# Speeding up Adaptive Stress Testing: Reinforcement Learning using Monte Carlo Tree Search with Neural Networks and Memoization

**Brage Lytskjold**[1] **and Ole Jakob Mengshoel**[1]

[1] Norwegian University of Science and Technology
bragelyt@gmail.com, ole.j.mengshoel@ntnu.no

**Abstract**

When developing an autonomous safety-critical system, it is crucial that the system is able to act safely in a wide range of scenarios. One way to help validate this is through Adaptive Stress Testing (AST). AST is used to stress test a safety-critical system by altering the variables of a simulated environment, finding likely sequences of environment disturbances that result in failure events such as collisions. AST search times increase drastically for systems with long simulation times. In this paper we study several computational methods that promise to reduce search time. Inspired by computer game agents including AlphaZero, we introduce neural networks to select the most promising paths when traversing the search tree as well as performing simulation rollouts that are more likely to result in failure events. Also studied is the method of memoization, which amounts to periodically storing simulation states for commonly traversed search tree nodes. This allows us to speed up future simulations without impacting search accuracy. The efficiency of the proposed speed-up methods is studied with AST for simulation-based validation of an autonomous ferry.

## 1 Introduction

**Context.** Testing the limits of a system by observing how it behaves in a range of challenging scenarios is known as scenario-based stress testing. While such stress testing can be performed in the real world, one often encapsulates the system under test (SUT) in a simulator (Lee et al. 2015; Porres, Azimi, and Lilius 2020; Koren et al. 2018; Hjelmeland et al. 2022). Using a simulator is often essential as it allows us to tailor the scenarios with full control of external disturbances. The simulation approach is also scalable, safe, and cost efficient compared to real world testing. In this work we stress test an autonomous ferry using simulation.

We study scenario-based stress testing by means of adaptive stress testing (AST). AST aims to find the most likely failure event of a system structured as a Markov decision process (MDP) (Lee et al. 2015, 2020). AST has shown promise in several applications, including testing of aircraft collision avoidance systems (Lee et al. 2015, 2020), autonomous vehicle pedestrian collision avoidance systems (Koren et al. 2018), and maritime collision avoidance systems (Hjelmeland et al. 2022). In this paper we use AST (Lee et al. 2015) with Monte Carlo Tree Search (MCTS) for reinforcement learning (RL) to stress test the mA2 autonomous ferry (Brekke et al. 2022) via simulation.[1] MCTS is based on the Monte Carlo method, randomly sampling rewards of random solutions to a deterministic problem, using the rewards as heuristics. MCTS works by building a tree where each branching path down the tree represents different actions that form unique simulation-based solutions to the MDP. In the final tree structure, a state is represented by nodes, while different actions are represented by the edges. A natural result of this approach is a strong correlation between the simulation time and total search time for AST.

**Challenges.** While AST has shown promising results, there are also challenges (Corso et al. 2019; Hjelmeland et al. 2022). Specifically, we study the challenge of computational cost, which limits the scalability of AST. In mA2 pilot studies we found that MCTS was capable of finding collision avoidance (ColAv) failure events, even in scenarios with an exceedingly low rate of failures. However, we also observed a need for a large number of searches to get a reliable result. This can be a problem when the computational cost of simulation is high, as often happens in science and engineering. Since a simulator is an abstraction of the real world, we need to consider how realistic and accurate that abstraction is. There is a trade-off, as a more realistic simulator generally requires more computational resources, resulting in longer simulation times. Developing simulation for AST consequently becomes a balancing act between realistic simulations with more accurate validation results and less complex simulations with faster runtimes.

**Contributions.** Simulators are often used for crucial engineering validation testing. This includes validation of collision avoidance (ColAv) systems within transportation, including for autonomous vehicles (Lee et al. 2015; Porres, Azimi, and Lilius 2020; Koren et al. 2018; Corso et al. 2019; Lee et al. 2020; Hjelmeland et al. 2022). Unfortunately, ColAv simulators can be relatively costly to run, with simulation times of 1 second or higher. In this work,[2] we study

[1]Similar approaches, not using AST, for testing ship collision avoidance systems exist; Porres et al. (Porres, Azimi, and Lilius 2020) present a scenario-based testing system that aims to generate challenging test scenarios to stress-test autonomous vessels.

[2]This paper is based on Brage Lytskjold's MS thesis from NTNU (Lytskjold 2022).

methods to improve AST scalability, by reducing the impact of costly simulation runs, without negating the validity of the results. Specifically, we introduce Monte Carlo Forrest Search with Action Pruning (MCFS-AP). The MCFS-AP algorithm is based on the MCTS-PW algorithm (Lee et al. 2020), with the addition of constructing search tree differently and using neural networks similar to AlphaZero (Silver et al. 2017). Using MCFS-AP with ColAv simulators, we study the behaviour of mA2 in different scenarios, while applying disturbances to the ferry and environment throughout the simulations. Multiple simulators and variants of MCFS-AP are studied, enabling a safe, cost-effective, and scalable way of testing different dimensions of ColAv systems for autonomous ferries such as mA2.

## 2 Background and Related Work

**Testing Ship Collision Avoidance Systems.** (Porres, Azimi, and Lilius 2020) present a scenario-based testing system that aims to generate challenging scenarios to stress-test autonomous vessels. For the reward function, a subset of the COLREG[3] rules, a Collision Regulation rule set quantifying safe behaviour for marine vessels is used. By simulating and calculating the total reward, a scenario can be evaluated in regard to how well the autonomous vessel, the SUT, complies with COLREG rules while avoiding collisions. 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: around 10% given random initial states. The method is essentially a filtered random search, making the method ineffective for finding failure events if they are sufficiently rare. If applied to mA2, given the robustness of the system, the method is unlikely to perform well.

Hjelmeland *et al.* study AST applied to mA2 (Hjelmeland et al. 2022; Hjelmeland 2022). They demonstrate that AST can be used to find failures, specifically collisions with adversary vessels. Four different AST reward functions are considered, along with their respective impacts on simulation results including the types of failures discovered. Similar to our work, they use AST in the mA2 setting, and note that an important limitation is the computational requirement when using a detailed simulation. This limitation of AST when it comes to computational cost is exactly what we are studying in this paper. Specifically, we are investigating methods to improve the computational feasibility of AST, with a particular focus on marine ColAv and computationally expensive simulations.

**Adaptive Stress Testing (AST).** Adaptive stress testing (AST) aims to find the most likely failure event of a system structured as a Markov decision process (MDP) (Lee et al. 2015, 2020). RL is applied to the MDP and uses previous experiences to effectively search a typically vast state space, finding likely failure events. Such events can be used to further guide the development and improvement of the SUT. The search is directed by a reward function that counter-intuitively rewards collisions and close misses with high transition probabilities.

When introduced, AST was used to find probable failure events that were not handled by ACAS X, a novel collision avoidance system for aircraft (Lee et al. 2015). To search for likely failure events, Lee *et al.* apply Monte Carlo tree search (MCTS). MCTS searches thousands of simulations with varying sequences of disturbances to find likely sequences leading to near mid-air collisions (NMACs). MCTS directs search towards sequences that are similar to those that are most promising so far, while also considering less explored sequences. As a result, AST proves effective in uncovering failure states, allowing a comparison of the current collision avoidance system, TCAS, and ACAS X.

(Koren et al. 2018) apply two versions of AST to compare the use of MCTS and deep RL (DRL) when searching for failure events. Koren *et al.* implement a simulation containing an autonomous vehicle (AV) approaching a pedestrian crossing. The pedestrians are controlled by AST to find paths where the AV is unable to avoid hitting the pedestrians. Through three different experiments, Koren *et al.* find that AST using DRL outperforms MCTS, in the sense of finding more likely failure scenarios with a higher efficiency. (Corso et al. 2019) argue that the results found by Koren *et al.* (Koren et al. 2018) are mostly uninteresting, as most failures were caused by pedestrians walking into an already-stopped AV. Thus, it acted correctly in the given scenario. Another problem that Corso *et al.* study is the lack of variations in the failure scenarios found by Koren *et al.*. Their proposed solution is to introduce two enhancements to the reward functions. With the implementation of the augmented reward function, a short experiment is run, resulting in multiple failures caused by the AV acting improperly. Corso *et al.* provide interesting results in a domain beyond aerospace, thus inspiring our study in the marine domain.

**Monte Carlo Tree Search in Games.** Monte Carlo Tree Search is often used in deterministic, sequential board games, including in AlphaZero. AlphaZero has made substantial advances within *Computer Chess*, *Computer Shogi* (Silver et al. 2017), and *Computer Go* (Silver et al. 2016). A core concept introduced in AlphaZero is the use of MCTS in conjunction with both value and rollout policy networks. Through a simulation-based rollout search, MCTS seeks to find moves, among all legal moves, with the highest likelihood of a win. Aiding in this search are value and rollout policy networks. The value network uses previous games to better evaluate what actions to explore, while the rollout policy network improves the actions chosen in a look-ahead over games. Gradually generating a better rollout policy in turn improves the virtual oponent's skill. The resulting model learns through self-play, needing no human expert input. While very different from scenario-based stress testing using simulators, AlphaZero's use of MCTS in games inspires several choices made in our work (see Section 3).

**Costly Fitness Functions and Simulations.** The computational cost of running a simulation is a manifestation of a broader problem when AI is applied in science and engineering. This is the problem of fitness function evaluation cost (Snoek, Larochelle, and Adams 2012; Lee et al. 2021; Luong et al. 2021; Mengshoel et al. 2022). Here, "cost" may

[3]Convention on the International Regulations for Preventing Collisions at Sea

refer to computational cost, energy cost, engineering cost, or other costs. There are several variants of costly fitness functions (CFFs), including the following. First, fitness function evaluation may have consistently high computational cost, for instance when simulations are used in AI (Lee et al. 2015; Porres, Azimi, and Lilius 2020; Koren, Corso, and Kochenderfer 2020; Hjelmeland et al. 2022). CFF applications include black-box topology optimization via simulation of mechanical structures (Guirguis et al. 2020; Zavala et al. 2014) and neural architecture search in ML (White, Nolen, and Savani 2021). A second CFF variant exists when the cost of fitness function evaluation varies drastically over a search space (Snoek, Larochelle, and Adams 2012; Luong et al. 2021). Such CFFs can be found in ML. Specifically, ML feature selection using wrappers (Kohavi and John 1997) is an application that often have a varying CFF (Mengshoel et al. 2022).

Studies on handling CFFs fall into different categories. One can directly perform CFF minimization by using fitness function approximation or surrogates (Shi and Rasheed 2010). Non-revisiting methods form another approach (Yuen and Chow 2007; Lou, Yuen, and Chen 2021). Fitness function cost minimization via carefully optimizing an algorithm's heuristics and hyperparameters (DaCosta et al. 2008; Mengshoel, Wilkins, and Roth 2011; Snoek, Larochelle, and Adams 2012; Karafotias, Hoogendoorn, and Eiben 2015) is another approach. Here, methods for optimizing heuristics and hyperparameters to minimize the computational cost of problem of feature selection are important (Kohavi and John 1997; Mengshoel, Ahres, and Yu 2016; Yu, Kveton, and Mengshoel 2017; Mengshoel et al. 2022). Our research in this paper is related to these works, but specifically targets computational simulation cost associated with the sequential AST setting.

## 3 The MCFS-AP Method

We partition the states $S$ created with a simulation $\mathcal{S}$ into failure events $E \subset S$ and non-failure events $\bar{E} \subset S$. Failure events $E$ are of particular interest to us, as they represent collisions between mA2 and one or more other vessels or adversaries. When $\mathcal{S}$ terminates, at a terminal time $t_{\text{end}}$ with a corresponding state $s_{t_{\text{end}}}$, there is either a collision $s_{t_{\text{end}}} \in E$ or not $s_{t_{\text{end}}} \notin E$. Let $R_E$ be reward at failure and $I$ an indicator function. We define the term $I_E$ to indicate a collision: $I_E = R_E \cdot I(s_{t_{\text{end}}} \in E)$. Let $d$ be the closest distance between mA2 and an adversary. We define $I_{\bar{E}}$ to indicate a non-collision: $I_{\bar{E}} = d \cdot I(s_{t_{\text{end}}} \notin E)$. The following objective function is used for AST (Lee et al. 2020):

$$f(x_0, \ldots, x_{t_{\text{end}}}) = \left[ \prod_{t=0}^{t_{\text{end}}} P(x_t|s_t) + I_E - I_{\bar{E}} \right]. \quad (1)$$

Optimum $r^*$ is then defined as:

$$r^* = \max_{x_0, \ldots, x_{t_{\text{end}}}} f(x_0, \ldots, x_{t_{\text{end}}}). \quad (2)$$

In our domain, (1) and (2) express a goal to minimise the distance between our vessel and other objects until a collision occurs. Further, the joint likelihood of the disturbance sequence $[x_0, \ldots, x_{t_{\text{end}}}]$ should be maximised. After all, we seek the most likely sequence of disturbances that results in a failure event, a collision.

For a deterministic time step simulator $\mathcal{S}$ with a fixed initial state $s_0$, any state $s_n$ can be represented as a unique sequence of disturbances $[x_0, \ldots, x_{n-1}]$. This sequence consists of the disturbances that were originally applied to $\mathcal{S}$ at each time step from $s_0$ to $s_n$. Revisiting the state $s_n$ is done by retracing the disturbances, stepping through each disturbance $x_t$ in $[x_0, ..., x_{n-1}] = x_{0:n-1}$ from $s_0$ of $\mathcal{S}$. Due to the deterministic nature of $\mathcal{S}$, the same state transition is computed given the same disturbance, thus we we arrive at the desired state $s_n$. In other words, when given the same initial state and disturbance sequence, the SUT and environment should change in the same manner, resulting in the exact same set of states at every time step. Such deterministic and noiseless simulators have been shown to work well with AST (Koren et al. 2018) and is what we focus on here.

Using (1) we compute an $\hat{r}^*$ that approximates $r^*$. We are also interested in the corresponding state $\hat{s}_n^*$ or disturbance sequence $\hat{x}_{0:n-1}^*$. To compute these, we introduce Monte Carlo Forrest Search with Action Pruning (MCFS-AP), see Algorithm 1.

### 3.1 Monte Carlo Forrest Search (MCFS)

We now explain the pseudo-code of MCFS-AP (see Algorithm 1). MCFS-AP inputs and parameters are the simulator $\mathcal{S}$ as well as wrappers for the value and rollout policy neural networks. MCFS-AP outputs the most likely disturbance or action sequence $\hat{x}^*$ leading to a failure event, found via search by optimising (1). For simplicity we use in MCFS-AP just $\hat{x}^*$ instead of $\hat{x}_{0:n-1}^* = [\hat{x}_0^*, \ldots, \hat{x}_{n-1}^*]$.

The algorithm consists of several loops and nested loops. The outer **for**-loop, lines 2–24, builds $n_t$ search trees. The **for**-loop in lines 5–24 handles action pruning and state memoization. Here the simulator is fast forwarded to the state of current root, deepening as more actions are pruned in line 24 (see Section 3.2). The **for**-loop in lines 8–23 is similar to previous MCTS algorithms for AST (Lee et al. 2015, 2020) and consists of three **while**-loops that perform MCTS. The **while**-loop in lines 10–13 performs selection and progressive widening. For value computation $\mathcal{V}$ in line 12, $\mathcal{V}_0$ represents a baseline using normal UCT-selection (Kocsis and Szepesvári 2006) while $\mathcal{V}_1$ uses UCT with a value neural network weight (see Section 3.3). The **while**-loop in lines 14–17 performs rollout including simulation with $\mathcal{S}$. In line 16, the rollout action is computed using $\mathcal{R}$ (see Section 3.3). In lines 18–19, reward $\bar{r}$ and disturbances $\bar{x}$ from the simulation $\mathcal{S}$ are computed by means of (1) and the best-so-far $\hat{r}^*$ and $\hat{x}^*$ are potentially updated. Using (1), higher reward is given for a simulation that results in failure via high transition probabilities. The **while**-loop in lines 20–23 backpropagates up the search tree, updating the expected reward of a node using information from $\mathcal{S}$.

The main additions of MCFS-AP compared to previous MCTS methods for AST are: (i) the approach to search tree construction and (ii) the use of neural networks when selecting actions during the selection and rollout process.

Algorithm 1: MCFS-AP

**Input**: Simulator $\mathcal{S}$, Value Computation $\mathcal{V}$, Rollout Policy Computation $\mathcal{R}$

**Parameter**: Number of trees $n_t$, number of prune steps $n_d$, and search loops at each root $n_l$.

**Output**: Most likely sequence of actions $\hat{x}^*$ resulting in a failure event.

```
1:  $\hat{x}^*, \hat{r}^* \leftarrow \emptyset$
2:  for 1 to $n_t$ do
3:      $s_R \leftarrow$ new Node($\emptyset$)
4:      $\mathcal{S}$.initiate()
5:      for 1 to $n_d$ do
6:          $\mathcal{S}$.memoizeState()
7:          $s_C \leftarrow s_R$
8:          for 1 to $n_l$ do
9:              $\mathcal{S}$.setMemoizedState()
10:             while $s_C$.nrOfChildren $> 0$ do
11:                 $s_C$.progWiden()
12:                 $s_C \leftarrow \mathcal{V}(s_C)$
13:                 $\mathcal{S}$.step($s_C$.seedAction())
14:             while not $\mathcal{S}$.terminal() do
15:                 $\hat{s} \leftarrow \mathcal{S}$.getState()
16:                 rolloutAction $\leftarrow \mathcal{R}$.getAction($\hat{s}$)
17:                 $\mathcal{S}$.step(rolloutAction)
18:             $\bar{x}, \bar{r} \leftarrow \mathcal{S}$.getReward()
19:             $\hat{x}^*, \hat{r}^* \leftarrow \mathcal{S}$.getBestReward($\bar{x}, \bar{r}, \hat{x}^*, \hat{r}^*$)
20:             while $s_C$ is not $s_R$ do
21:                 $s_C$.update()
22:                 $s_C \leftarrow s_C$.parent()
23:             $s_C$.update()
24:         $s_R \leftarrow s_R$.getRobustChild()
25:         $\mathcal{S}$.step($s_R$.seedAction())
26: return $\hat{x}^*$
```

### 3.2 Action Pruning and Memoization

A key difference between MCFS-AP and the algorithm presented by Lee *et al.* (Lee et al. 2020) is our study of different methods $\mathcal{A}$ of handling disturbances or actions. Method $\mathcal{A}_0$, the baseline, starts every search at depth 0. With action pruning, $\mathcal{A}_1$, we periodically select actions early in the action sequence and actively prune branches of the tree that are not performing as well as the main branch. Our $\mathcal{A}_1$ method is illustrated in Figure 1. Action pruning in MCFS-AP uses a robust child strategy (Chaslot et al. 2008), introduced as final move selection for sequencial games. Lines 6, 9, and 24 in MCFS-AP concern action pruning. We pick, in line 24, the most visited child node of the current root to become the new root, removing all other paths down the tree. Root pruning is performed after $n_l$ search loops of a root node and repeated $n_d$ times.

Closely related to action pruning is memoization $\mathcal{M}$. Memoization $\mathcal{M}_1$ can be applied to the new root node, starting $\mathcal{S}$ at the state represented by the new root, avoiding recalculation of previous states (greyed out for $d = 1$ and $d = n_d$ in Figure 1). When memoization is not done, denoted $\mathcal{M}_0$, recalculation does take place.

The main advantage of MCFS-AP action pruning and memoization is this: By periodically making the search more shallow, we can stop calculating the earlier parts of the simulations. Given a deterministic time step simulator, the end-state of a given disturbance sequence is constant, removing the need for the simulation to rerun what has already been calculated. That said, MCFS-AP moves us slightly away from AST's black-box simulation idea, as we need to start simulation in any state and not just the initial state.

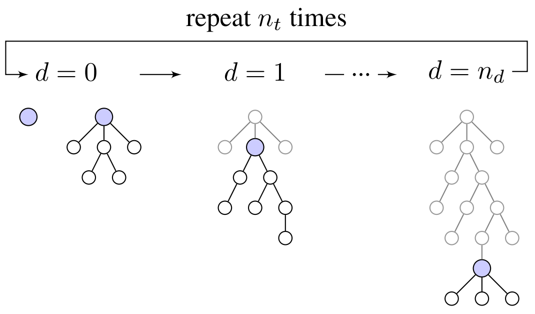


Figure 1: Construction of one MCFS-AP search tree, where nodes represent states and edges represent actions or disturbances. The root node (light blue) is moved downwards as the tree grows. Periodically, a child (black) of the current root node is picked as the new root and nodes in other branches connected to the current root are pruned (grey). After this, subsequent searches start from the new root. Pruning is done $n_d$ times per tree and $n_t$ such trees are constructed.

### 3.3 Rollout Policy and Value Neural Networks

To further improve the effectivity of the search for failure states, we have implemented a rollout policy network and a value network, inspired by Alpha Zero (Silver et al. 2017). The networks use experiences from previous MCFS-AP iterations to better direct the search for failure states.

**Rollout policy computation** $\mathcal{R}$ in line 16 of MCFS-AP proceeds in one of two ways: $\mathcal{R}_0$ is a baseline while $\mathcal{R}_1$ uses a rollout policy neural network. The baseline $\mathcal{R}_0$ is a random rollout policy. Both traditional MCTS and MCTS-SA use a random rollout policy, executing random seed-actions until reaching a terminal state. For $\mathcal{R}_1$, the *rollout policy neural network* gives a prediction of what action is most advantageous given the input state. Specifically, the rollout policy network $\rho_\pi$ uses previous experiences to pick advantageous seed-actions, aiming to give rollouts that terminate with higher rewards. Our rollout policy network takes an input vector $\hat{s}$ representing the state of the simulator. In our naval setting this vecor consists of the coordinates $x$ and $y$, as well as the angle $\theta$, 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 computation** $\mathcal{V}$ in line 12 of MCFS-AP proceeds in one of two ways: $\mathcal{V}_0$ is a baseline while $\mathcal{V}_1$ uses a value neural network. Either way, prediction uses augmented UCT selection:

$$\arg\max_a \left[ Q(s,a) + c_1 \sqrt{\frac{\log N(s)}{N(s,a)+1}} + c_2 v_\theta(\hat{s},a) \right]. \quad (3)$$

For $\mathcal{V}_0$, we put $c_2 = 0$ in (3), giving standard UCT. For $\mathcal{V}_1$, we put $c_2 \neq 0$, thus enabling the value network $v_\theta$ in (3). The *value neural network* $v_\theta$ aims to improve UCT selection by approximating the state-action function $Q(s,a)$. The value network is trained using state-action functions $Q(s,a)$ from previous trees. The goal of the value network is to predict the expected reward of taking any action $a$ from a given state $s$.

The value neural network takes an input vector $\hat{s}$, similar to the rollout policy network, alongside a disturbance $a$. The target is the state-action value $Q(s,a)$.

Both the rollout policy and value networks are trained using statistics from each fully explored tree. As MCFS-AP builds 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.[4]

## 3.4 Marine ColAv Simulation

We structure our system as a Markov decision process in order to apply RL. Thus, a ColAv simulator should advance through discrete state transitions.[5] This is achieved by using discrete time steps of equal length in a simulation $\mathcal{S}$. Following the deterministic time step simulator principles laid out above, we now discuss our two ColAv simulators.

**The Efficient ColAv Simulator $\mathcal{S}_0$.** To test the effectiveness of the AST system, an efficient but simple ColAv simulator $\mathcal{S}_0$ is implemented. The goal of $\mathcal{S}_0$ is to eliminate any noise and uncertainty, resulting in an efficient but low-fidelity simulation with two vessels moving at a fixed speed on a flat plane without other obstacles or disturbances. The SUT is a vessel unaware of its surroundings moving across the plane from left to right, in a straight line. The adversarial vessel starts at the bottom of the plane, facing upward. The heading of the adversarial vessel, given as the disturbance, is altered by RL in searching for a failure event. While $\mathcal{S}_0$ hugely oversimplifies the probable heading of a vessel, it allows for more understandable results through the simplicity of the transition model and intuitive solutions.

**The Complex ColAv Simulator $\mathcal{S}_1$.** The complex simulator $\mathcal{S}_1$ is for fine-grained stress testing of mA2. The simulator, illustrated in Figure 2, encapsulates mA2 and its desired path alongside a variable set of other vessels. A set of disturbances was implemented for AST to tune while searching for failure events. The main disturbance studied here is altering the delay in mA2's sensed position of the adversarial vessel, given by a delay function. A noise function is used to revert the sensed position to a previous one. Adding a noise vector equal to the difference between the current and a specified previous state allows the sensed position and heading to be altered, matching a previous state.

[4]Training was carried out taking the state represented in a root node, setting the positional data as input, and the statistics of the node as target values, before using propagation to tune the network weights (Lytskjold 2022).

[5]A distinction is made between simulation and simulation interface in the MS thesis (Lytskjold 2022). For simplicity we do not make that distinction here.

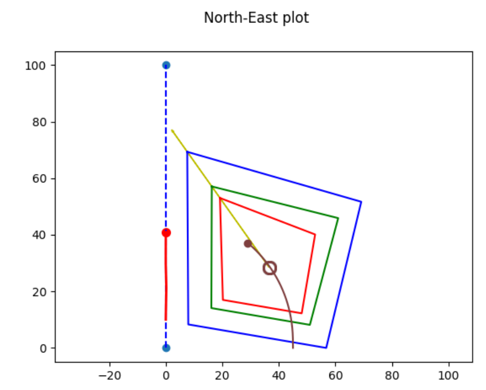


Figure 2: Top-down 2D view of the complex ColAv simulator $\mathcal{S}_1$. The current position and previous path of the mA2 ferry are shown, respectively, as a red circle and a "tail," overlaid on a dashed vertical blue line showing its desired path. An adversarial vessel's path is shown in brown while a yellow line shows current perceived heading. When sensing delay is added, the perceived (perhaps incorrect) position is shown as a brown circle. The perceived position is used to calculate safety zones as indicated by the three polygons (blue, green, and red) surrounding the adversary. Safety zones should not be entered by mA2. Here, a delay in perceived position of the adversarial vessel is introduced.

## 4 Experiments

### 4.1 Experimental Protocol and Setup

The system used for the experiments is an implementation of AST, using two simulator implementations (of $\mathcal{S}_0$ and $\mathcal{S}_1$). The $\mathcal{S}_1$ simulator is a much more detailed simulator than $\mathcal{S}_0$, potentially giving us more realistic and interesting simulation results. The downside of this is increased computational cost. Simulation time goes up, approximately, from 2.2 ms for $\mathcal{S}_0$ to 1 s for $\mathcal{S}_1$, a 450-fold increase.

The new RL agent implemented is MCFS-AP. The goal with our implementation is a generalised version of AST that is able to find failure events more efficiently. The complete implementation of MCFS-AP can be found on GitHub.[6]

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.

### 4.2 Experiment 1: AST ColAv Studies

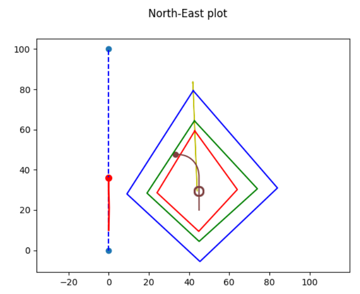
AST has been used to create many valuable mA2 validation scenarios (Hjelmeland et al. 2022; Hjelmeland 2022; Lytskjold 2022). We here focus on one scenario that illustrates an important class of scenarios involving sensing delays.

**Goal.** The goal of this experiment is to study how MCFS-AP is able to find failure events for this scenario: How can

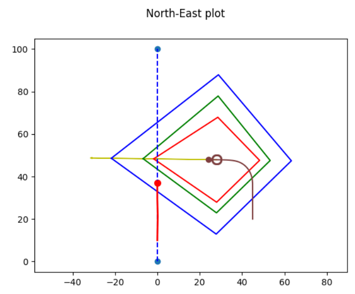
[6]MCFS-AP is fully implemented in Python: https://github.com/bragelyt/Flexible-Adaptive-Stress-Testing

Table 1: Ability of different variants of MCFS-AP versus MCTS-SA to find failure events. All models build build 200 trees. All MCFS-AP variants use action pruning $\mathcal{A}_1$ and memoization $\mathcal{M}_0$.

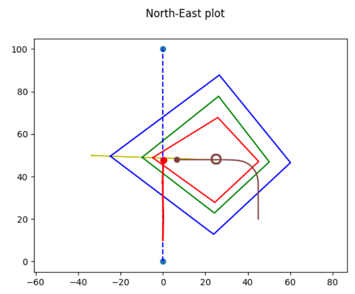
| RL agent<br>Nerual networks | MCFS-AP<br>RP/VN | MCFS-AP<br>RP/- | MCFS-AP<br>-/VN | MCFS-AP<br>-/- | MCTS-SA<br>-/- |
|---|---|---|---|---|---|
| Failures found | **78** | 74 | 41 | 59 | 71 |
| Failure rate | **39%** | 37% | 20.5% | 29.5% | 35.5% |
| Global best $\hat{r}^*$ | **29.092** | 28.650 | 28.784 | 28.061 | 28.245 |
| Average of best $\hat{r}^*$ | **5.818** | 4.776 | -1.426 | 1.927 | 2.304 |



(a) Major mA2 sensing delay, as reflected in the large distance between filled and unfilled brown circles, masks the adversarial vessel's actions.



(b) Reduced delay, reflected in the small distance between filled and unfilled brown circles, results in mA2 stopping for the adversarial vessel.



(c) The mA2 ferry is unable to avoid side collision with the adversary as its momentum is too high and there is again a major sensing delay.

Figure 3: Results of sensor delay experiment for the mA2 ferry (red filled circle) with rapid turning of the adversary (brown filled circle), shown in a top-down 2D view. The course of mA2 is indicated with a vertical line segment (solid red and dashed blue) at $x = 0$. Three time steps of the most likely failure event are shown, created using MCFS-AP/RP/VN with $\mathcal{S}_1$ simulation.

an adverserial sharp turn sensed with varying delay impact mA2's perceived course for the adversarial vessel?

**Metod and Data.** The complex simulator $\mathcal{S}_1$ is used with MCFS-AP/RP/VN, resulting a many interesting simulation runs (Lytskjold 2022). One of them is studied here.

**Results.** Examining the most likely failure event, illustrated in Figure 3, we see that varying delays are being added during the sharp turn. Clearly, such variation can mask an adversarial vessel's change in course. While not fully reflected here, the sensing delay fluctuates substantially. Towards the final time steps before the collision, the delay is again relatively high for a short time. This results in a miscalculation of the safety zones, see Figure 3c, tricking the ColAv system. This results in mA2 speeding up, making the momentum gained too great to slow down during the last time steps when it becomes clear that the two vessels may collide.

### 4.3 Experiment 2: Improving Search Efficiency

We study empirically three main components of MCFS-AP: the rollout policy network, the value network, and action pruning without memoization.

**Goal.** As all three components directly alter the paths chosen when traversing a search tree, this experiments aims to measure how this alteration impacts the search accuracy, as well as how the different components impact the search time. This experiment is run using simulator $\mathcal{S}_0$, in order to get much quantitative data. As the runtimes of this simulator are small, we are not using memoization here.

**Method and Data.** Five different models were implemented to test the impact of the three main components. The first three models are used to directly test the impact of the rollout policy network and the value network. Here we implement an MCFS-AP model for each network, using either the rollout policy network (RP), the value network (VN), or both. These models are referred to as MCFS-AP/RP/- (using $\mathcal{R}_1$ and $\mathcal{V}_0$), MCFS-AP/-/VN (using $\mathcal{R}_0$ and $\mathcal{V}_1$), and MCFS-AP/RP/VN (using $\mathcal{R}_1$ and $\mathcal{V}_1$) respectively.

We consider two baseline methods. The first baseline is MCFS-AP/-/- (using $\mathcal{R}_0$ and $\mathcal{V}_0$). The second baseline is our implementation of MCTS-SA (Lee et al. 2020). All MCFS-AP variants use action pruning $\mathcal{A}_1$ with a maximum prune depth of 18 and 500 nodes being added at each root. This results in 9,000 nodes and simulations per search. Each search using MCTS-SA runs 9,000 simulations. Each method's search is run 200 times, resulting in a total of 1,800,000 simulations per method. The neural nets used in the MCFS-AP variants are trained on data from the previous trees built by that method in the current experiment.

**Results.** To compare the efficiency of different methods, we build on the failure finding rate presented in Table 1. Specifically, we study how approximate relative failure finding rate depends on simulation runtime. The results, as presented in Figure 4, do not factor in time saved through memoization. In other words, we always use the full simulations. Importantly, failure rate is best for MCFS-AP/RP/VN in Table 1. This result is also reflected for high simulation runtimes in Figure 4, where both MCFS-AP/RP/VN and MCFS-AP/RP/- slightly outperform MCTS-SA for high $x$-

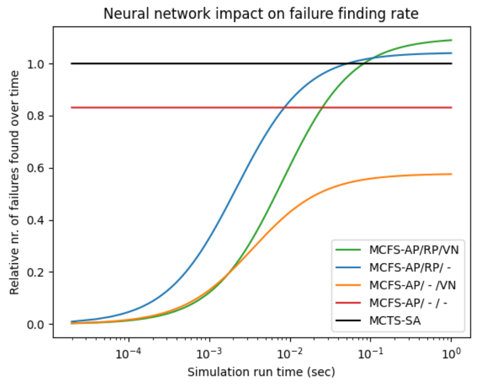


Figure 4: Relative rate of finding failures over time ($y$-axis), depending on full simulation run time ($x$-axis). Memoization is not enabled, but action pruning is: $\mathcal{M}_0$ and $\mathcal{A}_1$.

Table 2: 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 |

values.

## 4.4 Experiment 3: Impact of Memoization

A key idea in MCFS-AP is to mimic MCTS in sequencial games, locking down moves through action pruning. By memoizing an action or disturbance sequence for $\mathcal{S}$, we can later recall the result of this sequence, only requiring us to calculate states reached through later actions.

**Goal.** This experiment aims to measure memoization's impact on MCFS computation time. This is done through perfoming action pruning at incremental depths, also memoizing or saving the state reached through a locked sequence of disturbances or actions (see Figure 1).

**Method and Data.** The experiment is run using $\mathcal{S}_1$ with runtimes of around 1 s and a maximum of 25 steps. Memoization $\mathcal{M}_1$ is performed for each root chosen through action pruning down to a depth of $n_d = 22$. As a baseline MCTS-SA was used; it runs complete simulations. While MCTS-SA calculates all states in every simulation, MCFS-AP only calculates the states after the memoized state. At initial root depth of 0, MCFS-AP starts simulation at time step $t_0$ and thus computes all 25 steps. In contrast, at the maximum root depth of 22, MCFS-AP starts simulation at timestep $t_{22}$ and calculates only the last 3 time steps.

**Results.** Comparing the run time of MCFS-AP to the run time of MCTS-SA, which always performs full simulations, we see a substantial decrease in average simulation time, presented in Table 2. This decrease in average simulation time for MCFS-AP/-/- results in a faster overall search time, allowing MCFS-AP to search the same number of disturbance sequences as MCTS-SA at approximately 55% of the

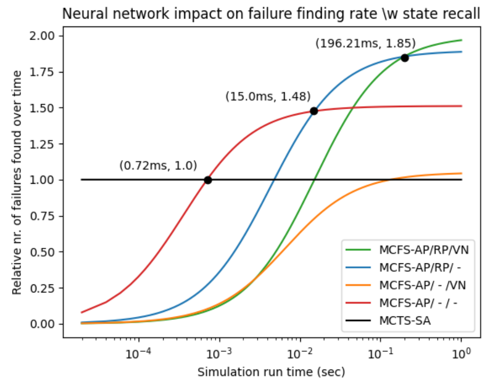


Figure 5: Relative rate of finding failures over time ($y$-axis), depending on average simulation time ($x$-axis). MCFS-AP methods use action pruning $\mathcal{A}_1$ and memoization $\mathcal{M}_1$.

time if the simulation time is sufficiently long.

In order to further understand the effect of memoization or action recall, we combine the findings of Figure 4 with the runtime improvements presented in Table 2. The results, presented in Figure 5, show an approximation of the efficiency of different methods (along $y$-axis) given varying simulation times (along $x$-axis). MCFS-AP/RP/- is best in the time interval from 15ms to 196.21ms, while MCFS-AP/RP/VN is best for higher simulation run times. This demonstrates the benefit of MCFS-AP/RP/VN when simulation times are high, which is a main focus in this work, and a problem that can also be found in other areas of science and engineering.

## 5 Conclusion and Future Work

Stress testing of engineered or natural systems can be performed by encapsulating them in a simulator. We seek to address the problem of long simulation times through our proposed method, Monte Carlo Forrest Search with Action Pruning (MCFS-AP). MCFS-AP aims to improve the efficiency of MCTS through two methods: (i) a neural network rollout policy and a value neural networ as well as (ii) periodically memoizing the internal state of the simulator, removing the need to calculate frequently visited states multiple times. Combining neural networks and memoization gives promising results. Memoization with state recall can reduce the average simulation time by 45% and when combined with the neural network models (MCFS-AP/RP/VN) results in a potential doubling of the method's efficiency.

Future work opportunities include the following. First, RL remains compute-intensive, even in light of the improvements discussed. For example, we are batching 20 simulation steps for each RL step in experiments. Consequently, further improvements in RL efficiency would be welcome. One could use parallel computing, as the methods discussed are amendable to such techniques. Second, while memoization was introduced as an add-on to action pruning, the two techniques are independent. Therefore, we propose to further study just memoization for nodes, depending on their depth and number of visits. This could, we hypothesize, reduce total search time.

## Acknowledgements

Bjørn-Olav Holtung Eriksen and Øyvind Smogeli gave essential help with the mA2 simulation. Ritchie Lee provided invaluable feedback on AST-related topics.

## References

Brekke, E. F.; Eide, E.; Eriksen, B.-O. H.; Wilthil, E. F.; Breivik, M.; Skjellaug, E.; Helgesen, Ø. K.; Lekkas, A. M.; Martinsen, A. B.; Thyri, E. H.; et al. 2022. milliAmpere: An Autonomous Ferry Prototype. In *Journal of Physics: Conference Series*, volume 2311.

Chaslot, G. G. M. J.-B.; Winands, M. H.; van den Herik, H. J.; Uiterwijk, J. W. H. M.; and Bouzy, B. 2008. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3): 343–357.

Corso, A.; Du, P.; Driggs-Campbell, K.; and Kochenderfer, M. J. 2019. Adaptive Stress Testing with Reward Augmentation for Autonomous Vehicle Validation. arXiv:1908.01046.

DaCosta, L.; Fialho, A.; Schoenauer, M.; and Sebag, M. 2008. Adaptive Operator Selection with Dynamic Multi-Armed Bandits. In *Proc. GECCO*, 913–920.

Guirguis, D.; Aulig, N.; Picelli, R.; Zhu, B.; Zhou, Y.; Vicente, W.; Iorio, F.; Olhofer, M.; Matusik, W.; Coello Coello, C. A.; and Saitou, K. 2020. Evolutionary Black-Box Topology Optimization: Challenges and Promises. *IEEE Transactions on Evolutionary Computation*, 24(4): 613–633.

Hjelmeland, H. W. 2022. *Safety Validation of Marine Collision Avoidance Systems using Adaptive Stress Testing*. Master's thesis, NTNU, Trondheim, Norway.

Hjelmeland, H. W.; Mengshoel, O. J.; Eriksen, B.-O. H.; and Lekkas, A. M. 2022. Identification of Failure Modes in the Collision Avoidance System of an Autonomous Ferry using Adaptive Stress Testing. In *14th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles*.

Karafotias, G.; Hoogendoorn, M.; and Eiben, A. E. 2015. Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation*, 19(2): 167–187.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proc. ECML*, 282–293.

Kohavi, R.; and John, G. H. 1997. Wrappers for Feature Subset Selection. *Artificial Intelligence*, 97(1-2): 273–324.

Koren, M.; Alsaif, S.; Lee, R.; and Kochenderfer, M. J. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, 1–7.

Koren, M.; Corso, A.; and Kochenderfer, M. J. 2020. The Adaptive Stress Testing Formulation. arXiv:2004.04293.

Lee, E. H.; Eriksson, D.; Perrone, V.; and Seeger, M. W. 2021. A Nonmyopic Approach to Cost-Constrained Bayesian Optimization. *CoRR*, abs/2106.06079.

Lee, R.; Kochenderfer, M. J.; Mengshoel, O. J.; Brat, G. P.; and Owen, M. P. 2015. Adaptive Stress Testing of Airborne Collision Avoidance Systems. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 6C2–1 – 6C2–13.

Lee, R.; Mengshoel, O. J.; Saksena, A.; Gardner, R.; Genin, D.; Silbermann, J.; Owen, M.; and Kochenderfer, M. J. 2020. Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning. *JAIR*, 1165–1201.

Lou, Y.; Yuen, S. Y.; and Chen, G. 2021. Non-Revisiting Stochastic Search Revisited: Results, Perspectives, and Future Directions. *Swarm and Evolutionary Computation*, 61.

Luong, P.; Nguyen, D.; Gupta, S.; Rana, S.; and Venkatesh, S. 2021. Adaptive Cost-Aware Bayesian Optimization. *Knowledge-Based Systems*, 232.

Lytskjold, B. 2022. *Adaptive Stress Testing: Reinforcement Learning using Monte Carlo Tree Search with Neural Network Policies*. Master's thesis, NTNU, Trondheim, Norway.

Mengshoel, O. J.; Ahres, Y.; and Yu, T. 2016. Markov Chain Analysis of Noise and Restart in Stochastic Local Search. In *Proc. IJCAI*, 639–646.

Mengshoel, O. J.; Flogard, E. L.; Yu, T.; and Riege, J. 2022. Understanding the Cost of Fitness Evaluation for Subset Selection: Markov Chain Analysis of Stochastic Local Search. In *Proc. GECCO*, 251–259.

Mengshoel, O. J.; Wilkins, D. C.; and Roth, D. 2011. Initialization and Restart in Stochastic Local Search: Computing a Most Probable Explanation in Bayesian Networks. *IEEE Transactions on Knowledge and Data Engineering*, 23(2): 235–247.

Porres, I.; Azimi, S.; and Lilius, J. 2020. Scenario-based Testing of a Ship Collision Avoidance System. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 545–552.

Shi, L.; and Rasheed, K. 2010. A Survey of Fitness Approximation Methods Applied in Evolutionary Algorithms. In Tenne, Y.; and Goh, C.-K., eds., *Computational Intelligence in Expensive Optimization Problems*, 3–28. Springer Verlag.

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; Driessche, G.; Schrittwieser, J.; Antonoglou, I.; et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815.

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proc. NeurIPS*, 2951–2959.

White, C.; Nolen, S.; and Savani, Y. 2021. Exploring the Loss Landscape in Neural Architecture Search. In *Proc. UAI*, 654–664.

Yu, T.; Kveton, B.; and Mengshoel, O. J. 2017. Thompson Sampling for Optimizing Stochastic Local Search. In *Proc. ECML-PKDD*, 493–510.

Yuen, S. Y.; and Chow, C. K. 2007. A Non-Revisiting Genetic Algorithm. In *Proc. CEC*, 4583–4590.

Zavala, G. R.; Nebro, A. J.; Luna, F.; and Coello, C. A. C. 2014. A Survey of Multi-Objective Metaheuristics Applied to Structural Optimization. *Structural and Multidisciplinary Optimization*, 49(4): 537–558.