Institute of Computer Science IV
University of Bonn
Bachelor of Science in Computer Science

Comparing the Robustness of Reinforcement Learning Based Routing Algorithms Against Adversarial Attacks in Reinforcement Learning

Bachelor Thesis

Submitted by

Florian Spelter

Matriculation Number: 3215131

Email: s6flspel@uni-bonn.de

Examiner: Prof. Dr. Michael Meier[1] and Dr. Paulo H. L. Rettore[2]

Supervisor: Johannes Loevenich[2]

[1]Head of Institute Computer Science IV, University of Bonn, Germany
[1]Head of Institute, Fraunhofer FKIE, Wachtberg, Germany
Email: meier@cs.uni-bonn.de

[2]Research Scientists, Fraunhofer FKIE, Bad Godesberg, Germany
Email: [paulo.lopes.rettore, johannes.loevenich,] @fkie.fraunhofer.de

In collaboration with the Fraunhofer Institute for Communication, Information Processing and Ergonomics (FKIE), Bonn, Germany

February 14, 2023

1 Abstract

Reinforcement Learning (RL) has been widely applied in diverse fields, including communication, and particularly military communication, where robustness of communication networks is crucial due to high-security standards. To ensure secure and reliable connections, several RL-based routing algorithms have been developed. This thesis evaluates the robustness of established RL algorithms against external disturbances, with a specific focus on adversarial attacks, and investigates their coping mechanisms. Additionally, the thesis explores the effects of natural disturbances on these algorithms.

2 Acknowledgement

My personal thanks go to my examiners, namely Prof. Dr. Michael Meier and Dr. Paulo H. L. Rettore and my advisor Johannes Loevenich at Fraunhofer FKIE who made this thesis possible in the first place. Their long-term and continuing support, suggestions, motivation, and patience throughout the process of the entire thesis are indispensable.

Moreover, I am very grateful to my fellow students Luca Liberto, Jonas Bode and Tobias Hürten. The cooperation with these extraordinary students over the last year considerably affected the quality of this thesis.

Contents

3 Introduction

Reinforcement Learning (RL) algorithms have been widely used in mobile ad-hoc networks (MANETs) for the purpose of routing. Tactical Networks (TN) are MANETs employed in military operations, where high levels of security and reliability are crucial, since the success of an operation and the lives of soldiers may depend on the functioning of these communication networks. In addition, there has been a significant increase in the public use of MANETs in recent years due to the growing prevalence of wireless networks in daily life. As a result, the safety and reliability of these networks must be guaranteed in a variety of applications.

For decades, Reinforcement Learning (RL) has been used to provide solutions for routing in mobile ad-hoc networks (MANETs). These networks are often used in the context of military operations and public applications, making security and reliability essential. To this end, numerous approaches have been introduced to provide a secure and reliable path of communication. Recent research, such as [1], has demonstrated that RL can be used to increase the security and robustness of links in MANETs. However, as shown in [2], these RL approaches are not perfect, and interference from outside can drastically decrease their performance. Adversarial attacks are a potential source of interference, with the goal of reducing an RL agent's performance by perturbing its observation and causing the agent to take non-optimal actions.

Although [2] has demonstrated that adversarial attacks can reduce the performance of RL-based routing algorithms, several important questions remain unanswered. In particular, it is unclear whether the attacks succeeded because of a fundamental weakness in RL or if using a different approach would yield different results. This thesis aims to address this question by comparing multiple RL-based routing algorithms of varying complexity. Our goal is to determine whether additional security measures are necessary for providing a secure environment or if selecting the appropriate algorithm for a given task is sufficient for achieving a robust and secure routing solution.

The remainder of this thesis is structured as follows. Section 4 introduces and explains the essential topics such as reinforcement learning, mobile ad-hoc networks (MANETs), their parameters, and tactical networks. In Section 5, we present the different algorithms evaluated in this thesis and the approaches used to test their robustness. Additionally, we describe the environment used to train and evaluate the algorithms. Section 6 provides an in-depth analysis of the algorithms performance, and we establish a baseline for performance comparison to explain our findings regarding adversarial attacks. Finally, in Section 7, we summarize our results, discuss ways to extend the research, and pose questions that require further investigation.

4 Background

In the following a summary of the most important terms regarding this thesis is provided. This includes the most important elements of Reinforcement Learning mainly taken from [3], an explanation of mobile ad-hoc networks and their rising significance and at last, the particularities and expectations of tactical networks.

4.1 Reinforcement Learning

4.1.1 Origin

As the name suggests, Reinforcement Learning has its origin in the theories of learning[3]. Its goal is for an agent, that can interact with an environment, to learn what action in a given situation leads to a maximal numerical reward. The typical interaction between an agent and the environment is shown in Figure 1. Given a current state s the agent performs an action a in the environment. As a return the agent receives a reward r and a new state s'. The learning is achieved by trying different actions in a given situation and receiving different rewards depending on the action taken, thus leading to the agent learning what action yields the highest reward. This method is called learning by trial-and-error. The second major characteristic of Reinforcement Learning is the future reward or delayed reward. This characteristic describes the need for the learning agent to not only look at the best possible action in a given situation, but also to anticipate what taking a certain action means for future rewards. One important distinction to make when working with Reinforcement Learning is that RL is not supervised learning, which is another very prominent approach in machine learning as a whole. While supervised learning uses labeled data to train an agent RL does not. RL forces the agent to learn from its own experience and maximising the reward but not to learn a hidden pattern.

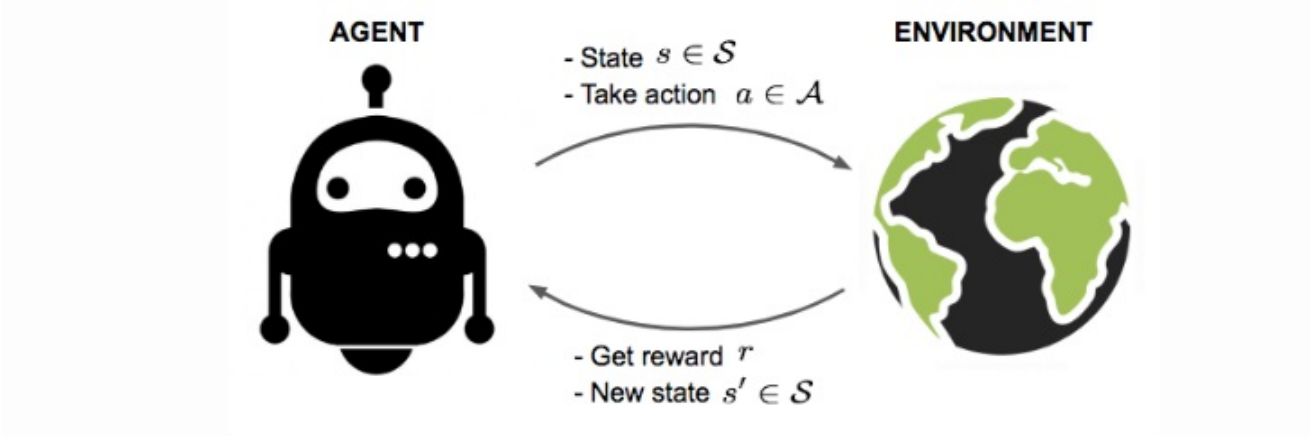


Figure 1. General relationship between an RL agent and the environment

4.1.2 Exploration and Exploitation

Like already mentioned the agent learns via trial-and-error. But what happens when the agent has found an action with a decent reward and assumes that this action is the best for a given situation and moves on without testing the remaining actions for their reward and this way maybe missing out on even better solutions? This problem is mostly referred to as Exploration vs Exploitation and describes the dilemma of having the agent use its already existing knowledge to achieve a reward but simultaneously trying out new things to increase the reward in the future. To balance this is a very important task while working with RL and later on we introduce ways to manage this.

4.1.3 Examples

As addressed above already, RL aims to train an agent to achieve a goal in an uncertain environment. This goal could be for a new born gazelle to learn how to run as soon as possible or even an every day task like preparing breakfast can be broken down into a lot of different steps or actions, like grabbing a spoon or pouring milk in a bowl, with each action leading closer to the overall goal: preparing breakfast[3].

It's easy to see that RL is applied and can be used in a lot of different settings. To emphasize this a small selection of real life applications of RL are provided below:

- Healthcare: To bypass the long procedure of identifying the necessary treatment for a patient, RL has been used to speed up that process[4].
- Computer vision: RL can be used to identify the most specific location of a target in a picture[5].
- Games: RL agents have been able to learn and play at the top level in one of the most complex board games ever created, namely GO[6].

This small selection of examples offers a great variety of possible applications and problems that can be solved via RL and shows the power and importance of it. But to actually apply RL to these problems a lot of prerequisites have to be fulfilled.

4.1.4 Elements

The above Sections already introduced multiple elements necessary to use RL when tackling a problem. One needs an agent to train, an environment for the agent to work in with an objective to achieve and a reward telling the agent whether his current course of action is leading him towards his goal or not. Additionally there are three more elements of importance for RL mentioned in[3], namely the policy, the value function and the model.

The Policy of an agent returns an action for the agent to take, given the current situation, meaning the state the agent is in. This state can consist of multiple different variables depending on the environment the agent is in. It can be his current location or time spend doing something. Depending on the complexity of the environment and the task this policy can either be a simple table, but also very complex forms taking extensive computation to receive an action.

As already mentioned the reward is the feedback used to tell the agent whether his current action is leading towards the goal or not. The agent receives a reward after every action and its sole purpose is to maximise the overall reward received. This reward is then used to update the policy, thus maybe leading to the agent choosing a different action with a higher reward the next time he encounters this state.

The next element important for RL is the value function. In the beginning of this Section the concept of a delayed reward has already been introduced. The task of the value function is to tell what actions are good in the long run without focusing solely on the current situation and the single reward received. A small real life example to further the understanding would be going to work on a rainy day. While the action to go out in the rain and get wet may result in a lower

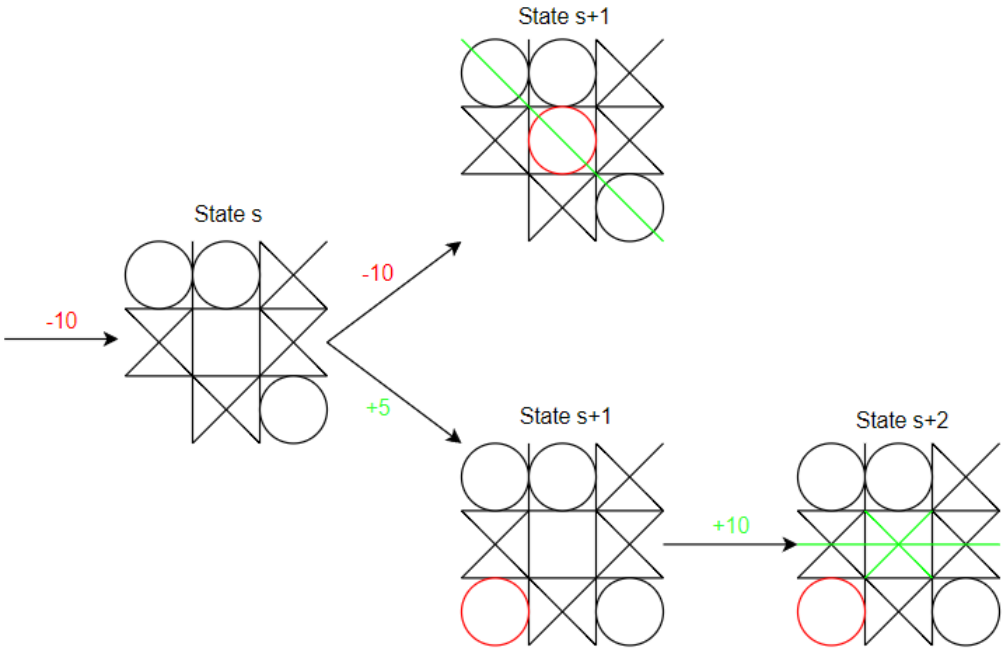


Figure 2. A game of Tic-Tac-Toe from the point of view of X. The value of state s is -10 because circle can win in the next action. The agent learning Tic-Tac-Toe would try to avoid reaching state s if it plays as X but would try to reach state s if playing as circle.

reward then staying at home the overall result of going to work, keeping your job and earning money would lead to a higher reward then staying at home. This would be the purpose of the value function, to take these future states into account and create a numerical representation. Another simple example of Tic-Tac-Toe is provided in Figure 2. The value of a state is the direct product of it's future states.

The last element of RL is the model. This element is not present in all applications of RL and it mimics the behavior of the environment[3]. Models allow the prediction of results of the next state and next reward, thus leading to better planning by knowing the results of actions not taken before. Methods using the model to learn are called model-based methods. Else they are called model-free.

4.1.5 History

Reinforcement learning has a long and rich history. It consists of three different threads that existed independent of each other before coming together in the late 1980s and thus creating our modern understanding of reinforcement learning[3], with the first thread being the already introduced concept of trial and error learning. The second thread didn't have anything to do with learning in the first place and was mostly concerned about optimal control and its solution using value functions. Its goal was to design a controller that can either minimize or maximize a measure of a dynamical system's behavior[3]. But the third thread, namely temporal-difference-learning, which is thoroughly explained in 4.1.9, tied these threads together and created RL as it's known now.

4.1.6 Markov Decision Processes

Markov Decision Processes(MDPs) are a mathematically idealized form of the RL problem for which precise theoretical statements can be made[3]. In an MDP the learner is called the agent and everything else beside the agent is called the environment. The agent can interact with the environment resulting in rewards. Figure 3 depicts the procedure in an MDP. The agent performs an action a_t in the environment according to the current state s_t. The environment returns the reward that the action produced and the new state s_{t+1}, which is then fed to the agent again. Each of these sequences take one time step, with the goal of the agent still being to maximize the overall reward. While this thesis is only dealing with finite MDPs, meaning that after an finite number of time steps an episode terminates, this doesn't have to be the case in every application and has to be considered thoroughly. One episode describes such a sequence or trajectory and looks like this:

$$S_0, A_0, R_0, S_1, A_1, R_2, S_2, ...S_n, A_n, R_n, S_{n+1} \tag{1}$$

For each sequence there is a probability function defining the dynamics of the MDP:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, At - 1 = a\} \tag{2}$$

This function describes the probability at time step t of ending up in state s' with the reward r if the agent is in state s and takes action a at time step t-1. This function shows the flexibility of a

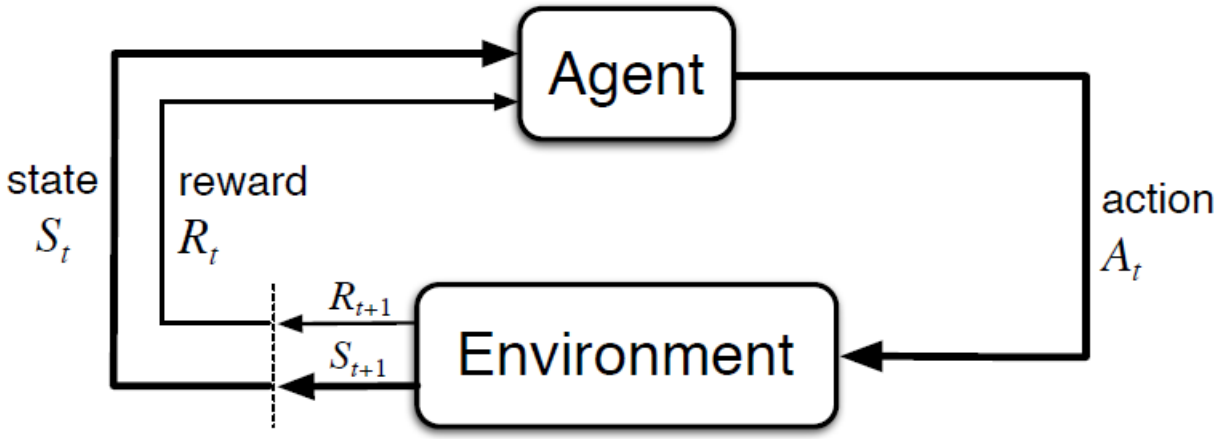


Figure 3. One sequence at time step t in a MDP from [3]

MDP. It is not forced to a strict environment or task but can be modified to one's heart's content.

It is important to realise the fine distinction between the agent and the environment. In [3] the environment is defined as anything that cannot be changed arbitrarily by the agent. So instead [3] introduces the concept of the agent-environment boundary which acts as a line of separation between the two. This can be located in different places depending on the purpose.

As already mentioned the agent receives a reward in every time step. This reward depends on the action the agent took in state s and the reward is directly linked to the overall goal the agent is supposed to achieve, which means when implementing such a reward its important that the reward depicts the goal in every time step. For example. when the goal is for an agent to do something as fast as possible, like running from a start point to an end point, the reward could always

be -1 when doing an action and 100 when reaching the goal. This would lead to the agent learning the fastest way from the start to the end within as few time steps as possible. The concept of an episode has already been established, however now this definition is going to be expanded by adding a distinction between agent-environment actions that break into clear sequences called episodic tasks, and the one's where this break isn't clear called continuing tasks. For the above example with the start and end point a clear break can be made. A terminal state is reached if either the agent has reached the goal or after T time steps. In regards to our overall goal to maximize the cumulative reward in the long run the return would look like

$$G_t = r_{t+1} + r_{t+2} + ... + r_T \tag{3}$$

But this doesn't work for continuing tasks because T may be ∞. To still be able to compute the return G_t a variable called the discount factor γ is introduced in [3]. Its purpose is to reduce the rewards that are far in the future. This leads to

$$G_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4}$$

with $0 < \gamma < 1$. Depending on the chosen value for γ the future rewards have a stronger or weaker influence on the overall return.

With the return, the previously mentioned value function can now be defined as

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} = \gamma^k r_{t+k+1}|S_t = s], \text{for all } s \in S \tag{5}$$

,with π being the policy dictating the agents actions. This value function is also called the state-value function for policy π because it only refers to the state s without taking the action the agent takes into account. With regards to the action taken in state s the action-value function for policy π can now be defined as

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} = \gamma^k r_{t+k+1}|S_t = s, A_t = a] \tag{6}$$

. The last important concept to introduce is the one of optimal policies and value functions. For the previous example of some race from a starting point to the finish line, the optimal policy would be the one policy which yields the highest return, thus being the policy that selects the actions which lead the agent the fastest to the goal. Because the state-value function and action-value function both depend on the policy, this simply leads to

$$v_*(s) = max_\pi v_\pi(s) \tag{7}$$

and

$$q_*(s,a) = max_\pi q_\pi(s,a) \tag{8}$$

with $*$ denoting the optimal function.

These definitions leave room for one question at hand. How does one find and compute the optimal policy for a given goal? One way of achieving this is via dynamic programming and Bellman Optimality Equations.

4.1.7 Dynamic Programming

According to [3] the key idea of Dynamic Programming(DP) is the usage of value functions to search for good policies. This is due to the fact that once an optimal value function has been found that satisfies the Bellman optimality equation

$$v_*(s) = max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_{(*)}(s')] \tag{9}$$

or

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma max_{a'} q_{(*)}(s',a')] \tag{10}$$

it can be used to compute the optimal policy. These Bellman equations can then be used as the update rules for DP algorithms.

Finding the optimal policy consists of two steps. Policy evaluation followed by policy improvement. In the first step the state-value is updated. To do this, the following Bellman equation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \tag{11}$$

can be used to update the value function. The update rule looks like this

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_{k-1}(s')] \tag{12}$$

for every k > 1 and with v_0 being initialised randomly. This algorithm is called iterative policy evaluation and updates the state value of each state by using its old value and the expected immediate reward and all other one-step transitions possible under the policy[3].

After evaluating the policy the second step, policy improvement, is applied. Here a decisions has to be made, whether to stick to the policy by choosing the actions a according to $\pi(s)$ or if another action is chosen according to $\pi'(s)$. One way to decide this is by comparing the state-value function $v_\pi(s)$ to the action-value function $q_\pi(s,\pi'(s))$. It is important to note that π and π' behave exactly the same except for state s. If

$$q_\pi(s,\pi'(s)) > v_\pi(s) \tag{13}$$

the policy is simply adjusted to π' in state s. The question remains how to choose π'. One way is to choose greedy meaning

$$\pi'(s) = argmax_a q_\pi(s,a) \tag{14}$$

These two steps combined are called policy iteration and it describes the process of increasing the quality of our policy in every step like:

$$\pi_0 \Rightarrow^E v_{\pi_0} \Rightarrow^t \pi_1 ... \Rightarrow^t \pi_* \Rightarrow^E v_* \tag{15}$$

Because of the way the evaluation and improvement work its guaranteed that there is a strict improvement over the previous policy in every step. But this method is limited by the interactions between the two steps and the necessity for one step to finish for the other to begin. Generalized policy iteration(GPI) is the general idea of letting the two steps interact with each other and is used in almost all RL methods[3]. The general idea is that once the two steps stop producing change a optimal policy and value function has been achieved.

Overall DP can be used to solve nowadays problems and solve MDPs with millions of states. However one big downside of DP is that one needs complete knowledge over the environment to utilize it. In the following, two methods requiring only experience, meaning sequences of states, actions and rewards from actual or simulated interaction with an environment[3], are introduced.

4.1.8 Monte Carlo Methods

Monte Carlo(MC) methods learn by averaging sample returns over episodes. The value estimation and policy are only changed once an episode is over. The approach is very similar to the approach of DP. The only major difference is that MC methods aren't computing value functions from knowledge but from sample returns. Just like in DP this process can be divided in three steps. First the computation of a value-function v_π and action-value function q_π, then policy improvement and finally combining those by GPI[3].

While in DP it was sufficient to estimate the state values this is not the case for MC due to no knowledge over the model. Instead the action values have to be estimated. There are two approaches for this. Either the every-visit MC method or the first-visit MC method. A visit of a state s and action a occurs in an episode whenever state s is visited and action a is performed. Like the name suggests the every-visit approach uses the average return for all visits as the estimation value while the first-visit only takes the first visit into account when creating the average return for an episode. But these approaches are missing one key point mentioned earlier, exploration. It is important that all state-action pairs are encountered to make sure that an optimal policy can be found. To ensure that a stochastic policy with a nonzero probability of selecting all actions in each state has to be used. The two most important variants of this approach are On and Off-policy methods. While On-policy methods only use one policy to evaluate and improve off-policy methods use two different policies. One policy is used to generate data while a second policy is improved. The policy that is being improved or learned is called the target policy. The policy used to generate data is called the behavior policy.

4.1.9 Temporal-Difference Learning

This leads us to our final and most important idea for this thesis, Temporal-Difference(TD) learning. The key concept for TD is a combination of the two prior ideas. For once TD methods learn directly from experience just like MC methods and, just like in DP, there is no need to wait for an episode to finish to run the update step. For the most simple TD method the update

$$V(S_t) \Leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{16}$$

occurs in every transition from state S_t to S_{t+1}. The parameter α is a constant that scales the change made for $V(S_t)$. If α is close to 0 the update performed is minimal. If it's chosen bigger or close to 1 the size of the update depends a lot more on the reward R_{t+1} achieved and the difference between the state-value of S_{t+1} and S_t. This part is also called the TD error. Now one question may arise in what method is the best? It is obvious that both TD and MC are more well rounded then DP because they do not require any knowledge over the environment to learn. But between TD and MC no clear answer has yet been found. One big advantage of TD however is that in case episodes take a lot of time to finish MC methods also take a lot of time to apply an update while TD performs them immediately[3].

But how do TD methods actually solve the control problem? To answer this question, in the following two algorithms are thoroughly examined and as already stated it is important to find a compromise between exploration and exploitation when applying a learning algorithm. This is once again achieved by either of the two previously introduced ideas: On and Off-policy. The

first algorithms is an example for an an On-policy approach, because these are often more simple in their way of learning and can overall be seen as a mere special case of off-policy learning in which the target and the behavior policy are the same. One example of an on-policy method is the SARSA approach.

The name SARSA originates from the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q_(S_t, A_t)] \tag{17}$$

that is very similar to the update rule introduced in the begging of this Section except it now uses the action-value function instead of the state-value function. The reasoning behind that is the same as in MC methods. One update step consists of the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ and thus resulting in the name SARSA. This is also a good moment to introduce the ϵ-greedy policy which is often used in SARSA. The main idea behind the ϵ-greedy policy to ensure exploration by defining an ϵ that is both < 1 and > 0. When choosing an action to perform according to the ϵ-greedy policy, one picks action a according to

$$a = \begin{cases} max_a Q(s,a) & \text{with probability 1-}\epsilon \\ rand(a) & \text{with probability } \epsilon \end{cases} \tag{18}$$

, meaning that with a prbability of ϵ, a random action a is performed, thus ensuring exploration. As can be seen below in Algorithm 1 SARSA utilizes the ϵ-greedy policy twice. The algorithm

Algorithm 1 SARSA(on-policy TD control) from [3]

Algorithm parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialize Q(s,a), for all s $\in S^+$, $a \in \mathbf{A}(s)$, arbitrarily except that Q(terminal,·)=0
for each episode **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g. ϵ-greedy)
    for each step of episode **do**
        Take action A, observe R, S'
        Chose A' from S' using policy derived from Q(e.g. ϵ-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S', A') - Q_(S, A)]$
        $S \leftarrow S', A \leftarrow A'$;
    end for
end for

begins by initializing all action-values for all states s $\in S^+$ and actions a $\in A(s)$ at random. The only exception are the action-values of terminal states. After that the algorithms iterates over all episodes by initializing S and choosing an action a according to the ϵ-greedy policy. Afterwards the actual training and updating of the action-values starts. For every step in the episode that is currently being used, the prior chosen action a is performed. The environment then returns a reward R and a new state S' which is observed by the algorithm. Now all that is left to do to utilize the update rule for SARSA 16, is to select the next action for the state S'. This is once again achieved via the ϵ-greedy policy. Then the action-value for state S and action A is updated. At last the old state S and action A are updated to S' and A' respectively. Because the same policy is used for the entire algorithm, SARSA is considered on-policy. It is also important to consider

that while this thesis has its main focus on the ϵ-greedy strategy, this doesn't have to be the case when using SARSA.

Now an example for an off-policy algorithm is going to be investigated. Q-Learning is one of the most prominent RL methods and this thesis is later on presenting several different algorithms based on Q-Learning and their implementations. Q-Learning was developed in 1989[7] by Watkins and is one of the big early breakthrough in RL[3]. The update rule for Q-learning is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \cdot max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{19}$$

and the big difference to the SARSA update rule is in the TD error. The TD error no longer uses action A_{t+1} but instead choose the action a greedy in state S_{t+1}. One big advantage of Q-learning is that the action-value function Q directly approximates q_*. The overall procedure of Q-Learning can be seen in Algorithm 2. Just like in SARSA, the algorithms begins by initializing the action-value function for all states s and actions a arbitrarily with an exception, once again, for terminal states. Afterwards Q-Learning iterates over all episodes and initialize a starting state s. But since there is no longer the need for the action to be performed in the next step, because it is now chosen greedy, only action a for state s has to be selected according to the ϵ-greedy policy, After performing the chosen action the update rule is immediately applied to update the corresponding action-value.

Algorithm 2 Q-Learning(off-policy TD control) from [3]

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize Q(s,a), for all s $\in S^+$, $a \in \mathbf{A}(s)$, arbitrarily except that Q(terminal,·)=0
for each episode **do**
    Initialize S
    for each step of episode **do**
        Chose A from S using policy derived from Q(e.g., ϵ-greedy)
        Take action A, observe R,S'
        $Q(S, A) \Leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', A) - Q_(S, A)]$
        $S \Leftarrow S'$
    end for
    until S is terminal
end for

To get a better understanding of the different approaches in SARSA and Q-Learning and where their strength and weaknesses are, [3] provides a simple problem and two agents that are trained with either SARSA or Q-Learning and their results. The environment consists of a gridworld like Figure 4. The goal for the agents is to go from the starting point S to the goal G. For every step they take they get a reward of -1. If they walk into the cliff they get a reward of -100 and are put back to the starting position. Now there are obviously several different paths to take, but the optimal path that used as little steps as possible leads directly past the cliff. Because the actions are chosen by an ϵ-greedy policy, there is a chance to fall into the cliff if the agent is in a field directly next to it. A second path would be the path that doesn't walk along the cliff. The agent may take more steps to reach the goal but has overall the lower probability to fall into the cliff. With an ϵ=0.1 an agent taught via Q-learning learns the optimal path and an agent using SARSA learns the safer

path. But because of that, and Q-learning using an ϵ-greedy policy the overall sum of rewards shown in Figure 5 are less for Q-learning because of the risk of falling into the cliff[3]. However in case of ϵ being gradually reduced both methods would converge towards the optimal policy. This method of decreasing the epsilon over time is called epsilon decay. It's a very useful extension of the regular ϵ-greedy policy because it keeps the exploration aspect but also introduces a way to avoid the risk of an already trained agent to perform wrong actions at random. However, having the ϵ decay over time can become problematic if the environment changes and a new path has to be found. This problem is going to be investigated as part of this thesis as well.

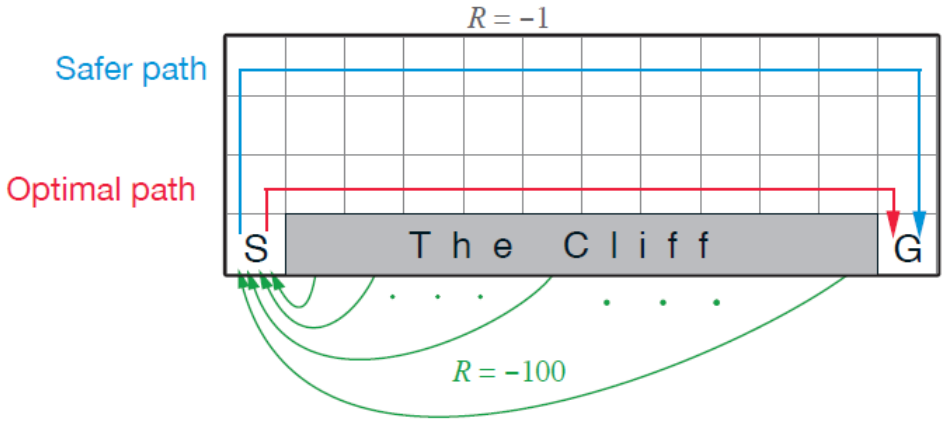


Figure 4. A gridworld with two paths from the start to the goal from [3]

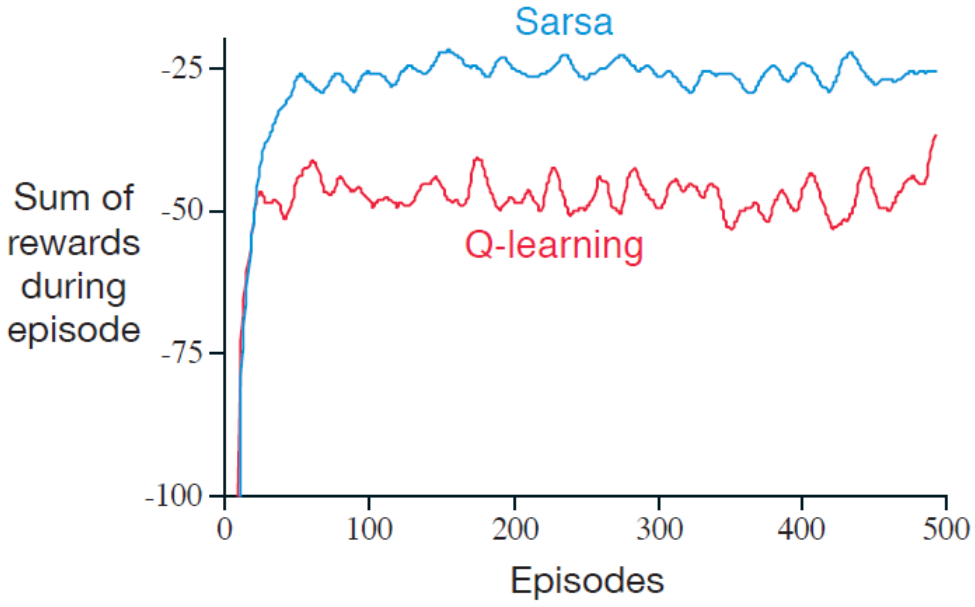


Figure 5. Rewards achieved per episode for 500 episodes from [3]

This concludes the Section about RL and the different approaches possible. Before taking a look at the different algorithms utilizing Q-Learning and the environment used to train on, one more important subject of this thesis has to be introduced first to understand these fully.

4.2 Mobile Ad-Hoc Network

Wireless ad hoc networks or mobile ad-hoc networks(MANETs) are a specific form of wireless communication networks that don't rely on an infrastructure with a base station but instead consists of an autonomous transitory association of mobile nodes that communicate with each other over wireless links[8]. If a package is send from one of these nodes to another it travels across the links. Additionally if the sending node and the receiving node aren't directly linked other nodes act as a router that relay the packet to its destination. As seen in Figure 6 nodes can be all kind of different devices mostly powered by battery. MANETs find use in all different kind of fields such as Emergency services to organize search and rescue operations, at home to connect a handy to a computer or even in tactical networks to ensure military communication in operations. The last part is also the main focus of this thesis. Each node in the network can send a packet to another node in the network and the time it takes for a package to arrive at it's destination depends on different parameters such as packet loss, data rate and queue length. Because of the increasing importance of MANETs in every day life the stability and security of these networks have to be a top priority while still maintaining a fast overall performance. To achieve this, different routing protocols have been published in regards to these problems. Later on an in depth look at different routing algorithms and their approach to solving these problems is provided.

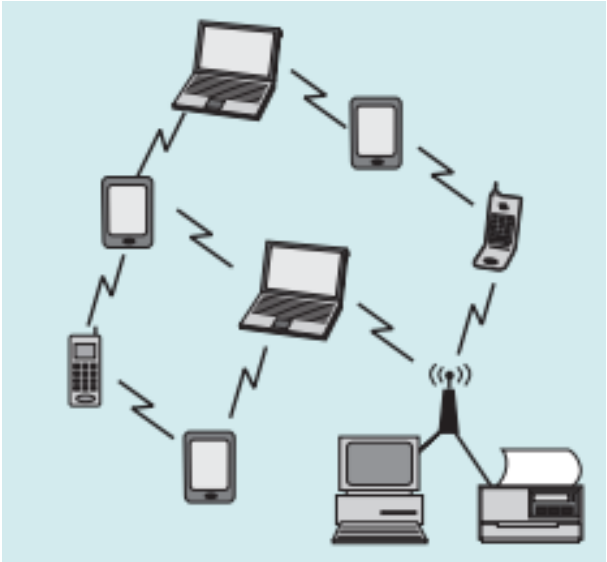


Figure 6. An example for an mobile ad-hoc network from [8] with different devices as nodes.

4.3 Tactical Networks

Tactical Networks are a special case of communication networks and refer to networks used for military communication and operations. Because of this, the already high dependency on a safe and stable connection is even more urgent than in every day scenarios due to the possibility that the life of operators or the success of the mission depend on these networks to work even in the most extreme circumstances. Furthermore, they not only have to be robust towards natural disturbance but additionally towards influences by an attacker. There have been proposals to use RL

to ensure the robustness of such networks in [1], but also proof that at least some RL algorithms are vulnerable towards attacks in [2]. In the following Section the environment simulating a tactical network and four different routing algorithms with the goal to be able to send packages from a source to a destination node in this environment while still being able to handle any kind of disturbance, are introduced. Additionally the different approaches to test their robustness are explained thoroughly.

5 Methodology

5.1 Environment

The environment is based on the environment used in [2]. A MANET is being modeled as a edge-weighted graph G:=(V,E) that consists of a set of nodes V and network links E. The weights on the links are presented as a triple that consists of the parameters queue length q_e, packet loss l_e and the data range d_e with $q_e \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$,$l_e \in \{0.0, 0.05, 0.1, ..., 1\}$ and $0.0 \leq d_e \leq 9.6$. The network conditions are simulated using data from experiments from [2]. With a way to initialize the weights on the edges the agent's goal is to be deployed at a source node S and to find a way ϕ through the network to end up in the destination node T. To make sure that every node is connected to all other nodes in some way a Barabási-Albert(BA) model is used to create the network. This allows the modeling of the observation space as a combination of next neighbors $V_i \subseteq V$ and the network parameters. Together they form the input for the RL agent as following: For each edge e =(v_i, v_j) from the current node v_i to a node v_j the agent receives the triple q_e, l_e, d_e for this specific edge.

The agent can either take a legal or an illegal action in each node v_i with an illegal action being the try to move to a node v_j that is not connected to v_i or a node that is not a forward neighbor. A node is considered a forward neighbor if the distance from it to the destination is shorter then the distance from the current node. In Figure 7. an example graph is provided, where the green node 18 is the current position of the agent and the orange node 3 is the destination. An example for a legal action would be node 13. To prevent the agent from learning illegal actions a strong negative reward is returned if one is performed. This is to make sure the agent doesn't try to hop to nodes that he can't get to. In case of a legal action the reward depends on the three network parameters as seen in Equation 20 and if the destination is reached a reward of 100 is achieved. The three weights w_1, w_2 and w_3 can be tuned according to the importance of the different parameters. For this thesis the weights are initialized with 1, but can be modified if deemed necessary.

$$r(e) = \begin{cases} 100, \text{ reached target node} \\ \frac{1}{w_1} \cdot q_e + \frac{1}{w_2} \cdot (1 - (1 - l_{e_{v_i}})) + \frac{1}{w_3} \cdot \mathbb{E}[d_e] \\ -5, \text{ selecting an illegal action} \end{cases} \tag{20}$$

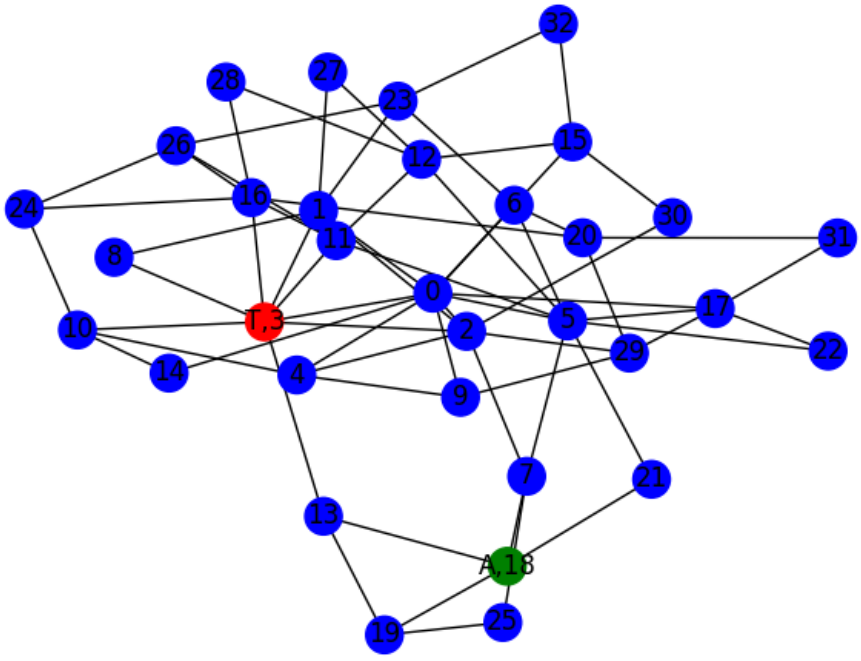


Figure 7. One example of a graph used to train the agents. The green node is the current position of the agent and the red the target node.

5.2 Algorithms

In this Section the different algorithms used to train an agent in the environment with the main focus being off-policy algorithms are introduced. A total of four different routing algorithms, that are all related to Q-Learning, are investigated. First QRouting[9] which also acts as the founder of RL based routing approaches using Q-Learning[10]. The second algorithm is CQRouting from [11] which introduced confidence values. The last two algorithms are CQplus, which was published in [12] and DeepCQplus published in [13]. Both are an extension of CQRouting and promise more stability and robustness.

5.2.1 QRouting

QRouting was first introduced 1993 by [9]. It's the first RL based algorithm that uses Q-learning[10]. Just like Q-learning, Qrouting uses a Q-function and Q-values to determine which actions to take. The state is given by the current node of the agent i and the node the agent wants to deliver a package to called d. The Q-value is denoted as $Q_i(d, j)$ with j being the next node the agent sends the package to. After the node j has been selected and the action performed to get there, node j returns an estimate for the remaining time to reach the destination

$$\theta_j(d) = \min_{k \in N_g(j)} Q_j(d, k) \tag{21}$$

with $N_g(j)$ being a list of all neighbor nodes of j. The update rule for QRouting is

$$Q_i(d, j) = (1 - \alpha) \cdot Q_i(d, j) + \alpha \cdot (q t_i + T x T_{i,j} + \gamma \theta_j(d)) \tag{22}$$

with α being the learning rate, qt_i the queue length and $TxT_{i,j}$ being the link cost. For our case this would be the data rate and the packet loss from node i to node j. Additionally a discount factor γ can be used to increase or decrease the impact of the estimate. To ensure exploration the previously introduced ϵ-greedy policy is utilized as well when choosing an action. The ϵ is initialized with 0.2 and is slowly decaying over the training period. It's important to note that Equation 21 uses the minimum instead of the maximum Q-value unlike regular Q-learning. This is due to the fact that the goal is to find the fastest path through the network and higher link costs lead to higher Q-values. Algorithm 3 provides a detailed explanation of QRouting. It's easy to see that the Q-Values can be stored as a 3 dimensional list. The Q-Values are calculated according to Equation 21 and 22. The updates are performed as long as the termination condition isn't hit. The condition threshold can be something like a reward threshold over the last n episodes that has to be reached.

Algorithm 3 QRouting from [10]

$Q_i(\star,\star)$ is the Q-value matrix of node i.
l·Q_i matrix may be randomly initialized.
while Termination condition not hit **do**
    if Packet to send is ready **then**:
        Select next hop j according to ϵ-greedy policy
        Send packet to node j
        Node i immediately gets back j's estimate for the time remaining in the trip to destination d calculated by Equation 21
        Node i updates its delivery delay estimate using Equation 22
    end if
    Repeat until Termination condition
end while

5.2.2 CQRouting

Confidence-based QRouting or CQRouting is an extension of QRouting. The main idea being the addition of a second parameter for the Q-Values called confidence value(C-Value)[11]. Every Q-Value is associated with a C-Value between 0 and 1 with 1 representing maximum confidence in the corresponding Q-Value. Just like in QRouting, the Q-Values are randomly initialized and the C-Values are 0 for all C-Values except C(y,y). In this case the C-Values are always 1. The update rule for CQRouting is given by

$$\Delta Q_x(y,d) = \eta(C_{old}, C_{est})(Q_y(\hat{z},d) + q_y + \delta - Q_z(y,d)) \tag{23}$$

with C_{est} being the C-Value to the Q-Value from Equation 21. The only new addition is the $\eta(C_{old}, C_{est})$ part. It fulfills the same purpose as the learning rate α in QRouting and it is chosen based on:

- Confidence in the old Q-Value is low or
- Confidence in the estimated Q-Value is high

One way to achieve that proposed in [11] is

$$\eta(C_{old}, C_{est}) = max(C_{est}, 1 - C_{old}) \tag{24}$$

. All C-Values get updated in every step except in the case mentioned above. There are 2 cases regarding the intensity of the upgrade namely:

- If the Q-Value of the corresponding C-Value has not been updated in the last time step the C-Values decay by some constant $\gamma \in (0, 1)$
- If the Q-Value is updated then the corresponding C-Value is updated based on the C-Values corresponding to the Q-Values used in the Q-Value update

The update rule for the C-Values is

$$C_{upd} = \begin{cases} \lambda C_{old} & \text{First case} \\ C_{old} + \eta(C_{old}, C_{est})(C_{est} - C_{old}) & \text{Second case} \end{cases} \tag{25}$$

and in the second case the update uses the old C-Value and adds Equation 24 multiplied by the difference between the estimated C-Value and the old C-Value. The η used in Equation 25 is the same as in the update rule for the Q-Values and depending on its size the estimate plays a larger or smaller part in the overall update. Just like QRouting, CQRouting is using an ϵ-greedy policy to ensure exploration. Due to the addition of the C-Values it is to be expected that a stable route is found faster than regular QRouting, while also being more certain that the found path is stable and thus making CQRouting a more stable algorithm than his predecessor.

5.2.3 CQplus

CQplus takes the concept of CQRouting and adds an additional approach to tackle the problem of frequent topology changes in tactical networks which is not regarded in classical CQRouting [12]. The solution for this problem provided by CQplus is a combination of RL as in CQRouting and an Adaptive Routing framework which determines whether the connection to the destination node is stable. If the connection is stable one can simply proceed as in CQRouting, meaning the shortest path is selected via Q-Values. This process is referred to as unicast and it describes the straight delivery of packages in a network from one node to another. In the case of an unstable network the packages are instead broadcast. Broadcasting describes the flooding of the network with the package that is to be delivered to make sure it arrives at it's destination. One can easily see that this idea is an improvement to robustness due to visiting more nodes and thus more exploration of the network. This approach is also referred to as Smart Robust Routing(SRR) in literature. Additionally the estimate from Equation 21 is now called ACK, due to this being the term used in more recent publications. A normal interaction between two nodes is displayed in Figure 8. First a package is sent from a node to a neighbor node. Once the package is received the neighbor node returns an ACK package which contains information. To be more precise, these ACK packages inform about the amount of hops it still takes to reach the destination node from the respective node and the delay. It's also important to note that the update rule for the C-Values

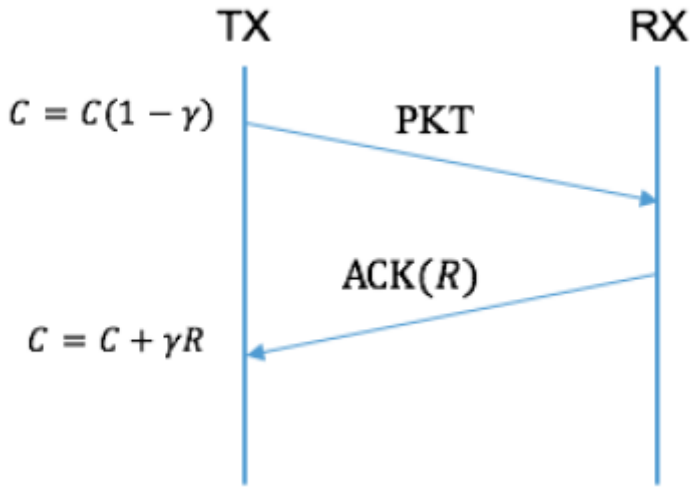


Figure 8. Interaction between two nodes. A package in send to a node and as a return it receives ACK[12]

$$C_i^{n+1}(d,j) = \begin{cases} (1-\gamma)C_i^n(d,j) & \text{if transmission fails} \\ (1-\gamma)C_i^n(d,j) + \gamma C_{ACK}^n & \text{else} \end{cases} \tag{26}$$

has been slightly adjusted in [12] in contrast to [11]. To determine whether to unicast or broadcast, first the next node has to be determined using

$$j^* = \underset{j}{\operatorname{argmin}}\, Q_i(d,j)(1 - C_i(d,j)) \tag{27}$$

. To decide whether to act according to Equation 27 and go to node j^* or instead broadcast a probabilistic choice is made by

$$p_B = \epsilon + (1-\epsilon)(1 - C_i(d,j^*)) \tag{28}$$

with ϵ acting almost the same as in the previously introduced ϵ-greedy policy by making sure that the probability of broadcast being non zero even for a C-Value of 1. This allows for additional exploration. Due to the way broadcast works, packages can theoretically loop between nodes. To prevent this CQplus uses a duplicate packet detection(DPD) to prevent this from happening. In case of this happening the looping package is dropped as shown in Algorithm 4. However, due to the way the environment used in this thesis operates, package duplication is not possible and is therefore not regarded in the results Section. Overall CQplus is a clear improvement over regular CQRouting as it provides a solution for a very common problem in tactical networks. This should lead to a more robust agent, whether the agent has to deal with normal causes of unstable tactical networks or influence from third parties. It is to be expected that, compared to the prior introduced algorithms, the ability to broadcast should lead to a faster learning of the network by CQplus. Due to the choice to tie the decision whether to broadcast or unicast to the respective C-Value of the current state it is to be expected that there is a lot of traffic in the beginning of the training.

Algorithm 4 CQplus from [12]

```
Receive incoming packet at node i
if Packet is ACK then
    Collect C_ACK, Q_ACK, d, j
    Update Q-Value
    Update C-Values
else
    if Packet traversed a loop then
        Drop Packet;
    end if
    Select j*
    Send ACK
    if Copy already forwarded then
        Drop package
    end if
    if node i is destination then
        Send Upstream
    end if
    p_B ⇒ ϵ + (1 − ϵ)C_i(d, j*)
    if Broadcast(p_B then
        nexthop=every node in neighborhood
        for j ∈ Neighborhood do
            update C_i(d, j)
        end for
    else
        nexthop=node according to action a
        update C-Value forC_i(d, j)
    end if
    Forward packet to nexthop
end if
```

5.2.4 DeepCQplus

At last [13] introduced yet another extension of QRouting and to be more specific of CQplus. The authors propose a different approach to the switching decision used in CQplus by utilizing multi-agent deep reinforcement learning(MADRL). According to [13] this leads to a robust agent that can even be deployed in scenarios with different sizes,traffic profiles and mobility patterns they haven't encountered during their training unlike their predecessors. The Deep Neural Network's(DNN) input features for a node i consist of two lists $q_t(i)$ and $c_t(i)$ with

$$q_t(i) = [q_t(i,i_1), q_t(i,i_2), ..., q_t(i,i_K)] \tag{29}$$

and respectively

$$c_t(i) = [c_t(i,i_1), c_t(i,i_2), ..., c_t(i,i_K)] \tag{30}$$

. These lists are ordered with respect to Equation 21 and each consist of K elements with the first element being the Q- and C-Value of the node i_1 that minimizes Equation 21. Additionally

the input feature of the DNN include $\Delta c_t(i) = c_t(i) - c_{t-1}(i)$ and $\Delta q_t(i) = q_t(i) - q_{t-1}(i)$ with both being the difference in Q- and C-Values from time step t-1 to t. The last two element of the input are the previous action $a_{t-1}(j)$ performed in some node j and $p_{t-1}(i)$, indicating whether the received package is the result of unicast or broadcast. All these element together form the input

$$o_t(i) = [c_t(i), h_t(i)\Delta c_t(i)\Delta h_t(i), a_{t-1}(j), p_{t-1}(i)] \tag{31}$$

The reward function presented in [13] is

$$r_t = w_1 1_D - w_2 1_Z - w_3 \frac{N_{ack}}{N} \tag{32}$$

with 1_D being the reward for packet delivery. This value depends on the contribution of node i in reaching the destination. 1_Z is a penalty applied when a node i send a package but didn't receive an ACK package. The last value is the amount of ACK packages received after N packages have been send from node i. Once the destination is reached from a node, this node receives a C_{ACK} of 1 and a Q_{ACK} of 0 from the destination node. Algorithm 5 shows the process of DeepCQplus. As mentioned above the major difference between CQplus and DeepCQplus lies in the last if statement where the DNN policy is used to choose between broadcast and unicast. It is to be exptected that due to having an intelligent decision making that the swap between the two should lead to an overall robuster algorithm that produces less traffic in the network.

5.3 Attacks

As already mentioned the reward indicates the performance of an agent in the environment. A high reward is achieved if an optimal path through the network is chosen. The goal of an adversarial is to reduce the overall reward, thus making the agent choose a non optimal action. Now follows a closer look at three different type of attacks, the naive and the gradient attack taken from [14] and the adversarial policy attack from [15], their strategy to achieve this goal and also the introduction of two different types of attack timings. All three attacks are adversarial attacks, meaning they aim to perturb the observation of the victim agent to force him to make wrong decisions. Figure 9 provides an example from [16] that shows two different adversarial attacks attacking a game of pong. In the first case the action taken by the victim would be down to block the ball, but due to the perturbed observation it receives, the agent instead doesn't act at all leading to conceding points and hence to a lower reward. In the second case the perturbed observation shows the ball to be at another position, therefore leading the agent to take the wrong action by moving downwards instead of upwards. This small example highlights the strength of these adversarial attack and the harm they can cause. These adversarial samples that lead the victim to choose non-optimal actions are crafted by applying a perturbation δ to the original observation. The way this δ is crafted is the main difference between the three attacks. It is important to note that a simple perturbation of the list of neighbors, to force the agent to make wrong decisions, won't take place, but instead the focus is on the quality parameters introduced earlier by adding the perturbation δ to these.

Algorithm 5 DeepCQplus from [13]

1: Receive incoming packet at node i
2: **if** Packet is ACK **then**
3: Update Q-and C-Values
4: **else**
5: **if** Packet traversed a loop **then**
6: Drop Packet;
7: **end if**
8: Select j^*
9: Send ACK
10: **if** packet is already in queue **then**
11: Find best nexthop j^* from 27
12: Compute C_{ACK} and Q_{ACK}
13: Drop packet but return ACK
14: **end if**
15: **if** packet is not duplicate **then**
16: Add packet to queue
17: **else**
18: Do not add packet to queue
19: **end if**
20: **end if**
21: **if** ACK packet is not received **then**
22: Do not update Q- and C-Value
23: **end if**
24: **if** Queue is not empty **then**
25: Form the input to the DNN policy by using Equation 29 and 30
26: DeepCQplus routing: choose

$$\begin{cases} Broadcast & \text{with probability } \pi_\theta(a=1|o_t;\theta) \\ Unicast & \text{with probability } \pi_\theta(a=0|o_t;\theta) \end{cases}$$

27: **end if**

5.3.1 Naive Attack

The first attack used in this thesis is the naive attack or naive approach and, as the name suggests it is the most basic of the three. The δ is chosen by drawing n samples from a β-distribution resulting in n_i (with $1 \leq i \leq n$). As seen in Algorithm 6 the β distribution receives an α and a β as input. For this thesis both are always 1. After sampling the noise one can then calculate δ_i

$$\delta_i := \epsilon \cdot 2(n_i - 0.5) \tag{33}$$

and add it to the current state s. Now a_{adv} is calculated, as is shown in Algorithm 6 line 6. The target is then fed this new action to see whether the resulting Q-Value is worse then q^*, the action chosen by the target without the attack. Formula 34 depicts this progress. The reason for trying to increase the Q-Value instead of minimizing it like it's done in [17] and [14] is due to the fact

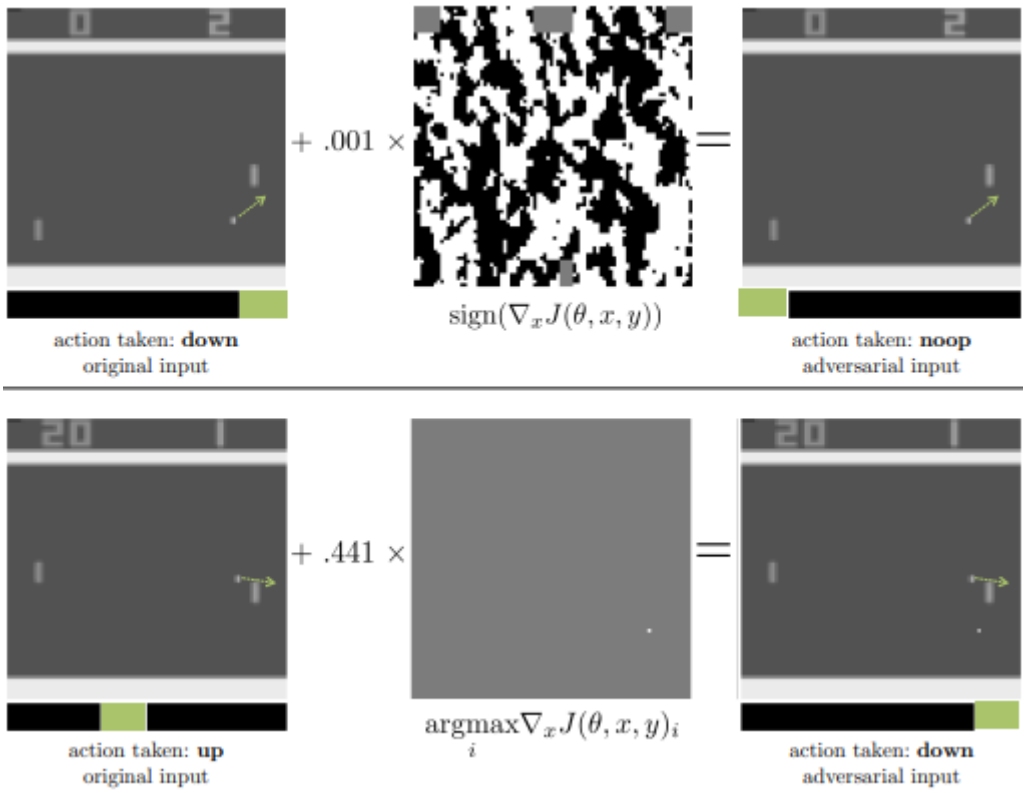


Figure 9. Two attacks performed on a victim. In both cases the perturbation leads the agent to make a wrong decision but depending on the perturbation and attack the agent chooses different actions. Taken from [16]

that the goal in QRouting is to minimize the Q-Values. Hence choosing a perturbation that leads to the biggest increase in the Q-Value reduces the performance the most. The n used to indicate the amount of times noise is sampled is always 10 for the experiments and the ϵ used to scale the noise is always 0.35.

$$\delta = argmax_{\delta_i, 1 \leq i \leq n} Q(s+\delta_i, \pi(s+\delta_i)) \tag{34}$$

5.3.2 Gradient Attack

The gradient attack can be seen as an extension of the naive attack and, as the name suggests, utilizes the gradient of the observation. One major flaw of the naive approach is that there is a small possibility that the δ_i modifies the Q-value to be even higher for the best action[2]. The gradient approach however bypasses this possibility by multiplying the direction of the gradient of an objective function to the absolute value of the perturbation, thus not only making the best action less likely but also making the worst possible action more probable[2]. Essential for this approach to work is the possibility to convert all Q-Values into a conditional probability mass function by passing them through a softmax layer. This turns the Q-values into a stochastic policy $\pi^*(a|s)$ which is used in the objective function

$$J(s, \pi^*) = -\sum_{i=1}^{n} n_i log \pi_i^* \tag{35}$$

Algorithm 6 Naive attack from [14]

NAIVE($Q^{target}, Q, s, \epsilon, n, \alpha, \beta$)
$a^* = \underset{a}{\operatorname{argmin}} Q(s,a), q^* = \min_a Q^{target}(s,a)$
for i =1 :n **do**
    $n_i = beta(\alpha, \beta)$
    $s_i = s + \epsilon \cdot 2 \cdot (n_i - 0.5)$
    $a_{adv} = \underset{a}{\operatorname{argmin}} Q(s_i, a)$
    $Q^{target}_{adv} = Q^{target}(s, a_{adv})$
    if $Q^{target}_{adv} \geq q^*$ **then**
        $Q^* = Q^{target}_{adv}$
        $s_{adv} = s_i$
    end if
end for
return s_{adv}

with i being all actions in state s, π^*_i being the probability mass function and n_i=P(a_i) being 1 if action i is the worst action for an adversarial probability distribution P(a_i) and 0 for every other action. The perturbation is then calculated by dividing the gradient of the object function by its L_2 norm and multiplying it by the noise from the naive approach

$$\delta_i = \epsilon \cdot 2(n_i - 0.5) \frac{\Delta_s J(s, \pi^*)}{||\Delta_s J(s, \pi^*)||} \tag{36}$$

This process is also depicted in Algorithm 7. Just like in the naive approach, ϵ is 0.35,n=10 and α or β are 1. The same reasoning as before can also be applied to the adjustments made to [14] and [17] by once again maximizing the Q-Value instead of minimizing it.

5.3.3 Adversarial Policy

The third attack that is going to be performed is the adversarial policy attack. Instead of crafting the perturbed state by sampling some noise, an RL agent is trained to craft the perturbed state. The agent is using proximal policy optimization(PPO)[18] and is trained on the same environment as the victim model. Like any other agent its task is to maximize the reward. The reward is the highest when the action performed by the victim model yields the lowest reward. To achieve this, one way is to simply invert the reward achieved by the victim after receiving the perturbed observation from the adversarial policy agent. This inverted reward can then be returned to the training agent to measure the success of its attack.

While all three attacks have the same goal, they all have different approaches and different amount of knowledge to utilize in their attack. The naive attack and the adversarial policy are both black box attacks due to not having any knowledge about the victim's properties. The gradient approach however has complete knowledge of the victim and uses this in its attack. Later on the question already asked in [2], whether this additional knowledge leads to a stronger attack overall is thoroughly investigated. As seen in Figure 10, the last thing left to do to perform an adversarial attack is determining the attack timing. This can be achieved in numerous way. This thesis is focusing on two such attack timings, namely the uniform attack and the strategically-timed attack.

Algorithm 7 Gradient attack from [14]

GRAD($Q^{target}, Q, s, \epsilon, n, \alpha, \beta$)
$a^* = \underset{a}{\operatorname{argmin}}\, Q(s,a), q^* = \min_a Q^{target}(s,a)$
$\pi^* = softmax(Q^{target}$
grad=$\Delta_s J(s, \pi^*)$)
$grad\_dir = \frac{grad}{||grad||}$
for i =1 :n **do**
    $n_i = beta(\alpha, \beta)$
    $s_i = s + \epsilon \cdot 2 \cdot (n_i - 0.5) \cdot grad\_dir$
    $a_{adv} = \underset{a}{\operatorname{argmin}}\, Q(s_i, a)$
    $Q^{target}_{adv} = Q^{target}(s, a_{adv})$
    if $Q^{target}_{adv} \geq q^*$ **then**
        $Q^* = Q^{target}_{adv}$
        $s_{adv} = s_i$
    end if
end for
return s_{adv}

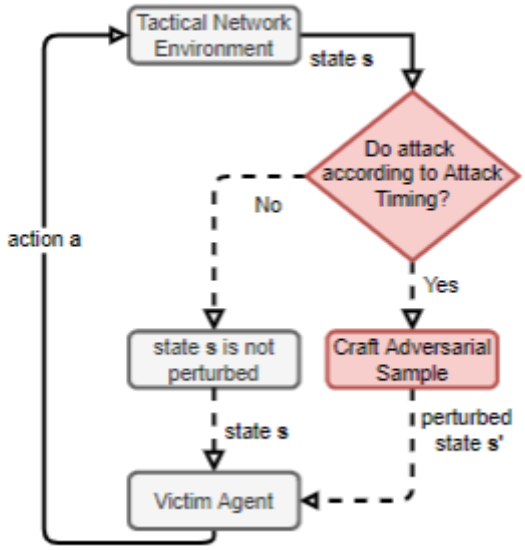


Figure 10. A general sequence of an adversarial attack. Given a state s the first decision to be made is whether the attack timing is right. If yes, the adversarial sample is crafted and the perturbed state s' is fed to the victim agent from [2].

5.3.4 Uniform Attack and Strategically-Timed Attack

The first thing that comes to mind when theorizing about attack patterns would be to attack in every time step to make sure that as many wrong actions as possible can be caused for the victim agent. This idea is called the uniform attack and it acts as a baseline to attack timings. But this

approach contradicts the overall goal to be as stealthy as possible while attacking. To solve this problem the second approach that is utilized is the strategically-timed attack. This attack aims to only perform attacks in important and critical states. To determine a critical state one approach would be to take the best action a^* and the worst possible action a_w of a state s and compare them to each other. This difference can be expressed in a function c(s) and if its result is bigger than a given threshold β the attack is performed. Because all the algorithms use Q-Values, it allows to simply calculate c(s) using

$$c(s) = \max_a Q(s,a) - \min_a Q(s,a) = Q(s,a^*) - Q(s,a_w) \tag{37}$$

and for a given β this leads to

$$attack = \begin{cases} 1 & \text{if } c(s) > \beta \\ 0 & \text{else.} \end{cases} \tag{38}$$

5.4 Natural Events

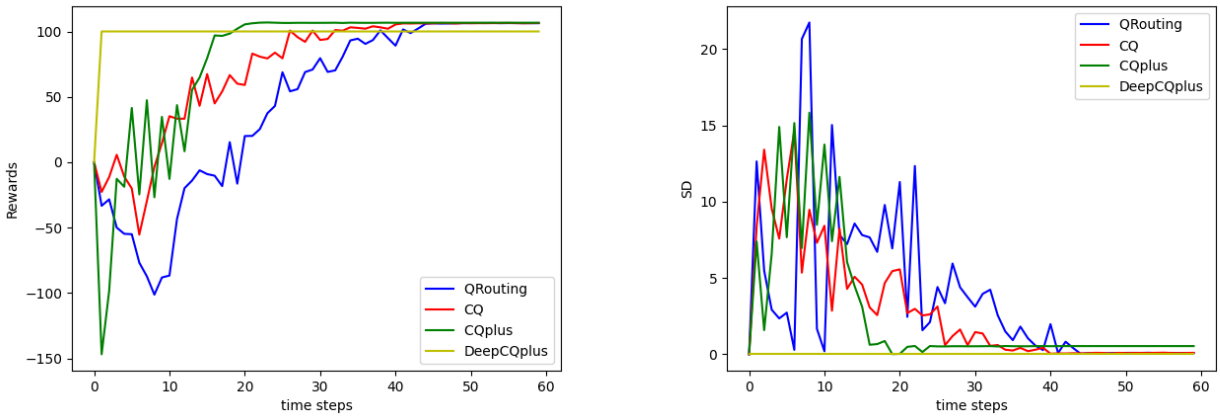
This thesis isn't only investigating adversarial attacks but instead tries to test the robustness of the algorithms against a natural cause as well. The last approach to test the robustness of the RL algorithms is edge or node removal. This describes the sudden disconnect of one node from the network. This can happen especially in tactical networks due to the extreme natural circumstances they may have to deal with. To simulate these scenarios a random edge or node is removed from the network after the RL algorithm has been training for some time already. The most interesting case for this thesis is when the edge that is removed is also the edge currently used by the RL algorithm to solve the path finding problem. This is why the main focus is on this in the following results Section.

6 Results

In this Section the different outside influences, namely adversarial attacks and edge removal, are interacting with the different algorithms to test their robustness. All the tests are performed on the previously introduced environment.

6.1 Baseline

First a baseline has to be set that is used to compare the different algorithms. Figure 11 shows the average rewards achieved over 60 time steps by the four RL algorithms. For Q- and CQRouting ϵ is set to be 0.2. The biggest stand out of the four is DeepCQplus routing which jumps directly to an average reward of 103. On the other hand, CQplus starts with the lowest average rewards at -143 but takes only 15 time steps reach a reward of over 100 as well and even manages to outperform DeepCQplus. The same can be said about CQRouting which has the same mean reward as CQplus after 34 time steps. The worst performing algorithm is QRouting which takes more than 40 time steps to reach approximately the same reward as the other two. Additionally the standard deviation(SD) of the four algorithms is provided in Figure 11. The lowest SD has DeepCQplus,



(a) mean reward **(b)** SD

Figure 11. The mean reward and SD of the four algorithms over 60 time steps

which was to be expected, considering its performance regarding the mean reward. The other three show comparable behavior in their SDs for the first 20 time steps. This window aligns with their mean reward because, after about the same number of time steps they reach an average reward of 100, their SD becomes 0 or close to 0. The very low SD of the different algorithms leads to the conclusion that after a certain amount of time steps all algorithms learn to route a package from the source to the target node without performing a lot or even any illegal actions.

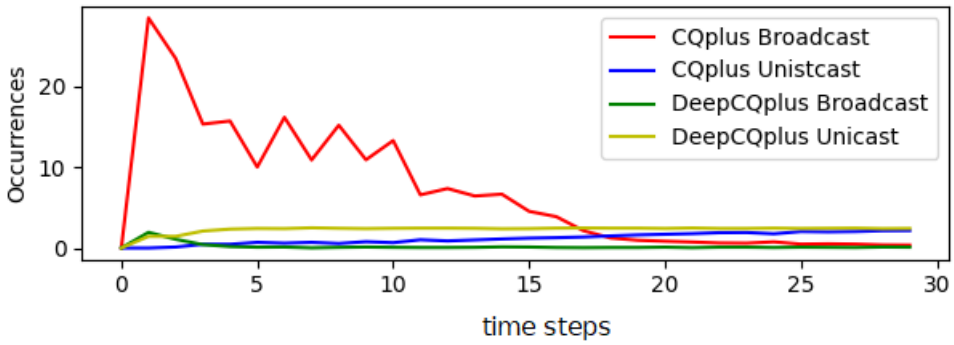


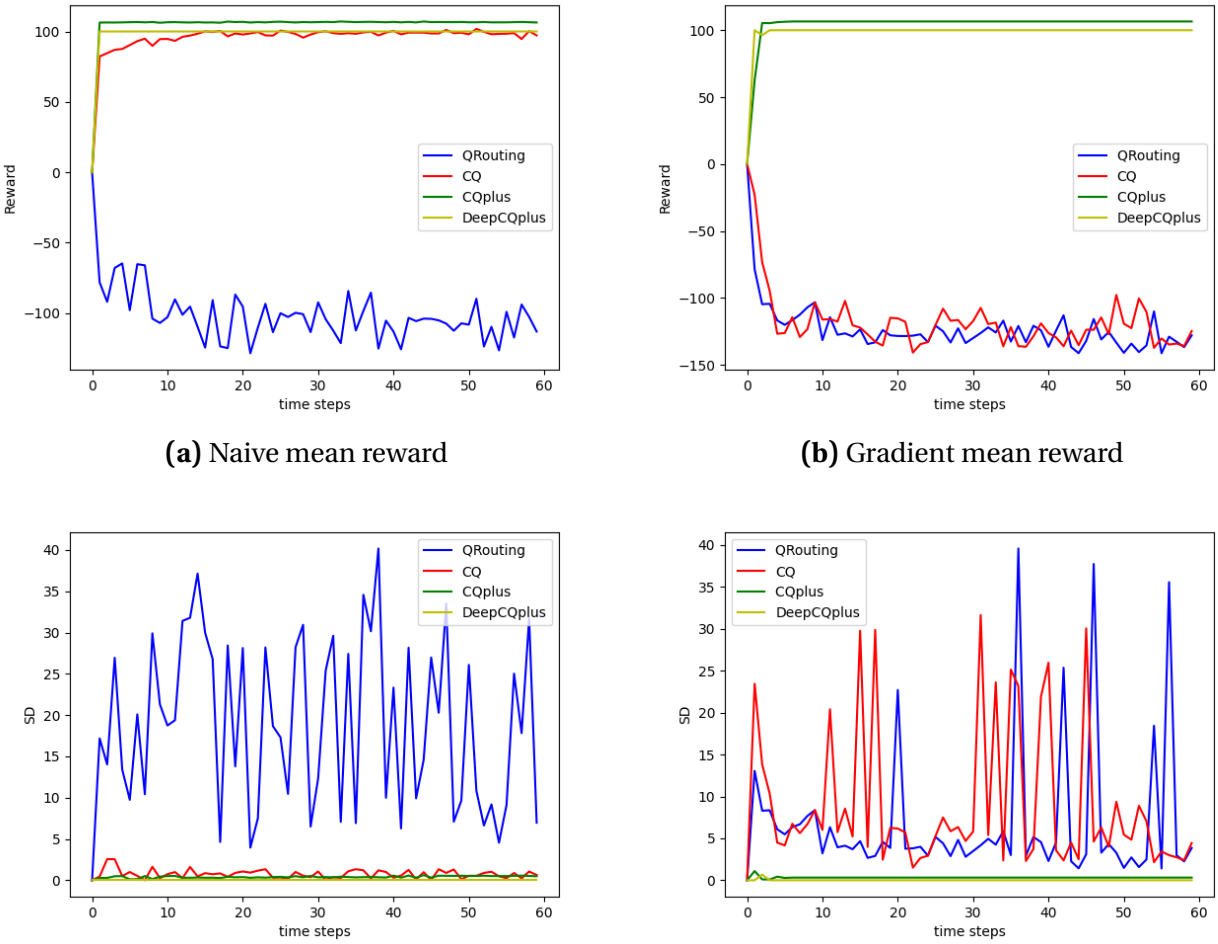
Figure 12. Amount of Unicast and Broadcast performed by CQplus and DeepCQplus.

In Figure 12. the behavior of CQplus and DeepCQplus is shown with a special focus on the comparison of unicast and broadcast performed by each algorithm. The most outstanding observation is that CQplus performs significantly more broadcasts than DeepCQplus with an all time high of 28. The extensive use of broadcast lasts for 17 time steps at which point unicast is the primary form of communication. The DNN policy of DeepCQplus seems to learn to only broadcasts in the first few time steps and even then just a small amount. Afterwards it immediately stops broadcasting after reaching an average reward of 103 and proceeds to only unicast for the remaining duration of the training. This also seems to be the reason behind the large difference in performance, especially in the early time steps between CQplus and DeepCQplus. The reason for the extensive use of broadcast in case of CQplus is that it takes some time for the C-Values to be adjusted and to be closer to 1, hence leading to a higher probability of broadcasts in

the early period of learning. However, the additional use of broadcasts in the beginning enables CQplus to explore the network more thoroughly, therefore leading to the difference in the mean reward shown in Figure 11. The decision by the DNN policy of DeepCQplus to perform the immediate swap from broadcast to unicast after finding one path and to no longer explore different options may be caused by the environments way of rewarding. The environment rewards finding the destination with 100 and all other steps leading to it with way smaller rewards. Modifying the environment to reward more exploration higher may lead to different decision making by the agent and to overall different results.

6.2 Naive and Gradient Attack

For now the main attack strategy is the uniform attack. Figure 13 shows the average rewards



(a) Naive mean reward

(b) Gradient mean reward

(c) Naive SD

(d) Gradient SD

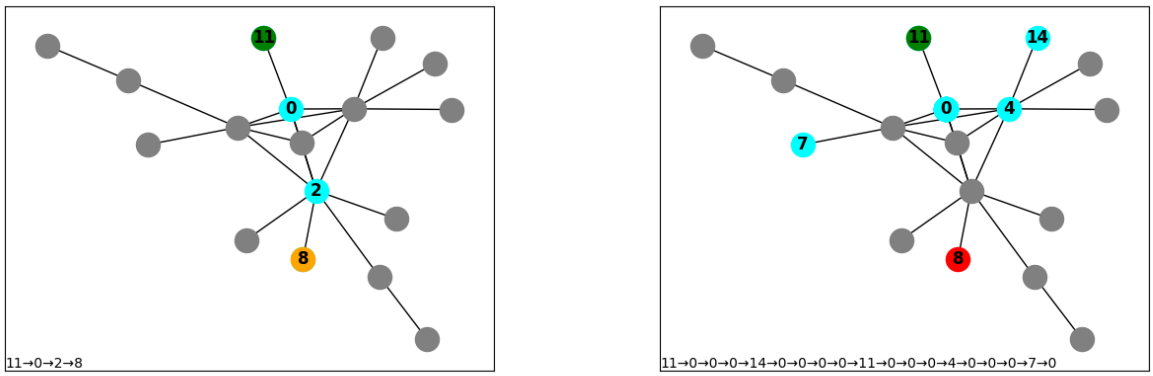
Figure 13. Average rewards of the 4 algorithms while being attacked by the naive and gradient attack in 60 time steps and their SD.

achieved by the 4 algorithms while being attacked by the naive and the gradient attack respectively. The experiments were performed on an already training agent to simulate a natural environment. The parameters of the different attacks are stated as above, with $\epsilon = 0.35$ and n=10. There are two major conclusions that can be drawn from these experiments. On one hand, that CQplus and DeepCQplus do not seem to be affected by the two attacks at all. This is due to the way the two algorithms calculate their Q- and C-Values. While adversarial attacks only perturb the observation, more specific only some parameters of the observation like queue length or packet loss, CQplus and DeepCQplus don't use these information to calculate their Q- and C-Values. The only arguments taken from the observation are the current position of the agent and the target node, which are not affected by the attack. It has been shown in Figure 11 that in case of DeepCQplus this additional protection against adversarial attacks is only punished in a slightly lower mean reward. For CQplus there is no downside at all. Hence both algorithms appear to be incredible useful when dealing with this type of attack.

For CQRouting the naive approach seems to only have a very small impact in the beginning of the attack but after only 10 time steps the average reward of the agent is back to about 100. While this result is still slightly worse compared to the baseline, it is safe to say that CQRouting manages to cope with the naive attack, especially compared to its predecessor in QRouting. The opposite can be said when CQRouting is attacked by the gradient approach. Q- and CQRouting seem to perform very similar with both achieving an average reward of below -100 after a few time steps. Figure 13 also shows the SD for the algorithms while being attacked by the naive attack in 13c and by the gradient attack in 13d respectively. There are two interesting points to me made here. Firstly, that the SD for CQRouting is almost 0 for the naive attack, strengthening the assumption that CQRouting can almost completely overcome it. Secondly, that while QRouting has a rather high SD for the naive attack, the SD for the gradient attack is quite low except some occasional outliers. A possible explanation for this behavior is that the additional knowledge utilized by the gradient attack leaves no room for the QRouting agent to learn at all. Instead he performs constantly illegal actions except for some rare cases. This explanation is back upped by the mean reward of the gradient attack for QRouting, which shows way less variance compared to the naive attack. In Figure 14 a comparison between an undisturbed CQRouting agent and a CQRouting agent being attacked by the gradient attack is provided. Here one can see that while the baseline finds a straight way to the destination node with as little time steps as possible, the same agent struggles severely when attacked by the gradient attack. In this case the agent tries to move to random nodes in the graph, often making illegal actions, like trying to move to node 14 from node 0, leading to a very low reward. After 18 time steps the episode terminates without the agent reaching the destination node, therefore resulting in an overall reward of about -100. These investigations lead to the conclusion that the results for CQRouting are very similar to the results of [2] and that the additional knowledge of the gradient attack is sufficient to perform a very effective attack. Additionally the advantage of the C-Value can be seen by comparing CQRouting and QRouting. The most important result of this Section however, is that both CQplus and DeepCQplus are not affected by the attacks at all, thus making them an excellent option when trying to handle adversarial attacks.

6.3 Adversarial Policy

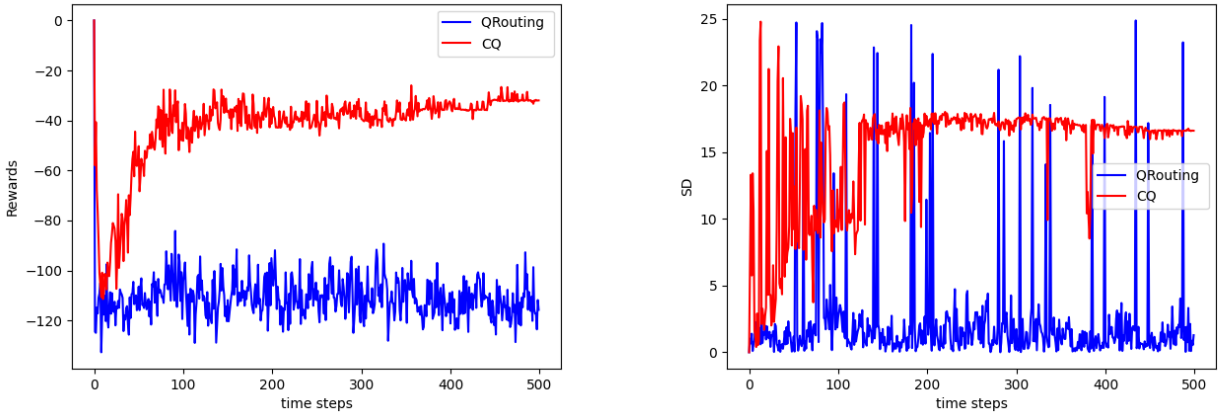
The last attack investigated in this thesis is the adversarial policy approach. It has already been discussed that both CQplus and DeepCQplus are not affected by adversarial attacks. Hence the



(a) Baseline CQ **(b)** Gradient CQ

Figure 14. Figure 14a shows a CQ agent without any interference after 30 time steps of training. The blue nodes are nodes that the agent tried to reach. Figure 14b shows a CQ agent attacked by the gradient attack.

main focus is on Q - and CQRouting. Figure 15a shows the performance of the two algorithms.



(a) mean reward **(b)** SD

Figure 15. Average rewards of QRouting and CQRouting while being attacked by the adversarial policy in 500 time steps and their SD

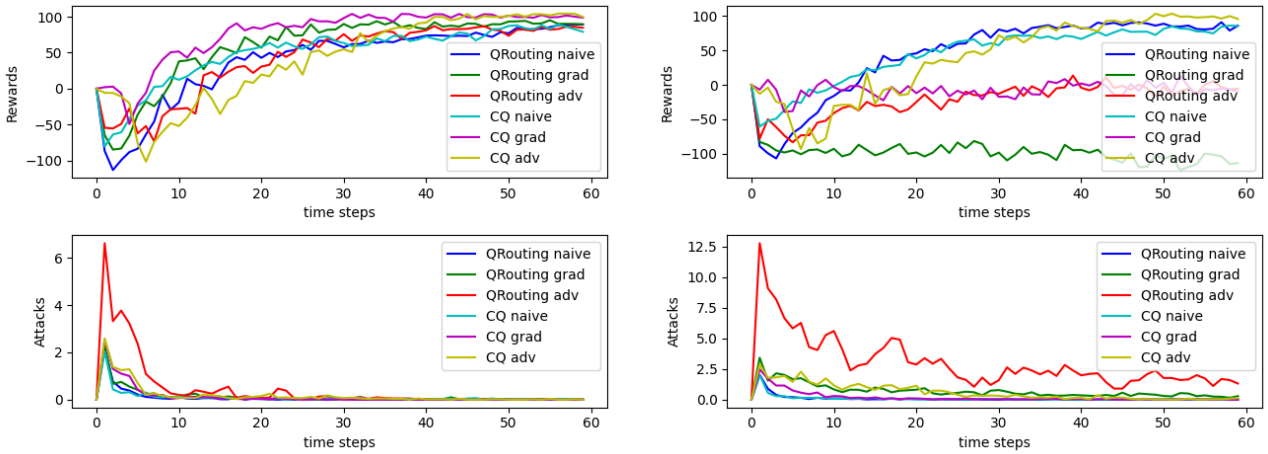
The parameters are the same as for the previous Section with $\epsilon = 0.2$ and n=10. Again QRouting performs very poor,averaging a reward of -120. CQRouting starts off at about the same mean reward but steadily improves for the first 90 episodes and ends up at a mean reward of -40. While this can still be seen as an overall bad result CQRouting shows significant more robustness compared to regular QRouting.

Another interesting observation can be made in Figure 15b. The Figure shows once again the SD of the two algorithms. This time around tho, QRouting has a smaller SD than CQRouting with the SD of CQRouting increasing relative to the the mean rewards achieved, while QRouting

has occasional outliers that surpass the SD of CQRouting, but mostly averages a SD of about 5. This behavior is comparable to that of QRouting being attacked by the gradient attack. Overall this Section once again strengthens the assumptions in [2], that depending on the knowledge available to the attack the effectiveness differs. Also the results of the experiments so far lead to the conclusion that the addition of the confidence value increases the robustness of an algorithm towards adversarial attacks.

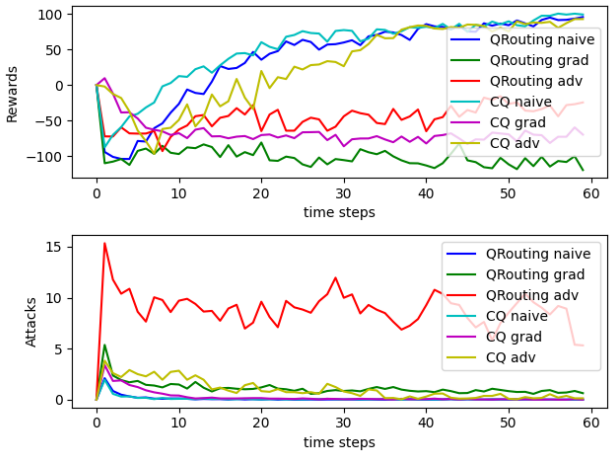
6.4 Strategically-Timed Attack

Up until now this thesis only utilized the uniform attack strategy, meaning an attack is performed in every single time step. Since it's in the best interest of the attacks to be as stealthy as possible, to be able to attack for as long as possible, the strategical-timed attack strategy offers a simple way to ensure this. In Figure 16 a comparison between 12 different β values is given. There can be seen that the performance of some of the attacks vary very strongly depending on the chosen β. Due to our previous investigation it is already known that the β doesn't matter for the naive attack against CQRouting and Figure 16 confirms this. This is due to the fact that the uniform attack is a special case of the strategically-timed attack with $\beta = 0$. Another interesting observation is that

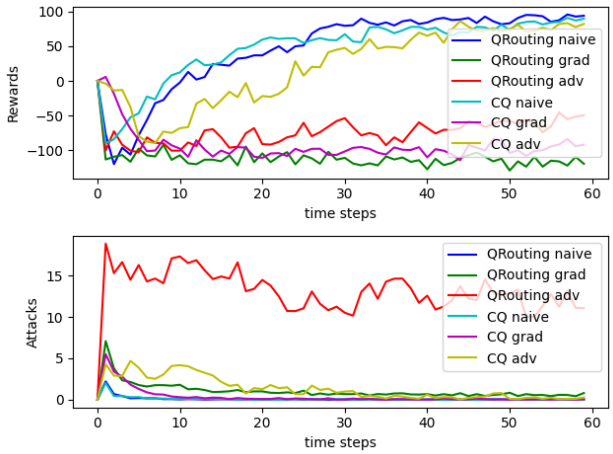


(a) $\beta = 0.99$ **(b)** $\beta = 0.98$

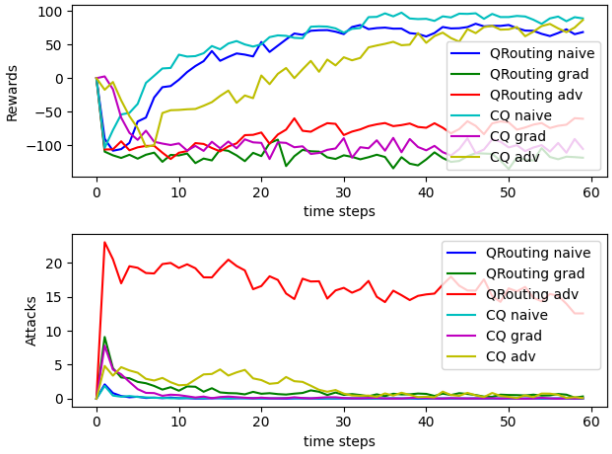
for $\beta = 0.99$ none of the attacks manage to have success but with the slight decrease to 0.98 and 0.97 respectively, the gradient approach manages to attack successfully, further strengthening the assumption that more knowledge leads to better results while also showing that CQRouting is slightly more robust than QRouting to the attack. Additionally the second figure for each β shows the average amount of attacks performed in each episode. Noticeable is that especially in the first few episodes a much larger number of attacks occur compared to later episodes. This behavior is the same for all βs but naturally the amount of overall attacks increases with a decreasing β. The reason for this may be the fact that the attacks are performed versus an already training agent. Thus especially in the beginning there is a high very high certainty in the correct action. But once the perturbed observations are used to calculate the new Q-Values the certainty for one action to be correct gets reduced leading to an overall decrease of attacks due to the difference between the worst and the best action becoming smaller. The only real exception is the adversary policy for QRouting which shows the described behavior only for a $\beta = 0.99$ and afterwards attacks with



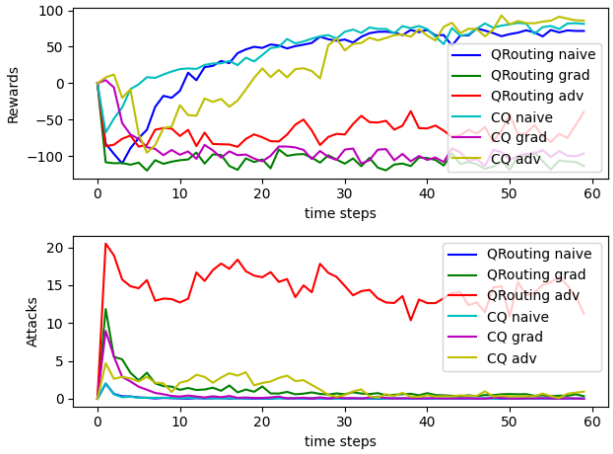
(c) $\beta = 0.97$



(d) $\beta = 0.96$



(e) $\beta = 0.95$

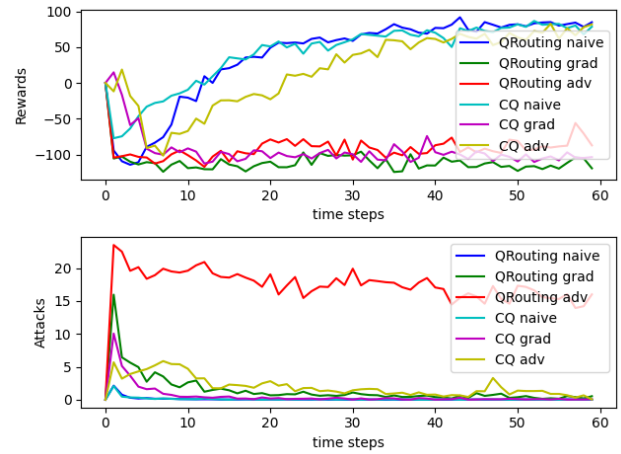


(f) $\beta = 0.94$

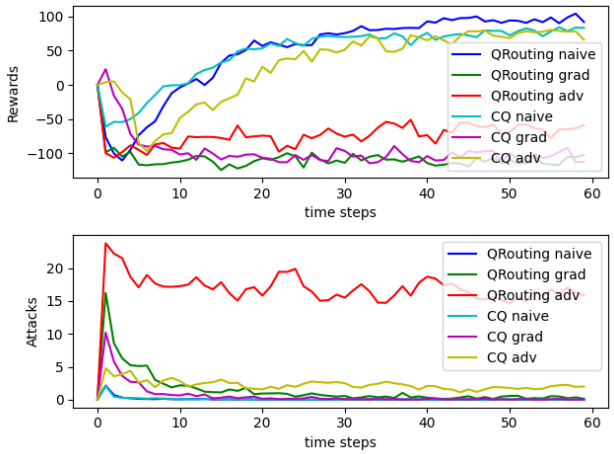
a high frequency very consistent. For $\beta = 0.91$ and smaller the adversary policy for CQ shows a similar behavior but for less average attacks. CQRouting also shows comparable behavior to the previous Section while being attacked by the adversarial policy. It starts off performing very poorly but manages to steadily increase its mean reward up to 100 for very high βs. With the decrease of β the mean reward starts to slowly decrease and at β=0.88 the average reward after 60 time steps is only 50. These results lead to a few interesting conclusions regarding the algorithms and the attacks. For once, it encourages the previous conclusions that the additional C-Values used to compute the Q-Values lead to a more robust algorithm. Additionally the effectiveness of the gradient attack is shown obviously. With only a very small amount of attack especially for later time steps it manages to still decrease the reward just as good as the uniform attack. This makes the gradient attack an even more prominent choice for an attacker because the very low frequency of attacks may result in a harder detection.

6.5 Edge Removal

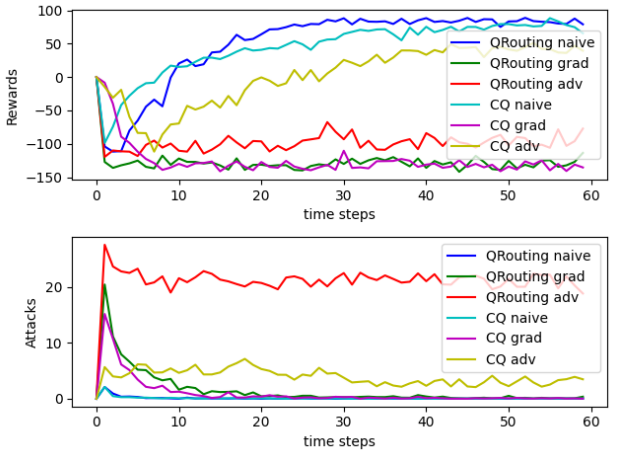
The last form of influence is not performed by an adversary, but is instead a natural occurrence. As mentioned in the previous Section, edge removal is occurring to see how an RL agent, that



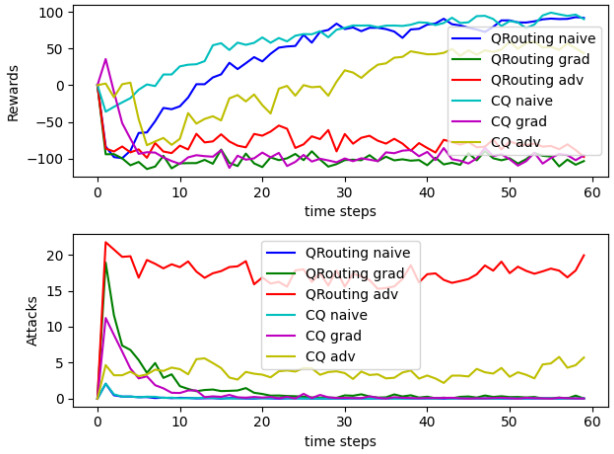
(g) $\beta = 0.93$



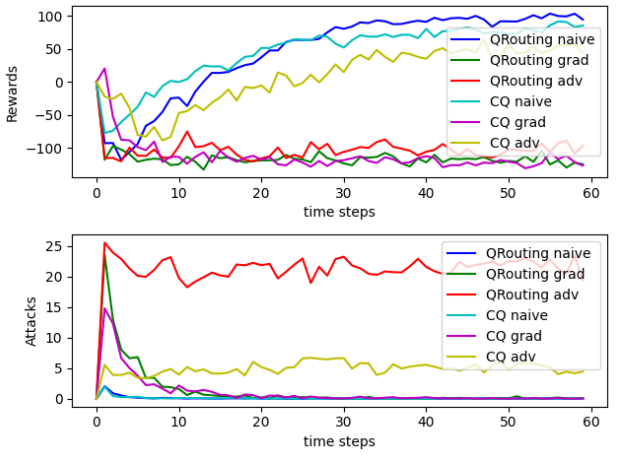
(h) $\beta = 0.92$



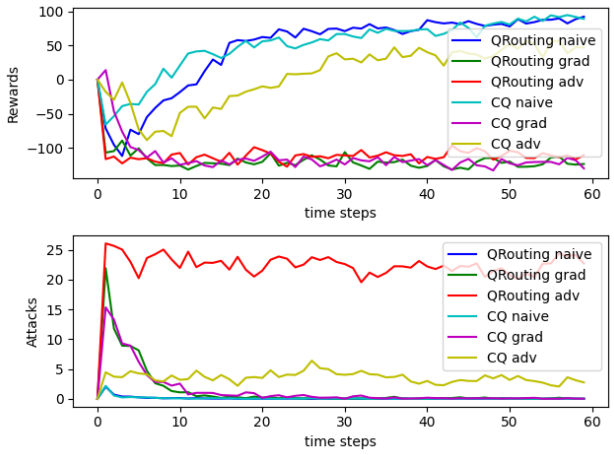
(i) $\beta = 0.91$



(j) $\beta = 0.90$

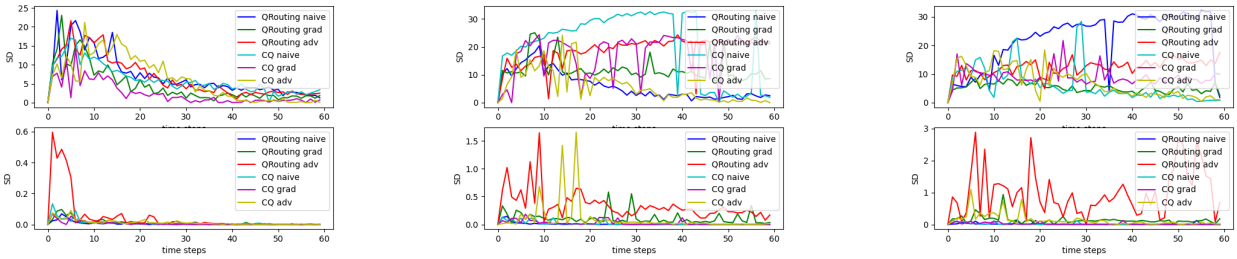


(k) $\beta = 0.89$



(l) $\beta = 0.88$

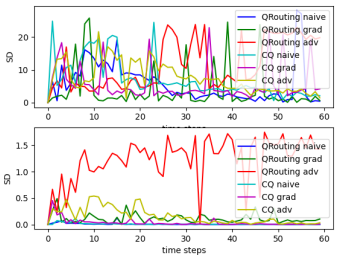
Figure 16. Performance of the naive, gradient and adversarial policy attack using the strategically-timed attack strategy for βs between 0.88 and 0.99



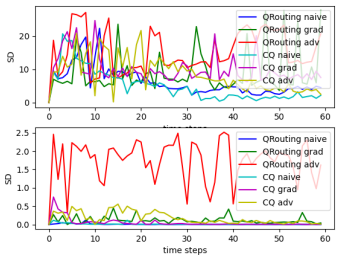
(a) $\beta = 0.99$

(b) $\beta = 0.98$

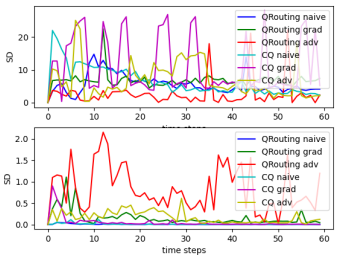
(c) $\beta = 0.97$



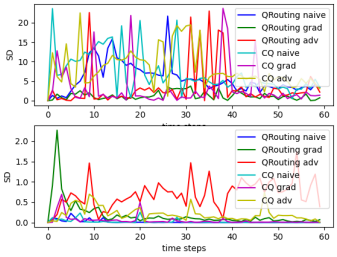
(d) $\beta = 0.96$



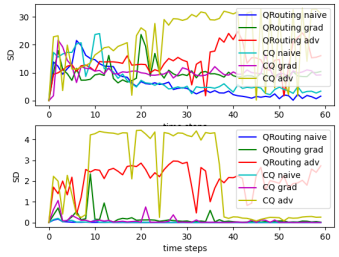
(e) $\beta = 0.95$



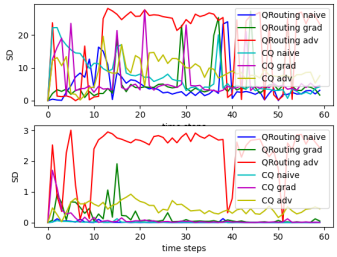
(f) $\beta = 0.94$



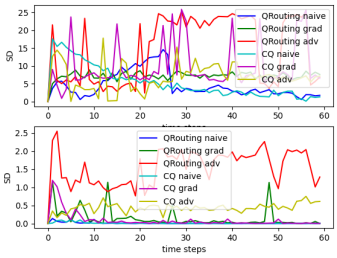
(g) $\beta = 0.93$



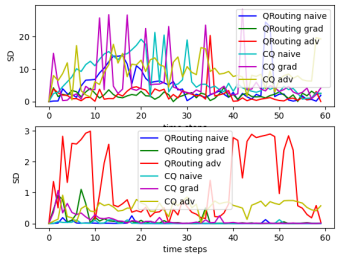
(h) $\beta = 0.92$



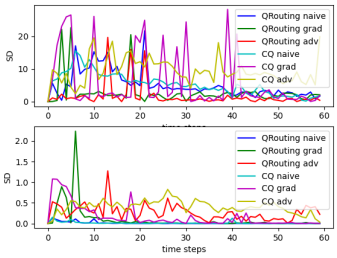
(i) $\beta = 0.91$



(j) $\beta = 0.90$



(k) $\beta = 0.89$



(l) $\beta = 0.88$

Figure 17. SD of the Naive, Gradient and Adversarial Policy Attack using the Strategically-Timed attack strategy for β between 0.88 and 0.99

has already been trained for a number of time steps, deals with it. The edge that is removed is the one that is currently used by the RL agent to reach the target from a given state. In Figure 18 the mean reward for the four algorithms is shown. After 30 time steps the currently used edge is removed and all algorithms have a strong decrease in the average reward. Both CQ and DeepC-Qplus have the lowest change in the average reward and they bounce back immediately to the old average. For DeepCQplus the new average reward is even higher than the old average. One

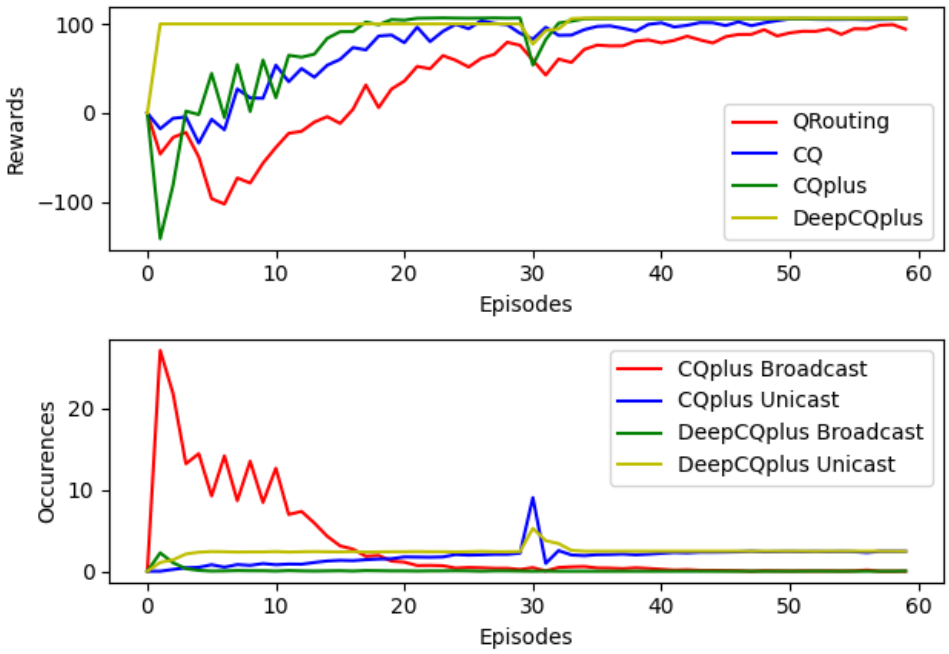
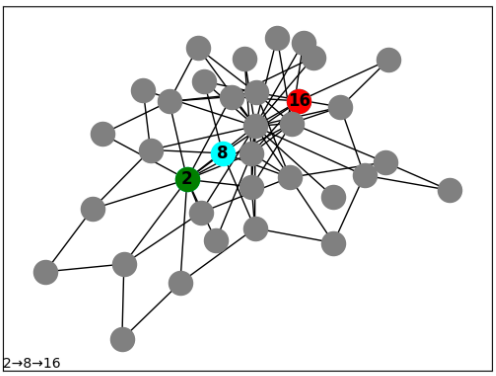
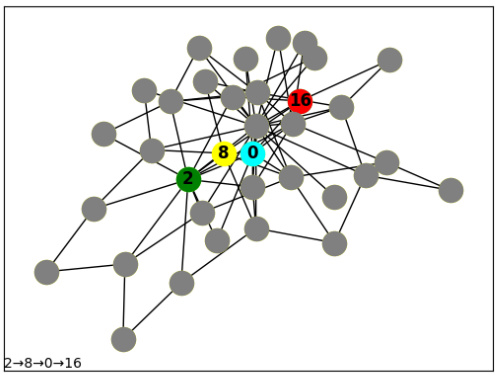


Figure 18. Average rewards over 500 episodes achieved by the 4 algorithms while having the current edge of the best action removed after 30 time steps and amount of unicasts and broadcasts performed by CQplus and DeepCQplus

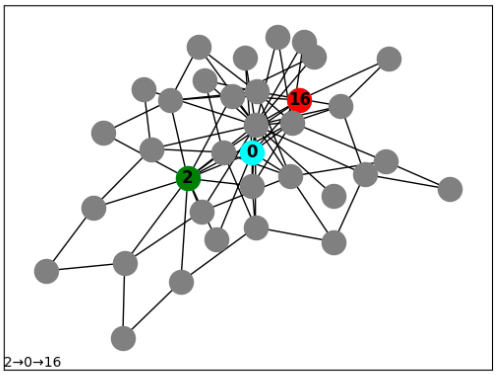
possible explanation for this behavior is the lack of exploration by DeepCQplus while training. It has been shown that DeepCQplus has the worst average reward of the four algorithms meaning that there are better paths still available. Once the edge is removed the new found path may be better than the old path, hence leading to an overall increase of the mean reward. Another interesting observation is that while CQplus and DeepCQplus immediately produce a stable average reward both QRouting and CQRouting take some time to adjust again. This can be also explained by the broadcast and unicast strategy that is also shown in Figure 18. For both CQplus and DeepCQplus a strong increase of performed unicasts can be seen in the same time step the edge is removed. The additional use of unicast results in the lower average reward because at the initial removal of the edge the agents still try to use it, therefore performing an illegal action and receiving a negative reward. After learning that the current path is unavailable DeepCQplus performs a single broadcast and immediately swaps back to unicast after finding a new path. CQplus performs slightly more broadcasts in the time steps immediately following the removal of the edge, but afterwards both continue with a consistent use of unicast. In Figure 19 a visual example is again provided. In this case a DeepCQplus agent is encountered with the removal of an edge to a node he is currently using to reach its destination. As one can see in 19b, the agent tries to reach the removed node exactly one time. This explains the slight decrease in the overall reward. Afterwards he immediately searches for a new node to reach his destination and finds it in node 0. In the next time step the agent no longer tries to reach node 8 but instead uses the new found path, therefore demonstrating the fast adaptability of DeepCQplus and the advantage of deciding intelligent when to broadcast and to unicast. The last Figure 20 regards a question from an earlier Section whether a decaying ϵ or a stable ϵ yields a better reward overall. As the Figure shows, there is one significant difference between the two approaches for CQRouting. With the



(a) Before edge removal



(b) At edge removal



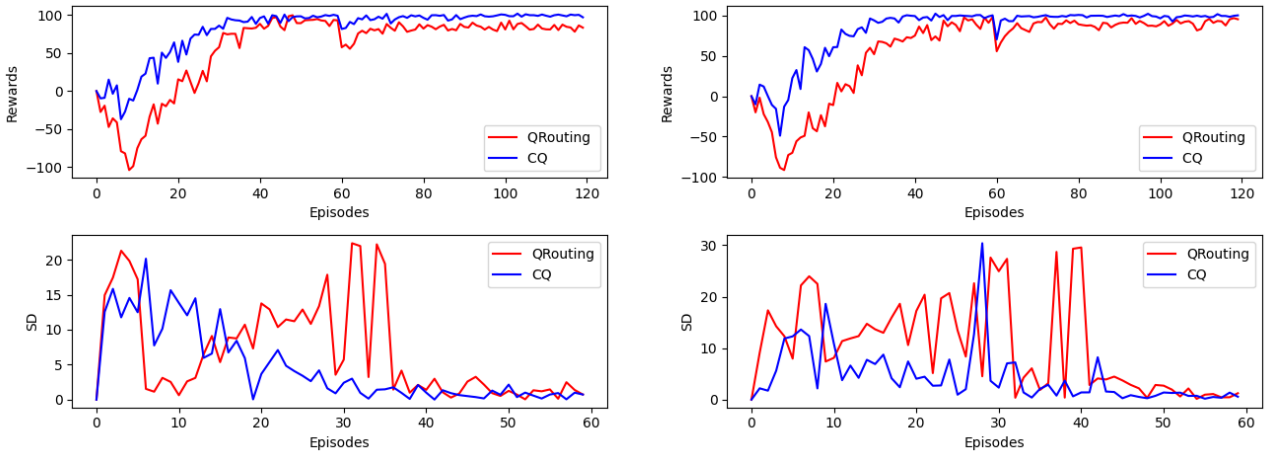
(c) After edge removal

Figure 19. An example of DeepCQplus handling the removal of an edge. Figure 19a shows the DeepCQplus agent before the edge is removed, Figure 19b is the agent in the time step of the edge removal. Figure 19c shows the agent after finding a new path.

stable ϵ there is only a small reduction in the mean reward while the decaying ϵ has a slightly bigger reduction. For QRouting there is no major difference visible.

6.6 Overhead

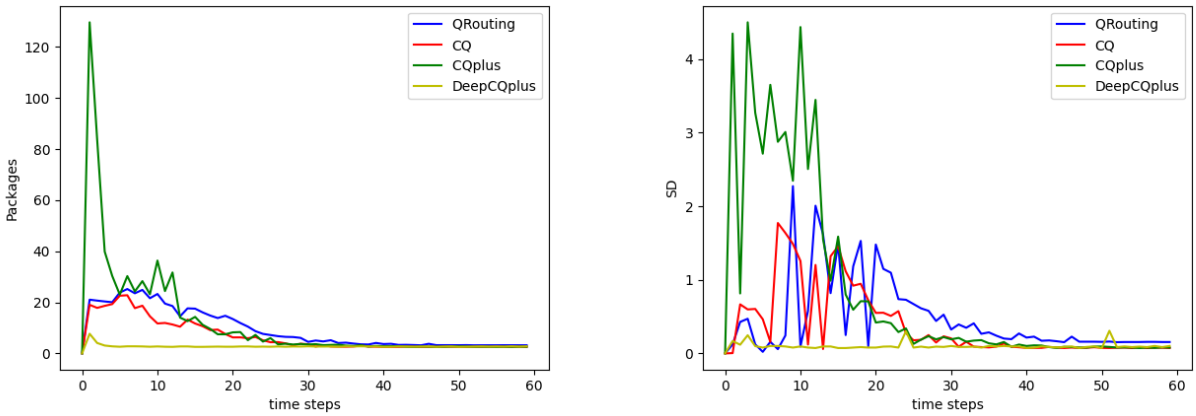
While CQplus and DeepCQplus have obvious upsides compared to QRouting and CQRouting in the usage of unicast and broadcast and their robustness versus adversarial attacks, there is one final question to be asked regarding these two, in whether CQplus or DeepCQplus should be the preferred choice. It has been shown that CQplus is able to achieve a slightly higher average reward than DeepCQplus due to extensive exploration. Usually this would lead to the conclusion that CQplus is simply a better choice than DeepCQplus, however there are two factors that have to be investigated before making this assumption. Firstly, the additional time it takes for CQplus to learn to achieve the higher reward and secondly, the extensive use of broadcasts performed



(a) Edge removal for Q- and CQ-Routing with a constant learning rate $\epsilon = 0.2$ and their SD

(b) Edge removal for Q- and CQ-Routing with a decaying ϵ and their SD

Figure 20. Figure 20a and 20b compare the impact of an decaying ϵ vs a stable ϵ for Q- and CQ-Routing.



(a) Mean packages per time step

(b) SD of packages per time step

Figure 21. The mean amount of packages created in a time step by CQplus and DeepCQplus and their SD

compared to DeepCQplus. In Figure 21 a comparison between the two in regards to the overall amount of packages send in the different time steps is shown. Here one can clearly see that CQplus creates way more packages than all other algorithms, thus leading to a very high overhead in the first few time steps. With these two pieces of information in mind it is not possible to declare a clear winner between the two but instead the decision has to be made depending on the urgency for an agent to learn a network fast and whether it is acceptable to create as much overhead as CQplus does.

With this the results Section is concluded. In the next Section a summary is provided and additionally ideas for future work regarding the topic of this thesis are discussed.

7 Conclusion

The experiments discussed earlier managed to achieve some interesting results. Firstly, the main question of this thesis was whether there are currently RL routing algorithms that can handle adversarial attacks, which can be answered with yes. This thesis has shown that different routing algorithms have different results when being attacked by adversarial attacks. While the most basic and oldest approach, namely QRouting, did not manage to cope with the attacks and had its performance drastically decreased by all three attacks, there was already a big improvement by simply adding a confidence value in CQRouting. CQRouting managed to withstand the naive attack almost completely, while also performing better then its predecessor when being attacked by the adversarial policy. However the gradient attack still resulted in the agent completely losing all its prior knowledge and performing comparable to QRouting, strengthening the already existing assumption that additional knowledge of the attacked agent results in an increase of performance by the attack. The last two algorithms, namely CQplus and DeepCQplus however, managed to completely ignore the attacks due to having a different approach to calculating their Q- and C-Values. This makes them a very fitting choice when the fear of adversarial attacks persists.

Additionally this thesis investigated the impact of natural influences like edge removal. Here was also shown that CQplus and DeepCQplus outperform their two competitors, but on a way smaller scale. However there were also downsides discovered when dealing with the two algorithms. For once CQplus has a very high overhead especially in the beginning of training. While DeepCQplus is the exact opposite of CQplus in this regard by having the smallest overhead of all four investigated algorithms, it also has a major flaw. Unlike the other algorithms DeepCQplus does not maximize its reward but instead takes the first path encountered. For future work multiple topics opened up while researching for this thesis. For once, DeepCQplus has to be tested in an environment that punishes little exploration more severely to see if this alters the decision making of the DNN policy. At the same time a different and more complex environment may lead to different results regarding the other algorithms as well. A second topic that needs further investigation are the type of algorithms and attacks discussed in general. This thesis only compared a small amount of algorithms and attacks with a lot of different approaches yet to be researched.

References

[1] S. Kaviani, B. Ryu, E. Ahmed, K. Larson, A. Le, A. Yahja, and J. H. Kim, "Deepcq+: Robust and scalable routing with multi-agent deep reinforcement learning for highly dynamic networks," 2021.

[2] F. Spelter, J. F. Loevenich, J. Bode, T. Hürten, L. Liberto, P. H. Rettore, and R. R. F. Lopes, "Adversarial attacks against reinforcement learning based tactical networks: A case study," *MILCOM*, 2022.

[3] R. S. Sutton and A. G.Barto, "Reinforcement learning an introduction second edition," 2018.

[4] C. Yu, J. Liu, and S. Nemati, "Reinforcement learning in healthcare: A survey," 2020.

[5] J. C. Caicedo and S. Lazebnik, "Active object localization with deep reinforcement learning," 2015.

[6] J. S. David Silver, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," 2017.

[7] C. J. WATKINS, "Q-learning," 1992.

[8] J. Hoebeke, I. Moerman, B. Dhoerdt, and P. Demeester, "An overview of mobile ad hoch networks: Applications and challenges," 2004.

[9] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," 1993.

[10] Z. Mammeri, "Reinforcement learning based routing in netowrk: Review and classification of approaches," 2019.

[11] S. Kumar and R. Miikkulainen, "Confidence-based q-routing: An on-line adaptive network routing algorithm," 1998.

[12] M. Johnston, C. Danilov, and K. Larson, "A reinforcement learning approach to adaptive redundancy for routing in tactial networks," 2019.

[13] S. Kaviani, B. Ryu, E. Ahmed, K. Larson, A. Le, A. Yahja, and J. H. Kim, "Deepcq+: Robust and scalable routing with multi-agent deep reinforcement learning for highly dynamic networks," 2021.

[14] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary, "Robust deep reinforcement learning with adversarial attacks," 2017.

[15] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell, "Adversarial policies: Attacking deep reinforcement learning," 2021.

[16] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adverasrail attacks on neural network policies," 2017.

[17] J. F. Loevenich, P. H. Rettore, R. R. F. Lopes, and A. Sergeev, "A bayesian inference model for dynamic neighbor discovery in tactical networks," *Procedia Computer Science*, vol. 205, pp. 28–38, 2022. 2022 International Conference on Military Communication and Information Systems (ICMCIS).

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proxial policy optimization algorithms," 2017.