From Regular Expression Matching to Parsing

Philip Bille
Technical University of Denmark, DTU Compute, Denmark
phbi@dtu.dk

Inge Li Gørtz
Technical University of Denmark, DTU Compute, Denmark
inge@dtu.dk

Abstract

Given a regular expression R and a string Q, the regular expression parsing problem is to determine if Q matches R and if so, determine how it matches, e.g., by a mapping of the characters of Q to the characters in R. Regular expression parsing makes finding matches of a regular expression even more useful by allowing us to directly extract subpatterns of the match, e.g., for extracting IP-addresses from internet traffic analysis or extracting subparts of genomes from genetic data bases. We present a new general techniques for efficiently converting a large class of algorithms that determine if a string Q matches regular expression R into algorithms that can construct a corresponding mapping. As a consequence, we obtain the first efficient linear space solutions for regular expression parsing.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases regular expressions, finite automata, regular expression parsing, algorithms

Digital Object Identifier 10.4230/LIPIcs.MFCS.2019.71

Related Version A full version of the paper is available at `https://arxiv.org/abs/1804.02906`

Funding Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178)

1 Introduction

A regular expression specifies a set of strings formed by characters combined with concatenation, union (`|`), and Kleene star (`*`) operators. For instance, `(a|(ba))*` describes the set of strings of `a`s and `b`s, where every `b` is followed by an `a`. Regular expressions are a fundamental concept in formal language theory and a basic tool in computer science for specifying search patterns. Regular expression search appears in diverse areas such as internet traffic analysis [14, 27, 17], data mining [11], data bases [19, 21], computational biology [23], and human computer interaction [16].

Given a regular expression R and a string Q, the *regular expression parsing problem* [15, 8, 10, 24, 25, 18] is to determine if Q matches a string in the set of strings described by R and if so, determine how it matches by computing the corresponding sequence of positions of characters in R, i.e., the mapping of each character in Q to a character in R corresponding to the match. For instance, if $R =$ `(a|(ba))*` and $Q =$ `aaba`, then Q matches R and $1, 1, 2, 3$ is a corresponding parse specifying that $Q[1]$ and $Q[2]$ match the first `a` in R, $Q[3]$ match the `b` in R, and $Q[4]$ match the last `a` in R[1]. Regular expression parsing makes finding matches of a regular expression even more useful by allowing us to directly extract subpatterns of the match, e.g., for extracting IP-addresses from internet traffic analysis or extracting subparts of genomes from genetic data bases.

[1] Another typical definition of parsing is to compute a parse tree (or a variant thereof) of the derivation of Q on R. Our definition simplifies our presentation and it is straightforward to derive a parse tree from our parses.

To state the existing bounds, let n and m be the length of the string and the regular expression, respectively. As a starting point consider the simpler *regular expression matching problem*, that is, to determine if Q matches a string in the set of strings described by R (without necessarily providing a mapping from characters in Q to characters in R). A classic textbook algorithm to matching, due to Thompson [26], constructs and simulates a non-deterministic finite automaton (NFA) in $O(nm)$ time and $O(m)$ space. An immediate approach to solve the parsing problem is to combine Thompson's algorithm with backtracking. To do so, we store all state-sets produced during the NFA simulation and then process these in reverse order to recover an accepting path in the NFA matching Q. From the path we then immediately obtain the corresponding parse of Q since each transition labeled by a character uniquely corresponds to a character in R. This algorithm uses $O(nm)$ time and space. Hence, we achieve the same time bound as matching but increase the space by an $\Omega(n)$ factor. We can improve the time by polylogarithmic factors using faster algorithms for matching [22, 3, 4, 6, 7], but by a recent conditional lower bound [2] we cannot hope to achieve $\Omega((nm)^{1-\varepsilon})$ time assuming the strong exponential time hypothesis. Other direct approaches to regular expression parsing [15, 8, 10, 24, 25, 18] similarly achieve $\Theta(nm)$ time and space (ignoring polylogarithmic factors), leaving a substantial gap between linear space for matching and $\Theta(nm)$ space for parsing. The goal of this paper is to address this gap.

1.1 Results

We present a new technique to efficiently extend the classic state-set transition algorithms for matching to also solve parsing in the same time complexity while only using linear space. Specifically, we obtain the following main result based on Thompson's algorithm:

▶ **Theorem 1.** *Given a regular expression of length m and a string of length n, we can solve the regular expression parsing problem in $O(nm)$ time and $O(n+m)$ space.*

This is the first bound to significantly improve upon the combination of $\Theta(nm)$ time and space. The result holds on a comparison-based, pointer machine model of computation. Our techniques are sufficiently general to also handle the more recent faster state-set transition algorithms [22, 4, 3] and we also obtain a similar space improvement for these.

1.2 Techniques

Our overall approach is similar in spirit to the classic divide and conquer algorithm by Hirschberg [13] for computing a longest common subsequence of two strings in linear space. Let A be the *Thompson NFA* (TNFA) for R built according to Thompson's rules [26] (see also Figure 1) with m states, and let Q be the string of length n.

We first decompose A using standard techniques into a pair of nested subTNFAs called the *inner subTNFA* and the *outer subTNFA*. Each have roughly at most 2/3 of the states of A and overlap in at most 2 boundary states. We then show how to carefully simulate A to decompose Q into substrings corresponding to subparts of an accepting path in each of the subTNFAs. The key challenge here is to efficiently handle cyclic dependencies between the subTNFAs. From this we construct a sequence of subproblems for each of the substrings corresponding to the inner subTNFAs and a single subproblem for the outer subTNFA. We recursively solve these to construct a complete accepting path in A. This strategy leads to an $O(nm)$ time and $O(n \log m + m)$ space solution. We show how to tune and organize the recursion to avoid storing intermediate substrings leading to the linear space solution in Theorem 1. Finally, we show how to extend our solution to obtain linear space parsing solutions for other state-set transition algorithms.

2 Preliminaries

Strings. A string Q of length $n = |Q|$ is a sequence $Q[1] \dots Q[n]$ of n characters drawn from an alphabet Σ. The string $Q[i] \dots Q[j]$ denoted $Q[i, j]$ is called a substring of Q. The substrings $Q[1, i]$ and $Q[j, n]$ are the ith prefix and the jth suffix of Q, respectively. The string ϵ is the unique empty string of length zero.

Regular Expressions. First we briefly review the classical concepts used in the paper. For more details see, e.g., Aho et al. [1]. We consider the set of non-empty regular expressions over an alphabet Σ, defined recursively as follows. If $\alpha \in \Sigma \cup \{\epsilon\}$ then α is a regular expression, and if S and T are regular expressions then so is the *concatenation*, $(S) \cdot (T)$, the *union*, $(S)|(T)$, and the *star*, $(S)^*$. The *language* $L(R)$ generated by R is defined as follows. If $\alpha \in \Sigma \cup \{\epsilon\}$, then $L(\alpha)$ is the set containing the single string α. If S and T are regular expressions, then $L(S \cdot T) = L(S) \cdot L(T)$, that is, any string formed by the concatenation of a string in $L(S)$ with a string in $L(T)$, $L(S)|L(T) = L(S) \cup L(T)$, and $L(S^*) = \bigcup_{i \geq 0} L(S)^i$, where $L(S)^0 = \{\epsilon\}$ and $L(S)^i = L(S)^{i-1} \cdot L(S)$, for $i > 0$. The *parse tree* $\mathcal{T}^P(R)$ of R (not to be confused with the parse of Q wrt. to R) is the rooted, binary tree representing the hierarchical structure of R. The leaves of $\mathcal{T}^P(R)$ are labeled by a character from Σ or ϵ and internal nodes are labeled by either $\cdot$, $|$, or $*$.

Finite Automata. A *finite automaton* is a tuple $A = (V, E, \Sigma, \theta, \phi)$, where V is a set of nodes called *states*, E is a set of directed edges between states called *transitions* either labeled ϵ (called ϵ-transitions) or labeled by a character from Σ (called character-transitions), $\theta \in V$ is a *start state*, and $\phi \in V$ is an *accepting state*[2]. In short, A is an edge-labeled directed graph with a special start and accepting node. A is a *deterministic finite automaton* (DFA) if A does not contain any ϵ-transitions, and all outgoing transitions of any state have different labels. Otherwise, A is a *non-deterministic automaton* (NFA). When we deal with multiple automatons, we use a subscript $_A$ to indicate information associated with automaton A, e.g., θ_A is the start state of automaton A.

Given a string Q and a path P in A we say that Q and P *match* if the concatenation of the labels on the transitions in P is Q. Given a state s in A we define the *state-set transition* $\delta_A(s, Q)$ to be the set of states reachable from s through paths matching Q. For a set of states S we define $\delta_A(S, Q) = \bigcup_{s \in S} \delta_A(s, Q)$. We say that A *accepts* the string Q if $\phi_A \in \delta_A(\theta_A, Q)$. Otherwise A *rejects* q. For an accepting path P in A, we define the *parse* of P for A to be the sequence of character transitions in A on P. Given a string Q accepted by A, a parse of Q is a parse for A of any accepting path matching Q.

We can use a sequence of state-set transitions to test acceptance of a string Q of length n by computing a sequence of state-sets $S_0, \dots, S_n$, given by $S_0 = \delta_A(\theta_A, \epsilon)$ and $S_i = \delta_A(S_{i-1}, Q[i])$, $i = 1, \dots, n$. We have that $\phi_A \in S_n$ iff A accepts Q. We can extend the algorithm to also compute the parse of Q for A by processing the state-sets in reverse order to recover an accepting path and output the character transitions. Note that for matching we only need to store the latest two state-sets at any point to compute the final state-set S_n, whereas for parsing we store the full sequence of state-sets.

Thompson NFA. Given a regular expression R, we can construct an NFA accepting precisely the strings in $L(R)$ by several classic methods [20, 12, 26]. In particular, Thompson [26] gave the simple well-known construction shown in Figure 1. We will call an NFA constructed

[2] Sometimes NFAs are allowed a *set* of accepting states, but this is not necessary for our purposes.

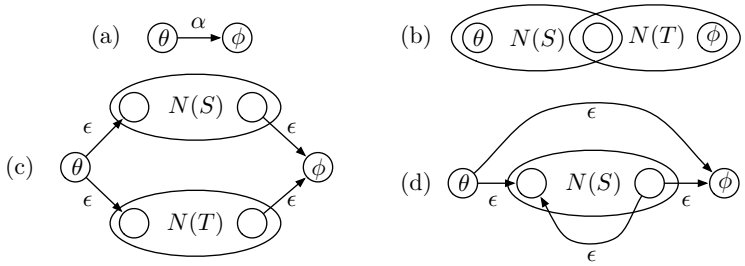

Figure 1 Thompson's recursive NFA construction. The regular expression $\alpha \in \Sigma \cup \{\epsilon\}$ corresponds to NFA (a). If S and T are regular expressions then $N(ST)$, $N(S|T)$, and $N(S^*)$ correspond to NFAs (b), (c), and (d), respectively. In each of these figures, the leftmost state θ and rightmost state ϕ are the start and the accept nodes, respectively. For the top recursive calls, these are the start and accept states of the overall automaton. In the recursions indicated, e.g., for $N(ST)$ in (b), we take the start state of the subautomaton $N(S)$ and identify with the state immediately to the left of $N(S)$ in (b). Similarly the accept state of $N(S)$ is identified with the state immediately to the right of $N(S)$ in (b).

with these rules a *Thompson NFA* (TNFA). A TNFA $N(R)$ for R has at most $2m$ states, at most $4m$ transitions, and can be computed in $O(m)$ time. Note that each character in R corresponds to a unique character transition in $N(R)$ and hence a parse of a string Q for $N(R)$ directly corresponds to a parse of Q for R. The parse tree of a TNFA $N(R)$ is the parse tree of R. With a breadth-first search of A we can compute a state-set transition for a single character in $O(m)$ time. By our above discussion, it follows that we can solve regular expression matching in $O(nm)$ time and $O(m)$ space, and regular expression parsing in $O(nm)$ time and $O(nm)$ space.

TNFA Decomposition. We need the following decomposition result for TFNAs (see Figure 2). Similar decompositions are used in [22, 3]. Given a TNFA A with $m > 2$ states, we decompose A into an *inner subTNFA* A_I and an *outer subTNFA* A_O. The inner subTNFA consists of a pair of *boundary states* θ_{A_I} and ϕ_{A_I} and all states and transitions that are reachable from θ_{A_I} without going through ϕ_{A_I}. Furthermore, if there is a path of ϵ-transitions from ϕ_{A_I} to θ_{A_I} in A_O, we add an ϵ-transition from ϕ_{A_I} to θ_{A_I} in A_I (following the rules from Thompson's construction). The outer subTNFA is obtained by removing all states and transitions of A_I except θ_{A_I} and ϕ_{A_I}. Between θ_{A_I} and ϕ_{A_I} we add a special transition labeled $\beta_{A_I} \notin \Sigma$ and if A_I accepts the empty string we also add an ϵ-transition (corresponding to the regular expression $(\beta_{A_I} \mid \epsilon)$). The decomposition has the following properties. Similar results are proved in [22, 3] (see also full version [5] for a the proof).

▶ **Lemma 2.** *Let A be any TNFA with $m > 2$ states. In $O(m)$ time we can decompose A into inner and outer subTNFAs A_O and A_I such that*

(i) *A_O and A_I have at most $\frac{2}{3}m + 8$ states each, and*
(ii) *any path from A_O to A_I crosses θ_{A_I} and any path from A_I to A_O crosses ϕ_{A_I}.*

3 String Decompositions

Let A be a TNFA decomposed into subTNFAs A_O and A_I and Q be a string accepted by A. We show how to efficiently decompose Q into substrings corresponding to subpaths matched in each subTNFA. The algorithm will be a key component in our recursive algorithm in the next section.

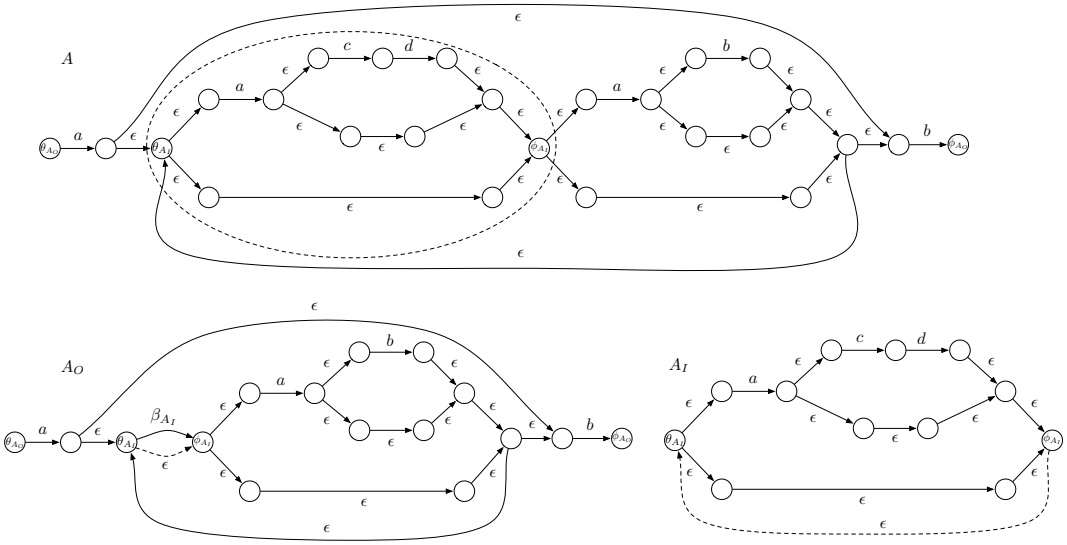

Figure 2 Decomposition of TNFA A into subTNFAs A_O and A_I. The dotted ϵ-transition in A_O exists since A_I accepts the empty string, and the dotted ϵ-transition in A_I exists since there is a path of ϵ-transitions from ϕ_{A_I} to θ_{A_I}.

Given an accepting path P in A, we define the *path decomposition* of P wrt. A_I to be the partition of P into a sequence of subpaths $\mathcal{P} = \overline{p}_1, p_1, \overline{p}_2, p_2, \ldots, \overline{p}_\ell, p_\ell, \overline{p}_{\ell+1}$, where the *outer subpaths*, $\overline{p}_1, \ldots, \overline{p}_{\ell+1}$, are subpaths in A_O and the *inner subpaths*, $p_1, \ldots, p_\ell$ are the subpaths in A_I. The *string decomposition* induced by $\mathcal{P}$ is the sequence of substrings $\mathcal{Q} = \overline{q}_1, q_1, \overline{q}_2, q_2, \ldots, \overline{q}_\ell, q_\ell, \overline{q}_{\ell+1}$ formed by concatenating the labels of the corresponding subpath in A. A sequence of substrings is a substring decomposition wrt. to A_I if there exists an accepting path that induces it. Our goal is to compute a string decomposition in $O(nm)$ time and $O(n+m)$ space, where n is the length of Q and m is the number of states in A.

An immediate idea would be to process Q from left to right using state-set transitions and "collapse" the state set to a boundary state b of A_I whenever the state set contains b and there is a path from b to ϕ_A matching the rest of Q. Since A_O and A_I only interact at the boundary states, this effectively corresponds to alternating the simulation of A between A_O and A_I. However, because of potential cyclic dependencies from paths of ϵ-transition from ϕ_{A_I} to θ_{A_I} in A_O and θ_{A_I} to ϕ_{A_I} in A_I we cannot immediately determine which subTNFA we should proceed in and hence we cannot correctly compute the string decomposition. For instance, consider the string $Q = aaacdaabaacdacdaabab$ from Figure 3. After processing the first two characters (aa) both θ_{A_I} and ϕ_{A_I} are in the state set, and there is a path from both these states to ϕ_A matching the rest of Q. The same is true after processing the first six characters ($aaacda$). In the first case the substring consisting of the next three characters (acd) only matches a path in A_I, whereas in the second case the substring consisting of the next two characters (ab) only matches a path in A_O. A technical contribution in our algorithm in the next section is to efficiently overcome these issues by a two-step approach that first decomposes the string into substrings and labels the substrings greedily to find a correct string decomposition.

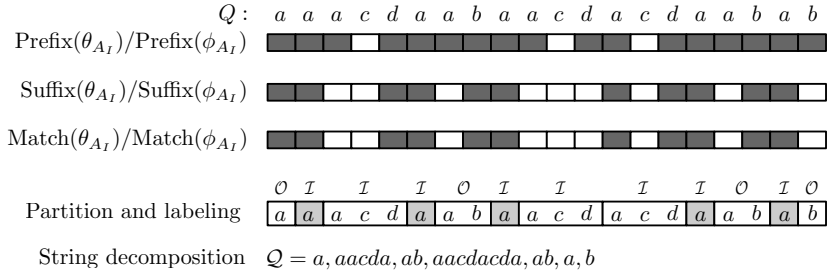

Figure 3 The string decomposition of the string $Q = aaacdaabaacdacdaabab$ wrt. A_I in Figure 2 and the corresponding suffix/prefix match sets. The dark grey blocks in the prefix, suffix and match sets are the positions contained in the sets. The blocks in the partition of the string are labeled $\mathcal{O}$ and $\mathcal{I}$ for outer and inner, respectively. The grey blocks in the partition are the substrings that can be parsed by both the inner and outer automaton. According to our procedure these blocks are labeled *inner*.

3.1 Computing String Decompositions

We need the following new definitions. Let i be a position in Q and let s be a state in A. We say that (i, s) is a *valid pair* if there is a path from θ_A to s matching $Q[1, i]$ and from s to ϕ_A matching $Q[i + 1, n]$. For any set of states X in A, we say that (i, X) is a valid pair if each pair (i, x), $x \in X$, is a valid pair. An accepting path P in A *intersects* a valid pair (i, X) if some state $x \in X$ is on the path, the subpath of P from θ_A to x matches $Q[1, i]$, and the subpath of P from x to ϕ_A matches $Q[i + 1, n]$.

Our algorithm consist of the following steps. In step 1, we process Q from left to right and right to left to compute and store the *match sets*, consisting of all valid pairs for the boundary states θ_{A_I} and ϕ_{A_I}. We then use the match sets in step 2 to process Q from left to right to build a sequence of valid pairs for the boundary states that all intersect a single accepting path P in A matching Q, and that has the property that all positions where the accepting path P contains θ_{A_I} or ϕ_{A_I} correspond to a valid pair in the sequence. Finally, in step 3 we construct the string decomposition using a greedy labeling of the sequence of valid pairs. See Figure 3 for an example of the computation in each step.

Step 1: Computing Match Sets

First, compute the *match sets* given by

$$\text{Match}(\theta_{A_I}) = \{i \mid (i, \theta_{A_I}) \text{ is a valid pair}\}$$
$$\text{Match}(\phi_{A_I}) = \{i \mid (i, \phi_{A_I}) \text{ is a valid pair}\}$$

Thus, $\text{Match}(\theta_{A_I})$ and $\text{Match}(\phi_{A_I})$ are the positions in Q that correspond to a valid pair for the boundary states θ_{A_I} and ϕ_{A_I}, respectively. To compute these, we first compute the *prefix match sets*, $\text{Prefix}(s)$, and *suffix match sets*, $\text{Suffix}(s)$, for $s \in \{\theta_{A_I}, \phi_{A_I}\}$. A position i is in $\text{Prefix}(s)$ if there is a path from θ_A to s accepting the prefix $Q[1, i]$, and in $\text{Suffix}(s)$ if there is a path from s to ϕ_A accepting the suffix $Q[i + 1, n]$. To compute the prefix match sets we perform state-set transitions on Q and A and whenever the current state-set contains either θ_{A_I} or ϕ_{A_I} we add the corresponding position to $\text{Prefix}(s)$. We compute the suffix match sets in the same way, but now we perform the state-set transitions on Q from right to left and A with the direction of all transitions reversed. Each step of the state-set transition takes $O(m)$ time and hence we use $O(nm)$ time in total.

Finally, we compute the match sets Match(s), for $s \in \{\theta_{A_I}, \phi_{A_I}\}$, by taking the intersection of Prefix(s) and Suffix(s). In total we use $O(mn)$ time and $O(n+m)$ space to compute and store the match sets.

Step 2: Computing Valid Pairs

We now compute a sequence of valid pairs

$$V = (i_1, X_1), (i_2, X_2), \ldots, (i_k, X_k)$$

such that $0 \leq i_1 < \cdots < i_k \leq n$ and $X_j \subseteq \{\theta_{A_I}, \phi_{A_I}\}$ and with the property that the states of all pairs intersect a single accepting path P in A and at all places where P is equal to either θ_{A_I} or ϕ_{A_I} correspond to a valid pair in V.

To compute the sequence we run a slightly modified state-set transition algorithm: For $i = 0, 1, \ldots, n$ we set $S_i = \delta_A(S_{i-1}, Q[i])$ (for $i = 0$ set $S_0 := \delta_A(\theta_A, \epsilon)$) and compute the set

$$X := \{x \mid x \in \{\theta_{A_I}, \phi_{A_I}\} \text{ and } i \in \text{Match}(x)\} \cap S_i \ .$$

Thus X is the set of boundary states in S_i that corresponds to a valid pair computed in Step 1. If $X \neq \emptyset$ we add (i, X) to the sequence V and set $S_i := X$.

We argue that this produces a sequence V of valid pairs with the required properties. First note that by definition of X we inductively produce state-set $S_0, \ldots, S_n$ such that S_i contains the set of states reachable from θ_A that match $Q[1, i]$ and the paths used to reach S_i intersect the states of the valid pairs produced up to this point. Furthermore, we include all positions in V where S_i contains θ_{A_I} or ϕ_{A_I}. It follows that V satisfies the properties.

Each of modified state-set transition uses $O(m)$ time and hence we use $O(nm)$ time in total. The sequence V uses $O(n)$ space. In addition to this we store the match sets and a constant number of active state-sets using $O(n+m)$ space.

Step 3: Computing the String Decomposition

We now convert the sequence $V = (i_1, X_1), (i_2, X_2), \ldots, (i_k, X_k)$ into a string decomposition. First, we construct the partition $q_0, \ldots, q_{k+1}$ of Q such that $q_0 = Q[1, i_1]$, $q_j = Q[i_j + 1, i_{j+1}]$, and $q_{k+1} = Q[i_k + 1, n]$. Note that q_0 and q_{k+1} may be the empty string. Next, we convert this partition into a string decomposition by first labeling each substring as inner or outer and then greedily merging these substrings to obtain an alternating sequence forming a string decomposition. We discuss these steps in detail below.

Labeling. We label the substrings as follows. First label q_0 and q_{k+1} with outer. For the rest of the substrings, if $X_i = \{\theta_{A_I}\}$ and $X_{i+1} = \{\phi_{A_I}\}$ then label q_i with inner, and if $X_i = \{\phi_{A_I}\}$ and $X_{i+1} = \{\theta_{A_I}\}$ then label q_i with outer. If either X_i or X_{i+1} contain more than one boundary state then we use state-set transitions in A_I and A_O to determine if A_I accepts q_i or if there is a path in A_O from ϕ_{A_I} to θ_{A_I} that matches q_i. If a substring is accepted by A_I then it can be an inner substring and if there is a path in A_O from ϕ_{A_I} to θ_{A_I} that matches q_i then it can be an outer substring. If a substring can only be either an inner or an outer substring then it is labeled with inner or outer, respectively. Let q_i be a substring that can be both an inner or an outer substring. We divide this into two cases. If there is an ϵ-path from ϕ_{A_I} to θ_{A_I} then label all such q_i with inner. Otherwise label all such q_i with outer. See also Algorithm 1.

Algorithm 1 Labeling.

Input: A sequence V of valid pairs $(i_1, X_1), \ldots (i_k, V_k)$ and the corresponding partition $q_0, \ldots, q_{k+1}$ of Q.
Output: A labeling of the partition

```
1  The (possible empty) substrings q_0 and q_{k+1} are labeled outer.
2  for i = 1 to k do
3      if X_i = {θ_{A_I}} and X_{i+1} = {φ_{A_I}} then                    /* Case 1 */
4          label q_i inner.
5      else if X_i = {φ_{A_I}} and X_{i+1} = {θ_{A_I}} then               /* Case 2 */
6          label q_i outer.
7      else if X_i or X_{i+1} contains more than one boundary node then /* Case 3 */
8          Use standard state-set transitions in A_I and A_O to determine if A_I accepts q_i
            or if there is a path in A_O from φ_{A_I} to θ_{A_I} that matches q_i.
9          if q_i is only accepted by A_I then                          /* Subcase 3a */
10             label q_i inner
11         else if q_i is only accepted by A_O then                     /* Subcase 3b */
12             label q_i with outer
13         else /* q_i is accepted by both A_I and A_O                          */
14             There are two cases.
15             if there is an ε-path from φ_{A_I} to θ_{A_I} then       /* Subcase 3c */
16                 label q_i inner.
17             else label q_i outer.                                    /* Subcase 3d */
18         end
19     end
20 end
```

For correctness first note that q_0 and q_{k+1} are always (possibly empty) outer substrings. The cases where both $|X_i| = |X_{i+1}|$ (case 1 and 2) are correct by the correctness of the sequence of valid pairs V. Due to cyclic dependencies we may have that X_i and X_{i+1} contain more than one boundary state. This can happen if there is an ϵ-path from θ_{A_I} to ϕ_{A_I} and/or there is an ϵ-path from ϕ_{A_I} to θ_{A_I}. If a substring only is accepted by one of A_I (case 3a) or A_O (case 3b) then it follows from the correctness of V that the labeling is correct. It remains to argue that the labeling in the case where q_i is accepted by both A_I and A_O is correct. To see why the labeling in this case is consistent with a string decomposition of the accepting path consider case 3c. Here, it is safe to label q_i with inner, since if we are in ϕ_{A_I} after having read q_{i-1} we can just follow the ϵ-path from ϕ_{A_I} to θ_{A_I} and then start reading q_i from here.The argument for case 3d is similar.

Except for the state-set transitions in case 3 all cases takes constant time. The total time of all the state-set transitions is $O(nm)$. The space of V and the partition together with the labeling uses $O(n)$ space.

String decomposition. Now every substring has a label that is either inner or outer. We then merge adjacent substrings that have the same label. This produces an alternating sequence of inner and outer substrings, which is the final string decomposition. Such an alternating subsequence must always exist since each pair in V intersects an accepting path.

In summary, we have the following result.

▶ **Lemma 3.** *Given string Q of length n, and TNFA A with m states decomposed into A_O and A_I, we can compute a string decomposition wrt. A_I in $O(nm)$ time and $O(n+m)$ space.*

4 Computing Accepting Paths

Let Q be a string of length n accepted by a TNFA A with m states. In this section we show how to compute an accepting path for Q in A in $O(nm)$ time and $O(n+m)$ space. Since an accepting path may have length $\Omega(nm)$ (there may be $\Omega(m)$ ϵ-transitions between each character transition) we cannot afford to explicitly compute the path in our later $o(nm)$ time algorithms. Instead, our algorithm will compute *compressed paths* of size $O(n)$ obtained by deleting all ϵ-transitions from a path. Note that the compressed path stores precisely the information needed to solve the parsing problem.

To compute the compressed path we define a recursive algorithm $\text{Path}(A, Q)$ that takes a TNFA A and a string Q and returns a compressed accepting path in A matching Q as follows. If $n < \gamma_n$ or $m < \gamma_m$, for constants $\gamma_n, \gamma_m > 0$ that we will fix later, we simply run the naive algorithm that stores all state-sets during state-set simulation from left-to-right in Q. Since one of n or m is constant this uses $O(nm) = O(n+m)$ time and space. Otherwise, we proceed according to the following steps.

Step 1: Decompose. We compute a decomposition of A into inner and outer subTNFAs A_I and A_O and compute a corresponding string decomposition $\mathcal{Q} = \overline{q}_1, q_1, \overline{q}_2, q_2, \overline{q}_\ell, q_\ell, \overline{q}_{\ell+1}$ for Q.

Step 2: Recurse. We build a single substring corresponding to all the subpaths in A_O and ℓ substrings for A_I (one for each subpath in A_I) and recursively compute the compressed paths. To do so, construct $\overline{q} = \overline{q}_1 \cdot \beta_{A_I} \cdot \overline{q}_2 \cdot \beta_{A_I} \cdots \beta_{A_I} \cdot \overline{q}_{\ell+1}$. Recall, that β_{A_I} is the label of the special transition we added between θ_{A_I} and ϕ_{A_I} in A_O. Then, recursively compute the compressed paths

$$\overline{p} = \text{Path}(A_O, \overline{q})$$
$$p_i = \text{Path}(A_I, q_i) \qquad 1 \le i \le \ell$$

Step 3: Combine. Finally, extract the subpaths $\overline{p}_1, \overline{p}_2, \ldots, \overline{p}_{\ell+1}$ from $\overline{p}$ corresponding to the substrings $\overline{q}_1, \overline{q}_2, \ldots, \overline{q}_{\ell+1}$ and return the combined compressed path

$$P = \overline{p}_0 \cdot p_1 \cdot \overline{p}_1 \cdot p_2 \cdot \overline{p}_2 \cdots p_\ell \cdot \overline{p}_{\ell+1}$$

Inductively, it directly follows that the returned compressed path is a compressed accepting path for Q in A.

4.1 Analysis

We now show that the total time $T(n, m)$ of the algorithm is $O(nm)$. If $n < \gamma_n$ or $m < \gamma_m$, we run the backtracking algorithm using $O(nm) = O(n+m)$ time and space. If $n \ge \gamma_n$ and $m \ge \gamma_m$, we implement the recursive step of the algorithm using $O(nm)$ time. Let n_i be the length of the inner string q_i in the string decomposition and let $n_0 = \sum_{i=1}^{\ell+1} |\overline{q}_i|$. Thus, $n = \sum_{i=1}^{\ell+1} n_i$ and $|\overline{q}| = n_0 + \ell$. In step 2, the recursive call to compute $\overline{p}$ takes $O(T(n_0+\ell, \frac{2}{3}m+8))$ time and the recursive calls to compute $p_1, \ldots, p_\ell$ take $\sum_{i=1}^{\ell} T(n_i, \frac{2}{3}m+8)$

time. The remaining steps of the algorithm all take $O(nm)$ time. Hence, we have the following recurrence for $T(n, m)$.

$$T(n,m) = \begin{cases} \sum_{i=1}^{\ell} T(n_i, \frac{2}{3}m + 8) + T(n_0 + \ell, \frac{2}{3}m + 8) + O(mn) & m \geq \gamma_m \text{ and } n \geq \gamma_n \\ O(m+n) & m < \gamma_m \text{ or } n < \gamma_n \end{cases}$$

It follows that $T(n, m) = O(nm)$ for $\gamma_n = 2$ and $\gamma_m = 25$ (see full version [5] for a detailed proof).

Next, we consider the space complexity. First, note that the total space for storing R and Q is $O(n + m)$. To analyse the space during the recursive calls of the algorithm, consider the recursion tree $\mathcal{T}_{\text{rec}}$ for Path(A, Q). For a node v in $\mathcal{T}_{\text{rec}}$, we define Q_v of length n_v to be the string and A_v with m_v states to be the TNFA at v. Consider a path $v_1, \ldots, v_j$ of nodes in $\mathcal{T}_{\text{rec}}$ from the root to leaf v_j corresponding to a sequence of nested recursive calls of the algorithm. If we naively store the subTNFAs, the string decompositions, and the compressed paths, we use $O(n_{v_i} + m_{v_i})$ space at each node v_i, $1 \leq i \leq j$. By Lemma 2(i) the sum of the sizes of the subTNFAs on a path forms a geometrically decreasing sequence and hence the total space for the subTNFAs is $\sum_{i=1}^{j} m_{v_i} = O(m)$. However, since each string (and hence compressed path) at each node v_i, $1 \leq i \leq j$, may have length $\Omega(n)$ we may need $\Omega(n \log m)$ space in total for these. We show how to improve this to $O(n + m)$ space in the next section.

4.2 Squeezing into Linear Space

We now show how to improve the space to $O(n + m)$. To do so we show how to carefully implement the recursion to only store the strings for a selected subset of the nodes along any root to leaf path in $\mathcal{T}_{\text{rec}}$ that in total take no more than $O(n)$ space.

First, consider a node v in $\mathcal{T}_{\text{rec}}$ and the corresponding string Q_v and TNFA A_v. Define χ_v^Q to be the function that maps each character position in Q_v (ignoring β_{A_I} transitions) to the unique corresponding character in Q and χ_v^A to be the function that maps each character transition (non-ϵ transition) in A_v to the unique character transition in A. Note that these are well-defined by the construction of subproblems in the algorithm. At a node v, we represent χ_v^Q by storing for each character in Q_v a pointer to the corresponding character in Q. Similarly, we represent χ_v^A by storing for each character transition in A_v a pointer to the corresponding character transition in A. This uses $O(n_v + m_v)$ additional space. It is straightforward to compute these mappings during the algorithm directly from the string and TNFA decomposition in the same time. With the mappings we can now output transitions on the compressed path as pairs of position in Q and transitions in A immediately as they are computed in a leaf of $\mathcal{T}_{\text{rec}}$. Thus, when we have traversed a full subtree at a node we can free the space for that node since we do not have to wait to return to the root to combine the pieces of the subpath with other subpaths.

We combine the mappings with an ordering of the recursion according to a *heavy-path decomposition* of $\mathcal{T}_{\text{rec}}$. Let v be an internal node in $\mathcal{T}_{\text{rec}}$. The *string length* of v is n_v. We define the *heavy child* of v to be a child of v of maximum string length among the children of v. The remaining children are *light children* of v. We have the following key property of light children (see full version [5] for a proof).

▶ **Lemma 4.** *For any node v with light child u in $\mathcal{T}_{rec}$, we have that $n_u \leq \frac{3}{4}n_v + O(1)$.*

We order the recursive calls at each node v as follows. First, we recursively visit all the light children of v and upon completing each recursive call, we free the space for that node. Note that the mappings allow us to do this. We then construct the subproblem for the heavy child of v, free the space for v, and continue the recursion at the heavy child.

To analyse the space of the modified algorithm, consider a path a path $v_1, \ldots, v_j$ of nodes in $\mathcal{T}_{\text{rec}}$ from the root to a leaf v_j. We now have that only nodes v_i, $1 \leq i < j$ will explicitly store a string if v_{i+1} is a light child of v_i. By Lemma 4 the sum of these lengths form a geometrically decreasing sequence and hence the total space is now $O(n)$. In summary, we have shown the following result.

▶ **Theorem 5.** *Given a TNFA with m states and a string of length n, we can compute a compressed accepting path for Q in A in $O(nm)$ time and $O(n+m)$ space.*

Note that the algorithm works in a comparison-based, pointer model of computation. By our discussion this immediately implies the main result of Theorem 1.

5 Speeding up the Algorithm

We now show how to adapt the algorithm to use the faster state-set simulation algorithms such as Myers' algorithm [22] and later variants [4, 3] that improve the $O(m)$ bound for a single state-set transition. These results and ours all work on a unit-cost word RAM model of computation with w-bit words and a standard instruction set including addition, bitwise boolean operations, shifts, and multiplication. We can store a pointer to any position in the input and hence $w \geq \log(n+m)$. For simplicity, we will show how to adapt the tabulation-based algorithm of Bille and Farach-Colton [4].

5.1 Fast Matching

Let A be a TNFA with m states and let Q be a string of length n. Assume first that the alphabet is constant. We briefly describe the main features of the algorithm by Bille and Farach-Colton [4] that solves the matching problem in $O(nm/\log n + n + m)$ time and $O(n^\varepsilon + m)$ space, for any constant $\varepsilon > 0$. In the next section we show how to adapt the algorithm to compute an accepting path.

Given a parameter $t < w$, we will construct a global table of size $2^{ct} < 2^w$, for a constant c, to speed up the state-set transition. We decompose A into a tree MS of $O(\lceil m/t \rceil)$ micro TNFAs, each with at most t states. For each $M \in MS$, each child C is represented by its start and accepting state and a pseudo-transition connecting these. By similar arguments as in Lemma 2 we can always construct such a decomposition.

We represent each micro TNFA $M \in MS$ uniquely by its parse tree using $O(t)$ bits. Since M has at most t states, we can represent the state-set for M, called the *local state-set* and denoted S_M, using t bits. Hence, we can make a universal table of size $2^{O(t)}$ that for every possible micro TNFA M of size $\leq t$, local state-set S_M, and character $\alpha \in \Sigma \cup \{\epsilon\}$ computes the state-set transition $\delta_M(S_M, \alpha)$ in constant time.

We use the tabulation to efficiently implement a global state-set transition on A as follows. We represent the state-set for A as the union of the local state-sets in MS. Note that parents and children in MS share some local states, and these states will be copied in the local state-sets.

To implement a state-set transition on A for a character α, we first traverse all transitions labeled α in each micro TNFA from the current state-set. We then follow paths of ϵ transition in two depth-first left-to-right traversal of MS. At each micro TNFA M, we compute all states reachable via ϵ-transitions and propagate the shared states among parents and children in MS. Since any cycle free path in a TNFA contains at most one *back transition* (see [22, Lemma 1]) it follows that two such traversals suffices to to correctly compute all states in A reachable via ϵ-transitions.

With the universal table, we process each micro TNFA in constant time, leading to an algorithm using $O(|MS|/t + n + m) = O(nm/t + n + m)$ time and $O(2^t + m)$ space. Setting $t = \varepsilon \log n$ produces the stated result. Note that each state-set uses $O(\lceil m/t \rceil)$ space. To handle general alphabets, we store dictionaries for each micro TNFA with bit masks corresponding to characters appearing in the TNFA and combine these with an additional masking step in state-set transition. The leads a general solution with the same time and space bounds as above.[3]

5.2 Fast Parsing

We now show how to modify our algorithm from Section 4 to take advantage of the fast state-set transition algorithm. Let $t < w$ be the desired speed up as above. We have the following two cases.

If $n \geq t$ and $m \geq t$ we implement the recursive step of the algorithm but replace all state-set transitions, that is when we compute the match sets and valid pairs, by the fast state-set transition algorithm. To compute the suffix match sets we need to compute fast state-set transitions on A with the direction of all transitions reversed. To do so, we make a new universal table of size $2^{O(t)}$ for the micro TNFAs with the direction of transitions reversed. We traverse the tree of micro TNFAs with two depth traversals as before except that we now traverse children in right to left order to correctly compute all reachable states. It follows that this uses $O(nm/t)$ time.

Otherwise ($n < t$ or $m < t$), we use backtracking to compute the accepting path as follows. First, we process Q from left-to-right using fast state-set transitions to compute the sets $S_0, \ldots, S_n$ of states reachable via paths from θ_A for each prefix of Q. We store each of these state-sets. This uses $O(nm/t + n + m) = O(n + m)$ time and space. Then, we process Q from right-to-left to recover a compressed accepting path in A. Starting from ϕ_A we repeatedly do a fast state-set transition A with the direction of transition reversed, compute the intersection of the resulting state-set with the corresponding state-set from the first step, and update the state-set to a state in the intersection. We can compute the intersection of state-sets and the update in $O(m/t)$ time using standard bit wise operations. We do the state-set transitions on the TNFA with directions of transitions reversed as above. In total, this uses $O(nm/t + n + m) = O(n + m)$ time.

In summary, we have the following recurrence for the time $T(n, m)$.

$$T(n,m) = \begin{cases} \sum_{i=0}^{\ell} T(n_i, \frac{2}{3}m + 8) + T(n_0 + \ell, \frac{2}{3}m + 8) + O\left(\frac{nm}{t}\right) & n \geq t \text{ and } m \geq t \\ O(n + m) & m < t \text{ or } n < t \end{cases}$$

Similar to Section 4.1 it follows that $T(n, m) = O(nm/t + n + m)$, for $25 \leq t < w$. The space is linear as before. Plugging in $t = \epsilon \log n$ and including the preprocessing time and space for the universal tables we obtain the following logarithmic improvement of Theorem 1.

▶ **Theorem 6.** *Given a regular expression of length m, a string of length n, we can solve the regular expression parsing problem in $O(nm/\log n + n + m)$ time and $O(n + m)$ space.*

[3] Note that the time bound in the original paper has an additional $m \log m$ term [4]. Using atomic heaps [9] to represent dictionaries for micro TNFAs this term is straightforward to improve to $O(m)$. See also Bille and Thorup [6, Appendix A].

References

1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

2 Arturs Backurs and Piotr Indyk. Which Regular Expression Patterns are Hard to Match? In *Proc. 57th FOCS*, pages 457–466, 2016.

3 Philip Bille. New Algorithms for Regular Expression Matching. In *Proc. of the 33rd ICALP*, pages 643–654, 2006.

4 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.

5 Philip Bille and Inge Li Gørtz. From Regular Expression Matching to Parsing. *Arxiv preprint arXiv:1804.02906*, 2019.

6 Philip Bille and Mikkel Thorup. Faster Regular Expression Matching. In *Proc. 36th ICALP*, pages 171–182, 2009. Full version with appendix available at `http://www2.compute.dtu.dk/~phbi/files/publications/2009fremC.pdf`.

7 Philip Bille and Mikkel Thorup. Regular Expression Matching with Multi-Strings and Intervals. In *Proc. 21st SODA*, pages 1297–1308, 2010.

8 Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

9 Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.

10 Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proc. 31st ICALP*, volume 3142, pages 618–629, 2004.

11 Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. 25th VLDB*, pages 223–234, 1999.

12 Victor M. Glushkov. The Abstract Theory of Automata. *Russian Math. Surveys*, 16(5):1–53, 1961.

13 D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

14 Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Monitoring Regular Expressions on Out-of-Order Streams. In *Proc. 23nd ICDE*, pages 1315–1319, 2007.

15 Steven M Kearns. Extending regular expressions with context operators and parse extraction. *Software: Practice and Experience*, 21(8):787–804, 1991.

16 Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: multitouch gestures as regular expressions. In *Proc. SIGCHI*, pages 2885–2894, 2012.

17 Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM*, pages 339–350, 2006.

18 Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proc. 7th SPIRE*, pages 181–187, 2000.

19 Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. 27th VLDB*, pages 361–370, 2001.

20 R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE Trans. on Electronic Computers*, 9(1):39–47, 1960.

21 Makoto Murata. Extended path expressions of XML. In *Proc. 20th PODS*, pages 126–137, 2001.

22 E. W. Myers. A Four-Russian Algorithm for Regular Expression Pattern Matching. *J. ACM*, 39(2):430–448, 1992.

23 Gonzalo Navarro and Mathieu Raffinot. Fast and Simple Character Classes and Bounded Gaps Pattern Matching, with Applications to Protein Searching. *J. Comp. Biology*, 10(6):903–923, 2003.

24 Lasse Nielsen and Fritz Henglein. Bit-coded Regular Expression Parsing. In *Proc. 5th LATA*, pages 402–413, 2011.

25 Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *Proc. 14th PPDP*, pages 79–90, 2012.

26 K. Thompson. Regular Expression Search Algorithm. *Commun. ACM*, 11:419–422, 1968.

27 Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ANCS*, pages 93–102, 2006.