Theoretical Computer Science 409 (2008) 486–496

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Fast and compact regular expression matching

Philip Bille [a,*], Martin Farach-Colton [b]

[a] IT University of Copenhagen, 2300 Copenhagen S, Denmark

[b] Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA

ARTICLE INFO

Article history:
Received 22 February 2007
Received in revised form 9 July 2008
Accepted 25 August 2008
Communicated by A. Apostolico

Keywords:
Regular expression matching
Approximate regular expression matching:
String edit distance
Subsequence indexing
Four russian technique

ABSTRACT

We study 4 problems in string matching, namely, regular expression matching, approximate regular expression matching, string edit distance, and subsequence indexing, on a standard word RAM model of computation that allows logarithmic-sized words to be manipulated in constant time. We show how to improve the space and/or remove a dependency on the alphabet size for each problem using either an improved tabulation technique of an existing algorithm or by combining known algorithms in a new way.

1. Introduction

We study 4 problems in string matching on a standard word RAM model of computation that allows logarithmic-sized words to be manipulated in constant time. This model is often called the *transdichotomous model*. We show how to improve the space and/or remove a dependency on the alphabet size for each problem. Three of the results are obtained by improving the tabulation of subproblems within an existing algorithm. The idea of using tabulation to improve algorithms is often referred to as the *Four Russian Technique* after Arlazarov et al. [1] who introduced it for boolean matrix multiplication. The last result is based on a new combination of known algorithms. The problems and our results are presented below.

Regular expression matching. Given a regular expression R and a string Q, the Regular Expression Matching problem is to determine if Q is a member of the language denoted by R. This problem occurs in several text processing applications, such as in editors like Emacs [24] or in the Grep utilities [30,21]. It is also used in the lexical analysis phase of compilers and interpreters, regular expressions are commonly used to match tokens for the syntax analysis phase, and more recently for querying and validating XML databases, see e.g., [12,13,16,6]. The standard textbook solution to the problem, due to Thompson [25], constructs a non-deterministic finite automaton (NFA) for R and simulates it on the string Q. For R and Q of sizes m and n, respectively, this algorithm uses $O(mn)$ time and $O(m)$ space. If the NFA is converted into a deterministic finite automaton (DFA), the DFA needs $O(\frac{m}{w}2^{2m}\sigma)$ words, where σ is the size of the alphabet Σ and w is the word size. Using clever representations of the DFA the space can be reduced to $O(\frac{m}{w}(2^m+\sigma))$ [31,23]. Efficient average case algorithms were given by Baeza-Yates and Gonnet [4].

Normally, it is reported that the running time of traversing the DFA is $O(n)$, but this complexity analysis ignores the word size. Since nodes in the DFA may need $\Omega(m)$ bits to be addressed, we may need $\Omega(m/w+1)$ time to identify the next node in the traversal. Therefore the running time becomes $O(mn/w+n+m)$ with a potential exponential blowup in the

[*] Corresponding author. Tel.: +45 72 18 52 71.
E-mail addresses: beetle@itu.dk (P. Bille), farach@cs.rutgers.edu (M. Farach-Colton).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.08.042

space. Hence, in the transdichotomous model, where w is $\Theta(\log(n+m))$, using worst-case exponential preprocessing time improves the query time by a log factor.

The fastest known algorithm is due to Myers [17], who showed how to achieve $O(mn/k + m2^k + (n+m)\log m)$ time and $O(2^k m)$ space, for any $k \leq w$. In particular, for $k = \epsilon \log n$, for constant $0 < \epsilon < 1$, this gives an algorithm using $O(mn/\log n + (n+m)\log m)$ time and $O(mn^\epsilon)$ space.

In Section 2, we present an algorithm for Regular Expression Matching that takes time $O(nm/k + n + m\log m)$ time and uses $O(2^k + m)$ space, for any $k \leq w$. In particular, if we pick $k = \epsilon \log n$, for constant $0 < \epsilon < 1$, we are (at least) as fast as the algorithm of Myers, while achieving $O(n^\epsilon + m)$ space.

We note that for large word sizes ($w > \log^2 n$) one of the authors has recently devised an even faster algorithm using very different ideas [7]. This research was done after the work that led to the results in this paper.

Approximate regular expression matching. Motivated by applications in computational biology, Myers and Miller [18] studied the Approximate Regular Expression Matching problem. Here, we want to determine if Q is within *edit distance* d to any string in the language given by R. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to transform one string into the other. Myers and Miller [18] gave an $O(mn)$ time and $O(m)$ space dynamic programming algorithm. Subsequently, assuming as a constant sized alphabet, Wu, Manber and Myers [32] gave an $O(\frac{mn\log(d+2)}{\log n} + n + m)$ time and $O(\frac{m\sqrt{n}\log(d+2)}{\log n} + n + m)$ space algorithm. Recently, an exponential space solution based on DFAs for the problem has been proposed by Navarro [22].

In Section 3, we extend our results of Section 2 and give an algorithm, without any assumption on the alphabet size, using $O(\frac{mn\log(d+2)}{k} + n + m\log m)$ time and $O(2^k + m)$ space, for any $k \leq w$.

String edit distance. We conclude by giving a simple way to improve the complexity of the String Edit Distance problem, which is defined as that of computing the minimum number of edit operations needed to transform given string S of length m into given string T of length n. The standard dynamic programming solution to this problem uses $O(mn)$ time and $O(\min(m,n))$ space. The fastest algorithm for this problem, due to Masek and Paterson [14], achieves $O(mn/k^2 + m + n)$ time and $O(2^k + \min(n,m))$ space for any $k \leq w$. However, this algorithm assumes a constant size alphabet. For long word sizes faster algorithms can be obtained [19,5]. See also the survey by Navarro [20].

In Section 4, we show how to achieve $O(nm\log^2 k/k^2 + m + n)$ time and $O(2^k + \min(n,m))$ space for any $k \leq w$ for an arbitrary alphabet. Hence, we remove the dependency of the alphabet at the cost of a $\log^2 k$ factor to the running time.

Subsequence indexing. We also consider a special case of regular expression matching. Given text T, the Subsequence Indexing problem is to preprocess T to allow queries of the form "is Q a subsequence of T?" Baeza-Yates [3] showed that this problem can be solved with $O(n)$ preprocessing time and space, and query time $O(m\log n)$, where Q has length m and T has length n. Conversely, one can achieve queries of time $O(m)$ with $O(n\sigma)$ preprocessing time and space. As before, σ is the size of the alphabet.

In Section 5, we give an algorithm that improves the former results to $O(m\log\log\sigma)$ query time or the latter result to $O(n\sigma^\epsilon)$ preprocessing time and space.

2. Regular expression matching

Given an string Q and a regular expression R the Regular Expression Matching problem is to determine if Q is in the language given by R. Let n and m be the sizes of Q and R, respectively. In this section we show that Regular Expression Matching can be solved in $O(mn/k + n + m\log m)$ time and $O(2^k + m)$ space, for $k \leq w$.

2.1. Regular expressions and NFAs

We briefly review Thompson's construction and the standard node set simulation. The set of *regular expressions* over Σ is defined recursively as follows:

- A character $\alpha \in \Sigma$ is a regular expression.
- If S and T are regular expressions then so is the *catenation*, $(S)\cdot(T)$, the *union*, $(S)|(T)$, and the *star*, $(S)^*$.

Unnecessary parentheses can be removed by observing that $\cdot$ and $|$ are associative and by using the standard precedence of the operators, that is $*$ precedes $\cdot$, which in turn precedes $|$. Furthermore, we will often remove the $\cdot$ when writing regular expressions. The *language* $L(R)$ generated by R is the set of all strings matching R. The *parse tree* $T(R)$ of R is the rooted and ordered tree representing the hierarchical structure of R. All leaves are represented by a character in Σ and all internal nodes are labeled $\cdot$, $|$, or *. We assume that parse trees are binary and constructed such that they are in one-to-one correspondence with the regular expressions. An example parse tree of the regular expression $ac|a^*b$ is shown in Fig. 2(a).

A *finite automaton* A is a tuple $A = (G, \Sigma, \theta, \Phi)$ such that,

- G is a directed graph,
- Each edge $e \in E(G)$ is labeled with a character $\alpha \in \Sigma$ or ϵ,
- $\theta \in V(G)$ is a *start node*,
- $\Phi \subseteq V(G)$ is the set of *accepting nodes*.

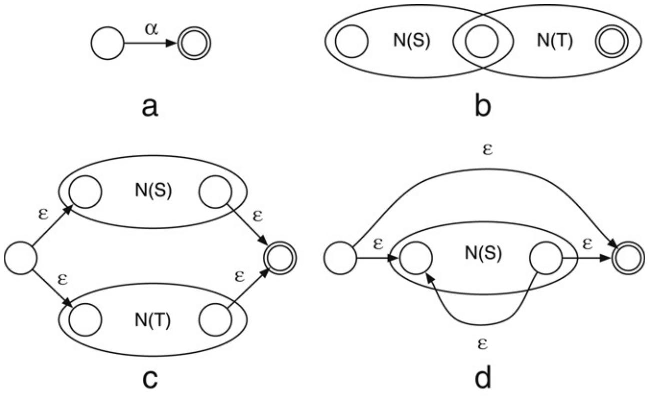

Fig. 1. Thompson's NFA construction. The regular expression for a character $\alpha \in \Sigma$ correspond to NFA (a). If S and T are regular expression then $N(ST)$, $N(S|T)$, and $N(S^*)$ correspond to NFAs (a), (b), and (c), respectively. Accepting nodes are marked with a double circle.

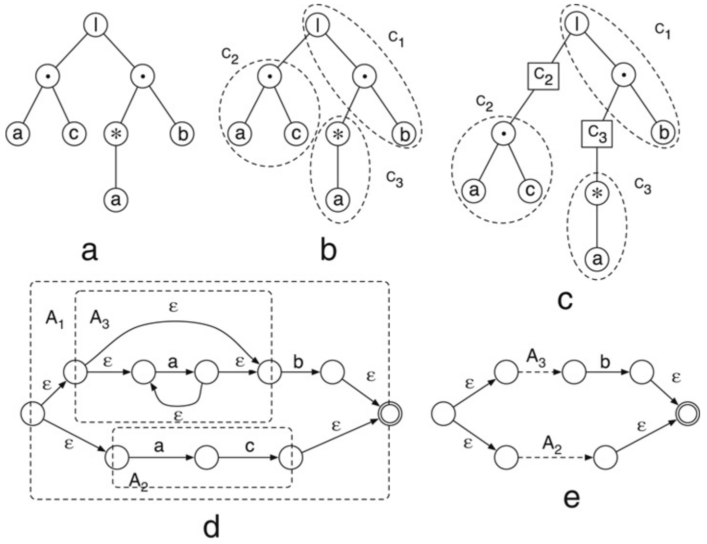

Fig. 2. (a) The parse tree for the regular expression $ac|a^*b$. (b) A clustering of (a) into node-disjoint connected subtrees C_1, C_2, and C_3. Here, $x = 3$. (c) The clustering from (b) extended with pseudo-nodes. (d) The automaton for the parse tree divided into subautomata corresponding to the clustering. (e) The subautomaton A_1 with pseudo-edges corresponding to the child automata.

A is a *deterministic finite automaton* (DFA) if A does not contain any ϵ-edges, and for each node $v \in V(G)$ all outcoming edges have different labels. Otherwise, A is a *non-deterministic automaton* (NFA). We say that A *accepts* a string Q if there is a path from θ to a node in Φ which spells out Q.

Using Thompson's method [25] we can recursively construct an NFA $N(R)$ accepting all strings in $L(R)$. The set of rules is presented below and illustrated in Fig. 1.

- $N(\alpha)$ is the automaton consisting of a start node θ_α, accepting node ϕ_α, and an α-edge from θ_α to ϕ_α.
- Let $N(S)$ and $N(T)$ be automata for regular expression S and T with start and accepting nodes θ_S, θ_T, ϕ_S, and ϕ_T, respectively. Then, NFAs for $N(S \cdot T)$, $N(S|T)$, and $N(S^*)$ are constructed as follows:
  $N(ST)$: Merge the nodes ϕ_S and θ_T into a single node. The new start node is θ_S and the new accepting node is ϕ_T.
  $N(S|T)$: Add a new start node $\theta_{S|T}$ and new accepting node $\phi_{S|T}$. Then, add ϵ edges from $\theta_{S|T}$ to θ_S and θ_T, and from ϕ_S and ϕ_T to $\phi_{S|T}$.
  $N(S^*)$: Add a new start node θ_{S*} and new accepting node ϕ_{S*}. Then, add ϵ edges from θ_{S*} to θ_S and ϕ_{S*}, and from ϕ_S to ϕ_{S*} and θ_S.

By construction, $N(R)$ has a single start and accepting node, denoted θ and ϕ, respectively. θ has no incoming edges and ϕ has no outcoming edges. The total number of nodes is at most $2m$ and since each node has at most 2 outgoing edges that the total number of edges is less than $4m$. Furthermore, all incoming edges have the same label, and we denote a node with incoming α-edges an *α-node*. Note that the star construction in Fig. 1(d) introduces an edge from the accepting node of $N(S)$ to the start node of $N(S)$. All such edges in $N(R)$ are called *back edges* and all other edges are *forward edges*. We need the following important property of $N(R)$.

Lemma 1 (*Myers [17]*). *Any cycle-free path in $N(R)$ contains at most one back edge.*

For a string Q of length n the standard node-set simulation of $N(R)$ on Q produces a sequence of node-sets $S_0, \dots, S_n$. A node v is in S_i iff there is a path from θ to v that spells out the ith prefix of Q. The simulation can be implemented with the following simple operations. Let S be a node-set in $N(R)$ and let α be a character in Σ.

$\mathsf{Move}(S, \alpha)$: Compute and return the set of nodes reachable from S via a single α-edge.
$\mathsf{Close}(S)$: Compute and return the set of nodes reachable from S via 0 or more ϵ-edges.

The number of nodes and edges in $N(R)$ is $O(m)$, and both operations are implementable in $O(m)$ time. The simulation proceed as follows: Initially, $S_0 := Close(\{\theta\})$. If $Q[j] = \alpha$, $1 \leq j \leq n$, then $S_j := \mathsf{Close}(\mathsf{Move}(S_{j-1}, \alpha))$. Finally, $Q \in L(R)$ iff $\phi \in S_n$. Since each node-set S_j only depends on S_{j-1} this algorithm uses $O(mn)$ time $O(m)$ space.

2.2. Outline of algorithm

Our result is based on a new and more compact encoding of small subautomata used within Myers' algorithm [17] supporting constant time Move and Close operations. For our purposes and for completeness, we restate Myers' algorithm in Sections 2.3 and 2.4, such that the dependency on the Move and Close operations on subautomata is exposed. The new encoding is presented in Section 2.5.

2.3. Decomposing the NFA

In this section we show how to decompose $N(R)$ into small subautomata. In the final algorithm transitions through these subautomata will be simulated in constant time. The decomposition is based on a clustering of the parse tree $T(R)$. Our decomposition is similar to the one given in [17,32]. A *cluster* C is a connected subgraph of $T(R)$. A *cluster partition* CS is a partition of the nodes of $T(R)$ into node-disjoint clusters. Since $T(R)$ is a binary tree, a bottom-up procedure yields the following lemma.

Lemma 2. *For any regular expression R of size m and a parameter x, it is possible to build a cluster partition CS of $T(R)$, such that $|CS| = O(m/x)$ and for any $C \in CS$ the number of nodes in C is at most x.*

An example clustering of a parse tree is shown in Fig. 2(b).

Before proceeding, we need some definitions. Assume that CS is a cluster partition of $T(R)$ for a some yet-to-be-determined parameter x. Edges adjacent to two clusters are *external edges* and all other edges are *internal edges*. Contracting all internal edges induces a *macro tree*, where each cluster is represented by a single *macro node*. Let C_v and C_w be two clusters with corresponding macro nodes v and w. We say that C_v is a *parent cluster* (resp. *child cluster*) of C_w if v is the parent (resp. child) of w in the macro tree. The *root cluster and leaf clusters* are the clusters corresponding to the root and the leaves of the macro tree.

Next we show how to decompose $N(R)$ into small subautomata. Each cluster C will correspond to a subautomaton A and we use the terms child, parent, root, and leaf for subautomata in the same way we do with clusters. For a cluster C, we insert a special *pseudo-node* p_i for each child cluster $C_1, \dots, C_\ell$ in the middle of the external edge connecting C and C_i. Now, C's subautomaton A is the automaton corresponding to the parse tree induced by the set of nodes $V(C) \cup \{p_1, \dots, p_\ell\}$. The pseudo-nodes are alphabet placeholders, since the leaves of a well-formed parse tree must be characters.

In A, child automaton A_i is represented by its start and accepting node θ_{A_i} and ϕ_{A_i} and a *pseudo-edge* connecting them. An example of these definitions is given in Fig. 2. Any cluster C of size at most x has less than $2x$ pseudo-children and therefore the size of the corresponding subautomaton is at most $6x$. Note, therefore, that automata derived from regular expressions can be thus decomposed into $O(m/z)$ subautomata each of size at most z, by Lemma 2 and the above construction.

2.4. Simulating the NFA

In this section we show how to do a node-set simulation of $N(R)$ using the subautomata. We compactly represent node-set of each subautomata in a bit string and in the next section we will show how to manipulate these node-set efficiently using a combination of the Four Russian Technique and standard word operations. This approach is often called *bit-parallelism* [2].

Recall that each subautomaton has size less than z. Topologically sort all nodes in each subautomaton A ignoring back edges. This can be done for all subautomata in total $O(m)$ time. We represent the current node-set S of $N(R)$ compactly using a bitvector for each subautomaton. Specifically, for each subautomaton A we store a *characteristic bitvector* $\vec{B} = [b_1, \dots, b_z]$, where nodes in $\vec{B}$ are indexed by the their topological order, such that $\vec{B}[i] = 1$ iff the ith node is in S. If A contains fewer than z nodes we leave the remaining values undefined. For simplicity, we will refer to the *state* of A as the node-set represented by the characteristic vector stored at A. Similarly, the state of $N(R)$ is the set of characteristic vectors representing S. The state of a node is the bit indicating if the node is in S. Since any child A' of A overlap at the nodes $\theta_{A'}$ and $\phi_{A'}$ we will ensure that the state of $\theta_{A'}$ and $\phi_{A'}$ is the same in the characteristic vectors of both A and A'.

Below we present appropriate move and ϵ-closure operations defined on subautomata. Due to the overlap between parent and child nodes these operations take a bit b which will use to propagate the new state of the start node. For each subautomaton A, characteristic vector $\vec{B}$, bit b, and character $\alpha \in \Sigma$ define:

$\mathsf{Move}^A(\vec{B}, b, \alpha)$: Compute the state $\vec{B}'$ of all nodes in A reachable via a single α edge from $\vec{B}$. If $b = 0$, return $\vec{B}'$, else return $\vec{B}' \cup \{\theta_A\}$.

$\mathsf{Close}^A(\vec{B}, b)$: Return the set $\vec{B}'$ of all nodes in A reachable via a path of 0 or more ϵ-edges from $\vec{B}$, if $b = 0$, or reachable from $\vec{B} \cup \{\theta_A\}$, if $b = 1$.

We will later show how to implement these operations in constant time and total $2^{O(k)}$ space when $z = \Theta(k)$. Before doing so we show how to use these operations to perform the node-set simulation of $N(R)$. Assume that the current node-set of $N(R)$ is represented by its characteristic vector for each subautomaton. The following Move and Close operations recursively traverse the hierarchy of subautomata top-down. At each subautomata the current state of $N(R)$ is modified using primarily Move^A and Close^A. For any subautomaton A, bit b, and character $\alpha \in \Sigma$ define:

```
Move(A, b, α): Let $\vec{B}$ be the current state of $A$ and let $A_1, \ldots, A_\ell$ be children of $A$ in topological order of their start node.
    1. Compute $\vec{B}' := \mathsf{Move}^A(\vec{B}, b, \alpha)$.
    2. For each $A_i$, $1 \le i \le \ell$,
        (a) Compute $f_i := \mathsf{Move}(A_i, b_i, \alpha)$, where $b_i = 1$ iff $\theta_{A_i} \in \vec{B}'$.
        (b) If $f_i = 1$ set $\vec{B}' := \vec{B}' \cup \{\phi_{A_i}\}$.
    3. Store $\vec{B}'$ and return the value 1 if $\phi_A \in \vec{B}'$ and 0 otherwise.
```

```
Close(A, b): Let $\vec{B}$ be the current state of $A$ and let $A_1, \ldots, A_\ell$ be children of $A$ in topological order of their start node.
    1. Compute $\vec{B}' := \mathsf{Close}^A(\vec{B}, b)$.
    2. For each child automaton $A_i$, $1 \le i \le \ell$,
        (a) Compute $f_i := \mathsf{Close}(A_i, b_i)$, where $b_i = 1$ if $\theta_{A_i} \in \vec{B}'$.
        (b) If $f_i = 1$ set $\vec{B}' := \vec{B}' \cup \{\phi_{A_i}\}$.
        (c) $\vec{B}' := \mathsf{Close}^A(\vec{B}, b)$.
    3. Store $\vec{B}'$ and return the value 1 if $\phi_A \in \vec{B}'$ and 0 otherwise.
```

The "store" in line 3 of both operations updates the state of the subautomaton. The node-set simulation of $N(R)$ on string Q of length n produces the states $S_0, \ldots, S_n$ as follows. Let A_r be the root automaton. Initialize the state of $N(R)$ to be empty, i.e., set all bitvectors to 0. S_0 is computed by calling $\mathsf{Close}(A_r, 1)$ twice. Assume that S_{j-1}, $1 \le j \le n$, is the current state of $N(R)$ and let $\alpha = Q[j]$. Compute S_j by calling $\mathsf{Move}(A_r, 0, \alpha)$ and then calling $\mathsf{Close}(A_r, 0)$ twice. Finally, $Q \in L(R)$ iff $\phi \in S_n$.

We argue that the above algorithm is correct. To do this we need to show that the call to the Move operation and the two calls to the Close operation simulates the standard Move and Close operations.

First consider the Move operation. Let S be the state of $N(R)$ and let S' be the state after a call to $\mathsf{Move}(A_r, 0, \alpha)$. Consider any subautomaton A and let $\vec{B}$ and $\vec{B}'$ be the bitvectors of A corresponding to states S and S', respectively. We first show by induction that after $\mathsf{Move}(A, 0, \alpha)$ the new state $\vec{B}'$ is the set of nodes reachable from $\vec{B}$ via a single α-edge in $N(R)$. For $\mathsf{Move}(A, 1, \alpha)$ a similar argument shows that new state is the union of the set of nodes reachable from $\vec{B}$ via a single α-edge and $\{\theta_A\}$.

Initially, we compute $\vec{B}' := \mathsf{Move}^A(\vec{B}, 0, \alpha)$. Thus $\vec{B}'$ contains the set of nodes reachable via a single α-edge in A. If A is a leaf automaton then $\vec{B}'$ satisfies the property and the algorithm returns. Otherwise, there may be an α-edge to some accepting node ϕ_{A_i} of a child automaton A_i. Since this edge is not contained A, ϕ_{A_i} is not initially in $\vec{B}'$. However, since each child is handled recursively in topological order and the new state of start and accepting nodes are propagated, it follows that ϕ_{A_i} is ultimately added to $\vec{B}'$. Note that since a single node can be the accepting node of a child A_i and the start node of child A_{i+1}, the topological order is needed to ensure a consistent update of the state.

It now follows that the state S' of $N(R)$ after $\mathsf{Move}(A_r, 0, \alpha)$, consists of all nodes reachable via a single α-edge from S. Hence, $\mathsf{Move}(A_r, 0, \alpha)$ correctly simulates a standard Move operation.

Next consider the two calls to the Close operation. Let S be the state of $N(R)$ and let S' be the state after the first call to $\mathsf{Close}(A_r, 0)$. As above consider any subautomaton A and let $\vec{B}$ and $\vec{B}'$ be the bitvectors of A corresponding to S and S', respectively. We show by induction that after $\mathsf{Close}(A, 0)$ the state $\vec{B}'$ *contains* the set of nodes in $N(R)$ reachable via a path of 0 or more *forward* ϵ-edges from $\vec{B}$. Initially, $\vec{B}' := \mathsf{Close}^A(\vec{B}, 0)$, and hence $\vec{B}'$ contains all nodes reachable via a path of 0 or more ϵ-edges from $\vec{B}$, where the path consists solely of edges in A. If A is a leaf automaton, the result immediately holds. Otherwise, there may be a path of ϵ-edges to a node v going through the children of A. As above, the recursive topological processing of the children ensures that v is added to $\vec{B}'$.

Hence, after the first call to $\mathsf{Close}(A_r, 0)$ the state S' contains all nodes reachable from S via a path of 0 or more forward ϵ-edges. By a similar argument it follows that the second call to $\mathsf{Close}(A_r, 0)$ produces the state S'' that contains all the nodes reachable from S via a path of 0 or more forward ϵ-edge and 1 back edge. However, by Lemma 1 this is exactly the set of nodes reachable via a path of 0 or more ϵ-edges. Furthermore, since $\mathsf{Close}(A_r, 0)$ never produces a state with nodes that are not reachable through ϵ-edges, it follows that the two calls to $\mathsf{Close}(A_r, 0)$ correctly simulates a standard Close operation.

Finally, note that if we start with a state with no nodes, we can compute the state S_0 in the node-set simulation by calling $\mathsf{Close}(A_r, 1)$ twice. Hence, the above algorithm correctly solves Regular Expression Matching.

If the subautomata have size at most z and Move^A and Close^A can be computed in constant time the above algorithm computes a step in the node-set simulation in $O(m/z)$ time. In the following section we show how to do this in $O(2^k)$ space for $z = \Theta(k)$. Note that computing the clustering uses an additional $O(m)$ time and space.

2.5. Representing subautomata

To efficiently represent Move^A and Close^A we apply the Four Russian trick. Consider a straightforward code for Move^A: Precompute the value of Move^A for all $\vec{B}$, both values of b, and all characters α. Since the number of different bitvectors is 2^z and the size of the alphabet is σ, this table has $2^{z+1}\sigma$ entries. Each entry can be stored in a single word, so the table also uses a total of $2^{z+1}\sigma$ space. The total number of subautomata is $O(m/z)$, and therefore the total size of these tables is an unacceptable $O(\frac{m}{z} \cdot 2^z\sigma)$.

To improve this we use a more elaborate approach. First we factor out the dependency on the alphabet, as follows. For all subautomata A and all characters $\alpha \in \Sigma$ define:

$\mathsf{Succ}^A(\vec{B})$: Return the set of all nodes in A reachable from $\vec{B}$ by a single edge.
$\mathsf{Eq}^A(\alpha)$: Return the set of all α-nodes in A.

Since all incoming edges to a node are labeled with the same character it follows that,

$$\mathsf{Move}^A(\vec{B}, b, \alpha) = \begin{cases} \mathsf{Succ}^A(\vec{B}) \cap \mathsf{Eq}^A(\alpha) & \text{if } b = 0, \\ (\mathsf{Succ}^A(\vec{B}) \cap \mathsf{Eq}^A(\alpha)) \cup \{\theta_A\} & \text{if } b = 1. \end{cases}$$

Hence, given Succ^A and Eq^A we can implement Move^A in constant time using bit operations. To efficiently represent Eq^A, for each subautomaton A, store the value of $\mathsf{Eq}^A(\alpha)$ in a hash table. Since the total number of different characters in A is at most z the hash table Eq^A contains at most z entries. Hence, we can represent Eq^A for all subautomata is $O(m)$ space and constant worst-case lookup time. The preprocessing time is $O(m)$ w.h.p.. To get a worst-case preprocessing bound we use the deterministic dictionary of [11] with $O(m \log m)$ worst-case preprocessing time.

We note that the idea of using $\mathsf{Eq}^A(\alpha)$ to represent the α-nodes is not new and has been used in several string matching algorithms, for instance, in the classical Shift-Or algorithm [2] and in the recent optimized DFA construction for regular expression matching [23].

To represent Succ compactly we proceed as follows. Let $\hat{A}$ be the automaton obtained by removing the labels from edges in A. Succ^{A_1} and Succ^{A_2} compute the same function if $\hat{A}_1 = \hat{A}_2$. Hence, to represent Succ it suffices to precompute Succ on all possible subautomata $\hat{A}$. By the one-to-one correspondence of parse trees and automata we have that each subautomata $\hat{A}$ corresponds to a parse tree with leaf labels removed. Each such parse tree has at most x internal nodes and $2x$ leaves. The number of rooted, ordered, binary trees with at most $3x$ nodes is less than 2^{6x+1}, and for each such tree each internal node can have one of 3 different labels. Hence, the total number of distinct subautomata is less than $2^{6x+1}3^x$. Each subautomaton has at most $6x$ nodes and therefore the result of Succ^A has to be computed for each of the 2^{6x} different values for $\vec{B}$ using $O(x2^{6x})$ time. Therefore we can precompute all values of Succ in $O(x2^{12x+1}3^x)$ time. Choosing x such that $x + \frac{\log x}{12+\log 3} \le \frac{k-1}{12+\log 3}$ gives us $O(2^k)$ space and preprocessing time.

Using an analogous argument, it follows that Close^A can be precomputed for all distinct subautomata within the same complexity. By our discussion in the previous sections and since $x = \Theta(k)$ we have shown the following theorem:

Theorem 1. *For regular expression R of length m, string Q of length n, and $k \le w$,* Regular Expression Matching *can be solved in $O(mn/k + n + m \log m)$ time and $O(2^k + m)$ space.*

3. Approximate regular expression matching

Given a string Q, a regular expression R, and an integer $d \ge 0$, the Approximate Regular Expression Matching problem is to determine if Q is within edit distance d to a string in $L(R)$. In this section we extend our solution for Regular Expression Matching to Approximate Regular Expression Matching. Specifically, we show that the problem can be solved in $O(\frac{mn\log(d+2)}{k} + n + m \log m)$ time and $O(2^k + m)$ space, for any $k \le w$.

Our result is achieved through a new encoding of subautomata within an algorithm by Wu et al. [32] in a style similar to the above result for Regular Expression Matching. For completeness we restate the algorithm of Wu et al. [32] in Sections 3.1 and 3.2. The new encoding is given in Section 3.3.

3.1. Dynamic programming recurrence

Our algorithm is based on a dynamic programming recurrence due to Myers and Miller [18], which we describe below. Let $\Delta(v, i)$ denote the minimum over all paths $\mathcal{P}$ between θ and v of the edit distance between $\mathcal{P}$ and the ith prefix of Q.

The recurrence avoids cyclic dependencies from the back edges by splitting the recurrence into two passes. Intuitively, the first pass handles forward edges and the second pass propagates values from back edges. The *pass-1 value* of v is denoted $\Delta_1(v, i)$, and the *pass-2 value* is $\Delta_2(v, i)$. For a given i, the *pass-1 (resp. pass-2) value of* $N(R)$ is the set of pass-1 (resp. pass-2) values of all nodes of $N(R)$. For all v and i, we set $\Delta(v, i) = \Delta_2(v, i)$.

The set of *predecessors* of v is the set of nodes $\mathrm{Pre}(v) = \{w \mid (w, v) \text{ is an edge}\}$. We define $\overline{\mathrm{Pre}}(v) = \{w \mid (w, v) \text{ is a forward edge}\}$. For notational convenience, we extend the definitions of Δ_1 and Δ_2 to apply to sets, as follows: $\Delta_1(\mathrm{Pre}(v), i) = \min_{w \in \mathrm{Pre}(v)} \Delta_1(w, i)$ and $\Delta_1(\overline{\mathrm{Pre}}(v), i) = \min_{w \in \overline{\mathrm{Pre}}(v)} \Delta_1(w, i)$, and analogously for Δ_2. The pass-1 and pass-2 values satisfy the following recurrence:

$$\Delta_2(\theta, i) = \Delta_1(\theta, i) = i \qquad 0 \le i \le n.$$

$$\Delta_2(v, 0) = \Delta_1(v, 0) = \min \begin{cases} \Delta_2(\overline{\mathrm{Pre}}(v), 0) + 1 & \text{if } v \text{ is a } \Sigma\text{-node}, \\ \Delta_2(\overline{\mathrm{Pre}}(v), 0) & \text{if } v \neq \theta \text{ is an } \epsilon\text{-node}. \end{cases}$$

For $1 \le i \le n$,

$$\Delta_1(v, i) = \begin{cases} \min(\Delta_2(v, i-1) + 1, \Delta_2(\mathrm{Pre}(v), i) + \lambda(v, Q[i]), \Delta_1(\overline{\mathrm{Pre}}(v), i) + 1) & \text{if } v \text{ is a } \Sigma\text{-node}, \\ \Delta_1(\overline{\mathrm{Pre}}(v), i) & \text{if } v \neq \theta \text{ is an } \epsilon\text{-node}, \end{cases}$$

where $\lambda(v, Q[i]) = 1$ if v is a $Q[i]$-node and 0 otherwise,

$$\Delta_2(v, i) = \begin{cases} \min(\Delta_1(\mathrm{Pre}(v), i), \Delta_2(\overline{\mathrm{Pre}}(v), i)) + 1 & \text{if } v \text{ is a } \Sigma\text{-node}, \\ \min(\Delta_1(\mathrm{Pre}(v), i), \Delta_2(\overline{\mathrm{Pre}}(v), i)) & \text{if } v \text{ is a } \epsilon\text{-node}. \end{cases}$$

A full proof of the correctness of the above recurrence can be found in [18,32]. Intuitively, the first pass handles forward edges as follows: For Σ-nodes the recurrence handles insertions, substitution/matches, and deletions (in this order). For ϵ-nodes the values computed so far are propagated. Subsequently, the second pass handles the back edges. For our problem we want to determine if Q is within edit distance d. Hence, we can replace all values exceeding d by $d + 1$.

3.2. Simulating the recurrence

Our algorithm now proceeds analogously to the case with $d = 0$ above. We will decompose the automaton into subautomata, and we will compute the above dynamic program on an appropriate encoding of the subautomata, leading to a small-space speedup.

As before, we decompose $N(R)$ into subautomata of size less than z. For a subautomaton A we define operations Next_1^A and Next_2^A which we use to compute the pass-1 and pass-2 values of A, respectively. However, the new (pass-1 or pass-2) value of A depends on pseudo-edges in a more complicated way than before: If A' is a child of A, then all nodes preceding $\phi_{A'}$ depend on the value of $\phi_{A'}$. Hence, we need the value of $\phi_{A'}$ before we can compute values of the nodes preceding $\phi_{A'}$. To address this problem we partition the nodes of a subautomaton as described below.

For each subautomaton A topologically sort the nodes (ignoring back edges) with the requirement that for each child A' the start and accepting nodes $\theta_{A'}$ and $\phi_{A'}$ are consecutive in the order. Contracting all pseudo-edges in A this can be done for all subautomata in $O(m)$ time. Let $A_1, \ldots, A_\ell$ be the children of A in this order. We partition the nodes in A, except $\{\theta_A\} \cup \{\phi_{A_1}, \ldots, \phi_{A_\ell}\}$, into $\ell + 1$ *chunks*. The first chunk is the nodes in the interval $[\theta_A + 1, \theta_{A_1}]$. If we let $\phi_{A_{\ell+1}} = \phi_A$, then the ith chunk, $1 \le i \le \ell + 1$, is the set of nodes in the interval $[\phi_{A_{i-1}} + 1, \theta_{A_i}]$. A leaf automaton has a single chunk consisting of all nodes except the start node. We represent the ith chunk in A by a characteristic vector $\vec{L_i}$ identifying the nodes in the chunks, that is, $\vec{L_i}[j] = 1$ if node j is in the ith chunk and 0 otherwise. From the topological order we can compute all chunks and their corresponding characteristic vectors in total $O(m)$ time.

The value of A is represented by a vector $\vec{B} = [b_1, \ldots, b_z]$, such that $b_i \in [0, d + 1]$. Hence, the total number of bits used to encode $\vec{B}$ is $z \lceil \log d + 2 \rceil$ bits. For an automaton A, characteristic vectors $\vec{B}$ and $\vec{L}$, and a character $\alpha \in \Sigma$ define the operations $\mathsf{Next}_1^A(\vec{B}, \vec{L}, b, \alpha)$ and $\mathsf{Next}_2^A(\vec{B}, \vec{L}, b)$ as the vectors $\vec{B}_1$ and $\vec{B}_2$, respectively, given by:

$$\vec{B}_1[v] = B[v] \quad \text{if } v \notin \vec{L}$$

$$\vec{B}_1[v] = \begin{cases} \min(\vec{B}[v] + 1, \vec{B}[\mathrm{Pre}(v)] + \lambda(v, \alpha), \vec{B}_1[\overline{\mathrm{Pre}}(v)] + 1) & \text{if } v \in \vec{L} \text{ is a } \Sigma\text{-node}, \\ \vec{B}_1[\mathrm{Pre}(v)] & \text{if } v \in \vec{L} \text{ is an } \epsilon\text{-node} \end{cases}$$

$$\vec{B}_2[v] = B[v] \quad \text{if } v \notin \vec{L}$$

$$\vec{B}_2[v] = \begin{cases} \min(\vec{B}[\mathrm{Pre}(v)], \vec{B}_2[\overline{\mathrm{Pre}}(v)] + 1) & \text{if } v \in \vec{L} \text{ is a } \Sigma\text{-node}, \\ \min(\vec{B}[\mathrm{Pre}(v)], \vec{B}_2[\overline{\mathrm{Pre}}(v)]) & \text{if } v \notin \vec{L} \text{ is an } \epsilon\text{-node}. \end{cases}$$

Importantly, note that the operations only affect the nodes in the chunk specified by $\vec{L}$. We will use this below to compute new values of A by advancing one chunk at each step. We use the following recursive operations. For subautomaton A, integer b, and character α define:

```
$\mathsf{Next}_1(A, b, \alpha)$:  Let $\vec{B}$ be the current value of $A$ and let $A_1, \dots, A_\ell$ be children of $A$ in topological order of their start node.
    1. Set $\vec{B}_1 := \vec{B}$ and $\vec{B}_1[\theta_A] := b$.
    2. For each chunk $L_i$, $1 \le i \le \ell$,
        (a) Compute $\vec{B}_1 := \mathsf{Next}^A_1(\vec{B}_1, \vec{L}_i, \alpha)$.
        (b) Compute $f_i := \mathsf{Next}_1(A_i, \vec{B}_1[\theta_{A_i}], \alpha)$.
        (c) Set $\vec{B}_1[\phi_{A_i}] := f_i$.
    3. Compute $\vec{B}_1 := \mathsf{Next}^A_1(\vec{B}_1, \vec{L}_{\ell+1}, \alpha)$.
    4. Return $\vec{B}_1[\phi_A]$.

$\mathsf{Next}_2(A, b)$:  Let $\vec{B}$ be the current value of $A$ and let $A_1, \dots, A_\ell$ be children of $A$ in topological order of their start node.
    1. Set $\vec{B}_2 := \vec{B}$ and $\vec{B}_2[\theta_A] := b$.
    2. For each chunk $L_i$, $1 \le i \le \ell$,
        (a) Compute $\vec{B}_2 := \mathsf{Next}^A_2(\vec{B}_2, \vec{L}_i)$.
        (b) Compute $f_i := \mathsf{Next}_2(A_i, \vec{B}_2[\theta_{A_i}])$.
        (c) Set $\vec{B}_2[\phi_{A_i}] := f_i$.
    3. Compute $\vec{B}_2 := \mathsf{Next}^A_2(\vec{B}_2, \vec{L}_{\ell+1})$.
    4. Return $\vec{B}_2[\phi_A]$.
```

The simulation of the dynamic programming recurrence on a string Q of length n proceeds as follows: First encode the initial values of the all nodes in $N(R)$ using the recurrence. Let A_r be the root automaton, let S_{j-1} be the current value of $N(R)$, and let $\alpha = Q[j]$. Compute the next value S_j by calling $\mathsf{Next}_1(A_r, j, \alpha)$ and then $\mathsf{Next}_2(A_r, j, \alpha)$. Finally, if the value of ϕ in the pass-2 value of S_n is less than d, report a match.

To see the correctness, we need to show that the calls Next_1 and Next_2 operations correctly compute the pass-1 and pass-2 values of $N(R)$. First consider Next_1, and let A be any subautomaton. The key property is that if p_1 is the pass-1 value of θ_A then after a call to $\mathsf{Next}_1(A, p_1, \alpha)$, the value of A is correctly updated to the pass-1 value. This follows by a straightforward induction similar to the exact case. Since the pass-1 value of θ after reading the jth prefix of Q is j, the correctness of the call to Next_1 follows. For Next_2 the result follows by an analogous argument.

3.3. Representing subautomata

Next we show how to efficiently represent Next^A_1 and Next^A_2. First consider Next^A_1. Note that again the alphabet size is a problem. Since the $\vec{B}_1$ value of a node in A depends on other $\vec{B}_1$ values in A we cannot "split" the computation of Next^A_1 as before. However, the alphabet character only affects the value of $\lambda(v, \alpha)$, which is 1 if v is an α-node and 0 otherwise. Hence, we can represent $\lambda(v, \alpha)$ for all nodes in A with $\mathsf{Eq}^A(\alpha)$ from the previous section. Recall that $\mathsf{Eq}^A(\alpha)$ can be represented for all subautomata in total $O(m)$ space. With this representation the total number of possible inputs to Next^A_1 can be represented using $(d+2)^z + 2^{2z}$ bits. Note that for $z = \frac{k}{\log(d+2)}$ we have that $(d+2)^z = 2^k$. Furthermore, since Next^A_1 is now alphabet independent we can apply the same trick as before and only precompute it for all possible parse trees with leaf labels removed. It follows that we can choose $z = \Theta(\frac{k}{\log(d+2})$ such that Next^A_1 can precomputed in total $O(2^k)$ time and space. An analogous argument applies to Next^A_2. Hence, by our discussion in the previous sections we have shown that,

Theorem 2. *For regular expression R of length m, string Q of length n, and integer $d \ge 0$* Approximate Regular Expression Matching *can be solved in $O(\frac{mn\log(d+2)}{k} + n + m\log m)$ time and $O(2^k + m)$ space, for any $k \le w$.*

4. String edit distance

The String Edit Distance problem is to compute the minimum number of edit operations needed to transform a string S into a string T. Let m and n be the size of S and T, respectively. The classical solution to this problem, due to Wagner and Fischer [29], fills in the entries of an $m+1 \times n+1$ matrix D. The entry $D_{i,j}$ is the edit distance between $S[1..i]$ and $T[1..j]$, and can be computed using the following recursion:

$$\begin{aligned} D_{i,0} &= i \\ D_{0,j} &= j \\ D_{i,j} &= \min\{D_{i-1,j-1} + \lambda(i,j), D_{i-1,j} + 1, D_{i,j-1} + 1\} \end{aligned}$$

where $\lambda(i,j) = 0$ if $S[i] = T[j]$ and 1 otherwise. The edit distance between S and T is the entry $D_{m,n}$. Using dynamic programming the problem can be solved in $O(mn)$ time. When filling out the matrix we only need to store the previous row or column and hence the space used is $O(\min(m,n))$. For further details, see the book by Gusfield [10, Chap. 11].

The best algorithm for this problem, due to Masek and Paterson [14], improves the time to $O(\frac{mn}{k^2} + m + n)$ time and $O(2^k + \min(m, n))$ space, for any $k \leq w$. This algorithm, however, assumes that the alphabet size is constant. In this section we give an algorithm using $O(\frac{mn \log^2 k}{k^2} + m + n)$ time and $O(2^k + \min(m, n))$ space, for any $k \leq w$, that works for any alphabet. Hence, we remove the dependency of the alphabet at the cost of a $\log^2 k$ factor.

We first describe the algorithm by Masek and Paterson [14], and then modify it to handle arbitrary alphabets. The algorithm uses the Four Russian Technique. The matrix D is divided into *cells* of size $x \times x$ and all possible inputs of a cell is then precomputed and stored in a table. From the above recursion it follows that the values inside each cell C depend on the corresponding substrings in S and T, denoted S_C and T_C, and on the values in the top row and the leftmost column in C. The number of different strings of length x is σ^x and hence there are σ^{2x} possible choices for S_C and T_C. Masek and Paterson [14] showed that adjacent entries in D differ by at most one, and therefore if we know the value of an entry there are exactly three choices for each adjacent entry. Since there are at most m different values for the top left corner of a cell it follows that the number of different inputs for the top row and the leftmost column is $m3^{2x}$. In total, there are at $m(\sigma 3)^{2x}$ different inputs to a cell. Assuming that the alphabet has constant size, we can choose $x = \Theta(k)$ such that all cells can be precomputed in $O(2^k)$ time and space. The input of each cell is stored in a single machine word and therefore all values in a cell can be computed in constant time. The total number of cells in the matrix is $O(\frac{mn}{k^2})$ and hence this implies an algorithm using $O(\frac{mn}{k^2} + m + n)$ time and $O(2^k + \min(m, n))$ space.

We show how to generalize this to arbitrary alphabets. The first observation, similar to the idea in Section 3, is that the values inside a cell C does not depend on the actual characters of S_C and T_C, but only on the λ function on S_C and T_C. Hence, we only need to encode whether or not $S_C[i] = T_C[j]$ for all $1 \leq i, j \leq x$. To do this we assign a code $c(\alpha)$ to each character α that appears in T_C or S_C as follows. If α only appears in only one of S_C or T_C then $c(\alpha) = 0$. Otherwise, $c(\alpha)$ is the rank of α in the sorted list of characters that appears in both S_C and T_C. The representation is given by two vectors $\vec{S}_C$ and $\vec{T}_C$ of size x, where $\vec{S}_C[i] = c(S_C[i])$ and $\vec{T}_C[i] = c(T_C[i])$, for all i, $1 \leq i \leq x$. Clearly, $S_C[i] = T_C[j]$ iff $\vec{S}_C[i] = \vec{T}_C[j]$ and $\vec{S}_C[i] > 0$ and $\vec{T}_C[j] > 0$ and hence $\vec{S}_C$ and $\vec{T}_C$ suffices to represent λ on C.

The number of characters appearing in both T_C and S_C is at most x and hence each entry of the vectors is assigned an integer value in the range $[1, x]$. Thus, the total number of bits needed for both vectors is $2x \lceil \log x + 1 \rceil$. Hence, we can choose $x = \Theta(\frac{k}{\log k})$ such that the input vectors for a cell can be represented in a single machine word. The total number of cells becomes $O(\frac{mn}{x^2}) = O(\frac{nm \log^2 k}{k^2})$. Hence, if the input vectors for each cell is available we can use the Four Russian Technique to get an algorithm for String Edit Distance using $O(\frac{nm \log^2 k}{k^2} + m + n)$ time and $O(2^k + \min(m, n))$ space as desired.

Next we show how to compute vectors efficiently. Given any cell C, we can identify the characters appearing in both S_C and T_C by sorting S_C and then for each index i in T_C use a binary search to see if $T_C[i]$ appears in S_C. Next we sort the characters appearing in both substrings and insert their ranks into the corresponding positions in $\vec{S}_C$ and $\vec{T}_C$. All other positions in the vectors are given the value 0. This algorithm uses $O(x \log x)$ time for each cell. However, since the number of cells is $O(\frac{nm}{x^2})$ the total time becomes $O(\frac{nm \log x}{x})$, which for our choice of x is $O(\frac{nm(\log k)^2}{k})$. To improve this we group the cells into *macro cells* of $y \times y$ cells. We then compute the vector representation for each of these macro cells. The vector representation for a cell C is now the corresponding subvectors of the macro cell containing C. Hence, each vector entry is now in the range $[0, \ldots, xy]$ and thus uses $\lceil \log(xy + 1) \rceil$ bits. Computing the vector representation uses $O(xy \log(xy))$ time for each macro cell and since the number of macro cells is $O(\frac{nm}{(xy)^2})$ the total time to compute it is $O(\frac{nm \log(xy)}{xy} + m + n)$. It follows that we can choose $y = k \log k$ and $x = \Theta(\frac{k}{\log k})$ such that vectors for a cell can be represented in a single word. With this choice of x and y we have that $xy = \Theta(k^2)$ and hence all vectors are computed in $O(\frac{nm \log(xy)}{xy} + m + n) = O(\frac{nm \log k}{k^2} + m + n)$ time. Computing the distance matrix dominates the total running time and hence we have shown:

Theorem 3. *For strings S and T of length n and m, respectively,* String Edit Distance *can be solved in $O(\frac{mn \log^2 k}{k^2} + m + n)$ time and $O(2^k + \min(m, n))$ space.*

5. Subsequence indexing

The Subsequence Indexing problem is to preprocess a string T to build a data structure supporting queries of the form:"is Q a subsequence of T?" for any string Q. This problem was considered by Baeza-Yates [3] who showed the trade-offs listed in Table 1. We assume throughout the section that T and Q have n and m, respectively. For properties of automata accepting subsequences of string and generalizations of the problem see the recent survey [8].

Using recent data structures and a few observations we improve all previous bounds. As a notational shorthand, we will say that a data structure with preprocessing time and space $f(n, \sigma)$ and query time $g(m, n, \sigma)$ has complexity $\langle f(n, \sigma), g(m, n, \sigma) \rangle$

Let us consider the simplest algorithm for Subsequence Indexing. One can build a DFA of size $O(n\sigma)$ for recognizing all subsequences of T. To do so, create an accepting node for each character of T, and for node v_i, corresponding to character

Table 1
Trade-offs for Subsequence Indexing

Space	Preprocessing	Query
$O(n\sigma)$	$O(n\sigma)$	$O(m)$
$O(n\log\sigma)$	$O(n\log\sigma)$	$O(m\log\sigma)$
$O(n)$	$O(n)$	$O(m\log n)$

$T[i]$, create an edge to v_j on character α if $T[j]$ is the first α after position i. The start node has edges to the first occurrence of each character. Such an automaton yields an algorithm with complexity $\langle O(n\sigma), O(m)\rangle$.

An alternative is to build, for each character α, a data structure D_α with the positions of α in T. D_α should support fast successor queries. The D_α's can all be built in a total of linear time and space using, for instance, van Emde Boas trees and perfect hashing [27,28,15]. These trees have query time $O(\log\log n)$. We use these vEB trees to simulate the above automaton-based algorithm: whenever we are in state v_i, and the next character to be read from P is α, we look up the successor of i in D_α in $O(\log\log n)$ time. The complexity of this algorithm is $\langle O(n), O(m\log\log n\rangle$.

We combine these two data structures as follows: Consider an automaton consisting of nodes $u_1, \dots, u_{n/\sigma}$, where node u_i corresponds to characters $T[\sigma(i-1), \dots, \sigma i - 1]$, that is, each node u_i corresponds to σ nodes in T. Within each such node, apply the vEB based data structure. Between such nodes, apply the full automaton data structure. That is, for node w_i, compute the first occurrence of each character α after $T[\sigma i - 1]$. Call these *long jumps*. A edge takes you to a node u_j, and as many characters of P are consumed with u_j as possible. When no valid edge is possible within w_j, take a long jump. The automaton uses $O(\frac{n}{\sigma}\cdot\sigma) = O(n)$ space and preprocessing time. The total size of the vEB data structures is $O(n)$. Since each u_i consist of at most σ nodes, the query time is improved to $O(\log\log\sigma)$. Hence, the complexity of this algorithm is $\langle O(n), O(m\log\log\sigma)\rangle$. To get a trade-off we can replace the vEB data structures by a recent data structure of Thorup [26, Thm. 2]. This data structure supports successor queries of x integers in the range $[1, X]$ using $O(xX^{1/2^\ell})$ preprocessing time and space with query time $O(\ell+1)$, for $0 \le \ell \le \log\log X$. Since each of the n/σ groups of nodes contain at most σ nodes, this implies the following result:

Theorem 4. Subsequence Indexing *can be solved in* $\left\langle O(n\sigma^{1/2^\ell}), O(m(\ell+1))\right\rangle$, *for* $0 \le \ell \le \log\log\sigma$.

Corollary 1. Subsequence Indexing *can be solved in* $\langle O(n\sigma^\epsilon), O(m)\rangle$ *or* $\langle O(n), O(m\log\log\sigma)\rangle$.

Proof. We set ℓ to be a constant or $\log\log\sigma$, respectively. □

We note that using a recent data structure for rank and select queries on large alphabets by Golynski et al. [9] we can also immediately obtain an algorithm using time $O(m\log\log\sigma)$ and space $n\log\sigma + o(n\log\sigma)$ *bits*. Hence, this result matches our fastest algorithm while improving the space from $O(n)$ words to the number of bits needed to store T.

Acknowledgments

Thanks to the anonymous reviewers for many detailed and insightful comments. The first author's research was supported by the Danish Agency for Science, Technology, and Innovation.

References

[1] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev, On economic construction of the transitive closure of a directed graph, Dokl. Acad. Nauk 194 (1970) 487–488 (in Russian). English translation in Soviet Math. Dokl. 11, 1209–1210, 1975.

[2] R. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82.

[3] R.A. Baeza-Yates, Searching subsequences, Theor. Comput. Sci. 78 (2) (1991) 363–376.

[4] R.A. Baeza-Yates, G.H. Gonnet, Fast text searching for regular expressions or automaton searching on tries, J. ACM 43 (6) (1996) 915–936.

[5] R.A. Baeza-Yates, G. Navarro, Faster approximate string matching, Algorithmica 23 (2) (1999) 127–158.

[6] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, M. Arenas, Efficient incremental validation of XML documents, in: Proceedings of the 20th International Conference on Data Engineering, 2004, p. 671.

[7] P. Bille, New algorithms for regular expression matching, in: Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, 2006, pp. 643–654.

[8] M. Crochemore, B. Melichar, Z. Troníček, Directed acyclic subsequence graph: Overview, J. Discrete Algorithms 1 (3–4) (2003) 255–280.

[9] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: A tool for text indexing, in: Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms, 2006, pp. 368–373.

[10] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge, 1997.

[11] T. Hagerup, P.B. Miltersen, R. Pagh, Deterministic dictionaries, J. Algorithms 41 (1) (2001) 69–85.

[12] H. Hosoya, B. Pierce, Regular expression pattern matching for XML, in: Proceedings of the 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL, 2001, pp. 67–80.

[13] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB, 2001, pp. 361–370.

[14] W. Masek, M. Paterson, A faster algorithm for computing string edit distances, J. Comput. System Sci. 20 (1980) 18–31.

[15] K. Mehlhorn, S. Nähler, Bounded ordered dictionaries in $o(\log\log n)$ time and $o(n)$ space, Inform. Process. Lett. 35 (4) (1990) 183–189.

[16] M. Murata, Extended path expressions of XML, in: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS, 2001, pp. 126–137.

[17] E.W. Myers, A four-russian algorithm for regular expression pattern matching, J. ACM 39 (2) (1992) 430–448.

[18] E.W. Myers, W. Miller, Approximate matching of regular expressions, Bull. Math. Biol. 51 (1989) 5–37.

[19] G. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, J. ACM 46 (3) (1999) 395–415.

[20] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.

[21] G. Navarro, NR-grep: A fast and flexible pattern-matching tool, Software — Practice Experience 31 (13) (2001) 1265–1312.

[22] G. Navarro, Approximate regular expression searching with arbitrary integer weights, Nordic J. Comput. 11 (4) (2004) 356–373.

[23] G. Navarro, M. Raffinot, New techniques for regular expression searching, Algorithmica 41 (2) (2004) 89–116.

[24] R.M. Stallman, Emacs the extensible, customizable self-documenting display editor, SIGPLAN Not. 16 (6) (1981) 147–156.

[25] K. Thompson, Regular expression search algorithm, Commun. ACM 11 (1968) 419–422.

[26] M. Thorup, Space efficient dynamic stabbing with fast queries, in: Proceedings of the Symposium on Theory of Computing, STOC, 2003, pp. 649–658.

[27] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (3) (1977) 80–82.

[28] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Mathematical Systems Theory 10 (1977) 99–127.

[29] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.

[30] S. Wu, U. Manber, Agrep — a fast approximate pattern-matching tool, in: Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162.

[31] S. Wu, U. Manber, Fast text searching: Allowing errors, Commun. ACM 35 (10) (1992) 83–91.

[32] S. Wu, U. Manber, E.W. Myers, A subquadratic algorithm for approximate regular expression matching, J. Algorithms 19 (3) (1995) 346–360.