Applying Markov Decision Process to Maze Navigator Robot Systems

Haider Abbasi, Izhar Ahmed, Georgios Karanasios, Huw Ranscombe-Smith
School of Computer Science, University of Birmingham, UK, B15 2TT

***Abstract*—The present paper focuses on the application of the Markov Decision Process (MDP), in the concept of the Policy Iteration algorithm, to real robotic systems. To do this, we created a system using the Stage Robot Operating System (ROS) middleware and applied it to a case study in the context of a simulated maze-solving robot. In this context, we highlight the use of MDP for policymaking, implemented by ROS nodes and coded in python programming language. Following the results of our experiments, this paper also describes conclusions about how successful MDPs are in finding the shortest path to different maze maps and how useful they can be for many man-made work environments.**

I. Keywords

Maze-solving robot system, Markov Decision Processes (MDP), Policy Iteration algorithm, Stage ROS simulation, Python programming language.

II. Introduction

NOwadays, in almost every work environment which requires dangerous activities (e.g., working in contaminated, dusty environments, lifting very heavy objects etc.), humans have been replaced by robots. However, in autonomous robotics, the biggest challenge for an efficient navigation strategy design is the guarantee that the robot will move as fast and secure as possible.

The purpose of this paper is to examine the above circumstances in practice. In particular, it deals with a robot system that lets robots find the shortest path inside a map. Because the project was implemented in a simulation environment, a maze map traversal is a great abstraction for many man-made environments that robots would be expected to operate in, such as warehouses or industrial sites.

Moreover, the robot system is based on information and strategy around Markov Decision Process (MDP), especially around the Policy Iteration technique, for a single-robot navigation algorithm where the robot can be considered as a dynamic obstacle. That is, the robot system can perform independent actions. After the selected action, the relationship with the target is evaluated, and rewards or penalty is given to the robot.

In the following sections, relevant works from other researchers will be discussed and the system's architecture will be analysed in detail. In addition, the performance of the robot application code was evaluated through a variety of experiments, which will be scrutinised along with the results at the end of this report.

III. Related work

As mentioned above, the subject of this report is based on the optimal finding of a robot path using the Markov Decision Process (MDP). In this field, there are lots of research projects that are focusing on the same methodology to achieve different goals. For instance, the research by Hartley, T., Mehdi, Q. and Gough N. [2] describes the process of applying the MDP algorithm to a 2D real-time computer game called Gem Raider (similar to Pac-man). In this game, the hero must collect all the gems from the map by avoiding every obstacle or the enemies.

The main difference between the two projects is the goals that the actor must achieve. On the one hand, the actor of the Gem Raider game (a Non-Playable Character, NPC) needs to stop the player from collecting the points and completing the game. The NPC has to chase the player while avoiding the obstacles. On the other hand, the maze problem includes just one robot that must find the shortest path inside the map.

However, the implementation of MDP is very similar for both projects. As it has been already applied to the Gem Raider game, the maze is split into a set of states and the robot must move from one state to another until it reaches the terminal state. Each state transition is dependent only on the current state that the robot has observed and the actions that he has chosen. In other words, every robot's decision is described by a set of transition probabilities between states after any given action. Also, in order for the robot to find the shortest path, each state (or cell), in the map, has a reward value. Therefore, if the robot e.g., hits the wall, it will receive a negative reward but if it moves to a cell that is getting it closer to the exit, it will receive a positive reward.

Another related work is Mochan Shrestha's paper [3] which explores different approaches to replicating MDP policy. It has been observed that policy iteration selects the best policy that has the steepest performance gradient for the next repetition. This approach is useful when actions in different states are correlated and a standard policy iteration cannot be applied. In this, many other questions need to be answered e.g., when the function converges and under what conditions it finds the local minimum. For example, in the game Rock, Paper and Scissors, there are three modes and three actions (the hero and opponent can play any of the above). The reward is given +1 for the victory, 0 for the draw and -1 for the defeat of the game. This game helps us to conclude how the policy gradient is used to improve the policy.

In addition, research into other MDP algorithms was considered necessary. Littman's soccer domain [4] uses Q-learning

to find the optimal policy with an extra action state. This game consists of grid mapping, and two players who can take actions (down, up, right, left, stand) simultaneously one after the other. In this game, there are two players, one having the ball called as an attacker and the other one known as a defender. The attacker drops the possession of the ball by running into the defender box, if a player goal the ball (scores the point), then the board resets, and the other player got the ball. Goals are given one point worth similar to the reward of our project's terminal state, a negative point for self-goal.

Although two-player Markov games are a genuinely confined class of multi-specialist conditions, they are of free intrigue and incorporate Markov choice cycles as an extraordinary case. Applying experiences from the hypothesis of helpful furthermore multi-player games could likewise demonstrate productivity in spite of the fact that observing valuable associations might be testing. Therefore, taking the Q-learning technique into account, the project can use it as a comparison example with the Policy Iteration.

IV. Algorithms and Framework

The project system is based on the Policy Iteration algorithm belonging to the Markov Decision Process (MDP) mathematical framework. This framework consists of an agent and an environment.

For our environment, we have different kinds of mazes, one of them is a square maze with different types of cells (e.g., cells that are free or occupied or define the exit of the maze - target cell). For example, we take a rat (agent or robot) and cheese (target cell) maze model. In this, the rat would be "encouraged" in order to find the cheese, with the shortest distance which could be possible.

The steps which are available for the agent would only be right, left, up and down. The agent only lives in the grid system, and the walls are the boundary of the agent, and it gives direction to the agent. As our main goal is to make the agent reach the target cell in minimum time and the minimum shortest distance.

Regarding the Markov Decision Process algorithm, it consists of "states" which are a set of tokens, which means that the whole system consists of many states. These states can be aligned in x and y coordinates. Then we have the action state, which means the possible direction available for the agent. Moreover, the algorithm provides a transition function that consists of the current state and action and the resultant state. In other words, it provides the probability of reaching the resultant state.

Through a mathematical aspect, the above state can be expressed as:

$$s' = T(s, a)$$

Where s is the current state, α is action state and s' is the next state. Note that the current and next state can be the same one.

The probability of ending up in the state s from state s' after the action α, is given by:

$$0 < T(s, a, s'), < 1$$

Note that for all actions α and states s and s' the sum of the probability is given by:

$$T(s, a, s') = 1$$

The transition model is important because it describes the rules of the game, it tells what we have to do next and what are the possible outcomes of it. As the agent changes its start state with every move i.e., the next state is the new start state.

The last thing in the MDP is the reward function:

$$r = R(s, a))$$

it is useful as it seems to be the domain of knowledge. There could be different kinds of rewards like a reward for entering into the state, the reward for being in a state and taking an action, a reward for being in a state and taking an action and ending up in a new state. But the main reward we have, reward for entering into the state.

The above statements are combined and declared as the problem. Now, the solution for MDP is called policy:

$$\pi(s) \rightarrow a$$

It is a function that takes a state and returns an action, which means for any given state in which the robot or agent is, it tells the robot to take an action. We have a special policy called optimal policy π*, which is a policy that maximises the long-term reward.

The resultant total reward can be written as:

$$A = R(s1, a1) + (s2, a2) + (s3, a3) + \ldots\ldots\ldots + (sn, an)$$

Policy iteration starts with an arbitrary policy $\pi 0$ which is an approximation to the optimal policy that works best. Then, the process can be split into two stages to obtain a sequence of monotonically improving policies and values [5]:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

where $\rightarrow E$ denotes a policy evaluation and $\rightarrow I$ denotes a policy improvement.

Policy evaluation [5] calculates the state-value function $\upsilon\pi$ for an arbitrary policy π:

$$\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big]$$

where $\pi(a|s)$ is the probability of taking action α in state s under policy π. The existence and uniqueness of $\upsilon\pi$ are guaranteed as long as either $\gamma < 1$ eventual or termination is guaranteed from all states under the policy π.

Policy improvement [5] finds better editions of the above policy in case that there is a new action which for some state improves the expected value. In other words, to consider the new greedy policy, π', given by:

$$\max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_{\pi'}(s')\Big]$$

where argmaxα denotes the highest / maximized value of action α (with ties broken arbitrarily).

Overall, the combination of the above two processes completes the policy iteration pseudocode [5] (Figure 1) used as the main prototype for implementing the project code.

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$

3. Policy Improvement
   policy-stable $\leftarrow true$
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 1 - Policy Iteration Pseudocode (using iterative policy evaluation) for estimating $\pi \approx \pi*$

V. Implementation

For our code implementation, we used Python as the main programming language, and it is structured based on the Object-Oriented Programming paradigm. The final project's source code can be found on GitHub [1].

Also, we decided to use ROS Stage as the simulator for our robot. This provided adequate facilities to run our algorithm and test our robot, without adding any unnecessary complexity. Particularly, the decision to use Stage with no noise added to the movement of our robot, allowed us to simplify many of the calculations in our algorithm.

For the first piece of the work was necessary to synchronize the map file (yaml and world files). This ensures that the coordinate system for the robot and the occupancy grid line up. To do this, the resolution in the yaml file was set to 0.1 (every pixel is 0.1 meters). In the world file, a few variables needed to be set to line up the coordinate system. The floor plan needed to have its dimensions defined. In the case of our first maze, the image is 198x196 pixels. Considering our resolution defined in the yaml file, the dimensions for this maze are 19.8x19.6m. We then need to offset this by half of the distance, so in the case of the first one, the offset will be 9.9m and 9.8m. This ensures that the origin is set to the bottom left corner.

As soon as the files of the maze environment have been successfully synchronised, loaded to ROS Stage and the robot starts from the desired origin inside the map, the next step was to write the first half of the code. The first part receives all the necessary information from the environment and responsible for this is the $subscriberNode$ class. In particular, the Subscriber executes the listener function in order to subscribe to the Occupancy grid and Odometry topic. In this way, we can make call-backs and access the data referring to the maze's occupancy grid values and the robot's position and orientation.

Then, the $maze\_dictionary$ python script comes in the background. The main idea was to collect all the information needed for the policy iteration and access them at once by generating a dictionary. Therefore, during the Subscriber's call-back of the Occupancy Grid topic, the maze's dictionary is created including information such as each grid cell's coordinate indexes and values, reward and value of the state, policy direction etc. Each grid cell has 0 probability if it is known, -1 unknown and -100 if it is an obstacle, thus, using that information we can define the corresponding reward for each case. Also, in order to define the terminal stage point, we needed the actual coordinate values of each cell, which from the mathematical perspective, are calculated using the resolution info of the grid. Note that each cell of the occupancy grid is the size of a single pixel.

We originally converted this directly to the entries within our dictionary, however, this resolution was too detailed for our robot to follow accurately. To remedy this, we made each 2x2 set of cells within the occupancy grid act as one entry in the dictionary. They effectively reduced the resolution of the grid used to calculate the policy, which in this case made it easier for the robot to follow accurately and had the added benefit of improving the policy calculation time.

The final part of the implementation of the maze grid is the $wall\_buffer$ function. An issue we noticed during testing was that the robot would frequently attempt to hug the wall, as this would be what it deemed to be the shortest path. The issue is that the robot is not a point object, so it would collide with the walls when doing this. To correct this, we added a function to add a buffer around the walls. After the initial dictionary has been created, the function simply loops through each entry. If that entry is a wall, then every entry surrounding it is given a large cost like that of the wall. This discourages the policy from calculating a path so close to the real maze walls. The buffer size can be set to control the strength of this effect.

On the other half of the code, the $publisherNode$ class plays the main role. It is responsible for navigating the robot to the terminal state by receiving all the path instructions from the policy iteration.

The implementation of the policy was handled in its own file $policy\_iteration.py$. Within this file, there are three functions: $policy\_generation$, $policy\_evaluation$ and $policy\_iteration$. The first two handle the main two parts of policy iteration, and the third links the two together. This means that only the third function is intended to be called by the other files.

$Policy\_generation$ takes a maze dictionary and calculates an appropriate policy for it. This is used to generate both the initial policy and recalculate the policy after the evaluation step. First, the function loops through each cell. For each cell, it does the following. The algorithm looks at each of the four surrounding cells and calculates their value. It then decides which neighbouring cell has the largest value. If two or more are tied for the largest value, one is picked at random. This is then decided to be the direction of the policy, which is assigned to the dictionary.

There are a couple of nuisances for this function of worthy note. For the calculation of the value, we have decided to omit

the transition function. This is because we are not simulating any noise for the movement of our robot on stage. As a result, the transition function for every movement should be 1. This means that it can be left out of the calculation for simplicity. If we were to continue the project at a later date, this would be one of the first features we implement. The other feature of this algorithm is the process is skipped if the cell being checked is in a terminal state. The policy for these cells is irrelevant, so there is no need to calculate it here.

$Policy\_evaluation$ is the second half of the algorithm. It loops through each cell in the maze grid dictionary and calculates its value using the policy. By using the policy, only one value calculation is calculated for each cell. There is a variable called theta which controls how many times this process should be repeated. The largest value of each loop is recorded in a variable delta. When delta is smaller than theta, the values are considered to have converged and the loop terminates. Each cell in the dictionary has the calculated value stored with it. As with the $policy\_generation$ function, the transition function is omitted from the value calculation due to the assumption that it is 1 for every transition and therefore redundant.

Finally, the $policy\_iteration$ function ties these two together. First, it generates an initial policy for the given cell grid. It then enters a loop with the following conditions. The loop repeats until either the policy is stable (there is no change when it is regenerated) or a maximum number of iterations is reached. The latter was included to reduce the run time of the algorithm. It is unclear how consistently the policy will converge completely; we believe that due to the random element of the policy generation function it may have converged but remain oscillating between two states in a small portion of a map. However, it seems from our testing that allowing the algorithm to run for a sufficient number of iterations ($\sim$1000) converges enough to provide a usable policy. Within this loop, the evaluation and generation functions are alternatively called. When the loop is broken, the optimal policy is considered found and the dictionary containing this result is returned.

Subsequently, the final dictionary of the policy iteration will be passed to the $talker(publisher)$, and it will be used by the robot movement to navigate the maze and reach the terminal state. The robot's movement is implemented by the mover function which constantly receives from the Subscriber the robot's current position and heading (the real orientation given by the yaw in the odometry topic) and the path instructions from the policy dictionary. In other words, for each time step, the subscriber node tells where the robot is on the map and based on that, the Publisher checks the dictionary what is the next move to publish.

The motion procedure has been split into two simple stages. First, the algorithm examines the current navigation of the robot and based on where the robot heads, it rotates the robot with the required degrees in order next to move North, South, East or West. Note, that the desired angular speed z is calculated by the control rotation speed kp multiplied by the difference between the target rotation tr (converted in radians) and the current angle of the robot yaw [6]:

$$z = kp * (tr - yaw)$$

Note that if the difference is big, the robot will rotate faster.

For the smoother motion of the robot, it will be able to move one step forward only when the rotation has finished. In summary, the $talker$ implements the whole procedure to ROS Stage environment iteratively and stops as soon as the current position of the robot is in the terminal state.

The last and also very important part of the implementation code is the $main.py$ file. It is the connecting link as it centralizes the Publisher and Subscriber nodes into one. Primarily, it calls the Subscriber's $listener$ function to collect all the required data for the policy iteration and robot motion (e.g., maze dictionary, robot current orientation and position etc.) and transfers the current Subscriber object which has all these recent data as a parameter to the Publisher's $talker$ function.

VI. Experimental Setup

There are two major aspects that we will need to evaluate for our implementation: the primary being accuracy and the secondary being speed. The whole purpose of this project is for the robot to be capable of determining the fastest path through a maze. If the route it decides upon is wildly inaccurate then it will have been a failure.

However, speed will be an important factor too. We want the robot to operate in real-time. If it ends up hanging on the maze calculations for a disproportionate length of time (e.g., it traverses the maze in less than a minute, but then takes over ten minutes in its calculation) then it proves our implementation to be impractical.

When we create the mazes, we will know the layout, as well as the start and goal locations. With this knowledge, we will know the ground truth shortest path. Once our robot has determined the shortest path using reinforcement learning, we will compare these two results and give a percentage difference. Ideally, there should be no difference, but small differences in the route that are the same length (e.g., "going up and then left" compared to "going left and then up") may occur.

The second measure of success will be more arbitrary. We are yet to decide what speed for the algorithm we will consider acceptable, but it could be something along the lines of calculating the shortest path that can take no more than double the time it took to traverse the entire maze, for example. We can also compare the performance of MDP with different types of reinforcement learning, such as q-learning.

Finally, we will be observing the robot using Stage package during runtime. This will enable us to make some visual comments on its performance. It will be interesting to see how it behaves, if we can spot any quirks in its behaviour that are clearly inefficient, or if it challenges our intuitions of what we expect to happen but still performs well by our other evaluation standards.

VII. Experimental Results

We conducted multiple experiments to evaluate the performance of Policy iteration algorithm. We chose to vary the map size, complexity and reward/terminate states to evaluate performance. The algorithm is used to find the shortest path in multiple environments. Initially, we tested the robot one maze with the terminal state in different positions but mostly set it in the centre (because it was observed as the most complicated/difficult path). This was done to ensure that different aspects of the environment are tested, for instance having multiple corners in the robot path, turning in different directions. This test was helpful in determining how effective the robot movements are for various rotations. The robot performed well in these scenarios, as seen in our observations of its performance on stage. Figures 2 and 3 showcases the different maps that the robot is tested on and their respective end state.

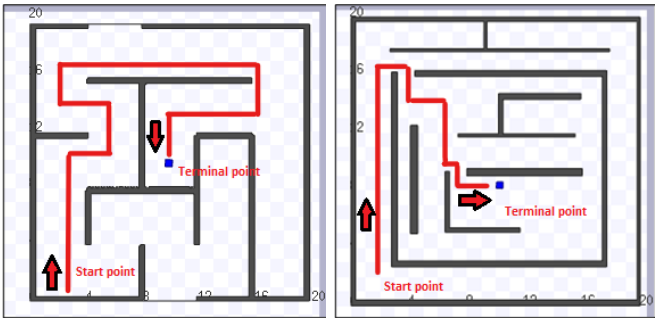

Figure 2 and 3 - Experimental results in different mazes

The performance of the algorithm varies from map to map. The bigger the size of the map, the longer the algorithm takes to converge. The algorithm does not always fully converge; however, it is still capable of finding a sufficiently optimal path. For the smaller map fewer iterations to achieve this in comparison to the larger maps. Once the optimal policy is found the robot, it quickly reaches the goal state when executing the policy. The computation time is measured by taking the difference in the start and end of the algorithm runtime. As seen in table 1, the policy computation takes more time than robot movements.

Maps (2000 Iterations)	Policy iteration	Robot movement in map
Map 1 Maze	423.58 sec	230.sec
Map 2 Room	412.80 sec	93.49 Sec
The time may vary based on device the algorithm is tested on.		

Table 1 - Experimental results on the performance of policy iteration algorithm and robot motion, tested on different maps

This has been tested by running the policy on maps with different layouts. In total, three maps were tested and in two maps the algorithm worked well, and robot movement worked accordingly. In some maps, the robot does face a few problems where the gap between the walls is very tight this is due to the rotation as there is no room for the robot to rotate.

As we are aware of the ground truth, we can observe from the robot movements in the map that it does find the shortest path to some extent but there are variabilities in terms of robot movements. Looking at individual cells it might not perform well but overall, it does find the shortest path.

Different terminal states were tested to ensure the policy generalises well. The change in the position of the terminal state does not have any effect on the performance. The robot can reach to terminal state from any state once the initial policy is computed. This has been tested by placing the robot in different locations and it does reach the final state.

The above testing case was also applied with different values regarding the control rotation speed kp and linear velocity x of the robot, $kp = 1.7$ and $x = 0.5$ decided as the most suitable.

Additionally, during the experiment process, the $help.py$ script was created in order to write functions that will make the testing process much easier. Precisely, once the policy is generated, it has been saved to a pickle file so we can use the latest policy for testing without running the policy iteration from scratch (working with the same terminal and any initial state). Therefore, since the algorithm takes a long time to find the policy, saving it in a pickle file made the testing process much faster. More functions such as the $test\_policy$ function and $save\_to\_text\_file$ function (for better quality in the analysis view) are also part of the $help.py$ script.

VIII. Conclusions and Future Work

In conclusion, the main idea of this work came from the fact that humanity needs robots that can perform a variety of tasks in a dangerous work environments. Part of this process is the ability of the robot to go to the right place following the shortest path. Thus, the aim of the project was to explore the capabilities of MDP through the Policy Iteration algorithm and apply this technique from scratch in a simulated robot system that solves a maze. Relevant researches refer to games such as Gem Raider and Rock Paper Scissors where the common key is the use of positive/negative rewards and value states which help the hero to reach his target.

Besides, the report described the most important parts of the code implementation, emphasizing the combination between the subscriber node and the maze creator in addition to the Publisher node and the mover function which follow the path instructions of the policy iteration dictionary. At last, most of the suggested experiments are conducted, the accuracy of the algorithms is evaluated, the layout of the different maps is tested, and robots perform well in most scenarios.

Further work in this area will involve exploring other methods of building the maze such as using the SLAM algorithm. Another interesting aspect would be to convert the maze to a hide and seek game between two robots. As soon as the algorithm performs well enough in real-time, it could potentially be used in a scenario where one robot attempts to chase down another within the maze.

Another improvement would be the introduction of the transition function into the cell value calculations. As stated earlier in our report, we decided to omit this for simplicity by assuming perfect transitions. This is realistic as the ROS Stage simulator has no noise in the motion of the robot. In later versions, we would include the transition function to our algorithm, and introduce artificial noise into Stage to reflect this.

Furthermore, running different MDP pathfinding algorithms (e.g., comparing the Q-learning algorithm with Policy Iteration) on different maps and comparing their shortest path result, would give us a better idea of how effective our policy code is.

Finally, the algorithm could be expanded by finding the quickest route between multiple points. In this way, MDP can show its real strengths as it will provide multiple-goal segments providing points, for which a path between can be optimised.

REFERENCES

[1] Abbasi, H., Ahmed, I., Karanasios, G. and Ranscombe-Smith, H., 2021. Intelligent Robotic Group 18 Final Project. [online] GitHub. Available at: ⟨https://github.com/hayderab/Group_18_finalProject_InteRobotic⟩ [Accessed 14 December 2021].

[2] Hartley, T., Mehdi, Q. and Gough, N., 2012. *Applying Markov Decision Processes To 2d Real Time Games*. The Research Institute in Advanced Technologies (RIATec), University Of Wolverhampton, School of Computing and Information Technology.

[3] Shrestha, M., 2020. Policy Gradient Theorem for One Step MDPs. [online] Mochan Shrestha. Available at: ⟨http://mochan.info/reinforcement-learning/machine-learning/2020/06/23/PGT-for-One-Step-MDPs.html⟩ [Accessed 9 December 2021].

[4] Littman, M., 1994. Markov games as a framework for multi-agent reinforcement learning. [online] Courses.cs.duke.edu. Available at: ⟨https://courses.cs.duke.edu/spring07/cps296.3/littman94markov.pdf⟩ [Accessed 9 December 2021].

[5] Sutton, R. and Barto, A., 2021. Reinforcement Learning: An Introduction. pp 74-81 [online] Incompleteideas.net. Available at: ⟨http://incompleteideas.net/book/RLbook2020.pdf⟩ [Accessed 14 December 2021].

[6] Yagüe, J., 2019. Learning player abilities based on multimodal data. [online] Projekter.aau.dk. Available at: ⟨https://projekter.aau.dk/projekter/files/312865554/Master_Thesis_190930_JorgeVilla.pdf⟩ [Accessed 7 December 2021].