OULUN AMMATTIKORKEAKOULU

Joonas Ruotsalainen

COMPARING PATH-FINDING ALGORITHMS AND MACHINE LEARNING MODEL

Comparing A-Star algorithm, Q-learning and PPO in game development perspective

COMPARING PATH-FINDING ALGORITHMS AND MACHINE LEARNING MODEL

Comparing A-Star algorithm, Q-learning and PPO in game development perspective

Joonas Ruotsalainen
Final projects
Winter 2024
Degree Programme in Modern
Software and Computing Solutions
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Modern Software and Computing Solutions

Author(s): Joonas Ruotsalainen
Title of the thesis: Comparing path-finding algorithms and machine learning model
Thesis examiner(s): Teemu Leppänen
Term and year of thesis completion: 2024 Pages: 72

This thesis focused on comparing A-star algorithm and some of its variants against Q-learning and Proximal Policy Optimization algorithms in terms of path finding and in game development perspective. Both Q-learning and proximal policy algorithm are reinforcement learning algorithms which is a subsection of machine learning. The goal of the thesis was to analyse the viability of using reinforcement learning in path finding in game development instead of traditional algorithms and discover the strengths and weaknesses of each method and possible future developments. The thesis subject was a personal topic of interest of the writer. At first the thesis introduced different kinds of path finding techniques in game development like navigation mesh and way-points based navigation. It described the A-star algorithm in detail using pseudocode and compared the standard algorithm to different A-star variants, for example D-star lite. Next, the author described the basics of machine learning, neural networks and reinforcement learning using Markov Decision process before going deeper into Q-learning algorithm and proximal policy algorithm. After this the thesis described experiments done using a Minigrid library in order to gather knowledge and data regarding the differences of the algorithms. During the thesis a simple simulation game was developed where a deep reinforcement learning agent needs to navigate though a simple maze from start to end position. The simulation game was used in experiments to gather knowledge and performance data of the algorithms. The chapters described the different types of scenarios developed and the results for each algorithm. At the end of the thesis the author had conclusions of the results and ideas for future development. The main conclusion was that using reinforcement learning for path finding is not viable because of the complexity and cost of the training. A-Star was also significantly more performant finding the path. However, author suggested studying alternative deep reinforcement learning algorithms which might yield better results. Author also explained that it might make sense to use reinforcement learning in other areas of the game development.

Keywords: Neural network, path-finding, machine learning, reinforcement learning

CONTENTS

1 INTRODUCTION

AI and Machine learning have been on the news in 2023 and with the introduction of different AI tools to generate text, images, and process data we are in a verge of change in our personal and working lives. The different kinds of AI tools will be replacing and changing jobs in the future and causing disruption, both good and bad, in work life, organizations, governments, and businesses. New jobs will be created and some types of jobs will disappear. Things that were extremely difficult to develop a couple of years ago are now possible and achievable with the help of AI related tools. Programmer's productivity can be increased by tools like ChatGPT for code generation, searching for information and learning new concepts. Large language models which are developed to generate and process text can understand complex structures and extract data by providing instructions. It is important to gain knowledge, understand and learn about the AI and machine learning.

By providing details of your skills, job history and education with a job application text to a large language model AI, it could generate you a tailor made CV and a cover letter written specifically for a certain job description. The AI tool to generate this could even search the internet for additional information of the company to make the letter even more specific for the job, thus saving you time and effort.

On the other hand, criminals or scammers may use AI tools to generate specific scams or phishing attempts with good written language and in an automated way. You initiate a chat with an AI tailored to chat with you and trying to get your personal information for criminal purposes which previously needed a real person.

If you are a real-estate agent you may take photos of empty houses and flats for sale. You could feed the photos to an AI which modifies the photos by adding furniture and other decoration. AI could also generate sales pitch based on different details like location, amount of rooms and price. In the future it could even "look" at the photos you took and make a sales pitch based on that.

Self-driving cars have developed a lot in the past 10 years. Starting from basic warning systems like lane-departure alarm we are now almost in a state were a human driver does not need to pay attention to the road and traffic. An AI receives constant input from different sensors of the car and makes decisions based on those. In the future AI probably replaces delivery services, taxis, public transportation, and we will have fully automated order delivery from e-commerce website to your front door.

Software developer could take raw data from a database and feed it into a large language model asking to generate code for a user interface component which visualizes the data. Testing the code could be done by AI analysing different code paths and functions and writing automated unit tests. Old code could be refactored into a better code base by asking AI to rewrite it using different libraries or even programming languages.

In game development game prototypes could be developed faster by generating game assets and graphics using generative AI. Games could be more interesting when some parts of the gameplay are generated dynamically by AI, new story-lines are AI generated for each playtime and dialog with game characters could be more realistic and human like. Instead of scripting all dialog game designers could describe characteristics, mood, and relationship to other characters. AI could then generate dialog dynamically and respond to players input. AI could be also used to provide feedback to the player. Aimlabs is a company which develops a training software for players who like to play first-person shooter type of games. The software includes an AI feature called Discovery. “Discovery is aware of your game-play at all times, and is able to give you immediate feedback after tasks, in addition to generating personalized training tasks to help you with specific aspects of training.” (Aimlabs 2023).

Figure 1: Game asset generated using ChatGPT with following prompt: Could you create me an illustration of a fantasy world for a computer game? The game is an adventure game where player controls group of human like characters. The characters could make spells, use bow and arrow and use swords and axes.

Gameplay could be also improved by reinforcement learning. The algorithm could learn the moves of the player and adjust the gameplay difficulty dynamically. Reinforcement learning is a type of machine learning where an algorithm learns from feedback and rewards and by taking actions based on the different states of the environment. This could provide challenging and enjoyable gameplay when the game AI adapts its strategy and difficulty dynamically instead of pre-defined logic introduced by the programmer. One part of this could be path-finding in which this thesis focuses on. Traditional path finding algorithms like A-star are good and efficient at finding paths. The main assumption is that at best machine learning model would get similar results than traditional algorithms. However, machine learning model might adapt better to dynamically changing environments and have a possibility to learn to perform additional tasks like opening doors in a maze or exploring unknown environments.

The goal of the thesis is to study differences of reinforcement learning in path finding against traditional A-star algorithm. It compares the performance, implementation details and evaluates the viability of implementing reinforcement learning in games for path-finding purposes based on the results.

2 ARTIFICIAL INTELLIGENCE IN GAMES

Navigation in a game world is an important part of gameplay and design. Game objects need to be able to move around in the game world efficiently. The world might have obstacles, different kinds of terrains or paths in which the game objects need to interact with. Performant algorithms are often needed to achieve desired features of the game. A single frame in the game loop might perform numerous calculations to update different states of the game. For example calculating collisions and physics, updating object positions based on players input, network request handling, and constructing vertices and other data to be sent to the graphics card in order to draw image to the screen. In order to have smooth playable experience the frame calculation time should not exceed 16.6 milliseconds which equals to roughly 60 frames per second. Lower than 60 frames per second may cause stuttering and increased latency between player input and what happens on the screen which may lead to decrease experience when playing the game. A study published in 2014 investigated a relationship between frame time and latency when a test subject tried to click moving targets on a screen. The study concluded that low frame rates have significant performance cost (Benjamin F. Janzen, Robert J. Teather 2014, 5.). There are also games where the frame rate does not impact much of the experience. For example in the game of chess the board needs to be re-drawn only after a player has made his move. In this case the frame-rate is irrelevant.

“Artificial intelligence, the main field of computer science into which reinforcement learning falls, is a discipline concerned with creating computer programs that display human-like intelligence” (Miguel Morales 2020, 1.). In games artificial intelligence usually means the behaviour of the objects in the game which are not player controlled. For example in a game of chess AI would be the opponent when played against computer.

Reinforcement learning is a section in AI field where an agent is “taught” to choose suitable actions in an environment based on rewards or feedback. The agent tries to select an action which would produce the maximum reward over time. In a game of chess the environment would be the board, pieces and the rules, agent would be either white or black player and actions would be the moves player takes during the game. Reward could be the determined by how good the move was. After each move the agent would receive feedback reward from the environment and update the internal mechanism to select the next action. The figure 2 below shows this basic function of an agent AI.

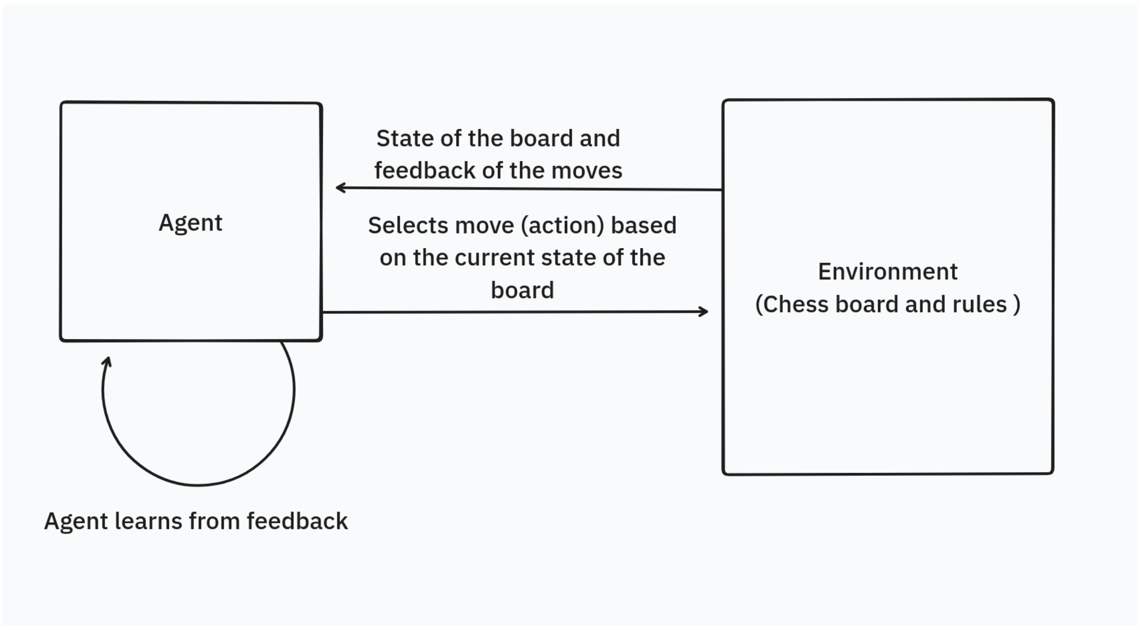

State of the board and feedback of the moves
Agent
Selects move (action) based on the current state of the board
Environment
(Chess board and rules)
Agent learns from feedback

Figure 2: Basic reinforcement learning

The purpose of path-finding is to find a route from a starting position to a desired location, and it’s a fundamental problem in various applications. In robotics a vacuum robot needs to find a path across a room or a taxi driver who needs to find a way to customer’s location when using navigation application. There are different aspects related to path finding. Finding the shortest path might not always be the goal. For example deciding the selection between shortest and fastest path. Selecting the shortest path in a traffic jam might not be the fastest one. A path might need to be recalculated in dynamic changing environment. There might be multiple goals, unknown areas or weighted locations in the graph. For example navigation application might build a path based on weights which selects a more scenic route through nature even when it is not the shortest one. Air traffic might control might build a path based on safety considerations and avoid flying planes inside certain airspace.

Path-finding is closely related to graphs. It involves finding the optimal or efficient path between two points within a network of interconnected nodes and edges. The network may be represented as a mathematical abstraction called a graph.

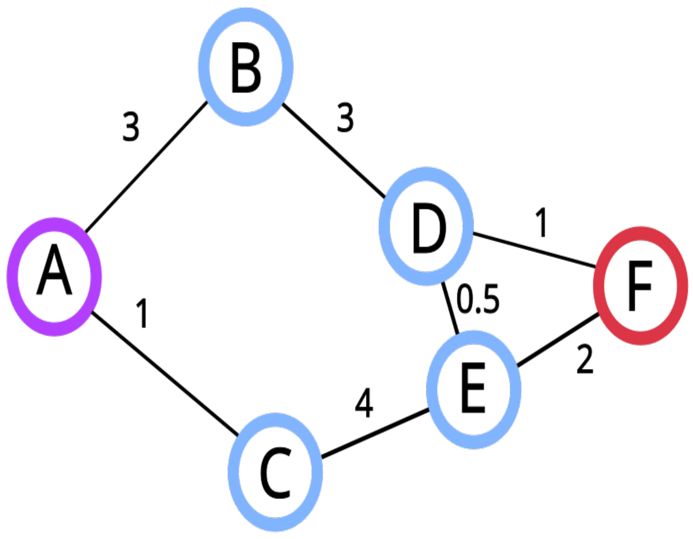


Figure 3: A graph of connected nodes and distances between them.

The image 3 above represents a simple graph. Each letter represents a node and lines are connections between the nodes. The connections are also called as edges and nodes as vertices. The number in the image represents a distance between nodes. The distances are often also called as weights and a graph displayed in the picture 3 would be called a weighted graph. The weight or distance could be considered as a cost for moving from one node to another. In the example A node is the starting position and F the ending position. Shortest path from A to F in this case would be A → C → E → D → F and the distance would be 6.5 units. Another example path could be A → B → D → F which has fewer nodes visited but is longer path of 7 units. It is also possible for weights to be negative. The edges could also have a direction, in which case they are called arcs. For example, you could go from A to C but not from C to A. Graphs with directed edges are called directed graph and the opposite is undirected graph. In the figure 3 above nodes could be considered to be cities and weights kilometres between them, as an example. The figure 3 could also represent an Ethernet network graph where the nodes are computers and edges are network connections between them. A path finding algorithm would have to find optimal route for the network packages.

2.1 Common path finding algorithms

According to a 2020 study “The A* algorithm is the most popular technique applied to path-finding in game development.” (Abdul Rafiq 2020, 8.).

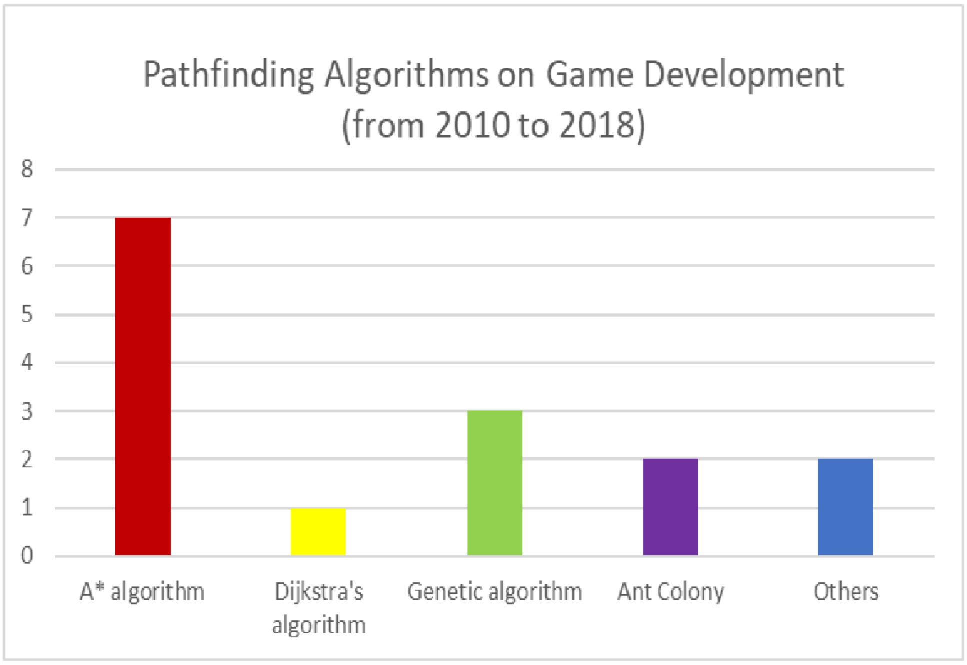


Figure 4: Pathfinding Algorithms on Game Development (Abdul Rafiq 2020, 8.)

The figure 4 above displays most common algorithms in game development based on Abdul Rafig’s study of 10 different game development related published papers between 2007 and 2018. The figure 4 shows that A-star was researched in 7 different papers. The following chapter introduces a couple of common path finding algorithms.

Greedy algorithms make the most optimal decision at each step considering the current situation. Djikstra’s algorithm is a greedy search algorithm which works in weighted directed graph $G = (V, E)$ where all weights are positive. It incrementally builds the shortest path. Heuristics is a problem-solving technique finding approximate solutions which might not always be the perfect but is good enough to solve the problem. Heuristic techniques are often used when classic methods are not performant enough. In case of a path finding algorithms, the heuristic functions usually estimate the cost of the cheapest path. Heuristic methods may be admissible which means that it never overestimates the cost. In the context of graph search heuristic function approximates the distance between two nodes. Djikstra's algorithm is not heuristic and always guarantees to find the shortest path.

Genetic algorithm is a heuristic algorithm inspired by the natural selection in evolution. It is used to find solutions to optimization and search problems. The idea is to build multiple potential paths and evolve them in to the optimal one. It uses techniques such as inheritance, mutation, selection, and crossover. For example in crossover technique two paths could be combined to explore new potential paths. In mutation technique random changes are introduced to the path which might lead to new discoveries of previously unseen paths or optimizations to existing ones. Genetic algorithm is not the fastest method to find a path. However, it is flexible in dynamic environments and adapts by its nature.

Another nature inspired algorithm is called Ant colony algorithm. It is based on the way ants find a path in nature using pheromones. In the first stage multiple potential paths are constructed but iteratively over time the optimal path is followed the most often which reinforces the pheromones and ends up having the strongest pheromone trail. The pheromones evaporate on less optimal paths.

A-star is a best-first search path finding algorithm which uses admissible heuristic methods. Being best-first type of algorithm it searches the graph and selects the most promising node. It does that by using a heuristic function which gives priority to some nodes compared to others, and it guides the algorithm forward towards the goal. A-star is very similar to Djikstra's algorithm but differs by the use of the heuristic function.

2.2 Grid-based navigation

Grid-based environments are based on a simple 2D-grid where the positions on the grid may be distinctly represented using x and y coordinate system. Grid-based navigation is often called also as tile map. Each tile is typically same size and shape. Tile map usually consists of squares but could be also built using hexagons or other shapes. Each node in a square based grid could be connected to either by four or eight other nodes depending on diagonal movement.

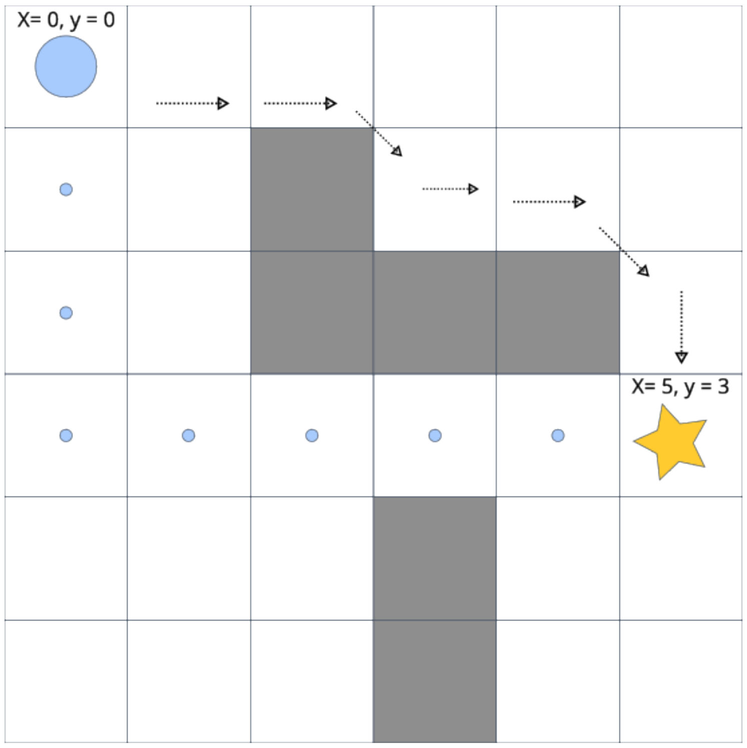


Figure 5: Grid based environment and movement

In the picture 5 above a circle is located in the top left corner and its coordinates are (0x, 0y). A star is located at coordinates (5x, 3y). In 4-way connected grid the movement from the circle to the star would take eight steps and on the 8-way only six steps. The grey tiles are obstacles.

Grids are an efficient and a simple way to navigate because the space is limited to the grid and not continuous and therefore the number of possible locations are finite. The distance between each adjacent tile is typically also equal making distance calculations easier.

2.3 Way-points

Way-points are a set of predefined points or coordinates on the map which help the game AI to navigate. This enables the player or the game developer to manually choose the desired path through the map and a path-finding algorithm might not be necessarily needed. Way-points may be also used to guide the path-finding algorithm to produce a designed path or optimize scenarios where there are multiple nodes and calculations to find paths would be too costly. They could be considered as intermediary goals guiding the agent to the final destination.

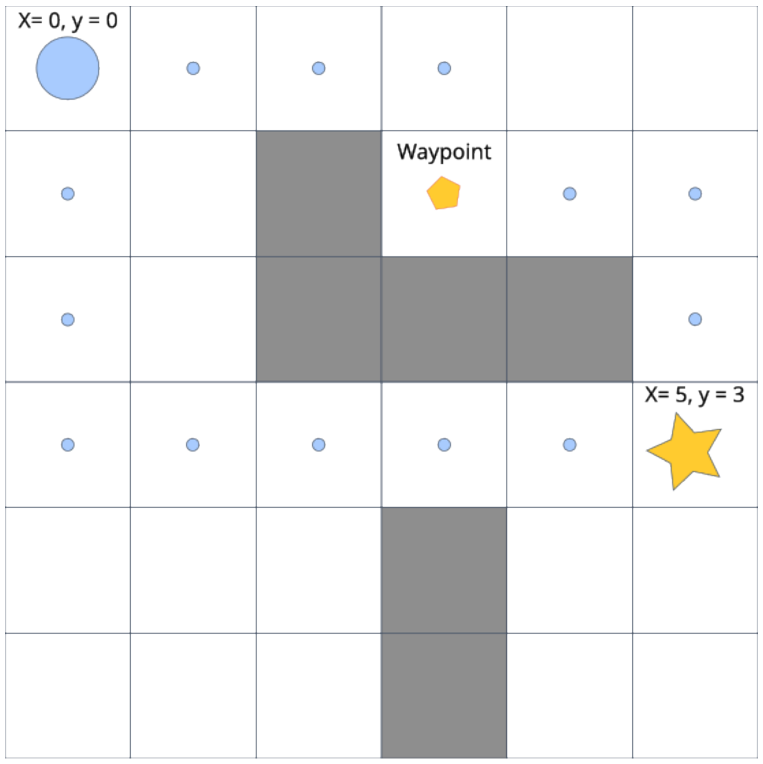


Figure 6: Waypoint navigation

In the image 6 above there are two equal length paths. Agent may be directed to select the upper path using a way-point.

2.4 Non-grid navigation

If fine-grained movement is needed in the map a continuous navigation may be implemented. In this kind of environment agent is not confined into a grid or specific way-points but can navigate freely in the environment. This enabled fluid movement in the environment but increases complexity because of the continuous set of points.

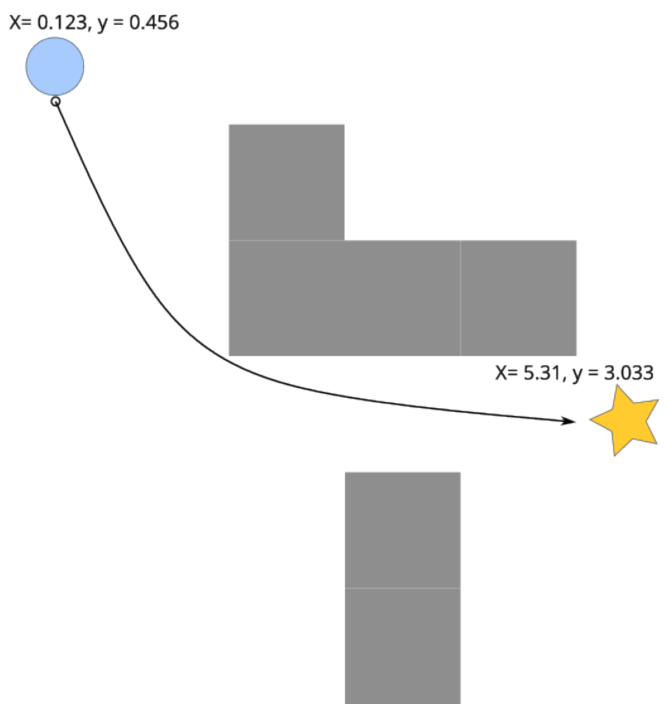


Figure 7: Continuous navigation space

The image 7 above displays a continuous navigation space. A fluid path is drawn from start to finish, and it’s not constrained inside a grid. In the next chapter a navigation mesh is introduced which can be used to reduce the complexity of continuous navigation spaces. Complexity could be further increase by introducing a third dimension and z-axis. The movement would then happen in a 3D-space.

2.5 Navigation mesh

Navigation mesh is often used in a complex environment to reduce the amount of nodes that the path-finding algorithm needs to process in order to find the optimal path. It defines traversable areas for the agent using 2-dimensional polygons which in the end can be represented as a graph. When the navigable areas are per-defined the environment may avoid doing expensive calculations for example collision detection with walls or physic simulations. It also simplifies the navigation space in 3D environments because calculations may be done in 2D space.

Navigation mesh may be also applied to a tile map which reduces the amount of possible nodes and thus, helps to reduce memory usage of A-star algorithm.

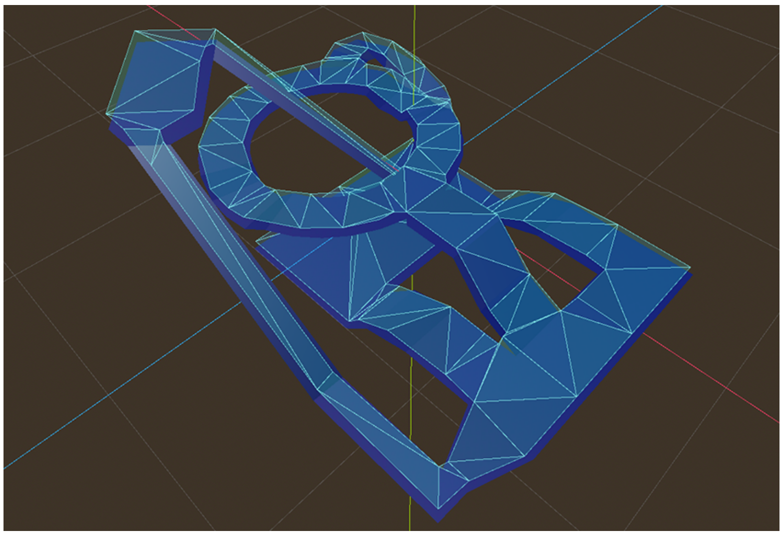


Figure 8: 3D navigation mesh (Godotengine 2023)

Picture 8 above displays a 2D navigation mesh in 3D environment generated in Godot game engine.

2.6 Effect of navigation system to graph based search algorithm

All of these navigation systems described above can all be abstracted and represented as a graph. A-star is a graph traversal algorithm and therefore navigation system does not affect the functionality of the algorithm as long as it can be represented as a graph.

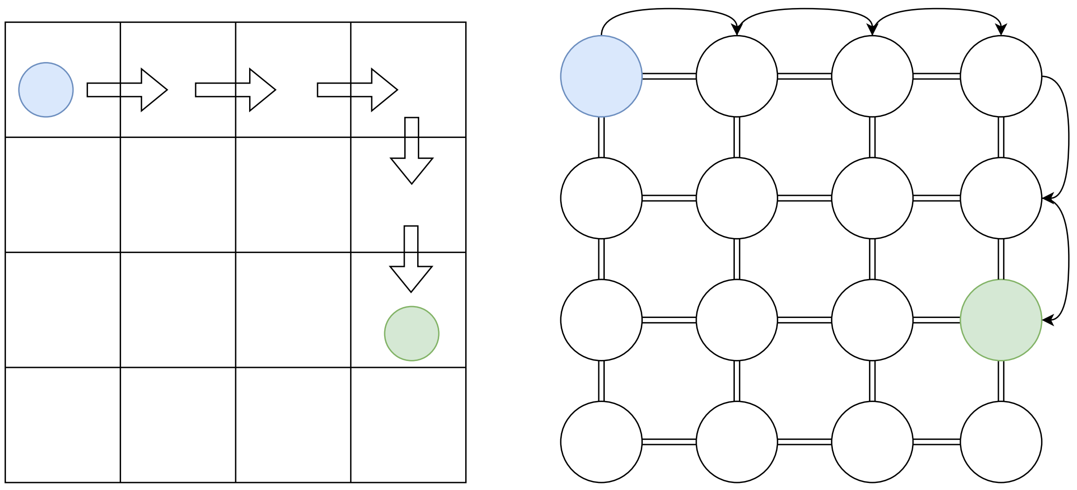


Figure 9: Grid based environment as a graph

Figure 9 above demonstrates that a simple grid may be also represented as a graph. Diagonal movement is prohibited, and each node is 4-way connected. Each square in the grid mirrors a node in the graph and distance between neighbouring nodes are uniform.

2.7 A-star algorithm

The following chapter explains how the A-star algorithm works, different ways to estimate the distance to the goal using heuristic function and some different variants of A-star.

```
1   AStar(start, goal)
2       frontier = priority queue
3       came_from = Map
4       cost_so_far = Map
5       add start to frontier, 0 cost
6       came_from[start] = start
7       cost_so_far[start] = 0
8
9       loop while frontier has items
10          current = get lowest cost from frontier
11          remove current from frontier
12          if current equals goal
13              break from loop
14
15          neighbours = get neighbouring nodes from current
16          loop each n in neighbours
17              movement_cost = get cost from n to current
18              new_cost = cost_so_far[current] + movement_cost
19              if cost_so_far[n] or new_cost < cost_so_far[n]
20                  heuristic = estimate from n to goal
21                  cost_so_far[n] = new_cost + heuristic
22                  add n to frontier with calculated cost
23
24      path = []
25      current = goal
26      loop while start does not equal current
27          add current to path
28          current = came_from[current]
29
30      add start to path
31      return reversed path
```

The frame above presents a pseudo code of basic A-star algorithm and the following chapter will explain the fundamental logic and steps. On line 2 a priority queue is initialized which is a primary component of A-star. Priority queue is a data structure which has a priority associated which each data point stored in the queue. When fetching data points from the queue either the one with the lowest or highest data point is returned depending on the implementation. In the A-star algorithm nodes are stored to the priority queue using movement cost serving as the priority. On line 10 the algorithm fetches a node from the priority queue and the lowest cost is returned. The returned node is also removed from the queue. The design of the priority queue ensures that this operation is efficient and optimized.

On line 3 to 7 the algorithm initializes variables needed for the algorithm. Came_from map is used to construct the path at the end of the algorithm after the search iterations are finished. It keeps track of the visited nodes and which was the node the algorithm came from. Cost_so_far map tracks the cost calculation on each node. A map is a data-structure of key-value pairs where each unique key points to a specific value. Map structure allows for a quick data access by using a key to retrieve its corresponding value. In the pseudo code implementation a node is used as a key.

On line 9 the algorithm starts the loop which continues until the priority queue is empty. The algorithm checks if the loop could be terminated early on line 12. If the current location is the goal node the loop may stop and the algorithm can continue building the actual path. On line 15 the algorithm fetches the neighbouring nodes which are linked the current node. In grid-based navigation system there are either 4 or 8 adjacent nodes depending on diagonal movement as explained in chapter 2.2. Each neighbouring node is then looped through and, movement cost or weight is calculated on line 17. In grid based navigation the movement cost or weight from current node to the neighbour is always 1 but in other types of graphs it would be the weight of the graph edge. On line 18 the algorithm calculates the tentative cost which is the distance from starting position to the neighbour. If the tentative cost is less than the current cost so far the algorithm updates the cost_so_far map variable and adds the current neighbour to the frontier priority queue. On the line 21 the algorithm calculates the cost of the node. Heuristic function is used to estimate the distance between the current neighbour to the goal.

A-star algorithm could be summarized using a following formula: $f(n)=g(n)+h(n)$. A-star selects the node which minimized the f(n) value where n is the current node. The f(n) value is calculated on line 21. In the pseudocode g(n) is the calculation on line 18. The heuristic calculation h(n) happens on line 20.

Heuristic function should never overestimate the distance in order to keep the algorithm admissible. One example to calculate the heuristic function in 4-way grid system is to use Manhattan distance. “The distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $|x_1 - x_2| + |y_1 – y_2|$.” (Paul E. Black, 2019). On the other hand, in 8-way grid heuristic could be calculated using Chebyshev distance.

After the algorithm finishes the main loop it constructs the path backwards by using the came_from map. On line 26 a new loop is started which compares the starting node to the current node from the map. The loop finishes when the two nodes are equal. The current node is added to a path array defined on line 24 and on the next line the variable which holds the current node is updated to the next node from the map. On line 30 the starting node is added to the path as first step in the path and finally the full path returns in reversed order. Including the starting node and reversing the path are optional steps at the end of the algorithm.

Heuristic function can be used to affect the behaviour of the algorithm. For example there might be multiple equal paths to the goal. Adjusting the heuristic function and, it’s return value the algorithm could be guided to select a desired path. Multiple variants of A-star algorithm exists. Iterative deepening A-star is a variation of the algorithm which has lower memory consumption than the standard algorithm. “It works by always generating a descendant of the most recently expanded node, until some depth cutoff is reached, and then backtracking to the next most recently expanded node and generating one of its descendants. Therefore, only the path of nodes from the initial node to the current node must be stored in order to execute the algorithm.” (Richard E. Korf, 1985). However, a downside of Iterative deepening A-star is that it might visit nodes multiple times during the execution of the algorithm. The algorithm uses heuristics to calculate a threshold value which is an estimated cost from the start to the goal. It runs depth-first search by exploring paths using the threshold value. However, if goal is not found the threshold is increased and search is done again which might cause some of the previously explored nodes to be revisited.

Another variant of A-star is Lifelong Planning A-star. “The first search of Lifelong Planning A* is the same as that of A* but all subsequent searches are much faster because it reuses those parts of the previous search tree that are identical to the new search tree” (S. Koenig and M.

Likhachev, 2001). The LPA* algorithm does not need to recalculate the entire graph in cases of changes to the nodes which should lead better performance than standard A* (S. Koenig and M. Likhachev, 2001).

D* Lite is an algorithm which is based on Lifelong Planning A-star. D* Lite is optimized for unknown environments and can reach goals using much fewer calculations than A-star (Sven Koenig, Maxim Likhachev, 2002). It is able to calculate the path while following the path towards the goal.

3 DEEP REINFORCED LEARNING

This chapter gives overview and theoretical background of machine learning and a specialized area ML called Deep Reinforcement Learning. Machine learning has multiple kinds of applications from image to speech recognition, image categorization, fraud detection, stock trading, statistics and spam filtering. It has also been used to teach computers to play video games successfully and often play better than humans.

Machine learning is a field in computer science which studies methods to enable computers to learn to perform different kind of tasks. In traditional imperative programming a programmer writes step-by-step instructions known as algorithms in code for computer to execute and to perform tasks. In machine learning the computer learns the steps to achieve the desired outcome during a process called training. The training process involves adjusting internal weights and parameters to reduce the difference between the predicted output and the actual target values. Using different kinds of machine learning algorithms a developer builds a model which is a representation of the system. The model is then trained by training algorithms and by inputting data to the model. There are three main branches in machine learning: supervised, unsupervised and reinforcement learning (Miguel Morales 2020, 1.).

In supervised learning the training data consist both input features and their corresponding output labels. The input features represent different properties of the data. For example stock price at a certain time or x and y position of the player. The labels provide guidance to the training algorithm in which direction to adjust the model. For example in training dataset call MNIST the data consists of 60000 grayscale images of handwritten digits and their corresponding labels (Keras 2023). The goal of supervised learning is to generalize and to develop a model which could make accurate predictions or classifications on new, unseen data. A model trained using MNIST data should be then able to accurately predict new handwritten digits which are not in the training data.

In unsupervised learning the training data consists only input data without output labels classifying the data. A model tries to find different kinds of patterns, structures, relationships, and clusters in the data. For example unsupervised learning could be used to detect different customer segments in a database of an e-commerce website. For example if a customer often spends more than 100 euros on electronics there is also 25% chance that he also buys extended warranty. It can be also used to detect anomalies and produce product, film or music recommendations.

Reinforcement learning is learning by trial and error by taking different kinds of actions and receiving feedback known as rewards from the environment. There are no previously collected datasets in reinforcement learning. Instead, the algorithm learns its environment based on the rewards which it receives by performing different kinds of actions.

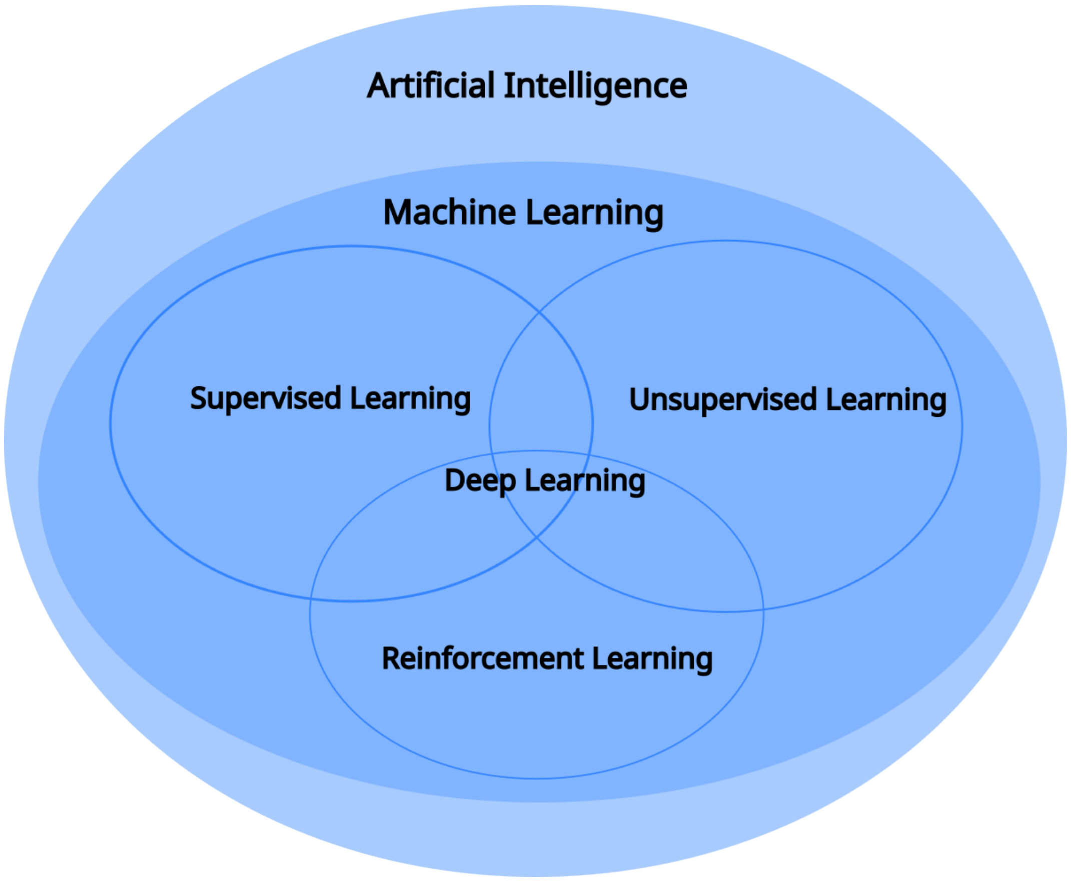


Figure 10: Machine Learning

3.1 Deep learning and Neural networks

Francois Chollet explains deep learning well in his book Deep Learning with Python, Second edition: “Deep learning is one of many branches of machine learning, where the models are long chains of geometric transformations, applied one after the other. These operations are structured into modules called layers: deep learning models are typically stacks of layers—or, more generally, graphs of layers. These layers are parameterized by weights, which are the parameters learned during training. The knowledge of a model is stored in its weights” (Francois Chollet 2021, 14.1.1)

Neural networks consist of connected nodes which are called neurons. These neurons are organized into multiple layers sometimes multiple levels deep, hence the name deep learning. Each neuron takes input from the previous layer, applies a mathematical transformation, and produces an output which is then sent to the next layer. The input layer receives the initial input data and features for processing. Hidden layers process the data and output layer produces the final prediction or classification as shown in the figure 9. Neural networks are inspired by human brain structure although having much fewer connections.

The learning in neural network occurs by adjusting weights in connections between neurons based on the difference of the model’s predictions and actual target values. If a model is trying to predict handwritten digits a prediction could be a number that the model outputs and target value the actual handwritten number. The weights are adjusted until the model produces predictions at satisfactory accuracy.

A Loss function calculates the difference between the prediction and target values, and it can be used to evaluate how well the algorithm is performing. Loss function is used during the learning or training process and when verifying the model results against test data. Optimization algorithms adjust the weights of the model trying to minimize the output of the loss function. For example gradient descent is an optimization method which adjusts the weights to a direction that reduces the loss.

One example of a loss function is the mean-squared error (MSE) which computes the average of the squared differences between the true and predicted values. It's often used in regression problems which involve producing predictions based on input features. The goal of a model is to minimize the MSE during training.

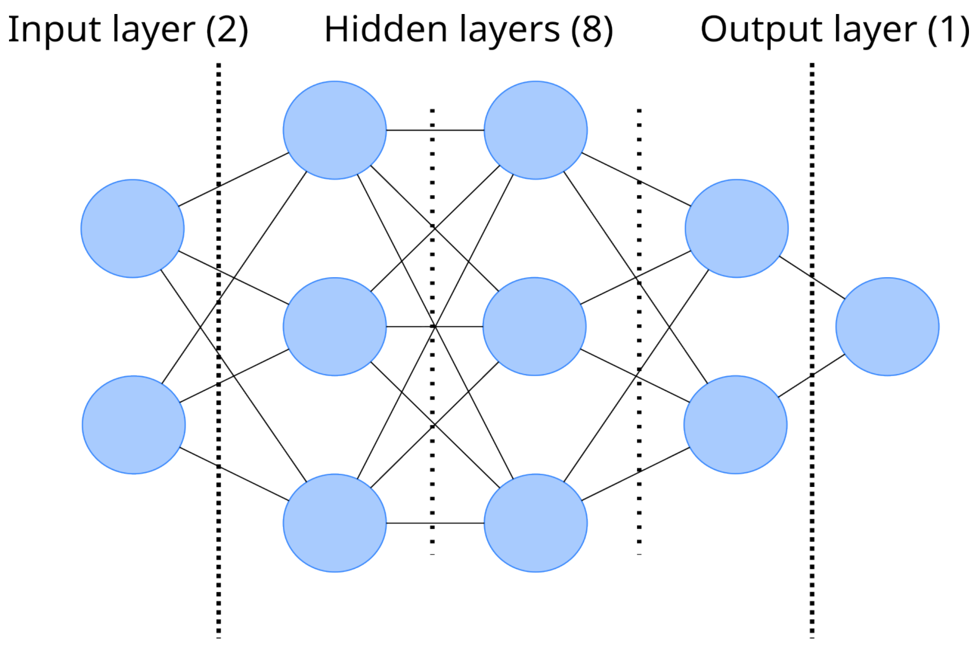


Figure 11: Neural network

The image 11 above illustrates a simple neural network with 11 nodes. The input layer has 2 nodes and therefore is expecting two feature inputs. For example location of a house and size in square meters. The first two hidden layers have two nodes each and the third layer two. The last output layer has one node which means that the network produces one prediction, for example the predicted price of a house based on the location of the house and size in square meters. Each node in a layer is connected to each node in the second layer.

There are multiple types of neural networks of which couple are listed before.

- Convolutional Neural Networks (CNN) which are usually used for image and video processing. The network consists a convolutional layer which is effective detecting edges, features and, corners in image data.

- Recurrent Neural Networks (RNN) is used to analyse sequential data or time series data and is used in fields like speech processing.
- Generative Adversarial Networks (GAN) is a neural network which is used to generate new data which matches the training data.

3.2 Deep reinforcement learning

As mentioned in the previous chapters deep reinforcement learning is a subset of machine learning which utilizes neural networks and deep learning models. The purpose of reinforcement learning is to learn how to act and not just classify or predict. For example driving a car, controlling a hand of a robot or optimizing energy consumption of an office building.

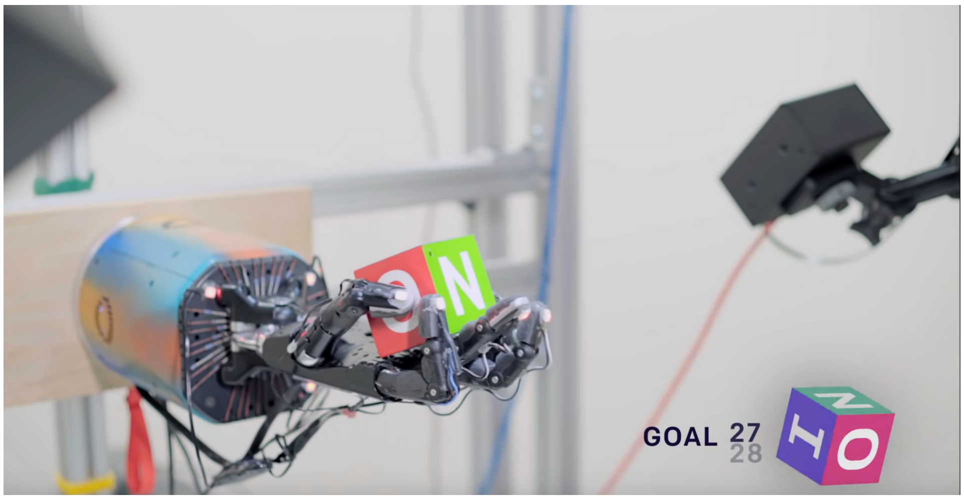


Figure 12: OpenAI's dactyl system controlling robot hand (OpenAI 2018.)

One difference in reinforcement learning compared to supervised or unsupervised learning is the element of time. The algorithm is influenced by previous time steps in the data. Time step could be any discrete point or step when the reinforcement learning algorithm is interacting with the environment. For example one turn in the game of chess.

Deep reinforcement learning algorithms interact with data continuously in order to decide which action to take. Every action it takes modifies the data. Deep learning agent could get positions and angles of different parts of a robot as an input features. It takes an action to move to a certain direction and receive feedback based on that action. The action has modified the positions and agent receives the update angles and positions in the next time step.

In deep reinforcement learning the goal is known but the steps to achieve the goal are unknown. The agent does not know what is the right action to take in every time step. Therefore, the algorithm receives rewards or penalties based on the actions. When the algorithm moves the robot to the right direction towards the goal it receives positive reward. On the other hand, if algorithm takes an action which causes the robot to collide with a wall it receives negative reward. Note that in reinforcement learning term reward could mean positive or negative. The reward reinforces the learning process and hence is called reinforcement learning. During training the algorithm cumulatively adds all the rewards together and tries to maximize the overall reward.

3.3 Markov decision process

Following chapters explain the core terminology of the reinforcement learning using a mathematical framework called Markov decision process.

Reinforcement learning problems may be described using a mathematical framework called Markov decision process. It models the decision-making process of an agent in a different situations or states when the decision outcomes are party random. The next chapters explain the key components related to Markov decision processes.

Agent

Agent is the decision maker in the MPD. In reinforcement learning the agent would be the algorithm which takes actions and makes decisions based on the observations of the environment.

Environment

The environment defines the task and the goal that the reinforcement agent aims to achieve. The environment is any dynamic process that produces data that is relevant to achieving our objective (Alexander Zai, Brandon Brown, 2020). The environment also defines to set of possible actions that agent may take. The possible actions in the environment is called the action space. The agent operates in the boundaries of the environment.

Action

The agent takes an action which modifies the environment and data. The algorithm analyses the current state and attempts to take the best possible action in order to maximize the reward. Discrete actions have a finite number of possible choices. For example move a chess piece forward or select other possible move in the game. Other example would be a classic “frozen lake” environment. Frozen lake is a usually 4×4 grid based environment where the agent is placed to the top left corner of the grid. Bottom right corner is the target where agent must try to move to. There are 4 holes in the lake which the agent must avoid. The actions in the discrete environment would be moving up, down, left or right.

Continuous actions on the other hand have a continuous range of possible values for the agent to take. For example apply X amount of force to a break pedal. Classical example of a continuous environment would be a “mountain car” environment in two-dimensional world. A car is placed at the bottom of a mountain and the agent’s goal is to move it to the top. Agent may apply force to the car to accelerate to either direction left or right. In mountain car environment the force that can be applied is limited between -1 and 1.

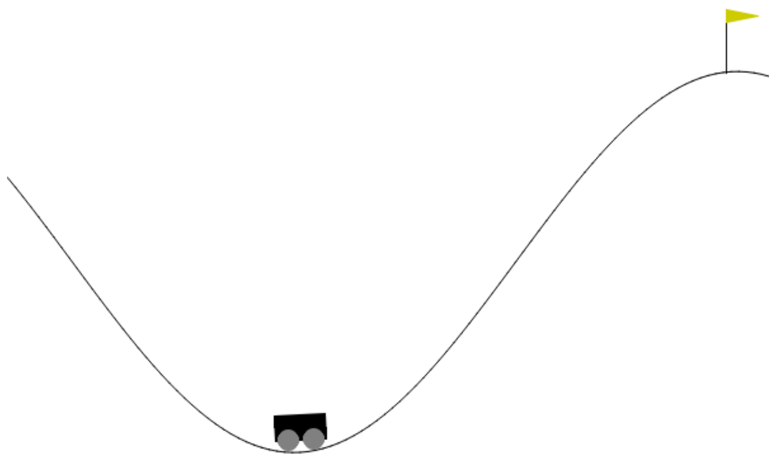


Figure 13: Mountain car environment

In mathematical notations action is indicated using letter A, A_t means action at a given time step and A_s is available set of actions in state s. For example on state s A_s could be “move left” or “move right”.

State

The state represents the current condition or configuration of the environment at a particular time. The agent observers the state and takes actions based on that. Environment transitions to a new state after each action agent takes. In the previous mountain car example state would consist the velocity of the car and position of the car along the x-axis. State is indicated using letter S and S_t in a give time step and initial state $s_0 \in S$.

An MDP needs to satisfy the Markov property. This means that the future state needs to be only dependent on the current state and action, and not on the sequence of states and actions that preceded it. In other words, the history of actions and states do not affect the next state and action. Each state is independent.

Reward

A reward is the term for the feedback which the environment provides to the agent based on the action. The reward is usually a numerical value which could be either positive or negative. Agent's performance is evaluated based on the cumulative reward. In the mountain car example the agent receives reward of value 100 if it reaches to the top of the mountain. Each time step a negative reward is also calculated which is -0.1 * action^2. Action in this case would be a value between -1 to 1 and would indicate the applied force to the car. R or R_t is used to indicate reward.

Discount

Discount factor is a parameter which discounts the future rewards. The parameter may be used for agent to prefer immediate rewards from actions or consider longer term rewards, and it can be used to adjust the agent's decision-making process. Discount is indicated using Greek letter gamma ɣ and in MPD it must be a value between 0 and 1.

Policy

Policy defines the agent's behaviour. Its function is to tell agent which action is the best action in any given state. Deterministic policies mean that each state in the environment is mapped to an action. For example in frozen lake example policy could specify that selecting action "down" is the best action on the first tile.

Stochastic policy on the other hand does not tell which specific action is the best to take. Instead, it specifies a probability distribution and agent may use these probabilities to select the best action. Policy is indicated using pi π, deterministic policy using notation $\pi : S \rightarrow A$ and stochastic policy using $\pi : S \times A \rightarrow R$. The best action at a give state in deterministic policy is $\pi(S)$ and $\pi(S, a)$ indicates the probability for an action on a state in stochastic policy.

START **Right**	Right	Right	Down
Right	Right	Down	Down
Up	**BLOCKED**	Down	Down
Up	**BLOCKED**	Right	**GOAL**

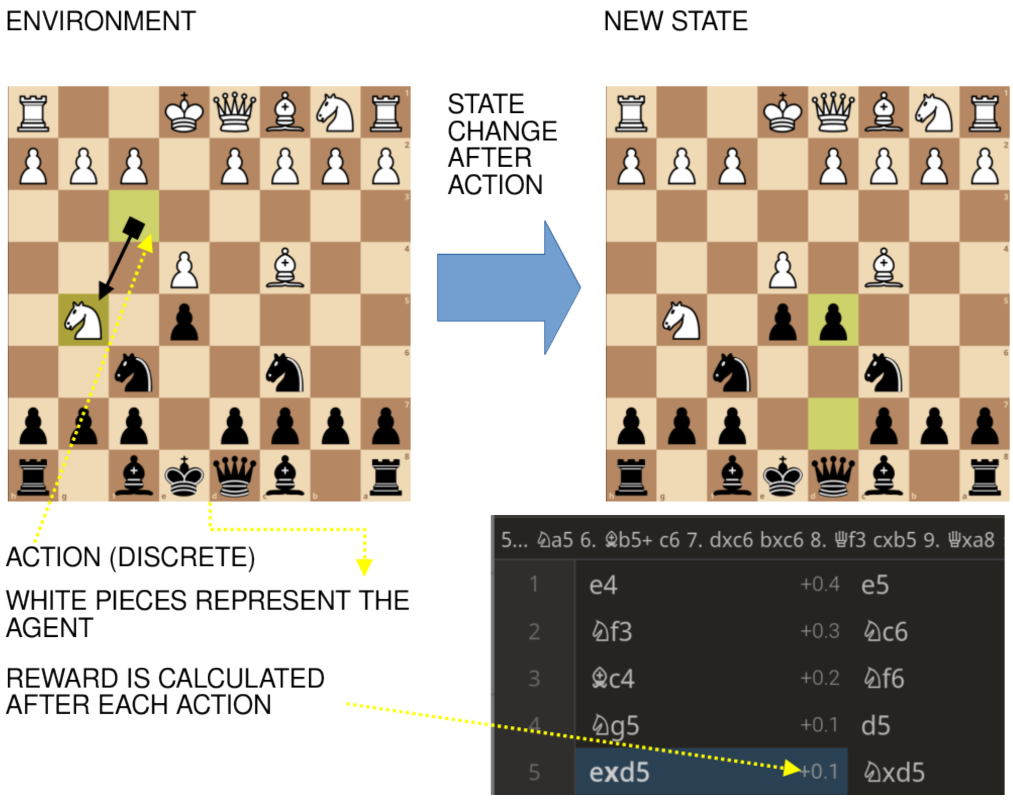
START **Right 50%** **Left 0%** **Up 0%** **Down 50%**	Right 5% Left 5% Up 10% Down 80%	Right 80% Left 5% Up 10% Down 5%	Right 5% Left 5% Up 10% Down 80%
Right 70% Left 10% Up 10% Down 10%	Right 80% Left 5% Up 10% Down 5%	Right 80% Left 5% Up 10% Down 5%	Right 5% Left 5% Up 10% Down 80%
Right 5% Left 5% Up 80% Down 10%	**BLOCKED**	Right 80% Left 5% Up 10% Down 5%	Right 5% Left 5% Up 10% Down 80%
Right 5% Left 5% Up 80% Down 10%	**BLOCKED**	Right 80% Left 5% Up 10% Down 5%	**GOAL**

Figure 14: Policies

Image 14 displays the difference between deterministic (left) and stochastic (right) policies. Deterministic tells the best action to take at any give state. Stochastic on the other hand the probability of an action.

Observation

Each time step agent "observes" data encapsulated to state from the environment. The state may be fully observable where agent has all data from the state available or partially observable where agent has only limited information of the environment. Agent selects the next action based on the observations.


ENVIRONMENT
NEW STATE
STATE CHANGE AFTER ACTION
ACTION (DISCRETE)
WHITE PIECES REPRESENT THE AGENT
REWARD IS CALCULATED AFTER EACH ACTION

5... ♘a5 6. ♗b5+ c6 7. dxc6 bxc6 8. ♕f3 cxb5 9. ♕xa8

1	e4	+0.4	e5
2	♘f3	+0.3	♘c6
3	♗c4	+0.2	♘f6
4	♘g5	+0.1	d5
5	exd5	+0.1	♘xd5

Figure 15: Chess example

In a game of chess the board, rules, pieces, and the chess engine analysis would represent the environment. White or black player would be the agent. Agent would be able to take discrete actions defined by chess rules. Continuous actions would not make sense in this environment. You would not move pieces by millimetres. Instead, you select the best move based on strict rules. For example in the picture 15 above white player would be the agent and has just taken action to move knight from the tile f3 to g5. After this a new state is calculated where black has moved pawn to d5. Reward +0.1 is analysed by Stockfish which is an open source chess engine. Stockfish is a strong chess engine that analyses chess positions and computes the optimal moves (Stockfish 2023). Agent would then observe the new chess position and reward and select the best action based on the new state.

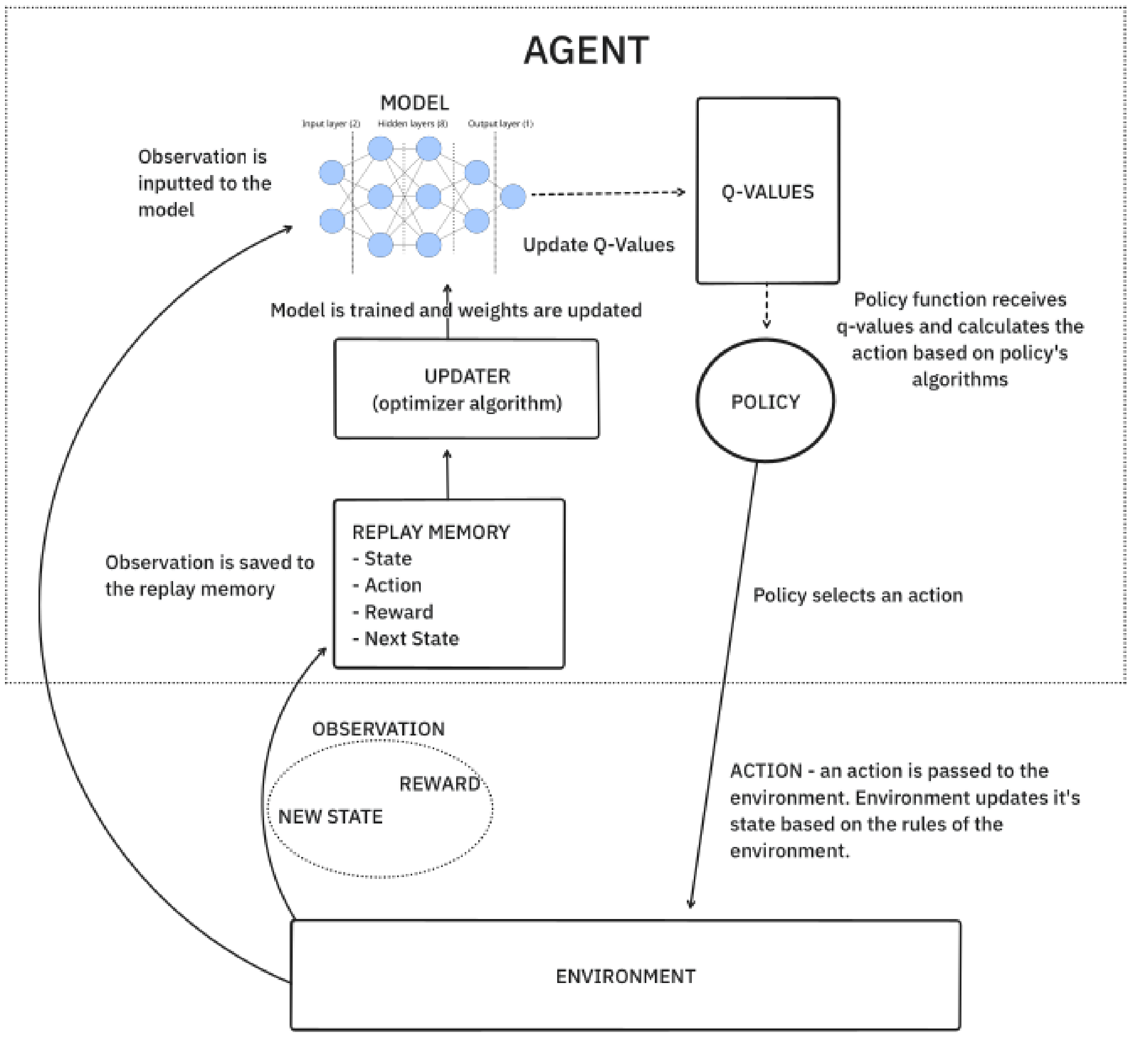

AGENT
MODEL
Input layer (2)
Hidden layers (8)
Output layer (1)
Observation is inputted to the model
Q-VALUES
Update Q-Values
Model is trained and weights are updated
Policy function receives q-values and calculates the action based on policy's algorithms
UPDATER (optimizer algorithm)
POLICY
REPLAY MEMORY
- State
- Action
- Reward
- Next State
Observation is saved to the replay memory
Policy selects an action
OBSERVATION
REWARD
NEW STATE
ACTION - an action is passed to the environment. Environment updates it's state based on the rules of the environment.
ENVIRONMENT

Figure 16: Deep Q-learning learning flow

The figure 16 above displays the basic reinforcement learning flow.

1. An agent receives observation from the environment.
2. The agent stores the observation to the replay memory.
3. A sample is fetched from the memory and using an optimizer algorithm the weights in the model are updated.
4. The observation is inputted to the model and passed through the network which updates the Q-values.
5. Agent’s policy reads the Q-values and calculates the best action based on the policy’s algorithms.
6. Policy selects and action which is sent to the environment.
7. The environment processes the action and updates its state.

Markov decision process is a mathematical tuple: (S, A, P_a, R_a) which consist states, actions, probability that the action leads to state $P_a(s', s)$ and immediate reward received after transitioning to a new state $R_a(s', s)$. In discounted MPDs a discount factor is also included in the tuple: (S, A, P_a, R_a, γ).

A state value function V(s) represents the expected cumulative reward agent is expecting to receive when following a specific policy. The main goal of the Markov decision process is to find an optimal policy which produces maximum discounted cumulative reward. The value function of this optimal policy is called optimal value function V*. The optimal policy maximizes the value function on all states.

A Q-function estimates the expected total reward received by taking an action in a state and then following policy afterwards. The Q-function could be considered to be the quality indicator of a given action on a particular state. A Q-value could be calculated for each action on each state and the next action could be selected based on that.

The MPD may be solved using multiple methods. The next chapter describes Q-learning which is one way to find the optimal policy.

3.4 Deep Q-learning

Q-learning is a model-free reinforcement learning algorithm which provides a way for an agent to learn the optimal policy in Markovian type of environment described in chapter 3.2.1. Model-based algorithms learn the model of the environment and is therefore able to predict the states and rewards. Agent may plan ahead because it knows the possible choices and what is the expected outcome of choosing a specific action. Often the full model of the environment is not available to the agent and the agent has to discover and explore the environment. Model-free algorithms do not know the environment beforehand and explore it during the training.

Q-learning uses a q-table similar than represented in figure 12 earlier. It initializes the q-table with q-values and selects the best action based on that. After each action the reward is measured and the q-table is updated accordingly. In large environments the q-table would also grow large causing higher requirements for computing and memory. In deep q-learning the q-table is replaced with neural network which is used to predict the best actions.

3.4.1 Exploration and exploitation

One of the challenges of agent's decision-making process is exploration-exploitation. Agent must make decisions between of exploring something new and possibly discovering a more optimal solution or exploiting already learnt information to select the best action. Selecting the balance between exploration and exploitation is important. Too little exploration could lead to non-optimal solution. However, too much exploration could also lead the agent to select non-optimal actions.

One strategy to mitigate the problem is called an epsilon-greedy strategy where the agent takes an exploration action based on probability which could reduce overtime. This allows the agent to increase exploitation gradually (Baeldung 2023).

3.4.2 Double Q-learning

Q-learning has a tendency to overestimate the q-values because it always tends to select the highest estimated Q-value. However, because of noise, stochastic nature or other factors of the environment it often leads to overestimation which may compound over time and leading non-optimal learning. Double Q-learning tries to solve this decoupling the selection and evaluation of the action (Hado van Hasselt, Arthur Guez, David Silver 2015). It does this by having two update functions for the Q-value which are randomly updating each other.

3.4.3 Prioritized replay method

As shown in the figure 14 the agent stores history of the actions, rewards, and states to replay memory which could be considered as the experience of the agent. Later the values from the replay memory could be fetched to adjust the agent's policy and value function towards more optimal solution. One important question in replay methods is which experiences to replay during learning. In simple implementation the agent replays the memory in a same order than they were originally discovered. However, more advanced methods have been developed which one of them is prioritized replay.

Prioritized replay method fetches experiences from the memory based on the importance or significance of the experience. “The key idea is that an RL agent can learn more effectively from some transitions than from others. Transitions may be more or less surprising, redundant, or task-relevant. Some transitions may not be immediately useful to the agent, but might become so when the agent competence increases (Schmidhuber, 1991).” (Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, 2016, 1.).

One way to prioritize the transitions which are in the replay memory is to use temporal difference error which represents the difference between predicted Q-value and the target Q-value. Those transitions which have high errors would be higher priority in the replay memory (Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, 2016, 3.). However, sampling in a way that each transition at least a small chance of being selected, is needed in order to prevent the agent learning only from experiences with high temporal difference error. Noise can also cause sudden errors to the replay memory which may be selected too often without sampling (Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, 2016, 3.). The replay memory is updated periodically to reflect the agent’s learning.

Prioritized replay may lead to more efficient learning because agent learns from experiences which are considered informative. However, computing requirements during training are increased because the need to update the temporal difference errors and keep track of them. This could be worth the increase because it could increase the learning speed by a factor of 2 when training agent to play Atari games (Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, 2016, 7.).

3.5 Proximal Policy Optimization algorithm (PPO)

Another popular deep reinforcement learning algorithm is the Proximal Policy Optimization algorithm PPO which is developed by OpenAI. Like Q-learning it is also a model-free algorithm. However, it is a policy gradient algorithm which means that it is trying to optimize the policy of the agent in order to maximize the cumulative reward. It does not need a value function or Q-table like Q-learning, and instead it optimizes the policy directly. PPO uses neural network to output the action directly while deep Q-learning uses neural network to first to update the values in the Q-table.

Adjusting different parameters of the algorithms has big impact on the agent's learning process and final performance of the model. Finding the right parameters is often tedious trial-and-error process (Mariam Kiran, Melis Ozyildirim 2020). PPO algorithm tries to make the parameter tuning simpler and more stable by ensuring only small updates to the policy on each iteration by clipping the possible policy update values to a small range. Therefore, it avoids updates which might be irreversible by being too large and causing dramatic changes to the agent's policy (Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo and Youn-Hee Han, 2020, 3.). It does this by keeping track of the new and old policy and comparing them on each iteration making sure that the difference is not great.

PPO is an actor-critic algorithm. Actor is a neural network component which is responsible for predicting the best action and maximizing the reward. Critic component is also a neural network which evaluates the chosen action and tries to minimize the error between prediction and result. The networks may even share some hidden layers for selecting actions. The actor model learns which action is optimal in each state by following the observation-action-reward process explained earlier. The critic model on the other hand learns to evaluate if the selected action resulted the agent being in a better state in the environment. It provides feedback which is used to optimize the actor model. (Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo and Youn-Hee Han, 2020, 4.)

PPO also needs a memory buffer where it keeps samples of experiences that the agent has learnt. During training PPO samples small continuous batches from the memory at a random starting point. This helps the algorithm to learn efficiently because it allows the algorithm to take multiple optimization steps in a single epoch. It also reduces memory usage.

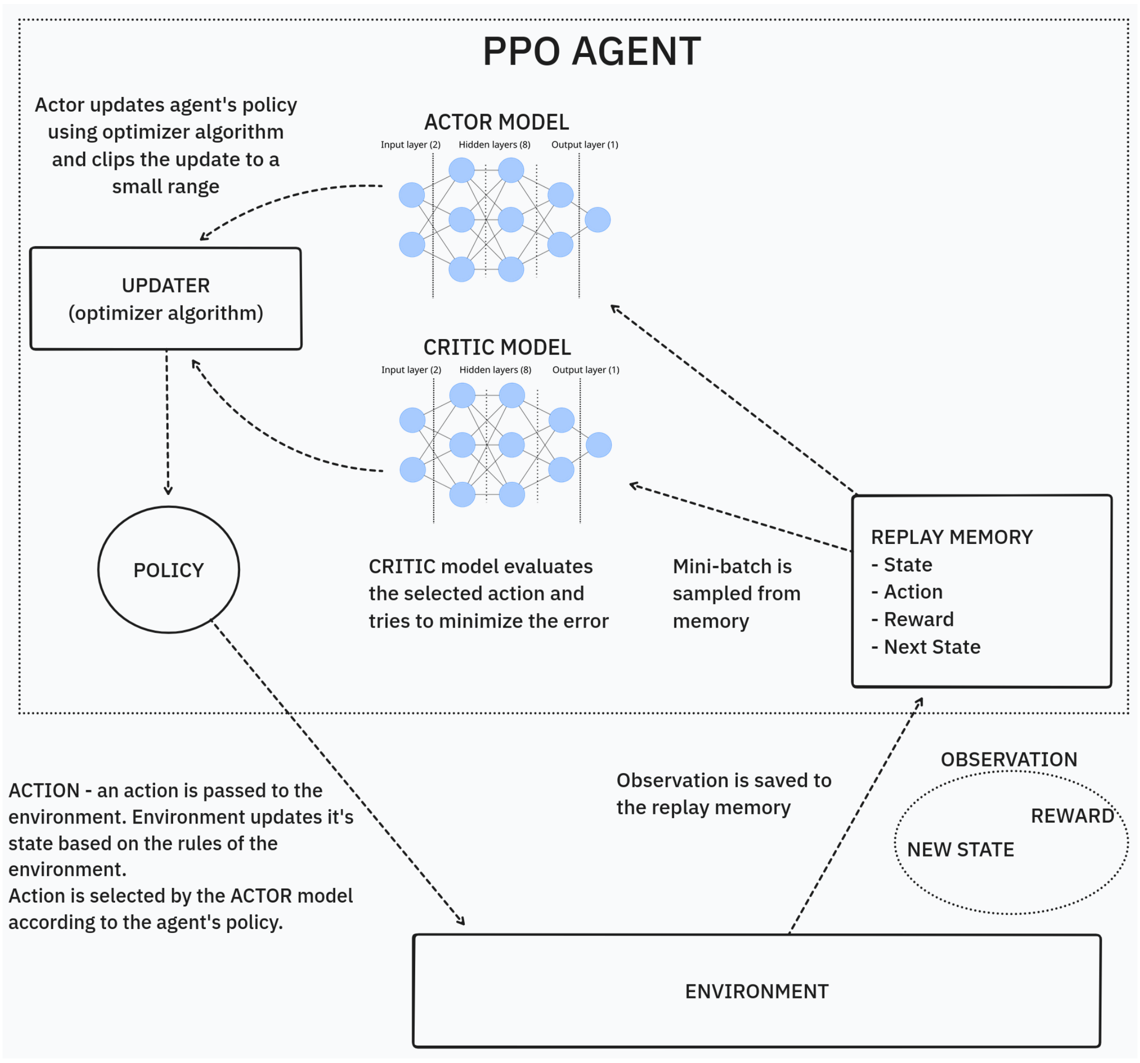

PPO AGENT
Actor updates agent's policy using optimizer algorithm and clips the update to a small range
ACTOR MODEL
Input layer (2)
Hidden layers (8)
Output layer (1)
UPDATER (optimizer algorithm)
CRITIC MODEL
Input layer (2)
Hidden layers (8)
Output layer (1)
POLICY
CRITIC model evaluates the selected action and tries to minimize the error
Mini-batch is sampled from memory
REPLAY MEMORY
- State
- Action
- Reward
- Next State
ACTION - an action is passed to the environment. Environment updates it's state based on the rules of the environment.
Action is selected by the ACTOR model according to the agent's policy.
Observation is saved to the replay memory
OBSERVATION
REWARD
NEW STATE
ENVIRONMENT

Figure 17: PPO algorithm

Figure 17 shows the basic flow of PPO agent. Compared to the Q-learning agent it has two neural networks.

4 MINIGRID EXPERIMENTS

In purpose of gathering data, gaining knowledge and investigating the implementation details of Q-learning, PPO algorithm and A-star algorithms a straightforward simulation game was implemented as a part of this thesis. The simulation game was used to measure the speed and correctness of the Q-learning agent and A-star algorithm. When evaluating the results the impact of the frame time was taken into account because it's an important aspect of a smooth gameplay.

The next chapters will introduce experiments done using the Minigrid library. The goal is to evaluate the difference of training a deep reinforcement agent and implementation of A-star algorithm in different kinds of environments. For example in some cases A-star could be used directly to solve Minigrid scenario. However, in other cases some modification might be needed, or it might not be possible at all to solve using a-star alone. On the other hand, reinforcement learning might be able to learn to solve the problem without any help.

Following Minigrid scenarios were examined:

- Environment with dynamic changing obstacles
- Environment with a task – Pick up a key and open a door in order to reach the goal.
- Environment where agent or algorithm must avoid certain paths

4.1 The Minigrid library

Minigrid library contains simple and easily configurable grid world environments to conduct reinforcement learning research (Farama foundation, 2023). Farama foundation provides also tools to train and test models which work in Minigrid type of environment. The Minigrid library and associated tools are written using Python programming language and uses PyGame multimedia library to visualize agent’s behaviour in the environment.

RL Starter Files project was used to help with training the agents. It contains tools and scripts to train agents in the Minigrid environment using PPO and A2C algorithms. It also has useful scripts to visualize the training process, and save and evaluate the trained models.

The Minigrid library provides multiple different kinds of scenarios for experiments. As an example, One scenario is a “Locked Room” environment. “The environment has six rooms, one of which is locked. The agent receives a textual mission string as input, telling it which room to go to in order to get the key that opens the locked room. It then has to go into the locked room in order to reach the final goal. This environment is extremely difficult to solve with vanilla reinforcement learning alone.” (Locked Room, Farama Foundation 2023).

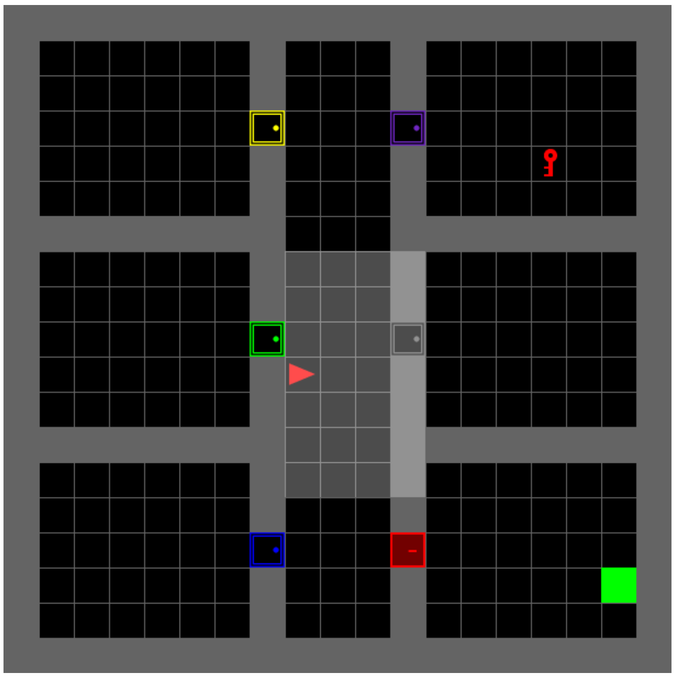


Figure 18: Locked room environment by Farama foundation

Picture 18 above is an example environment provided by the Minigrid library. In order to solve the environment the agent which is represented by a triangle must move to the yellow square. In order to do that the agent must first go to pick up the red key which opens the red door. The agent also has a limited field of view which are displayed as lighter coloured squares. This requires the agent to first explore the environment before being able to solve it.

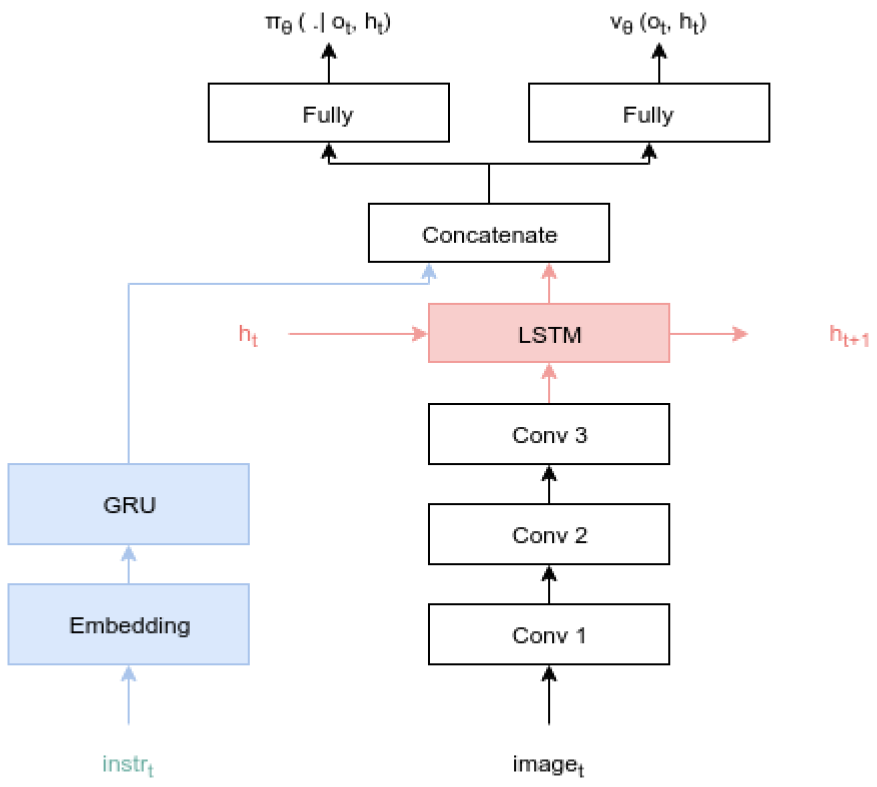


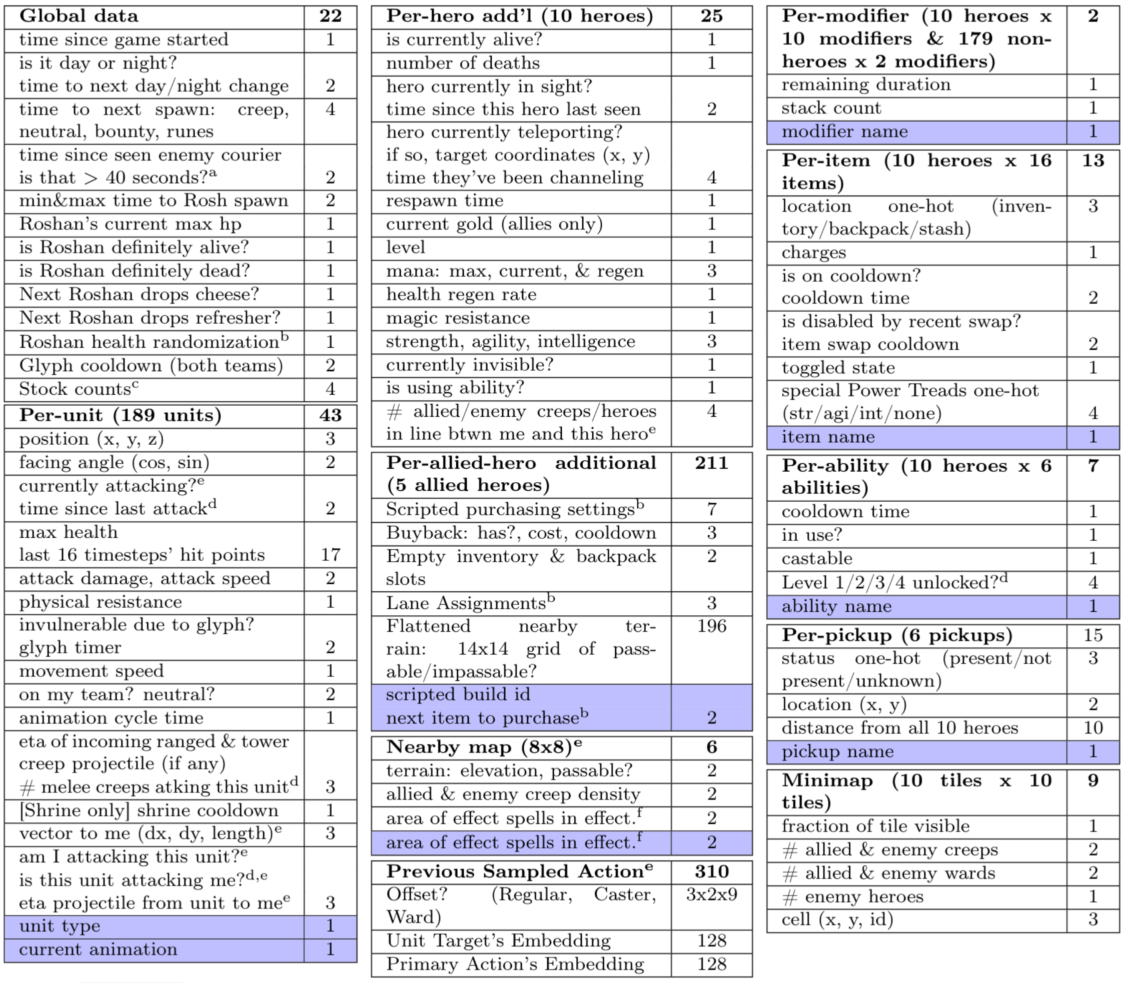
Figure 19: RL Starter Files default model (rl-starter-files 2023)

The image 19 above shows the default model used by the RL Starter Files project. The default model was not modified in the experiments. The model has two inputs. The first input is the environment or state of the Minigrid which is inputted to the model as images in raw pixel format in 256x256 resolution. The second optional input is a set of text instructions to the agent describing the goal. The model has a long short term memory LSTM which is disabled by default. LSTM helps the model to remember dependencies of inputs and helps agent make decisions based on current and historical inputs. The memory is controlled using a numeric recurrence parameter. Adjusting the parameter allows the network to back-propagate through multiple time steps which enables the agent to learn patterns over sequences of inputs. Back propagation is a process where the model weights are adjusted based on previous errors. Larger value for recurrence will make the agent consider longer sequences of input.

The model has three convolution layers for processing the input image. After these layers, optional memory layer is applied. The memory is needed in some Minigrid scenarios for example in “RedBlueDoors” where the agent has to open a red door first and then blue door. Memory is needed in order to agent to remember the sequence. The text instructions input is first converted into a dictionary for retrieving words using indices before it’s passed to GRU layer. Gated recurrent unit (GRU) is a mechanism to process sequential data and is useful in natural language processing. Like LSTM, it is disabled by default. Both inputs are concatenated and then passed to actor and critic model. By default, actor model has two linear layers and uses tanh activation function. The critic model has the same network structure as the actor model. The hyperbolic tangent function (tanh) outputs values between -1 and 1.

Figure 20: Screenshot of the Dota 2 game

The environments in video games are often much more complex, and extracting a raw pixel data for reinforcement learning agent would yield noisy data. Therefore, training the agent to play an actual computer game would be more challenging than very limited and simplified Minigrid environment. The action spaces in video games are usually much wider and the pixel input greater than 256x256 at least in modern games and requiring lots of computing power to train. For example OpenAI trained an agent called Five to play a game called Dota 2 which is a team versus team real time strategy game with multiple heroes and items. It requires complex tactics and team coordination in order to win. The observation space of the OpenAI Five was around 16000 inputs (OpenAI 2021, 45.). Some of these inputs are presented in a figure 19. In order to play the game the agent needed much more inputs than just raw pixel data. “Instead of using the pixels on the screen, we approximate the information available to a human player in a set of data arrays” (OpenAI 2021, 39.). The following quote summarizes the underlying computing requirements well: “OpenAI Five is a single training run that ran from June 30th, 2018 to April 22nd, 2019. After ten months of training using 770±50 PFlops/s·days of compute, it defeated the Dota 2 world champions in a best-of-three match and 99.4% of human players during a multi-day online showcase.” (OpenAI 2021, 8.). The OpenAI Five neural network consists of 158,502,815 parameters (OpenAI 2021, 45.).



Global data	**22**
time since game started	1
is it day or night? time to next day/night change	2
time to next spawn: creep, neutral, bounty, runes	4
time since seen enemy courier is that > 40 seconds?[a]	2
min&max time to Rosh spawn	2
Roshan's current max hp	1
is Roshan definitely alive?	1
is Roshan definitely dead?	1
Next Roshan drops cheese?	1
Next Roshan drops refresher?	1
Roshan health randomization[b]	1
Glyph cooldown (both teams)	2
Stock counts[c]	4

Per-unit (189 units)	**43**
position (x, y, z)	3
facing angle (cos, sin)	2
currently attacking?[e] time since last attack[d]	2
max health last 16 timesteps' hit points	17
attack damage, attack speed	2
physical resistance	1
invulnerable due to glyph? glyph timer	2
movement speed	1
on my team? neutral?	2
animation cycle time	1
eta of incoming ranged & tower creep projectile (if any) # melee creeps atking this unit[d]	3
[Shrine only] shrine cooldown	1
vector to me (dx, dy, length)[e]	3
am I attacking this unit?[e] is this unit attacking me?[d,e] eta projectile from unit to me[e]	3
unit type	1
current animation	1

Per-hero add'l (10 heroes)	**25**
is currently alive?	1
number of deaths	1
hero currently in sight? time since this hero last seen	2
hero currently teleporting? if so, target coordinates (x, y) time they've been channeling	4
respawn time	1
current gold (allies only)	1
level	1
mana: max, current, & regen	3
health regen rate	1
magic resistance	1
strength, agility, intelligence	3
currently invisible?	1
is using ability?	1
# allied/enemy creeps/heroes in line btwn me and this hero[e]	4

Per-allied-hero additional (5 allied heroes)	**211**
Scripted purchasing settings[b]	7
Buyback: has?, cost, cooldown	3
Empty inventory & backpack slots	2
Lane Assignments[b]	3
Flattened nearby terrain: 14x14 grid of passable/impassable?	196
scripted build id next item to purchase[b]	2

Nearby map (8x8)[e]	**6**
terrain: elevation, passable?	2
allied & enemy creep density	2
area of effect spells in effect.[f]	2
area of effect spells in effect.[f]	2

Previous Sampled Action[e]	**310**
Offset? (Regular, Caster, Ward)	3x2x9
Unit Target's Embedding	128
Primary Action's Embedding	128

Per-modifier (10 heroes x 10 modifiers & 179 non-heroes x 2 modifiers)	**2**
remaining duration	1
stack count	1
modifier name	1

Per-item (10 heroes x 16 items)	**13**
location one-hot (inventory/backpack/stash)	3
charges	1
is on cooldown? cooldown time	2
is disabled by recent swap? item swap cooldown	2
toggled state	1
special Power Treads one-hot (str/agi/int/none)	4
item name	1

Per-ability (10 heroes x 6 abilities)	**7**
cooldown time	1
in use?	1
castable	1
Level 1/2/3/4 unlocked?[d]	4
ability name	1

Per-pickup (6 pickups)	**15**
status one-hot (present/not present/unknown)	3
location (x, y)	2
distance from all 10 heroes	10
pickup name	1

Minimap (10 tiles x 10 tiles)	**9**
fraction of tile visible	1
# allied & enemy creeps	2
# allied & enemy wards	2
# enemy heroes	1
cell (x, y, id)	3

Figure 21: Dota 2 observation space of OpenAI Five (OpenAI 2021. 38.)

Based on results from Minigrid experiments and the previous research by OpenAI, using traditional algorithms like A-star or D*-lite would be better in path finding in game development because of the required computing power and development time. Using machine learning to other parts of a game could be feasible. For example in strategy games where the player plays against a programmed AI machine learning could be used to build the strategy of the computer controlled opponent players. The model could be even tuned during the game to adapt itself to human player's behaviour (Toni Lääveri 2017, 74.). In games with lots of different options, parameters and strategies this could be challenging to program using dynamic programming.

4.2 Dynamic obstacles environment

Dynamic obstacles environment is an empty Minigrid room with moving obstacles. In a computer game the obstacles could represent enemies trying to catch the player.

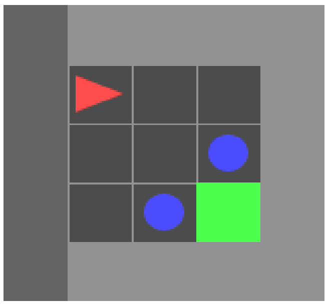


Figure 22: Dynamic obstacles environment

Agent must reach the goal without colliding any of the moving obstacles, and it will receive maximum penalty if collision with an obstacle happens. Reward for successfully reaching the goal is calculated using formula *1 – 0.9 * (how many steps / maximum steps for the environment)*. The environment has an action space of left, right, and forward. Left and right actions rotate the agent and forward moves it one step. The environment was studied in order to gain knowledge how machine learning algorithms learn in dynamically changing environments and how A-star algorithm could be implemented to solve it.

Two models were trained. The first agent was trained using PPO algorithm in 6×6 sized environment for period of one million frames. Discount value was set to 0.99, learning rate to 0.001 and batch size to 256. No memory was used during training. The second agent was trained in 8×8 sized environment using same parameters. However, the model did not converge until memory was included to the model using a recurrence value of 4.

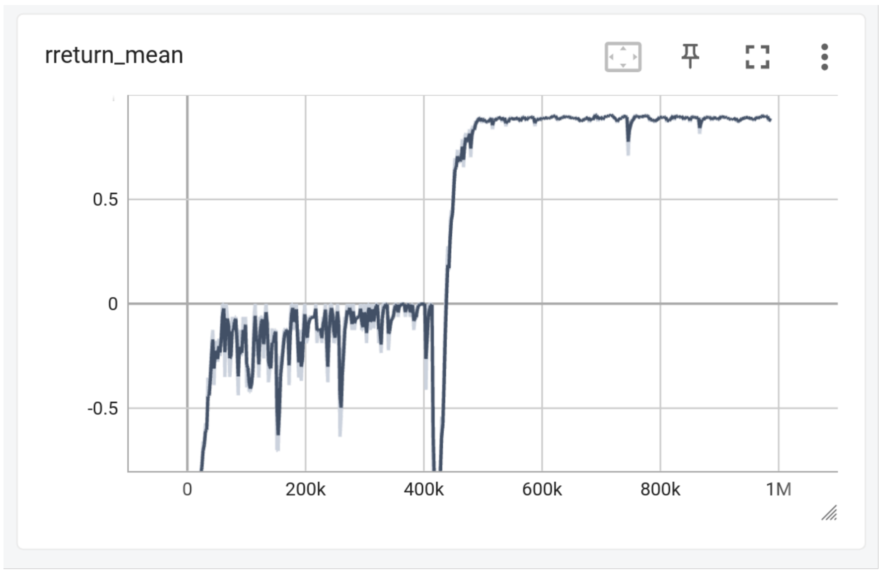


Figure 23: Mean return for 6×6 dynamic obstacles environment

Figure 23 shows the training plot for 6×6 environment. The model converged after around 500000 frames.

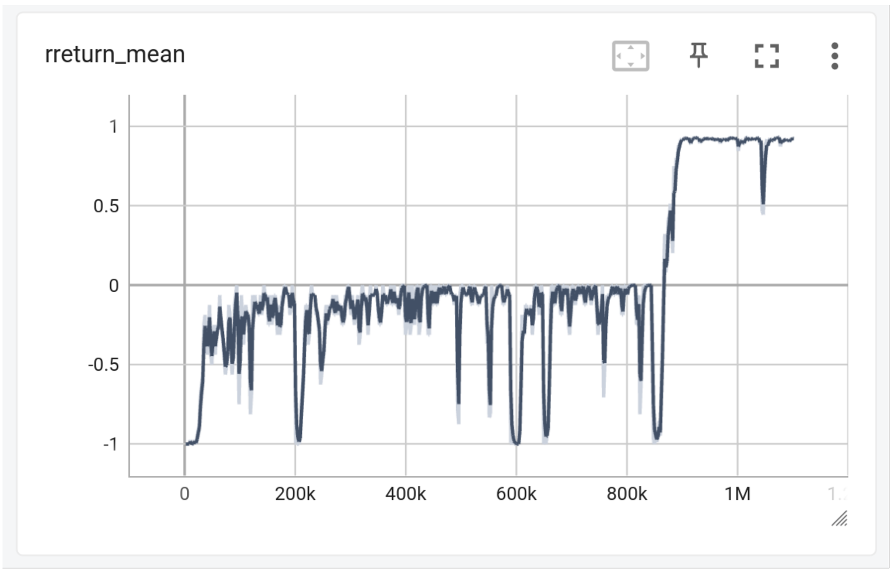


Figure 24: Mean return for 8×8 dynamic obstacles environment

Figure 24 plots the 8x8 environment training and the model converged after 900000 frames. Even though there are only 28 more tiles in the grid on 8x8 than in 6x6 environment the training was harder, required more episodes and LSTM memory. One more experiment was done using recurrence with value of 8. The model converged dramatically faster after this as shown on the figure 25 below.

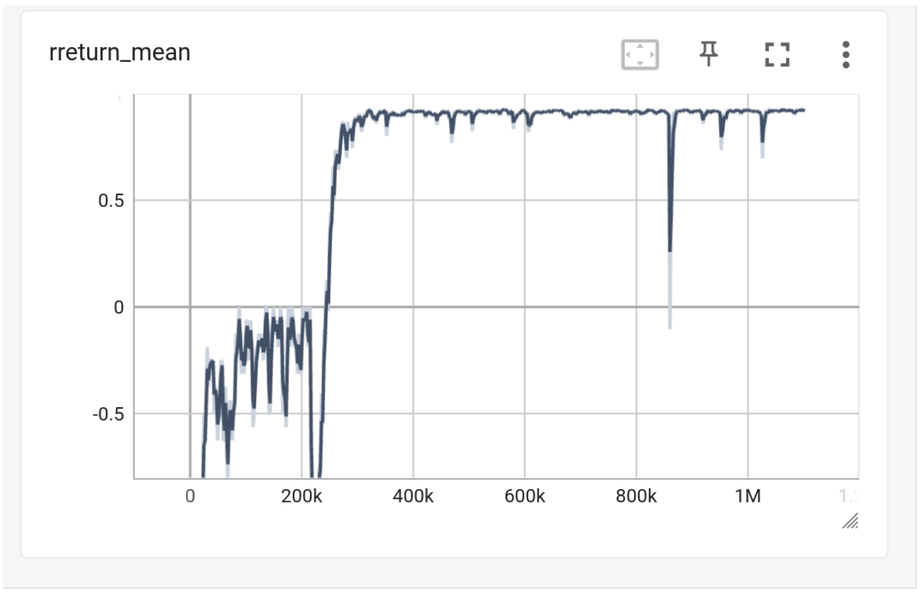


Figure 25: 8×8 environment trained with more memory

In a small 8x8 environment A-star algorithm could recalculate the path after every step in case the environment has changed which might not scale to larger environments with numerous nodes especially because the reordering of the priority queue (Sven Koenig, Maxim Likhachev 2002, 5.). However, D*-lite algorithm could be more suitable in this case because by its nature only updated parts which affect the calculated path are recalculated, thus reducing processing needs. The algorithm first calculates the path without the dynamic obstacles and then start traversing towards the goal. This solves navigation problems in unknown terrain (Sven Koenig, Maxim Likhachev 2002). If obstacle is found on the original path the algorithm would then compute the updated path. It has a method to detect the edge cost changes of the graph, but it does not make assumptions how they change (Sven Koenig, Maxim Likhachev 2002, 5.). In case of appearing obstacles like in the Minigrid environment the edge cost would be changed to infinity. D*-lite uses priority queue like A-star but also uses heuristic methods avoiding the need to reorder the priority queue.

4.3 Door key environment

Door key environment has a key which unlocks a door. The agent must first pick up a key and unlock the door before it can reach the goal.

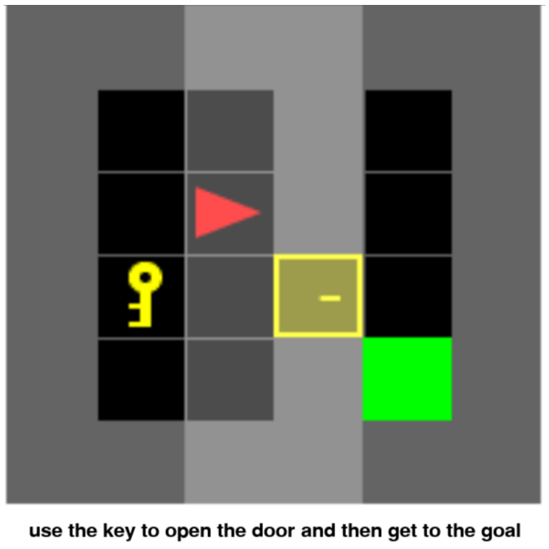


Figure 26: Minigrid's door key environment

The reward is calculated using the same formula as in the Dynamic obstacles environment except there is no penalty. No reward is received if agent fails to reach the goal before maximum amount of steps is reached. In addition to left, right, and forward actions the door key environment also has “pick up” and “toggle” actions. Pick up takes the key and toggle opens or closes the door.

The agent was trained using PPO algorithm with one million frames. As shown in the image 27 the agent learnt the environment after around 80000 frames.

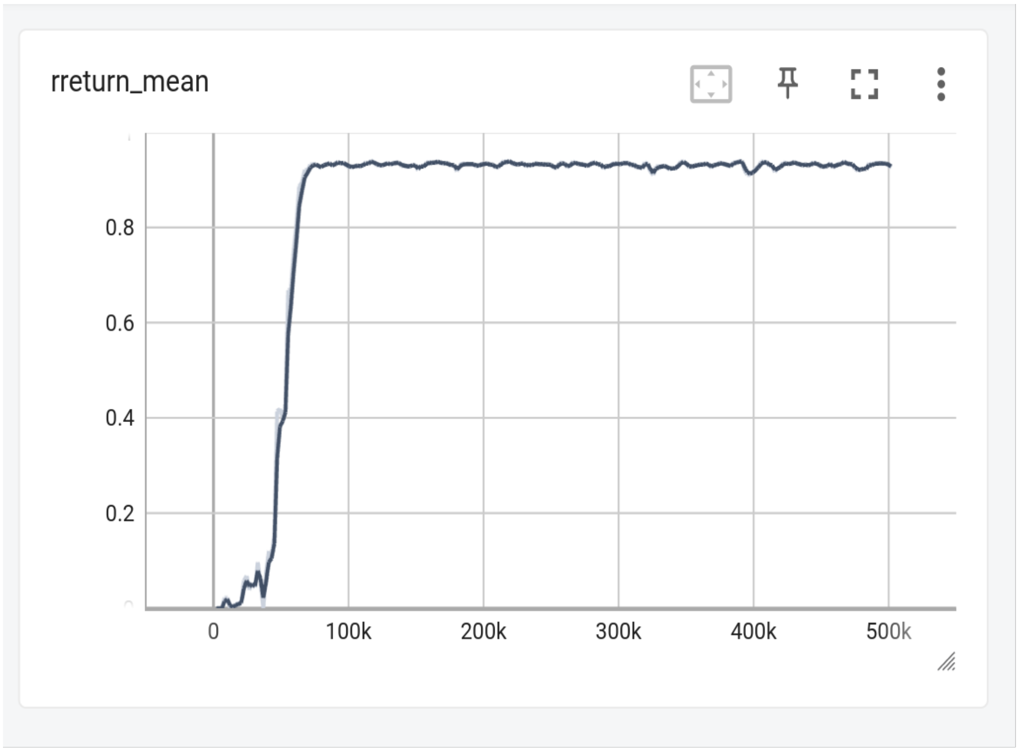


Figure 27: Training result of Door key environment

The environment was chosen to study challenges different tasks presents to A-star algorithm. A-star alone would not be able to solve the environment. Instead, dynamic programming principles would need to be used to break the problem into multiple smaller components. Go to the key, then go to the door, then unlock the door, and finally go to the goal position. First A-star algorithm would be used to find path from start to the key. After reaching the key a "pick up" action would be triggered in order to hold the key. After this a path would have to be recalculated from the current position to the door, and after reaching to the door toggle action needs to be triggered. Final path calculation is from the open door to the goal. The environment can be solved in a straightforward manner using dynamic programming and standard A-star algorithm.

However, one advantage that the machine learning algorithm has compared to the dynamic programming is that the agent may explore the steps in order to reach the goal by itself without being explicitly specified what needs to be done. The agent knows the actions and receives rewards based on its actions which will help the agent to learn the way to reach the goal.

4.4 Lava gap environment

Lava gap environment has some tiles filled with lava which the agent must avoid. Touching the lava immediately terminates the episode and returns maximum penalty. Reward is calculated using the same formula as in environments above. It has same action and observation than dynamic obstacles environment.

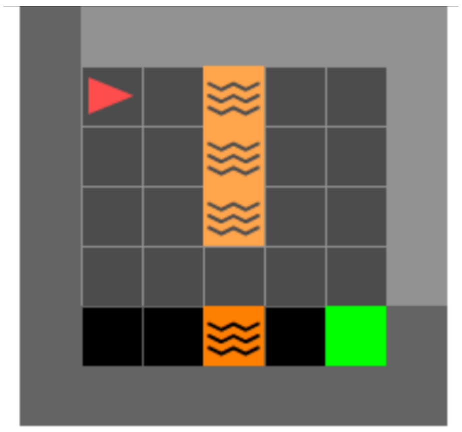


Figure 28: Lavagap environment

Lavagap environment is trivial to solve using A-star by simply considering lava tiles as impassable nodes by adjusting the weight in the graph to infinity. After this A-star may calculate the path without any adjustments. As seen in figure 29 the PPO algorithm also managed to train the agent in around 120000 frames.

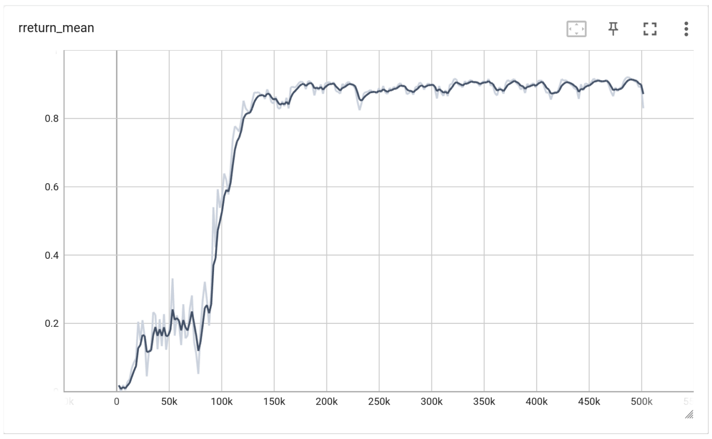


Figure 29: Training result of Lava gap environment

5 SIMULATION GAME EXPERIMENTS

During this thesis a simple simulation game was developed in order to study Deep Q-Learning machine learning algorithm and A-star path finding. A 4-way connected grid-based navigation system was selected for this thesis because of ease and straightforwardness of implementation and simple and clear way of visualization of the algorithms in the environment. Diagonal movement was prohibited in the environments. Different scenarios were developed as a plain numeric CSV files as its easy to edit them using basic spreadsheet tool. The grid size depends on the rows and columns of the scenario CSV-file. Grids used in this thesis were also be relatively small and therefore benefit of implementing a navigation mesh is insignificant.

C++ programming language and a machine learning library called Mlpack was chosen to develop the simulation. It contains wide variety of different machine learning algorithms (Mlpack, 2023). Deep Q-learning was implemented to the simulation game using Mlpack library. Other relevant libraries used were Armadillo and Ensmallen.

“Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use” (Armadillo, 2023). Armadillo implements different kinds of matrix calculations and data structures which were used when developing the Markov compatible environment to the simulation game.

“ensmallen is a high-quality C++ library for non-linear numerical optimization” (Ensmallen, 2023). Ensmallen implements different optimizers used during the training of the machine learning model. Both Armadillo and Ensmallen are dependencies of Mlpack.

A-star algorithm was implemented using the logic explained in the theory chapter 2.7. The implementation was adapted to work in the environment developed in the simulation game. Both Q-learning and A-star work in the same environment using the same CSV based scenarios in order not to give any advantage to other implementation. Manhattan distance was used to calculate the heuristic value in A-star. Unordered map and priority queue data structures were used from the C++ standard library. The code was compiled using GCC compiler version 13.2.1 with O3 flag turned on during compliation which optimizes the code for performance.

Armadillo library is optimized to use multithreading when doing matrix calculations. Mlpack library takes advantage of this feature which might give some advantage in speed to the Q-learning reinforcement learning agent compared to the A-star algorithm. A-star was implemented being a single-threaded process. This difference is taken into account in evaluation.

The simulation tracks how many milliseconds it takes to build a path from start to goal and how many steps are required. The tracking is done by doing 10000 test runs in a row and calculating the mean average time in milliseconds. All tests were done on the same computer. CPU usage was measured using a perf tool and memory using a tool called memusage. Note that the time to build the path differs from perf tool timing because the simulation game tracks only the time to build the actual path. Perf tool measures the whole program execution which includes parts which are not relevant to build the path like parsing command line parameter or reading the scenario CSV file.

5.1 Training the model

Following image 30 displays the network graph of the model. The model was kept simple, and the thesis does not focus optimizing it.

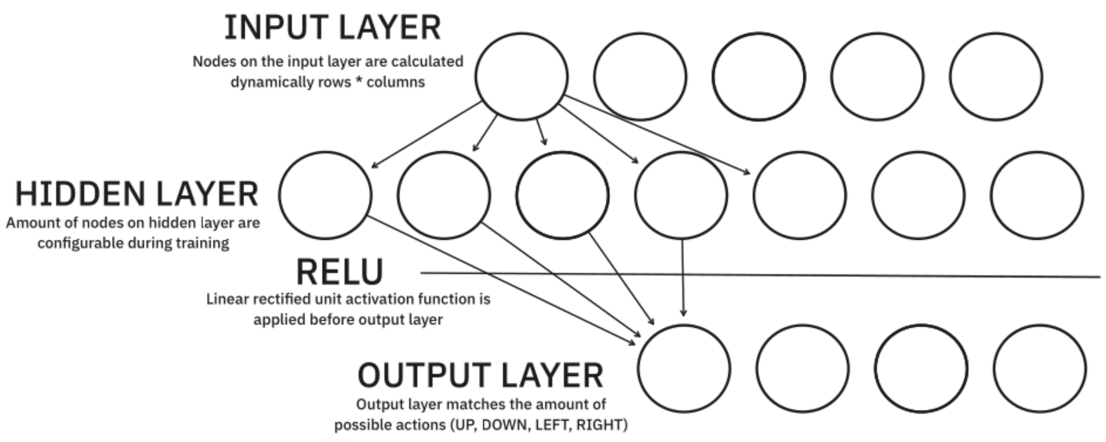

INPUT LAYER
Nodes on the input layer are calculated dynamically rows * columns
HIDDEN LAYER
Amount of nodes on hidden layer are configurable during training
RELU
Linear rectified unit activation function is applied before output layer
OUTPUT LAYER
Output layer matches the amount of possible actions (UP, DOWN, LEFT, RIGHT)

Figure 30: Standard neural network graph used during tests

The input layer of the neural network was dynamically calculated by multiplying the amount of rows and columns in the grid. The amount of nodes on the hidden layer was configurable and the nodes on the output layer matches the amount of possible different actions agent may take.

The Environment has an action space of left, right, up and down. The environment terminates in two cases: the goal is reached, or maximum steps is reached. There are multiple configuration options:

- Success reward is a reward which the agent receives when successfully achieving the goal.
- Maximum steps reward is received when maximum number of steps is reached and the environment is terminated. Reward may also be a negative number, so penalties are also supported.
- Bonus reward is added to the observation when agent hits an obstacle. This may also be negative in order to guide the agent to avoid obstacles.
- Maximum steps is a value which indicates the amount of steps agent is allowed to take before the environment terminates and returns reward for maximum steps.

The observation space of the agent is the whole grid in numeric format where each tile has a number representing the type of the tile, for example the position of the agent is number 4. The input is passed to the network in a matrix form with a single row and dynamically changing amount of columns. The original scenario matrix of 8x8 for example, is reshaped in to 1x16 matrix. This was done to follow similar pattern than OpenAI used in their Dota 2 research. “The complex multi-array observation space is processed into a single vector” (OpenAI 2019, 4.). The input was also normalized before sending it to the agent.

The training algorithm implements following parameters:

- Number of episodes to train. An episode represents a sequence of actions which agent performs in the environment starting from initial state to the terminal state. For example starting at the first position and moving until reaching the goal or until the environment terminates because the agent has taken too many steps. Usually the agent requires many episodes in order to converge.

- Optimizer algorithm – Adam optimizer from Ensmallen library was used. The purpose of the optimizer is to update the weights of the model.
- Policy – Epsilon greedy policy was used for Q-learning in the simulation game. Greedy policy always selects the highest estimated Q-value and is used to balance the exploration-exploitation selection. Policy supports configuring epsilon, minimum epsilon, decay rate and anneal interval. Epsilon may be used to control the exploration rate of the agent. For example value 1 would mean that the agent explores almost all the time and value close to 0 would cause the agent to select actions greedily based on Q-values. Decay rate decreases the epsilon over time and causes shift from exploration to exploitation. Anneal interval controls how often epsilon is decayed, and it's measured in episodes. For example the training algorithm could be configured to trigger the decay of epsilon every 50 episodes.
- Replay method or memory – Prioritized replay was used. It supports parameters for alpha, capacity, and batch size. Replay method was described in chapter 3.4.3.
- Step size or learning rate – The learning rate controls the magnitude how much the weights are being adjusted. Finding the correct value for learning rate is a process of trial and error. Having a too small value will cause the model to learn very slowly but having it too large might cause the agent to fail to converge or produce and optimal solution.
- Discount – This is the discount factor or gamma described in Markov decision process chapter 3.2.1.
- Exploration steps – Agent can be configured to explore the environment in order to gain knowledge of it before starting to update the weights of the model.
- Double Q-learning – The implementation supports turning double q-learning on or off. It is described in chapter 3.3.2.
- Step limit – This configures the maximum amount of steps agent will take in order to reach the goal. If the limit is reached agent aborts the episode.

The main training loop is presented in the following pseudocode. On the line 1 the scenario is loaded from a CSV file and the environment is generated based on that on the line 2. Q-learning agent is created on line 3 and average return of the agent is tracked using a variable defined on line 4. The main training loop starts on the line 6, and it loops until the agent has completed the configured amount of episodes. On the line 7 the agent performs a single episode and returns the total return of the episode. At the end of the loop the environment is reset back to the initial state. Finally, the average return is reported after the main loop.

```
1  CSVFile = loadScenarioFromCSV(filePath)
2  Environment = initializeEnvironment(CSVFile)
3  Agent = QlearningAgent(Environment, trainingParameters)
4  averageReturn = runningStatisticsOfScalars
5
6  loop trainingParameters.amountOfEpisodes
7        totalReturn = agent.Episode()
8        averageReturn.add(totalReturn)
9        Environment.Reset()
10
11 print "Average return " + averageReturn.calculateMean()
12
```

The following pseudocode demonstrates the agent's episode function and at which point agent updates the weight and learns. It references to the line 7 in the pseudocode above.

```
1  state = Environment.InitialSample()
2  totalReturn = 0
3  steps = 0
4
5        while not Environment.isTerminal(state)
6              actionValues = Agent.Network.Predict(state)
7              action = Agent.Policy.Sample(state, actionValues)
8              reward, nextState = Environment.Sample(action, state)
9              totalReturn = totalReturn + reward
10             steps = steps + 1
11             Agent.replayMethod.Store(state, action, reward,
12 nextState, isTerminal)
13             state = nextState
14             if steps > trainingConfig.explorationSteps
15                   Agent.UpdateWeights()
16
17 return totalReturn
18
```

On the line 1 the initial state is received from the environment. The received state is 1 dimensional array of numbers and could be considered to be the first observation. On the lines 2 and 3 the total return and the amount of steps the agent has taken during the episode is tracked. The loop of the episode starts at the line 5 and continues until the environment has reached a terminal state. On the line 6 the agent's network predicts the probabilities for each action and on the line 7 the best action is selected based on the policy which the agent is following. The selected action and current state is then passed to the environment which returns the reward for that action and the next state on the line 8.

The sample method of the environment translates the selected action into a movement on the grid, checks for collisions with walls, and then updates the agent position on the grid. After this the environment checks if the agent has reached the goal position and if the environment is in a terminal state. After this the environment returns the reward and the next updated state.

On the line 11 the agent stores the states, action and the received reward to the replay memory which could be later used when updating the weights. The replay memory could be considered as an experience of the agent, and it does not reset between episodes. Decay rate parameter may be used to adjust the rate at which old experiences are discarded in favour of new experiences. On the line 14 the code checks if the agent has performed more steps than configured amount of exploration steps. If the agent is done exploring the "UpdateWeights" method is called which updates the network and the agent learns. At the end total return of the episode is returned.

5.2 Scenarios

The experiments were done using three different scenarios with different grid sizes: 8x8, 10x10 and 12×12. Each scenario has basic maze, starting point for the agent and a goal. Training time and difficulty dramatically increases on larger grid sizes and because of limited time and resources larger grids were scoped out from the thesis. As seen with Minigrid environments multiple episodes are needed even in a small grid. However, with A-star algorithm one larger scenario of 50×50 grid was also tested because the algorithm is able to produce a path in a reasonable time and does not require time for training.

For example, it took 56 minutes to train 12×12 scenario with 2000 episodes using 230 nodes and double Q-learning. The training was done using CPU with 12 cores and 24 threads running at 3.7GHz and 32 GB of system memory. The cores were fully utilized during training in multithreaded training process. The training could have been faster using a supported GPU, but the author did not have such hardware available.

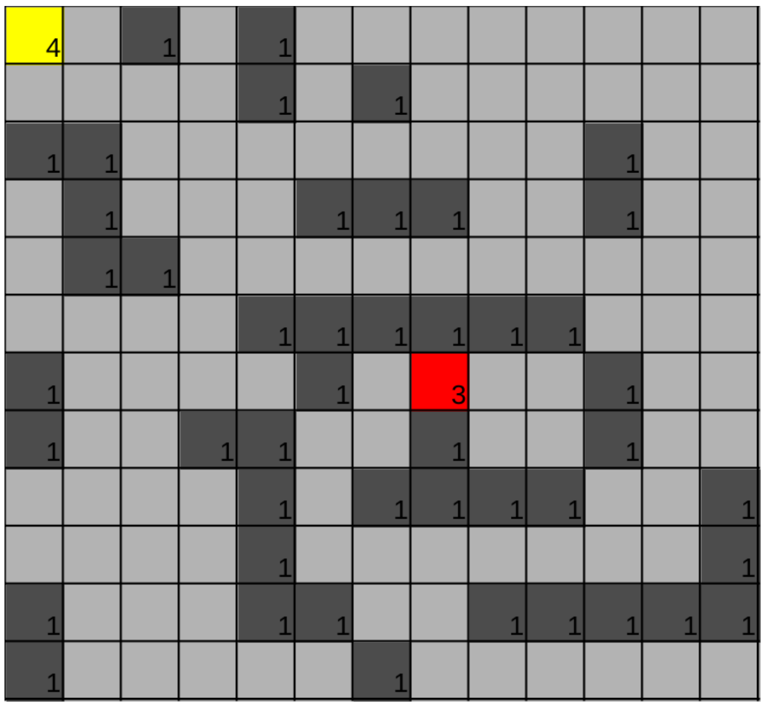


Figure 31: Simple 12x12 scenario

Figure 31 above displays example of the 12x12 environment. Number 1 represents wall, number 4 is the starting position of the agent and number 3 is the goal. The empty areas are represented with number 0 in the raw CSV file. The simulation program loads the CSV file and converts it into a matrix and handles it in memory.

5.2.1 Scenario 8x8

Following table below indicates the parameters used to train the agent for the scenario with 8 tiles width and height.

Reward on success	100
Maximum steps per episode	100
Episodes	200
Gamma	0.9
Exploration steps	3000

Step size	0.001
Hidden network layers	12
Epsilon	1
Anneal interval	1000
Minimum epsilon	0.1
Decay rate	0.99
Batch size	600
Capacity of replay memory	8000
Alpha	0.3
Double Q-learning	enabled

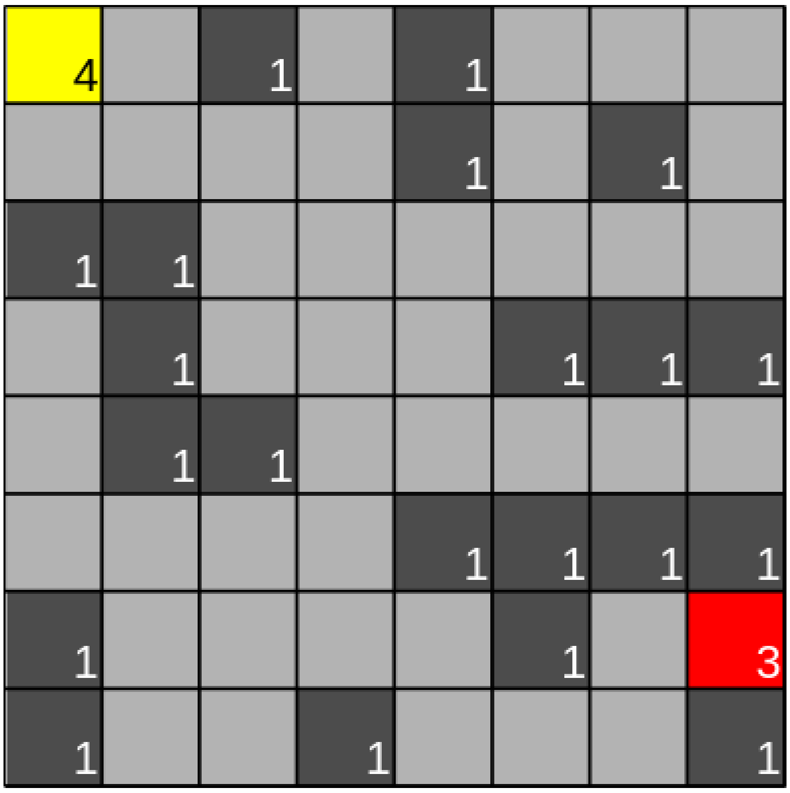


Figure 32: 8x8 scenario

Figure 32 shows the scenario used for 8x8 tests. The path is relatively simple and straight forward.

The table below shows the performance results taken from perf tool.

Metric	A-Star	Q-Learning
Task Clock	91.51 msec	22,154.91 msec
CPUs Utilized	1	23.71
Context Switches	0	0
CPU Migrations	0	0
Page Faults	285	1934
Cycles	461162625	112409556671
Stalled Cycles Frontend	326596	177896052
Stalled Cycles Backend	174762	180411714
Instructions	2232980352	17144859465
Branches	366868076	3606252130
Branch Misses	69102	11097782
Time Elapsed	0.091951832 sec	0.934264056 sec
User Time	0.091762 sec	21.970696 sec
System Time	0.000000000 sec	0.112634 sec

The table below shoes the memory usage report from memusage program.

Metric	A-Star Value	Q-Learning Value
Heap Total	134318088	61456647
Heap Peak	173403	1251230
Stack Peak	5504	34432
Malloc Calls	3404615	54506
Malloc Total Memory	134313944	61442799

Realloc Calls	0	1
Realloc Total Memory	0	200
Calloc Calls	9	33
Calloc Total Memory	4144	13648
Free Calls	3404644	54535
Free Total Memory	134237788	61367436

The agent was able to find a path in 0.09756 milliseconds. A-star algorithm resolved the path in 0.0094 milliseconds. This is 164% difference. Both methods produced equal and correct path. Only 12 hidden nodes were needed for the neural network and 200 training episodes.

5.2.2 Scenario 10×10

In 10x10 grid the parameters to train the agent model were same than in 8x8 grid.

Reward on success	100
Maximum steps per episode	100
Episodes	200
Gamma	0.9
Exploration steps	3000
Step size	0.001
Hidden network layers	12
Epsilon	1
Anneal interval	1000
Minimum epsilon	0.1
Decay rate	0.99
Batch size	1000
Capacity of replay memory	10000

Alpha	0.3
Double Q-learning	enabled

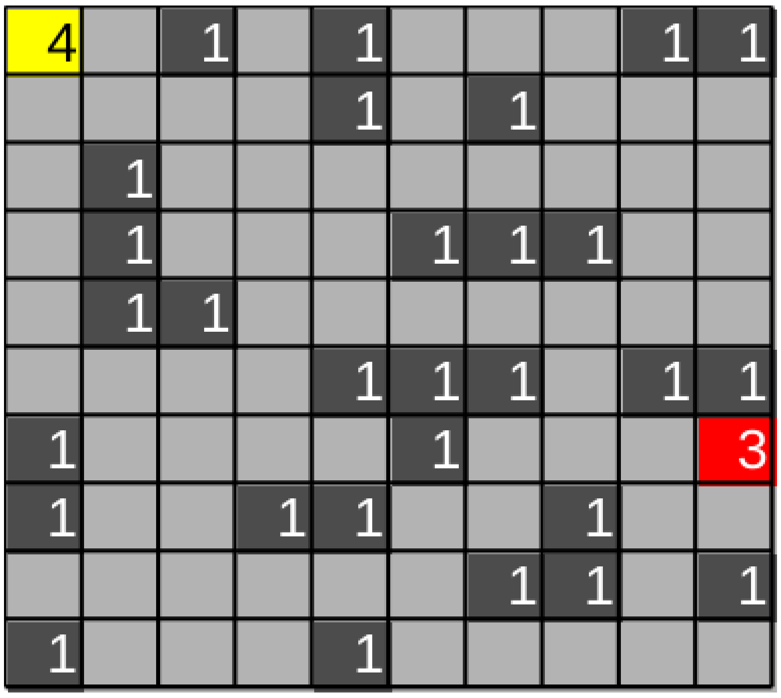


Figure 33: 10x10 scenario

The picture 33 above shows the scenario for 10x10 grid and the table below displays the results from perf tool.

Metric	A-Star	Q-Learning
Task Clock	129.15 msec	45,574.12 msec
CPUs Utilized	1	23.79
Context Switches	0	0
CPU Migrations	0	0
Page Faults	283	1737
Cycles	646978546	231033978869
Stalled Cycles Frontend	830430	255149289

Stalled Cycles Backend	437637	467571695
Instructions	3074571144	37008271578
Branches	507766718	7431999777
Branch Misses	179095	27175035
Time Elapsed	0.13 sec	1.92 sec
User Time	0.13 sec	45.24 sec
System Time	0	0.16 sec

The table below shows the memusage program report.

Metric	**A-Star**	**Q-Learning**
Heap Total	207460975	63069097
Heap Peak	175331	2332867
Stack Peak	5504	36320
Malloc Calls	4534913	54581
Malloc Total Memory	207456831	63055249
Realloc Calls	0	1
Realloc Total Memory	0	200
Calloc Calls	9	33
Calloc Total Memory	4144	13648
Free Calls	4534976	54644
Free Total Memory	207380675	62979886
Failed Calls	0	0

A-Star was able to solve the path in 0.0122 milliseconds and Q-learning agent in 0.20918. The difference between the algorithms is 177%. The difference was very small compared to 8x8 grid.

5.2.3 Scenario 12x12

Training the agent in 12x12 scenario took around 24 minutes. The table below shows the training parameters. The scenario seemed to be much more complex to solve for Q-learning agent and much more capacity was needed for replay memory. Agent was also trained for 1000 episodes and with 32 hidden layers in the neural network. Lots of exploration steps were also needed in order the agent to find the goal and gather data of the environment.

Reward on success	100
Maximum steps per episode	200
Episodes	1000
Gamma	0.9
Exploration steps	100000
Step size	0.001
Hidden network layers	32
Epsilon	1
Anneal interval	80
Minimum epsilon	0.1
Decay rate	0.99
Batch size	10000
Capacity of replay memory	30000
Alpha	0.3
Double Q-learning	enabled

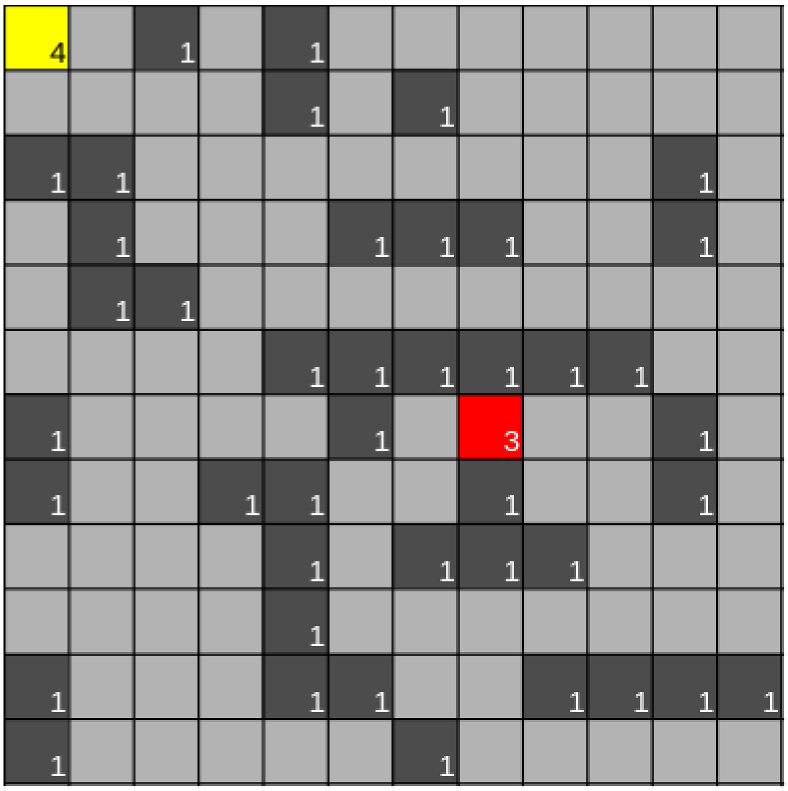


Figure 34: 12x12 scenario

The image 34 above shows the grid of the 12x12 scenario. This scenario was more complicated than the previous ones. Direct path to the goal is blocked and there are two possible ways to reach to goal. The table below shows the perf tool results.

Metric	A-Star	Q-Learning
Task Clock	615.02 msec	90,701.98 msec
CPUs Utilized	1	23.69
Context Switches	0	0
CPU Migrations	0	0
Page Faults	283	2539
Cycles	3116252089	456035836119
Stalled Cycles Frontend	5730643	650481894
Stalled Cycles Backend	1259980	745496899

Instructions	15281440166	80939267017
Branches	2438871954	15994840553
Branch Misses	775664	44179810
Time Elapsed	0.615339501 sec	3.828502132 sec
User Time	0.613536 sec	90.203932 sec
System Time	0 sec	0.242531 sec

The table below shows the memusage report.

Metric	A-Star	Q-Learning
Heap Total	774619878	127329944
Heap Peak	176459	4100667
Stack Peak	5504	37472
Malloc Calls	18577774	64637
Malloc Total Memory	774615734	127316096
Realloc Calls	0	1
Realloc Total Memory	0	200
Calloc Calls	9	33
Calloc Total Memory	4144	13648
Free Calls	18577862	64725
Free Total Memory	774539578	127240733
Failed Calls	0	0

A-star was able to solve the path correctly in 0.0604 ms and Q-learning in 0.2829 ms. The difference between the algorithms was 129%. The assumption is that the large amount of obstacles in a maze like environment caused the agent to reach the goal much slower pace, and it often got stuck in corners or dead ends. Q-learning utilized almost all 24 CPU threads with 456,035,836,119 CPU cycles and 80,939,267,017 instructions which indicates how much work was done. On the other hand A-star utilized only 1 CPU thread as expected from single threaded algorithm. CPU has performed fewer cycles 3,062,096,586 and 15,267,370,665 instructions. Memory usage for A-star was higher. Totally it used 738 MB heap memory during the execution. However, during the execution the peak allocation of heap memory was 3.91 MB. This indicates the maximum amount of memory that was allocated at any point during the process. Q-learning allocated totally 121 MB heap memory, and it peaked at 0.168 MB. A-star memory usage could be improved by more efficient data structure.

5.2.4 A-star at 50x50 scenario

A-star algorithm was able to solve the 50x50 grid scenario in 0.7418 milliseconds on average of 10000 test runs and the solved shortest path was 119 steps. The figure 35 below shoes the calculated path using green colour on 50x50 grid.

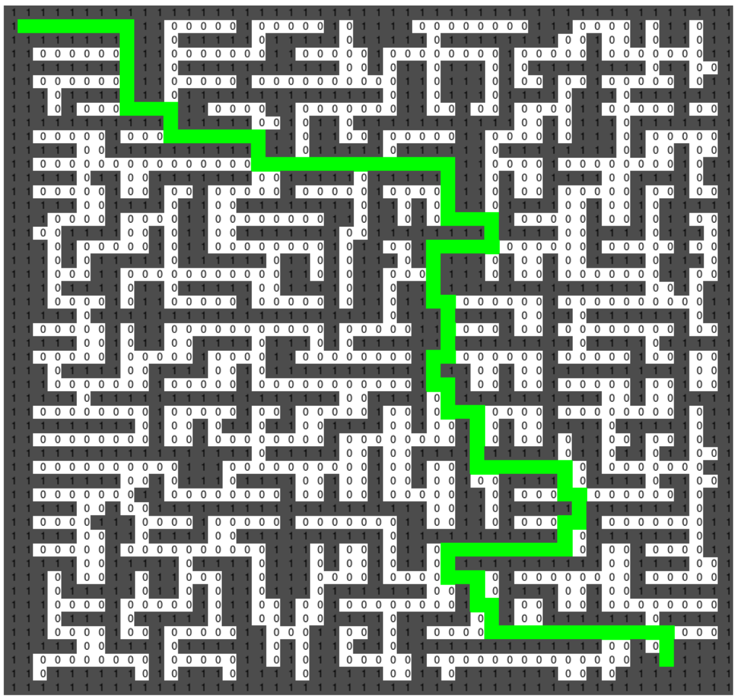


Figure 35: A-star path calculation on 50x50 grid

The calculation is still fast even though the processor has to do numerous operatios and explore multiple nodes in order to find the correct path. Q-learning was not attempted for 50x50 scenario because of the lack of resources and possible long training time required. However, would be interesting to get results in a future development.

5.3 Analysis

As expected and indicated by the results, A-star algorithm was significantly faster than Q-learning agent even though A-star was running in a single-threaded process compared to multithreaded Q-learning process. The neural network within the deep reinforcement learning agent has to process the state information to predict the Q-values on every step. This is done using matrix calculations which are computationally intensive. A-star performs only relatively simple calculations. Manhattan distance was used for calculation of the heuristic function. It involves straight forward arithmetic operations which do not require as much computing power. The use of a priority queue data structure allows the A-star algorithm access and remove the node with the highest priority efficiently because it can avoid going through the entire list of nodes. The nodes are already stored to the queue based on the priority.

In order to keep game play smooth the CPU calculations on each frame should not take too much time. As mentioned in the chapter two, each frame in the game should not take longer than 16.6 milliseconds in order to achieve 60 frames per second game-play. As seen in all the perf tool measurements, Q-learning fully utilized the CPU while predicting the optimal path. This might not leave any room for adding additional threads for other types of processing. A-star on the other hand utilized only once CPU and was still faster than Q-learning. Based on the test scenarios the processor will have plenty of time to do any additional calculations before it affects frame time. For example, it took only 0.7418 milliseconds in 50x50 grid to calculate the path using A-star.

The performance of the path finding could be further improved by dividing the game world graph into smaller hierarchical graphs in case there are lots of nodes. For example the world could be divided into larger areas and the algorithm would first calculate the path to the area where the final destination is located. A player needs to navigate from a building to another building in a game. First graph would represent the rooms and areas in the current location. After player has navigated out the building a new graph would be calculated representing the streets of the area and path would be calculated from the building to the destination building. Final graph would be calculated when player reaches the destination building and goes inside. Using this graph player would then navigate to the final destination. This could yield significant computational advantage

(Harri Antikainen, 2013). However, the complexity of the implementation increases because the transition points of the graphs would need to be defined.

In a simple example and using the A-star results from the scenarios, if an agent needs to navigate through 8x8 graph and then 12×12 sub-graph in order to enter the final 50×50 graph. This would take 0.8116 milliseconds based on calculation 0.0094 + 0.0604 + 0.7418. If we added 0.5 milliseconds for transitioning from the current graph to the next the result would be 1.8116 milliseconds. This would still leave 14.7884 milliseconds for the processor to do additional calculation without affecting the desired frame rate.

Frame rate	**Milliseconds per frame**	**Time left after A-star (ms)**
30	33.33	31.52
60	16.66	14.78
144	6.94	5.12
240	4.16	2.35

The table above displays comparison against different frame rates. In fast-paced game targeting 144 frames per second there would only be 5.12 milliseconds left for other calculations and just 2.35 when targeting 240 fps. However, this issue could be solved by multithreading where A-star calculates the path in a separate thread while main processing is done in other threads. Based on the experiments and the table above A-star will have no timing issues calculating the path as long as the path calculation is done in a separate thread.

A-star and its variants are well researched, and many ready-made implementations exists in different programming languages. In game development and in programming in general it usually makes sense to use existing well tested components in order to save resources. A-star and its variants are flexible and may adapt to different kinds of environments by adjusting different aspects and logic of the algorithm. For example by adjusting heuristics calculation methods based on the problem. Additional logic can be included to the calculation, for example avoiding or preferring certain types of nodes or including waypoints to the path. Variants like D-star avoid reordering the priority queue when recalculating the path which adapts to dynamic environments.

6 DISCUSSION AND CONCLUSIONS

As seen in the OpenAI Five model experiments with Dota 2 game, training a deep reinforcement learning agent is challenging, time-consuming and very resource intensive. Even with small test environments like Minigrid, it took time and required some trial-and-error effort in order to get the model to converge and solve the environment. However, the OpenAI Five research indicated that a fine-tuned model is able to beat human players in a complex game with lots of different parameters affecting the actions of the agent. Training and experimenting with such an agent is possible only in games with high budgets and resources.

There is a chance for model drifting as the game or software develops further. Model drifting means degrading of the model's performance over time. In addition to the code also the model needs to be kept up to date which needs resources and time. In order to train the agent the reward mechanism needs to be developed to the game-play which also requires additional time. Using advanced techniques like "Surgery" developed by OpenAI, could reduce the time and cost retraining a model. Using surgery method OpenAI was able to incorporate changes and new versions to the deep learning model without restarting the training process (OpenAI 2019).

Based on the research and acquired results, and the strong arguments in favour of A-star as mentioned in the chapter 5.3, it is better to use traditional A-star and it's variants for sole purpose of path finding instead of training a deep reinforcement learning model. This thesis focused only on Q-Learning and PPO algorithms. Some other reinforcement learning algorithms might however yield better results and with less resources. However, as mentioned in the previous chapters implementing a reinforcement learning algorithm is challenging and time-consuming. A-star on the other hand, is straight forward to implement and has good performance. This is another argument against reinforcement learning for path finding.

There might be other game-play areas which could benefit from reinforcement learning other than path finding. In a future research it would make sense to focus on a single reinforcement learning algorithm. The research could focus on aspects of the game-play which traditionally require lots of logic based programming for the AI. For example in a turn based strategy game a player needs to manage a city, resources, and army and make different kinds of decisions to improve position. The opponent AI could be trained to take the best decision using reinforcement learning. The model could be trained using a super computer during development and then distributed to player's device. Gaming computers often have powerful GPUs which could calculate predictions efficiently. Latest mobile device chips now-days also include an "AI-chip" built in. In turn based games all processing can be done at the end of each turn which would not affect to the enjoyment of the game as much as in real-time games.

Playing against an agent which always beats human players is not enjoyable. Reinforcement learning agent by its nature tries to maximize the reward it receives from the environment. This could add additional challenge when training a model because it should not be "too good" when playing against humans. This could potentially be mitigated by adjusting the agent's reward mechanism so that in a long run greater reward is accumulated if the agent does not win all the time. Player feedback could be collected during game-play and the model could be adjusted based on that. The trained model could be then distributed to players in regular game updates.

It would be interesting to research how large language models would perform as an alternative to reinforcement learning. The rules of the game could be inputted as a prompt to the model. The model could be asked to provide output which can be easily parsed using a programming language, so it could be translated to actions in a game. Also, one target of research could be to see how large language models solve mazes that this thesis experimented with.

REFERENCES

Abdul Rafiq 2020. Pathfinding Algorithms in Game Development. Search date 2.9.2023. https://iopscience.iop.org/article/10.1088/1757-899X/769/1/012021

Aimlabs. 2023. Discovery. Search date 16.12.2023. https://aimlabs.com/discovery

Alexander Zai, Brandon Brown 2020. Deep Reinforcement Learning in Action. United States, New York: Manning Publications, 1. Search date 17.6.2023

Armadillo 2023. About. Search date 26.8.2023. https://arma.sourceforge.net/

Baeldung.com. 2023. Epsilon-Greedy Q-learning. Search date 28.10.2023. https://www.baeldung.com/cs/epsilon-greedy-q-learning

Benjamin F. Janzen and Robert J. Teather 2014. Is 60 FPS Better than 30? The Impact of Frame Rate and Latency on Moving Target Selection. Search date 2.9.2023. http://dx.doi.org/10.1145/2559206.2581214

Ensmallen 2023. flexible C++ library for efficient numerical optimization. Search date 26.8.2023. https://ensmallen.org/

Farama foundation 2023. Locked Room. Search date 20.8.2023. https://minigrid.farama.org/environments/minigrid/LockedRoomEnv/

Farama foundation 2023. Minigrid. Search date 20.8.2023. https://minigrid.farama.org/

Francois Chollet 2021. Deep Learning with Python, Second Edition. United States, New York: Manning Publications, 1. Search date 17.6.2023

Godotengine 2023. Using NavigationMeshes [online image]. Search date 19.8.2023. https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_using_navigationmeshes.html#navmesh-for-3d-gridmaps

Hado van Hasselt, Arthur Guez and David Silver. 2015. Deep Reinforcement Learning with Double Q-learning. Search date 28.10.2023. https://arxiv.org/pdf/1509.06461.pdf

Harri Antikainen. 2013. Using the Hierarchical Pathfinding A* Algorithm in GIS to Find Paths through Rasters with Nonuniform Traversal Cost. Search date 20.1.2024. https://doi.org/10.3390/ijgi2040996

Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo and Youn-Hee Han. 2020. Federated Reinforcement Learning for Training Control Policies on Multiple IoT Devices. Search date 30.10.2023. https://doi.org/10.3390/s20051359

Keras. 2023. MNIST digits classification dataset. Search date 21.10.2023.
library. Search date 26.8.2023. https://mlpack.org/static/pub/2023mlpack.pdf

Mariam Kiran and Melis Ozyildirim. 2022. HYPERPARAMETER TUNING FOR DEEP REINFORCEMENT LEARNING APPLICATIONS. Search date 29.10.2023. https://arxiv.org/pdf/2201.11182.pdf

Miguel Morales 2020. Grokking Deep Reinforcement Learning. United States, New York: Manning Publications, 1. Search date 17.6.2023

Mlpack 2023. mlpack 4: a fast, header-only C++ machine learning

OpenAI 2018. Learning Dexterity. Youtube. Search date 17.6.2023. https://www.youtube.com/watch?v=jwSbzNHGflM. Screenshot 00:00:42.

OpenAI. 2021. Dota 2 with Large Scale Deep Reinforcement Learning. https://arxiv.org/abs/1912.06680

Paul E. Black 2019. Manhattan distance. Search date 23.9.2023. https://xlinux.nist.gov/dads/HTML/manhattanDistance.html

Richard E. Korf. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search*. Search date 23.9.2023. https://www.cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf

S. Koenig and M. Likhachev. 2001. Incremental A*. Search date 23.9.2023. https://proceedings.neurips.cc/paper_files/paper/2001/file/a591024321c5e2bdbd23ed35f0574dde-Paper.pdf

S. Koenig and M. Likhachev. 2002. Improved fast replanning for robot navigation in unknown terrain. Search date 23.9.2023. https://ieeexplore.ieee.org/document/1013481

Stockfish chess engine. 2023. Documentation. Search date 7.10.2023. https://disservin.github.io/stockfish-docs/pages/Home.html

Sven Koenig and Maxim Likhachev. 2002. D* Lite. Search date 30.9.2023. https://aaai.org/papers/00476-d-lite/
Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. 2016. PRIORITIZED EXPERIENCE REPLAY. Search date 28.10.2023. https://arxiv.org/pdf/1511.05952.pdf

Toni Lääveri. 2017. Integrating AI for Turn-Based 4X Strategy Game. https://www.theseus.fi/handle/10024/134060