Log-Based Behavioral System Model Inference Using Reinforcement Learning

Pandelis Symeonidis
Delft University of Technology
Delft, Netherlands

Mitchell Olsthoorn
Delft University of Technology
Delft, Netherlands

Annibale Panichella
Delft University of Technology
Delft, Netherlands

ABSTRACT

System behavior models are highly useful for the developers of the system as they aid in system comprehension, documentation, and testing. Even though methods to obtain such models exist, e.g. profiling, tracing, source code inference and existing log-based inference methods, they can not successfully be applied to the case of large, real-time systems. Profiling and tracing add overhead which may alter the system's behavior and source code inference does not scale for systems of this magnitude. Existing log-based approaches also suffer due to the intrinsic scalability issues of deriving a minimal model as proven by Gold [13]. In this work, this issue is tackled by applying Reinforcement Learning to the model inference problem. First, an initial model is created from the traces and then Q-Learning is applied to shrink this model into a concise and accurate representation of the system. The approach is evaluated using log traces produced by the XRP Ledger Consensus Protocol [1]. Its effectiveness is assessed based on the accuracy and the conciseness of the inferred models as well as the execution time of the algorithm to infer the model. Results show that the Q-Learning implementation used in this work, is not able to converge to consistent action values. These results might be implementation specific meaning future work should experiment with and extend the current implementation of the algorithm, or due to assumptions made in this work about the underlying systems do not hold. Future work should apply this approach to a different system so as to assess its feasibility.

1 INTRODUCTION

Behavioral system models are becoming an increasingly important tool for software verification, maintenance, and testing. They aid software comprehension [5], anomaly detection [27], test case generation and can help in regression testing [10]. As systems grow in size and complexity, deriving such a model manually becomes increasingly difficult.

Many techniques have been proposed to understand the behavior of a system [2, 5, 28, 30], one of them being source-code model inference in which the system's behavior is modeled by analyzing the software's source code [6]. However, as systems grow larger the source code grows with them making this technique unscalable. Techniques such as profiling and tracing have also been proposed as a way to follow the data flow in the system. However, these techniques do not scale well for large, real-time systems because they instrument the code leading to overhead which could affect the performance and change the behavior of the system. On the contrary, logs are already incorporated in most software systems and hence do not impose additional overhead. On top of that, logs contain semantically useful information about the system providing a more high-level behavioral view of the system. Hence, an approach using logs could solve the shortcomings of some of the existing techniques.

Existing log-based techniques ([20, 30]) perform well, deriving highly accurate models. However, since deriving an exact yet minimal model is proven by Gold to be a NP-complete problem [13], current techniques suffer from scalability issues. Attempts have been made to tackle this using parallelization [31], however a fully scalable approach is yet to be proposed. This work aims to tackle the intrinsic scalability problem of inferring an accurate yet concise model from a system's log traces by approximating the model using AI-based techniques.

Many finite state machine inference techniques belong to the family of "state merging" techniques. In these algorithms, the first step is to infer an initial model describing the input traces. A common approach is using a Prefix Tree Acceptor (PTA) which is the most specific form of FSM accepting all the logs [7]. Then, an algorithm is used to minimize it by merging states until a desirable solution has been attained. In this work, the technique used to merge states is Reinforcement Learning (RL) and specifically Q-Learning (QL) as merging nodes in the initial model can be modeled as a sequential decision process where it has to be decided how to merge a specific region of nodes. The essence of RL is that an agent learns through experience by interacting with their environment, getting positive reinforcement, i.e. a reward, when making a "good" choice, and negative reinforcement when making a "bad" choice. In this case, we attempt to learn what the best way is to merge a region inside the initial model, depending on its topology. That is also the assumption we make about the underlying system, i.e. that this topology is adequate to dictate how the region should be merged.

This research aims to evaluate the novel RL-based model inference approach based on logs produced by the XRP Ledger Consensus Protocol. The XRP Ledger is a distributed, real-time system producing approximately 4 GB of logs every day. The approach is assessed by evaluating the resulting models using accuracy and conciseness-based metrics. On top of that, the scalability of the approach is evaluated by recording the execution time of the approach for different data set sizes.

The initial results are discouraging. The Q-values learned by the algorithm do not converge over time leading to incomplete results. We identify to potential reasons. First, due to the current representation of the state space being insufficient, action values are not able to generalize to the entire model. Second, the assumption made about the nature of the system described above seems to not hold in this particular case study. Further work will thus have to

[1] https://xrpl.org/intro-to-consensus.html

test the assumption on a different system as well as expand on the internal algorithm's implementation. However, the model is able to produce promising intermediately results reaching high levels of compression albeit them being results of random walks through the initial model.

The paper is structured as follows. Section 2 first first gives the definitions of the most important concepts used in this work and then highlights related work. Next, section 3 explains approach proposed. After, section 4 introduces the research questions, the experimental protocol, implementation details, and parameter settings. Section 5 answers the research questions and highlights the most important results. Section 6 discusses the threats to this work's validity, both externally and internally, and section 7 reflects on the ethical aspects of this work including its reproducibility. Finally, section 8 concludes this work and discusses potential future work.

2 BACKGROUND AND RELATED WORK

This section highlights the most important concepts used in this work (2.1) and analyzes existing related work in the topic of model inference based on system traces (2.2)

2.1 Background

The model inference technique presented in this work is based on a set of log traces, $\mathcal{S}$. A *log trace* is a sequence of log statements. A log statement consists of two parts. The first part is a *timestamp*, recording the date and time the log statement was created. The second part is the *log message* which contains information about the logged event (set by the developer). This *log message* consists of a static part called the *log template* which identifies the event using a regular expression, and a dynamic part which consists of information inserted into the template at run-time. For instance, in the log statement: **2020-Mar-01 16:22:46.57 Sending X to Y**, the first part in bold is the *timestamp* and the rest is the *log message*. The **X** and **Y** are dynamic as they may vary and thus the *log template* can be described using the following regular expression: **Sending [A-Z] to [A-Z]**. We define the set of all unique *log templates* of the system as $\mathcal{E}$.

To identify a *log statement* by its *log template* the *Syntax Tree* (ST) is used. The ST is an in-memory prefix tree [14], represented by the 3-tuple $ST = \langle \mathcal{N}, \mathcal{M}, r \rangle$, where $\mathcal{N}$, is the set of nodes in the tree, $\mathcal{M}$, is the set of directed edges in the tree and r is the root of the tree. Each node in the ST holds a regular expression. Each of the leaves of the ST, is associated with a *log template* of $\mathcal{E}$, such that $\langle n_0 n_1 \cdots n_k \rangle = e \rightarrow n_0 = r \land \forall i_{0 \leq i \leq k} \langle n_i, n_{i+1} \rangle \in M$.

The inferred model is represented with a *Finite State Machine* (FSM). A FSM is a model of computation represented by a 5-tuple $\langle Q, \Sigma, \delta, q_0, \mathcal{F} \rangle$, with a set of states Q, an input alphabet Σ (set of symbols), a starting state q_0, a set of final states $\mathcal{F}$ and a state-transition function $\delta : Q \times \Sigma \rightarrow Q$, which describes the transitions in the model. In this work when using the term FSM we assume it is *deterministic*. That means, the range of the transition function δ is always a single state. If it were not deterministic it could return multiple states.

The proposed technique takes as input an initial FSM and attempts to minimize it while keeping it as accurate as possible. The initial model used is the *Unique State Graph* (USG) which is a FSM where each state in Q, represents, i.e. accepts, one or more *log template* $e \in \mathcal{E}$. We describe this as $e \in q$ if the state q represents *log template* e. The USG is defined under the following constraint: $\forall e \in \mathcal{E}, \exists_{\leq 1} q \in Q, (e \in q)$.

To minimize the USG, Reinforcement Learning (RL) is used. RL is an area of AI focused on an agent learning what decisions to make in order to maximize some future cumulative reward [24]. Formally, this problem is modeled as finding the optimal policy of a Markov Decision Process (MDP) [16]. It is defined by a state space, $\mathcal{S}$, an action space, $\mathcal{A}$, and by the "one-step dynamics" of the system defining the resulting states and rewards of specific actions. These are described by a function:

$$p(s', r|s, a) = Pr\{\mathcal{S}_{t+1} = s', \mathcal{R}_{t+1} = r|\mathcal{S}_t = s, \mathcal{A}_t = a\} \quad (1)$$

giving the probabilities of each possible next state and reward while being at a particular state and taking a particular action.

Reinforcement learning algorithms aim at estimating *value functions*, $v_\pi(s)$, which given a state, s estimate how "good" it is to be in this state given a specific *policy*, π. "Goodness" is defined as the expected return when following this policy. A policy is defined as a function giving the probabilities $\pi(a|s)$ of taking action a while being at state s. Intuitively, the policy describes the agent's decision-making. Moreover, we define the action-value function, $q_\pi(s, a)$, giving the expected return when at a specific state, s, and taking action, a, after which continuing using policy π.

Finally, the goal of a reinforcement learning algorithm is to find a policy π that maximizes the reward over the long run. This can be reduced to finding the optimal value function or action-state function as using these functions we know what action is best to take at each state.

2.2 Related Work

Several approaches have been used to infer FSMs from a system's execution traces. One of the first approaches, proposed in 1972 by Biermann and Feldman, is the *k-Tail* algorithm [2]. In this approach, the log traces are parsed into an initial model, the Prefix Tree Acceptor (PTA), representing all possible execution paths. Then, this PTA is minimized by merging states using a heuristic. Namely, to merge two states if they share the same future of length k. Another approach, the *Evidence-Driven State Merging* (EDSM) algorithm, also known as *BlueFringe*, proposed by Lang et al. in 1998 won the Abbadingo competition [18] and was also later used as a benchmark in the STAMINA competition [29]. This work finds equivalent states to merge in the PTA by confining its search to an area close to the root of the PTA and using the heuristic that states with the greatest overlapping suffixes are more likely to be equivalent[29].

Other techniques have been proposed to infer different types of models. For instance, the tool *MINT*, proposed by Walkinshaw in 2016 [30], builds an Extended FSM (EFSM) [4] by combining the EDSM algorithm and a data classifier, where the classifier learns patterns between the data values relating to an event and the events that follow. This work aims to tackle issues of inflexibility and non-determinism in other EFSM inference approaches like *GK-tails*. In short, *GK-tails* proposed by Lorenzoli et al. in 2008 [20], builds on the *k-tails* algorithm where state transitions also hold corresponding data values. When states are merged, the values on each transition become more general.

As Gold pointed out [13], deriving a minimal yet consistent model is an undecidable problem. To deal with this inherent scalability issue of model inference, distributed approaches have been proposed using the Map-Reduce framework [8]. For instance, a distributed version based on *k-tails*, called *k-tails*^{MR}, splits the traces into groups based on the event type and then collects all sub-traces of the same group to create a FSM [31]. However, this approach is very application-specific since the input data needs to be encoded to be used by the Map-Reduce framework. Another approach, *SCALER* [23], creates a FSM for each component of the system and then combines them by making use of dependencies between components as reflected in the traces of the system.

3 APPROACH

The model inference technique follows a "state merging" methodology ([7, 29, 30],) consisting of two parts: creating an initial model of the log traces and then minimizing it to a still accurate but concise representation of the system. This section first discusses how the initial model is obtained (3.1 and then describes the approach used to minimize it using RL (3.2).

3.1 Initial model

The initial model inference takes as input the set of log traces, $\mathcal{S}$, and returns the initial FSM, as described in algorithm 1. With the use of the *Syntax Tree*, the log traces are parsed into the *Unique State Graph* (USG) as defined is section 2.1. Since the USG is by definition unique in terms of the log templates, $\mathcal{E}$, it can have a maximum number of nodes equal to $|\mathcal{E}|$. This greatly assists in tackling the scalability issues of other approaches as the model to be minimized is already highly concise. Moreover, the use of the USG solves the issue of non-determinism in the graph. A non-deterministic graph has various disadvantages relating to its analysis as there is an explosion of possible paths when following a specific sequence [15]. Since, the nodes in the USG are unique, a case of non-determinism is impossible.

Other work uses as an initial model use a Prefix Tree Acceptor (PTA), a tree-shaped automaton that exactly accepts the input log traces, $\mathcal{S}$ [29]. This approach was considered in this study, however, as we aim to tackle the scalability problems of related methods, the extensive magnitude of the PTA discouraged its use. Empirically, a data set of 160 log traces, results in a PTA with 160,322 nodes whereas the corresponding USG has 87 nodes with 157 transitions. Furthermore, the size of the initial DFA used in the "state merging" model is detrimental to the execution time of the approach. Moreover, the use of the PTA resurfaces the issue of non-determinism which, as explained, we strive to avoid. Thus, even though the PTA is the most accurate initial model, its excessive size as well as the scalability and determinism-related qualities lead to the choice of the USG.

It is important for this approach to note the following **assumptions** about the underlying system:

AS.1 Every unique log template originates from a single point in the system's codebase, i.e. each template represents a unique state of the system.

AS.2 The topology of the USG around a specific node encodes enough information to dictate how this region best should be merged.

Algorithm 1: USG parser

```
  input : The Syntax Tree, st, the set of all log traces, S, an
          empty starting node s_0
  output: The inital model (USG)
1  nodeToTemplateMap ← empty mapping
2  model ← empty FSM
3  foreach trace ∈ S do
4  |   node ← s_0
5  |   foreach statement ∈ trace do
6  |   |   template ← syntaxTree.get(statement)
7  |   |   if template ∈ nodeToTemplateMap then
8  |   |   |   existing ← nodeToTemplateMap[template]
9  |   |   |   addEdge(prefixTree, node, existing)
10 |   |   |   node ← existing
11 |   |   else
12 |   |   |   child ← newState(template)
13 |   |   |   nodeToTemplateMap[template] ← child
14 |   |   |   addChild(prefixTree, child, parent)
15 |   |   |   node ← child
16 |   |   end
17 |   end
18 |   return model
19 end
```

3.2 The Approach

Taking as input the Unique State Graph (USG) described in section 3.1, the aim is to condense the model while keeping it an accurate representation of the system. The definition of accurate is not absolute. As Gold [13] and Angluin [1] showed in 1978 obtaining a complete accurate yet miniaml FSM is an undecidable problem. Thus, the actual goal of model inference techniques is to approximate such models, as Lang showed is possible in 1992 [17]. This fact, in combination with the ability of this problem to modeled as a subsequent decision process where an agent decides how to merge nodes in the graph motivates the use of RL in this work.

The big picture of the algorithm is as follows. The agent traverses the graph from its starting node in a depth first manner. At each node, it calculates the state it is in based on the topology of the region around it. The agent observes this state and takes either a random action with a small probability or the best action it has found so far out of a defined set of allowed actions. The possible actions include different ways to merge the current node with its children. After the action has been taken, the resulting model is evaluated and a reward is given trading off accuracy for compression. If it performs a merge that worsens the model (i.e. gets a negative reward) the negative reward is given and the episode is ended. After, the agent restarts from the starting node of the USG and this process is repeated until either the whole model has been

traversed or no progress is observed over a predefined number of time steps.

Specifically, the popular vanilla Q-Learning (QL) algorithm is applied. QL is one of the most popular RL algorithms and is straightforward to implement, a which is crucial given the restricted time frame of this study. QL aims at estimating the optimal action-state function, $q_\pi(s, \alpha)$. It is described by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_\alpha Q(S_{t+1}, \alpha) - Q(S_t, A_t)] \quad (2)$$

This shows that after the agent takes an action A_t at state S_t at time step t, it updates its Q-value based on the immediate reward it receives and the maximum expected return from the next state, S_{t+1}. The actual policy used by the agent can vary, but the most popular is the ϵ-greedy policy ([24]) which is defined as follows: with a probability ϵ, a random action is selected and with a probability $1 - \epsilon$, the best action is chosen based on the Q-values. This way a trade-off between exploration and exploitation exists where the agent can explore new actions in order to learn their Q-values but also exploit already known Q-values to make the best decisions. The value of ϵ is dropped over time as the agent should stop exploring and focus on the best learned values. The general formulation of the algorithm is shown in algorithm 2.

Algorithm 2: Q-learning

```
input: The learning rate, α, the ε-greedy parameter ε, the
       discount factor, γ, the environment, env
1  while stop is False do
2      done ← False
3      state ← reset(env)
4      while done is False and stop is False do
5          actions ← getValidActions(env)
6          random ← rand(0, 1)
7          if random < ε then
8              action ← random action
9          else
10             action ← max_α Q[state, α], a ∈ actions
11         end
12         s′, r, done, stop ← step(env, a)
13         Q[s, a] ← Q[s, a] + α × (r + γ max_α Q(s′, α) − Q[s, a])
14         s ← s‘
15     end
16 end
```

Formally, the environment the agent operates in is defines as follows.

3.2.1 State Space. The state the agent is in depends on the topology of the USG around a specific node. It consists of a combination of the following variables:

- The number of outgoing edges (5 options). A number, 0-4, representing the amount of edges. Option **4**, represents ≥ 4[2]
- Normalized Shannon's entropy (NSE) [22] on the outgoing edges probabilities' (5 options). Using the frequency counts on the USG, the probabilities of following an edge are calculated. Then, the NSE is calculated using equation 3 and discretized into five buckets of length 0.2 representing the 5 options.

$$S = -\frac{1}{log(n)} \sum_{k=1}^{n} p_k * log(p_k) \quad (3)$$

Thus, there are $5 \times 5 = 25$ unique states the agent can be in.

3.2.2 Action Space. The action space spans different ways in which the current node can be merged into its children based on the frequencies on their respective edges. In this approach, we allow only one *type* of merge. This merge is a union between the respective log templates on the nodes. More concretely, if $\delta(s_0, X) = s_1$ and $\delta(s_1, Y) = s_2$ and we merge $s_0 ands_1$, we have a node $s_{0\cup1}$ with $\delta(s_{0\cup1}, X) = s_{0\cup1}$ and $\delta(s_{0\cup1}, Y) = s_2$. The reason behind this decision is two-fold, one it is commonly used in related work [7] and second due to the cost to implement other *types* of merges considering the restricted time-frame of this study.

The complete action space can be seen in table 1. Actions are constricted based on the state the agent is in. For instance a 2-way merge is not possible if the node only has one child. This action-validity matrix can be found in appendix B, in table 5, where a 1 represents an action is allowed and a 0 an action is not allowed.

Table 1: Action space with unique identifiers. Each action represents a way to merge a node with its outgoing children based on the frequencies on their respective edges.

Id	Action
0	Dont merge
1	Merge all children into current node
2	Merge most frequent child
3	Merge second most frequent child
4	Merge third most frequent child
5	Merge two most frequent children
6	Merge two least frequent children
7	Merge two most and least frequent children

3.2.3 Reward. The agent is rewarded after every merge by trading off accuracy for conciseness. Since the initial model is highly accurate as described in section 3.1, the goal is to trade some accuracy for a more concise model. Specifically, after a merge, the resulting model is evaluated based on the model's f1-score, specificity, and compression as explained in detail in section 4.4. Using these metrics the accuracy (α) and the conciseness (c) are computed in terms of the average of the f1-score and specificity and the compression ratio respectively. Thus we have,

$$\alpha = 0.5 \times (2 \times \frac{recall*precision}{recall+precision}) + 0.5 \times specificity$$

$$c = 1 - \frac{\#\ nodes}{\#\ initial\ nodes}$$

[2]The reason behind this is that having a 4 way or more split in code is a rare situation and can practically only be achieved using a *switch* statement which we do not come across often in code.

Then the following cases are identified based on the change in accuracy, $\delta\alpha$, and the change in compression, δc:

- if α < accuracy threshold, reward = $\delta\alpha$
- if $\delta\alpha < 0$, reward = δc
- else if $\delta c > 0$, reward = δc * 10
- else if $\delta c <= 0$, reward = $\delta\alpha$

Intuitively, as long as the accuracy is above a threshold value defined by the user of the algorithm, we reward the agent positively for reducing the accuracy for a gain in compression, analogously to the size of the compression gain. If the action manages to both increase accuracy and compression we give a scaled reward to signify the value of this action. If both accuracy and compression decrease or the threshold is exceeded, a negative reward equal to the loss of accuracy is awarded. The motivation being that the punishment should scale with the degree of decrease in accuracy

3.2.4 State transitions and episodes. Transitions from state to state happen by traversing the USG in a depth first fashion. The agent starts at the starting node of the USG keeping track of a set of visited nodes so as to avoid visiting the same node twice and getting stuck in cycles. It marks the current node as visited, adds all of the outgoing nodes which have not yet been visited to the stack. After performing an action in this state it pops the first node in the stack and sets that as the current node. However, if the action resulted in merging an unvisited node into the current one, we keep this node as our current one as it has connections to other unvisited nodes. Once a negative reward is encountered, the episode is ended. The reason behind this is we want to encourage the agent to traverse the model without performing merges that will negatively impact the model. Thus, once the whole USG has been traversed without receiving a negative reward we stop the algorithm and take the model produced by this traversal as the final model.

3.2.5 Algorithm hyper-parameters. The QL algorithm consists of multiple parameters, namely:

- The learning rate, α, dictates how much the new value is accepted vs the old one. So, a learning rate of 0 means we do not learn anything new and keep the old value and the larger α becomes, the more weight is given to the new value
- The discount factor, γ, dictates how much importance is given to future rewards. A discount factor of 0 means no importance is given to the future rewards at all and the agent learns to take actions that maximize its immediate reward.
- The value of ϵ, dictates the exploration vs exploitation trade-off. It gives the probability the agent will explore, by picking a random action, versus being greedy and picking the best action.

4 EMPIRICAL STUDY

This section discusses the details of the empirical study starting by presenting the research questions this study aims to answer (4.1), the benchmark used (4.2), implementation details and parameter settings (4.3) and details about the experimental protocol followed (4.4).

4.1 Research Questions

The goal of this empirical study is to evaluate the effectiveness of the reinforcement learning-based approach in deriving a concise and accurate state model from a set of log traces. On top of that, the scalability of this approach is assessed. The research questions are as follows.

RQ1. How accurate and how concise is the model inferred by the RL-based technique?

RQ2. How well does the RL-based algorithm scale?

4.2 Benchmark

This work makes use of logs produced by the XRP ledger, a real-time system producing approximately 100GB of log data every day, across all components and all levels. However, we focus on the Consensus Protocol, a major component of the system. This component is very well documented, produced an abundance of logs, and has been used in previous studies [21] where it was modeled both empirically and theoretically. This allows us to compare the models produced in this approach with the theoretical counterpart and evaluate the results of the approach.

4.3 Implementation and parameter settings

The approach was implemented as a Python 3.8 program and is publicly available at [26]. The reinforcement learning implementation is done using the OpenAI gym library [3], the de facto standard library in reinforcement learning research. OpenAI gym is a framework aiming at standardizing RL research, consisting of multiple standardized environments and agents but also providing the ability to fully customize the approach.

As far as the hyper-parameter settings are concerned, their values are shown in table 2. These values are based on related work ([11]) and accepted standard values in literature. The time frame of this study did not allow for careful fine-tuning of these parameters however these values even though not fine-tuned still produce acceptable results and future work could look into optimizing them.

Table 2: Model hyper-parameters

Parameter	Value
α	0.2
ϵ	0.3 decayed to 0.1
γ	0.7
accuracy threshold	0.9
k (KFCV)	5

4.4 Experimental Protocol

The research questions are answered by evaluating the models produced by the approach using the metrics introduced in section 4.4.1 and by evaluating the scalability of the approach as explained in section 4.4.2.

4.4.1 Accuracy and conciseness. To answer RQ1, we evaluate the model based on its accuracy and conciseness. To evaluate the model's accuracy, a data set was created consisting of training

and test traces following a 4:1 split. Concretely, 800 traces with an average of 800 statements each, were picked randomly as a training set. The test set consists of 100 positive samples and 100 negative samples. The 100 positive samples consist of 100 additional log traces produced by the Consensus Protocol which the model should accept. The 100 negative samples consist of 100 negative traces, i.e. log traces the model should reject. To acquire negative samples, we adopt a technique used in related work ([23, 29]) in which existing traces are mutated. Specifically, we apply three different types of mutations:

- Remove a random log statement
- Swap two consecutive log statements
- Swap two random log statements

To verify the mutated trace is indeed an incorrect one, we use the initial model. If it accepts the trace, the trace is actually correct. Hence, mutations are applied until it is rejected.

Accuracy is evaluated following related work [9, 30] based on recall, specificity, precision, and the f1-score. To evaluate the model the test traces are used as input to the model, leading to one of the following results: True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN).

Based on these classes the metrics are calculated as follows:

- Recall: How many of the positive traces are accepted

$$Recall = \frac{\#TP}{\#TP + \#FN}$$

- Specificity: How many of the negative traces are rejected

$$Specificity = \frac{\#TN}{\#TN + \#FP}$$

- Precision: How many of the accepted traces are actually positive

$$Precision = \frac{\#TP}{\#TP + \#FP}$$

As far as conciseness is concerned, the model is evaluated based on three metrics:

- Compression ratio

$$1 - \frac{Compressed\ size}{Uncompressed\ size} = 1 - \frac{\#\ nodes}{\#\ inital\ nodes}$$

- Number of nodes
- Number of transitions

To calculate the metrics, k-fold cross validation (KFCV) is used [12]. This method, also used in other model inference studies ([30]), splits the data set into k partitions. The k-1 folds are used as a training set and the remaining fold as a test set. This is repeated k times, until all folds have been used as the test set, hence making efficient use of the whole data set. The metrics are then calculated by averaging over the k folds.

4.4.2 Scalability. To evaluate the scalability of the approach, we recorded the running time of the algorithm on different sized data sets. In particular, 5 data sets were created ranging from 100 to 1000 traces with an interval of 200 following a 4:1 split into training and test sets. KFCV was then used on each data set and the average run time across the k folds was calculated. All experiments were run on a Windows machine with an Intel(R) Core(TM) i7-8750H CPU at 2.20GHz with 16 GB of RAM.

5 RESULTS

In this section, we report and analyze the results of the study by answering the research questions presented in section 4.1, namely about the accuracy and conciseness of the inferred model in section 5.1 and the scalability of the RL-based approach in section 5.2.

5.1 Accuracy and conciseness

The amount of time steps completed in each episode of the QL algorithm across episodes is plotted in figure 1. As seen in the graph, the amount of steps randomly fluctuates around approximately 40 steps. The algorithm did not manage to converge and stopped due to not making any progress across 10 episodes. This indicates that the agent was not able to traverse the entire USG. These observations can be attributed to two factors.

Firstly, these results suggest that using the current state representation, as explained in section 3.2.1, the action values can not generalize to the whole model and thus the agent is not able to traverse the whole graph. The 25 unique states, used to model the topology around a node based on its outgoing edges and their frequencies, appear to be too broad and thus the algorithm is unable to converge to one specific action being optimal for each state.

Secondly, assumption **AS.2**, regarding the topology of the USG, appears to not hold in this case. If it did hold, the agent would have been able to learn the optimal action for each state in the graph and use it to traverse the USG. However, since it is not able to perform a complete traversal this points to the assumption not holding for the underlying system.

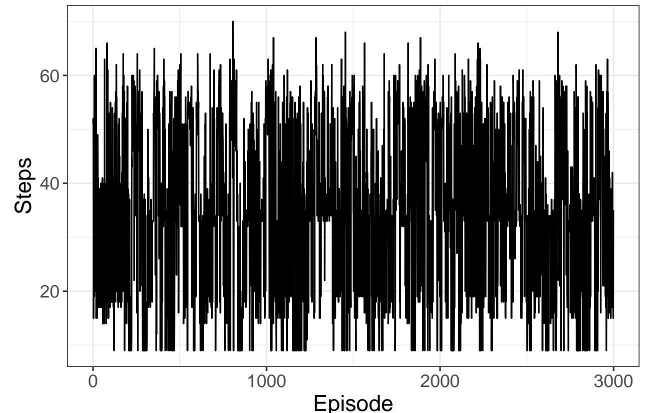

Figure 1: Amount of time steps until the episode is ended plotted across all episodes.

Table 3, shows some of the most important results gathered during the execution of the approach. As visible in the table, the initial model, i.e. the USG, is highly accurate with a recall of 0.98 and specificity of 1. As far as conciseness is concerned, it consists of an average of 102 nodes and 200 transitions. Moreover, we have collected the best, worst, and average model produced across the run of the algorithm. This "goodness" is described in terms of the number of time steps completed in the episode resulting in the corresponding model. Since the goal of the approach is to manage a complete traversal through the USG, the number of time steps

Table 3: Evaluation of recall, specificity, precision, compression and number of nodes and transitions for the best, worst and average model produced by the approach.

Model	Recall	Specificity	Precision	Compression	Nodes	Transitions
initial (USG)	0.98	1.00	1.00	0.00	102	200
best	0.99	0.84	0.86	0.56	44	100
worst	0.99	0.81	0.84	0.08	94	167
average	0.99	0.85	0.87	0.34	67	131

completed is analogous to the percentage of the graph traversed. As can be seen in the table, the compression improves with the traversal length reaching a maximum of 56% with 44 nodes and 100 transitions. Also, we notice that the recall is constantly high, whereas specificity is the measure that varies the most. This can be explained as follows. As the recall in the USG is already at a value of 0.98 and the merges performed during the approach only make the model more general, all positive traces accepted by the USG will always be accepted, hence it can only increase. However, as merges are completed and the model generalizes, more negative traces might get accepted that were previously rejected.

Furthermore, it is worth pointing out some other interesting results. In table 6 found in the appendix, we highlight some runs of the algorithm that did manage to converge. These mostly apply to models based on small a data set (100 traces) since they have a simpler topology and thus can be traversed with more ease by the agent. One such run managed to condense the initial USG by 91% while keeping both recall and specificity at 90% which is a promising result. Another run on a data set of 600 traces managed to converge in only 3 episodes condensing the initial model by 69% and achieving a value of 1 for recall and 0.88 for specificity.

Given these results, it can be concluded that the reward function manages to successfully trade off accuracy for an increase in the conciseness of the model. However, since the action values are not able to converge, these results are most likely constructs of almost random traversals and merges through the graph managing to get to the end without negatively affecting the model. Thus, the results in table 6 are mostly the product of random walks through the USG and not based on meaningful information priorly learned.

We conclude that,

> **RQ1.** Under the current state representation, assumption **AS.2** does not hold in this case and thus we cannot generalize the actions for the whole model. However, the reward function seems to guide the agent to successfully trade off accuracy for conciseness leading to promising, although randomly generated, final models.

5.2 Scalability

Figure 2 displays the execution time of the approach based on varying data set sizes. As visible, the correlation between the two variables seems to be linear despite one outlier. This decrease of the runtime at the 800 trace data set could suggest that the runtime of the algorithm is related to the topology of the *USG*. Each data set produces a similar (in terms of nodes) yet different (in terms of connections) *USG* and the convergence of the approach depends on the existence of patterns in the topology of the graph.

Nevertheless, a linear relationship would coincide with the theoretical complexity analysis of the approach. As the approach takes as input the USG, which node count is capped by the unique log templates produced by the system, the input model is practically the same for each run of the algorithm. The only thing affecting its execution time is the model evaluation after each action. This part passes the test set (positive and negative traces) through the resulting model and classifies them accordingly. These classes are then used to compute the metrics described in the experimental protocol (section 4.4). Thus, the evaluation's time complexity is $O(n)$ where n is the size of the data set. Hence, the whole approach, theoretically, also scales linearly with the size of this data set.

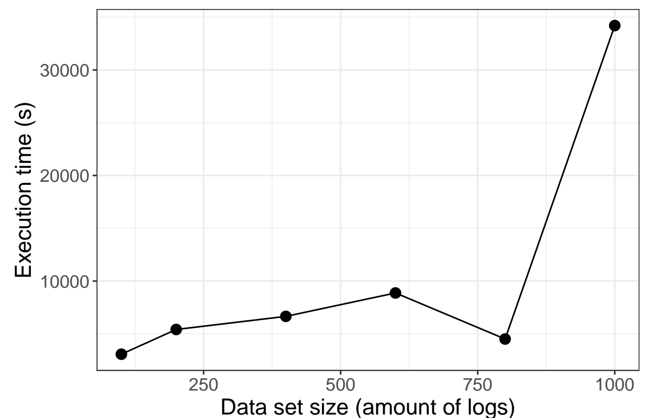

Figure 2: Approach execution time in seconds for different data set sizes.

However, it has to be noted that as these execution time values are the result of KFCV, the runs either terminate because of the algorithm converging or due to the algorithm not converging and the stopping condition being reached. A high amount of variance was observed for the k different runs. Given that the majority of runs do not converge (as seen in table 6), this could be attributed to the randomness associated with the learning phase each run can take arbitrarily long to reach the stopping condition. This leads us to conclude that the results about the scalability of the approach are inconclusive and should be further examined. We conclude that,

> **RQ2.** The results are inconclusive due to the algorithm not converging. However, initial results seem to depict a linear relationship which agrees with our theoretical complexity analysis.

6 THREATS TO VALIDITY

This section outlines the threats to the validity of this work, externally (6.1) and internally (6.2), and how we mitigate them in this work.

6.1 Threats to external validity

Threats to the external validity of this study refer to the generalizability of this work. The study is based on a specific case study, the Consensus Protocol of the XRP Ledger. On top of this, we make an assumption based on this specific system, **AS.2**. As discussed in the results (5), this assumption does not seem to hold in this specific case. However, we cannot draw a general conclusion as the assumption is possible to still hold for other systems. Future work will, thus, need to test this hypothesis.

A second assumption is made, namely **AS.1**. In this case, this assumption was manually tested by examining the source code. However, this assumption might not hold for other systems or may not be able to be tested due to there being no access to the source code. In that case, the *USG*, used as an initial model,will not be a representative model of the underlying system and thus this approach will not hold.

6.2 Threats to internal validity

Threats to the internal validity of this study include threats to the quality of the evaluation performed. First, an argument could be made about the choice of metrics used to evaluate the inferred models. For accuracy, we choose the f1-score, recall, specificity, and precision following related work [19, 30] and DFA inference competitions (Abbadingo [18] and STAMINA [29]). For conciseness, again following related work ([23, 30]), we evaluate the number of nodes and the number of transitions as well as the amount by which the model has been compressed.

Moreover, a threat to the internal validity could be the choice of the data set used in the work. Firstly, the traces were acquired from an official node of the XRP ledger. Secondly, regarding the size of the data set, doubts could be raised if it is large enough to evaluate the scalability of the approach. Comparing to alternative approaches ([23]) which use a maximum of 35K log statements, we evaluate on significantly more traces. Namely 800 traces each consisting of an average of 800 statements, thus 640K statements in total.

7 RESPONSIBLE RESEARCH

Traditional ethical concerns like conflicts of interest or human research subjects do not apply in this case because of the nature of the work. However, a crucial aspect to be considered is its reproducibility. To adhere to the scientific method and to ensure work is credible it must be reproducible. Without reproducibility, the work loses its credibility as it cannot be validated by the scientific community. For this reason, a lot of emphases was put on making sure this work is fully reproducible. Firstly, the implementation of the algorithm presented is publicly available [26]. Moreover, the data set used in the study is also publicly available [25] and the values of the parameters used are cited in table 2. This way, anyone who wants to validate the results by running the algorithm or experiment with the work itself is able to do so. All external requirements with their specific required versions are documented and extensive documentation has been written, both inside the code and as an external document which is all available with the code.

Furthermore, due to the nature of RL algorithms, this work contains the element of randomness. Specifically, when the agent is exploring it is randomly picking an action. On top of this, when evaluating the inferred model using k-fold cross validation, randomness is involved in splitting the data set. To cope with statistical variation due to this randomness all results presented in this work are averages taken from various independent runs of the algorithm. Moreover, for this work to be fully reproducible the option to seed the random generator is given and the seed used to achieve the results in this work is also given, meaning the random number generator will produce the exact sequence of numbers every time ran. Specifically, the seeds used can be found in the appendix in table 4.

8 CONCLUSIONS AND FUTURE WORK

This paper aims to present and evaluate a novel Reinforcement learning-based model inference algorithm based on a system's log traces. The principal motivation behind this approach is to attempt to tackle the intrinsic scalability issue of deriving a consistent FSM. Hence, an AI-based approach is proposed which infers a model by first parsing the traces into an initial model and minimizing it. The approach implements the popular Q-Learning algorithm by traversing the initial model and at each node, deciding how to merge, if at all, the specific region based on its topology, i.e. number of states and edge frequencies. After taking action, the resulting model is evaluated and a reward is given to the agent for successfully trading off accuracy for conciseness. If the reward is negative the process is restarted until the action values learned can generalize to the whole model and thus complete a traversal.

The empirical evaluation of the final models is based on a data set of 1000 traces (800 statements each) produced by the XRP Ledger Consensus Protocol. Results show that using the state representation used in this work the action values are not able to generalize to the whole model and thus the algorithm does not manage to traverse the whole initial model. This is, probably, because of the assumption made throughout this work, i.e. that the topology of the graph around a node contains the information necessary to dictate how to merge this region, does not hold in this case. However, throughout each attempt to traverse the initial model, the reward function guides the agent successfully trading of the accuracy of conciseness and although randomly generated, providing promising results. As far as the scalability of the approach is concerned, since the algorithm does not converge, no definite conclusions can be drawn. Theoretically, however, because of the choice of using the Unique State Graph as the initial model, meaning the size of the initial model remains constant, the execution time of the approach should scale linearly with the test set used to evaluate the model.

Future work should look into applying this approach to other systems, to examine if the assumption mentioned holds and leads to more promising results. On top of this, the state space should be experimented with by enriching it with more information such as the number of incoming edges and frequencies of incoming edges. However, it is important to point out that there is a tradeoff between

the size of the state space and the learning time of the approach. The larger the state space, the more action values will have to be learned leading to additional learning time.

Moreover, in this study, only a particular kind of merge was executed. Future work could attempt to include other types of merges, for instance, merging nodes A and B leads to a single node, C, accepting exactly the sequence A → B. Additionally, other types of actions could be considered. For example, completely removing a specific branch or merging a set of children into one child.

Finally, due to the restricted time frame of the work, we were not able to optimize the hyper-parameters of the model and followed related work as well as standard values used in the literature. Thus, future work could look into fine-tuning the Q-Learning hyperparameters as well as the accuracy threshold set in the reward function.

REFERENCES

[1] Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and control* 39, 3 (1978), 337–350.

[2] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers* 100, 6 (1972), 592–597.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[4] Kwang-Ting Cheng and Avinash S Krishnakumar. 1993. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*. IEEE, 86–91.

[5] Jonathan E. Cook and Alexander L. Wolf. 1998. Discovering Models of Software Processes from Event-Based Data. *ACM Trans. Softw. Eng. Methodol.* 7, 3 (July 1998), 215–249. https://doi.org/10.1145/287000.287001

[6] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 439–448.

[7] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel Van Lamsweerde. 2005. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering* 31, 12 (2005), 1056–1073.

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[9] Seyedeh Sepideh Emam and James Miller. 2018. Inferring extended probabilistic finite-state automaton models from software executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 1 (2018), 1–39.

[10] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

[11] Eyal Even-Dar, Yishay Mansour, and Peter Bartlett. 2003. Learning Rates for Q-learning. *Journal of machine learning Research* 5, 1 (2003).

[12] Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[13] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320.

[14] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. 2014. *Data structures and algorithms in Java*. John Wiley & Sons.

[15] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. 2009. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)* 41, 2 (2009), 1–76.

[16] Ronald A Howard. 1960. Dynamic programming and markov processes. (1960).

[17] Kevin J Lang. 1992. Random DFA's can be approximately learned from sparse uniform examples. In *Proceedings of the fifth annual workshop on Computational learning theory*. 45–52.

[18] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*. Springer, 1–12.

[19] David Lo, Leonardo Mariani, and Mauro Santoro. 2012. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software* 85, 9 (2012), 2063–2076.

[20] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*. 501–510.

[21] M. Roelvink. 2020. Log inference on the Ripple Protocol: testing the system with an empirical approach. (2020).

[22] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[23] Donghwan Shin, Salma Messaoudi, Domenico Bianculli, Annibale Panichella, Lionel Briand, and Raimondas Sasnauskas. 2019. Scalable inference of system-level models from component logs. *arXiv preprint arXiv:1908.02329* (2019).

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

[25] Pandelis Symeonidis, Tommaso Brandirali, Calin Georgescu, and Thomas Werthenbach. 2021. *XRP Ledger Consensus Protocol Traces Dataset*. https://doi.org/10.5281/zenodo.5090200

[26] Pandelis Symeonidis, TommasoBrandirali, Thomas Werthenbach, and Calin Georgescu. 2021. *Pandelissym/WhatTheLog: Log-Based Behavioral System Model Inference Using Reinforcement Learning*. https://doi.org/10.5281/zenodo.5090175

[27] Alfonso Valdes and Keith Skinner. 2000. Adaptive, model-based monitoring for cyber attack detection. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 80–93.

[28] Neil Walkinshaw, John Derrick, and Qiang Guo. 2009. Iterative refinement of reverse-engineered models by model-based testing. In *International Symposium on Formal Methods*. Springer, 305–320.

[29] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. 2013. STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering* 18, 4 (2013), 791–824.

[30] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.

[31] Shaowei Wang, David Lo, Lingxiao Jiang, Shahar Maoz, and Aditya Budi. 2015. Scalable parallelization of specification mining using distributed computing. In *The Art and Science of Analyzing Software Data*. Elsevier, 623–648.

A SEED VALUES

In table 4 we note all seeds used throughout this study to ensure full reproducibility.

Table 4: Seeds used.

run	seed
KFCV data set creation	0
KFCV (fold 1)	0
KFCV (fold 1)	1
KFCV (fold 1)	2
KFCV (fold 1)	3
KFCV (fold 1)	4

B ACTION SPACE

Table 5 highlights the allowed actions at each particular state. Next to the unique identifier of each state we include the decoded (human readable) version of the state. This is a concatenation of the two variables making up the state space: the number of outgoing edges and the entropy of the outgoing edges' probabilities. We repeat all possible actions the agent can take in table 1.

Table 1: Action space with unique identifiers. Each action represents a way to merge a node with its outgoing children based on the frequencies on their respective edges (repeated from page 4).

Id	Action
0	Dont merge
1	Merge all children into current node
2	Merge most frequent child
3	Merge second most frequent child
4	Merge third most frequent child
5	Merge two most frequent children
6	Merge two least frequent children
7	Merge two most and least frequent children

Table 5: Action validity matrix. A value of 1 symbolizes an action is allowed in a specific state whereas a 0 symbolized said action is not allowed.

	Actions							
State (decoded[a])	0	1	2	3	4	5	6	7
0 (0.0)	1	0	0	0	0	0	0	0
1 (0.1)	1	0	0	0	0	0	0	0
2 (0.2)	1	0	0	0	0	0	0	0
3 (0.3)	1	0	0	0	0	0	0	0
4 (0.4)	1	0	0	0	0	0	0	0
5 (1.0)	1	0	1	0	0	0	0	0
6 (1.1)	1	0	1	0	0	0	0	0
7 (1.2)	1	0	1	0	0	0	0	0
8 (1.3)	1	0	1	0	0	0	0	0
9 (1.4)	1	0	1	0	0	0	0	0
10 (2.0)	1	0	1	0	0	0	0	0
11 (2.1)	1	1	1	1	0	0	0	0
12 (2.2)	1	1	1	1	0	0	0	0
13 (2.3)	1	1	1	1	0	0	0	0
14 (2.4)	1	1	1	1	0	0	0	0
15 (3.0)	1	0	1	0	0	0	0	0
16 (3.1)	1	1	1	1	1	1	1	1
17 (3.2)	1	1	1	1	1	1	1	1
18 (3.3)	1	1	1	1	1	1	1	1
19 (3.4)	1	1	1	1	1	1	1	1
20 (4.0)	1	0	1	0	0	0	0	0
21 (4.1)	1	0	0	0	0	0	0	0
22 (4.2)	1	0	0	0	0	0	0	0
23 (4.3)	1	0	0	0	0	0	0	0
24 (4.4)	1	0	0	0	0	0	0	0

[a] The decoded states indicate the two components of the state space. The first part is the number of outgoing edges and the second the entropy of the outgoing edges' probabilities

C ADDITIONAL RESULTS

Table 6 lists the complete results gathered throughout the evaluation process. We underline the runs of the algorithm that managed to converge by completing a traversal through the *USG*. Moreover, we highlight other interesting results.

Table 6: Additional results. Underlined episodes signify converged runs of the algorithm. Highlighted cells point out other interesting results.

Data set size	Seed	Episodes	Accuracy			Conciseness			Duration
			Recall	Specificity	Precision	Compression	Nodes	Transitions	
100	0	3000	0.90	0.80	0.82	0.42	56	101	6732
100	1	182	0.90	0.90	0.90	0.91	8	23	564
100	2	2551	0.90	0.90	0.90	0.84	15	38	7804
100	3	56	1.00	0.90	0.91	0.85	15	42	237
100	4	13	0.90	0.90	0.90	0.82	17	45	18
400	0	507	1.00	0.82	0.85	0.18	89	170	3963
400	1	1172	0.98	0.85	0.87	0.14	93	176	11738
400	2	1635	0.95	0.88	0.88	0.30	76	151	14146
400	3	439	1.00	0.75	0.80	0.16	92	177	3031
400	4	84	0.92	0.78	0.80	0.18	80	136	393
600	0	361	0.98	0.82	0.84	0.21	84	161	3649
600	1	3000	0.95	0.87	0.88	0.36	70	152	37569
600	2	120	1.00	0.77	0.81	0.16	92	168	941
600	3	253	1.00	0.67	0.75	0.19	82	139	2150
600	4	3	1.00	0.88	0.90	0.69	34	79	51
800	0	70	0.99	0.80	0.83	0.13	85	167	1005
800	1	550	0.99	0.81	0.84	0.17	80	149	6693
800	2	586	0.98	0.80	0.83	0.17	82	149	6055
800	3	464	1.00	0.86	0.88	0.33	65	116	5763
800	4	178	0.99	0.84	0.86	0.15	83	157	3033
1000	0	348	1.00	0.80	0.83	0.07	95	168	7944
1000	1	66	1.00	0.85	0.87	0.08	94	166	858
1000	2	1627	0.99	0.82	0.85	0.08	94	167	31747
1000	3	85	0.98	0.80	0.83	0.08	94	167	1185
1000	4	1538	0.98	0.76	0.80	0.08	94	167	31170