3D Interactive Maze

A Multi-Agent System Problem

Isabel Soares
AASMA
Instituto Superior Técnico
Lisboa, Portugal
isabel.r.soares@tecnico.ulisboa.pt

João Dias
AASMA
Instituto Superior Técnico
Lisboa, Portugal
joao.ribeiro.dias@tecnico.ulisboa.pt

Rodrigo Sousa
AASMA
Instituto Superior Técnico
Lisboa, Portugal
rodrigo.b.sousa@tecnico.ulisboa.pt

ABSTRACT

Our team developed a solution to overcome the resolution of a maze.

The problem is addressed in a 3D environment, where two agents will try to escape a maze with some obstacles. There will be locked doors and unreachable heights (or at least without any cooperation). Also, they are only aware of their position and state, although they are able to communicate with other agents and send or receive information.

There are several ways to achieve the end of the maze successfully. Having that into account, we decided to try different architectures approaches, such as reactive, deliberative, hybrid and learning algorithms.

With the development of this project, we intended to find the advantages and disadvantages of different architectures by testing all of them with our defined metrics.

1 Introduction

The scope of our project is to solve a 3D maze. However, our agents face some obstacles, as locked doors and “unreachable” heights, that need other agent to help the first to surpass those difficulties. This can be achieved by pressing a pressure plate or by placing a block next to the “unreachable” heights. So firstly, it is important to explain that it is impossible for a single agent to finish the maze, because it cannot unlock doors by itself, as it needs to leave the pressure plate, what will be proved. Then, related to the scope of having multiple agents, we hope to find the most efficient way of solving the maze in an interactive world that is shared by these two agents.

Although the strategies used are of the students’ choice, our project will have to follow the context of the AASMA course, such as exploring the multiagent theme.

The main objectives of the project were both the creation of the best strategical way of surpassing the obstacles in order to complete the maze, and the comparison between multiple tested strategies to point out the advantages or/and disadvantages of each one.

With the recent developments of agents that are capable of navigating the real world comes the need of algorithms and architectures that adequately solve the problem of navigating a 3D interactive world. Through this project we were hoping to gather a better grasp on the needs for this specific kind of systems as well as the strategies that they employ or that naturally surge on learning agents that face this problem. To measure the pros and cons among different architectures the team specified some metrics, explained in more detail later.

2 Implementation

The environment is a 3 dimensional one as referred above, that was built and displayed with the use of a game engine in python called Ursina [1].

Figure 1: Map number 7 created by the team, has all the possible elements

All blocks are built and placed in a unit-based system, meaning that each block can be represented by a cube with one unit in each side and placed in a position (x, y, z) described only by integers. The only blocks that differ from this are the agents and doors (which can both be simplified by a cube which has double the height of a normal block) and the pressure plates (which occupy a given position but only take around 10% of the height of a normal block). Every agent has a defined colour and a finite number of blocks of a different colour, but with the same colour of another agent (seen in the figure above as red and blue). The agent is able

to move only forwards, rotate right or left and climb a block that has the same colour and is ahead of the agent. The agents are also able to place a block ahead of themselves as well as break it. Apart from the difficulty associated to reach the end of the maze, the goal block, it has some obstacles as locked doors that can be unlocked if an agent is standing on the pressure plate associated to that door as well as some walls for which they need other agents' blocks to surpass.

When it comes to observations, the agent is able to observe the blocks around him since these are the ones that are strictly necessary for it to navigate the world, the rest is either simply unknown to the agent, or it will be stored in a temporary memory that agent stores of the world (if such capability is given to the agent).

We used a multiagent system, as a single agent will more than likely not be able to escape the maze without cooperation, coordination and negotiation with other agents. We tested some strategies such as reactive, deliberative, hybrid and even learning agents, since with our work we wish to gather a better understanding of the effects that each one has on the problem resolution.

2.1 Map Loading and Graphical Interface

Since we wanted to easily develop new maps or change existing ones, the maps are created through a JSON like formatting, where each block its position and relevant attributes are set. Our code will then, in run time, build the map according to what is specified in its corresponding file. The process of running was also simplified for a better experience, in order to run any given map with any given architecture the user simply needs to place these choices as arguments when running the code.

In order to get a better grasp of process of solving the maze as it happens the team invested some time developing its graphic interface. At the top left there is a panel where the user is able to see in real time the current value for the metrics, either in general (containing the sum of the various agents) or for each one of the agents, by selecting the agent it wishes to see. In this very same panel, the user is able of selecting how long one game tick should equate to in seconds.

In the right top side of the screen, it also exists another panel. In this panel the user can see which messages are being sent to the world. By predefinition the messages being displayed are the requests for help, but the panel allows for the selection of currently being helped messages and information messages about the world. For each one of these messages is displayed the name of the agent which sent it, the position relevant to the information and either the action requested or the type of block that is being transmitted as information to the other agents.

The metrics at each step as well as a log of the output of the various agents is saved on every run to a directory called exports. By doing this we can run all the agents and subsequently analyse the various results and elaborate graphics and charts that allow for comparison among agent's architectures.

2.2 Agent

In this section we would like to explain the agent general architecture and capabilities. We will later specify the approach for each one of them.

The team tried to provide the same set of rules and possibilities to every agent, in order to compare them and gather a good grasp about which one of the agents is more adequate for each situation and this kind of problems.

The agent is able to choose one action per timestep. One can either stay, move in front in the same height, up or down, rotate left and right, create a block in the same height, up or down and break that block.

Every agent can also perceive the blocks that are around it up to 2 blocks up and down. They can then discriminate the type of entity or block that are in one of those positions, whether they are walls, agents' blocks, goal blocks, doors, or pressure plates. The agent can also understand if it has permissions to pass a door or a wall.

2.2.1 Reactive Adapted Agent

The first agent approach that our team tried was the reactive one. Probably the architecture with the least complex architecture, but helpful to guide the rest of our research to find other type of approaches. Although a purely reactive agent does not have internal state, we quicky understood that we would need something more than only a set of instructions to lead the agents to the end of the maze.

Having that into account, we defined five types of states that an agent could have to execute the actions accordingly to those states. The agent could be in a random state, be the one responsible for pressing the pressure plate, the agent that pass the door, the one that creates blocks or finally climb up other agents' blocks.

Our solution also allowed agents to communicate between each other to understand in which state they should be.

The pseudocode below illustrates the decision making of the agents. Note that the process is not the complete, since there are some complex functions that contain some background instructions and others, such as the ones responsible for communication with messages, that change the state of the agent for instance.

```
if door_in_front and permission and not prohibited:
    if state == random:
        send_message() # asking for help to open the door
    elif state == door and not door_open:
        stay()
```

```
    elif state == door and door_open:
        move()
        send_solve_message() # to finish the help
elif state == pressure_plate and pressure_plate:
    stay()
elif wall_in_front and permission and state == random:
    send_message() # asking for help to climb the wall
elif state == wall and wall_in_front and permission:
    climb()
    send_solve_message() # to finish the help
elif state == create_block and not block_in_front:
    create_block()
elif state == random and agent_block_in_front_with_same_color:
    climb()
elif state == random and own_goal_block:
    stay()
elif (state == door or state == wall) and count_after_help == 500:
    send_solve_message()
else:
    random_action()
```

It is easly understandable by the pseudocode that the agent still has a decision making based on the present.

Both agents start with a random posture, choosing random actions. However, when it gets in a position with a door or a wall in front of it, it sends a message to the other agent asking for help to pass the obstacle. It it in that moment that one becomes responsible for passing the door and the other opens it pressing the pressure plate. When the obstacle is marked as complete, the agent that was asking for help, finishes the request sending a message saying that it does not need help anymore.

The team noticed that when there is an obstacle A that need to be surpassed by both agents, if for instance one of them passes that obstacle A and asks for help about an obstacle B without the other agent having passed A, we could have a problem, because one would be trying to help the other agent in vain. Thus, we have a condition specifying that after 500 iterations being helped, if the obstacle was not surpassed, both agents return to the random state and try to find other obstacles.

Finally when an agent reaches the goal block, the chosen action is stay until the other agent also reaches the goal or until they trigger another obstacle if needed to be solved.

We also have some specificities that deviate from the sense and react posture. When an agent passes a door, it saves the door position in order to neither go back, nor ask other agents for help to open the door again. Also, when some agent receives a message and need to help other by creating a block, that message comes with a position where the block should be placed, and the agent saves that position internally to know where to place the block.

2.2.2 Deliberative Agent

The deliberative approach was the first one where we were able of delegating a higher level of logic dependent on beliefs, which can be increased and improved during the ‘exploration’ of the agent. We started off by defining the overall logic of the deliberation process of the agent, focusing mainly on the logic that would tolerate the update of beliefs from perceptions gathered around the agent.

For this reason, it is important to define first some general rules that were established specifically for this architecture. The agents could not communicate with each other, but they could indeed make beliefs / assumptions about other agents from their own beliefs. The agent would take as percepts:

- the distance to the goal
- everything in a cube of size 3 x 5 x 3 centered on the agent
- they could know the position of a door that is opened by a pressure plate in this perception cube.

Regarding the **belief update function**, there are **three different logics implemented** on it on which the function can be subdivided:

The general logic of the belief function takes the percepts and constructs a belief for each position. It analyses and stores it in a map internal to the agent. If a previous belief already was stored at that specific position, then the value is merely replaced by this new belief.

The logic behind the goal blocks is essential since the main objective of the agent is to reach the goal block, meaning it is very important to discover its position, or at least its approximate position. In the first iteration of the deliberative agents (when it does not have any hypothesis for the position of its goal) it takes the distance to the goal. It creates a spherical surface with the floor of the distance has the smaller radius and the ceiling of the distance has the bigger radius. In the subsequent iterations, it implements the same logic for the current distance to its goal, but this time, it updates the hypothesis for the goals by intersecting both sets (the previously stored and the new set now computed). It does this until it either reaches a single value or has taken as direct percept its own goal.

Finally, the last important part of the belief update function deals with agents. Since we know agents are uniquely identified on the problem by their name, we wish to have only one stored on our belief map, because having more than one belief position for a single agent would be wrong and could cause future problems. To solve this problem every time, we take as direct percept an agent we search through our current beliefs and delete any belief that involves this agent in another position. When we started developing the rest of the hybrid agent, we also detected the need for estimating the position of other agents, for example, when dealing with doors. By this we mean that at step t if we have agent in our percepts and on step t + 1, we no longer have this

very same agent in our percepts we conclude that he must have moved in the position that he was facing (away from us) and update our beliefs according to this rule.

The next part to implement was the function which **dealt with updating the desires of the agents**. The logic on this function is quite simple: we simply used a list as priority queue to store the desires of the agents. On every iteration, the agent always has the (basic) desire of reaching its goal. If no other desire is added, this is the desire to which it will create an intention further ahead.

After adding the basic desire of reaching the goal the agents checks whether it believes that any other agent need him to place a block and if so, it adds this desire. The agent believes that another agent needs a block placed if it encounters a wall with two units of height. As a matter of fact, this logic is not perfect for sure, but we are limited due to the restriction that the agents cannot communicate with each other. So, we need to make assumptions, and we can only assume that if we reach a wall that the other agent might want to climb it. In more difficult and complex maps, this logic, would not work (or at least it would take a long time to eventually get it right) since there could exist multiple walls where the other agents would not be interested in going and the number of blocks is limited.

Then we check for the need of placing a block, the agent checks whether any other agent needs a door to be opened. Once again, we needed to make assumptions, although this time our assumption is stronger since it will not cause big problems if incorrect. We assume that an agent wants a door to be opened if it is standing next to it.

The counterpart of the last one, we check whether an agent wants a door to be opened back. The concept is not intuitive. So, let's assume that we just got through a door, what we are checking is whether there is any agent behind a door which we got through which also wants to pass it. We verify this by checking whether we believe an agent is on the other side of the door. We compute the distance to both the agent and the door as well as the cosine metric between them and if the distance to the agent is greater than the distance to the door and the cosine is above a certain threshold, then, we believe that it wants the door to be opened back and this agent will remain to believe this until it sees the agent elsewhere.

Lastly, we check whether the agent has been stuck in the same position for too long (a parameter specified by us controls what the agents considers as too long). If the agent surpasses this value, then we add the desire of roam around to it. So that it feels motivated to explore and (eventually) figure out if another agent needs help.

Having dealt with the **desires of the agent,** we now have to deal with the **intention update** of the agent. The logic behind this function is quite simple. If the agent has as selected desire to reach its goal, it randomly chooses one of the hypotheses for its goal and sets this has the intention. If the agent has as desire to either open or open back a door, it searches in its beliefs for the pressure plate that opens the corresponding door. In the case, it does not know of it, it will set as the intention position None (this is not a problem since the functions used to build the plan deal with this possibility). If the agent desires to place the block the same method used for the belief return the position to place the blocks and sets it as intention.

The last essential step for the deliberative agent is the **building of a plan**, since the problem can be described as solving a maze, most of the functions elaborated dealt with elaborating a path of positions or actions which lead to a successful execution of the intention. We have three different functions which elaborate a plan: *build_path_plan*, which is the most general, *build_path_plan_block* which does almost the same with the small difference that at the end it takes into consideration the placement of an agent block (either right in, above or under the position ahead of the agent) and finally *build_longest_path_plan* which elaborates the longest path possible according to its beliefs.

The method *build_path_plan* is used when the agent has the intention of reaching its goal, simply wants to explore unknown positions, open a door or open back a door. The method first tries to develop a path of positions which lead to the objective position from its beliefs, if it is not able to do so, it iterates the various unknown (but possible) paths that can compute from its beliefs and chooses the smallest path that leads closer to the final position, in the case that there are no unexplored options the agent simply remains still.

The method *build_path_plan_block* is used when the agent has the intention of placing a block. As for its logic its exactly the same with the difference that if it can find a valid path of actions, according to its beliefs, that leads to the position where it wants to place a block, it corrects this path as to place the block correctly. In the case that it cannot formulate a valid path to the position it results exactly on the same as the previous method.

The method *build_longest_path_plan* is used when the agent has the intention of roaming around. This method has the simplest logic, (although the name does not imply it) it tries to explore unknown position, we do this since it does not make sense to roam around randomly through the map if it still has positions which is not sure about, and then, in the case that it has nothing else to explore, it tries to formulate the longest path possible through the map, not repeating positions or actions in each position.

Many other methods were used in order to better segmentate the problem of, building paths, converting paths from positions to actions, and checking the correctness of previously established plans (which is used to evaluate the soundness of the plan formulated).

2.2.3 Hybrid Agent

For the hybrid architecture, we followed a simple vertical layered architecture. We chose this since by doing this we would focus on

the interactions between the layers, giving specific tasks to each one of the layers instead of trying to implement a function that would make the final deliberation for the agent from the output of the various layers.

Our agent has three layers: a **Reactive Layer**, a **Deliberative Layer** and finally a **Communication Layer**. Each of the layers has a memory associated to it, where information pertinent to the layer is stored, as an example, the Deliberative Layer Memory stores its beliefs. Each one of these layers is able to communicate either with the layer on top or on the bottom, by sending messages through the main control module. The percepts are initially sent to the Reactive Layer and it's the Reactive Layer that has the task of sending the chosen action back to the main control module.

In order to facilitate the communication between layers we implemented a *Message* class. This class stores the layer from which the message was sent, the indication whether the message should be sent up or down, the type of message (which is an enumerator defined also in this class) and finally the content to be sent.

The main control module, the agent itself first takes the percepts of the world as well as certain agent's properties such has its position, current forward direction, its name, number of blocks, etc... These percepts are then sent to the Reactive Layer which redirects this information to the Deliberative Layer.

The Deliberative Layer then deals with these percepts just as it dealt with them in the Deliberative Agent. After dealing with them, then it requests the communication layer for the messages that were sent to this agent, either requests for help or information about the world. After, the communication layer sends this information to the Deliberative Layer, which deals with this information in a similar way to the percepts. It is important to know that the Deliberative Layer gives precedence to information it took has percepts then information gathered through the Communication Layer (since this one might be outdated). The Deliberative Layer then makes its deliberation similar to the one it made before.

The main difference between the Deliberative Agent and the Deliberative Layer, is that the former had to make assumptions about the other agents, and the layer does not make them. The Deliberative Layer, for example, only assumes that the other agents need help opening the door if it receives explicitly a message from an agent saying that, the same happens with the placement of blocks. The capability of agents of communicating simplifies the overall logic of the deliberative agent, because of this improvement, we do not need the notion of opening back a door to an agent since, if the agent on the other side requires a door to be opened, it simply requests it.

Since we no longer needed to make assumptions about the other agent state / desire, we could implement logic that would allow for the breaking of blocks, defined in the method *build_path_plan_break_block*. The way we implemented relied on the principle that when the agents was unable of reaching its goal and no other position was unknown to it, then he would believe that it had to break its block in order for the other agent to place his. This implementation would not have worked for the deliberative architecture since it could not know for a fact if the other agent still needed this block to be placed.

Apart from the logic of breaking blocks, the only logic that needed a bit of change was the one that involves the request of help in placing a block. When the agent requests that another agent places a block, this agent needs to move away from this position, in a way that allows the other agent of reaching this position and of placing the block. With the intent of easily solving these problems, we simply send the first agent as far away as possible. Eventually this agent will receive information that the block was placed and will come back.

After the deliberation process is completed, the deliberative layer sends to the communication layer pertinent information about the world that wants to be shared, for example doors, pressure plates, agents block, winning blocks, and its own position. As a fact, this logic was not explicitly necessary but it increases the overall efficiency of the process.

Lastly the Communication Layer will return the flow of process to the Deliberative Layer which will then send the chosen action back to the Reactive Layer, which will then send the action to the Control Module has the selected action.

2.2.4 Reinforcement Learning Agent

The last approach, followed a different path, taking into account the learning capabilities of choosing the best position values to follow the correct decision-making path.

We used a Q-Learning approach, which is a famous model-free reinforcement learning algorithm that finds the optimal policy by getting successive rewards while the agent explores the environment. With that approach we would have a Q-function to store the values of total rewards following a certain action in a specific state.

As we wanted to have a good grasp of different architectures and the project was not fully dedicated to reinforcement learning, we decided to follow a simple approach. So, our team followed a Single-Agent Reinforcement Learning, where each agent would have their own Q-value matrix, as well as the rewards would be different for each one.

Because our environment is dynamic, the idea was that each one of the agents would have different states, as it was in the reactive approach, referred before. Recalling what was stated, the five states would be the random state, the state for the door, the state for the pressure plate, the one for climbing the wall and finally the creation of blocks.

Regarding the states mentioned, the agent receives different rewards for every action in each of the states. In order not to contaminate the Q-table of each state, every time an agent changes state, it also resets its Q-table. We can perceive this as

levels. Every state is a level that an agent must learn and pass concerning the specific rewards for that level.

Like the reactive approach, the agent changes the state when it has a door in front and sends a message to other agent to change its state for pressure plate as well. The same happens with wall and create block. When a task is finished, after the message communication, they both return to random state again.

For most positions the agent does not receive any reward, i.e., reward = 0, except in some specific ones, having into account its current state. The rewards for each state are as following:

Random state:

- every position: $\frac{1}{distance\ to\ goal}$

Door state:

- door's position: 1
- position adjacent to door and looking to it: 0.2
- position adjacent to door and not looking to it: 0.1

Pressure plate state:

- pressure plate's position: 1

Wall state:

- position adjacent to wall: 1

Create block state:

- position adjacent to a position adjacent to wall: 1

Having the rewards established, we needed to define the parameters to learn our Q-values. The Q-learning function is the following:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (r_t + \gamma * maxQ(s_{t+1}, a) - Q(s_t, a_t)),$$

where $Q(s_t, a_t)$ is the Q-value in the state s_t following the action a_t, α is the learning rate, r_t the reward received and γ the discount factor.

The action choice is done based on an epsilon-greedy approach, where we opt between exploration - choose a random possible action - and exploitation - choose the best action regarding the learned Q-values until that moment - according to an epsilon value. In this case we thought that it would be a good idea if epsilon started with the value 1, what would make the agent choose a random action, and explore the maximum states and get almost every possible type of reward to fill in the Q-table early. Then, applying an epsilon decay algorithm, its value would decrease, what would make the agent be more willing to exploit to the best action following the policy. This value would reach a low value (0.2) after some iterations to conclude the learning process and choose the best path.

We tried different combinations of learning rates and discount factors. Our best results were learning rate = 0.9 and discount factor = 0.6 for the agents' random state and the learning rate = 0.4 and discount factor = 0.9 for the agents' obstacle states (door, pressure plate, wall and create block). The choice of these values will be explained latter in the evaluation section,

The agent also saves the door position when it passes that door so that it does not go back. Once an agent surpasses a wall obstacle, the other agent that created the block, signs that block not to break in the future.

3 Evaluation

First, and before starting to compare the architectures between each other, it is important to explain, that the team tested different values for the learning rate and the discount factor, as well as the epsilon. These values were referred before within the implementation of the Reinforcement Learning. But notice that the results were gathered through the use of a script that tested the agent for different combinations of these parameters. The epsilon was also tested, but we opt not to use a static one and instead the epsilon decay. Our results were also supported by some research the team did and reasoning about what would make sense in the context. As we provide rewards according to the distance to goal, this could have a high learning rate, as the rewards are very expressive and complete, and a lower discount factor, as the future rewards are not so important in this case (learning rate = 0.9 and discount factor = 0.6). The same is not true for the stages with obstacles and solutions, because we give almost only a punctual reward in one position, what would require a low learning rate and high discount factor (learning rate = 0.4 and discount factor = 0.9). So, our results support the theory and vice versa.

To evaluate both the characteristics of each agent and the differences between each model, whether being reactive, deliberative, hybrid or with learning, we tested seven different maps. The maps used had the following obstacles:

- map 1 – reach goal.
- map 2 – door for both agents; reach goal.
- map 3 – wall for both agents; reach goal.
- map 4 – door for both agents; wall for both agents; reach goal.
- map 5 – door for Agent ORANGE; reach goal.
- map 6 – wall for Agent BLUE; reach goal.
- map 7 – door for both agents; wall for Agent BLUE; reach goal.

The metrics used for this project were:

- the capacity of the agent or agents of escaping or not the maze;
- the time they need to accomplish it (measured in time steps);
- the number of steps they needed to escape (measured in physical steps) - since an agent has the possibility of not moving, this will differ from the time steps mentioned before;
- the number of blocks placed by agents in the environment;

- the number of blocks removed by agents from the environment;
- the number of times they press the pressure plates;
- the number of messages that were sent to other agents.

These metrics were both measured:

- individually;
- collectively (which may give insights to the roles that different agents take in the problem-solving strategy).

Having all of that into account we then plotted graphs for every map and metric so that we could compare the models.

We will do an overview of all the maps with the aspects that are repeated among all of them and then analyze the specificities of each agent and each map.

It is undoubtful that the worst architecture for every map is the one with Reinforcement Learning. This is due to the fact the model needs time to converge to the obstacle or solution. And as we used a Single Agent approach, they learn “per levels” and every level is a new stage that needs the first thousands of iterations to watch every state and the posterior ones that continue learning while starting to get near to the desired obstacle or solution.

The Reactive approach revealed himself as the one with the less complex algorithm, but not enough if we want to reach the goal

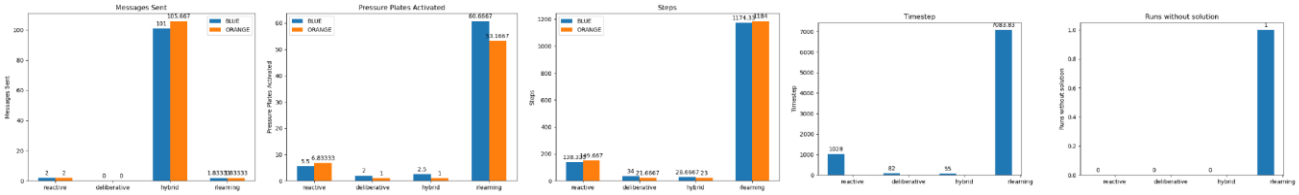

Figure 2: Map 2

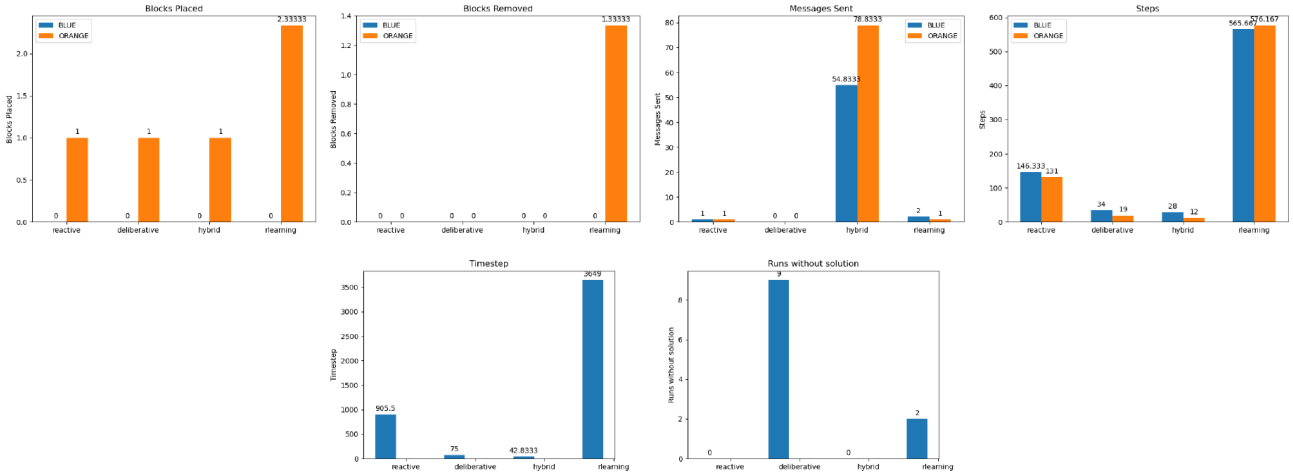

Figure 3: Map 6

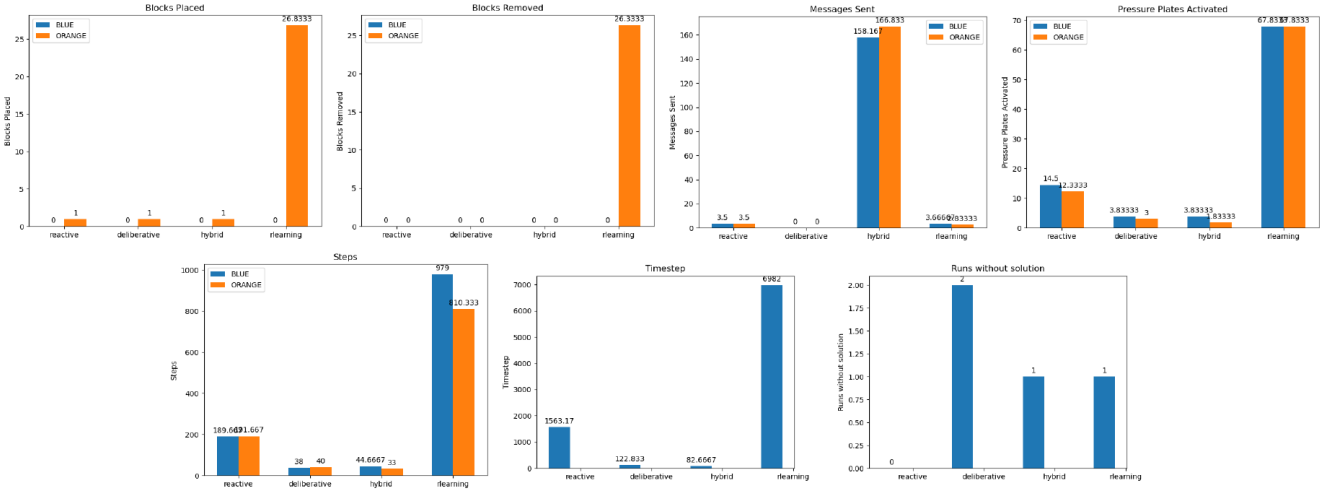

Figure 4: Map 7

faster, as it does not guarantee convergence and the random movements until it founds an obstacle or solution can take a lot of time. Although we do not have guarantees of convergence, as our maps were small, this architecture proved to be the one that found a solution the most often for this context, in this case every time. The same cannot be said about the Hybrid and Deliberative approaches, that did not reach a solution on time sometimes. However, these models were the fastest for every map, with the Hybrid leading, which was expected due to the high-level planning and communication abilities.

If we take a look to the number of blocks placed and removed, we can notice a big difference between the Reinforcement Learning and the other three architectures, as the Reactive and Deliberative only create the block in the correct place when asked to do so or make the assumption that Is needed, respectively. The learning algorithm does not do it so blindly. The agent receives a message asking for help, but only changes the reward when placing the block, so it keeps creating and breaking blocks until they learn where to place the block and that this should not be broke. The Hybrid agent does not place blocks nor remove them often, but sometimes that can also happen.

We can also notice that in terms of messages sent into the world, the Hybrid has clearly sent many messages in comparison to the rest of the architectures. This was expected and intentional in our end. Since in our Hybrid architecture the deliberative layer takes a big relevance, and this layer stores the beliefs it has about the world. These beliefs are the only pieces of information used, when developing a plan and more importantly a path to an objective. It is important to keep these beliefs updated. With the addition of a communication channel, when compared to simple Deliberative architecture, we wanted to take full advantage of this aspect so we send every information that might be pertinent to the other agent (doors, pressure plates, agent position, goal blocks and agents' blocks), which we can verify that it helped in terms of steps it takes to solve each map.

Now going into the various maps and their specificities, the map number 4 was the worst for the learning algorithm, since it has a wall obstacle near the door obstacle. And when an agent surpasses the first obstacle and asks for help with the second, the other agent is not able to help because it is still stuck before. This does not happen with the reactive agent for instance, since we change the state after 500 iterations as explained before. As the learning agent already needs a lot of time, we could not take the same measure (that would have to be something more than 5000 iterations), due to our time limit that we wanted to fulfill. The map number 6 is an interesting one that we wanted to talk specifically. Although simple from a design approach it caused a lot of problems to the deliberative architecture, the problem being that the agent that got first to the wall was the one that placed the block, but this was the wrong block to place in that situation. Since they are unable to communicate, they are unable of indicating that the block placed is not necessary and that it shall be removed. From the point that that block is placed there is no going back, and they are stuck in an impossible situation.

Finally, it is a trivial conclusion, but we also showed that the number of steps, blocks placed/removed, pressure plates pressed, and messages sent increased with the complexity of the map.

4 Conclusion & Future Work

After the analysis of the results that we got after evaluation, we can easily conclude that the fastest architecture is the one with Hybrid agents. The Reactive approach is better if we want to increase our probabilities of finding a solution. However, none of these architectures can guarantee convergence and that can be something that we could work in the future. The team got very interesting results and we consider our work very successful, because we supported all that was expected, such as the learning algorithm being the slowest, the reactive being a simple approach but efficient and the deliberative and hybrid being much faster and showing the impact that having a communication channel has on results.

For future work, we consider that other Learning Algorithms could be explored, like coupled or joint action. ones. We are not expecting a better performance than the Hybrid approach, or at least in number of steps needed to conclude the maze, but maybe we could have more convergence guarantees. As for the Hybrid approach we feel like testing other approaches to segmenting the implementation of building a plan, more specifically a path, could lead to interesting results.

CCS CONCEPTS

• Computing methodologies ~ Artificial intelligence ~ Distributed artificial intelligence ~ Multi-agent systems

• Computing methodologies ~ Artificial intelligence ~ Distributed artificial intelligence ~ Cooperation and coordination

KEYWORDS

Artificial Intelligence, Multi-Agents, 3D problem, Cooperation

REFERENCES

[1] Ursina Engine, https://www.ursinaengine.org/