Sebastian Penttinen

COLREG compliant collision avoidance using reinforcement learning

Sebastian Penttinen 38630
Master's thesis in computer science
Supervisors: Marina Waldén, Fatima Shokri-Manninen
Faculty of Science and Engineering
Åbo Akademi University
2020

Sebastian Penttinen

Abstract

The maritime industry could benefit from autonomous vessels. The benefits would come from decreased accidents, fewer casualties at sea, and cost savings. The decreased accidents and fewer casualties would come as a result of taking the human factor out of the industry. The cost savings would originate from the reduced number of personnel needed to operate vessels. There are challenges on the way to making autonomous vessels a reality. The main challenge is algorithms capable of safely navigating and controlling autonomous vessels. This thesis will try to aid in the solution of this problem by exploring the possibilities reinforcement learning could bring to the industry. The goal is to implement a proof of concept collision avoidance system using reinforcement learning. The implemented reinforcement learning agent needs to follow the COLREG rules.

A proof of concept solution is implemented in Python and tested in a simulator. The results seem promising. Further work is, however, needed to make the developed reinforcement learning agent fully follow the COLREG rules.

Keywords: Reinforcement learning, Maritime autonomous systems, Collision avoidance, COLREG, Safety, Navigation

Preface

I would like to thank my supervisor Marina Waldén for all the help and guidance with my thesis. I would also like to thank Fatima Shokri-Manninen for the interesting discussions and support provided when I felt lost in my topic. Thanks also to Ivan Porres for the feedback on my code and for presenting the interesting opportunity to work on the simulator. Also thank you to Kim Hupponen for the great teamwork when building the simulator. Lastly, a great thank you to Johan Lilius for hiring me to write my thesis and to my colleagues at the IT department for the great working atmosphere and support.

Sebastian Penttinen
Turku, 14 April 2020

Contents

1. Introduction

Autonomous vessels have risen in demand during recent years. The rise in demand is due to two factors. Firstly, human errors at sea are partly responsible for 75-96% of loss of life at sea according to reports [1]. Additionally, it has been shown that 89-96% of collisions at sea are caused by human error [1]. Reports have also shown that 56% of these collisions occurred since the COLREG rules [2] where violated [3]. The second factor playing a part in the increased demand is the natural cost savings that come from having smaller crews operating vessels. Autonomous vessels with no crew onboard are also possible. The challenge with fully autonomous vessels is that it puts a high demand on the vessels and their safety. The autonomous vessels would need to make decisions without human intervention in every step and possible scenario at sea. This means that they would, for instance, have to be able to avoid collisions on their own. This is not a trivial or easily solvable issue with a high level of complexity.

This thesis will provide a step towards a solution for fully autonomous vessels by implementing a proof of concept collision-avoidance procedure for autonomous vessels. The collision-avoidance procedure will follow the COLREG rules [2]. The collision-avoidance will be based on *Reinforcement Learning* (RL). The programming language used for the implementation is *Python*. The implementation will utilize the Python version of the libraries *Tensorflow* [4], *Keras* [5], *Keras-rl* [6] and *Open Ai gym* [7].

The practical part of the thesis will consist of the construction and implementation of an RL agent that follows the COLREG rules. The agent will initially be built as a proof of concept and will serve as a basis for further research and investigation. The proof of concept in question will be implementing a collision avoidance system based on RL. The collision avoidance system will be able to follow the COLREG rules in the four base collision avoidance situations found in the COLREG rules and, additionally, the agent should handle a multiple vessel scenario. The base situations are head-on, overtaking, crossing from left, and crossing from right. The different COLREG situations will be described in greater detail in chapter two. The agent should be able to generalize. Hence, safe completion of a multiple vessel scenario is also a goal.

The proof of concept RL system will be constructed as a part of a larger implementation of a ship simulator. The simulator will handle the visualization of the RL system and add real-life environment factors such as physics to the RL training environment. The

simulator itself is built by Kim Hupponen [8] as a related but separate master's thesis.

In order to understand the context and implementation of the RL agent the thesis will cover and explain; safety and collision avoidance in a maritime setting, reinforcement learning, Q-learning, and deep Q-learning. The simulator agents are placed in for testing purposes will also be explained.

The thesis is structured the following way. First, context will be given in chapter two by describing maritime safety and collision avoidance. In chapter three, maritime autonomous systems will be described. In chapter four, the theory behind RL will be covered. Chapter five describes the implementation of the RL system and the simulator it is placed inside. In chapter six, the results are presented and discussed. Chapter seven outlines the conclusions of the thesis.

2. Maritime safety regulations

According to the International Maritime Organization (IMO), shipping is one of the world's most dangerous industries [9]. IMO believes the best way to tackle the dangers in the shipping industry is to make regulations that all nations partaking in the industry have to follow. One such regulatory treaty is the International Convention for the Safety of Life at Sea (SOLAS). SOLAS dates back to 1914 when it was drafted in response to the Titanic accident. It has since then been updated and the last version is from 1974 and amendments have been continuously added. The SOLAS treaty regulates a vast number of things, from mandatory life-saving equipment to specifications for nuclear ships [9].

Despite the conventions and treaties, shipping is a dangerous field. The inherent danger of the field is mostly due to humans being a part of it. Where there are humans involved, there will also be human mistakes. According to Rothblum's report, "Human Error and Maritime Safety" [1], 75-96% of casualties at sea were due to some degree of human error. In 2019 alone, 3174 maritime casualties and accidents were reported according to Safety4Sea [10]. This means that if the human factor could be taken out of the industry, potentially over 2000 accidents could be prevented. A large number of those accidents may have caused loss of human lives. The need for automation resulting in greater safety is clear.

2.1 COLREG

The *Convention on the International Regulations for Preventing Collisions at Sea* (COLREG) is the regulations addressing collision avoidance at sea. The COLREG rules can be thought of as the road rules for the sea. The same way as there are regulations in place to govern the use of roads, COLREG governs the rules of sailing the sea. COLREG in its current form was established in 1972 by IMO [2]. COLREG contains in total 41 rules. The most interesting rules for the proof of concept implemented in this thesis are the rules 8, 13, 14, 15, 16 and, 17.

Rule 8 defines in what manner actions should be taken in order to avoid a collision. Rule 8 outlines that all actions to be taken have to be done in ample time and in line with good seamanship. Additionally, all the actions taken shall be large enough to clearly signal to other vessels what actions have been taken. Altering the course shall be done

as the primary action to avoid a collision if there is enough space available. The passing distance between vessels also needs to be safe. The need to take further collision avoiding actions shall be assessed until the situation is over. If there is no other way to avoid a collision or more time is needed in order to avoid one, a vessel can take actions to slow down or reverse [2].

Rule 13 outlines the rules for overtaking another vessel. According to rule 13, any action taken to overtake another vessel needs to first comply with the rules defined within rule 8. Additionally, the overtaking vessel needs to always avoid getting in the way of the vessel it is overtaking. Rule 13 also gives the degrees of approach (22.5) for when a situation should be considered an overtaking. In addition to the degrees, the rule also addresses night-time conditions when it would be difficult to assess the angle of approach. The rule states that it is an overtaking scenario if the overtaking vessel, when it comes up on the vessel to be overtaken, only is able to see the vessel's stern light (see Figure 2.2) and none of its sidelights. Knowing if it is an overtaking scenario or not might be difficult in some cases. Rule 13 deals with this by specifying that if there is any doubt about whether it is an overtaking vessel or not, one should assume it is an overtaking vessel and act according to rule 13. The rule also outlines that the overtaking vessel needs to make sure that it does not overtake in such a manner that it creates a crossing scenario by having an interesting course with the vessel it just overtook [2]. An illustration of the intended path of the overtaking vessel can be seen in Figure 2.1.

The 14th COLREG rule defines how to act when there is a head-on situation. According to the rule, the vessels that are meeting head-on (have reciprocal courses) or nearly head-on both need to adjust their courses so they pass each other on the port side of the other vessel, meaning that both vessels turn towards their starboard. Rule 14 specifies that there is a head-on situation, if a vessel sees another vessel coming towards it ahead or close to ahead. At night, the head-on situation is assessed according to the rule deemed to exist if the masthead light is visible or both sidelights are visible. Figure 2.2 gives an intuition of the angles at which the lights are visible. Just as in rule 13, rule 14 also specifies that if there is any uncertainty whether a situation is a head-on situation or not, the situation should be treated as one [2]. An illustration of the corrective actions to be taken by the vessels in accordance with rule 14 can be found in Figure 2.1.

COLREG rule 15 governs crossing situations. It states that when two vessels have intersecting paths (crossing paths) with a risk of collision, the vessel that has the other vessel involved in the situation on its starboard side needs to keep out of the way. This should be done by not crossing in front of the other vessel and instead cross behind the other vessel. The rule also states that this action only should be taken if the situation allows for it, meaning that in certain situations it could be acceptable to cross in front if there are no other option [2].

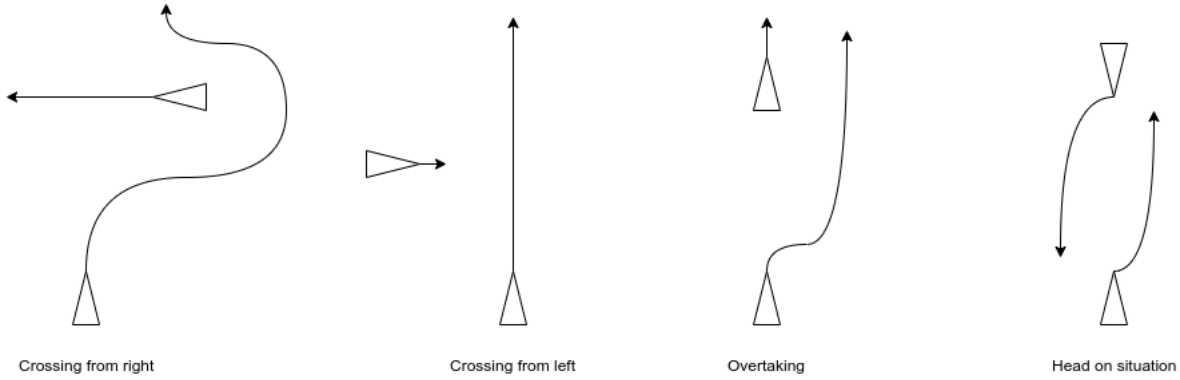


Figure 2.1: COLREG Rules

Rule 16 of the COLREG rules specify what the give-way vessel should do in a scenario. The rule is simple and straightforward in stating that the give-way vessel should take every available action to keep out of the way. Additionally, these actions also need to be done as early as possible [2]. Figure 2.1 also illustrates the actions to be taken by vessels in crossing situations.

The 17th COLREG rule states what the stand-on vessel should do. According to the rule, the stand-on vessel should keep its current speed and course. The stand-on vessel can also itself take actions to avoid a collision, if it appears as if the other vessel (give-way) is not following the COLREG rules and neglecting to take collision-avoiding actions. The rule also states that if the situation has gone so wrong that it is deemed that actions by the give-way ship alone are not going to avoid a collision, the stand-on ship should do whatever it takes to avoid a collision. The rule continues to state that in the crossing scenario the stand-on vessel should not alter its course towards port if the other vessel is on its port side. Essentially, this means the stand-on vessel should not close the distance between the vessels. Lastly, rule 17 reminds that it does not relieve the give-way vessel in rule 16 from its obligation to keep out of the way [2].

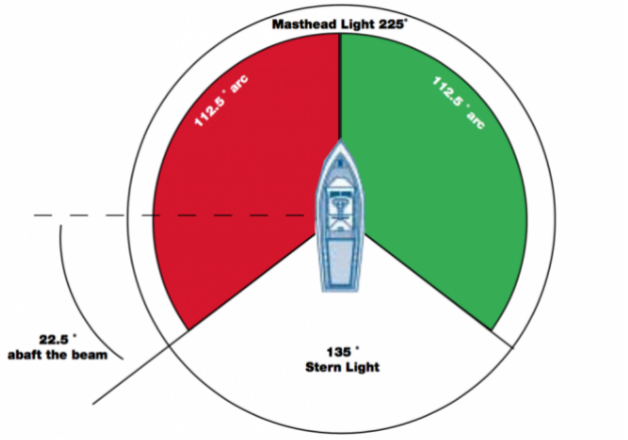


Figure 2.2: Navigation lights [11]

The COLREG rules are written for humans and, hence, are full of vague definitions that depend on human interpretation. One such example is rule 8 stating that all actions should be taken in ample time and in accordance with good seamanship. Good seamanship is understood by a human captain with experience in the field. Programming and defining good seamanship into an RL agent that controls an autonomous vessel is non-trivial. This means that such human elements of the rules may not be sufficiently captured and implemented in the agent. If they are captured, they might be biased towards what the author in this case believes is ample space and good seamanship. This can be a drawback of the implementation, since the author has a background in computer science and limited experience from the maritime field.

3. Maritime autonomous systems

In the maritime space, autonomous vessels go by the names *unmanned surface vessel* (USV) or *autonomous surface craft* (ASC). Both names hint at the removal of personnel from the vessel. It is this lack of personnel onboard a vessel that defines it as being an autonomous vessel, as defined by Liu et al. [12]. In addition, Liu et al. define that a USV needs to be capable of highly non-linear dynamics [12]. Essentially, they need to exhibit some form of intelligent behavior.

Autonomous vessels are not a new phenomenon. They have been developed for about 20 years already [13]. Most of the USVs tested and prototyped so far have, according to Liu et al. [12], been semi-autonomous, meaning that they still have a human involved in the loop at some point. They can be compared to fully-autonomous vessels that are completely independent of human input and guidance.

Liu et al. [12] point out the numerous benefits that could be gained by implementing autonomous vessels. The benefits of automation would first come in the form of cost savings. These savings are related to the reduced manpower needed to operate the vessels. Without the need to maintain a crew onboard, the vessel's operational range and reliability would also be increased. Safety in the industry would also increase [12]. As pointed out before, the vast number of accidents in the shipping industry stems from human mistakes and errors [14]. The unmanned vessels could also be used in dangerous missions or missions with a higher risk associated with them, since no human life would be on the line [12]. An interesting benefit of autonomous vessels is that they tend to be lighter and more nimble compared to traditional vessels. This means that they can sail shallower waters and waters otherwise not accessible by traditional vessels. Autonomous vessels also have the capability of carrying a heavier load compared to traditional vessels [12]. This means further reducing costs and emissions, if fewer trips are needed to move the same amount of cargo.

Despite all the advantages of autonomous vessels and prototype implementations in research and governmental use, autonomous vessels have had very little commercial adoption [13]. The lack of commercial use so far is due to the challenges that come along with sailing on the sea with others. An autonomous vessel needs to be able to interact with other vessels and take them into consideration when sailing [13]. Fully-autonomous vessels are not yet seen due to the challenges with reliable and completely safe opera-

tion. They need to be able to handle vastly different and complex situations [12]. In essence, this means making the autonomous vessels comply with the COLREG rules and safely handle ambiguous situations. Unclear and ambiguous situations will occur when autonomous vessels operate in a field that includes humans.

4. Reinforcement learning

One approach to overcome the challenges related to developing autonomous vessels is *Reinforcement Learning* (RL). RL is able to generalize and find the optimal order in which to take actions and make decisions, given a set of rules. Making autonomous vessels sail and navigate safely can be thought of as an optimization problem. In what order and how long should a vessel take certain action in order to accomplish a goal within certain parameters? RL can be used to learn this.

This chapter will provide background and intuition about RL. The chapter starts with some general history and background on RL. After this, the mathematical basis of RL and the parts that make up an RL system are explained. Lastly, ways of solving the RL problem are presented.

4.1 History

Sutton and Barto cover the history of RL in their book *Reinforcement Learning: An Introduction* [15]. In their book, they explain that RL has two different independent starting points that later merged into RL as known today. The two starting points of RL were "learning by trial and error" and "optimal control" [15].

The optimal control starting point is based on the work done by Richard Bellman and others to control dynamical systems, more exactly the work Bellman and others did relating to defining states of dynamical systems and value functions. The function they invented is now called the *Bellman equation*. The Bellman equation is discussed later in this chapter. The field of study that tries to solve optimal control problems by solving Bellman equations became what now is known as *Dynamic Programming* [15].

Trial and error learning is a much older starting point as it has its roots in psychology, according to Sutton and Barto. They state that some form of trial and error learning was discussed as early as in the late 1800s. According to Sutton and Barto, the first one to combine trial and error learning with computers was Alan Turing. They mention that Alan Turing, in one of the earliest ideas about artificial intelligence, considered a pleasure and pain system following the ideas around trial and error learning [15].

4.2 Background

Reinforcement Learning is machine learning combining different disciplines [16]. The disciplines making up the combination are *Dynamic Programming* and *Supervised Learning*. This combination makes it possible for RL to solve problems that dynamic programming and supervised learning are not able to solve on their own. Dynamic programming is a mathematical field focused on solving control and optimization problems. A drawback of dynamic programming is its limited ability to handle extremely large and complex problems. Supervised learning is a machine-learning method used to find parameters for functions. The found parameters are used to approximate outputs from functions. The drawback of supervised learning is that it requires data to accomplish the approximations. The data required is often in the form of a set of questions and answers. The need for data is a drawback. This is because there are many situations where data is not available or it is too expensive to acquire. It could also simply be the case that the correct answers are not yet known [16].

Harmon et al. also correctly state that RL is a powerful method. It has the possibility of solving problems thought to be unsolvable. The power of RL is in its generality. The generality comes from RL working on the basis of trial and error. In other words, an RL system is given a goal to achieve and then uses trial and error to achieve the goal [16].

To provide intuition for how RL works let us consider the example given by Harmon et al. The authors give an example of an RL system learning how to ride a bicycle. The goal of the RL system is simple. It is supposed to learn how to ride a bicycle without falling over. The possible actions the RL system can take is turning the handlebars of the bicycle to the left or right. The RL system starts and, during the first try, it does some actions (turning of the handlebars) and ends up in a state where the bicycle is tilting 45 degrees to the right. At this point, the system has to choose what action to take. In the example, the authors decided that the system chooses to turn the handlebars to the left and, as a result, the system will crash the bike. As a direct result of the crash, the system will receive feedback, also called a *reinforcement*. This reinforcement will lead to the RL system learning that it will receive a negative reinforcement from turning the handlebars to the left when the bicycle is tilting 45 degrees to the right.

The RL system starts over again. In the next run, the RL system once again does a series of actions leading to a state where the bicycle is tilting 45 degrees to the right. Based on the knowledge the RL system gained from the previous attempt it knows that it will receive a negative reinforcement if it turns the handlebars to the left in this state. Instead of the known bad action, it chooses to turn the handlebars to the right. This action causes the system to once again crash the bike and receive a negative reinforcement.

However, this time the system learns more than from the previous run. It does not only learn that turning the handlebars to the right is associated with a negative reinforcement when the bicycle is tilting 45 degrees to the right. The RL system will learn that being in the state of 45 degrees of tilt to the right is associated with negative reinforcement. This is important, since the RL system will utilize this information in future attempts. In the following attempt, the RL system ends up in a state of 40 degrees of tilt to the right. The following action it chooses to take is turning the handlebars to the right. This action leads to the bicycle titling 45 degrees to the right. The RL system now realizes it will receive the negative reinforcement and can stop. This attempt also provides the RL system with additional useful knowledge. The system learns that turning to the right when the tilt is 40 degrees to the right will result in negative reinforcement. Once the system has performed enough of these trial and error runs, it will learn what actions it has to perform in order to avoid the negative reinforcements or, in other words, it learns to ride the bicycle [16].

This example by Harmon and Harmon explains the intuition behind RL nicely. It manages to capture the essence that RL systems learn what states are suboptimal to be in and not just what actions are undesired.

4.3 Mathematical basis

This section will cover the mathematical basis of RL. The essential mathematical foundation of RL is based on the *Markov Decision Process* (MDP). This claim is based on the statement by Sutton et al. [15] in their book where they state that MDPs are considered the traditional way of formalizing a sequential decision-making process. Further, Sutton et al. claim that almost any RL problem can be expressed as an MDP. They even go as far as stating that MDPs are the ideal mathematical way of representing RL problems [15].

According to Silver [17], MDPs are a formal way to represent fully observable environments for RL. This is in line with the earlier presented statements by Sutton et al. Both Silver and Sutton et al. explain that an MDP starts from the Markovian property. The Markovian property means that all possible future states are fully independent from all previous states if the current state is known. The mathematical representation of the Markovian property is given by Silver. A process is Markovian if the following holds [17]:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$$

As seen in the formula, a state can be Markovian only if the next state is only dependent on the current state [17].

The next step towards an MDP is the Markov process (also called a Markov chain). A

Markov process is a random memoryless process that has the Markov property. A Markov chain is defined as the following tuple [17]:

$$\langle S, P \rangle$$

where S is the annotation for the finite set of states, and P is a state transition probability matrix. A state transition probability matrix defines the probability of transitioning from all states s to all following states s'. Silver gives the formal mathematical definition of a probability matrix as the following [17]:

$$P_{ss'} = P[S_{t+1} = s' | S_t = s]$$

The transition matrix P looks as follows. Each row in the matrix has a sum equal to one [17].

$$P = \begin{bmatrix} P_{11} & \dots & P_{1n} \\ \vdots & & \vdots \\ P_{n1} & \dots & P_{nn} \end{bmatrix}$$

Expanding further on the Markov chain and taking another step towards an MDP is the Markov reward process. The Markov reward process is annotated with the following four tuples [17].

$$\langle S, P, R, \gamma \rangle$$

The annotation for S and P is the same as in the Markov chain. In the four tuples, R is the annotation for the reward function and γ (gamma) is the annotation for the discount factor. The mathematical annotation for the reward function is the following [17]:

$$R_s = E[R_{t+1} | S_t = s]$$

The formula gives the expected immediate reward. It works in the following way: if at time t the current state is s, then at time $t+1$ the specified reward will be given. In other words, the formula gives the reward for being in a specific state at a specific moment. The discount factor or discount rate γ is used to avoid dealing with infinite returns that can happen if the Markov process contains a cycle, which means looping around the same cycle in the process and never reaching the end. The discount rate also gives the present value of future rewards in the return value [15]. The discount rate is always in the range $0 \leq \gamma \leq 1$. Different rewards can be obtained depending on how the discount factor is chosen. A discount factor closer to zero will lead to short-sightedness and greediness.

Immediate rewards will be preferred over later ones. If the value of the discount factor is closer to one, it will have the effect of far-sighted rewards being preferred, in other words not always taking the best reward with the hopes of receiving a better reward in the future [17].

As stated by Sutton et al. [15], the goal of an RL agent is to maximize the cumulative future reward. This notion can be represented in a formal way using return. The following formula is used for representing return [15].

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

In the above formula, G_t is the total discounted reward for the time step t. As stated above, γ (discount factor) then gives the value for the future rewards at time t. The reward received after $k+1$ steps is given by $\gamma^k R$ [15] [17].

The next step after the Markov reward process is the MDP. In an MDP, all states are Markovian. The formal definition of an MDP is the following [17]:

$$\langle S, A, P, R, \gamma \rangle$$

In the annotation, S is as before the finite set of states while A stands for the finite set of actions that can be taken. The state transition probability matrix is annotated with P, R is the reward function and γ is the discount factor. When adding the actions to the Markov reward process, the reward function also needs to be updated to include the actions. The reward function that also takes actions into consideration is defined in the following way [17]:

$$R_s^a = E[R_{t+1} | S_t = s, A_t = a]$$

The formula is the same as in the reward function for a Markov reward process, except the addition of the action. The addition of the action to the formula means the reward received from the function will also be dependent on the action taken in a state. The state transition probability matrix is also changed when adding an action to the Markov reward process. The probability of transitioning to the next state is now also dependent on the action taken in a given state. The formal definition for the state transition is then expressed in the following way [17]:

$$P_{ss'}^a = P[S_{t+1} = s' | S_t = s, A_t = a]$$

4.3.1 Bellman equation

Evaluating future states from the current state is also needed in RL to know the value of the current state. This can be done with the *Bellman equation*. The Bellman equation is, according to Sutton et al., used to show the relationship between a state's value and the values of states that come after this state. This works by taking a state and looking forward from that state and figuring out what future states can be reached from this state. This looking ahead also needs to take into consideration all the different actions an agent could choose to take in the various states with respect to the current policy used by the agent. Based on the actions, the agent could end up in very different states as well as the variance in the rewards received depending on the actions taken [15].

According to Sutton et al. the Bellman equation tackles all these branching possibilities by averaging all of them and weighting them by the probability they have of happening. This then forms the basis for calculating value functions. The Bellman function is therefore useful and used in numerous implementations of RL [15].

Sutton et al. annotate the Bellman equation in the following way [15]:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r + \gamma v_\pi(s') \right], \text{for all } s \in S$$

As seen in the formula, it expresses the values of a state given a policy. As discussed, the formula takes into account the possibilities of transitioning to different states. It also discounts all the future rewards with gamma [15]. The discounting of future values will be discussed later on in this chapter.

4.4 Elements of reinforcement learning

Sutton et al. define the components to create an RL system as follows: agent, environment, policy, reward, value function and, in some cases, a model of the environment [15]. Each of the components will be covered in greater detail below.

4.4.1 Agent and environment

Sutton et al. [15] define the agent as the part of the RL system that interacts with an environment in order to learn about the environment. This information is then used to learn how to accomplish a predetermined goal. To do this the agent needs to know something about the environment. It can either know the whole state of the environment or just a small part of it. The agent also needs to be capable of changing the state of the environment through actions. As previously mentioned, the agent tries to achieve a goal. This

goal is also needed by the agent since it guides its actions. The goal is usually to get the state of the environment to be a certain one. This is achieved by the agent changing the state with its actions until the state is the same as the goal state [15].

The term environment is quite broad, since it is considered to be everything that is outside of the agent. Essentially, the environment is the setting the agent is placed in and can change with its actions. The environment will respond to the agent's actions by giving the agent a new state and thereby a new situation. This interaction goes on continuously; when the agent is presented with a new state, it will again take an action and the environment responds with a new state. This can also be seen as the agent changing the environment, as presented in the previous section. The environment is also the basis of the rewards, since the rewards have their basis in what state the environment is in. Rewards will be presented later in this chapter. This series of actions and responses is illustrated by Sutton et al. (see Figure 4.1). The figure illustrates the agent taking an action at time t based on the state of the environment at time t and reward at time t. The environment responds to the agent's taken action by giving it a new state and reward at time $t+1$ [15].

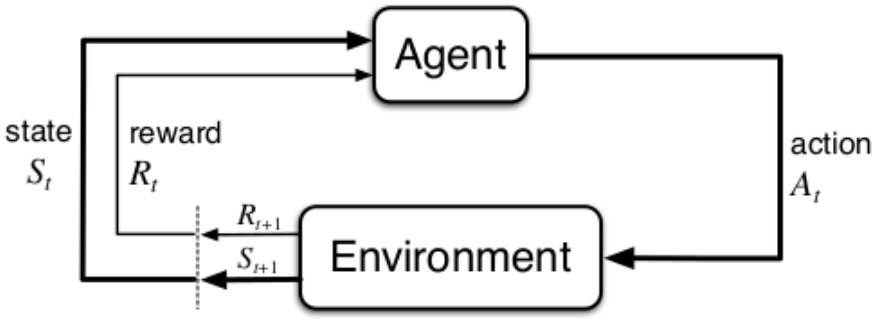


Figure 4.1: Sutton's et al. illustration of the agent and environment interactions [15].

4.4.2 Policy

According to Sutton et al. [15], the *policy* states how an RL agent will behave or act in a given state. In other words, the policy defines what action an agent will take in a specific state and therefore fully defines the behavior of an agent [17]. This is accomplished by maintaining a mapping. A policy maps states of the environment to actions the agent should take in those states [15]. To learn this policy or mapping is the goal of the agent. Silver presents a formal mathematical definition of a policy. A policy can be thought of as a distribution over actions given states, as seen in the formula [17].

$$\pi(a|s) = P[A_t = a|S_t = s]$$

The formula explains that a policy fully defines what action will be taken in which state. Additionally, it can be seen from the formula that a policy is independent of time and history, it only needs the current state [17].

An example of how a policy (π) works in the context of an MPD is the following, $M = \langle S, A, P, R, \gamma \rangle$ where π is given. If a sequence of states is given, for instance, $S_1, S_2 \ldots$, it is considered a Markov process and annotated the following way [17]:

$$\langle S, P^{\pi} \rangle$$

If a sequence of states and rewards $S_1, R_2, S_2, \ldots$ is given it is considered a Markov reward process and annotated in the following way [17]:

$$\langle S, P^{\pi}, R^{\pi}, \gamma \rangle \text{ where } P^{\pi}_{s,s'} = \sum_{a \in A} \pi(a|s) P^{a}_{s,s'} \text{ and } R^{\pi}_{s} = \sum_{a \in A} \pi(a|s) R^{a}_{s}$$

The first addition to the formula $P^{\pi}_{s,s'} = \sum_{a \in A} \pi(a|s) P^{a}_{s,s'}$ specifies that the probability of moving from a state to a following state is dependent on the policy and action taken to follow the policy. The second addition $R^{\pi}_{s} = \sum_{a \in A} \pi(a|s) R^{a}_{s}$ specifies that the reward gained is dependent on the policy and the action that will be taken when following that policy [17].

Sutton et al. [15] said that the objective of the agent was to learn a mapping or policy. That is true, but more precisely the policy that is the most interesting and sought after is the optimal policy. Silver [17] explains that the optimal policy is the policy that gives the optimal value function (discussed in the following section). Both Sutton et al. and Silver mention that an optimal policy defines a partial ordering over policies, as show in the following formula [17] [15].

$$\pi \geq \pi' \text{ if } v_{\pi}(s) \geq v_{\pi'}(s), \forall s$$

In the formula, π stands for a policy and $v_{\pi}(s)$ is the notation for the value of a state following a specific policy (the value function is explained in the following section) [17]. As seen in the formula, a policy is better or more optimal than another policy if the value of the policy is greater than the other policy in all states [17].

The key take-aways about optimal policies are mentioned by Sutton et al. [15] and summarized into a theorem by Silver [18]. The theorem states the following: there exits such an optimal policy π_* which is equal or better than all other policies. This is formally mathematically annotated in the following way. $\pi_* \geq \pi, \forall \pi$. The theorem also includes the notion that an optimal policy also achieves the optimal state value and optimal action-value functions. This is annotated in the following way, $v_{\pi *}(s) = V_*(s)$ and $q_{\pi *}(s,a) = q_*(s,a)$. Silver also explains how to find an optimal policy. An optimal policy can be

acquired by maximizing over $q_*(s,a)$ in the following way [17]:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in A} q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

The formula states that if we maximize over the action-value function, we will find the best policy when we have found the action that has the highest return [17].

4.4.3 Value function

A value function is the component of an RL system determining the desired outcome in the long run. This is achieved by defining values for states. The definition of a value is the expected accumulated reward if the agent were to start in that state. Hence, the value function will be able to show the long-term desired states by considering the most likely orders of states. It takes into account what states and expected rewards are most probable to follow a given state. This long-term notion of the value function is important since the reward (presented later) will only consider what is an immediately desired action. The value function will take into account that taking the greatest reward in every step may not lead to the overall greatest reward. The value function makes it possible to discover if there may be a state that always has a low reward associated with it, but following states may always yield high rewards or vice versa [15].

A formal mathematical way of expressing a value function is provided by Silver [17],

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

As seen in the formula, the value of a state is the expected return starting from a state s and following a policy π. Different actions also have a value associated with them. The value of a given action in a given state is defined by the action-value function. The function is annotated in the following way [17]:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$

As seen in the function, the action-value function is the expected return when starting from state s, taking action a and following policy π. As explained previously this function will tell the value of taking a specific action in a specific state. This allows actions to be ranked. It allows us to determine what action is more desired to take in this state to benefit in the long run [17].

In addition to the action-value function, there are also optimal value functions. The optimal value functions are the functions solving or having the optimal performance in

an MDP. There are two different value functions. The first function is the optimal state value function $v_*(s)$. The second one is the optimal action-value function $q_*(s,a)$. The definition for the optimal state-value function is the maximum value function that can be achieved from trying out all policies. The state-value function is annotated in the following way [17]:

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

The optimal action-value function, however, is the maximum action-value function that can be obtained from trying out all policies [17]. It can be given as follows:

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

The value function essentially makes sure the agent does not become greedy and consider all options. The best policy is not always to take everything now. Lower intermediate rewards are acceptable if they lead to higher rewards overall. An example to provide intuition on this would be the case of driving your car from point a to point b. The distance between the points require you to fuel your car along the way. The goal is to get there as fast as possible. The shorter the time is the higher the reward will be. Stopping for fuel will cost you time and give you an immediate lower reward. However, if you do not stop for fuel you will not make it there. Here the value function would make sure that the stop for fuel and an intermediate lower reward is accepted since it will lead to making it to point b and ultimately having a higher reward.

4.4.4 Reward

Sutton et al. explain the *reward* as being the RL agent's feedback for the actions it takes. The reward can be given at different times, either directly after a taken action or after each trial and error run. The agent tries to find the policy that will maximize this reward. Thus, the reward defines the goal for the RL agent to achieve. This is done by engineering a reward function to specify what are desired and undesired actions to be taken by the agent. Since the agent tries to find a policy to maximize the reward, it is also the primary source on information regarding changing a policy. The reward is the feedback on of the changes made to a policy were good or bad. The agent may have taken an action resulting in a low reward and, based on this, the policy can be changed and the agent will take a different action in the next iteration. Then in the next iteration taking a different may result in a higher reward and the agent knows the policy change was good [15].

4.4.5 Model

According to Sutton et al., a *model* of the environment can be a part of an RL system. It is not an essential part but a model can aid an RL system by representing the environment and allow assumptions to be made on the behavior of the environment. This is achieved by a model taking a state and an action and then predicting the following state and reward. This allows for planning. RL systems without a model are purely based on trial and error to explore the environment and are not able to plan. Utilizing a model and planning essentially means being able to consider what actions taken in the current state mean for future states [15].

4.5 Solving the RL problem

Theory and mathematical functions are good but do not provide much value unless they can be used and implemented into practical applications that solve problems. Implementing agents and finding solutions to the RL problem have had different approaches during the years. New methods and ways of tackling the problem have been introduced and often new methods expand on older ones for better solutions. In this section, two ways of solving the RL problem will be presented. The agent implemented in this thesis uses the lastly presented Deep Q-learning method.

4.5.1 Q-learning

Q-learning was introduced in 1989 by Christopher Watkins in his Ph.D. thesis titled "Learning from Delayed Rewards" [19]. In the thesis, Watkins presents a model-free RL type called *Q-learning*. In 1992, Watkins and Peter Dayan followed up the Ph.D. thesis. [20]. Q-learning is an easy way for agents to learn how to perform optimally in controlled Markovian domains. The learning process is done by letting the agent experience consequences of its actions. However, Q-learning does not require that the agents maintain maps of the domain they are located in [20]. Learning happens by the agent trying out all possible actions in all different states again and again. In all states, it evaluates all actions regarding which effects they have on the immediate reward received and also the value of the state the action is taken in. By the repeated evaluations of every action in every state, the agent is able to learn which actions are the best overall. How good an action is or the value of an action is judged by the long-term discounted reward expected to be achieved when taking that action [20].

4.5.2 Deep Q-learning

In 2015, the paper "Human-level control through deep reinforcement learning" [18] by Mnih et al. was published. It introduced a new way to tackle RL problems. A new algorithm was developed that was able to solve different and varying difficult tasks. The result was a new RL agent known as a *Deep Q-Network* (DQN). The new agent combined traditional RL with deep neural networks and overcame issues known to occur when having a neural network approximate Q-values.

The approach started with the regular premiss of an RL system, having an agent interact with an environment to explore the environment, take actions that change the environment, and receive rewards. The goal of the agent was also, as usual, to find a policy that maximizes the expected cumulative future reward. The new thing with Deep Q-learning is that it is able to approximate the action-value function (Q-function) non-linearly, essentially meaning that a neural network can approximate the Q-function. The action values are known to diverge and be unstable if nonlinear approximations (neural network approximation) are used. The reasons behind these challenges are the correlations between sequences of the agent's observations and a policy can change drastically even for small changes to the action-value function. The authors combatted the known issues by introducing *experience replay* and periodic iterative updates to the parameters of the neural network [18].

Experience replay is a method drawing inspiration from biology, namely, the replay of past experiences that happens during sleep in actual brains (dreaming). Experience replay randomizes data to significantly reduce or remove the existing correlation between an agent's subsequent observations of the environment. This works by saving the agent's observations from every time step in a data set. Then, during training when the Q-learning updates are done, the updates are done on uniformly randomly selected samples from the agent's past experiences [18]. This meaning that the agent will still learn but subsequent observations are not affecting each other.

The periodic updates to the parameters of the neural network, according to the authors, also reduce correlation. This is done by fixing the target neural network parameters between updates and only updating the target network parameters with the Q-network parameters after a specified number of iterations. Doing this allows approximation of the Q-value with a larger neural network and, hence, increasing the complexity of tasks an RL agent can solve [18].

5. Proof of concept agent

The *Simple Ship Sim* program is meant to serve as a training and testing environment for RL agents and algorithms. A user can create his own RL agent or algorithm for solving maritime-related challenges. This user-supplied agent would be using the included environment for training. Testing of the user agent would be done using the simulator. The program is built as a larger project with a team. The author's responsibility in the development is everything RL related and an initial user agent to be placed in the simulator. The requirements set on the initial user agent are that it is able to follow the COLREG rules in the basic head-on, overtaking, and overtaking scenarios. Additionally, the agent needs to be able to safely navigate in a multi-vessel scenario.

This chapter will cover the author's practical part of the project. The initial project design was given as a template by Ivan Porres. The author further refined this design and implemented it to work with RL. The author developed the custom environment, RL agent, main command-line interface, configuration files and *ShipAgent* module. The modules *helpers* and *ship* were developed together with Kim Hupponen [8] from the template given by Porres. Hupponen implemented the *sim* and *ui* modules. In order to understand the implemented proof of concept agent the context has to be explained. Hence, this chapter also contains the parts that the author developed together with Kim Hupponen [8] and parts developed solely by Hupponen. Ivan Porres, in addition to the template, also provided guidance and comments on the code written for the project.

The chapter will first cover the goals to be achieved, then the structure of the program is explained, and following this, the libraries used in the implementation are covered, then the simulator is explained for context and, lastly, how to use the simulator is outlined. Since the proof of concept agent is a part of the simulator, the agent itself will be covered in the explanation of the simulator.

5.1 Goal

The goal the author set out to reach was defined as making a proof of concept solution investigating if it is possible to teach RL agents to follow COLREG rules. The different scenarios chosen were an overtaking of a slower ship, a crossing from right and left, a head-on scenario, and a multiple vessel scenario. The goal would be considered achieved

if the RL agent follows the COLREG rules in these scenarios and is able to compromise on the rules, if necessary. This created agent would then serve as an initial agent for the *Simple Ship Sim* program.

5.2 Program structure

The initial structure of the simulator was made by Ivan Porres, as a starting point. The program structure was then further expanded by the author to reduce code duplication, increase maintainability, and easy extensions of the program. The program contains the following modules. The initially given basis contained the following modules; agent, ship, sim (simulation), and UI (user interface). The initial interaction between the modules was drawn up in the code and implemented in a bare-bones way. The environment for training agents, implemented agents, neural network models, a helpers module, main command-line interface, and the use of configuration files were added during the development. All modules will be described in greater detail below.

The entry point for the program is the main module, as seen in Figure 5.1. It provides a command-line interface for running the program. It follows the standard unix flags and parameters for passing arguments to a program. The agent module is the home for all implemented agents. Here the agents take decisions using the pre-trained neural network and call the environment module to make the actions happen. The simulator (*Sim*) module is responsible for translating desired actions into actual actions in the simulator, as seen in Figure 5.1. The simulator module also handles the physics that concerns vessels, for instance, how tight should the turning circle of a vessel be. The UI or user interface is responsible for rendering the user interface, for instance, it renders the map, ships, and buttons. The config module seen in Figure 5.1 is responsible for loading configuration files and passing on the information to the main and ship module. The helper module is a central location for sharing code between the modules. Among other things, it provides information about ships to the user interface and aids with calculations needed in the simulation, as seen in Figure 5.1.

5.3 Libraries

Libraries make it easier and faster to implement solutions since all code does not have to be rewritten by the programmer. Libraries are also more extensively tested and are used by many people. This means that a great deal of the bugs in the library has already been found and corrected, compared to a self-made implementation of a similar feature set as offered by a library. Hence, industry-standard libraries are used in the implementation of

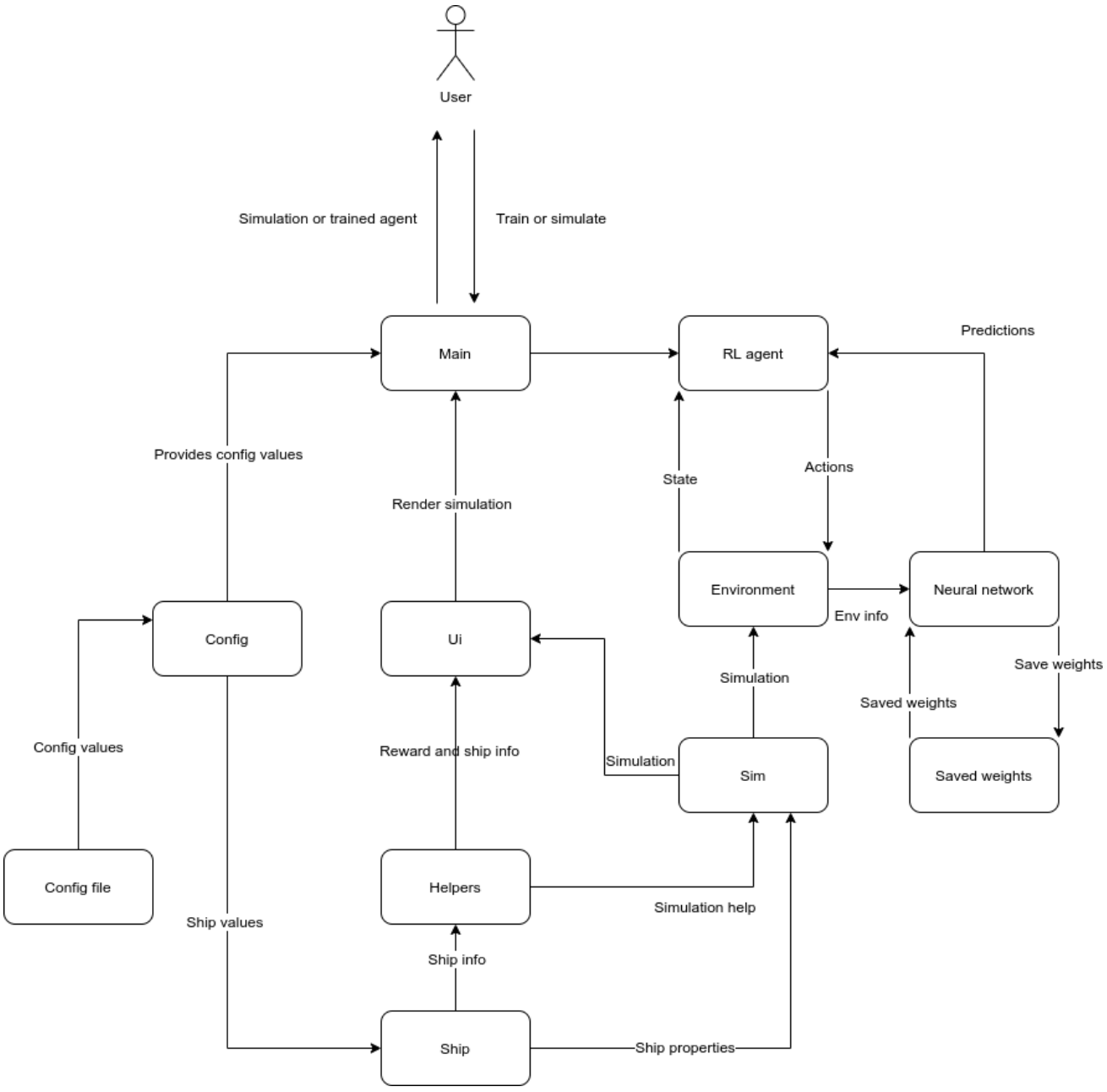


Figure 5.1: Program structure

the deep-learning parts in the case study. To train the neural network *Tensorflow* [4] was chosen. To create and specify the structure of the neural network *Keras* [5] was used. It is difficult to deal with multidimensionality without *NumPy* [21] and hence it was included. Since the developed agent is the main focus of the thesis, the author implementing all of the mathematics and basic functionality of an RL agent is not very productive. Hence, a library with a predefined agent was chosen, namely *Keras-rl* [6]. In the library, all the basics are taken care of and the user can focus on optimizing their agent to solve a task. The agent needed an environment specifically designed for maritime scenarios. Since there was no such environment a custom one had to be made by the author. For making RL agent environments, the industry standard is *Open Ai Gym*. Building a user interface from the ground up is not the point of the thesis and very time-consuming. Hence, a library offering easy creation of user interfaces was needed and the choice fell on *PySide2* [22].

5.3.1 Tensorflow

Tensorflow is an open-source library originally built by Google to be used internally by Google. In 2015 Google, however, released Tensorflow as open-source. Tensorflow is meant to help with development and training of machine-learning models [4] [23].

Tensorflow was chosen for this project since it is one of the industry standards. Furthermore, Tensorflow comes with good documentation and tutorials making it easy to use. Tensorflow was also chosen since it goes well together with the Keras API and together they make a powerful package.

5.3.2 Keras

Keras is an open-source neural network application programming interface (API) built in Python. Keras is able to utilize different back-ends such as Tensorflow or Theano. The main focus of Keras and why it was built is to make fast experimentation possible. To achieve this; fast and simple prototyping, support for different neural networks, and running on both GPUs and CPUs are features of Keras [5] [24].

Keras was chosen for the project since it is the current industry-standard and has useful documentation and user guides, making it easy to debug and use, reducing the time it takes to prototype and try different neural networks.

5.3.3 Keras-rl

Keras-rl is an open-source library providing deep reinforcement-learning algorithms implemented in Python. The main feature of Keras-rl is already present in the name. It

provides easy integration with the Keras library. The other main feature of Keras-rl is that it is compatible with the Open Ai Gym library. The Keras-rl library, hence, makes it easy and fast to implement RL algorithms. Keras-rl is also built to be extendable by users making the library suit their needs [6]. The Keras-rl library was chosen because of its ease of use and easy integration with the rest of the tech stack.

5.3.4 NumPy

NumPy is a Python library developed to be used for scientific computing and has become the industry standard. One of its main features is its support of large and powerful N-dimensional array objects. NumPy also provides easy-to-use functions to reshape and work with these arrays [21]. The N-dimensional array objects are the reason for including the library in the project. The NumPy arrays are needed due to the multidimensionality of the inputs to the neural network. Without using NumPy, it would be more difficult to deal with this multidimensionality.

5.3.5 PySide 2

PySide2 is the name of the library module for the *Qt5* framework for Python. The library is used to develop user interfaces by providing ready-made user interface elements. The style of the elements lies toward classic desktop applications [22]. The already made user interface elements are the primary cause for the use of the *PySide2* module in the project. It would take significantly longer to develop the same user interface elements again compared to using the elements provided with the library. Now the implementation just needs to specify what elements should be included and define the layout and content to be displayed by the elements.

5.3.6 Open Ai Gym

Open Ai Gym is a library and toolkit used to compare and develop RL algorithms. The Gym library itself is more a collection of pre-made test problems, also called environments. Gym was mainly built to tackle, according to Open AI, the main problems in the RL field. The problems are in their opinion the lack of good benchmarks and not enough standardization of the environments used in scientific publications [7].

Open Ai Gym was chosen because of its industry standard and ease of use. There are many papers and documentation surrounding it. Open Ai Gym also works well with Tensorflow. Open Ai Gym also allows its users to build their own environments. This use case was needed for the thesis, since there were no already made maritime or ship environments.

5.4 Simple Ship Sim modules

This section outlines the different modules of the implemented system and the functions they serve. This section begins with the custom *Open Ai Gym environment*, after this the RL agent's structure is explained, following this the different ship agents and the configuration loading is described. After this, the *helper* module is outlined, and following this, the simulation module and the user interface are explained. Lastly, the ship object and the main module are described.

5.4.1 Open Ai Gym environment

The custom Open Ai Gym environment follows the basic outline for implementing a Gym environment. The methods implemented are the *init* method to create the Gym object, a *step* method to take steps in the environment, a *reset* method to reset the environment after an episode and a *render* method to visually show the agent training. Each of the methods and their use in the environment will be covered in greater detail below.

This *init* method is the standard constructor for creating an object in Python. The method can be found in listing 5.1. The constructor in the environment initializes the variables that will be used later. The most important part is the creation of the action and observation space. The action space in this environment is discrete or continuous. The action space is initialized using the *spaces* method included in the Gym library. The discrete implementation contains 15 different actions the agent can take. This means that the rudder's potential degree values have been split up to make it possible to set the rudder in discrete increments. The environment also includes an option to create a continuous action space. This is needed if a user wants to implement a continuous agent. The difference between the discrete and continuous environment is that the actions are not discretized. In the continuous environment, an action corresponds directly to a rudder degree. The action itself is the rudder degree to be set. Hence, the actions can take on values between -35 and 35. A continuous agent was not developed for the proof of concept. However, for the generality of the environment the author felt it was necessary to give a potential end user the option to develop a continuous agent.

The observation space is initialized with the minimum and maximum values observations of the environment can take. In this environment, the returned observations are comprised of a NumPy array with a size of seven. The returned observations contain the following information from the environment. The first value in the array is the x coordinate of the ship. The x coordinate can take values from all the navigable space. In this implementation, the x coordinate was chosen to have a minimum value of 0 and a maximum value of 9400. The second observation value is the y coordinate of the ship. It

takes on the values from 0 to 6100 which also corresponds to the navigable area of a ship. The third value in an observation is the bearing of the ship in radians. The observation corresponds to the current bearing of the ship in the environment. The fourth value in an observation is the speed of the ship in knots, taking on values in the range of 0 to 15.6 knots. The numbers are taken from the specifications of a typical oil tanker. The fifth value in an observation is the current degrees of the rudder. This can take on values from -35 to 35 in accordance with typical ship specifications. The sixth and seventh values in the observation are the distance to the closest vessel and its speed. The distance to the closest vessel takes on values between 0 and 9000. The speed of the closest vessel takes on values between 0 and 15.6.

In the constructor for the environment, a config file is also loaded. The config file provides the user with an easy way of specifying his own scenarios in the environment. The config file is in a Yaml [25] format further increasing ease of use. In the config file, a user can specify the attributes of the ships he wishes to include and their starting positions in the environment.

The step method in the environment is responsible for taking steps in the environment and returning observations, also this method is found in listing 5.1. The step method takes an action as its argument. The step methods first check if any conditions to end an episode are met. If there are no conditions for determining whether an episode is over, the environment would loop forever and the agent would not receive the necessary feedback it needs in order to learn. The conditions for termination in this environment is if the agent has collided, sailed out of bounds or reached its destination. The environment step method also calls the simulation object's step method to advance the simulation. In the environment's step method, the action passed to the method in the arguments is also taken. This is done by calling the helpers module and the *take_action* method. After taking an action, the agent needs to know the quality or how good that action was to take in that particular state. Hence, the *get_reward* method from helpers is called and returns the reward. Both the *take_action* method and reward function will be covered in greater detail below. Lastly, since the step method needs to return an observation to the agent it calls the *get_state* method in the helper module. This is needed since the action taken by the agent has changed the state. The step method lastly returns the following four tuples: the first value is the state, the second value is the reward, the third value is the boolean variable done used to know if the episode is over and, lastly, it returns the info dictionary. In the info dictionary, the developer of the environment can specify additional information that might be useful for an end user to know. In this environment the info dictionary is not in use, however, it is still an expected return parameter and, hence, it is included.

Taking a step in the environment is the same as transitioning from one state to another

in an MDP. The agent starts in a state and takes the action it deems to be the best. The decision is based on the discounted value of the state, as discussed in the theory. This is the three tuples S, A, and γ of an MDP. In what state the agent will end up as a result of the action is dependent on the state transition probability matrix. The probabilities are not visible to the user of the system and depend on the Open Ai Gym environment. In the case of this environment, the probability of ending up in a state is always the same. There is no branching. The same action taken in the same state will always result in the same next state. The part missing from a complete MDP is the reward and it will be covered later in this chapter.

The *reset* method is used to reset the environment after a finished episode run. In the reset method, the environment is reset into its starting values as defined in the config file and in the constructor. Additionally, to increase the generality of the agent it is placed in a new scenario. This naturally increases the exploration of the environment, since the agent is exposed to more different scenarios. The assigning of a new scenario is done randomly.

Render is the method called if one wishes to visualize the environment during training. This can be helpful to see what the agent is actually doing in the environment during training. The main drawback of doing so is that the training is significantly slowed down. The render function is not in heavy use in the developed environment, since a completed training can just as easily be rendered in the simulator.

```
def __init__(self, isDiscrete):
    self.world = None
    self.AgentVessel = None
    self.state = None
    self.isDiscrete = isDiscrete
    self.config = None

    if self.isDiscrete:
        self.action_space = gym.spaces.Discrete(15)

    else:
        self.action_space = gym.spaces.Box(
            low=-35, high=35, shape=(1,), dtype=np.float32
        )

    self.observation_space = gym.spaces.Box(
        np.array(
            [[0, 0, -1.3, 0, -35, 0, 0]]
        ),  # Min observations for x,y,bearing,speed,rudder, distance to
```

```
            land, distance to closest other vessel, other vessels speed
        np.array(
            [[9400.0, 6100, 6.5, 15.6, 35, 9000, 15.6]]
        ),  # Max possible observations
        dtype=np.float32,
    )

    self.config = config.load()["scenarios"]

def step(self, action):
    done = False
    info = dict()

    # Episode over if there has been a crash
    if self.AgentVessel.collision == True:
        done = True

    if self.AgentVessel.x > 9400 or self.AgentVessel.x < 0:
        done = True

    if self.AgentVessel.y > 6100 or self.AgentVessel.y < 0:
        done = True

    if self.AgentVessel.done:
        done = True

    self.world.step(1000)
    helpers.take_action(action, self.AgentVessel, self.isDiscrete)

    reward = helpers.get_reward(self.AgentVessel, self.world.ships)

    self.state = helpers.get_state(self.AgentVessel, self.world.ships,
        False)

    return self.state, reward, done, info
```

Listing 5.1: Environment code

5.4.2 RL agent

To make the vessels function autonomously they need an agent working as the brains and making decisions. The RL agent was implemented using the *Keras-rl* library and the *DQN* method previously discussed. The library was used since it contained already implemented and tested RL algorithms. This significantly increases the quality of the software, since the library is more mature and less likely to contain bugs compared to the author implementing the same functionality. The use of the *Keras-rl* library also shortened the development time. The library is also used since the mathematics discussed earlier are already implemented. The user does not need to implement all the basic mathematics.

Using the library is easy, as seen in listing 5.2. First, an environment is supplied, the environment is then used to obtain the action and observation spaces. These are needed when creating the neural network. The action space is needed for the neural network's output layer. Since the neural network essentially creates a mapping from the state or observations to an action, it needs to know how many outputs it should map to. The observations are needed as the input. The network is built in a separate module to decrease code duplication. The network is needed both in the training and simulation and, therefore, sharing of the code is needed. When the neural network is created the memory for storing the observations is defined. This is the memory utilized by the experience replay. The limit is the maximal number of episodes to store. The window length is the number of entries from the memory to be returned when querying the memory for experience replay.

When the memory is created the next thing to define is the exploration policy to be used. As seen in listing 5.2, the author chose to go with the Boltzmann-Gibbs policy. The *Keras-rl* library calls this method the BoltzmannQPolicy. Naming aside, Wiering [26] explains that the Boltzmann-Gibbs exploration policy works by looking at different actions and their Q-values. Then, based on the Q-values, it will assign the actions different probabilities. This essentially means that the method will give a probability of taking an action in a state. The policy will work so that the agent will explore more when states have Q-values that are very close to each other and explore less when the Q-values differ more. The agent will not explore much in areas that seem to be worse (there is a clearly higher Q-value present). Exploring in directions that look terrible is still, however, needed since wrong Q-values could be introduced by noise [26]. The author chose the Boltzmann-Gibbs method since it is widely used in the field of RL. The method is not without critique and may not always guarantee optimal behavior, as shown by Cesa-Bianchi et al. [27]. Despite this, the method worked well for the proof of concept.

After the exploration policy is set the agent is defined. The *Keras-rl* library provides a *DQNAgent*. The agent needs to know what neural network model to use, the number of possible actions, the amount of memory, the number of warmup steps to do, target

model update and, lastly, it needs the exploration policy. The model parameter for the *DQNAgent* is the neural network mentioned before. The number of actions parameters is the number of actions possible to take in the environment. The agent needs to know the number of possible actions, since they are used to know the shape of the Q-values. Basically, the agent needs to know what actions it can take. The memory parameter is the memory defined before. The number of warmup steps just means that the model will use a smaller learning rate during this interval. This is done to reduce overfitting on early observations. The target model update comes directly from the specification of a deep Q-learning agent [18] as presented before in the thesis. The parameter sets how often the weights in the neural network should be updated. Lastly, the policy is the exploration policy defined before.

When the agent is defined, it needs to be compiled before use. In the compilation, the *Adam* optimizer is used with the learning rate set to 0.001. The error metric is also set in the compilation, in this agent mean absolute error is used. When the agent is compiled it can be trained. The training happens with the use of the *fit* method, as seen in listing 5.2. The *fit* method will return a *Keras* history object that contains information about loss, reward, and other information the user might want to have access to in order to evaluate the training. The training is done by passing the environment, the number of steps, a visualize parameter, and setting the printout parameter. The environment is passed to the fit method since it is needed for the agent to train in. From the environment, the agent among other things gets the reward and state, as discussed before in the thesis. The number of steps parameter is used to set how many steps in the environment the agent will take before stopping training. The visualize parameter defines if the environment will be shown while the agent is training. Showing the agent's actions as they happen in the environment can be a useful tool for understanding the agent better, but visualizing will slow down the training significantly. The verbose parameter is used to define in what manner the method will report back the progress of the training. When the training is done the neural network weights need to be saved, otherwise, the training will be lost. This means that a simulation could not be run without training the agent again. Saving the weights means that they can be used again and will save a significant amount of time when training only has to be done once. If anything is changed in the agent, the training needs to be done again and the weights also need to be saved again.

```
def train(numberOfSteps):
    env = gym_env.ShipGym(True)

    # Get the possible actions and the shape of the observation space
    actions = env.action_space.n
    observations = env.observation_space.shape

    model = DQNModel.dqn(actions, observations)
    memory = SequentialMemory(limit=2000, window_length=1)
    policy = BoltzmannQPolicy()
    dqn = DQNAgent(
        model=model,
        nb_actions=actions,
        memory=memory,
        nb_steps_warmup=10000,
        target_model_update=0.01,
        policy=policy,
    )

    dqn.compile(Adam(lr=0.001), metrics=["mae"])

    hist = dqn.fit(env, nb_steps=numberOfSteps, visualize=False, verbose=1)
    dqn.save_weights(
        "Weights/DQN/Weights" + str(numberOfSteps) + ".h5f", overwrite=True
    )
```

Listing 5.2: Code for training of the RL agent

5.4.3 Ship agents

The program contains different ship agents with various behavior. The agents are used when running a simulation to show the agent's actually learned policies. All agents trained implement the abstract *ShipAgent* class. The abstract class is simple and straight forward, only requiring the implementation of a step method. All agents are hence easily extendable and all follow the same structure, making it a simple task to swap out the agent to run in the simulator. The implemented ship agents are a basic agent, a DQN-agent, and a semi-random DQN-agent. Each of the different agents will be described in greater detail below.

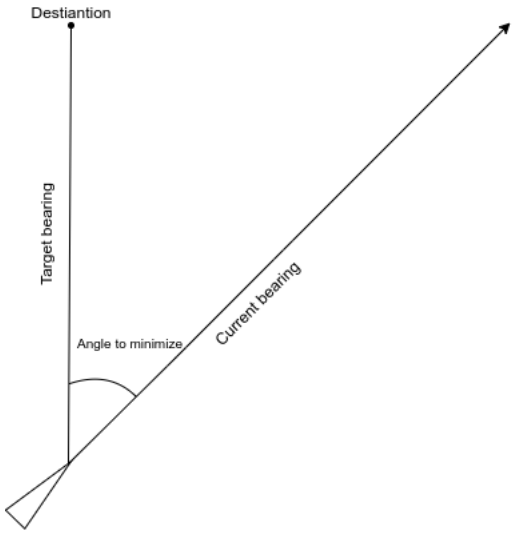


Figure 5.2: Target bearing vs current bearing

5.4.3.1 Basic agent

The first implemented agent is a *basic agent*. The basic agent is not implemented with any RL. Despite the lack of RL implementation, the agent is important to describe. The basic agent is used as the controller of environment ships that are supposed to be the stand-on vessel in the COLREG scenarios. The basic agent works by using the built-in *atan2* Python function to calculate the difference in the target bearing compared to the ship's current bearing. The basic agent will then adjust the ship's bearing until the difference between the target bearing and the current bearing is zero. This approach is simple and will take the shortest direct route towards the target destination. It will not consider obstacles or ships in the way. The *atan2* function is used since the degrees between two points in a 2D-plane are of interest. This is illustrated in Figure 5.2. In this case, the basic agent will adjust the course towards its port side to decrease the angle. This easy approach works quite well if the agent is given ample time to adjust the course.

The problem with this approach is that the agent may never reach the destination point and end up circling it, as shown in Figure 5.3. This happens if the distance between the ship and destination is short and the angle to reduce is large. Once the agent has made the course changes, the ship will have moved forward and the course adjustment will end up not being enough. The agent adjusts again and will fail to change course fast enough and the ship will have moved forward and the adjustment will not be sufficient. The distance and angle for when this happens are different depending on the vessel. The factor determining if this will happen is the turning radius of the ship. The circle the vessel will end up making around the destination will be equal to the radius of the ship, meaning the ship cannot turn steeply enough to end up at the destination.

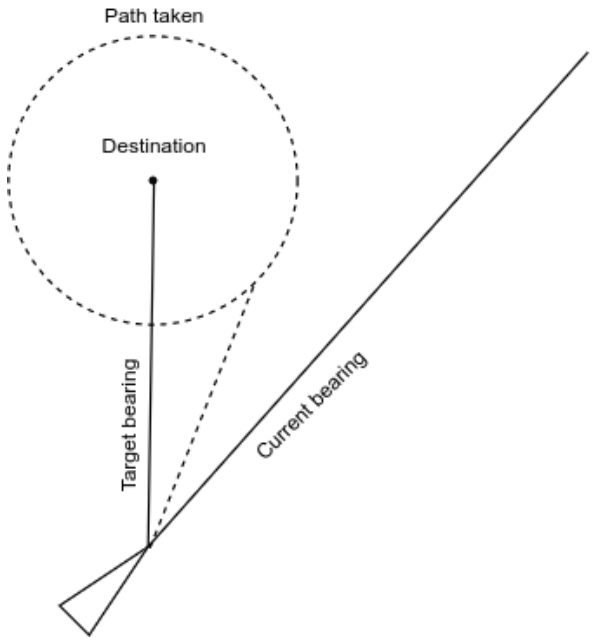


Figure 5.3: Basic agent shortcomings

5.4.3.2 DQN-Agent

The second implemented ship agent is the *DQN-Agent*. It is based on RL and is the main agent of the thesis. The agent needs to be trained before it can be used. This is since the agent needs to have saved weights to load the neural network with. These weights are obtained from previous training on scenarios and the agent will use the learned policies from training and use them in simulation runs. In addition to the neural network weights, the agent also needs an instance of the custom ship environment. This is since the agent needs to know the action and observation spaces so it can load the weights correctly. Loading the weights essentially means building the same neural network with the same weights as achieved at the end of training. The weights to be loaded can be declared by the user from the command line when he starts a simulation. If no weights are given by the user, the agent will use the default weights that come with the simulator. Since the *DQN-Agent* extends the *ShipAgent* class, the *DQN-Agent* also needs to have a step method. The step method for the *DQN-Agent* is straightforward and simple. It uses the *get_state* method form the helpers to obtain the current state of the environment. When it has the current state, it predicts the following best action to take in that state. To do this it uses the state as the input to the neural network and receives an action back. It then uses the *take_action* function from the helpers to take the action and progress in the simulation. The *DQN-Agent* also has a simple *string* method to ease the printout in the user interface of what agent is used.

5.4.3.3 Semi-random DQN-Agent

The agent *SemiRandomDQNAgent* is an additional agent to the simulator. There was a need to implement some kind of unpredictable behavior element in the simulation. This is the case since the behavior of the regular *DQN-Agent* is predictable. It will always take the same action in the same state with the same weights. The unpredictable behavior would in a way represent a human captain making a mistake or disregarding the COLREG rules. This would be needed for testing how robust the agent *DQN-Agent* is towards unexpected situations that may arise. This is a good testing feature, since the *DQN-Agent* agent has not encountered such situations in training. Testing with the agent *SemiRandomAgent* was left out from the result, since it goes out of scope for the thesis but is grounds for further research and investigation.

The implementation of *SemiRandomDQNAgent* is similar to the implementation of the agent *DQN-Agent*. *SemiRandomDQNAgent* extends *DQN-Agent* to have access to the same loading of weights without code duplication. The difference is that *SemiRandomDQNAgent* has a random component added. To implement this the step method from *DQN-Agent* is overridden in *SemiRandomDQNAgent*. In its current implementation, *SemiRandomDQNAgent* will take a random action instead of the neural network predicted one in 10% of actions taken. The number of random actions taken can be tweaked not to be such a high number. It is unlikely that a human would make mistakes that often. A single wrong action taken would not affect the outcome of a situation that much. This is since an action is taken at every increment of the simulator and the increments of the simulator are small. The agent would simply make the correct action in the increment and the wrong action would not be noted. To change this and simulate more of a real mistake the wrong action will be taken 100 times in succession by *SemiRandomDQNAgent*. The random element is implemented with Python's included *randint* function from the *random* module. In the agent *SemiRandomDQNAgent* implementation an integer between one and ten (with one and ten included) is selected at random. If the random integer is one, then the agent will again choose a random integer in the range of the possible actions and take it. If the random integer is not equal to one, *SemiRandomDQNAgent* will function in the same way as the agent *DQNAgent* and take the predicted action. *SemiRandomDQNAgent* also includes the *to string* method for easy displaying of the agent's name in the user interface.

5.4.4 Configuration files

For easy extension and creation of different scenarios, the author chose to implement configuration files. The configuration files are specified in the Yaml [25] format. The use

of Yaml is based on the ease of use and the fact that Yaml files are easily readable by humans. This makes the process of creating the configuration files much easier for the user. The user can create separate training scenarios and separate simulation scenarios. This process is better, since the agent can then be placed in a previously unencountered scenario and the agent's ability to generalize can be assessed. The sample training file contains the different COLREG scenarios previously decided to be included in the proof of concept.

The configuration files follow the structure seen in listing 5.3. First, the user starts with the *scenarios* key, under this key all the different scenarios to be used in the training or simulation is defined. When the top key is defined, the user lists the different scenarios located under it, for instance, *CrossingLeft*. When a scenario is named, it is used as the key for that specific scenario. Under this key, the user then specifies the *ships* key that, in turn, contains all the different ships in that scenario. Inside the key *ships*, the user then lists all vessels to be included, for instance *vessel1*. Then, using this key, the user defines all the ship properties he wishes to include.

In the configuration file, the user also specifies what map he wants the ships to sail on. If no map is wanted, the user can also specify land coordinates to be inserted in the simulation. If the user chooses, he can also insert additional land points to a map.

```
scenarios:
    CrossingLeft:
        ships:
            vessel1:
                id: "UR"
                agent: "basic"
                position:
                    x: 5118.00
                    y: 5963.00
                speed: 3.00
                bearing: 2
                mass: 90000000.0
                length: 250.0
                width: 42
                max_speed: 8.03
                tactical_diameter: 1197.9
                rudder_time: 11.8
                rudder_area: 16
                rudder_coefficient: 1.1
                destination:
```

```
                x: 4150.00
                y: 9000.00
map:
    path: "maps/turku.png"
    width: 800
    height: 600
    scale: 0.22
```

Listing 5.3: Example of Yaml configuration file

5.4.5 Helper module

The helper module was added to the program to reduce code duplication. The environment, simulator, and user interface in many cases needed access to the same information. Instead of duplicating code, the *helpers* module was created. This addition naturally increases the maintainability of the project significantly.

The helper *config* was created to allow the user of the program to easily define scenarios in config files. The *config* module was then created to have a central location for loading config files. As previously mentioned, Yaml was chosen as the format for the config files.

5.4.5.1 Distance modules

The general helper module contains a lot of methods to ease the development of the program. The first method specified in the module is the *distanceTo* method. The method is used in the program to calculate the distance between two points in the world using the Pythagorean theorem. The Pythagorean theorem is $a^2 + b^2 = c^2$ [28]. To use the theorem for calculating distances in a 2D-plane the formula is $distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ [29]. This distance calculation is, for instance, used in the program to calculate the distance from the ship's current position to its target destination.

The next helper method is an extension of the previous method and is called *distance_to_closest_vessel*. The method's purpose is, as specified in the name, to give the distance to the closest vessel from the ego vessel. In addition to the closest distance, the method also has an option to obtain the id of the closest ship. This is useful for finding out additional attributes of the closest ship, as seen in later methods.

Not only the distance to other ships in the simulation is important to know. The distance to land is also important. The main use for the *distance_to_land* method is to know if there has been a collision with land or a collision is about to occur. Some versions of the agent also used the distance to the closest land point in their state. The idea behind

this is that the agent needs to know if it is approaching land or not. In later versions of the agent, the distance to land was removed. Since the agent's performance without it was better. This, however, resulted in the drawback of the agents not knowing where land is on the map. This drawback was acceptable, since collision avoidance in more open waters with ships cluttering the waters was more interesting to optimize for. Hence, collision avoidance with static objects, such as landmasses, was left out.

5.4.5.2 Bearing modules

The *adjust_bearing* method is also located in the general helpers module. It handles the changing of a ship's bearing using the calculations described above in the basic agent.

To identify the different COLREG scenarios the relative bearing between vessels is also needed. To know this the *ships_relative_angle* method was added to the helpers module. The method works by first finding out the id of the closest ship using the above mentioned optional return of the id for the closest distance to a vessel. When the id is known, it is just a matter of looping the ship objects and comparing which one of them has the id in question. When the correct ship object has been found, its object representation can be used to obtain the x and y coordinates needed in the *atan2* calculation to obtain the relative angle between the vessels. The coordinates used in the calculation is the ego ships x and y coordinates and the nearest ships x and y coordinates. The Python math library implementation of the *atan2* method returns the angle in radians. The angle is then converted to degrees using the built-in conversion method in Python's *math* library.

5.4.5.3 Modules for faster prototyping

The method *take_action* also needed to be shared. It became significantly easier to prototype and try different actions the RL agent is able to take when the changes only had to be made in a single place. The implementation of how actions are taken depends on the type of agent that is used. What all agent types have in common is that the actions they can take only affect the angle of the rudder. In the case of a continuous agent the action passed to the method will correspond to a rudder degree. The actions are also trimmed to be inside the allowed ranges of the rudder when dealing with continuous agents. In the case of a discrete agent, actions are discretized into five-degree segments inside the allowed rudder range. This means that the action passed to the method will be mapped to a degree of rudder turn and not used directly. The actions passed from the agent are also not taken into consideration before the "danger zone" is reached. In the implementation, this happens when the distance to another vessel is less than 550 units. When the ship is in this state of higher risk of collision, the autopilot is switched off and the ship goes into collision avoidance mode. Then the agent is free to navigate the ship in order to avoid a

collision. The autopilot is given back control of the vessel again when the distance between the vessels become large enough again or the ships have passed each other and the dangerous or risky situation is over.

Also the code for the reward function is shared. The reward function is implemented in the method *get_reward* (See listing 5.4). The current reward needed to be displayed in the user interface of the program and sharing the code between the environment and the user interface made most sense. The reward function does as previously mentioned guide the agent's behavior. The amount of reward to be received for different states was mostly found through trial and error. There are most likely also other rewards that would produce the same behavior. The resulting final reward function can be found in listing 5.4.

The agent should receive a large negative reward or penalty for colliding or crashing, since this is the worst-case scenario and should at all cost be avoided. The check to see if the agent has been in a collision is the first *if* clause in the reward function, as seen in listing 5.4. It does a simple comparison to see if the collision attribute of a ship is true or not. If it is true the ship has collided and the negative reward of -10000 should be added to the reward. The next part of the reward function guides the agent to make decisions to close the distance towards the destination. Even in a collision avoidance situation, the agent should favor actions that close the gap towards the destination. For instance, the agent should favor an evasive manoeuver allowing it to still move towards the destination instead of turning the ship around. The code for this is found directly after the first *if* clause. The negated distance to the destination is added to the reward. This means that the closer to the destination the agent comes the higher the reward will be since the reward value will then increase towards zero, reducing the negative reward.

Following this, the reward function handles the desired behavior of not getting too close to other vessels. The further away from another vessel the agent can keep the ship while avoiding a collision the lesser the risk of collision. It is, according to the author, more likely that a collision will take place if the ships are just a few meters apart compared to a few hundred meters. A threshold value of 200 units was therefore added. The distance between the vessels needs to be lower than this before a negative reward is given to the agent. In other words, the agent will be punished for sailing to close to other vessels. The code for this is found in the second *if* clause in listing 5.4. If the distance to the closest vessel from the agent ship is below 200 units, a negative reward of 1000 will be given to the agent. The reward function can also give out a positive reward. The reward function does this, if the agent gets to the specified destination. If this happens, a positive reward of 1000 will be given to the agent. The code for this is found in the third *if* clause in listing 5.4. The *if* clause checks if the *done* attribute of the ships is true or false.

The next part of the reward function guides the agent to follow the COLREG rules.

This is achieved by punishing the agent if the correct evasive action has not been taken in the scenario. The scenarios are recognized using the relative angle of approach. The different COLREG scenarios can be identified from this angle. The code for this is found in if clauses four to seven in listing 5.4. The fourth if clause handles the reward for being in the overtaking or head-on scenario with the agent ship approaching from south towards north. These scenarios have the relative angles of approach in the range -80 to -50. In these cases, the agent should favor evasive action towards right/starboard. In the case of an overtake, the agent is according to the COLREG rules actually free to overtake on both the port side and the starboard side. However, in the case of a head-on situation both the encountering ships should take evasive actions towards the starboard side. Hence, the author chose to make the agent favor evasive actions towards the starboard side. The favoring of actions towards starboard is accomplished by penalizing the agent if it does not set the rudder to at least 15 degrees. A positive rudder degree means that the ship will turn towards starboard. The 15 degrees were chosen by the author, since five and ten degrees of turn is a too small an adjustments to effectively avoid a collision and still leave ample space to the other vessel.

With the head-on and overtaking scenarios going from the south towards north taken care of, the same scenario in a north-to-south going trajectory needs to be taken care of. The code and process are the same as in the previous scenario, except that the relative angles of approach are different. When going from north towards south, the relative angles for the overtaking and head-on scenario will be in the range -180 to -140. With the overtaking and head-on scenarios covered, the crossing scenarios need to be handled. The crossing scenarios are also dealt with in the same fashion as the overtaking and head-on scenarios, the only difference again being the angles of approach. For a crossing scenario with the agent coming from the north and the stand-on vessel going from west towards east, the relative angle of approach will be between 160 and 90 degrees. The crossing scenario with the stand-on vessel coming from the north and going towards south with the give-way vessel coming from east and going towards west the relative angle of approach will be in the range -130 to -100.

The reward function is not complete and some situations are not covered in the function. Only two direction-permutations of the COLREG scenarios are handled. Missing still is, for instance, cases with an overtaking scenario in a west to eastbound or east to west-bound direction. These situations where not handled, since they are outside the scope of this thesis. Despite the agent's shortcomings in certain situations, the results from the scenarios are promising. The results will be covered later on in the results chapter.

As mentioned previously, the reward function is the last part needed to complete the MDP used in this implementation. The reward function follows the formula presented

for rewards in the MDP theory section. The reward received will be dependent on both the state and the action taken. The same action in different states will not give the same reward.

```
def get_reward(ship, ships):
"""Returns the reward for a taken action as an integer"""
reward = 0
# avoid collision
if ship.collision == True: # if clause 1
    reward += -10000

# reward getting closer to destination
reward += (
    distanceTo(
        ship.x, ship.target_destination_x, ship.y,
            ship.target_destination_y,
    )
) * -1

# Do not want to be too close to other ship
# if clause 2
if distance_to_closest_vessel(ship, ships) < 200:
    reward += -1000

# if clause 3
if ship.done:
    reward += 1000

# if clause 4
if -80 <= ship.angle_of_approach <= -50: # head on and overtaking:
    south to north
    if (
        ship.rudder < 15
    ):  # correction to the right needs to be taken, will turn towards
        the right if the rudder is larger than 0
        reward += -2000

# if clause 5
if -130 <= ship.angle_of_approach >= -100: # crossing from right east
    to west
```

```
        if ship.rudder < 15:
            reward += -2000

    # if clause 6
    if -180 <= ship.angle_of_approach >= -140: # Head-on overtaking: north
        to south
        if ship.rudder < 15:
            reward += -2000

    # if clause 7
    if 160 >= ship.angle_of_approach >= 90: # crossing right west to east
        if ship.rudder < 15:
            reward += -2000
    return reward
```

Listing 5.4: Reward function

5.4.5.4 General helpers

The ship object's values need to be constantly updated as the simulation progresses and the agent takes different actions. The ship object has numerous different attributes and to update each one of them in the locations where the changes happen was not satisfactory. Instead, the helper method *update_ship_info* was created. The method handles all the updating of a ship's attributes in one place and is easily callable when a change has happened and the values need updating. The values are updated using the ship object's setter methods.

In the state, the agent needs to know the closest vessel speed. To make this happen the *get_closest_vessel_speed* method was created. It works by getting the ego ship and the other ships in the environment. It then looks up the id of the closest ship and then uses the id to obtain the speed attribute from the closest vessel and returns it.

The state representation was moved out of the environment to reduce the code duplication since the state was needed in multiple parts of the program. The state is composed of seven different observations of the environment. These seven attributes are: the agent ship's x coordinate, the y coordinate, bearing, speed, rudder degree, the distance from the agent ship to the closest other ship and, finally the speed of the closest ship to the agent. All the different values in the state have a purpose. The coordinates are naturally needed in order for the agent to know its location within the environment. The bearing is used by the agent to know in what direction it is traveling. The speed is important so the agent knows how fast it is going. The rudder degree is used by the agent to know in what di-

rection it is currently steering the ship. The distance to the closest vessel is useful for the agent to know, since it will be punished for being too close to another vessel. If it does not know the distance to the closest vessel, it cannot avoid taking this penalty efficiently. The speed of the closet vessel is also useful information for the agent to know since it then knows how fast the distance between itself and the closest vessel will be closing. The state needs to be reshaped before it can be returned. The shape of the state needs to be different if the state is used in training or in the simulation. This is since in the running of the simulation the batch size is left out and needs to be inserted. To overcome this the method passes along a flag to determine which state shape is expected by the caller.

5.4.6 Sim

The *sim* module is responsible for handling the simulation, keeping track of where all the ships are currently located, and knowing where land is in the simulation. The module also has a step method to advance the simulation. This method is called both when running a simulation and in training to advance the environment. As part of this, the step method in the *sim* module is responsible for updating the ship's information in accordance with what happens during a step. Since the step method knows what happens as the result of a step in the environment, it is able to determine if there has been a collision. The collision can be with land or with another ship and the *sim* module can also determine if the ship has reached its final destination. The update of the environment is done by calling the ship object's step method for all ships in a simulation. The module is also responsible for loading the ships from the config files into the simulator when running a simulation. Then based on the config file, the module assigns the agents to the ships. Resets of simulations, if the user wants to reset the simulation and run it again, is also handled inside the module.

5.4.7 User interface

The user interface has nothing in common with the RL agent, but it is a vital part of the whole built system and the reader needs to at least have a partial understanding of the structure of the user interface in order to get the correct picture of the whole system.

The user interface is composed of six classes with their own responsibilities. The classes work together to make up the user interface. The different classes are the *MainWindow*, *ShipView*, *WorldScene*, *MapView*, *TextDrawer* and, lastly, the *ButtonForm*. The *MainWindow* class is, just as the name hints, responsible for rendering the main application window to contain the other scenes, widgets, and buttons. The *ShipView* is responsible for handling the rendering of ships. The segregation of the ships into a separate view is needed to handle the updating of a single ship's state separately from the other ships

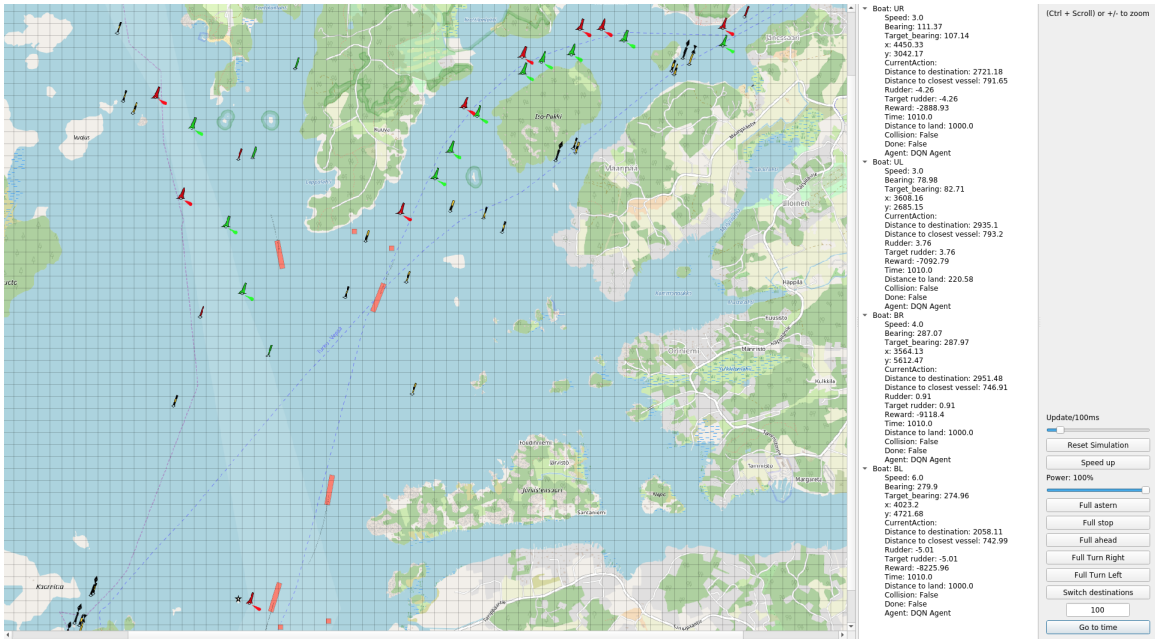


Figure 5.4: User interface

in the same scenario. One ship could have crashed while the others are still fine and the representation of this in the user interface is easier when each ship is an instance of a *ShipView*. The *WorldScene* is responsible for handling and updating everything inside the user interface. The *MapView* handles the user interactions with the map, such as scrolling and zooming. Displaying of all information related to ships is handled by the *TextDrawer* class. The buttons in the user interface are handled by the *ButtonForm* class.

The user interface gives the user control over the speed at which the simulation is progressing. The user can choose from realtime simulation to a thousand times speed up. The speedup is needed since in realtime the simulations are slow and long-running. If a user wants to prototype or test some changes, they cannot be expected to run the simulation in realtime and wait about 30 minutes for the simulation to finish.

The layout of the user interface can be seen in Figure 5.4. The ships are represented by the red rectangles, the text drawer with the information about the ships can be seen at the right in the picture. The controls for setting speed, power and other attributes of the vessels can also be seen at the right in Figure 5.4.

5.4.8 Ship

The *ship* module is, just as the name suggests, a ship object. It contains all the properties and logic of a ship. Ships used in the simulator have their properties taken from a typical oil tanker. The ship object has many properties just as a real ship. The most important ones for the RL are the properties of the ship used in the state. The properties in the state

that directly come from the ship object are the *x* and *y* coordinates of the ship, the bearing, speed and rudder angle. The other properties in the state are calculated using these values. The distance to the closest ship from the agent ship uses both ships' coordinates and the closest vessel's speed also comes from the ship object. The physics of vessel handling is also located in the ship class. Examples of these physics are drag force of the water and the force of the rudder, i.e. how effective the rudder is at adjusting the yaw of the ship. All the physics calculations for the ships are located in the ship object's step method, since the physics set bounds and affect the outcome of a step.

5.4.9 Main

Main serves as the entry point for the simulator. The user can start a training, verification, or simulation run from *main* depending on the arguments passed in the command-line interface. To aid with the parsing of arguments the standard Python module *getopt* [14] is used. The different options or command-line arguments supported are training, simulation, verification, path, and help. The training argument *"-t"* or *"–train"* needs to also include the type of agent one wishes to train with. The agent can be an agent operating on a discrete action space or a continuous action space. To train a discrete agent the argument *"-td"* would be passed and for training a continuous agent *"-tc"*. Additionally, the training argument needs to be followed by the number of training steps one wishes to perform. The full argument to train a discrete agent for 20 000 steps would then be *"python main.py -td 20000"*.

For running verifications in the simulator, the argument *"-v"* or *"–verify"* would be used. Then the external verification tool would take over. The implementation of the verification mode is very basic and more a proof of concept that the simulator can be used that way. The implementation of the verification mode goes outside the scope of this thesis and hence was left very brief.

The simulation mode can be started using the argument *"-s"* or *"–simulate"*. Additionally, the simulation mode needs to know what type of agent the user wants to simulate, a discrete, or continuous. For this reason the simulation also needs the flag *"d"* or *"c"* added as in the case of training. The simulation also needs to be told what weights it should use for the neural network. The user, therefore, can specify a path to where the weights can be found. The full argument for running a simulation with the weights saved from a previous training would then be *"python main.py -sd 'Weights/DQN/Weights20000.h5f'"*. If the user does not provide a path in the command line, the default included pre-trained weights will be used.

5.5 Using the simulator

Using the simulator is straight forward. All that is needed from the user is some basic skills using the terminal in order to specify the correct options for the program. If the user does not want to create his own scenarios to test the included agent or his own agents the user can choose to run one or several of the included pre-made scenarios. If a user wants to run his own scenarios, he needs to create his own configuration file outlining the scenario. The included configuration files of different scenarios work as a model for the user to follow. A user with the necessary coding skills can supply his own agent. This can be challenging for a new user if he is not familiar with the structure of the simulator and would require the user to read the source code to understand how to supply the new agent correctly. In this case, the user needs to create a custom agent that trains on the custom Open Ai Gym ship environment and add a ship agent that implements the *ShipAgent* class with its step method. The user would also need to make sure that the correct agent is called when a simulation is executed.

6. Results

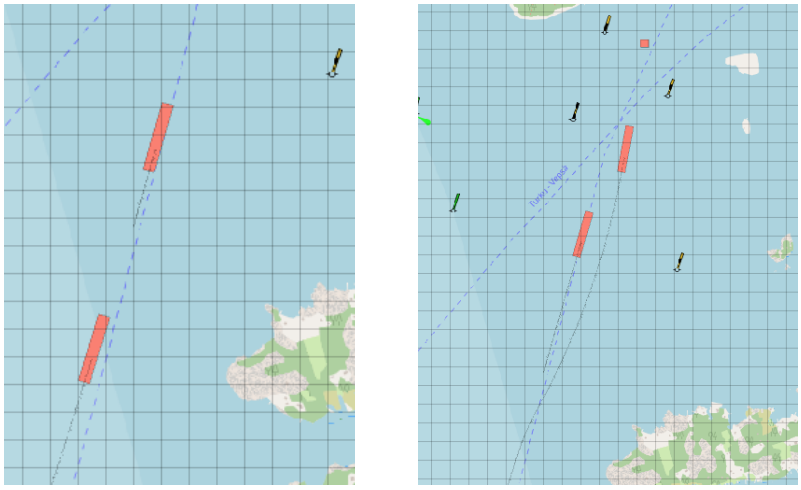
The result section reviews all the different scenarios defined as the initial test scenarios and presents the results when the agent is placed in the scenario within the simulator. Additionally, the results from a more complex scenario with four different vessels navigating the same waters are presented. The results for the overtaking, head-on, and crossings were obtained after 300,000 steps of training on the *allScenarios* configuration file. The results with four vessels were obtained after training for 100,000 steps on the same *allScenarios* configuration file.

6.1 Overtaking

Overtaking is one of the first defined scenarios for testing the agent's performance on COLREG rules. An overview of the essential parts of the scenario is found in Figure 6.1. It shows the vessels before and after the overtake. The agent vessel being the bottom vessel and the environment vessel being the top vessel in the figure. The routes the vessels have taken are shown as the dotted black lines after the ships.

The scenario starts with the agent vessel coming up on the slower moving vessel from behind, as seen in Figure 6.1(a). The agent vessel wants to overtake the vessel in front and starts the overtake on the starboard side of the vessel in front. The agent vessel makes a course correction towards its starboard side to pass the other vessel safely. When the agent vessel has overtaken the other vessel, it again adjusts its course to sail towards the destination, as seen in Figure 6.1(b).

The agent's performance in the overtaking case is good. It follows the COLREG rules concerning overtaking. The scenario demonstrates that the agent has successfully learned a policy for overtaking scenarios in a south to northbound direction. This is directly in line with the goals specified at the beginning of the thesis.



(a) Starting point (b) Complete

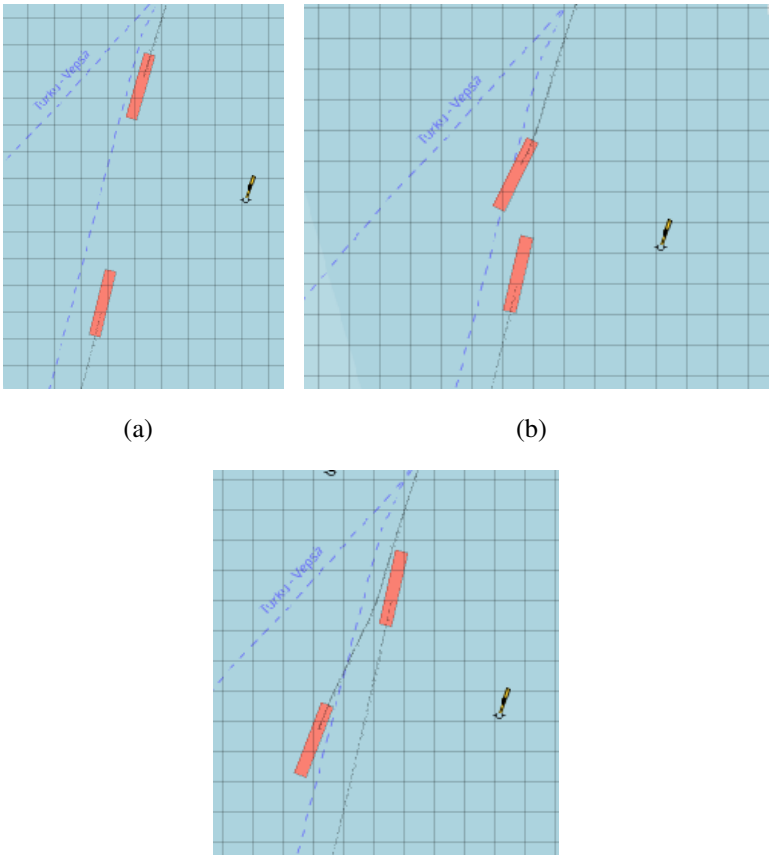
Figure 6.1: Overtaking

6.2 Head-on

The head-on scenario is the second scenario to test the agent's performance. An overview of the head-on scenario is found in Figure 6.2. As seen in the figure, the vessels have courses that would lead them to crash head-on, if no action were taken. In the scenario, the agent vessel is the top one and the environment vessel is the bottom one in the figure. Also in this scenario, the paths that the vessels take can be seen as the trailing black dots. According to the COLREG rules, both vessels would alter their course to their starboard side to avoid a collision in this case. To simplify the scenario only the agent vessel will take corrective actions to avoid a collision.

The scenario starts with the ships coming up on each other head-on, as seen in Figure 6.2(a). To avoid the collision the agent vessel starts to make corrective actions by altering its course to its starboard side, as seen in Figure 6.2(b). When the collision has been avoided and the ships have sailed past each other, the agent adjusts its course towards the destination point it was sailing to before it had to avoid a collision, as seen in Figure 6.2(c).

The agent's performance in a head-on collision avoidance scenario is also good. The agent manages to follow the COLREG rules that apply to it in the scenario. The scenario demonstrates that the agent has successfully also learned a policy for dealing with head-on collision scenarios in a north to south-bound direction. This is also in line with the goals specified.



(a) (b)

(c)

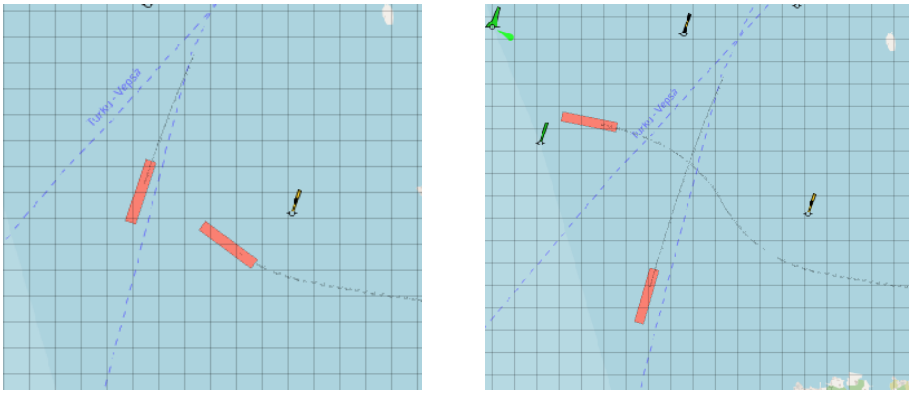
Figure 6.2: Head-on scenario: (a) Starting point (b) Midway point (c) Scenario complete

6.3 Crossings

According to the set goals, the agent also needs to be able to deal with crossing scenarios in accordance with the COLREG rules. There are generally two different crossing scenarios, a crossing from left and a crossing from right. In these crossing scenarios, the agent vessel is never the stand-on vessel, since it is not as interesting an investigation to see if the agent will do nothing compared to if it will avoid the collision with corrective actions.

The initial setup of the crossing-from-right scenario can be seen in Figure 6.3(a). The agent vessel in the scenario is the rightmost vessel. The environment vessel is the top vessel. In this scenario, the two vessels will collide if no evasive actions are taken. The dotted lines after the vessels also in this scenario show the paths they have taken. In Figure 6.3(a), it is observable that the agent ship has started an evasive maneuver to its starboard side to avoid the collision. In Figure 6.3(b), it is visible that the actions taken by the agent vessel were successful in avoiding a collision and in line with the COLREG rules for the scenario. The agent vessel avoided the collision by going behind the crossing vessel

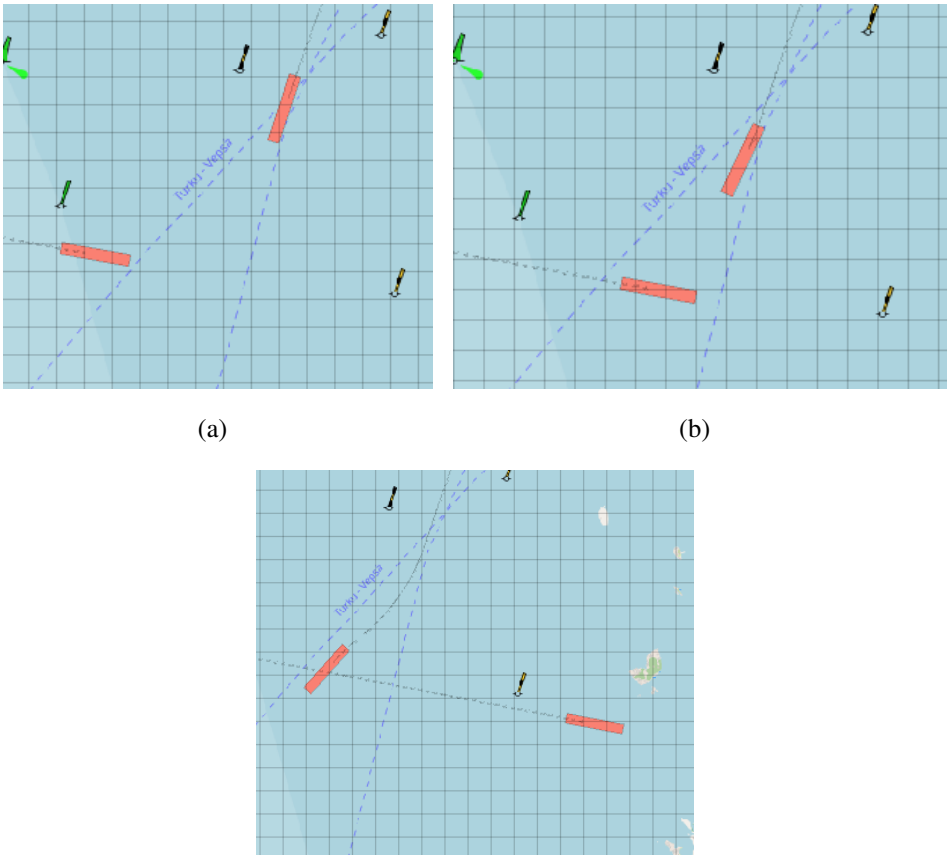
and avoided an intersecting path with the other vessel altogether. This is proof enough to conclude that the agent, in addition to the previous policies, has learned a policy for dealing with crossing scenarios from the right in an east to west direction.



(a) Starting point (b) Complete

Figure 6.3: Crossing Right

A crossing scenario from the left is not that different from a crossing scenario from the right. It can be thought of as the same scenario flipped. An overview of the crossing from left scenario is seen in Figure 6.4(a). In this case, the agent vessel is the top vessel and the environment vessel is the vessel to the left in the figure. Since this scenario is a crossing, the agent vessel needs to avoid the collision by altering its course to its starboard to avoid intersecting courses. The agent vessel begins this corrective action to its starboard side in ample time, as seen in Figure 6.4(b). The agent vessel's course correction is enough to avoid a collision in this scenario, as seen in Figure 6.4(c). When the agent vessel crosses the dotted line and intersects the environment vessel's path, the environment vessel is already a safe distance away from the agent ship. The running of this scenario shows the agent's ability to also learn policies for crossing scenarios in a north to south-bound direction with the environment ship approaching from the west.



(a) (b)

(c)

Figure 6.4: Crossing from left: (a) Starting point (b) Midway (c) Complete

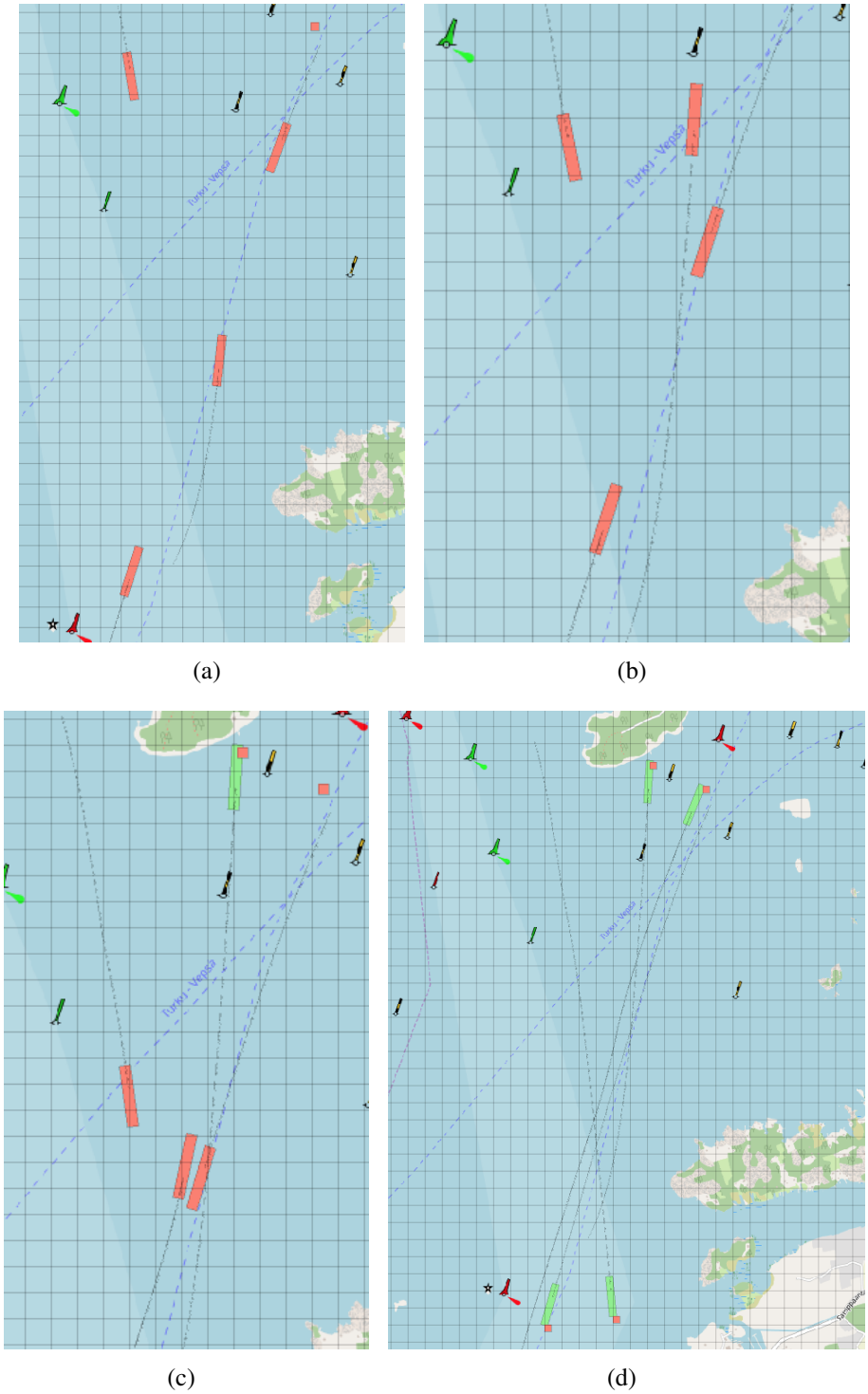
6.4 Four vessels

The most interesting goal defined in the thesis goals is a multiple vessel scenario. The agent was never trained in a scenario with multiple ships present. All training scenarios include the agent ship and an environment ship; hence, there were never more than two vessels present. In the scenario with four vessels, all vessels are agent ships and are controlled independently from one another. Taking multiple vessels into consideration at once while still following COLREG rules is very interesting. It is not always clear what scenario the agent vessel is in.

The initial setup of the scenario can be seen in Figure 6.5(a). As seen in the figure, two of the ships have a northbound heading and the other two ships have a southbound heading. This makes an interesting scenario when they all meet at almost the same time in the middle, as seen in 6.5(b). Interesting to observe is that the middle vessel in Figure 6.5(a) decides to take the penalties it will get from altering its course towards its starboard side and intersecting the oncoming vessel's path. The distance is still ample enough be-

tween the vessels and does not result in a collision. This demonstrates that the agent is capable of making tradeoffs and taking a penalty in order not to receive the greater penalty it would get in the case of a collision. In Figure 6.5(c) it can be observed that the middle vessel has safely reached its final destination, as indicated by its change in color from red to green. Additionally, notable in Figure 6.5(c) is that two of the vessels end up very close to each other. They will get penalties for this but not as large penalties as in the case of a crash. This once again shows the agent's ability to compromise. All the ships in the multiple vessel scenario end up at their final destinations safely, as indicated by their green colors in Figure 6.5(d).

This chapter has covered the performance of the proof of concept agent in standard COLREG scenarios. The agent's performance in the scenarios was promising. It managed to deal with head-on and overtaking scenarios in accordance with the COLREG rules. It leaves ample space and does the correct evasive actions. Crossing scenarios are also handled by the agent. When exposed to scenarios not seen before the agent also shows promise. It manages to avoid collisions in a testing scenario with multiple ships present. To avoid collisions whit multiple ships present the agent uses its ability to generalize and make compromises. It knows to take intermediate penalties to achieve a higher reward in the end.



(a) (b)

(c) (d)

Figure 6.5: Four ships scenario: (a) Starting point (b) Midway (c) One collision avoided (d) Complete.

7. Discussion

The RL agent was created to be a part of a larger maritime simulator project at Åbo Akademi University. The project needed an initial navigational agent to be placed in the simulator and the author's thesis was a good candidate. The author's developed agent also needed a simulator to be tested, which led to the author's thesis being a part of the larger project. Ivan Porres supervised the project and provided guidance during the development. Porres also supplied and designed the initial template for the project. Kim Hupponen [8] developed the simulator and simulation parts of the simulator. This means that the *sim* module and user interface were primarily developed by Hupponen. The *sim* module is responsible for handling the simulation, keeping track of where all the ships are currently located, and knowing where land is. The author's part in the user interface development was minimal. The author mostly contributed with information about the ships and agents that were needed in order to ease the debugging of the agents. The ship object properties dealing with physics and modeling were designed by Kim Hupponen. The author contributed to the ship object mainly on the parts needed to extend the environment state and display information needed when debugging the RL agent.

The author's main development focus was the RL agent and environment to train the agent in. The author also assisted and solved bugs and issues in the simulator when needed to advance in the development. The author made some modifications to the traditional structure of an Open Ai Gym environment to make the environment better fit the simulator. The *take_action* method that usually is found inside the environment was moved outside to the *helpers* module to reduce code duplication and make prototyping faster. The taking of actions was needed in multiple places and hence it made the most sense to share the code. This achieved faster prototyping since changes only had to be made at a single place in the code.

Another part that is usually part of the environment is the reward function. This function was also needed in multiple places in the code. This led to the author moving the function outside the environment to the helper module. The reward function was the part of the author's practical work that ended up taking the longest before arriving at a satisfactory result. The development was mostly done through careful consideration and trial and error. What to include and in which ways required contemplation and thought. Although trial and error were involved, it was never pure guessing. Which states to be considered

not desired or suboptimal and which states to consider desirable was for the most part clear to the author. Therefore this was the easiest part of the reward function to decide on. Trial and error were used to arrive at the rewards given out for different actions. The reward function is currently not handling all permutations of COLREG scenarios. The author did some prototyping with adding all permutations to the reward function. The initial results where promising but still required additional work and testing. The author hence decided to leave out the permutations of situations.

The state of the environment also traditionally lives inside the environment. As previously this would have caused code duplication and hence the author moved the code for keeping track of the state outside of the environment. This makes dealing with changes in the state easier since the changes only need to be made in a single place.

The author ended up making additional agents for the simulator that were not originally planned to be included. There was a need to introduce some form of randomness or unexpected elements. The additional agent was added since the human error element of maritime navigation needed to be included in the simulator. This introduced randomness can then be used to test how robust different navigational algorithms are against unexpected events and previously not seen situations. To introduce these elements the agent *SemiRandomDQNAgent* was added.

A simplification that is worth exploring in future work is the author's hardcoded threshold for when the RL agent obtains the control of the ship to do collision avoidance. This value is currently based on the author's intuition about what can be considered ample space and in due time to start collision avoidance procedures. In future work, it would be interesting to teach the agent to learn what this value should be. It is almost certain that this hard-coded value is not suitable for all situations. The author believes that better performance could be achieved if the agent would learn when to switch from autopilot navigation to collision avoidance. The agent would then take the whole situation into account.

8. Conclusion

In this thesis, the possibility of using RL in a maritime setting has been explored. The exploration was done as a proof of concept implementation of an RL agent that could control vessels inside a simulator. The goals set up for the proof of concept was to explore if it was possible to make an RL agent follow the COLREG rules in four different base scenarios listed in the COLREG rules. Additionally, the RL agent's performance was evaluated in a multi-vessel scenario.

The agent's performance and ability to accomplish the goals were evaluated with test runs in the simulator and validated with visual inspections by the author. The lack of mathematical proof for the validity of the scenarios is a factor that makes the results less formal. The simulator used to test the agents was developed alongside the agents and in-house at Åbo Akademi University. This means that the simulator is still quite new and may have yet undiscovered bugs or lacking features that would greatly have influenced the results obtained in this thesis. The simulator is yet to be published, which makes the independent validation of these results not possible at this time.

Despite the lack of formal proof, the results from the trial runs with the agent are promising. The agent manages to learn the desired behavior in all of the test cases. Especially interesting is that the agent manages to solve a more complex and diverse scenario than trained on. This is an indication of the agent's ability to generalize and not just remember different situations. It is this ability to generalize that gives RL its great potential.

Automation shows great promises for solving safety problems and introducing cost savings in the long run. Further development and testing are, however, needed. The test cases performed in this thesis alone are not enough proof to conclude that the approach is suitable for large-scale adoption. The test cases show that RL agents can be thought basic behavior mimicking COLREG rules in toy examples of simple scenarios inside a simulator. The real world is infinitely more complex and diverse than ever possible to program into a simulator. Significant real-world testing and further development of the RL agent would be needed in order to actually implement a suitable commercial application.

8.1 Future work

During the implementation of the agent and simulator, several compromises had to be made. One omission that is worth exploring in future work on the agents is the drawback that the vessels are not aware of where land is. This, as previously mentioned, is due to the simplifications made to focus on the COLREG scenarios. Collision avoidance with land was omitted and not considered to be as relevant for the proof of concept. The agent is also only aware of the one other ship in the environment at a time. It is only aware of the closest vessel to itself at any given point. It would be interesting to explore whether there are any differences in performance if the agent is aware of more vessels. Since avoiding a collision with the closest ship without knowing about a second close ship might lead to the agent vessel ending up in an unnecessary risky or dangerous situation, that could have been avoided completely if the agent had been aware of the second vessel's presence.

The basic navigation agent is fine and accomplishes its job of providing the environment vessels with basic navigation abilities. The basic abilities are good enough for the environment vessels and for navigating the agent vessel when there are no collisions to be avoided. The basic agent is, however, somewhat lacking in its performance. In future work, it would be interesting to explore and implement a better navigational agent that is able to accept way-points along the way that it has to navigate through. This would provide the basis to also implement path planning. This could be utilized to design more complex situations and longer missions for the vessels. The vessels could have to sail through some points to reach their final destination. For instance, the points could be positioned so the vessel always stays in the fairway. Now the basic agent will take the straightest path to its goal. This could also be a basis to try avoiding collisions while always staying inside the fairway.

Another basis for future work is that currently, the simulator requires coding skills and familiarity with the terminal to use. In future versions of the simulator, a more novice way of creating configuration files could be beneficial. A form of drag and drop of ships into the environment where the user could rotate the ships to change their bearing and specify speed and other related things directly from the user interface would make the simulator easier to use. The addition of new agents to the simulator could also be simplified since it requires coding skills. The interface and coding necessary to implement one's own agents into the simulator could be improved in future versions of the simulator by changing the structure of how the simulator loads and uses agents. The agents could, for instance, be made to contain all relevant details in a single file.

9. Svensk sammanfattning

9.1 Introduktion

Under de senaste åren har efterfrågan av obemannade fartyg ökat. Denna ökning beror främst på två faktorer. Den första faktorn är de mänskliga misstag som inträffar inom sjöfartsindustrin. Enligt rapporter beror över 75 % av alla dödsfall som inträffar till sjöss på mänskliga misstag, dessutom beror 89-96 % av alla kollisioner också på mänskliga misstag [1]. Nämnvärt är också att utredningar har visat att 56 % av dessa kollisioner inträffade på grund av att COLREG-reglerna [2] inte följdes. Den andra faktorn som påverkar är de potentiella inbesparingar som kan göras om man kan upprätthålla fartyg med minskad personal. Helt obemannade fartyg är alltså möjliga. Dessa medför dock utmaningar. Ett helt obemannat fartyg behöver vara kapabelt att ta beslut helt utan mänsklig inblandning. Att lösa detta problem är inte lätt eftersom det innehåller många dimensioner och vinklingar.

Denna avhandling är avsedd att utforska en potentiell lösning på detta problem. I avhandlingen behandlas användningen av *förstärkt inlärning* (eng. reinforcement learning) för att skapa en agent som kan navigera fartyg säkert och i enlighet med COLREG-reglarna. Den implemeterade agenten är endast en prototyp och är avsedd att fungera som en utgångspunkt för vidare forskning. Agenten kommer primärt att fokusera på att lära sig undvika kollisioner i omkörningar, möten, situationer med korsande kurser och ett scenario med flera fartyg. Agenten implementeras som en del av ett större projekt. Projektet i fråga går ut på att utveckla en simulator för marina förhållanden. Simulatorn är avsedd att hjälpa användare i utvecklingen av algoritmer för navigering av fartyg.

9.2 Sjöfartens regelverk

Enligt *International Maritime Organization* (IMO) är den maritima industrin en av världens mest riskfyllda industrier [2]. IMO anser att riskerna inom den maritima industrin bäst hanteras genom olika regelverk och förordningar. Ett av de mest centrala regelverken inom den maritima industrin är *Convention on the International Regulations for Preventing Collision at Sea* (COLREG). COLREG är regelverket som hanterar hur kollisioner ska undvikas till sjöss. Regelverket kan ses som vägregler för sjöfart. COLREG innehåller

41 olika regler. De mest relevanta reglerna för denna avhandling är reglerna 8, 13, 14, 15, 16 och 17. Regel 8 beskriver hur kollisioner kan undvikas. Regel 8 bestämmer att alla undvikande manövrar som faryg gör ska tas i god tid och i enlighet med god sjömannaetik. Manövrarna som görs ska också vara tillräckligt stora för att andra fartyg enkelt ska se dem. Tillräckligt med utrymme ska också lämnas mellan fartygen. Dessutom ska ett fartyg i en riskfylld situation konstant övervaka om det finns behov att vidta vidare undvikande manövrar. Regel 8 fastslår också att den primära, undvikande manövern ska vara att ändra kurs. Ändringar i hastighet ska göras endast för att vinna tid eller som sista utväg för att undvika en kollision [2].

Regel 13 bestämmer hur omkörningar av andra fartyg ska göras. En omkörning av ett annat fartyg behöver först och främst följa alla bestämmelser i regel 8. Det fartyg som kör om ett annat fartyg ska alltid undvika att komma i vägen för fartyget som blir omkört. Regel nummer 14 bestämmer hur fartyg ska agera när de möter varandra för mot för. I denna situation ska de båda involverade fartygen ändra sina kurser mot styrbord. Detta gör att fartygen passerar varandra på ett ökat avstånd. Regel nummer 15 dikterar hur situationer där fartyg har korsande kurser ska skötas. Regel 15 bestämmer att det fartyg som har ett annat fartyg på sin styrbords sida ska hålla sig ur vägen. Detta ska göras genom att väja undan utan att hamna framför det andra fartyget. Med andra ord ska det väjande fartyget köra bakom det andra fartyget. Regel 16 berättar hur fartyget som väjer ska agera. Kontentan av regeln är att fartygen ska hålla sig ur vägen för varandra. Regel 17 bestämmer vad fartyget som inte väjer bör göra. Regeln säger att fartyget ska behålla sin kurs och hastiget. Fartyget ska endast väja om det är enda sättet att undvika en kollision. Detta kan hända om det väjande fartyget inte följer reglerna eller om det har gått så fel att en kollision inte längre kan undvikas endast genom att det väjande fartyget agerar ensamt. Då behöver också det andra fartyget vidta åtgärder för att till varje pris undvika en kollision [2].

9.3 Obemannade fartyg

Enligt Liu et al. [12] är definitionen på ett obemannat fartyg är avsaknaden av personal ombord. Dessutom behöver ett obemannat fartyg uppvisa någon form av intelligent beteende. Obemannade fartyg är inte ett nytt koncept utan de har redan utvecklats i 20 år [13]. De hittils utvecklande, obemannade fartygen har varit halvautonoma, vilket betyder att de har till viss del förlitat sig på mänsklig vägledning och stöd, detta till skillnad från helautonoma fartyg som fungerar helt utan mänsklig inblandning [12]. Liu et al. påpekar många fördelar med helautonoma fartyg, de är billiga i drift eftersom minskad personal innebär minskade löneutbetalningar. Dessutom ökar säkerheten genom att mänskliga mis-

stag minimeras. En oväntad fördel är att helautonoma faryg tenderar att vara lättare och smidigare, vilket innebär att de kan transportera mera gods i samma last jämnfört med traditionella fartyg. Detta innebär också inbesparingar genom att samma mängd gods kan transporteras med färre resor [12].

Trots alla dessa fördelar med helautonoma fartyg har de ännu inte tagits i bruk i större utsträcking [13]. Detta beror främst på de utmaningar som ännu finns. Utmaningarna beror främst på att fartygen inte är ensamma på havet. De behöver vara kapabla till interaktion med andra fartyg och att ta andra fartyg i beaktande när de utför beslut [13]. Dessa situationer med interaktion och beaktande av andra fartyg kan var ytterst komplexa [12]. Utmaningen ligger därför korfattat i att få de helautonoma fartygen att följa COLREG-reglerna och att kunna hantera all tvetydiga eller osäkra situationer som kan uppstå på havet.

9.4 Förstärkt inlärning

Förstärkt inlärning går ut på att genom upprepade försök och misslyckanden lära en agent att lyckas lösa problem [16]. Förstärkt inlärning är en kombination av *dynamisk programmering* och *övervakad inlärning*. Genom att kombinera metoderna kan förstärkt inlärning lösa problem som metoderna på egen hand inte klarar av att lösa. Fördelen med förstärkt inlärning är dess förmåga att generalisera. Detta gör metoden lämplig att kunna lösa problem som tidigare ansetts vara omöjliga [16].

En implementation av ett förstärkt inlärningssystem innehåller olika delar som behandlas nedan. Först och främst innehåller ett förstärkt inlärningssystem en agent och en miljö. Agenten interagerar med miljön för att utforska och lära sig saker om den. Det agenten lär sig om miljön använder den för att nå det mål som satts och för att nå målet behöver agenten känna till miljön. Oftast känner agenten till vilket tillstånd miljön befinner sig i. Vad som skall räknas till miljö inom ett förstärkt inlärningssystem är inte entydigt definerat, men den vanligaste definitionen menar att allt som inte är agenten i sig själv hör till miljön. Miljön och agentens samverkan baserar sig på att agenten kan ändra miljöns tillstånd genom olika handlingar, exempelvis genom att agenten förflyttar sig. Miljön svarar på dessa handlingar genom att presentera ett nytt tillstånd åt agenten. Agenten reagerar på detta nya tillstånd med en ny handling och så vidare. Dessa handlingar som agenten utför har olika belöningar. Agenten försöker lösa uppgiften den har getts genom att ändra på miljöns tillstånd och om en ändring går i rätt riktning ges agenten en belöning för att veta att den har gjort en korrekt handling. Om agenten har agerat dåligt kan den bli bestraffad eller bli utan belöning. Det agenten försöker lära sig är en regel för hur den skall handla i olika tillstånd som miljön befinner sig i. Det är denna regel

som sedan definerar en agents beteende. Det agenten försöker komma fram till är vilken regel som kommer att resultera i maximal belöning. En annan viktig del av ett förstärkt inlärningssystem är hur belöningar ges. Den användare som bygger systemet måste definera en matematisk funktion för hur belöningar skall ges och är ansvarig för att se till att belöningar ges på ett sätt som leder till att agenten löser den givna uppgiften [15].

Det finns olika metoder för hur man skall designa agenter och förstärkta inlärningssystem. I avhandling används metoden *Deep Q-learning* som presenterades av Mnih et al. [18]. Metoden använder djup maskininlärning för att estimera värden för olika handlingar en agent kan utföra i olika tillstånd. Agenten lär sig därmed vilken handling som är bäst att utföra i olika tillstånd [18].

För att förstå hur en agent tar sina beslut och hur förstärkt inlärning fungerar behövs matematik. Den matematik som utgör grunden för förstärkt inlärning är en *Markovian Decison Process* (MDP). MDP har även kallats den optimala och traditionella metoden för att representera en sekvens av beslut matematiskt [15]. MDP går i korthet ut på att representera tillstånd och handlingar som kan utföras i dessa tillstånd matematiskt. Inkluderat i en MDP är även sannolikheten för att byta tillstånd, belöningar som ges i tillstånd och en avdragsfaktor. En avdragsfaktor är en faktor med vilken framtida belöningar för tillstånd beaktas i det nuvarande tillståndet. Framtida belöningar kan med hjälp av avdragnings faktorn viktas olika. Faktorn kan användas för att få agenten att alltid välja den belöning som för stunden ser bäst ut utan att beakta framtida belöningar. Alternativet till denna kortsynthet är att sätta faktorn så att agenten försöker planera för större belöningar i framtiden och inte agerar girigt [15].

9.5 Implementation

Agenten implementerades som en del av ett större projekt inom Åbo Akademi. Det större projektet gick ut på att skapa en simulator för testning av olika navigeringsalgoritmer för fartyg. Min uppgift var att som en del av projektet implementera en första navigeringsalgoritm och jag valde att implementera denna algoritm med hjälp av förstärkt inlärning. Själva agenten implementerades med hjälp av *Keras-rl*-biblioteket [6] för att inte behöva implementera grundläggande funktioner. Dessutom är ett bibliotek mindre sannolikt att innehålla fel då flera personer redan har använt biblioteket och hittat de flesta felen. Jag implementerade även en specialmiljö att träna agenten i genom att det inte fanns färdiga miljöer för maritima förhållanden att tillgå. En belöningsfunktion behövde utecklas för agenten och är den viktigaste delen hos en agent. Med funktionen styrs beteende som agenten kommer att lära sig och använda för att lösa den givna uppgiften. Funktionen fungerar genom att beakta vinkeln mellan två fartyg. Utgående från vinkeln kan det sedan

bestämmas i vilket COLREG-scenario som agenten befinner sig i just nu. Därefter kan agenten bestraffas om den inte utför korrekt undanmanöver.

9.6 Resultat

Den implementerade agenten lyckas lära sig att undvika kollisioner i omkörningar, mötanden och i situationer med korsande kurser. Agenten följer även COLREG-reglerna när den undviker kollisionerna. Dessutom lyckas agenten undvika kollisioner i ett mera komplicerat scenario än den har sett under träning. Det mera komplicerade scenariot innehåller fyra fartyg som seglar mot varandra. Vilka COLREG-regler som gäller under vilken tidpunkt i scenariot är tvetydigt. Agenten lyckas ändå undvika att kollidera med andra fartyg. Detta är ett tecken på att agenten klarar av att generalisera. Detta är det som vill uppnås med agenten, den blir inte låst att endast följa reglerna. Den inser att det kan vara skäl att ta mindre bestraffningar som en följd av att bryta mot regler. Slutresultatet är ändå en högre belöning genom att den lyckas undvika den enorma bestraffning som skulle bli utdelad ifall den skulle ha kolliderat med ett annat fartyg.

9.7 Slutsats

I denna avhandling har utforkskats möjligheten att använda förstärkt inlärning för att lösa utmaningarna som finns med hel-autonoma fartyg genom att implemetera en prototyp av ett förstärkt inlärningssystem. Målet som sattes upp var att studera om det är möjligt att lära en förstärkt inlärningsagent att följa COLREG-reglerna. Agentens prestationer utvärderades visuellt i en simulator. De första resultaten är lovande och kan fungera som en bas för fortsatt forskning. Automation visar sig vara lovande för att minimera risker och spara pengar inom sjöfartsindustrin. Testerna och slutsatserna i denna avhandling är inte tillräckliga för att bevisa att förstärkt inlärning är lämplig för storskalig användning i navigeringssystem, men de påvisar att det finns förutsättningar. Testerna visar att agenter med förstärkt inlärning kan lära sig COLREG-regler inom ramen för kontrollerade experiment i en simulator. Långtgående testning och utveckling med riktiga fartyg krävs som vidare utveckling.

References

[1] A. M. Rothblum, *Human error and marine safety*, 2000.

[2] I. M. Organization, *Colreg*, 1972. [Online]. Available: http://www.imo.org/en/About/Conventions/ListOfConventions/Pages/COLREG.aspx.

[3] T. Statheros, G. Howells, and K. M. Maier, "Autonomous ship collision avoidance navigation concepts, technologies and techniques", English, *The Journal of Navigation*, vol. 61, no. 1, pp. 129–142, Jan. 2008. [Online]. Available: http://journals.cambridge.org/abstract_S037346330700447X.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[5] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[6] M. Plappert, *Keras-rl*, https://github.com/keras-rl/keras-rl, 2016.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[8] K. Hupponen, *Simple ship sim, a simulator for testing machine-learning algorithms for ships tbp*, 2020.

[9] I. M. Organization, *International maritime organization*. [Online]. Available: http://www.imo.org/.

[10] Safety4Sea, *3,174 maritime casualties and incidents reported in 2019*. [Online]. Available: https://safety4sea.com/23073-maritime-casualties-and-incidents-reported-in-2019/.

[11] R. C. M. Sea and Rescue, *Characteristics of navigation lights*. [Online]. Available: https://rcmsar12.org/boating-resources/vessel-navigation-lights/characteristics-navigation-lights/ (visited on 03/30/2020).

[12] Z. Liu, Y. Zhang, X. Yu, and C. Yuan, "Unmanned surface vehicles: An overview of developments and challenges", English, *Annual Reviews in Control*, vol. 41, pp. 71–93, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.arcontrol.2016.04.018.

[13] J. E. Manley, "Unmanned surface vehicles, 15 years of development", English, IEEE, Sep 2008, pp. 1–4, ISBN: 0197-7385. [Online]. Available: https://ieeexplore.ieee.org/document/5152052.

[14] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2. print. Cambridge, Mass. [u.a.]: MIT Press, 2015, ISBN: 9780262193986.

[16] M. E. Harmon and S. S. Harmon, "Reinforcement learning: A tutorial", English, Tech. Rep., Jan. 1997. [Online]. Available: http://www.dtic.mil/docs/citations/ADA323194.

[17] S. David, *Lecture 2: Markov decision process*, 2015. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning", English, *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/25719670.

[19] C. J. C. H. Watkins, "Learning from delayed rewards", English, PhD thesis, 1989. [Online]. Available: http://catalog.crl.edu/record=b1459262.

[20] C. J. C. H. Watkins and P. Dayan, "Q-learning", English, *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992. [Online]. Available: https://www.openaire.eu/search/publication?articleId=od1874::d032ca5d74b9f6624a16d96869f30f56.

[21] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[22] G. Community, *Pyside2*. [Online]. Available: https://wiki.qt.io/Qt_for_Python (visited on 03/03/2020).

[23] Tensorflow, *Tensoflow.org*. [Online]. Available: https://www.tensorflow.org/.

[24] G. Community, *Keras.io*. [Online]. Available: https://keras.io/.

[25] I. d. N. Oren Ben-Kiki Clark Evans, *Yaml ain't markup language (yaml^{TM}) version 1.2*, 2009. [Online]. Available: https://yaml.org/spec/1.2/spec.html.

[26] M. Wiering, "Explorations in efficient reinforcement learning", PhD thesis, 1999.

[27] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, "Boltzmann exploration done right", English, May 2017. [Online]. Available: https://arxiv.org/abs/1705.10257.

[28] E. W. Weisstein, *Pythagorean theorem*. 2020. [Online]. Available: http://mathworld.wolfram.com/PythagoreanTheorem.html (visited on 02/17/2020).

[29] M. O. Reference, *Distance between two points (given their coordinates)*, 2011. [Online]. Available: https://www.mathopenref.com/coorddist.html (visited on 02/17/2020).