Membership Testing for Semantic Regular Expressions

YIFEI HUANG, University of Southern California, USA
MATIN AMINI, University of Southern California, USA
ALEXIS LE GLAUNEC, Rice University, USA
KONSTANTINOS MAMOURAS, Rice University, USA
MUKUND RAGHOTHAMAN, University of Southern California, USA

This paper is about semantic regular expressions (SemREs). This is a concept that was recently proposed by Chen et al. [9] in which classical regular expressions are extended with a primitive to query external oracles such as databases and large language models (LLMs). SemREs can be used to identify lines of text containing references to semantic concepts such as cities, celebrities, political entities, etc. The focus in their paper was on automatically synthesizing semantic regular expressions from positive and negative examples. In this paper, we study the *membership testing problem*:

(1) We present a two-pass NFA-based algorithm to determine whether a string w matches a SemRE r in $O(|r|^2|w|^2 + |r||w|^3)$ time, assuming the oracle responds to each query in unit time. In common situations, where oracle queries are not nested, we show that this procedure runs in $O(|r|^2|w|^2)$ time. Experiments with a prototype implementation of this algorithm validate our theoretical analysis, and show that the procedure massively outperforms a dynamic programming-based baseline, and incurs a $\approx 2\times$ overhead over the time needed for interaction with the oracle.

(2) We establish connections between SemRE membership testing and the triangle finding problem from graph theory, which suggest that developing algorithms which are simultaneously practical and asymptotically faster might be challenging. Furthermore, algorithms for classical regular expressions primarily aim to optimize their time and memory consumption. In contrast, an important consideration in our setting is to minimize the cost of invoking the oracle. We demonstrate an $\Omega(|w|^2)$ lower bound on the number of oracle queries necessary to make this determination.

CCS Concepts: • **Theory of computation** → **Regular languages**.

Additional Key Words and Phrases: Regular expressions, oracle-backed grammars, membership testing

ACM Reference Format:
Yifei Huang, Matin Amini, Alexis Le Glaunec, Konstantinos Mamouras, and Mukund Raghothaman. 2025. Membership Testing for Semantic Regular Expressions. *Proc. ACM Program. Lang.* 9, PLDI, Article 197 (June 2025), 24 pages. https://doi.org/10.1145/3729300

1 Introduction

Since their introduction in the 1970s, regular expression matching engines such as grep [34] and sed [22] have emerged as the workhorses of data extraction and transformation workflows. Regular expressions are well-suited to describe simple syntactic structures in text, such as for identifying sequences of tokens and for matching delimiter boundaries. Apart from their ease of use, they are backed by well-understood theoretical foundations and efficient matching algorithms.

Authors' Contact Information: Yifei Huang, University of Southern California, Los Angeles, CA, USA, yifeih@usc.edu; Matin Amini, University of Southern California, Los Angeles, CA, USA, matinami@usc.edu; Alexis Le Glaunec, Rice University, Houston, TX, USA, afl5@rice.edu; Konstantinos Mamouras, Rice University, Houston, TX, USA, mamouras@rice.edu; Mukund Raghothaman, University of Southern California, Los Angeles, CA, USA, raghotha@usc.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/6-ART197
https://doi.org/10.1145/3729300

Chen et al. [9] recently proposed extending classical regular expressions with primitives to query large language models (LLMs). For example, one might search for mentions of Turing award winners in some text using the pattern $\Sigma^*\langle\text{Turing award winner}\rangle\Sigma^*$, assuming an oracle exists which can identify their names. One might also conceivably apply such patterns to search for mentions of Eastern European cities, respiratory diseases endemic among humans, to flag passwords and SSH keys accidentally left behind in source code commits, or to query external services such as DNS and WHOIS while categorizing spam email. The motivation is that classical regular expressions are a bad tool to describe such semantic categories, which are better resolved by asking a search engine, an LLM, or by querying other external sources of information such as databases of award winners or atlases of major cities.

The major focus in [9] was on synthesizing semantic regular expressions from user-provided examples of positive and negative data. We assert that a more fundamental problem involves *membership testing*: Does a string w match a given semantic regular expression r? This problem will be our subject of study in this paper.

In contrast to [9], which uses an expansive formalism (including regular expression operators such as intersection and complement), we will focus on a simplified core subset of SemREs (oracle refinement + the classical regular expression operators—union, concatenation, and Kleene-*). This allows us to isolate the impact of oracle refinement on the complexity of membership testing.

Still, even for this simplified fragment, it is relatively straightforward to show an $\Omega(|w|^2)$ lower bound on the number of oracle queries needed (and hence on the worst-case running time) to make this determination. See Theorem 4.1. It follows that any automata-based matching algorithm—which are frequently associated with linear time algorithms—must necessarily perform non-trivial additional work.

In this context, our primary contribution is a two-pass NFA-based matching algorithm: The first pass recognizes syntactic patterns required by the underlying classical regular expression, and constructs a data structure which we call a query graph. The query graph encodes outstanding query-substring pairs that need to submitted to the oracle for verification and Boolean relationships among the results of these queries. The second pass uses dynamic programming to evaluate the query graph and compute the final result.

The principal challenge is in designing the query graph so that it can both be rapidly constructed and efficiently evaluated. One particularly tricky part of this procedure occurs while constructing the query graph, when we must track the string indices at which oracle queries begin and end. Recall that because classical Thompson NFAs also contain ϵ-transitions, their evaluation requires computing their transitive closure after processing each subsequent character of the input string. Unfortunately, in our case, these ϵ-closures might themselves involve complicated query structures that need to be carefully processed. Our solution is a novel gadget which summarizes all possible patterns in which substrings may need to be examined by the oracle using only $\Theta(|r|)$ vertices in the query graph. At this point, query graph construction reduces to a relatively straightforward tiling of this gadget after processing each character. Taken together, our end-to-end membership testing algorithm runs in $O(|r|^2|w|^2 + |r||w|^3)$ time, while making $O(|r||w|^2)$ oracle queries.

In Section 4.2, we show that SemRE matching is at least as hard as finding triangles in graphs. While subcubic (i.e., $O(n^{3-\epsilon})$ time) algorithms are known for this latter problem [41], they all build on algorithms for fast (integer) matrix multiplication, and are consequently slower in practice. This suggests that matching algorithms which are both practical and asymptotically faster might be challenging to devise.

This hardness result fundamentally relies on SemREs with nested queries. An example of such a pattern is $(\Sigma^*(\Sigma^* \wedge \langle\text{City}\rangle)\Sigma^*) \wedge \langle\text{Celebrity}\rangle$, which identifies celebrities whose names also include

the names of cities, such as `Paris Hilton`. We expect that common SemREs will not utilize such nested queries. In their absence, we can show that our algorithm runs in $O(|r|^2|w|^2)$ time.

We also include the results of an experimental evaluation: We use a benchmark dataset consisting of two text files and nine SemREs, and compare our matching algorithm to a simple dynamic programming-based baseline. Our algorithm turns out to have 101× the throughput of the baseline, and a 2× overhead over the cost of consulting the oracle. Furthermore, our algorithm makes 51% fewer oracle queries than the baseline. Given the costs of large-scale LLM use [25], our hope is that this algorithm will enable the practical application of oracle-backed regular expressions.

Summary of Contributions. To summarize, our paper makes the following contributions:

(1) We initiate the study of the membership testing problem for semantic regular expressions (SemREs), and present an automaton-based matching algorithm that runs in $O(|r|^2|w|^2 + |r||w|^3)$ time with $O(|r||w|^2)$ queries. Additional assumptions, such as the absence of nesting, permit faster performance guarantees to be made.
(2) We report on a set of SemREs and corresponding text files that can be used to benchmark membership testing algorithms for SemREs. Experiments with this dataset validate our theoretical analysis, and indicate that our algorithm outperforms a simple dynamic-programming based baseline.
(3) We establish the first lower bounds on SemRE membership testing: (*a*) That at least $\Omega(|w|^2)$ oracle queries are necessary in the worst case, and (*b*) that subcubic algorithms for membership testing could be used to find triangles in graphs in subcubic time.

2 Semantic Regular Expressions

2.1 The Formalism

Fix a finite alphabet Σ, a space of possible queries Q, and let 🔮 : $Q \times \Sigma^* \to$ Bool be the external oracle. We choose this symbol to evoke the idea of the oracle being a magical crystal ball. We define *semantic regular expressions* (SemRE) as the productions of the following grammar:

$$r ::= \bot \mid \epsilon \mid a \mid r_1 + r_2 \mid r_1 r_2 \mid r^* \mid r \wedge \langle q \rangle, \tag{1}$$

for $a \in \Sigma$ and $q \in Q$. Observe that the only difference from classical regular expressions [32] is the final production rule $r ::= \cdots \mid r \wedge \langle q \rangle$. Each semantic regular expression r identifies a set of strings $[\![r]\!] \subseteq \Sigma^*$, defined recursively as follows:

$$\left.\begin{aligned} [\![\bot]\!] &= \emptyset, \\ [\![\epsilon]\!] &= \{\epsilon\}, \\ [\![a]\!] &= \{a\}, \\ [\![r_1 + r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!], \\ [\![r_1 r_2]\!] &= [\![r_1]\!][\![r_2]\!], \\ [\![r^*]\!] &= \bigcup_{i \geq 0} [\![r]\!]^i, \text{ and} \\ [\![r \wedge \langle q \rangle]\!] &= \{w \in [\![r]\!] \mid 🔮(q, w) = \text{true}\}, \end{aligned}\right\} \tag{2}$$

where the concatenation and exponentiation of sets of strings $A, B \subseteq \Sigma^*$ are defined as usual:

$$\begin{aligned} AB &= \{w_1 w_2 \mid w_1 \in A, w_2 \in B\}, \\ A^0 &= \{\epsilon\}, \text{ and} \\ A^{n+1} &= A^n A, \text{ for } n \geq 0 \text{ respectively.} \end{aligned}$$

Once again, the only difference from classical language theory is in the definition $[\![r \wedge \langle q \rangle]\!]$ of oracle queries. Given a SemRE r and a string $w \in \Sigma^*$, the central problem of this paper is to determine whether $w \in [\![r]\!]$. An easy solution is to use dynamic programming and memoization to operationalize Equation 2. This is the approach used in the SMORE implementation by Chen et al. [9].[1] This algorithm would require $O(|r||w|^3)$ time in the worst case. *Can we do better?*

2.2 Examples

In common use, one might set Σ to be the set of ASCII or Unicode code points. One might choose $Q = \{\text{Sportsperson}, \text{Scientist}, \text{Eastern European city}, \dots\}$. Of course, 🔮 would depend on the backing oracle, but for the sake of concreteness, one might assume 🔮$(\text{Sportsperson}, w) = \text{true}$ for $w \in \{\texttt{Simone Biles}, \texttt{Lionel Messi}, \texttt{Roger Federer}, \dots\}$, and that 🔮$(\text{Scientist}, w) = \text{true}$ for $w \in \{\texttt{Albert Einstein}, \texttt{Marie Curie}, \texttt{Charles Darwin}, \dots\}$.

We start with two simple examples of semantic regular expressions: Lines of text containing political news might match the expression $\Sigma^*(\Sigma^* \wedge \langle \text{Politician} \rangle)\Sigma^*$, while rosters of sports teams might be described by the pattern $((\Sigma^* \wedge \langle \text{Sportsperson} \rangle)\texttt{", "})^*(\Sigma^* \wedge \langle \text{Sportsperson} \rangle)$.

Note 2.1. Observe the particularly common idiom, $\Sigma^* \wedge \langle q \rangle$. When the intent is clear, we will shorten this and simply write $\langle q \rangle$. For example, we will write $\Sigma^*\langle \text{Politician} \rangle\Sigma^*$ instead of the more verbose $\Sigma^*(\Sigma^* \wedge \langle \text{Politician} \rangle)\Sigma^*$, and $(\langle \text{Sportsperson} \rangle\texttt{", "})^*\langle \text{Sportsperson} \rangle$ instead of the second SemRE above.

Note 2.2 (Role of the alphabet). Throughout this paper, we will assume that $\Sigma = \{a, b, c, \dots\}$ is finite. Given the number of ASCII and Unicode code points, this assumption needs deeper consideration while actually implementing these algorithms. Our prototype implementation operates over streams of 8-bit bytes (using Rust's default UTF-8 encoding) and supports three types of character classes: (*a*) The wildcard Σ (or "`.`"), which is matched by any symbol from the alphabet, $a \in \Sigma$. In other words, the character class Σ is simply notational shorthand and a minor optimization over the SemRE $a + b + c + \cdots$. (*b*) `[a–b]` which matches all characters $c \in \Sigma$ such that $a \leq c \leq b$, and (*c*) `[^a–b]`, which matches all characters $c \in \Sigma$ that do not match `[a–b]`. By constructing an effective Boolean algebra over character classes, symbolic alphabets [31, 38] provide a more principled approach to these issues, and is a good direction for future work.

Example 2.3 (Credential leaks). A common class of software vulnerabilities arises when developers embed passwords, SSH keys, and other secret credentials in their source code, which subsequently gets accidentally publicized as part of an open source software release. Several popular programs (such as Gitleaks [29]) have been written to automatically discover hardcoded secrets in source code. One approach to discover these forgotten credentials is to examine all string literals occurring in the repository and flag those which look like secrets. While classical regular expressions are ideal for recognizing string literals, an external oracle, such as an LLM, is better suited to determine whether a specific string literal might be a password or SSH key. This approach can be summarized using the following semantic regular expression:[2]

$$\begin{aligned} r_{\text{pass}} &= \texttt{"}(\Sigma_s + \text{Esc})^* \wedge \langle \text{Password or SSH key} \rangle\texttt{"}, \text{ where} \\ \Sigma_s &= \Sigma \setminus \{\texttt{"}, \text{BS}\} \text{ and} \\ \text{Esc} &= \text{BS } \{\texttt{b}, \texttt{t}, \texttt{n}, \texttt{f}, \texttt{r}, \texttt{"}, \texttt{'}, \text{BS}\}, \end{aligned} \tag{3}$$

and where BS is the backslash character.

[1] See Lines 271–394 of the file `Smore-main/lib/interpreter/executor.py` in the artifact that may be downloaded from https://doi.org/10.5281/zenodo.8144182.

[2] After accounting for escape sequences, as per https://docs.oracle.com/javase/specs/jls/se22/html/jls-3.html#jls-3.10.5.

Assumption 2.4 (Deterministic oracles). We have implicitly assumed that 🔮 : $Q \times \Sigma^* \to$ Bool is a deterministic function. On the other hand, the token sequences produced by LLMs are fundamentally non-deterministic. This arises from multiple factors, including its temperature, sampling strategy, and even non-determinism arising from floating point calculations performed on the GPU. We compensate for these issues by lowering the temperature to 0 and caching results to previously submitted queries, thereby forcefully determinizing the oracle.

Example 2.5 (Non-existent file paths). Source code also frequently contains hardcoded file paths, which progressively become obsolete as bitrot sets in. A SemRE similar to the following might be used to automatically detect such paths:

$$r_{\text{file}} = (\Sigma_f^* \text{ FS } (\Sigma_f^* + \text{FS})^+ + \Sigma_f^+ \text{ FS}) \wedge \langle \text{Non-existent file path} \rangle, \text{ where} \tag{4}$$
$$\Sigma_f = \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{0}, \ldots, \mathtt{9}, \mathtt{-}, \text{PD}, \text{US}\}, \text{ and where}$$

PD and US are the period and underscore characters, and FS is the forward slash character commonly used to separate directories in Unix file paths.

Note 2.6. In the above example, the oracle would be an external script that determines the (non-)existence of specific file paths. Although many SemREs would use LLMs as the background oracle, 🔮, we emphasize that the oracle may also be constructed using other sources of information, such as databases, calculators, or other external tools.

Example 2.7 (Identifier naming conventions). Software maintainers love to compile elaborate conventions for how to name variables, identifiers, and other program elements,[3] and also love to complain about programmers not following these conventions. Ignoring keywords and non-ASCII characters, Java identifiers may be recognized using the regular expression $\Sigma_l(\Sigma_l + \{\mathtt{0}, \ldots, \mathtt{9}\})^*$, where $\Sigma_l = \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \$, \text{US}\}$. With an appropriate oracle, they simply need to scan their project for identifiers of the form:

$$r_{\text{id}} = (\Sigma_l \ (\Sigma_l + \{\mathtt{0}, \ldots, \mathtt{9}\})^*) \ \wedge \langle \text{Inappropriately named Java identifier} \rangle \tag{5}$$

where the oracle presumably confirms that the name is meaningful, correctly uses camel case or snake case formats, and is drawn from the appropriate grammatical categories (such as verbs or nouns).

Example 2.8 (Pharmaceutical spam). Spam email routinely contains advertisements for pharmaceutical products and weight loss supplements. Assume access to an LLM which can reliably recognize the names of these products. One might then use the following SemRE to flag the subject lines of these email messages:

$$r_{\text{spam},1} = \mathtt{Subject:}\ \Sigma^* \ [\text{Medicine name}] \ \Sigma^*, \tag{6}$$

where $[q]$ is shorthand for $\Sigma^+ \wedge \langle q \rangle$. One might additionally wish to restrict their search to single whole words, so they might use the following SemRE instead:

$$r_{\text{spam},2} = \mathtt{Subject:}\ \Sigma^* \text{ WS } \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}\}^+ \wedge \langle \text{Medicine name} \rangle \text{ WS } \Sigma^*, \tag{7}$$

where WS is the whitespace character. For a somewhat humorous example of classical regular expressions being used in a similar setting, we refer the reader to Wikipedia's blacklist of forbidden article titles.[4]

[3] For example: https://google.github.io/styleguide/javaguide.html#s5-naming.
[4] https://en.wikipedia.org/wiki/MediaWiki:Titleblacklist.

Example 2.9 (Domains, 1). Another common indicator of spam email is when the sender's domain no longer exists. This can be checked using services such as DNS, Whois or PyFunceble [12]. One can then filter email from senders that satisfy the pattern:

$$\begin{aligned} r_{\text{edom}} &= \Sigma_e^+ \ \texttt{@}\ (\Sigma_e^+ \ \text{PD}\ \Sigma_a^{\{1,3\}}) \wedge \langle \text{Domain does not exist} \rangle, \text{ where} \\ \Sigma_e &= \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}, \texttt{0}, \ldots, \texttt{9}, \texttt{-}, \text{PD}\}, \text{ and} \\ \Sigma_a &= \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\}. \end{aligned} \tag{8}$$

Here the notation $r^{\{i,j\}}$ is syntactic sugar for bounded repetition:

$$r^{\{i,j\}} = r^i + r^{i+1} + \cdots + r^j.$$

Example 2.10 (Domains, 2). A third characteristic of spam email is the presence of URLs from phishing domains. Security researchers make repositories of these domains publicly available. See, for example, https://openphish.com/. When scanning their inbox, one might therefore wish to run the SemRE:

$$r_{\text{wdom},1} = (\texttt{http(s?)://} + \texttt{www}\ \text{PD})\ (\Sigma_e^+ \ \text{PD}\ \Sigma_a^{\{1,3\}}) \wedge \langle \text{Phishing domain} \rangle. \tag{9}$$

We use the usual syntactic sugar $r?$ for $r + \epsilon$.

In other situations, one might be interested in recent domains, say those registered after 2010. One can once again use the Whois service to answer this question:

$$r_{\text{wdom},2} = (\texttt{http(s?)://} + \texttt{www}\ \text{PD})\ (\Sigma_e^+ \ \text{PD}\ \Sigma_a^{\{1,3\}}) \wedge \langle \text{Domain registered after 2010} \rangle. \tag{10}$$

Example 2.11 (Foreign IP addresses.). While analyzing some packet capture data, a security researcher might be interested in mentions of IP addresses that are outside the company's intranet. They might use their knowledge of the local network topology to set up an oracle and look for matches for the following SemRE:

$$r_{\text{ip}} = ((\Sigma_d^{\{1,3\}}\ \text{PD})^3\ \Sigma_d^{\{1,3\}}) \wedge \langle \text{Foreign IP address} \rangle, \tag{11}$$

where $\Sigma_d = \{\texttt{0}, \ldots, \texttt{9}\}$.

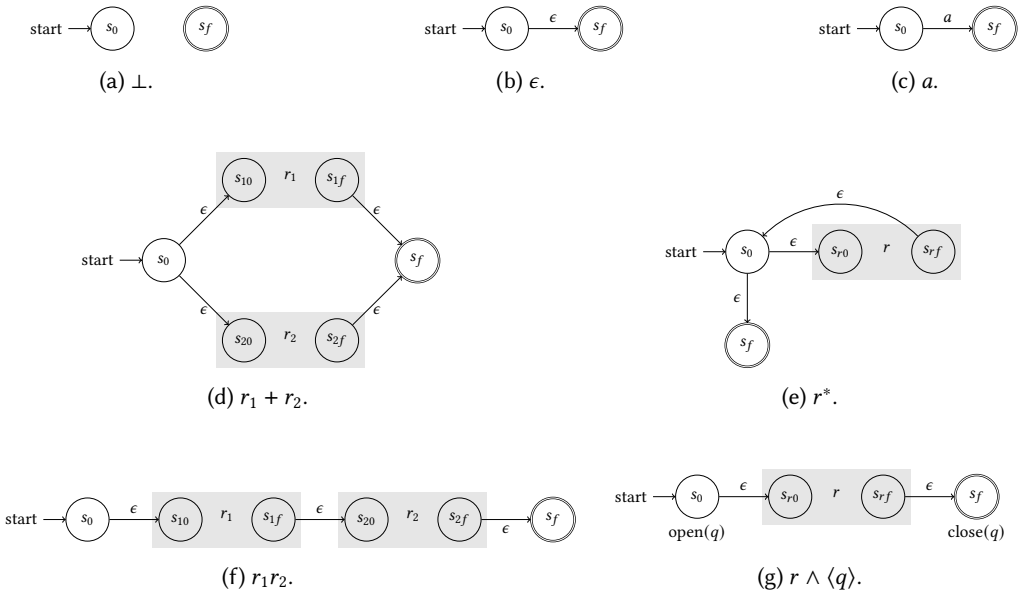
3 Matching Semantic Regular Expressions Using Query Graphs

We will now present an algorithm to determine whether $w \in [\![r]\!]$ for a given SemRE r and input string w. Our procedure conceptually proceeds in two passes: It first uses an NFA-like algorithm to identify syntactic structure in w. Notably, we do not discharge any non-trivial oracle queries during this pass, but instead construct a data structure called a query graph which encodes the Boolean structure among the results of these oracle queries. The second pass uses dynamic programming to evaluate the query graph, discharge the necessary oracle queries, and produce the final result.

3.1 Converting SemREs into Semantic NFAs

We start with a straightforward extension of classical NFAs: A *semantic NFA* (SNFA) is a 5-tuple, $M = (S, \Delta, \lambda, s_0, s_f)$, where: (*a*) S is a finite set of states, (*b*) $\Delta \subseteq S \times (\Sigma \cup \{\epsilon\}) \times S$ is the transition relation, (*c*) $\lambda : S \to \{\text{open}, \text{close}\} \times Q \cup \{\text{blank}\}$ is the state labeling function, (*d*) $s_0, s_f \in S$ are the start and end states respectively, and such that: (*e*) for each sequence of transitions, $s_0 \to s_1 \to s_2 \to \cdots \to s_f$, the corresponding sequence of state labels $\lambda(s_0)\lambda(s_1)\lambda(s_2)\cdots\lambda(s_f)$ forms a well-parenthesized string ($|Q|$ different types of parentheses, one for each query in Q).

Remark 3.1. The main difference with classical NFAs is the state labeling $\lambda(s)$. We use this to mark substrings that subsequently need to be submitted to the oracle for validation. The semantics of SemREs in Equation 2 imply that these substrings are well-nested. Well-parenthesized structures



(a) $\bot$. (b) ϵ. (c) a.

(d) $r_1 + r_2$. (e) r^*.

(f) $r_1 r_2$. (g) $r \wedge \langle q \rangle$.

Fig. 1. Recursive construction of the semantic NFA M_r given a SemRE r. The states s_0 and s_f of $M_{r \wedge \langle q \rangle}$ are respectively labelled with the open and close query markers for q. The formal construction may be found in Appendix A.

will consequently play a central role in our algorithm, including in the above definition of SNFAs and in the definition of query graphs in Section 3.2.

We will write $s \to^a s'$ to indicate the transition $(s, a, s') \in \Delta$. When the state label $\lambda(s)$ is of the form (open, q) or (close, q), we instead write $\text{open}(q)$ and $\text{close}(q)$ respectively.

Given a SemRE r, we construct an SNFA M_r by following a traditional Thompson-like approach. We visualize this construction in Figure 1, but postpone its formal description to Appendix A.1. To argue its correctness, we need to reason about (feasible) paths through the SNFA. A path $\pi = s_1 \to^{w_1} s_2 \to^{w_2} s_3 \to^{w_3} \cdots \to^{w_n} s_{n+1}$, for $n \geq 0$, consists of an initial state s_1, and a (possibly empty) sequence of transitions emerging from s_1. The corresponding string $\text{str}(\pi)$ is the sequence of characters, $\text{str}(\pi) = w_1 w_2 w_3 \ldots w_n$. (Informally,) We say that the path π is *feasible* if the oracle accepts all its queries. Formally, π is feasible if any of the following conditions hold:

(1) $\pi = s_1$ is the empty path where $\lambda(s_1) = \text{blank}$, or
(2) $\pi = s_1 \to s_2 \to \cdots \to s_n \to s_{n+1}$, where $\lambda(s_1) = \text{open}(q)$ and $\lambda(s_{n+1}) = \text{close}(q)$ for some query q, the subpath $s_2 \to \cdots \to s_n$ is feasible, and $\mathcal{O}(q, \text{str}(\pi)) = \text{true}$, or
(3) $\pi = \underbrace{s_1 \to \cdots \to s_k}_{\pi_1} \to \underbrace{s_{k+1} \to \cdots \to s_n}_{\pi_2}$, where both subpaths π_1 and π_2 are feasible.

We say that a string w is accepted by an SNFA M if there a feasible w-labelled path $\pi = s_0 \to \cdots \to s_f$. The following theorem asserts the correctness of the constructed SNFA M_r:

Theorem 3.2. *Pick a SemRE r and let M_r be the SNFA resulting from the construction of Figure 1. For each string w, $w \in [\![r]\!]$ iff M_r accepts w.*

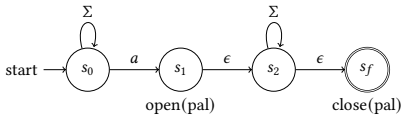


Fig. 2. SNFA which accepts strings of the form $\Sigma^* a\langle \text{pal} \rangle$. Assume the query pal recognizes palindromes. The machine nondeterministically finds an occurrence of the character a, and confirms that the subsequent suffix is a palindrome.

Proof sketch. The forward direction can be proved by induction on the derivation of $w \in [\![r]\!]$. To prove the converse, we start by observing that whenever a combined path

$$\pi = \underbrace{s_1 \longrightarrow \cdots \longrightarrow s_n}_{\pi_1} \longrightarrow \underbrace{s_{n+1} \longrightarrow \cdots \longrightarrow s_{n+m}}_{\pi_2}$$

is feasible and both subpaths π_1 and π_2 are well-parenthesized, then π_1 and π_2 are both themselves feasible paths. The final proposition may then be proved by induction on r, and by case analysis on the presented proof of feasibility. □

3.2 The Query Graph Data Structure

Given a SemRE r and string w, our goal is to determine whether $w \in [\![r]\!]$. So far, our approach of converting r into a corresponding SNFA M_r has been fairly standard. Per Theorem 3.2, we just need to determine whether a feasible w-labelled path exists from the initial to the final states of M_r. In the classical setting, we would do this by maintaining the set of reachable states, and determining whether the accepting state s_f was reachable. More formally, we would iteratively define the set of reachable states S_w after processing w as:

$$S_\epsilon = \{s_1 \in S \mid s_0 \rightarrow^{\epsilon^*} s_1\}, \text{ and}$$
$$S_{wa} = \{s_3 \in S \mid \exists s_1 \in S_w, s_2 \in S \text{ such that } s_1 \rightarrow^{a} s_2 \rightarrow^{\epsilon^*} s_3\},$$

where $s \rightarrow^{\epsilon^*} s'$ denotes the transitive closure of ϵ-transitions in the NFA. When r is a (classical) regular expression, $w \in [\![r]\!]$ iff $s_f \in S_w$.

The complication (Part 1). The classical algorithm fundamentally relies on the following Markovian property of NFA executions: One only needs to know $S_w \subseteq S$ and w' to determine whether the combined string $ww' \in [\![r]\!]$. On the other hand, consider the SNFA in Figure 2, which accepts strings of the form $r_{\text{pal}} = \Sigma^* a\langle \text{pal} \rangle$, and assume the query pal accepts palindromic strings. Consider the prefixes:

$$w_1 = babc \text{ and } w_2 = bacb.$$

After processing these prefixes, the machine may be in any of the states:

$$S_{w_1} = S_{w_2} = \{s_0, s_2, s_f\}.$$

Now consider the common suffix,

$$w_3 = cb.$$

Observe that the combined string $w_1 w_3 = babccb \in [\![r_{\text{pal}}]\!]$, while $w_2 w_3 = bacbcb \notin [\![r_{\text{pal}}]\!]$. It follows that the SemRE membership testing algorithm needs to maintain additional information beyond just the frontier S_w of currently reachable states.

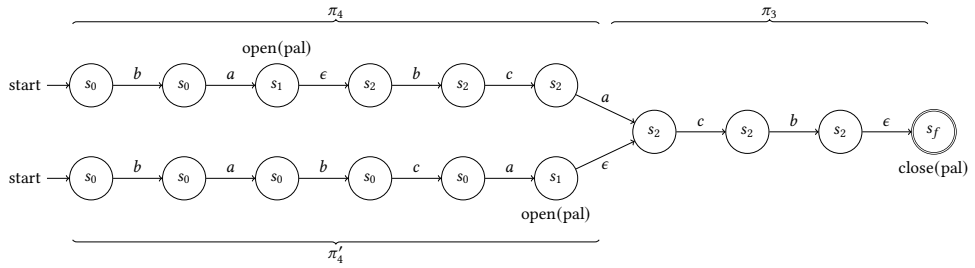


Fig. 3. Two prefix paths π_4 and π'_4 from the initial state s_0 to the intermediate state s_2. The SNFA in question is the one from Figure 2. Both paths correspond to the same string $w_4 = babca$. The suffix path π_3 moves the machine from s_2 to the final state s_f along the string $w_3 = cb$. Notice that the combined path $\pi_4\pi_3$ is feasible (so the machine accepts $w_4w_3 = babcacb$) but $\pi'_4\pi_3$ is not. The SNFA evaluation algorithm therefore needs to track the indices at which queries were opened along each path through the automaton.

The complication (Part 2). Even for a single string, different paths leading to the same state may behave differently. For example, consider the string $w_4 = babca$, so that

$$w_4w_3 = ba\overbrace{bca\underbrace{cb}_{\cancel{\langle \text{pal} \rangle}}}^{\langle \text{pal} \rangle} \in [\![r_{\text{pal}}]\!].$$

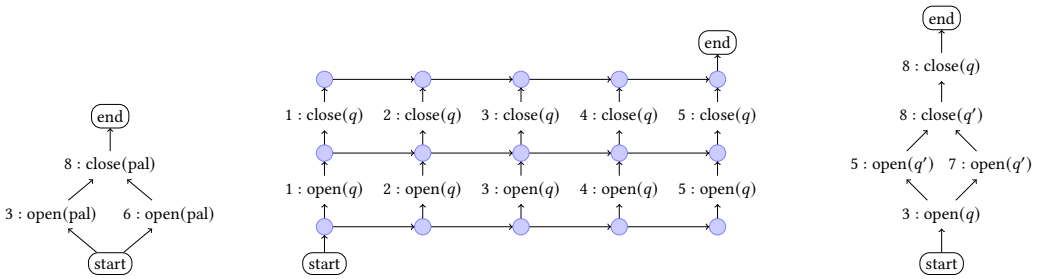
The machine may nondeterministically take the $s_0 \xrightarrow{a} s_1$ transition either on the first or the second occurrence of the symbol a, resulting the two paths π_4 and π'_4 in Figure 3. Both of these paths can then be extended using the path π_3 (corresponding to the suffix w_3). Notice that the combined path $\pi_4\pi_3$ is feasible, while $\pi'_4\pi_3$ is not feasible, even though both π_4 and π'_4 end at the same state s_2. Both paths π_4 and π'_4 have an outstanding open query; the difference is the position in the string at which this query was opened. For each path, the SNFA evaluation algorithm therefore also needs to maintain the string indices at which queries were opened and closed.

Our solution. Our solution is to unroll the SNFA while processing the input string. We maintain a compact representation of all possible paths in a data structure called the query graph. Fix a string $w = w_1w_2 \dots w_n$, and recall that Q is the set of all possible oracle queries. Formally, a *query graph*, $G = (V, E, \text{start}, \text{end}, \text{idx}, l)$, is a directed acyclic graph (DAG) with vertices V and edges $E \subseteq V \times V$, with identified initial and final vertices, $\text{start}, \text{end} \in V$, and whose vertices are associated with string indices, $\text{idx} : V \to \{1, 2, \dots, n+1\}$ and labelled using $l : V \to \{\text{open}, \text{close}\} \times Q \cup \{\text{blank}\}$ such that for each path p from start to end:

(1) The index labels are monotonically increasing along p, and
(2) the open and close marks and associated queries form a well-parenthesized string.

The query graph is essentially an unrolling of the SNFA. Therefore, as before, we define its semantics in terms of the feasibility of its paths. Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path through the query graph starting from a specified state v_1. We say that p is feasible if any of the following conditions hold:

(1) $p = v_1$ is the empty path, where $l(v_1) = \text{blank}(i_1)$ for some i_1, or
(2) $p = v_1 \to v_2 \to \cdots \to v_k \to v_{k+1}$, where:
  (a) the subpath $v_2 \to \cdots \to v_k$ is feasible, and
  (b) $l(v_1) = \text{open}(q)$ and $l(v_{k+1}) = \text{close}(q)$ for some query $q \in Q$, and
  (c) $\text{🔮}(q, w_iw_{i+1} \cdots w_{j-1}) = \text{true}$, where $i = \text{idx}(v_1)$ and $j = \text{idx}(v_{k+1})$, or



(a) G_1. (b) G_2. (c) G_3.

Fig. 4. Examples of query graphs. 4a: Corresponding to possible parse trees for the string $w_4w_3 = babcacb$ according to the SemRE $r_{\text{pal}} = \Sigma^* a \langle \text{pal} \rangle$. We indicate the labels on each node v by writing $\text{idx}(v) : l(v)$. The path through the open node on the left is feasible if 🔮$(\text{pal}, bcacb) = \text{true}$ and the path on the right is feasible if 🔮$(\text{pal}, cb) = \text{true}$. $[\![G_1]\!] = \text{true}$ if either of these paths is feasible. 4b: Corresponding to the string $w = w_1w_2w_3w_4$ according to the pattern $\Sigma^* \langle q \rangle \Sigma^*$. The unlabelled vertices are all marked blank. Observe how each open node may be delimited by any subsequently reachable matching close node. Our construction in Section 3.3 exploits similar sharing to produce a query graph with only $O(|r||w|)$ vertices. 4c: Query graph with "*nested*" queries. Corresponding to the string $w = babcbc$ and the SemRE $r_{\text{nest}} = \Sigma^* a (\Sigma^* b \langle q' \rangle) \wedge \langle q \rangle$. $[\![G_3]\!] = \text{true}$ iff the Boolean formula 🔮$(q, cbcbc) \wedge$ (🔮$(q', cbc) \vee$ 🔮(q', c)) evaluates to true.

(3) $p = \underbrace{v_1 \longrightarrow \cdots \longrightarrow v_k}_{p_1} \longrightarrow \underbrace{v_{k+1} \longrightarrow \cdots \longrightarrow v_n}_{p_2}$, where both subpaths p_1 and p_2 are feasible.

We associate the entire query graph G with a Boolean value $[\![G]\!] \in \text{Bool}$, such that $[\![G]\!] = \text{true}$ iff there is a feasible path from start to end.

We present some examples of query graphs in Figure 4. Graph G_1 corresponds to the string $w_4w_3 = babcacb$ and the pattern $\Sigma^* a \langle \text{pal} \rangle$ from earlier in this section. In general, notice that the answer to the membership problem, i.e., whether $w \in [\![r]\!]$, depends on some Boolean combination of the results of no more than $O(|r||w|^2)$ oracle queries. By allowing a single close node to demarcate multiple open queries, Figure 4b illustrates how this Boolean function might nevertheless be encoded using a query graph with only $O(|r||w|)$ vertices. Of course, it is not yet clear that small query graphs can be constructed for arbitrary instances of the membership testing problem: We devote Section 3.3 to actually constructing a compact query graph such that $[\![G_r^w]\!]$ is true iff $w \in [\![r]\!]$. It is also not clear how $[\![G]\!]$ can be computed: We show how to solve this problem in Section 3.4.

3.3 Constructing the Query Graph

Given an SNFA M and input string $w \in \Sigma^*$, our goal in this section is to construct a query graph G_M^w such that $[\![G_M^w]\!] = \text{true}$ iff M accepts w. We hope to construct G_M^w by making one left-to-right pass over the string. Furthermore, at a high level, we hope to iteratively maintain a "*frontier*" of vertices in G_M^w corresponding to the reachable states of M. After processing each subsequent character w_i, we create a new "*layer*" of vertices in the growing query graph, and add edges between these last two layers according to the w_i-labelled transitions of M.

As an example, consider the SNFA M_{q^*} shown in Figure 5a, which accepts strings of the form $(\Sigma^* \wedge \langle q \rangle)^*$. Let $w = abc$. Depending on how w is partitioned, it is accepted in one of four different

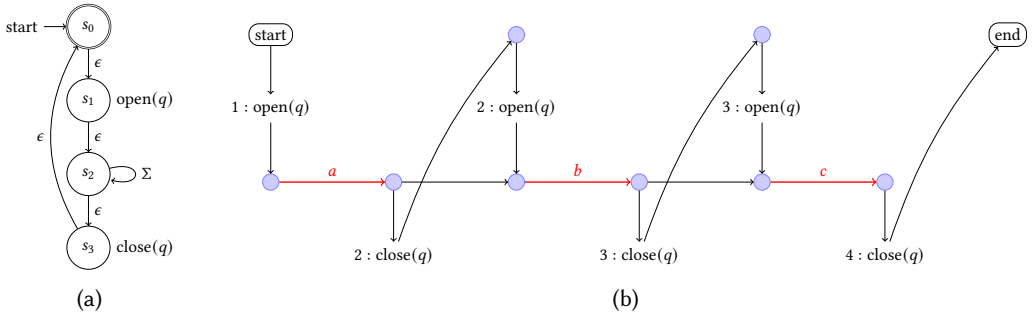


Fig. 5. 5a: The SNFA M_{q^*} accepts strings of the form $(\Sigma^* \wedge \langle q \rangle)^*$. Let $w = abc$. The query graph $G^{abc}_{q^*}$ in Figure 5b describes the four possible ways in Equation 12 by which M_{q^*} might accept w. Note that edges in the query graph are unlabelled: The red coloured labels a, b and c are there simply to hint to the reader that these edges can be thought of as originating from the corresponding $s_2 \to s_2$ transition of the SNFA.

Fig. 6. (Non-)Examples of tentatively well-parenthesized paths. Assume $q_3 \neq q_2$. The path in Figure 6a is tentatively well-parenthesized while the path in Figure 6b is not.

cases: when

$$\left.\begin{aligned} \text{🔮}(w, abc) &= \text{true, or} \\ \text{🔮}(w, a) \wedge \text{🔮}(w, bc) &= \text{true, or} \\ \text{🔮}(w, ab) \wedge \text{🔮}(w, c) &= \text{true, or when} \\ \text{🔮}(w, a) \wedge \text{🔮}(w, b) \wedge \text{🔮}(w, c) &= \text{true.} \end{aligned}\right\} \tag{12}$$

Each of these alternatives corresponds to a different path through the SNFA. Observe that these alternatives differ only in the sequence of ϵ-transitions exercised after processing each character, which result in the query q being closed (and possibly immediately reopened) at different locations in the input string. The query graph $G^{abc}_{q^*}$ in Figure 5b collectively summarizes the effect of these alternatives: The edges labelled a, b and c in red indicate new layers of the query graph that are added after each subsequent character is processed. These are straightforward to add. The remaining edges all correspond to ϵ-transitions in the SNFA. The technical meat of this section is therefore in designing an *inter-character gadget* which summarizes the effect of ϵ-transitions.

3.3.1 Preliminaries: Tentatively Feasible Subpaths, Query Contexts, and ϵ-Feasibility. Before discussing the gadget, we note that although every complete $s_0 \to^* s_f$ path is well-parenthesized, its subpaths might have unmatched close and open states. We start by investigating the structure of these subpaths.

Let $\pi = s_1 \to s_2 \to \cdots \to s_k$ be a path through the SNFA starting at a specified state s_1. We say that π is *tentatively well-parenthesized* if the sequence of state labels $\lambda(\pi) = \lambda(s_1)\lambda(s_2)\ldots\lambda(s_k)$ is a substring of some well-parenthesized string. For example, the path in Figure 6a is tentatively well-parenthesized but the one in Figure 6b is not.

Given a tentatively well-parenthesized path π, let σ_c be the list of its unbalanced close nodes, and let σ_o be the list of its unbalanced open nodes. For example, for the path π_1 in Figure 6a,

$\sigma_{c,1} = [\text{close}(q_1)]$ and $\sigma_{o,1} = [\text{open}(q_3)]$. Given a tentatively well-parenthesized path π, we refer to the pair (σ_c, σ_o) as its *query context*, which we will denote by $\text{qcon}(\pi) = (\sigma_c, \sigma_o)$.

We then note, without elaboration, that it is possible to compute the query context $\text{qcon}(\pi)$ of a combined path:

$$\pi = \underbrace{s_1 \rightarrow^* s_2}_{\pi_1} \rightarrow \underbrace{s_3 \rightarrow^* s_4}_{\pi_2},$$

merely by consulting $\text{qcon}(\pi_1)$ and $\text{qcon}(\pi_2)$.

Now, recall that all paths from the start state s_0 to the end state s_f in the SNFA are well-parenthesized. A natural consequence is that all paths through M (regardless of origin and destination) are also tentatively well-parenthesized. Furthermore, for each state s, and for each pair of paths $\pi = s_0 \rightarrow^* s$ and $\pi' = s_0 \rightarrow^* s$, $\text{qcon}(\pi) = \text{qcon}(\pi')$. Because all paths to a given state s share the same query context, we can now speak of the query context $\text{qcon}(s)$ of individual states s, without identifying any particular path $\pi = s_0 \rightarrow^* s$.

Assumption 3.3. We assume that every state s in M is both reachable from the start state, $s_0 \rightarrow^* s$, and can itself reach the final state, $s \rightarrow^* s_f$. The SNFA M_r constructed in Section 3.1 automatically satisfies this property if r does not contain any occurrence of $\perp$. Such sub-expressions can be eliminated by the application of simple rewrite rules and using the observation that $w \notin [\![\perp]\!]$.

We say that a tentatively well-parenthesized path π through the SNFA is *tentatively feasible* if the oracle accepts all of its balanced queries. Similar ideas of tentative well-parenthesization, query contexts, and tentative feasibility can also be developed for query graphs.

Finally, we note that the feasibility of well-parenthesized ϵ-labelled paths depends on responses from the oracle for queries of the form (q, ϵ). Before constructing the gadget, we therefore begin by determining all pairs of states, (s_1, s_2), such that there exists an ϵ-labelled feasible path from s_1 to s_2. In this case, we will write $s_1 \rightarrow_F^{\epsilon^*} s_2$. The set of all such pairs can be calculated using a graph search algorithm in $O(|r|^2)$ time. The details of this algorithm can be found in Appendix A.2.1.

3.3.2 The Inter-Character Gadget. The purpose of the gadget is to summarize ϵ-labelled paths through the SNFA. As an example, consider the SNFA in Figure 7a, and say that we are currently in state s_5. Before processing the next character, the machine might choose to either stay in the same state, or close and immediately reopen one or both of the queries q_1 and q_2. Because of the well-parenthesized property of query opens and closes, q_1 must be closed *before* q_2 is closed, and can only be reopened *after* q_2 is reopened. This immediately suggests a three layer structure, where queries are sequentially closed in Layer 1, (re-)opened in Layer 2, and where the last layer performs any remaining ϵ-transitions that do not affect query context. We visualize the resulting gadget in Figure 7b, and highlight the possible taken from state s_5 in Figure 7c.

We formally define the gadget Gad as the triple:

$$\begin{aligned}
\text{Gad} &= (V_{\text{Gad}}, E_{\text{Gad}}, l_{\text{Gad}}), \text{ where} \\
V_{\text{Gad}} &= S \times \{1, 2, 3\}, \\
E_{\text{Gad}} &= E_{\text{Gad},11} \cup E_{\text{Gad},12} \cup E_{\text{Gad},22} \cup E_{\text{Gad},23}, \text{ and} \\
l_{\text{Gad}}(v) &= \begin{cases} \text{close}(q) & \text{if } v = (s,1) \text{ and } \lambda(s) = \text{close}(q), \\ \text{open}(q) & \text{if } v = (s,2) \text{ and } \lambda(s) = \text{open}(q), \text{ and} \\ \text{blank} & \text{otherwise.} \end{cases}
\end{aligned} \tag{13}$$

Here, $E_{\text{Gad},11}$, $E_{\text{Gad},12}$, $E_{\text{Gad},22}$, and $E_{\text{Gad},23}$ refer to edges between the respective layers. Informally, edges in $E_{\text{Gad},11}$ result in the progressive closing of currently open queries and $E_{\text{Gad},22}$ progressively

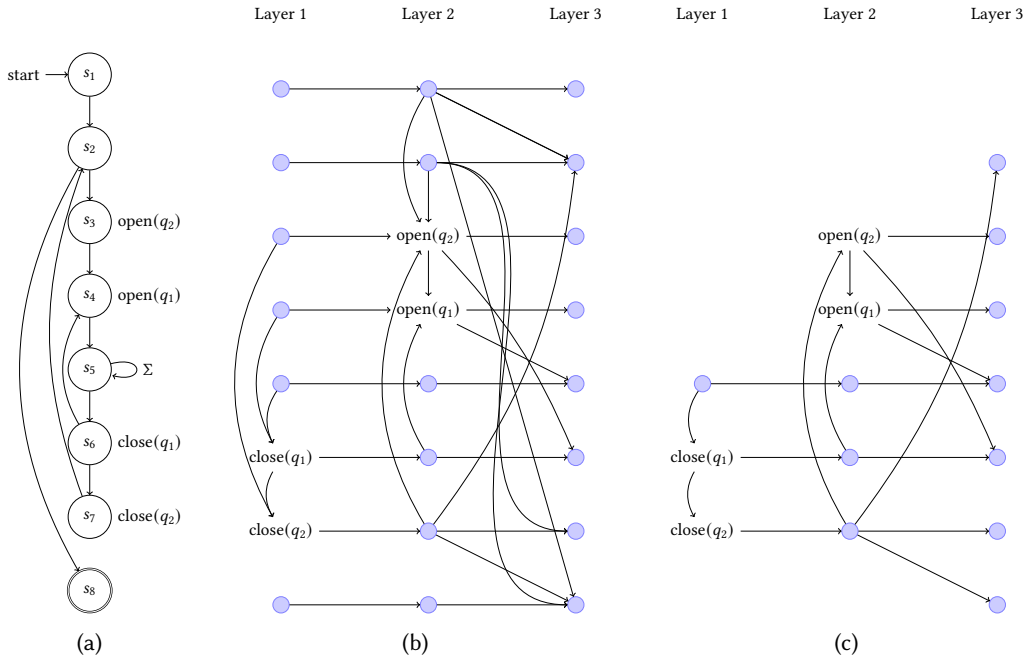


Fig. 7. Construction of the gadget to summarize ϵ-paths. The SNFA in Figure 7a accepts strings of the form $(\langle q_1 \rangle^* \wedge \langle q_2 \rangle)^*$. Assume for the sake of this example that ϵ is accepted by all queries. There is a path to s_5 after processing each character w_i of the input string. The query context of s_5 is $[\text{open}(q_2), \text{open}(q_1)]$. Before processing the next character, w_{i+1} using the $s_5 \longrightarrow s_5$ transition, the machine might make four choices: (*a*) stay in the same state, or (*b*) close and immediately reopen the query q_1, by following the sequence of transitions, $s_5 \longrightarrow s_6 \longrightarrow s_4 \longrightarrow s_5$, or (*c*) close and immediately reopen both queries: $s_5 \longrightarrow s_6 \longrightarrow s_7 \longrightarrow s_2 \longrightarrow^* s_5$, or (*d*) move to the final state s_8. These alternatives are described in Figure 7c. Observe how the gadget in Figure 7b describes these and all other eventualities from the states of the SNFA.

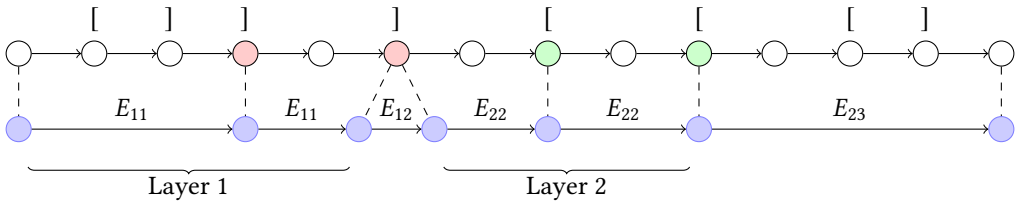


Fig. 8. Schematic procedure for how tentatively feasible ϵ-paths are broken down across the three layers of the gadget. All edges indicate ϵ transitions, and brackets,] and [indicate close and open nodes respectively. Unmatched have been shaded red and green. The edges in Layers 1 and 2 of the gadget summarize sequences of transitions between subsequent pairs of shaded nodes.

(re-)opens queries. The direct connections in $E_{\text{Gad},12}$ provides shortcut connections to bypass these reopening actions. The last set of edges, $E_{\text{Gad},23}$ summarizes ϵ-labelled feasible paths that do not visibly change the query context. These edges are formally defined as follows:

$$E_{\text{Gad},11} = \left\{ (s,1) \longrightarrow (s''',1) \;\middle|\; \begin{array}{l} s, s', s'', s''' \in S, q \in Q \text{ such that} \\ s' \longrightarrow_F^{\epsilon^*} s'' \text{ and } s \longrightarrow^{\epsilon} s' \text{ and } s'' \longrightarrow^{\epsilon} s''' \text{ and } \lambda(s''') = \text{close}(q) \end{array} \right\} \cup$$

$$\left\{(s,1) \to (s''',1) \;\middle|\; \begin{array}{l} s, s''' \in S, q \in Q \text{ such that} \\ s \to^{\epsilon} s'' \text{ and } \lambda(s''') = \text{close}(q) \end{array} \right\},$$

$$E_{\text{Gad},12} = \{(s,1) \to (s,2) \mid s \in S\},$$

$$E_{\text{Gad},22} = \left\{(s,2) \to (s''',2) \;\middle|\; \begin{array}{l} s, s', s'', s''' \in S, q \in Q \text{ such that} \\ s' \to_F^{\epsilon^*} s'' \text{ and } s \to^{\epsilon} s' \text{ and } s'' \to^{\epsilon} s''' \text{ and } \lambda(s''') = \text{open}(q) \end{array} \right\} \cup$$

$$\left\{(s,2) \to (s''',2) \;\middle|\; \begin{array}{l} s, s''' \in S, q \in Q \text{ such that} \\ s \to^{\epsilon} s'' \text{ and } \lambda(s''') = \text{open}(q) \end{array} \right\},$$

$$E_{\text{Gad},23} = \{(s,2) \to (s',3) \mid s, s', s'' \in S \text{ such that } s \to_F^{\epsilon^*} s'' \text{ and } s'' \to^{\epsilon} s' \text{ or } s = s'\}.$$

Our central claim about this construction is that there is a correspondence between ϵ-labeled tentatively feasible paths in the SNFA and paths in the gadget. We formalize and prove this claim in Lemma A.3 of Appendix A.2.2. We schematically visualize the breakdown of tentatively feasible ϵ paths across the three layers of the gadget in Figure 8.

3.3.3 The Final Query Graph Construction. Let $w = w_1 w_2 \cdots w_n$, and recall that S is the set of states in the SNFA. We construct the final query graph G_M^w by tiling $n+1$ copies of the gadget, and connecting each pair of adjacent gadget instances, $i-1$ and i, based on the transitions made by M while processing the character w_i:

$$\left.\begin{aligned} G_M^w &= (V, E, \text{start}, \text{end}, \text{idx}, l), \text{ where} \\ V &= S \times \{1,2,3\} \times \{1,2,\ldots,n+1\} \\ &= \{(s,k,i) \mid (s,k) \in V_{\text{Gad}} \text{ and } 1 \le i \le n+1\}, \text{ and} \\ E &= E_c \cup E_\epsilon, \text{ where} \\ E_c &= \{(s,3,i) \to (s',1,i+1) \mid s, s' \in S \text{ and transition } s \to^{w_i} s' \text{ for } 1 \le i \le n\}, \\ E_\epsilon &= \{(s,k,i) \to (s',k',i) \mid (s,k) \to (s',k') \in E_{\text{Gad}}, 1 \le i \le n+1\}, \text{ and} \\ \text{start} &= (s_0, 1, 1), \\ \text{end} &= (s_f, 3, n+1), \\ \text{idx}(s,k,i) &= i, \text{ and} \\ l(s,k,i) &= l_{\text{Gad}}(s,k). \end{aligned}\right\} \quad (14)$$

The following lemmas establish a connection between paths in M and through G_M^w. Their proofs may be found in Appendix A.2.3.

Lemma 3.4 (Path correspondence, Part 1). *Pick a path $\pi = s_0 \to^* s$ through M. If π is tentatively feasible, then there is also a tentatively feasible path $p = \text{start} \to^* (s, 3, |\text{str}(\pi)| + 1)$ through G_M^w with the same query context.*

Lemma 3.5 (Path correspondence, Part 2). *Pick a path $p = \text{start} \to^* v$ in G_M^w, where $v = (s,3,i)$ for $s \in S$. There is a tentatively well-parenthesized path $\pi = s_0 \to^* s$ in the SNFA with the same query context, and such that if p is tentatively feasible, then so is π.*

From Lemma 3.5, it follows that each path $p = \text{start} \to^* \text{end}$ is well-parenthesized. It is also easy to see that string indices monotonically increase along each edge $e \in E$, and that G_M^w is a DAG. It follows that G_M^w is a well-formed query graph. The following theorem is now a direct consequence of Lemmas 3.4 and 3.5, and establishes the correctness of the query graph:

Theorem 3.6. *M accepts w iff $[\![G_M^w]\!] = \text{true}$.*

$$\textsc{As}\ \frac{}{\mathrm{Alive}(\mathrm{start})} \qquad \textsc{Ab}\ \frac{l(v) = \mathrm{blank} \quad v' \to v \quad \mathrm{Alive}(v')}{\mathrm{Alive}(v)}$$

$$\textsc{Ao}\ \frac{l(v) = \mathrm{open}(q) \quad v' \to v \quad \mathrm{Alive}(v')}{\mathrm{Alive}(v)} \qquad \textsc{Ac}\ \frac{l(v) = \mathrm{close}(q) \quad \mathrm{Matched}(v) \neq \emptyset}{\mathrm{Alive}(v)}$$

$$\textsc{M}\ \frac{l(v) = \mathrm{close}(q) \quad v' \to v \quad \mathrm{Alive}(v') \quad v'' \in \mathrm{Backref}(v') \quad \text{🔮}(q, w_{\mathrm{idx}(v'')} \cdots w_{\mathrm{idx}(v)-1})}{v'' \in \mathrm{Matched}(v)}$$

$$\textsc{Bb}\ \frac{l(v) = \mathrm{blank} \quad v' \to v \quad \mathrm{Alive}(v')}{\mathrm{Backref}(v') \subseteq \mathrm{Backref}(v)} \qquad \textsc{Bo}\ \frac{l(v) = \mathrm{open}(q) \quad \mathrm{Alive}(v)}{v \in \mathrm{Backref}(v)}$$

$$\textsc{Bc}\ \frac{l(v) = \mathrm{close}(q) \quad \mathrm{Alive}(v) \quad v' \in \mathrm{Matched}(v)}{\mathrm{LOQ}(v') \subseteq \mathrm{Backref}(v)}$$

Fig. 9. Inference rules for evaluating the query graph. $\mathrm{LOQ}(v')$ is defined in Equation 17. Rule Bc has a flavor similar to the calculation of the grandparents of a vertex in a graph. Repeated application of this rule is responsible for the dominant $O(|r||w|^3)$ term in the final time complexity of our procedure. The rule is vacuous for the case of non-nested SemREs, because $\mathrm{LOQ}(v') = \emptyset$.

3.4 Evaluating Query Graphs Using Dynamic Programming

Given a SemRE r and a string w, we sequentially translate r into an SNFA M (using Figure 1) and construct a query graph G (using Equation 14). Our final goal is now to evaluate $[\![G]\!]$. Recall that all paths from start to end are well-parenthesized, that a start–end path is feasible if the oracle accepts all of its queries, and that the entire graph G evaluates to true if there is a feasible path. We now show how to determine this using dynamic programming.

We start by introducing a few new properties of vertices in the query graph: First, we say that a vertex v in G is *alive* if there exists a tentatively feasible path from start to v. Naturally:

Theorem 3.7. $[\![G]\!] = \mathrm{true}$ *iff* $\mathrm{Alive}(\mathit{end}) = \mathrm{true}$.

We present inference rules for determining the living nodes of a query graph in Figure 9. Rules As, Ab and Ao (for the starting node, blank and open nodes respectively) are immediate. On the other hand, upon encountering a close node v with $l(v) = \mathrm{close}(q)$, we need to discharge the queries that have just been fully delimited. In principle: we must recall all tentatively feasible paths leading to v, find the last unclosed open vertex on each of these paths, and discharge these queries. We do this efficiently by also propagating the set of *backreferences* during query graph evaluation.

Let v be a vertex in G, and consider some tentatively feasible path $p = \mathrm{start} \to^* v$. We define the backreference of p, $\mathrm{Backref}(p)$, as its last unclosed open vertex. The backreference of the vertex v is simply defined as the set of backreferences along each tentatively feasible path to v:

$$\mathrm{Backref}(v) = \{\mathrm{Backref}(p) \mid \text{Tentatively feasible path } p = \mathrm{start} \to^* v\}. \tag{15}$$

Upon encountering a close node, v, we consult the backreferences $\mathrm{Backref}(v')$ at each of its immediate predecessors v', and discharge the relevant queries. Formally, let $p = \mathrm{start} \to^* v$ be a tentatively feasible path with $l(v) = \mathrm{close}(q)$. We define $\mathrm{Matched}(p)$ as the matching open for this last vertex v. Aggregating this quantity along all tentatively feasible paths leading to v, we write:

$$\mathrm{Matched}(v) = \{\mathrm{Matched}(p) \mid \text{Tentatively feasible path } p = \mathrm{start} \to^* v\}. \tag{16}$$

To complete the calculation of Alive(v), we simply note that if v is a close node, then v is alive iff Matched(v) is non-empty (see Rules M and Ac).

The remaining rules in Figure 9 focus on computing the backreferences. Once again, the calculation for blank and open nodes is straightforward. See Rules Bb and Bo. Lastly, let v be a living close node, and consider how one might calculate its backreferences. Informally, we have just popped the last open query from the query context, and we need to "*bring forward*" the backreferences from each immediate predecessor of the popped open vertices. This motivates the definition:

$$\mathrm{LOQ}(v') = \bigcup_{v'' \to v'} \mathrm{Backref}(v''), \tag{17}$$

and its use in Rule Bc. At this point, the following result is straightforward:

Lemma 3.8. *The inference rules in Figure 9 are sound and complete.*

One may therefore mechanically compute Alive(v), Backref(v) and Matched(v) by repeatedly applying the evaluation rules until fixpoint. This completes our description of the SemRE membership testing procedure.

3.5 Performance Analysis of Our Algorithm

Let r be a semantic regular expression, and $w \in \Sigma^*$ be the input string. Let M_r be the SNFA constructed using Figure 1, and $G^w_{M_r}$ be the resulting query graph. The proof of the following claim be found in Appendix A.3. In short, it follows from bounding the sizes of the auxiliary sets, Backref(v), Matched(v) and LOQ(v) and the observation that the query graph is a DAG and a subsequent accounting of the cost of applying each of the inference rules from Figure 9 to each of the vertices in the query graph.

Theorem 3.9. *Assume the oracle responds in unit time. We can determine whether $w \in [\![r]\!]$ in $O(|r|^2|w|^2 + |r||w|^3)$ time and with at most $O(|r||w|^2)$ calls to the oracle. If r does not contain nested queries, then the running time of our algorithm is $O(|r|^2|w|^2)$. Furthermore, if* skel(r) *is unambiguous, then the algorithm runs in $O(|r|^2|w|)$ time and discharges $O(|r||w|)$ oracle queries.*

Here, we write skel(r) ("*skeleton*") for the underlying classical regular expression, which may be calculated by stripping away all oracle refinements occurring within the SemRE r. Also recall that a (classical) regular expression r is unambiguous if every string admits a single parse tree [6].

4 The Complexity of Matching Semantic Regular Expressions

We will now establish some (admittedly simple) lower bounds on the complexity of the membership testing problem. Our first result will show that any SemRE matching algorithm needs to make at least quadratic (in the string length) number of oracle invocations in the worst case. Next, in Section 4.2, we will reduce the problem of finding triangles in graphs to that of matching SemREs.

4.1 Query Complexity

Note that we have so far not made any assumptions about the set of possible oracle queries Q. Our first result can be stated in one of two forms, depending on whether Q is finite or not:

Theorem 4.1. *If the set of queries $Q = \{q_1, q_2, \dots\}$ is unbounded, then $\Omega(|r||w|^2)$ oracle invocations are necessary to determine whether $w \in [\![r]\!]$. If the query space Q is finite but non-empty, then $\Omega(|w|^2)$ oracle invocations are necessary to determine whether $w \in [\![r]\!]$.*

The proof in Appendix B essentially fleshes out the observation that matching $\Sigma^*\langle q\rangle\Sigma^*$ requires us to examine *every* substring of w.

Note 4.2. Note that Theorem 4.1 only describes the asymptotic growth of the worst-case query complexity. On the other hand, specific SemREs might admit more efficient matching algorithms. For example, whether a string w matches $(\Sigma \wedge \langle q \rangle)\Sigma^*$ can be determined with only a single oracle query, by checking whether $\mathcal{O}(q, w_1) = \text{true}$ (where w_1 is the first letter of w).

Note 4.3. We also note that Theorem 4.1 only holds in the black-box setting when one is not allowed to make any further assumptions about the oracle. In specific situations, such as when the oracle is a set of strings or performs a computation that is easy to characterize, it might be possible to perform additional optimizations so that these lower bounds no longer hold.

4.2 Membership Testing is as Hard as Finding Triangles

The gap between the worst case time complexity of the SNFA algorithm and the lower bound of Theorem 4.1 is frustrating. We therefore conclude this section with a defense of our algorithm.

Recall that the $O(|r||w|^3)$ term arises because of the time needed to compute the updated set of backreferences at each close node in the query graph. This calculation is essentially multiplying the Boolean vector listing backreferences from the current node with the matrix containing the global set of backreferences. This observation led us to compare the SemRE membership testing problem with problems in Boolean matrix multiplication and graph theory.

Let $G = (V, E)$ be an undirected graph with $V = \{v_1, v_2, \ldots, v_n\}$. We say that G has a *triangle* if there are distinct vertices $v_i, v_j, v_k \in V$ with edges $\{v_i, v_j\}, \{v_j, v_k\}, \{v_k, v_i\} \in E$. We will now construct a SemRE r_Δ and a string w_G such that $w_G \in [\![r_\Delta]\!]$ exactly when G has a triangle.

Let $\Sigma = \{1, 2, \ldots, n, \#\}$ be an alphabet with $n + 1$ symbols, and where # is a special delimiter symbol. Punning with the set of edges, let $E \in Q$ be a query such that for non-empty strings $w = w_1 w_2 \ldots w_k \in \Sigma^*$,

$$\mathcal{O}(E, w) = \text{true} \iff \{w_1, w_k\} \in E.$$

Now consider the following SemRE,

$$r_\Delta = \Sigma^*\#\underbrace{(\Sigma \cdot \underbrace{(\Sigma\Sigma^*\#\Sigma) \wedge \langle E \rangle}_{r_{ij}} \cdot \underbrace{(\Sigma\Sigma^*\#\Sigma) \wedge \langle E \rangle}_{r_{jk}} \cdot \Sigma) \wedge \langle E \rangle}_{r_{ik}}\Sigma^*. \tag{18}$$

The marked subexpressions r_{ij}, r_{jk} and r_{ik} respectively identify portions of the string that identify the endpoints of edges $\{v_i, v_j\}$, $\{v_j, v_k\}$, $\{v_k, v_i\}$ respectively. From the semantics of r_Δ and the definition of $\mathcal{O}(E, w)$, it follows that:

Lemma 4.4. *Let $w_G = \#11\#22\#33\cdots\#nn$. Then, $w_G \in [\![r_\Delta]\!]$ iff G contains a triangle.*

One objection to this reduction might be the potentially large alphabet with $|\Sigma| = O(|V|)$. This can be easily remedied by rendering the vertices of V in binary, and rewriting r_Δ and w_G using the ternary alphabet $\Sigma_3 = \{0, 1, \#\}$. From here, it follows that:

Theorem 4.5. *Given a string w, if we can determine whether $w \in [\![r_\Delta]\!]$ in $D(|w|)$ time, then we can determine whether a given graph G contains triangles in $D(|V| \log(|V|))$ time.*

There are triangle finding algorithms which run in $O(|V|^{3-\epsilon})$ time. For example, express the set of edges E as an adjacency matrix and observe that G contains a triangle iff the trace (sum of diagonal elements) of E^3 is non-zero. Still, these procedures rely on algorithms for fast (integer / Boolean) matrix multiplication, and are therefore impractical. No purely "*combinatorial*" algorithms are known that can find triangles in $O(|V|^{3-\epsilon})$ time [41, 42]. We conclude that finding practical algorithms which perform SemRE membership testing in $O(|r|^c|w|^{3-\epsilon})$ time would be very challenging.

Table 1. Benchmark SemREs and their statistics. For the r_{id} SemRE, the regular expressions pad_1 and pad_2 are added as padding to search for identifiers in longer lines of code. Here: $\text{pad}_1 = (\Sigma^*(\Sigma \backslash \Sigma_l))?$ and $\text{pad}_2 = (\Sigma^*(\Sigma \backslash (\Sigma_l \cup \{0, \ldots, 9\})))?$.

Dataset	Name	SemRE	Oracles	$\lvert r \rvert$	# Matched line
Code	pass	$\Sigma^* r_{\text{pass}} \Sigma^*$	LLM	26	$\approx$ 4,330
	file	$\Sigma^* r_{\text{file}} \Sigma^*$	File system	226	8,706
	id	$\text{pad}_1\, r_{\text{id}}\, \text{pad}_2$	LLM	250	$\approx$ 423,165
Spam	edom	$\Sigma^* r_{\text{edom}} \Sigma^*$	Whois	307	365,112
	spam,1	$\Sigma^* r_{\text{spam},1} \Sigma^*$	LLM	31	$\approx$ 46,666
	spam,2	$\Sigma^* r_{\text{spam},2} \Sigma^*$	LLM	86	21,096
	wdom,1	$\Sigma^* r_{\text{wdom},1} \Sigma^*$	Phishing website list	263	192
	wdom,2	$\Sigma^* r_{\text{wdom},2} \Sigma^*$	Whois	263	482
	ip	$\Sigma^* r_{\text{ip}} \Sigma^*$	IP geolocation	153	135,320

Note 4.6. Note the fundamental need for nested queries to express r_Δ: $(\cdots \wedge \langle q_1 \rangle \cdots) \wedge \langle q_2 \rangle$. The above reduction therefore does not contradict the promises of Theorem 3.9 for non-nested SemREs. The Paris Hilton SemRE previously mentioned in the introduction, $(\Sigma^*(\Sigma^* \wedge \langle \text{City} \rangle)\Sigma^*) \wedge \langle \text{Celebrity} \rangle$, is one of the few natural examples of SemREs with nested queries that we can write. We expect that most SemREs arising in practice would be unaffected by the reduction just described.

5 Experimental Evaluation

We conducted an experimental evaluation of the algorithm from Section 3 in which we attempted to answer the following questions:

RQ1. How quickly can the algorithm process input data?
RQ2. How heavily does the algorithm use the oracle?
RQ3. How does the length of the input string affect matching speed?

Benchmark SemREs and input data. We used the SemREs from Section 2.2 to filter lines of text in a corpus of spam emails and a corpus of Java code downloaded from three popular GitHub repositories. We removed lines with non-ASCII characters and lines greater than 1,000 characters in length. After this pruning the two datasets had 4,622,992 and 5,461,188 lines and measured 61 MB and 72 MB respectively. Table 1 presents some statistics of each SemRE, including their size, the backing oracle, and the number of matched lines in the datasets. Because processing the entire datasets with the LLM oracle would require several weeks of computation for r_{pass}, r_{id} and $r_{\text{spam, 1}}$, we only provide estimates of the number of matched lines based on random sampling. Also note that we have padded the SemREs so the system looks for lines of text with substrings that have the desired patterns.

Implementation and baselines. We have implemented our algorithm in a system called grep^O, a grep-like tool for matching semantic regular expressions. We implemented a simple baseline algorithm based on dynamic programming, as discussed in Section 2.1. This algorithm recursively processes subexpressions of the given SemRE r and substrings of the input string w, and uses top-down dynamic programming with memoization to determine whether $w \in [\![r]\!]$. The entire system consists of approximately 1,800 lines of Rust code, and has an interface similar to grep: given the SemRE, oracle, and input file, it prints the matched lines.

Table 2. SemRE matching performance. We report reciprocal throughput (average time needed to process each line) for the two algorithms, and aggregate oracle use statistics.

	RT, Total (ms · line^{-1})		RT, Matched (ms · line^{-1})		Oracle calls (line^{-1})		Oracle fraction		Query length (chars · line^{-1})	
SemRE	SNFA	DP	SNFA	DP	SNFA	DP	SNFA	DP	SNFA	DP
pass	**38.2**	464.2	**891.0**	3624.0	**0.125**	1.767	0.999	0.999	**1.93**	23.895
file	≤ **0.1**	1.7	**0.1**	5.5	0.001	0.001	0.001	≤ 0.001	**0.033**	0.036
id	**175.7**	1569.7	**311.0**	1545.5	**3.101**	20.541	0.999	0.999	**22.218**	111.249
ip	≤ **0.1**	159.3	**0.1**	10.8	0.113	**0.095**	0.162	≤ 0.001	1.205	**1.016**
edom	≤ **0.1**	16.9	**0.3**	2.9	0.09	**0.082**	0.417	≤ 0.001	0.993	**0.651**
spam,1	7.9	**6.3**	763.5	**525.0**	**0.032**	0.037	0.999	0.891	0.892	**0.386**
spam,2	**0.4**	23.5	**39.5**	1728.1	**0.016**	0.117	0.987	0.960	**0.076**	0.415
wdom,1	**0.2**	35.9	**0.4**		0.424	**0.186**	0.925	0.003	6.15	**2.616**
wdom,2	≤ **0.1**	2.6	**0.4**	7.3	**0.025**	0.055	0.351	0.002	**0.24**	0.523

Implementing the underlying oracles required some care: For example, the external Whois service will temporarily block the users if too many queries are submitted in a short period of time. To avoid overloading this service, we instead prepopulated a local database with necessary information for all domains in the input data. We also provided a predefined list of phishing websites, an IP geolocation database, and a filesystem interface. For SemREs using an LLM oracle, we used LLaMa3-8B with 8-bit quantization [23, 28]. Because the throughput of the LLM is low, and to mitigate issues with nondeterminism (Recall Assumption 2.4), its use was mediated by a query cache.

5.1 RQ1: Efficiency in Processing Input Data

To measure the effectiveness of our algorithm in processing semantic regular expressions, we compared its throughput and that of the baseline DP-based implementation for each SemRE from Table 1. We present the results in Table 2. For each semantic regular expressions, we report the reciprocal throughput (average time needed to process each line) for both the entire run and when restricted to lines that match r. Because of the low throughput of the baseline and of the LLM-based oracle, we set a timeout of 40 minutes for each run and report statistics obtained for the text processed within this period.

Overall, we observe that grep^O has a much higher throughput both for the file as a whole, and also when restricted to matched lines. Although this speedup varies depending on the SemRE in question, averaged across all SemREs (using geometric means), and compared to the DP-based baseline, grep^O has 101× the throughput for the whole dataset, and 12× the throughput when focusing on the matched lines in particular.

We notice that this speedup is lower when the oracle that has a very long response time (i.e., for the LLM-based oracle). This is unsurprising, and is caused by the fact that performance for these SemREs is dominated by the response time of the oracle, as we will see in the next section. Nevertheless, our algorithm still outperforms the baseline in all but one of the LLM-based SemREs. The only exception is $r_{\text{spam, 1}}$ where the DP-baseline has a slightly higher throughput. We speculate that this performance inversion is due the underlying skeleton being too permissive: matching essentially reduces to guessing which substring is accepted by the LLM, so that the simplicity of the baseline algorithm results in reduced overhead.

5.2 RQ2: Extent of Oracle Use

We expect oracle use to have non-zero cost (either in terms of running time, actual cost, or in terms of limits on daily use). We therefore measured how heavily the two algorithms used the oracle in terms of number of calls, time spent within the oracle, and average length of submitted queries.

Once again, we report these statistics in Table 2. Overall, grep^O makes 51% fewer oracle calls than the baseline, which in turn spends 3× the time inside the oracle as our algorithm.

These additional "useless" queries are due to a combination of factors: One reason is because of lines where the skeleton eventually turns out to not match. Recall that the skeleton is the classical regular expression $\mathrm{skel}(r)$ that one may obtain by removing oracle refinements. Naturally, $[\![r]\!] \subseteq [\![\mathrm{skel}(r)]\!]$, so any oracle use for strings $w \notin [\![\mathrm{skel}(r)]\!]$ is wasted work. A second reason is due to a lack of lookahead even for lines where the skeleton matches. For example, in the case of the spam medication SemRE, while the baseline would correctly use whitespaces on the left to demarcate the beginning of medicine names, it would not be able to exploit the symmetric property of whitespace demarcation on the right.

Note 5.1. As a counterpoint to our entire paper, a critical reader might wonder whether it would make sense to simply submit the entire string to the LLM and suitably phrase the SemRE matching problem as a prompt. Of course, this approach would only work when the backing oracle is an LLM, and would lack the semantic guarantees of SemREs. We additionally point out that even for LLM-backed SemREs, grep^O submits much fewer tokens to the oracle than the average line length, so LLM use is massively reduced.

5.3 RQ3: Scalability with Respect to Input String Length

Finally, we investigate the empirical impact of string length on the throughput of our algorithm. We reran the experiment on the same dataset but filtered out all lines longer than 200 characters. We show the median throughput of the matching algorithms as a function of line length in Figure 10. We also show the overall distribution of line lengths in our dataset.

Given the overall differences in throughput, it is unsurprising that the median runtime of grep^O increases much more slowly compared to the baseline (notice the log-scaling applied to the y-axis). We observe similar trends both for LLM and non LLM oracles. The "*cloud*" of data points appearing to the top right of the plots for r_{id} occur because of cache misses in the oracle implementation leading to actual LLM queries. This cloud of cache misses is particularly acute for this SemRE because identifiers appear in a large fraction of lines.

6 Related Work

SMORE. The story of our paper begins with SMORE [9], which is itself set in a broader context of systems that use LLMs for various data extraction [10, 39] and code generation tasks [24]. Other examples of work in this space include L^*LM [37], which uses an LLM as oracle to guide the L^* algorithm (Angluin's famous procedure for learning DFAs [2]) and systems that extract structured data from language models [5, 18].

The overarching difference between SMORE and our paper is in the focus: Chen et al. [9] consider the problem of synthesizing semantic regular expressions from data, rather than membership testing. Their implementation uses the dynamic programming approach for membership testing that we used as a baseline in Section 5. This is a natural approach to testing membership: an early reference is [15]. In addition, SemREs, as defined by Equations 1 and 2, appear to be simpler than the SMORE formalism (Figures 3 and 4), in a few ways:

(1) SMORE-style SemREs appear to permit string transformations during processing by the oracle. This allows languages such as $\{w \in \Sigma^* \mid w \text{ is a date and } \mathrm{month}(w) = \mathrm{May}\}$ to be represented. Because the backing oracle 🔮 is externally defined and we make no assumptions about the query space Q, they can both be suitably extended so that such languages remain expressible in our simpler core formalism.

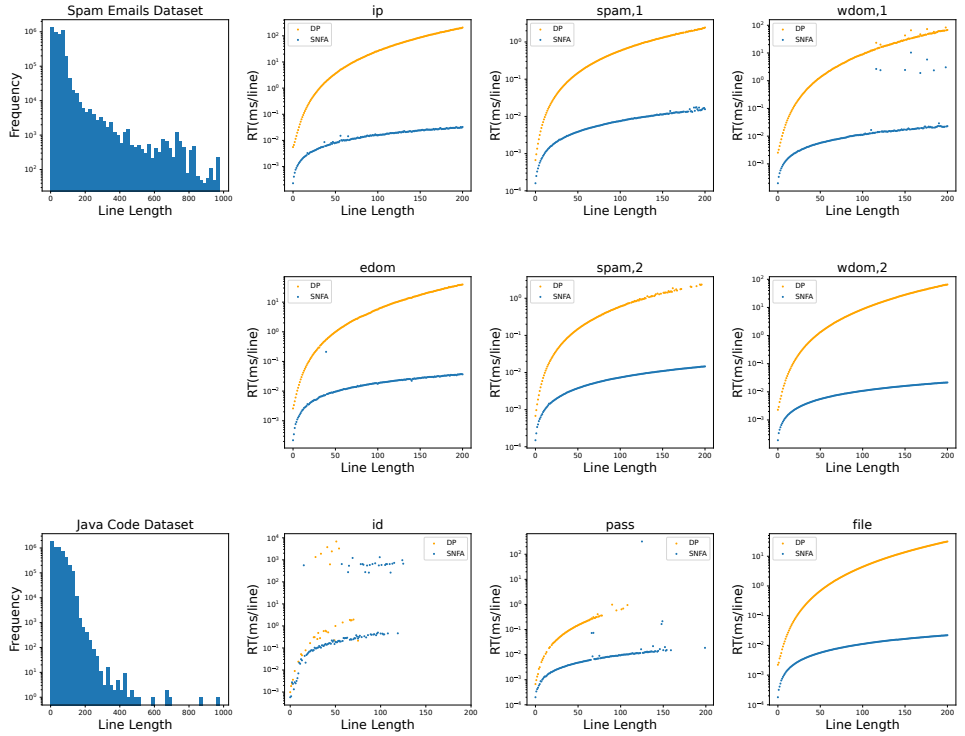


Fig. 10. Line length distribution and variation of running time across the datasets. We only report running time measurements when the algorithm processes at least 10 lines with that length.

(2) In addition to the basic constructors, SMORE-style SemREs also permit negation, intersection, and bounded repetition. Efficient membership testing for languages defined using these operations is significantly harder than for classical regular expressions. See, for example, [19, 30] and [20]. To keep our paper simple, we have ignored these constructs. Handling them would be an excellent direction for future work.

On the other hand, our work is also more general, because the oracle can be realized using any external mechanism. While oracle machines have been the subject of extensive theoretical study, to the best of our knowledge, our paper is the first practical implementation of such ideas.

KAT. Another closely related idea is that of KAT (Kleene algebras with tests) [17]. KAT has a similar flavor as SemREs, with the major difference being in the nature of oracle tests: While KAT allows individual characters of a string to encode uninterpreted tests, SemREs allow entire substrings to be presented to an oracle for validation.

Visibly pushdown languages. The well-nested query structure induced by SemREs is reminiscent of visibly pushdown languages [1]. We highlight some differences: First, visibly pushdown automata (VPAs) require an explicitly tagged alphabet of calls, returns, and local actions. As a consequence, the locations where the stack is pushed an popped is obvious (i.e., "*visible*") upon seeing the input string. On the other hand, identifying positions where queries are opened and closed is the key computational problem associated with SemREs. Think of the $\Sigma^*\langle\text{Politician}\rangle\Sigma^*$ pattern. As a consequence, for a fixed VPA M, we only require $O(|w|)$ time to determine whether M accepts an input string w. This is in opposition to our $\Omega(|w|^2)$ lower bound of Theorem 4.1. A second

difference is that VPAs allow for unbounded call depths, whereas the query context in an SNFA can never be deeper than the nesting depth.

Membership testing for classical and extended regexes. The problem of membership testing for regular expressions has also been extensively studied. We refer the reader to Cox's excellent survey on the topic [11]. Practical implementations use a range of techniques, including backtracking [13], automata simulation [14, 33, 40], and various forms of derivatives [3, 7]. Extended regular expressions, i.e., with intersection and complement, are of particular interest to us, because of the similarities between intersection, $r_1 \cap r_2$, and the way oracle access is mediated in SemREs: $r \wedge \langle q \rangle$. It is now folklore that these operations do not add expressive power to classical regular expressions. Still, efficient membership testing algorithms for this class is challenging, as extended regular expressions are non-elementarily more succinct than NFAs: We refer the reader to [30].

Petersen [27] showed that the problem of regular expressions with intersection is LOGCFL-complete. The reduction in Section 4.2 shows SemRE membership testing is similarly related to several other problems of interest, including triangle finding, Boolean matrix multiplication [41], context-free grammar (CFG) parsing [21], and Dyck reachability [8, 26]. These connections also raise the possibility of applying techniques such as the Four Russians Method [4] or ideas similar to Valiant's reduction of CFG parsing to Boolean matrix multiplication [35] to develop asymptotically faster algorithms. Of course, this reduction does not work well in practice, so developing practical subcubic algorithms would be challenging.

Derivative-based algorithms. Derivatives [3, 7] provide an alternative characterization of regular expression matching algorithms. Given a regular expression r and an initial character a, the (Brzozowski) derivative $\partial_a(r)$ is another regular expression that describes possible suffixes that extend this initial character: $[\![\partial_a(r)]\!] = \{w' \mid aw' \in [\![r]\!]\}$. For example, it can be seen that $\partial_a((ab)^*) = b(ab)^*$. There is a purely syntactic approach to calculating derivatives, and this greatly simplifies implementation. Naturally, one can ask whether similar derivative-based approaches can be applied to the setting of SemREs. We think that this is a good direction of future work. They might even allow us to gracefully handle extended regex operators, in a manner similar to [36]. However, while derivatives have promise to simplify some parts of our algorithm, we anticipate the core difficulties—i.e., nested queries and associated hardness results—will remain.

7 Conclusion

In this paper, we studied the membership testing problem for semantic regular expressions (SemREs). We described an algorithm, analyzed its performance both theoretically and experimentally, and proved some lower bounds and hardness results. We think that SemREs are an exciting formalism, and that their applications and associated computational problems deserve greater scrutiny.

Artifact Availability Statement

The artifact that supports the findings of this paper can be downloaded from Zenodo [16].

Acknowledgements

This research was supported in part by the US National Science Foundation awards CCF 2313062, CCF 2146518 and CCF 2107261. We thank our anonymous reviewers for their feedback, which helped to greatly improve this paper.

References

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 202–211. doi:10.1145/1007352.1007390

[2] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[3] Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319. doi:10.1016/0304-3975(95)00182-4

[4] Vladimir Arlazarov, Yefim A. Dinitz, M. A. Kronrod, and Igor Faradzhev. 1970. On Economical Construction of the Transitive Closure of an Oriented Graph. *Soviet Mathematics Doklady* 11 (1970), 1209–1210.

[5] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is Programming: A Query Language for Large Language Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.

[6] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Comput.* C-20, 2 (1971), 149–153. doi:10.1109/T-C.1971.223204

[7] Janusz Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. doi:10.1145/321239.321249

[8] Jakob Cetti Hansen, Adam Husted Kjelstrøm, and Andreas Pavlogiannis. 2021. Tight Bounds for Reachability Problems on One-Counter and Pushdown Systems. *Inform. Process. Lett.* 171 (2021), 106135. doi:10.1016/j.ipl.2021.106135

[9] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 287 (Oct. 2023), 30 pages. doi:10.1145/3622863

[10] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah Smith, and Tao Yu. 2023. Binding Language Models in Symbolic Languages. In *The Eleventh International Conference on Learning Representations*. OpenReview.net. https://openreview.net/forum?id=lH1PV42cbF

[11] Russ Cox. 2010. *Implementing Regular Expressions*. Technical Report.

[12] Nissar Chababy et al. 2023. *PyFunceble*. https://github.com/funilrys/PyFunceble

[13] Philip Hazel et al. 1997. *PCRE - Perl Compatible Regular Expressions*. https://www.pcre.org/

[14] Russ Cox et al. 2010. *RE2: A Principled Approach to Regular Expression Matching*. https://opensource.googleblog.com/2010/03/re2-principled-approach-to-regular.html https://github.com/google/re2.

[15] John Hopcroft and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[16] Yifei Huang, Matin Amini, Alexis Le Glaunec, Konstantinos Mamouras, and Mukund Raghothaman. 2025. *Membership Testing for Semantic Regular Expressions*. doi:10.5281/zenodo.15048870

[17] Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Transactions on Programming Languages and Systems* 19, 3 (1997), 427–443. doi:10.1145/256167.256195

[18] Michael Kuchnik, Virginia Smith, and George Amvrosiadis. 2023. Validating Large Language Models with ReLM. *Proceedings of Machine Learning and Systems* 5 (2023), 457–476.

[19] Orna Kupferman and Sharon Zuhovitzky. 2002. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In *Mathematical Foundations of Computer Science*. Springer, 446–458.

[20] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (Oct. 2023), 30 pages. doi:10.1145/3586044

[21] Lillian Lee. 2002. Fast Context-Free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *J. ACM* 49, 1 (2002), 1–15.

[22] Lee McMahon. 1974. *sed*. AT&T Bell Laboratories.

[23] Meta. 2024. *LlaMa3*. Retrieved 11 Jul, 2024 from https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. OpenReview.net. https://openreview.net/forum?id=iaYcJKpY2B_

[25] OpenAI. 2024. *Pricing*. Retrieved 18 May, 2024 from https://openai.com/api/pricing/

[26] Andreas Pavlogiannis. 2023. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (feb 2023), 5–25. doi:10.1145/3583660.3583664

[27] Holger Petersen. 2002. The Membership Problem for Regular Expressions with Intersection Is Complete in LOGCFL. In *STACS*. Springer, 513–522.

[28] QuantFactory. 2024. *LlaMa3*. Retrieved 11 Jul, 2024 from https://huggingface.co/QuantFactory/Meta-Llama-3-8B-GGUF

[29] Zach Rice. 2022. *Gitleaks*. https://gitleaks.io/

[30] Grigore Roşu. 2007. An Effective Algorithm for the Membership Problem for Extended Regular Expressions. In *Foundations of Software Science and Computation Structures (FoSSaCS) (Lecture Notes in Computer Science, Vol. 4423)*. Springer, 332–345. doi:10.1007/978-3-540-71389-0_24

[31] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 372–378.

[32] Michael Sipser. 2012. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.
[33] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. doi:10.1145/363347.363387
[34] Ken Thompson. 1973. *grep*. AT&T Bell Laboratories.
[35] Leslie Valiant. 1975. General Context-Free Recognition in Less than Cubic Time. *J. Comput. System Sci.* 10, 2 (1975), 308–315. doi:10.1016/S0022-0000(75)80046-8
[36] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proceedings of the ACM on Programming Languages* 9, POPL, Article 1 (2025), 32 pages. doi:10.1145/3704837
[37] Marcell Vazquez-Chanlatte, Karim Elmaaroufi, Stefan Witwicki, and Sanjit Seshia. 2024. *L*LM: Learning Automata from Examples using Natural Language Oracles*. Technical Report. arXiv:2402.07051 [cs.LG]
[38] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. 2012. Symbolic Finite State Transducers: Algorithms and Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 137–150. doi:10.1145/2103656.2103674
[39] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-trained Models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 100 (2021), 25 pages. doi:10.1145/3485477
[40] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang
[41] Virginia Vassilevska Williams and Ryan Williams. 2010. Subcubic Equivalences between Path, Matrix and Triangle Problems. In *IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 645–654. doi:10.1109/FOCS.2010.67
[42] Huacheng Yu. 2018. An Improved Combinatorial Algorithm for Boolean Matrix Multiplication. *Information and Computation* 261 (2018), 240–247. doi:10.1016/j.ic.2018.02.006 ICALP 2015.

Received 2024-11-15; accepted 2025-03-06