Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Brage Lytskjold

Improving Adaptive Stress Testing: Reinforcement Learning using Monte Carlo Tree Search with Neural Network Policies

Master's thesis in Computer Science
Supervisor: Ole Jakob Mengshoel
January 2022

NTNU
Norwegian University of
Science and Technology

Brage Lytskjold

Improving Adaptive Stress Testing: Reinforcement Learning using Monte Carlo Tree Search with Neural Network Policies

Master's thesis in Computer Science
Supervisor: Ole Jakob Mengshoel
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

When developing an autonomous safety-critical system, it is crucial that the autonomous agent is able to act safely in a wide range of different scenarios. Each scenario is defined by a range of factors that can include weather and visibility, internal delay and sensor noise, external forces, and encounters with other agents. This results in a virtually infinite number of possible scenarios, and testing the system for all scenarios becomes virtually impossible, especially for complex systems. One way to approach this problem is through Adaptive Stress Testing (AST). AST is a framework used to stress tests a safety-critical system by altering the variables of a simulated environment, finding the most likely sequence of environment disturbances that results in a failure scenario.

AST searches are most commonly directed using a reinforcement learning agent, which for this thesis are variations of Monte Carlo Tree Search (MCTS). MCTS is a state-of-the-art search algorithm that randomly samples a large number of simulated scenarios to create a heuristic. This results in long search times for testing systems with costly simulations. In our thesis, we look at ways to improve the efficiency of MCTS when applied to complex simulators, with a single simulation time of 1 second or more.

We address the long search times through our proposed model, Monte Carlo Forrest Search with Action Pruning (MCFS-AP). MCFS-AP aims to improve the efficiency of MCTS through two methods. The first method is the implementation of a neural network rollout policy and a value neural network. The rollout policy is used to improve the reward of simulated samples, while the value network is trained to predict the state-action value of the possible actions in a state. The second method is to periodically store the internal state of the simulator, allowing us to skip the calculation of the given state in future simulations.

The proposed model is applied in experiments to two simulators to compare the impact of the changes made to MCTS. Finally, we apply our model to stress test the collision avoidance system of the milliAmpere 2, an autonomous ferry prototype designed and built by NTNU.

Sammendrag

Ved utvikling av et autonomt sikkerhetskritisk system er det avgjørende at den autonome agenten er i stand til å handle trygt i en lang rekke ulike scenarier. Hvert scenario avhenger av en rekke faktorer som kan inkludere vær og sikt, intern forsinkelse og støy, eksterne krefter, og møter med andre agenter. Dette resulterer i et tilnærmet uendelig antall mulige scenarier, og å teste systemet for alle scenarier blir svært vanskelig, spessielt for komplekse systemer. En måte å nærme seg dette problemet på er gjennom Adaptive Stress Testing (AST). AST er et rammeverk som brukes til å stressteste et sikkerhetskritisk system ved å endre variablene i et simulert miljø, og finne den mest sannsynlige sekvensen av miljøforstyrrelser som resulterer i et feilscenario.

AST søket etter den mest sannsynlige feiltilstanden bruker vanligvis Reinforcement Learning agent. Dette gjør vi også denne oppgaven, gjennom bruken av Monte Carlo Tree Search (MCTS). MCTS er en toppmoderne søkealgoritme som tilfeldig prøver et stort antall simulerte scenarier for å lage en heuristikk. Dette resulterer i lange søketider for testsystemer med kostbare simuleringer. I oppgaven vår ser vi på måter å forbedre effektiviteten til MCTS når det brukes på komplekse simulatorer, med en enkelt simuleringstid på 1 sekund eller mer.

I vår foreslåtte modell forbedrer vi effektiviteten til MCTS-søket gjennom to metoder. Det første trinnet er implementeringen av en policy for utrulling av nevrale nettverk og et verdinevralt nettverk. Utrullingspolicyen brukes til å forbedre belønningen av simulerte prøver, mens verdinettverket er opplært til å forutsi tilstandshandlingsverdien til mulige handlinger. Det andre trinnet i å forbedre effektiviteten er å periodisk lagre den interne tilstanden til simulatoren, slik at vi kan hoppe over beregningen av den gitte tilstanden i fremtidige simuleringer.

Den foreslåtte modellen brukes i eksperimenter på to simulatorer for å sammenligne virkningen av endringene som er gjort på MCTS. Til slutt bruker vi modellen vår for å stressteste kollisjonsunngåelsessystemet til milliAmpere 2, en autonom fergeprototype designet og bygget av NTNU.

Preface

This thesis was written and carried out by Brage Lytskjold and constitutes his Master of Science in Computer Science degree at the Norwegian University of Science and Technology (NTNU). Professor Ole Jakob Mengshoel at the Department of Computer Science under the Faculty of Information Technology and Electrical Engineering at NTNU supervised the thesis.

I want to thank Professor Mengshoel for his guidance and feedback throughout this project. I would also like to thank the people at Zeabuz for providing the milliAmpere 2 simulator used in this thesis. Lastly, I would like to acknowledge Professor Keith L. Downing and his course IT3105 for a valuable introduction to MCTS in sequential games.

Brage Lytskjold
Trondheim, June 12, 2022

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this section, we introduce the main motivation for this thesis, before introducing the core concepts. After that, we define the thesis goals alongside the research questions addressed to reach the goals. Finally, we present a summary of the thesis structure.

1.1 Motivation

Scenario based stress testing is an approach for testing the limits of a system by observing how the system behaves in a range of edge case scenarios. All though scenario based stress testing of physical systems can be performed in real life, we often encapsulate the system under test (SUT) in a simulated environment. The simulator is an abstraction of the real world, modeling the behaviours and reactions of the environment the SUT interacts with. Using a simulator for testing can be beneficial as it allows us to tailor the scenarios with full control of external disturbances, while also being scalable, safe and cost efficient compared to real life testing.

As the simulator is an abstraction of the real environment a factor we need to consider is how realistic and accurate the abstraction is. This question introduces a trade off as a more realistic simulator is generally more computationally heavy, resulting in a longer simulation time. Defining the simulation for a given system under test becomes a balancing act between realistic simulations with more accurate testing data and less complex simulations with faster run times. In this thesis, we will attempt to alter this balance by reducing the number and length of simulation runs without negating the validity of the results.

1.2 Adaptive Stress Testing

Adaptive Stress Testing (AST) is a framework for scenario based stress testing. AST is used to find the most likely scenarios where the SUT is unable to maintain desirable behaviour. This is performed by altering a simulated environment in order to find a combination of the most likely and challenging scenarios for the system under test. AST uses reinforcement learning to find these alterations, given as a set of parameters that apply disturbances to the simulated environment to which the system under test (SUT) has to adapt.

When Adaptive Stress Testing was introduced by Lee *et al.* [1] in 2015, it was used to stress test a new generation of collision avoidance systems for commercial airplanes (ACAS X). The development of such a safety-critical system requires robust functionality, guaranteed to handle a wide range of scenarios. Proving this robustness is hard, as searching through all variations of every possible scenario is virtually impossible. The solution to this was to select a large set of difficult historical scenarios before altering them through the use of AST.

1.3 The milliAmpere ferries

The milliAmpere 1 and 2 are a set of autonomous urban ferry prototypes developed by NTNU. Real-world tests of the ferries were started in Q1 2021 and trial operation is scheduled to begin during summer 2022 [2]. As the autonomous ferry is a safety-critical system, it is necessary that the system performs well in a range of scenarios. This requires more rigorous stress testing in a simulated environment, allowing us to look at a range of edge case behaviours in extreme scenarios without putting people or property in danger.

Zeabuz has currently implemented simulators for a safe, cost-effective, and scalable way of training and testing different parts of the autonomous system [3]. The simulator used in this thesis is implemented to test the collision avoidance system (ColAv) within the autonomous system. This simulator, hereafter called the Zeabuz simulator, allows us to examine the ColAv behaviour of milliAmpere 2 in any given scenario, while applying disturbances to the ferry and environment throughout the simulation.

1.4 Goals

The goals of this thesis is partly based on the findings of our project thesis on the viability of the AST framework, introduced by Lee *et al.* [1], using MCTS as the reinforcement learning agent searching for failure events. The main principle of Monte Carlo tree search (MCTS) is, as the name suggests, the Monte Carlo method, which approximates numerical properties through a large number of samples generated via random simulations. A natural result of this is the strong correlation between the sampling time and the search time.

In our preliminary work, we studied Adaptive Stress Testing (AST) using MCTS as a reinforcement solver, applying the framework to a fast and interpretable simulator. Here we found that MCTS was capable of finding failure events, even in scenarios with an exceedingly low rate of failures. However, we observe that the search became inefficient and unstable when the failure events become sufficiently resulting in the need for a large number of searches to get a reliable result.

The reliance of MCTS on a large number of simulation was not a large hindrance using the simple simulator in our preliminary study, but when applied to more costly simulators, like the Zeabuz simulator, this becomes a problem.

To address some of these problems we propose a new model which we call Monte Carlo Forrest Search with Action Pruning (MCFS-AP). MCFS-AP introduces a set of alterations to MCTS in order to speed up simulations while also decreasing the number of simulations in a search. Following our motivation and the proposed MCFS-AP model, we arrive at the two main goals of this thesis.

They are as follows.

- **Goal 1:** Implement MCFS-AP and examine the effects that the proposed changes have on the efficiency of MCTS as a reinforcement solver for AST.
- **Goal 2:** Apply Adaptive Stress Testing to the simulated milliAmpere 2 autonomous ferry to examine the possible weaknesses and limits of the collision avoidance system in the autonomous agent.

To achieve these goals, we first implement different variations of MCFS-AP along with a baseline MCTS model. The models are then tested, running a set of experiments using both the fast and interpretable simulator, as well as the complex milliAmpere 2 simulator. The efficiency of MCFS-AP and the milliAmpere 2 autonomous ferry is examined by addressing the following research questions.

- **Research question 1:** How does the use of neural networks to select simulations affect the failure-finding rate of MCTS?
- **Research question 2:** How does recalling internal simulator states affect the failure-finding rate of MCTS?
- **Research question 3:** How well does milliAmpere 2 handle varying amounts of added delay in the perceived position of other vessels?
- **Research question 4:** How well does milliAmpere 2 handle challenging encounters with other vessels?

1.5 Thesis structure

The background theory is introduced in Chapter 2 before related works are discussed in Chapter 3. This introduces the main ideas and concepts applied for the experimental method presented in Chapter 4. Chapter 5 introduces the experiments and results of the thesis, which are discussed in more detail in Chapter 6. Chapter 6 concludes by addressing the research questions, before proposing future work.

It should be noted that some parts of this thesis, especially surrounding parts of the background theory as well as related works, overlap with the report from the fall of 2022 as the theory has not changed.

Chapter 2

Background

To lay the foundation for our implementation of the AST framework, this chapter introduces some key concepts within AST and reinforcement learning. Finally, we discuss previous work done within stress testing of marine vessels, AST, and Monte Carlo tree search.

2.1 Stress testing

When searching for failure events for a system under test (SUT), we can model an environment, define a set of initial conditions, and simulate the system inside the environment until the system has either achieved or failed its goal. When simulating a complex system, the number of states can be large. Furthermore, allowing conditions within the simulation to change throughout the simulation increases the number of possible simulations exponentially. If the failure rate of the SUT is sufficiently low, finding simulations in which the system fails through an exhaustive search becomes virtually impossible. To extensively test such an SUT, other methods must be used.

One way to do this is to implement an adversary to the system under test, as illustrated in Figure 2.1. Here, the adversary examines the results of previous simulations before making an informed guess as to what changes can be made so that the SUT might fail. By directing the search towards promising simulations, most simulations can be ignored, resulting in a drastic reduction in the search time.

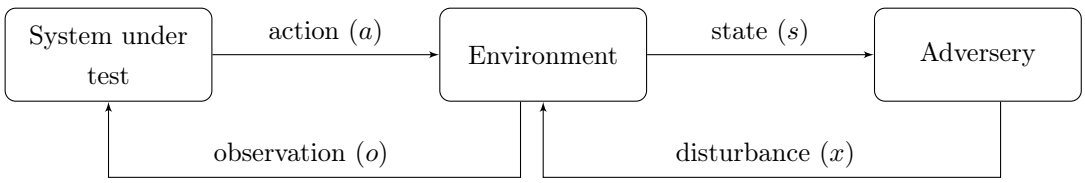


Figure 2.1: Adversarial stress testing. In order to stress the system under test (SUT) an Adversary, influencing the simulated environment is introduced. The Adversary aims to alter conditions within the environment, finding challenging scenarios for the SUT

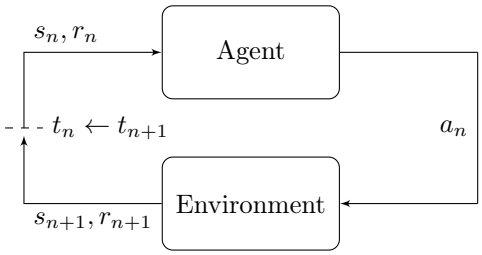
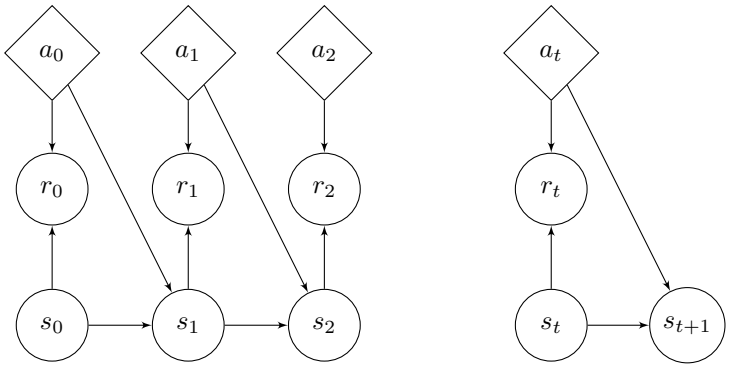


Figure 2.2: Partially observable Markov decision process. An action a is passed to the environment, resulting in the observable state representation s and reward r.



(a) General representation (b) Stationary representation

Figure 2.3: Stationary Markov decision process is defined by applying the he Markov assumption to a Markov decision chain, shown in figure a.

2.2 Markov Decision Process

A sequential decision problem is a problem of maximising an objective function given a set of decisions in a stochastic environment. When the model is known and the environment is observable, the problem becomes a Markov decision process (MDP) [4]. In a partially observable MDP an agent chooses an action a_t at time t, receives the resulting reward r_{n+1} and observes the new state s_{t+1} as illustrated in Figure 2.2. The mechanics behind the state transition are not known to the agent. In this paper, we will only consider partially observable MDPs where the outcome of action a_t on a state s_t is deterministic, always resulting in the same state s_{t+1}.

A central part of MDP is the Markov assumption. This assumption states that the transition between two states depends only on the current first state and the action taken in that state, not on any prior states or actions. By this assumption, we can simplify the sequential model to a stationary MDP, illustrated in Figure 2.3. As the Markov assumption tells us that the next state s_{t+1} is dependent only on the current state s_t, we can remove the previous states as long as the current state s_t is set.

Although an MDP without a maximum number of time steps can be studied, this article will focus on finite MDP, which are sequential decision problems with a finite number of time steps, given by t_{end}. By establishing a maximum number of time steps for a sequence, we resolve the need for reward discounting, as the number of actions used to calculate each reward is the same. In addition, if we determine that every action is of equal importance to the outcome of the final state, the total reward is given by the sum of all rewards calculated from every state transition. This

results in the following formula for the total reward R.

$$R = \sum_{t=0}^{t_{end}} r_t \tag{2.1}$$

The task of finding the optimal solution to the MDP now becomes a search for the sequence of actions $[a_0, ..., a_{t_{end}}$ from the initial state s_0 to a terminal states $s_{t_{end}}$ that maximises the total reward. This is an optimisation problem that might be solved using reinforcement learning, a subset of machine learning algorithms that chooses actions in an environment and tries to optimise a reward function.

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are machine learning models inspired by biological brains. ANNs consist of perceptrons connected by directed edges, a structure inspired by biological neurones and axons. In most ANNs, the perceptrons are divided into layers, resulting in an input layer, an optional set of hidden layers, and an output layer. The preceptrons in one layer then connected to all perceptrons in the next layer through the directed edges. A neural network with two hidden layers is illustrated in Figure 2.4

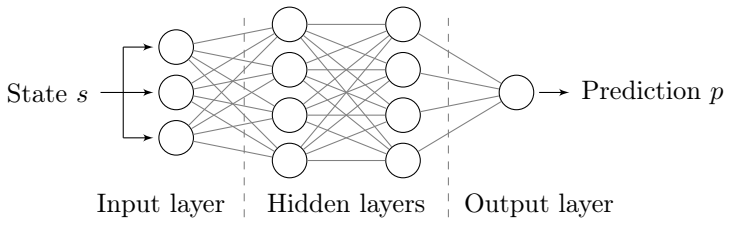


Figure 2.4: An artificial neural network with one input layer, two hidden layers and an output layer.

The edges between layers propagate values from perceptions in one layer to the next, augmenting each value using the weight stored at each edge. When the weighted values reach the target perceptron, they are put into an activation function, propagated throught the hidden layers before the resulting is given by the output layer. Throughout this propagation, each perceptron receives a set of weighted values from the previous layer, which is then put into an activation function. The resulting value of the activation function is then sent to the perceptrons in the next layer, repeating the process.

Training the network is done by passing an input looking at the output of a network and comparing it to the desired target output. The prediction error is calculated and propagated back through the network, updating the weights along the edges to minimise the error. In our work, the target values are calculated from the results of the simulations and the expected results calculated by MCTS.

2.4 Reinforcement learning

Machine learning (ML) is a branch of artificial intelligence that is devoted to automatically improving computer algorithms by "learning" from previous experience. One branch of ML is reinforcement learning. In reinforcement learning, our goal is to optimise the behaviour of an agent that executes a set of actions in an environment. The action sequence is then evaluated on the basis of the reward or punishment that is given to the agent after the action sequence has been executed.

Rewards and punishments given to the reinforcement learning agent are calculated by the reward function, which should reflect the objective of the agent. The reward function defines the desired outcome of the scenario and will vary from domain to domain. A reinforcement learner playing chess might be rewarded for winning a game, no reward for tie, and punished for losing. In our thesis, the adversarial vessel is rewarded when a crash occurs and punished in relation to how far the vessels came from crashing and how unlikely its behaviour was.

2.5 Adaptive Stress Testing

Adaptive Stress Testing (AST) is a framework to find the most likely failure state of the system under test. To achieve this, we apply reinforcement learning to search for the most likely sequence of disturbances to the simulated environment that, when applied throughout the simulation, results in a failure event. In practise, this is done by reducing the problem seen in Figure 2.1 to a Markov decision process (MDP) and solving it using reinforcement learning. Redefining the problem as an MDP is done through the following two steps.

2.5.1 Restructuring the problem

The first step necessary to redefine the problem is restructuring the problem illustrated in Figure 2.1 as a Markov decision process. This is achieved by encapsulating the system under test and the environment within a simulator object, resulting in the model illustrated in Figure 2.5. Here, the adversarial agent passes a disturbance x to the simulator at each time step. The disturbance is applied to the environment before the system under test chooses a new action, trying to reach its goal despite the disturbance. After each chosen action, the simulator transitions to a new state and the reward for that transition is calculated. A representation of the state s and the reward r is returned to the adversarial agent, allowing the agent to assess the outcome of the chosen disturbance.

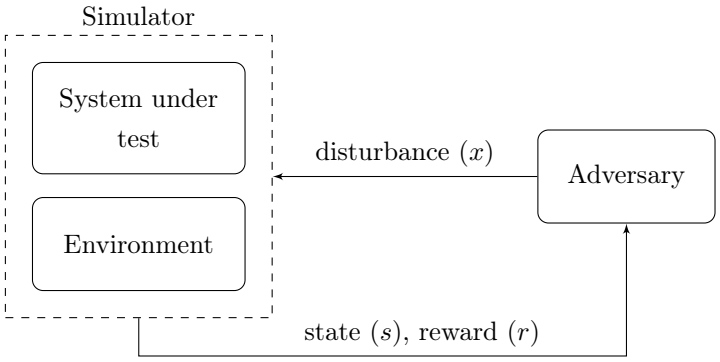


Figure 2.5: Adversarial stress testing treated as a Markov decision process by encapsulating the system under test inside the simulator.

Disturbance x is a general term for a wide variety of influences on the environment that will vary between different applications and domains. In a naval setting, the disturbance may contain the direction and strength of the current, the actions and responses of other vessels, and the wind. The importance of disturbances lies in their influence on the environment and, in turn, their influence on the actions chosen by the system under test.

2.5.2 Objective function

The second part needed before we have a complete MDP is an objective function which is the basis for the rewards given to the reinforcement learner. The objective should mirror the objective of the AST framework, which is to find the most likely sequence of disturbances that lead to a failure event. This objective can be divided into two parts. The primary objective is to find failure events, and the secondary objective is to find the most likely sequence of disturbances [1].

The objective function for directing the search towards a failure event can be expressed as

$$\max_{x_0,\ldots,x_{end}} \big[R_e * (s_{t_{end}} \in E) - d * (s_{t_{end}} \notin E)\big], \tag{2.2}$$

where E is a set of all states that are failure events, R_e is a non-negative reward, and d is a positive metric of the distance the current state is from a failure event. When applying this to our autonomous vessel testing, we must define some key expressions in this formula. A failure event can be defined as the autonomous vessel colliding with another object or vessel resulting. The distance metric d can be described as the closest distance between the autonomous vessel and another object or vessel. This results in the first part of the objective function being to minimise the distance the AV comes from a collision until a collision occurs. Further, the location of the collision does not matter as we only look at the state giving the highest reward, so our primary objective is to minimise the closest overall distance from a collision throughout the simulation.

To find the most likely sequence of disturbances, we can apply the chain rule for Bayesian networks [4]. Looking at the Bayes Theorem, we find that the joint probability of a set of states is given by

$$P(s_0, ..., s_{t_{end}}) = P(s_0)P(s_1|s_0)P(s_2|s_1, s_0)...P(s_{t_{end}}|s_{t_{end}-1}, ..., s_0), \tag{2.3}$$

The Bayesian chain rule tells us that a transition to a new state is only dependant upon the current state. If we assume this to be true, we can make the following simplification.

$$P(s_n|s_{n-1}...s_0) = P(s_n|s_{n-1}) \tag{2.4}$$

Applying this to 2.3, we see that the joint probability of a sequence of states $s_0, ..., s_{t_{end}}$ is given by

$$P(s_0, ..., s_{t_{end}}) = P(x_0) * \prod_{i=1}^{t_{end}} P(s_i|s_{i-1}), \tag{2.5}$$

As we always start in the initial state, we get $P(x_0) = 1$, further simplifying the transition probability to

$$P(s_0, ..., s_{t_{end}}) = \prod_{i=1}^{t_{end}} P(s_i|s_{i-1}), \tag{2.6}$$

The final step in constructing our objective function is to combine both Function 2.2 and Function 2.6.

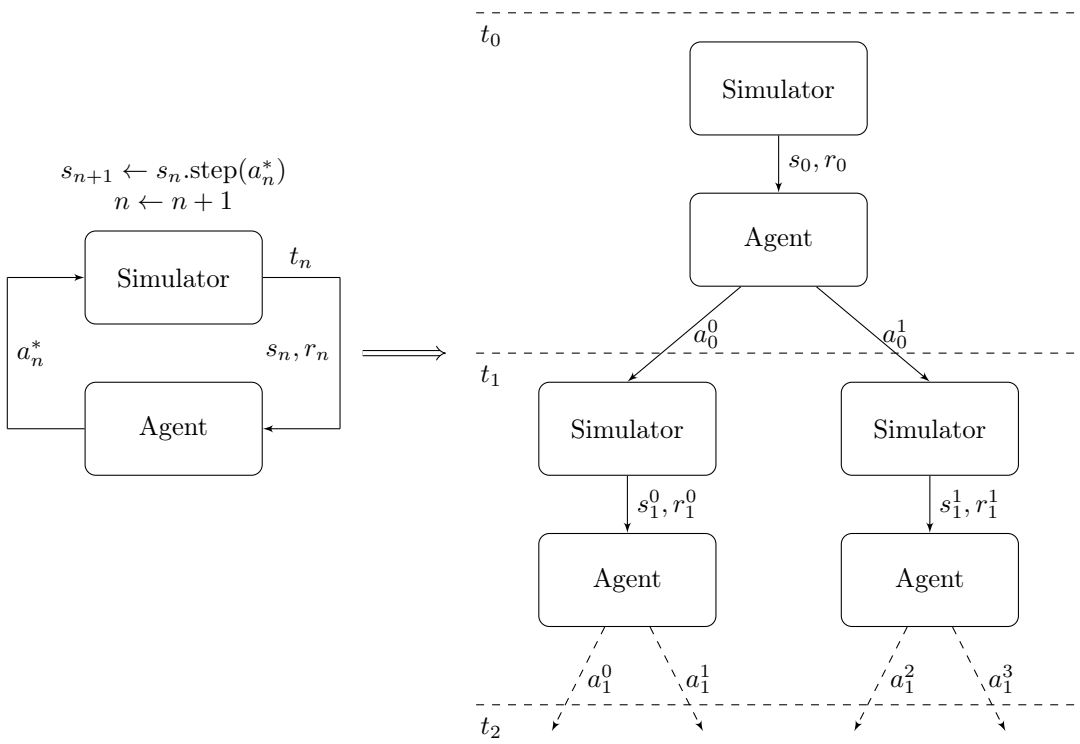
$$\max_{x_0,...,x_{t_{end}}} \left[\prod_{i=1}^{t_{end}} P(s_i|s_{i-1}) + R_e * (s_{t_{end}} \in E) - d * (s_{t_{end}} \notin E) \right], \tag{2.7}$$

resulting is the main objective function of AST. When applied to our naval domain, our objective is to minimise the distance between the autonomous vessel and other objects until a collision occurs. Furthermore, the joint probability of the sequence occurring should be maximised, resulting in the most probable sequence of disturbances that result in a collision.

2.6 Monte Carlo Tree Search

Monte Carlo Tree Search, introduced by Coulom [5], is a reinforcement learner used to find optimal solutions to Markov decision processes. MCTS is based on the Monte Carlo method, randomly sampling rewards of random solutions to a deterministic problem, using the reward as heuristics for future solutions. MCTS achieves this by building a node tree where each branching path down the tree represents different actions a_n^* that form unique solutions to the MDP. This restructuring from the circular MDP restructuring of AST, seen in Figure 2.5, to a tree structure is illustrated in Figure 2.6. In the final tree structure, the state s_n^* is represented by nodes, while the different actions a_n^* are represented by the edges.

MCTS has recently gained popularity after proving its usefulness in solving grid-based games [6]. It is a state-of-the-art heuristic search algorithm within a subset of reinforcement learning algorithms called Q-learning that aims to find optimal solutions to finite Markov decision processes. The common trait between Q-learners is that they do not need a model of the environment, purely estimating the result of actions in different states. To achieve this, a Q-learner builds a state-action value function $Q(s, a)$ that estimates the expected total, or final, the reward of choosing an action a in a state s.



(a) MDP (b) MCTS

Figure 2.6: Restructuring a Markov Decision Problem as tree structure. In a partially observable Markov decision process the agent chooses an action a_n^* altering the environment, before receiving the new state along with a reward. This is the basis of the tree structure in Monte Carlo Tree Search.

To calculate the state-value function $Q(s, a)$ historical data about the traversal of the tree is also stored. Along with what action was taken, a given edge in the tree also contains the number of times that edge has been traversed and the total reward accumulated through that edge. The state-action value function can then be calculated by the average reward passing through a node a.

The tree is built iteratively through four steps, illustrated in Figure 2.7. During selection, the first step of MCTS, the tree is traversed to a leaf node, using a selection policy to select actions, represented by edges to child nodes, to traverse. One such selection policy is Upper Confidence bounds applied to Trees (UCT) introduced by Kocsis *et al.* and [7]. UCT is a bandit-based approach, addressing the exploration-exploitation dilemma not only by selecting actions based on previous results but also by selecting actions that have not been frequently explored. This is done by adding an exploration bias to the selection policy, selecting the action in a node that maximises the following equation.

$$UCT = \max_{a \in A(s)} \left[Q(s,a) + \sqrt{\frac{\log N(s)}{N(s,a)}} \right] \tag{2.8}$$

After traversing the search tree until a leaf node is reached, the leaf node is expanded. Normal expansion is done by adding child nodes along edges with possible actions to the leaf node. A

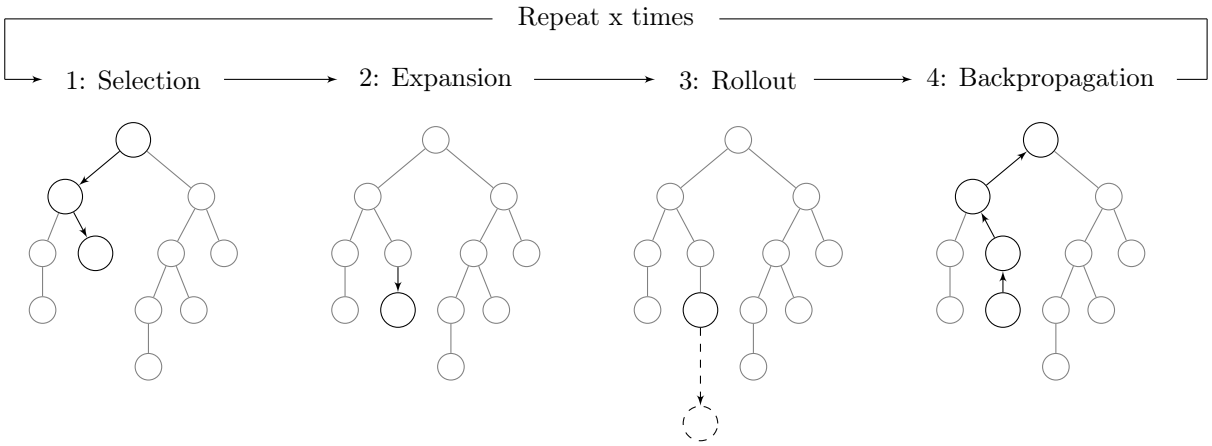


Figure 2.7: The four steps of Monte Carlo tree search.

challenge with this approach arises when the action space is vast or infinite, resulting in it being inadvisable or impossible to add all actions to a state. One proposed solution to this is Progressive Widening, which is presented in Section 2.6.1.

The final two steps are rollout and backpropagation. During rollout, a set of simulations is run and the rewards of the simulations are calculated. During simulations, actions are chosen at each time step based on the rollout policy, which in a traditional Monte Carlo method is equivalent to picking random actions. These rewards are then passed back to the tree during the last step. Here, the number of traversals and the total accumulated reward are updated in the traversed nodes and edges, storing the information for future selection. Nodes representing the states simulated in the rollout are not added to the search tree.

2.6.1 Progressive Widening

When the action space becomes excessively large or infinite, a general implementation of MCTS encounters a problem. If only a small subset of actions is added in the expansion step, there is a risk that the optimal action is not added to the tree, but if too many actions are added, the tree becomes too wide, and the search times grow exponentially.

One solution to this is to implement MCTS with progressive widening (MCTS-PW) [8], [9]. The idea behind MCTS-PW is to expand nodes throughout the selection process, adding more child nodes to frequently visited nodes. By adding more child nodes to frequently traversed nodes, we ensure that the tree is explored to a sufficient depth while still allowing for later optimisation of paths by introducing new nodes along promising paths. The addition of a child node is made if the traversed node meets the following condition.

$$|\bar{X}(s)| < kN(\bar{s})^{\alpha} \tag{2.9}$$

Here, s is the traversed node, $|\bar{X}(s)|$ is its number of children, and $N(s)$ is the number of times s has been traversed. k and α are parameters that control the rate at which new nodes are added. This check is done at every node throughout the selection process, allowing new nodes to be added

along an existing branch, and not only at leaf nodes, like when using normal expansion.

2.7 Simulator observably

Within simulated systems, the observability of internal states can vary from system to system. In this paper, we will study testing on simulators with a limited insight into the workings of the system under test called a black-box simulator [1]. When performing black-box testing of a system under test, we only have access to the input variables and the output in the form of the current state of the environment.

Our approach to testing testing the black-box testing is using reinforcement learning, trying to map the input variables to desirable environment states, measured by an objective function. The objective function will vary from domain to domain, but some of the optimisation processes are the same. This allows us to generalise the framework, allowing for easy implementation on a wide range of domains.

Chapter 3

Related works

In this chapter, we discuss previous work with the main focus being on developments within adaptive stress testing. When discussing domain-specific AST implementations, we focus on the testing of autonomous vehicles, as the domain, and challenges arising from it, is highly comparable to seagoing vessels. In addition to discussing AST, some interesting developments within the naval domain, as well as Monte Carlo tree search, are also discussed to help direct further improvements.

3.1 COLREG compliance

Convention on the International Regulations for Preventing Collisions at Sea (COLREG) is a set of 41 rules introduced in 1977 by the International Marine Organization. COLREG aims to mandate behaviour at sea to prevent vessel collisions. A majority of the rules are based on common sense or directly mandate technical specifications of vessels, and as such are not applicable to autonomous vessels [10]. Rules 11-18, however, directly mandate the *conduct of vessels in sight of one another* with clear instructions on behaviour in order to avoid collisions at sea. Previous work has been done to stress test marine vessels and to introduce the COLREG as a tool for finding failures in autonomous vessel behaviour.

3.1.1 Scenario-based Testing of a Ship Collision Avoidance System

In their article, Porres *et al.* (2020) [11] presents a scenario-based testing system that aims to generate challenging test scenarios to stress the autonomous vessel. The article presents a reward function based on the discussed subset of COLREG rules, rules 11-18, as well as the measurement of the distance between the boats. Here, the distance to the approaching vessel is calculated in relation to the distance to the goal, penalising near misses far away from the goal. By running a simulation and calculating the total reward, the scenario can be evaluated in regard to how well the autonomous vessel under test (ego vessel) complied with COLREG rules in addition to avoiding collisions. In other words, the reward function not only evaluates if the ego vessel avoids collisions but also if the vessel abides by basic naval navigation and yielding rules.

To test the ego vessel, a set of scenarios are generated, and the reward given by the scenario is predicted by a neural network (NN). The scenarios with the highest predicted likelihood of failure

are then simulated, and the rewards are fed back into the NN, making future predictions more accurate. Throughout the search, a set of challenging scenarios are generated, helping pinpoint where the autonomous vessel being tested fails to indicate behaviour in the ego vessel that needs to be improved.

The most interesting part of the paper, in our opinion, is the application of the COLREG rule set. Being able to quantify how well the autonomous vessel adheres to common naval norms, predictability and, in a way, the blame of the collisions can be approximated, allowing the reward function to distinguish between undesirable behaviour in the ego boat and other boats, only penalising the reward of the ego boat.

Although this method shows promising results in uncovering failure states of the given system under test, one should take into account the relatively high failure rate of the system of around 10% given random initial states. We therefore argue that the method should not, however, be considered for use in testing fairly robust systems, as the initial states are generated randomly and not constructed by the learner. As such, the method presented is essentially a random search with a filter, which makes the method ineffective for finding failure events if they are sufficiently rare. If applied to the milliAmpere 2 system, given the robustness of the system, the method is unlikely to perform well.

3.2 Adaptive Stress Testing

Adaptive stress testing (AST) is a testing framework that aims to find the most likely failure event of a system structured as a Markov decision process (MDP) [12]. A reinforcement learner is applied to the MDP and uses previous experiences to effectively search the often very vast state space, finding likely failure events, which can be used to further guide the development and improvement of the system under test. The search is directed by a reward function that rewards collisions and close misses with high transition probabilities.

We have not been able to find previous implementations of AST in a naval setting, but AST has shown promise in a variety of other fields, testing airplane collision avoidance systems [1], [13], autonomous vehicle pedestrian collision avoidance [14], and the robustness of financial systems when it comes to detecting credit card fraud [15]. This section will go through a few applications to illustrate some of the developments made within the AST framework.

3.2.1 Airborne Collision Avoidance Systems

When introduced by Lee *et al.* [1] in 2015, the AST framework was used to find probable failure events that were not counteracted by ACAS X, an airplane collision avoidance system currently in development. To test the system, ACAS X was encapsulated within a simulator, advising a virtual airplane pilot on collision avoidance manoeuvres. During the setup, two or three airplanes were initiated in a series of hand-picked initial states, and stress tested by searching for the defined failure states, near midair collisions (NMACs). As such, the search is done, not by generating sets of initial conditions, but by altering time step conditions, referred to as *disturbances*, throughout the simulation run time, searching for likely sequences of disturbances that lead to failure states.

To search for the most likely failure event, Lee *et al.* applies Monte Carlo tree search (MCTS). MCTS searches thousands of simulations with varying sequences of disturbances to find the likely sequences leading to NMACs. Throughout the search, MCTS directs the search towards sequences, similar to those that have proven the most promising, while also endorsing exploration of less explored sequences. As a result, the framework proved to be effective in uncovering failure states, allowing a comparison of the current collision avoidance system and ACAS X. The results found through the work of Lee *et al.* helped ACAS X become the new international standard for airborne avoidance collision systems.

3.2.2 Autonomous Vehicles

Further developments to the AST framework were done by Koren *et al.* (2018) [14], in an article that applied two versions of the framework in order to test pedestrian collision avoidance systems in autonomous vehicles (AV). In their work, Koren *et al.* implements a simulation containing an AV approaching a pedestrian crossing. The pedestrians are controlled by the AST framework to find paths where the AV is unable to avoid hitting the pedestrians. Where Koren *et al.* differs from Lee *et al.* is by using two different implementations of AST, each with different reinforcement solvers, one using MCTS and the other, referred to as the deep reinforcement learner (DRL), employs a neural network to control pedestrians.

Through three different experiments, Koren *et al.* found that AST using DRL convincingly outperformed MCTS, concluding that DRL finds more likely failure scenarios than MCTS, and it finds them more efficiently. Furthermore, the report also acknowledges the ability of the AST framework when autonomous vehicles with modular components, allowing testing to be pinpointed on a sensor or decision system of a vehicle in simulation.

Something that is not addressed by Koren *et al.* is who is to blame for the failure state found. In the work by Lee *et al.*, all airplanes involved in a failure event are controlled by the system under test, while Koren and Alsaif et al. introduce an external adversarial agent in the form of a pedestrian.

3.2.3 Reward augmentation

In their work, Corso *et al.* (2019) [16] argue that the results found by Koren *et al.* (2018) [14] are mostly uninteresting, as most failures were caused by pedestrians walking into a stationary AV that has already stopped and, therefore, acted correctly in the given scenario. Another problem that Corso *et al.* addresses is the lack of variations in the failure scenarios found by Koren *et al.* Since most of the scenarios found by AST are very similar, Corso *et al.* argues that the method is missing potentially valuable feedback from other failure states.

Their solution is to introduce two new enhancements to the reward functions. The first reward function aims to discourage failure states not caused by the AV by introducing a negative reward for every time step when the AV is not acting within the Responsibility-Sensitive Safety (RSS) policy. RSS is a set of mathematical functions that define what a car should do given its position and speed in relation to the position and speed of other vehicles or obstructions. The resulting reward function allows the search to be directed towards scenarios where the AV is not reacting properly to the movement of the pedestrian, resulting in partial blame for the collision.

The second reward function introduced is a *Trajectory Dissimilarity* (TD) reward. The reward function compares failure trajectories, or disturbance sequences, rewarding the sequences that are the most dissimilar compared to other sequences. This effectively discourages crowding, resulting in a greater variety of failure states.

After the implementation of the two reward functions, a short experiment is run, resulting in multiple failures caused by the AV not stopping. Cosco *et al.* concludes by arguing that studying the wide variety of what they call *the most relevant failure cases* can better aid in the validation of autonomous vehicles.

3.3 Monte Carlo Tree Search in sequential games

A huge part of the recent increase in popularity of Monte Carlo Tree Search is largely contributed to its use in deterministic, sequential board games. In this section, we introduce some developments of MCTS in game AI, as they lay the foundations of a lot of the choices made in our proposed model.

3.3.1 AlphaZero

AlphaZero is a generalised framework that has made substantial advances within *Computer Chess* and *Shogi* (a Chinese relative of chess) [17] as well as *Computer Go* [18] where AlphaGo, a predecessor of AlphaZero, became the first computer Go program to beat a professional human player [19].

Though not directly related to the stress testing of marine vessels, an insight into the AlphaZero framework might be interesting, as it revolutionised the use of MCTS within large state space. It should also be noted that AlphaZero is a complex framework, so the following section will only briefly address some core concepts that may apply to AST.

AlphaZero is based on the use of Convolutional Neural Networks (CNN) in conjunction with Monte Carlo Tree Search (MCTS). Unlike most other engines using domain knowledge and handcrafted expert reward models, AlphaZero has zero human expert input but builds its own policy through self-play, optimising the likelihood of a win. The framework starts by initially playing random games against itself, building its own policy by learning how to routinely win against each iteration of itself.

To achieve this, the framework operates on a few different core concepts. The first is the simulation-based, look-ahead (or rollout) search used through MCTS. Using statistical knowledge of previously played games, MCTS searches moves with an estimated high likelihood of wins or moves that have not been sufficiently explored. The goal is to map out the moves resulting in favourable future games of a given action while directing the search away from moves that are likely to be unfavourable for the player, allowing a thorough search through an otherwise immense search space that would be impossible to search extensively.

However, only using MCTS is not enough to achieve superhuman performance in chess, Go, or Shogi. The second part added to AlphaZero is a CNN used as *policy network*. The basic idea of this addition is that by storing previous experiences in the policy network, the rollout performed

to estimate the values of a given action will improve. A better estimate of actions will improve the quality of AlphaZero games, which, in turn, improves the policy network and its estimations. By allowing AlphaZero to play against itself, the policy network should improve incrementally, in turn improving the model as a whole.

Throughout learning, the AlphaZero framework plays multiple consecutive sets of finished games. Every chosen move in a game is based on a newly generated search tree running multiple rollouts. evaluating possible moves. After the game, all generated trees are used to train the policy network. As a new search tree is constructed for each game, all estimates in the tree are based on the newly improved network policy, allowing for a quick turnaround.

Chapter 4

Method

This section introduces our implementation of the AST framework, presented in Lee *et al.* (2020) [13]. The implementaiton of AST is done through the three main parts of our architecture which we call Flexible Aadaptive Stress Testing (FAST). The FAST architecture, illustrated in Figure 4.1, is divided into three main components, namely *Reinforcement learning agent*, *Simulator interface*, and *Simulator*.

This section also introduces Monte Carlo Forrest Search with Action Prunin (MCFS-AP), a reinforcement learner designed and implemented for the FAST architecture. The main additions to MCFS-AP, over previous MCTS methods applied the AST framework, are the approach to search tree construction and the use of neural networks when selecting actions during the selection and rollout process as well. Our goal is to implement a generalised version of the AST framework that is able to find more failure states more efficiently. This section will discuss our implementation of the three components of FAST. The complete implementation can be found on GitHub[1].

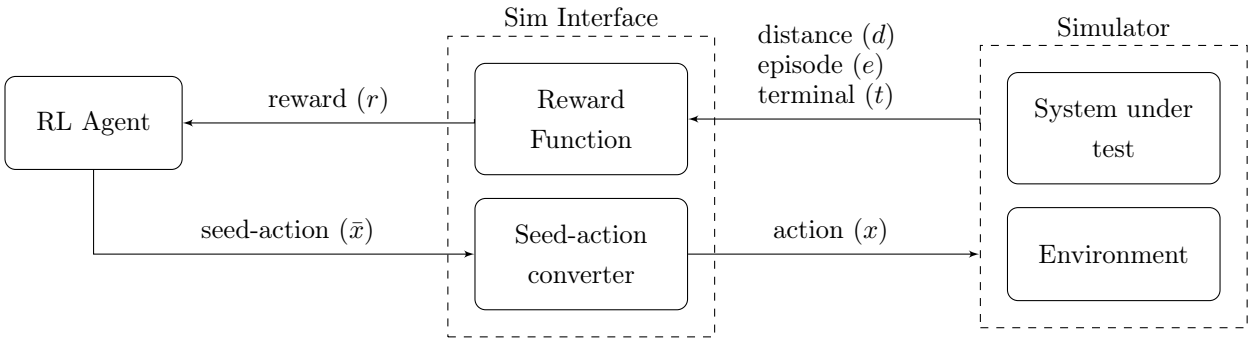


Figure 4.1: Flexible Adaptive Stress Testing (FAST). The Simulator class contains a simulated environment as well as the system under test while the reinforcement learning (RL) agent that is controlling the simulator based on previous rewards. To allow a generalized reinforcement learner and a given Simulator to communicate the Simulator Interface is implemented, translating seed-actions of the reinforcement learner into disturbances for the simulated environment, as well as calculating the reward of the executed action.

[1]The FAST architecture using MCFS-AP Python fully implemented in Python:
https://github.com/bragelyt/Flexible-Adaptive-Stress-Testing

4.1 Simulator

There are two main goals of this paper. The first goal is to implement and test our own version of the AST framework, while the second goal is to stress test the autonomous ferry, milliAmpere 2. To achieve these two goals, two separate simulators are used. To test the efficiency of the implementation, a simple simulator containing a predictable system and interpretable environment was implemented. For stress testing of the milliAmphere 2 ferry, a slightly altered version of a complex simulator supplied by Zeabuz is used. This section will discuss the general requirements for a simulator as well as the implementation of the two simulators used in our experiments.

4.1.1 Time step simulator

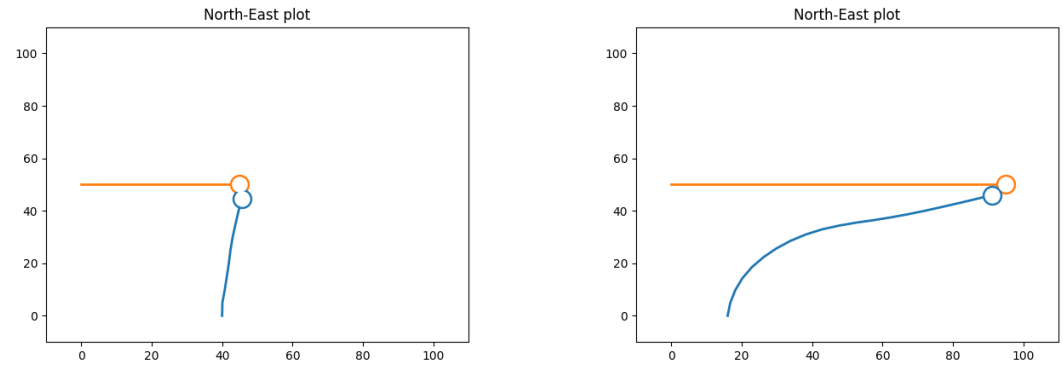
To apply a reinforcement learner to the simulator, we structure the system as a Markov decision process, implying that the simulator should advance through discreet state transitions. This is achieved by implementing a time step simulator, where the progression of time is divided into discrete time steps of equal length. Furthermore, all transitions between states should be deterministic given a specific disturbance and state. In other words, when given the same initial state and a given set of disturbance the system under test and environment should change in the same manner, resulting in the exact same set of states at every time step. Although more probabilistic simulators can work, deterministic and noiseless simulators have been shown to work well with AST [14].

When using a deterministic time step simulator, any previously explored state s_n can be represented as a unique sequence of disturbances $[x_1, ..., x_n]$. The given disturbance sequence is the set of disturbances that was originally applied to the simulator at each time step from the initial state to the given state s_n. Revisiting the state s_n is done by retracing the disturbance trace, stepping through each disturbance $[x_1, ..., x_n]$ from the initial state in the simulator. As the deterministic nature of the simulator calculates the same state transition given the same disturbance, we arrive at the desired state s_n.

4.1.2 The Interpretable simulator

To test the effectiveness of the AST system, we first need to implement a system in an environment to be stress tested. Using the complex Zeabuz simulator however, might introduce uncertainties and make implementation, troubleshooting, and understanding the AST method unnecessarily time-consuming. As a result of this, a simple simulator was implemented, containing a predictable system and an interpretable environment. We will now discuss this simulator.

The goal when determining the setup for this simpler simulation was to eliminate any noise and uncertainty, resulting in a low-fidelity scenario with two boats at a fixed speed on a flat plane without other obstacles or disturbances. The system under test (ego vessel) is a vessel unaware of its surroundings moving across the plane from left to right, in a straight line. The second vessel (adversarial vessel) starts at the bottom of the plane, facing upward. The heading of the adversarial vessel, given as the disturbance, is altered by the reinforcement learner searching for a failure event where the two vessels collide.



(a) Easy initial condition

(b) Harder initial condition

Figure 4.2: Simple simulators with different failure rates. Figures present near optimal solutions for the simple simulator with different initial conditions. The adversarial vessel and its path is given by the blue circle and line, while the ego vessel and its path is shown in orange. The shift in starting position from (a) to (b) makes it harder for the ego vessel to catch up to the adversarial vessel.

To vary the probability that a failure event occurs in a random simulation, the starting point of the adversarial vessel can be altered. Setting the initial position to a position less advantageous for the adversarial vessel results in a lower margin of error throughout the disturbances chosen to generate a collision state and a lower failure rate for the ego vessel. This in turn results in a more difficult search, as more simulations must likely be sampled in order to find a falure event. A visualisation of two setups with varying probability for failure events is shown in Figure 4.2

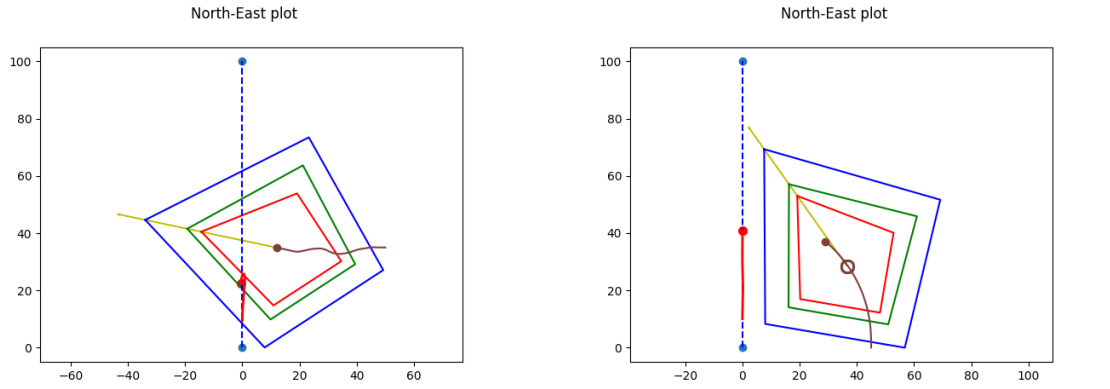
The reward function for the simple simulator is set through defining the complete set of failure events E, the distance d a state is from a failure event, and the transition probability $P(s|x)$ between states. We define d as the Euclidean distance between the ego and the adversarial vessel, and E as all states where d is less than a threshold value d_{min}. Finally, the transition probability between states is calculated by how similar a two subsequent disturbances are.

As step rewards are given by $\log(P(s|x))$, the optimal solution should be a smooth and direct curve that intercepts the ego vessel, in as few steps as possible. Although this is a huge oversimplification of the probable heading of a vessel, it allows for a more understandable results through the simplicity of the transition model and intuitive optimal solution.

4.1.3 Zeabuz simulator

The second simulator is supplied by Zeabuz and is used to stress test milliAmphere 2 The simulator, shown in Figure 4.3, encapsulates a milliAmphere 2 ferry and its desired path alongside a variable set of other vessels at sea. The simulator is continuous, where the started simulator does not pause before reaching a terminal state. In order to apply AST to the simulator, discrete time steps is implemented, allowing AST to introduce new disturbances at every time step. The length of a simulator varies depending on the scenario, but will typically last between 25 and 50 timesteps.

A set of disturbances was implemented for the reinforcement agent to tune while searching for failure states. The first disturbance is to control the heading of the adversarial vessels. To achieve



(a) Zeabuz simulator with heading of the adversarial vessel being used as disturbance, while sensor noise and delay is removed.

(b) Zeabuz simulator with delay of perceived position of adversarial vessel being altered, while its path is predetermined and constant.

Figure 4.3: Visualizations of ongoing Zeabuz simulations. The position and previous path of the autonomous ferry is shown as a red circle and tail, going along a dotted blue line showing its desired path. The path of an adversarial vessel is shown in brown while the yellow line shows current perceived position and heading. When delay is added, the perceived position is shown as a brown circle. The perceived position is to calculate safety zones indicated by the three squares surrounding the adversarial vessels. Safety zones should not be crossed by the autonomous ferry.

this, a steerable vessel class was implemented, allowing different headings to be manually set at each time step throughout the simulation. This class allows headings to be used either directly as disturbance or via a predefined set of headings to set the same route for adversarial ships for each simulation.

The second form of disturbance implemented is noise in the position data provided to the ColAv system. This noise function heavily builds on an existing noise generation function already implemented in the simulator. Our implementation overwrites this method and forces the generated noise to reflect the noise vectors given by AST at every time step. This method was later used to alter the position of the adversarial vessel within a given threshold in order to find failure events.

The final disturbance is altering the delay in the perceived position of the adversarial vessel, given by a delay function. Here, the noise function is used to revert the perceived position to a previous state. By adding a noise vector equaling the difference between the current and a specified previous state allows the perceived position and heading to be altered, matching the given previous state.

Overall, the Zeabuz simulator is a much more realistic simulator than the simple simulator, potentially giving us more interesting and applicable results from the simulations. The downside of this realism is that each simulation is more computationally costly. This results in an increase in the simulation time from 2.2 ms, using the simple simulator, to 1 s using the Zeabuz simulator, a 450-time increase.

4.2 Simulator Interface

One of the main strengths of AST is the versatility of the method. An important aspect that allows for this is the black box testing approach to the simulator. The reinforcement learner has minimal knowledge of the functionality of the system under test, only mapping the input values to the output rewards. As a result of this, there is only a small predetermined set of functionalities that AST needs to access from the simulator to search for failure states. On the basis of this, we have chosen to create an abstract simulator interface class as a central part of the FAST architecture. A version of this class is implemented for each simulator to handle the translation between the generalised reinforcement learner and the domain-specific simulator, based on the predetermined set of functions required by the reinforcement learner. This allows both the reinforcement learner and the simulator to remain unchanged, resulting in a flexible model, allowing us to easily swap between different agents or simulators.

As seen in Figure 4.1, the two main tasks of the simulator interface class are to translate the generalised seed-actions of the reinforcement learner into actionable values, and to calculate the state reward given the resulting state of the simulator, as well as previous states and actions. Implementing the simulator interface removes the need for domain-specific interpretations within AST while allowing for a minimal amount of alterations to the simulator. The simulator interface is implemented with the following five functions:

- *Initiate()* constructs the simulator within the sim interface.
- *ResetState()* sets the simulator back to the initial state.
- *Step($\bar{x}$)* takes a seed-action $\bar{x}$, executes the adjoined action on the simulation, and returns the reward generated by the action. The reward function is located in the simulator interface, as the reward function will vary between different simulators and use cases. Separating the reward function from the main MCTS-SA class allows easier implementation with other simulators.
- *IsTerminal()* returns true if the simulator is currently in a terminal state, false if not. Terminal states occur when a failure event has occurred or if the maximum number of time steps has been reached.
- *GetTerminalReward()* returns the the terminal reward of the current simulator, given that it has reached a terminal state. Returns the shortest distance d throughout the simulation, or the failure reward R_e if a failure state has occurred.
- *SetState($\bar{x}_0, ..., \bar{x}_n$)* sets the simulator to the state given by the input seed-action sequence $\bar{x}_0, ..., \bar{x}_n$.
- *GetState()* returns the seed-action trace $[\bar{x}_0, ..., \bar{x}_n]$ leading up to the current state. The state is stored as a seed-action sequence within the simulator interface. Starting at the initial state and stepping through the given seed-action sequence should result in the simulator reaching the current state.
- *RememberState()* and *RecallState()* are only required when implementing state recall. *RememberState()* saves the internal simulation state, while *RecallState()* takes the remembered state and sets the simulator directly to that state, without stepping through the seed-actions trace. These replace *SetState($\bar{x}_0, ..., \bar{x}_n$)* and *GetState()*

- *GetNNReresentation* is only required if the neural networks are used in the reinforcement learner. This function returns a representation of the state suitable as the input for the neural network, which varies between domains. In our case, the current coordinates and heading of vessels are returned.

4.2.1 Reward function

The reinforcement calculation is one of the most integral parts of the simulator interface, as the reward is the heuristic that directs the search of the reinforcement learner. The reward function used in our experiments vary dependent on what disturbances are being used, but they are all heavily based on the objective function presented in Chapter 2.5.2. The only alteration done to the objective function is the calculation of the joint transition probability.

When looking at the objective function, given by Equation 2.7, the transition probability p is given by $\prod_{i=1}^{t_{end}} P(x_i|x_{i-1})$. In our experiments this function is altered in a few ways. The main difference is altering the function from a product to a sum, where we now calculate

$$\bar{P}(s_0, ..., s_{t_{end}}) = \sum_{t=1}^{t_{end}} \log(P(s_t|s_{t-1})). \tag{4.1}$$

Although the outputs of the two functions are different, the results of maximising the two are equivalent [13]. The change is done to more severely punish highly unlikely sequences. While a highly unlikely sequence of disturbances would return 0 given the original function, the sum function returns a strictly negative reward where the reward of sequences that are highly unlikely goes towards negative infinity, strongly emphasising the fact that they should not be further explored.

The final modification of the function makes the calculation of the state transitions more interpretable. As the state transition within the system under test is deterministic, the state transition can be calculated based on the likelihood of a disturbance occurring and not by the difference between two states, which in a black-box simulator is hard to interpret. As the state s_t is deterministically given by a disturbance x^*_{n-1} occurring in the state s_{t-1} we can rewrite Equation 4.1 as

$$\bar{P}(s_0, ..., s_{t_{end}}) = \sum_{t=0}^{t_{end-1}} \log(P(x_t|s_t)), \tag{4.2}$$

Combining Equation 4.2 with the rest of the objective function, given by Equation 2.7, the reward r of applying a disturbance x in a state s is given by

$$r(s,x) = \begin{cases} \log(P(x|s)) & \text{if } s \text{ is not terminal} \\ r_e & \text{if } s \text{ is terminal and } s \in E \\ -d & \text{if } s \text{ is terminal and } s \notin E \end{cases} \tag{4.3}$$

where E is the set of all failure events, r_e the positive reinforcement given by a failure event, d the closest distance to a failure event, and $P(x|s)$ the transition probability calculated by the simulator

interface. To calculate the total reward R of a complete simulation, the sum of the reward r of each state transition throughout the simulation is calculated, given by Equation 4.4.

$$R = \sum_{t=0}^{t_{end}} r(s_t, x_t), \tag{4.4}$$

Maximising R results in the achievement of the objective function of AST. By searching for a sequence of disturbances that cause the system to end in a failure event, while also refining the search to maximise $\log(P(x|s))$, we should end up with a result giving the most likely failure event.

4.2.2 Calculating transition probabilities

When calculating transition probabilities, we rely on the Markov assumption, introduced in Section 2.2. When defining the transition probabilities between states, some alterations of this assumption are made with respect to how disturbances develop over time. In our work we use two disturbances, steering and delay. To calculate the two differences we define two types of transition possibilities, of which the disturbances either partly or fully fall into. We call these conditional and unconditional transition probabilities.

Conditional transition probabilities are the disturbances in which we assume that the Markov chain rule is in force. These disturbances have an internal moment of inertia, supporting our assumption that the current disturbance value x_t is dependent on the previous disturbance value x_{t-1}. In our experiments we define the steering of the vessel as a disturbance with conditional transition probabilities. When observing the alterations in the heading of a vessel between consecutive states, we assume that the new state should not vary much from the previous state. Although being a vast oversimplification of how a human-controlled vessel moves, the assumption does capture some of the behaviour of a vessel.

When calculating a conditional disturbance, we calculate the Euclidean distance between the current and the previous state before normalising the result. In order to achieve this, we can take advantage of our representation of the current state being the sequence of seed-actions leading to that state. As our seed-actions are a pseudo-random float between 0 and 1 that are being linearly mapped to a given disturbance, the transition probability for a conditional disturbance is given by

$$P_c(\bar{x}_t|s_t) = 1 - |\bar{x}_t - \bar{x}_{t-1}|, \tag{4.5}$$

where $\bar{x}_{t-1}$ is given by s_t. Here, two similar disturbance values given by seed-actions $\bar{x}_t = \bar{x}_{t-1}$ return a transition probability of 1, while two completely opposite disturbance values, $\bar{x}_t = 1, \bar{x}_{t,1} = 0$, return a transition probability of 0, heavily penalising the sudden change in disturbance value. When applying this to a simulation using the heading of the adversarial vessel as the disturbance, we will search for the sequence of states ending in a crash with the least amount of abrupt changes to the heading.

The second form of disturbances is unconditional disturbances. Unlike conditional disturbances, the values of these disturbances are stochastic and are randomly distributed, without taking previous states into account. In our experiments we assume that the delay is partly unconditional, as the amount of noise in a given time step is highly unpredictable.

Calculating unconditional disturbances is done using seed-action values. Although there is a bit more variation depending on the distribution of the disturbance values, a simple implementation assuming a normal distribution of disturbance values is given by

$$P_c(\bar{x}_t|s_t) = 1 - |c - \bar{x}_t|, \tag{4.6}$$

where c is the most likely disturbance value. As all unconditional disturbances implemented in our experiments are distributed with a median value at 0 we get a function where the higher the disturbance value is, the lower the transition probability. When using noise as a disturbance, this results in us searching for the sequence states, resulting in a crash where the least amount of sensor noise has been applied.

4.3 Our reinforcement learning agent

The final component of the FAST architecture is the reinforcement learning agent. In this chapter, we introduce our implementation of the Monte Carlo Tree Search, which we call Monte Carlo Forest Search with Action Pruning (MCFS-AP). MCFS-AP differs from MCTS by search using a set of smaller trees, aggressively pruning the action space at each tree through action selection. This is done by periodically locking in the most promising action from the root, ignoring all other actions in subsequent searches. MCFS-AP is implemented with progressive widening, using rollout policy and value networks during the rollout and selection steps, respectively. The base algorithm is based on the MCTS-PW algorithm presented by Lee *et al.* [13], with the addition of neural networks inspired by the AlphaZero framework presented by Silver *et al.* [17].

4.3.1 Monte Carlo Tree Search with Seed-Actions

MCFS-AP is based on the Monte Carlo Tree Search with Seed-Actions (MCTS-SA) model presented by Lee *et al.* [13]. MCTS-SA uses progressive widening to build a single search tree in a continuous search space. The first alteration from MCTS-SA is that we divide MCST-SA into the four stages of an MCTS search loop. This separates the nodal operations from the tree construction, allowing for easy implementation of different variations of MCTS and MCFS.

4.3.2 The search loop

The basis for all versions of the MCTS presented in this thesis is the four search stages, briefly presented in Section 2.6. Throughout the four stages, we traverse the tree, check if we want to add new possible actions at a node, simulate a complete disturbance sequence before calculating the reward of the sequence, and update the nodes traversed with the new reward. In this section, we will study how each step is implemented in MCFS-AP and what design choices differentiate MCFS-AP, MCTS-SA, and traditional MCTS. The search loop is implemented in Algorithm 1 through the function LOOP(s_{root}) implemented on line 27.

- **Selection:**

  The first stage in the MCTS search loop is to traverse the existing node tree to find and a leaf node. When traversing the tree, we are only visiting nodes that contain disturbance values that have already been part of a previous simulated disturbance sequence. Selection is performed iteratively, starting at the root node and selecting a disturbance to arrive at a child node of the root node. The chosen child node is then set as the current node before one of its children is chosen and set as the current node, restarting the process. As all nodes have been previously visited, we can use the rewards gained through the children of the nodes to evaluate the children. This evaluation is given by the state-action value function $Q(s, x)$, which is the average reward that resulted in the selection of a disturbance x in a state s.

  Choosing which child to traverse is not solely based on the state-action value function $Q(s, x)$, but using UCT selection. The idea behind UCT selection is the introduction of an exploration coefficient, weighing the selection in favour of children that have been traversed fewer times. MCFS-AP differs from MCTS-SA by introducing a value network, which predicts the state-action value Q of a node during UCT selection. This is discussed in more detail in Section 4.3.5.

  The implementation of the Selection step starts at line 30. During this loop, we iteratively apply UCT to a node, traverse the chosen child, and then set the child as the current node. This is repeated until we reach a leaf node in the search tree, either as a result of the node not being previously visited or because the node is in a terminal state in the simulation.

  Throughout the selection process, we keep track of the previous transition probability and, upon arriving at a leaf node, store the transition probability for the last disturbance within the leaf node. In this way, all nodes keep track of their respective transition probability, which are later used in the backpropagation process.

- **Expansion:**

  An alteration from traditional MCTS made in both MCTS-SA and MCFS-AP is the implementation of progressive widening [8]. As we are operating in a continuous state space, the number of possible disturbance values, and in turn children, in a node can be virtually infinite. As a result, we cannot add all children at a node after selection, but we have to progressively add more children as needed.

  The progressive widening of nodes is done in parallel with the selection process using, checking if a node should be expanded when it is traversed. Here, the progressive widening criteria, given by Equation 2.9, is checked, and a child node with a pseudo-random seed-action is added to the current node. This is done before the UCT check, as the new node is highly likely to be explored due to its high exploration bias.

- **Rollout:**

  The rollout rollout process is implemented from line 37. The rollout is performed by selecting a set of disturbances using our rollout policy and stepping through the simulation with the set of disturbances, until a terminal state is reached. A cumulative transition probability is calculated, which contains the step reward for every disturbance applied to the simulation through the rollout. Finally, the complete rollout reward is calculated by adding the terminal-state reward to the cumulative transition probability.

  MCTS-SA uses a random rollout policy, picking random actions to be applied to the simulation until a terminal state is reached. This is then performed a number of times before

the average reward gained through all simulations is returned. As we are using relatively time-consuming simulations, we want to avoid this approach in MCFS-AP, instead opting for a rollout policy neural network, discussed in more detail in Section 4.3.5.

The rollout policy is trained on previous rewards to give predictions on what disturbances are desirable during rollout. This, in turn, results in better rollouts that give more direct evaluations of child nodes, as there is less deviation in rollouts, reducing the amount of rollouts needed for the evaluation to converge.

- **Backpropagation:**

  Backpropagation is the last step of the search loop in MCTS, dedicated to updating the node evaluations of the nodes traversed during the current loop. After calculating the reward accumulated by the rollout from a leaf node, we step backward through the traversed nodes, updating the number of traversals and the average reward, using the rollout reward and the transition probabilities stored in each node. Throughout the rollout, implemented in Algorithm 4, the step reward of the current node is added to the total expected reward and the evaluation of the node is updated. This is repeated by setting the parent node as the current node until we reach the root node.

  After updating rewards up to the initial root node, the total reward accumulated through the rollout and backpropagation is returned back to the search handler. This loop is then repeated a number of times before the search handler returns the best reward and disturbance trace found throughout all loops.

Tree node objects

A large structural difference between MCFS-AP and the MCTS-SA algorithm presented by Lee *et al.* is the introduction of tree node objects that keep track of the current search tree, replacing lookup tables. Tree node objects contain pointers to adjacent nodes and statistics of the encapsulated state in the form of current and predicted rewards and traversal statistics. As MCTS is an iterative process, we only access sequential nodes. Having a linked node structure results in us only having to search for the desired node in a list of neighbouring nodes, instead of searching through the complete tree, improving run time.

4.3.3 Action pruning and State recall

A substantial change between MCFS-AP and algorithm presented by Lee *et al.* [13] is our choice to periodically select actions early in the action sequence and actively pruning branches of the tree that are not performing as well as the main branch. This method is based on the use of MCTS in sequential games, where executing an action invokes a new state, limiting the reachable states. Implementing this in MCTS is done by periodically picking a child node of the current root node, setting it as the new current root, and pruning all other branches connected to the previous root. After choosing the new current root, subsequent searches start from the given root, resulting in a more shallow and aggressive search.

The main advantage of action pruning is that by periodically making the search more shallow, we can stop calculating the earlier parts of the simulations. We call this state recall. Given a deterministic simulator, the calculated state of a chosen sequence of disturbances is constant,

removing the need for the simulation to rerun the sequences of the simulations that have already been calculated. However, MCFS-AP does move slightly away from the black-box simulation idea behind our implementation of AST, as we need to be able to initiate the simulation in any given state and not just the initial state.

Action pruning is implemented in Algorithm 1 in the function SEARCH starting at line 3. During the search function, a series of n_{trees} search trees are constructed, each starting in the initial state of the problem. After a set of n_{loops} searches have been performed, the root child nodes are examined, before defining the child most traversed as the new root, implicitly pruning the other root child nodes. After stepping the simulator to the state represented by the new root, we store the simulator's internal state. A new set of n_{loops} is performed, each starting from the new root state, set directly using the saved internal state. This loop is then repeated n_{depth} times, periodically moving the current root further down the search tree.

4.3.4 Monte Carlo Forrest Search

A huge challenge within reinforcement learning is the exploration vs. exploitation trade off, which is the balancing act of sufficiently exploring the search space, while also having to refine the assumed optimal solution. If the solutions found are not sufficiently exploited we will not converge at all.

Action pruning is an exploitative approach to search, as we early on decide to stop exploration of early time steps opting to rather refine the best solution found. To address the exploration vs exploitation we introduce Monte Carlo Forrest Search (MCFS), building multiple, small aggressive trees, forcing diversity in order to introduce exploration. The goal of this approach is to aggressively converge at multiple different local optima, in the hopes that the best local optima found is close to the global optima.

4.3.5 Neural networks

To further improve the effectivity of the search for failure episodes, we have implemented a policy network and a value network, inspired by Alpha Zero [17]. The goal of the networks is to use experiences from previous iterations to better direct the search for failure states.

This section will discuss the motivation behind the neural networks, and their general structure. The implementation of the actual networks are not discussed in detail, as they are mainly ment as a proof of concept. The networks are implemented using PyTorch, and can be further examined on GitHub[2].

Rollout policy network

As discussed in Section 2.6, rollout is normally performed based on the rollout policy given by the model. Both traditional MCTS and MCTS-SA use a random rollout policy, executing random seed-actions until reaching a terminal state. In MCFS-AP we replace the random policy with a rollout policy network π_r that uses previous experiences to pick advantageous seed-actions, giving rollouts

[2]Neural network implementation using PyTorch:
https://github.com/bragelyt/Flexible-Adaptive-Stress-Testing/tree/main/models

that terminate with higher rewards. The construction of a rollout is implemented in Algorithm 1 from line 37 with the rollout policy network π_r selecting the rollout seed-action at line 39.

The rollout policy network gives a prediction of what action is most advantageous given the input state. This is achieved though splitting the seed-actions into discreet ranges and predicting a seed-action range distribution, giving a higher weight to more promising seed-action ranges, in the given state, and using roulette to pick a seed-action value range. A seed-action is then randomly chosen from within the seed-action range.

Our implementation of the rollout policy network takes an input vector consisting of the coordinates x and y, as well as the angle θ, of each vessel in the simulation. The target value is a distribution of seed-actions, given by a list of length k where an element d_n is the relative frequency of seed-actions $\bar{x} \in \left[\frac{n}{k}, \frac{n+1}{k}\right]$. The target distribution in a state is found by getting the distribution of traversals of the seed-actions out of the node representing the given state.

Value network

The other neural network used is a value network, which aims to improve UCT selection. This is approached by using previous results to predict the expected reward of different seed-actions in a given state. The predicted reward is added to the UCT selection formula, resulting in Equation 4.7.

$$UCT(s, \bar{x}) = \max_{\bar{x}} \left[Q(s, \bar{x}) + c_1 \sqrt{\frac{\log N(s)}{N(s, \bar{x}) + 1}} + c_2 V^{\pi}(s, \bar{x}) \right] \tag{4.7}$$

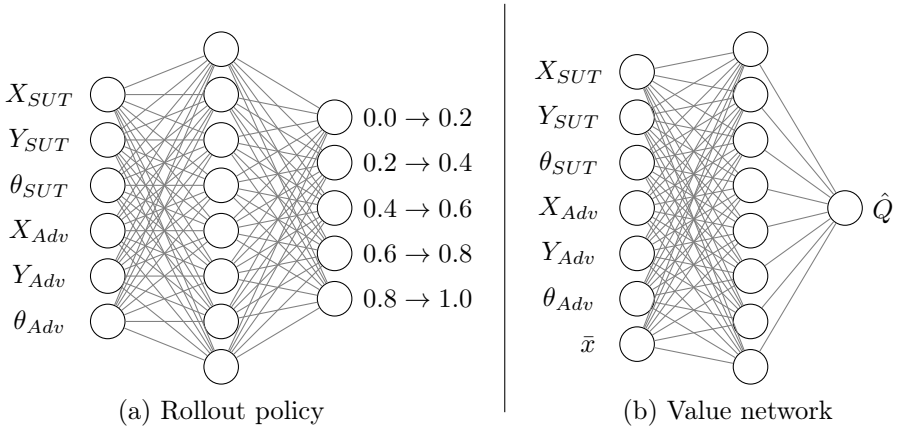
The neural network takes an input vector similar to the rollout policy network, but with the addition of the seed-action chosen in the given state. The target value of the network is the current expected result $Q(s, \bar{x})$ of the given state.

Both networks were trained at the end of each fully explored tree, given in Algorithm 1 at line 24. As we are building multiple trees using action pruning, the statistics at the current root of each depth are used for training. The reason behind this is that root nodes are likely to be frequently traversed, providing the most accurate predictions. Training was carried out taking the state represented in a rote node, setting the positional data as input, and the statistics of the node as target values, before using propagation to tune the network weights. A simplified network model that highlights the input and output layers is illustrated in Figure 4.4.

It should be noted that the fact that both policy networks use the positional state of the vessels might limit the method somewhat, as it once again moves the method away from a pure black-box approach. If this information on positional data is not available, other representations of the state could be used. This was not explored in this work, as access to the positional state of the vessels was not a problem with the simulators we used.

4.4 Summary of our model

The resulting model used for the experiments is an implementation of the FAST architecture, with a simulator interface implemented for each of the two simulators used. The simulator interfaces,



(a) Rollout policy (b) Value network

Figure 4.4: A simplified model of the value and policy network used in our model.

discussed in Section 4.2 allows a single implementaiton of the reinforcement learner to be applied to both simulators, through converting seed-actions into disturbances and calculating the rewards of full simulations.

The main reinforcement learner agent implemented is MCFS-AP. MCFS-AP differs from the MCTS-SA model [13] by introducing Monte Carlo Forrest Search, presented in Section 4.3.4, constructing multiple smaller search trees. Action Pruning with state recalling, presented in 4.3.3, is also introduced in order to make the construction of each search tree faster and more aggressive.

The implemented MCFS-AP model also has the possibility of enabling two neural networks, guiding the search. The first neural network is a rollout policy network, presented in Section 4.3.5, which replaces the random rollout policy of MCTS-SA with a policy based on previous searches, performing more optimal rollouts. The second neural network is a value network, presented in Section 4.3.5, which is introduced during UCT selection, predicting the evaluations of an actions in a given state.

Algorithm 1 Monte Carlo Forest Search with Action Pruning

▷ Input: Seed-action simulator interface $\bar{S}$, MCTS class
▷ Returns: Seed-action trace $\bar{X}^*$ with the highest reward R^*
function SEARCH($\bar{S}, n_{trees}, n_{depth}, n_{loops}$) ▷ **Complete search**
    $R^* \leftarrow -\infty$
    $\bar{X}^* \leftarrow \emptyset$
    $\bar{S}.initiate()$
    for 1 **to** n_{trees} **do** ▷ Number of searches
        $s_{root} \leftarrow$ NEWNODE($\emptyset$, None) ▷ Create new root node
        $s_{root}.traversals \leftarrow 1$
        $\bar{S}.reset()$
        for 1 **to** n_{depth} **do** ▷ Periodic root pruning
            for 1 **to** n_{loops} **do** ▷ Root exploration
                $\bar{S}$.recallState() ▷ Recall sim state to current root state
                $R, \bar{X} \leftarrow$ LOOP(s_{root})
                if $R > R^*$ **then** ▷ Store best reward and sequence
                    $R^*, \bar{X}^* \leftarrow R, \bar{X}$
            queueForTraining(s_{root}) ▷ Neural net training queue
            $s_{root}, \bar{x} \leftarrow$ PRUNEROOT(s_{root})
            if $\bar{x}$ is None **then** ▷ If terminal state
                break
            $s_{root} \leftarrow s_{root}$.child($\bar{x}$)
            $\bar{S}.step(s_{root}.action)$ ▷ Action Pruning
            $\bar{S}$.rememberState() ▷ Update current root state
        trainNetworks()
    return $R^*, \bar{X}^*$

function LOOP(s_{root}) ▷ **Single traverse of node tree**
    $s_n \leftarrow s_{root}$
    $p \leftarrow$ None
    while $s_n.traversals > 0$ and not $\bar{S}$.isTerminal() **do** ▷ **Selection**
        progWiden(s_n) ▷ Add child if PW condition met
        $s_{n+1}, \bar{x} \leftarrow \mathrm{UCT}(s_n)$
        $\bar{X} \leftarrow \bar{X} \cup \bar{x}$
        $p \leftarrow \bar{S}$.step($\bar{x}$) ▷ Transition probability of step
        $s_n \leftarrow s_{n+1}$
    $s_n.stepReward \leftarrow p$ ▷ Transition prob stored in new node
    $P_{rollout} \leftarrow 0$ ▷ **Rollout**
    while not $\bar{S}$.isTerimanl() **do**
        $\bar{x} \leftarrow \pi_R(\bar{S}.state)$ ▷ Rollout policy prediction
        $P_{rollout} \leftarrow P_{rollout} + \bar{S}$.step($\bar{x}$)
    $R_t \leftarrow \bar{S}$.getTerminalReward() ▷ **Backpropagation**
    $R \leftarrow$ BACKPROPAGATE($P_{rollout} + R_t$)
    return $R, \bar{X}$

Algorithm 2 Tree node constructor

function NEWNODE($\bar{x}, s_{parent}$)
  $\bar{x}_s \leftarrow \bar{x}$
  $s.parent \leftarrow s_{parent}$)
  $e.traversals \leftarrow 0$
  $s.evaluation \leftarrow 0$ ▷ $Q(s_{parent}, s)$
  $s.stepReward \leftarrow$ None
  $s.children \leftarrow \emptyset$
  return s

Algorithm 3 UCT with value network

▷ Input: Value policy network V^π
function UCT(s)
  $R_{max} \leftarrow -\infty$
  $s_{best} \leftarrow$ None
  for $s_{child}, \bar{x}_{child}$ **in** $s.children$ **do**
    $R \leftarrow s_{child}.evaluation + c_1\sqrt{\frac{\log N(s)}{N(s,\bar{x}_{child})}} + c_2 V^\pi(s, \bar{x}_{child})$ ▷ UCT with V^π
    if $R > R_{max}$ **then** ▷ Remember best child
      $R_{max}, s_{best} \leftarrow R, s_{child}$
  return $s_{child}, \bar{x}_{child}$

Algorithm 4 Backpropagation with transition cumulation

function BACKPROPAGATE(s_n, r)
  while s_n.parent **is** not None **do**
    $r \leftarrow r + s_n.stepReward$
    $s_n.traversals \leftarrow s_n.traversals + 1$
    s_n.**updateEvaluation**(r)
    $s_n \leftarrow s_n$.parent
  $s_n.traversals \leftarrow s_n.traversals + 1$
  return r

Algorithm 5 Action pruning

function PRUNEROOT(s)
  $N_{max} \leftarrow 0$
  $s_{best} \leftarrow$ None
  for s_{child} **in** $s.children$ **do**
    if $s_{child}.traversals > N_{max}$ **then**
      $N_{max}, s_{best} \leftarrow s_{child}.traversals, s_{child}$
  return $s_{best}, \bar{x}_{child}$

Chapter 5

Results

This work presents two main results. The first result revolves around our proposed reinforcement learner, MCFS-AP, testing if there is an increase in search accuracy and speed. To examine this, we test the effect of implementing rollout policy and value networks, as well as the simulation speed increase of the use of action pruning with state recall. The goal here is to examine whether the benefits of these measures are substantial enough to warrant implementation when using AST, and if so, what problems it should be considered for.

We present the second set of results from applying the most promising MCFS-AP learner to the FAST architecture, stress testing multiple possible scenarios that an autonomous marine vessel can encounter. These results are intended to test the effectiveness of the MCFS-AP method and to highlight possible weaknesses in the autonomous system.

Table 5.1: MCTS parameters.

Parameter	Value
Progressive widening	
Expansion coefficient k	0.5
Expansion exponential α	0.6
UCT w/ Value network	
Exploration coefficient c_1	17
Value network coefficient c_2	0.7
UCT w/o Value network	
Exploration coefficient c	10

All parameters of progressive widening and UCT selection were kept constant for all experiments, across all MCTS-SA and MCFS-AP models. The values used are based on the findings in the preliminary work and are the values that are found to best suit an MCTS-SA implementation without neural networks. This should counteract any bias introduced by tuning the MCFS-AP to better suit models using action pruning or neural networks. All core parameters are presented in Table 5.1. Our implementation is written in Python, using PyTorch for implementation of the neural networks. Seed-actions are generated using the Python stock random library. The experiments were run using a single 3.6GHz CPU core with 16 GB of 3.2GHz DDR4 RAM. GPU acceleration is not used for any of the models.

5.1 MCTS with rollout policy and value network: Simple simulator

To test the impact of the neural networks on the ability of the model to find failure states, a set of MCFS-AP models were applied to the interpretable simulator described in Section 4.1.2. The different models performed 200 searches, each based on 9,000 simulations. All models used the MCTS parameters described in Table 5.1. In this section, we present the parameters used in the setup, as well as the different experiments performed.

Setup

All searches in this experiment were carried out using the same setup of the interpretable simulator, with the parameters presented in Table 5.2. It should be noted that the starting position and heading of the steerable vessel were chosen to minimise the number of failure events.

Table 5.2: Simulator parameters.

Parameter	Value
Sim world size	100 x 100
Vessel step length	5
Crash distance threshold	5
Collision reward R_e	30
Steerable action range (deg.)	[-5, 5]
Steerable vessel start pos.	[16, 0]
Steerable vessel start vector	[0, 1]
Straight vessel start pos.	[50, 0]
Straight vessel start vactor	[0, 1]

The goal of this experiment is to test the impact of the value network and the rollout policy network introduced in Section 4.3.5. This is done by running five different models. The two first models are used to directly test the impact of the rollout policy network and the value network, by implementing a MCFS-AP model for each network. The two MCFS-AP models are implemented by using either the rollout policy network (RP) or the value network (VN) and will be referred to as MCFS-AP/RP/- and MCFS-AP/-/VN. In addition to this, we implemented one model using both neural networks, referred to as MCFS-AP/RP/VN. As a baseline, we implement two models. The first model is a baseline MCFS-AP/-/- model, without neural networks, and the second model is our implementation of MCTS-SA, presented by Lee *et al.* [13]. All MCFS-AP models implement action pruning, with a maximum prune depth of 18 and 500 nodes being added at each root, resulting in 9000 nodes and simulations in each search. Each search using MCTS-SA runs 9000 simulations. Each model search is run 200 times, resulting in a total of 1,800,000 simulations per model.

Ability to find failure events

During the experiment, each model saves the best solution found, the average reward of all solutions, and the best reward from every search. Using this, the average best reward of all searches and just the searches resulting in a failure episode, are calculated. Models using action pruning store the best reward found at every root, allowing us to calculate the average depth of the first and best failure event found at each search. The results are presented in Table 5.3.

Table 5.3: Models ability to find failure events over 200 trees built.

RL agent Nerual networks	MCFS-AP RP/VN	MCFS-AP RP/-	MCFS-AP -/VN	MCFS-AP -/-	MCTS-SA -/-
Nr. of runs	200	200	200	200	200
Failures found	**78**	74	41	59	71
Failure rate	**39%**	37%	20.5%	29.5%	35.5%
Global best R	**29.092**	28.650	28.784	28.061	28.245
Avg. best R	**5.818**	4.776	-1.426	1.927	2.304
Avg. best R when E	**27.941**	27.056	27.710	27.091	25.718
Avg. depth, first E	3.410	**3.203**	5.390	4.847	
Avg. depth, max R	11.395	**10.47**	14.195	13.375	

Run time impact

To estimate the effect that neural networks have on the run time of a search, the searches were also timed. Using the total run time of the action pruning model without neural networks, the impact of the neural networks is calculated, giving us the added run time to each simulation. The results are presented in Table 5.4.

Table 5.4: Neural network (NN) addition to runtime for simulations. All models are using periodic root pruning in order to.

RL agent Nerual networks	MCFS-AP RP/VN	MCFS-AP RP/-	MCFS-AP -/VN	MCFS-AP -/-	MCTS-SA -/-
Total Runtime (**h:m:s**)	4:12:19	1:51:49	1:13:28	0:08:06	0:06:41
NN Runtime addition (**h:m:s**)	4:04:13	1:43:43	1:05:22	0:00:00	0:00:00
Nr. of simulations	1,800,000	1,800,000	1,800,000	1,800,000	1,800,000
NN Runtime pr. sim (**ms**)	8.141	3.457	2.179	0	0

Another result which can be calculated from the run times is the added run time (hereby referred to as overhead) for node handling and tree pruning of the root pruning method. As the interpretable simulator has a simulation time of 0.127 ms pr simulation, we can calculate the total simulation time of the experiments being around 228 s, giving MCTS-SA an overhead of 173 s. If we assume a best-case reduction to a relative 55% simulation *+time using root pruning, as found in Section 5.2, we get a worst-case overhead of 361 seconds.

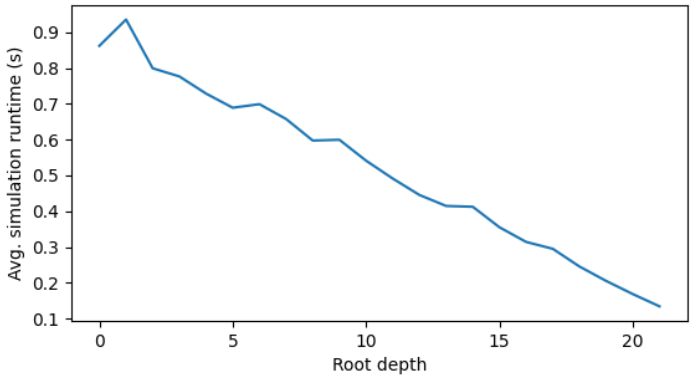


Figure 5.1: Results of state recalling. Graph shows average run times of simulations at different depths. The maximum depth of each simulations is 25.

5.2 Run time impact of state recall: Zeabuz simulator

Another measure implemented to increase the effectiveness of all MCFS-AP models is the use of action pruning with state recall, presented in Chapter 4.3.3. This experiment was carried out using the Zeabuz simulator, using MCTS-SA as a baseline. The Zeabuz simulator was chosen as the impact of state recall is negligible for the simple simulator, where a majority of the search time is taken up by node tree operations and not simulations. The scenario chosen for the experiment is uncrashable and contains the autonomous ferry and an adversarial vessel. Lack of failure states results in the simulation running the maximum length of 25 states. Action pruning is performed periodically at each depth $T_{prune} = \{t_0, t_1, ..., t_{22}\}$ after 20 searches at each depth.

When the simulation state was internally set to reflect the current state of the root node, the number of simulation steps needed to be calculated is reduced as the depth of the root node increased. This can be observed in Figure 5.1, where the average simulation times for simulations starting at different depths are presented. With a root depth of 0, the simulations start at time step t_0, calculating all 25 steps of the simulation. At the maximum root depth of 22 we set the simulation state at $t_2 2$ only calculating the last three time steps.

Comparing the run time of MCFS-AP to the run time of MCTS-SA, which always calculates all simulation times, we see a decrease in average simulation time, presented in Table 5.5. This decrease in average simulation time results in a faster overall search time, allowing MCFS-AP to searching the same number of disturbance sequences as MCTS-SA at approximately 55% of the time if the simulation time is sufficiently long.

Table 5.5: Simulation run times of MCFS-AP and MCTS-SA.

Parameter	MCFS-AP/-/-	MCTS-SA
Simulations run	440	440
Total run time (s)	227.335	412.521
Avg sim. run time (s)	0.517	0.938

5.3 Stress testing the milliAmpere 2

A real-life test case was performed applying our model for adaptive stress testing of the collision avoidance system of the milliAmpere 2. Seven scenarios were constructed, each scenario chosen to challenge and exploit plausible weaknesses of the autonomous ferry. Of the scenarios chosen, four use the sensor delay as a disturbance and three use the heading of the adversarial vessel.

General setup

All experiments use the FAST architecture with MCFS-AP/RP/VN, referred to as MCFS-AP for these experiments, as the reinforcement learner. This choice is based on the long simulation times of the Zeabuz simulator and the results discussed in Section 6.1.2.

All scenarios used to test the behaviour of milliAmphere 2 were performed within the Zeabuz simulator using the parameters presented in Table 5.6. One thing to note is the step resolution of 0.1. This is the result of each vessel step consisting of 30 sequential calculations, each resulting in a small step. The initial states for each experiment were chosen to test different aspects of the ColAv system. The different setups are discussed in more detail before each experiment and during the discussion in Section 6.2.

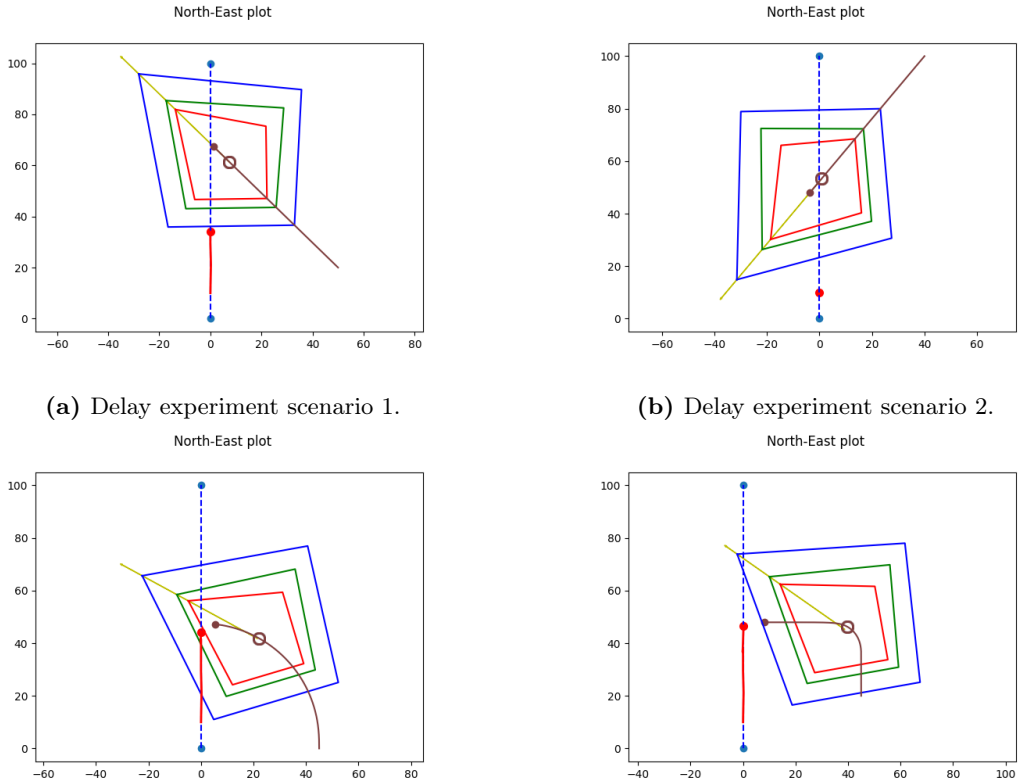
Table 5.6: Zeabuz simulator parameters.

Parameter	Value
Crash distance threshold	1
Collision reward R_e	100
Vessel step length	3
Step resolution	0.1
milliAmpere initial pos	[0, 0]
milliAmpere target pos	[0, 100]

5.3.1 Delay experiments

In the experiments discussed in this section, we are looking at delay used as the disturbance being tuned by the FAST architecture in order to find failure states. Delay, in this setting, refers to difference in perceived position of the adversarial vessel, and can be a result of processing time and uncertainties in the perception systems of milliAmpere 2. In our experiments we are simulating delay through setting previous positions as the perceived position of the adversarial vessel.

Adding a delay to the perceived state will alter the position of the safety zones, which in turn can impact the ColAv systems ability to act responsibly. The delay in a simulation step is illustrated by the perceived position of the adversarial vessel being given by a brown circle. The alterations to the safty zones are also reflected in the figures.



(a) Delay experiment scenario 1.

(b) Delay experiment scenario 2.

(c) Delay experiment scenario 3.

(d) Delay experiment scenario 4.

Figure 5.2: Best simulations for delay experiment. The results from stress testing four scenarios to explore the impact of sensor delay on collision avoidance. Each figure shows the state of the simulator at a time step during the best event found by the searches. The perceived position of the vessel is given by the brown circle. Safety zones are calculated based on the perceived position.

Setup

To test the minimum amount of input delay before a collision occurs, the four scenarios illustrated in Figure 5.2 were created. In the first two scenarios, the adversarial vessel is going in a straight line, crossing the desired path of the milliAmpere 2 vessel. These scenarios are constructed to attempt to test the worst case orientation of the safety zones, to allow the adversarial vessel to approach milliAmpere 2, without being detected in sufficient time.

In the third and fourth scenarios, the adversarial vessels turn in order to cross the desired path of the milliAmpere vessel. Here we test milliAmpere 2 how delays affects the ability of milliAmphere 2 to stop for crossing vessels making sharp turns.

Results

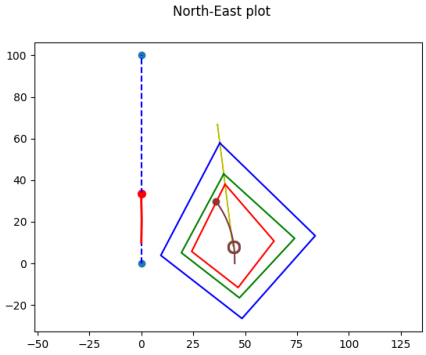
Of the four scenarios, only scenario four resulted in a crash, with scenario three resulting in a near miss. We will now take a closer look at the most likely failure state found for the four scenarios in order to examine the behaviour of the milliAmpere vessel.

The MCFS-AP agent was unable to find failure events for the straight course scenarios. In both scenarios we observed that the delay has little impact on the ColAv behaviour, as the milliAmpere 2 waits for the vessel to pass, illustrated in 5.2b, before following at a safe distance, illustrated in 5.2a. This is a result of the milliAmpere predicting the future path of the adversarial vessel, opting to stop in order to wait for the vessels to pass. As the course of the adversarial vessels is straight, any added delay does not affect the predicted future path, resulting in little change in the ColAv behaviour.

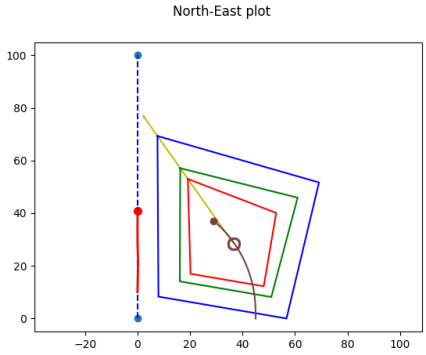
For the scenarios in Figure 5.2c and 5.2d, the adversarial vessel makes a turn. As a result, we see that alterations in sensor delay can also result in alterations in the perceived direction of the adversarial vessel. This strengthens the influence of the AST method and introduces more intelligent behaviour in the search result.

During the course of the closest encounter seen for the scenario in Figure 5.2c, we observe that this deviation to delay can confuse the ColAv system, resulting in a close encounter between the two vessels. In Figure 5.3, we can examine the development of the delay over the course of the simulation. In the first steps of the simulation, we observe that a large amount of delay is added, hiding the change of course, reducing information and, in turn, the reaction time of the ColAv system. When the milliAmpere 2 aproaches the interception point of the two vessels, the delay is reduced, in order to get milliAmpere 2 to stop. In the last time steps before the close encounter of the two vessels, the delay is turned up, to reduce information to the ColAv system, making the collision avoidance harder. Despite this, the milliAmpere 2 vessel never reaches the interception point of the two vessels, avoiding a crash.

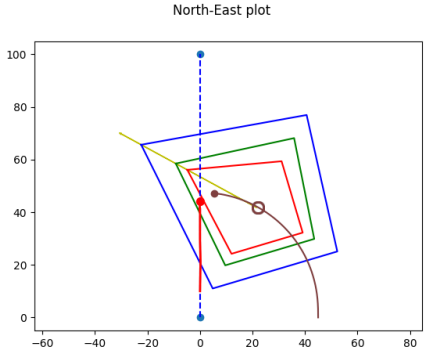
The behaviour observed in the scenario in Figure 5.2c is mirrored in the most likely failure event found for the scenario in Figure 5.2d. Here MCFS-AP is able to find failure events for this scenario, as the sharp turn gives applied delay a larger impact to the perceived course of the adversarial vessel. Examining the most likely failure event, illustrated in Figure 5.4, we once again see that large delays are being added during the turn, masking the adversarial vessels change in course. In the next time steps, the delay fluctuates, periodically dropping in order to affect the ColAv system, slowing the milliAmpere into a position close to the intersection point of the two vessels. Towards the final time steps before the collision, the delay is once again turned back up for a short time. This results in a miscalculation of the safety zones, illustrated in Figure 5.4c, tricking the ColAv system. This results in milliAmpere 2 speeding up, making the momentum gained to great to slow down during the last time steps where the ColAv system realises that the two vessels are on a collision course.



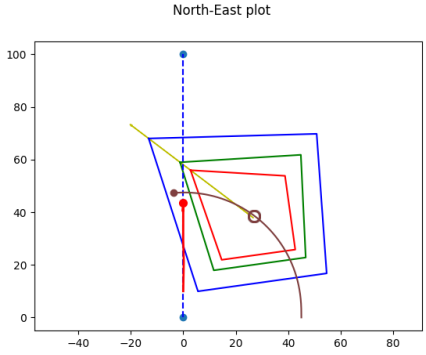
(a) Large amounts of delay early in the simulation masks the adversarial vessels actions.



(b) Reduced delay results in the milliAmpere stopping to let the adversarial vessel pass.

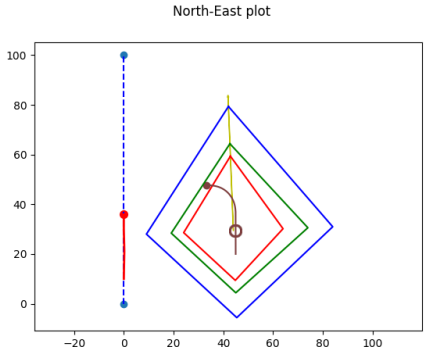


(c) Noise is added in the final steps to try and trick milliAmpere 2 into moving forward.

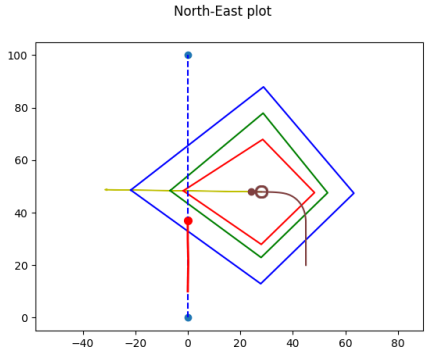


(d) The adversarial vessel passes milliAmpare 2 without a collision.

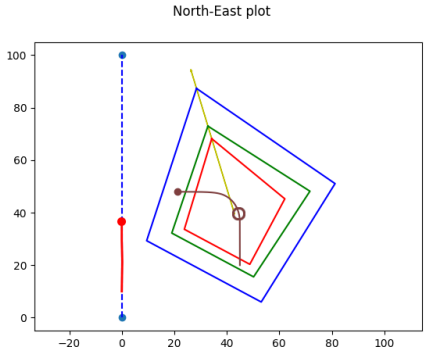
Figure 5.3: [Result of delay experiment with slow turning]. Four time steps from the simulation closest to a collision. The delay added given by the perceived position of the adversarial vessel, illustrated with a brown circle.



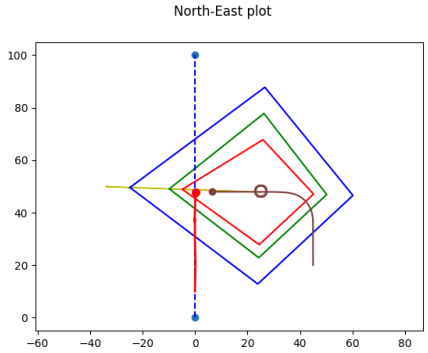
(a) Delay masks the adversarial vessels actions.



(b) Reduced delay results in the milliAmpere stopping for the adversarial vessel.

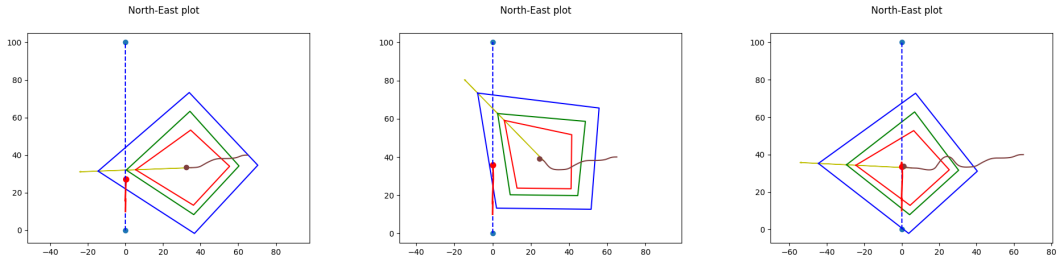


(c) Reintroduction of a large delay add momentum to the milliAmpere

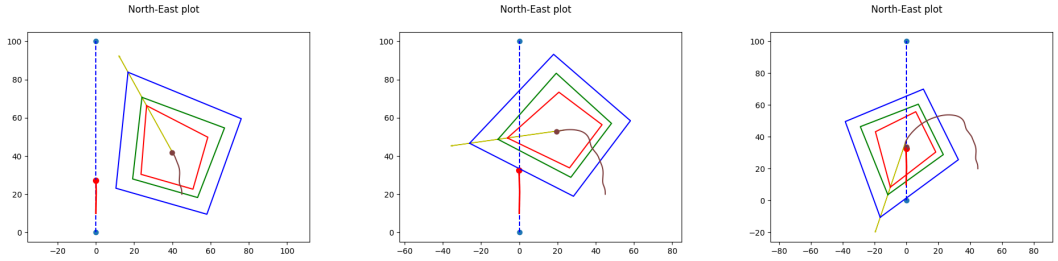


(d) The milliAmpere is unable to avoid side collision as its momentum is too high.

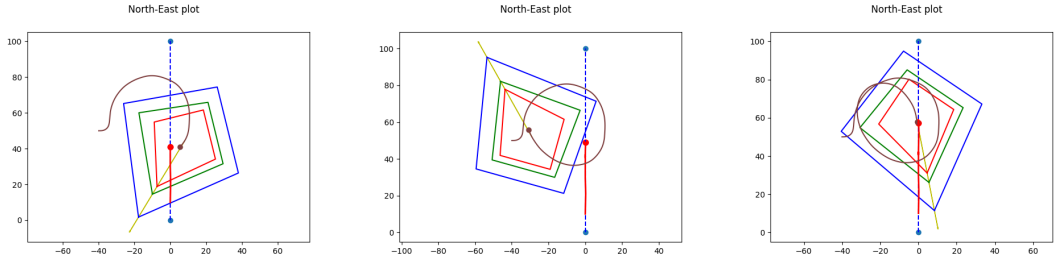
Figure 5.4: Result of delay experiment with rapid turning. Four time steps from the simulation of the most likely failure event.



(a) Scenario with the adversarial vessel starting starboard side of milliAmpere 2.



(b) Scenario with the adversarial vessel starting out parallel to milliAmpere 2.



(c) Scenario with the adversarial vessel starting port side of milliAmpere 2.

Figure 5.5: Best simulations for delay experiment. Three time steps from the best simulation found when stress testing three scenarios using the steering course of the adversarial vessel as the disturbance. Safety zones are calculated within the ColAv system based on perfect knowledge of the position of adversarial vessel.

5.3.2 Steering experiment

This section discusses experiments where the disturbance of the simulation is given by alterations to the course of the adversarial vessel. The milliAmpere 2 has a perfect knowledge of the position and heading of the adversarial vessel. The goal of this experiment is to test the ColAv systems ability to avoid head on collisions, looking for the most predictable course to a crash that system is unable to avoid.

Setup

For experiments using steering as a disturbance, three different scenarios were constructed and tested. In all three experiments, there is one adversarial vessel that is controlled by our AST implementation. This is done by passing a desired alteration to the heading to the adversarial vessel at each time step. The best simulations found for each scenario is illustrated in Figure 5.5.

Results

AST was able to uncover failure states for all three experiment scenarios. This was in general done by steering towards the milliAmpere vessel, resulting in it stopping, before crashing into it. However, the disturbance traces found by the searches are of varying quality.

The transition probability of the steering disturbance is defined as a conditional disturbance, given by Equation 4.5, in order to reward regular and predictable courses. As such the optimal solution should try to plot a "smooth" course for the adversarial vessels. Keeping this in mind when looking at the solution proposed for the scenario in Figure 5.5b, we can observe what appears to be a somewhat optimal route chosen by AST. Although there are rapid fluctuations in heading in the earlier time steps, we see a sudden shift to a smooth course. Further, we observe that this second half of the path originally looks to cross in front of the milliAmpere, causing the ColAv system to stop the autonomous vessel. After stopping the milliAmpere, the adversarial vessels head down, crashing into the autonomous vessel head on.

A less smooth, more chaotic approach can be observed in the scenario in Figure 5.5a, which is the simulation with the lowest reward. This scenario is the scenario in which MCFS-AP was able to find a failure event given the least amount of simulations. This early, high reward combined with the aggressive search of MCFS-AP resulted in the solver not being able to refine the given solution, resulting in the lowest increase to total reward over the search.

The behaviour of the adversarial vessel seen in Figure 5.5c is the behaviour that we considered to be the least intelligent, including more than twice the number of time steps compared to the other simulations. Despite this, it is the simulation with the highest total reward of any of the tree simulations. Looking at the course of the adversarial vessel, we can observe the reason behind this, as the chosen path has very little alteration. This is also seen in the disturbance sequence, where every disturbance in the final 67high of the 69 time steps stays within a 10% range of the possible action space, resulting in high transition probabilities and a high final reward.

Chapter 6

Discussion and Conclusion

In Section 1.4 we introduce the research goals and questions for this thesis. In this section, we will discuss the results presented in Section 5, before addressing the research questions in Section 6.3 and proposing future work in Section 6.4.

6.1 MCFS-AP with neural networks and state recall

The first research goal of this thesis, presented in Section 1.4, is to implement our proposed model and examine the efficiency of our proposed reinforcement learner MCFS-AP. To achieve this, variations of MCFS-AP, using neural networks and state recall, presented in Sections 4.3.5 and 4.3.3, are tested, and the results are presented in Section 5.1. In this section, we discuss these results and whether this approach is worth considering for other test cases.

6.1.1 Decreasing number of needed simulations through neural networks.

The first action taken in order to improve the simulation times in a search is the implementation of two neural networks, a rollout policy network and a value network. The goal of the two networks is to use the knowledge from previous simulations to direct the current search. To achieve this, we train the value network to predict the expected result of applying an action in a state, in order to better direct the selection process. The rollout policy is trained to predict desirable disturbances from a state, which is used to replace the random rollout policy, common in MCTS, to perform more desirable rollouts.

Looking at the results presented in Table 5.3, we can study the ability of different models to find failure events. Looking at the failure rate of the different models, we see that both models that use the rollout policy network are able to find failures more frequently. These models also find the disturbance sequences with the highest rewards and are able to find desirable failure disturbance sequences using fewer simulations. The reason for this is likely that the rollouts performed throughout the search benefit from the rollout policy, generally performing rollouts which increase the chance of finding a more optimal solution in a rollout. This in turn results in

predicted rewards converging faster, as a lot of variation is lost in the lack of randomness from the default random rollout policy.

A surprising result is the negative impact of the value network, where 20.5% of the searches find a failure event, compared to the model that does not run any neural networks, which finds a failure event in 29.5% of all searches. Although this suggests poor performance of the value network, reducing the efficiency of the base model by 30%, it should be noted that no decrease in efficiency is observed when the rollout policy is implemented, where the model using both networks outperforms, if only barely, the model only using the rollout network. It is hard to pinpoint exactly what contribution the value network has to these results, but it is likely that the value network benefits from the lower variations in predicted rewards as a result of the rollout rewards.

Despite the improvements over the model without neural networks, we still see the impact of action pruning in the difference between the networkless model, with its 29.5% failure finding rate, and the baseline MCTS-SA model, which finds failures in 35.5% of its searches, done by constructing one large static tree containing 9000 nodes at each search. The greedy approach introduced by action pruning, locking down early parts of the disturbance sequence, heavily impacts the models' ability to find failure events. A slight positive with the greedy approach of action pruning seems to be that when a failure is found, the aggressive approach seems to be more efficient at refining the sequence, as the average best reward of searches where a failure event was found is higher. This is to be expected, as the greedy approach spends more time exploring a single branch, resulting in a higher likelihood of finding a better solution given that the branch results in a failure state.

Lastly, we need to consider the added run time added through the training and predictions done by the neural networks. Looking at the run times presented in Table 5.4, the increase in run time might seem huge, at worst increasing the run time from a total run time of 8 minutes using no neural networks to a total run time of 4 hours and 12 minutes for the model running both neural networks. Though this seems like a huge cost we should we can calculate an average added time to each simulation reveling that the the networks add around 2 to 8 ms of run time at each simulations. Further more we know that the training and predictions performed by the networks are constant and independent form the run time of the simulations as they are performed by the MCTS using previously acquired node statistics. By adding this fixed addition to different simulation run times, we can calculate the average number of simulations that can be run in a set amount of time. Normalizing these results for the networkless model we get the results illustrated in Figure 6.1.

Here we can see the impact the neural networks has on the total run time as the length of an average simulation changes. We see the diminishing impact as the simulation times approaches 0.1 seconds, and the almost negligible effect at a simulation time of 1 second, or the time the Zeabuz simulation runs at.

Further more we can adjust this graph to reflect the number of searches that result in a failure we can approximate the number of failures found over time as a function of the total simulation run time. The values are once again normalize for the MCTS-SA model, and illustrated in Figure 6.2, which now also include the networkless action pruning model labeled No NN, assuming similar run times pr simulation.

Here we see the relative rate of finding failure states for the models using the rollout policy network overtake the MCTS-SA, reaching maximum increase amount of failure states found of about 109%

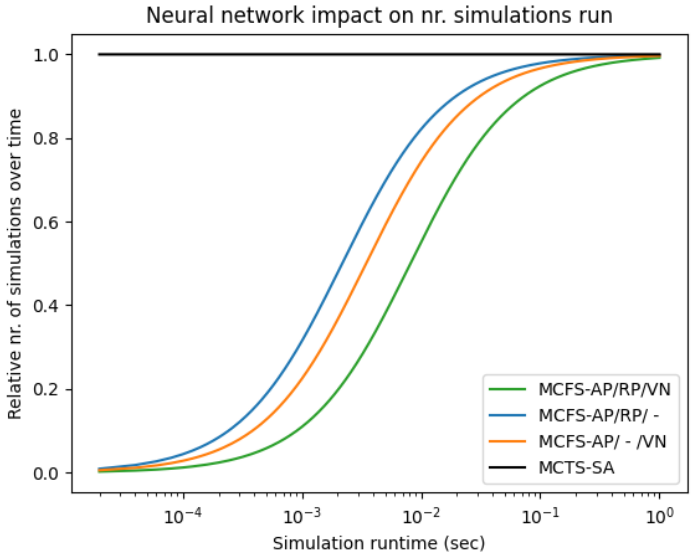


Figure 6.1: Relative run time of a full simulation. Relative run time pr simulation over time with the addition of neural networks, depending on simulation run time.

and 104% relative to the MCTS-SA baseline. The value network and the no network model perform poorly approaching a failure finding rate of 57% and 83% respectively relative to MCTS-SA.

6.1.2 Decreasing average simulation run time through action pruning and state recall.

Up until this point we have only considered the negative impact of the aggressiveness of action pruning, but now we take into account the impact of state recalling, presented in Section 4.3.3. State recall remembers and sets the internal simulation state after every action pruned, allowing us to start future simulations at the new root state of the search. This reduces the amount of time steps calculated for later simulations, resulting in a increase in search time as the simulation time grows.

To find the decrease in simulation run time an experiment, the more complex Zeabuz simulator was used, as the effect on longer simulation time would be easier to observe. The experiment, presented in Section 5.2, runs simulations to a depth of 25 time steps, iteratively pruning at every depth down to a depth of 22, the same relative depth pruned to in the results presented above. The results presented in Table 5.5 show a drastic reduction in the total simulation time, where the pruned model has a run time that is 55% the run time of the MCTS-SA model. This is the result of the simulations getting shorter over time, where the first simulations last for the full 25 time steps, while the last simulations only simulate the last 3 time steps. This gives an average simulation length of 14 time steps, which is 56% the total length of 25 time steps in every simulation run by the MCTS-SA model.

This reduction in simulation time is not the result of not calculating new time steps, but the result of recalling an old state for the current root instead of calculating it for every simulation. This has no impact on the actual search, as the system under test is deterministic and the calculated root state should always be the same.

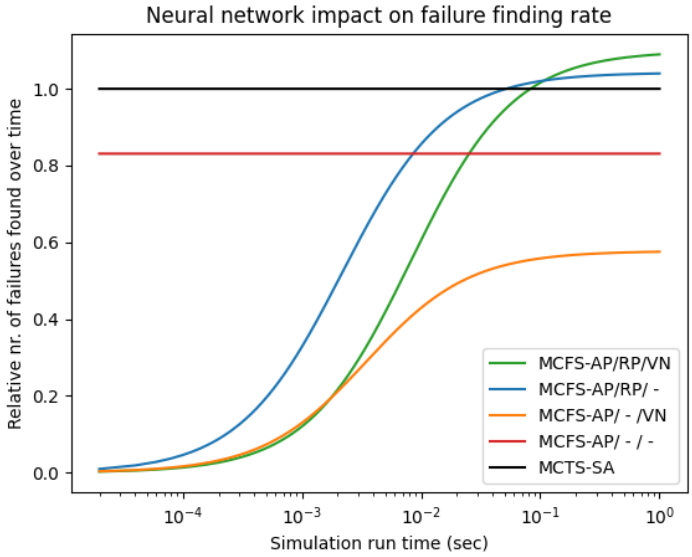


Figure 6.2: Relative rate of finding failures over time, depending on full simulation run time. State recall is not enabled, but action pruning is.

If we look at the findings in Figure 6.2 we can factor in the time saved by state recall resulting in more simulations being run over time. The new calculations are illustrated in Figure 6.3

The resulting figure gives us an approximation of the efficiency of the different models depending on the simulation time. If we examine the crossing point of the models, we can see that the MCFS-AP/-/- model using action pruning outperforms MCTS-SA when simulation times exceed 0.72 ms. Further, we see that the rollout policy model MCFS-AP/RP/- surpasses MCFS-AP/-/- at 15ms run times, before MCFS-AP/RP/- is surpassed by the dual network model MCFS-AP/RP/VN at simulation times of approximately 200 ms. We also see that at simulation times of 1 s, MCFS-AP/RP/VN and MCFS-AP/RP/- model approach a relative failure-finding rate of 1.96 and 1.88 respectively. The baseline MCFS-AP/-/- model with action pruning and stat recall approaches a relative failure-finding rate of 1.51.

Furthermore, we find that implementation of the value network alone is not advisable, as it performs strictly worse than the model that does not implement neural networks. This is expected as the subpar results of the value network model and the added run time both result in worse performance.

However, it should be noted that the results in the graph are not definite, as more data should be gathered on a series of simulations of varying difficulties and run times to confirm the approximations made within Figure 6.3.

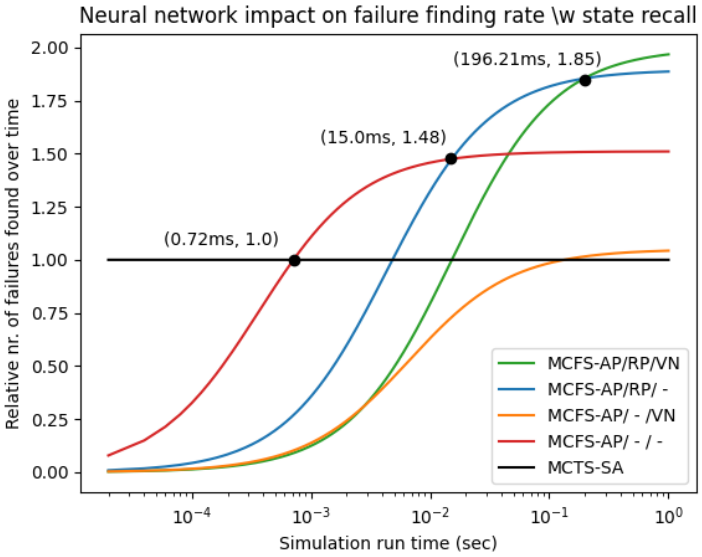


Figure 6.3: Relative rate of finding failures over time, dependent on avg. simulation time with state recall. MCFS-AP models use state recalling.

6.2 Adaptive stress test of the milliAmpere 2

The second goal of this thesis is to apply AST to the milliAmpere 2 autonomous ferry to examine the possible weaknesses and limits of the collision avoidance (ColAv) system. The simulator encapsulating this the system under test (SUT), is developed by Zeabuz, though access to the source code has been given, allowing for propper implementation of state recall. Given the findings presented in Section 6.1 and an average simulation time of approximately 1 second, we were confident when applying our model, with neural networks and action pruning, to stress test the autonomous ferry.

For the experiments, we applied our implementation of AST to a range of scenarios in which a disturbance was altered, either in the form of sensor delay or the course of the adversarial vessels. In this section, we discuss the results of the experiments presented in Section 5.3.

6.2.1 Delay experiments

To examine how sensor delay affects the performance of the ColAv system, four scenarios were stress tested in Section 5.3.1. AST was implemented with the FAST architecture, using MCFS-AP/RP/VN as the reinforcement learner. This implementation was chosen based on the findings presented in Figure 6.3.

The first two scenarios are relatively uneventful as the headings of the adversarial vessels are constant with an intersection point in front of the milliAmpere 2. In these scenarios, we see that the ColAv system always yields to other vessels, waiting for them to pass. Despite being a passive and slow approach to collision avoidance, the behaviour of the vessel becomes predictable while also minimising the risk and severity of collisions with other vessels.

A possible adjustment that could be made to these experiments is an alteration in the calculation of the distance to collision d. In the current configuration d is the shortest Euclidean distance

between the vessels. Although this is a fairly accurate heuristic for collisions, we can observe that the milliAmpere 2 always moves along its planed path with negligible deviation. As a result, we know that collisions will occur along the planned path at the intersection point of the two vessels. Our goal now becomes getting the milliAmpere 2 to stay at the intersection point at the time of crossing, resulting in the distance between the vessels at that time being a possibly better heuristic.

When observing the search results from scenarios three and four, we see a more interesting behaviour by the AST method, trying to crash the milliAmpere 2. Over the span of the simulation, we observe how the AST indirectly controls the milliAmpere vessel by alternating the sensor delay, applying small delays to keep the milliAmpere 2 stationary, or long delays to trick the vessel into moving forward. However, it should be noted that, despite finding failure events in scenario four and very close encounters in scenario three, we see that the disturbance sequences are highly unlikely, requiring a high amount of noise in both the sensor reading and interpretations.

6.2.2 Steering experiments

Stress testing by directly controlling the course of the adversarial vessel resulted in failure events for the three scenarios tested. In all scenarios the adversarial vessel is steered towards the milliAmpere, causing it to stop before the collision occurs. All scenarios show cautious behaviour by the milliAmpere vessel, slowing down and stopping when approached, before being struck by the adversarial vessels. The courses plotted by the adversarial vessels can easily be considered overly reckless and highly unlikely, showing destructive intentions.

In addition, we argue that the failure events found through our searches, using direct control of the adversarial vessels, are not very informative, as the transition probabilities are based on the assumption that the most likely action a captain makes is to continue doing the previous. This oversimplification of how a human-controlled vessel acts results in highly unlikely actions, even given high transition probabilities. This problem has previously been addressed by Corso *et al.* [16], but the proposed solution of penalising time steps where the SUT operates in an irresponsible manner was not effective for this experiment, as the milliAmpere 2 always stops long before a collision occurs.

Another problem arises with our model when considering that the objective of AST is to find the most likely failure event. Our implementation of AST used in these experiments employs both neural networks and action pruning, which results in a greedy network that generally repeats previous successful behaviour. Although this may be positive in harder simulations, we suspect that the greedy, somewhat repetitive nature of the model does not allow sufficient exploration in simulations with a high probability of failure events occurring.

This combination of an exploitational model and the high frequency of failure events results in conversion to suboptimal solutions for all scenarios in this experiment. This is especially apparent for the solution to Scenario 1, where a failure event was fond in the third simulation, stepping through a full sequence of pseudo-random seed-actions. The large positive reward and the gready nature of our model will then outweigh the exploration bias in later selection phases. The resulting narrow search tree, found in the first search, is then used to train the rollout policy and value networks, which will find similar suboptimal solutions in later searches. This new tree will once again result in a reinforcement of the rollout policy and value networks, converging the search towards a variation of the suboptimal solution.

6.3 Conclusion

Taking into account the findings discussed in Sections 6.1 and 6.2, we present the conclusion of the thesis. To present this, we go back to the beginning of the thesis and address the four research questions presented in Section 1.4.

6.3.1 RQ1: How does the use of neural networks to select simulations affect the failure-finding rate of MCTS?

The use of neural networks had varying effects on the failure-finding rate of the models. Despite the differences in efficiency, we do see improvements, especially using the rollout policy network. There are still uncertainties, especially with respect to the value network, but an increase in efficiency was seen for simulators with longer simulation times.

6.3.2 RQ2: How does recalling internal simulator states affect the failure-finding rate of MCTS?

Recalling the internal states of the simulator was implemented in conjunction with action pruning. State recall shows promising results, reducing the average simulation time by 45%. However, the efficiency of state recall is hampered by the implementations chosen with the reliance on action pruning, which was found to reduce the failure-finding rate of the model.

6.3.3 RQ3: How well does milliAmpere 2 handle varying amounts of added delay in the perceived position of other vessels?

Examining the results of the delay experiment, we find that the milliAmpere 2 performs well when delays are introduced, requiring large amounts of fluctuating, well-timed delay in order to crash. The possibility for such values to be possible in real life is hard to say for certain without deeper knowledge of the milliAmpere 2 system, though it seems highly unlikely.

6.3.4 RQ4: How well does milliAmpere 2 handle challenging encounters with other vessels?

Despite finding failure events for all experiments in which the heading of the adversarial vessel was controlled by AST, we feel certain in the ability of milliAmpere 2 when it comes to avoiding other vessels, given information about their position. All failure events found were strictly caused by the adversarial vessel crashing into the autonomous ferry, after the ferry had stopped to make way for the adversarial vessel.

Although we can argue that the behaviour of milliAmpere 2 is overly passive, we cannot argue that the behaviour is safe with respect to other vessels, given the scenarios we have tested. The relative novel approach to collision avoidance minimises the risk of collisions depending on the human controlled vessel to act irrationally. By yielding to human-controlled vessels and relying on them to resolve challenging scenarios in their own time, the milliAmpere 2 prototype acts

predictably and humbly, allowing humans to ease into the idea of fully autonomous vessels. As such, we conclude that milliAmpere 2 and its collision avoidance system seem like a good step towards the introduction of autonomous ferries.

6.4 Future work

Despite promising performance at high simulation times, we still see that further improvements can be made to the MCTS implementation. The first alteration we suggest is to move away from action pruning, instead implementing the recall of simulator states at the node level. By storing the states within the tree nodes, we are able to skip larger parts of the simulations, in theory further reducing the simulation times. If every node, or just the nodes traversed a sufficient amount of times, should store its simulator state, is hard to anticipate and would have to be tested.

Another major hurdle of the current model is the iterations needed to train the neural networks. A huge improvement to this would be to build general-purpose neural networks that can be applied to a range of scenarios. Building such a model would require a separate training routine, searching through a set of different scenarios, while also generalising the input, relying more heavily on the vessels relative positions to each other rather than fixed coordinates.

Finally, it should be noted that the model has only been properly compared with models using MCTS. As Koren *et al.* [14] has showed that the use of deep neural networks as reinforcement learners convincingly outperforms MCTS-SA it would be interesting to test if DNN outperforms our model as well and whether a hybrid of the two models is beneficial.

Bibliography

[1] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat and M. P. Owen, 'Adaptive stress testing of airborne collision avoidance systems', in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, pp. 6C2-1 - 6C2–13.

[2] Zeabuz. 'NTNU and the milliampere ferries'. (2021), [Online]. Available: https://zeabuz.com/milliampere/ (visited on 26th Nov. 2021).

[3] ——, 'The autonomy system'. (2022), [Online]. Available: https://zeabuz.com/technology (visited on 2nd Jun. 2022).

[4] M. J. Kochenderfer, C. Amato, G. Chowdhary *et al.*, 'Sequential problems', in *Decision Making Under Uncertainty: Theory and Application*. 2015, pp. 18–19, 77–112.

[5] R. Coulom, 'Efficient selectivity and backup operators in monte-carlo tree search', in *5th International Conference on Computer and Games*, Jun. 2006.

[6] G. G. M. J.-B. Chaslot, S. Bakkes, I. Szita and P. Spronck, 'Monte-carlo tree search: A new framework for game ai', *Bijdragen*, Jan. 2008.

[7] L. Kocsis and C. Szepesvári, 'Bandit based monte-carlo planning', *Machine Learning: ECML*, vol. 2006, pp. 282–293, Sep. 2006.

[8] G. G. M. J.-B. Chaslot, M. H. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk and B. Bouzy, 'Progressive strategies for monte-carlo tree search', *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[9] R. Coulom, 'Computing elo ratings of move patterns in the game of go', *ICGA Journal*, vol. 30, Jun. 2007.

[10] IMO. 'Convention on the international regulations for preventing collisions at sea, 1972 *COLREGs*'. (1972), [Online]. Available: https://imo.org/en/About/Conventions/Pages/COLREG.aspx (visited on 6th Jun. 2022).

[11] I. Porres, S. Azimi and J. Lilius, 'Scenario-based testing of a ship collision avoidance system', in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 545–552.

[12] M. Koren, A. Corso and M. J. Kochenderfer, *The adaptive stress testing formulation*, 2020.

[13] R. Lee, O. J. Mengshoel, A. Saksena *et al.*, *Adaptive stress testing: Finding likely failure events with reinforcement learning*, 2020.

[14] M. Koren, S. Alsaif, R. Lee and M. J. Kochenderfer, *Adaptive stress testing for autonomous vehicles*, 2018.

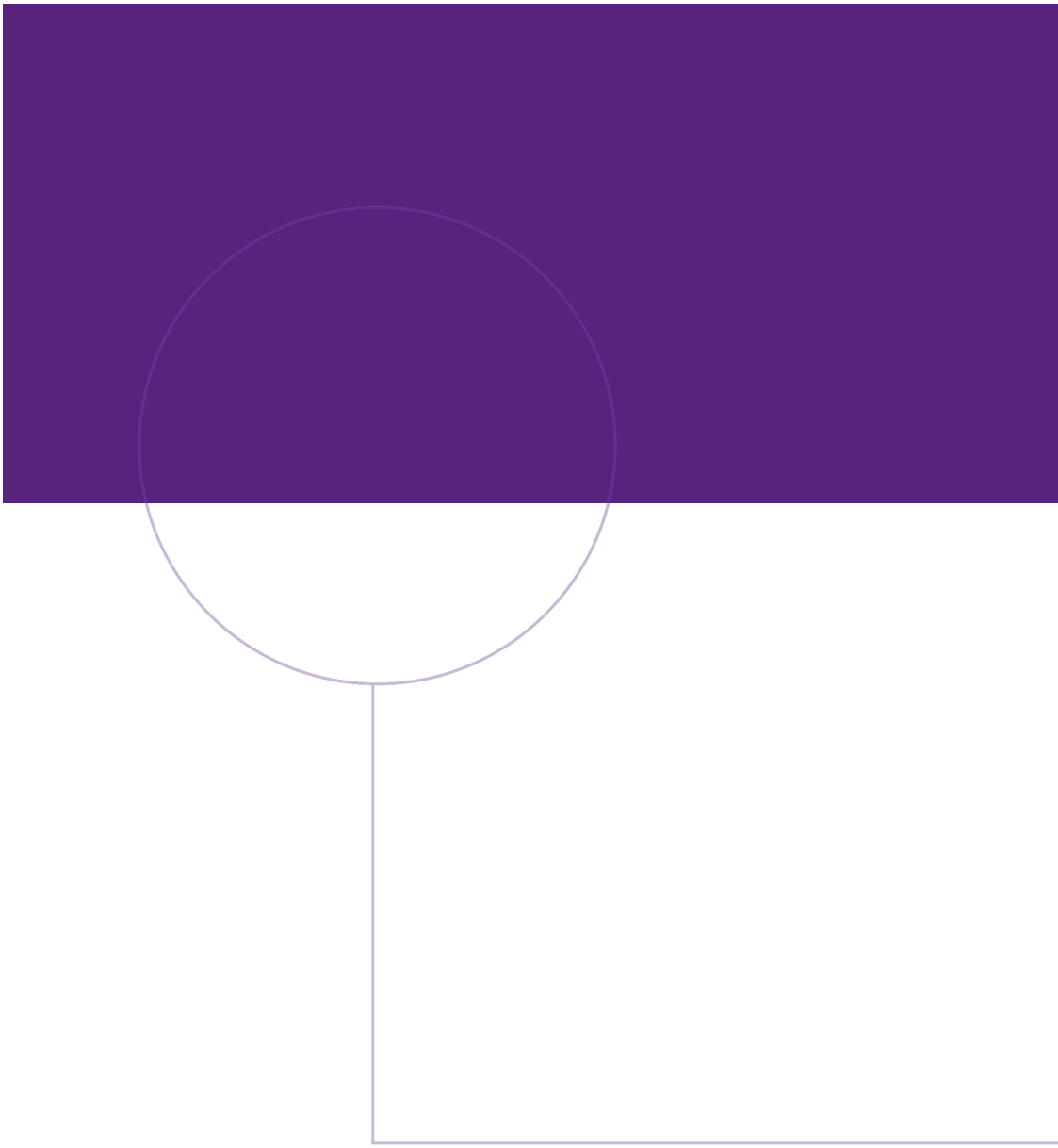
[15] K. El-Awady, *Adaptive stress testing for adversarial learning in a financial environment*, 2021.

[16] A. Corso, P. Du, K. Driggs-Campbell and M. J. Kochenderfer, *Adaptive stress testing with reward augmentation for autonomous vehicle validation*, 2019.

[17] D. Silver, T. Hubert, J. Schrittwieser *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017.

[18] D. Silver, A. Huang, C. Maddison *et al.*, 'Mastering the game of go with deep neural networks and tree search', *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[19] D. Silver and D. Hassabis. 'Alphago: Mastering the ancient game of go with machine learning'. (2016), [Online]. Available: https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html (visited on 6th Dec. 2021).



NTNU

Norwegian University of
Science and Technology