APPROXIMATE REGULAR EXPRESSION SEARCHING WITH ARBITRARY INTEGER WEIGHTS *

GONZALO NAVARRO†
Dept. of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. We present a bit-parallel technique to search a text of length n for a regular expression of m symbols permitting k differences in worst case time $O(mn/\log_k s)$, where s is the amount of main memory that can be allocated. The algorithm permits arbitrary integer weights and matches the complexity of the best previous techniques, but it is simpler and faster in practice. In our way, we define a new recurrence for approximate searching where the current values depend only on previous values. Interestingly, our algorithm turns out to be a relevant option also for simple approximate string matching with arbitrary integer weights.

ACM CCS Categories and Subject Descriptors: E.1. Data structures; F.2.2. Nonnumerical Algorithms and Problems — Computations on discrete structures, Pattern matching, Sorting and searching; H.3. Information Storage and Retrieval

Key words: Approximate string matching, string matching with differences, regular expression searching, bit-parallelism, computational biology.

1. Introduction and related work

The need to search for regular expressions arises in many text-based applications, such as text retrieval, text editing and computational biology, to name a few. A *regular expression* is a generalized pattern composed of (i) basic strings, (ii) union, concatenation and Kleene closure of other regular expressions. Readers unfamiliar with the concept and terminology related to regular expressions are referred to a classical book such as [1]. We call m the length of our regular expression, not counting operator symbols. The alphabet is denoted by Σ, and n is the length of the text.

The traditional technique to search for a regular expression [1] first builds a nondeterministic finite automaton (NFA) and then converts it to a deterministic finite automaton (DFA), which is finally used to search the text in $O(n)$ time. This is worst-case optimal in terms of n. The main problem has been always the preprocessing time and space requirement to code the DFA, which can be as high as $O(2^{2m}|\Sigma|)$ if the classical Thompson's NFA construction algorithm [13] is used.

---

*An extended abstract of this paper appeared in *Proc. ISAAC'03* [10].

†Partially supported by Fondecyt grant 1-020831.

Received ...; revised ...; accepted ....

Thompson's construction produces up to $2m$ states, but it has interesting properties, such as ensuring a linear number of edges, constant indegree and outdegree, etc.

An alternative NFA construction is Glushkov's [4, 3]. Although it does not provide the same regularities of Thompson's, this construction has other interesting properties, such as producing the minimum number of states ($m + 1$) and that all the edges arriving at a node are labeled by the same character. The corresponding DFA needs only $O(2^m|\Sigma|)$ space, which is significantly less than the worst case using Thompson's NFA. Nevertheless, this is still exponential in m.

Two techniques have been classically used to cope with the space problem. The first is to use lazy DFAs, where the states are built only when they are reached. This ensures that no more than $O(n)$ extra space is necessary. The second choice [13] is to directly use the NFA instead of converting it to deterministic. This requires only $O(m)$ space, but the search time becomes $O(mn)$. Both approaches are slow in practice if the regular expression is large.

Newer techniques have provided better space-time tradeoffs by using hybrids between the NFA and the DFA. Based on the Four Russians technique, which precomputes large tables that permit processing several NFA states in one shot, it has been shown that $O(mn/\log s)$ search time is possible using $O(s)$ space [6]. The use of Thompson's automaton is essential for this approach which, however, is rather complicated. Simpler solutions obtaining the same complexities have been obtained later using bit-parallelism, a technique to pack several NFA states in a single machine word and update them as a single state. A first solution [16], based on Thompson's construction, uses a table of size $O(2^{2m})$ that can be split into t tables of size $O(2^{2m/t})$ each, at a search cost of $O(tn)$ table inspections. A second solution [11] uses Glushkov's automaton and uses t tables of size $O(2^{m/t})$ each, which is much more efficient in space usage. In both cases, $O(mn/\log s)$ search time is obtained using $O(s)$ space.

Several applications in computational biology, data mining, text retrieval, etc. need an even more sophisticated form of searching. In addition to the regular expression, an integer threshold k is given, so that we have to report the text substrings that can match the regular expression after performing several character insertions, deletions and substitutions, whose total *cost* or *weight* does not exceed k. In most real applications, there are different weights associated to insertions, deletions, and substitutions, depending on the characters involved. This problem is called "approximate regular expression searching", as opposed to "exact" searching.

Instead of being just active or inactive, every NFA node has now $k + 2$ possible states, according to the weight of the differences needed to match the text (0 to k, or more than k). If one applies the classical DFA construction algorithm, the space requirement raises to $O((k + 2)^{2m})$ using Thompson's NFA and $O((k + 2)^m)$ using Glushkov's NFA. A dynamic programming based solution with $O(mn)$ time and $O(m)$ space exists [7]. Although this is an achievement because it retains the time complexity of the exact search version and handles real-valued weights, it is still slow. The Four Russians technique has been gracefully extended to this problem [17], obtaining $O(mn/\log_k s)$ time using $O(s)$ space. Again, this algorithm is rather complicated.

Since bit-parallel solutions have, for many related problems, yielded fast and simple solutions, one may wonder what have they achieved here. For the case of unitary costs (that is, all the weights are 1), bit-parallel solutions exist which resort to simulating $k + 1$ copies of the NFA used for exact searching. They achieve $O(ktn)$ time using $O(2^{2m/t})$ space [16] (implemented in the software *Agrep* [15]) or $O(2^{m/t})$ space (implemented in the software *Nrgrep* [9]). This yields $O(kmn/\log s)$ time using $O(s)$ space, which is inferior to the achievement of the Four Russians technique. Despite this less attractive complexity, bit-parallel solutions are by far the fastest for moderate sized regular expressions. Yet, they are restricted to the simpler case of unitary costs.

The aim of this paper is to overcome the technical problems that have prevented the existence of a simple $O(mn/\log_k s)$ time and $O(s)$ space bit-parallel solution to approximate regular expression searching with arbitrary integer weights. We build over Glushkov's NFA and represent the state of the search using $m\lceil 1 + \log_2(k+2)\rceil$ bits. We then use t tables of size $O((k+2)^{m/t})$ each and reach $O(tn)$ search time.

Table I illustrates the context of our contribution. We point out, however, that bit-parallel complexities assume that the computer can handle words of arbitrary length in constant time. If we use the RAM model, where the computer can handle words of $w = \Theta(\log n)$ bits in constant time, previous bit-parallel complexities get multiplied by $O(m/w)$ and ours by $O(m \log(k)/w)$. In this setting, we match the Four Russians complexity only when $m = O(\log_k n)$.

	Exact Searching	Approximate Searching	
		Unit Cost	General Costs
Dynamic Programming	$O(mn)$ [13] Thompson		$O(mn)$ [7] Myers & Miller
Four Russians	$O(mn/\log s)$ [6] Myers		$O(mn/\log_k s)$ [17] Wu, Manber & Myers
Bit Parallelism	$O(mn/\log s)$ [16, 11] Wu & Manber Navarro & Raffinot	$O(kmn/\log s)$ [16, 9] Wu & Manber Navarro	$O(mn/\log_k s)$ [10] This paper

Table I: Our contribution in context.

We use the following terminology for bit-parallel algorithms. A *bit mask* is a sequence of bits, where the lowest bit is written at the right. Typical bit operations are infix "|" (bitwise *or*), infix "&" (bitwise *and*), prefix "∼" (bit complementation), and infix "<<" (">>"), which moves the bits of the first argument (a bit mask) to higher (lower) positions in an amount given by the argument on the right. Additionally, one can treat the bit masks as numbers and obtain specific effects using the arithmetic operations "+", "−", etc. Exponentiation is used to denote bit repetition, e.g., $0^3 1 = 0001$, and $[x]_\ell$ represents an integer x using ℓ bits. Finally, $X \times x$, where X is a bit mask and x is a number, is the exact result of the multiplication, that is, a bit mask where x appears in the places where X has 1's (superimpositions are solved with summation, as in usual multiplication, but we never use that feature).

2. A bit-parallel exact search algorithm

We describe in this section the exact bit-parallel solution we build on [11]. The classical algorithm to produce a DFA from an NFA [1] consists in making each DFA state represent a set of NFA states that may be active at some point. Our way to represent the states of a DFA (i.e., the sets of states of an NFA) is a bit mask of $O(m)$ bits. The bit mask has in 1 the bits that belong to the set. We use set notation or bit mask notation indistinctly.

The description of Glushkov's NFA construction algorithm is outside the scope of this paper [4, 3]. We just show an example in Fig. 1 and remark some of its properties. Given a regular expression of m characters (not counting operator symbols), the algorithm defines $m+1$ *positions* numbered 0 to m (one per position of a character of Σ in the regular expression, plus an initial position 0). Then, the NFA has exactly one state per position, the initial state corresponding to position 0. Two tables are built: $B(\sigma)$, the set of positions of the regular expression that contain character σ; and $Follow(x)$, the set of NFA states that can be reached from state x in one transition[1]. From these two tables, the transition function of the NFA is computed: $\delta : \{0 \ldots m\} \times \Sigma \rightarrow \wp(\{0 \ldots m\})$, such that $y \in \delta(x, \sigma)$ if and only if from state x we can move to state y by character σ. The algorithm gives also a set of final states, *Last*, which again will be represented as a bit mask.

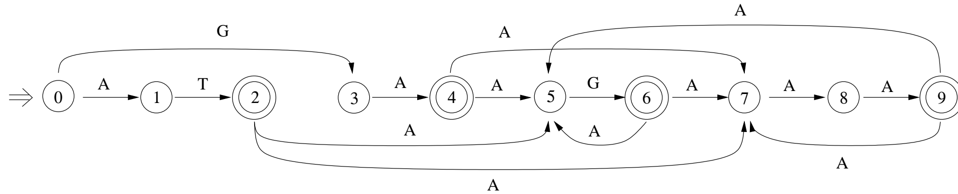


Fig. 1: Glushkov's NFA for the regular expression `"(AT|GA)(AG|AAA)*"`.

Important properties of Glushkov's construction follow. (1) The NFA is ε-free. (2) All the arrows leading to a given NFA state are labeled by the same character: the one at the corresponding position. (3) The initial state does not receive any transition. (4) $\delta(x, \sigma) = Follow(x) \cap B(\sigma)$.

Property (4) permits a compact representation of the DFA transitions. The construction algorithm is written so that tables B and $Follow$ represent the sets of states as bit masks. We use B as is and build a large table J, the deterministic version of $Follow$. That is, J is a table that, for every bit mask D representing a set of states, stores $J[D] = \bigcup_{i \in D} Follow(i)$. Then, by Property (4) it holds that, if the current set of active states is D and we read text character σ, then the new set of active states is $J[D] \cap B[\sigma]$. For search purposes, we set state 0 in $J[D]$ for every D and in $B[\sigma]$ for every σ, and report every text position j where $D \cap Last \neq \emptyset$. (In fact, state 0 needs not be represented, since it is always active when searching.)

[1] This is computed from the regular expression, since the NFA does not yet exist. Also, to simplify the discussion, we assume that $Follow(0) = First$, the states reachable from the initial state.

Hence we need only $O(2^m + |\Sigma|)$ space instead of the $O(2^m|\Sigma|)$ space of the classical representation. Space-time tradeoffs are achieved by splitting table J. The splitting is done as follows. We build two tables J_1 and J_2, which give the set of states reached from states $0 \ldots \ell$ and $\ell + 1 \ldots m$, respectively, with $\ell = \lfloor (m+1)/2 \rfloor$. Then, if we accordingly split the current set of states D into left and right submasks, $D = D_1 : D_2$, we have $J[D] = J_1[D_1] \cup J_2[D_2]$. Tables J_1 and J_2 need only $O(2^{m/2})$ space each. This generalizes to using t tables, for an overall space requirement of $O(t2^{m/t})$ and a search cost of $O(tn)$ table accesses.

3. A new recurrence for approximate searching

Let us first give an exact formulation for our problem. Let R be a regular expression generating language $\mathcal{L}(R) \subseteq \Sigma^*$. Let m be the number of characters belonging to Σ in R. Let $T_{1\ldots n} \in \Sigma^*$ be the text, a sequence of n symbols. The problem is, given R, T, and $k \in \mathbb{N}$, to report every text position j such that, for some $j' \leq j$ and $P \in \mathcal{L}(R)$, $ed(T_{j'\ldots j}, P) \leq k$. Here $ed(A, B)$, the edit distance, is the minimum sum of weights of a sequence of character insertions, deletions and substitutions needed to convert A into B. The weights are represented by a function ω, such that $\omega(a, b)$ is the cost to substitute character a by character b in the text, $\omega(a, \varepsilon)$ is the cost to delete character a from the text, and $\omega(\varepsilon, b)$ is the cost to insert character b in the text. Function ω satisfies $\omega(a, a) = 0$, nonnegativity, and the triangle inequality.

The classical dynamic programming solution for approximate string matching [12], for the case where R is a simple string $P_{1\ldots m}$, recomputes for every text position j a vector $C_{0\ldots m}$, where $C_i = \min_{j' \leq j} ed(T_{j'\ldots j}, P_{1\ldots i})$. Hence every text position j where $C_m \leq k$ is reported. C is initialized as $C_i = \sum_{r=1}^{i} \omega(\varepsilon, P_r)$, and then updated to C' at text position j using dynamic programming:

$$C'_i \leftarrow \min(\omega(T_j, P_i) + C_{i-1},\ \omega(T_j, \varepsilon) + C_i,\ \omega(\varepsilon, P_i) + C'_{i-1})$$

where $C'_0 = 0$. The first component refers to a character matching or substitution, the second to deleting a text character, and the third to inserting a character in the text.

If we have a general regular expression R built using Glushkov's algorithm, with positions 1 to m, this generalizes as follows. For each NFA state i, we call L_i the set of strings recognized by the automaton if we assume that the only final state is i. Then $C_i = \min_{j' \leq j, P \in L_i} ed(T_{j'\ldots j}, P)$ is computed as follows:

$$C'_i \leftarrow \min(\omega(T_j, R_i) + \min_{i' \in Follow^{-1}(i)} C_{i'},\ \omega(T_j, \varepsilon) + C_i,\ \omega(\varepsilon, R_i) + \min_{i' \in Follow^{-1}(i)} C'_{i'}) \quad (1)$$

where R_i is the only character such that $i \in B(R_i)$: Thanks to Property (2), we know that all the edges arriving at state i are labeled by the same character, R_i. C_0 is always 0 because it refers to the initial state, so $L_0 = \{\varepsilon\}$. Vector C is initialized as $C_i = \omega(\varepsilon, R_i) + \min_{i' \in Follow^{-1}(i)} C_{i'}$.

A more convenient form for us of the above recurrence is

$$C'_i \leftarrow \min(S_i(T_j) + \min_{i' \in Follow^{-1}(i)} C_{i'},\ D(T_j) + C_i,\ I_i + \min_{i' \in Follow^{-1}(i)} C'_{i'}) \qquad (2)$$

where $S_i(a) = \omega(a, R_i)$, $D(a) = \omega(a, \varepsilon)$, and $I_i = \omega(\varepsilon, R_i)$.

Note that the main difference in the generalization is that, in the case of a single pattern, every state i has a unique predecessor, state $i-1$. Here, the set of predecessor states, $Follow^{-1}(i)$, can be arbitrarily complex. In the third component of Eq. (2) (insertions in the text) we have a potential dependence problem, because in order to compute C' for state i we need to have already computed C' for states that precede i, in an automaton that can perfectly contain cycles. There are good previous solutions to this circular dependence problem [7], but these are not easy to apply in a bit-parallel context.

We present a new solution now. The central idea is to explicitly consider all the NFA paths that arrive at each state from each other state. It turns out that paths up to a fixed length need to be considered because there is a limit k on the costs of the interesting paths. The recurrence we obtain will be more complex but free of circular dependences. In the next section we show how to precompute all those paths so as to have a fast serach algorithm.

We will use the form $i^{(r)}$ in minimization arguments, whose range is as follows: $i^{(0)} = i$ and $i^{(r+1)} \in Follow^{-1}(i^{(r)})$. Also, we will denote $S_{i^{(r)}} = S_{i^{(r)}}(T_j)$ and $D = D(T_j)$. Let us now unfold the recurrence of Eq. (2):

$$C'_i \leftarrow \min(S_i + \min_{i^{(1)}} C_{i^{(1)}}, D + C_i, I_i + \min_{i^{(1)}} \min(S_{i^{(1)}} + \min_{i^{(2)}} C_{i^{(2)}}, D + C_{i^{(1)}}, I_{i^{(1)}} + \min_{i^{(2)}} C'_{i^{(2)}}))$$

where after a few manipulations we obtain

$$\begin{aligned} C'_i \leftarrow \min \quad (\quad & D + C_i,\ \min_{i^{(1)}}(S_i + C_{i^{(1)}}), \min_{i^{(1)}}(I_i + S_{i^{(1)}} + \min_{i^{(2)}} C_{i^{(2)}}), \\ & \min_{i^{(1)}}(I_i + D + C_{i^{(1)}}), \min_{i^{(1)}}(I_i + I_{i^{(1)}} + \min_{i^{(2)}} C'_{i^{(2)}}) \ ) \end{aligned}$$

The term $\min_{i^{(1)}}(I_i + D + C_{i^{(1)}})$ can be removed because, by definition of C_i, $C_i \leq \min_{i^{(1)}} I_i + C_{i^{(1)}}$ (third component of Eq. (2) applied to the computation of C), and we have already $D + C_i$ in the minimization. We factor out all the minimizing operators and get

$$C'_i \leftarrow \min(D + C_i,\ \min_{i^{(1)}, i^{(2)}} \min(S_i + C_{i^{(1)}}, I_i + S_{i^{(1)}} + C_{i^{(2)}},\ I_i + I_{i^{(1)}} + C'_{i^{(2)}}))$$

By unfolding $C'_{i^{(2)}}$ and doing the same manipulations again we get

$$\begin{aligned} C'_i \ \leftarrow \ & \min(D + C_i, \\ & \min_{i^{(1)}, i^{(2)}, i^{(3)}} \min(S_i + C_{i^{(1)}},\ I_i + S_{i^{(1)}} + C_{i^{(2)}},\ I_i + I_{i^{(1)}} + S_{i^{(2)}} + C_{i^{(3)}}, \\ & \qquad I_i + I_{i^{(1)}} + I_{i^{(2)}} + C'_{i^{(3)}})) \end{aligned}$$

and we can continue until the latter term exceeds $k + C'_{i^{(r+1)}}$, which is not interesting anymore. The resulting recurrence does not depend anymore on C', and will become our working recurrence:

$$C'_i \leftarrow \min(D + C_i,\ \min_{r \geq 0}\ \min_{i^{(1)}, \ldots, i^{(r)}} \sum_{0 \leq u < r} I_{i^{(u)}} + S_{i^{(r)}} + C_{i^{(r+1)}}) \qquad (3)$$

4. A bit-parallel approximate search algorithm

We will represent the C_i vector in a bit mask. Each cell C_i will range in the interval $0 \ldots k+1$, so we will need $\ell = \lceil \log_2(k+2) \rceil$ bits to represent it. The reason is that, if a cell value is larger than $k+1$, we can assume that its value is $k+1$ and the outcome of the search will be the same [14]. For technical reasons that are made clear soon, we will need an extra bit per cell, which will always be zero. Since C_0 is always 0, it does not need to be represented. Hence we need $m(1+\ell)$ bits overall. The bit mask will represent the sequence of cells $C = 0[C_m]_\ell \, 0[C_{m-1}]_\ell \ldots 0[C_2]_\ell \, 0[C_1]_\ell$. We use as many computer words as needed to store C (a single cell will not be split among computer words).

From the parsing of the regular expression, we receive the tables B and $Follow$, where the sets are represented as bit masks of length $m+1$ (see previous work for details [11]). We will preprocess B so as to produce bit-parallel versions of I_i, D and S_i:

- We recall that $I_i = \omega(\varepsilon, R_i)$, and therefore it depends only on the pattern. Its bit-parallel version is $I = 0[I_m]_\ell \ldots 0[I_1]_\ell$.
- Recall also that $D(a) = \omega(a, \varepsilon)$, which depends on the last text character read, $a = T_j$, but not on the NFA state i. Hence its bit-parallel version is essentially $D[a] = (0[D(a)]_\ell)^m$.
- Finally, recall that $S_i = \omega(a, R_i)$, where $a = T_j$. Its bit-parallel version is $S[a] = 0[S_m(a)]_\ell \ldots 0[S_1(a)]_\ell$.

In all the masks above, any value larger than $k+1$ is converted to $k+1$. The computation of I, $D[\,]$ and $S[\,]$ from ω and B is shown in Fig. 2. Fig. 3 shows an example for the NFA of Fig. 1.

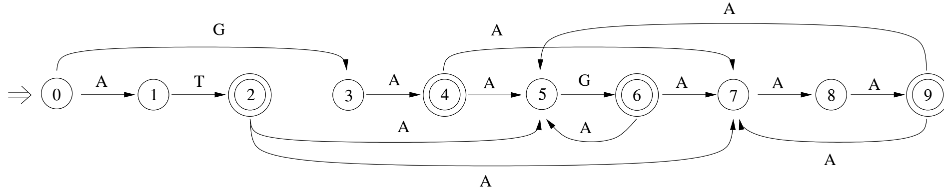
```
CalcWeights (ω, B, k, m, ℓ)
1.  I ← 0^{(1+ℓ)m}
2.  For c ∈ Σ Do
3.      D[c] ← (0[min(ω(c,ε), k+1)]_ℓ)^m
4.      S[c] ← 0^{(1+ℓ)m}
5.      For i ∈ 1...m Do
6.          If B[c] & 0^{m−i}10^{i−1} ≠ 0^m Then
7.              I ← I | 0^{(1+ℓ)(m−i)}0[min(ω(ε,c), k+1)]_ℓ 0^{(1+ℓ)(i−1)}
8.              For c′ ∈ Σ Do
9.                  S[c′] ← S[c′] | 0^{(1+ℓ)(m−i)}0[min(ω(c′,c), k+1)]_ℓ 0^{(1+ℓ)(i−1)}
```

Fig. 2: Computation of tables I, D and S from ω and B.

Our next tool is a table J (an extended version of the simpler table J of Section 2), which maps bit masks of length $m(1+\ell)$ into bit masks of length $m(1+\ell)$, as follows:

$$J[\,0[C_m]_\ell \, 0[C_{m-1}]_\ell \ldots 0[C_2]_\ell \, 0[C_1]_\ell\,] \quad = \quad 0[M_m]_\ell \, 0[M_{m-1}]_\ell \ldots 0[M_2]_\ell \, 0[M_1]_\ell$$

w(e,A) = 1, w(e,C) = 1, w(e,G) = 2, w(e,T) = 3
w(A,e) = 2, w(C,e) = 3, w(G,e) = 1, w(T,e) = 2
w(A,C) = 2, w(A,G) = 2, w(A,T) = 1



$I =$	1	3	2	1	1	2	1	1	1
$D['A'] =$	2	2	2	2	2	2	2	2	2
$S['A'] =$	0	1	2	0	0	2	0	0	0

Fig. 3: Examples of I, D, and S masks for the NFA of Fig. 1, considering some arbitrary weights $\omega()$ given on top. Note that there is one entry in the masks for each NFA state except the initial one.

where

$$M_i \;=\; \min_{i' \in Follow^{-1}(i)} C_{i'}$$

That is, for each search state C, J indicates how the values in C propagate through NFA edges. If several states i' propagate to a single state i, we choose the minimum value. We account for the zeros propagated from the unrepresented initial state 0.

Let us now consider the recurrence of Eq. (3). Assume that C is our current search state. The first part of the minimum $(D + C_i)$ is easily obtained in bit-parallel, as $E \leftarrow C + (0[D]_\ell)^m$. If D turns out to be larger than $k + 1$ we set $D = k + 1$. The result of the sum can give us values as large as $2(k + 1)$ in the counters. Our extra bit per cell can hold the overflow, but we have to replace the values of the overflown counters by $k + 1$ in order to continue our process[2]. We detect the overflown counters by precomputing $W \leftarrow (10^\ell)^m$ and doing $Z \leftarrow E \ \& \ W$. Then, $Z \leftarrow Z - (Z >> \ell)$ will be a sequence of all-0 or all-1 cells, where the all-1 ones correspond to the overflown counters. These are restored to $k + 1$ by doing $E \leftarrow (E \ \& \ \sim Z) \mid (0[k + 1]_\ell)^m \ \& \ Z)$.

Let us call H the second, complex part of the main minimum of Eq. (3). Once we obtain H, we have to obtain $C' \leftarrow Min(E, H)$, where Min takes the element-wise minimum over two sequences of values, in bit-parallel.

Bit-parallel minimum can be obtained with a technique similar to the one used above to restore overflown values. Say that we have to compute $Min(X, Y)$, where X and Y contain several counters (nonnegative integers) properly aligned. We need the extra highest bit per counter, which is always zero. We use mask W and perform the operation $Z \leftarrow ((X \mid W) - Y) \ \& \ W$. The result is that, in Z, each highest bit

[2] A simple choice is to use $2 + \ell$ bits per counter, since the upcoming minimizations will take care of the overflows, but we show that it can be done anyway with $1 + \ell$ bits.

is set if and only if the counter of X is larger than that of Y. We now compute $Z \leftarrow Z - (Z >> \ell)$, so that the counters where X is larger than Y have all their bits set in Z, and the others have all the bits in zero. We now choose the minima as $Min(X, Y) \leftarrow (Y \,\&\, Z) \mid (X \,\&\, {\sim} Z)$. Similarly, $Max(X, Y) \leftarrow (X \,\&\, Z) \mid (Y \,\&\, {\sim} Z)$. Fig. 4 summarizes our min/max procedures.

Min $(X,\ Y)$
1. $W \leftarrow (10^{\ell})^m$
2. $Z \leftarrow ((X \mid W) - Y) \,\&\, W$
3. $Z \leftarrow Z - (Z >> \ell)$
4. **Return** $(Y \,\&\, Z) \mid (X \,\&\, {\sim} Z)$

Max $(X,\ Y)$
1. $W \leftarrow (10^{\ell})^m$
2. $Z \leftarrow ((X \mid W) - Y) \,\&\, W$
3. $Z \leftarrow Z - (Z >> \ell)$
4. **Return** $(X \,\&\, Z) \mid (Y \,\&\, {\sim} Z)$

Fig. 4: Bit-parallel minima and maxima.

Having overcomed these initial obstacles, we focus now on the most complex part: the computation of H. Let us consider $A = J[C] + S[T_j]$, and assume that we have again solved overflow problems in A[3]. The i-th element of A is, by definition of J, $A_i = S_i + \min_{i' \in Follow^{-1}(i)} C_{i'}$. Now, consider $J[A] + I$. Its i-th value is

$$\begin{aligned} I_i + \min_{i' \in Follow^{-1}(i)} A_{i'} &= I_i + \min_{i' \in Follow^{-1}(i)} \left(S_{i'} + \min_{i'' \in Follow^{-1}(i')} C_{i''}\right) \\ &= \min_{i^{(1)}, i^{(2)}} \left(I_i + S_{i^{(1)}} + C_{i^{(2)}}\right) \end{aligned}$$

If we compute $J[J[A] + I] + I$, we have that its i-th value is $\min_{i^{(1)}, i^{(2)}, i^{(3)}} (I_i + I_{i^{(1)}} + S_{i^{(2)}} + C_{i^{(3)}})$, and so on. Let us define $f(A) = J[A] + I$ and $f^{(r)}(A)$ as the result of taking r times f over A. Then, we have that

$$f^{(r)}(A) \;=\; \min_{i^{(1)} \ldots i^{(r)}} \left(\sum_{0 \le u < r} I_{i^{(u)}} \;+ S_{i^{(r)}} + C_{i^{(r+1)}} \right)$$

and hence the H we look for is

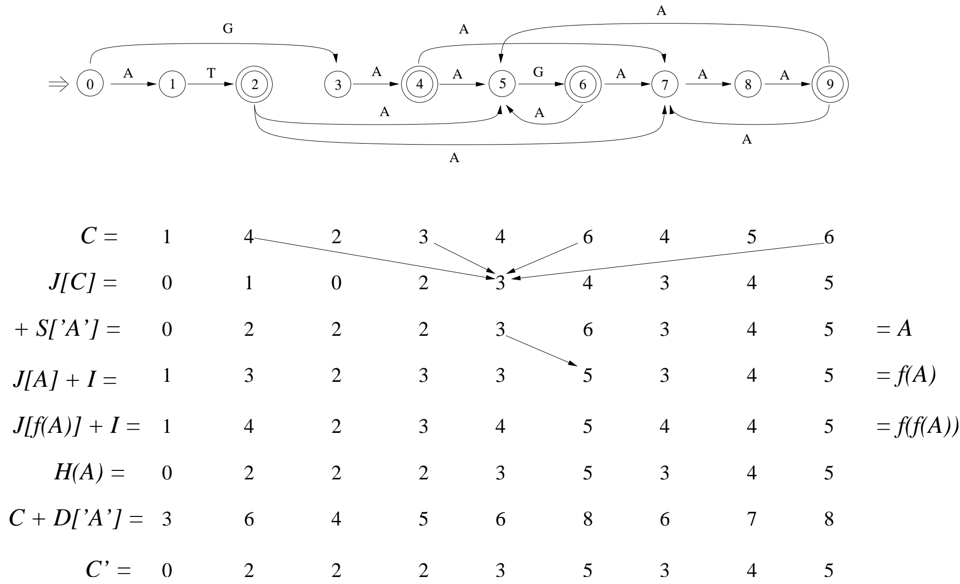
$$H[A] \;=\; Min\left(A, f(A), f^{(2)}(A), f^{(3)}(A), \ldots\right) \qquad (4)$$

Fig. 5 continues our example, showing how a new C' vector is computed from a current C vector.

To conclude, we have to report every text position where it holds $C_i \le k$ for a final state i. The parsing yields an $(m + 1)$-bits long mask of final states, $Last$. We will precompute a mask $F = 0[F_m]_\ell\ 0[F_{m-1}]_\ell \ldots 0[F_2]_\ell\ 0[F_1]_\ell$, so that $F_i = 1$ if i is final and $F_i = 0$ otherwise[4]. Hence, we have a match if and only if $C \,\&\, (F \times (2^{\ell} - 1)) \neq F \times (k + 1)$. Note that $F \times x$ is a bit mask of m counters X_i such that $X_i = x$ if $F_i = 1$ and $X_i = 0$ otherwise.

[3] The extra work for this can be avoided either by, as before, using counters of $2 + \ell$ bits, or by precomputing all the allocated cells of H, as it will be clear soon.

[4] We assume that the initial state is not final, as otherwise the problem is trivial.



C =	1	4	2	3	4	6	4	5	6	
J[C] =	0	1	0	2	3	4	3	4	5	
+ S['A'] =	0	2	2	2	3	6	3	4	5	= A
J[A] + I =	1	3	2	3	3	5	3	4	5	= f(A)
J[f(A)] + I =	1	4	2	3	4	5	4	4	5	= f(f(A))
H(A) =	0	2	2	2	3	5	3	4	5	
C + D['A'] =	3	6	4	5	6	8	6	7	8	
C' =	0	2	2	2	3	5	3	4	5	

Fig. 5: Example of Fig. 3 continued. Assume the current C values are those of the top row. The automaton arrows permit carrying them onto other states by $J[C]$ (those reaching state 5 are shown with arrows). Then, A is computed by adding $J[C] + S[\texttt{'A'}]$, being `'A'` the current text character. Then we repeatedly compute $f(A) = J[A] + I$, $f(f(A))$, as far as necessary (in this example we stopped at $f(f(A))$ as it is uniformly larger than $f(A)$). The arrow in the third row shows how the value at state 5 permits reducing the value at state 6 when moving from A to $f(A)$. Then $H[A]$ takes the columnwise minimum of A, $f(A)$, $f(f(A))$, etc. Actually $H[A]$ is precomputed for every possible A so this process takes constant time. Finally, the new C value is the columnwise minimum between $H[A]$ and $C + D[\texttt{'A'}]$.

Fig. 6 gives the search code. To initialize C we take H over an initial state where all the counters are $k + 1$. **Glushkov_Parse** is in charge of parsing the regular expression and delivering tables B, $Follow$ and bit mask $Last$. We then precompute all the tables using **Preprocess**.

Search $(T_{1...n},\ R,\ k,\ \omega)$
1. $(B, Follow, Last, m) \leftarrow$ **Glushkov_Parse**(R)
2. $(D, S, J, H, F, \ell) \leftarrow$ **Preprocess**$(B, Follow, Last, m, k, \omega)$
3. $C \leftarrow H[(0[k+1]_\ell)^m]$
4. **For** $j \in 1 \ldots n$ **Do**
5. $\quad A \leftarrow J[C] + S[T_j]$
6. $\quad C \leftarrow Min(C + D[T_j], H[A])$
7. $\quad$ **If** $C \mathbin{\&} (F \times (2^\ell - 1)) \neq F \times (k+1)$ **Then** Report text position j

Fig. 6: Our search algorithm. We disregard the process of restoring overflows after additions.

The preprocessing is given in Fig. 7. Although it looks complicated, it is conceptually simple. There are first several technical functions. **Expand** takes a sequence

of $m + 1$ bits, ignores the first, and introduces ℓ zero bits between each pair of bits, so as to align them to our representation. **Next** computes the successor of a number made up of m digits in the range $0 \ldots k+1$. **Fill** fills all the $(k+2)^m$ possible entries of a table U, where each entry is formed by m numbers in the range $0 \ldots k+1$. The idea is that, given table $u[i, v]$ telling which positions get which values if position i has value v, U tells, given an m-tuple of numbers, the minimum value obtained by each position from any of the values in its argument. **Fill** starts with the entry where all the numbers are $k + 1$ and then computes all the possible values for the i-th number, with the invariant that all the possible values of numbers at smaller positions are already computed (values at positions larger than i being $k + 1$). G is a bit mask that traverses all these possible values, and $curr$ is the current value of the i-th number in G. $U[G]$ is computed as the minimum between what we already have with value $k + 1$ for the i-th number and $u[i, curr]$.

The purpose of **Fill** is to compute tables J and H, which is done in **Preprocess**. For J, we fill a table e of m-tuples such that $e[i, v]$ has value v in states reachable from state i via $Follow$, and $k+1$ elsewhere. Building J from e via **Fill** is precisely what we need: For each m-tuple of state values, J gives the minimum value that reaches each state via $Follow$. Similarly, for H we fill a table $h[i, v]$ corresponding to the value of $H[A]$ when the i-th value of A is v and the rest is $k + 1$. The fixed point of Eq. (4) is computed in lines 11–13. Then, we build all the combinations of A using **Fill** as before. Note that we do not return I because it is embedded in the computation of H.

5. Analysis and space-time tradeoffs

The search time of our algorithm is clearly $O(n)$. The preprocessing time includes $O(|\Sigma|^2 m)$ for **CalcWeights** and $O(k^2 m^2)$ to compute h (since for each of the km cells we iterate as long as we reduce some counter, which can happen only $m(k+1)$ times). However, the dominant preprocessing complexity is the $O((k + 2)^m)$ space and time needed to fill J and H. If this turns out to be excessive, we can horizontally split tables J and H.

The splitting is based on the following property. Let J be a table built over m counters. Let $C = C^1 : C^2$ be a splitting of mask C into two submasks, a left and a right submask. If we define J_1 and J_2 so that they propagate counters only from the first and second half of mask C, respectively, then $J[C^1 : C^2] = Min(J_1[C^1], J_2[C^2])$ because of the definition of J. (Note that J_1 and J_2 can propagate values to states of any half.) The same is valid for H: we can split the argument A into two halves A^1 and A^2, and preprocess the propagations of values from the first and second half in H_1 and H_2, so that $H[A^1 : A^2] = Min(H_1[A^1], H_2[A^2])$. Note, in particular, that J and H have been built by adding the effects of new states one by one, precisely because they can be decomposed in this way.

In general, we can split J and H into t tables $J_1 \ldots J_t$ and $H_1 \ldots H_t$, such that J_i and H_i address the counters roughly from $(i - 1)m/t$ to $im/t - 1$, that is, m/t counters. Each such table has $(k + 2)^{m/t}$ entries, for a total space requirement of $O(t(k + 2)^{m/t})$. The cost is that, in order to perform each transition, we need to pay

Expand$(X,\ m,\ \ell)$
1. $EX \leftarrow 0^{(1+\ell)m}$
2. **For** $i \in 1 \ldots m$ **Do**
3. **If** $X\ \&\ 0^{m-i}10^i \neq 0^{m+1}$ **Then** $EX \leftarrow EX \mid 0^{(m-i)(1+\ell)}0^\ell 10^{(i-1)(1+\ell)}$
4. **Return** EX

Next$(G,\ \ell,\ m,\ lim)$
1. **For** $i \in 1 \ldots m$ **Do**
2. $val \leftarrow (G >> (1+\ell)(i-1))\ \&\ 0^{(1+\ell)(m-1)}01^\ell$
3. **If** $val < lim$ **Then**
4. $G \leftarrow G + 0^{(1+\ell)(m-i-1)}0^\ell 10^{(1+\ell)(i-1)}$
5. **Return** G
6. $G \leftarrow G\ \&\ 1^{(1+\ell)(m-i-1)}0^{1+\ell}1^{(1+\ell)(i-1)}$

Fill$(U,\ u,\ k,\ \ell,\ m)$
1. $U[(0[k+1]_\ell)^m] \leftarrow u[0,0]$
2. **For** $i \in 1 \ldots m$ **Do**
3. $G \leftarrow (0[k+1]_\ell)^{m-i}0^{(1+\ell)i}$
4. **For** $j \in 0 \ldots (k+2)^i - 1$ **Do**
5. $curr \leftarrow (G >> (1+\ell)(i-1))\ \&\ 0^{(1+\ell)(m-1)}01^\ell$
6. $U[G] \leftarrow Min(U[G + 0^{(1+\ell)(m-i)}0[k+1-curr]_\ell 0^{(1+\ell)(i-1)}],\ u[i, curr])$
7. $G \leftarrow$ **Next**$(G, \ell, m, k+1)$
8. **Return** U

Preprocess $(B,\ Follow,\ Last,\ m,\ k,\ \omega)$
1. $\ell \leftarrow \lceil \log_2(k+2) \rceil$
2. $(I, D, S) \leftarrow$ **CalcWeights** $(\omega,\ B,\ k,\ m,\ \ell)$
3. $F \leftarrow$ **Expand**$(Last, m, \ell)$
   // Computation of J
4. **For** $i \in 0 \ldots m$ **Do**
5. $EFollow[i] \leftarrow$ **Expand**$(Follow[i], m, \ell)$
6. **For** $v \in 0 \ldots k+1$ **Do**
7. $e[i,v] \leftarrow (0[k+1]_\ell)^m - (EFollow[i] \times (k+1-v))$
8. **Fill** $(J,\ e,\ k,\ \ell,\ m)$
   // Computation of H
9. **For** $i \in 1 \ldots m$ **Do**
10. **For** $v \in 0 \ldots k+1$ **Do**
11. $h[i,v] \leftarrow (0[k+1]_\ell)^{m-i}0[v]_\ell(0[k+1]_\ell)^{i-1}$
12. **While** $h[i,v] \neq Min(h[i,v], J[h[i,v]] + I)$ **Do**
13. $h[i,v] \leftarrow Min(h[i,v], J[h[i,v]] + I)$
14. $h[0,0] \leftarrow (0[k+1]_\ell)^m$
15. **Fill** $(H,\ h,\ k,\ \ell,\ m)$
16. **Return** (D, S, J, H, F, ℓ)

Fig. 7: Our preprocessing.

for t table accesses so as to compute

$$J[C^1 : C^2 : \ldots C^t] = Min(J_1[C^1],\ J_2[C^2],\ \ldots J_t[C^t])$$
$$H[A^1 : A^2 : \ldots A^t] = Min(H_1[A^1],\ H_2[A^2],\ \ldots H_t[A^t])$$

which makes the search time $O(tn)$ in terms of table accesses. If we have $O(s)$ space, then we solve for $s = t(k+2)^{m/t}$, to obtain a search time of $O(tn) = O(mn/\log_k s)$.

Let us consider how good can be our complexity with infinite space. Our overall cost, adding preprocessing and searching, is $O(t(k+2)^{m/t} + tn)$. The optimum is reached for $t = m/\log_{k+2} n$, where the overall complexity becomes $O(mn/\log_k n)$. The amount of space we need to achieve this best possible complexity is $s = \Omega(mn/\log_k n)$.

At this point, a note on the underlying computation model is relevant. We are assuming that we can handle bit masks of $O(m \log k)$ bits in $O(1)$ time. If we stick to the RAM model of computation, where the computer can only handle words of $w = \Theta(\log n)$ bits in constant time, we should multiply the cost of our algorithm by $O(m \log(k)/w)$ to account for the need to use many computer words. Hence, our complexity in the RAM model is actually $O(m^2 n \log^2(k)/(w \log n))$. On the other hand, the previous $O(kmn/\log n)$ bit-parallel technique [16] needs only $O(m)$ bits, and therefore its RAM cost is $O(km^2 n/(w \log n))$. Still our complexity is superior by a factor of $O(k/\log^2 k)$.

6. Experimental results

In this section we evaluate our algorithm experimentally and compare it against previous work. The algorithms we have compared are:

DP: The classical dynamic programming solution [7]. The code was originally from G. Myers and we modified it to work with integer values and fixed threshold k. This algorithm is by far the slowest in our experiments, but it handles the more general problem of real-valued arbitrary weighted scoring schemes and affine gap costs, as opposed to our algorithm, which handles just integral-cost differences. However, it is a good baseline to compare how the more specific algorithms improve upon it. Since the algorithm performance is insensitive to costs and thresholds, we run it only for $k = 1$ and unit-cost differences.

RUS: The four-russians approach [17]. Unfortunately, the code used for this paper seems to be lost [18], so we have used an algorithm implemented using the same technique, but which handles exact search only, at $O(mn/\log s)$ time and $O(s)$ space [6]. The code is from G. Myers. The exact searching algorithm gives us a lower bound on which would have been the performance of the version that searches permitting k differences.

GREP: The technique of copying $k+1$ exact searching automata and updating them one by one [16, 15, 9]. The code is a highly optimized modification of the "forward scanning" of *nrgrep* where we removed the transposition

error, manually coded the cases $1 \leq k \leq 8$ and up to 2 computer words, and reduced reporting to just counting all the matches. The performance is at least as good as that of *agrep* because the DFA is smaller [11]. This algorithm can handle only unit-cost differences, and seems not extensible to arbitrary weights. Hence, it is not an alternative to our algorithm, but it serves to show how simpler is the problem of unit-cost differences.

OURS: Our algorithm handling arbitrary integer-cost differences, where we have manually coded the cases of up to 2 computer words and tables split into up to 6 subtables. The code is a plain implementation of Fig. 7 and Fig. 6. Since our algorithm performance is insensitive to the weight function ω, we ran it with unit-costs differences for simplicity (our code, however, does not take advantage of this). For each data point, we choose the best among the alternatives of using 1 to 6 tables. The best is generally the one with fewest tables so that the machine can hold them comfortably. In our experiments we never used more than about 5 megabytes of memory.

All the algorithms are carefully coded, use similar buffer schemes, and just count the number of occurrences. We used `gcc` with all the code optimizations. Our machine is a 64-bit Digital Alphaserver 600 5/266 with 266 MHz 21164 Alpha-processors and 768 Mb of RAM, running Digital Unix 4.0B. The machine was not performing other heavy tasks while the experiments ran. We measure user times (CPU times).

We searched 10 megabytes of English text extracted from the Wall Street Journal 1987 [5]. Each data point corresponds to an average over 100 different search patterns (the same for all the algorithms).

The choice of patterns is always problematic when dealing with regular expressions, since there is no clear concept of what a random regular expression is and, as far as we know, there is no public repository of regular expressions available, except for a dozen of trivial examples. We have chosen to generate random regular expressions as follows:

(1) We choose m and pick a random text substring of length m.

(2) We choose an *operator density* $0 \leq \alpha \leq 1$.

(3) We apply a recursive procedure to convert a string of length ℓ into a regular expression:

   (a) An empty string is converted into an empty regular expression. In the rest, we assume a nonempty string.
   (b) With probability $1 - \alpha$ we choose that the expression will be the concatenation of two subexpressions: a left part of ℓ' characters and a right part of $\ell - \ell'$ characters, where ℓ' is chosen uniformly in the range $1 \leq \ell' \leq \ell - 1$. We recursively convert both subparts into regular expressions e_1 and e_2. The resulting expression is $e_1 \cdot e_2$. If $\ell = 1$ we simply write down the string character.
   (c) Otherwise, if the parent in the recursion has just generated a Kleene closure operator "$*$", we choose to add a union operator "|", if not, we choose with the same probability among a Kleene closure and a union.

(d) If we chose that the expression will have a union operator, we choose a left part of ℓ' characters and a right part of $\ell - \ell'$ characters, where ℓ' is chosen uniformly in the range $0 \leq \ell' \leq \ell$. We recursively convert both subparts into regular expressions e_1 and e_2. The resulting expression is $e_1|e_2$.

(e) If we chose to add a Kleene closure operator "$*$" at the end of the string, we recursively generate a regular expression e_1 for the string. The resulting expression is e_1*.

(f) To avoid problems with the different softwares and with operators symbols in the text, any non-alphanumerical character is converted to underscore.

The above procedure is just one of the many possible alternatives to generate random regular expressions one could argue for, but it has a few advantages. First, it permits determining the length m (number of characters of Σ) in advance. Second, it takes the characters from the text, respecting its distribution. Third, it permits us to choose expressions with more or less operators (other than concatenation) by varying α. We will show experiments with $\alpha = 0.05$, $\alpha = 0.10$ and $\alpha = 0.20$. Examples obtained from our tests, with $m = 10$, are `"I(n|(s)*urance )"`, `"(co|ntr(a)*(c(t)*|or))"`, and `"(((_)*f|ro)m_l|((a)*|st))"`, respectively.

We first show how the search cost increases with m, for the minimum threshold $k = 1$. Fig. 8 shows the results. As it can be seen, DP is by far the slowest algorithm (albeit, as explained, can handle real-valued weights), and its cost grows noticeably as the density α of the regular expressions increases. RUS turns out to be clearly slower than our algorithm, usually twice as slow, and its cost also grows slightly with α. GREP, on the other hand, is much faster than our algorithm (up to three times faster) but, as explained, it handles unit-cost differences only (all $\omega() = 1$). Neither GREP nor OURS are affected by the density α.

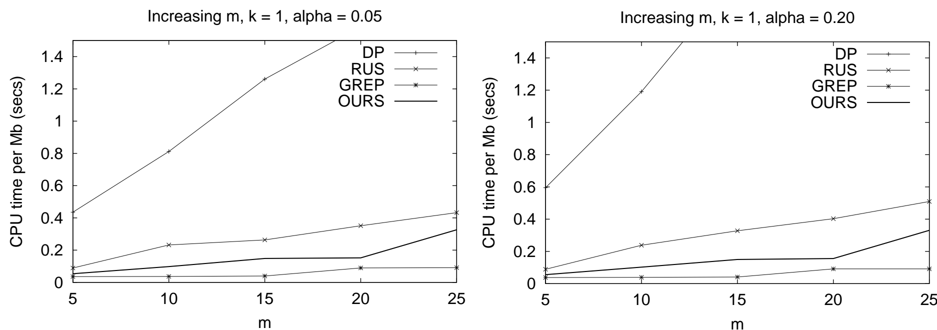


Fig. 8: Comparison for increasing pattern length m and fixed $k = 1$. We show two different pattern densities, $\alpha = 0.05$ on the left and $\alpha = 0.20$ on the right.

From now on we consider an intermediate density $\alpha = 0.10$. Fig. 9 shows the behavior of the algorithms for fixed m and increasing k. We have included a new algorithm called "RUS (extrap)", which is just an extrapolation of the cost of RUS for $k > 0$. Since their complexity depends on k as $\log(k+2)$, we have multiplied the

complexity of the exact search by $\ln(k+2)/\ln(2)$. We believe that this estimation is optimistic because the code to handle differences is indeed more complex, but anyway this has to be taken for what it is: just our extrapolation.

Both for length $m = 15$ and $m = 20$, OURS becomes slower than RUS only for $k > 5$, even when RUS works just for $k = 0$. Our extrapolation of RUS is well above OURS. On the other hand, the $O(k)$ complexity of GREP is clear as compared to our $O(\log k)$ complexity. However, it seems unlikely that GREP becomes slower than our algorithm before our algorithm becomes slower than DP (there should be another jump for $m = 20$ and $k = 6$ because three computer words would be needed to hold the masks). We remark again that GREP only works for unit-cost differences.

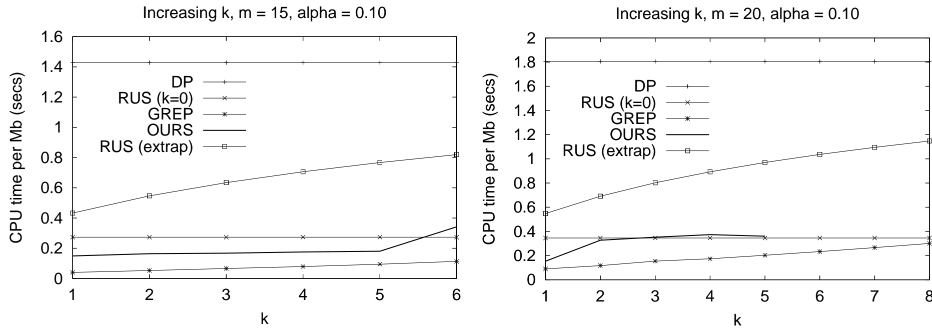


Fig. 9: Comparison for fixed pattern length $m = 15$ (left) and $m = 20$ (right) and increasing k. We show density $\alpha = 0.10$.

Finally, we show in Fig. 10 the effect of increasing m where k is a fixed fraction of m. The results bring no new surprises. Our algorithm is faster than RUS ($k = 0$) up to $m = 20$, and consistently faster than our extrapolation of RUS. On the other hand, DP is much slower and GREP is significantly faster than our algorithm.

Note that OURS shows jumps when we start using more computer words for the simulation, and the lines stop when we need more than 6 computer words since, as we explained, we coded the algorithm with up to that number of words.

As a final note, we remark that the best table partitioning choice never made us use more than 5 megabytes of memory. This means that the best choices have negligible preprocessing. This is expected, as a large preprocessing time means that large tables are precomputed and hence the locality of reference to access them is low, which worsens the search time in addition to the preprocessing time.

7. Conclusions

We have presented a bit-parallel algorithm to solve the problem of approximate searching for regular expressions with arbitrary integer weights. The algorithm is simple and has the same complexity of the best previous solution, $O(mn/\log_k s)$ time with $O(s)$ space. In practice, however, we show that our algorithm clearly outperforms previous solutions.

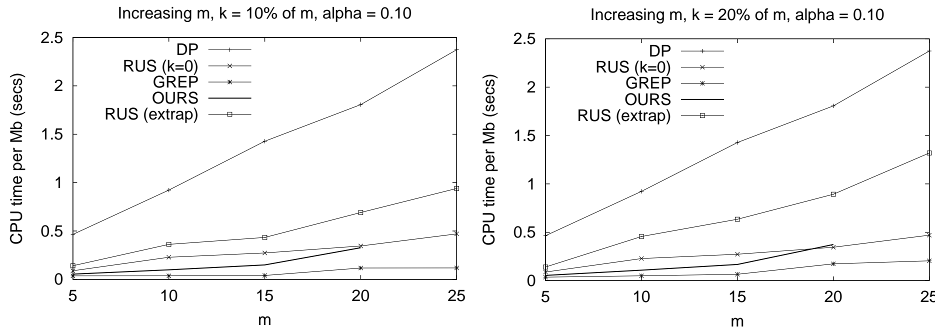


Fig. 10: Comparison for increasing m and fixed fraction $k/m = 10\%$ (left) and $k/m = 20\%$ (right). We show density $\alpha = 0.10$.

In our way, we have found a new recurrence for the problem, where the current values depend only on previous values. This is usually the main complication when combining the circular dependence of the classical recurrence (current values depending on current values) with the possible cycles of the automaton. We believe that our solution can be useful in other scenarios.

It is easy to extend the solution to the case where the regular expression contains classes of characters, that is, positions that match several possible characters. We simply have to take the minimum over the characters of the class when precomputing tables I and S.

It is interesting that our solution is also relevant for approximate searching of simple strings using arbitrary weights. Current bit-parallel solutions handle only the case of unitary costs, or at most a fixed integer cost per operation [8]. Recently, Bergeron [2] extended this result showing that, when all the insertions and deletion costs are fixed at c and substitution costs are arbitrary integers, a simple string can be searched for in $O(mnc\log(c)/w)$ time. Our approach permits arbitrary insertion, deletion and substitution costs and has better complexity given enough memory space, $s > c\log c/\log^2 k$.

It would be interesting to study how one can take advantage of the simpler structure of a string pattern in order to simplify our algorithm in this case. For example, table $J[C]$ is simply $J[C] = C << (1 + \ell)$. However, we have not found a relevant simplification for table H.

On the other hand, we have also shown that much better solutions exist for the case of searching for regular expressions with unit-cost differences. Although these solutions, also based on bit-parallelism, have worse complexity $O(kmn/\log s)$, they are significantly faster in practice. Despite that unit-costs is an oversimplification for many real-world applications, a clear goal for future work is to develop an algorithm whose efficiency approaches that of the best algorithms for the unit-cost case. A way to simplify the computation of bit-parallel minimum would be important in this sense.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
[2] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–65, 2002.
[3] G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.
[4] V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
[5] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.
[6] E. Myers. A four-russian algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):430–448, 1992.
[7] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51:7–37, 1989.
[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
[9] G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.
[10] G. Navarro. Approximate regular expression searching with arbitrary integer weights. In *Proc. 14th International Symposium on Algorithms and Computation (ISAAC'03)*, LNCS 2906, pages 230–239, 2003.
[11] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proc. 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 1–12, 2001.
[12] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.
[13] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
[14] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
[15] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of the USENIX Technical Conference*, pages 153–162, 1992.
[16] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.
[17] S. Wu, U. Manber, and E. W. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346–360, 1995.
[18] S. Wu, U. Manber, and G. Myers. Personal communication with each. 2002.