Automatic Inference of Fault Trees using Reinforcement Learning
Master Thesis

Jander Gilbers

Supervisors:
Lisandro A. Jimenez-Roa, Matthias Volk, Moritz Hahn and Mariëlle Stoelinga

December 4, 2022

Contents

1 Introduction

1.1 Problem description

Fault Tree Analysis (FTA) is a widely used method for dependability evaluation of systems [1]. It is thus an important part of risk analysis in industries such as medical and aerospace, where system failure could have drastic consequences on people's lives. Fault Trees (FTs) [2] are graphical representations of a system that show how failures of individual parts of a system can cause a system failure. Traditionally, FTs are manually built with expert knowledge. This can be tedious and error-prone, which means it is desirable to find other, automatic ways to infer FTs.

The automatic inference of Fault Trees is a process where the structure of an FT is obtained without any manual input. There are three main groups of inference methods: knowledge-based, model-based and data-driven. Knowledge-based methods use known information about basic components of the system and their interactions to analyse the system, the information is known through human interaction [3]. Model-based methods translate existing models of the system into FTs [4], and data-driven methods use databases as their source of information.

There already exist a few data-driven Fault Tree inference algorithms. The LIFT [5], FT-BN [6], FT-EA [7] and FT-MOEA [8] algorithms. One of the main limitations of these methods is associated with scalability, which in this context refers to the ability to infer FT structures from datasets with larger number of basic events. This is challenging, as the solution space increases exponentially with the number of basic events, causing current algorithms to underperform in real-world industrial applications. Another hurdle that is often encountered is the noise of real-world data.

Research in other fields of computer science use machine and deep learning to infer/optimize graph-based models, such as designing convolutional-neural networks [9]. In that study, Q-learning is used to choose layers of the network in a stepwise fashion, resulting in a network that outperforms even handmade networks. In [10], symbolic equations are inferred via graphical neural networks, to obtain equations from data sets. In [11], reinforcement learning has been used as part of a solution for specifications of the Clock Constraint Specification Language (CCSL), to bridge the knowledge gap of requirement engineers to quickly and accurately figure out CCSL specifications from natural language-based design descriptions.

Inspired by the above work, we explore a path that, to the best of our knowledge, has not been explored yet. We look at inferring FTs using Reinforcement Learning (RL). RL focuses on goal-directed learning from interaction [12]. We take the first steps to see how RL can be used as a solution to the problem of FT inference. We answer the question: *How can Reinforcement Learning be used for the automatic inference of Fault Trees?* To do this we use classical Q-learning techniques, with a few optimizations. We have translated the inference of FTs into a RL framework. This framework has been implemented into an algorithm we call FT-RL. FT-RL was then optimized and tested with several case studies, to find out the limits of this approach.

This thesis is structured as follows. Section 2 provides an overview of the theoretical background, which discusses FTs and RL. Already existing solutions to the problem of FT inference are also found in that Section. The RL framework for FT inference can be found in Section 3. The implementation of the algorithm and the optimizations can also be found there. Sections 4 through 6 show the results of FT-RL. Section 4 shows the setup and case studies of the experiments. Section 5 shows the tuning of the hyper-parameters of FT-RL. Section 6 tests the performance of FT-RL, shows where its weaknesses lie and compares it to the state-of-the-art FT inference algorithm FT-MOEA.

1.2 Research objective

General research question: How can Reinforcement Learning be used to automate the inference of Fault Trees?

We will answer this question by trying to translate automatic FT inference to a Reinforcement Learning framework. Then we will implement this framework into an algorithm. The performance of this algorithm will then be tested and compared to the state-of-the-art. The research will have three main groups of sub-questions to answer. The Sections in which the questions are answered are shown as well for easy navigation.

R1. General:

- 1.1 How can FT inference be formulated as a reinforcement learning problem? (Section 3.1)

R2. Specific:

- 2.1 How should the rewards be given? (Section 3.1)
- 2.2 What metrics should be used? (Section 5)
- 2.3 What is the best prioritization of metrics for RL? (Section 5.1)

R3. Performance:

- 3.1 How fast can the algorithm produce a correct FT? (Section 6)
- 3.2 How much does performance decrease with increasing FT complexity, e.g. increasing number of BEs? (Section 6)

R4. Validation:

- 4.1 How does this new solution compare to existing techniques? (Section 6.6)

2 Background

This section presents the background knowledge needed to understand this research. It gives a basic understanding of Fault Trees and Reinforcement Learning. We also present the state-of-the-art on data-driven inference of Fault Trees.

2.1 Fault Trees

A *Fault Tree* (FT) is a model that is used in the risk analysis of a system [1]. A Fault Tree is a rooted directed acyclic graph, that represents how failures of subsystems propagate in a system. There are several extensions of FTs [2], such as dynamic fault trees, extended fault trees, repairable fault trees and fuzzy fault trees. Here we focus on static fault trees.

Definition 2.1 (Fault Tree) *A Fault Tree (FT) is a 5-tuple* $F = \langle BE, G, T, I, TE \rangle$ *where:*

- BE *is a set of basic events, which are parts of the system that can fail on their own.*
- G *is a set of logic gates.*
- $E = BE \cup G$ *denotes the set of all elements.*
- $T : G \longrightarrow GateType$ *is a function that describes the type of each gate, with* $GateType = \{And, Or\}$.
- $I : G \longrightarrow \mathcal{P}(E) \backslash \emptyset$ *describes the inputs of each gate (*$\mathcal{P}(E)$ *is the powerset of* E*). The inputs of a gate cannot be empty.*
- $TE \in E$ *is a unique root called the top event.*

The graph of an FT is a directed acyclic graph formed with (E, I). The TE in that graph is reachable by all other elements. The possible gates in G are the AND and OR gates:

- And gate is a tuple {And, I, O} where O outputs *true* if every i ∈ I occurs.
- Or gate is a tuple {Or, I, O} where O outputs *true* if at least one i ∈ I occurs.

Figure 1 shows an example of a FT. The Fault Tree is from a case study on a Container Seal Design (CSD) [13], that will also be used later in the experiments in Section 5. The Fault Tree contains two OR gates, two AND gates and six basic events. The TE is not displayed, but is implied above the highest OR gate.

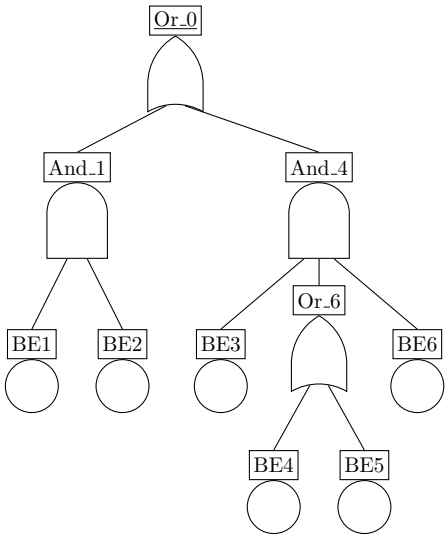


Figure 1: A Fault Tree abstracted from a case study on Container Seal Design (CSD). It has two OR gates, two AND gates and six basic events.

2.1.1 Failure data

The failure dataset is a labelled binary dataset [14]. Each label represents one of the events: a BE or the TE. The values 0 or 1 represent the failure status of the event, where 1 indicates failure. Table 1 gives an example that corresponds to the FT in Figure 1. The possible BE combinations together with the resulting TE status are shown. The Count is the amount of times that combination of BEs has been observed. This failure data (FD) can be used to evaluate FTs. This can be done with a function P that gives the value of the TE for each set of BEs in the N datapoints in the failure dataset. For this example: $P^{FD}(0,0,0,0,1,0) = 0$ and $P^{FD}(1,1,0,0,0,0) = 1$. This function can then be compared to the P function of another failure dataset of an inferred FT. More overlap between the two P functions means that the inferred FT is more similar to the original failure dataset. This approach of comparing an FT to the failure data was used by a previous research on automatic FT inference [8].

BE1	BE2	BE3	BE4	BE5	BE6	TE	Count
0	0	0	0	0	0	0	745
0	0	0	0	0	1	0	376
0	0	0	0	1	0	0	451
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	0	0	0	0	1	711
1	1	0	0	0	1	1	264
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	1	0	1	87
1	1	1	1	1	1	1	101

Table 1: Example failure data set, based on Figure 1.

2.1.2 Minimal Cut Sets

From the failure data, we can identify a *Minimal Cut Set (MCS)*. This is the minimal set of BEs which are needed to yield a system failure in TE.

Definition 2.2 (Minimal cut sets) *A cut set $C \subseteq BE$ for FT $\mathcal{F}$ is a set of BE such that $\mathcal{F}$ fails if all BE in C fail, i.e. Top $\in$ f(C). Where for a set of failed BE C in $\mathcal{F}$, function f(C) gives all failed elements. A cut set C is minimal if each subset C' $\subsetneq$ C is not a cut set. The minimal cuts sets (MCS) for an FT $\mathcal{F}$ are denoted by $M_{\mathcal{F}}$.*

For example, looking again at Figure 1, BE1 and BE2 together cause the TE to fail, thus the set ⟨BE1, BE2⟩ is part of the MCS. BE1 with BE2 and BE3 would also cause the TE to fail, but BE3 can be left out, thus the set ⟨BE1, BE2, BE3⟩ is not part of the MCS. The MCS for the example FT are: ⟨⟨BE1, BE2⟩, ⟨BE3, BE4, BE6⟩ and ⟨BE13 BE5, BE6⟩. These MCSs can also be used to evaluate the correctness of an FT. The comparison of the MCS obtained from the failure data to the MCS obtained from an FT is a good indication of the accuracy of the FT w.r.t. the failure data.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is the computational approach to learning from interaction [12]. Figure 2 shows a schematic view of the RL process. RL makes use of an *agent*. At each time step t, the agent evaluates the state s that it is in, and chooses the best action a that it is allowed to do in that state. This action then influences the environment. At step $t+1$, the environment gives the agent a new state and a *reward*. With this reward, the agent learns which actions to take. The agents' aim is to maximize the reward. This model is a Markov decision process.

Definition 2.3 (Markov decision process) *A Markov decision process (MDP) is a tuple of four elements* $M = \langle S, A, P, R \rangle$ *with:*

- S *- a set of States* $s \in S$
- A *- a set of Actions* $a \in A$
- $P_a(s, s')$ *- the transition probability function from state* s *to* s'*, when action* a *is taken. Whith* $S \times S \rightarrow$ *[0,1] and* $\sum_{s' \in S} P_a(s, s') = 1$*.*
- $R_a(s, s')$ *- the reward function from state* s *to* s'*, when action* a *is taken. It gives the reward value* r*.*

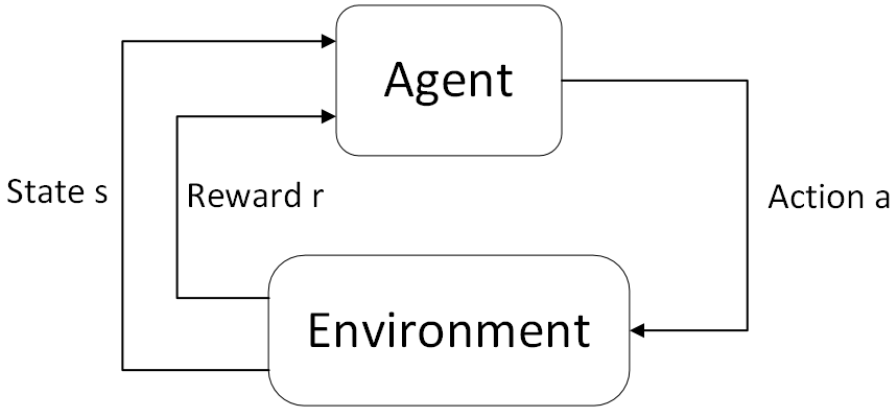


Figure 2: View of the main RL concept.

With these tools we can explain the following important terms in RL:

- **Policy**
  A mapping of the set of states S to the set of actions A, $S \rightarrow A$. The agent uses the policy to determine which actions to take in a given state. An optimal policy will take actions so that the maximum possible amount of reward is obtained when the process is finished.
- **Reward Signal**
  The reward r is given to the agent at each time step. It is the agents job to pick the right actions to maximize the overall received reward.
- **Value Function**
  The value function helps the agent with picking actions that give long term rewards. The value of taking an action can be seen as the expected total reward of taking that action. This means that an action could have a high immediate reward, but a low value.

2.2.1 Q-learning

Q-learning is a RL method that keeps track of Q-values to let the agent make decisions [15]. The *policy* makes use of these Q-values, by always picking the highest one. Q-values are calculated with a current state and a selected action. Q-values are updated each time an action is taken, taking in the reward and value of that action in the current state.

The formula for Q-learning [12] at time t is:

$$Q(s_t, a_t) := (1 - \alpha)Q(s_t, a_t) + \alpha[r + \gamma \max_a Q(s_{t+1}, a)] \tag{1}$$

Where s is the state, a is the action, r is the reward, α is the learning rate and γ is the diminishing factor. This update to the Q-values occurs at each time step t. s_{t+1} is obtained from taking action a_t on state s_t. In other words: the term $max_a Q(s_{t+1}, a)$ is the maximum Q-value for the *next* state.

This is the *value function* that helps with long term rewards. The discount factor γ determines how much these future rewards are valued compared to the immediate reward r. Finally, α determines how much the reward and future rewards are valued compared to the old Q-value.

2.2.2 Exploration and Exploitation

Since RL is all about learning from experience, the agent first needs to have experiences to learn from. In the *exploration* phase of RL, the agent tries random solutions to see which ones work best. Since this phase explores randomly, it avoids local maxima in the reward space. Once the agent has had many different experiences, it can start the *exploitation* phase, where it uses what it learned to make choices. In this second phase the agent will not always pick the best choices, as neither fully exploring nor fully exploiting will give a good result. It will instead use an ϵ greedy strategy [9], where either an exploratory (random) or a exploitative (greedy) action is taken with probability $1 - \epsilon$ and ϵ, respectively. Throughout the exploitation phase, ϵ is then raised from 0 to 1 to get more exploitative actions.

In the context of Q-learning, all Q-values are zero at the start. Only by having an exploration phase, can the Q-values be updated with rewards. The exploitation then uses these Q-values to make the greedy decisions.

2.3 Fault Tree inference: State-of-the-art

2.3.1 Fault Tree inference

The automatic inference of Fault Trees is a process where the structure of an FT is obtained without any manual input. There are three main groups of inference methods: knowledge-based, model-based and data-driven. Knowledge-based methods use known information about basic components of the system and their interactions to analyse the system, the information is known through human interaction [3]. Model-based methods translate existing models of the system into FTs [4], and data-driven methods use databases as their source of information. We will use a data-driven method where failure data is used as the input to automatically obtain an FT structure.

2.3.2 Data-driven inference of Fault Trees

One of the first developments for automatic FT inference that is data driven were the LIFT algorithms [5]. The LIFT algorithms use a machine learning method to narrow down possible causal relationships between events. This method has problems with complexity of FTs, since it raises the computation time exponentially. Additionally the algorithm may learn causal relations that are only correlation relations in reality.

Another method was proposed that talks about creating FTs with Bayesian Networks [6]. A Bayesian Network is a probabilistic graphical model. This model can be learned from data and then be transformed into an FT. It needs exhaustive data to construct the BN with its root nodes, intermediate nodes and leaf nodes. These nodes are then translated into the BEs, Intermediate Events and TE of an FT, respectively. Intermediate Events are the combinations of BEs through Gates. This method is robust to noise, but the need for exhaustive knowledge of all parts of the system still makes real data difficult to work with, as especially intermediate parts may not be known.

One of the later developments for automatic FT inference is the use of evolutionary algorithms [7], with the latest development being the use of a multi-objective evolutionary algorithm [8]. These methods consist of the random modification of a set of parent FTs. These parent FTs are then tested with metrics, that determine which FTs can move on to the next generation and become the new parents. This evolution cycle continues inferring better FTs until a convergence criterion is met. The main problems with this solution is its scalability, and the fact that it cannot fully be used with noisy data. Noise in the data makes it not reliable to use the Minimal Cut Sets (MCSs), since there is no way to know whether the MCSs are correct. Without noise the MCSs greatly improve the optimization process of this approach.

The two problems of scalability and inability to use real (noisy) data occur frequently in the existing solutions. A case study on domestic heaters [16] talks about the problems with high runtimes and that real data can have problems due to its noisy, and sometimes incomplete nature. In their work, they have used a combination of the C4.5 decision-tree learning algorithm [17] and LIFT algorithms [5] to achieve their goals.

2.3.3 Reinforcement learning

Reinforcement learning can be used for many applications, from learning traffic signals [18] to learning the board game Go [19]. A recent example of RL being used is as a solution for specifications of the Clock Constraint Specification Language (CCSL) [11]. The authors use RL to bridge the knowledge gap of requirement engineers to quickly and accurately figure out CCSL specifications from natural language-based design descriptions. It does so by learning a Directed Acyclic Graph (DAG), and evaluating the reward based on the completed DAG. This is very similar to FTs, as they are also DAGs and can only be evaluated once they are completed.

The initial inspiration to use RL for FT inference came from a study that designs convolutional neural network architectures with regular Q-learning [9]. The authors present a Markov decision process that sequentially chooses the layers of the convolutional neural network. There are several different layer types that can be added at each step, with rewards being received equal to the validation accuracy. This way the learning agent is able to learn to create a convolutional neural network that outperforms previous methods and hand-crafted networks. A main restriction of this approach is that it limits the state-action space to achieve convergence. Extensions to deep learning techniques presented in [20] are suggested as future work.

The deep Q-network [20] is thoroughly explained in [21] which goes into the theory of deep Q-learning. This deep Q-learning has been used to generate directed acyclic graphs [22], where they use adjacency matrices as representations of the state. A big limitation of this work however is how they distributed the rewards. In a graph with 10 nodes, only one to five out of $1.018 * 10^{13}$ states would result in a non-zero reward, meaning they could not test on large graphs. The case of life-cycle control for large multi-component engineering systems [23] shows an RL algorithm that schedules maintenance and inspection policies for these systems, which have very large state-action spaces. This shows that deep Q-learning could be a good second step for inferring FTs with RL, as it could solve the scalabilty issue. However, it is not a step we will take in this research. We need to formulate FT inference as a RL problem first, which is the goal of this thesis.

A study in the field of quantum computing [24] and its follow-up study [25], use deep RL to automatically find architectures of quantum gates. The authors want to generate a particular quantum state with as few gates as possible, which is not a trivial task. This is similar to inference of FTs, which is also looking for a possible sequence of gates which is as small as possible but still generates the desired result. It is possible to discover a nearly-optimal gate sequence with this method. Here they stumble upon the scalability problem, where the sampling of the large amount of different paths requires significant computational resources.

3 Data-driven inference of Fault Tree models via Reinforcement Learning

Here we will present how we formulated the inference of fault tree models as a Reinforcement Learning problem, and how it was implemented and used to obtain the results of the following section.

3.1 Fault Tree inference in a Reinforcement Learning framework

As described in Section 2, a RL problem can be seen as a standard Markov decision process (MDP), that was previously defined in Definition 2.3. It contains a set of states S, a set of actions A, the transition probability function P and the reward function R. Here we present how we put these elements into the context of automatic FT Inference. First of all, our problem is deterministic, meaning that the transition probability function can be replaced by a function $S \times A \to S$. A state-action pair will always result in the same new state. The task of the learning agent will be to sequentially choose elements to add to an unfinished FT until it is finished.

State space: Each state is an unfinished FT. To define this properly, we take the definition of a FT, as described in Definition 2.1, and extend it to an Expandable Fault Tree (EFT). That extends the FT definition with one new set: Expansion Points (EP).

Definition 3.1 (Expandable Fault Tree) *An Expandable Fault Tree (EFT) is a 6-tuple EFT = ⟨BE, G, T, I, TE, EP⟩ where:*

- *BE, G, T, I and TE are as for fault trees (see Definition 2.1).*
- *EP is a set of* Expansion Points *which are dummy events similar to BEs.*
- *The set of all elements is given by $E := BE \cup G \cup EP$.*
- *Each gate has at most one EP as input: $|I(g) \cap EP| \leq 1$ for all $g \in G$. A gate g with no EP as input $I(g) \cap EP = \emptyset$ is called a* completed *gate.*

An EFT where all gates are completed is an FT. Expansion Points are the indication where the EFT can be expanded. Thus all Gs that are added will come with an EP by default. The TE will also come with a EP by default. Gates are completed when their EP is removed, meaning the EFT can no longer be expanded at that gate.

Action space: Action (A) is defined as a set of functions that maps an EFT to a new EFT ($EFT \to EFT$). Actions are done at an EP in the EFT. It is required to have a function that returns a gate G that has an EP. This function is F(EFT) $\longrightarrow$ G. F assures that the same G will be returned every time for the same state. This ensures that each specific state action pair will always give the same resulting EFT. The function F is also used to check for completion of a FT. Once no more EP can be found, F does not return a G. This indicates that the EFT is now a finished FT.

Now we show the actions A = ⟨{Add(E, EFT)|E ∈ AND,OR,BE}, Stop(EFT)⟩, where:

- Add(E, EFT): Adds element E to the gate found with F(EFT), where E ∈ AND,OR,BE.
- Stop(EFT): removes the EP of the EFT found with F(EFT), marking that gate G as completed.

Reward: The reward is calculated with three metrics: the accuracy based on the dataset (ϕ_d), the size of the FT (ϕ_s) and the accuracy based on the MCSs (ϕ_c).

To calculate the accuracy based on the dataset ϕ_d, we use function P^{FD} that gives the value of the TE for each set of BEs in the N datapoints in the failure dataset. We compare each value of P^{FD} (P_i^{FD}) to each value of the P_i^{FT}, the P function of the inferred FT. Each i corresponds to one of the BE combinations of P.

$$\phi_d = \frac{\sum_i^N x_i}{N}, \text{ where } \begin{cases} x_i = 1 & \text{if } P_i^{FD} = P_i^{FT} \\ x_i = 0 & \text{if } P_i^{FD} \neq P_i^{FT} \end{cases} \tag{2}$$

The ϕ_d metric has values [0, 1], where 1 is most desirable.

To calculate the size metric ϕ_s, we add the number of elements of the FT together, which are the number of Gates plus the number of BEs, and then divide them by a maximum size (S_{max}) that the FT may have. The exact value of S_{max} is explained later in Section 3.4.4. We divide by this maximum size so FTs will be punished for their size proportionally to how big they should be. Instead of punishing them for their absolute size.

$$\phi_s = \frac{|E|}{S_{max}} \tag{3}$$

This technically also means that ϕ_s has values [0, 1], but since the S_{max} is a soft cap, it could have values larger than 1. The value of 0 is obviously never reached, but it is the most desirable.

The exact same method as in FT-MOEA [8] is used to calculate the MCS accuracy. In [8], the RV-coefficient [26] is used, which measures the similarity between the MCS matrix of the failure dataset (M_D) and the MCS matrix of a given FT (M_F).

$$\phi_c = \frac{tr(M_D M_F^T M_F M_D^T)}{\sqrt{tr(M_D M_D^T)^2 tr(M_F M_F^T)^2}} \tag{4}$$

Where tr(.) is the *trace* of the matrix. While FT-MOEA uses the MCS error, we use the accuracy, which is simply not subtracting the result from 1. The ϕ_c metric has values [0, 1], where 1 is most desirable.

With these metrics, we can calculate the reward:

$$r = p_d\phi_d - p_s\phi_s + p_c\phi_c \tag{5}$$

where each p is a parameter that gives weight to the metrics. The user can give these as numbers at the beginning of the algorithm. (We will study what the best values of these are later in Section 5.1.) Note the minus sign in front of the p_s. This is the size metric. It is desired to have smaller FTs, hence less reward will be given for larger FTs. Note that these metrics can only be calculated for a finished FT. Therefore there will be no intermediate rewards.

Policy: The policy is based on Q-learning. Each encountered state-action pair receives a Q-value. The policy is then to always take the action in the current state that has the highest Q-value. In this sense the policy never directly changes, but the Q-values change, causing different actions to be taken.

Obtaining the best FT: Each FT that is inferred gets evaluated. To get the best FT out of this process, the FT with the highest reward is saved. This saved FT is then compared to each new FT in the process, keeping the FT that is the best. At the end of a pre-set amount of runs, or when no improvements have been found in the last pre-set amount of cycles, this best FT is presented as the result of the run. There are actually three different best FTs being tracked, one with the best reward, one with the best accuracy based on the dataset and one with the best accuracy based on the MCSs. This is done so a user can see and choose wether they prefer a more accurate FT or a smaller FT. In the ideal case these three saved FTs would be the exact same FT.

Formulation as MDP: Now we can formulate the MDP from Definition 2.3 with the definitions we gave. $M = \langle S, A, P, R, \gamma \rangle$. Where:

- S - a set of EFT
- A - a set of Actions A = $\langle$\{Add(E, EFT)|E $\in$ AND,OR,BE\}, Stop(EFT)$\rangle$
- P - the transition probability function, which is always 1.
- R - the reward function $r = p_d\phi_d - p_s\phi_s + p_c\phi_c$
- γ - the discount factor, which limits the importance of future rewards.

3.2 Example of FT inference

Here we will show an example of how an FT is inferred. The example is shown in Figure 3. We start with a completely empty FT. Then we pick one of the two gates to start with: AND or OR. Figure 3a shows the start with an OR gate. Note the EP in yellow below the gate. This is the expansion point. BE1 and an AND gate are added to the EFT with Add actions. Then, from 3c to 3d, the OR gate is completed with the Stop action. The EP is removed. The algorithm then keeps adding elements until the AND gate is complete as well and an inferred FT is obtained.

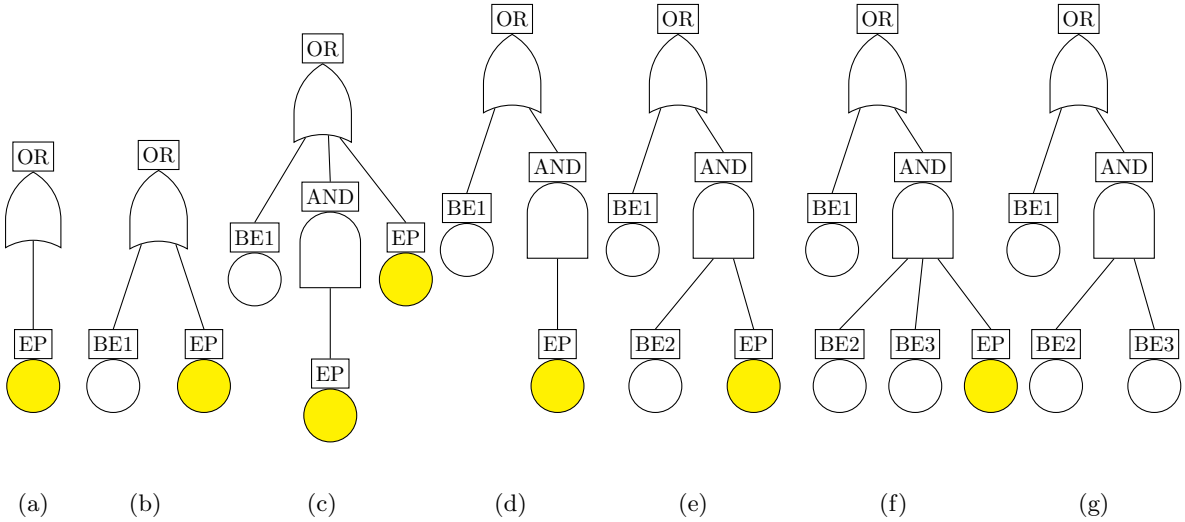


Figure 3: An example of an FT being inferred though EFTs and taking actions.

3.3 Basic Algorithm

Now that we know how to formulate FT inference as a RL problem, we can present the algorithm used to generate the results.

The algorithm infers many FTs through a couple of *cycles*. Each cycle has an amount of *rounds*, where each round corresponds to one inferred FT. After an FT is inferred, it is evaluated and compared to the best previously found FT. Eventually the algorithm will stop and return the best FT it found throughout its cycles.

A basic overview of the algorithm can be seen in Figure 4. At Step 1 the process is initialized. The algorithm needs the failure data and parameters. These parameters are the weights of the reward function, the amount of rounds per cycle, the amount of cycles, and the convergence criteria. At Step 2 a new cycle is started, which leads into Step 3 where a new FT is started. The process of inferring an FT is as follows: begin with a TE, add one of the possible set of elements to the TE, keep adding elements until the FT is finished. (Step 4) After each addition of an element, the Q-values are updated. (Step 5) Once the FT is finished, it can be evaluated in Step 6 and saved in Step 7. This process of Step 3 through 7 is one round.

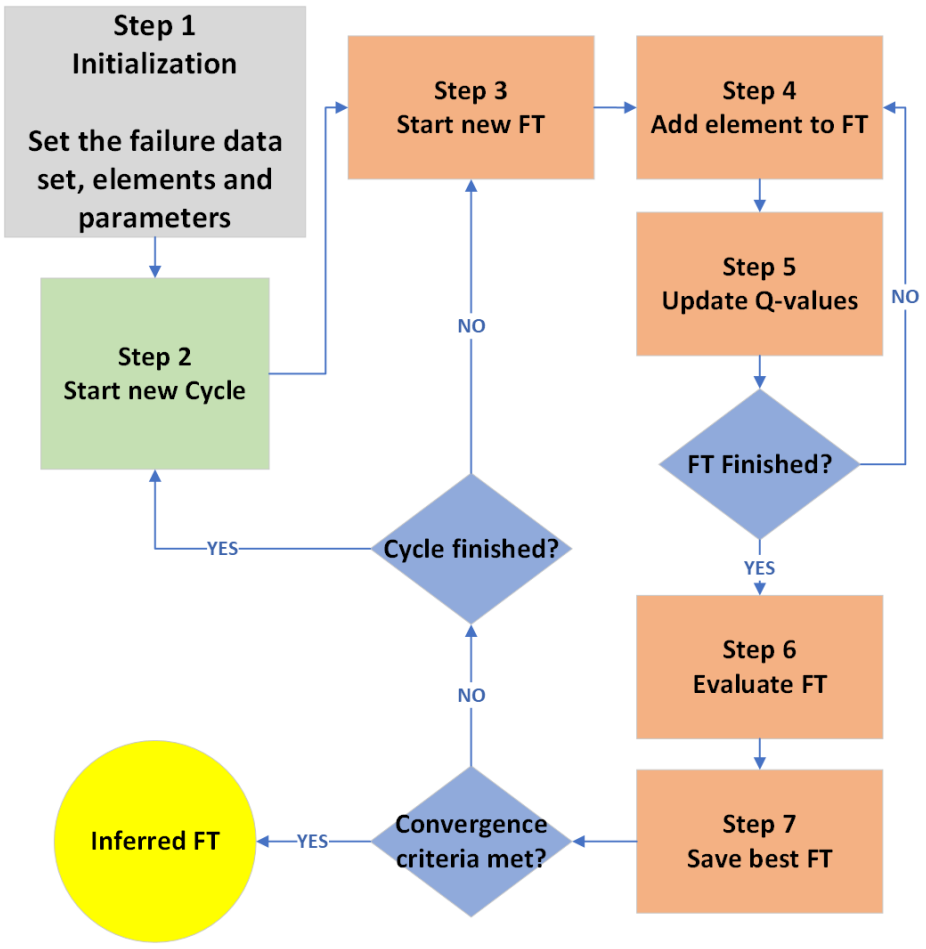


Figure 4: Schematic overview of the algorithm process.

We now show pseudo-code for the most important parts of the algorithm. Algorithm 1 shows the main loop, where the cycles are done. The weights of the reward function (p_d, p_s and p_c), the amount of cycles, rounds per cycle and of course the failure data have to be given as input. When this algorithm is done, the final inferred FT is obtained. A new FT is created for each round at line 5, it is then evaluated at line 6 to obtain the results. The Q-values are updated and the results are stored. The storing

of the results also compares it to the previous best results so the best FT can be kept. The finishing criteria are checked at line 9. If the criteria are met, the loops end and the final results are returned.

Algorithm 1 FT-RL pseudo code

Input failure data, parameters: p_d, p_s, p_c, rounds, cycles
Output final results: best FT, metrics

```
1:  q_values = Q(p_d, p_s, p_c, data)
2:  elements = get_elements(data)
3:  for cycles : do
4:      for rounds : do
5:          ft = create_new_ft(cycle, round, elements, q_values)
6:          results = evaluate_ft(ft)
7:          update_final_q_value()
8:          store_results(results)
9:          check_finishing_criteria()
10:     end for
11: end for
12: return final_results
```

The FT structure is inferred with $create\_new\_ft()$, which is shown in Algorithm 2. This covers steps 4 and 5 of Figure 4. First, it initializes a new EFT in line 2. Then a set of actions is created based on the elements that are provided. ϵ is set in line 4, this is needed for the exploration and exploitation phases of the RL. When exploring, ϵ is set to zero. When exploiting, ϵ is gradually raised to one. The round input is used to determine the phase. It needs to know the current cycle, for the locking in optimization described later in Section 3.4.2.

Starting at line 5, actions are taken to add elements to the EFT. Line 6 is the function described in Section 3.1 that locates the first EP of the EFT. It returns the expandable element. If no expandable element is found, the FT is done and returned. If there is an expandable element, line 10 makes a list of possible actions on that element. Then, line 11 checks if a random or guided action needs to be taken. If a guided action is taken, the Q-values are used and the best action to take is obtained. If a random action is taken, one is chosen at random from the list of possible actions. After that line 16 does the action that has been selected. Line 17 updates the Q-values according to the action taken. Actions continue to be taken until the FT is completed, or a maximum amount of iterations is reached.

Algorithm 2 Creation of FT

Input round, cycle, elements, q_values
Output FT

```
1: procedure CREATE_NEW_FT(:)
2:     ft = FaultTree()
3:     actions = create_set_of_actions(elements)
4:     ϵ = set_epsilon(round, cycle)
5:     for max_iterations do
6:         el = ft.get_expandable()
7:         if el is None then
8:             return ft
9:         else
10:            possible_actions = get_possible_actions(el, actions)
11:            if random.random() > ϵ then
12:                ac = q_values.get_best_action(ft, possible_actions)
13:            else
14:                ac = get_random_action(ft, possible_actions)
15:            end if
16:            ac.do_action(ft, element)
17:            q_values.update_q_values()
18:        end if
19:    end for
20:    return ft
21: end procedure
```

3.4 Optimizations

To help the algorithm find good FTs faster, several optimizations and additions have been added to the basic algorithm. In this section we will discuss the details on the added Backpropagation, Locking in of choices, Maximum size and Semi-Randomization.

3.4.1 Backpropagation

The first problem that was encountered with the basic algorithm was the fact that *many* states did not receive any updates to their q-values. Out of several million states, less than one percent actually had their q-values updated. The states that had updated q-values were mainly the states that finished the FT, with their action being the stop action. This is because of two factors: the fact that there are no intermediate rewards (rewards are only given for completed FT), and the fact that the state space is very large. This means that in order for rewards to propagate through the states, the same states have to be reached again by the algorithm randomly, so the value function can see the previous rewards. This is less likely the larger the state space is. Other RL problems with intermediate rewards do not have this issue.

To tackle this problem we introduce backpropagation of the reward. Because the process is deterministic, we can save the full state-action sequence that has been done to reach a certain reward. This state-action sequence $(s_0, a_0), (s_1, a_1) \dots (a_n, s_n)$ has $n+1$ state-action pairs, each with a corresponding Q-value Q_i. The last state action pair before the completion of the FT will have $n = 0$, this state action pair will receive the full reward. Each state action pair after that will receive the reward with an extra power of the learning rate and diminishing factor. The Q-values for each of the pairs is then updated as follows:

$$Q_n = Q_n + \alpha^{n+1}\gamma^n r \tag{6}$$

where α, γ and r are the learning rate, diminishing factor and reward, respectively. This formula mimics the behavior of all states seeing the final reward with the value-function part from Equation 1. This backpropagation enforces the final reward to be seen by all intermediate state-action pairs. Many Q-values are updated this way. Each inferred FT contributes to finding the best FT. The backpropagation is implemented in FT-RL by altering Algorithm 1 line 7. Here not only the final Q-value is updated, but also backpropagated.

To demonstrate the backpropagation visually, we look at an example of an FT being inferred. The example was shown earlier in Figure 3, but is now displayed again here in Figure 5 for easy reference.

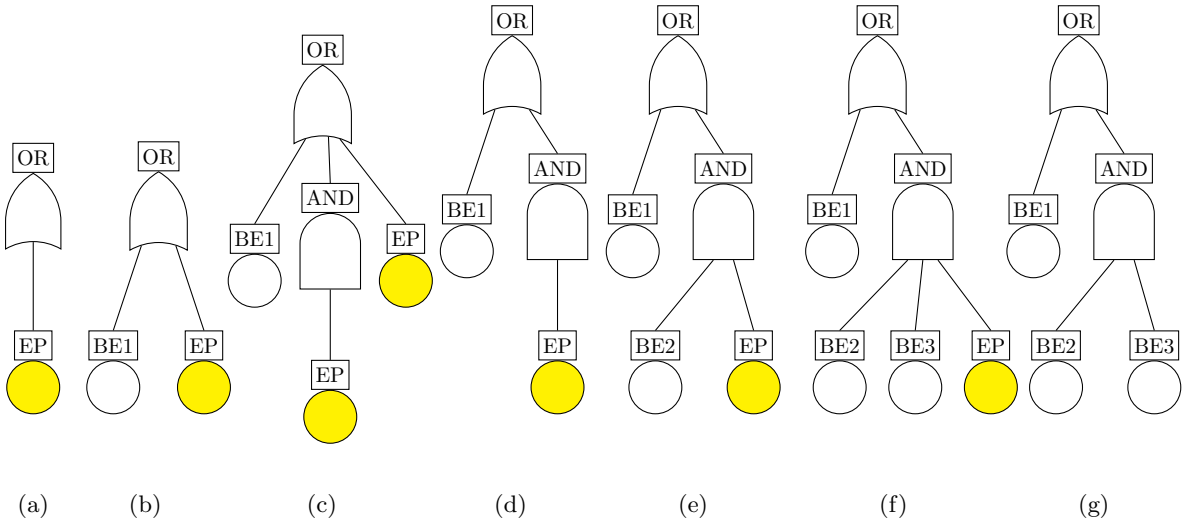


Figure 5: An example of an FT being inferred though EFTs and taking actions.

Table 2 shows the state action pairs of the example, with the Q-values. In the beginning, all Q-values are set to zero. When actions are taken, the Q-values are updated based on Eq. 1. This means that at the start, all updates to the Q-values result in 0. The actions taken in the example are highlighted in pink. When the last stop action is taken, the FT can be evaluated and a reward calculated. This reward is then used to update all the Q-values for the state action pairs that led to this reward, as is shown in the second part of Table 2.

Table 2: Shows the Q-values for each state action pair of the example in Figure 5. The Actions highlighted in pink are the ones that have been taken to get to the next state. The Q-values in green after the backpropagation show that the reward has been backpropagated through the state action pairs.

Empty state		State a		State b		State c		State d	
Action	Q	Action	Q	Action	Q	Action	Q	Action	Q
Add(AND)	0	Add(AND)	0	Add(AND)	0	Add(AND)	0	Add(AND)	0
Add(OR)	0	Add(OR)	0	Add(OR)	0	Add(OR)	0	Add(OR)	0
		Add(BE1)	0	Add(BE1)	0	Add(BE1)	0	Add(BE1)	0
		Add(BE2)	0	Add(BE2)	0	Add(BE2)	0	Add(BE2)	0
		Add(BE3)	0	Add(BE3)	0	Add(BE3)	0	Add(BE3)	0
						Stop	0		

State e		State f		FT g	
Action	Q	Action	Q	Evaluation	Value
Add(AND)	0	Add(AND)	0	Reward	12
Add(OR)	0	Add(OR)	0	Accuracy	0.8
Add(BE1)	0	Add(BE1)	0	MCS	0.7
Add(BE2)	0	Add(BE2)	0	Size	5
Add(BE3)	0	Add(BE3)	0		
		Stop	0		

Backprop:

Empty state		State a		State b		State c		State d	
Action	Q	Action	Q	Action	Q	Action	Q	Action	Q
Add(AND)	0	Add(AND)	0	Add(AND)	6.6	Add(AND)	0	Add(AND)	0
Add(OR)	5.3	Add(OR)	0	Add(OR)	0	Add(OR)	0	Add(OR)	0
		Add(BE1)	5.9	Add(BE1)	0	Add(BE1)	0	Add(BE1)	0
		Add(BE2)	0	Add(BE2)	0	Add(BE2)	0	Add(BE2)	8.1
		Add(BE3)	0	Add(BE3)	0	Add(BE3)	0	Add(BE3)	0
						Stop	7.3		

State e		State f		FT g	
Action	Q	Action	Q	Evaluation	Value
Add(AND)	0	Add(AND)	0	Reward	12
Add(OR)	0	Add(OR)	0	Accuracy	0.8
Add(BE1)	0	Add(BE1)	0	MCS	0.7
Add(BE2)	0	Add(BE2)	0	Size	5
Add(BE3)	9	Add(BE3)	0		
		Stop	10		

3.4.2 Locking in choices

Because of the large state spaces that can occur in this field, it is necessary to explore in a more clever manner than purely random. A survey on exploration methods [27], shows there are many ways to improve the exploration in RL. We use a random-based method, where we alter the randomness of the exploration. One of these alterations is the *locking in* of choices. (Further alterations are discussed in Section **??**.) The cycles in the algorithm are used for the "locking in" of choices. In the first cycle, the agent has no knowledge. It will explore purely random, and then exploit. After the first cycle it will have a decent idea of what choices to make to get a good FT. The next cycles will make use of this knowledge. In a new cycle, we give the very first choice the agent makes a very large likelihood to be *exploiting*, instead of exploring. After that choice the normal exploration will continue. The Nth cycle will have its first N choices being "locked in". The idea here is that the further exploration cycles will explore closer to the desired FT. Algorithm 3 shows how this is implemented. It is used as a substitution for the *random*() call in line 11 of Algorithm 2.

This approach was tested on the COVID case study (see Section 4.1) and Figure 6 shows the results. The average ϕ_d metric of 20 runs is shown per round. The exploring then exploiting pattern of the

Algorithm 3 Locking in

```
1: procedure LOCK_IN_CHOICE:
2:     if exploration_phase < 0: then
3:         enforce_epsilon -= 1/exploration_phase
4:         max(enforce_epsilon, 0)
5:     else
6:         enforce_epsilon = 0
7:     end if
8:     if lock_in: then
9:         return random() > enforce_epsilon + epsilon
10:    end if
11:    return random() > epsilon
12: end procedure
```

cycles causes the saw-like shape. After each peak a cycle ends. It can clearly be seen that the "lock in" functionality causes the agent to explore better FTs on average. While it does not clearly reach higher average ϕ_d for the total run, we can see that it reaches a higher average ϕ_d faster.

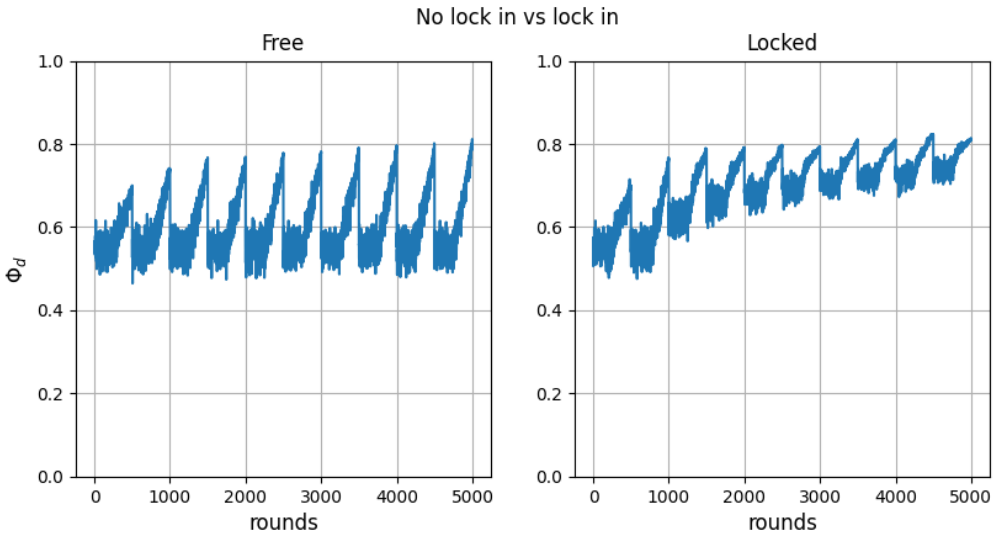


Figure 6: Average accuracy metric ϕ_d of 20 runs with 10 cycles with each 500 Rounds. With free exploration on the left and "locked in" exploration on the right. The experiment was done with the COVID case study, shown in Table 3 in Section 5.

3.4.3 Avoiding redundant states

Two measures have been taken to prevent certain redundant structures in the FT. First of all, if a gate already contains a BE as its child, it is not possible to add the same BE as a direct child a second time. This is present in Algorithm 2 line 10. The possible actions are adjusted to remove any BE additions that are already present as children. Secondly, the children of a gate are always ordered. This way the order of insertion does not matter. If BE2 is added before BE1 to a gate for example, as the state will always see them as BE1 and BE2, in that order.

3.4.4 Max size

Theoretically, the state space of inferring an FT with even as little as three elements (one gate and two basic events), can already be infinite. A gate can have a gate as its child, which can also have a gate as its child, and this can continue indefinitely. In order to limit the state space, we introduce the idea of a maximum size of the FT. From the MCS, we can determine the disjunctive normal form [14]. This is a FT built from just the MCS. This will contain all the information necessary, but will often

be larger than needed. Thus the size of this form will be the maximum size of a randomly inferred FT as anything larger will definitely be redundant information. The maximum size is calculated by:

$$S_{max} = \text{Number of MCS} + \sum_{m \text{ in MCS}} |m| + 1 \text{ , where } |m| \text{ is the number of BEs in a MCS } m \tag{7}$$

This maximum size is used to keep the value of the size metric ϕ_s between [0, 1]. This makes finding a good weight parameter p_s (Section 5.1) for the reward function (Eq. 5) easier.

3.4.5 Semi-random generation

We do not have zero information when we start exploring the state space. As already mentioned before, a maximum size of the desired FT is already known. We can use this fact to guide the exploration. During early development, we noticed that the algorithm tended to produce FTs that were very small. This can be due to many factors. In a first version, every element has an equal chance of being added to the FT. This caused the structure of the FT to be dependent on the ratio between the amount of BEs with the amount of Gs. After two elements were added to a gate, we put the stopping action at 50%, since putting it at an equal chance of the rest would mean the chance of stopping a gate is quite low, and the FTs would grow very large.

This called for a more sophisticated method of random generation. We landed on the following: we distinguish expanding and finishing actions:

- Expanding actions: addition of a gate.
- Finishing actions: addition of a BE or the Stop action.

Then we set a size difference parameter s_d, which is simply the current size of the EFT (EFT_s) divided by the max size (S_{max}).

$$s_d = \frac{\text{EFT}_s}{S_{max}} \tag{8}$$

As this size difference parameter gets closer to 1, the finishing actions are prioritized more. When the size difference parameter is close to 0, the expanding actions are prioritized more. This way FTs of sizes up to the maximum size are being explored. This is desirable since the desired size is smaller than the maximum size. One pitfall of this approach is that the max size can get really large when the failure dataset contains many MCSs. Since this semi-random generation guides the FTs to a size that is around the max size, it will generate very large FTs, which takes a long time. As a precaution the max size of an FT can reach no higher than 100. This restriction is suitable for the case studies done in this research, but can be easily altered. This semi-random generation is part of FT-RL in Algorithm 2. Line 14 is replaced with this semi-random action selection.

4 Experimental setup

In this section we show the experimental setup. Most of the experiments were done with five case studies, described in Section 4.1. The failure data of these case studies was used as input into the algorithm implementation. We call this implementation of the data driven Fault Tree inference with reinforcement learning: FT-RL. Each run of FT-RL results in an inferred FT. The metrics of this FT is then used as part of the results in the following Sections 5 and 6. The experiments were run on a Dell R7515 server, with an AMD 7302P CPU (16/32 cores), 64G RAM and on a 240GB SSD.

4.1 Case studies

We test the performance of the FT-RL algorithm based on five case studies. These case studies are the same ones used in previous research FT-MOEA [8], so comparison between FT-RL and FT-MOEA can be done easily. The case studies, shown in Table 3, are: Container Seal Design (CSD); Pressure Tank (PT); COVID-19 infection risk (COVID); Mono-propellant propulsion system (MPPS); Spread Monitoring System (SMS).

Table 3: Case studies with their relevant information. The number of unique BEs (w), total BEs (W), AND gates (#AND), OR gates (#OR), FT size (FT_{size}), Minimal Cut Sets (#MCS), order of MCSs (O-MCSs), and space complexity (O(2^w)).

Case study	Ref.	w	W	#AND	#OR	FT_{size}	#MCSs	O-MCSs	O(2^w)
CSD	[13]	6	6	2	2	10	3	{2,3,3}	64
PT	[13]	6	6	1	4	11	5	{1,1,2,2,2}	64
COVID19	[28]	9	21	9	3	33	6	{3,4,4,4,4,4}	512
MPPS	[13]	8	12	3	8	23	7	{2,2,2,2,2,2,2}	256
SMS	[29]	13	17	0	8	25	13	{1,1,1,1,1,1,1,1,1,1,1,1,1}	8192

4.2 FT-RL implementation

The FT-RL algorithm is implemented in Python. It uses several libraries such as numpy, scipy and sympy. Since the algorithm takes in failure data (see Section 3.3 for details), and not an FT, the failure data is first generated. The failure data is generated by collecting all possible BE combinations from the FTs and propagating them through the FT to obtain the TE. Each count is set to 1. The failure data is complete and noise-free. This failure data is then given to the algorithm as input. The algorithm keeps track of the best FT it found throughout its runtime, and then saves the best FT and its metrics at the end.

5 Hyper-parameters tuning

In this section we investigate the hyper-parameters of FT-RL. The algorithm has many input parameters: weights for the reward metrics, amount of rounds and amount of cycles as the most important ones. To find optimal hyper-parameters, we will carry out many runs of FT-RL, while varying these parameters independently. The metrics of the resulting FTs will give a clear indication of which values for the parameters are desirable. In this Section, we will answer the specific research question R2.

In Section 5.1 we want to find out what the best prioritization of the reward metrics is. There will always be a trade-off between the size and the accuracy of the resulting FT. By varying the weights of the reward function, we can find the best prioritization.

In Section 5.2 we evaluate the amount of rounds per cycle. This influences the amount of FTs that are generated in each exploration and exploitation phase. There is a trade-off between completion time and the other metrics. We want to find out what the best length is, by varying the amount of rounds per cycle.

In Section 5.3 we investigate the ending criteria. We want to find out how many cycles we should give FT-RL to find better FTs. If FT-RL does not find a new, better FT within X amount of cycles, the run will be stopped. We vary the amount of X to find out the best trade-off between completion time and the other metrics.

5.1 Parametric analysis on reward function (p_d, p_s, p_c)

As stated earlier in Section 3.1, the reward is calculated according to Eq. 5. There are three parameters p_d, p_s and p_c that serve as weights for the metrics ϕ_d, ϕ_s and ϕ_c, respectively.

To test the influence of these parameters, for each of the five case studies we set p_d, p_s, and p_c to 0.5. We then vary one of the three parameters between 0 and 1 in steps of 0.2. Note that the *difference* between the values of these parameters has an influence on the outcome of the algorithm, but not their absolute values. Setting all parameters to 1 will give the same result as setting them all to 0.1.

The algorithm is fed with the failure data of the case studies, and has run for 5 cycles with 1000 rounds in each cycle. (The optimal amount of rounds and cycles will be explored later, these numbers were chosen so the experiments would not take too much time while still giving decent accuracy results.) After the 5 cycles the algorithm stops and gives the best FT it could infer in that time. Figure 7 shows the varying of parameter p_d, Figure 9 the varying of p_s, and Figure 8 the varying of both p_d and p_c.

Figure interpretation: Result Figures always show four rows of plots. The first row depicts the reward r or time in minutes, the second one the accuracy metric based on data ϕ_d, the third row the size metric ϕ_s and the last row the accuracy metric based on the MCSs ϕ_c. The columns (separated by dotted green lines) correspond to the different values of the paramter on the x-axis. The results are shown as boxplots, which is the average of 20 runs each. The boxes show the standard deviation, and the circles show outliers. The different case studies are shown in different colors: CSD in blue, PT in green, COVID in purple, MPPS in pink and SMS in red. They are also abbreviated and labelled on the top plot: CS (CSD), PT, CO (COVID), MP (MPPS) and SM (SMS).

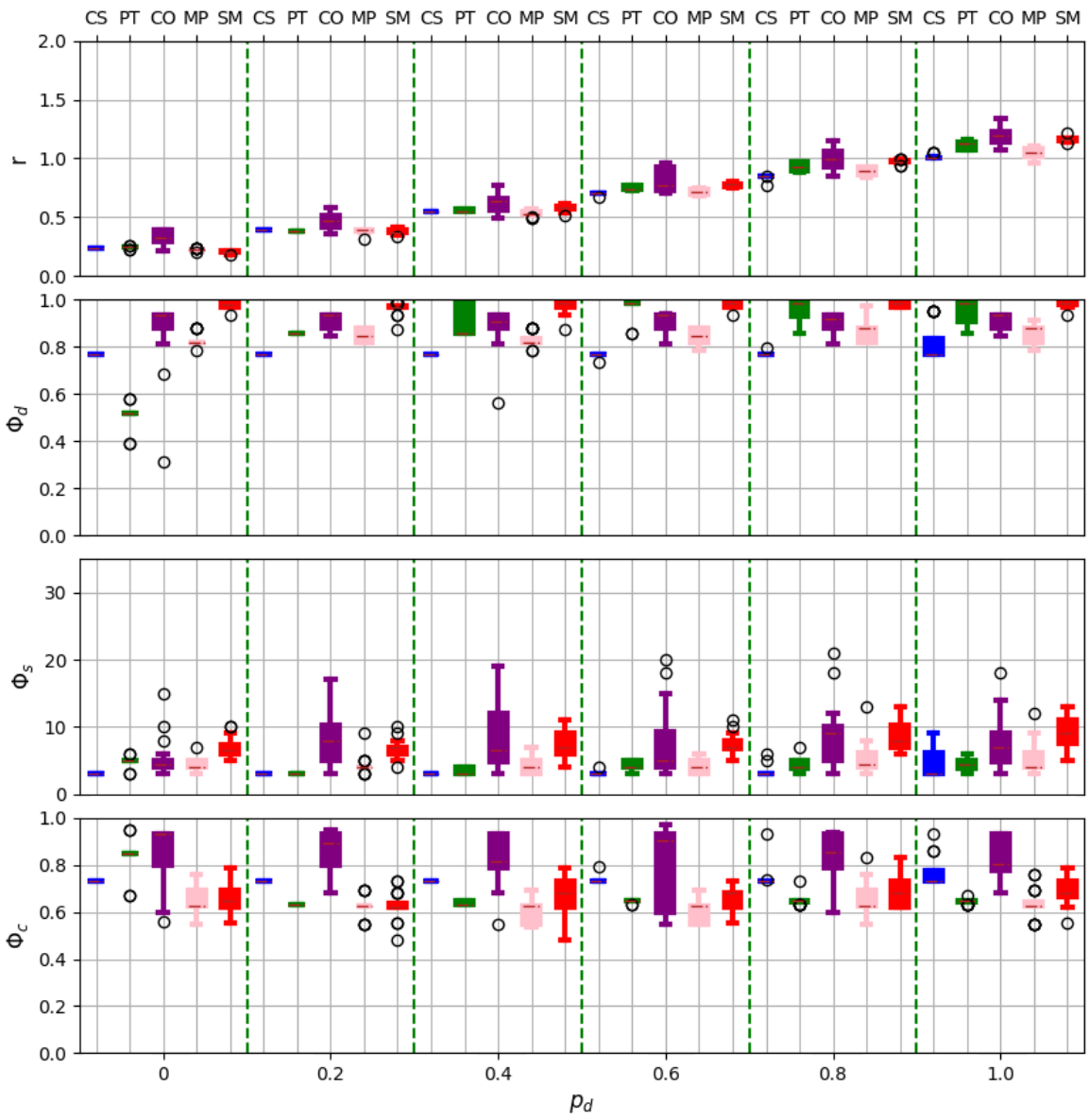


Figure 7: Results for varying values of weight p_d (accuracy based on the dataset ϕ_d). The other two parameters p_s and p_c are set to 0.5. p_d is varied between 0 and 1 in steps of 0.2, the dotted green lines indicate these steps.

Since the metrics ϕ_d and ϕ_c are indirectly linked, it is interesting to vary them together. This is shown in Figure 8. Note here that when p_d and p_c are 0, all metrics are zero. This is the result of a limitation of the algorithm, which works on maximizing rewards. When p_d and p_c are 0, it will never gain any positive rewards and thus cannot perform its job.

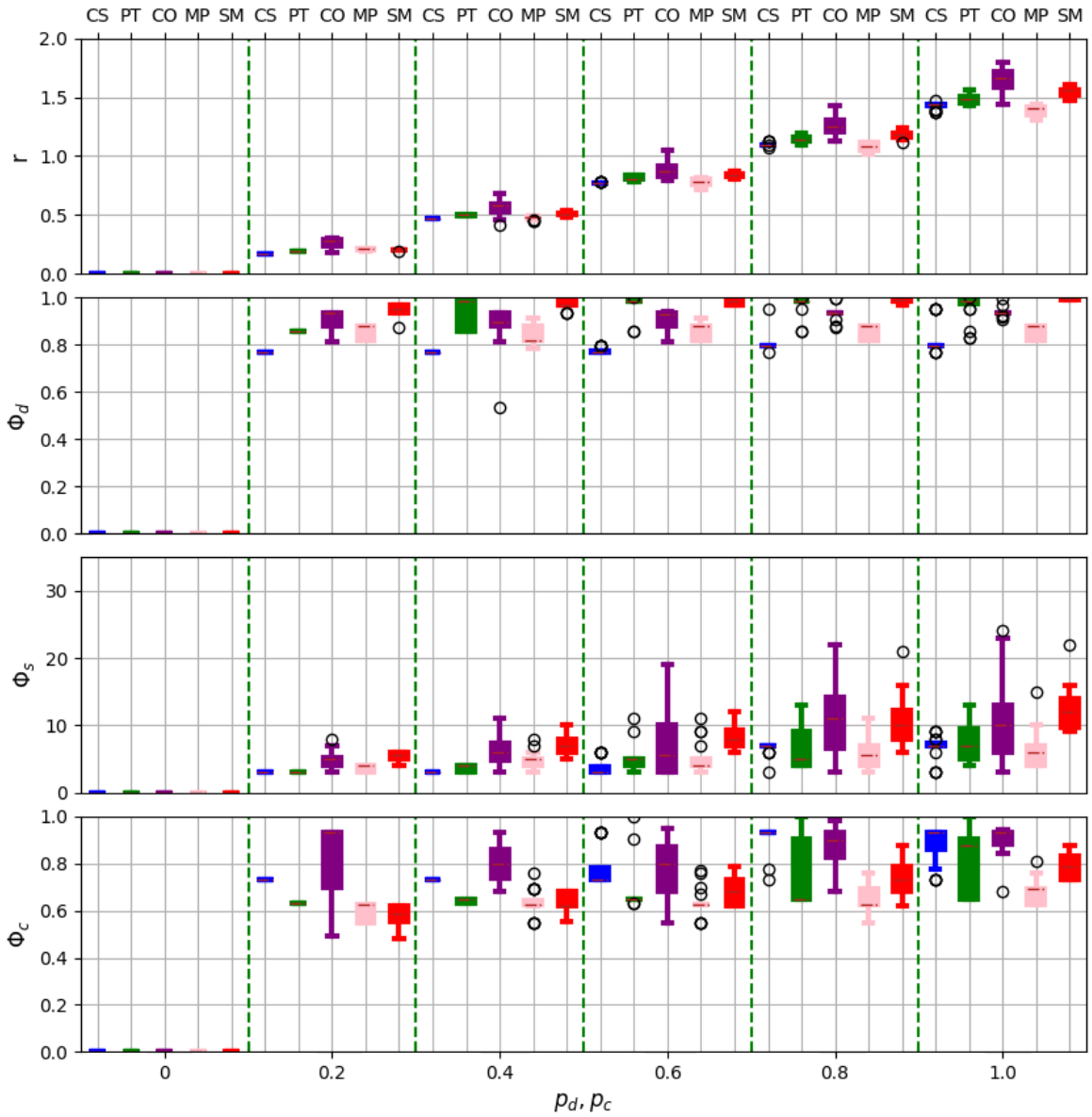


Figure 8: Result of the parametric analysis. Here the parameter p_d for the accuracy based on the dataset ϕ_d and the parameter p_c for the accuracy based on the MCSs ϕ_c is varied. The p_s parameter is set to 0.5. p_d and p_c are varied together between 0 and 1 in steps of 0.2, the dotted green lines indicate these steps.

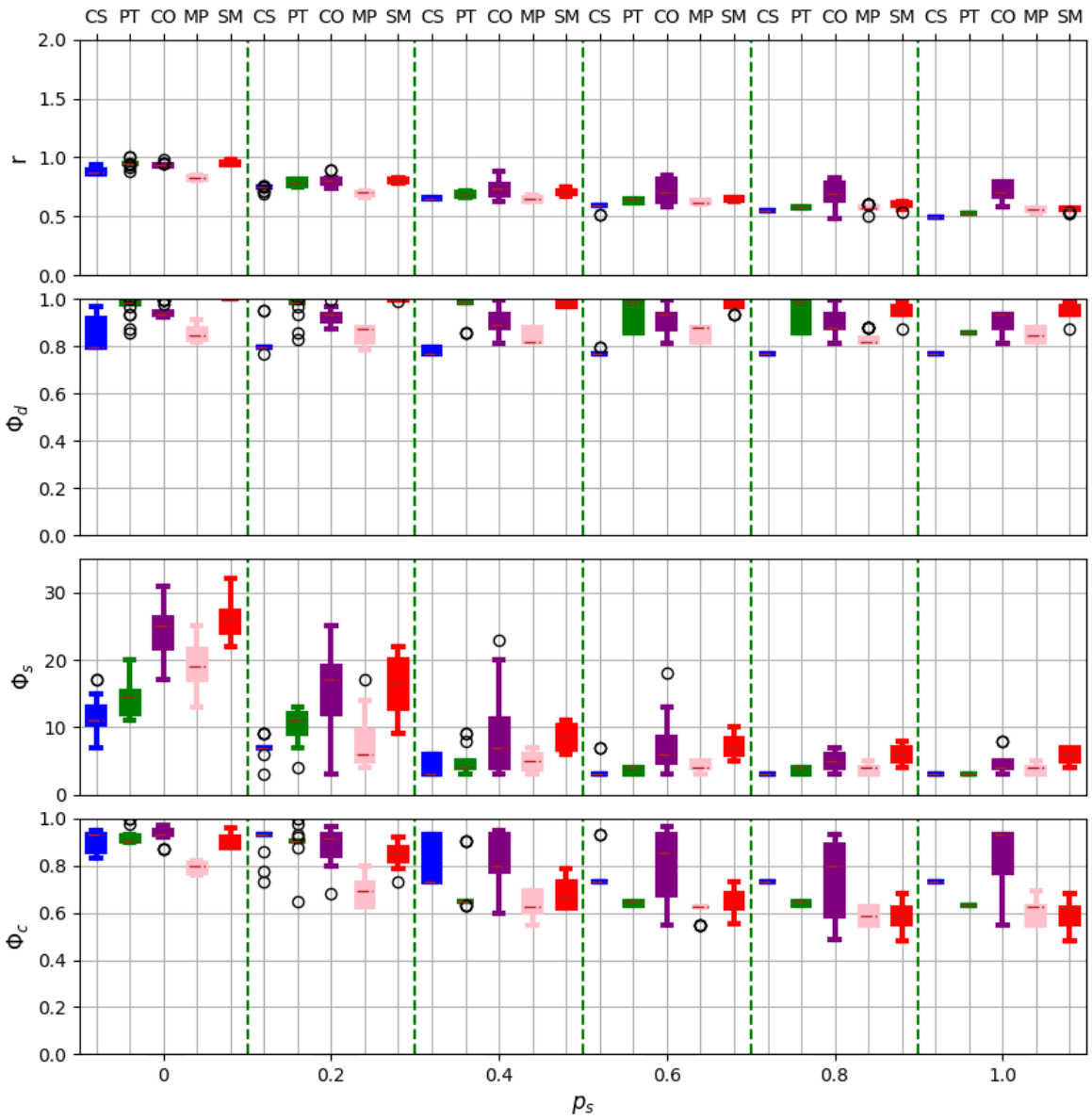


Figure 9: Result of the parametric analysis. Here the parameter p_s for the accuracy based on the dataset ϕ_d is varied. The other two parameters p_d and p_c are set to 0.5. p_s is varied between 0 and 1 in steps of 0.2, the dotted green lines indicate these steps.

Now that we can see the results, we can discuss which values for the parameters are the best. Figure 7 shows the varying p_d. The reward naturally steadily goes up, as the reward for a high ϕ_d simply increases with p_d. Only the PT case clearly responds positively to a higher p_d. The rest of the cases do not change much in this metric. This could be considered strange but keep in mind that the ϕ_d and ϕ_c are somewhat linked. They are both accuracy metrics ultimately based on the failure data, they simply look at different aspects of it. Keeping ϕ_c steadily at 0.5 also appears to makes sure that ϕ_d has a high resulting value.

Figure 8 shows both the accuracy metrics being changed at the same time. While ϕ_c increases for all cases with increasing p_d and p_c, ϕ_d does once again generally not show much change. The fact that the accuracy metrics do not go to 100% is disappointing. We will investigate why this is further in Sections 6.2 through 6.4. The size ϕ_s gets larger with higher accuracy weights.

Figure 9 shows the varying of parameter p_s for the size. When it increases, the size and reward go down, as is expected. The ϕ_d shows that it goes down as well, but not as clearly as ϕ_c. Since we care about all three of the metrics, but more about the accuracy, we will take the values of 0.5, 0.2 and

0.5 for the parameters p_d, p_s and p_c respectively. Setting ϕ_s to 0.2 gives a great drop in size, while generally maintaining the high accuracy metrics.

5.2 Amount of rounds per cycle

Each cycle has an amount of rounds. This amount is another input for the algorithm, so it can also be varied. In order to determine a good amount of rounds, we ran the program for 5 cycles, with a varying amount of rounds. The parameters for the reward function are set to 0.5, 0.2 and 0.5 as found by the previous experiment. The results can be seen in Figure 10.

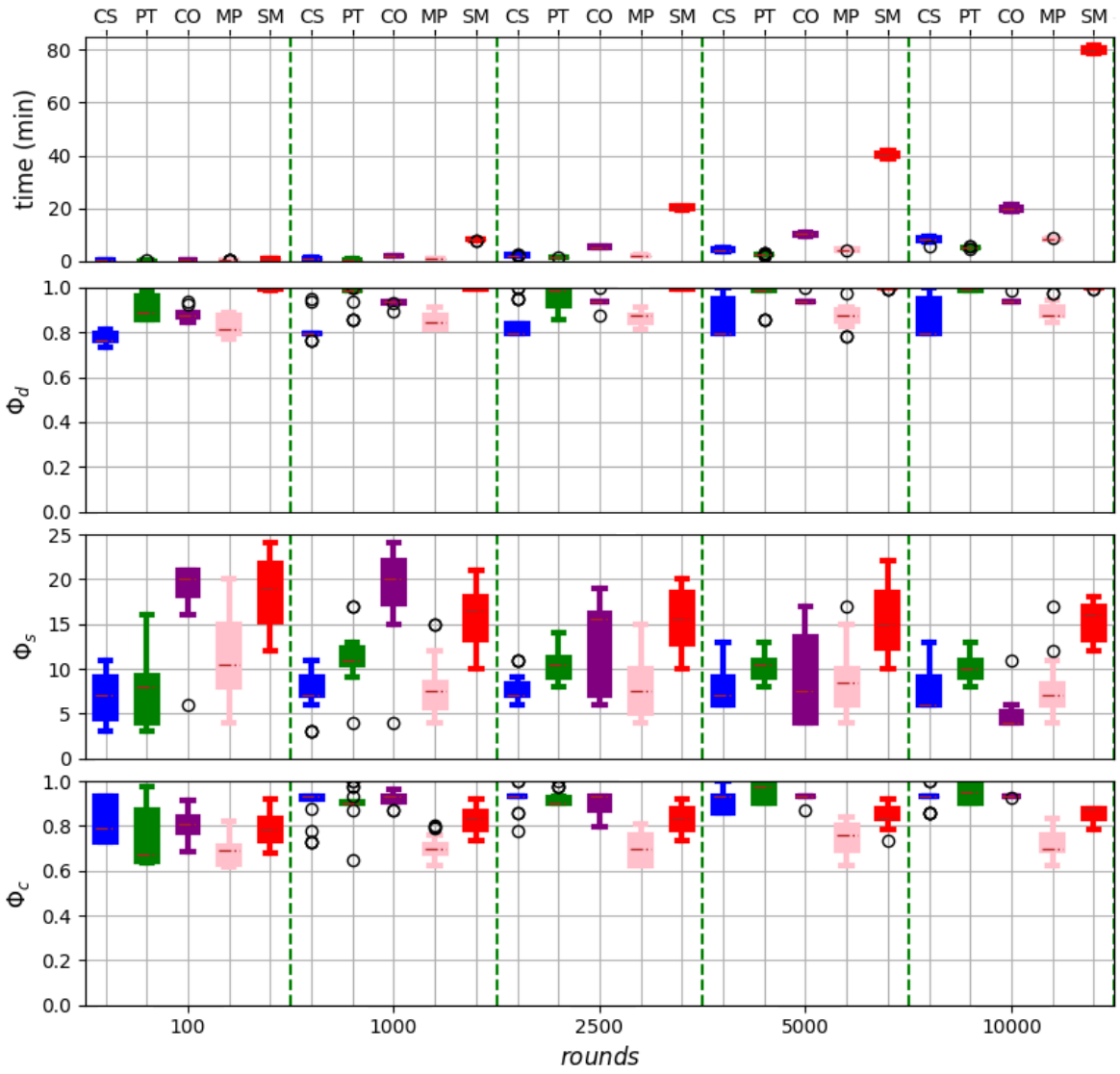


Figure 10: Result of rounds analysis. The time, ϕ_d, ϕ_s and ϕ_c metrics are shown.

Naturally the results are improving with more rounds per cycle. More rounds means more time for the algorithm to find good FTs. In general though, there is no clear trend for the metrics. The improvements on the accuracy metrics seem to stop at about 2500 rounds. Increasing the amount of rounds past 2500 does not significantly help the accuracy. The sizes also generally do not improve much after the 2500 rounds. Only the COVID case seems to benefit a lot from more rounds for its size metric. The time to completion however, does clearly increase with each increase in rounds. We will set the rounds to 2500 for following experiments, as the trade-off between the metrics and the time of completion. It does become more apparent that there is no perfect setup of the parameters that will cover every case optimally.

5.3 Ending criteria

In order to obtain results for the completion time of the algorithm, we need an ending criteria. The ending criteria consists of either finding an FT that has 100% accuracy, or the algorithm not being able to find a better result in the last X amount of cycles. The next experiment is about finding which value for X is a good one. The reward function parameters are set to 0.5, 0.2 and 0.5. The rounds per cycle are set to 2500. We have varied X for 2, 5, 10 and 20 cycles. The results can be seen in Figure 11.

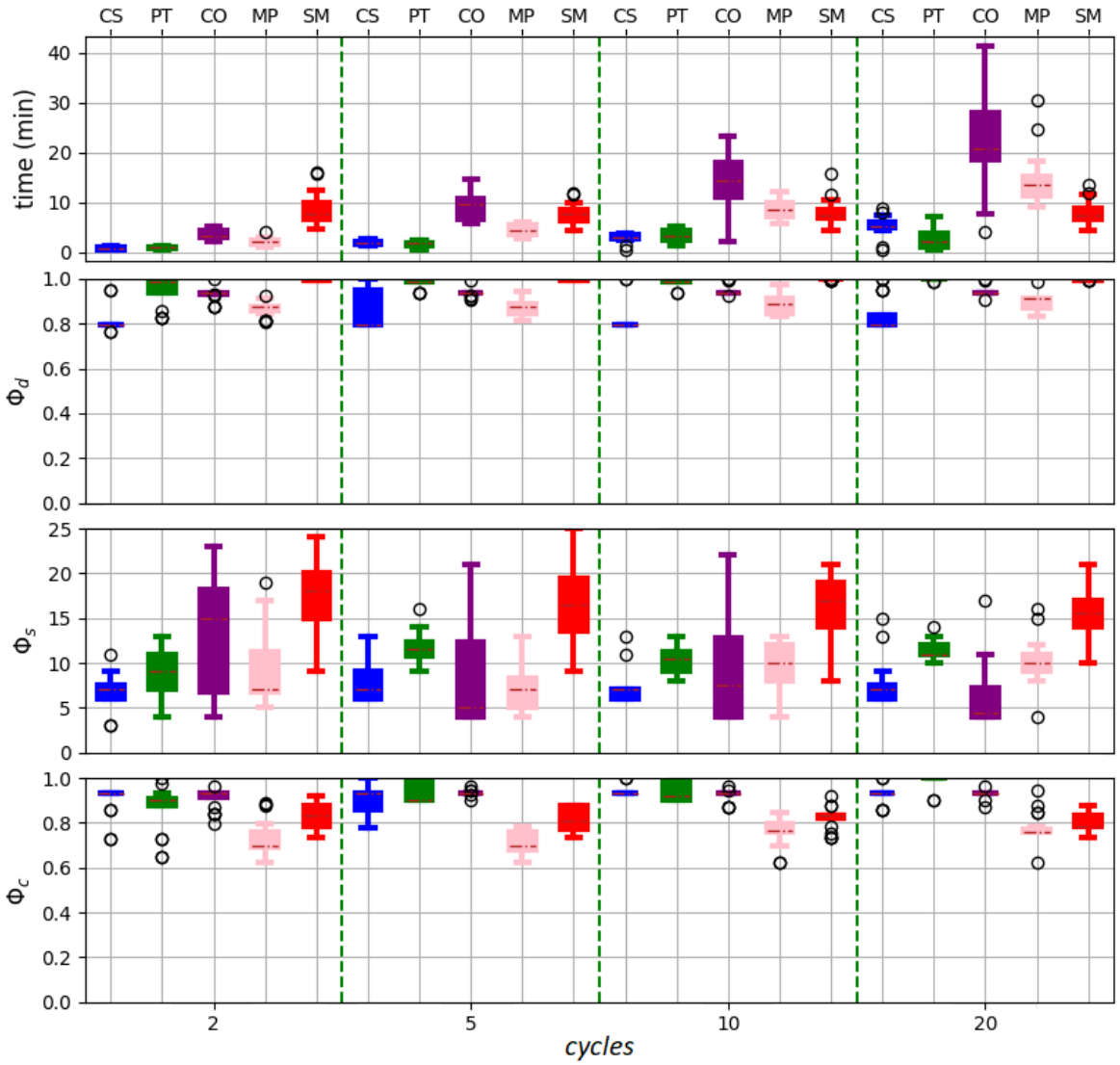


Figure 11: Result of end criteria analysis. The x-axis shows the amount of cycles in which FT-RL did not find a better FT before it was allowed to terminate.

The differences between the end cycles are very minor for the accuracy metrics. CSD finds better ϕ_d at 5 cycles, while PT finds better ϕ_c at 20 cycles. The rest of the cases do not change a lot. The size metric also does not change much, but actually rises for some cases with more cycles. Naturally the time for completion rises with increased cycles. To mitigate the increase in time we will set the end criteria cycles to 5 cycles.

6 Performance evaluation and validation

The previous section showed optimal values for many of the input parameters of FT-RL. We have set the reward function parameters p_d, p_s and p_c as 0.5, 0.2 and 0.5, respectively. The rounds per cycle are set to 2500 and the end criteria cycles to 5. With this setup we now test the performance of FT-RL. First we look at the scalability and time. We answer the research questions of how long it takes to infer an FT and how performance decreases with increasing FT complexity. Then we compare FT-RL to the state-of-the-art method of data driven FT inference: FT-MOEA [8], to answer the question how FT-RL compares to existing methods.

6.1 Scalability and time

For the scalability we have done two tests, one with artificial cases and one with the real case studies. The attributes of the ground truth FTs, used to generate the failure data for the artificial cases, are shown in Table 4. The artificial cases were made by hand to see how an increase of unique BEs in the failure dataset affects the results. The artificial cases do not only vary the amount of BEs, but also the amount of MCSs and their complexity.

Figure 12 shows the results of the experiment on the artificial cases. Note the green dotted line, points on the right of the line have been obtained without using the MCSs accuracy metric, meaning $p_c = 0$ in the reward calculations for those runs. This was done due to the immense amount of time that the MCS calculations can take with such large orders and amounts of MCSs. To give an indication: case 7 with MCS calculations enabled could not complete even one cycles in more than 10 hours. Figure 13 shows the results of FT-RL using the real case studies from Table 3. For these the same setup was used, and the MCS calculations were enabled for all of them.

Table 4: Cases used for the scalability analysis. The number of unique BEs (w), total BEs (W), AND gates (#AND), OR gates (#OR), Minimal Cut Sets (#MCS), order of MCSs (O-MCSs), and space complexity (O(2^w)). The cases are fabricated to see how much an increase of the amount of unique BEs affects the performance of the system.

Case	w	W	#AND	#OR	FT_{size}	#MCSs	O-MCSs	O(2^w)
1	2	2	1	0	3	1	{2}	4
2	4	4	1	1	6	2	{3,3}	16
3	6	6	2	1	9	2	{5,5}	64
4	8	8	3	1	12	3	{5,5,6}	256
5	10	10	3	2	15	6	{6,6,6,6,7,7}	1024
6	12	12	3	2	17	8	{7,7,7,7,7,7,8,8}	4096
7	14	14	3	2	19	12	{7,7,7,7,7,7,7,7,7,10,10,10}	16384

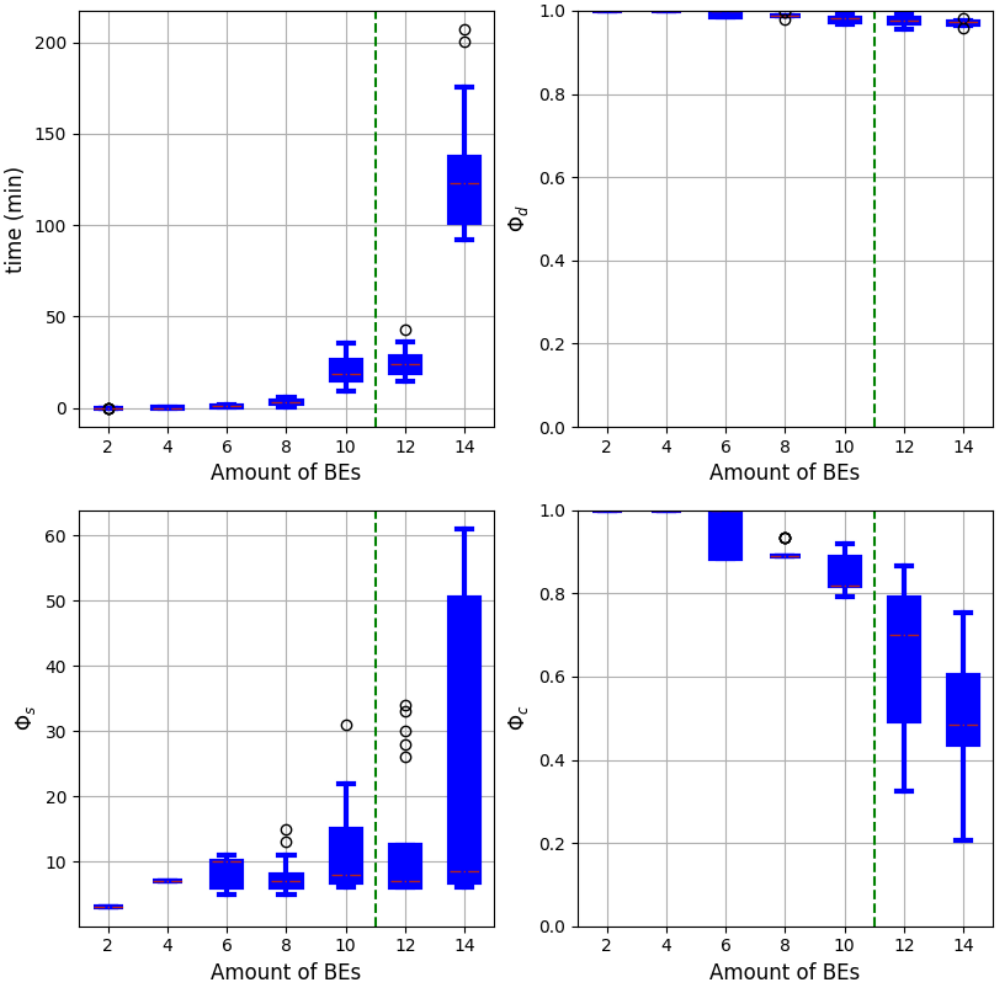


Figure 12: Result of scalability analysis. The time in minutes and the three metrics ϕ_d, ϕ_s and ϕ_c are shown. The boxplots are the average of 20 runs. Points on the right of the green dotted line are obtained without using the MCSs accuracy metric, meaning $p_c = 0$ in the reward calculations for those runs. The cases used are shown in Table 4.

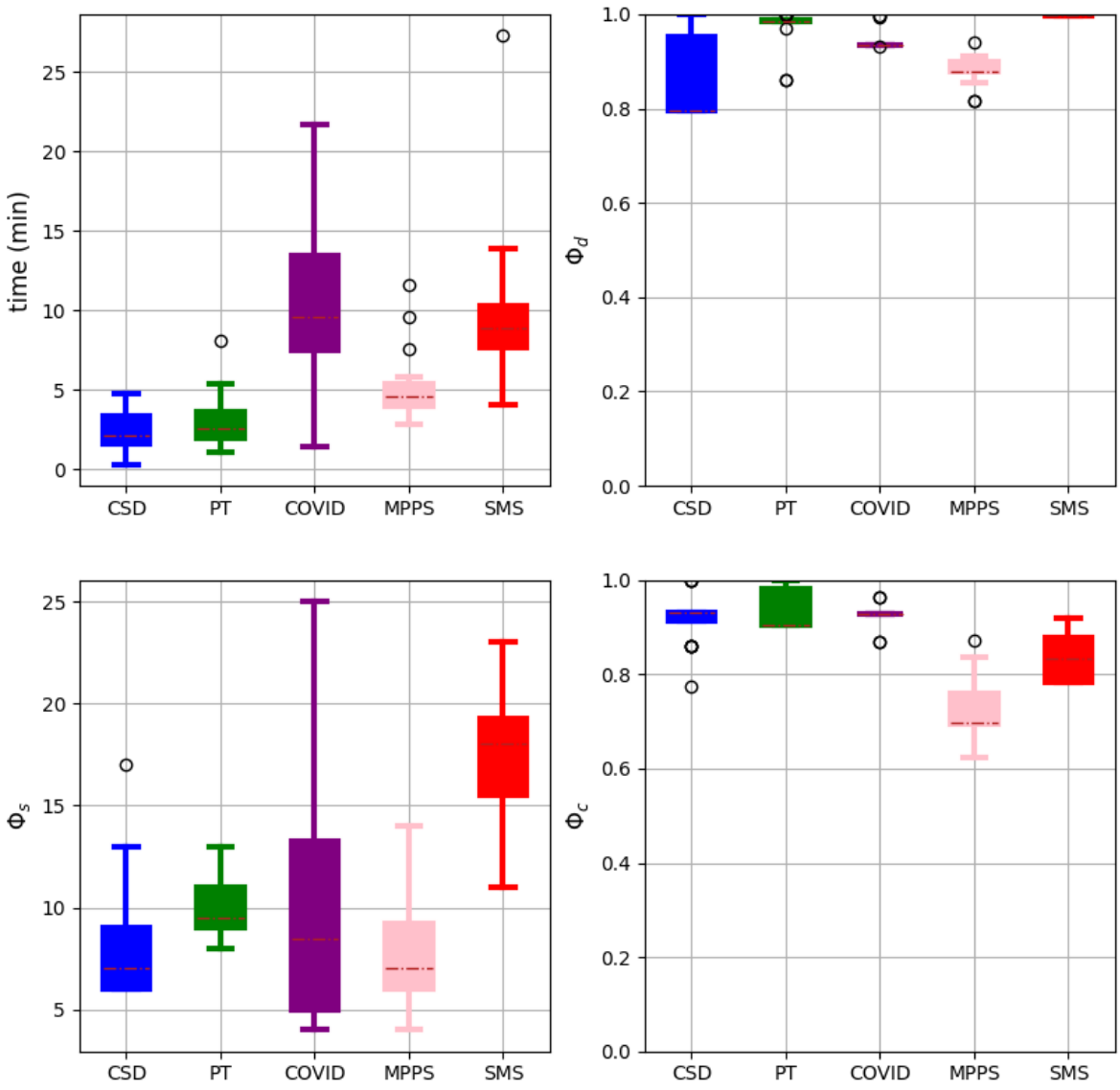


Figure 13: Boxplots of the results of FT-RL for the case studies of Table 3.

Looking at the artificial cases of Figure 12, it can be seen that the time increases significantly for larger amounts of BEs and increasing MCSs complexity. So much so that the runs on the right of the green line are without the MCS calculations (meaning $p_c = 0$). The MCS calculations take a very long time for a larger number of BEs. Even without these MCS calculations, the algorithm still takes two hours on average to complete for an FT with 14 BEs. After these two hours however, the accuracy is quite close to 100%. A slight drop can be seen for the ϕ_d as the BE amount increases. The ϕ_c drops a lot with increasing BEs and MCSs complexity, especially after the green line where it is no longer taken into account for calculating the rewards. This behavior of course, makes sense. The variance in size (ϕ_s) also increases significantly with increasing BEs, while the average stays below 10. When compared to the sizes of the ground truths in Table 4, it can be seen that the inferred FTs are smaller for the larger BE amounts. This is a reason why 100% accuracy cannot be achieved.

Figure 13 shows the results of FT-RL on the real case studies presented in Table 3. For most cases FT-RL was unable to find the ground truth FT. It can be seen that FT-RL struggles with finding 100% accurate FTs for these real cases, and that the range of accuracies it finds are rather narrow. It finds similar FTs very consistently for the PT, COVID, MPPS and SMS cases. Especially the SMS case shows that FT-RL is particularly good at scoring high on ϕ_d for MCSs of low order, even if the amount of unique BEs is larger.

Surprisingly, the SMS case does not need more than 10 minutes on average to complete. This is surprising because of the previous experiment shown in Figure 12. Since the SMS case has 13 unique BEs, one would expect an average run to complete in more than an hour, especially considering that the MCSs calculations are enabled for the SMS case. This indicates that the amount of unique BEs is not the main time consuming part of the algorithm. Naturally, the state space still grows exponentially with each added unique BE, but if the order of the MCSs are not high, FT-RL can find a solution close to 100% rather quickly.

The CSD case is interesting, as it does not contain many BEs, but it has an average of only 80% accuracy. Looking at Table 3 we may find an answer: the CSD case O-MCSs is {2,3,3}. This together with the other cases brings up the argument that FT-RL struggles with higher O-MCSs. Especially the SMS case shows this, which has a very high number of BEs, but only O-MCSs of 1. Its result of being near a 100% accuracy suggests that the O-MCSs is the main limiting factor. The reason that the CSD case goes to 80% and COVID only goes to a bit more than 90% on average, while COVID is the more complex case, is probably exactly because CSD is less complex. When the CSD case misses one of its MCSs, it impacts the overall accuracy more than when the COVID case misses one of its MCSs.

The artificial cases however, show that higher O-MCSs cannot be the main problem. They have very high O-MCSs, while still being able to find accuracies of above 90%. This shows that something else is going on. FT-RL struggles with MCSs that have high amounts of BEs in them, that are unique to only that MCS. We will explore this idea further, later in this section.

6.2 Visualization of inferred FTs

Here we would like to show some of the interesting inferred FTs that FT-RL has obtained for the case studies. Figure 14 shows the ground truth and two inferred FTs for the CSD case. One with higher reward, one with higher accuracy.

Figure 14b shows the FT with reward: 0.75, ϕ_d: 0.95, ϕ_c: 0.86 and ϕ_s: 9. Figure 14c shows another obtained FT with reward: 0.71, ϕ_d: 1.0, ϕ_c: 1.0 and ϕ_s: 17. Here it is clearly shown that the higher accuracy of the second result does not compensate for the larger size it has. The FT in 14b is missing one of the MCSs needed to get the 100% accuracy score, namely the MCS with BE4.

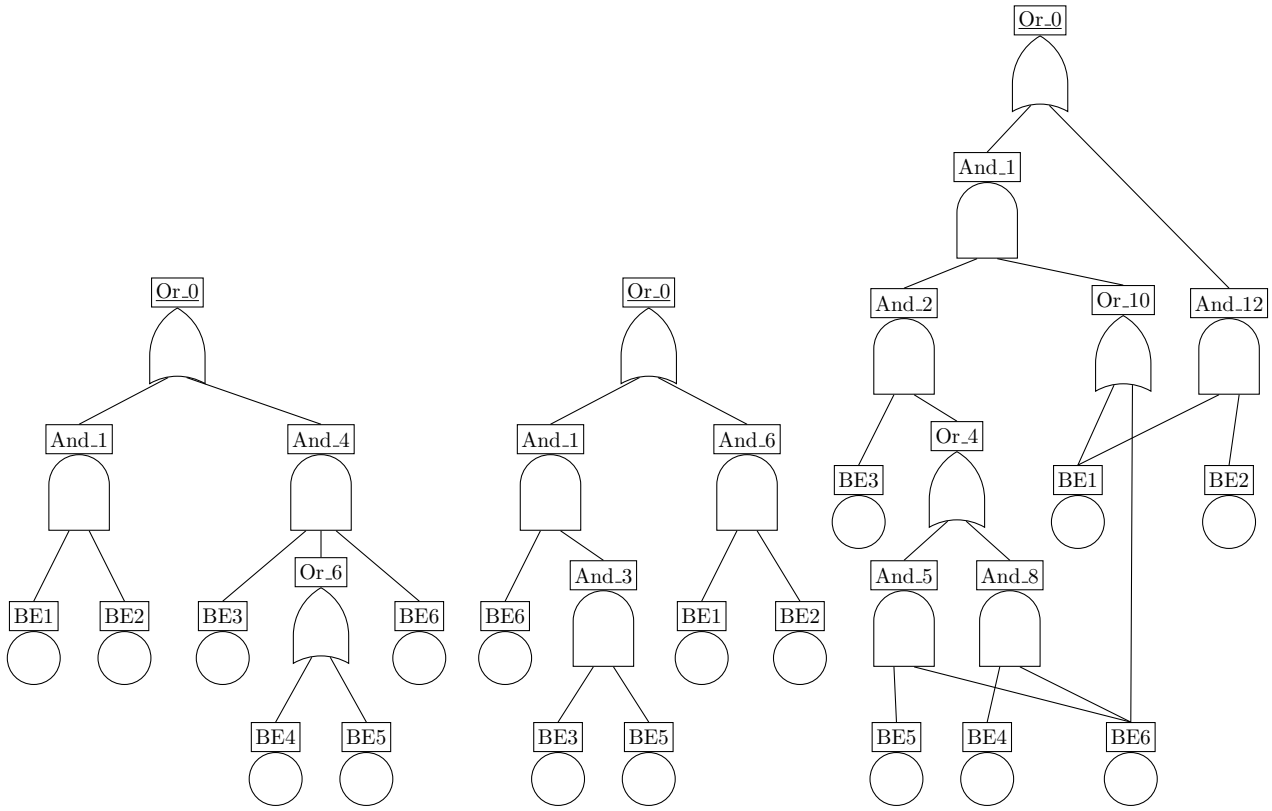
Figure 15 shows the results for the PT case: the ground truth and an inferred FT that is 100% accurate, and even more optimized than the ground truth FT.
The other cases can be seen in Appendix B.

6.3 Local optima

The results of the scalability and time analysis, show that FT-RL gets stuck in local optima for some cases. The narrow boxplots lower than 100% accuracy are an indication of that. The greedy nature of the algorithm is most likely the cause of this. It can occur that adding a specific element generally gives a better reward than adding any other element, while that element does not lead to the optimal solution.

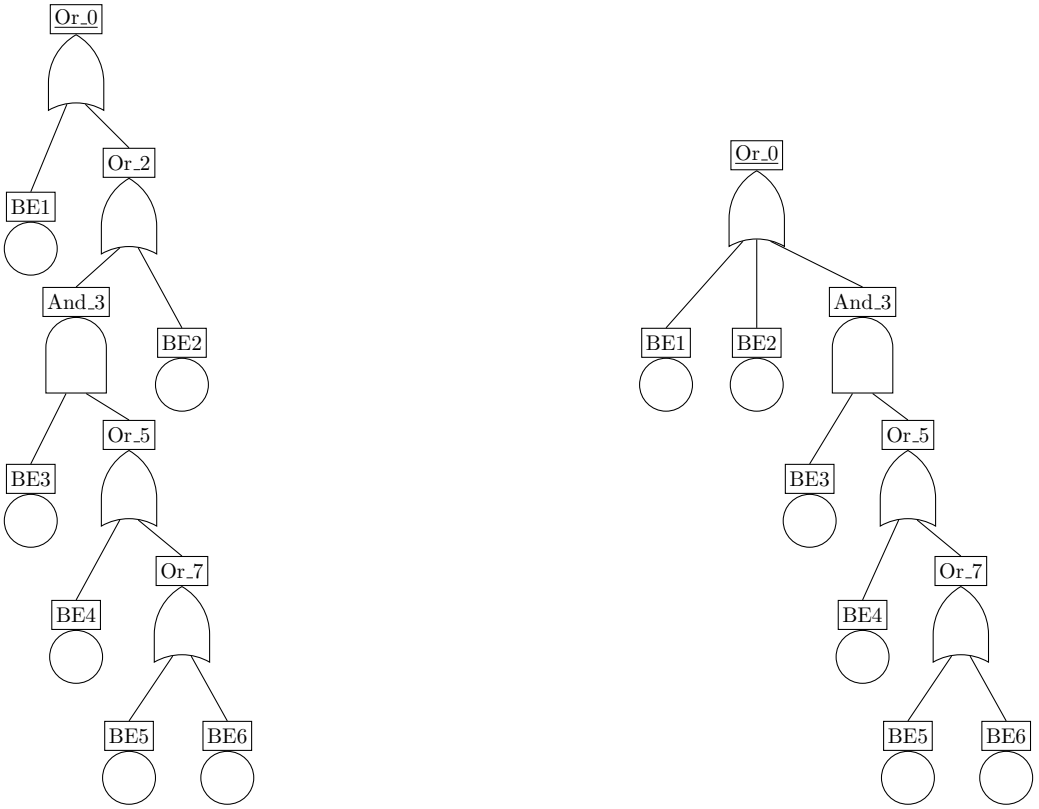
To illustrate this idea, we take a look at a subtree from the CSD case shown in Figure 14a. The subtree starts at And_4. We will try to understand why FT-RL picked the subtree starting at And_1 in Figure 14b the way it did. For that we first show the failure data of the subtree in Table 5. Then we depict the following scenario: the algorithm has started with an AND gate, and added BE6. At this point it can add another BE, or a gate.

In this example we will show why FT-RL will choose an AND gate, instead of the OR gate that is present in the ground truth. For brevity we show the cases for all "gate with two BEs" combinations. We then compare the TE results of each of these with the ground truth subtree. This is similar to how the accuracy metric ϕ_d is calculated. The results are shown in Table 6. It is clear that the best option would be to choose an AND gate, while the 100% accuracy solution lies with an OR gate. The algorithm does not know the results of adding the BE3 after the choice of the gate, so it will choose the AND gate.



(a) Ground truth (b) (c)

Figure 14: Ground truth and resulting FTs for the CSD case. (a) shows the ground truth, (b) shows a result with a higher reward (r: 0.75, ϕ_d: 0.95, ϕ_c: 0.86, ϕ_s: 9), (c) shows a result with a optimal accuracy (r: 0.71, ϕ_d: 1.0, ϕ_c: 1.0, ϕ_s: 17).



(a) Ground truth (b)

Figure 15: (a) Ground truth and (b) resulting FT (r: 0.85, ϕ_d: 1.0, ϕ_c: 1.0, ϕ_s: 10) for the PT case.

Table 5: Ground truth failure data for subtree AND(BE3,BE6,OR(BE4,BE5))

BE3	BE4	BE5	BE6	TE
1	1	1	1	1
1	1	1	0	0
1	1	0	1	1
1	1	0	0	0
1	0	1	1	1
1	0	1	0	0
1	0	0	1	0
1	0	0	0	0
0	1	1	1	0
0	1	1	0	0
0	1	0	1	0
0	1	0	0	0
0	0	1	1	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	0

Table 6: TE results for each of the subtrees. Red highlights the results that differ from the ground truth. Yellow highlights the subtree found by FT-RL. Green highlights the subtree that should have been found to later obtain the ground truth. AND and OR gates are abbreviated by A and O respectively.

Ground Truth A(3,6,O(4,5))	A(6,A(3,5))	A(6,A(3,4))	A(6,A(3,6))	A(6,A(4,5))	A(6,A(4,6))	A(6,A(5,6))
1	1	1	1	1	1	1
0	0	0	0	0	0	0
1	0	1	1	0	1	0
0	0	0	0	0	0	0
1	1	0	1	0	0	1
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Ground Truth A(3,6,O(4,5))	A(6,O(3,5))	A(6,O(3,4))	A(6,O(3,6))	A(6,O(4,5))	A(6,O(4,6))	A(6,O(5,6))
1	1	1	1	1	1	1
0	0	0	0	0	0	0
1	1	1	1	1	1	1
0	0	0	0	0	0	0
1	1	1	1	1	1	1
0	0	0	0	0	0	0
0	1	1	1	0	1	1
0	0	0	0	0	0	0
0	1	1	1	1	1	1
0	0	0	0	0	0	0
0	1	1	1	1	1	1
0	0	0	0	0	0	0
0	0	1	1	1	1	1
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	0	0	0	0	0	0

6.4 MCS order

The scalability results in Figure 13 clearly show that some cases do better than others. There is not a clear variable that seems to be the cause of this. The number of unique BEs, total BEs, AND gates (#AND), OR gates (#OR), Minimal Cut Sets (#MCS), order of MCSs (O-MCSs) nor space complexity (O(2w)) clearly scale with decreasing accuracy. The trio of the CSD case, the SMS case and case 7 from Table 4, give the idea that another parameter is the cause for a drop in accuracy.
The CSD case shows that a smaller FT can have low accuracy results. The SMS case shows that a large amount of MCSs with low order can still have high accuracy results. Case 7 shows that large amount of MCS with high order can still have high accuracy based on data ϕ_d. This means it is not the size, nor the order of the MCSs that causes the drop in accuracy. One interesting observation, is that the CSD case has quite a few unique BEs in their MCSs, while the SMS case and case 7 do not. We hypothesize that the amount of unique BEs in the MCSs (U-MCSs), is the problem. To confirm this, another experiment has been done. The cases are shown in Table 7 and Figure 16, with their MCSs in Table 8, where the overlapping and unique BEs are clearly shown. Figure 17 shows the drop

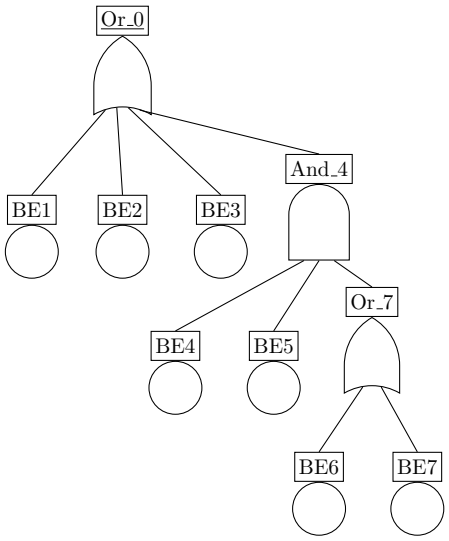
in accuracy with higher amounts of U-MCSs. Case 1 in blue, shows it is able to get very close to 100% accuracy. Case 2 in green, already performs much poorer, being around 90% accuracy. Case 3 in purple, with the most amount of unique BEs in their MCSs, shows the poorest performance. The other aspects of these FTs are kept mostly the same, this shows that higher U-MCS indicates poorer performance of FT-RL.

Table 7: Cases used for the O-MCSs analysis. The number of unique BEs (w), total BEs (W), AND gates (#AND), OR gates (#OR), Minimal Cut Sets (#MCS), order of MCSs (O-MCSs), amount of unique BEs in the MCSs (U-MCSs), and space complexity (O(2^w)). The cases are fabricated to see how much an increase of the amount of unique BEs affects the performance of the system.

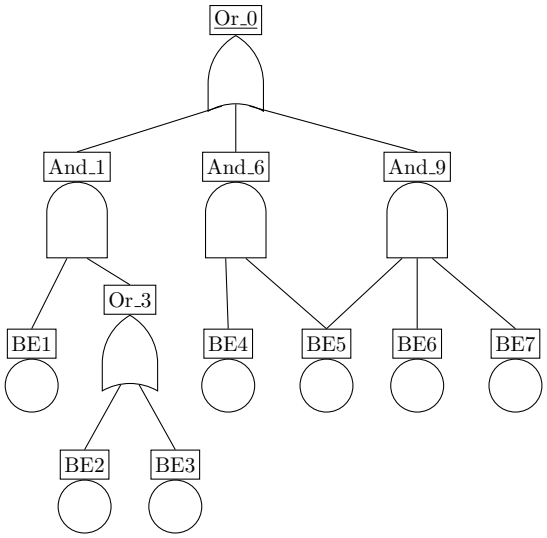
Case	w	W	#AND	#OR	FT_{size}	#MCSs	O-MCSs	U-MCSs	O(2^w)
1	7	7	1	2	10	5	{1,1,1,3,3}	{1,1,1,1,1}	128
2	7	8	3	2	13	5	{2,2,2,3}	{1,1,1,2}	128
3	7	7	3	1	11	3	{3,2,2}	{3,2,2}	128

Table 8: MCS of the cases. Unique BEs in the MCS are highlighted in red.

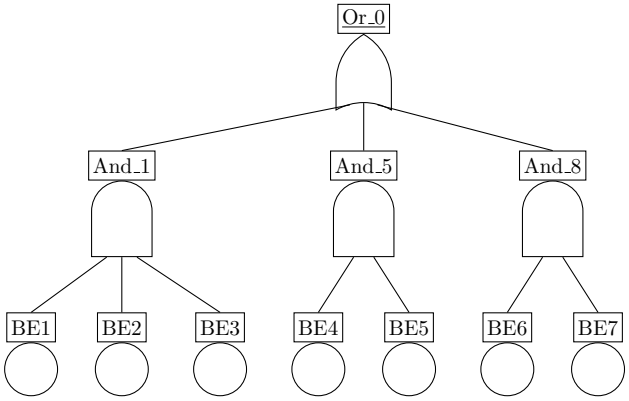
Case 1	BE1	BE2	BE3	BE4	BE5	BE6	BE7
MCS1	1	0	0	0	0	0	0
MCS2	0	1	0	0	0	0	0
MCS3	0	0	1	0	0	0	0
MCS4	0	0	0	1	1	1	0
MCS5	0	0	0	1	1	0	1
Case 2	BE1	BE2	BE3	BE4	BE5	BE6	BE7
MCS1	1	1	0	0	0	0	0
MCS2	1	0	1	0	0	0	0
MCS3	0	0	0	1	1	0	0
MCS4	0	0	0	0	1	1	1
Case 3	BE1	BE2	BE3	BE4	BE5	BE6	BE7
MCS1	1	1	1	0	0	0	0
MCS2	0	0	0	1	1	0	0
MCS3	0	0	0	0	0	1	1



(a) Case1



(b) Case2



(c) Case3

Figure 16: FT cases used to understand the unique BEs in MCSs.

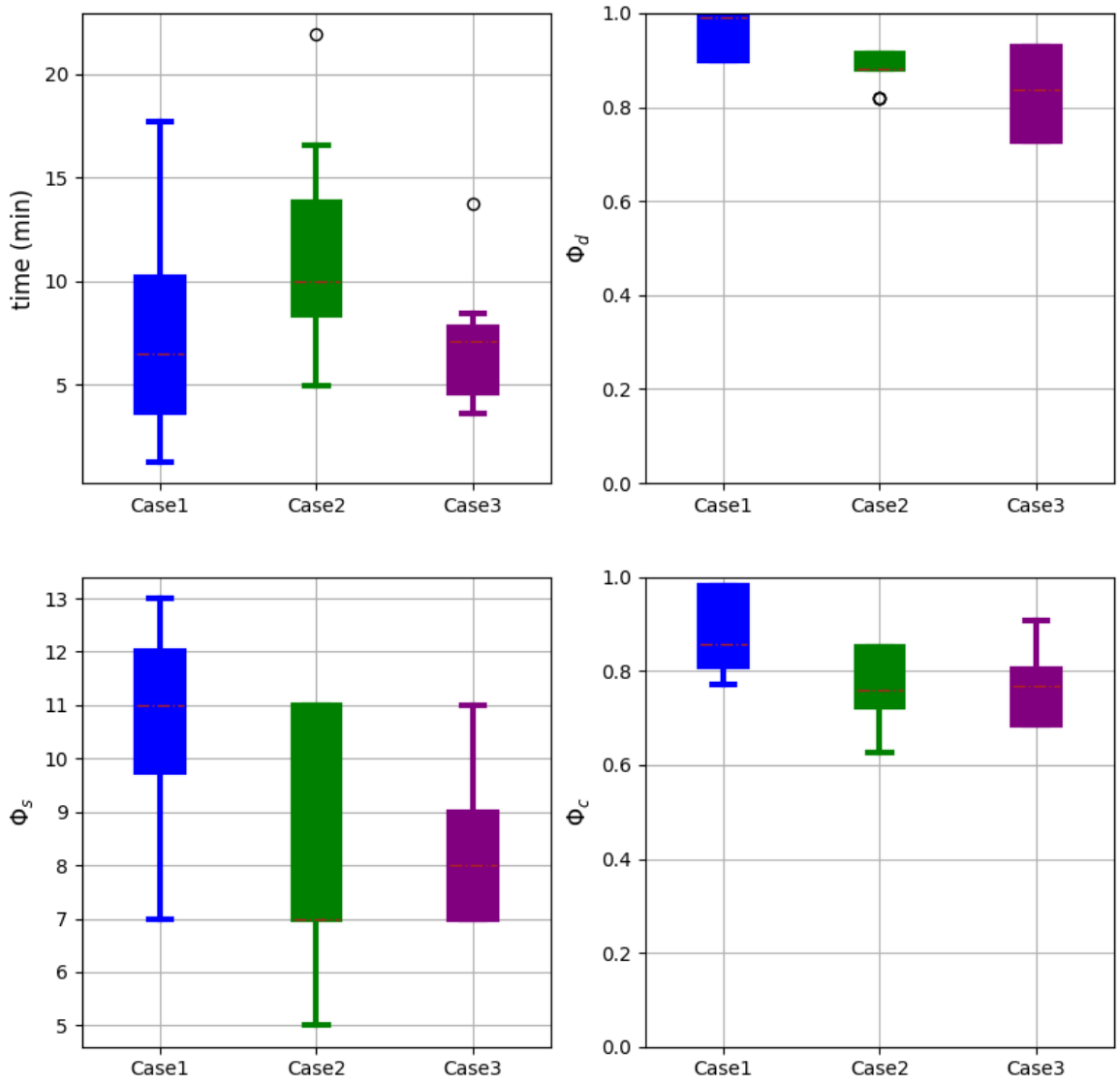


Figure 17: Results of the analysis on increasing U-MCS. The cases are shown in Table 7.

The cause of the drop in accuracy is the random nature of the algorithm. It is simply more likely to randomly find structures of MCSs that have overlapping BEs, rather than unique ones. To illustrate this, we paint two scenarios. We look at a decision the algorithm will make at the point where it has started with an OR gate with an AND gate as its child OR(AND(:),(:)). From this point it can add one of the four BEs and more gates. We look at the case where it adds two BEs.

Scenario 1: overlapping BEs
Ground truth: OR(AND(2,1),AND(2,3),AND(2,4)) or AND(2,OR(1,3,4)), which has MCSs {BE1,BE2}, {BE2,BE3} and {BE2,BE4}.

Scenario 2: unique BEs
Ground truth OR(AND(1,2),AND(3,4)), which has MCSs {BE1,BE2} and {BE3,BE4}.

There are twelve total BE combinations that can be added to the AND gate at this point. Each of these combinations is shown in Table 9. A checkmark means that one of the MCSs is found, while an x means that no MCS is found. It is clear that for scenario 1, it is easier to randomly find one of the MCSs. This causes the addition of BE2 to generally get a higher reward than the other BEs, causing it to be chosen more often, causing the other MCSs to have a higher chance to be found as

well. (For example by adding BE2 and then an OR gate).

Table 9: Scenarios

Scenario 1							
Added	MCS	Added	MCS	Added	MCS	Added	MCS
BE1, BE2	✓	BE2, BE1	✓	BE3, BE1	x	BE4, BE1	x
BE1, BE3	x	BE2, BE3	✓	BE3, BE2	✓	BE4, BE2	✓
BE1, BE4	x	BE2, BE4	✓	BE3, BE4	x	BE4, BE3	x
Scenario 2							
Added	MCS	Added	MCS	Added	MCS	Added	MCS
BE1, BE2	✓	BE2, BE1	✓	BE3, BE1	x	BE4, BE1	x
BE1, BE3	x	BE2, BE3	x	BE3, BE2	x	BE4, BE2	x
BE1, BE4	x	BE2, BE4	x	BE3, BE4	✓	BE4, BE3	✓

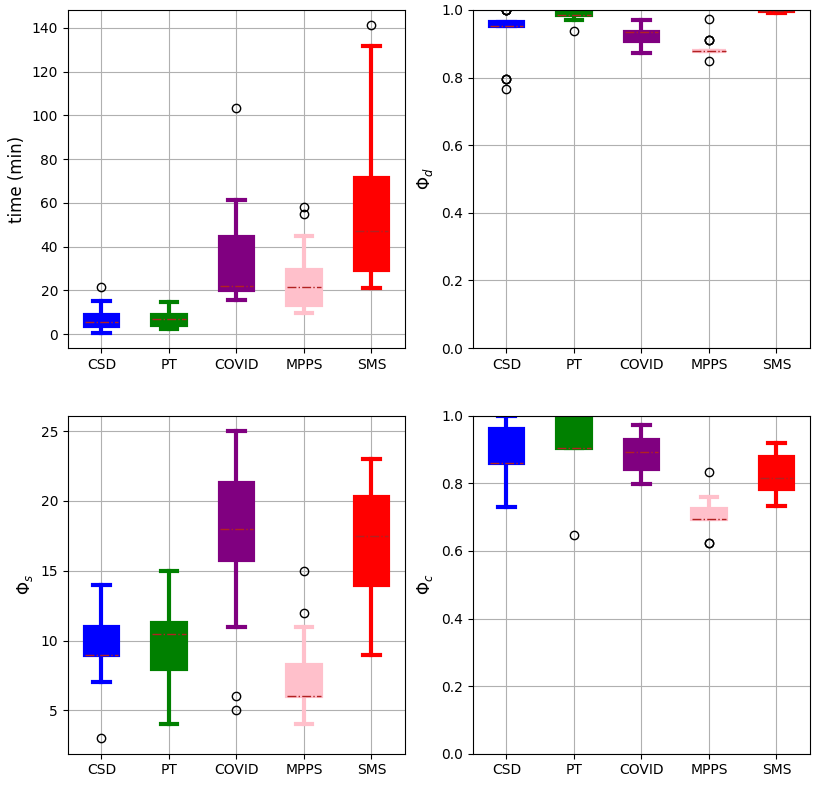
6.5 Intermediate rewards

Intermediate rewards are the rewards given when going from an intermediate state to another. It is the r in Eq. 1. Giving these intermediate rewards to EFTs was considered not possible. The reward is calculated with metrics that can only be calculated with a completed FT. This is why backpropagation of the reward was used. It is possible to evaluate FTs that are not completed, but only if parts of the FT are ignored or filled out. Since intermediate rewards have the potential to make the algorithm much better, we evaluated unfinished FTs as follows: AND gates with no children default to FALSE and OR gates with no children default to TRUE. The alternative would be to ignore gates with no children in the evaluation. That would mean there is no distinction between the AND or OR gate. To keep the distinction, we give them the default values. FALSE meaning no failure and TRUE meaning failure. To see how well the intermediate rewards do on their own, the backpropagation was turned off for this experiment. The results are shown in Figure 18.

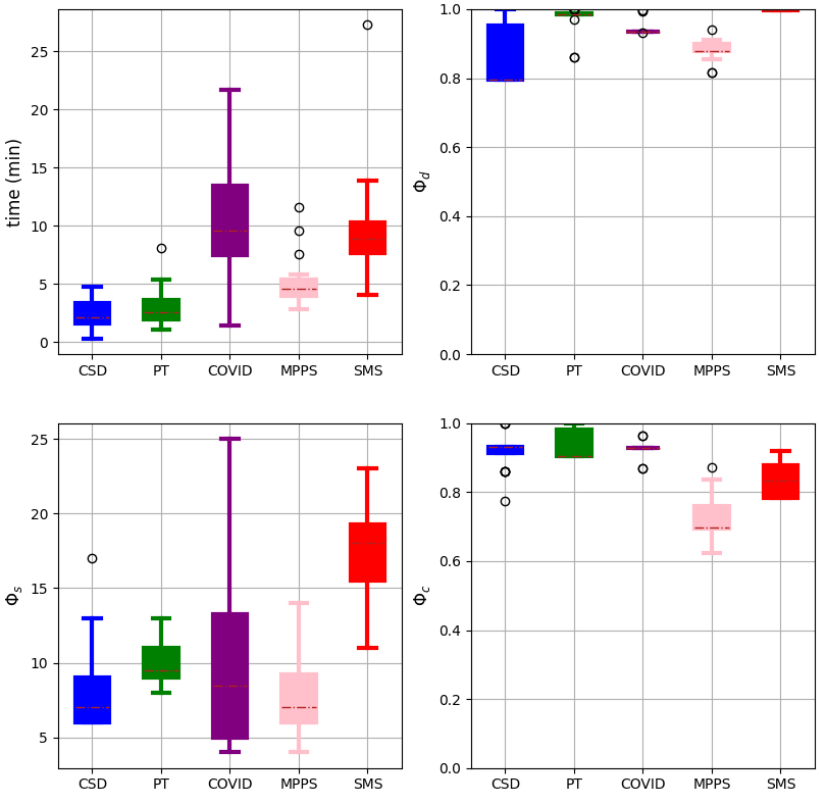
It is clear that the intermediate reward approach gives similar results to no intermediate rewards with backpropagation. The main difference is that backpropagation makes the process much faster. The runs with intermediate rewards take about 20 minutes on average while the ones with backpropagation take 5 minutes on average. It is interesting to see that for the CSD case, the intermediate rewards seem to be giving better results.

6.6 Comparison to FT-MOEA

We now display how FT-RL compares to FT-MOEA [8]. We have ran both FT-RL and FT-MOEA by taking the same case studies as before, shown in Table 3. And then generating their failure data as input for both the algorithms. The results can be seen in Figure 19.



(a)



(b)

Figure 18: (a) Results of FT-RL with intermediate rewards instead of backpropagation. (b) Results with no intermediate rewards and backpropagation.

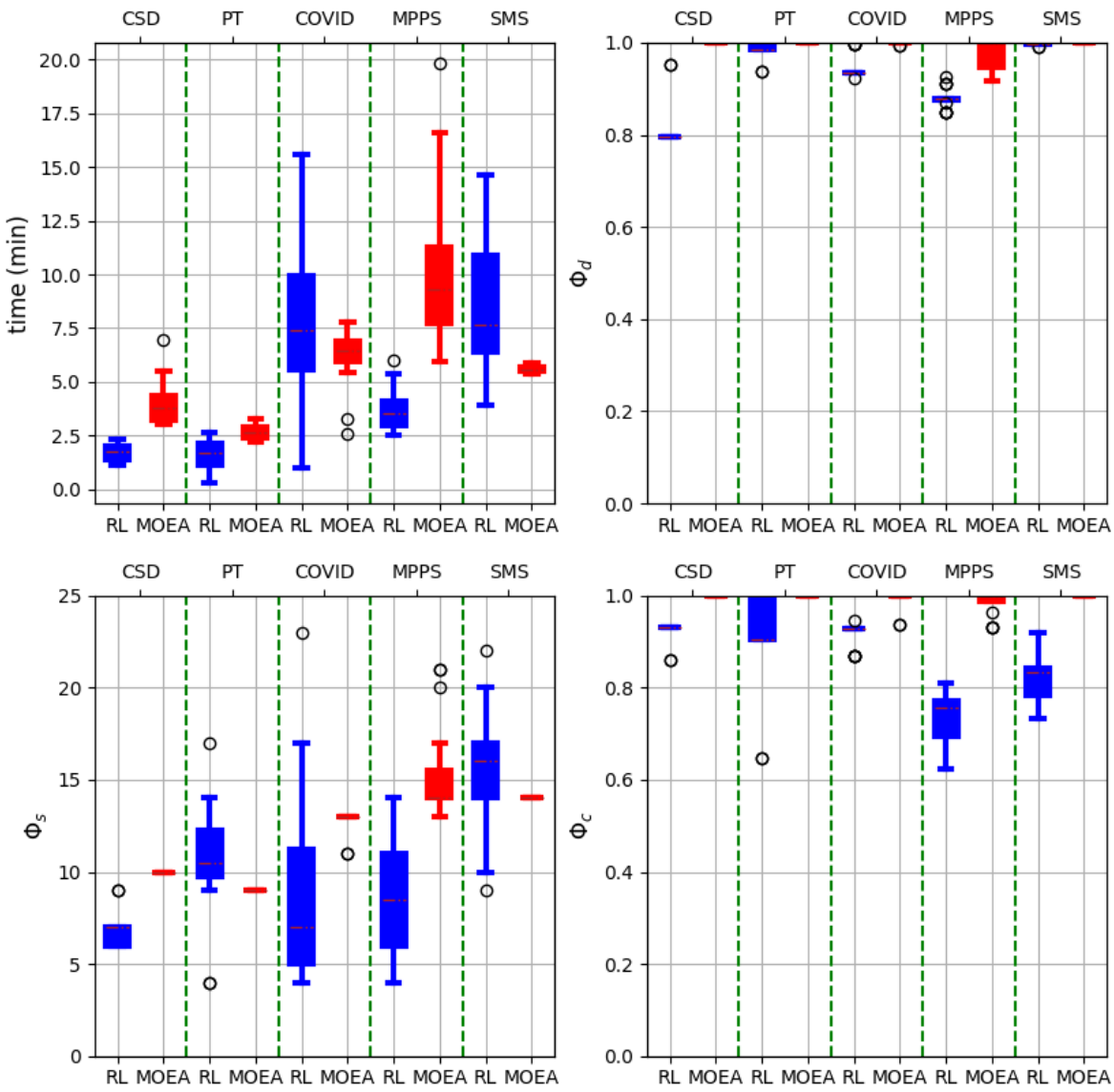


Figure 19: Comparison between FT-RL and FT-MOEA for the case studies of Table 3. FT-RL results are in blue and FT-MOEA results are in red. The dotted green lines are there for clear separation between the case studies.

It is quite clear that FT-MOEA outperforms FT-RL. Where FT-RL has better times, FT-MOEA has better accuracy. The size metric shows that FT-RL has a far broader range of sizes, and seems to give smaller FTs, while FT-MOEA is very consistent.

7 Conclusion

We investigate the data-driven inference of Fault Tree (FT) models, which is the process that produces an FT model given compatible input information. We do this via Reinforcement Learning (RL) by defining a framework based on Q-learning that can automatically infer FTs from a failure data set. We call the implementation of this framework FT-RL. We answer **RQ1**: *How can FT inference be formulated as a reinforcement learning problem?* with the following definition:

- *Environment*: FT-RL translates the inference of FTs into a Markov decision process. FT-RL environment is defined as this MDP process that sequentially adds elements to an Expandable Fault Tree (EFT), until a complete FT is obtained.
- *State-space*: We define Expandable Fault Trees (EFT), which is an extension of FTs that contain Expansion Points (EP). These EPs are children of uncompleted Gates in the EFT. EPs mark the Gates as expandable. Additional elements can be added to the EFT at the EPs. Each EFT is considered as a state.
- *Actions-space*: FT-RL has two actions, namely *Add* and *Stop*. The first add elements (i.e., basic event or gates) to an EFT at the EPs only. The second deletes EPs only. Once no more EPs are available, FT-RL reaches the final state. This is the inferred FT.
- *Rewards*: For the inferred FT, FT-RL computes a reward based on three metrics namely accuracy based on the dataset (ϕ_d), accuracy based on Minimal Cut Sets (MCS) (ϕ_c), and Fault Tree size (ϕ_s). The above metrics are weighted in the reward function, by the parameters p_d, p_c, and p_s, respectively, which are part of FT-RL input parameters. This item answers **RQ2.1**: *How should the rewards be given?* and **RQ2.2**: *What metrics should be used?*

The methodology that FT-RL follows can be summarized into 6 steps, which corresponds to *Step 1. initialization* where input parameters such as the failure data and the parameters p_d, p_c, and p_s are defined. *Step 2. start new cycle*, starts a new cycle of FTs being inferred. *Step 3. start new FT*, infers the new FT by starting an EFT and sequentially adding elements to it until an FT is obtained. *Step 4. evaluation* the obtained FT is evaluated with the reward. *Step 5. comparison* the obtained FT is compared to previous obtained FTs, the best one is kept. *Step 6. convergence* if the best obtained FT adheres to the convergence criteria, the process is complete. If not, a new FT is started at Step 3. The convergence criteria are that there is unchanged performance metrics after 5 number of cycles, or an FT with 100% accuracy metrics is obtained.

The experimental setup for testing FT-RL is based on five case studies from the literature. These case studies act as ground truths, from which we generate synthetic failure data sets that are then used by FT-RL to infer FTs. These case studies provide us with information on the optimal hyper-parameters of FT-RL and the overall performance of the algorithm.

To answer **RQ2.3**: *What is the best prioritization of metrics for RL?* we carry out a hyper-parameters tuning, where we found that based on the case studies, FT-RL achieves the best overall performance when the weight for ϕ_d and ϕ_c is $p_d = p_c = 0.5$, and for ϕ_s is $p_s = 0.2$. Prioritizing the accuracy metrics over the size metric leads to better encoding of the logic present in the failure dataset.

From the five case studies, FT-RL scored an accuracy based on the dataset between 80% to 100%, and accuracies based on the MCSs of 70 to 90%, in 3 to 10 minutes on average. The accuracies and times vary significantly between the different cases. This answers **RQ3.1**: *How fast can the algorithm produce a correct FT?* We found that the current implementation of FT-RL is not consistent, as it infers different FT each time the algorithm runs for the same failure data input.

To answer **RQ3.2**: *How much does performance decrease with increasing FT complexity, e.g. increasing number of BEs?* we look at the five case studies again, together with 7 artificial cases. We have found that FT-RL underperforms in cases that have minimal cut sets with no shared basic events, i.e., the FTs have independent sub-trees. This might be due to the greedy nature of FT-RL causes it to fall into local optima. FT-RL also takes significantly more time for FTs with higher order minimal cut sets. The time to complete scales exponentially, meaning larger FT sizes are not feasible.

To answer **RQ4.1**: *How does this new solution compare to existing techniques?* We have compared FT-RL with the state-of-the-art algorithm FT-MOEA (based on multi-objective evolutionary

algorithms). We found that FT-RL consistently scores lower in the accuracy metrics. While only sometimes scoring better for the time and size metrics.

Among the benefits of FT-RL we can mention that it can consistently find relatively small FTs in similar amount of time as the state-of-the-art. The FT-RL framework allows for much flexibility in prioritization of accuracy or size, and can easily be expanded to address some of the limitations it has. Among FT-RL limitations we can mention that it consistently does not find 100% accurate FTs for many cases. It has much trouble with finding unique sub-tree structures. The time to complete for larger FT sizes also grows exponentially, causing larger sizes to not be feasible with this method. These limitations as framed as future work, which is discussed in the following section.

8 Future Work

In this section, we discuss what parts of FT-RL can be improved, and what experiments can still be interesting to do.

8.1 Additional analysis for FT-RL

- Parametric analysis: Since there are many variables and input parameters to FT-RL, it is worthwhile to explore additional setups. We found that there was not one optimal setup of parameters for all cases. More combinations of reward weights, cycles and round amounts should be tested on a broader range of cases.
- Expansion points: The way that expansion points in EFTs are found could influence the types of FTs found. In all experiments here, EPs were found in a depth-first manner. Finding them in a breath-first manner could maybe solve some of the scenarios where the algorithm greedily picks the wrong option. But it may also cause new scenarios like that. The parametric analysis could also be more refined. The accuracy metrics seem to snap in place between 0 and 0.2, taking more steps between these values may show interesting behavior.
- Intermediate rewards: Exploring different ideas for the intermediate rewards could result in higher accuracies. We saw that the CSD case had a higher average accuracy with intermediate rewards than with backpropagation. More experimentation should be done to find out why. A combination of the intermediate rewards with the backpropagation should also be tested.

8.2 To overcome current FT-RL drawbacks

- Regarding Minimal cutsets: The calculations of the MCS accuracy metric proved to be very time-consuming. For two of the artificial cases we had to turn off the MCS calculations, which made its completion speed go from an estimated 50 hours, to about 2 hours on average. Granted these artificial cases had far larger MCS matrices than the real case studies, but this needs to be addressed if FTs with bigger, more complex MCSs want to be inferred.
- Optimizing FT structure: FT-RL currently finds an FT that has both high accuracy and low size. A different approach could be tried where the process is split into two parts: start by finding an FT that has high accuracy, then optimize that FT to reduce its size. This second part would require another approach that can take an FT as its input (e.g., FT-MOEA [8]). This algorithm would then have actions that can remove parts of the FT. This way, the tradeoff between accuracy and size could be alleviated. Another optimization for the structure that we missed was restricting the children of a gate so it cannot contain the same gate as the parent gate. This simple restriction can already optimize the size.
- Learning method: A different RL method can also be tried. FT-RL uses Q-learning, but there are many other forms of RL, and also other deep learning techniques. A logical next step to take would be to explore deep Q-learning [21]. These deep learning techniques with neural networks could alleviate the problems the greedy Q-learning algorithm causes.
- Additional gates: The range of FTs that FT-RL is able to infer can be extended, by the addition of more gate types. Exploring the idea of adding subtrees or structures as actions instead of just adding single elements can also be explored. Focus could be put on the MCSs, especially the MCSs with unique BEs. This can be derived from the failure data, then a subtree can be made that with these unique BEs. This subtree can then be added at once with an action, making it almost guaranteed that that MCS is found.

Appendices

A Duplicate gates optimization

One of the future work suggestions was to remove duplicate gates. This means that a gate cannot have its same gate type as a direct child. Since this was a rather quick implementation, this experiment has been done last minute. The results are not taken into consideration on the rest of the discussion. The results are shown in Figure 20.

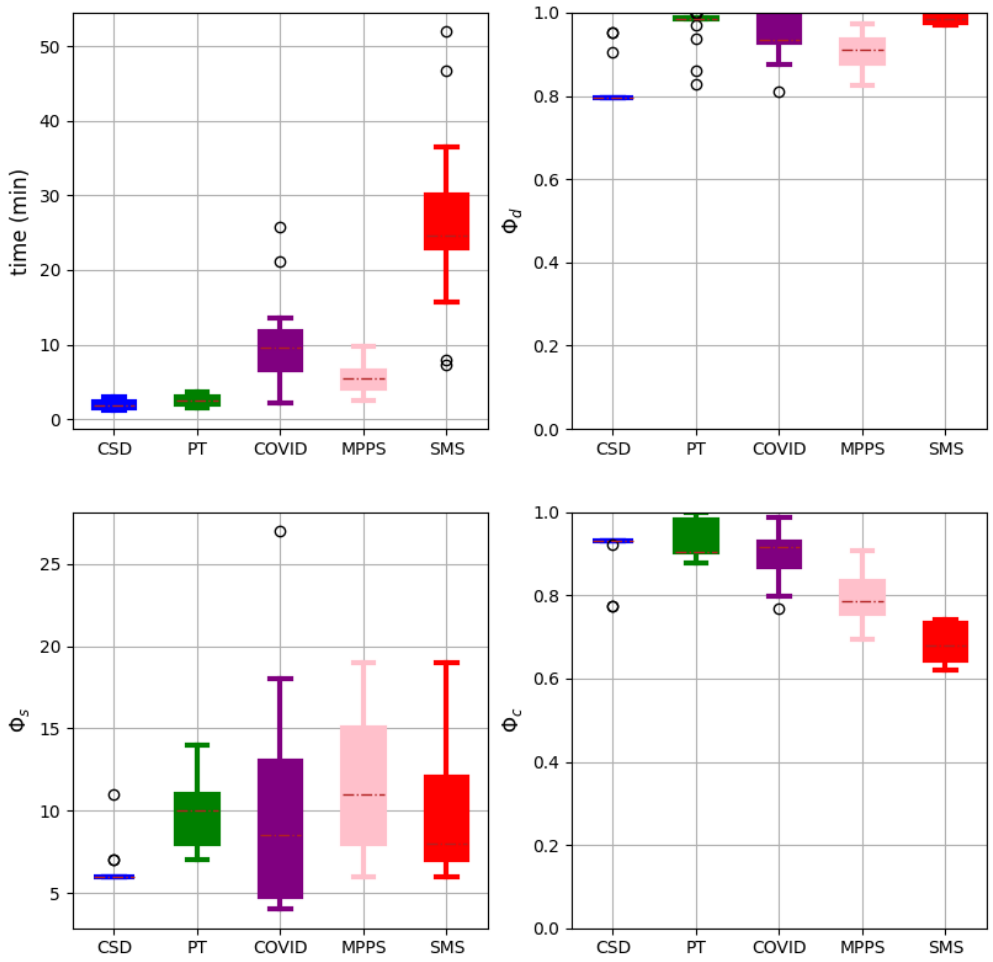


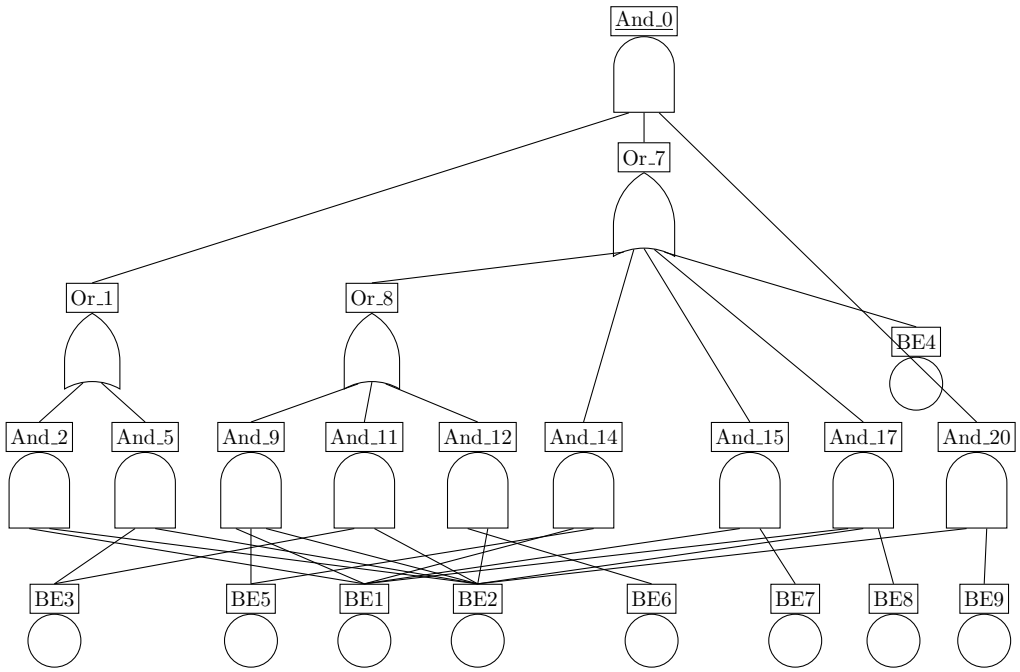
Figure 20: Results of FT-RL on the case studies of Table 3 with the added duplicate gates optimization.

Unfortunately, there are no major differences to be seen between Figure 20 and Figure 13.

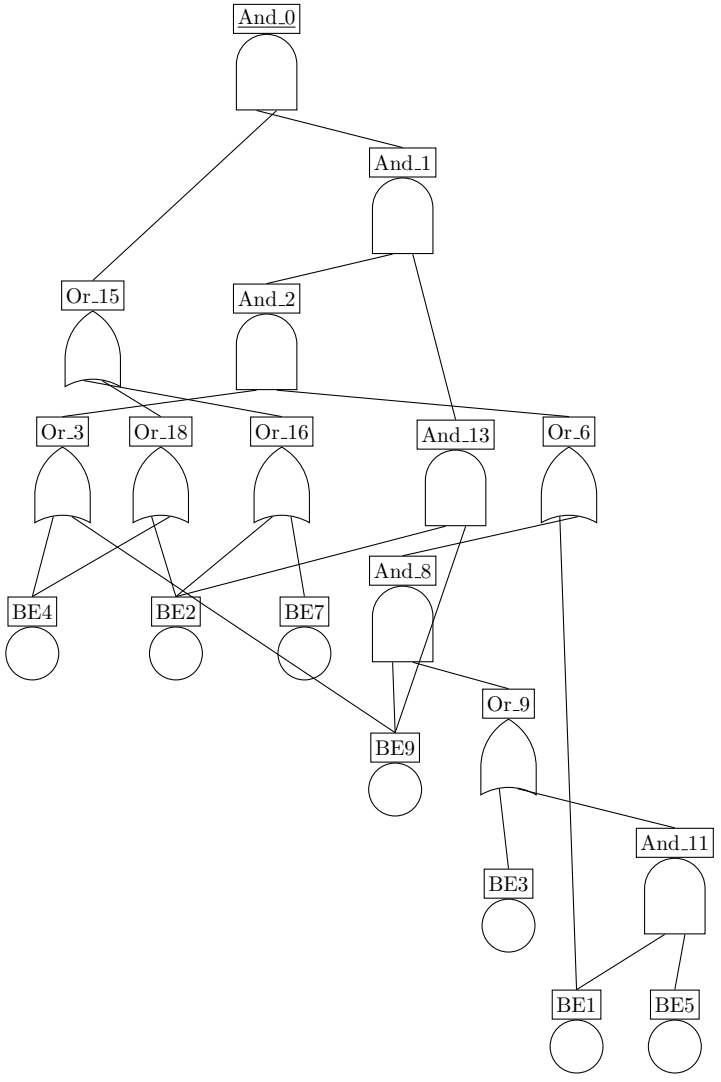
B Visualization of inferred FTs case studies

Here we display the visualizations of the ground truth FTs and their inferred FTs by FT-RL for the cases of Table 3 that were not discussed in the thesis. Figure 21 shows the COVID case. Figure 22

shows the MPPS case. The inferred FT is missing BE2 and BE3 completely. It does capture the behavior with the OR gates for BE5, 6, 7 and 8. And the MCS of BE1 with BE4. It has reward: 0.78, ϕ_d: 0.94, ϕ_c: 0.87 and ϕ_s: 12. And Figure 23 shows the SMS case.

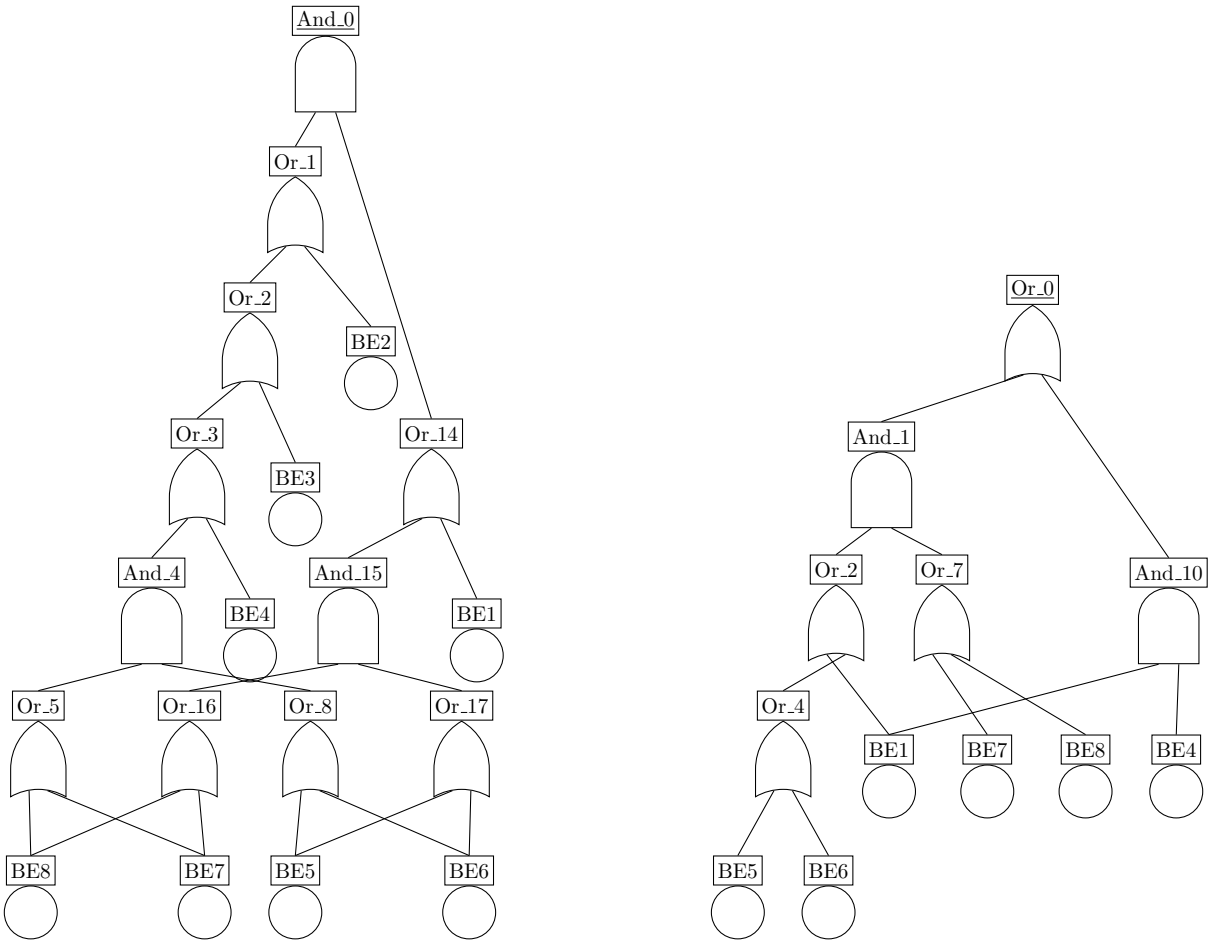


(a) Ground truth



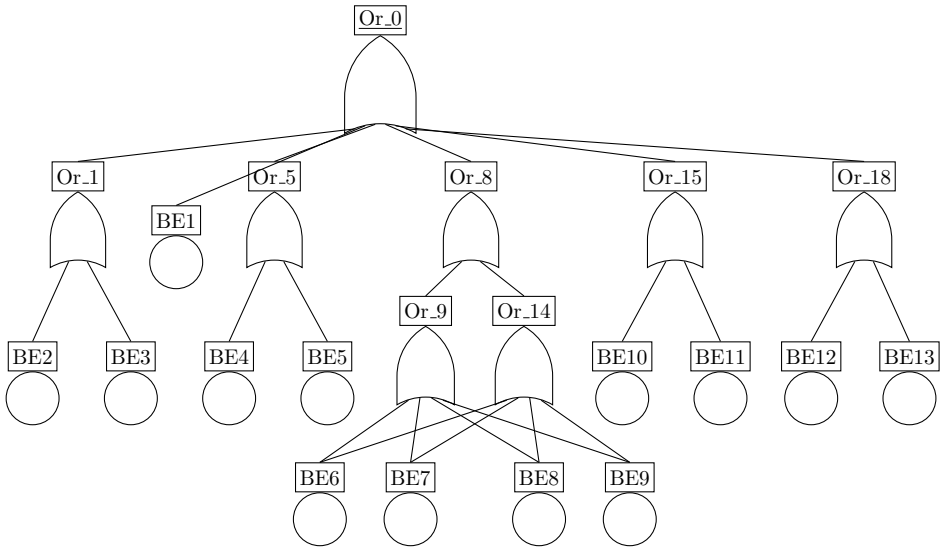
(b) Resulting FT

Figure 21: Ground truth (a) and resulting FT (b) for the COVID case

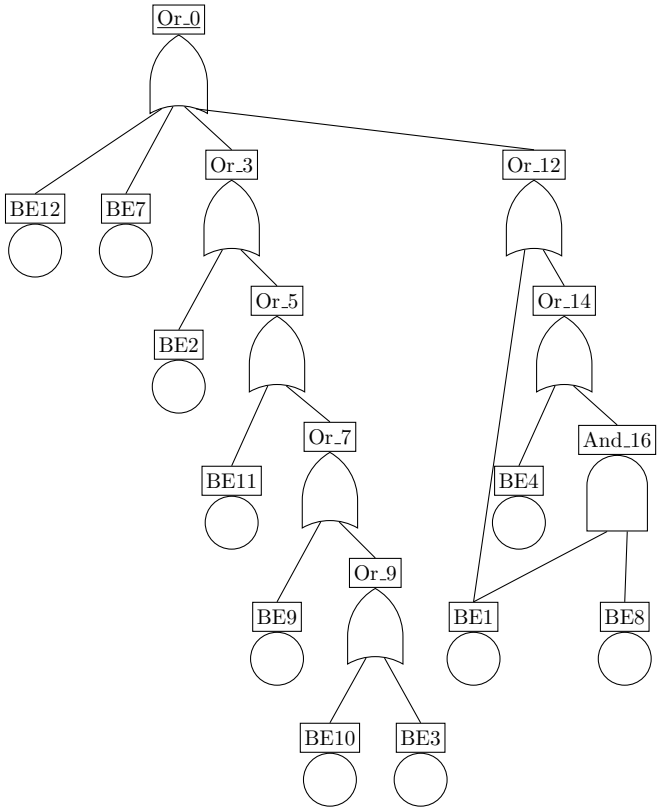


(a) Ground truth (b)

Figure 22: (a) Ground truth and (b) resulting FT (r: 0.78, ϕ_d: 0.94, ϕ_c: 0.87, ϕ_s: 12) for the MPPS case.



(a) Ground truth



(b) Resulting FT

Figure 23: Ground truth (a) and resulting FT (b) for the SMS case.

References

[1] Sohag Kabir. An overview of fault tree analysis and its application in model based dependability analysis. *Expert Systems with Applications*, 77:114–135, 2017.

[2] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.*, 15:29–62, 2015.

[3] A. Carpignano and A. Poucet. Computer assisted fault tree construction: a review of methods and concerns. *Reliability Engineering & System Safety*, 44(3):265–278, 1994. Special Issue On Advanced Computer Applications.

[4] Faida Mhenni, Nga Nguyen, and Jean-Yves Choley. Automatic fault tree generation from sysml system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 715–720, 2014.

[5] Meike Nauta, Doina Bucur, and Mariëlle Stoelinga. Lift: learning fault trees from observational data. In *International Conference on Quantitative Evaluation of Systems*, pages 306–322. Springer, 2018.

[6] ARR Linard, Marcos LP Bueno, Doina Bucur, and Mariëlle Ida Antoinette Stoelinga. Induction of fault trees through bayesian networks. *Proceedings of the 29th European Safety and Reliability Conference (ESREL)*, 2019.

[7] Alexis Linard, Doina Bucur, and Mariëlle Stoelinga. Fault trees from data: Efficient learning with an evolutionary algorithm. In Nan Guan, Joost-Pieter Katoen, and Jun Sun, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 5th International Symposium, SETTA 2019, Shanghai, China, November 27-29, 2019, Proceedings*, volume 11951 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2019.

[8] Lisandro Arturo Jimenez-Roa, Tom Heskes, Tiedo Tinga, and Mariëlle IA Stoelinga. Automatic inference of fault tree models via multi-objective evolutionary algorithms. *IEEE transactions on dependable and secure computing*, 2021.

[9] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[10] Miles Cranmer, Alvaro Sanchez Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. *Advances in Neural Information Processing Systems*, 33:17429–17442, 2020.

[11] Ming Hu, Jiepin Ding, Min Zhang, Frédéric Mallet, and Mingsong Chen. Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications Using Reinforcement Learning. In *RTSS 2021 - IEEE Real-Time Systems Symposium*, pages 227–239, Dortmund / Virtual, Germany, December 2021. IEEE.

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

[13] Michael Stamateatos, William Vesely, Joanne Dungan, Joseph Fragola, Joseph Minarick III, and Jan Railsback. *Fault tree handbook with aerospace applications*. 2002.

[14] Lisandro Arturo Jimenez-Roa, Matthias Volk, and Mariëlle Stoelinga. Data-driven inference of fault tree models exploiting symmetry and modularization. In Mario Trapp, Francesca Saglietti, Marc Spisländer, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 46–61, Cham, 2022. Springer International Publishing.

[15] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. King's College, Cambridge United Kingdom, 1989.

[16] Bart Verkuil, Carlos E. Budde, and Doina Bucur. Automated fault tree learning from continuous-valued sensor data: a case study on domestic heaters. *CoRR*, abs/2203.07374, 2022.

[17] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[18] Jinwon Yoon, Kyuree Ahn, Jinkyoo Park, and Hwasoo Yeo. Transferable traffic signal control: Reinforcement learning with graph centric state representation. *Transportation Research Part C: Emerging Technologies*, 130:103321, 2021.

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[21] Zhuoran Yang, Yuchen Xie, and Zhaoran Wang. A theoretical analysis of deep q-learning. *CoRR*, abs/1901.00137, 2019.

[22] Laura D'Arcy, Padraig Corcoran, and Alun D. Preece. Deep q-learning for directed acyclic graph generation. *CoRR*, abs/1906.02280, 2019.

[23] C.P. Andriotis and K.G. Papakonstantinou. Managing engineering systems with large state and action spaces through deep reinforcement learning. *Reliability Engineering & System Safety*, 191:106483, 2019.

[24] En-Jui Kuo, Yao-Lung L Fang, and Samuel Yen-Chi Chen. Quantum architecture search via deep reinforcement learning. *arXiv preprint arXiv:2104.07715*, 2021.

[25] Esther Ye and Samuel Yen-Chi Chen. Quantum architecture search via continual reinforcement learning. *arXiv preprint arXiv:2112.05779*, 2021.

[26] Paul Robert and Yves Escoufier. A unifying tool for linear multivariate statistical methods: the rv-coefficient. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 25(3):257–265, 1976.

[27] Pawel Ladosz, Lilian Weng, Minwoo Kim, and Hyondong Oh. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85:1–22, 2022.

[28] Tarik Bakeli and Adil Alaoui Hafidi. Covid-19 infection risk management during construction activities: An approach based on fault tree analysis (fta). *Journal of Emergency Management (Weston, Mass.)*, 18(7):161–176, 2021.

[29] Ayhan Mentes and Ismail H Helvacioglu. An application of fuzzy fault tree analysis for spread mooring systems. *Ocean Engineering*, 38(2-3):285–294, 2011.