ABSTRACT

Explanatory note 87 pp., 21 figs., 5 tables, 15 sources, 1 appendix.

MALMO CHALLENGE, MULTI-AGENT ENVIRONMENT, BEHAVIOR RECOGNITION, REINFORCEMENT LEARNING, Q-LEARNING, MINECRAFT

The object of research (development) is the strategy of interaction of artificial intelligence agents to achieve a common goal.

The aim of the work is to develop a model of agent behavior in a multi-agent environment to recognize the intentions of a partner and collaborate with him to achieve a common or their own secondary goal.

In the course of the work, a model of agent behavior based on artificial intelligence was developed for navigation in a two-dimensional space on a field with sparse obstacles and determining the intentions and goal of a partner. Two programs were developed: one for training the agent in a simplified two-dimensional environment, the second for testing its abilities in a three-dimensional environment close to reality. The three-dimensional environment is the Minecraft game, which requires continuous movement of agents (as opposed to discrete movement in a simplified environment) and provides a complex dynamic component of the environment. According to the results of testing both programs, the obtained accuracy of determining intentions and the probability of achieving a given goal by agents reaches 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%.

CONTENT

DEFINITIONS, DESIGNATIONS AND ABBREVIATIONS

The following terms and corresponding definitions are used in this explanatory note:

VKR – final qualifying work

GUI – Graphical User Interface

AI – Artificial Intelligence

INTRODUCTION

This work has the same goals as the Malmo Collaborative AI Challenge, on which it is based. The main goals include the need to research and develop AI agents that can collaborate with humans and other agents to solve a common task, as well as to answer a number of questions: how can software agents recognize someone's intentions and understand what task their partner is trying to solve, what behavior patterns contribute to solving a common task, and how can a group of agents communicate and coordinate the actions of all members of the group. Since research in the field of AI-based agent interaction is relatively new, not all of these questions have been answered precisely yet, which suggests the need for further development in this area and testing of different scenarios of agent interaction.

This work is based on an extended version of the Malmo challenge stage called Pig chase. The solutions to the official Pig chase are not applicable to the extended version, since the environment of agent interaction is significantly different. This provides an opportunity to experiment with machine learning methods different from those already used by participants in the official competition.

1. CURRENT STATE OF THE ISSUE

1.1. History of the issue

There are many definitions of artificial intelligence (hereinafter AI). As an example, we can cite the definition from the article “What is Artificial Intelligence” from 2007 - “It is the science and technology of making intelligent machines, especially computer programs. This field of science is closely related to the task of studying human intelligence using computers, but AI should not be limited to the application of biologically observable methods.” [1] However, the active study of AI began in the 1950s, when Alan Turing asked the question “Can machines think?” in his work, proposing a test containing a set of tasks that allow you to distinguish a person from AI. With the passage of time and the development of AI methods, the test has been repeatedly changed and supplemented, remaining relevant to this day under the name “Turing Test”.

One of the most influential works is Artificial Intelligence: A Modern Approach, last updated in 2022. In it, Stuart Russell and Peter Norvig distinguish two approaches to developing intelligent systems: human and ideal. In the first case, the system thinks and behaves like a human, which corresponds to Turing's definition, in the second, it thinks and behaves strictly rationally.

In its simplest form, AI is the ability of a computer to independently solve creative problems that are usually performed by a human. AI includes methods of machine learning, one of the subdivisions of which, in turn, is deep learning. These concepts are now often found in scientific papers. In particular, they are used in systems that predict or classify data based on the information fed to them.

There are two main types of artificial intelligence: weak (Artificial Narrow Intelligence, ANI) and strong (Artificial General Intelligence, AGI). The first type is aimed at solving a specific narrow range of problems. However, it is quite difficult to call it “weak”, since in some tasks it reaches the level of a person, and sometimes even exceeds it. The most famous examples are voice assistants such as Alice and self-driving vehicles. The second type is currently only a theory and is an AI whose intelligence corresponds to human intelligence in all criteria, moreover, strong AI has self-awareness, it is able to solve various problems, learn and plan the future. Sometimes a third type is distinguished - artificial superintelligence, which is theoretically capable of surpassing a person in the above-mentioned criteria. Despite the fact that at the moment there is no practical implementation of either the second or third types, scientists continue research in this area.

The first and one of the most successful narrow-focus AI is the Deep Blue computer, created with the sole purpose of teaching an AI agent to play chess. An AI agent (intelligent agent) is defined as "a program that independently performs a task specified by a computer user over long periods of time." [11] The agent is able to navigate in the environment (possibly virtual) and perform certain actions to receive rewards specified by the environment itself. Already in 1997, it became the first computer to win a match against world champion Garry Kasparov. This computer used a search tree for optimal moves up to 20 moves ahead. The function of assessing the quality of each move was configured manually, but was later optimized using software methods and was actively used in various fields. Moreover, the computer was programmed with the history of many games of grandmasters from all over the world, which served as the basis for assessing the quality of moves. Despite Deep Blue's success, it was far from the most powerful computer in 1997, leaving room for further improvement.

In 2013, DeepMind repeated the success of Deep Blue by training AI on Atari Games to the level of professional players. Despite the fact that the games in question provide a small set of possible actions, the AI was trained without using intermediate rewards from the system, but only on pixels from the computer screen and the results of each game. Some time later, another AI from DeepMind – AI AlphaGo defeated the world champion in the game with the same name AlphaGo.

There have been attempts to develop AI for StarCraft 2, but that game offers a much larger range of possible actions and a more detailed world. The player also needs to understand the intentions of their opponent and strategize accordingly. At the moment, existing AI implementations have not shown such high results as their ancestors in simpler games.

Collective intelligence is not limited to AI, but has been an integral part of human communication since the time of cave paintings. Drawings allowed people to share ideas and plan group hunts, greatly increasing their chances of success. People who can intelligently distribute actions among team members are extremely important in working with both other people and AI agents.

With the advent of first writing and then the Internet, people gained access to a huge amount of information, which certainly has a positive effect on the possibility of optimally distributing actions between members of different groups. With the increase in the volume of available information, programs began to appear that could process data much more efficiently than humans, which raised the question of creating software agents that could unite with people to achieve common goals.

Since the 20th century, there have been learning AI systems that are not able to collaborate with other agents or people, and there are also collaborative systems that facilitate collaboration between people via a computer, without a built-in learning mechanism. To create a collaborative AI, these two systems need to be combined. The first time such a combination was considered was in the multidisciplinary study “radical innovation”, which highlighted the requirements for creating a collaborative AI and showed a possible future for such systems. However, this work mainly describes different approaches to creating the system in question, such as social and developmental psychology, neuroscience, robotics, language models, and AI is only a part of the article. Based on “radical innovation”, a similar article was later published (Stefik & Price, 2023), but this time the focus was on AI. The article discusses scientific techniques and technologies for creating a collaborative AI and proposes an action plan for their implementation. As the author of the article “Roots and Requirements for Collaborative AI” notes in translation from English: collaborative agents must be trained in the same way as humans, in a collaborative environment, otherwise they will turn out to be fragile demonstration systems that will fail when performing real practical tasks. [2]

Joseph Licklider, in his work “Human-Computer Symbiosis,” presented the idea of integrating AI into human teams to better understand complex situations and plan actions. The idea is that a computer is given a specific task with input values and produces a numerical answer, while a person sets goals, formulates hypotheses, and evaluates solutions to problems. Joseph Licklider noted that in the future, computers will also help people set optimal goals, and in general, such symbiosis will be more productive than a computer solving a problem alone or in a group of people. However, one of the biggest obstacles to implementing plans was the inability to communicate with a computer in human language. In this regard, collaborative systems were created that allow team members to exchange information through a computer, but not directly with the computer itself.

1.2. Contemporary issues

When developing collaborative AI agents, the following most pressing problems (tasks) are identified [10]:

1. Each agent needs to promote coordination and strive to achieve common goals;
2. The agent must be able to model the tasks, intentions and behavior of other team members based on the state of the system (whether everything is going according to plan, whether the partner needs help, etc.);
3. All agents (especially human-controlled ones) must behave predictably;
4. Agents must be manageable;
5. The agent's state and intentions should be as simple and clear as possible to the rest of the team;
6. Agents must also be able to detect and process relevant messages about the status and intentions of their partners;
7. The agents themselves must have the ability to coordinate common (team) goals;
8. Planning and self-management technologies must be joint in nature;
9. Agents must be able to manage their attention;

10. All team members must participate in optimizing the costs of overall coordination.

When building collaborative systems, situations often arise where one of the agents has failed the task for some reason and is no longer able to play its role. In order for the system to continue functioning, such an agent must be signaled about the danger that has occurred or is approaching. In some situations, an agent can change its predetermined role without external dangers, which has a very negative effect on the team's joint progress, since it contradicts the original plan.

To remain a useful element of the system throughout its operation, each agent needs to understand the situation it is in. For example, to understand the position of other agents in the virtual environment, whether the other agents are following the initial plan, and if not, to try to understand their strategy for overcoming the difficulties that have arisen. Since agents are not yet able to communicate with each other like humans, most often they use a small set of specific actions that participants can take. This allows us to turn the problem of behavior processing into a simple classification problem. However, if some agents are under the complete control of humans, this approach becomes inapplicable.

The more predictable the agents are, the better each of them can model the current state of the system. If an agent frequently changes its goals or works suboptimally, it becomes difficult even for a human to understand its strategy. It can also be noted that the more adaptive the agent is, the less predictable it is. Most agent control algorithms are designed to be as transparent and specific as possible. This is the only way to build a reliable collaborative system.

Full transparency in the development of agent control mechanisms allows for the preservation of predictability of their behavior. Quite often, there are cases when it is necessary to intervene in such mechanisms with their subsequent modification, for example, to improve the performance of an agent. To meet all these requirements, special behavior policies are used that can be edited without changing the agent code itself. Having a full set of possible agent actions makes it easier to classify individual elements of its behavior.

There is a belief that good automated systems should be invisible to humans.[10] But with collaborative systems, the opposite is true: the more obvious each agent's actions are, the easier it is to understand its overall strategy. When observing the system, a person should not have questions like "What is this agent trying to do?" and "What does it plan to do next?" In a team made up entirely of humans, members tend to rely on their own thoughts when trying to analyze the behavior of others, but among agents, this approach is impossible, since it often leads to errors and misunderstandings, reducing the level of trust in AI.

Each agent's communication of behavior and intentions may not be enough to create a quality system if agents are unable to process them correctly. Based on the messages received from other agents, agents must form the current state of all team members, the environment, and the task at hand. The main problem is the inability of most agents to recognize the nuances of the participants' behavior unless they are directly pointed out. It is believed that until AI reaches human levels, asymmetry in coordination will prevent them from processing human-sent messages correctly.[10]

In cases where the situation (task, goal) changes and agents can no longer adhere to the original plan, it is necessary to provide the possibility of discussing a new strategy between agents. If agents cannot change the goal or explain this change to other participants, they will start from the original general plan. Usually, such situations are resolved by mechanisms and algorithms that accept the team goals of agents as input and return ready-made strategies with processing of all situations that may arise during its implementation. Obviously, this approach is not applicable if at least one of the agents is controlled by a person.

When working in a team, people divide the entire task into subtasks and, upon completion of each of them, discuss the results and think through further actions. For the system to work well, it is necessary to add a similar planning method to the interaction of agents. To implement it, agents should

1. keep the overall picture of the plan as detailed as possible, using all available feedback from other team members;
2. detect potential hazards that arise when continuing to follow the current plan and initiate re-planning if necessary;
3. evaluate the viability of changes made to the plan by other participants;
4. manage the rescheduling mechanism, including hiring additional agents if the process requires more effort than is currently available;
5. redistribute roles between agents if the plan has changed;
6. change the types of communication to better understand the information by agents.

When implementing the planning mechanism, it is worth considering that at the re-planning stage, agents may find themselves at different stages of implementing the general initial plan.

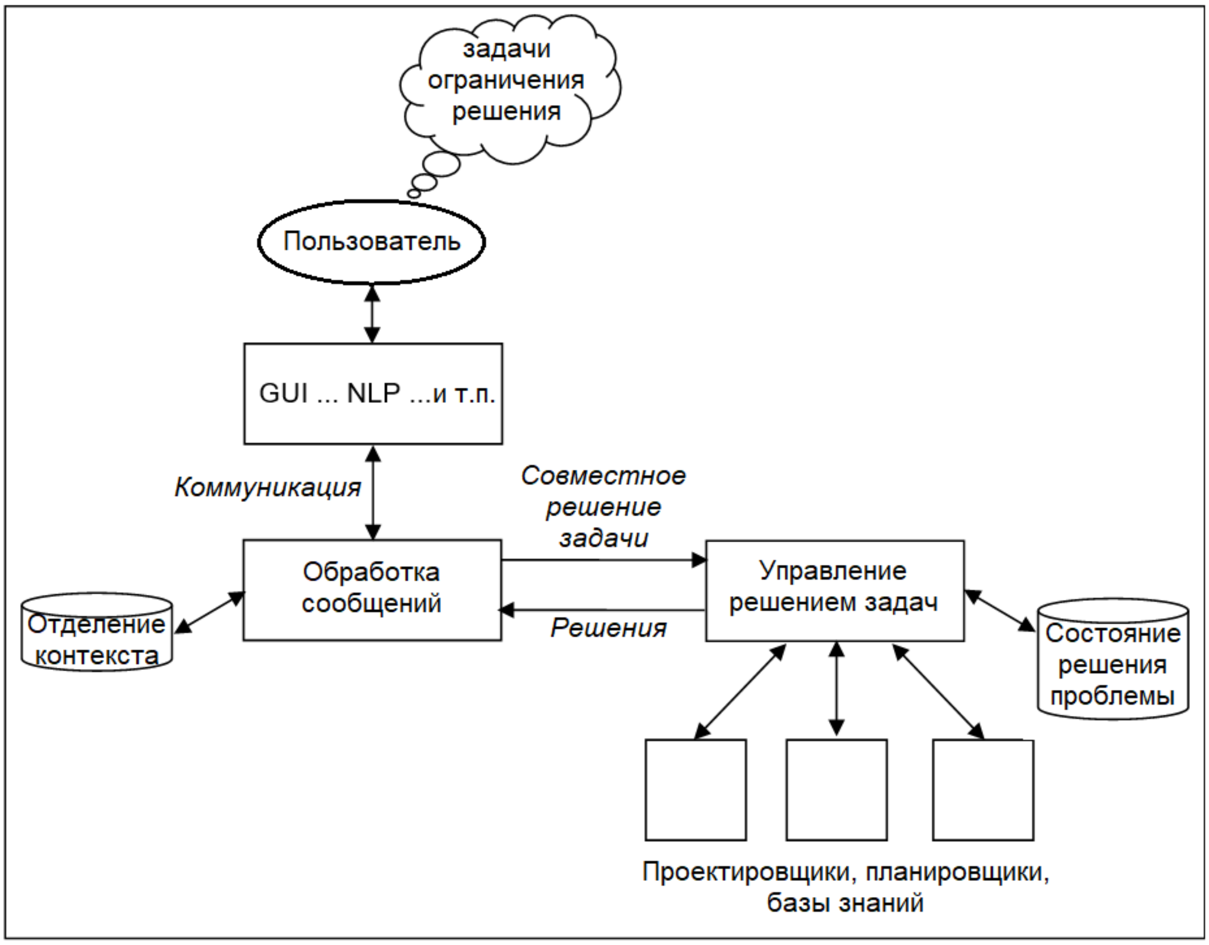
When communicating status and behavior messages, agents must specifically direct the attention of other participants to only the important details, ignoring minor actions and other noise. Agents need to understand what messages to expect from each team member and what to pay attention to, based on their internally formed model of the participants and the specific situation. If this process is not implemented correctly, situations may arise in which the automated system begins to compensate for various malfunctions without the observers noticing, and at some point it reaches its limit. At the moment when the capabilities of the automated system are exceeded, the failure may already be irreversible. The real breakthrough will occur when agents can communicate as freely as an organized team of people. However, at the moment, the questions "How can we determine that an agent is failing its task if the task itself has not yet failed?" and "How can an agent signal that the limit of its capabilities is about to be reached?" are still relevant. Signals about crossing a certain boundary of the agent's capabilities are often used to solve this problem. But such methods are rarely used in practice, since they do not take into account the context of the task and signal a malfunction too early, or, conversely, too late, when failure can no longer be avoided.

Coordinating the actions of agents, sending messages, and processing them can be time-consuming. Therefore, agents (like humans) should optimize these methods to reduce the costs of their implementation. Teamwork requires a constant investment of resources to maintain these processes, so it is impossible to completely eliminate costs (without resources, each agent will pursue its own goal and try to achieve it alone). However, reducing costs is only part of the process of optimizing the strategy. In addition, agents should adapt to their operators, and not try to adapt them to themselves just to save resources. The adaptation process can consume a lot of resources, since along with adapting to the new environment, agents must maintain predictability in their behavior, as mentioned earlier. The main problem is the insufficient level of adaptation of agents to the needs and knowledge of people.

1.3. Ways to solve problems

Collaborative planning systems often include features that are not found in traditional planning systems. For example, the development of a plan in such systems must be stepwise. In other words, it requires considering small parts of the plan in detail, compiling a list of possible actions, and making decisions on each part separately, and only after all parts are analyzed is a complete picture of the actions formed. Another important component of a planning system is stability. Changes in the final plan should be minimal when new constraints on possible solutions arise. If the system were to create the entire plan anew every time new difficulties arose, the final list of actions could change dramatically. Also, a fixed search strategy should not be used when creating a plan, since there is no specific order in which its subtasks are considered when forming a plan. Another important characteristic of the system follows from this criterion: openness to innovation. Under human guidance, the system should be able to form and analyze plans that it would not have created under other circumstances. [7]

The two main tasks that arise when creating a human-computer collaborative planning system are defining a context-independent level of problem solving and creating a hybrid plan analysis system. The first task is a layer between the human-computer communication system and the plan analysis system. The abstract structure of such a system is shown in Figure 1.1.


задачи
ограничения
решения
Пользователь
GUI ... NLP ...и т.п.
Коммуникация
Совместное решение задачи
Обработка сообщений
Управление решением задач
Решения
Отделение контекста
Состояние решения проблемы
Проектировщики, планировщики, базы знаний

Drawing1.1 – Structure of the joint planning system [7]

The figure shows a user with the goals he is trying to achieve, possible constraints on solving the problem and, possibly, part of the solution itself. This information is transmitted to the system via GUI (Graphical user interface) and NLP (Natural Language Processing), which are the most frequently used means of human-computer communication. The most famous and currently available examples of using such an interface are virtual assistants Alice, Siri and others. The system then transmits the received information to the task management center, which, in turn, creates and saves the current state of the solution. Once the user's intentions have been interpreted and the state generated, the task management unit calls traditional artificial intelligence methods: classification, planning, knowledge base search, etc., which are most suitable in the context of the current task. With such a system structure, functionality (AI components) can be added dynamically, right during operation, if required by the task. However, traditional methods are not always suitable for sequential planning, when constraints can be added and removed, or in situations where the user offers a ready-made part of the solution.

The purpose of the problem solving control center is to support joint planning and maintain a history of problem solving states. To successfully perform its functions, it is necessary to define the following requirements for the system structure:

1. The user's goals and objectives must have a hierarchical structure: they must be able to be detailed and focused on individual parts of the overall task, in which all levels are linked by constraints on possible solutions;
2. Each level of the hierarchy is an abstract description of all its components, and the relationship between them consists of constraints and assumptions of independence;
3. It must be possible to formulate a specific action plan based on the hierarchy of tasks, satisfying all constraints and taking into account all possible solutions and their duration at each stage;
4. The plan described in the previous paragraph should be presented to the user with a detailed description of the environmental conditions throughout the implementation of the plan, taking into account possible external factors and methods of their treatment.

These 4 requirements allow traditional AI methods to be applied directly to finding solutions to the problem and to maintain maximum efficiency of user-computer interaction. Context-independent algorithms of temporal logic [7] (this is "logic in whose statements the time aspect is taken into account; it is used to describe sequences of phenomena and their interrelation on a time scale" [12]) are used to create an abstract plan based on a hierarchical structure of goals. Namely, relying on the abstract plan, traditional AI methods form specific actions that are ultimately transferred to the user of the system. Usually, the user, having received the action plan, adds constraints and requests a plan that takes them into account. Together with the sequence of actions itself, the system identifies three states of the environment: before, during and after the plan is executed, which allows the user to more accurately assess the result of the AI's work.

However, despite the efforts of the task management center, the problem of correctly interpreting tasks using classical AI methods remains.The advantages of a planner that can handle the task context on its own are speed, expressiveness (since the user needs the most complete information about actions, plans, time, and the state of the environment), completeness (the agent does not have to evaluate all existing plans, but must support the ability to gradually approach any of them), and flexibility (the agent must adapt to changes made by the user).

1.4. Conclusion on the section

Thus,At the moment, there are already quite a few approaches to creating joint AI systems, most of which are human-computer joint planning systems. However, the specific methods used to implement the systems can vary greatly and depend on the task set for the system. The topic of creating joint AI, despite its first mentions in the 1950s in the works of Alan Turing, began to be actively studied and implemented in everyday life only in the 2020s, which shows the need to continue research and search for more efficient and accurate planning algorithms in various areas of information technology.

For further research into collaborative planning systems, it would be useful to develop a similar system, for example for the Malmo competition, bringing the virtual environment closer to reality. In this case, despite the simplicity of the task of binary classification of agent behavior, new algorithms would be needed, since the methods used in the original version of the competition would lose their relevance in the extended version.

2. Description of the development

This work is based on the Malmo project [9], developed by Microsoft. The goal of the project is to explore the possibilities of AI and machine learning, in particular the reinforcement learning method. “Reinforcement learning is a machine learning method in which our system (agent) learns through trial and error. The idea is that the agent interacts with the environment, learning in parallel, and is rewarded for performing actions.” [4] On the project page, the organizers ask the following questions:

1. How to develop AI that can navigate complex virtual environments?
2. Can it learn, including from humans, to interact with the environment?
3. Can such an AI store and transmit its knowledge accumulated over its entire existence to solve new problems?

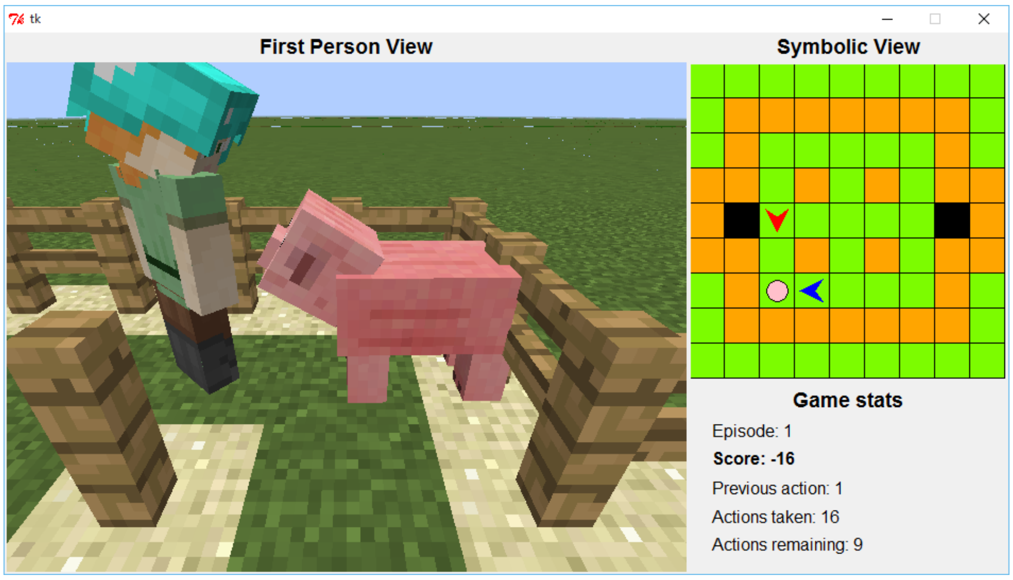
The project provides users with a platform for various experiments, consisting of the Minecraft game, which allows players to make almost any changes to the virtual world and interact within it; modifications of this game to allow control of in-game characters through scripts written in Python; and several studies, which are given as examples of the use of the platform. In addition to the platform for experimenting with AI, the Malmo project provides the opportunity to participate in competitions dedicated to various areas of AI.

This paper proposes a solution to the 2017 Malmo Collaborative AI Challenge (also known as Pig chase) [8]. This challenge is about human-software agent interaction to achieve a common goal. Particular attention is paid to the following questions:

1. How can AI agents recognize someone's intentions?
2. How can AI agents understand what behaviors are beneficial for teamwork?
3. How can agents communicate and coordinate strategies to achieve a goal?

These questions are of particular interest because no precise answer has yet been found to any of them.

The task is a mini-game, based on the theoretical game of "stag hunt", in which two agents must work together to get the highest reward, or act alone for a chance to get a smaller reward. The mini-game takes place on a plane with a perimeter limited by obstacles; in addition to the two agents, there is another entity on the plane (a pig, as the name of the stage suggests). Each agent has a choice: to catch the entity (by driving it into a corner or between obstacles, limiting all escape routes), receiving the maximum reward, or to give up and leave the game, moving to a predetermined position and receiving a small reward. [8]



Episode: 1
Score: -16
Previous action: 1
Actions taken: 16
Actions remaining: 9

Drawing2.1 – Still from the Pig chase mini-game [8]

In Figure 2.1, two images representing the same state of the system can be seen: through the eyes of one of the agents navigating in a three-dimensional environment (first-person view) and a simplified two-dimensional symbolic diagram. The upper right image reflects the full state of the system at the current moment in time. It shows directional arrows denoting the position and orientation of two agents, a pink circle for the entity's position, green squares for free space, orange squares for obstacles, and black squares for game exit points. Below the symbolic diagram are numerical statistics of the state: the number of games completed during the program's execution, the reward value for the current agent (the agent from whose point of view the screenshot on the left was taken), the numeric identifier of the agent's previous action, the number of actions performed by the agents during the game, and the maximum possible number of actions taken before the game ends. If the agents exceed the maximum number of actions before catching the entity or giving up, the game ends automatically, and the final reward for each agent is equal to the corresponding value in the Score field.

In this work, the game level was expanded: the dimensions of the plane were increased, obstacles were added, and the position of the exit point was moved. The level used in all further illustrations looks like this (without taking into account the obstacles that limit the level along the perimeter, since the goal is always within the level):

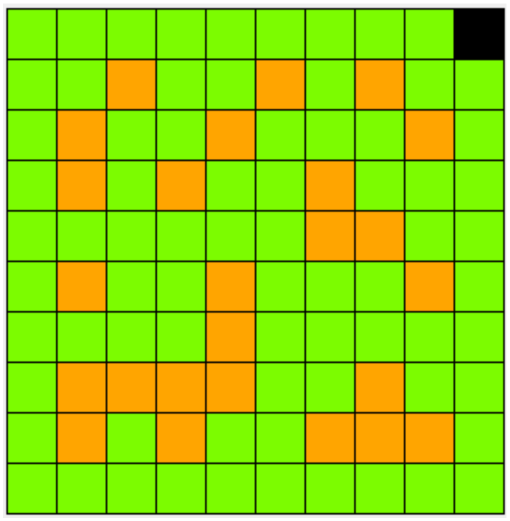


Figure 2.2 – Symbolic diagram of the level used in this work

Level expansionis necessary for creating and testing new AI algorithms, since many solutions based on popular traditional reinforcement learning algorithms have already been proposed for the level defined in the original competitions. In particular, algorithms that learn and test on fixed levels. In other words, for such algorithms to work, the structure of the level (obstacles) must remain constant. The extended version assumes the possibility of testing agents on levels with dynamically generated obstacles.

2.1. Principles, materials and methods

Since the Malmo project focuses on reinforcement learning methods, the Q-learning algorithm is recommended for multi-agent environments.[13]

Q-Learning is a model-free reinforcement learning algorithm that allows an agent to learn the cost of each possible action in each state of the system. [13] The most basic implementation of the algorithm includes a table (Q-table) whose columns are the agent's actions and whose rows are the system states. Each cell reflects the cost of an action in the current state, which the agent learned during model training. The model is trained by receiving rewards for actions performed, provided by the virtual environment. The model remembers which actions in each state lead to a positive outcome, and which lead to punishment for the agent. When the program is launched, the Q-table can be filled with either zeros or random values. Then, after each action of the agent, one cell of the table is updated, located at the intersection of the agent's previous action () and the state of the system at the time of this action (), according to a formula called the Bellman equation or utility function [13]:$a_t s_t$

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

Where Q is the Q-table, is the learning coefficient, which determines the degree of confidence in the model of new information, and is the discount coefficient, which allows the system to prefer short-term or long-term rewards. The search for the maximum in the expression occurs relative to the available set of actions in the state after the action is performed. There is no general rule for calculating the values of the coefficients; they are selected separately for each situation, based on the results of the system's operation. If the agent's next action leads to the end of the game, the table is not updated at this stage. The learning algorithm looks like this:$\alpha\gamma$ $\max Q(s_{t+1}, a)$ $s_{t+1} a_t$

1. The agent performs an action – either random or corresponding to the largest value (reward) in the Q-table opposite the current state of the system;
2. The agent updates the value in the Q-table cell mentioned in the previous point (even if a random action was selected);
3. The randomness coefficient of the agent's behavior is reduced; now it more often relies on a model formed in the form of a Q-table;
4. Go to step 1 if the end of training signal was not received.

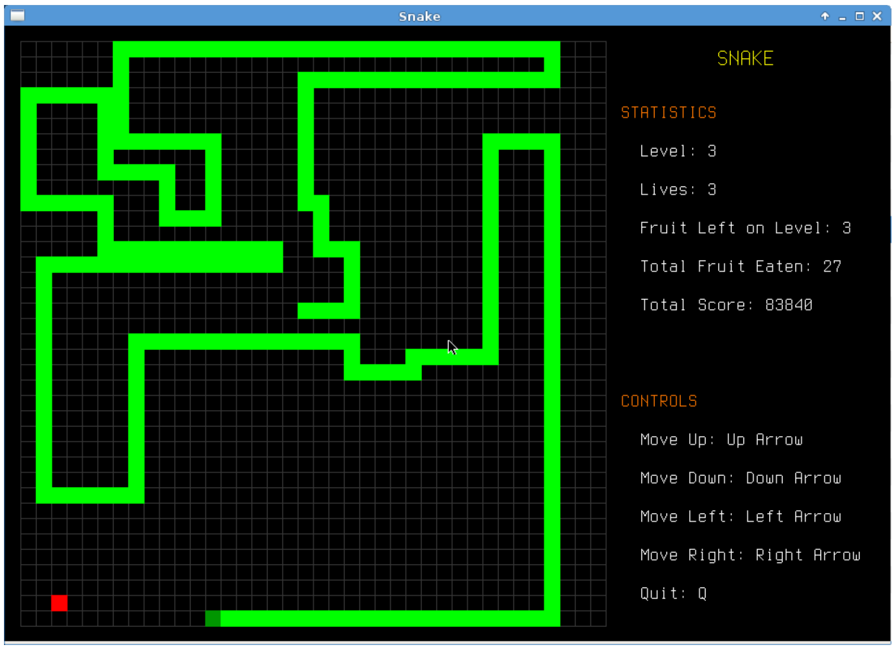
Thus, when randomness is eliminated from the agent's behavior, each action it chooses yields the greatest reward value. Randomness is necessary for the agent to fully understand the environment through exploration. The reward itself, stored in a cell of the Q-table, is a weighted sum of all future rewards, starting from the current state of the system. Since the set of all states and transitions between them is a Markov decision process, with a sufficiently long learning phase and a static environment, the values of the Q-table should reach an optimum for all possible states. [13]

However, as mentioned in [13], the multi-agent environment needs to be transformed into a single-agent environment (to which Q-learning is applied). This is due to the convergence requirement of the Q-learning method: the environment must be static. In other words, if the same action in the same state of the system can lead to different reward values, the optimal state of the Q-table will never be reached. In this regard, the development of the program was divided into the implementation of two directly independent algorithms: the algorithm for moving the agent along the level and the algorithm for recognizing the partner's intentions. The solution to both subproblems is based on Q-learning.

Among the methods proposed by the participants of the competition, there are algorithms that do not use any additional means of communication between agents other than their location on the level. Such methods best emphasize the importance of the need for not only high-quality behavior recognition, but also high-quality demonstration of one's intentions. In this regard, it was decided to create an algorithm that requires minimal communication between agents, such as transmitting their current coordinates on the level.

As an example, consider the method proposed by the participants who took first place in the 2017 competition [5]. Their intention recognition algorithm assumes two strategies, one of which is followed by the partner agent throughout the game. The agent can act either completely randomly or in the most optimal way to achieve the common goal. It is implied that the partner cannot change the strategy during the game itself. Next, they transform the multi-agent environment into a single-agent environment, declaring the partner as part of the virtual environment. The small size of the original level allows you to build a tree of all possible situations on the playing field during model training. At the testing stage, this will allow you to determine the partner's strategy quite accurately and quickly using the Monte Carlo search tree algorithm. The probability value of each strategy obtained at this stage is processed through Bayesian inference. The disadvantages of this algorithm include the need for a small level size to save all states in RAM and maintain high operating speed.

Since the work consists of two parts, we should start with the implementation of predictable behavior of the goal-oriented agent. Due to the increase in the size of the level used in this thesis, it became impossible to store all possible states in memory, therefore, the method described in the previous paragraph will not work. One of the most famous games in which it is necessary to achieve a goal bypassing obstacles on a plane is "Snake". Its concept is very close to the mini-game Pig chase: the snake exists on a two-dimensional plane divided into discrete cells, the goal is located in one of these cells, and the role of obstacles is played by the body of the snake itself, growing as the game progresses.



```
Snake

SNAKE

STATISTICS
  Level: 3
  Lives: 3
  Fruit Left on Level: 3
  Total Fruit Eaten: 27
  Total Score: 83840

CONTROLS
  Move Up: Up Arrow
  Move Down: Down Arrow
  Move Left: Left Arrow
  Move Right: Right Arrow
  Quit: Q
```

Figure 2.3 – Illustration of the game "Snake"

From Figure 2.3, we can see that the controls in Snake are also the same as the possible actions of each agent in Pig Chase: moving left, right, down, and up. The biggest difference is the handling of the character's collision with an obstacle or the edge of the level. In the case of Snake, the game progress is completely reset, while in the case of Pig Chase, the agent does not move from the spot, while decreasing the counter of the maximum number of actions until the end of the game. Since a collision with an obstacle does not globally affect the agent's progress towards the goal, this difference can be neglected and a learning model similar to the learning model of Snake can be applied.

Fortunately, the most common machine learning models used for snake games are based on the Q-learning method. Consider the states that make up the Q-table for the snake game. [3]

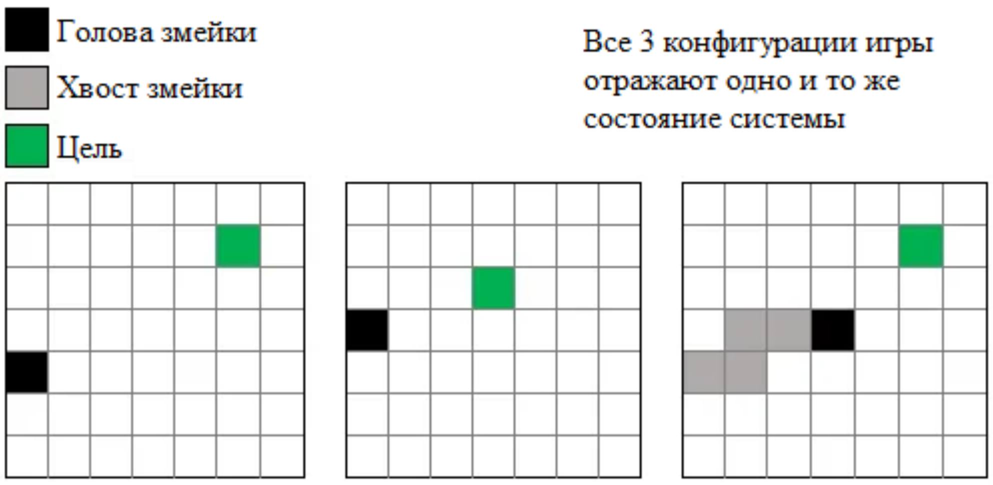

Голова змейки
Хвост змейки
Цель
Все 3 конфигурации игры отражают одно и то же состояние системы

Figure 2.4 – Visualization of system states for the game “Snake” [3]

As mentioned earlier, it is impossible to store all combinations of the positions of the snake's head, target, and obstacles (each block of the snake's tail) for the field shown in Figure 2.4 as an example. Therefore, the state of the system is collected from the following parameters:

1. The horizontal position of the snake's head relative to the target (is it to the right, left, or at the same level)
2. The vertical position of the snake's head relative to the target (similar to point 1)
3. The presence of obstacles in adjacent cells to the left, right, above and below.

The third item is a combination of four binary values corresponding to each side. Thus, the Q-table has 3 * 3 * 24 = 144 rows and 4 columns corresponding to each movement of the snake's head to adjacent cells, which is significantly less than the number of all combinations of environmental parameters. It is worth noting that the level boundaries, although not shown in Figure 2.2, are treated similarly to obstacles.

In [3], positive rewards are given for moving the snake's head towards the goal and for reaching the goal, and negative rewards are given for colliding with an obstacle (the tail or the edge of the level) and moving the head away from the goal. However, the reward system in Pig chase is different: the agent is punished for every action performed, regardless of its impact on the gameplay, and is rewarded only for reaching the goal. Due to the change in the level structure, the agent's reward system should also be edited.

Since it was decided not to introduce any additional communication methods between agents to determine the intentions and predict the behavior of the partner, but to draw conclusions only based on its position, the Q-learning method is also used in the second part of the task. Despite the simplicity of this method, it should show good results, since when considering the original level (without expansion and other modifications), in most situations it is possible to limit oneself to a fixed set of behavior classification rules, regardless of the algorithm for achieving the goal by the agent. For example, in [5], the participants preferred the planning method to the model learning method. That is, in their work, at the learning stage, a search tree was built in order to subsequently draw up (plan) the most optimal action plan based on it.

2.2. Theoretical research and calculations

To achieve optimal values in the cells of a Q-table of this size, several hundred or thousands of training episodes are needed. The Minecraft environment is too demanding for training the model, so it was decided to create a simplified version of the virtual environment. Since both agents and the creature move only horizontally, the 3D level can be represented as a 2D array containing free cells and cells with obstacles. An example of such an array is shown in Figure 2.2. Agents and creatures are able to move discretely to neighboring cells of this array, achieving their goals according to the same rules as in 3D space.

The learning and discounting coefficients are taken directly from [3] due to the similarity of the tasks: 0.7 and 0.5 respectively, however, the randomness coefficient of actions, initially equal to 0.9, is not fixed throughout the entire training period, but decreases by 1.3 times after each episode. The value is selected in such a way that by the 10,000th step of the agent there is almost no randomness left in its actions. Since at this stage only the algorithm for achieving the goal by one agent is designed, it is necessary to change this goal accordingly: the coordinates of the agent and the goal must coincide, only in this case the current game (episode) ends.

The quality of the first half of the learning process can have a strong impact on the resulting model. There may be situations when the agent, making random movements, fails to reach the goal several times in a row. In such a case, the model may stop paying attention to the direction to the goal, which is taken into account when forming the state of the system. The situation is aggravated by the lack of difference between rewards when the agent moves towards the goal or in the opposite direction (this difference is present in the implementation of Q-learning for the Snake game in [3]). Such indifference of the system can be reflected in an infinite looped transition of the agent between two neighboring cells. There are 3 solutions to this problem:

1. Prevents movement to the cell of the agent's previous location;
2. Large negative reward for moving to a cell of the agent's previous location;
3. Adding a small randomness of movement during agent testing (so that there is a chance to exit the cycle by moving to another adjacent cell);

The first method completely cuts off the possibility of entering cycles from two adjacent cells, but it leads to another problem. It can be found by examining the following situation in detail:

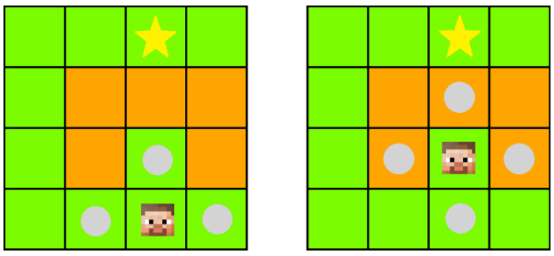


Figure 2.5 – The agent cannot take a step back and ends up in a dead end

Figure 2.5 shows a situation in which the agent, based on the current state of the system, does not detect obstacles on the way to the goal (the star) and takes a step towards it, finding itself trapped. Since the goal can be located "above" at any distance, this situation often occurs in practice, therefore, removing the ability to take a step back is irrational.

The ability to punish the agent is an optimal option to force it to avoid loops. This way, the last action of the agent is added to the parameters of the system state (which is similar in meaning to the direction of the last visited cell, but more standardized). However, this solution only saves from the simplest cycles involving two neighboring cells. The agent can avoid punishment by using four cells, which is also a situation that often occurs in practice. It is irrational to create an array of all visited cells, since the target may change its position, which will lead to a search for a longer path, and the size of the data describing the system state will increase significantly.

For more complex loops, it is necessary to introduce a small probability of the agent performing a random action. This randomness should be present both during the training period and during testing of the agent, since a collision with an obstacle in the Minecraft environment does not lead to any serious consequences. The chosen value for the probability of performing a random action is 0.00005. It is quite insignificant for episodes in which the agent does not enter the loop.

Side effects that break such cycles include exceeding the maximum number of steps per episode, which restarts the game, and target movement, which is an integral part of the virtual environment.

The movement of a random agent cannot be implemented by choosing a random neighboring cell. Before performing an action, the agent cuts off all valid movements that lead to a cell with an obstacle, and then makes a random choice from the list of remaining actions. This approach makes the intentions of a random agent more predictable. Entity movement does not include all the logic of creature behavior in Minecraft, since it is random and creatures are in the same position most of the time.

Q-training of the intention recognition model is performed only after obtaining the optimal values in the Q-table in the previous step, so that the difference between the behaviors of the two types of agents is most obvious. Since the behavior and actions of agents can only be observed in time, the data describing the state of the system in the agent navigation algorithm along the level is insufficient. The most obvious solution would be to create a chain of states sorted by time. However, this solution will lead to a large memory consumption, and therefore should be optimized. Since a goal-oriented agent must approach the goal over time, which distinguishes it from an agent with random behavior, a new state of the system for each agent can be composed from the following parameters:

1. Distance from agent to target in L1 metric (Manhattan distance);
2. The ordinal number of the agent's step from the beginning of the current episode (game).

Instead of the Euclidean distance, it is preferable to use the Manhattan distance, since for any state of the system it is equal to some integer, which makes it easier to classify when referring to the Q-table. In this case, the interval of possible distance values is [0, 18] for the level shown in Figure 2.2 (maximum 9 cells horizontally and 9 vertically relative to the goal). The interval of admissible ordinal numbers is located from 0 to the maximum number of steps in one game. This solution allows us to determine the agent's intentions using only the distance from him to the goal at each discrete moment of time. However, since randomness plays a large role in the agent's position relative to the goal, the time for the values in the Q-table cells to converge to the optimum may be large. In this regard, it was decided to group the values of the ordinal numbers of steps.

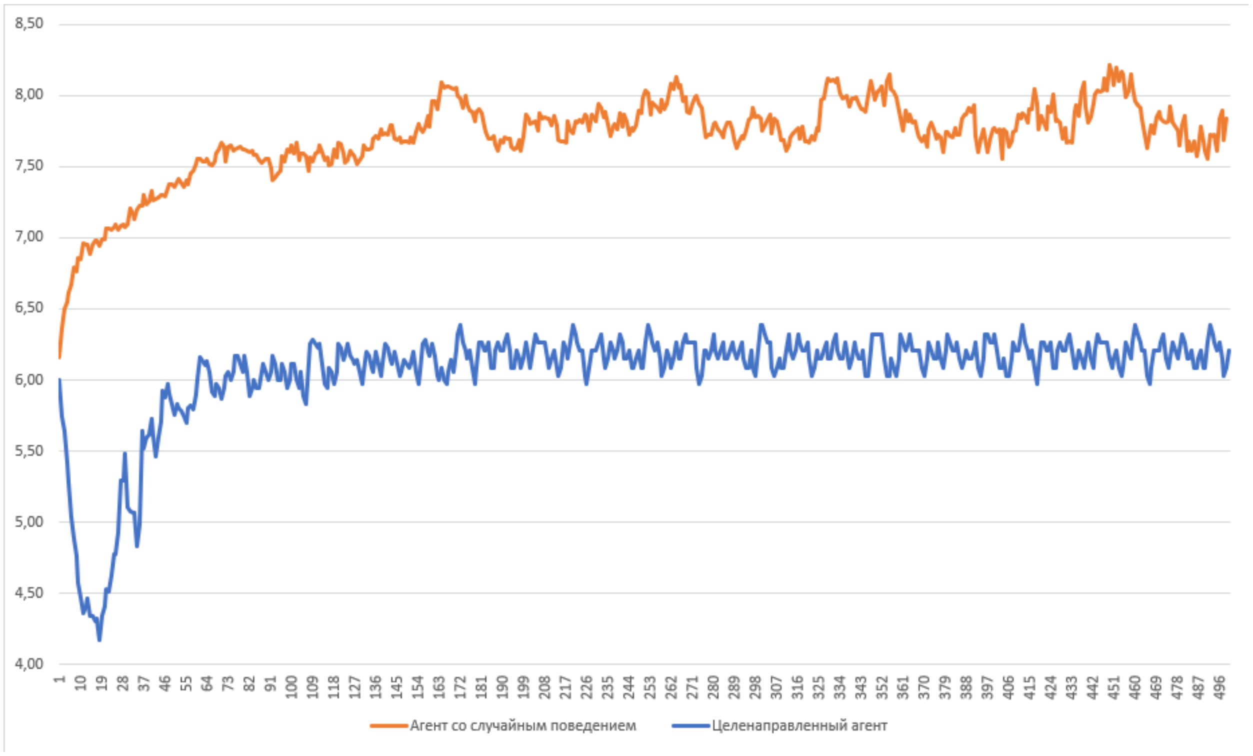


Figure 2.6 – Average distance of agents to the target at each step

Figure 2.6 illustrates the average distance from agents to the goal at each step of the game. The maximum allowed number of steps per episode in this example is 500. It is worth paying special attention to the fact that if the game ends before the 500th step of the agent, the remaining steps are ignored and the array of distances corresponding to them is not filled with anything. In other words, the higher the step number on the abscissa axis, the less data was collected for this step. In this figure, it can be seen that the differences between the random agent (marked in orange) and the goal-oriented agent (marked in blue) appear already at the first step. Then, the greatest difference between the distances is observed at approximately step 19.

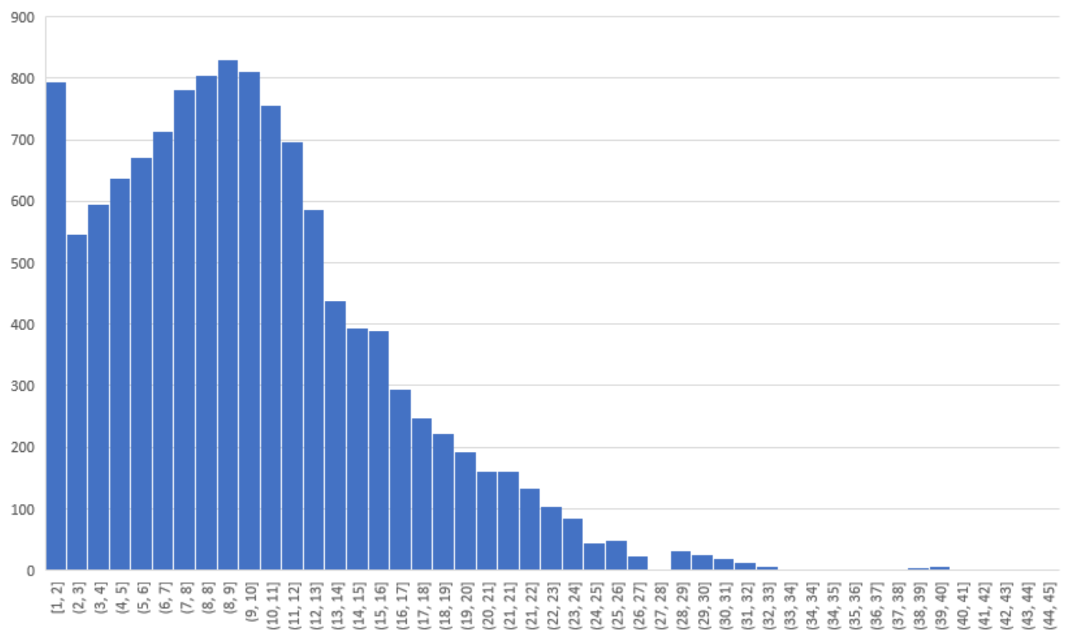


Figure 2.7 – Number of completed games for the ordinal number of each step of the goal-oriented agent from the beginning of the episode

However, based on Figure 2.7, it can be noted that most episodes end already at the 9th step of the goal-oriented agent, therefore, it is irrational to divide the ordinal numbers of the steps into two groups relative to the 19th step. At the same time, at the 9th step, the difference between the distances of the two types of agents is not large enough. It is worth noting that the agent predicting the intentions of its partner cannot immediately end the game. First, it needs to get to the specified coordinates (the goal). Therefore, since the average distance to the goal at the 19th step is approximately 4, it is possible to divide the ordinal numbers into groups relative to the 15th step. As the histogram in Figure 2.7 shows, a large number of games were completed before the 15th step, which allows us to choose it as the dividing point for the two groups of ordinal numbers of the steps. Thus, the ordinal numbers of the agent's steps from the beginning of the game were divided into 3 groups: up to and including step 1, up to and including step 15, and the rest.

As for the Q-table columns of the intent determination method, there are only two: the agent follows random behavior and goal-oriented behavior. Instead of describing actions, as in the Q-table used to navigate the agent through the level, here they act as a classification of the partner's behavior. The result of the classification is used to determine the goal of the agent performing behavior analysis. The goal can be either the coordinates of a creature or the coordinates of a premature end of the game point.

To enable the transfer of the trained model to the testing environment, both Q-tables are written to a file. On the Minecraft side, a goal-oriented agent with similar functionality is implemented using Python and a modification provided by the Malmo project, performing actions based on the values stored in the file with the Q-table. Part of the agent's functionality was adapted to work with a structure describing the state of the new environment. The part responsible for training both models was completely cut out. A more detailed comparison of the state structures is provided in the next section.

2.3. Model, block diagrams of the program code

This section contains flow charts and class diagrams for both the simplified environment and the Minecraft game environment, organized using a modification of the game provided by the Malmo project. All code for this thesis is written in Python. The diagrams are created using Visio.

The main elements of the simplified environment operation algorithm are shown in Figure 2.8.

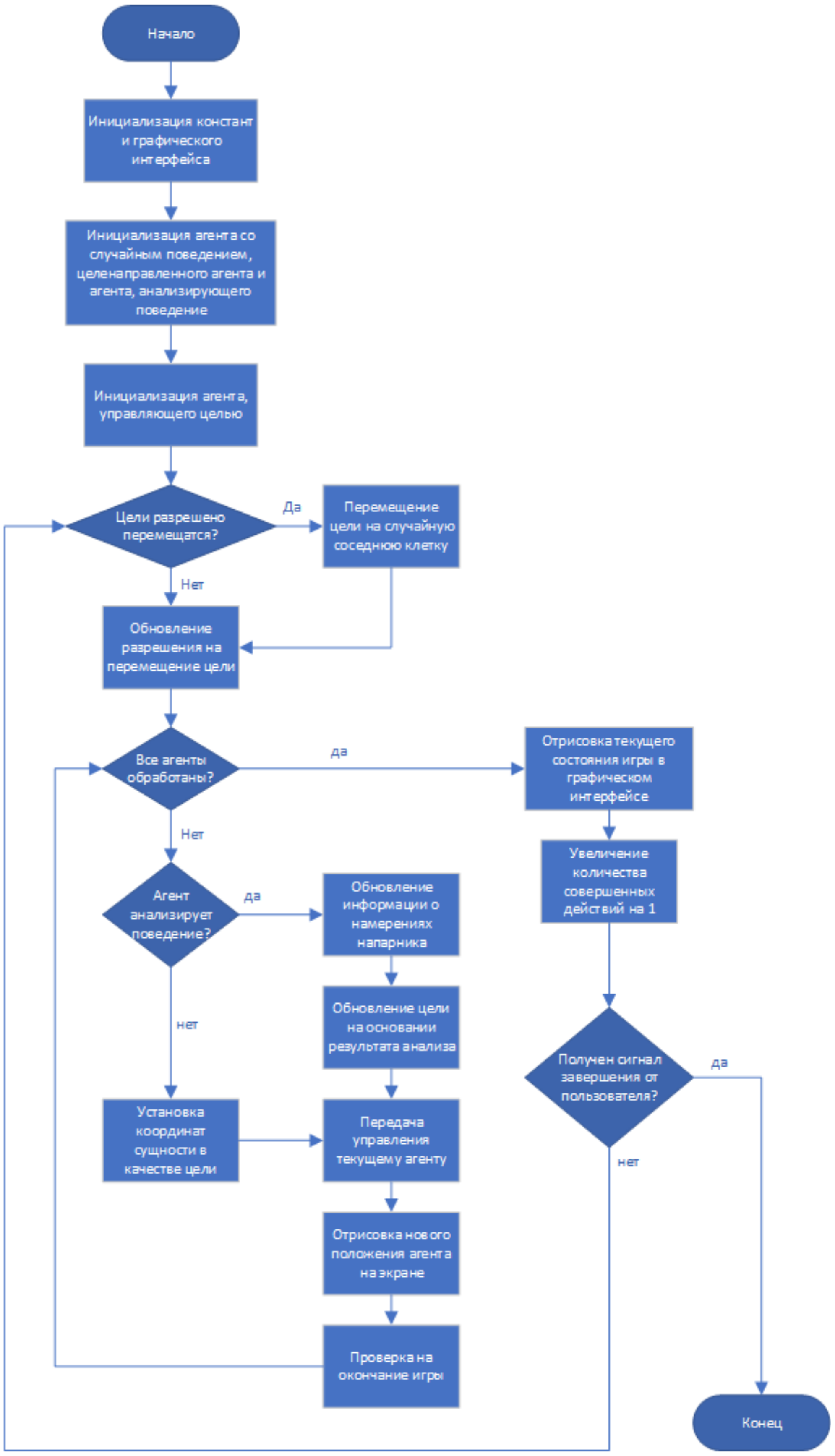

Начало
Инициализация констант и графического интерфейса
Инициализация агента со случайным поведением, целенаправленного агента и агента, анализирующего поведение
Инициализация агента, управляющего целью
Цели разрешено перемещатся?
Да
Перемещение цели на случайную соседнюю клетку
Нет
Обновление разрешения на перемещение цели
Все агенты обработаны?
да
Отрисовка текущего состояния игры в графическом интерфейсе
Нет
Увеличение количества совершенных действий на 1
Агент анализирует поведение?
да
Обновление информации о намерениях напарника
нет
Обновление цели на основании результата анализа
Получен сигнал завершения от пользователя?
да
Установка координат сущности в качестве цели
Передача управления текущему агенту
нет
Отрисовка нового положения агента на экране
Проверка на окончание игры
Конец

Figure 2.8 – Generalized algorithm for the operation of a simplified virtual environment

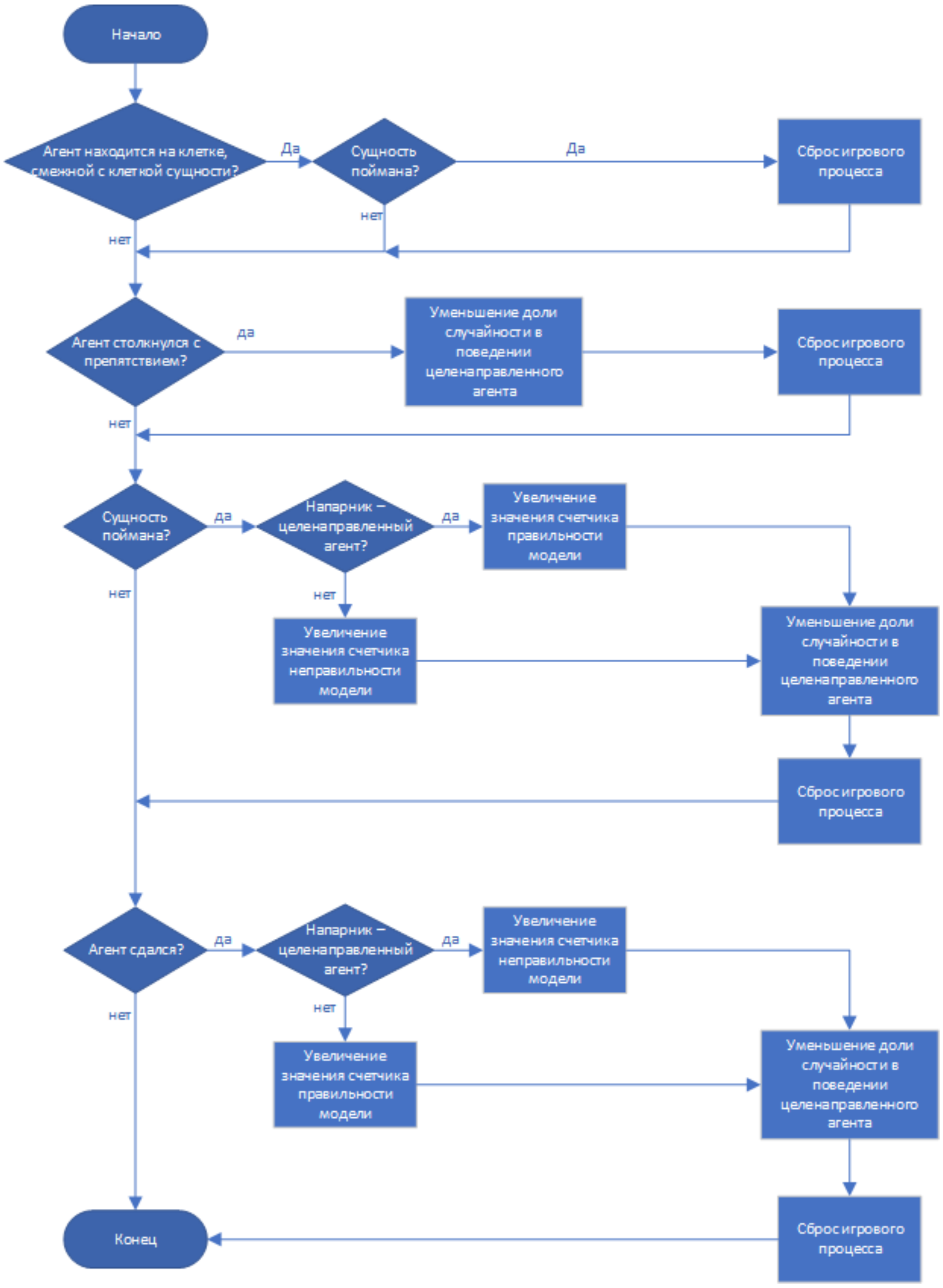

Начало
Агент находится на клетке, смежной с клеткой сущности?
Да
Сущность поймана?
Да
Сброс игрового процесса
нет
нет
Агент столкнулся с препятствием?
да
Уменьшение доли случайности в поведении целенаправленного агента
Сброс игрового процесса
нет
Сущность поймана?
да
Напарник – целенаправленный агент?
да
Увеличение значения счетчика правильности модели
нет
нет
Увеличение значения счетчика неправильности модели
Уменьшение доли случайности в поведении целенаправленного агента
Сброс игрового процесса
Агент сдался?
да
Напарник – целенаправленный агент?
да
Увеличение значения счетчика неправильности модели
нет
нет
Увеличение значения счетчика правильности модели
Уменьшение доли случайности в поведении целенаправленного агента
Конец
Сброс игрового процесса

Figure 2.9 – Details of the “Check for game end” block

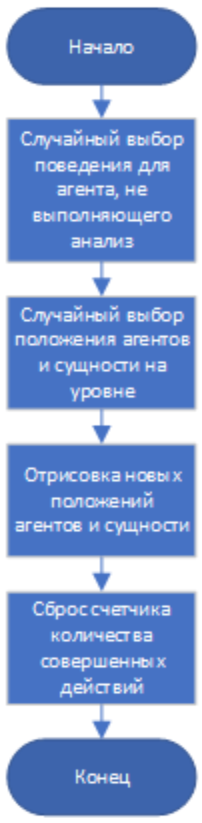

Начало
Случайный выбор поведения для агента, не выполняющего анализ
Случайный выбор положения агентов и сущности на уровне
Отрисовка новых положений агентов и сущности
Сброс счетчика количества совершенных действий
Конец

Figure 2.10 – Details of the “Reset Game Progress” block

In Figure 2.8, you can see the ability to terminate the algorithm only at the user's request. This is due to the fact that after the model is trained, the program automatically switches to testing mode, displaying the gameplay in real time. The user exits the testing mode by independently terminating the program by closing the graphical interface window. The source code of the algorithm in Figures 2.8, 2.9, and 2.10 is given in Appendix A, Listing A.1.

The entity movement is performed through the agent class with a randombehavior. This is reflected in the block "Initialization of the agent controlling the target". The instance of this specific agent is not included in the list of system agents, therefore, its behavior is not analyzed. It is also not taken into account when checking the condition "Are all agents processed?". The periods of movement and stop of the entity are determined by a special timer, which can take values from -5 to 10. If the timer value is negative, the entity is allowed to move, and control is transferred to the class implementing the behavior of the "random" agent. If the value is positive, the entity stands still. Each action performed in the system brings the timer closer to 0 by one (performs a decrement or increment action). As soon as the timer value reaches 0, a new random value is selected in the range [-5, 10]. These two sentences describe the block "Updating permission to move the target".

Block "Check for the end of the game"is shown in detail in Figure 2.9. Events that lead to the end of the game include the agent colliding with an obstacle or the edge of the level, successfully capturing an entity, and one of the agents prematurely ending the game. It can be seen that before each reset of the game process, goal-oriented agents have a smaller proportion of randomness in their behavior. In other words, the interval in which a randomly chosen number must fall for the agent to take a random action narrows. For an agent with random behavior, this block is ignored.

The "Game Process Reset" block is placed in a separate figure 2.10 due to its numerous repeated uses in figure 2.9. It is responsible for resetting all agent achievements for the last episode, except for the general statistics that the user needs to evaluate the results of the program. Also, within this block, a random selection of behavior is made for the agent being evaluated. At the code level, an instance of the agent class of one type is replaced by an instance of another type, or remains in its place.

Let's consider the hierarchy of agents, presented as a class diagram in Figure 2.11. The hierarchy is built in such a way as to maximally facilitate the transfer of agent functionality to the testing environment, that is, to integrate it into the Malmo project. Observer is an abstract class representing an object with coordinates (position), a limit on the maximum number of steps per game (max_fitness), information about the last action (last_move), the state of goal achievement (goal_achieved), and the availability of the goal (edible). The availability of the goal is understood as the possibility of surrounding the entity. If 2 agents are defined in the system, then at least 2 obstacles must be located on the cells adjacent to the goal, in which case the goal will be achievable.

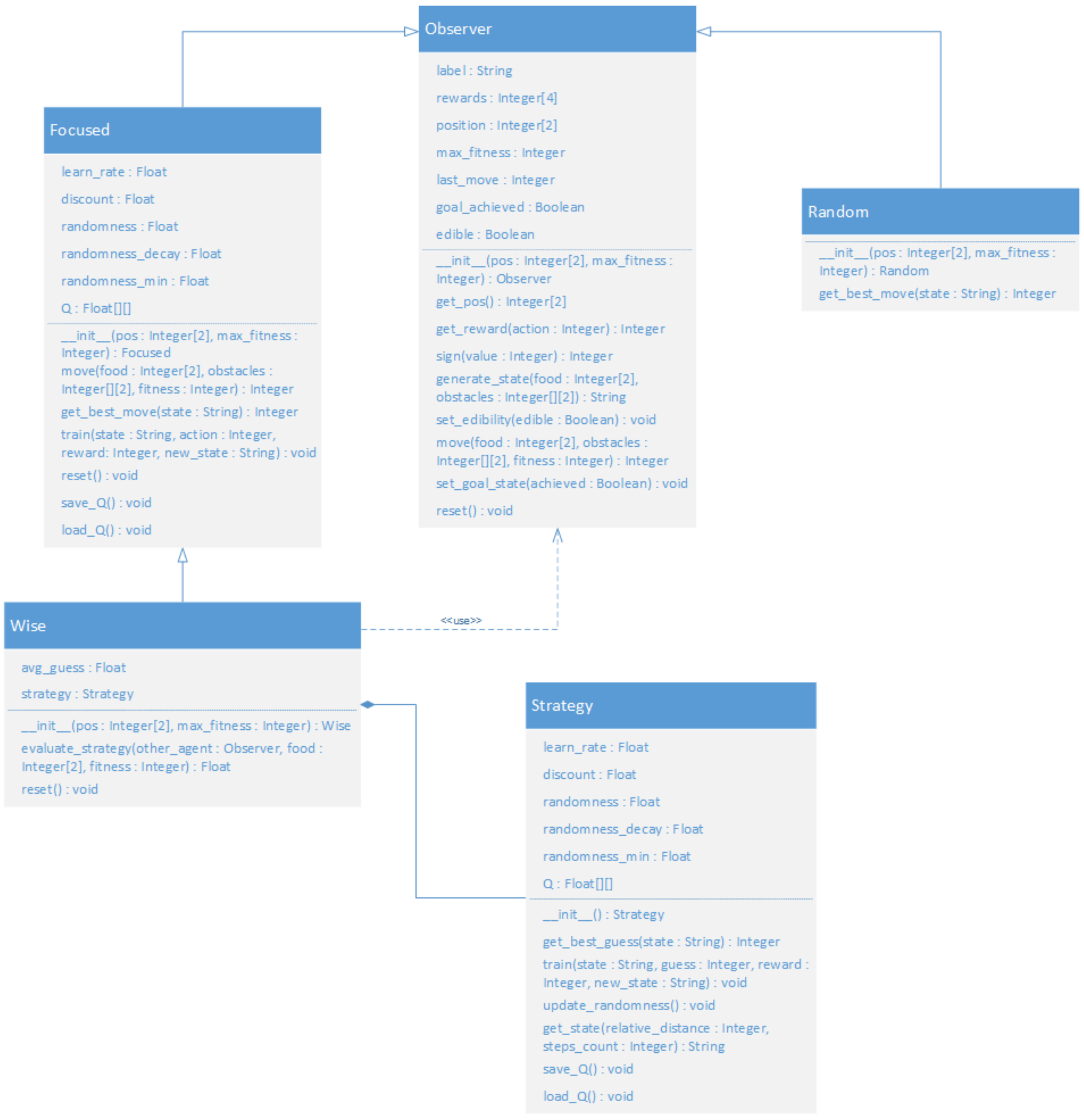

Observer
label : String
rewards : Integer[4]
position : Integer[2]
max_fitness : Integer
last_move : Integer
goal_achieved : Boolean
edible : Boolean
__init__(pos : Integer[2], max_fitness : Integer) : Observer
get_pos() : Integer[2]
get_reward(action : Integer) : Integer
sign(value : Integer) : Integer
generate_state(food : Integer[2], obstacles : Integer[][2]) : String
set_edibility(edible : Boolean) : void
move(food : Integer[2], obstacles : Integer[][2], fitness : Integer) : Integer
set_goal_state(achieved : Boolean) : void
reset() : void
Focused
learn_rate : Float
discount : Float
randomness : Float
randomness_decay : Float
randomness_min : Float
Q : Float[][]
__init__(pos : Integer[2], max_fitness : Integer) : Focused
move(food : Integer[2], obstacles : Integer[][2], fitness : Integer) : Integer
get_best_move(state : String) : Integer
train(state : String, action : Integer, reward: Integer, new_state : String) : void
reset() : void
save_Q() : void
load_Q() : void
Random
__init__(pos : Integer[2], max_fitness : Integer) : Random
get_best_move(state : String) : Integer
Wise
avg_guess : Float
strategy : Strategy
__init__(pos : Integer[2], max_fitness : Integer) : Wise
evaluate_strategy(other_agent : Observer, food : Integer[2], fitness : Integer) : Float
reset() : void
<<use>>
Strategy
learn_rate : Float
discount : Float
randomness : Float
randomness_decay : Float
randomness_min : Float
Q : Float[][]
__init__() : Strategy
get_best_guess(state : String) : Integer
train(state : String, guess : Integer, reward : Integer, new_state : String) : void
update_randomness() : void
get_state(relative_distance : Integer, steps_count : Integer) : String
save_Q() : void
load_Q() : void

Figure 2.11 – Agent class diagram

The Observer class also contains information about rewarding agents for performing the corresponding actions. The rewards array is a named list that includes such actions as "step", "step back", "collision with an obstacle" and "achieving the goal". There is no division between achieving the overall goal and completing the game, since the goal depends entirely on the behavior of the partner. The position array, which is also a named list, stores the "x" and "y" coordinates of the object on the level map. The goal_achieved binary variable is set to true when the object is located close to the goal. It is necessary to exclude any actions on the part of the agent if the goal has already been achieved. The variable resets its value to false if the goal changes its position and the goal achievement condition is no longer met. The class has a constructor that accepts the initial position of the object on the level map and a limit on the maximum number of steps. The sign function is an auxiliary one and returns the sign of the value number (1 if positive, -1 if negative, otherwise 0). The generate_state method converts the coordinates of the object, food, and obstacles into a string representing the current state of the system and suitable for indexing the Q-table by rows. The move function is responsible for moving the object around the level, issuing rewards, and processing global events in the system. It checks the correctness of the object's movement: its collision with an obstacle, going beyond the level boundaries, exceeding the maximum number of steps allowed per game; handles the agent's return to the previous position (a step back) and calls the Q-table update function for agents that need it. The reset function resets the goal_achieved variable.

The Focused class is based on the abstract Observer class and represents a goal-oriented agent. Its variables include constants required for Q-learning: learning rates (learn_rate) and discount rates (discount), randomness (randomness), minimum randomness threshold (randomness_min), and randomness decay factor (randomness_decay). The most important variable is the Q-table itself, which is a two-dimensional named list. The class constructor accepts the same values as the parent constructor and initializes the variables by filling the entire Q-table with zeros. The move function adds conditions for refusing to take actions: if the goal is unachievable or if the goal has already been reached. If these conditions are not met, the parent move function is called. The get_best_move function was added, returning either the best action based on the Q-table values, or a random one depending on the randomness variable. The train function updates the value of the Q-table cell located at the intersection of state and action (input parameters of the function) in accordance with the Bellman equation. As mentioned earlier, the reset function has been supplemented with the functionality of reducing the randomness of the agent's behavior. The save_Q and load_Q methods are used to write and read the file with the Q-table, respectively.

The second basic type of agent is an agent with random behavior(Random class). Its constructor does not initialize variables, but immediately calls the constructor of the Observer class. The get_best_move function finds a subset of actions that do not immediately end the game in the current state of the system. Then a random action is selected from this subset.

The most important class is the Wise agent class, which inherits the functionality of the Focused agent and analyzes the behavior of any object (Observer). This class adds two variables: avg_guess, which reflects the confidence in the partner's focus averaged over all of his actions performed during one episode, and strategy, a class responsible for analyzing behavior. In the constructor, the avg_guess value is set to 1 (complete confidence in the partner's readiness to achieve the common goal) and an object of the Strategy class is initialized, the constructor of which has no input parameters. In the evaluate_strategy function, the partner's behavior is evaluated and the result is returned: 0 if, in the algorithm's opinion, the partner acts randomly and is not ready to cooperate, or 1 otherwise. To minimize algorithm errors, this binary value is entered into the avg_guess variable in a ratio of 1 to 4, thereby slightly changing the overall confidence of the system. Returning to Figure 2.8, the "update target based on [teammate's behavior] analysis" block selects an entity as a target if the avg_guess variable exceeds the threshold of 0.5, otherwise the target becomes the coordinates of the premature termination point. The reset function reduces the randomness of the Strategy class object (since it is also based on the Q-learning method) and resets the avg_guess value to 1, since the behavior of the agent being evaluated may change in the next episode.

The source code and class hierarchy of agents are given in Appendix A., Listing A.2. This code should be placed in a file named agents.py, since the code in Listing A.1 contains references to this file.

ClassStrategy has a structure similar to the Focused class (excluding the Observer class). The get_best_guess function is functionally equivalent to get_best_move. The get_state method accepts the distance from the analyzed agent to the target in the L1 metric and the number of the group to which the number of actions performed since the beginning of the current episode belongs. The string returned by the get_state method is used to index the corresponding Q-table by rows. The source code for the Strategy class is given in Appendix A, Listing A.3. The class should be located in the strategy.py file, since Listing A.2 contains references to it.

ClassDesign has nothing to do with processing actions in a virtual environment, but is used only for the convenience of presenting the results of the program's work. It will be discussed in the corresponding section.

Only one Wise class was moved to the testing environment (Malmo project), renamed to CustomAgent, since the goal-oriented agent and the agent with random behavior are provided by default by the modification. The modified hierarchy and structure of the agent classes is shown in Figure 2.12.

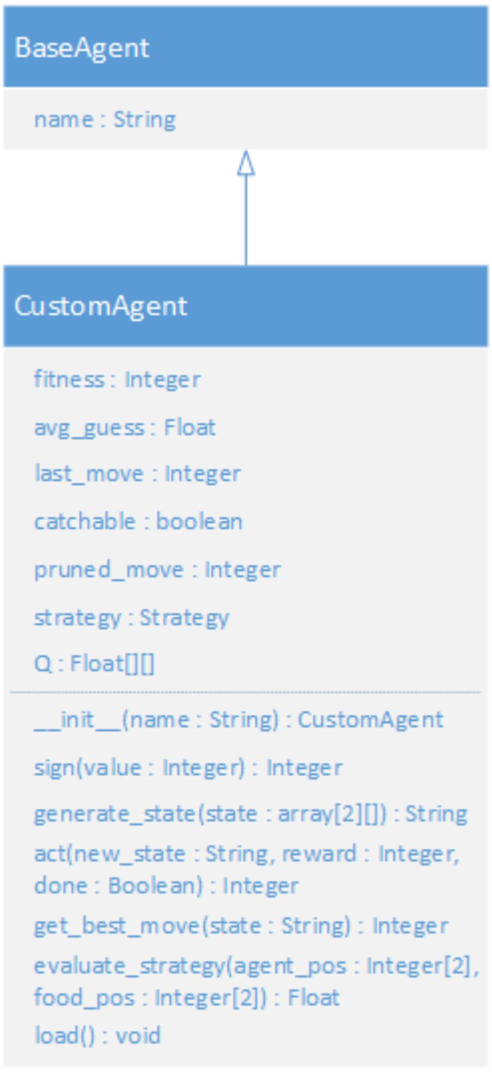

BaseAgent
name : String
CustomAgent
fitness : Integer
avg_guess : Float
last_move : Integer
catchable : boolean
pruned_move : Integer
strategy : Strategy
Q : Float[][]
__init__(name : String) : CustomAgent
sign(value : Integer) : Integer
generate_state(state : array[2][]) : String
act(new_state : String, reward : Integer, done : Boolean) : Integer
get_best_move(state : String) : Integer
evaluate_strategy(agent_pos : Integer[2], food_pos : Integer[2]) : Float
load() : void

Figure 2.12 – Hierarchy of agent classes in the Malmo project

BaseAgent is an abstract (base) agent class in the Malmo project environment, similar to the Observer class in the simplified environment. In Figure 2.12, only one variable belonging to this class is listed: name, since this is the only inherited variable used in CustomAgent.

The contents of the constructor and the sign, evaluate_strategy, and load (analogous to load_Q) functions are no different from the contents of the corresponding functions of the Wise class. The generate_state function includes some actions, presented as blocks in Figure 2.8. Its only input parameter, state, contains the positions of all objects on the map and the level map itself. The entity's position is extracted from it and a check is made for the possibility of catching it (the result is written to the catchable variable, previously called edible). Then the evaluate_strategy function is called, updating the avg_guess variable. This is followed by an innovation: pruning of invalid actions. Invalid actions are those that lead to an early end to the game if the agent is not yet ready to give up. In other words, a subset of actions is formed that will not lead the agent to the coordinates of the end point of the game if the agent believes that its partner is ready to cooperate. Since there is only one end point of the game, the number of pruned actions cannot exceed 1. The found action is stored in the pruned_move variable. Further functionality of generate_state repeats the generate_state method of the Observer class. The act function, just like the move function from the simplified environment, returns the identifier of the action performed by the agent. However, its content is very different. The input parameter done is checked, signaling the end of the current episode and requiring resetting all parameters to default values (similar to the reset function). Then the possibility of catching the entity is checked. If this is not possible, the agent remains in its place, otherwise it accesses the Q-table to find the best action. Before exiting the function, the current action is saved in last_move and fitness is incremented. Inside the get_best_move function, the deletion of the pruned action (whose identifier is contained in pruned_move) is added before comparing the options in the Q-table. The list of rewards, like the entire part responsible for training the model, was cut out of the CustomAgent class.

From the classStrategy also had the model training part cut out, but the remaining structure was not changed at all, since the CustomAgent class is the only one interacting with it.

Agent Action Listincreased by one element: the "inaction" action was added, since in this environment the agent does not have the ability to simply "skip a turn".

2.4. Conclusion on the section

In this paper, two models were developed and trained: a model for navigation of an AI agent on a two-dimensional level with obstacles and a model for classifying the behavior of a partner. Both models are based on the Q-learning method, since they require a small amount of discrete input and output data, which allows storing all possible states of the system in memory as a Q-table.

Despite the relatively small number of possible system states, the model requires optimization, since the training should only end when the AI agent can choose the most optimal action for each state of the system. In other words, the more states the system has, the longer it takes to reach the optimal values in the Q-table. The behavior classification model was optimized by categorizing the number of steps taken by the analyzed agent since the beginning of the current game.

Thus, it is necessary to use the simplest methods of managing AI agents in collaborative systems to increase controllability, transparency and, accordingly, predictability.

3. DEVELOPMENT RESULTS

In this section, the results of the algorithm's operation will be presented both in a simplified environment and in a testing environment, since the forms of presentation of the results in these environments differ.

Participants in the Malmo Challenge provide their results in the form of a demonstration of their algorithm in two scenarios: with a random partner and a goal-oriented partner. [5] This demonstration is usually enough to show that the algorithm works. Some participants also demonstrate a third type of collaboration: a human with an agent analyzing their behavior.

The participants who posted their solution in [6] abstracted from the environment provided by the Malmo project and used deep learning methods. “Deep learning is a set of machine learning methods (supervised, semi-supervised, unsupervised, and reinforcement) based on representation learning rather than task-specific algorithms.” [14] The input to their algorithm is a set of pixels obtained directly from a window with a 3D representation of the virtual environment. In other words, “through the eyes of the agent”. This approach allowed the participants to imagine the results of the algorithm from a variety of perspectives: the accuracy of the environment analysis, the average reward for each episode, the average reward for each step across all episodes, etc. Here, the accuracy is determined by the sum of the errors made during one episode.

Despite the fact that it is impossible to conduct a direct comparison of the results with other participants due to changes in the rules of the original competitions and different time constraints, the results of the VKR algorithm will be presented in the most detailed form, combining various methods of presenting the results by the participants who took prize places.

3.1. Experimental methodology

The experiment in a simplified environment is carried out according to the method presented in the form of an algorithm in Figure 2.8. This algorithm supports two modes of operation: training a model with subsequent testing and testing a ready-made Q-table. Since the model of the agent's movement along the level plays an important, but secondary role, only the model of the partner's behavior analysis is trained. To ensure that the training is sufficient, the testing period begins with the 200,000th step of the agent, counting from the start of the program. The most important result provided by the program interface is the proportion of correct determinations of the partner's intentions, which is in the interval [0, 1]. The proportion is calculated and averaged over all played episodes according to the following formula:

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

Where wins is the number of episodes in which the agent correctly classified the behavior of its partner, and loses is the number of episodes containing errors in the agent's determination of intentions. The values of these variables are updated in Figure 2.9 in the blocks "Increase in the model correctness counter value" (wins) and "Increase in the model incorrectness counter value" (loses), respectively.

In addition to the proportion of episodes with correct intent determination, the program provides a graphical user interface.It allows you to observe the gameplay in real time and compare your understanding of the "partner's" behavior with the understanding of the analyzing agent. The structure of the class containing the GUI elements is shown in Figure 3.1.

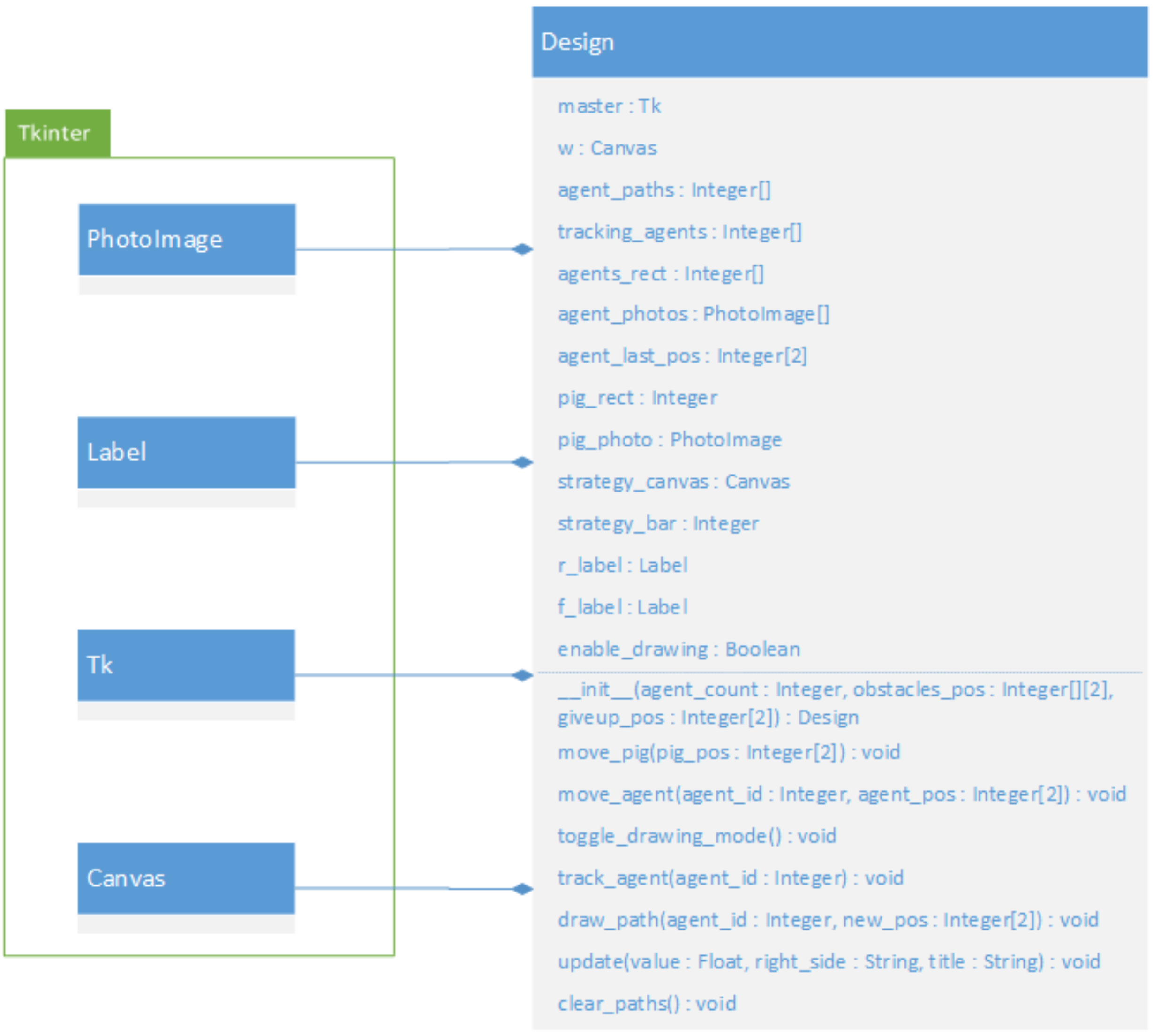

Tkinter
PhotoImage
Label
Tk
Canvas
Design
master : Tk
w : Canvas
agent_paths : Integer[]
tracking_agents : Integer[]
agents_rect : Integer[]
agent_photos : PhotoImage[]
agent_last_pos : Integer[2]
pig_rect : Integer
pig_photo : PhotoImage
strategy_canvas : Canvas
strategy_bar : Integer
r_label : Label
f_label : Label
enable_drawing : Boolean
__init__(agent_count : Integer, obstacles_pos : Integer[][2], giveup_pos : Integer[2]) : Design
move_pig(pig_pos : Integer[2]) : void
move_agent(agent_id : Integer, agent_pos : Integer[2]) : void
toggle_drawing_mode() : void
track_agent(agent_id : Integer) : void
draw_path(agent_id : Integer, new_pos : Integer[2]) : void
update(value : Float, right_side : String, title : String) : void
clear_paths() : void

Figure 3.1 – Structure of the Design class

The graphic library is usedTkinter for Python. "Tkinter (from the English Tk interface) is a cross-platform event-oriented graphical library based on Tk tools (widely used in the world of GNU/Linux and other UNIX-like systems, also ported to Microsoft Windows)" [15]. This library allows you to create simple GUIs suitable for presenting the results of a program.

The main component of the classThe design shown in Figure 3.1 is master, a top-level widget responsible for the parameters and content of the entire graphical window. The w and strategy_canvas widgets of the Canvas class contain most of the payload: the game field and the results of the agent's behavior analysis, respectively.

The field is a 10 by 10 grid along the abscissa and ordinate axes. Agents are located on the field, the identifiers of whose graphic elements are saved in the agents_rect variable. Agents have unique textures to make them stand out more against the background and obstacles. The textures are of the PhotoImage type provided by the Tkinter library and are located in agent_photos. The entity has similar characteristics and the corresponding variables: pig_rect and pig_photo. Obstacle elements are not saved in memory, since they remain static until the program is finished.

Drawing the movement of objects across the level is carried out through functionsmove_pig and move_agent respectively.

Beyond the level structure and the main characters on the canvasw shows the path taken by the analyzed agent. The entire agent path for the current episode is saved in agent_paths and cleared by calling the clear_paths method. The path consists of individual elements and is replenished with each agent movement. Moreover, for a better understanding of the dynamics of the object's movement, new elements of the path are highlighted in bright shades of color, while old elements fade over time. The functionality of drawing the path and changing the color is contained in the draw_path function. The user is given a choice of agents whose path should be tracked. An agent can be added by calling track_agent by passing the function the agent's ordinal number in the array of all system agents (except for the agent controlling the entity's movement).

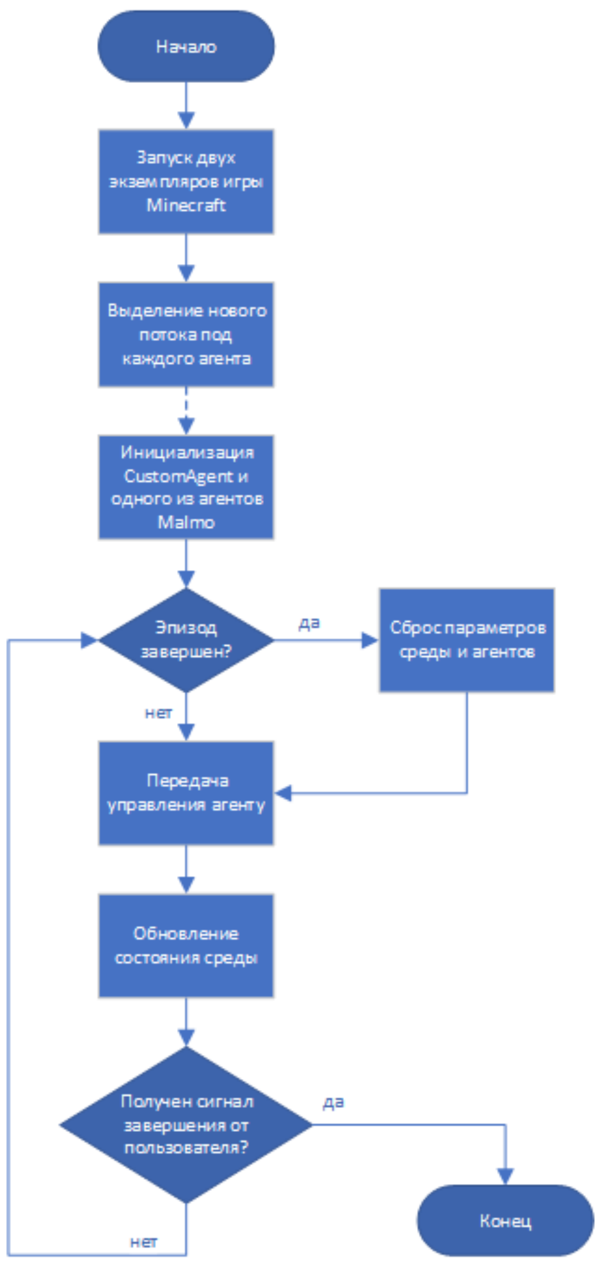
The next major component of the GUI is the strategy_canvas, which displays the agent's opinion of his partner's intentions. The strategy_bar is a status bar with a range of [0, 1]. It reflects the value of the avg_guess variable of the Wise class. On the left side of the bar is 0, labeled "R[andom]" (the r_label variable), and on the right is 1, labeled "F[ocused]" (the f_label variable). The current value of avg_guess is marked by a large dot that "slides" across the status bar throughout the episode.

It is worth noting that the graphic window is not updated whenchanging the state of the graphic elements, but only when calling the update function. The method is responsible for drawing all widgets in their updated state and maintaining the relevance of the window title containing the share of episodes with correctly defined intentions and the real class of the current analyzed agent.

Since the program supports model training, there must be a functionality to disable the GUI. This is contained in the functiontoggle_drawing_mode, which can freeze and unfreeze the entire GUI when called. The method takes no parameters, as it inverts the current freeze state each time it is called.

Class source codeDesign is given in Appendix A, Listing A.4. The class must be in the design.py file, since Listing A.1 contains references to it.

The algorithm for testing agents in the Malmo environment is shown as a flowchart in Figure 3.2. Before the testing procedure, it is necessary to ensure a sufficient number of running instances of the Minecraft game. In the case of a final qualification round, there must be at least two of them – one for each agent. The missing instances are checked and launched using a blocking operation in the "Launching two instances of the Minecraft game" block. As soon as the instances become ready, the program allocates a new thread for each agent. This is necessary because, unlike the simplified environment, in the Minecraft game all processes occur simultaneously, not step by step. All subsequent actions shown in Figure 3.2 occur simultaneously in two threads. At the very beginning of the thread function, the current agent is initialized: either an agent of the CustomAgent class (Figure 2.12), or one of the agents provided by the Malmo environment (at the user's choice). It is also possible to select a second instance of the CustomAgent class.



Начало
Запуск двух экземпляров игры Minecraft
Выделение нового потока под каждого агента
Инициализация CustomAgent и одного из агентов Malmo
Эпизод завершен?
да
Сброс параметров среды и агентов
нет
Передача управления агенту
Обновление состояния среды
Получен сигнал завершения от пользователя?
да
Конец
нет

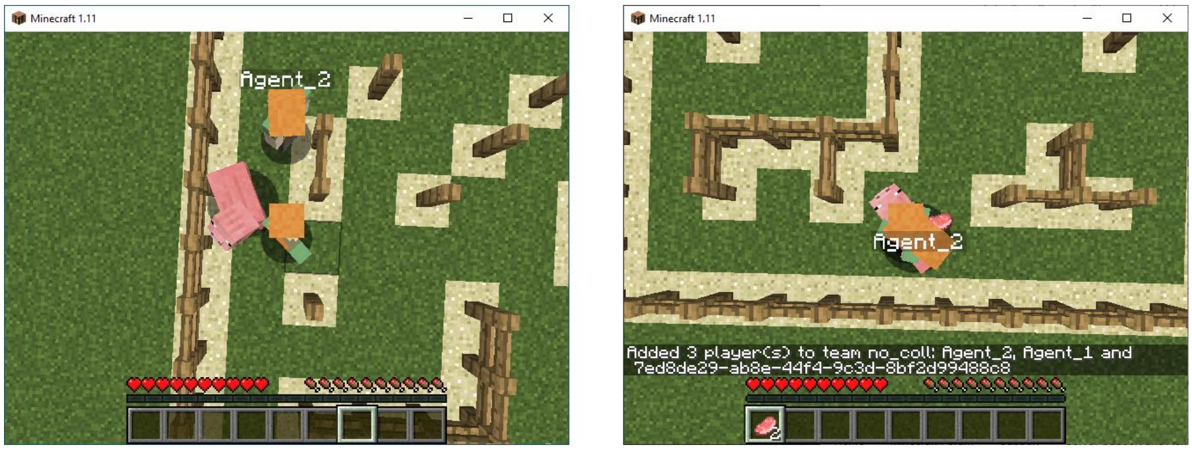
Figure 3.2 – Algorithm for testing agents in the Malmo environment

Then the main infinite cycle of updating the environment state begins until the user independently terminates the program execution. The "Transfer control to the agent" block calls the act function of the agent class belonging to the corresponding thread. The testing stage ends when the Ctrl+C key combination is sent to the program console.

Game environmentMinecraft introduces its own challenges that are absent when training in a simplified environment. One of the main obstacles is the collision of objects located at the same coordinates. In other words, objects cannot pass through each other, as they did during the model training stage. In practice, situations often arise when one of the agents, located between several obstacles, leads to the creation of a full-fledged labyrinth on the level map. Unfortunately, the model based on Q-learning is not able to pass dynamic (changing over time) labyrinths, since this contradicts the requirement of convergence of the algorithm. For best results, it is necessary to disable the processing of object collisions. To do this, simply enter the following commands in the Minecraft game window, pressing Enter after entering the text from each item:

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

This method disables collision handling for all objects currently in the virtual environment. You must re-enter command #3 if the set of objects changes since the last time it was executed. To ensure that the commands were executed successfully, the left side of Figure 3.3 shows collision handling enabled, and the right side shows it disabled.



```
Added 3 player(s) to team no_coll: Agent_2, Agent_1 and
 7ed8de29-ab8e-44f4-9c3d-8bf2d99488c8
```

Figure 3.3 – Result of disabling collision handling

In both cases, the model was tested using the level shown in Figure 2.2. The level structure includes 25 obstacles (excluding the level boundaries) that form a sparse labyrinth. It was for this type of level that the agent navigation algorithm was created. The level has several features: a recess, shown in Figure 2.5, and the absence of obstacles adjacent to the edges of the level. The first is necessary for correct processing and testing of the agent's return to the previous location, the second is for optimizing the search for a path to the goal.

3.2. Experimental results

The result of the implementation of the GUI for a simplified environment is presented in Figure 3.4.

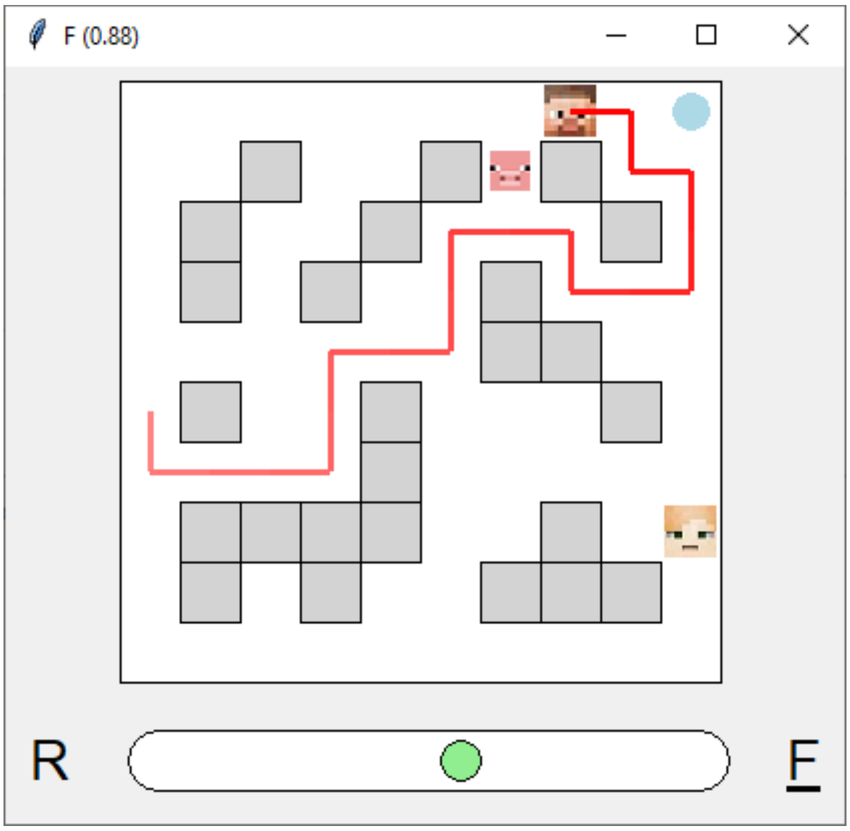


Figure 3.4 – GUI for a simplified environment

Based on the designation "F" present in the window title (upper left corner), it can be seen that the analyzed agent is currently ready to cooperate. There you can also see the calculated share of episodes in which intentions were determined correctly. In Figure 3.4, the share is 0.88 of the total number of episodes, but in practice its value may fluctuate. On the game field itself, you can see the path taken by the analyzed agent, which, given the possibility of moving the entity, is practically the shortest. The status line shows the confidence of the analyzing agent in the purposefulness of its partner. The green color of the dot confirms the correctness of the agent's conclusion.

An example of an agent with random behavior is shown in Figure 3.5. This time, the dot in the status bar is shifted to the left, which allows us to conclude that the partner is not ready to cooperate. Therefore, the analyzing agent moves to the blue dot to prematurely end the game.

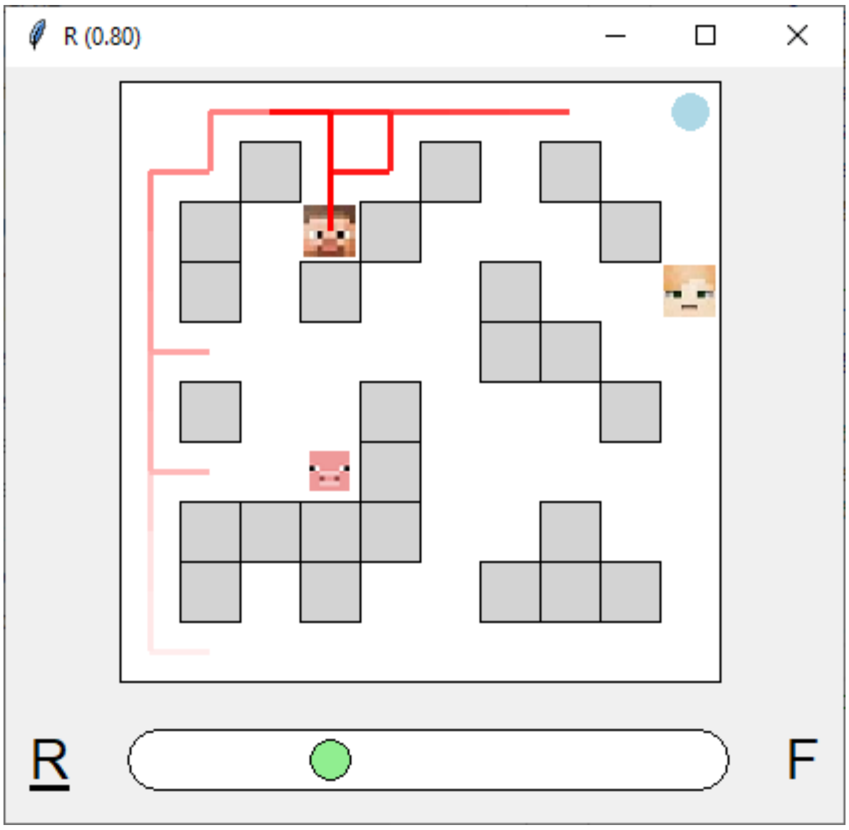


Figure 3.5 – GUI displaying an agent with random behavior

Getting the correct intent detection rate for episodes in the Malmo environment is much more difficult, as it takes much longer to run 100,000 episodes. Instead, here are two screenshots showing an entity being captured and a premature game termination, respectively.

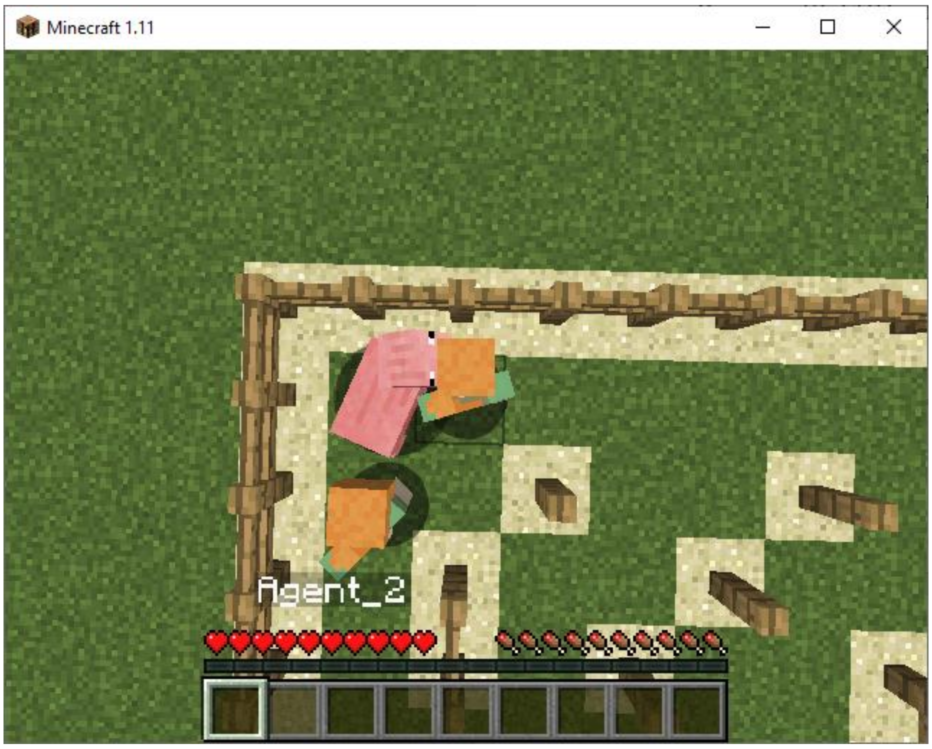


Figure 3.6 – Agent cooperation in the Malmo environment

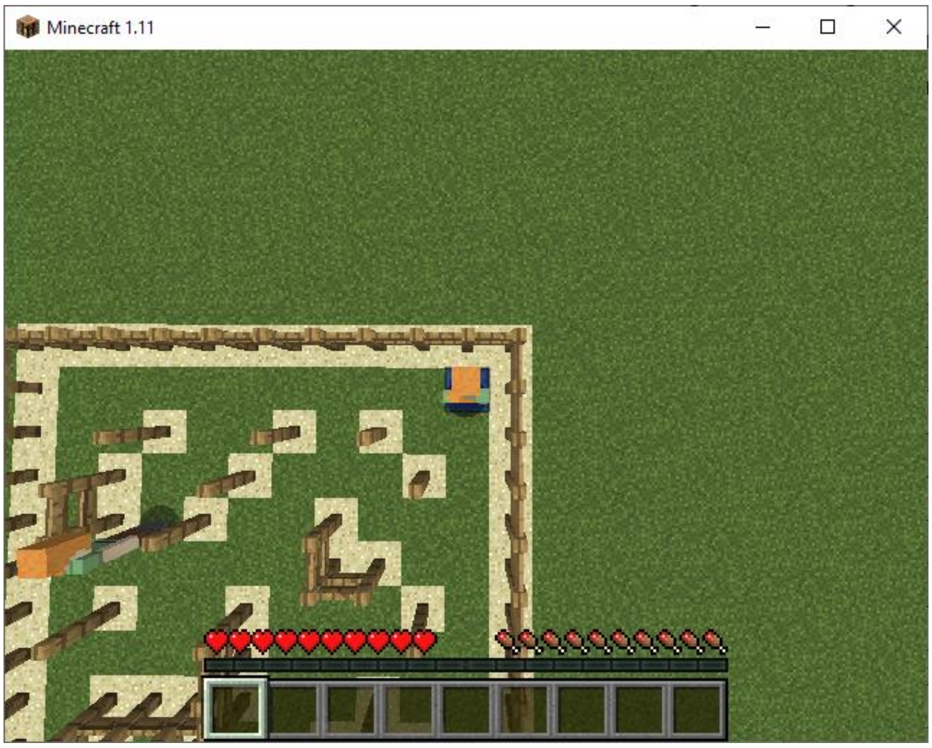


Figure 3.7 – Premature termination of the game in the Malmo environment

It is pointless to conduct such an analysis of the agent’s navigation algorithm by level, since, if we do not take into account the rareand without the agent getting into cycles, Q-learning is designed to solve exactly such problems, and the agent is able to reach the goal in any part of the level from any given point, bypassing all obstacles. However, the statistics of distances to the goal and successful completion of games for each step of the agent are shown in Figures 2.6 and 2.7, respectively.

3.3. Evaluation of results

The accuracy rate of 0.88 in Figure 3.4 is high enough for a model invariant to the dynamics of the analyzed agent's movement. However, even when averaging the rate over 100,000 games, there is a dependence on the quality of the first training episodes of the model. For example, in Figure 3.5, the same accuracy is 0.8, although the number of episodes that contributed to the formation of the parameter value has not changed.

Including the path traveled by the agent being analyzed in the state of the environment can have a positive effect on the accuracy of the model.The path to the goal taken by the random agent in Figure 3.5 cannot be called the shortest. It has many self-intersections and is concentrated in one region of the level that does not contain the final goal, which greatly distinguishes it from the path of the goal-oriented agent in Figure 3.4.

With collision handling disabled in the Malmo environment, agents should show similar results, since the only difference between this environment and the simplified one is the entity behavior.

The behavior of a goal-oriented agent can be called optimal. The agent can reach any cell of the field from any position in 18 steps. At the same time, analyzing Figure 2.7, it successfully reaches the goal, changing its position, in 8-9 steps. Even if this path is not the shortest, the agent copes well with its task, because this algorithm is relevant for any level, the structure of which is a sparse labyrinth. Moreover, if all cells of the Q-table have reached optimal values, the same model can control the agent at different levels without the need for retraining.

In Figure 2.6, there is a sharp increase in the distance from the agent to the goal after step 20. However, based on Figure 2.7, it can be seen that most games end before step 33. It can be concluded that, most likely, agents that do not finish the game before step 33 enter the cycle and remain in it until the end of the episode. Let us apply Fourier analysis to the values of the graph in Figure 2.6, limiting ourselves to the range [245, 500]. The result is shown in Figure 3.8.

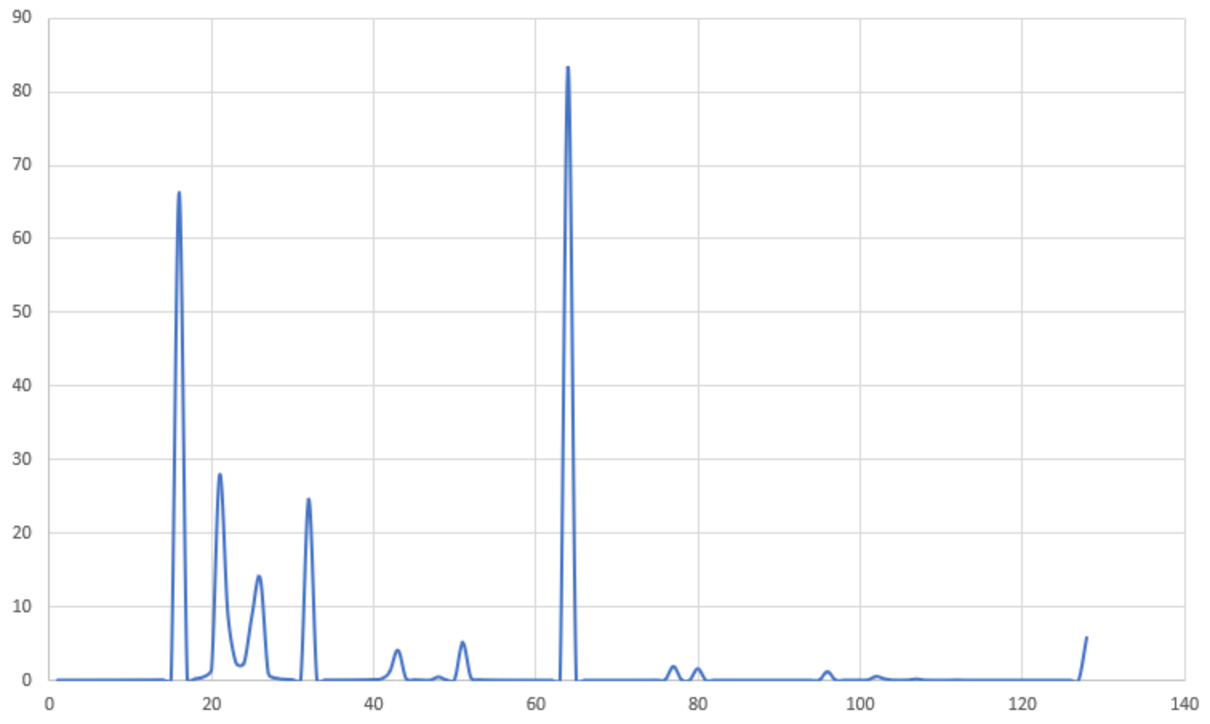


Figure 3.8 – Fourier analysis to determine cyclic motions

The graph clearly shows several extremes, which increases the probability of cyclic movements in the agent's behavior. Since agents that are not in a cycle or have escaped from it rarely require more than 33 steps to successfully complete the game, it can be concluded that the information in the graph in the range [33, 500] is provided only by agents that have entered a cycle. This explains the sharp increase in distance in the graph. However, this problem is not critical, since it occurs extremely rarely, since the movement of an entity over time can take agents out of cycles.

3.4. Conclusion on the section

According to the testing results, the navigation model copes well with the task, while the binary behavior classification model, despite the accuracy of 90%, in some situations can incorrectly interpret the behavior of a goal-directed agent, for example, if the agent has not chosen the shortest path to the goal. At the same time, the intent analysis model relies on the agent navigation model during training.

However, even when trained on the basis of the navigation algorithm in a simplified environment, the model is able to correctly classify the behavior of different agents in the Minecraft game environment.

Thus, both models turned out to be of sufficient quality for use in low-responsibility systems that require collaboration of AI agents with minimal communication between them. However, adding the analysis of the dynamics of agent movement over time to the model can improve accuracy and allow the possibility of embedding the model in more important systems.

4. FEASIBILITY STUDY

4.1.Introduction

In this thesis, an algorithm for the collaboration of AI agents to achieve a common team goal was developed. The share of AI in the market is rapidly growing, which means that it is necessary to continue studying various approaches and algorithms to increase the productivity and safety of AI systems. The developed algorithm will help establish communication between different systems, in particular those whose main task is related to movement on the surface (for example, cargo delivery or territory exploration). The algorithm is also capable of identifying malfunctions of systems with which collaboration is required based on their behavior, which allows operational systems to change the action plan directly during its execution.

This section contains the calculation of costs required for the sale of the product, wages of workers and other types of necessary deductions. At the end of the section, a conclusion is made about the need to introduce this product to the market.

4.2.Evaluation of labor intensity

The labor intensity is calculated in man-days spent on the completion and assessment of the work. The final qualifying work was divided into parts, the names and approximate completion dates of which are given in the table on page 3. The labor intensity assessment is made for both the student and the supervisor. Since each part has a minimum and maximum permissible completion date, to calculate its estimated duration, a weighted average of these two values is calculated using the formula

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

Where is the expected duration of work, and are the minimum and maximum permissible durations, respectively.$t_j^0 t_{min} t_{max}$

Thus, in the table4.1 provides calculations of the expected duration for each part of the work.

Table 4.1. – Expected duration of each part of the work

Item No.	Name of works	Duration of work		
		t_{min}	t_{max}	t_j^0
1	Review of literature on the topic of the work	14	21	16.8
2	Program development	62	72	66
3	Testing the program	2	7	4
4	Checking the functionality of the program by the manager	1	1	1
5	Fixing bugs and adding functionality where needed	8	14	10.4
6	Cleaning the program code	2	5	3.2
7	Preparation of an explanatory note	10	20	14
8	Evaluation of the explanatory note by the manager	2	7	4
9	Design of illustrative material	5	7	5.8

4.3. Cost determination

The list of costs for manufacturing a product includes employee wages, mandatory deductions, depreciation charges, equipment operation, third-party services and consumables.

4.3.1 Determining wage costs

The average monthly salary of a programmer trainee is 40,000 rubles before tax, and the salary of a scientific supervisor is 52,270 rubles/month. Since the duration of work in Table 4.1 is measured in days, it is necessary to divide these values by 21 working days. This gives 1,905 rubles/day and 2,489 rubles/day, respectively.

Based on the data in Table 4.1, the costs of basic wages are calculated using the formula:

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

WhereT_i– the total duration of work of the i-th performer in days, – the salary of the i-th performer in rubles/day, k – the number of performers (in this case 2).C_i

The costs of additional wages are calculated using the formula

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

Where is the standard for additional wages, adopted at 14%.$H_{\text{доп}}$

4.3.2 Determining the costs of mandatory deductions

Mandatory deductions are necessary to ensure social, pension and health insurance. They are calculated on the basis of the basic and additional wages according to the formula

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

Where is the standard for deductions for insurance premiums for compulsory social, pension and medical insurance, adopted at 30%, is deductions from wages for social needs.$H_{\text{доп}}З_{\text{соц}}$

4.3.3 Determination of depreciation costs

Before calculating the costs of depreciation charges, it is necessary to list the funds and software used in this final qualifying work.

Table 4.2. – Tools and software used in this final qualifying work

Resources	Quantity, pcs.	Price, RUB	Amount, RUB
Lenovo laptop	1	62990.00	62990.00
HP DeskJet Printer	1	6390.00	6390.00
Windows 10 Home	1	4937.00	4937.00
Microsoft Office	1	1254.00	1254.00
Total			75571.00

Depreciation charges are calculated using the following formula:

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

Where$Ц_{\text{п.н.}i}$– the initial cost of the i-th asset, – the annual depreciation rate of the i-th asset as a percentage, calculated using the formulaH_{ai}

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

WhereN_i– the useful life of the i-th asset in years. For a Lenovo laptop, this period is 5 years, for an HP DeskJet inkjet printer – 4 years. The concept of depreciation charges is not applicable to software. However, writing a final qualifying work takes less than 2 years, therefore, it is necessary to calculate depreciation charges for the period of their actual use using the formula

$$\text{А}_{i\text{ВКР}} = \text{А}_i * \frac{T_{i\text{ВКР}}}{12}$$

Where$T_{i\text{ВКР}}$– the period of using the tool for writing the final qualifying work.

Table 4.3. – Depreciation charges

Means	Initial cost, RUB	Annual depreciation rate, %	Depreciation charges for 1 year, RUB.	Depreciation charges for the period of writing the final qualifying work, RUB.
Lenovo laptop	62990	20.0	12598	6299
HP DeskJet Printer	6390	25.0	1598	133
Total				6432

4.3.4 Determining the cost of operating equipment

The category of equipment operation includes the consumption of electricity by a laptop and printer, the cost of which is calculated using the formula

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

Where k is the number of devices (in this case 2), is the electricity consumption of the i-th device in kW*h, is the total operating time of the i-th device in hours, is the electricity cost tariff, equal to 5.7 rubles/kW*h in the city of St. Petersburg for 2023.$P_i t_i C$

The printer was actively used for an average of 2 hours during its entire operation, and the laptop was used for 756 hours (6 hours each working day from November to May).

4.3.5 Costs of third party services

To conduct research and write a program, you need Internet access, the cost of which is 500 rubles/month for both the programmer and the manager. The costs of using the services of third-party organizations are calculated using the formula

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

Where k is the number of services (in this case 2), is the cost of the i-th service, is the period of use of the i-th service, VAT is the value added tax, equal to 20% in 2023.$q_i t_i$

4.3.6 Costs of consumables

The cost of consumables is calculated using the formula

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

Where k is the number of consumables,G_i– the consumption rate of the i-th material per unit of production in pieces, – the cost of the i-th material in rubles, – the rate of transportation and procurement costs, taken as 10%. The consumables used are listed in Table 4.4.$C_i H_{\text{т.з}}$

Table2.4. – Consumables and their cost

Material	G_i, pcs.	Cost, RUB	Costs, RUB
Paper (500 sheets)	1	387	425.7
HP 3YM60AE Cartridge	4	1142	5024.8
Total			5450,5

4.4. Project Cost Estimation

The total cost of the product consists of all the costs calculated in section 4.3. All costs and the total amount are shown in table 4.5.

Table 4.5. – Cost of product development

Cost	Amount, RUB
Basic salary	241426,00
Additional salary	33799,64

Mandatory deductions	82567,69
Depreciation charges	6432,00
Purchase of special equipment	62990,00
Operation of equipment	478.57
Third party services	4800,00
Consumables	5450,50
Total	437944,40

The most expensive component of development is the basic salary.

4.5. Conclusion on the section

The cost of developing an AI agent collaboration system is 437,944 rubles, which makes it affordable for almost any business that actively uses AI in its work. This system will be able to take over some of the human work, which will reduce the burden on people monitoring the state and behavior of AI agents. Even if part of the systems fails, the agents will signal the incident and try to continue working independently, generating a new action plan.

CONCLUSION

In this thesis, in accordance with its objective, a program was developed to recognize the behavior strategies of AI agents and a mechanism for their navigation on a plane was created to achieve a common goal in a multi-agent environment. The goal was successfully achieved - the developed agent is able to navigate on a plane with obstacles, which are a sparse labyrinth, and determine the intentions of its partner with an accuracy of up to 90%. The implementation of this work showed the high complexity of creating AI agent collaboration systems even for solving such a simple task. This explains the currently low prevalence of such systems for business applications.

Systems for coordinating two or more AI agents are widely used today, and their use is predicted to be even more successful in the future, especially in areas with a branched structure of autonomous entities, large amounts of data, high dynamics, and poor predictability of behavior. Examples include managing a large number of local structures within a large organization, self-organizing groups of unmanned vehicles or aircraft, or synchronized operation of small power plants in distributed power grids. For example, agents can play the role of material deliverers who need to meet at a given point, having its coordinates. One agent will hand over the materials to another so that the second can continue delivering them. If one of the agents fails, the second will be able to analyze the behavior of the first and signal a failure in the system without reaching the designated meeting place. Another example: both agents are flying drones exploring the territory for electromagnetic guns or other dangers. Each agent has a point on the map where they must end up after completing the study. If one of the agents observes any strange behavior in his partner, such as a deviation from the course or a complete stop, he interrupts the investigation procedure and returns to the base.

To further advance this development, it is necessary to consider models that analyze the agent's behavior in more detail. It is necessary to take into account the "entanglement" of their path: the number of self-intersections, curvature taking into account obstacles, etc. It is also worth paying attention to time analysis– observe changes in the agent's behavior throughout the game. The ability to change the goal of any agent directly during the plan execution can play an important role. At the moment, only the agent performing the behavior analysis can change its goal during the same episode. This is a rather strong limitation of the current work.

Most likely, in the future, multi-agent systemsextend to all kinds of devices (a set of smart Internet things) that have sensors and are capable of influencing various objects in the outside world. Such devices will be able to not only process signals following certain instructions, but also independently make decisions and build action plans.

LIST OF USED SOURCES

1. John McCarthy. What is artificial intelligence? // Stanford, CA 94305. – 2007. – November 12 [Electronic resource]. URL:https://www-formal.stanford.edu/jmc/whatisai.pdf(date accessed: 09.05.2023).
2. Mark Stefik. Roots and Requirements for Collaborative AI // Palo Alto Research Center. – 2023. – March 17 [Electronic resource]. URL:https://arxiv.org/ftp/arxiv/papers/2303/2303.12040.pdf (date accessed: 09.05.2023).
3. Jason Lee. Teaching a computer how to play Snake with Q-Learning // Towards Data Science. – 2020. – July 23 [Electronic resource]. URL:https://towardsdatascience.com/teaching-a-computer-how-to-play-snake-with-q-learning-93d0a316ddc0 (date accessed: 09.05.2023).
4. Alex Maszański. What is reinforcement learning and how does it work. We explain with simple examples // proglib. – 2021. – December 24 [Electronic resource]. URL:https://proglib.io/p/chto-takoe-obuchenie-s-podkrepleniem-i-kak-ono-rabotaet-obyasnyaem-na-prostyh-primerah (date accessed: 09.05.2023).
5. Adrià Garriga-Alonso. The Malmo Collaborative AI Challenge // Github. – 2017. – May 23 [Electronic resource]. URL:https://github.com/rhaps0dy/malmo-challenge (date accessed: 09.05.2023).
6. Florin Gogianu. Malmo Challenge Overview // Github. – 2017. July 23 [Electronic resource]. URL:https://github.com/village-people/flying-pig (date accessed: 09.05.2023).
7. James Allen, George Ferguson. Human-machine collaborative planning // University of Rochester. – 2020. – June 5 [Electronic resource]. URL:https://www.researchgate.net/publication/228911168_Human-machine_collaborative_planning (date accessed: 09.05.2023).

8. Funtowicz Morgan, Vitaly Kurin, Katja Hofmann. Malmo Collaborative AI Challenge – Pig Chase // Github. – 2017. – May 11 [Electronic resource]. URL:https://github.com/microsoft/malmo-challenge/blob/master/ai_challenge/pig_chase/README.md(date accessed: 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 [Electronic resource]. URL:https://github.com/Microsoft/malmo(date accessed 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 [Electronic resource]. URL:https://www.ihmc.us/wp-content/uploads/2021/04/17.-Team-Players.pdf (date accessed: 09.05.2023).
11. Intelligent agent // Wikipedia. [Electronic resource]. URL:https://ru.wikipedia.org/wiki/Intelligent_agent (date accessed: 09.05.2023).
12. Temporal logic // Wikipedia. [Electronic resource]. URL:https://ru.wikipedia.org/wiki/Temporal_logic (date accessed: 09.05.2023).
13. Q-learning // Wikipedia. [Electronic resource]. URL:https://en.wikipedia.org/wiki/Q-learning (date accessed: 09.05.2023).
14. Deep learning // Wikipedia. [Electronic resource]. URL:https://ru.wikipedia.org/wiki/Deep_learning (date accessed: 09.05.2023).
15. Tkinter // Wikipedia. [Electronic resource]. URL:https://ru.wikipedia.org/wiki/Tkinter (date accessed: 09.05.2023).

APPENDIX A. PROGRAM SOURCE CODE LISTINGS

Listing A.1 – Source code of the main.py file

```
# Import agents from file agents.py (Listing A.2)
from agents import Focused, Random, Wise
from tkinter import *
# Import the Design class from the design.py file (Listing A.4)
from design import *
import random
import time

# Setting the number of agents in the system (1 or 2)
NUM_OF_AGENTS = 2
# Maximum number of steps allowed for each agent
MAX_FITNESS = 200
# The structure of the obstacles of a sparse maze in which there
is a path between any two cells
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}]
# Game Over Point
giveup_pos = {'x' : 9, 'y' : 0}

# Initialize the graphical user interface
design = Design(NUM_OF_AGENTS, obstacle_pos, giveup_pos)

# Random selection of a free cell on the field
def generate_object():
    global agents_brain
    # Get positions of all agents
    agents_pos = [agent.get_pos() for agent in agents_brain]
    obj_pos = {}
    while True:
        # Random selection of a cell by two coordinates
        obj_pos['x'] = int(9*random.random())
        obj_pos['y'] = int(9*random.random())
        # If the cell is not occupied by agents or obstacles
        if obj_pos not in agents_pos and obj_pos not in
obstacle_pos:# and edibility() >= 4-NUM_OF_AGENTS:
            return obj_pos

# Initialization of goal-oriented, wise (analyzing) agents and
agent with random behavior
# Since NUM_OF_AGENTS is set to 2, the system only takes into
account the first two agents
# When starting a new episode, either the first or third agent
from the list takes first place
```

```
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))
# Activate tracking of the path of the analyzed agent
design.track_agent(0)
# Generate initial target position
target_pos = generate_object()
# Drawing the target
design.move_pig(target_pos)

# Checking if the target is surrounded by obstacles
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)
    # If the target is surrounded by at least 2 obstacles, it
can be caught
    return obstacle_count > 1

caught_per_agent = [False] * NUM_OF_AGENTS
# Start a new game when the goal is reached
def catch(agent):
    global target_pos, caught_per_agent, fitness, steps,
target_brain, target_move_timer
    # Agents must be in different cells to capture the entity
    if NUM_OF_AGENTS > 1 and agents_brain[agent].get_pos() ==
agents_brain[1-agent].get_pos():
        return
    caught_per_agent[agent] = True
    # If every agent has caught the entity (entity is
surrounded), the episode restarts
    if not all(caught_per_agent):
        return
    # Random selection of agent for a new game (targeted or with
random behavior)
    agent_id = int(2*random.random())
    agents_brain[0], agents_brain[3] = agents_brain[2*agent_id],
agents_brain[2*(1-agent_id)]
    # Temporary freeze of the target at the beginning of the
game (for the first 5 steps of agents)
    target_move_timer = 5
    # Random selection of cells for placement of agents in a new
game
    for a in range(NUM_OF_AGENTS):
        agents_brain[a].position = generate_object()
    # Updating the target's position in the class responsible
for its movement
    target_brain.position = target_pos = generate_object()
    # Reset game progress
    fitness = 0
    caught_per_agent = [False] * NUM_OF_AGENTS
    # Draw the target in a new position
    design.move_pig(target_pos)
    # Clearing the path taken by the analyzed agent
    design.clear_paths()

# Restart the game
def restart_game():
    global caught_total, caught_per_agent
    # Reset game progress
    caught_total = -1
    caught_per_agent = [False] * NUM_OF_AGENTS
    for agent in range(NUM_OF_AGENTS):
        agents_brain[agent].reset()
        # Start a new game for each agent
        catch(agent)

# Start of the main program loop
steps = 0 # Total number of steps taken by each agent across all
episodes
fitness = 0 # Number of steps taken by each agent during the
current episode
game_loses = 1 # Number of episodes in which the agent
incorrectly determined the intentions of the partner
game_wins = 1 # Number of episodes in which the agent correctly
determined the intentions of the partner
target_brain = Random(target_pos, MAX_FITNESS) # Agent that
controls the movement of the entity
target_move_timer = 5 # Timer that allows/prohibits entities
from moving
while True:
    # Move target entity randomly
    if target_move_timer < 0:
        target_res = target_brain.move(target_pos, obstacle_pos,
fitness=0)
        target_pos = target_brain.get_pos()
        # If the target has moved, it is no longer caught
        for agent in range(NUM_OF_AGENTS):
            caught_per_agent[agent] = False
        design.move_pig(target_pos)

    # Updating the timer that allows/prohibits entities from
moving
    if target_move_timer == 0:
        # Maximum 5 steps of movement and 10 of rest
        target_move_timer = int(15 * random.random() - 5)
    else:
        target_move_timer -=
target_brain.sign(target_move_timer)

    # Updating the position of agents
    for agent in range(NUM_OF_AGENTS):
        # If the agent can analyze behavior, the analysis is
performed
        if hasattr(agents_brain[agent], "evaluate_strategy"):
            # Obtaining up-to-date data on a partner's behavior
in the form of probability
            avg_guess =
agents_brain[agent].evaluate_strategy(agents_brain[0],
target_pos, fitness)
            binary_guess = round(avg_guess)

            # Selecting the strategy of the analyzing agent in
accordance with the current data
            target = [giveup_pos, target_pos][binary_guess]
            agents_brain[agent].set_edibility([True,
edible()][binary_guess])

            # If the partner presumably does not want to
cooperate, forget about the essence
            if binary_guess == 0:
                caught_per_agent[agent] = False
        # If not, default parameters are set
        else:
            target = target_pos # target is an entity
            agents_brain[agent].set_edibility(edible())

        # Transfer control to the current agent
        result = agents_brain[agent].move(target, obstacle_pos,
fitness)

        # Render the agent in the new position
        design.move_agent(agent, agents_brain[agent].get_pos())

        # Check for global changes in the game
        if result == 1 and target == target_pos: # Current agent
has reached the target
            catch(agent)
        # The goal was achieved by all agents, or the agent
encountered an obstacle
        if result < 0 or all(caught_per_agent):
            # Update game counter according to rules
            game_wins += agents_brain[0].label != "R"
            game_loses += agents_brain[0].label == "R"
            restart_game()
        if agents_brain[agent].get_pos() == giveup_pos: #
Current agent has given up
            # Update game counter according to rules
            game_wins += agents_brain[0].label == "R"
            game_loses += agents_brain[0].label != "R"
            restart_game()

        # Updating goal achievement information in agent class

agents_brain[agent].set_goal_state(caught_per_agent[agent])
    # Updating graphic information in the application window
    design.update(avg_guess, agents_brain[0].label,
agents_brain[0].label + " (" + '{0:.2f}'.format(game_wins /
(game_wins + game_loses)) + ")")

    # If the goal is achievable in the current state of the
environment
    currently_edible = edible()

    # After step 199990 the agent starts rendering the GUI
    if steps == 199990:
        design.toggle_drawing_mode()
    # Every 10000 steps the training status of the model is
displayed
    if steps % 10000 == 0 and currently_edible:
        print("Wise agent made", steps, "steps")
    # Environment processes need to be slowed down so that the
user can monitor progress
    if 199990 < steps:
        time.sleep(0.1)

    # If agents do not move, the step counter does not change
    if currently_edible:
        steps += 1
        fitness += 1
```

Listing A.2 – Source code of the file agents.py

```
# Import the Strategy class from the strategy.py file (Listing
A.4)
from Strategy import Strategy, get_L1distance_from_target
from random import random
import json

# Base class of moving object
class Observer:
    label = "O" # Class label
    # List of rewards provided by the environment
    rewards = {'step' : -1, 'step_back' : -200, 'loss' : -1000,
'caught' : 50, 'stay' : -10}

    # The constructor takes the initial position of the object
and the maximum number of steps allowed per game
    def __init__(self, pos, max_fitness) -> None:
        self.position = pos
        self.max_fitness = max_fitness
        self.last_move = 0 # The previous action performed by
the agent
```

```
        self.goal_achieved = False # Whether the goal was
achieved (the agent is on the adjacent cell from the entity)
        self.edible = False # Is the goal achievable (is it
possible to catch the entity)

    # Agent position getter
    def get_pos(self):
        return self.position

    # Receiving the amount of the reward for the corresponding
action
    def get_reward(self, action):
        return self.rewards[action]

    # Getting the sign of a number
    def sign(self, value):
        if value < 0: return -1
        elif value > 0: return +1
        else: return 0

    # Formation of the system status line from the target and
obstacle positions
    def generate_state(self, target, obstacles):
        # Horizontal (-1,0,1) and vertical (-1,0,1) positions of
the target relative to the agent
        target_related = 3*self.sign(target['y']-
self.position['y']) + self.sign(target['x']-self.position['x'])
        # Presence of an obstacle to the left of the agent
        left_side = int({'x' : self.position['x']-1, 'y' :
self.position['y']} in obstacles)
        # Presence of an obstacle below the agent
        bottom_side = int({'x' : self.position['x'], 'y' :
self.position['y']+1} in obstacles)
        # Presence of an obstacle to the right of the agent
        right_side = int({'x' : self.position['x']+1, 'y' :
self.position['y']} in obstacles)
        # Presence of an obstacle above the agent
        upper_side = int({'x' : self.position['x'], 'y' :
self.position['y']-1} in obstacles)
        # Combination of obstacles on all sides
        obstacle_comb = (left_side << 3) + (bottom_side << 2) +
(right_side << 1) + upper_side
        # Combination of all states in one line
        return str(4 * (15*target_related + obstacle_comb) +
self.last_move)

    # Setting the reachability of a target from outside the
class
    def set_edibility(self, edible):
        self.edible = edible

    # Agent performing an action and receiving a reward
    def move(self, target, obstacles, fitness):
        # Current state of the environment
        state = self.generate_state(target, obstacles)
        # Getting the best or random agent move
        move = self.get_best_move(state)

        # Result of agent's action (not reward), neutral by
default
        result = 0

        # Getting new agent coordinates on the plane
        tempX = self.position['x'] + (move == 1) -(move == 3)
        tempY = self.position['y'] + (move == 2) -(move == 0)
        # Agent Position Update
        self.position = {'x' : tempX, 'y' : tempY}
        # If the agent has gone beyond the level, collided with
an obstacle or
        # exceeded the maximum number of steps allowed in one
episode
        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')
        # If the agent has reached the goal (is in that cell or
adjacent to it)
        elif abs(tempX-target['x']) + abs(tempY-target['y']) <=
1:
            result = 1 # Goal has been achieved
            reward = self.get_reward('caught')
        # Processing agent movement to previous position
        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')

        # Getting a new system state
        newState = self.generate_state(target, obstacles)
        # Train the model based on the last step and the reward
        self.train(state, move, reward, newState)
        # Update last step of agent
        self.last_move = move
        # -1 - agent violated the rules, 0 - state did not
change, 1 - agent achieved the goal
        return result

    # Setting the goal achievement state from outside the class
    def set_goal_state(self, achieved):
        self.goal_achieved = achieved
    # Reset game progress
    def reset(self):
        self.goal_achieved = False

# Goal-oriented agent class
class Focused(Observer):
    label = "F" # Class label

    def __init__(self, pos, max_fitness) -> None:
        super().__init__(pos, max_fitness)
        # Parameters for Q-learning
        self.learn_rate = 0.1
        self.discount = 0.9
        self.randomness = 0.0 # 0.9 for training
        self.randomness_decay = 1.3
        self.randomness_min = 0.00005

        # Loading a ready-made Q-table for a targeted agent
        self.load_Q()
        # To train the navigation model, you need to uncomment
the code and
        # run the program with one agent
        # Cannot be run in parallel with training of the intent
detection model,
        # this will not give the desired result
        #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):
        # If the goal is unachievable or has already been
achieved, the goal-directed agent should not move
        if self.goal_achieved == True or not self.edible:
            return 0
        # Call the parent agent move function
        return super().move(target, obstacles, fitness)

    # Get the best action according to Q-table or choose a
random one to explore the environment
    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())

    # The model training function updates the value in the Q-
table cell at the intersection of state and action
```

```
    def train(self, state, action, reward, newState):
        # The value is not updated if the action resulted in the
game ending
        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']))

    # Reducing randomness in agent behavior when training a
model
    def reset(self):
        super().reset()
        if self.randomness > self.randomness_min:
            self.randomness /= self.randomness_decay

    # Save Q-table to file
    def save_Q(self):
        open("Agent.json", "w").write(json.dumps(self.Q))

    # Load Q-table from file
    def load_Q(self):
        self.Q = json.load(open("Agent.json", "r"))

# Class of a "wise" agent that analyzes the behavior of the
Observer object
class Wise(Focused):
    label = "W" # Class label

    def __init__(self, pos, max_fitness) -> None:
        super().__init__(pos, max_fitness)
        # Before starting a new game, the partner is considered
as a target by default
        self.avg_guess = 1
        # Initialize the Strategy object to define intents
        self.strategy = Strategy()

    # Function of evaluating the partner's behavior
    def evaluate_strategy(self, other_agent, target_pos,
fitness):
        # Getting the distance from the agent to the target in
L1 metric
        dist_to_target =
get_L1distance_from_target(other_agent.get_pos(), target_pos)
        # Delay for more precise definition
        if fitness > 3:
            # Getting the current state of a partner
            strat_state =
self.strategy.get_state(dist_to_target, fitness)
            # Assessing his behavior
            strat_guess =
self.strategy.get_best_guess(strat_state)
```

```
            # A reward, the size of which depends on the
correctness of the behavior determination
            strat_reward = 1 if ["R", "F"][strat_guess] ==
other_agent.label else 0
            # Train a model based on the partner's state and
reward
            self.strategy.train(strat_state, strat_guess,
strat_reward, strat_state if strat_guess == 1 else
str(int(36*random())))
            # Update partner targeting probability
            self.avg_guess = 0.80 * self.avg_guess + 0.20 *
strat_guess
        # If the partner is suspected of having random behavior,
reset the goal achievement state
        if round(self.avg_guess) == 0:
            self.goal_achieved = False
        # Return the new probability value
        return self.avg_guess

    # Reducing the randomness of training an intention detection
model
    def reset(self):
        super().reset()
        self.strategy.update_randomness()
        self.avg_guess = 1

# Agent class with random behavior
class Random(Observer):
    label = "R" # Class label

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

    # Choosing the best action that does not immediately end the
game
    def get_best_move(self, state):
        # Analysis of the presence of obstacles around the agent
        obstacle_info = int(state) // 4% 15
        # Trimming invalid actions
        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())])

    # Model training function is missing
    def train(self, state, action, reward, newState):
        pass
    # There is no progress reset function
    def reset(self):
        pass
```

ListingA.3 – Source code of the Strategy class (file strategy.py)

```
from random import random
import json

# The class is responsible for analyzing the agent's behavior
class Strategy:
    def __init__(self) -> None:
        # Parameters for Q-learning
        self.learn_rate = 0.1
        self.discount = 0.9
        self.randomness = 0.9
        self.randomness_decay = 1.0005
        self.randomness_min = 0.00005

        # Initialize Q-table with zeros
        # You can use self.load_Q() to load a ready table
        self.Q = {}
        # The minimum distance to the target is (0,0) and the
maximum is (9,9), so the L1 distance is in the range [0, 18]
        for dist in range(20):
            #3 categories: agent's first step, first 15 steps
and the rest
            for stage in range(3):
                shift = 3 * dist + stage
                self.Q[str(shift)] = {}
                # Two "answer" options: agent is purposeful or
has random behavior
                for strategy in range(2):
                    self.Q[str(shift)][str(strategy)] = 0

    # Select the most appropriate behavior class or random when
training the model
    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())

    # The model training function updates the value in the Q-
table cell at the intersection of state and 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()))

    # Reducing randomness in predicting agent intentions
    def update_randomness(self):
        if self.randomness > self.randomness_min:
            self.randomness /= self.randomness_decay
    # Get the current state of the agent in the system based on
its distance to the target and
    # number of steps since the start of the game
    def get_state(self, relative_distance, steps_count):
        # Dividing the number of steps taken into categories
        step_group = 0 if steps_count <= 1 else 1 if steps_count
<= 15 else 2
        # Forming the status bar
        state = 3 * relative_distance + step_group
        return str(state)

    # Save Q-table to file
    def save_Q(self):
        open("Strategy.json", "w").write(json.dumps(self.Q))

    # Load Q-table from file
    def load_Q(self):
        self.Q = json.load(open("Strategy.json", "r"))

# Getting the L1 metric distance between two points on a plane
def get_L1distance_from_target(agent_pos, target_pos):
    return abs(agent_pos['x']-target_pos['x']) +
abs(agent_pos['y']-target_pos['y'])
```

Listing A.4 – Source code of the Design class (file design.py)

```
from tkinter import *
from tkinter import ttk

# The class is responsible for the graphical user interface
class Design:
    # Path elements taken by the monitored agent
    agent_paths = []
    # Tracked agent IDs
    tracking_agents = []

    # The constructor accepts the number of agents in the
system, the positions of obstacles and
    # premature game termination points on the plane
    def __init__(self, agent_count, obstacles_pos, giveup_pos) -
> None:
        # Setting up the application's graphical window
        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)
        # Parameter for updating the contents of the graphic
window (False in training mode)
        self.enable_drawing = False

        # Creating a background for the level
        self.w.create_rectangle(0, 0, 300, 300, fill='white')
        # Creating a graphical representation of agents
        self.agents_rect = []
        self.agent_photos = []
        self.agent_last_pos = []
        for agent_id in range(agent_count):
            # Agent images are in player№.png files
            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]))
            # Save previous position for drawing paths
            self.agent_last_pos.append(None)
        # Creating a graphical representation of obstacles
        for obstacle in obstacles_pos:
            # Obstacles are marked in grey
            self.w.create_rectangle(30*obstacle['x'],
30*obstacle['y'], 30*(obstacle['x']+1), 30*(obstacle['y']+1),
fill='lightgray')
        # Create a graphical representation of the target
(texture in the pig.png file)
        self.pig_photo = PhotoImage(file = 'pig.png')
        self.target_rect = self.w.create_image(0, 0,
image=self.pig_photo)
        # Create a graphical representation of the game's
premature termination point
        # Marked with a blue circle
        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")

        # Adding a status bar for confidence in the partner's
focus
        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)
        # Create a graphical representation of the status bar
        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")
        # Adding tooltips to the sides of the status bar
        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)

    # Rendering the entity in a new position
    def move_pig(self, target_pos):
        # Checking if the graphic window needs to be updated
        if not self.enable_drawing:
            return
        self.w.moveto(self.target_rect, 30*target_pos['x']+5,
30*target_pos['y']+5)

    # Render the agent in the new position
    def move_agent(self, agent_id, agent_pos):
        # Checking if the graphic window needs to be updated
        if not self.enable_drawing:
            return
        # Drawing the path taken by the agent
        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)

    # Toggle the parameter responsible for the need to update
the graphics window
    def toggle_drawing_mode(self):
        self.enable_drawing = not self.enable_drawing

    # Enable path tracking for agent with agent_id
    def track_agent(self, agent_id):
        self.tracking_agents.append(agent_id)

    # Drawing the path taken by the agent
    def draw_path(self, agent_id, new_pos):
        # Check if this agent is being monitored
        if agent_id not in self.tracking_agents:
            return
        # adding a new path element
        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))
        # Update path color to show the passage of time
        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))
        # Update agent's last position
        self.agent_last_pos[agent_id] = new_pos

    # Refreshing the contents of the graphics window
    def update(self, value, right_side, title):
        # Checking if the graphic window needs to be updated
        if not self.enable_drawing:
            return
        # Update window title
        self.master.title(title)
        # Update status bar
        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")
        # Render all updates on the screen
        self.master.update()

    # Clear all agent paths when resetting game progress
    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()
```