Approximate Regular Expression Searching with Arbitrary Integer Weights[⋆]

Gonzalo Navarro

Dept. of Computer Science, Univ. of 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 best previous complexities, but it is much 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.

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* (RE) is a generalized pattern composed of (i) basic strings, (ii) union, concatenation and Kleene closure of other REs [1]. We call m the length of our RE, 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 RE [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 [10] is used. Thompson's construction produces up to $2m$ states, but it has interesting properties, such as ensuring a linear number of edges and constant in/out-degree.

An alternative NFA construction is Glushkov's [3, 2]. Although it does not provide the same regularities of Thompson's, this construction has other useful 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 [10] 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 RE is large.

⋆ Partially supported by Fondecyt grant 1-020831.

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 automaton states in one shot, it has been shown that $O(mn/\log s)$ search time is possible using $O(s)$ space [4]. 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 [12], 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 [8] 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: An integer threshold k is given, so that we have to report the text substrings that can match the RE after performing several character insertions, deletions and substitutions, whose total *cost* or *weight* does not exceed k. Each operation may have a different weight 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 [5]. 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 [13], 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 [12] or $O(2^{m/t})$ space [6]. This yields $O(kmn/\log s)$ time using $O(s)$ space, inferior to the achievement of the Four Russians technique. Despite this worse complexity, bit-parallel solutions are by far the fastest for moderate sized REs. Yet, they are restricted to 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 RE 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 $O((k+2)^{m/t})$ and reach $O(tn)$ search time.

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.

An extended version of this paper, with all the details, can be found in [7].

2 A Bit-Parallel Exact Search Algorithm

We describe in this section the exact bit-parallel solution we build on [8]. 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.

Glushkov's NFA construction algorithm can be found in [3, 2]. We just remark some of its properties. Given a RE 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 RE, 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 RE 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.

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 very 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

[1] This is computed from the RE, since the NFA does not yet exist. Also, for simplicity, we assume that $Follow(0) = First$, the states reachable from the initial state.

position j where $D \cap \mathit{Last} \neq \emptyset$. (In fact, state 0 needs not be represented, since it is always active when searching.)

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

We start with an exact formulation for our problem. Let R be a RE generating language $L(R) \subseteq \Sigma^*$. 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 $P \in L(R)$, $ed(T_{j'\ldots j}, P) \leq k$. The edit distance, $ed(A, B)$, 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, where $\omega(a, b)$ is the cost to substitute character a by character b in the text, $\omega(a, \varepsilon)$ is the cost to delete text character a, and $\omega(\varepsilon, b)$ is the cost to insert character b in the text. Function ω satisfies $\omega(a, a) = 0$, nonnegativity, and triangle inequality.

The classical dynamic programming solution for approximate string matching [9], 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 $C_i = i$ 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 RE R built using Glushkov's algorithm, with positions 1 to m, this generalizes as follows. 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:

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

where $S_i(a) = \omega(a, R_i)$, $D(a) = \omega(a, \varepsilon)$, $I_i = \omega(\varepsilon, R_i)$, and R_i is the only character such that $B(R_i) = \{i\}$: 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 $L_0 = \{\varepsilon\}$.

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 Recurrence (1) (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 [5], but these are not easy to apply in a bit-parallel context.

We present a new solution now. 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 Recurrence (1):

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

where after a few manipulations we obtain

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

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 Recurrence (1) 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

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

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:

$$C'_i \leftarrow \min(D + C_i,\ \min_{r \geq 0}\ \min_{i^{(1)} \ldots i^{(r)}} \sum_{0 \leq u < r} I_{i^{(u)}}\ + S_{i^{(r)}} + C_{i^{(r+1)}}) \qquad (2)$$

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 [11]. For technical reasons that are made clear soon, we will need an extra bit per cell, which

```
CalcWeights ($\omega$, $B$, $k$, $m$, $\ell$)
1.  $I \leftarrow 0^{(1+\ell)m}$
2.  For $c \in \Sigma$ Do
3.      $D[c] \leftarrow (0[\min(\omega(c,\varepsilon), k+1)]_\ell)^m$
4.      $S[c] \leftarrow 0^{(1+\ell)m}$
5.      For $i \in 1 \ldots m$ Do
6.          If $B[c]$ & $0^{m-i}10^{i-1} \neq 0^m$ Then
7.              $I \leftarrow I \mid 0^{(1+\ell)(m-i)}0[\min(\omega(\varepsilon,c), k+1)]_\ell 0^{(1+\ell)(i-1)}$
8.              For $c' \in \Sigma$ Do
9.                  $S[c'] \leftarrow S[c'] \mid 0^{(1+\ell)(m-i)}0[\min(\omega(c',c), k+1)]_\ell 0^{(1+\ell)(i-1)}$
```

Fig. 1. Computation of tables I, D and S from ω and B.

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 RE, we receive the tables B and $Follow$, where the sets are represented as bit masks of length $m+1$ (see previous work for details [8]). We will preprocess B so as to produce bit-parallel versions of I_i, D and S_i. These will be called I, $D[\sigma]$ and $S[\sigma]$, respectively. The computation of these values from ω and B is shown in Figure 1.

We use a table J (an extended version of previous simpler table J), which maps bit masks of length $m(1+\ell)$ into bit masks of length $m(1+\ell)$, as follows:

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

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 Recurrence (2). 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. 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 Recurrence (2). 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 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 $Min(X, Y) \leftarrow (Y \ \&\ Z) \mid (X \ \&\ \sim Z)$.

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^2[2]. The i-th element of A is, by definintion of J, $A_i = S_i + \min_{i' \in Follow^{-1}(i)} C_{i'}$. Now, consider $J[A] + I$. Its i-th value is

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

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

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

and hence the H we look for is

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

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[3]. 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.

Figure 2 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 RE and delivering tables B, *Follow* and bit mask *Last*. We then precompute all the tables using **Preprocess**.

The preprocessing is given in Figure 3. Although it looks complicated, it is conceptually simple. Function **Expand** takes a sequence of $m+1$ bits, ignores

[2] The extra work for this can be avoided by precomputing all the allocated cells of H, as it will be clear soon.

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

```
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.       $A \leftarrow J[C] + S[T_j]$
6.       $C \leftarrow Min(C + D[T_j], H[A])$
7.       If $C$ & $(F \times (2^\ell - 1)) \neq F \times (k+1)$ Then Report text position $j$
```

Fig. 2. Search algorithm. We disregard the restoring of overflows after additions.

the first, and introduces ℓ zero bits between each pair of bits, so as to align them to our representation. J is computed by ranging over all the $(k+2)^m$ possible search states, starting with a state where all the counters are $k+1$ and then computing all the possible values for state i, with the invariant that all the possible values of states $< i$ (with states larger than i having value $k+1$) are already computed. G is a bit mask that traverses all these possible values, and *curr* is the current value of state i in G. $J[G]$ is computed as the minimum between what we already have with value $k+1$ for state i and the *curr* value for the states in $Follow[i]$. **Next** computes the next value for G. The processing for H is very similar, except that we first compute $h[i, v]$ as the desired value of $H[A]$ when the i-th value of A is v and the rest is $k+1$. Then, we build all the combinations of A using h with the same technique 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.

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 $H[A^1 : A^2] = Min(H_1[A^1], H_2[A^2])$. 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 $im/k - 1$, that is, m/t counters. Each such table has $(k+2)^{m/t}$ entries, for a total space requirement of $O(t(k+2)^{m/t})$.

```
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.       $G \leftarrow G$ & $1^{(1+\ell)(m-i-1)}0^{1+\ell}1^{(1+\ell)(i-1)}$

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 $EFollow[i] \leftarrow$ Expand($Follow[i], m, \ell$)
 5.   $J[(0[k+1]_\ell)^m] \leftarrow (0[k+1]_\ell)^m - (EFollow[0] \times (k+1))$
 6.   For $i \in 1 \ldots m$ Do
 7.       $G \leftarrow (0[k+1]_\ell)^{m-i}0^{(1+\ell)i}$
 8.       For $j \in 0 \ldots (k+2)^i - 1$ Do
 9.           $curr \leftarrow (G >> (1+\ell)(i-1))$ & $0^{(1+\ell)(m-1)}01^\ell$
10.           $J[G] \leftarrow Min(J[G + 0^{(1+\ell)(m-i)}0[k+1-curr]_\ell 0^{(1+\ell)(i-1)}],$
                          $(0[k+1]_\ell)^m - (EFollow[i] \times (k+1-curr)))$
11.       $G \leftarrow$ Next($G, \ell, m, k+1$)
              // Computation of $H$
12.   For $i \in 1 \ldots m$ Do
13.       For $v \in 0 \ldots k+1$ Do
14.           $h[i,v] \leftarrow (0[k+1]_\ell)^{m-i}0[v]_\ell(0[k+1]_\ell)^{i-1}$
15.           While $h[i,v] \neq Min(h[i,v], J[h[i,v]] + I)$ Do
16.               $h[i,v] \leftarrow Min(h[i,v], J[h[i,v]] + I)$
17.   $H[(0[k+1]_\ell)^m] \leftarrow (0[k+1]_\ell)^m$
18.   For $i \in 1 \ldots m$ Do
19.       $G \leftarrow (0[k+1]_\ell)^{m-i}0^{(1+\ell)i}$
20.       For $j \in 0 \ldots (k+2)^i - 1$ Do
21.           $curr \leftarrow (G >> (1+\ell)(i-1))$ & $0^{(1+\ell)(m-1)}01^\ell$
22.           $H[G] \leftarrow Min(H[G + 0^{(1+\ell)(m-i)}0[k+1-curr]_\ell 0^{(1+\ell)(i-1)}],\ h[i, curr])$
23.       $G \leftarrow$ Next($G, \ell, m, k+1$)
24.   Return $(D, S, J, H, F, \ell)$
```

Fig. 3. Our preprocessing.

Now, in order to perform each transition, we need to pay for t table accesses so as to compute $J[C^1 : C^2 : \dots C^t] = Min(J_1[C^1],\ J_2[C^2],\ \dots J_t[C^t])$ and $H[A^1 : A^2 : \dots A^t] = Min(H_1[A^1],\ H_2[A^2],\ \dots H_t[A^t])$, which makes the search time $O(tn)$. If we have $O(s)$ space, then we solve for $s = t(k+2)^{m/t}$, to obtain a search time of $O(tn) = O(mn/\log_k s)$.

6 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, namely $O(mn/\log_k s)$ time with $O(s)$ space. For lack of space we cannot present our experimental results in this paper, but they are available in [7]. There we show that, in practice, our algorithm clearly outperforms all previous solutions.

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, for example the simpler problem of approximate string matching with integer weights.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.
2. G. Berry and R. Sethi. From regular expression to deterministic automata. *Theor. Comp. Sci.*, 48(1):117–126, 1986.
3. V. Glushkov. The abstract theory of automata. *Russ. Math. Surv.*, 16:1–53, 1961.
4. E. Myers. A four-russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.
5. E. Myers and W. Miller. Approximate matching of regular expressions. *Bull. Math. Biol.*, 51:7–37, 1989.
6. G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.
7. G. Navarro. Approximate regular expression searching with arbitrary integer weights. Tech.Rep. TR/DCC-2002-6, Dept. of Computer Science, Univ. of Chile, July 2002. `ftp.dcc.uchile.cl/pub/users/gnavarro/aregexp.ps.gz`.
8. G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proc. WAE'01*, LNCS 2141, pages 1–12, 2001.
9. P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1(4):359–373, 1980.
10. K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
11. E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
12. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, 1992.
13. S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *J. of Algorithms*, 19(3):346–360, 1995.