновение с препятствием в среде игры Minecraft не приводит ни к каким серьезным последствиям. Выбранное значение

вероятности совершения случайного действия равняется 0.00005. Оно совсем незначительно для эпизодов, в которых вход агента в цикл отсутствует.

Побочными эффектами, разрывающими такие циклы, являются превышение количества максимального количества шагов за эпизод, перезапускающее игру, и перемещение цели, являющееся неотъемлемой частью виртуальной среды.

Перемещение агента со случайным поведением не может быть реализовано посредством выбора случайной соседней клетки. Перед совершением действия агент отсекает все допустимые перемещения, приводящие в клетку с препятствием, после чего производит случайный выбор из списка оставшихся действий. Такой подход делает намерения агента со случайным поведением более предсказуемыми. Перемещение сущности не включается в себя всю логику поведения существ в игре Minecraft, поскольку оно является случайным и существа большую часть времени находятся в одном положении.

Q-обучение модели распознавания намерений осуществляется только после получения оптимальных значений в Q-таблице на предыдущем этапе, чтобы разница между поведениями двух типов агентов была наиболее очевидна. Поскольку поведение и действия агентов можно наблюдать только во времени, данных, описывающих состояние системы в алгоритме навигации агентов по уровню, недостаточно. Самым очевидным решением будет создать отсортированную по времени цепочку состояний. Однако это решение приведет к большому потреблению памяти, вследствие чего должно быть оптимизировано. Поскольку целенаправленный агент должен приближаться к цели с течением времени, что отличает его от агента со случайным поведением, можно составить новое состояние системы для каждого агента из следующих параметров:

1. Расстояние от агента до цели в метрике L1 (манхэттенское расстояние);
2. Порядковый номер шага агента с начала текущего эпизода (игры).

Вместо евклидова расстояния предпочтительно использовать манхэттенское, поскольку при любом состоянии системы оно равняется некоторому целому числу, что позволяет легче его классифицировать при обращении к Q-таблице. В таком случае интервал возможных значений расстояния равен [0, 18] для уровня, изображенного на рисунке 2.2 (максимум 9 клеток по горизонтали и 9 по вертикали относительно цели). Интервал допустимых порядковых номеров располагается от 0 до максимального количества шагов за одну игру. Данное решение позволяет определять намерения агента, используя только расстояние от него до цели в каждый дискретный момент времени. Однако, поскольку большую роль в положении агента относительно цели играет случайность, время сходимости значений в ячейках Q-таблицы к оптимуму может оказаться большим. В связи с этим было принято решение сгруппировать значения порядковых номеров шагов.

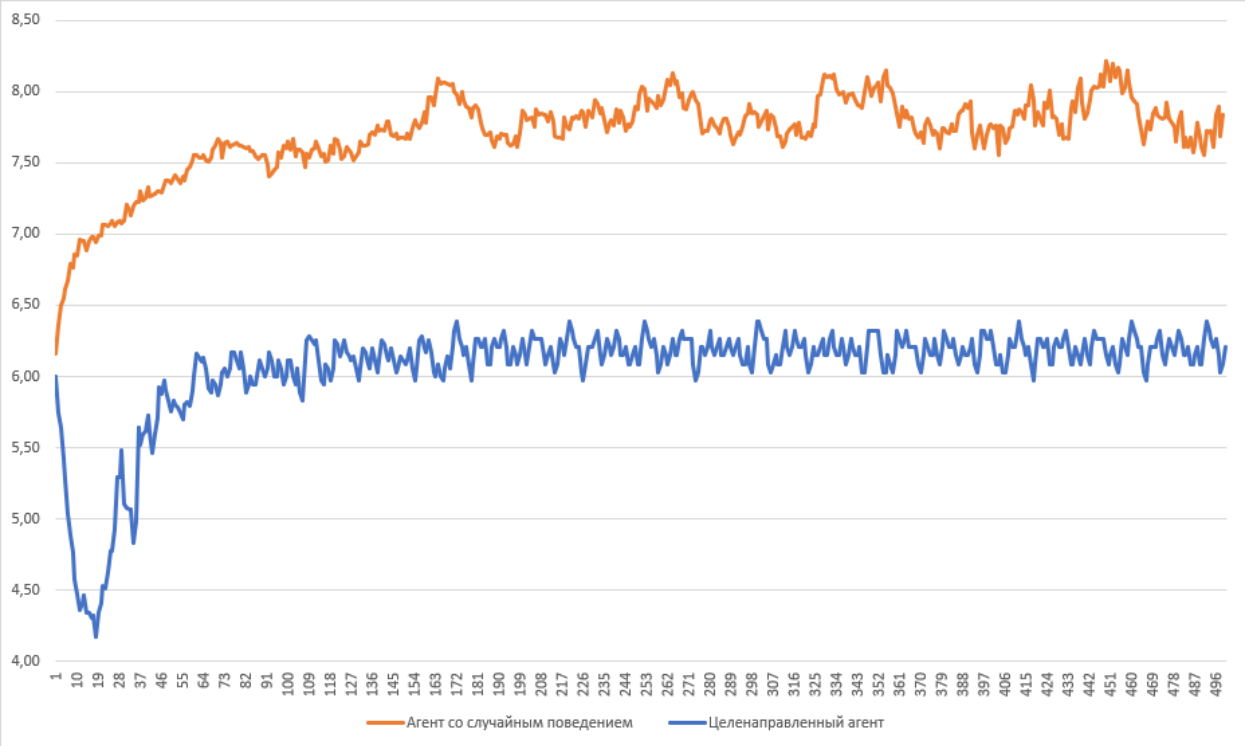


Рисунок 2.6 – Среднее расстояние агентов до цели на каждом шаге

Рисунок 2.6 иллюстрирует среднее расстояние от агентов до цели на каждом шаге игры. Максимально допустимое количество шагов за один эпизод в данном примере равняется 500. Стоит обратить отдельное внимание на то, что если игра заканчивается раньше 500-ого шага агента, оставшиеся шаги

игнорируются и массив расстояний, соответствующим им, ничем не заполняется. Другими словами, чем больше номер шага по оси абсцисс, тем меньше данных для этого шага было собрано. На данном рисунке можно заметить, что различия между агентом со случайным поведением (отмечен оранжевым) и целенаправленным агентом (отмечен синим) появляются уже на первом шаге. Затем, наибольшая разница между расстояниями наблюдается примерно на 19 шаге.

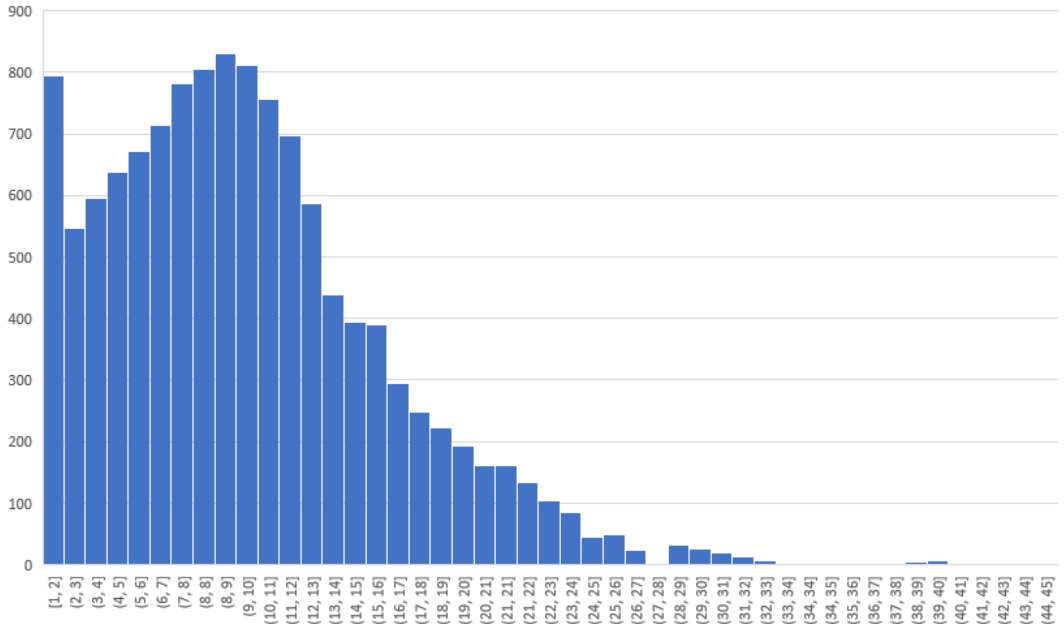


Рисунок 2.7 – Количество завершенных игр для порядкового номера каждого шага целенаправленного агента с начала эпизода

Однако исходя из рисунка 2.7, можно отметить, что большинство эпизодов заканчиваются уже на 9 шаге целенаправленного агента, соответственно, нерационально разделять порядковые номера шагов на две группы относительного 19 шага. В то же время на 9 шаге разница между расстояниями двух типов агентов недостаточно большая. Стоит обратить внимание на то, что агент, предсказывающий намерения напарника, не может мгновенно завершить игру. Для начала ему необходимо добраться до заданных координат (цели). Следовательно, поскольку среднее расстояние до цели на 19 шаге равняется примерно 4, разделить порядковые номера на группы можно относительно 15 шага. Как показывает гистограмма на рисунке 2.7, большое количество игр было завершено до 15 шага, что

позволяет выбрать его в качестве точки разделения двух групп порядковых номеров шагов. Таким образом, порядковые номера шагов агента от начала игры были разделены на 3 группы: до 1 шага включительно, до 15 шага включительно и остальные.

Что касается столбцов Q-таблицы метода определений намерений – их всего два: агент придерживается случайного поведения и целенаправленного. Вместо описания действий, как в Q-таблице, используемой для навигации агента по уровню, здесь они играют роль классификации поведения напарника. Результат классификации используется для определения цели агента, выполняющего анализ поведения. Целью могут являться либо координаты существа, либо координаты точки преждевременного завершения игры.

Для возможности переноса обученной модели в среду для тестирования предусмотрена запись обеих Q-таблиц в файл. На стороне игры Minecraft посредством языка Python и модификации, предоставленной проектом Malmo, реализован целенаправленный агент с аналогичной функциональностью, совершающий действия на основании значений, хранящихся в файле с Q-таблицей. Часть функциональности агента была адаптирована под работу со структурой, описывающей состояние новой среды. Часть, отвечающая за обучение обеих моделей, была полностью вырезана. Более подробное сравнение структур состояния производится в следующем разделе.

### 2.3. Модель, блок-схемы кода программы

В данном разделе приведены блок схемы алгоритмов и диаграммы классов как для упрощенной среды, так и для среды игры Minecraft, организованной посредством модификации игры, предоставленной проектом Malmo. Весь код для данной ВКР написан на языке Python. Диаграммы созданы при помощи Visio.

Основные элементы алгоритма работы упрощенной окружающей среды приведены на рисунке 2.8.

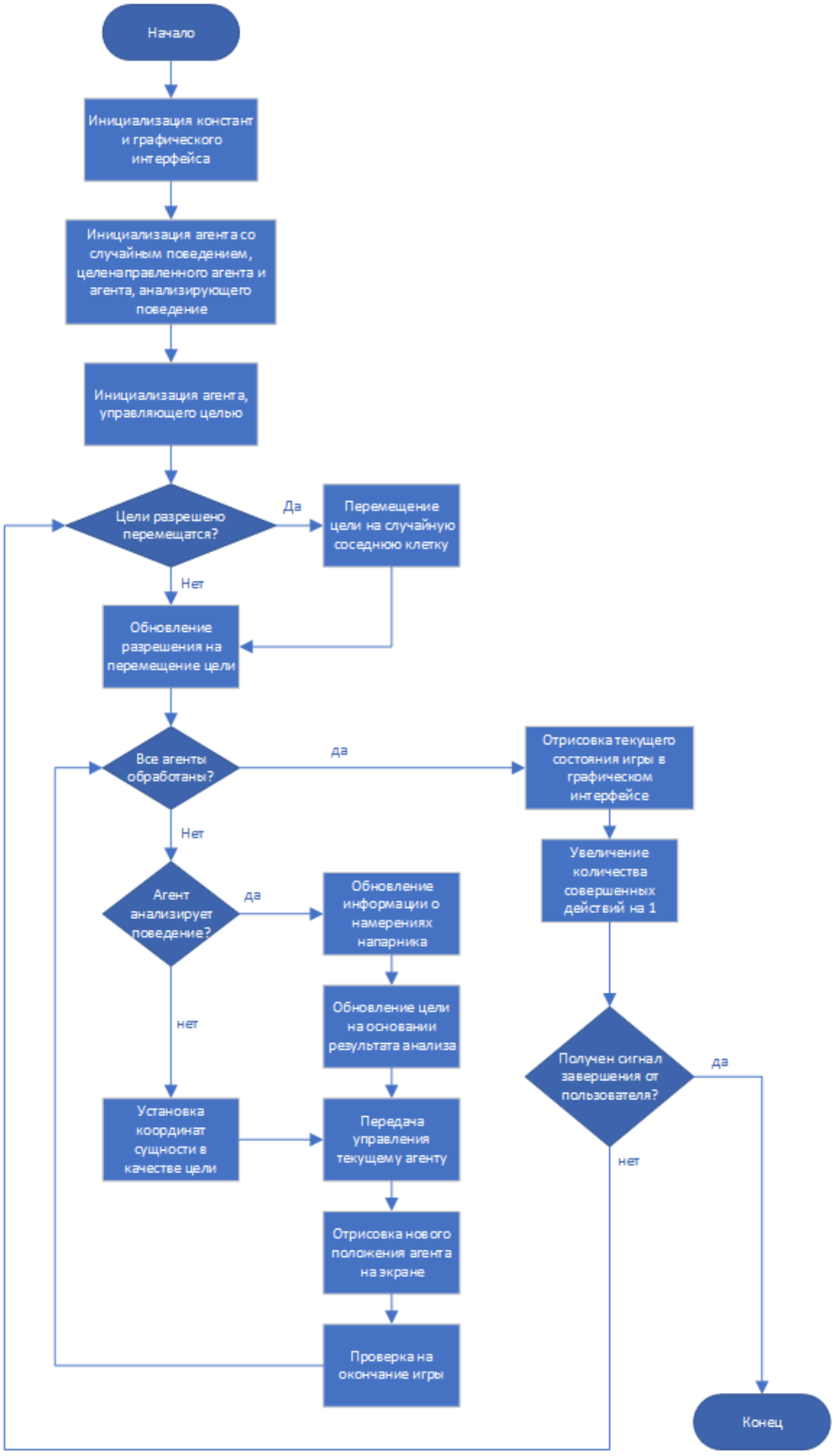


Рисунок 2.8 – Обобщенный алгоритм работы упрощенной виртуальной среды

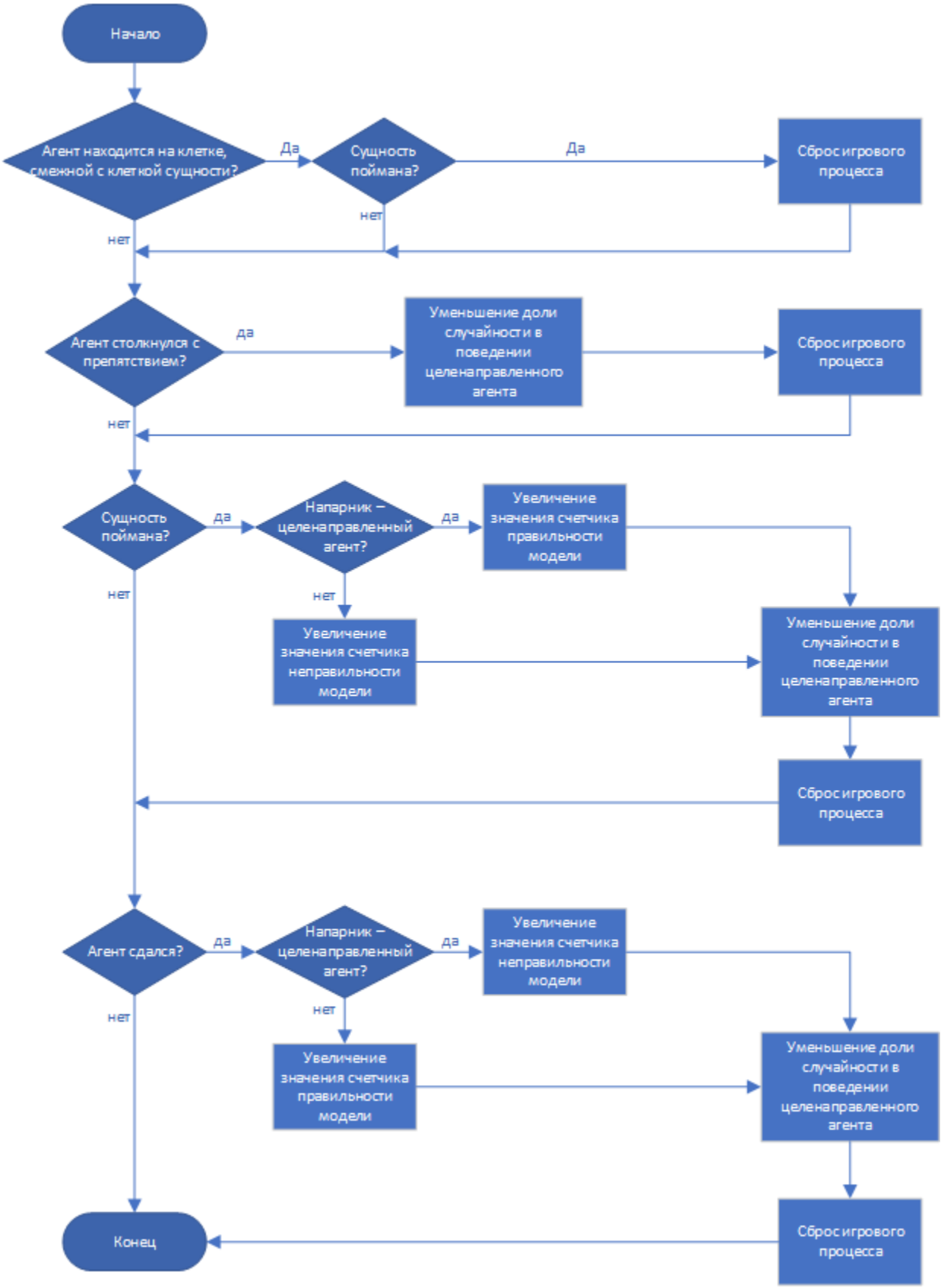


Рисунок 2.9 – Детализация блока «Проверка на окончание игры»

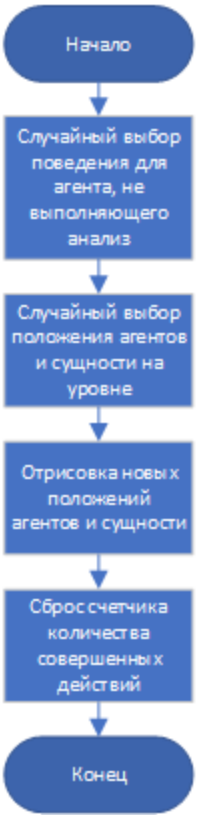


Рисунок 2.10 – Детализация блока «Сброс игрового прогресса»

На рисунке 2.8 можно заметить возможность завершения работы алгоритма только по требованию пользователя. Это связано с тем, что после окончания обучения модели программа автоматически переходит в режим тестирования, отображая игровой процесс в реальном времени. Из режима тестирования пользователь выходит, самостоятельно завершив выполнение программы путем закрытия окна графического интерфейса. Исходный код алгоритма на рисунках 2.8, 2.9 и 2.10 приведен в приложении А, Листинг А.1.

Перемещение сущности производится через класс агента со случайным поведением. Это отражается в блоке «Инициализация агента, управляющего целью». Экземпляр этого конкретного агента не входит в список агентов системы, следовательно, его поведение не анализируется. При проверке условия «Все агенты обработаны?» он также не учитывается. Периоды движения и остановки сущности определяются специальным таймером, который может принимать значения от -5 до 10. При отрицательном значении таймера сущности разрешено двигаться, и управление передается классу, реализующему поведение «случайного» агента. При положительном

значении сущность стоит на месте. Каждое совершенное действие в системе приближает таймер к 0 на единицу (выполняет действие декремента или инкремента). Как только значение таймера достигает 0, выбирается новое случайное значение в диапазоне [-5, 10]. Эти два предложения описывают блок «Обновление разрешения на перемещение цели».

Блок «Проверка на окончание игры» детально изображен на рисунке 2.9. События, приводящие к окончанию игры, включают в себя столкновение агента с препятствием или краем уровня, успешную поимку сущности, и преждевременное завершение игры одним из агентов. Можно заметить, что перед каждым сбросом игрового процесса у целенаправленных агентов уменьшается доля случайности в них поведении. Другими словами, сужается интервал, в который должно попасть случайно выбранное число, чтобы агент предпринял случайное действие. Для агента со случайным поведением данный блок игнорируется.

Блок «Сброс игрового процесса» вынесен в отдельный рисунок 2.10 в связи с многочисленными повторными использованиями его в рисунке 2.9. Он отвечает за сброс всех достижений агентов за последний эпизод, кроме общей статистики, которая необходимо пользователю для оценки результатов работы программы. Также внутри этого блока производится случайный выбор поведения для оцениваемого агента. На уровне кода экземпляр класса агента одного типа заменяется экземпляром другого типа, либо остается на своем месте.

Рассмотрим иерархию агентов, представленную в виде диаграммы классов на рисунке 2.11. Иерархия построена таким образом, чтобы максимально облегчить перенос функционала агентов в среду тестирования, то есть, встроить в проект Malmo. Observer является абстрактным классом, представляющим объект, имеющий координаты (position), ограничение на максимальное количество шагов за игру (max_fitness), сведения о последнем действии (last_move), состояние достижения цели (goal_achieved) и доступность цели (edible). Под доступностью цели понимается возможность

окружения сущности. Если в системе определено 2 агента, то на соседних от цели клетках должно располагаться как минимум по 2 препятствия, в таком случае цель будет достижима.

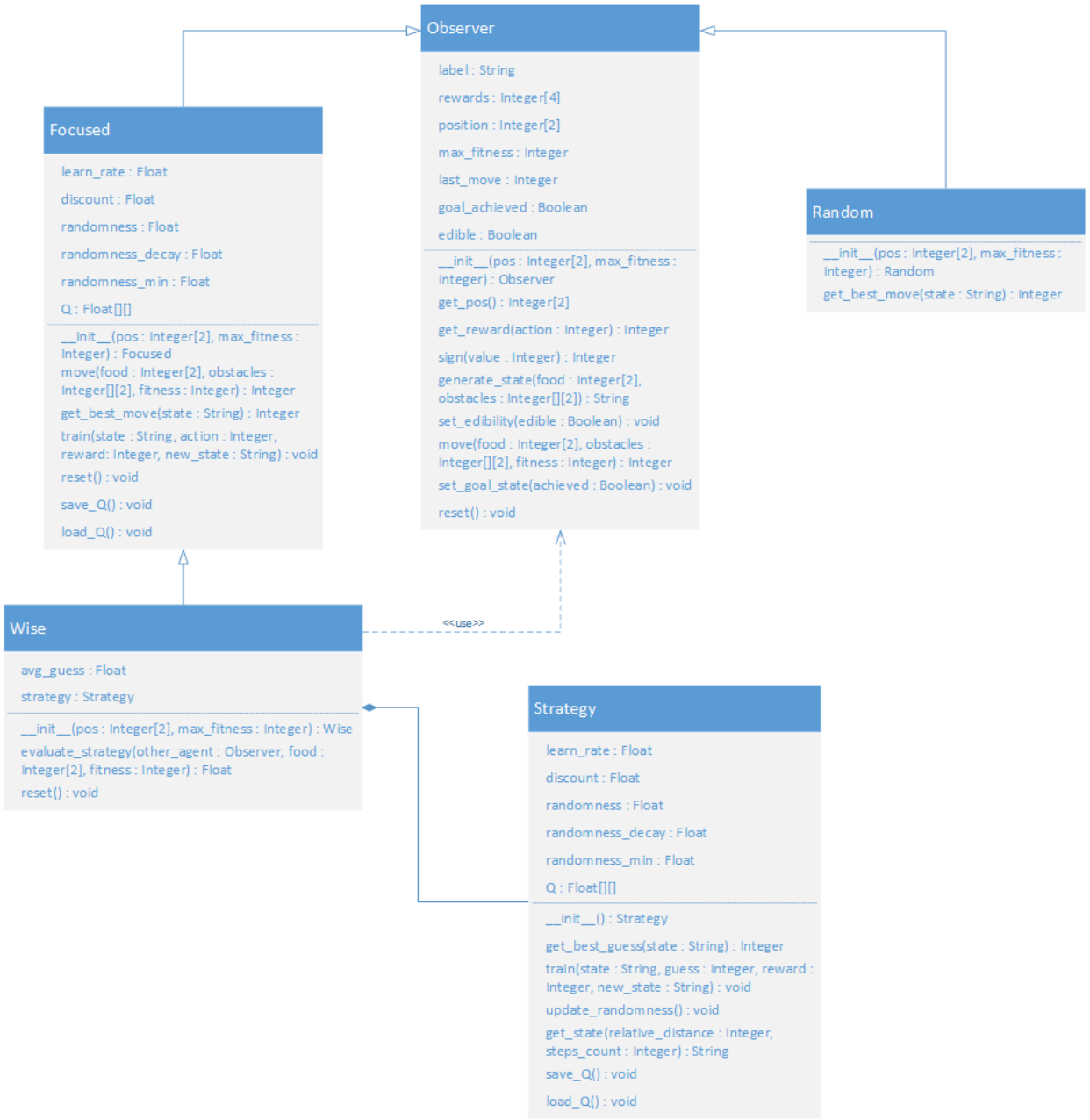


Рисунок 2.11 – Диаграмма классов агентов

Также в классе Observer содержатся сведения о награждении агентов за выполнение соответствующих действий. Массив rewards является именованным списком, включающим в себя такие действия, как «шаг», «шаг назад», «столкновение с препятствием» и «достижение цели». Разделения между достижением общей цели и завершением игры нет, поскольку цель полностью зависит от поведения напарника. В массиве position, тоже

являющимся именованным списком, хранятся «x» и «y» координаты объекта на карте уровня. Бинарная переменная goal_achieved устанавливается в значение true, когда объект располагается вплотную к цели. Она необходима для исключения любых действий со стороны агента, если цель уже достигнута. Переменная сбрасывает значение до false, если цель меняет свое положение и условие достижения цели больше не выполняется. Класс имеет конструктор, принимающий начальное положение объекта на карте уровня и ограничение на максимальное количество шагов. Функция sign является вспомогательной и возвращает знак числа value (1, если положительное, -1, если отрицательное, иначе 0). Метод generate_state преобразует координаты объекта, еды и препятствий в строку, представляющую текущее состояние системы и пригодную для индексирования Q-таблицы по строкам. Функция move отвечает за перемещение объекта по уровню, выдачу наград и обработку глобальных событий в системе. Она проверяет корректность перемещения объекта: его столкновение с препятствием, выход за границы уровня, превышение максимально допустимого количества шагов за игру; обрабатывает возвращение агента на предыдущую позицию (шаг назад) и вызывает функцию обновления Q-таблицы для агентов, которым это необходимо. Функция reset сбрасывает переменную goal_achieved.

Класс Focused основывается на абстрактном классе Observer и представляет собой целенаправленного агента. В состав его переменных входят константы, необходимые для Q-обучения: коэффициенты обучения (learn_rate) и дисконтирования (discount), случайность (randomness), минимальный порог случайности (randomness_min) и коэффициент уменьшения случайности (randomness_decay). Самой важной переменной является сама Q-таблица, являющаяся двумерным именованным списком. Конструктор класса принимает те же значения, что и конструктор родителя, и инициализирует переменные, заполняя всю Q-таблицу нулями. Функция move добавляет условия отказа от действий: если цель недостижима или если цель уже достигнута. При невыполнении этих условий вызывается

родительская функция move. Была добавлена функция get_best_move, возвращающая либо наилучшее действие на основании значений Q-таблицы, либо случайное в зависимости от переменной randomness. Функция train обновляет значение ячейки Q-таблицы, находящейся на пересечении state и action (входных параметров функции), в соответствии с уравнением Беллмана. В функцию reset, как было упомянуто ранее, добавлена функциональность уменьшения случайности поведения агента. Методы save_Q и load_Q служат для записи и чтения файла с Q-таблицей соответственно.

Вторым базовым типом агента является агент со случайным поведением (класс Random). Его конструктор не инициализирует переменные, а сразу вызывается конструктор класса Observer. Функция get_best_move находит подмножество действий, не приводящих к немедленному завершению игры в текущем состоянии системы. Затем производится случайный выбор действия из этого подмножества.

Самым важным классом является класс мудрого агента (Wise), наследующий функциональность целенаправленного агента (Focused) и выполняющий анализ поведения любого объекта (Observer). Данный класс добавляет две переменные: avg_guess, отражающий уверенность в целенаправленности напарника усредненную по всем его действиям, совершенным в течении одного эпизода, и strategy – класс, отвечающий за анализ поведения. В конструкторе значение avg_guess устанавливается в 1 (полная уверенность в готовности напарника достичь общей цели) и инициализируется объект класса Strategy, конструктор которого не имеет входных параметров. В функции evaluate_strategy выполняется оценка поведения напарника и выносится результат: 0, если, по мнению алгоритма, напарник действует случайным образом и не готов кооперироваться, или 1 в обратном случае. Для минимизации ошибок алгоритма это двоичное значение заносится в переменную avg_guess в соотношении 1 к 4, тем самым немного изменяя общую уверенность системы. Возвращаясь к рисунку 2.8,

блок «обновление цели на основании анализа [поведения напарника]» выбирает сущность в качестве цели, если значение переменной avg_guess превышает порог 0.5, иначе целью становятся координаты точки преждевременного завершения игры. Функция reset производит уменьшение случайности для объекта класса Strategy (так как он тоже основан на методе Q-обучения) и сбрасывает значение avg_guess до 1, поскольку поведение оцениваемого агента в следующем эпизоде может измениться.

Исходный код и иерархия классов агентов приведены в приложении А, Листинг А.2. Данный код должен размещаться в файле с названием agents.py, поскольку код в Листинге А.1 содержит ссылки на данный файл.

Класс Strategy имеет структуру, аналогичную классу Focused (без учета класса Observer). Функция get_best_guess по функциональности соответствует функции get_best_move. Метод get_state принимает расстояние от анализируемого агента до цели в метрике L1 и номер группы, к которой принадлежит количество действий, совершенных с начала текущего эпизода. Строка, возвращенная методом get_state используется для индексирования соответствующей Q-таблицы по строкам. Исходный код класса Strategy приведен в приложении А, Листинг А.3. Класс должен находиться в файле strategy.py, поскольку в Листинге А.2 содержатся ссылки на него.

Класс Design не имеет отношения к обработке действий в виртуальной среде, а используется только для удобства представления результатов работы программы. Он будет разобран в соответствующем разделе.

В среду для тестирования (проект Malmo) был перенесен только один класс Wise, поменявший название на CustomAgent, поскольку целенаправленный агент и агент со случайным поведением по умолчанию предоставляются модификацией. Измененная иерархия и структура классов агентов представлена на рисунке 2.12.

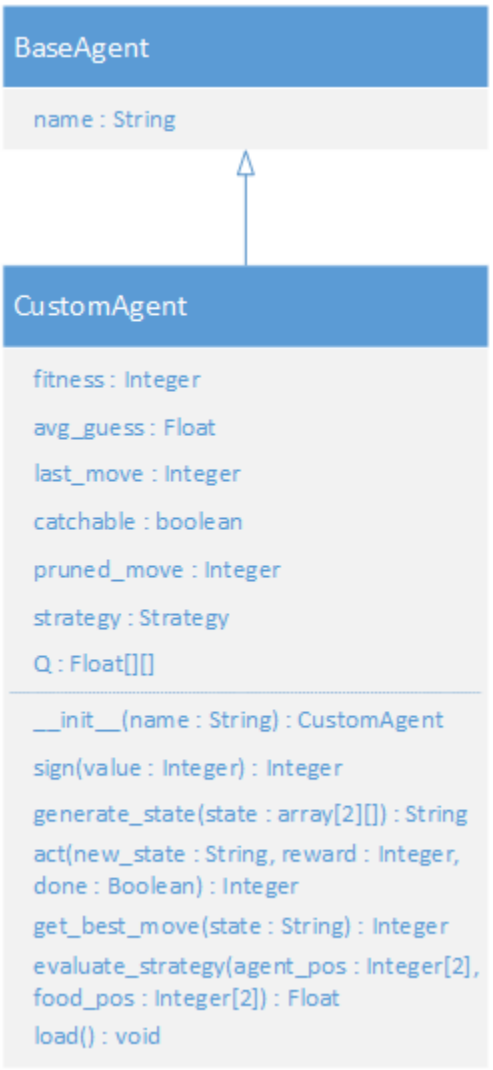


Рисунок 2.12 – Иерархия классов агентов в проекте Malmo

BaseAgent является абстрактным (базовым) классом агента в среде проекта Malmo, наподобие класса Observer в упрощенной среде. На рисунке 2.12 указана только одна переменная, принадлежащая этому классу: name, поскольку это единственная унаследованная переменная, используемая в CustomAgent.

Содержимое конструктора и функций sign, evaluate_strategy и load (аналог load_Q) ничем не отличается от содержимого соответствующих функций класса Wise. Функция generate_state включила в себя некоторые действия, представленные в виде блоков на рисунке 2.8. Ее единственный входной параметр state содержит положение всех объектов на карте и саму карту уровня. Из него извлекается положение сущности и производится проверка на возможность ее поимки (результат записывается в переменную catchable,

прежде несущую название edible). Затем вызывается функция evaluate_strategy, обновляющая переменную avg_guess. За этим следует нововведение: обрезка недопустимых действий. Недопустимыми считаются действия, которые приводят к преждевременному завершению игры, если агент еще не готов сдаваться. Другими словами, формируется подмножество действий, которые не приведут агента на координаты точки завершения игры, если агент считает, что его напарник готов кооперироваться. Поскольку существует только одна точка завершения игры, количество обрезанных действий не может превышать 1. Найденное действие заносится в переменную pruned_move. Дальнейший функционал generate_state повторяет метод generate_state класса Observer. Функция act, так же как функция move из упрощенной среды, возвращает идентификатор действия, совершенного агентом. Однако ее содержимое сильно отличается. Проверяется входной параметр done, сигнализирующий о завершении текущего эпизода, и требующий сброса всех параметров к значениям по умолчанию (по аналогии с функцией reset). Затем проверяется возможность поимки сущности. При отсутствии такой возможности агент остается на своем месте, иначе обращается к Q-таблице за поиском наилучшего действия. Перед выходом из функции текущее действие сохраняется в last_move и инкрементируется fitness. Внутри функции get_best_move добавляется удаление обрезанного действия (идентификатор которого содержится в pruned_move) перед тем, как производить сравнение вариантов по Q-таблице. Список наград, как и вся часть, ответственная за обучение модели, был вырезан из класса CustomAgent.

Из класса Strategy также была вырезана часть с обучением модели, но оставшаяся структура не претерпела никаких изменений, поскольку класс CustomAgent единственный взаимодействующий с ней.

Список действий агента увеличился на один элемент: добавилось действие «бездействия», поскольку в данной среде у агента нет возможности просто «пропустить ход».

### 2.4. Вывод по разделу

В данной работе были разработаны и обучены две модели: модель навигации ИИ-агента по двумерному уровню с препятствиями и модель классификации поведения напарника. Обе модели основываются на методе Q-обучения, поскольку требуют небольшого количества дискретных входных и выходных данных, что позволяет хранить все возможные состояния системы в памяти в виде Q-таблицы.

Несмотря на относительно небольшое количество возможных состояний системы, модель требует оптимизации, так как обучение должно заканчиваться только когда ИИ-агент сможет выбирать самое оптимальное действие для каждого состояния системы. Другими словами, чем больше состояний имеет система, тем больше времени требуется для достижения оптимальных значений в Q-таблице. Модель классификации поведения была оптимизирована путем категоризации количества шагов, совершенных анализируемым агентом с начала текущей игры.

Таким образом, необходимо использовать наиболее простые методы управления ИИ-агентами в совместных системах для повышения управляемости, прозрачности и, соответственно, предсказуемости.

# 3. РЕЗУЛЬТАТЫ РАЗРАБОТКИ

В этом разделе буду приведены результаты работы алгоритма как в упрощенной среде, так и в среде для тестирования, поскольку формы представления результатов в этих средах различаются.

Участники соревнований Malmo Challenge предоставляют свои результаты в виде демонстрации работы своего алгоритма в двух сценариях: с напарником, действующим случайным образом, и целенаправленным напарником. [5] Обычно этой демонстрации достаточно, чтобы показать работоспособность алгоритма. Некоторые участники также продемонстрировали третий вид коллаборации: человек с агентом, анализирующим его поведение.

Участники, разместившие свое решение в [6], абстрагировались от среды, предоставленной проектом Malmo, и использовали методы глубокого машинного обучения. «Глубокое обучение (глубинное обучение; англ. Deep learning) – совокупность методов машинного обучения (с учителем, с частичным привлечением учителя, без учителя, с подкреплением), основанных на обучении представлениям, а не специализированных алгоритмах под конкретные задачи.» [14] Входными данными для их алгоритма является набор пикселей, полученных напрямую из окна с трехмерным представлением виртуальной среды. Другими словами, «глазами агента». Такой подход дал участникам возможность представить результаты работы алгоритма с самых разных точек зрения: точность анализа окружающей среды, средняя величина наград за каждый эпизод, средняя по всем эпизодам величина наград за каждый шаг и т.д. Здесь точность определяется суммой ошибок, совершенных в течение одного эпизода.

Несмотря на то, что провести прямое сравнение результатов с другими участниками невозможно в связи с изменением правил оригинальных соревнований и разными временными ограничениями, результаты работы алгоритма ВКР будут представлены в наиболее подробном виде, совмещая

различные способы представления результатов участниками, занявшими призовые места.

### 3.1. Методика эксперимента

Эксперимент в упрощенной среде проводится по методике, представленной в виде алгоритма на рисунке 2.8. Данный алгоритм поддерживает два способа работы: обучение модели с последующим тестированием и тестирование готовой Q-таблицы. Поскольку модель перемещения агента по уровню играет хотя и важную, но второстепенную роль, производится обучение только модели анализа поведения напарника. Для гарантии достаточности обучения период тестирования начинается с 200000 шага агента, отсчитывая от запуска программы. Самым важным результатом, который предоставляется интерфейсом программы, является доля правильных определений намерений напарника, находящаяся в промежутке [0, 1]. Доля вычисляется и усредняется по всем сыгранным эпизодам по следующей формуле:

$$result = \frac{wins}{wins + loses}$$

Где wins – количество эпизодов, в которых агент правильно классифицировал поведение напарника, а loses – количество эпизодов, содержащих ошибки в определении намерений агентом. Обновление значений данных переменных производится на рисунке 2.9 в блоках «Увеличение значения счетчика правильности модели» (wins) и «Увеличение значения счетчика неправильности модели» (loses) соответственно.

В дополнение к доле эпизодов с правильным определением намерений программа предоставляет графический интерфейс пользователя. Он позволяет наблюдать за игровым процессом в реальном времени и сравнивать свое представление о поведении «напарника» с представлением анализирующего агента. Структура класса, содержащего элементы ГИП, представлена на рисунке 3.1.

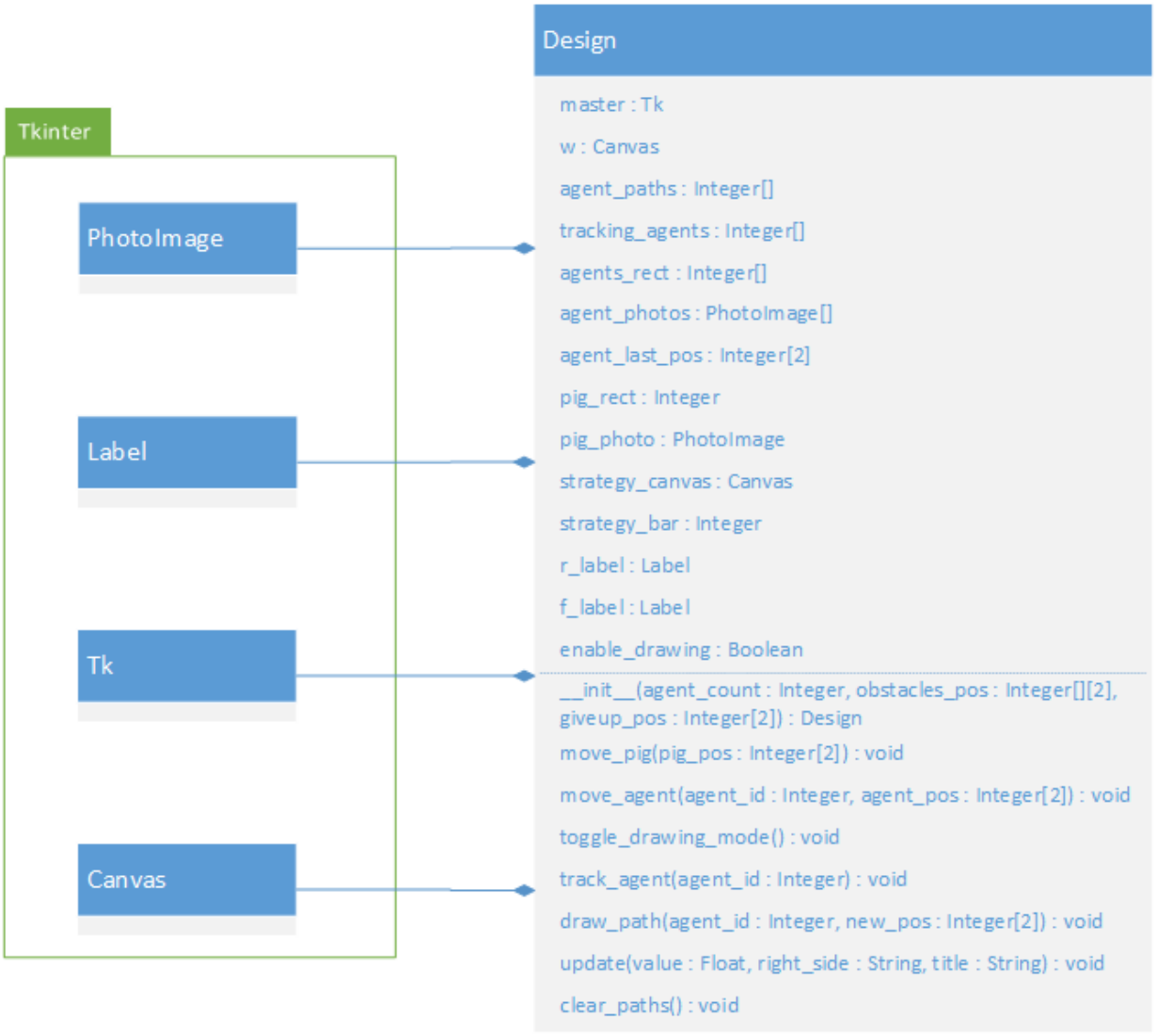


Рисунок 3.1 – Структура класса Design

В качестве графической библиотеки используется Tkinter для языка Python. «Tkinter (от англ. Tk interface) – кросс-платформенная событийно-ориентированная графическая библиотека на основе средств Tk (широко распространённая в мире GNU/Linux и других UNIX-подобных систем, портирована также и на Microsoft Windows)» [15]. Эта библиотека позволяет создавать простые ГИП, подходящие для представления результатов работы программы.

Главным компонентом класса Design, представленного на рисунке 3.1, является master – виджет верхнего уровня, отвечающий за параметры и содержание всего графического окна. Виджеты w и strategy_canvas типа класса Canvas содержат большую часть полезной нагрузки: игровое поле и результаты анализа поведения агентом соответственно.

Поле представляет собой сетку 10 на 10 клеток по осям абсцисс и ординат. На поле располагаются агенты, идентификаторы графических элементов которых сохраняются в переменную agents_rect. Агенты обладают уникальными текстурами для большего их выделения относительно фона и препятствий. Текстуры представляют собой тип PhotoImage, предоставляемый библиотекой Tkinter и находятся в agent_photos. Сущность имеет аналогичные характеристики и соответствующие им переменные: pig_rect и pig_photo. Элементы препятствий не сохраняются в памяти, поскольку остаются статичными до завершения работы программы.

Отрисовка перемещения объектов по уровню осуществляется через функции move_pig и move_agent соответственно.

Помимо структуры уровня и основных действующих лиц на холсте w изображается путь, проделанный анализируемым агентом. Весь путь агента за текущий эпизод сохраняется в agent_paths и очищается при вызове метода clear_paths. Путь состоит из отдельных элементов и пополняется при каждом движении агента. Более того, для лучшего понимания динамики перемещения объекта новые элементы пути выделяются яркими оттенками цвета, в то время как старые элементы со временем тускнеют. Функциональность отрисовки пути и изменение цвета содержится в функции draw_path. Пользователю предоставляется выбор агентов, путь которых необходимо отслеживать. Добавить агента можно посредством вызова track_agent передав функции порядковый номер агента в массиве всех агентов системы (за исключением агента, управляющего перемещением сущности).

Следующей важной составляющей ГИП является strategy_canvas, отображающий мнение агента касательно намерений его напарника. strategy_bar представляет собой строку состояния, имеющую диапазон допустимых значений равный [0, 1]. Она отражает значение переменной avg_guess класса Wise. С левой части строки находится 0, отмеченный надписью «R[andom]» (переменная r_label), с правой – 1 с пометкой

«F[ocused]» (переменная f_label). Текущее значение avg_guess помечается крупной точкой, «скользящей» по строке состояния в течение всего эпизода.

Стоит отметить, что обновление графического окна происходит не при изменении состояния графических элементов, а только при вызове функции update. Метод отвечает за отрисовку всех виджетов в их обновленном состоянии и поддержание актуальности названия окна, содержащего долю эпизодов с правильно определенными намерениями и реальный класс текущего анализируемого агента.

Поскольку программа поддерживает обучение модели, должна присутствовать функциональность отключения ГИП. Она содержится в функции toggle_drawing_mode, способной замораживать и размораживать весь графический интерфейс при ее вызове. Метод не принимает параметров, так как инвертирует текущее состояние заморозки при каждом вызове.

Исходный код класса Design приведен в приложении А, Листинг А.4. Класс должен находиться в файле design.py, поскольку в Листинге А.1 содержатся ссылки на него.

Алгоритм тестирования агентов в среде Malmo изображен в виде блок-схемы на рисунке 3.2. Перед процедурой тестирования необходимо обеспечить достаточное количество запущенных экземпляров игры Minecraft. В случае ВКР их должно быть не менее двух – свой под каждого агента. Проверка и запуск недостающих экземпляров осуществляются посредством блокирующей операции в блоке «Запуск двух экземпляров игры Minecraft». Как только экземпляры переходят в состояние готовности, программа выделяет по новому потоку для каждого агента. Это необходимо, поскольку, в отличие от упрощенной среды, в игре Minecraft все процессы происходят одновременно, а не пошагово. Все последующие действия, изображенные на рисунке 3.2, происходят одновременно в двух потоках. В самом начале функции потоков производится инициализация текущего агента: либо агента класса CustomAgent (рисунок 2.12), либо одного из агентов, предоставленных

средой Malmo (на выбор пользователя). Также возможен выбор второго экземпляра класса CustomAgent.

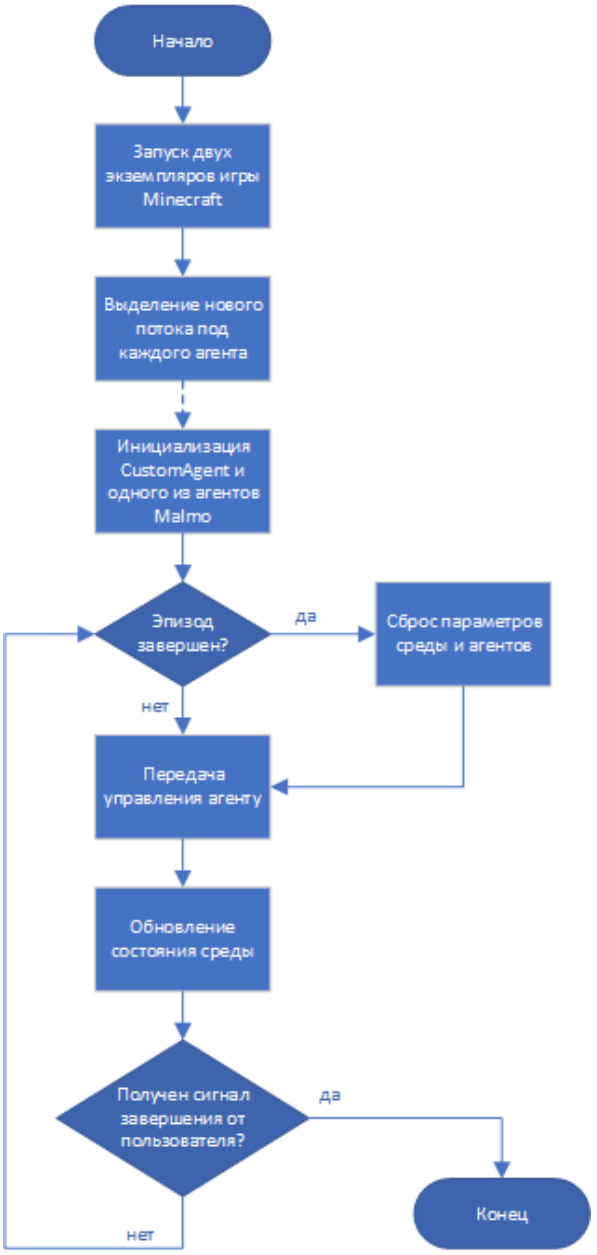


Рисунок 3.2 – Алгоритм тестирования агентов в среде Malmo

Затем начинается основной бесконечный цикл обновления состояния среды, пока пользователь самостоятельно не завершит выполнение программы. Блок «Передача управления агенту» вызывает функцию act класса агента, принадлежащего соответствующему потоку. Этап тестирования заканчивается по отправке комбинации клавиш Ctrl+C в консоль программы.

Среда игры Minecraft привносят свои трудности, отсутствующие при обучении в упрощенной среде. Одним из главных препятствий является

столкновение объектов, расположенных на один и тех же координатах. Другими словами, объекты не могут проходить сквозь друг друга, как они это делали на этапе обучении модели. На практике часто возникают ситуации, когда один из агентов, расположившись между несколькими препятствиями, приводит к созданию полноценного лабиринта на карте уровня. К сожалению, модель, основанная на Q-обучении, не способна проходить динамические (меняющиеся со временем) лабиринты, поскольку это противоречит требованию сходимости алгоритма. Для достижения наилучших результатов необходимо отключить обработку столкновений объектов. Для этого достаточно ввести следующие команды в окне игры Minecraft, нажимая Enter после ввода текста из каждого пункта:

1. /scoreboard teams add no_coll
2. /scoreboard teams option no_coll collisionRule never
3. /scoreboard teams join no_coll @e

Этот метод отключает обработку столкновений всех объектов, находящихся на данный момент в виртуальной среде. Необходимо повторно ввести команду №3, если набор объектов изменится с момента последнего ее выполнения. Чтобы убедиться в успешном выполнении команд в левой части рисунка 3.3 показана включенная обработка столкновений, в правой – ее отсутствие.


Рисунок 3.3 – Результат отключения обработки столкновений

В обоих случаях для тестирования модели использовался уровень, изображенный на рисунке 2.2. В структуру уровня входят 25 препятствий

(без учета границ уровня), образующих разреженный лабиринт. Именно для такого типа уровней создавался алгоритм навигации агента. Уровень имеет несколько особенностей: выемка, обозначенная на рисунке 2.5, и отсутствие препятствий, прилегающих к краям уровня. Первое необходимо для корректной обработки и тестирования возвращения агента на предыдущее местоположение, второе – для оптимизации поиска пути к цели.

### 3.2. Результаты эксперимента

Результат реализации ГИП для упрощенной среды представлен на рисунке 3.4.

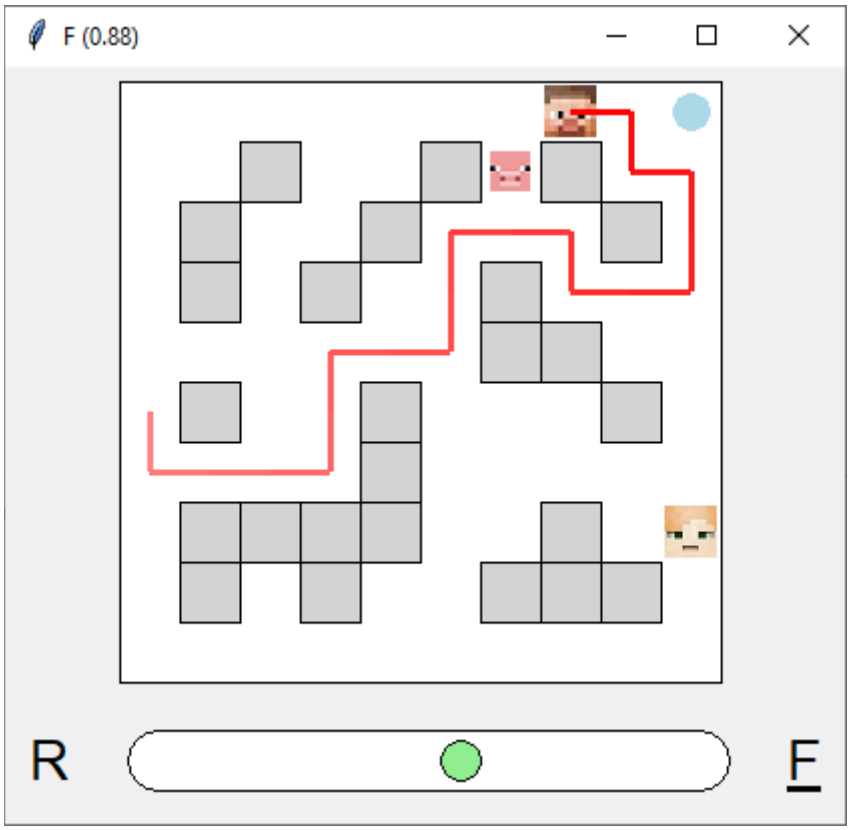


Рисунок 3.4 – ГИП для упрощенной среды

Исходя из обозначения «F», присутствующего в названии окна (левый верхний угол), можно видеть, что в данный момент анализируемый агент готов кооперироваться. Там же можно заметить вычисленную долю эпизодов, в которых намерения были определены правильно. На рисунке 3.4 доля составляет 0.88 от общего количества эпизодов, но на практике ее значение может колебаться. На самом игровом поле можно видеть путь,

пройденный анализируемым агентом, который, если учесть возможность перемещения сущности, является практически кратчайшим. Строка состояния показывает уверенность анализирующего агента в целенаправленности напарника. Зеленый цвет точки подтверждает правильность вывода агента.

Пример агента со случайным поведением приведен на рисунке 3.5. На этот раз точка в строке состояния смещена в левую сторону, что позволяет сделать вывод о неготовности напарника к кооперации. Следовательно, анализирующий агент направляется к синей точке для преждевременного завершения игры.

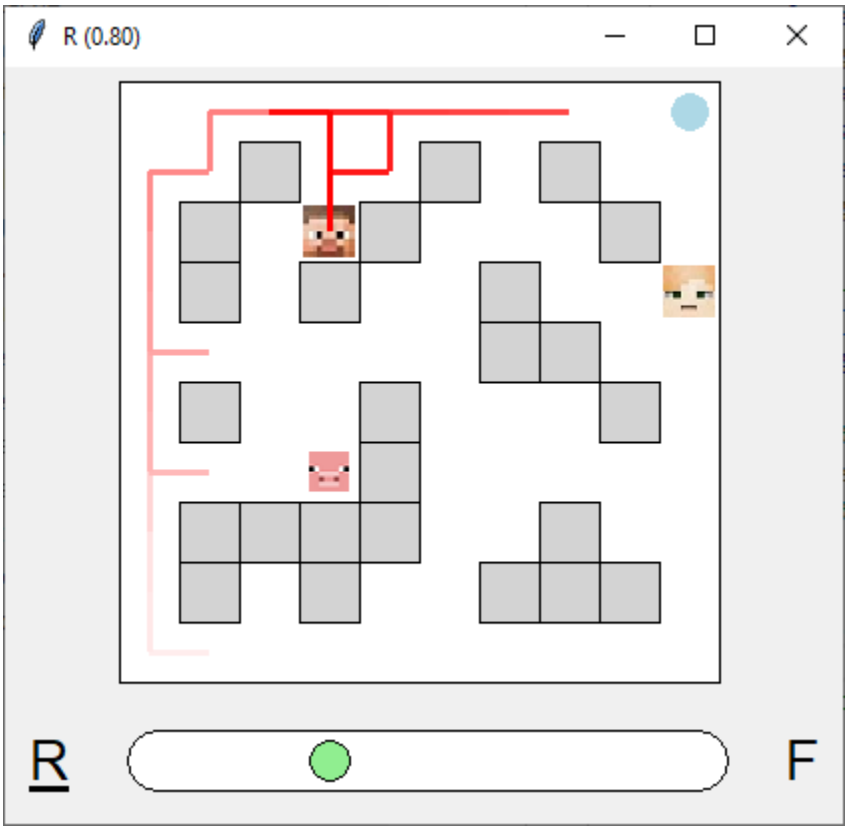


Рисунок 3.5 – ГИП, отображающий агента со случайным поведением

Получить долю эпизодов с правильным определением намерений в среде Malmo гораздо сложнее, поскольку на запуск 100000 эпизодов потребуется намного больше времени. Вместо этого можно привести два снимка экрана, отражающих поимку сущности и преждевременное завершение игры соответственно.

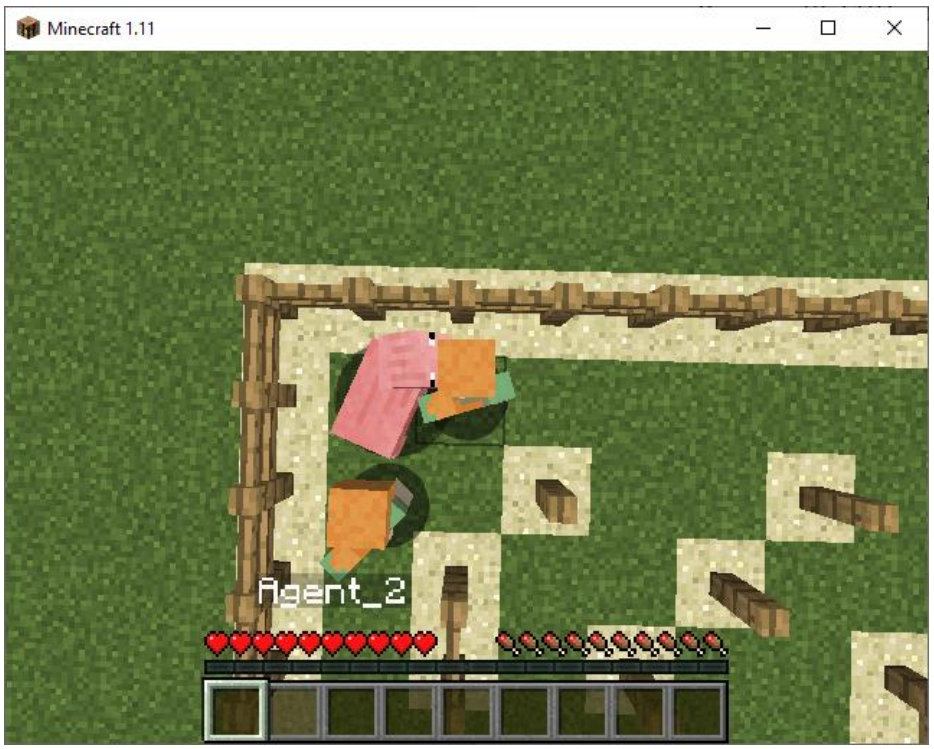


Рисунок 3.6 – Кооперация агентов в среде Malmo

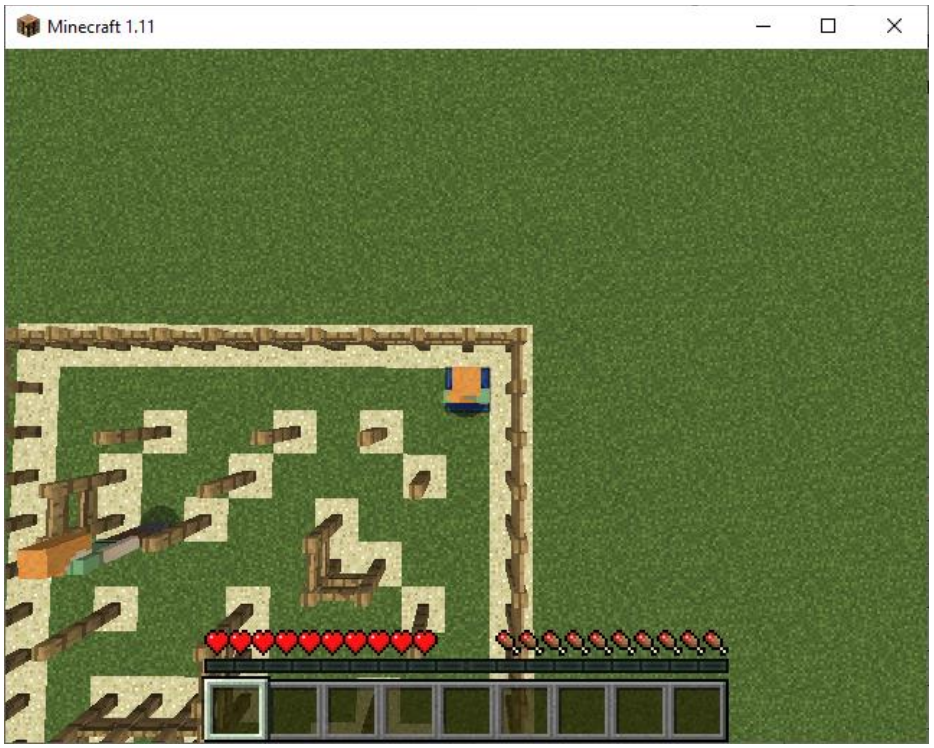


Рисунок 3.7 – Преждевременное завершение игры в среде Malmo

Проводить подобный анализ алгоритма навигации агента по уровню бессмысленно, поскольку, если не учитывать редкие попадания агента в циклы, Q-обучение предназначено для решения именно таких задач, и агент способен достичь цели в любой части уровня из любой заданной точки,

обходя все препятствия. Однако статистика расстояний до цели и успешного завершения игр для каждого шага агента приведена на рисунках 2.6 и 2.7 соответственно.

### 3.3. Оценка результатов

Показатель точности 0.88 на рисунке 3.4 достаточно высокий для модели инвариантной к динамике движения анализируемого агента. Однако даже при усреднении показателя по 100000 игр присутствует зависимость от качества первых эпизодов обучения модели. К примеру, на рисунке 3.5 эта же точность равна 0.8, хотя количество эпизодов, внесших свой вклад в формирование значения параметра, не изменилось.

Включение в состояние среды пути, пройденного анализируемым агентом, может положительно сказаться на точность модели. Путь до цели, проделанный агентом со случайным поведением на рисунке 3.5, не может быть назван кратчайшим. Он имеет множество самопересечений и сосредоточен в одной области уровня, не содержащей конечной цели, что сильно отличает его от пути целенаправленного агента на рисунке 3.4.

При отключенной обработке столкновений в среде Malmo агенты должны показывать аналогичные результаты, поскольку единственным отличием этой среды от упрощенной является поведение сущности.

Поведение целенаправленного агента можно назвать оптимальным. Агент может достигнуть любой клетки поля из любого положения за 18 шагов. При этом, анализируя рисунок 2.7, он успешно достигает цели, меняющей свое положение, за 8-9 шагов. Даже если этот путь не является кратчайшим, агент хорошо справляется со своей задачей, ведь данный алгоритм актуален для любого уровня, структура которого представляет собой разреженный лабиринт. Более того, если все ячейки Q-таблицы достигли оптимальных значений, одна и та же модель может управлять агентом на разных уровнях без необходимости переобучения.

На рисунке 2.6 наблюдается резкое увеличение расстояния от агента до цели после 20 шага. Однако исходя из рисунка 2.7, можно заметить, что большинство игр заканчиваются до 33 шага. Можно сделать вывод, что, вероятнее всего, агенты, не завершившие игру раньше 33 шага, входят в цикл и остаются в нем до конца эпизода. Применим анализ Фурье к значениям графика на рисунке 2.6, ограничившись диапазоном [245, 500]. Результат приведен на рисунке 3.8.

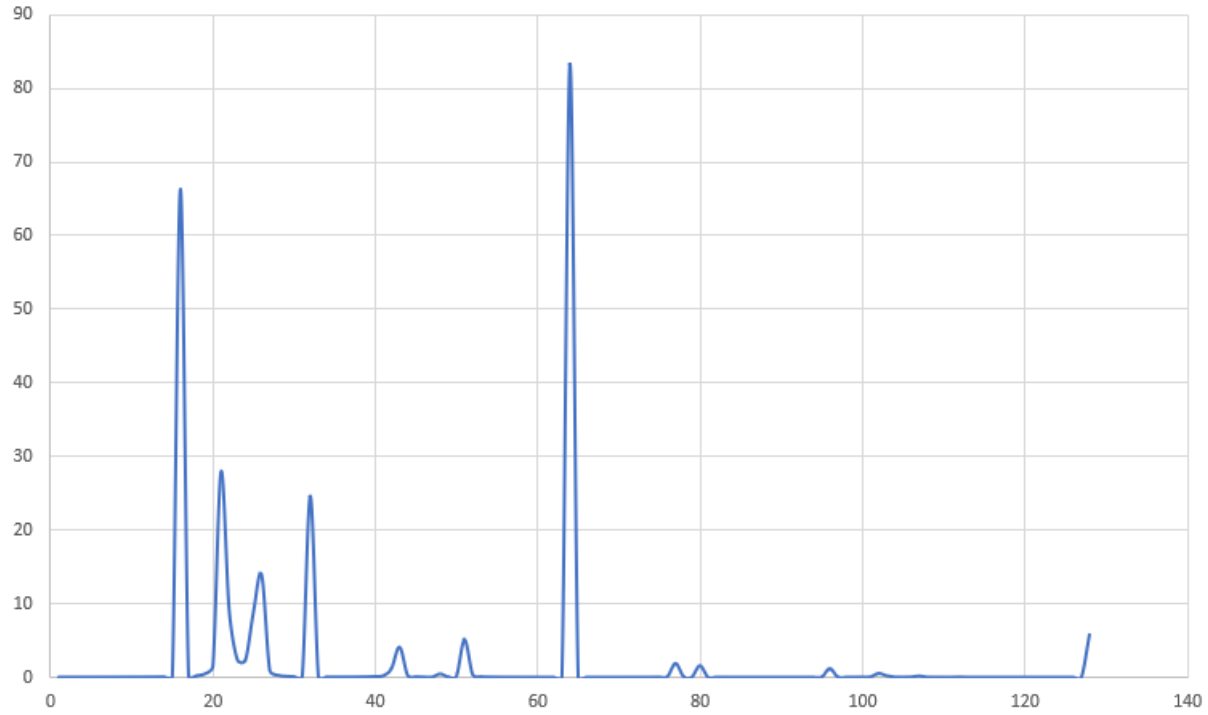


Рисунок 3.8 – Анализ Фурье для определения циклических движений

На графике явно выделяются несколько экстремумов, что увеличивает вероятность наличия циклических движений в поведении агента. Поскольку агентам, не попавшим в цикл или выбравшимся из него, редко требуется больше 33 шагов для успешного завершения игры, можно сделать вывод, что информация на графике в диапазоне [33, 500] предоставлена только агентами, попавшими в цикл. Это объясняет резкое увеличение расстояния на графике. Однако эта проблема не является критичной, ведь возникает она крайне редко, поскольку перемещение сущности с течением времени способно выводить агентов из циклов.

### 3.4. Вывод по разделу

По результатам тестирования модель навигации отлично справляется с решением поставленной задачи, в то время как модель бинарной классификации поведения, несмотря на точность в 90%, в некоторых ситуациях может неверно интерпретировать поведение целенаправленного агента, например, в случае, если агент выбрал не самый короткий путь к цели. В то же время модель анализа намерений полагается на модель навигации агентов во время обучения. Однако даже при обучении на основе алгоритма навигации в упрощённой среде, модель способна правильно классифицировать поведение различных агентов в среде игры Minecraft.

Таким образом, обе модели получились достаточно качественными для применения в системах с небольшой ответственностью, требующих коллаборации ИИ-агентов с минимальной коммуникацией между ними. Однако добавление в модель анализа динамики перемещения агентов во времени может повысить точность и допустить возможность встраивания модели в более важные системы.

# 4. ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ

## 4.1. Введение

В данной ВКР был разработан алгоритм коллаборации ИИ-агентов для достижения общей командной цели. Доля ИИ на рынке стремительно растет, а значит необходимо продолжение исследования различных подходов и алгоритмов для увеличения производительности и безопасности систем ИИ. Разработанный алгоритм поможет наладить коммуникацию между различными системами, в частности теми, основная задача которых связана с перемещением по поверхности (например, доставка грузов или исследование территории). Также алгоритм способен выявлять неполадки систем, с которыми требуется произвести коллаборацию, по их поведению, что позволяет работоспособным системам изменить план действий прямо во время его выполнения.

Данный раздел содержит расчет затрат, необходимых на реализацию продукта, заработный платы работников и другие виды необходимых отчислений. В конце раздела производится вывод о необходимости введения данного продукта на рынок.

## 4.2. Оценка трудоемкости

Трудоемкость вычисляется в человекоднях, затраченных на выполнение и оценку работы. ВКР была разделена на части, названия и примерные сроки выполнения которых приведены в таблице на странице 3. Оценка трудоемкости производится как для студента, так и для научного руководителя. Поскольку каждая часть имеет минимально и максимально допустимые сроки выполнения, для вычисления ее предположительной длительности рассчитывается взвешенное среднее от этих двух величин по формуле

$$t_j^0 = \frac{3t_{min} + 2t_{max}}{5}$$

Где $t_j^0$ – ожидаемая длительность работы, $t_{min}$ и $t_{max}$ – наименьшая и наибольшая допустимые длительности соответственно.

Таким образом, в таблице 4.1 приведены расчёты ожидаемой длительности для каждой части работы.

Таблица 4.1. – Ожидаемая длительность выполнения каждой части работы

| № п/п | Наименование работ | Длительность работы | | |
|---|---|---|---|---|
| | | $t_{min}$ | $t_{max}$ | $t_j^0$ |
| 1 | Обзор литературы по теме работы | 14 | 21 | 16,8 |
| 2 | Разработка программы | 62 | 72 | 66 |
| 3 | Тестирование программы | 2 | 7 | 4 |
| 4 | Проверка функциональности программы руководителем | 1 | 1 | 1 |
| 5 | Исправление ошибок и добавление функциональности при необходимости | 8 | 14 | 10,4 |
| 6 | Чистка программного кода | 2 | 5 | 3,2 |
| 7 | Оформление пояснительной записки | 10 | 20 | 14 |
| 8 | Оценка пояснительной записки руководителем | 2 | 7 | 4 |
| 9 | Оформление иллюстративного материала | 5 | 7 | 5,8 |

### 4.3. Определение затрат

В список затрат на изготовление продукта входят заработная плата работников, обязательные отчисления, амортизационные отчисления, эксплуатация оборудования, услуги сторонних организаций и расходные материалы.

#### 4.3.1 Определение затрат на заработную плату

Средняя месячная зарплата программиста-стажера равна 40000р до вычета налога, а оклад научного руководителя составляет 52270р/мес. Поскольку длительность работы в таблице 4.1 измеряется в днях, необходимо разделить данные значения на 21 рабочий день. Таким образом получается 1905р/день и 2489р/день соответственно.

Исходя из данных таблицы 4.1 вычисляются затраты на основную заработную плату по формуле:

$$З_{\text{осн.з\пл}} = \sum_{i=1}^{k} T_i * C_i = 120{,}2 * 1905 + 5 * 2489 = 241426 \text{ рублей}$$

Где $T_i$ – суммарная длительность работы i-того исполнителя в днях, $C_i$ – зарплата i-того исполнителя в р/день, k – количество исполнителей (в данном случае 2).

Расходы на дополнительную заработную плату вычисляются по формуле

$$З_{\text{доп.з/пл}} = З_{\text{осн.з\пл}} * \frac{H_{\text{доп}}}{100} = 241426 * 14\% = 33799{,}64 \text{ рубля}$$

Где $H_{\text{доп}}$ – норматив дополнительной заработной платы, принятый 14%.

### 4.3.2 Определение затрат на обязательные отчисления

Обязательные отчисления необходимы для обеспечения социального, пенсионного и медицинского страхования. Они вычисляются на основании основной и дополнительной заработных плат по формуле

$$З_{\text{соц}} = \left(З_{\text{осн.з\пл}} + З_{\text{доп.}\frac{\text{з}}{\text{пл}}}\right) * \frac{H_{\text{доп}}}{100} = 275225{,}64 * 30\% = 82567{,}692 \text{ рубля}$$

Где $H_{\text{доп}}$ – норматив отчислений на страховые взносы на обязательное социальное, пенсионное и медицинское страхование, принятый 30%, $З_{\text{соц}}$ – отчисления с заработной платы на социальные нужды.

### 4.3.3 Определение затрат на амортизационные отчисления

Прежде чем вычислить затраты на амортизационные отчисления, необходимо перечислить средства и программное обеспечение, используемые в данной ВКР.

Таблица 4.2. – Средства и ПО, использованные в данной ВКР

| Средства | Количество, шт. | Цена, руб. | Сумма, руб. |
|---|---|---|---|
| Ноутбук Lenovo | 1 | 62990.00 | 62990.00 |
| Принтер HP DeskJet | 1 | 6390.00 | 6390.00 |
| Windows 10 Home | 1 | 4937.00 | 4937.00 |
| Microsoft Office | 1 | 1254.00 | 1254.00 |
| Итого | | | 75571.00 |

Амортизационные отчисления вычисляются по следующей формуле:

$$А_i = Ц_{п.н.i} * \frac{H_{ai}}{100}$$

Где $Ц_{п.н.i}$ – первоначальная стоимость i-го средства, $H_{ai}$ – годовая норма амортизации i-того средства в процентах, рассчитывается по формуле

$$H_{ai} = \frac{100}{N_i}$$

Где $N_i$ – срок полезного использования i-го средства в годах. Для ноутбука Lenovo данный срок равен 5 годам, для струйного принтера HP DeskJet – 4 года. Для программного обеспечения понятие амортизационных отчислений неприменимо. Однако написание ВКР занимает меньше 2 лет, поэтому, необходимо рассчитать амортизационные отчисления для срока их реального использования по формуле

$$А_{iВКР} = А_i * \frac{T_{iВКР}}{12}$$

Где $T_{iВКР}$ – период использования средства для написания ВКР.

Таблица 4.3. – Амортизационные отчисления

| Средство | Первоначальная стоимость, руб. | Годовая норма амортизации, % | Амортизационные отчисления за 1 год, руб. | Амортизационные отчисления за время написания ВКР, руб. |
|---|---|---|---|---|
| Ноутбук Lenovo | 62990 | 20,0 | 12598 | 6299 |
| Принтер HP DeskJet | 6390 | 25,0 | 1598 | 133 |
| Итого | | | | 6432 |

### 4.3.4 Определение затрат на эксплуатацию оборудования

К категории эксплуатации оборудования можно отнести потребление электроэнергии ноутбуком и принтером, стоимость которого вычисляется по формуле

$$З_{э.э} = C * \sum_{i=1}^{k} P_i * t_i = 5{,}7 * (0{,}11 * 756 + 0{,}4 * 2) = 478{,}572 \text{ рублей}$$

Где k – количество устройств (в данном случае 2), $P_i$ – потребление электроэнергии i-тым устройством в кВт*ч, $t_i$ – суммарное время работы i-того устройства в часах, $C$ – тариф стоимости электроэнергии, равный в городе Санкт-Петербург 5,7руб/кВт*ч на 2023 год.

Принтер за все время работы в среднем активно использовался 2 часа, ноутбук – 756 часов (по 6 часов каждый рабочий день с ноября по май).

### 4.3.5 Затраты на услуги сторонних организаций

Для проведения исследования и написания программы необходим доступ в интернет, стоимость которого составляет 500р/мес как для программиста, так и для руководителя. Затраты на использование услуг сторонних организаций вычисляются по формуле

$$З_{\text{у.с.о.}} = \left(1 - \frac{\text{НДС}}{100}\right) * \sum_{i=1}^{k} q_i * t_i = 0{,}8 * 2 * (500 * 6) = 4800 \text{ рублей}$$

Где k – количество услуг (в данном случае 2), $q_i$ – стоимость i-той услуги, $t_i$ – период использования i-той услуги, НДС – налог на добавленную стоимость, равный 20% в 2023 году.

### 4.3.6 Затраты на расходные материалы

Затраты на расходные материалы рассчитываются по формуле

$$З_{\text{р.м}} = \left(1 + \frac{H_{\text{т.з}}}{100}\right) \sum_{i=1}^{k} G_i C_i$$

Где k – количество расходных материалов, $G_i$ – норма расхода i-того материала на единицу продукции в штуках, $C_i$ – стоимость i-того материала в руб., $H_{\text{т.з}}$ – норма транспортно-заготовительных расходов, принята 10%. Используемые расходные материалы приведены в таблице 4.4.

Таблица 2.4. – Расходные материалы и их стоимость

| Материал | $G_i$, шт. | Стоимость, руб. | Затраты, руб. |
|---|---|---|---|
| Бумага (500 листов) | 1 | 387 | 425,7 |
| Картридж HP 3YM60AE | 4 | 1142 | 5024,8 |
| Итого | | | 5450,5 |

### 4.4. Оценка стоимости проекта

Полная стоимость продукта складывается из всех затрат, вычисленных в пункте 4.3. Все затраты и итоговая сумма приведены в таблице 4.5.

Таблица 4.5. – Себестоимость разработки продукта

| Затраты | Сумма, руб. |
|---|---|
| Основная заработная плата | 241426,00 |
| Дополнительная заработная плата | 33799,64 |
| Обязательные отчисления | 82567,69 |
| Амортизационные отчисления | 6432,00 |
| Покупка спецоборудования | 62990,00 |
| Эксплуатация оборудования | 478,57 |
| Услуги сторонний организаций | 4800,00 |
| Расходные материалы | 5450,50 |
| Итого | 437944,40 |

Самой затратной составляющей разработки является основная заработная плата.

### 4.5. Вывод по разделу

Стоимость разработки системы коллаборации ИИ-агентов составляет 437944 рублей, что делает ее доступной практически для любого бизнеса, активно применяющего ИИ в своей работе. Данная система сможет забрать часть работы человека на себя, что позволит уменьшить нагрузку на людей, контролирующих состояние и поведение ИИ-агентов. Даже в случае сбоя части систем агенты просигнализируют о происшествии и попытаются продолжить работу самостоятельно, сгенерировав новый план действий.

# ЗАКЛЮЧЕНИЕ

В данной ВКР в соответствии с ее целью была разработана программа для распознавания стратегий поведения ИИ-агентов и создан механизм их навигации по плоскости для достижения общей цели в мультиагентной среде. Цель была успешно достигнута – разработанный агент способен ориентироваться на плоскости с препятствиями, представляющими собой разреженный лабиринт, и определять намерения своего напарника с точностью до 90%. Выполнение данной работы показало высокую сложность изготовления систем коллаборации ИИ-агентов даже для решения такой простой задачи. Это объясняет малую на данный момент распространенность подобных систем для бизнес-приложений.

Системы координации двух и более ИИ-агентов достаточно широко используются в настоящем и еще более успешное применение им предсказывают в будущем, особенно в областях с разветвленной структурой автономных сущностей, большим количеством данных, высокой динамикой и плохой предсказуемостью поведения. Примерами могут служить управление большим количеством локальных структур в составе крупной организации, самоорганизация группировок беспилотных автомобилей или летательных аппаратов, либо синхронизированная работа малых электростанций в распределенных электросетях. Например, агенты могут играть роль доставщиков материалов, которым необходимо встретиться в заданной точке, имея ее координаты. Один агент передаст материалы другому, чтобы второй мог продолжить их доставку. Если один из агентов вышел из строя, второй сможет проанализировать поведение первого и подать сигнал о сбое в системе, не доходя до назначенного места встречи. Другой пример: оба агента являются летательными дронами, исследующими территорию на наличие электромагнитных пушек или других опасностей. У каждого из агентов имеется точка на карте, в которой они должны оказаться по окончанию исследования. Если один из агентов наблюдает странности в

поведении своего напарника, такие как отклонение от курса или полную остановку, он прерывает процедуру исследования и возвращается на базу.

Для дальнейшего продвижения данной разработки необходимо рассмотреть модели, более детально анализирующие поведение агента. Необходимо учитывать «запутанность» их пути: количество самопересечений, кривизна с учетом препятствий и т.д. Также стоит уделить внимание временному анализу – пронаблюдать изменения в поведении агента за все время игры. Важную роль может сыграть возможность изменения цели любого агента прямо в процессе выполнения плана. На данный момент изменить свою цель в течении одного и того же эпизода может только агент, выполняющий анализ поведения. Это является довольно сильным ограничением текущей работы.

Скорее всего, в дальнейшем мультиагентные системы распространяться на всевозможные типы устройств (совокупность умных Интернет вещей), имеющих датчики и способных воздействовать на различные объекты внешнего мира. Такие устройства будут способны не просто обрабатывать сигналы, следуя определенным инструкциям, а самостоятельно принимать решения и строить планы действий.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. John McCarthy. What is artificial intelligence? // Stanford, CA 94305. – 2007. – 12 ноября [Электронный ресурс]. URL: https://www-formal.stanford.edu/jmc/whatisai.pdf (дата обращения: 09.05.2023).
2. Mark Stefik. Roots and Requirements for Collaborative AI // Palo Alto Research Center. – 2023. – 17 марта [Электронный ресурс]. URL: https://arxiv.org/ftp/arxiv/papers/2303/2303.12040.pdf (дата обращения: 09.05.2023).
3. Jason Lee. Teaching a computer how to play Snake with Q-Learning // Towards Data Science. – 2020. – 23 июля [Электронный ресурс]. URL: https://towardsdatascience.com/teaching-a-computer-how-to-play-snake-with-q-learning-93d0a316ddc0 (дата обращения: 09.05.2023).
4. Alex Maszański. Что такое обучение с подкреплением и как оно работает. Объясняем на простых примерах // proglib. – 2021. – 24 декабря [Электронный ресурс]. URL: https://proglib.io/p/chto-takoe-obuchenie-s-podkrepleniem-i-kak-ono-rabotaet-obyasnyaem-na-prostyh-primerah (дата обращения: 09.05.2023).
5. Adrià Garriga-Alonso. The Malmo Collaborative AI Challenge // Github. – 2017. – 23 мая [Электронный ресурс]. URL: https://github.com/rhaps0dy/malmo-challenge (дата обращения: 09.05.2023).
6. Florin Gogianu. Malmo Challenge Overview // Github. – 2017. 23 июля [Электронный ресурс]. URL: https://github.com/village-people/flying-pig (дата обращения: 09.05.2023).
7. James Allen, George Ferguson. Human-machine collaborative planning // University of Rochester. – 2020. – 5 июня [Электронный ресурс]. URL: https://www.researchgate.net/publication/228911168_Human-machine_collaborative_planning (дата обращения: 09.05.2023).

8. Funtowicz Morgan, Vitaly Kurin, Katja Hofmann. Malmo Collaborative AI Challenge – Pig Chase // Github. – 2017. – 11 мая [Электронный ресурс]. URL: https://github.com/microsoft/malmo-challenge/blob/master/ai_challenge/pig_chase/README.md (дата обращения: 09.05.2023).
9. Johnson M., Hofmann K., Hutton T., Bignell D. The Malmo Platform for Artificial Intelligence Experimentation. // Proc. 25th International Joint Conference on Artificial Intelligence, Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA. – 2016 [Электронный ресурс]. URL: https://github.com/Microsoft/malmo (дата обращения 09.05.2023).
10. Gary Klein, David D. Woods, Jeffrey M. Bradshaw, Robert R. Hoffman, Paul J. Feltovich. Ten Challenges for Making Automation a “Team Player” in Joint Human-Agent Activity // Institute for Human and Machine Cognition. – 2004 [Электронный ресурс]. URL: https://www.ihmc.us/wp-content/uploads/2021/04/17.-Team-Players.pdf (дата обращения: 09.05.2023).
11. Интеллектуальный агент // Wikipedia. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Интеллектуальный_агент (дата обращения: 09.05.2023).
12. Темпоральная логика // Wikipedia. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Темпоральная_логика (дата обращения: 09.05.2023).
13. Q-learning // Wikipedia. [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Q-learning (дата обращения: 09.05.2023).
14. Глубокое обучение // Wikipedia. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Глубокое_обучение (дата обращения: 09.05.2023).
15. Tkinter // Wikipedia. [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Tkinter (дата обращения: 09.05.2023).

# ПРИЛОЖЕНИЕ А. ЛИСТИНГИ ИСХОДНОГО КОДА ПРОГРАММЫ

Листинг А.1 – Исходный код файла main.py

```
# Импорт агентов из файла agents.py (Листинг А.2)
from agents import Focused, Random, Wise
from tkinter import *
# Импорт класса Design из файла design.py (Листинг А.4)
from design import *
import random
import time

# Установка количества агентов в системе (1 или 2)
NUM_OF_AGENTS = 2
# Макимально допустимое количество шагов для каждого из агентов
MAX_FITNESS = 200
# Структура препятствий разреженного лабиринта, в котором
существует путь между двумя любыми клетками
obstacle_pos = \
[{'x': 2, 'y': 1}, {'x': 5, 'y': 1}, {'x': 7, 'y': 1}, {'x': 1,
'y': 2}, {'x': 4, 'y': 2}, {'x': 8, 'y': 2}, {'x': 1, 'y': 3},
{'x': 3, 'y': 3}, {'x': 6, 'y': 3},\
{'x': 6, 'y': 4}, {'x': 7, 'y': 4}, {'x': 1, 'y': 5}, {'x': 4,
'y': 5}, {'x': 8, 'y': 5}, {'x': 4, 'y': 6}, {'x': 1, 'y':
7},{'x': 2, 'y': 7},{'x': 3, 'y': 7},{'x': 4, 'y': 7},\
{'x': 7, 'y': 7}, {'x': 3, 'y': 8}, {'x': 6, 'y': 8}, {'x': 7,
'y': 8}, {'x': 8, 'y': 8}, {'x': 1, 'y': 8}]
# Точка преждевременного завершения игры
giveup_pos = {'x' : 9, 'y' : 0}

# Инициализация графического интерфейса пользователя
design = Design(NUM_OF_AGENTS, obstacle_pos, giveup_pos)

# Случайный выбор свободной клетки на поле
def generate_object():
    global agents_brain
    # Получение положений всех агентов
    agents_pos = [agent.get_pos() for agent in agents_brain]
    obj_pos = {}
    while True:
        # Случайный выбор клетки по двум координатам
        obj_pos['x'] = int(9*random.random())
        obj_pos['y'] = int(9*random.random())
        # Если клетка не занята агентами или препятствиями
        if obj_pos not in agents_pos and obj_pos not in
obstacle_pos:# and edibility() >= 4-NUM_OF_AGENTS:
            return obj_pos

# Инициализация целенаправленного, мудрого (анализирующего)
агентов и агента со случайным поведением
# Поскольку NUM_OF_AGENTS принимает значение 2, система
учитывает только двух первых агентов
# При запуске нового эпизода первое место занимает либо первый,
либо третий агент из списка
```

```
agents_brain = []
agents_brain.append(Focused(generate_object(), MAX_FITNESS))
agents_brain.append(Wise(generate_object(), MAX_FITNESS))
agents_brain.append(Random(generate_object(), MAX_FITNESS))
# Акцивация отслеживания пути анализируемого агента
design.track_agent(0)
# Генерация начального положения цели
target_pos = generate_object()
# Отрисовка цели
design.move_pig(target_pos)

# Проверка на окруженность цели препятствиями
def edible():
    global target_pos
    obstacle_count =\
        ({'x' : target_pos['x']-1, 'y' : target_pos['y']} in
obstacle_pos or target_pos['x'] == 0) +\
        ({'x' : target_pos['x']+1, 'y' : target_pos['y']} in
obstacle_pos or target_pos['x'] == 9) +\
        ({'x' : target_pos['x'], 'y' : target_pos['y']-1} in
obstacle_pos or target_pos['y'] == 0) +\
        ({'x' : target_pos['x'], 'y' : target_pos['y']+1} in
obstacle_pos or target_pos['y'] == 9)
    # Если цель окружена минимум 2 препятствиями, ее возможно
поймать
    return obstacle_count > 1

caught_per_agent = [False] * NUM_OF_AGENTS
# Начало новой игры, когда цель достигнута
def catch(agent):
    global target_pos, caught_per_agent, fitness, steps,
target_brain, target_move_timer
    # Агенты должны находиться в разных клетках для поимки
сущности
    if NUM_OF_AGENTS > 1 and agents_brain[agent].get_pos() ==
agents_brain[1-agent].get_pos():
        return
    caught_per_agent[agent] = True
    # Если каждый агент поймал сущность (сущность окружена),
эпизод перезапускается
    if not all(caught_per_agent):
        return
    # Случайный выбор агента для новой игры (целенаправленный
или со случайным поведением)
    agent_id = int(2*random.random())
    agents_brain[0], agents_brain[3] = agents_brain[2*agent_id],
agents_brain[2*(1-agent_id)]
    # Временная заморозка цели на начало игры (на первые 5 шагов
агентов)
    target_move_timer = 5
    # Случайный выбор клеток для расположения агентов в новой
игре
    for a in range(NUM_OF_AGENTS):
```

```
        agents_brain[a].position = generate_object()
    # Обновление положения цели в классе, отвечающем за ее
перемещение
    target_brain.position = target_pos = generate_object()
    # Сброс игрового прогресса
    fitness = 0
    caught_per_agent = [False] * NUM_OF_AGENTS
    # Отрисовка цели в новом положении
    design.move_pig(target_pos)
    # Очистка пути, пройденного анализируемым агентом
    design.clear_paths()

# Перезапуск игры
def restart_game():
    global caught_total, caught_per_agent
    # Сброс игрового прогресса
    caught_total = -1
    caught_per_agent = [False] * NUM_OF_AGENTS
    for agent in range(NUM_OF_AGENTS):
        agents_brain[agent].reset()
        # Начало новой игры для каждого агента
        catch(agent)

# Начало основного цикла программы
steps = 0 # Суммарное количество шагов, сделанных каждым
агентом, по всем эпизодам
fitness = 0 # Количество шагов, сделанных каждым агентом за
текущий эпизод
game_loses = 1 # Количество эпизодов, в которых агент неверно
определил намерения напарника
game_wins = 1 # Количество эпизодов, в которых агент верно
определил намерения напарника
target_brain = Random(target_pos, MAX_FITNESS) # Агент,
управляющий перемещением сущности
target_move_timer = 5 # Таймер, разрешающий/запрещающий сущности
передвигаться
while True:
    # Перемещение сущности-цели случайным образом
    if target_move_timer < 0:
        target_res = target_brain.move(target_pos, obstacle_pos,
fitness=0)
        target_pos = target_brain.get_pos()
        # Если цель переместилась, она больше не является
пойманной
        for agent in range(NUM_OF_AGENTS):
            caught_per_agent[agent] = False
        design.move_pig(target_pos)

    # Обновление таймера, разрешающего/запрещающего сущности
передвигаться
    if target_move_timer == 0:
        # Максимум 5 шагов перемещния и 10 - отдыха
        target_move_timer = int(15 * random.random() - 5)
```

```
    else:
        target_move_timer -=
target_brain.sign(target_move_timer)

    # Обновление положения агентов
    for agent in range(NUM_OF_AGENTS):
        # Если агент умеет анализировать поведение, выполняется
анализ
        if hasattr(agents_brain[agent], "evaluate_strategy"):
            # Получение актуальных данных о поведении напарника
в виде вероятности
            avg_guess =
agents_brain[agent].evaluate_strategy(agents_brain[0],
target_pos, fitness)
            binary_guess = round(avg_guess)

            # Выбор стратегии анализирующего агента в
соответствии с актуальными данными
            target = [giveup_pos, target_pos][binary_guess]
            agents_brain[agent].set_edibility([True,
edible()][binary_guess])

            # Если напарник предположительно не желает
кооперироваться, забываем про сущность
            if binary_guess == 0:
                caught_per_agent[agent] = False
        # Если нет, выставляются параметры по умолчанию
        else:
            target = target_pos # цель - сущность
            agents_brain[agent].set_edibility(edible())

        # Передача управления текущему агенту
        result = agents_brain[agent].move(target, obstacle_pos,
fitness)

        # Отрисовка агента в новом положении
        design.move_agent(agent, agents_brain[agent].get_pos())

        # Проверка на глобальные изменения в игре
        if result == 1 and target == target_pos: # Текущий агент
достиг цели
            catch(agent)
        # Цели была достигнута всеми агентами, или агент
столкнулся с препятствием
        if result < 0 or all(caught_per_agent):
            # Обновление счетчика игр в соответствии с правилами
            game_wins += agents_brain[0].label != "R"
            game_loses += agents_brain[0].label == "R"
            restart_game()
        if agents_brain[agent].get_pos() == giveup_pos: #
Текущий агент сдался
            # Обновление счетчика игр в соответствии с правилами
            game_wins += agents_brain[0].label == "R"
```

```
            game_loses += agents_brain[0].label != "R"
            restart_game()

        # Обновление информации о достижении цели в классе
агента

agents_brain[agent].set_goal_state(caught_per_agent[agent])
    # Обновление графической информации в окне приложения
    design.update(avg_guess, agents_brain[0].label,
agents_brain[0].label + " (" + '{0:.2f}'.format(game_wins /
(game_wins + game_loses)) + ")")

    # Если цель достижима в текущем состоянии среды
    currently_edible = edible()

    # После 199990 шага агента начинается отрисовка графического
интерфейса пользователя
    if steps == 199990:
        design.toggle_drawing_mode()
    # Каждые 10000 шагов выводится статус обучения модели
    if steps % 10000 == 0 and currently_edible:
        print("Wise agent made", steps, "steps")
    # Процессы среды необходимо замедлить, чтобы пользователь
мог следить за прогрессом
    if 199990 < steps:
        time.sleep(0.1)

    # Если агенты не двигаются, счетчик шагов не меняется
    if currently_edible:
        steps += 1
        fitness += 1
```

Листинг А.2 – Исходный код файла agents.py

```
# Импорт класса Strategy из файла strategy.py (Листинг А.4)
from Strategy import Strategy, get_L1distance_from_target
from random import random
import json

# Базовый класс перемещающегося объекта
class Observer:
    label = "O" # Метка класса
    # Список наград, предоставляемых средой
    rewards = {'step' : -1, 'step_back' : -200, 'loss' : -1000,
'caught' : 50, 'stay' : -10}

    # Конструктор принимает начальную позицию объекта и
максимально допустимое количество шагов за игру
    def __init__(self, pos, max_fitness) -> None:
        self.position = pos
        self.max_fitness = max_fitness
        self.last_move = 0 # Предыдущее действие, соверешенное
агентом
```

```
        self.goal_achieved = False # Достигнута ли цель (агент
находится на соседней клетке от сущности)
        self.edible = False # Достижима ли цель (возможно ли
поймать сущность)

    # Геттер положения агента
    def get_pos(self):
        return self.position

    # Получение размера награждения за сооветствующее действие
    def get_reward(self, action):
        return self.rewards[action]

    # Получение знака числа
    def sign(self, value):
        if value < 0: return -1
        elif value > 0: return +1
        else: return 0

    # Формирование строки состояния системы из положений цели и
препятствийы
    def generate_state(self, target, obstacles):
        # Горизонтальное (-1,0,1) и вертикальное (-1,0,1)
положения цели относительно агента
        target_related = 3*self.sign(target['y']-
self.position['y']) + self.sign(target['x']-self.position['x'])
        # Наличие препятствия слева от агента
        left_side = int({'x' : self.position['x']-1, 'y' :
self.position['y']} in obstacles)
        # Наличие препятствия снизу от агента
        bottom_side = int({'x' : self.position['x'], 'y' :
self.position['y']+1} in obstacles)
        # Наличие препятствия справа от агента
        right_side = int({'x' : self.position['x']+1, 'y' :
self.position['y']} in obstacles)
        # Наличие препятствия сверху от агента
        upper_side = int({'x' : self.position['x'], 'y' :
self.position['y']-1} in obstacles)
        # Комбинация наличия препятствий со всех сторон
        obstacle_comb = (left_side << 3) + (bottom_side << 2) +
(right_side << 1) + upper_side
        # Комбинация всех состояний в одну строку
        return str(4 * (15*target_related + obstacle_comb) +
self.last_move)

    # Установка достижимости цели извне класса
    def set_edibility(self, edible):
        self.edible = edible

    # Совершения действия агентом и получение награды
    def move(self, target, obstacles, fitness):
        # Текущее состояние среды
        state = self.generate_state(target, obstacles)
```

```
        # Получение наилучшего или случайного движения агента
        move = self.get_best_move(state)

        # Результат действия агента (не награда), нейтральный по
умолчанию
        result = 0

        # Получение новых координат агента на плоскости
        tempX = self.position['x'] + (move == 1) -(move == 3)
        tempY = self.position['y'] + (move == 2) -(move == 0)
        # Обновление положения агента
        self.position = {'x' : tempX, 'y' : tempY}
        # Если агент вышел за пределы уровня, столкнулся с
препятствием или
        # превысил максимально допустимое колчичество шагов за
один эпизод
        if not 0 <= tempX < 10 or not 0 <= tempY < 10 or
self.position in obstacles or fitness > self.max_fitness:
            result = -1
            reward = self.get_reward('loss')
        # Если агент достиг цели (находит в той клетке или
смежной с ней)
        elif abs(tempX-target['x']) + abs(tempY-target['y']) <=
1:
            result = 1 # Goal has been achieved
            reward = self.get_reward('caught')
        # Обработка перемещения агента на предыдущую позицию
        elif self.last_move == 0 and move == 2:
            reward = self.get_reward('step_back')
        elif self.last_move == 1 and move == 3:
            reward = self.get_reward('step_back')
        elif self.last_move == 2 and move == 0:
            reward = self.get_reward('step_back')
        elif self.last_move == 3 and move == 1:
            reward = self.get_reward('step_back')
        else:
            reward = self.get_reward('step')

        # Получение нового состония системы
        newState = self.generate_state(target, obstacles)
        # Обучение модели на основании последнего шага и награды
        self.train(state, move, reward, newState)
        # Обновление последнего шага агента
        self.last_move = move
        # -1 - агент нарушил правила, 0 - состояние не
изменилось, 1 - агент достиг цели
        return result

    # Установка состояния достижения цели извне класса
    def set_goal_state(self, achieved):
        self.goal_achieved = achieved

    # Сброс игрового прогресса
```

```
    def reset(self):
        self.goal_achieved = False

# Класс целенаправленного агента
class Focused(Observer):
    label = "F" # Метка класса

    def __init__(self, pos, max_fitness) -> None:
        super().__init__(pos, max_fitness)
        # Параметры для Q-обучения
        self.learn_rate = 0.1
        self.discount = 0.9
        self.randomness = 0.0 # 0.9 для обучения
        self.randomness_decay = 1.3
        self.randomness_min = 0.00005

        # Загрузка готовой Q-таблицы для целенаправленого агента
        self.load_Q()
        # Для обучения модели навигации необходимо
раскомментировать код и
        # запустить программу с одним агентом
        # Нельзя запускать параллельно с обучением модели
определения намерений,
        # это не даст желаемого результата
        #self.Q = {} # Filling self.Q with all zeroes
        #for y in range(-1,2): # Lower, Upper, Equal
        #    for x in range(-1,2): # Left, Right, Equal
        #        for o in range(15): # Combinations of 4
obstacles around agent
        #            for prev in range(4): # Previous move
        #                shift = 4 * (15 * (3*y + x) + o) + prev
        #                self.Q[str(shift)] = {}
        #                for a in range(4): # Up, Right, Down,
Left
        #                    self.Q[str(shift)][str(a)] = 0

    def move(self, target, obstacles, fitness):
        # Если цель недостижима или уже достигнута,
целенаправленному агенту не следует двигаться
        if self.goal_achieved == True or not self.edible:
            return 0
        # Вызов родительской функции перемещения агента
        return super().move(target, obstacles, fitness)

    # Получить наилучшее действие в соответствии с Q-таблицей
или выбрать случайное для исследования среды
    def get_best_move(self, state):
        return int(max(self.Q[state], key=self.Q[state].get)) if
random() > self.randomness else int(4*random())

    # Функция тренировки модели обновляет значение в ячейке Q-
таблицы на пересечении state и action
    def train(self, state, action, reward, newState):
```

```
        # Значение не обновляется, если действие привело к
завершению игры
        self.Q[state][str(action)] *= 1 - self.learn_rate
        self.Q[state][str(action)] += self.learn_rate *\
            (reward + self.discount *
max(self.Q[newState].values()) * int(reward !=
self.rewards['loss']))

    # Уменьшение случайности в поведении агента при обучении
модели
    def reset(self):
        super().reset()
        if self.randomness > self.randomness_min:
            self.randomness /= self.randomness_decay

    # Сохранить Q-таблицу в файл
    def save_Q(self):
        open("Agent.json", "w").write(json.dumps(self.Q))

    # Загрузить Q-таблицу из файла
    def load_Q(self):
        self.Q = json.load(open("Agent.json", "r"))

# Класс "мудрого" агента, выполняющего анализ поведения объекта
Observer
class Wise(Focused):
    label = "W" # Метка класса

    def __init__(self, pos, max_fitness) -> None:
        super().__init__(pos, max_fitness)
        # Перед началом новой игры напарник по умолчанию
считается целенаправленным
        self.avg_guess = 1
        # Инициализация объекта Strategy для определения
намерений
        self.strategy = Strategy()

    # Функция оценки поведения напарника
    def evaluate_strategy(self, other_agent, target_pos,
fitness):
        # Получение расстояния от агента до цели в метрике L1
        dist_to_target =
get_L1distance_from_target(other_agent.get_pos(), target_pos)
        # Задержка для более точного определения
        if fitness > 3:
            # Получение текущего состояния напарника
            strat_state =
self.strategy.get_state(dist_to_target, fitness)
            # Оценка его поведения
            strat_guess =
self.strategy.get_best_guess(strat_state)
            # Награда, размер которой зависит от правильности
определения поведения
```

```
            strat_reward = 1 if ["R", "F"][strat_guess] ==
other_agent.label else 0
            # Обучение модели на основании состояния напарника и
награды
            self.strategy.train(strat_state, strat_guess,
strat_reward, strat_state if strat_guess == 1 else
str(int(36*random())))
            # Обновление вероятности целенаправленности
напарника
            self.avg_guess = 0.80 * self.avg_guess + 0.20 *
strat_guess
        # Если напарник предположительно имеет случайное
поведение, сброс состояния достижения цели
        if round(self.avg_guess) == 0:
            self.goal_achieved = False
        # Возврат нового значения вероятности
        return self.avg_guess

    # Уменьшение случайности тренировки модели определения
намерений
    def reset(self):
        super().reset()
        self.strategy.update_randomness()
        self.avg_guess = 1

# Класс агента со случайным поведением
class Random(Observer):
    label = "R" # Метка класса

    def __init__(self, pos, max_fitness) -> None:
        super().__init__(pos, max_fitness)

    # Выбор наилучшего действия, не приводящего к немедленному
завершению игры
    def get_best_move(self, state):
        # Анализ наличия препятствий вокруг агента
        obstacle_info = int(state) // 4 % 15
        # Обрезка недопустимых действий
        actions = ('3' if not (obstacle_info & 8) and
self.position['x'] != 0 else '') +\
            ('2' if not (obstacle_info & 4) and
self.position['y'] != 9 else '') +\
            ('1' if not (obstacle_info & 2) and
self.position['x'] != 9 else '') +\
            ('0' if not (obstacle_info & 1) and
self.position['y'] != 0 else '')
        return int(actions[int(len(actions)*random())])

    # Функция тренировки модели отсутствует
    def train(self, state, action, reward, newState):
        pass

    # Функция сброса прогресса отсутствует
```

```
    def reset(self):
        pass
```

Листинг А.3 – Исходный код класса Strategy (файл strategy.py)

```
from random import random
import json

# Класс отвечает за анализ поведения агента
class Strategy:
    def __init__(self) -> None:
        # Параметры для Q-обучения
        self.learn_rate = 0.1
        self.discount = 0.9
        self.randomness = 0.9
        self.randomness_decay = 1.0005
        self.randomness_min = 0.00005

        # Инициализация Q-таблицы нулями
        # Можно использовать self.load_Q() для загрузки готовой
таблицы
        self.Q = {}
        # Минимальное расстояние до цели равно (0,0) и
максимальное (9,9), поэтому L1 расстояние находится в пределах
[0, 18]
        for dist in range(20):
            # 3 категории: первый шаг агента, первые 15 шагов и
остальные
            for stage in range(3):
                shift = 3 * dist + stage
                self.Q[str(shift)] = {}
                # Два варианта "ответа": агент целенаправленный
или со случайным поведением
                for strategy in range(2):
                    self.Q[str(shift)][str(strategy)] = 0

    # Выбрать наиболее подходящий класс поведения или случайный
при обучении модели
    def get_best_guess(self, state):
        return int(max(self.Q[state], key=self.Q[state].get)) if
random() > self.randomness else int(2*random())

    # Функция тренировки модели обновляет значение в ячейке Q-
таблицы на пересечении state и guess (action)
    def train(self, state, guess, reward, newState):
        self.Q[state][str(guess)] *= 1 - self.learn_rate
        self.Q[state][str(guess)] += self.learn_rate * (reward +
self.discount * max(self.Q[newState].values()))

    # Уменьшение случайности в предсказании намерений агента
    def update_randomness(self):
        if self.randomness > self.randomness_min:
            self.randomness /= self.randomness_decay
```

```
    # Получить текущее состояние агента в системе на основании
расстояния от него до цели и
    # количества шагов с начала игры
    def get_state(self, relative_distance, steps_count):
        # Разделение количества сделанных шагов на категории
        step_group = 0 if steps_count <= 1 else 1 if steps_count
<= 15 else 2
        # Формирование строки состояния
        state = 3 * relative_distance + step_group
        return str(state)

    # Сохранить Q-таблицу в файл
    def save_Q(self):
        open("Strategy.json", "w").write(json.dumps(self.Q))

    # Загрузить Q-таблицу из файла
    def load_Q(self):
        self.Q = json.load(open("Strategy.json", "r"))

# Получение расстояния метрики L1 между двумя точками на
плоскости
def get_L1distance_from_target(agent_pos, target_pos):
    return abs(agent_pos['x']-target_pos['x']) +
abs(agent_pos['y']-target_pos['y'])
```

Листинг А.4 – Исходный код класса Design (файл design.py)

```
from tkinter import *
from tkinter import ttk

# Класс отвечает за графический интерфейс пользователя
class Design:
    # Элементы пути, пройденного отслеживаемым агентом
    agent_paths = []
    # Идентификаторы отслеживаемых агентов
    tracking_agents = []

    # Конструктор принимает количество агентов в системе,
положения препятствий и
    # точки преждевременного завершения игры на плоскости
    def __init__(self, agent_count, obstacles_pos, giveup_pos) -
> None:
        # Настройка графического окна приложения
        self.master = Tk()
        self.w = Canvas(self.master, width=310, height=310,
scrollregion=(-1,-5, 5, 5))
        self.w.grid(column=1, row=0, padx=10, pady=0)
        # Параметр обновления содержания графического окна
(False в режиме обучения)
        self.enable_drawing = False

        # Создание фона для уровня
```

```
        self.w.create_rectangle(0, 0, 300, 300, fill='white')

        # Создание графического представления агентов
        self.agents_rect = []
        self.agent_photos = []
        self.agent_last_pos = []
        for agent_id in range(agent_count):
            # Изображения агентов находятся в файлах player№.png
            self.agent_photos.append(PhotoImage(file =
f"player{agent_id}.png"))
            self.agents_rect.append(self.w.create_image(2, 2,
image=self.agent_photos[-1]))
            # Сохранение предыдущего положения для отрисовки
путей
            self.agent_last_pos.append(None)
        # Создание графического представления препятствий
        for obstacle in obstacles_pos:
            # Препятствия обозначаются серым цветом
            self.w.create_rectangle(30*obstacle['x'],
30*obstacle['y'], 30*(obstacle['x']+1), 30*(obstacle['y']+1),
fill='lightgray')
        # Создание графического представления цели (текстура в
файле pig.png)
        self.pig_photo = PhotoImage(file = 'pig.png')
        self.target_rect = self.w.create_image(0, 0,
image=self.pig_photo)
        # Создание графического представления точки
преждевременного завершения игры
        # Отмечается синим кругом
        self.w.create_oval(30*giveup_pos['x']+5,
30*giveup_pos['y']+5, 30*(giveup_pos['x']+1)-5,
30*(giveup_pos['y']+1)-5, fill='lightblue', outline="white")

        # Добавление строки состояния уверенности в
целенаправленности напарника
        self.strategy_canvas = Canvas(self.master, width=310,
height=40, scrollregion=(-5,-5, 5, 5))
        self.strategy_canvas.grid(column=1, row=1, padx=0,
pady=10)
        # Создание графического представления строки состояния
        self.strategy_canvas.create_oval(0, 0, 30, 30,
fill="white")
        self.strategy_canvas.create_oval(270, 0, 300, 30,
fill="white")
        self.strategy_canvas.create_rectangle(15, 0, 285, 30,
fill="white", outline="white")
        self.strategy_canvas.create_line(15, 0, 286, 0)
        self.strategy_canvas.create_line(15, 30, 286, 30)
        self.strategy_bar = self.strategy_canvas.create_oval(5,
5, 25, 25, fill="lightgreen")
        # Добавление подсказок по сторонам строки состояния
        self.r_label = ttk.Label(self.master, text="R",
font=("Arial", 20))
```

```
        self.r_label.grid(column=0, row=1, padx=10, pady=10)
        self.f_label = ttk.Label(self.master, text="F",
font=("Arial", 20))
        self.f_label.grid(column=2, row=1, padx=10, pady=10)

    # Отрисовка сущности в новом положении
    def move_pig(self, target_pos):
        # Проверка на необходимость обновления графического окна
        if not self.enable_drawing:
            return
        self.w.moveto(self.target_rect, 30*target_pos['x']+5,
30*target_pos['y']+5)

    # Отрисовка агента в новом положении
    def move_agent(self, agent_id, agent_pos):
        # Проверка на необходимость обновления графического окна
        if not self.enable_drawing:
            return
        # Отрисовка пути, пройденного агентом
        self.draw_path(agent_id, agent_pos)
        self.w.moveto(self.agents_rect[agent_id],
30*agent_pos['x']+2, 30*agent_pos['y']+2)

    # Переключение параметра, отвечающего за необходимость
обновления графического окна
    def toggle_drawing_mode(self):
        self.enable_drawing = not self.enable_drawing

    # Включить отслеживание пути для агента под номером agent_id
    def track_agent(self, agent_id):
        self.tracking_agents.append(agent_id)

    # Отрисовка пути, пройденного агентом
    def draw_path(self, agent_id, new_pos):
        # Проверка, отслеживается ли данный агент
        if agent_id not in self.tracking_agents:
            return
        # добавление нового элемента пути
        if self.agent_last_pos[agent_id] != None:

self.agent_paths.append(self.w.create_line(30*self.agent_last_po
s[agent_id]['x']+15,\

30*self.agent_last_pos[agent_id]['y']+15,\

30*new_pos['x']+15, 30*new_pos['y']+15, width=3))
        # Обновление цвета пути, чтобы показать течение времени
        for id, path in enumerate(self.agent_paths):
            red = 255 - 255 * (id+1) // len(self.agent_paths)
            self.w.itemconfig(path, fill="#%02x%02x%02x" % (255,
red, red))
        # Обновление последнего положения агента
        self.agent_last_pos[agent_id] = new_pos
```

```
    # Обновление содержимого графического окна
    def update(self, value, right_side, title):
        # Проверка на необходимость обновления графического окна
        if not self.enable_drawing:
            return
        # Обновление заголовка окна
        self.master.title(title)
        # Обновление строки состояния
        self.strategy_canvas.moveto(self.strategy_bar,
270*value+5)
        self.strategy_canvas.itemconfig(self.strategy_bar,
fill="lightgreen" if ["R", "F"][round(value)] == right_side else
"red")
        self.r_label.configure(underline=right_side != "R")
        self.f_label.configure(underline=right_side == "R")
        # Отрисовка всех обновлений на экране
        self.master.update()

    # Очистка всех путей агентов при сбросе игрового прогресса
    def clear_paths(self):
        if not self.enable_drawing:
            return
        for path in self.agent_paths:
            self.w.delete(path)
        self.agent_last_pos = [None] * len(self.agent_last_pos)
        self.agent_paths.clear()
```