# Behaviors Verification - Report

This document contains a description of the project Behaviors Verification done by Filipa Correia between 6/08/2021 and 08/10/2021.

Link:

- Github repo created by Filipa: https://github.com/iv4xr-project/rl-behaviors-verification

Other relevant links:

- Bruno's code: https://github.com/carreirabruno/Dissertation_BrunoCarreira_iv4XR
- Bruno's some other code: https://github.com/carreirabruno/RL-in-iv4XR
- Some other: https://github.com/carreirabruno/Tese_BrunoCarreira_Squary-Shappy
- Levels generator: https://github.com/iv4xr-project/labrecruits-levelgenerator

# Goal and Approach

The goal of this project is to verify if certain behaviors can occur in a specific game level or to measure the level's susceptibility for those behaviors. To simplify, we decided to start with the collaboration behavior. In other words, we are trying to evaluate how collaborative a game level is.

Our approach to address the simplified version of the problem was to create two types of agents and use their learned behaviors to evaluate the game level. This approach was proposed by Manuel Lopes and is detailed in a file named "Automatic_Level_Analy...".

# Implementation

## Scenarios / Game Levels

Each level has a set of buttons, some of them unlock doors to other rooms, while others are target buttons. The goal of each level is to press all target buttons.

The scenarios are loaded from a .txt file.

## Gridworld

The GridWorld class sets up the environment for the agents to learn their policies or to simply execute learnt policies. Therefore it includes methods such as:

- Reset or RandomReset
- CheckGoal
- GetState (GetStateFullObservability, GetStatePartialObservability)
- Learn (LearnDecentralized, LearnCentralized)
- Step (StepSingleAgent, StepDecentralized, StepCentralized)
- EvalAgents (EvalAgentsDecentralized, EvalAgentsCentralized)
- Render
- WriteAgentsQtableToFile
- LoadAgentsQtableFromFile

## Agents

I developed three types of agents: the centralized, the decentralized and the single agents. Common to all agents:

- Action space is [UP, DOWN, LEFT, RIGHT, PRESS, NOTHING].
- Action selection is done with Egreedy and when choosing the current best action for a certain state, ties are solved randomly.
    - E (or epsilon) decays linearly from 1 to MIN_EPSILON (set as 0.05) during 70% of the episodes. The last 30% of the episodes, E remains at 0.05.
- Rewards:
    - 100 for reaching the goal
    - -0.4 when choosing NOTHING
    - -1 for any other action

    - Notes: the reward of doing NOTHING constrains the time it takes for the agents to learn.
- Learning algorithm is DynaQ with 10 planning steps.

### Centralized Agent

The centralized agent is a single entity controlling the actions and knowledge of two players.

The state is a string with [pos_x_a0] + [pos_y_a0] + [pos_x_a1] + [pos_y_a1] + [state of all the buttons]. Because it maps the position of the two players (and all the buttons) it is accessible with a method called *GetStateFullObservability*.

The possible action is a numeric value mapping the action of each player. For instance 0 represents the player0 doing UP and player1 doing UP, while 1 represents the player0 doing UP and player1 doing DOWN.

The rewards were doubled to have fair comparisons with other agents.

### Decentralized Agents

The decentralized agents are two distinct instances, each representing one player.

The state is a string with [pos_x] + [pos_y] + [state of all the buttons]. Because it maps the position of only one player, it is accessible with a method called *GetStatePartialObservability*.

These agents learn in a decentralized way but they learn at the same time in the same environment. Therefore, what happens is that the agents actually learn to cooperate if that gives them a higher reward.

### Single Agents

These agents are independent of one another as they learn alone in the environment and therefore, the learnt policies correspond to an individualist behavior.

The learning method is in the Agent class file: *LearnDynaQ*. When implementing this type of agent, a problem occurred while evaluating two simultaneous agents of this type in the same environment. Because as soon as one of the agents opened one door, the other stopped to know what to do as it had never seen that state. To address this issue, at learning-time instead of resetting the agents to always start in the same state, I created the RandomReset. This method, 30% of the time, resets the environment to a random state (including position and button states).

## Running instructions

The main.py can be executed as follows:

- *main.py -learn* [AGENT_FLAG] [SCENARIO]
    - In this mode, the agents learn their policies for that specific scenario and, in the end, the Qtables are saved to a .txt file in the /policies/ folder.
- *main.py -run* [AGENT_FLAG] [SCENARIO]
    - In this mode, the agents load the Qtables saved in the /policies/ folder (if any) and execute the learnt actions step by step, while the console log visually displays their behavior.
- *main.py -compare* [AGENT_FLAG] [AGENT_FLAG] [SCENARIO]

- In this mode, you will get some evaluation measures comparing the two types of agents (given in the execution parameters).

[AGENT_FLAG] can either be *-centralized*, *-decentralized*, or *-singleagents*
[SCENARIO] can either be *-s1*, *-s2*, or *-s3*

# Results

## Training

If someone takes over this project, I think it's relevant to know how long it takes to train the agents, at least the centralized agents (which take the longest). The times were taken in a Macbook Air (Processor: 1,8 GHz Dual-Core Intel Core i5; Memory: 8 GB 1600 MHz DDR3).

| | Scenario 1 | Scenario 2 | Scenario 3 |
|---|---|---|---|
| MaxSteps | 500 | 800 | 1500 |
| NumEpisodes | 1000 | 1000 * | 2000 * |
| Time (Centralized) | ~2m | ~36m | ~1h30m |

* Ideally, these agents should train for a longer number of episodes.

## Measures

As follows, I will detail the used measures for the comparison of two types of agents.

### Reward difference (Δ reward)

Absolute difference between the accumulated rewards of the two types of agents. The accumulated rewards are returned by the *EvalAgents* method. For the centralized agents, the output of their behavior is joint in one accumulated reward for the two players. However, for both the decentralized and single agents, the output of their behavior is the accumulated reward for each player, which is then summed. For this reason, the rewards of the centralized agents are doubled or "as summed" (as mentioned in Section Implementation.Agents.Centralized Agent).

### Hitmap per Player Difference (Δ hitmap / player)

Average difference between the behavior trace of a player for each type of agent (X and Y). The method *CompareHitmapsPerAgent* receives the size of the grid, and the two traces of each type of agent. A trace is a list of positions, in which each two elements of the list are the position of the two players (agent0 followed by the position of agent1). I first create four zeros matrices with the size of the grid, one for each player (a0 and a1) of each type of agent (X and Y). While iterating over the traces, 1 unit is summed in the corresponding matrix cell each time a certain player of a certain type passed in that position of the grid.

Then I calculate the norm of the difference (numpy.linalg.norm) between each player of type X and the same player of type Y. Then the method returns the average of the two players.

Because the hitmap difference might be substantially different for one player and not for the other, I have additionally written the hitmap difference for each player (a0 and a1) in the results table. The *Δ hitmap per agent* is the average between *Δ hitmap a0* and *Δ hitmap a1*.

### Joint Hitmap Difference (Δ hitmap joint)

Average difference between the behavior trace of all the players for each type of agent (X and Y). The method *CompareHitmaps* does not compare for each player and instead creates a single hitmap of where all players of a certain type have been.

### Policies Difference (Δ policies)

The policy matrix is binary as the agents have a deterministic behavior. Instead of having a policy matrix for each player (as I did for the hitmaps per player), here I created a single joint policy matrix for the two players (a0 and a1) as the centralized agents do by default. While for the centralized agents the policy matrix is obvious to get from the QTable (method *GetPolicy*), for both the decentralized and single agents it needs to be inferred/transformed. The method *CreateJointPolicy* does that, i.e., converts the two Qtables of the two players that have an action space of [UP,...,NOTHING] into a single Qtable (and therefore policy matrix) where the action space is [UP-UP,UP-DOWN,...,NOTHING-NOTHING].

ISSUE: However, currently there is a “problem” or issue that makes this measure a bit unfair. The *CreateJointPolicy* calls the method *ChooseCurrentBestAction*, which solves ties randomly. As you might see in the following results table, the policy difference is always a high number. If you run the comparison between agents of the same type (C vs C, D vs D, and S vs S), the policy difference should be (or close to) zero and it is not because of this issue, i.e., there is a high number of ties along the gridspace that causes two instances of the same agent to choose different actions in the same state. I initially thought that I was not training the agents enough time to explore “everything”. But while writing this document I thought of another solution. Probably creating the policy matrix non-binary and accounting for ties should solve it. Ex.: if the two highest Qvalues are for DOWN and LEFT, choose those two actions with a probability of 0.5 each. To implement this, you should not call *ChooseCurrentBestAction* inside the *CreateJointPolicy* method.

## Level evaluation

| * | C vs D | C vs S | D vs S |
|---|---|---|---|
| Scenario 1 | Δ reward - 13.4<br>Δ hitmap a0 - 5.9<br>Δ hitmap a1 - 5.7<br>Δ hitmap / player - 5.8<br>Δ hitmap joint - 8.2<br>Δ policies - 89.9 | Δ reward - 36.6<br>Δ hitmap a0 - 10.2<br>Δ hitmap a1 - 43.3<br>Δ hitmap / player - 26.7<br>Δ hitmap joint - 44.3<br>Δ policies - 92.6 | Δ reward - 23.2<br>Δ hitmap a0 - 8.9<br>Δ hitmap a1 - 43.7<br>Δ hitmap / player - 26.3<br>Δ hitmap joint - 44.4<br>Δ policies - 89.2 |

| | | | |
|---|---|---|---|
| Scenario 2 | Δ reward - 1.4<br>Δ hitmap a0 - 6.1<br>Δ hitmap a1 - 5.6<br>Δ hitmap / player - 5.8<br>Δ hitmap joint - 8.2<br>Δ policies - 179.5 | Δ reward - 7.0<br>Δ hitmap a0 - 3.0<br>Δ hitmap a1 - 3.0<br>Δ hitmap / player - 3.0<br>Δ hitmap joint - 4.2<br>Δ policies - 229.5 | Δ reward - 8.4<br>Δ hitmap a0 - 4.5<br>Δ hitmap a1 - 5.5<br>Δ hitmap / player - 5.0<br>Δ hitmap joint - 7.1<br>Δ policies - 171.1 |
| Scenario 3 | Δ reward - 1.8<br>Δ hitmap a0 - 4.3<br>Δ hitmap a1 - 10.0<br>Δ hitmap / player - 7.2<br>Δ hitmap joint - 11.0<br>Δ policies - 183.1 | Δ reward - 1.8<br>Δ hitmap a0 - 4.1<br>Δ hitmap a1 - 9.0<br>Δ hitmap / player - 6.6<br>Δ hitmap joint - 9.8<br>Δ policies - 181.2 | Δ reward - 0<br>Δ hitmap a0 - 1.4<br>Δ hitmap a1 - 7.7<br>Δ hitmap / player - 4.6<br>Δ hitmap joint - 7.9<br>Δ policies - 145.4 |

*C - centralized agents, D - decentralized agents, S - single agents

# Discussion

The following analyses ignore the last measure (Δ policies) due to the issue reported before.

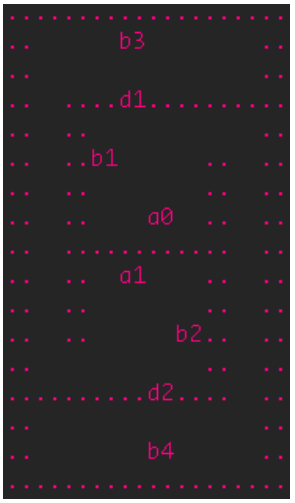
## Scenario 1


The first scenario is interesting because a1 is trapped and the only way out is if a0 presses the button b2 (to open the door d2). Player a0 is capable of solving the level alone if he presses b1 and then b3, which makes it possible for him alone to press the one and only goal/target button b4.

The behavior learnt by the *centralized agents (C)* corresponds to the collaborative behavior, i.e., a0 presses b2 and a1 goes directly to the target button b4 (13 steps). What the *single agents (S)* have learned to do, because they train alone in the environment, is precisely the opposite, i.e., a1 does nothing while a0 presses b1, b3 and finally b4 (43 steps). The *decentralized agents (D)*, as I mentioned before, have also learned cooperative behaviors as they are advantageous. However, because they don't fully observe the environment (i.e., they don't know the position of the other player), they take slightly more steps (20 steps) than the centralized agents. Specifically, in the centralized mode, a0 does not move much after pressing b2. Conversely, in the decentralized mode, a0 presses b1 after pressing b2. To sum up, C and D use both a collaborative strategy but D is slightly less efficient, and S uses a totally different strategy (that can be considered the least collaborative).

Based on this subjective description of their behavior (which you can also check by running *main.py -run ...* ), it was expected that numeric comparisons should highlight (1) higher differences between C vs S and D vs S than between C vs D. Moreover, it would be expected that (2) C vs S present a higher difference than D vs S. All the measures support the first expected result. However, the only measure that mirrors the second expected result is the comparison of rewards.

### Scenario 2

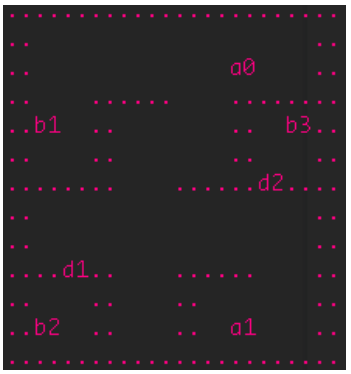



In the second scenario, no player is blocked in a room and there are two target buttons. Each of the target buttons is much closer to one of the players than to the other. As the target buttons are two and they are equally accessible to the players, to solve this level players can act in parallel, such as in two independent sub-tasks.

The behavior of the three types of agents is generically the same, i.e., a0 presses b1 and then b3, while a1 presses b2 and then b4. They slightly differ in the number of steps (C - 18 steps, D - 19 steps, S - 13 steps), but the expected numeric differences should be negligible (or zero by increasing the training time).

All the measures support that the behaviors of the three types of agent act similarly.

### Scenario 3




In the third scenario, there is only one target button and cooperation is advantageous to reach the goal slightly faster. In three modes, the learned behaviour generically corresponds to a0 pressing b1, a1 pressing b2 and then a0 is always the fastest to reach the target button b3. As pointed in Subsection Training, the centralized agents should train for a higher number of episodes. Currently, they trained for 2000 episodes and they reach the goal in 37 steps, but especially a1 is not acting optimally and chooses NOTHING before even pressing b2. The optimal solution should take 32 steps.

In both C and D, a0 is going towards b1, at the same time that a1 is going towards b2. However, in C mode, a1 stays still after pressing b2, while a0 is going towards the target button. Conversely, in D mode, because the agents do not know the position of the other player, they both go towards the target button (34 steps). In the S mode, both players start the level going towards b1, which is visible because a1 starts moving upwards in the central corridor before a0 reaches b1. As soon as d1 is open, a1 goes towards b2 and then both players go towards b3 to finish the level (34 steps).

Because C did not train for enough time to act optimally, it is hard to infer the best numeric measure to compare the behaviors in this game level. However, overall all measures suggest the three behaviors were similar and close (lower differences), which is indeed true.

# Other Notes

## How to measure cooperation?

By measuring the difference between two types of agents we get a sense of how close or far their behaviors are. But that does not tell much about the degree of cooperation of the level. In the first scenario, the differences in all measures were high suggesting a discrepancy

between the centralized behavior and any of the other two (although a higher difference when comparing with the single agents). But this was only “possible” because in that first scenario, one of the agents is trapped. Therefore, when the agents are more individualistic, as long as they don’t unlock the other agent, the behavior will be completely different.

The opposite example is the third scenario, in which no agent is trapped and even if they have learned to solve the puzzle alone, they don’t start looking for the same button (due to their proximity to the buttons) and end up cooperating the same way as the centralized agents do. My point is scenario 3 is also prone to cooperative behavior (in a subjective sense) but the behaviors of the agents end up being similar and we have no objective way to quantify it.

In other words, deploying two individualistic agents at the same time might still end up in a cooperative behavior, and the current approach to measure cooperation might still not be ideal to capture it. I was thinking if it would make more sense to execute the single agents alone rather than jointly. Another idea worth trying (to explore whether these measures are sensible) is exploring levels with different proximities to buttons and doors between the two players.

## What is cooperation?

It is not that simple to define cooperation or cooperative behaviour. Based on Bruno’s MSc thesis, the cooperative behavior should be reached with centralized agents (that basically learn the policy of two player jointly and with full observability), while the opposite behavior, non-cooperative or individualistic, should be reached with the decentralized agents (that basically learn one policy for each of the two players with partial observability because they don’t know the position of the other player). However, the way the rewards are implemented highly affect the behavior learned by the decentralized agents. The way I implemented it was similar to the centralized agents and produced similar cooperative behaviors (although sub-optimally). Bruno’s implementation was actually slightly different and resulted in a more individualistic behaviour. For that reason, I implemented the single agents, which learn their policies alone in the environment (independently of the other players). Rui Prada mentioned comparing centralized agents with single agents, might be a good approach although it considers a specific type of cooperation (see Steiner’s Taxonomy of Tasks).

Another relevant consideration regarding cooperation is that these three scenarios proposed by Bruno Carreira, were adapted from a single player game. The labrecruit game was single player and Bruno adapted it to be played by two players. However, these three examples might not mirror levels where cooperation is needed to finish the game level. Cooperation might be advantageous because players might together reach the goal faster compared to when they do it alone.

## Number of Goal Buttons

While I was writing the discussion section, I realised the number of goal buttons might affect the impact of cooperation. It might also depend on the locations of doors and of other buttons, but two or more goal buttons are prone to create independent sub-tasks among players (as seen for scenario 2). While having just one goal button (as in scenarios 1 and 3) may change the type of collaboration required, i.e., do the players need to synchronize? Or can we see “pressing door buttons” as different tsub-tasks? Maybe this issue is related with

the previous one discussing cooperation types and what does it mean to have cooperative and non-cooperative behaviors.

## Human traces

Finally, one last note is that I end up not using the human traces that Bruno Carreira collected in his user studies. The logs are in the github rep. The policies can not be taken from the traces (may inferred, I don't know…), but the other measures can be used to compare rewards and hitmaps.