Ben-Gurion University

Faculty of Engineering Science

Department of Software and Information Systems Engineering

Search in Artificial Intelligence

Prof. Ariel Felner

Course Project

Authors:	*IDs:*
Yuval Heffetz	302957139
Matan Zuckerman	201648219

February 12, 2019

Contents

1 Introduction

Heuristic search algorithms and reinforcement learning (RL) methods are often applied on similar or overlapping search tasks, even though they are using completely different approaches. While heuristic search algorithms such as A-star are used online and do not have a learning phase, RL methods learn the specified environment iteratively and improve over time.

Recently a new field called deep reinforcement learning (DRL) has evolved. DRL algorithms use deep neural networks along with RL methods to learn from experience and generalize across states so a policy can be predicted when a new state is encountered, allowing for online use of trained models. One such algorithm is Deep Q-Networks (DQN), which is a variation of the familiar q-learning algorithm. DQN uses a neural network for predicting the sum of future rewards for taking an action in a certain state (called "q-values").

In this project we propose a new algorithm that aims to integrate heuristic search with DRL by using DRL's predictive properties in a heuristic search algorithm. More specifically we use the predicted q-values of a trained DQN model to reorganize the open list in the A-star algorithm by modifying the heuristic function with respect to the order of the q-values, resulting in an algorithm we call *Q-star*, as will be detailed in section 4. As detailed in section 6, the proposed algorithm demonstrates improved behaviour compared to the A-star baseline when the q-values are accurately predicted.

We chose the maze pathfinding task for evaluating Q-star since it is a classic search task, solved successfully and optimally both by A-star and the q-learning algorithms [9]. In the following sections we provide a short background, list the goals of the project, detail the methodology and perform an evaluation of the algorithm compared to the baselines.

2 Background

2.1 Pathfinding

Pathfinding in computer games has been an investigated area for many years. It is just about the most popular but very difficult Artificial Intelligence (AI) problem.

Pathfinding could be used to give answers to the question "How do I get from source to destination?". In most cases, the path from source (current point) to the destination (next point) could possibly include several different solutions, but if possible, the solution needs to cover the following goals:

1. The way to get from source A to destination B.
2. The way to get around obstacles in the way.
3. The way to find the shortest possible path.
4. The way to find the path quickly.

Several search algorithms including A-star search algorithm, Bread-First search algorithm and Depth-First search algorithm, were created to solve the problem of finding the shortest path [3] and achieved great success in it.

One of the pathfinding tasks is to solve a 2 dimensional maze. Each maze contains a starting point from where the agent is at the beginning of the game. Ending point, The goal of the maze, where the agent suppose to go. Walls that are limiting the movement of the agent. In the maze, the pathways and walls are fixed

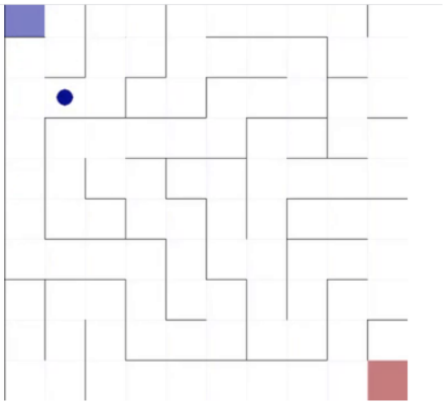


Figure 1: An example for a 2D maze game.

This task is highly prioritized in the video game industry. Many researches were conducted in order to propose a new algorithms or improve familiar ones in order to achieve better results [1][3][7]. Some of the researches acknowledged the outstanding performances of the A-star algorithm for this task and optimized it [3].

2.2 A Star

A-star is a best-first search algorithm, which is highly used in order to find an optimal path on weighted graphs. Starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does it by maintaining an open list and closed list. At each step of the algorithm, the node with the lowest cost function f(x) value is removed from the open list and goes into the closed list, and the children of than

node are added into the open list with their f value. The algorithm continues until a goal node was found or the open list is empty. The f value of the goal is then the cost of the shortest path. The cost function f(x) is defined as follow:

$$f(x) = g(x) + h(x)$$

x is the next node on the path, g(x) is the cost of the path from the start node to x, and h(x) is a heuristic function that estimates the cost of the cheapest path from x to the goal. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A-star is guaranteed to return a least-cost path from start to goal as was described in [5] under solution quality chapter (3.5.2).

For pathfinding, A-star algorithm examines the most offering unexplored location it has seen. When a location is examined, the A-star algorithm is completed when that location is the goal. In any other case, it helps make note of all that location neighbors for additional exploration. The time complexity of this algorithm is depending on heuristic used. There are several heuristics that can be used for a A-star algorithm in a pathfinding tasks in 2D maze problems. Each heuristic calculates in a different way the distance between the agent node and the goal node:

- **Manhattan distance**:
$$\sum_{i=1}^{n} |a_i - goal|$$
- **Diagonal Distance**:

  $$\sum_{i=1}^{n} D*(ax_i - goalx_i + ay_i - goaly_i) + (D_2 - 2*D)*\min(ax_i - goalx_i, ay_i - goaly_i)$$

  where D and D_2 are parameters to be chosen
- **Euclidean distance**:
$$\sqrt{\sum_{i=1}^{n} (a_i - goal)^2}$$

A-star is known as a simple and competitive algorithm in pathfinding tasks.

Algorithm 1 A-star

```
1: procedure A-star
2:     Initialize open list open ← ∅
3:     Initialize close list closed ← ∅
4:     for each state ∈ S do
5:         g(state) = 0
6:     Insert start state in open
7:     while open ≠ ∅ do
8:         sort(open)
9:         n ← open.pop()
10:        if goal(n) = True then return makePath(n)
11:        children ← expand(n)
12:        for c ∈ children do
13:            g(c) ← g(n) + 1
14:            h(c) = euclidean_distance(c, goal)
15:            if c ∩ closed = ∅ then
16:                open.push(c)
17:        closed.push(n)
```

2.3 Reinforcement Learning

Reinforcement learning (RL) is a sub-field of machine learning. It refers to the group of algorithms and methods in which an agent is interacting with an environment and learning the optimal policy to behave in this environment to achieve a goal or a maximum reward, and It is a powerful tool for sequential decision making problems [9].

In a reinforcement learning model an agent interacts with the environment over time. Under a specific policy π, the agent forms a trajectory of actions and states where in each time-step t, the agent has to choose an action a_t given the current state s_t that will bring it to the next state s_{t+1} (i.e. s'), with a reward r_t which is obtained from the reward function $R(s, a, s')$. The underlying goal of the agent is to maximize the return, which is defined in eq. 1:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

The return is the discounted accumulated reward with a discount factor $\gamma \in (0, 1]$ which, roughly speaking, denotes how much the agent cares about current rewards in respect to future rewards.

When an RL problem satisfies the Markov property, i.e., that a state is dependent only on the previous state and action, it is formulated as a Markov Decision Process

(MDP). MDP is a formalization of sequential decision making where the actions taken in a certain step can influence the future rewards of a trajectory [10]. In an MDP, we pick a random initial state with some probability and in each time-step we choose an action a_t, as a result of which the current state s_t transitions to some next state s_{t+1} with a probability defined by $P(s_{t+1}|s_t, a_t)$. In RL, the goal is to find the **best possible action** to take in each state to maximize the expected sum of discounted rewards. When this is given, the role of the reinforcement learning can be defined as finding a *policy* which is a mapping of all states and actions $\pi : S \to A$.

Q Function. A value function is the expected accumulated discounted future rewards used to measure how good a state is. Given a policy π we can define its value function $V^\pi : S \to R$ for taking actions according to π starting from state s:

$$V^\pi(s) = E_\pi[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, s_2) + ...|s_0 = s] \tag{2}$$

Similarly, we can compute the value of a state-action pair using $Q-function$. Given a policy π we can define its Q-function $Q^\pi : S \times A \to R$ for taking actions according to π starting from state s, first taking a certain action a:

$$Q^\pi(s, a) = E_\pi[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, s_2) + ...|s_0 = s, a_0 = a] \tag{3}$$

The optimal value function can be computed by $V^*(s) = max_\pi V^\pi(s)$ and the optimal Q-function by $Q^*(s, a) = max_\pi Q^\pi(s, a)$. Using the *Bellman equations* we can decompose equations 2 and 3 to a recursive form, so that the optimal Q-function will be:

$$Q^*(s, a) = \sum_{s'} p(s'|s, a)(R(s, a, s') + \gamma max_{a'} Q^*(s', a')) \tag{4}$$

Q-Learning algorithm. Q-learning is one of the most prominent RL algorithms successfully used in many domains, pathfinding in mazes being one of them. It is a dynamic programming algorithm built on the basis of the recursive bellman equation (eq. 4).

Q-learning is considered a *model-free* algorithm, meaning that is not using any prior knowledge on the environment and its underlying dynamics. It is using the temporal difference (TD) learning method to learn the Q-function $Q(s, a)$ directly from experience with TD error in an incremental way [10]. It's a prediction problem which follows an update rule to compute an estimation of the value function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, a, s') + \gamma Q(s', a) - Q(s, a)] \tag{5}$$

The learning rate of the agent is controlled by parameter α. In Q-learning we store the Q-value of each state-action pair in a **lookup table**. The complete process can

Algorithm 2 Q-Learning Algorithm

1: **procedure** Q-Learning
2: Initialize $Q(s, a)$ arbitrarily, $\forall s \in S$, $\forall a \in A$
3: **repeat** for each episode
4: Initialize s
5: **repeat** for each step of episode
6: Choose a from s using policy derived from Q
7: Take action a, observe r, s'
8: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)]$
9: $s \leftarrow s'$
10: **until** All steps are done
11: **until** s is terminal

be seen in algorithm 2. We use the $\epsilon-greedy$ policy to pick actions in each step (line 6), by selecting an action randomly with probability ϵ or greedily with probability $1 - \epsilon$, i.e. action with the maximum Q-value [4].

The Q-values in the lookup table are bound to converge to their optimal value after a certain number of steps, directing the RL agent in the optimal path by selecting the action with the highest Q-value in each state.

Since Q-learning has an offline learning stage it is understandably less efficient than A-star for problems such as pathfinding, however we suggest in this project to utilize the learnt q-values for improving the heuristic function of A-star for a faster search.

Deep Q-Networks. Q-learning assumes that the agent visits every state (i.e., location in the maze) during the learning stage, so for the pathfinding problem it can only be used on a maze that the agent was trained on. Thus, any suggested use of q-values in the A-star process can only make it longer and inefficient.

The solution for this problem is using a neural network (NN) for approximating the q-function instead of using a lookup table of states and actions. This method was suggested by Mnih et al. [8] and is called Deep Q-Networks (DQN). The DQN algorithm follows the same steps as algorithm 2, but instead of updating the q-values in the lookup table directly it is updating parameters θ which are the weights of a NN used for predicting the Q-value of each state-action pair $\hat{Q}(s', a; \theta)$. The NN inputs are the states of the environment and its output is the predicted q-value for each action in a state. The network's parameters may be updated in every step of the algorithm or every fixed number of steps, and the resulting network is supposed to be able to generalize across states, so when a new unseen before state is inserted as an input it will be able to accurately predict its q-values relying on past experiences.

DQN can be trained offline on thousands of different mazes and be applied online on

a new unseen before maze by predicting the q-value for each state-action pair of the maze. We intend to utilize this ability by using the predicted q-values to improve the heuristics of A-star algorithm without the need to train the agent on the new maze.

3 Goals and Expected Contribution

Performing a comparison between two different pathfinding algorithms in a 2D maze environment, namely the heuristic algorithm method A-star with the RL algorithm Q-learning, while exploring the properties of each of the two methods.

Exploring the possibility of integrating the two algorithms by utilizing the properties of q-learning and neural networks for improving the heuristic function used in A-star, hopefully reducing the run-time and number of visited nodes. The main goal is to find whether providing "extra information" from the RL algorithm to A-star heuristic will improve its performance.

Our main objectives for achieving the aforementioned are as follows:

- Creating a maze environment for the algorithms to operate on.
- Designing and implementing a new proposed algorithm, integrating DQN with A-star.
- Implementing DQN and A-star algorithms separately for the maze pathfinding task.
- Evaluating and comparing the proposed algorithms with the baselines on the maze environments we created.

As far as we can tell, there is no previous research for integrating heuristic search methods with deep reinforcement learning (DRL) and more specifically A-star with DQN. The expected contribution of our proposed algorithm is predominantly to reduce the number of nodes in the search path of A-star while maintaining its ability to find an optimal path by harnessing the recent advances in the fields of RL and deep NNs.

4 Methodology

In this section we will present the methods and methodology that will be used to fulfill the project's objectives. The methodology can be divided into two main stages:

- Fitting the DQN and A-star algorithms for the specific problem of pathfinding in a 2D maze, including the design of the DQN's reward function and NN architecture and determining the A-star's heuristic function.
- The proposed algorithm for maze pathfinding.

4.1 DQN

DQN was a major breakthrough when it was first published in 2015 [8], successfully solving complicated computer games using NN and Q-learning. However, DQN hyperparameter tuning and choice of network is very tricky and may have a vast influence on its performance. For the maze environment we chose to implement DQN with a convolutional NN (CNN) [6] since it was successful in similar computer game environments. For stabilizing the network and making the problem more like supervised-learning we implemented *experience replay buffer* and *fixed q-targets*, which are two methods detailed in [8].

Defining States and Actions

As mentioned above, the DQN's network should take as input a state and output the predicted q-value for each possible action. For this to happen, we first have to translate the maze environment to state vectors and action vectors in fixed sizes. We decided to represent a state as a 2D array in the shape of the maze's grid, with zeros representing empty cells and ones indicating wall locations. The cells occupied by the player and the target are filled with the numbers 50 and 100 respectively. An example for a state representation can be seen in fig. 2. The actions vector is the output vector of the network and is an array of size four, which is the number of possible actions - {Up, Right, Down, Left}.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 50 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 100 \end{bmatrix}$$

Figure 2: Example of a state representation used as input to the CNN for a 5X5 maze. In this example the player is at coordinate [1,2] and the goal is at [4,4]

Notice that this representation is only needed when we are using a NN in q-learning (i.e., DQN) but for the basic q-learning algorithm the state is merely the coordinates of the player.

Neural Network Architecture

The architecture we set for the DQN model is built from the following layers:

1. Convolution 1: Input size - grid.shape , filters - 16, kernel = [5,5], strides - 1, Padding = True,
2. Non-linear activation: Relu
3. Convolution 2: Input: Convolution 1, filters = 32, kernel = [3,3], strides - 1, Padding = True,
4. Non-linear activation: Relu
5. Convolution 3: Input: Convolution 2, filters = 64, kernel = [2,2], strides - 1, Padding = True,
6. Non-linear activation: Relu
7. Fully connected: output size 512
8. Non-linear activation Relu
9. Fully connected: output size 512
10. Output layer - Predictions of Q value for each action

The optimizer we chose for the model is Adam optimizer. An illustration of the network can be seen in fig. 3.

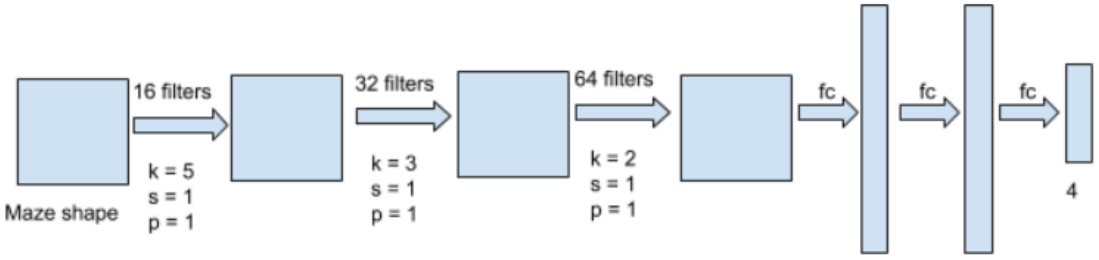


Figure 3: An illustration of the CNN architecture used for the DQN model

Reward Function

As described above, in RL environments each action taken in a state is rewarded according to a predefined reward function. The reward function can have a large influence on the performance of the model and determines the behaviour of the agent in the specific environment. In the maze environment there are four possible actions an agent can take, with three possible outcomes:

- **The agent transition into a valid cell**: The reward for this is **-0.01**.
  The reason for negative reward is to push the agent to reach the goal as fast as it can and not to "wander around" the maze.
- **The agent transition into a wall or outside the maze**: The reward is **-0.75** .
  We decided to be very strict with those actions and punish the agent so it will not learn to go into walls. In case the agent decided to go into a wall, the reward is calculated but the position of the agent is not changing from previous step.
- **The agent transition into the goal cell**: The reward is **+1**. This is the only positive reward the agent can get.

This reward function ensures that the agent will follow the shortest path (i.e., optimal) from the start state to the goal, assuming that the model accurately predicts the q-value of each state-action pair.

4.2 A-star

A-star algorithm does not require any special tuning or changes for fitting it to different environments, and specifically to the maze environment. The maze pathfinding problem is a classic search problem that A-star is known to excel at. The only decision that should be taken is which heuristic function should be used, a decision that may have a significant influence on the end result.

Heuristic Function

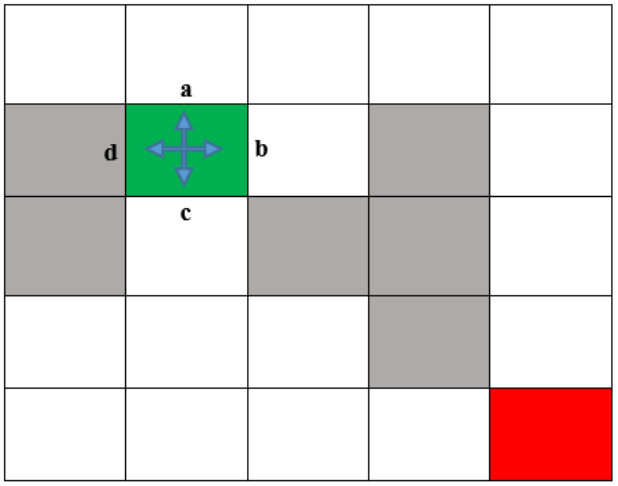
The cost function of A-star is built from two parts as we described in section 2. the heuristic function is one of them. In section 2 we also listed the three most common heuristic functions used for the pathfinding problem.

We decided to implement in this project the **euclidean distance heuristic**. This heuristic is admissible because the euclidean distance is always shorter than the distance an agent will need to travel in order to get to the goal. Therefore, using this heuristic function ensures the discovery of the optimal path for the agent as we learned in class.

4.3 Proposed Algorithm - Q-star

Our proposed algorithm, Q-star, is based on A-star algorithm but consists of a modification of the heuristic function, regardless of the chosen method of heuristics. The

heuristic function is modified in the direction of the normalized q-values, denoted by q-scores, in an attempt to improve the order of the nodes in the sorted open list, so nodes with higher q-values will be closer to the top of the list.



$$\hat{Q}_\theta(n) = [0.52, 0.61, 0.96, 0.2] \rightarrow \ Qscores = [0.773, 0.725, 0.581, 0.9127]$$

$$\boldsymbol{h(b)} = \sqrt{13} * 0.725 = \mathbf{2.6} > \boldsymbol{h(c)} = \sqrt{13} * 0.581 = \mathbf{2.09}$$

Figure 4: An example of the heuristic function modification in Q-star. In this example the walls are indicated by grey cells, the green cell is the location of the expanded node n and the red cell is the goal. The children of the node are the neighboring cells {a,b,c,d}. It is clear that without the Q-star modification, the heuristics of nodes b and c are the same ($\sqrt{13}$). After the multiplication in the $Qscores$, child c will get a smaller heuristic and thus will be selected before b when both are in the open list, directing the agent in the optimal path.

First, a DQN model is trained over an episodic environment of mazes, i.e, an environment of mazes in a fixed size but with randomly generated walls. In this environment each time the agent reaches the goal, the environment is being reset to a new maze with different wall locations. After trained over a few thousand mazes, the model's q-network $\hat{Q}_\theta$ should be able to generalize across states and to predict the correct q-value for each state-action pair.

The trained q-network is inserted as an input to the Q-star algorithm, detailed in alg. 3. The Q-star algorithm follows the steps of A-star, but, as can be seen in lines 12 and 15, introduces a change in the heuristic function.

The heuristic function in our proposed algorithm is computed as follows:

$$h(x) = euclidean\_distance(x, goal) * Qscores(x)$$

In this case we used euclidean distance as the base heuristic but it can be any other admissible and consistent heuristic function. We modify the base heuristic function by multiplying it with the q-score of the node which is computed in line 12 of alg. 3 as detailed in the procedure presented in alg. 4. $Qscores$ is a vector computed given the expanded node n. Each entry in the vector represents the q-score of a child of n which corresponds to an action taken in the state n.

The $Qscore$ vector is computed in the following way (see also alg. 4):

1. Calculate the $\hat{Q}_\theta(n)$ output for the specific state n. The output is a vector with q-value for each action corresponding to a child node c of n. The bigger the q-value is, the action is more desired. (line 2).
2. Find the minimum q-value (line 3) and push all values upwards so there will be no negative values (lines 4-6).
3. calculate the sum of the vector (line 7)
4. Normalize the vector so the sum of all values will be 1.0 and transform the values to (1-Normalized) (line 8). In this case, actions that $\hat{Q}_\theta(n)$ predicted to be less desired will get higher q-scores. All q-scores are between 0 and 1.

By multiplying the base heuristic with the q-score, the heuristic value of better actions (nodes) will be reduced while worse ones will keep the same value in the worst case. A-star algorithm will choose to expand the nodes with lower $f(x)$ which will lead to the expansion of nodes that the combination between A-star and DQN saw as the best to expand. An illustrated example can be seen in fig. 4.

Q-star algorithm is bound to produce the optimal path since it can be proven that the resulting heuristic function is admissible as long as the base heuristic function is admissible:

$$h(x) \leq h(x') : \text{ A-star has an admissible heuristic.}$$
$$h(x) * Qscores(x) \leq h(x) \leq h(x') : \text{ DQN retrieve values between 0 to 1}$$
$$h(x) * Qscores(x) \leq h(x')$$

Notice that there is no difference when using Q-learning algorithm instead of DQN in Q-star since the output of Q-learning is the same as DQN. The only difference is the input to alg. 3 that changes from q-network to a lookup table containing the q-values of each state-action pair.

We succeeded to combine two algorithms, A-star and Q-learning/DQN. As was discussed in the background A-star is an excellent algorithm for pathfinding in mazes. Adding the knowledge that DQN learned for each state and action it experienced in the past can increase A-star performance even further in means of visited states and run-time.

Algorithm 3 Q-star

Input: $\hat{Q}_\theta$ - Q-network with trained parameters θ

procedure Q-star
  Initialize open list $open \leftarrow \emptyset$
  Initialize close list $closed \leftarrow \emptyset$
  for each state $\in S$ **do**
    $g(state) = 0$
  Insert start state in $open$
  while $open \neq \emptyset$ **do**
    sort($open$)
    $n \leftarrow open.pop()$
    if $goal(n) = True$ **then return** $makePath(n)$
    $children \leftarrow expand(n)$
    $Qscores \leftarrow q\_scores(\hat{Q}_\theta, n)$ ▷ vector with normalized q-values
    for $c \in children$ **do**
      $g(c) \leftarrow g(n) + 1$
      $h(c) = euclidean\_distance(c, goal) * Qscores(c)$
      if $c \cap closed = \emptyset$ **then**
        $open.push(c)$
    $closed.push(n)$

Algorithm 4 q-scores

procedure q_scores ($\hat{Q}_\theta$, state)
  $Q \leftarrow \hat{Q}_\theta$(state) ▷ vector with q-value for each action
  $min\_q \leftarrow \min(Q)$
  if $min\_q < 0$ **then**
    $\delta \leftarrow min\_q * (-1)$
    $Q \leftarrow Q + \delta$
  $q\_sum \leftarrow sum(Q)$
  $Qscores \leftarrow 1 - (Q/sum\_q)$ ▷ vector with normalized q-values
return $Qscores$

5 Experimental Setup

In this section we will present the evaluation process and the experimental setup, including the maze environment we conducted our experiments on, the different experiments we devised, and metrics and measures we used for evaluating the proposed algorithm.

5.1 Maze Environment

We used for the purpose of this project the *maze explorer* environment of OpenAI's *gym* library[1]. This episodic environment is generating mazes in predefined size with random wall locations each time it is being reset but with same locations of start and goal states (upper left and lower right respectively). We used mazes in sizes of 10x10 and 12x12 since they were sufficient to examine the performance of the algorithm and for the comparison between the algorithms.

5.2 Experiments

We conducted two main experiments which will be described below: comparison between A-star, Q-learning and Q-star (based on Q-learning) trained on previously seen mazes for proof of concept and sanity check; comparison between A-star, DQN, and Q-star (based on DQN) on new unseen mazes for evaluating the proposed algorithm.

Experiment 1: Proof of Concept with Q-learning

The purpose of this experiment is to compare the performance between A-star, Q-star and Q-learning algorithms **on the same mazes which Q-learning was trained on**.

Assuming Q-learning outputs the optimal path to the goal when trained long enough on the same maze, using Q-learning in the Q-star algorithm will be able to tell us whether the algorithm improves the heuristics and the results in a "perfect world" where the q-values are definitely accurate. In this experiment we aim to check whether the q-values are indeed providing "extra knowledge" for the heuristic function when they are accurate and whether the performance of the Q-star will improve over A-star. Q-learning is also evaluated in order to examine whether this algorithm by its own manages to find the optimal path with the reward function that we used.

This experiment serves only as a proof of concept of our algorithm since the training phase in this experiment which is conducted on the same mazes that are later being used for testing the algorithm is clearly making the contribution of the algorithm irrelevant (in the time q-learning trains offline, A-star can find the path several times online).

We conducted this experiment on 10 mazes in size 12x12. Each time the Q-learning was trained on the specific maze until convergence. After finishing training, the

[1] https://github.com/mryellow/gym-mazeexplorer

three algorithms are tested on the same maze and all measures are collected. After 10 mazes the average score is calculated.

Experiment 2: Q-star Tested with DQN

The purpose of this experiment is to compare the performance between A-star, Q-star and DQN algorithms **on mazes which DQN was not trained on**.

For A-star there is no difference between the two experiment as A-star calculates its cost function online for each node in the graph. For DQN and Q-star (based on DQN) it's a completely different thing. In the first experiment the agent already visited all of the states in the tested maze and it "learned" what is the best path for it to take. In this experiment however, there is still an offline learning phase performed by the DQN model, but the learning is conducted on several thousand mazes that are **not** used for the online test phase. In this case the offline learning phase can take place only once and then the trained model's NN can be used countless times for online purposes and new mazes.

In this experiment we will be able to examine whether DQN can improve A-star in new mazes based on similar mazes it saw in the past. Here also we evaluated DQN separately in order to check if by its own it can be used for pathfinding tasks.

In this experiment we trained the DQN algorithm on 1700 different mazes in size 10x10. We introduced an additional parameter - the level of difficulty of the maze to help us gain more insight in the evaluation. The level of difficulty of the maze is determined by the number of nodes that the agent expands in A-star algorithm:

- **Level 1**: number of nodes < 30
- **Level 2**: $30 <$ number of nodes < 40
- **Level 3**: number of nodes > 40

After finishing training, we tested the three algorithms on 10 different new mazes in each level of difficulty. The results of the different metrics were collected and the average score was calculated.

5.3 Metrics

Below are the metrics which we used in order to compare the different algorithms in the experiments mentioned above:

- Number of states each algorithm visited in the online mode.
- Running time of the algorithm in online mode.

- Additional memory usage for Q-star algorithm (The memory complexity difference between the two algorithms is due to the "additional knowledge" Q-star has, which is on higher order of memory complexity compared to the other memory usages in both algorithms).

6 Results

6.1 Experiment 1

Table 1: Number of states and elapsed time over ten different mazes

	Num of States			**Elapsed Time [ms]**		
Maze #	**A-star**	**Q-star**	**Q-learn**	**A-star**	**Q-star**	**Q-learn**
1	53	35	20	2023	1383	727
2	42	21	20	1491	674	678
3	51	49	20	1644	1455	625
4	41	41	24	1347	1197	739
5	46	29	20	1347	874	633
6	57	57	20	1787	1862	687
7	48	22	20	1446	733	614
8	27	22	20	804	665	616
9	67	61	24	2007	1919	852
10	61	61	20	1908	1815	648
Average	**49.3**	**39.8**	**20.8**	**1580**	**1258**	**682**

The cost of using Q-star in this experiment is the **memory of the q-values lookup table**: 12x12 (size of the maze) x32 (bytes for float) x4 (valid actions) = 18432 bytes. Besides that, both algorithms are bound by the same complexity of space and therefore it's neglectable.

6.2 Experiment 2

Difficulty Level 1

Table 2: Number of visited states and elapsed time over mazes of difficulty level 1

	Num of States		**Elapsed Time [ms]**	
Maze #	**A-star**	**Q-star**	**A-star**	**Q-star**
1	29	25	984	1204
2	24	21	827	1273
3	27	23	865	937
4	27	31	925	1437
5	17	17	528	1034
6	19	19	684	1032
7	26	26	808	1760
8	17	17	524	936
9	27	27	859	1292
10	22	27	752	1385
Average	**23.5**	**23.3**	**776**	**1229**

Difficulty Level 2

Table 3: Number of visited states and elapsed time over mazes of difficulty level 2

	Num of States		**Elapsed Time [ms]**	
Maze #	**A-star**	**Q-star**	**A-star**	**Q-star**
1	37	27	1274	1676
2	30	31	1536	2177
3	39	21	2529	1721
4	37	37	1382	2296
5	37	39	1413	1914
6	35	35	1228	1661
7	37	37	1226	1879
8	33	31	1122	1959
9	33	32	1139	2172
10	31	21	975	1075
Average	**34.9**	**31.1**	**1382**	**1853**

Difficulty Level 3

Table 4: Number of visited states and elapsed time over mazes of difficulty level 3

	Num of States		**Elapsed Time [ms]**	
Maze #	**A-star**	**Q-star**	**A-star**	**Q-star**
1	41	41	1393	1887
2	41	41	1415	1708
3	40	29	1228	1350
4	40	19	1224	975
5	41	33	1309	1560
6	41	17	1331	951
7	42	37	1324	1652
8	41	35	1310	1835
9	45	45	1484	1990
10	43	43	1351	1782
Average	**41.5**	**34**	**1337**	**1569**

The cost of using Q-star in this experiment is the **memory of the trained Q-network**: 7110000 bytes. Besides that, both algorithms are bound by the same complexity of space and therefore it's neglectable.

The increase of time even when the number of states decreases with Q-star can be attributed to the time of loading the neural network model which occurs only once for each task.

7 Discussion

The results of the first experiment show that when the q-values of each state-action pair in the maze are converged and predict the true sum of future rewards for taking an action in a certain state, they introduce significant improvement when used in our proposed algorithm, Q-star. This improvement indicates that the knowledge transfer from the q-learning model to the heuristic function used by A-star does enhance the heuristics and shorten the search process when done according to our proposed algorithm.

The chosen reward function did cause the agent to find the optimal path in every test using the q-learning algorithm in the first experiment, indicating the choice was correct and that it can be used in the second experiment as well, and that it could not be blamed for unsuccessful pathfinding attempts by the DQN agent.

Not surprisingly, the second experiment did not result in a DQN agent that learns to navigate on its own in new mazes using only the past experiences in an optimal manner since it is known to be a very difficult task [2]. However, when the trained Q-network was used by the Q-star model in mazes of high difficulty, it did manage to point the agent to the right direction better than the euclidean distance heuristic, and thus reduce the number of states it visited (in average). This happened with an increase in memory usage as Q-network is using more memory than the lookup table.

A disadvantage of Q-star is the offline training time. In case the pathfinding task is needed to be done once and there is no past knowledge, A-star will perform better than the first algorithms. In case this is a frequent task than using Q-star which was trained on past experience will improve the performance. Another disadvantage is the memory size. Q-star, which can either be based based on a lookup table or Q-network, takes up significantly more memory than A-star which do not have the above additions. However, the additional memory of using a neural network is fixed and is not influenced by the size of the search space. In larger spaces this fixed increase should be irrelevant.

Both experiments demonstrated the potential of Q-star, but further research still needs to be done, especially regarding the model used by the algorithm to predict the q-values. The DQN model can be further explored and tuned for the maze problem so it will be able to navigate in new mazes. The field of deep reinforcement learning is evolving rapidly so an improved DQN that would achieve this goal is certainly a do-able task. When DQN will manage to accurately predict the q-values we can know for certain that Q-star can improve the performance of A-star, as was shown in the first experiment.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:7, 2015.

[2] Shurjo Banerjee, Vikas Dhiman, Brent Griffin, and Jason J Corso. Do deep reinforcement learning algorithms really learn to navigate? 2018.

[3] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[4] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2-3):209–232, 2002.

[5] Richard E Korf. heuristic search. In *heuristic search*, pages 0–253. Computer science departmnet, 2008.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Geethu Elizebeth Mathew and G Malathy. Direction based heuristic for pathfinding in video games. In *Electronics and Communication Systems (ICECS), 2015 2nd International Conference on*, pages 1651–1657. IEEE, 2015.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[9] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.