# FastNFA: A High-Performance FPRAS Implementation for the #NFA Problem

Alberto Heissl[1⋆] ⓡ Vinh-Hien D. Huynh[1⋆] ⓡ Juan Guevara[1⋆], Kuldeep S. Meel[2], and Timothy van Bremen[1]

[1] Nanyang Technological University, Singapore
alberto.heissl@studenti.unipd.it, hien@u.nus.edu, juan.guevara@mbzuai.ac.ae, timothy.vanbremen@ntu.edu.sg
[2] Georgia Institute of Technology, USA
meel@gatech.edu

**Abstract.** The #NFA problem asks for the number of words of a fixed length $n$ accepted by a non-deterministic finite automaton. We present FastNFA, a practical tool for solving the #NFA problem at scale in practice. FastNFA computes randomized approximations for the #NFA problem, and comes with rigorous $(\epsilon, \delta)$-guarantees on the accuracy of the estimates produced. Our experimental results show that FastNFA scales well beyond naïve implementations of existing theoretical algorithms, solving a full order of magnitude more benchmarks within a fixed timeout.

**Keywords:** non-deterministic finite automata · constrained counting · randomized algorithms

## 1 Introduction

In this paper, we introduce FastNFA, a novel tool for solving the following problem in practice:

> **#NFA**: given a non-deterministic finite automaton (NFA) $\mathcal{A} = (\mathcal{Q}, q_I, q_F, \mathcal{T})$ with $m$ states, and a number $n \in \mathbb{Z}$, determine $|\mathcal{L}_n(\mathcal{A})|$, that is, the number of words of length $n$ in the language accepted by $\mathcal{A}$.

The #NFA problem is a fundamental problem in computer science, since an NFA is arguably *the* canonical succinct representation of a regular language. From a theoretical perspective, the #NFA problem is known to be #P-complete; more specifically, it lies in the SpanL-complete complexity class [2]. In particular, this implies that many other natural problems can be reduced to #NFA, for example, counting satisfying assignments of a Boolean formula in disjunctive normal form (DNF), counting answers to regular path queries (RPQs) on graph databases,

⋆ Alberto Heissl, Vinh-Hien D. Huynh, and Juan Guevara contributed equally to this research. The symbol ⓡ denotes random author order. The publicly verifiable record of the randomization is available at https://www.aeaweb.org/journals/random-author-order evaluating RPQs on probabilistic graphs [3], and counting satisfying assignments of non-deterministic ordered binary decision diagrams (nOBDDs) [4], among others.

Recent years have seen significant progress on randomized approximation algorithms for the #NFA problem. In a breakthrough result, Arenas et al. [4] showed the existence of a *fully polynomial-time randomized approximation scheme* (FPRAS) for the #NFA problem; i.e., a polynomial-time randomized approximation algorithm returning the true count within a $(1 \pm \varepsilon)$-multiplicative threshold with high probability. However, the time complexity of their original algorithm ($\mathcal{O}(m^{17}n^{17}\varepsilon^{-14})$, up to poly-logarithmic factors), is untenably high for practical implementation. A growing line of work has focused on bringing down this complexity to the current best bound of $\mathcal{O}(m^3 n^2 \log(mn)\epsilon^{-2})$ [7, 8]; but until now no algorithm has been amenable to implementation in code due to impractically large constants.

In this paper, we describe the design and implementation[3] of FastNFA, the first algorithm capable of solving #NFA (and more generally, any problem in SpanL) at a large scale in practice. We do this by observing and addressing several shortfalls in the current best-known theoretical FPRAS for the #NFA problem, known as countNFA [8]. First, we observe that the internal parameters used in countNFA can be tightened for improved performance while retaining theoretical guarantees (Section 4.1). Secondly, we identify and exploit a *parallelism-performance tradeoff* (Section 4.2), where the expected runtime of individual runs of a key subroutine of the algorithm can be reduced at the price of requiring more (parallel) runs, and vice versa. Lastly, motivated by the observation that many NFAs contain only limited nondeterminism in practice, we introduce an optimization (Section 4.3) that avoids unnecessary computation for states that occur only as part of "deterministic lines", i.e., runs of the NFA without nondeterminism.

The effectiveness of the above improvements incorporated into FastNFA is supported by comprehensive experimental results presented in Section 5. In particular, we run an ablation study that sheds light on the impact of each individual optimization. Our results confirm a significant impact; in particular, FastNFA solves over 10× as many benchmarks within a 300 second timeout as a naïve implementation of the original countNFA algorithm (Figure 2).

## 2 Notation and Preliminaries

We write $[n]$ to denote the set $\{1, 2, \dots, n\}$. For $a$, $b$ and $\varepsilon$ three non-negative real numbers, we use $a \in (1 \pm \varepsilon)b$ to denote $(1-\varepsilon)b \leq a \leq (1+\varepsilon)b$; similarly, $a \in b(1 \pm \varepsilon)^{-1}$ stands for $\frac{b}{1+\varepsilon} \leq a \leq \frac{b}{1-\varepsilon}$ (with $\frac{b}{0}$ defined as $\infty$ when $b \neq 0$, and as 0 when $b = 0$). A *word* $w$ over an alphabet $\Sigma$ is a sequence $(w_1 \dots w_k)$ of length $|w| = k$ with each $w_i \in \Sigma$. $\lambda$ is denoted as the empty word (of length 0). We further denote by $\Sigma^*$ the set of all words (language) over $\Sigma$. For $w, w' \in \Sigma^*$, we denote by $w \cdot w'$ their concatenation. For $S \subseteq \Sigma^*$, we write $S \cdot w = \{w' \cdot w \mid w' \in S\}$.

[3] We will release the code upon final acceptance and publication of this paper.

*FPRAS* For a problem that, given an input $x$ of size $s$, to count a number $N(x) \in \mathbb{R}^+$, a *fully polynomial-time randomized approximation scheme* (FPRAS) is an algorithm that, given $x$, $\varepsilon > 0$, and $0 < \delta < 1$, runs in time polynomial in $s$, $1/\varepsilon$ and $\log(1/\delta)$, and returns an estimate $\tilde{N}$ with the guarantee that $\Pr\left[\tilde{N} \in (1 \pm \varepsilon)N(x)\right] \geq 1 - \delta$.

*NFA* A *non-deterministic finite automaton* (NFA) is a tuple $\mathcal{A} = (\mathcal{Q}, q_I, q_F, \mathcal{T})$ where $\mathcal{Q}$ is a finite set of states, $q_I \in \mathcal{Q}$ is the initial state, $\mathcal{T} \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a transition relation, and $q_F \in \mathcal{Q}$ is the final state. This research mainly examines the case when $\Sigma = \{0, 1\}$, though our algorithm is expandable to larger alphabet size. We assume a total order $\prec$ on the states. A *run* of $w$ on $\mathcal{A}$ is a sequence $\rho = (q_0, \ldots, q_{|w|})$ such that $q_0 = q_I$, $q_{|w|} = q_F$ and, for every $i < |w|$, $(q_i, w_{i+1}, q_{i+1}) \in \mathcal{T}$. A word $w$ is *accepted by* $q \in \mathcal{Q}$, when there is a run of $w$ on $\mathcal{A}$ that ends on $q$. The *language* of a state $q$, denoted $\mathcal{L}(q)$, is the set of words accepted by $q$, and the language of $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \mathcal{L}(q_F)$. For $n \in \mathbb{N}$, we define $\mathcal{L}_n(\mathcal{A})$ as the set of words of length $n$ in $\mathcal{L}(\mathcal{A})$. For $q \in \mathcal{Q}$ and $b \in \Sigma$, the *b-predecessors* of $q$ are given by a sequence $\mathsf{pred}(q, b) = (q' \mid (q', b, q) \in \mathcal{T})$ ordered consistently with $\prec$. We also define $\mathsf{pred}(q) = (q' \mid (q', 0, q) \in \mathcal{T} \text{ or } (q', 1, q) \in \mathcal{T})$, again with states ordered according to $\prec$.

*Unrolling* Given an NFA $\mathcal{A} = (\mathcal{Q}, q_I, q_F, \mathcal{T})$ and $n \in \mathbb{N}$, one may construct the *unrolled NFA* $\mathcal{A}^{\mathsf{u}}_n$ whose language is $\mathcal{L}_n(\mathcal{A})$ in time $\mathcal{O}(n|\mathcal{T}|)$. One can check whether $\mathcal{L}_n(\mathcal{A})$ is empty in time $\mathcal{O}(n|\mathcal{T}|)$; we hence assume $\mathcal{L}_n(\mathcal{A}) \neq \emptyset$. Since $n$ will be fixed, we will just write $\mathcal{A}^{\mathsf{u}} = (\mathcal{Q}^{\mathsf{u}}, q_I^0, q_F^n, \mathcal{T}^{\mathsf{u}})$. We now describe the construction of $\mathcal{A}^{\mathsf{u}}$. For each state $q \in \mathcal{Q}$ and $0 \leq \ell \leq n$, if $q$ is reachable in $\mathcal{A}$ by *some* word of length $\ell$, then $\mathcal{Q}^{\mathsf{u}}$ contains a state $q^\ell$, and $\mathcal{T}^{\mathsf{u}}$ ensures that $q^\ell$ is reachable by precisely *all* words of length $\ell$ that reach $q$ in $\mathcal{A}$. Formally, $q_I^0$ is in $\mathcal{Q}^{\mathsf{u}}$ and, for every $0 \leq \ell < n$, if $(q_1, b, q_2)$ is in $\mathcal{T}$ and $q_1^\ell$ is in $\mathcal{Q}^{\mathsf{u}}$, then $q_2^{\ell+1}$ is in $\mathcal{Q}^{\mathsf{u}}$ and $(q_1^\ell, b, q_2^{\ell+1})$ is in $\mathcal{T}^{\mathsf{u}}$. We write $\mathcal{Q}^\ell = \{q^\ell \mid q \in \mathcal{Q} \text{ and } q \text{ is reachable by words of length } \ell\}$ and $\mathcal{Q}^{<\ell} = \mathcal{Q}^0 \cup \cdots \cup \mathcal{Q}^{\ell-1}$ and similarly for $\mathcal{Q}^{\leq\ell}$, $\mathcal{Q}^{>\ell}$ and $\mathcal{Q}^{\geq\ell}$. Note that $\mathcal{Q}^0$ contains only $q_I^0$. Graphically, we see unrolled automata as directed acyclic graphs (DAGs) where the states are vertices and the transitions are edges labeled by the transition symbol $b$. We will again assume some total order $\prec$ on the states of $\mathcal{A}^{\mathsf{u}}$. Since in this paper we work only with unrolled NFA, we drop the superscript from state names.

*Derivation run* Let $w \in \mathcal{L}(q)$. The derivation run $\mathsf{run}(w, q)$ is defined inductively as follows:

- If $q = q_I$ then $w = \lambda$ and the only derivation run is $\mathsf{run}(\lambda, q_I) = q_I$.
- If $q \neq q_I$ then let $b$ be the last symbol of $w$, write $w = w' \cdot b$ and let $q' \in \mathsf{pred}(q, b)$ be the first $b$-predecessor of $q$ such that $w' \in \mathcal{L}(q')$. Then $\mathsf{run}(w, q) = \mathsf{run}(w', q') \xrightarrow{b} q$.

*Last common prefix state* Let $R$ and $R'$ be two runs in $\mathcal{A}^{\mathsf{u}}$ both starting at $q_I$. The *last common prefix state* of $R$ and $R'$, denoted by $\mathsf{lcps}(R, R')$, is $k^{\text{th}}$ state of

$R$ for the largest possible $k$ such that the $k^{\text{th}}$ prefix of $R$ equals the $k^{\text{th}}$ prefix of $R'$.

## 3 Algorithm

The following description is adapted from the original presentation of the countNFA algorithm in [8]. The overall goal is to compute a reliable approximation of $|\mathcal{L}_n(\mathcal{A})|$. The FPRAS algorithm begins by constructing the unrolled automaton $\mathcal{A}^{\mathsf{u}} = (\mathcal{Q}^{\mathsf{u}}, q_I, q_F, \mathcal{T}^{\mathsf{u}})$ for the given word length requirement $n$. It then processes the states of $\mathcal{A}^{\mathsf{u}}$ in a bottom-up manner, ensuring that all predecessors of a state $q$ are processed before $q$ itself.

For each state $q \in \mathcal{Q}^{\mathsf{u}}$, the algorithm computes two key quantities: an estimate $p(q)$, which approximates $|\mathcal{L}(q)|^{-1}$, and a collection of sample sets $S^1(q), \ldots, S^\gamma(q)$, which are all subsets of $\mathcal{L}(q)$. At a high level, for every state $q$ with predecessors $\mathsf{pred}(q) = (q_1, \ldots, q_k)$, the algorithm performs the following steps:

1. It uses the sample sets $(S^r(q_i))_{i\in[k],r\in[\gamma]}$ and the estimates $(p(q_i))_{i\in[k]}$ from the predecessors of $q$ to compute the new estimate $p(q)$.
2. It then constructs the new sample sets $(S^r(q))_{r\in[\gamma]}$ for state $q$ using its estimate $p(q)$ and the predecessors' sample sets.

The algorithm finishes after processing the final state $q_F$, and returns the final count estimate $N(q_F) = 1/p(q_F)$.

### 3.1 countNFA

The countNFA procedure (Algorithm 1) serves as the main entry point. It first computes the unrolled automaton $\mathcal{A}^{\mathsf{u}}$ and determines the necessary hyperparameters, such as the number of sampling sets $\gamma = n_s n_t$, based on the desired tolerance $\varepsilon$ and confidence $\delta$. To amplify the success probability to at least $1-\delta$, it executes $n_u$ independent runs of the core routine, countNFAcore (Algorithm 2), and returns the median of their outputs as the final estimate of $|\mathcal{L}_n(\mathcal{A})|$. We will return to the calculation of the four parameters $n_s$, $n_t$, $n_u$, and $\theta$ later in the paper.

**Algorithm 1:** countNFA$(\mathcal{A} = (\mathcal{Q}, q_I, q_F, \mathcal{T}), n, \varepsilon, \delta)$

```
1 Compute the unrolled NFA $\mathcal{A}^{\mathsf{u}} = (\mathcal{Q}^{\mathsf{u}}, q_I, q_F, \mathcal{T}^{\mathsf{u}})$ of $\mathcal{A}$ for length $n$
2 $(n_s, n_t, n_u, \theta) \leftarrow$ computeHyperparams$(|\mathcal{Q}^{\mathsf{u}}|, \delta, \varepsilon)$
3 for $j \leftarrow 1$ to $n_u$ do
4     est$_j \leftarrow$ countNFAcore$(\mathcal{A}^{\mathsf{u}}, n, n_s, n_t, \theta)$
5 return median(est$_1, \ldots,$ est$_{n_u}$)
```

### 3.2 countNFAcore

The procedure countNFAcore (Algorithm 2) is the core of the FPRAS. For the initial state $q_I$, it sets $p(q_I) = 1$ (line 1) and initializes all sample sets $S^r(q_I)$ to contain only the empty word, $\{\lambda\}$ (lines 3-4). The algorithm then proceeds layer by layer through the unrolled automaton. For each layer $\mathcal{Q}^i$, it iterates through every state $q \in \mathcal{Q}^i$ and calls estimateAndSample to compute the probability estimate $p(q)$ and the corresponding sample sets. To efficiently perform membership checks, a cache is maintained and updated for each layer via calls to computeCache(i) and updateCache(i). To ensure a polynomial running time, the procedure terminates and returns 0 if the total number of samples exceeds the threshold $\theta$ (line 10).

**Algorithm 2:** countNFAcore($\mathcal{A}^{\mathsf{u}}, n, n_s, n_t, \theta$)

```
1  $p(q_I) \leftarrow 1$
2  computeCache(0)
3  for $r \leftarrow 1$ to $n_s n_t$ do
4      $S^r(q_I) \leftarrow \{\lambda\}$
5  for $i \leftarrow 1$ to $n$ do
6      computeCache(i)
7      for $q \in \mathcal{Q}^i$ do
8          estimateAndSample(q)
9          if $\sum_{r \in [n_s n_t], q' \in \mathcal{Q}^{\mathsf{u}}} |S^r(q')| \geq \theta$ then
10             return 0
11     updateCache(i)
12 return $N(q_F)$
```

### 3.3 estimateAndSample

Given a state $q$ with predecessors $\mathsf{pred}(q) = (q_1, \dots, q_k)$, the estimateAndSample procedure computes $p(q)$ and the sample sets $(S^r(q))_{r \in [\gamma]}$. At this stage, for each predecessor $q_i$, an estimate $p(q_i)$ is known such that for any word $w \in \mathcal{L}(q_i)$, the probability of $w$ being in a sample set $S^r(q_i)$ is $p(q_i)$.

The procedure begins by computing $\rho(q) = \min(p(q_1), \dots, p(q_k))$ (line 1). It then uses the reduce procedure to normalize the predecessor sample sets (lines 2-3), creating temporary sets $(\bar{S}^r(q, q_i))_{r \in [\gamma], i \in [k]}$ where the sampling probability for any word is uniformly $\rho(q)$. Next, the union procedure is called to combine these normalized samples into new sets $(\hat{S}^r(q))_{r \in [\gamma]}$ (lines 4-5). For any word $w \in \mathcal{L}(q)$, the probability of $w$ being in $\hat{S}^r(q)$ is now $\rho(q)$, making $\rho(q)^{-1}|\hat{S}^r(q)|$ an unbiased estimator for $|\mathcal{L}(q)|$.

To obtain a robust estimate, the "median-of-means" technique is applied (lines 6-8). The $\gamma = n_s n_t$ estimates are partitioned into $n_t$ batches of size $n_s$, the mean estimation is computed separately for each batch, then the median of these means is taken. The inverse of this median is stored in $\hat{\rho}(q)$. Since $|\mathcal{L}(q)| \geq |\mathcal{L}(q_i)|$, it is expected that the final probability $p(q)$ is no larger than any $p(q_i)$. The algorithm thus sets $p(q) = \min(\rho(q), \hat{\rho}(q))$ (line 9). Finally, the procedure calls reduce one last time on the sets $\hat{S}^r(q)$ to produce the final sample sets $S^r(q)$ (lines 11-12), ensuring that for every $w \in \mathcal{L}(q)$, we have $\Pr\left[w \in S^r(q) \mid w \in \mathcal{L}(q)\right] = p(q)$.

**Algorithm 3:** estimateAndSample(q) with pred$(q) = (q_1, \ldots, q_k)$

```
1  $\rho(q) \leftarrow \min(p(q_1), \ldots, p(q_k))$
2  for $r \leftarrow 1$ to $n_s n_t$ and $i \leftarrow 1$ to $k$ do
3      $\bar{S}^r(q, q_i) \leftarrow$ reduce$\left(S^r(q_i), \frac{\rho(q)}{p(q_i)}\right)$
4  for $r \leftarrow 1$ to $n_s n_t$ do
5      $\hat{S}^r(q) \leftarrow$ union$\left(q, \bar{S}^r(q, q_1), \ldots, \bar{S}^r(q, q_k)\right)$
6  for $j \leftarrow 1$ to $n_t$ do
7      $M^j(q) \leftarrow \frac{1}{n_s \rho(q)} \sum_{r=1}^{n_s} |\hat{S}^{n_s(j-1)+r}(q)|$
8  $\hat{\rho}(q) \leftarrow \mathrm{median}(M^1(q), \ldots, M^{n_t}(q))^{-1}$
9  $p(q) \leftarrow \min(\rho(q), \hat{\rho}(q))$
10 $N(q) \leftarrow \frac{1}{p(q)}$
11 for $r \leftarrow 1$ to $n_s n_t$ do
12     $S^r(q) \leftarrow$ reduce$\left(\hat{S}^r(q), \frac{p(q)}{\rho(q)}\right)$
```

## 3.4 union

The purpose of union (Algorithm 4) is to construct a sample set $S'$ for a state $q$ from the sample sets $S_1, \ldots, S_k$ from pred$(q)$ with respect to their order $q_1 \prec q_2 \prec \cdots \prec q_k$. A key challenge is that a word may be reachable in $q$ via multiple predecessors, leading to duplicates. To ensure each word is generated through a unique path, the algorithm only includes words corresponding to their *derivation run*. Specifically, for a transition on symbol $b \in \Sigma$ from predecessors $q_i$ and $q_j$ (with $q_i \prec q_j$) to $q$, a word $w \cdot b$ is added to $S'$ from a sample $w \in S_i$ only if $q_i$ is the first predecessor in the order $\prec$ such that $w \in \mathcal{L}(q_i)$. This check for uniqueness is made efficient by using a precomputed cache, as detailed in [8].

## 3.5 reduce

The reduce method (Algorithm 5) adjusts the sampling density of a set. Given an input set $S$ and a target probability $p \in [0, 1]$, it constructs a new set $S'$ by retaining each element $s \in S$ with probability $p$, independently of all other elements.

```
Algorithm 4: union($q, S_1, \dots, S_k$)    (informal)
where $q$ has $k$ predecessors ordered as $q_1 \prec \cdots \prec q_k$ and $S_i \subseteq \mathcal{L}(q_i)$ for all $1 \le i \le k$

1 $S' \leftarrow \emptyset$
2 for $b \in \{0,1\}$ do
3     Let $J$ be the subset of $\{1,\dots,k\}$ such that $(q_j \mid j \in J) = b$-pred($q$)
4     for $j \in J$ and $w \in S_j$ do
5         if $w \notin \mathcal{L}(q_\ell)$ for every $\ell < j$ with $\ell \in J$ then
6             add $w \cdot b$ to $S'$
7 return $S'$
```

```
Algorithm 5: reduce($S, p$) with $p \in [0,1]$

1 $S' \leftarrow \emptyset$
2 for $s \in S$ do
3     Add $s$ to $S'$ with probability $p$
4 return $S'$
```

## 4 Technical Contributions

We now present FastNFA, a practical FPRAS for the #NFA algorithm implemented in C++ based on the original countNFA algorithm outlined in Section 3. FastNFA incorporates several theoretical and engineering enhancements, which we document here.

### 4.1 Optimization of parameters $n_t$ and $n_u$

At the start of the algorithm, countNFA computes four parameters—$n_s$, $n_t$, $n_u$, and $\theta$—that largely determine its runtime. Parameters $n_s$ and $n_t$ correspond to the median-of-means estimator in lines 6–8 of Algorithm 3: $n_s$ specifies the number of sampling sets used to compute each mean, while $n_t$ is the number of means whose median is taken. Their product, $n_s n_t = \gamma$, gives the total number of sampling sets computed per automaton state. On the other hand, the parameter $n_u$ denotes how many times the countNFAcore routine is repeated to reduce the overall error probability. Finally, $\theta$ is a hard limit on the total number of words stored across all sampling sets (line 10 of Algorithm 2).

For fixed inputs $\varepsilon, \delta$ and unrolled automaton $\mathcal{Q}^{\mathsf{u}}$, and defining $\kappa := \varepsilon/(1+\varepsilon)$, the original values of these parameters (as presented in [8]) are:

$$n_s = \left\lceil 4(n+1)\frac{(1+\kappa)^2}{\kappa^2(1-\kappa)} \right\rceil \qquad n_t = \left\lceil 8\ln\big(16|\mathcal{Q}^{\mathsf{u}}|\big) \right\rceil$$

$$n_u = \left\lceil 8\ln\big(\tfrac{1}{\delta}\big) \right\rceil \qquad \theta = \left\lceil 16(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}| \right\rceil$$

Table 1: Descriptive statistics of the ratio (i.e., estimate divided by exact value) between the value of $|\mathcal{L}_n(\mathcal{A})|$ estimated by countNFA with $(\varepsilon, \delta) = (0.8, 0.36)$ using the original $n_s$, $n_t$, $n_u$ parameters [8] and the exact count obtained by BruteNFA on 298 randomly-generated graphs. Note that all empirical results fall well within the theoretical guarantee.

| $n$ | $\mu$ | $\sigma$ | min | 25% | 50% | 75% | max | Theoretical Guarantee |
|---|---|---|---|---|---|---|---|---|
| 298 | 0.9994 | 0.0059 | 0.9375 | 0.9980 | 1.0000 | 1.0020 | 1.0087 | $[0.2, 1.8]$ |

We found that these parameters are prohibitively large for practical use. For example, setting $\varepsilon = 0.8$ and $\delta = 0.36$ (values used in practice, for instance, in [9]), we have that $n_s n_t \approx 608(n+1)\ln(16|\mathcal{Q}^u|)$ and $n_u = 9$. That means we have to compute about $608(n+1)\ln(16|\mathcal{Q}^u|)$ sampling sets for *each* state in the unrolled automaton (Algorithm 3), and repeat the entire procedure $n_u = 9$ times (Algorithm 1). This leads to impractically long runtimes in most cases.

*Empirical Study of Tightness* We first sought to better understand *empirically* the impact of $n_s$, $n_t$, and $n_u$ parameters on the output approximation quality. To do this, we developed an implementation of the FPRAS algorithm documented above in Section 3, as well as a brute force solution (documented in Algorithm 8 in Appendix G) for reference. Our experiments (Table 1) on 298 randomly-generated NFAs showed that estimates produced by countNFA always fell within a 10% tolerance (i.e., $N(q_F) \in (1 \pm \varepsilon)|\mathcal{L}(q_F)|$ is always satisfied with $\varepsilon = 0.1$). Since the accuracy observed in practice is much tighter than the theoretical guarantee of an 80% tolerance implied by the parameter $\varepsilon = 0.8$, this suggests that some of the bounds of the original analysis can be improved. Additionally, we noticed that the threshold $\theta$ is never reached in practice. We investigate this direction further below.

*Optimizing $n_u$* Recall that countNFA uses the median as a variance reduction technique in two key portions of the algorithm: first, in the median-of-means for estimating $|\mathcal{L}(q)|$ for every $q \in \mathcal{Q}^{\mathsf{u}}$ (line 8 of Algorithm 3); and second, in line 5 of Algorithm 1, when we take the median of $n_u$ independent estimates $\{\mathsf{est}_j\}_{1 \le j \le n_u}$, given by independent runs of countNFAcore.

Using the median is a natural idea. Take for example the median of independent countNFAcore runs just discussed. It is shown in [8] that for a single countNFAcore output est, we have $\Pr[\mathsf{est} \notin (1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|] \le 1/4$. Nevertheless, we desire our final estimate (i.e., the output of countNFA) to fall within $(1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|$ with small failure probability $\delta$. To reduce the probability of error from $1/4$ to $\delta$, we compute $n_u$ independent estimates by running countNFAcore several times, and report the median. We know that if the median was outside of the target range $(1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|$, then strictly more than $n_u/2$ of our independent estimates were outside of the range. Formally:

$$\Pr\left[\operatorname*{median}_{1\le j\le n_u} \mathsf{est}_j \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right] \le \Pr\left[\sum_{j=1}^{n_u} \mathbb{1}\left[\mathsf{est}_j \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right] > \frac{n_u}{2}\right]$$

Observing that $\mathbb{1}\left[\mathsf{est}_j \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right] \sim \text{Bernoulli}(1/4)$, we have the sum of indicator functions is a binomial random variable, resulting in:

$$\Pr\left[\operatorname*{median}_{1\le j\le n_u} \mathsf{est}_j \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right] \le \Pr\left[\text{Binomial}\left(n_u, \frac{1}{4}\right) > \frac{n_u}{2}\right]$$

The upper tail of the binomial distribution can be computed directly for arbitrary $n_u$, or bounded above using Chernoff's inequality as is done in [8]. In any case, if we guarantee that $n_u$ is large so that $\Pr\left[\text{Binomial}(n_u, 1/4) > n_u/2\right] \le \delta$, then the probability of our median-based estimator falling outside the desired $(1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|$ range will also be at most $\delta$.

In [8], the Chernoff inequality is applied to derive the bound

$$\Pr\left[\text{Binomial}\left(n_u, \frac{1}{4}\right) > \frac{n_u}{2}\right] \le e^{-n_u/8},$$

which implies that setting $n_u = \lceil 8\ln(1/\delta)\rceil$ suffices to achieve the desired confidence level $\delta$. However, computing the binomial tail exactly allows us to obtain a smaller value for the parameter $n_u$, defined as

$$n_u = \operatorname*{argmin}_{n\in\mathbb{N}} \left\{\Pr\left[\text{Binomial}\left(n, \frac{1}{4}\right) > \frac{n}{2}\right] \le \delta\right\}.$$

Since the original Chernoff-based value $\lceil 8\ln(1/\delta)\rceil$ already ensures the same confidence level, our refined value of $n_u$ must, by construction, be less than or equal to the original one. This reasoning will serve as a template for similar refinements introduced later in our analysis, motivating the following notation.

**Definition 1.** *For every* $0 \le p, \alpha \le 1$, *we define:*

$$\text{chase}(p,\alpha) = \operatorname*{argmin}_{n\in\mathbb{N}} \left\{\Pr\Big[\text{Binomial}(n,p) > \frac{n}{2}\Big] \le \alpha\right\}$$

*If there is no such* $n$ *that satisfies the requirement, we define* $\text{chase}(p,\alpha) = \infty$.

So we have our refined parameter $n_u = \text{chase}(1/4, \delta)$. The terminology comes from the idea that we are given an error probability $p$ and we are *chasing* a smaller error probability $\alpha$. Lemma 6 in Appendix D guarantees that $\text{chase}(\frac{1}{4}, \delta)$ is finite for every value of $\delta$.

*Optimizing $n_t$* In line 8 of Algorithm 3, we take the median of $n_t$ estimates $M^j(q)$ to reduce the chance of large errors in approximating $|\mathcal{L}(q)|$. Following the same reasoning used to derive a reduced value for $n_u$, we can similarly obtain a smaller value for $n_t$, defined as follows:

$$n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right)$$

The correctness of this refinement is more delicate. Roughly, the proof of correctness of $\mathsf{countNFA}$ in [8] requires us to guarantee:

$$\Pr\left[\underset{1\leq j\leq n_t}{\text{median}}\ M^j(q) \notin |\mathcal{L}(q)|(1\pm\varepsilon)\right] \leq \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|} \tag{1}$$

Moreover, since we are able to show that:

$$\Pr\left[M^j(q) \notin |\mathcal{L}(q)|(1\pm\varepsilon)\right] \leq \frac{1}{4}$$

then we have that $n_t = \text{chase}(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|})$ is the smallest number of repetitions needed to the ensure inequality (1), which is also ensured by the original, Chernoff-based parameter value $\lceil 8\ln(16|\mathcal{Q}^{\mathsf{u}}|)\rceil$. This description is not precise because the family of random estimates $\{M^j(q)\}_{1\leq j\leq n_t}$ is not mutually independent, so a more careful analysis is needed for validating the use of chase (see Appendix A).

*From constants to variables* Summarizing the analysis on $n_s$ and $n_t$ performed above, the refined parameter values are as follows:

$$n_u = \text{chase}\left(\frac{1}{4}, \delta\right) \qquad\qquad n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right) \tag{2}$$

While the other two parameters remain unchanged, namely

$$n_s = \left\lceil 4\cdot\frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)}\right\rceil \qquad\qquad \theta = \lceil 16\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|\rceil \tag{3}$$

The multiplicative constants 4 and 16 in $n_s$ and $\theta$, respectively, originate from arbitrary choices made in the proof of [8] to guarantee the algorithm's correctness. However, these choices do not account for their effect on the running time. Similarly, the value of $n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right)$ arises from the somewhat arbitrary decision to chase precisely $\frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}$. Theorem 1 shows that many of these constants can be allowed to vary freely, leading to the following theorem whose proof is given in Appendix C.

**Theorem 1.** *Let $x$, $y$, and $z$ be positive real numbers, with $x, y > 1$ and let:*

$$n_t = \text{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right) \qquad\qquad n_u = \text{chase}\left(\frac{1}{y}+\frac{2}{z}, \delta\right)$$

$$n_s = \left\lceil x\cdot\frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)}\right\rceil \qquad\qquad \theta = \lceil y\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}\rceil$$

*Then:*

$$\Pr\left[\operatorname*{median}_{1\le j\le n_u} \mathsf{est}_j \notin (1\pm\varepsilon)\,|\mathcal{L}(\mathcal{A})|\right] \le \delta$$

*provided that $n_t$ and $n_u$ are finite.*

Thus, for every trio of valid values of $x, y, z$, we can obtain different values for parameters $n_s$, $n_t$, $n_u$ and $\theta$ that still preserve the guarantees of correctness. The benefit of introducing these new variables is that we can now design a procedure to compute the optimal values of $x$, $y$, and $z$ that maximize the algorithm's execution speed. This procedure will be detailed in Section 4.2.

Observe that $x = 4, y = 16, z = 16$ recovers the parameters in (2) and (3). Since these parameters now have smaller values or remain unchanged with respect to the ones reported in [8], the asymptotic complexity of our algorithm remains the same.

## 4.2 Exploiting Parallelism

Recall that $n_u$ represents the number of times the countNFAcore procedure must be repeated to achieve $\delta$-guarantees on the probability of success. Since these $n_u$ executions are independent, they can be parallelized across multiple cores (see Section 4.4). However, for larger values of $\delta$, the optimal number of repetitions $n_u = \text{chase}\,(1/y + 2/z, \delta)$ is typically small, often just one or two. For instance, for the confidence level $\delta = 0.36$ and the original $x = 4$, $y = 16$, and $z = 16$, we have $n_u = 1$. Consequently, the available parallelism of modern multi-core processors remains largely underutilized in the current setting.

As mentioned in Section 4.1, the overall wall-clock runtime of the algorithm is largely governed by $n_s$ and $n_t$. Each automaton state requires $\gamma = n_s n_t$ sampling sets, and the most computationally expensive steps involve manipulating these sets (e.g., reduce and union operations).

In the baseline configuration, the parameters are set to $x = 4$, $y = 16$, and $z = 16$. Our objective is to adjust these values to minimize $n_s n_t$ while maintaining the constraint $n_u \le C$, where $C$ denotes the number of processor cores available. Let $\mathfrak{x}, \mathfrak{y}, \mathfrak{z}$ be the optimal parameters for $x, y$ and $z$ respectively. We thus have the following optimization problem:

$$\begin{aligned}(\mathfrak{x},\, \mathfrak{y},\, \mathfrak{z}) &= \operatorname*{argmin}_{x,\,y,\,z}\ \gamma \\ \text{subject to}\quad & n_u \le C\end{aligned}$$

If we define $w := \frac{1}{y} + \frac{2}{z}$ and substitute $n_s$ and $n_t$ with the expressions featured in Theorem 1, we have:

$$(\mathfrak{x},\, \mathfrak{y},\, \mathfrak{z}) = \operatorname*{argmin}_{x,\,y,\,z} \left\lceil x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)} \right\rceil \cdot \text{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right).$$

$$\text{subject to}\quad \text{chase}(w,\delta) \le C \quad\text{and}\quad w = \frac{1}{y} + \frac{2}{z}$$

An approximate solution to this optimization problem can be obtained by finding optimal values for $w$, $y$, $z$, and $x$, in that order. To minimize $\gamma$, we can proceed step by step:

**Step 1** Determine the largest value of $w$ such that $\mathrm{chase}(w, \delta) \leq C$, and denote it by $\mathfrak{w}$. Since the function $\mathrm{chase}(\cdot, \delta)$ is monotonically increasing (Lemma 5 in Appendix D), we can find this maximal $w$ efficiently using a binary search. Because $w$ is a continuous variable, the binary search stops when the difference between the upper and lower bounds becomes smaller than a predefined tolerance $\eta$. In practice, we found that setting $\eta = \delta/2^{10}$ provides a good balance between accuracy and computation time. Algorithm 7 in Appendix G describes the procedure for finding $\mathfrak{w}$. After determining $\mathfrak{w}$, our optimization constraint becomes $\frac{1}{y} + \frac{2}{z} \leq \mathfrak{w}$. From this, it follows directly that $z < \frac{2}{\mathfrak{w}}$.

**Step 2** Set $y$ to a sufficiently large constant. In our experiments, we used $y = 10^6 = \mathfrak{y}$. The variable $y$ appears only in the expression $\theta = \lceil y(1+\kappa) n_s n_t |\mathcal{Q}^{\mathsf{u}}| \rceil$, which represents the maximum number of words stored across all sampling sets and all states. The algorithm terminates if this threshold is reached (line 10 of Algorithm 2). In practice, as reported in Section 4.1, this threshold was never reached for the original setting $y = 16$. Therefore, increasing $y$ does not affect the algorithm's wall-clock runtime or its asymptotic complexity.

**Step 3** Set $\mathfrak{z} = \frac{2}{\mathfrak{w} - \frac{1}{\mathfrak{y}}}$. This value represents the smallest $z$ allowed by the optimization constraint found in Step 1. We choose the smallest possible $z$ because the function $n_t = \mathrm{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right)$ is monotonically increasing in $z$.
Note that $\mathfrak{z}$ is very close to the theoretical lower bound $\frac{2}{\mathfrak{w}}$, since we made $\mathfrak{y}$ very large.

**Step 4** With $y = \mathfrak{y}$ and $z = \mathfrak{z}$ fixed, the problem reduces to:

$$n_s(x) = \left\lceil x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)} \right\rceil \qquad n_t(x) = \mathrm{chase}\left(\frac{1}{x}, \frac{1}{\mathfrak{z}|\mathcal{Q}^{\mathsf{u}}|}\right) \qquad \mathfrak{x} = \operatorname*{argmin}_x n_s(x)\, n_t(x)$$

Treat the objective as a black box and perform a linear search over $x \in [1, b]$, where $b$ is chosen such that we can establish $\mathfrak{x} \leq b$ even without knowing $\mathfrak{x}$ a priori. Start at $x = 1$ because $\mathrm{chase}(p, \alpha)$ is defined only for probabilistic inputs (here $p = 1/x$), and stop at $x = b$ since the optimum cannot exceed this bound. It can be shown that for $\mathfrak{b} = \frac{\kappa^2(1-\kappa)}{(n+1)(1+\kappa^2)} n_s(4)\, n_t(4)$ we can guarantee $\mathfrak{x} \leq \mathfrak{b}$, so set $b = \mathfrak{b}$ as the search upper bound (see Appendix E for a proof of this fact). A linear search step size of $10^{-3}$ works well in practice and is used in our experiments.

## 4.3 Deterministic Line Optimization

Many NFAs, when unrolled, exhibit long sequences of states in which no nondeterminism is present. For example, consider encoding an NFA with a non-binary alphabet $\Sigma$ into one with the binary alphabet $\{0,1\}$: each transition over a character $a \in \Sigma$ can be encoded with a chain of $\mathcal{O}(\log(|\Sigma|))$ states using only transitions labeled with binary characters. Processing these chains one state at a time is inefficient, as the estimation and sampling steps are redundant due to the lack of nondeterminism. To exploit this common structure, we introduce an optimization for what we term *deterministic lines*.

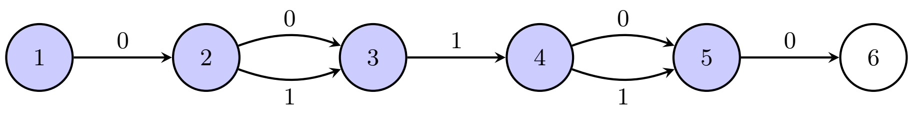

$$\begin{tikzcd} 1 \arrow[r, "0"] & 2 \arrow[r, bend left, "0"] \arrow[r, bend right, "1"'] & 3 \arrow[r, "1"] & 4 \arrow[r, bend left, "0"] \arrow[r, bend right, "1"'] & 5 \arrow[r, "0"] & 6 \end{tikzcd}$$

Fig. 1: Example of a deterministic line (highlighted in blue) within a DAG of length $l = 5$ with $d = 2$ double-edge transitions. It is represented with the two masks: $L_S = 0010$ and $L_F = 1010$.

**Definition 2 (Deterministic line).** *A deterministic line is a maximal path of states* $(q_1, q_2, \ldots, q_l)$ *in an unrolled NFA such that:*

- *Every state* $q_i$ *in the path, for* $i \in [1, \ldots, l]$*, has exactly one successor.*
- *Every state* $q_j$ *in the path, for* $j \in [2, \ldots, l]$*, has exactly one predecessor.*

**Deterministic line representation** We can represent the structure of a deterministic line of length $l$ containing $d$ double-edge transitions, where each double-edge corresponds to a pair of transitions labeled "0" and "1" between the same two consecutive states, using two bitmasks:

- **Structure Mask ($L_S$):** An $l$-bit vector where the $i$-th bit corresponds to the $i$-th transition: it is set on 0 for a transition on 0, and to 1 for a transition on 1. For double-edge transitions, this bit can be set to 0 as it will be ignored.
- **Fixed-Position Mask ($L_F$):** An $l$-bit vector where the $i$-th bit corresponds to the $i$-th transition: it is set on 0 if the transition is a double-edge transition, and to 1 otherwise.

**Accelerating Computation along Deterministic Lines** Instead of applying the standard estimateAndSample procedure to every state in a deterministic line, we can compute the final estimate and sample sets for $q_l$ directly from the information at $q_1$ (the exact implementation is documented in Algorithm 9 in Appendix G). This provides a significant computational shortcut.

**Theorem 2.** *Let a deterministic line from $q_1$ to $q_l$ contain $d$ double-edge transitions. The language size and sampling probability at $q_l$ can be computed deterministically from their values at $q_1$:*

$$|\mathcal{L}(q_l)| = 2^d \cdot |\mathcal{L}(q_1)| \qquad p(q_l) = \frac{p(q_1)}{2^d}$$

*Consequently, the full iterative estimation process for intermediate states of the line can be bypassed.*

**Efficient Membership Checking with Bitwise Operations** The deterministic line optimization provides significant performance and memory benefits by avoiding computation and storage for intermediate states. However, this creates a challenge for the `union` procedure, which relies on checking if a word $w$ from one predecessor's sample set $S(q_i)$ also belongs to the language of another predecessor, $\mathcal{L}(q_j)$, where $q_j$ might be the end of a deterministic line. Since we have discarded the intermediate cache information for states within the line, a direct lookup is impossible. To solve this, we introduce a highly efficient membership check that leverages bitwise operations.

**Theorem 3.** *Given a word $w$, let its $l$-bit suffix be $w_{suffix}$. The suffix is consistent with a line's structure (represented by $L_S$ and $L_F$) if and only if:*

$$(w_{suffix} \oplus L_S) \wedge L_F = 0$$

*where $\oplus$ is the bitwise XOR operation and $\wedge$ is the bitwise AND operation.*

The complete membership check is described in Algorithm 11 in Appendix G, which is used to reconstruct cache information on-the-fly when processing a state that follows a deterministic line.

**Lexicographical Ordering for Correct Caching** The deterministic line optimization introduces an asymmetry in caching. Because we discard cache information for intermediate states within a line, we can perform certain membership checks but not others:

- **Possible Check:** Given a word $w$ from a standard sample set $S(q_i)$, we *can* check if $w \in \mathcal{L}(q_j)$, where $q_j$ ends a deterministic line. This is done efficiently using the bitwise containsWord method.
- **Impossible Check:** Given a word $w'$ from a deterministic line's sample set $S(q_j)$, we *cannot* easily check if $w' \in \mathcal{L}(q_i)$ for some other state $q_i$. The necessary derivation information for $w'$ was never stored in the cache matrix.

The `union` procedure resolves non-determinism by iterating through a state's predecessors in a fixed order. To prevent the "impossible check" from ever being needed, we must ensure the "possible check" is always the one performed. We achieve this by establishing a new global state ordering, $\prec'$, that gives lexicographical priority to states that are part of a deterministic line. This new ordering is enforced by sorting the states within each layer and putting the states at the end of each deterministic line first.

### 4.4 Other Engineering Enhancements

**Handling Large and High-precision Floating-point Numbers** The size of a language $|\mathcal{L}(q)|$ can easily exceed the representational limits of standard integer types in C++. Consequently, the algorithm requires support for arbitrary-precision integers. To address this requirement, we employ the data types provided by the GNU Multiple Precision Arithmetic Library (GMP) [5], which enable accurate and reliable numerical computations throughout the algorithm.

**Merging the Two reduce Calls** In the estimateAndSample procedure, the reduce method is invoked twice for a given state $q$ with predecessor $q_i$: at the beginning of the iteration processing $q$,

$$\mathsf{reduce}\left(S^r(q_i), \tfrac{\rho(q)}{p(q_i)}\right)$$

and at the end of the previous iteration while processing $q_i$,

$$\mathsf{reduce}\left(\widehat{S}^r(q_i), \tfrac{p(q_i)}{\rho(q_i)}\right).$$

Since these calls are independent, we combined them into a single invocation:

$$\mathsf{reduce}\left(S^r(q_i), \tfrac{\rho(q)}{\rho(q_i)}\right), \qquad \tfrac{\rho(q)}{p(q_i)} \cdot \tfrac{p(q_i)}{\rho(q_i)} = \tfrac{\rho(q)}{\rho(q_i)}.$$

This optimization produces a modest performance improvement by reducing the number of reduce calls without altering the algorithm's correctness.

**Memory Management** The algorithm requires the generation of sampling sets for each state $q$ to estimate the size of the language $\mathcal{L}(q)$. In its original form, as presented in the reference work, the algorithm requires storing $S^r(q_i)$, $\widehat{S}^r(q)$, and $\bar{S}^r(q, q_i)$ for all $r \in [1, \dots, \gamma]$, $q \in \mathcal{Q}^u$, and $q_i \in \mathsf{pred}(q)$. In practice, such an approach is impractical, as it quickly leads to memory exhaustion.

An initial optimization consisted of storing sampling sets only for the current and immediately preceding layers. A similar strategy was applied to other data structures, such as $\mathsf{cache}_i$, where only the current layer is retained in memory.

Subsequently, we observed that three separate data structures were still being used to store samples, even though $\widehat{S}^r(q)$ and $\bar{S}^r(q, q_i)$ functioned solely as temporary storage during the calculation of $S^r(q_i)$. To address this, we slightly adapt the logic of estimateAndSample and union (see Algorithm 12 and 13 in Appendix G) so that the samples are reduced and merged on-the-fly, thus eliminating the need for these additional storage structures. The reduce logic is implemented directly inside union and no longer requires a separate method. This modification enables the removal of $\widehat{S}^r(q)$ and $\bar{S}^r(q, q_i)$, improving memory efficiency and using data locality for faster access.

Profiling revealed that memory allocation and access were among the main performance bottlenecks. Consequently, we applied several best practices to further optimize memory management, including pre-allocation and the use of references rather than value copying, resulting in a substantial improvement in execution time.

**Caching** We use the dense matrix data type from the Eigen library [6] to store caches (see Algorithm 2) as it allows for efficient matrix multiplication and memory allocation. In addition, to access $\mathsf{cache}_i(w, q)$ as required in the practical implementation of union (see Algorithm 14 in Appendix G, line 5), we maintain a hash table to map the words $w$ into the matrix's row index number.

**Parallelization** Recall that the countNFA procedure, described in Algorithm 1, requires $n_u$ runs of countNFAcore, where $n_u$ is calculated according to Section 4.1. Because each execution of countNFAcore is independent, we enable its parallel execution using the OpenMP API [10]. This parallelization allows an execution time reduction by an approximate factor of $n_u$, at the cost of increased memory consumption, as each independent run requires its own set of data structures.

## 5 Empirical Evaluation

To assess the practical performance of our proposed approach, we conducted an extensive empirical study. This work presents the first known implementation of the underlying theoretical framework, for which no standard benchmarks were previously available. Our evaluation is therefore designed to answer three primary research questions:

**RQ 1** What is the impact in performance of each of our optimizations in Section 4, and how do they compare to a brute force approach?

**RQ 2** How does the presence of deterministic structures in the input graphs impact the performance of the algorithm?

**RQ 3** How accurate are the estimates computed by the final version of our algorithm in practice (i.e., are they better than the theoretical guarantees imply)?

Table 2: Descriptive statistics of the ratio (i.e., estimate divided by exact value) between the value of $|\mathcal{L}_n(\mathcal{A})|$ estimated by FastNFA with $(\varepsilon, \delta) = (0.8, 0.36)$ and the exact count obtained by BruteNFA, on 305 randomly-generated graphs. Note that all empirical results fall well within the theoretical guarantee.

| $n$ | $\mu$ | $\sigma$ | min | 25% | 50% | 75% | max | Theoretical Guarantee |
|---|---|---|---|---|---|---|---|---|
| 305 | 1.0013 | 0.0105 | 0.9674 | 0.9986 | 1.0007 | 1.0045 | 1.0870 | $[0.2, 1.8]$ |

*Hardware and Benchmarks* All experiments were performed on the NSCC Singapore ASPIRE2A cluster [1]. Each run was allocated 2 CPU cores, 32 GB of RAM, and a timeout of 300 seconds (5 minutes). Our benchmark suite comprises a large set of randomly generated 0/1-labeled directed acyclic graphs (DAGs), consisting of 12 random graphs for each layer count from 20 to 200 layers, totaling 2,172 samples. The benchmarks are categorized into four structural classes based on the ratio of total states to the number of layers: `_1x` ($|\text{states}| = |\text{layers}|$), `_2x` ($|\text{states}| = 2 \times |\text{layers}|$), `_3x`, and `_5x`.

*Compared Algorithms* We use an ablation study, evaluating five distinct approaches:

– BruteNFA: A baseline implementation (Algorithm 8) developed to provide exact counts via exhaustive enumeration. Its exponential complexity makes it a viable reference only for smaller instances.
– countNFA: A direct translation of the original theoretical algorithm described in Section 3 into code. As we will show, this version proved impractical for most cases.
– FastNFA$^{\text{orig-params,no-det-opt}}$ An implementation of countNFA that further leverages the engineering optimizations described in Section 4.4.
– FastNFA$^{\text{no-det-opt}}$: A more efficient version of FastNFA$^{\text{orig-params,no-det-opt}}$, which incorporates not only the engineering enhancements but also the optimizations of the parameters $n_s$, $n_t$, and $n_u$ described in Section 4.1.
– FastNFA: Our final proposed algorithm, which builds on FastNFA$^{\text{no-det-opt}}$ to integrate the deterministic line optimization described in Section 4.3.

*Parameters* In line with previous work on FPRAS evaluation [9], we set the tolerance to $\varepsilon = 0.8$ and the confidence to $\delta = 0.36$ in our experiments.

**RQ1: Performance Impact of Optimizations** Figure 2 provides a comprehensive answer to our first research question by aggregating the performance of all implementations across the entire benchmark suite. The plot clearly illustrates the impact of both our theoretical and engineering contributions. The BruteNFA method, while exact, is quickly overwhelmed by its exponential complexity. Our first implementation of the original countNFA algorithm proved even less practical: it not only performed worse than BruteNFA but failed on the vast majority of the benchmark suite due to timeouts or memory exhaustion.

Our first attempt of a practical solution, FastNFA$^{\text{orig-params,no-det-opt}}$, focused on the engineering enhancements described in Section 4.4. While this version offered a drastic improvement over countNFA, empirical evaluation revealed a crucial bottleneck. Its poor scaling was a direct consequence of the large number of sampling sets, determined by the hyperparameter $\gamma = n_s \cdot n_t$. Consequently, FastNFA$^{\text{orig-params,no-det-opt}}$ still underperformed the BruteNFA baseline on most instances. This finding underscored a key insight: engineering enhancements alone were insufficient, and a truly practical implementation required theoretical changes to the algorithm's hyperparameters. The true breakthrough is evident in the performance of FastNFA$^{\text{no-det-opt}}$: by combining our theoretical contributions with engineering optimizations, this version exhibits the expected polynomial-time behavior, solving a substantial portion of the benchmark suite. Finally, FastNFA, the final version of our algorithm that also incorporates the deterministic line optimization from Section 4.3, shows the best overall performance.

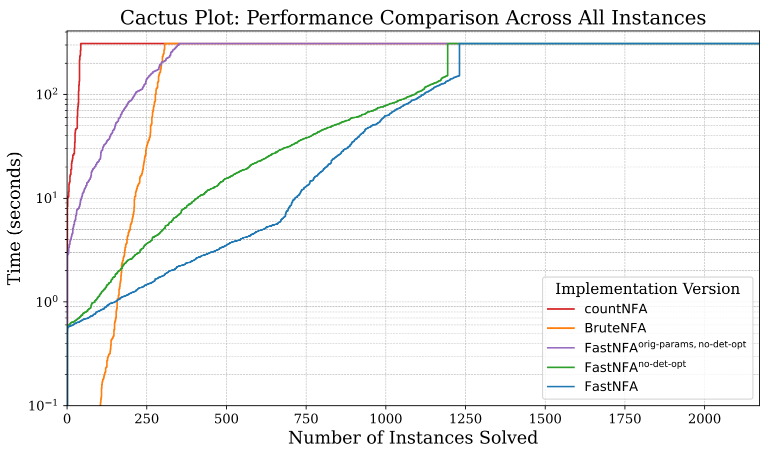
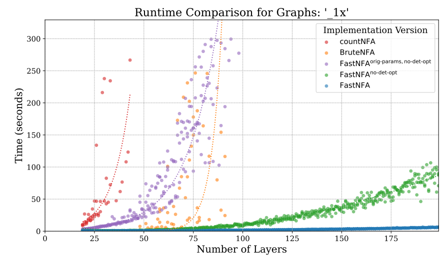


Fig. 2: Cactus plot comparing the performance of the implementations across the entire benchmark suite. The $y$-axis is on a logarithmic scale.

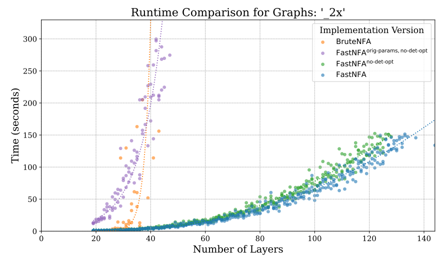
**RQ2: Impact of Deterministic Structures** The impact of deterministic structures is best analyzed by comparing performance across different graph categories, as shown in Figure 2. To quantify the structural differences, we measured the average *determinism ratio*, as the total number of states occurring as part of some deterministic lines, divided by the total number of states. The determinism ratio for each class of benchmarks is as follows: `_1x` (98.71%), `_2x` (70.87%), `_3x` (46.93%), and `_5x` (26.45%).

The `_1x` graphs, with a nearly pure deterministic structure, provide a clear showcase for the optimization. As seen in Figure 3a, our final algorithm leverages these lines to achieve a dramatic performance improvement. This powerful effect is characteristic of any benchmark class featuring such structures.

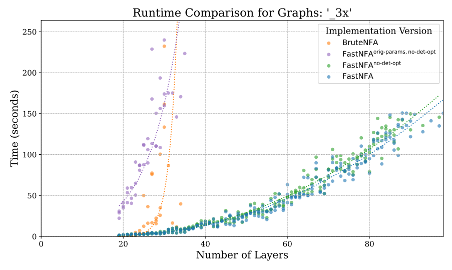
Conversely, as the determinism ratio decreases across the `_2x`, `_3x`, and `_5x` categories, the performance of our final version increasingly converges with that of the standard countNFA, as shown in Figures 3b–3d. This is a crucial result, as it demonstrates that our optimization incurs no measurable overhead on graphs lacking deterministic lines, confirming its effectiveness as a purely opportunistic and beneficial enhancement.



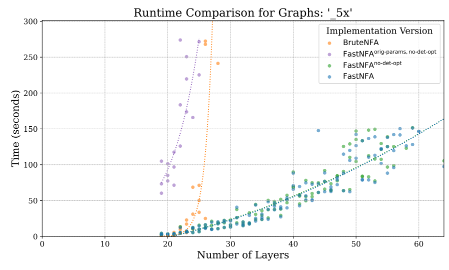
(a) Runtime on graphs with suffix `_1x`.



(b) Runtime on graphs with suffix `_2x`.



(c) Runtime on graphs with suffix `_3x`.



(d) Runtime on graphs with suffix `_5x`.

Fig. 3: Runtime performance comparison on different graph categories.

**RQ3: Accuracy of the Final Algorithm** The empirical accuracy of our final algorithm is summarized in Table 2. For all instances where an exact count was available from our brute-force solver, we validated the output of our FPRAS. The results confirm that all estimates were well within the theoretical bounds as constrained by $\varepsilon = 0.8$. Furthermore, the observed average and maximum relative errors were significantly lower than this bound, underscoring the reliability of our approach in practice.

## 6 Conclusion

We introduce FastNFA, a practical tool for approximating the number of words accepted by an NFA. Experimental results validated the efficiency of the novel optimizations used. In future work, we would like to further investigate the performance of FastNFA on a wider class of benchmarks, as well as consider how analogous ideas can be used for the problem of *sampling* words accepted by an NFA.

**Acknowledgments.** This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 project RS04/25, and by the Nanyang Technological University Start-up Grant.

## References

1. ASPIRE 2A | NSCC (May 2024), https://www.nscc.sg/aspire-2a/
2. Àlvarez, C., Jenner, B.: A very hard log-space counting class. Theor. Comput. Sci. **107**(1), 3–30 (1993)
3. Amarilli, A., van Bremen, T., Gaspard, O., Meel, K.S.: Approximating queries on probabilistic graphs (2025), https://arxiv.org/abs/2309.13287, accepted to *Logical Methods in Computer Science*
4. Arenas, M., Croquevielle, L.A., Jayaram, R., Riveros, C.: #NFA admits an FPRAS: efficient enumeration, counting, and uniform generation for logspace classes. J. ACM **68**(6), 48:1–48:40 (2021)
5. GNU Project: GNU Multiple Precision Arithmetic Library (GMP). Free Software Foundation, https://gmplib.org/, 6.3.0 (2023-07-30) edn. (2025), accessed August 12, 2025
6. Jacob, B., Guennebaud, G.: Eigen: A C++ linear algebra library (2021), https://eigen.tuxfamily.org, released August 18, 2021; MPL 2.0 license
7. Meel, K.S., Chakraborty, S., Mathur, U.: A faster FPRAS for #NFA. Proc. ACM Manag. Data **2**(2), 112 (2024)
8. Meel, K.S., de Colnet, A.: Towards practical FPRAS for #NFA: Exploiting the power of dependence. Proc. ACM Manag. Data **3**(2) (Jun 2025). https://doi.org/10.1145/3725253, https://dl.acm.org/doi/10.1145/3725253
9. Meel, K.S., Shrotri, A.A., Vardi, M.Y.: Not all FPRASs are equal: demystifying FPRASs for DNF-counting. Constraints **24**(3–4), 211–233 (Oct 2019). https://doi.org/10.1007/s10601-018-9301-x
10. OpenMP Architecture Review Board: OpenMP application program interface, version 6.0. https://www.openmp.org/specifications (Nov 2024), accessed via OpenMP website

## Appendix

### A Definition of the Random Process

It was established in Section 4.1 that setting $n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right)$ preserves the correctness of countNFA. This optimized choice of $n_t$ arises from the observation that, in the proof of correctness for countNFA, it is crucial to ensure the following inequality holds:[4]

$$\Pr\left[\operatorname*{median}_{1\leq j\leq n_t} M^j(q) \notin |\mathcal{L}(q)|(1\pm\varepsilon)\right] \leq \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}. \tag{4}$$

If we were to follow the same reasoning used to refine $n_u$, the value $n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right)$ should be justified in two steps:

1. Show that $\Pr\left[M_q^j \notin (1\pm\varepsilon)|\mathcal{L}(q)|\right] \leq \frac{1}{4}$;

[4] In fact, the proof in [8] is more intricate, as will be discussed later. The bound $\frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}$ only applies to a simplified version of countNFA that omits line 9 in Algorithm 2.

2. Then, by definition, $n_t = \text{chase}\left(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}\right)$ ensures inequality (4).

The difficulty with applying this reasoning directly is that, unlike the family $\{\mathsf{est}_j\}_{1\leq j\leq n_u}$, the collection $\{M^j(q)\}_{1\leq j\leq n_t}$ is not mutually independent. This is because all $n_t$ estimates depend on the sampling sets of the predecessors of $q$. This interdependence considerably complicates the analysis. To address the issue, [8] introduces a random process that simulates a variant of countNFAcore in which the size of the sets $S_r(q)$ is unbounded (i.e., omitting line 9 of Algorithm 2). They call this variant countNFAcore$^\star$. Using a coupling argument, they show how $M^j(q)$ can be replaced by a new random variable with more well-behaved properties, conditioned on sufficient knowledge of the algorithm's execution up to state $q$. This knowledge is formally represented by what they call the *history* of $q$. More precisely, the history $h$ for a set of states $Q \subseteq \mathcal{Q}^{\mathsf{u}}$ is a mapping $h : Q \to \mathbb{Q}$, and $h$ is *realizable* when there exists a run of countNFAcore$^\star$ that gives the value $h(q)$ to $p(q)$ for every $q \in Q$.

The random process comprises $n_s n_t$ independent copies identified by the superscript $r$. For $q \in \mathcal{Q}^{\mathsf{u}}$, $t \in \mathbb{Q}$ and $h$ a realizable history for $q$, we have several random variables $\mathfrak{S}^r_{h,t}(q)$ with domain all possible subsets of $\mathcal{L}(q)$. $\mathfrak{S}^r_{h,t}(q)$ simulates $S^r(q)$ in the situation where the value $p(q')$ for each ancestor $q'$ of $q$ has been set to $h(q')$ and where $p(q)$ is set to $t$ (so $t$ is restricted to values that can be given to $p(q)$ by the algorithm under $h$). The variables $\mathfrak{S}^r_{h,t}(q)$ are defined inductively. We refer the reader to the original paper for a more detailed discussion of the random process, while we limit ourselves to provide a intuitive explanation of the random variables defined within the process that are relevant to our discussion.

- As mentioned, $\mathfrak{S}^r_{h,t}(q)$ simulates the sampling set $S^r(q)$ when the countNFAcore$^\star$ run satisfies $p(q') = h(q')$ for every $q' \in \mathsf{ancestors}(q)$ and $t = p(q)$.
- We call $\hat{\mathfrak{S}}^r_h(q)$ the random variable that simulates $\hat{S}^r(q)$ when the countNFAcore$^\star$ run satisfies $p(q') = h(q')$ for every $q' \in \mathsf{ancestors}(q)$.
  It is important to note that the family of random variables $\{\hat{\mathfrak{S}}^r_h(q)\}_{1\leq r\leq n_s n_t}$ is mutually independent by construction, contrary to the original family of random sampling sets $\hat{S}^r(q)$. We refer the reader to [8] for the formal definition of $\hat{\mathfrak{S}}^r_h(q)$.
- Let $\rho_h(q) = \min_{q'\in\mathsf{ancestors}(q)} h(q')$ and $R_j$ the range of natural numbers $[n_s(j-1)+1, n_s j]$. Define $\mathfrak{M}^j_h(q) = \left(\sum_{r\in R_j} |\hat{\mathfrak{S}}^r_h(q)|\right) / (\rho_h(q) n_s)$. This is the random process equivalent to $M^j(q)$ in line 7 of Algorithm 3. It is the $j$-th mean of our median-of-means estimator.
  Each $\mathfrak{M}^j_h(q)$ is the normalized sum of a family of independent random variables $\{|\hat{\mathfrak{S}}^r_h(q)|\}_{r\in R_j}$. Moreover, the families corresponding to different indices $j$ are independent of one another. Therefore, the family $\{\mathfrak{M}^j_h(q)\}_{1\leq j\leq n_t}$ inherits this independence.

Going back to $n_t$, the definition of $n_t = \text{chase}(\frac{1}{4}, \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|})$ exploits the independence of the family $\{\mathfrak{M}^j_h(q)\}_{1\leq j\leq n_t}$. For the formal proof of correctness of this new value of $n_t$, see Theorem 1 in Appendix C.

## B High-level Overview of the Proof of Correctness of countNFA

A thorough but high-level overview of the proof of correctness of countNFA in [8] is given in this section, as it will be useful in later portions of our discussion.

Here, we use $\kappa := \frac{\varepsilon}{1+\varepsilon}$. An important property is that for any positive $a, b \in \mathbb{R}$, $a \in b(1 \pm \kappa) \implies a \in \frac{b}{1 \pm \epsilon}$.

1. Fact: The algorithm countNFA is correct if and only if

$$\Pr\left[\underset{1\le j\le n_u}{\text{median}}\, \mathsf{est}_j \in (1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|\right] \le 1 - \delta$$

   where $\mathsf{est}_j$ is the output of the $j-$th independent run of countNFAcore.

2. Observation: Let est the output of a single run of countNFAcore. A sufficient condition to guarantee that

$$\mathsf{est} = p(q_F)^{-1} \in (1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|.$$

   holds is that the following 2 events occur simultaneously
   - Event $E_1$: for every $q \in \mathcal{Q}^{\mathsf{u}}$, the estimate $p(q)$ satisfies

$$p(q) \in \frac{1 \pm \kappa}{|\mathcal{L}(q)|}$$

   - Event $E_2$: The sum of the sizes $|S^r(q)|$ are bounded

$$\sum_{q \in \mathcal{Q}^{\mathsf{u}}} |S^r(q)| \le \theta$$

     If we take the complement of these events, we can set up the following inequality

$$\Pr\left[\mathsf{est} \notin (1 \pm \varepsilon)|\mathcal{L}(\mathcal{A})|\right] \le \Pr\left[E_1^c \text{ or } E_2^c\right] \le \Pr\left[E_1^c\right] + \Pr\left[E_2^c\right].$$

     Where the superscript $^c$ denotes complement.

3. The proof proceeds to bound the probability of $E_1^c$. In particular, it will be established that

$$\Pr\left[E_1^c\right] = \Pr\left[\exists q \in \mathcal{Q}^{\mathsf{u}} :\ p(q) \notin \tfrac{1\pm\kappa}{|\mathcal{L}(q)|}\right] \le \tfrac{1}{16}.$$

   This is proved as follows:

   (a) Fix a particular $q$. Via a coupling argument, it is shown that

$$\begin{aligned}\Pr\left[p(q) \notin \tfrac{1\pm\kappa}{|\mathcal{L}(q)|}\right] &\le \Pr\left[\underset{1\le j\le n_t}{\text{median}}\, M^j(q) \notin \tfrac{1\pm\kappa}{|\mathcal{L}(q)|}\right] \\ &\le \Pr\left[\underset{1\le j\le n_t}{\text{median}}\, \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right]\end{aligned}$$

   It will be shown in Steps 3.b–3.g that

$$\Pr\left[\operatorname*{median}_{1\le j\le n_t} \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}$$

Afterwards, using a union bound, we conclude that

$$\begin{aligned}\Pr\Big[\exists q\in\mathcal{Q}^{\mathsf{u}} : \ p(q)\notin \tfrac{1\pm\kappa}{|\mathcal{L}(q)|}\Big] &\le \sum_{q\in\mathcal{Q}^{\mathsf{u}}} \Pr\left[p(q)\notin \tfrac{1\pm\kappa}{|\mathcal{L}(q)|}\right]\\ &\le \sum_{q\in\mathcal{Q}^{\mathsf{u}}} \Pr\left[\operatorname*{median}_{1\le j\le n_t} \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right]\\ &\le \sum_{q\in\mathcal{Q}^{\mathsf{u}}} \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|} = \frac{1}{16}.\end{aligned}$$

Which is what we wanted.

(b) Bound $\mathrm{Var}\big(|\hat{\mathfrak{S}}_h^r(q)|\big)$ explicitly using properties of the unrolled automaton.

(c) Using $\mathrm{Var}\big(\sum_{i=1}^{n_s} X_i\big) = \sum_{i=1}^{n_s}\mathrm{Var}(X_i)$ for $\{X_i\}_{1\le i\le n_t}$ mutually independent, derive an upper bound on $\mathrm{Var}\left(\mathfrak{M}_h^j(q)\right)$ as a function of $n_s$.

(d) Apply a concentration inequality (i.e., Chebyshev's) to obtain

$$\Pr\left[\mathfrak{M}_h^j(q)\notin\frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \frac{1}{n_s}\cdot\Big(\frac{\kappa^2(1-\kappa)}{(n+1)(1+\kappa^2)}\Big)$$

(e) Substituting $n_s = \lceil 4\cdot\frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)}\rceil$ in the previous inequality we get

$$\Pr\left[\mathfrak{M}_h^j(q)\notin\frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \frac{1}{4}.$$

(f) Use the Chernoff bound to obtain

$$\Pr\left(\operatorname*{median}_{1\le j\le n_t} \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right) \le e^{-n_t/8}.$$

(g) Setting $n_t = \lceil 8\ln(16n|\mathcal{Q}^{\mathsf{u}}|)\rceil$ in the previous inequality we get

$$\Pr\left(\operatorname*{median}_{1\le j\le n_t} \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right) \le \frac{1}{16|\mathcal{Q}^{\mathsf{u}}|}.$$

4. Now, we will show $\Pr\left[E_2^c\right] \le \frac{1}{8}$:

$$\Pr\left[\sum_{q\in\mathcal{Q}^{\mathsf{u}}} |S^r(q)| > \theta\right] \le \tfrac{1}{8}.$$

(a) A simple application of union bound tells us

$$\Pr\left[\sum_{q\in\mathcal{Q}^{\mathsf{u}}}|S^r(q)|>\theta\right]\le\underbrace{\Pr\left[\sum_{q\in\mathcal{Q}^{\mathsf{u}}}|S^r(q)|>\theta \text{ and } \exists q:\operatorname*{median}_{1\le j\le n_t}\mathfrak{M}_h^j(q)\notin\frac{|\mathcal{L}(q)|}{1\pm\kappa}\right]}_{\Pr[E_3]}$$
$$+\underbrace{\Pr\left[\exists q:\operatorname*{median}_{1\le j\le n_t}\mathfrak{M}_h^j(q)\notin\frac{|\mathcal{L}(q)|}{1\pm\kappa}\right]}_{\Pr[E_1^c]}$$

Where $E_3$ is defined as the event in the first summand

(b) It is shown in the step 2 that $\Pr[E_1^c]\le\frac{1}{16}$

(c) Use Markov's inequality and the random process to show that

$$\Pr[E_3]\le\frac{1}{\theta}\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|$$

(d) Setting $\theta=\lceil 16\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|\rceil$ in the inequality above we obtain

$$\Pr[E_3]\le\frac{1}{16}$$

(e) $\Pr\left[\sum_{q\in\mathcal{Q}^{\mathsf{u}}}|S^r(q)|\right]\le\Pr[E_3]+\Pr[E_1^c]\le\frac{1}{16}+\frac{1}{16}=\frac{1}{8}$

5. Combining Items 2 and 3 gives

$$\Pr[\mathsf{est}\notin(1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|]\le\Pr[E_1^c]+\Pr[E_2^c]\le\frac{1}{16}+\frac{1}{8}<\frac{1}{4}.$$

6. $\mathsf{countNFAcore}$ computes $n_u$ independent estimates $\mathsf{est}_1,\dots,\mathsf{est}_{n_t}$ and reports their median as final estimate. By Chernoff's inequality,

$$\Pr\left[\operatorname*{median}_{1\le j\le n_u}\mathsf{est}_j\notin(1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right]\le e^{-n_u/8}$$

7. Finally, choose $n_u$ so that the overall failure probability of $\mathsf{countNFA}$ is less than $\delta$. Plugging $n_u=\lceil -8\ln\delta\rceil$ in the inequality above, we obtain

$$\Pr\left[\operatorname*{median}_{1\le j\le n_u}\mathsf{est}_j\notin(1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|\right]\le\delta$$

Throughout the explanation, we can observe how each parameter $n_s$, $n_t$, $n_u$, and $\theta$ is assigned its original value.

– $n_s$ (the number of samples per mean) is fixed so that the probability of the mean misestimating $|\mathcal{L}(q)|$ is less than $\frac{1}{4}$. This interpretation follows directly from Steps 3.d–3.e.
– $n_t$ (the number of means from which the median is taken) is defined so that the probability of the median of means misestimating $|\mathcal{L}(q)|$ is less than $\frac{1}{16\mathcal{Q}^{\mathsf{u}}}$. This interpretation follows directly from Steps 3.f–3.g.

- $\theta$ (the upper bound on the number of words contained in all sampling sets across all states) is fixed such that $\Pr[E_3] \leq \frac{1}{16}$. This interpretation follows directly from Steps 4.c–4.d.
- $n_u$ (the number of repetitions of `countNFAcore`) is chosen so that the probability of the final output misestimating $|\mathcal{L}(\mathcal{A})|$ is less than $\delta$. This interpretation follows directly from Steps 6–7.

## C Theorems derived from overview

In the following lemmas and theorem, we will refer to Step 1 - 7, to specify portions of the high-level proof found in Appendix B.

**Lemma 1.** *Let $x$ be an arbitrary real number greater than 1. If $n_s = \lceil x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)} \rceil$, then*

$$\Pr\left[\mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right] \leq \frac{1}{x}$$

.

*Proof.* From Step 3.d, we have

$$\Pr\left[\mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right] \leq \frac{1}{n_s} \cdot \frac{\kappa^2(1-\kappa)}{(n+1)(1+\kappa^2)}$$

Since $n_s \geq x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)}$, it immediately follows that

$$\Pr\left[\mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right] \leq \frac{1}{x}$$

□

**Lemma 2.** *Let $x$ be an arbitrary real number greater than 1. Let $n_s = \lceil x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)} \rceil$. If $n_t = \text{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right)$ is finite, then*

$$\Pr[E_1^c] = \Pr\left[\exists q \in \mathcal{Q}^{\mathsf{u}} : \ p(q) \notin \tfrac{1 \pm \kappa}{|\mathcal{L}(q)|}\right] \leq \frac{1}{z}$$

.

*Proof.* From Lemma 1, $\Pr\left[\mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right] \leq \frac{1}{x}$. Thus, we have that

$$\Pr\left[\operatorname*{median}_{1 \leq i \leq n_t} \mathfrak{M}_h^j(q) \notin \frac{|\mathcal{L}(q)|}{1 \pm \kappa}\right] \leq \Pr\left[\text{Binomial}\left(n_t, \frac{1}{x}\right) > \frac{n_t}{2}\right]$$

We have that $n_t = \text{chase}(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|})$. Now, by the definition of chase and the fact that $\{\mathfrak{M}_h^j(q)\}_{1 \leq j \leq n_t}$ is a family of independent random variables,

$$\Pr\left[\underset{1\le i\le n_t}{\text{median}}\,\mathfrak{M}^j_h(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \Pr\left[\text{Binomial}\left(n_t, \frac{1}{x}\right) > \frac{n_t}{2}\right]$$
$$\le \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}$$

Now, we can proceed as done in Step 3.a.

$$\Pr[E_1^c] \le \Pr\left[\exists q : \underset{1\le j\le n_t}{\text{median}}\,\mathfrak{M}^j_h(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \sum_{q\in\mathcal{Q}^{\mathsf{u}}} \Pr\left[\underset{1\le j\le n_t}{\text{median}}\,\mathfrak{M}^j_h(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right]$$
$$\le \sum_{q\in\mathcal{Q}^{\mathsf{u}}} \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|} = \frac{1}{z}.$$

□

**Lemma 3.** *Let $y$ be an arbitrary positive real number greater than 1. Set $\theta = \lceil y\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|\rceil$. Then,*

$$\Pr[E_3] = \Pr\left[\sum_{q\in\mathcal{Q}^{\mathsf{u}}} |S^r(q)| > \theta \text{ and } \exists q : \underset{1\le j\le n_t}{\text{median}}\,\mathfrak{M}^j_h(q) \notin \frac{|\mathcal{L}(q)|}{1\pm\kappa}\right] \le \frac{1}{y}$$

*Proof.* From Step 4.c,

$$\Pr[E_3] \le \frac{1}{\theta}\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|$$

The inequality is immediate by plugging the proposed value. □

**Lemma 4.** *Let $x, y$ arbitrary positive real numbers greater than 1. Let $z$ be another positive real number. Let $n_s = \lceil x\cdot\frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)}\rceil$. Let $n_t = \text{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right)$. Let $\theta = \lceil y\cdot(1+\kappa)n_s n_t|\mathcal{Q}^{\mathsf{u}}|\rceil$. Then,*

$$\Pr[\mathsf{est} \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|] \le \frac{1}{y} + \frac{2}{z}.$$

*Provided $n_t$ is finite.*

*Proof.* From Step 2, we have that $\Pr[\mathsf{est} \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|] \le \Pr[E_1^c] + \Pr[E_2^c]$. By Step 4.a, $\Pr[E_1^c] + \Pr[E_2^c] \le \Pr[E_1^c] + \Pr[E_1^c] + \Pr[E_3]$. Using Lemma 2 and Lemma 3

$$\Pr[\mathsf{est} \notin (1\pm\varepsilon)|\mathcal{L}(\mathcal{A})|] \le \frac{1}{z} + \frac{1}{z} + \frac{1}{y}$$

□

**Theorem 1.** *Let $x$, $y$, and $z$ be positive real numbers, with $x, y > 1$ and let:*

$$n_t = \text{chase}\left(\frac{1}{x}, \frac{1}{z|\mathcal{Q}^{\mathsf{u}}|}\right) \qquad n_u = \text{chase}\left(\frac{1}{y} + \frac{2}{z}, \delta\right)$$
$$n_s = \left\lceil x \cdot \frac{(n+1)(1+\kappa^2)}{\kappa^2(1-\kappa)} \right\rceil \qquad \theta = \lceil y \cdot (1+\kappa) n_s n_t |\mathcal{Q}^{\mathsf{u}}| \rceil$$

*Then:*

$$\Pr\left[\operatorname*{median}_{1 \le j \le n_u} \mathsf{est}_j \notin (1 \pm \varepsilon)\, |\mathcal{L}(\mathcal{A})|\right] \le \delta$$

*provided that $n_t$ and $n_u$ are finite.*

*Proof.* From Lemma 4 we have

$$\Pr[\mathsf{est} \notin (1 \pm \varepsilon)\, \mathcal{L}(\mathcal{A})] = \frac{1}{y} + \frac{2}{z}$$

Which implies

$$\Pr\left[\text{Binomial}\left(n_u, \frac{1}{y} + \frac{2}{z}\right) > \frac{n_u}{2}\right]$$

And $n_u$ is exactly defined with chase so that we can guarantee

$$\Pr\left[\text{Binomial}\left(n_u, \frac{1}{y} + \frac{2}{z}\right) > \frac{n_u}{2}\right] \le \delta$$

□

## D Properties of Chase

We remind ourselves the definition of chase.

**Definition 1.** *For every $0 \le p, \alpha \le 1$, we define:*

$$\text{chase}(p, \alpha) = \operatorname*{argmin}_{n \in \mathbb{N}} \left\{ \Pr\left[\text{Binomial}(n, p) > \frac{n}{2}\right] \le \alpha \right\}$$

*If there is no such $n$ that satisfies the requirement, we define* $\text{chase}(p, \alpha) = \infty$.

The following lemmas are routine.

**Lemma 5.** $\text{chase}(\cdot, \alpha)$ *is non-decreasing in the range* $(0, 1)$. $\text{chase}(p, \cdot)$ *is non-increasing in the range* $(0, 1)$.

**Lemma 6.** $\text{chase}(p, \alpha)$ *is finite if* $p < 1/2$

**Lemma 7.** $\text{chase}(p, \alpha) = 1$ *if* $p \ge \alpha$.

**Lemma 8.** chase$(p,\alpha) = 2$ *if* $p^2 \leq \alpha \leq p$.

**Lemma 9.** chase$(p,\alpha) = \infty$ *if and only if* $p \geq \frac{1}{2}$ *and* $\alpha < p^2$.

To compute chase, we first check the finiteness condition in Lemma 9. If chase is finite, we do a linear scan starting from $n = 1$, until we obtain

$$\Pr\Big[\text{Binomial}(n,p) > \frac{n}{2}\Big] \leq \alpha$$

Properties of the binomial distribution and Lemmas 6 to 9 can be used to design a more efficient search. Such optimizations were not required because the linear scan performs sufficiently fast.

## E Optimizing $x$, $y$, $z$

In Section 4.2, we described a strategy to obtain the optimal values for $x, y, z$ that maximize execution speed, constrained on the number of CPU cores. Algorithm 6 summarizes the entire procedure.

During the optimization of $x$, we need to find the minimum of the objective function $n_s(x)n_t(x)$. Since this objective function behaves as a black box, we employ a linear search to identify the optimal $x$ over the range $[1, b]$, where $b$ is chosen such that we can establish $\mathfrak{x} \leq b$ even without knowing $\mathfrak{x}$ a priori. We begin at $x = 1$ because chase$(p,\alpha)$ is only defined for probabilistic inputs, and we terminate at $x = b$ since the optimal solution cannot exceed this upper bound. Our next step is to determine an appropriate value for $b$.

Notice that $n_s(x)n_t(x) \geq n_s(x)$ for $x > 1$ since $n_t(x) \in \mathbb{N} \cup \{\infty\}$. Let $x_0$ be an arbitrary anchor point where $n_s(x_0)n_t(x_0)$ is finite (By Lemma 6, any $x_0 > 2$ suffices; we choose $x_0 = 4$ arbitrarily). Therefore, if we find $b$ such that $n_s(b) \geq n_s(x_0)n_t(x_0)$, then

$$n_s(x_0)n_t(x_0) \leq n_s(b) \leq n_s(x) \leq n_s(x)n_t(x) \qquad \forall\, x > b$$

$n_s(b) \leq n_s(x)$ because $n_s(\cdot)$ is a linear function with a positive slope. Hence, no $x > b$ can minimize the objective, ensuring that the optimum $\mathfrak{x}$ lies within $[1, b]$. To compute $b$, we solve $n_s(b) = n_s(4)n_t(4)$:

$$b = \frac{\kappa^2(1-\kappa)}{(n+1)(1+\kappa^2)}\, n_s(4)n_t(4).$$

This $b$ satisfies $n_s(b) \geq n_s(x_0)n_t(x_0)$ for $x_0 = 4$, which is what we wanted. So $b$ provides practical upper bound on the linear search of $[1, b]$. The step size of the linear search $10^{-3}$ works well in practice and is used in our experimentation.

## F Theorems related to Deterministic Line Optimization

**Theorem 2.** *Let a deterministic line from $q_1$ to $q_l$ contain $d$ double-edge transitions. The language size and sampling probability at $q_l$ can be computed deterministically from their values at $q_1$:*

$$|\mathcal{L}(q_l)| = 2^d \cdot |\mathcal{L}(q_1)| \qquad p(q_l) = \frac{p(q_1)}{2^d}$$

*Consequently, the full iterative estimation process for intermediate states of the line can be bypassed.*

*Proof.* Let $p(q_1)$ be the probability that a word $w \in \mathcal{L}(q_1)$ is included in a sample set $S(q_1)$. Any such word $w$ is a prefix for exactly $2^d$ distinct words in $\mathcal{L}(q_l)$. Let $p(q_l) = p(q_1)/2^d$, it is necessary to apply reduce to each of these $2^d$ suffixes with probability $p(q_l)/p(q_1) = 1/2^d$. The number of words to retain for each prefix $w \in S(q_1)$ is therefore a random variable following the binomial distribution[5] $\mathrm{Binom}(2^d, 1/2^d)$. Since enumerating all $2^d$ continuations is infeasible for large $d$, we can directly sample from this distribution to determine how many words to generate. This allows us to create the final sample sets for $q_l$ efficiently, as detailed in Algorithm 10 in Appendix G.

**Theorem 3.** *Given a word $w$, let its $l$-bit suffix be $w_{suffix}$. The suffix is consistent with a line's structure (represented by $L_S$ and $L_F$) if and only if:*

$$(w_{suffix} \oplus L_S) \wedge L_F = 0$$

*where $\oplus$ is the bitwise XOR operation and $\wedge$ is the bitwise AND operation.*

*Proof.* The term $(w_{\text{suffix}} \oplus L_S)$ identifies all positions where the word's suffix differs from the line's structure. The bitwise AND with $L_F$ masks out these differences at all double-edge positions. The result is zero if and only if there are no mismatches at any fixed-transition positions. This allows for membership verification in $\mathcal{O}(1)$ time.

## G Algorithm listing

**Algorithm 6:** computeHyperparams($|\mathcal{Q}^{\mathsf{u}}|, \delta, \varepsilon$)

```
1  $C \leftarrow$ number of processor cores from system
2  $\kappa = \frac{\varepsilon}{1+\varepsilon}$
3  $\mathfrak{w} \leftarrow$ wSolver($\delta, \frac{\delta}{2^{10}}, C$)
4  $\mathfrak{y} \leftarrow 10^6$
5  $\mathfrak{z} \leftarrow \frac{2}{\mathfrak{w} - \frac{1}{\mathfrak{y}}}$
6  $b \leftarrow \frac{\kappa^2(1-\kappa)}{(n+1)(1+\kappa^2)} n_s(4) n_t(4)$.
7  for $x \in [1, b]$, with step size of $10^{-3}$. do
8      $n_t = \text{chase}(1/x, 1/(\mathfrak{z}|\mathcal{Q}^{\mathsf{u}}|))$
9      $n_s = \left\lceil \frac{(x)(1+\kappa^2)(n+\kappa)}{\kappa^2(1-\kappa)} \right\rceil$
10     keep $(n_s, n_t)$ if $\gamma = n_s n_t$ is minimal
11 $\theta \leftarrow \mathfrak{y}(1+\kappa) n_s n_t |\mathcal{Q}^{\mathsf{u}}|$
12 $n_u = \text{chase}(1/\mathfrak{y} + 2/\mathfrak{z}, \delta)$
13 return $(n_s, n_t, n_u, \theta)$
```

[5] In practice, the binomial distribution $\mathrm{Binom}(2^d, 1/2^d)$ is well-approximated by a Poisson distribution with parameter $\lambda = 1$ when $2^d$ is sufficiently large, since $\mathrm{Binom}(n, p) \xrightarrow{d} \mathrm{Poisson}(np)$ (i.e. the distributions converge) as $n \to \infty$ and $p \to 0$.

**Algorithm 7:** wSolver$(\delta, \eta, C)$
Binary search algorithm to find $w^*$ given tolerance $\delta$, search sensitivity $\eta$, and number of processor cores $C$

```
1  left ← $\delta$
2  right ← 1.0
3  while $right - left > \eta$ do
4      mid ← (left + right)/2
5      Update left and right according to the value of chase(mid, $\delta$)
6      if chase($mid, \delta$) $\leq C$ then
7          $w$ ← mid
8          left ← mid
9      else
10         right ← mid
11 return $w$
```

**Algorithm 8:** BruteNFA$(\mathcal{A}^{\mathsf{u}} = (\mathcal{Q}^{\mathsf{u}}, q_I^0, q_F^n, \mathcal{T}^{\mathsf{u}}))$

```
   // Initialize empty dictionary ‘‘words’’
1  for $q \in \mathcal{Q}^u$ do
2      words($q$) ← $\emptyset$
3  words($q_I^0$) ← $\{\lambda\}$
4  for $i \leftarrow 0$ to $n - 1$ do
5      for $q^i \in \mathcal{Q}^i$ do
6          for $b \in \{0, 1\}$ do
7              for each $q^{i+1} \in \mathcal{Q}^{i+1}$ with $q^i \in b$-pred$(q^{i+1})$ do
8                  for $w \in words(q^i)$ do
9                      add $w \cdot b$ to words($q^{i+1}$)
10 return |words($q_F^n$)|
```

**Algorithm 9:** estimateAndSample$(q)$ (with deterministic line)

```
1  if $q$ is the last state of some deterministic line then
2      $q_1$ ← first state of line
3      $d$ ← |double edges in line|
4      $S^r(q)$ ← deterministicSampler($S(q_1)$, line)
5      $p(q) \leftarrow \frac{p(q_1)}{2^d}$
6  if $q$ belongs to some deterministic line then
7      return ;                          // Skip ordinary computation
8  else
9      Continue as before...
```

**Algorithm 10:** deterministicSampler($(S^r(q_1))_{r\in[\gamma]}$, line)

**Input:** Sample sets $(S^r(q_1))_{r\in[\gamma]}$ for the line's start state; A deterministic line object line with $d$ double-edge transitions

**Output:** The final sample sets $(S(q_l))_{r\in[\gamma]}$ for the line's end state.

```
1  $S'(q_l) \leftarrow$ empty sets
2  for $r \leftarrow 1$ to $\gamma$ do
3      foreach $w_{prefix} \in S^r(q_1)$ do
4          $k \leftarrow$ sample from Binom$(2^d, \frac{1}{2^d})$
5          for $i \leftarrow 1$ to $k$ do
6              $w' \leftarrow w_{\text{prefix}}$
7              foreach transition $t$ in line do
8                  if $t$ is a double edge then
9                      $b \leftarrow$ randomly choose from $\{0, 1\}$
10                 else
11                     $b \leftarrow t$
12                 $w' \leftarrow w' \cdot b$
13             add $w'$ to $S'^r(q_l)$
14 return $S'(q_l)$
```

**Algorithm 11:** containsWord(line, $w$)

**Input:** A deterministic line object “line” (containing its start state language $\mathcal{L}(q_1)$, length $l$, structure mask $L_S$, and fixed-position mask $L_F$), and a word $w$.

**Output:** `true` if $w$ is in the language of the line's final state, `false` otherwise.

```
   // 1. Decompose the word
1  $w_{\text{prefix}} \leftarrow$ shift_right$(w, l)$ ;                // Get first $|w| - l$ bits
2  $w_{\text{suffix}} \leftarrow w \wedge ($shift_left$(1, l) - 1)$ ;        // Get last $l$ bits
   // 2. Check the prefix
3  if $w_{prefix} \notin \mathcal{L}(q_1)$ then
4      return false
   // 3. Check the suffix
5  if $((w_{suffix} \oplus L_S) \wedge L_F) \neq 0$ then
6      return false
7  return true
```

**Algorithm 12:** $\mathsf{estimateAndSample}(q)$ (optimized) with $\mathsf{pred}(q) = (q_1, \ldots, q_k)$

```
1  $\rho(q) = \min(p(q_1), \ldots, p(q_k))$
2  for $1 \le j \le n_t$ do
3      for $1 \le i \le n_s$ do
4          $r \leftarrow j \cdot n_s + i$
5          $S^r(q) = \mathsf{union}\,(q, S^r(q_1), \ldots, S^r(q_k))$
6          $\hat{M}^j(q) \leftarrow \hat{M}^j(q) + |S^r(q)|$
7      $M^j(q) \leftarrow \frac{\hat{M}^j(q)}{n_s \cdot \rho(q)}$
8  $\hat{\rho}(q) = \mathrm{median}(M^1(q), \ldots, M^{n_t}(q))^{-1}$
9  $p(q) = \min(\rho(q), \hat{\rho}(q))$
10 $N(q) = \frac{1}{p(q)}$
```

**Algorithm 13:** $\mathsf{union}(q, S_1, \ldots, S_k)$ (optimized) where $q$ has $k$ predecessors $q_1 \prec \cdots \prec q_k$ and $S_i \subseteq \mathcal{L}(q_i)\ \forall\ 1 \le i \le k$

```
1  $S' = \emptyset$
2  for $b \in \{0,1\}$ do
3      let $J$ be the subset of $\{1, \ldots, k\}$ such that $(q_j \mid j \in J) = b\text{-}\mathsf{pred}(q)$
4      for $j \in J$ and $w \in S_j$ do
5          if $w \notin \mathcal{L}(q_\ell)$ for every $\ell < j$ with $\ell \in J$
6          then add $w \cdot b$ to $S'$ with probability $\frac{\rho(q)}{\rho(q_j)}$
7  return $S'$
```

**Algorithm 14:** $\mathsf{union}(q, S_1, \ldots, S_k)$ with $q \in \mathcal{Q}^i$ (with caching)

```
1  $S' = \emptyset$
2  for $b \in \{0,1\}$ do
3      let $J$ be the subset of $\{1, \ldots, k\}$ such that $(q_j \mid j \in J) = b\text{-}\mathsf{pred}(q)$
4      for $j \in J$ and $w \in S_j$ do
5          if $\mathsf{cache}'_i(w \cdot b, q) \in [2^{k-j}, 2^{k-j+1})$ then add $w \cdot b$ to $S'$ with
             probability $\frac{\rho(q)}{\rho(q_j)}$
6  return $S'$
```