NR-grep: A Fast and Flexible Pattern Matching Tool

Gonzalo Navarro*

Abstract

We present *nrgrep* ("nondeterministic reverse *grep*"), a new pattern matching tool designed for efficient search of complex patterns. Unlike previous tools of the *grep* family, such as *agrep* and *Gnu grep*, *nrgrep* is based on a single and uniform concept: the bit-parallel simulation of a nondeterministic suffix automaton. As a result, *nrgrep* can find from simple patterns to regular expressions, exactly or allowing errors in the matches, with an efficiency that degrades smoothly as the complexity of the searched pattern increases. Another concept fully integrated into *nrgrep* and that contributes to this smoothness is the selection of adequate subpatterns for fast scanning, which is also absent in many current tools. We show that the efficiency of *nrgrep* is similar to that of the fastest existing string matching tools for the simplest patterns, and by far unpaired for more complex patterns.

Key words: Online string matching, regular expression searching, approximate string matching, *grep*, *agrep*, BNDM.

1 Introduction

The purpose of this paper is to present a new pattern matching tool which we have coined *nrgrep*, for "nondeterministic reverse *grep*". *Nrgrep* is aimed at efficient searching for complex patterns inside natural language texts, but it can be used in many other scenarios.

The pattern matching problem can be stated as follows: given a text $T_{1..n}$ of n characters and a pattern P, find all the positions of T where P occurs. The problem is basic in almost every area of computer science and appears in many different forms. The pattern P can be just a simple string, but it can also be, for example, a regular expression. An "occurrence" can be defined as exactly or "approximately" matching the pattern.

In this paper we concentrate on *online* string matching, that is, the text cannot be indexed. Online string matching is useful for casual searching (i.e. users looking for strings in their files and unwilling to maintain an index for that purpose), dynamic text collections (where the cost of keeping an up-to-date index is prohibitive, including the searchers inside text editors and Web interfaces[1]), for not very large texts (up to a few hundred megabytes) and even as internal tools of indexed schemes (as *agrep* [29] is used inside *glimpse* [15] or *cgrep* [17] is used inside compressed indexes [21]).

*Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Work developed while the author was at postdoctoral stay at the Institut Gaspard Monge, Univ. de Marne-la-Vallée, France, partially supported by Fundación Andes and ECOS/Conicyt.

[1] We refer to the "search in page" facility, not to confuse with searching the Web.

There is a large class of string matching algorithms in the literature (see, for example, [26, 8, 4]) but not all of them are practical. There is also a wide variety of fast online string matching tools in the public domain, most prominently the *grep* family. Among these, *Gnu grep* and Wu and Manber's *agrep* [29] are widely known and currently considered as the fastest string matching tools in practice. Another distinguishing feature of these software systems is their flexibility: they can search not only for simple strings, but they also permit classes of characters (that is, a pattern position matches a set of characters), wild cards (a pattern position that matches an arbitrary string), regular expression searching, multipattern searching, etc. *Agrep* also permits approximate searching: the pattern matches the text after performing a limited number of alterations on it.

The algorithmic principles behind *agrep* are diverse [30]. Exact string matching is done with the Horspool algorithm [12], a variant of the Boyer-Moore family [6]. The speed of the Boyer-Moore string matching algorithms comes from their ability to "skip" (i.e. not inspect) some text characters. *Agrep* deals with more complex patterns using a variant of Shift-Or [2], an algorithm exploiting "bit-parallelism" (a concept that we explain later) to simulate nondeterministic automata (NFA) efficiently. Shift-Or, however, cannot skip text characters. Multipattern searching is treated with bit-parallelism or with a different algorithm depending on the case. As a result, the search performance of *agrep* varies sharply depending on the type of search pattern, and even slight modifications to the pattern yield widely different search times. For example, the search for the string `"algorithm"` is 7 times faster than for `"[Aa]lgorithm"` (where `"[Aa]"` is a class of characters that matches `"A"` and `"a"`, which is useful to detect the word either starting a sentence or not). In the first case *agrep* uses Horspool's algorithm and in the second case it uses Shift-Or. Intuitively, there should exist a more uniform approach where both strings could be efficiently searched for without a significant difference in the search time.

An answer to this challenge is the BNDM algorithm [22, 23]. BNDM is based on a previous algorithm, BDM (for "backward DAWG matching") [9, 8]. The BDM algorithm (to be explained later) uses a "suffix automaton" to detect substrings of the pattern inside a text window (the Boyer-Moore family detects only suffixes of the pattern). As the Boyer-Moore algorithms, BDM can also skip text characters. In the original BDM algorithm the suffix automaton is made deterministic. BNDM is a recent version of BDM that keeps the suffix automaton in nondeterministic form by using bit-parallelism. As a result, BNDM can search for complex patterns and still keep a search efficiency close to that of simple patterns. It has been shown experimentally [22, 23] that the BNDM algorithm is by far the fastest one to search for complex patterns. BNDM has been later extended to handle regular expressions [24].

Nrgrep is a pattern matching tool built over the BNDM algorithm (hence the name "nondeterministic reverse *grep*", since BNDM scans windows of the text in reverse direction). However, there is a gap between a pattern matching algorithm and a real software. The purpose of this work is to fill that gap. We have classified the allowed search patterns in three levels:

Simple patterns: a simple pattern is a sequence of m classes of characters (note that a single character is a particular case of a class). Its distinguishing feature is that an occurrence of a simple pattern has length m as well, as each pattern position matches one text position.

Extended patterns: an extended pattern adds to simple patterns the ability to characterize individual classes as "optional" (i.e. they can be skipped when matching the text) or "repeatable" (i.e. they can appear consecutively a number of times in the text). The purpose of extended patterns is to capture the most commonly used extensions of the normal search patterns so as to develop specialized pattern matching algorithms for them.

Regular expressions: a regular expression is formed by simple classes, the empty string, or the "concatenation", "union" or "repetition" of other regular expressions. This is the most general type of pattern we can search for.

We develop a different pattern matching algorithm (with increasing complexity) for each type of pattern, so simpler patterns are searched for with simpler and faster algorithms. The classification has been made having in mind the typical search needs on natural language, and it would be different, say, for DNA searching. We have also this in mind when we design the error model for approximate searching. "Approximate searching" or "searching allowing errors" means that the user gives an error threshold k and the system is able to find the pattern in the text even if it is necessary to perform k or less "operations" in the pattern to match its occurrence in the text. The operations typically permitted are the insertion, deletion and substitution of single characters. However, transposition of adjacent characters is an important typing error [14] that is normally disregarded because it is difficult to deal with. We allow the four operations in *nrgrep*, although the user can specify a subset of them.

An important aspect that deserves attention in order to obtain the desired "smoothness" in the search time is the selection of an optimal subpattern to scan the text. A typical case is an extended pattern with a large and repeatable class of characters close to one end. For technical reasons that will be made clear later, it may be very expensive to search for the pattern as is, while pruning the extreme of the pattern that contains the class (and verifying the potential occurrences found) leads to much faster searching. Some tools (such as *Gnu grep* for regular expressions) try to apply some heuristics of this type, but we provide a general and uniform subpattern optimization method that works well in all cases and, under a simplified probabilistic model, yields the optimal search performance for that pattern. Moreover, the selected subpattern may be of a simpler type than the whole pattern and a faster search algorithm may be possible. Detecting the exact type of pattern given by the user (despite the syntax used) is an important issue that is solved by the pattern parser.

We have followed the philosophy of *agrep* in some aspects, such as the record-oriented way to present the results and most of the pattern syntax features and search options. The main advantages of *nrgrep* over the *grep* family are uniformity in design, smoothness in search time, speed when searching for complex patterns, powerful extended patterns, improved error model for approximate searching, and subpattern optimization.

In this paper we start by explaining the concepts of bit parallelism and searching with suffix automata. Then we explain how these are combined to search for simple patterns, extended patterns and regular expressions. We later consider the approximate search of these patterns. Finally, we present the *nrgrep* software and show some experimental results on it. Despite that the algorithmic aspects of the paper borrow from our previous work in some cases [22, 23, 24, 25], the paper has some novel and nontrivial algorithmic contributions, such as

- searching for extended patterns, which implies the bit parallel simulation of new types of restricted automata;

- approximate searching allowing transpositions for the three types of patterns, which has never been considered under the bit-parallel approach; and
- algorithms to select optimal search subpatterns in the three types of patterns.

The *nrgrep* tool is freely available under a *Gnu* license from `http://www.dcc.uchile.cl/~gnavarro/pubcode/`.

2 Basic Concepts

We define in this section the basic concepts and notation needed throughout the paper.

2.1 Notation

We consider that the text is a sequence of n characters, $T = t_1 \ldots t_n$, where $t_i \in \Sigma$. Σ is the alphabet of the text and its size is denoted $|\Sigma| = \sigma$. In the simplest case the pattern is denoted as $P = p_1 \ldots p_m$, a sequence of m characters $p_i \in \Sigma$, in which case it specifies the single string $p_1 \ldots p_m$. More general patterns specify a finite or infinite set of strings.

We say that P matches T at position i whenever there exists a $j \geq 0$ such that $t_i \ldots t_{i+j}$ belongs to the set of strings specified by the pattern. The substring $t_i \ldots t_{i+j}$ is called an "occurrence" of P in T. Our goal is to find all the text positions that start a pattern occurrence.

The following notation is used for strings. $S_{i\ldots j}$ denotes the string $s_i s_{i+1} \ldots s_j$. In particular, $S_{i\ldots j} = \varepsilon$ (the empty string) if $i > j$. A string X is said to be a prefix, suffix and factor (or substring), respectively, of XY, YX and YXZ, for any Y and Z.

We use some notation to describe bit-parallel algorithms. We use exponentiation to denote bit repetition, e.g. $0^3 1 = 0001$. We denote as $b_\ell \ldots b_1$ the bits of a mask of length ℓ, which is stored somewhere inside the computer word of fixed length w (in bits). We use C-like syntax for operations on the bits of computer words, i.e. "|" is the bitwise-or, "&" is the bitwise-and, "^" is the bitwise-xor, "~" complements all the bits, and "<<" moves the bits to the left and enters zeros from the right, e.g. $b_\ell b_{\ell-1} \ldots b_2 b_1 << 3 \;=\; b_{\ell-3} \ldots b_2 b_1 000$. We can also perform arithmetic operations on the bits, such as addition and subtraction, which operate the bits as the binary representation of a number, for instance $b_\ell \ldots b_x 10000 - 1 = b_\ell \ldots b_x 01111$.

In the following we show that the pattern can be a more complex entity, matching in fact a set of different text substrings.

2.2 Simple Patterns

We call a "simple" pattern a sequence of characters or classes of characters. Let m be the number of elements in the sequence, then a simple pattern P is written as $P = p_1 \ldots p_m$, where $p_i \subseteq \Sigma$. We say that P matches at text position $i + 1$ whenever $t_i \in p_i$ for $i \in 1 \ldots m$. The most important feature of simple patterns is that they match a substring of the same length m in the text.

We use the following notation to describe simple patterns: we concatenate the elements of the sequence together. Simple characters (i.e. classes of size 1) are written down directly, while other classes of characters are written in square brackets. The first character inside the square brackets can be "^", which means that the class is exactly the complement of what is specified. The rest is a simple enumeration of the characters of the class, except that we allow ranges: "x-y" means all the characters between x and y inclusive (we assume a total order in Σ, which is in practice the ASCII code). Finally, the character "." represents a class equal to the whole alphabet and "#" represents the class of all separators (i.e. non alphanumeric characters). Most of these conventions are the same used in Unix software. Some examples are:

- "[Aa]merican", which matches "American" and "american";
- "[^\n]Begin", which finds "Begin" if it is not preceded by a line break;
- "../../197[0-9]", which matches any date in the 70's;
- ".e[^a-zA-Z_]t#", which permits any character in the first position, then "e", then anything except a letter or underscore in the third position, then "t", and finishes with a separator.

Note that we have used "\n" to denote the newline. We also use "\t" for the tab, "\xHH" for the character with hex ASCII code *HH*, and in general "\C" to interpret any character C literally (e.g. the backslash character itself, as well as the special characters that follow).

It is possible to specify that the pattern has to appear at the beginning of a line by preceding it with "^" or at the end of the line by following it with "$". Note that this is not the same as adding the newline preceding or following the pattern because the beginning/end of the file signals also the beginning/end of the line, and the same happens with records when the record delimiter is not the end of line.

2.3 Extended Patterns

In general, an extended pattern adds some extra specification capabilities to the simple pattern mechanism. In this work we have chosen some features which we believe are the most interesting for typical text searching. The reason to introduce this intermediate-level pattern (between simple patterns and regular expressions) is that it is possible to devise specialized search algorithms for them which can be faster than those for general regular expressions. The operations we permit for extended patterns are: specify optional classes (or characters), and permit the repetition of a class (or character). The notation we use is to add a symbol after the affected character or class: "?" means an optional class, "*" means that the class can appear zero or more times, and "+" means that it can appear one or more times. Some examples are:

- "colou?r", which matches "color" and "colour";
- "[a-zA-Z_][a-zA-Z_0-9]*", which matches valid variable names in most programming languages (a letter followed by letters or digits);
- "Latin#+America", which matches "Latin" and "America" separated by one or more separator characters (e.g. spaces, tabs, etc.).

2.4 Regular Expressions

A regular expression is the most sophisticated pattern that we allow to search for, and it is in general considered powerful enough for most applications. A regular expression is defined as follows.

- Basic elements: any character and the empty string (ε) are regular expressions matching themselves.
- Parenthesis: if e is a regular expression then so is (e), which matches the same strings. This is used to change precedence.
- Concatenation: if e_1 and e_2 are regular expressions, then $e_1 \cdot e_2$ is a regular expression that matches a string x iff x can be written as $x = yz$, where e_1 matches y and e_2 matches z.
- Union: if e_1 and e_2 are regular expressions, then $e_1|e_2$ is a regular expression that matches a string x iff e_1 or e_2 match x.
- Kleene closure: if e is a regular expression then $e*$ is a regular expression that matches a string x iff, for some n, x can be written as $x = x_1 \ldots x_n$ and e matches each string x_i.

We follow the same syntax (with the precedence order $*$, $\cdot$, $|$) except that we use square brackets to abbreviate $(x_1|x_2|\ldots|x_n) = [x_1 x_2 \ldots x_n]$ (where the x_i are characters), we omit the concatenation operator $(\cdot)$, we add the two operators $e+ = ee*$ and $e? = (e|\varepsilon)$ and we use the empty string to denote ε, e.g. `"a(b|)c"` denotes $a(b|\varepsilon)c$. These arrangements make the extended patterns to be a particular case of regular expressions. Some examples are:

- `"dog|cat"`, which matches `"dog"` and `"cat"`;
- `"((Dr.|Prof.|Mr.)#)*Knuth"`, which matches `"Knuth"` preceded by a sequence of titles.

3 Pattern Matching Algorithms

We explain in this section the basic string and regular expression search algorithms our software builds on.

3.1 Bit Parallelism and the Shift-Or Algorithm

In [2], a new approach to text searching was proposed. It is based on *bit-parallelism* [1]. This technique consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

Figure 1 shows a non-deterministic automaton that searches for a pattern in a text. Classical pattern matching algorithms, such as KMP [13], convert this automaton to a deterministic form and achieve $O(n)$ worst case search time. The Shift-Or algorithm [2], on the other hand, uses bit-parallelism to simulate the automaton in its non-deterministic form. It achieves $O(mn/w)$

worst-case time, i.e. an optimal speedup over a classical $O(mn)$ simulation. For $m \leq w$, Shift-Or is twice as fast as KMP because of better use of the computer registers. Moreover, it is easily extended to handle classes of characters.

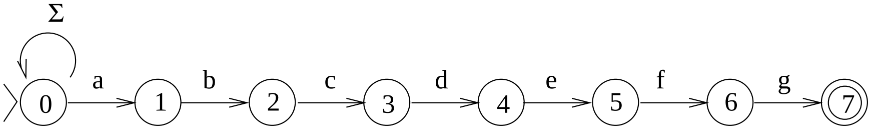

Figure 1: A nondeterministic automaton (NFA) to search for the pattern $P =$ "abcdefg" in a text.

3.1.1 Text Scanning

We present now the Shift-And algorithm [30], which is an easier to explain (though a little less efficient) variant of Shift-Or. Given a pattern $P = p_1 p_2 \ldots p_m$, $p_i \in \Sigma$ and a text $T = t_1 t_2 \ldots t_n$, $t_i \in \Sigma$, the algorithm builds first a table B which for each character stores a bit mask $b_m \ldots b_1$. The mask in $B[c]$ has the i-th bit set if and only if $p_i = c$. The state of the search is kept in a machine word $D = d_m \ldots d_1$, where d_i is set whenever $p_1 p_2 \ldots p_i$ matches the end of the text read up to now (another way to see it is to consider that d_i tells whether the state numbered i in Figure 1 is active). Therefore, we report a match whenever d_m is set.

We set $D = 0^m$ originally, and for each new text character t_j, we update D using the formula

$$D' \quad \longleftarrow \quad ((D << 1) \mid 0^{m-1}1) \ \ \& \ \ B[t_j]$$

The formula is correct because the i-th bit is set if and only if the $(i-1)$-th bit was set for the previous text character and the new text character matches the pattern at position i. In other words, $t_{j-i+1} \ldots t_j = p_1 \ldots p_i$ if and only if $t_{j-i+1} \ldots t_{j-1} = p_1 \ldots p_{i-1}$ and $t_j = p_i$. Again, it is possible to relate this formula to the movement that occurs in the NFA for each new text character: each state gets the value of the previous state, but this happens only if the text character matches the corresponding arrow. Finally, the "$\mid 0^{m-1}1$" after the shift allows a match to begin at the current text position (this operation is saved in the Shift-Or, where all the bits are complemented). This corresponds to the self-loop at the initial state of the automaton.

The cost of this algorithm is $O(n)$. For patterns longer than the computer word (i.e. $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time), with a worst-case cost of $O(mn/w)$ and still an average case cost of $O(n)$.

3.1.2 Classes of Characters and Extended Patterns

The Shift-Or algorithm is not only very simple, but it also has some further advantages. The most immediate one is that it is very easy to extend to handle classes of characters, where each pattern position may not only match a single character but a set of characters. If p_i is the set of characters that match the position i in the pattern, we set the i-th bit of $B[c]$ for all $c \in p_i$. No other change is necessary to the algorithm. In [2] they show also how to allow a limited number k of mismatches in the occurrences, at $O(nm \log(k)/w)$ cost.

This paradigm was later enhanced [30] to support allow wild cards, regular expressions, approximate search with nonuniform costs, and combinations of them. Further development of the bit-parallelism approach for approximate string matching yielded some of the fastest algorithms for short patterns [3, 18]. In most cases, the key idea was to simulate an NFA.

Bit-parallelism has become a general way to simulate simple NFAs instead of converting them to deterministic automata. This is how we use it in *nrgrep*.

3.2 The BDM Algorithm

The main disadvantage of Shift-Or is its inability to skip characters, which makes it slower than the algorithms of the Boyer-Moore [6] or the BDM [9, 8] families. We describe in this section the BDM pattern matching algorithm, which is able to skip some text characters.

BDM is based on a *suffix automaton.* A *suffix automaton* on a pattern $P = p_1p_2 \ldots p_m$ is an automaton that recognizes all the suffixes of P. A nondeterministic version of this automaton has a very regular structure and is shown in Figure 2. In the BDM algorithm [9, 8], this automaton is made deterministic.

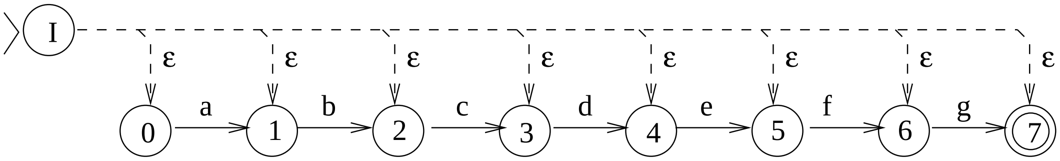

Figure 2: A nondeterministic suffix automaton for the pattern P = "abcdefg". Dashed lines represent ε-transitions (i.e. they occur without consuming any input).

A very important fact is that this automaton can be used not only to recognize the suffixes of P, but also factors of P. Note that there is a path labeled by x from the initial state if and only if x is a factor of P. That is, the NFA will not run out of active states as long as it has read a factor of P.

The suffix automaton is used to design a simple pattern matching algorithm. This algorithm runs in $O(mn)$ time in the worst case, but it is optimal on average ($O(n \log_\sigma m/m)$ time). Other more complex variations such as TurboBDM [9] and MultiBDM [8, 27] achieve linear time in the worst case.

To search for a pattern $P = p_1p_2 \ldots p_m$ in a text $T = t_1t_2 \ldots t_n$, the suffix automaton of $P^r = p_mp_{m-1} \ldots p_1$ (i.e the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm searches backward inside the window for a factor of the pattern P using the suffix automaton, i.e. the suffix automaton of the reverse pattern is fed with the characters in the text window read backward. This backward search ends in two possible forms:

1. We fail to recognize a factor, i.e we reach a window character σ that makes the automaton run out of active states. This means that the suffix of the window we have read is not anymore a factor of P. Figure 3 illustrates this case. We then shift the window to the right, its starting position corresponding to the position following the character σ (we cannot miss an

occurrence because in that case the suffix automaton would have found a factor of it in the window).

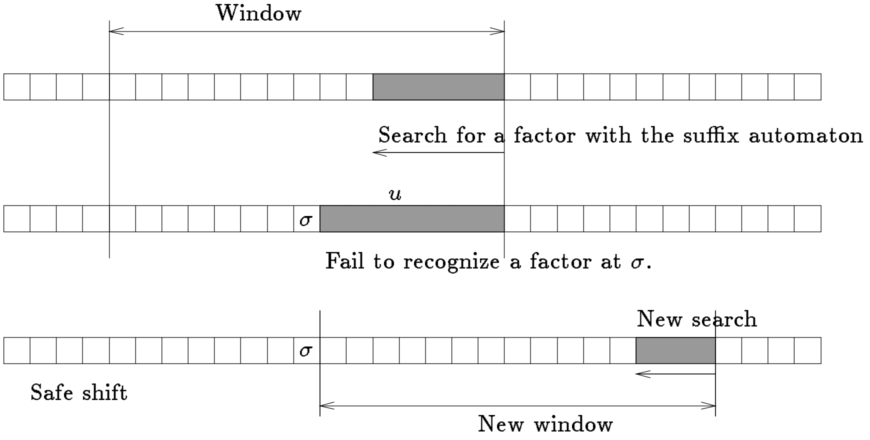
Window
Search for a factor with the suffix automaton
u
σ
Fail to recognize a factor at σ.
New search
σ
Safe shift
New window

Figure 3: Suffix automaton search.

2. We reach the beginning of the window, therefore recognizing the pattern P since the length-m window is a factor of P (indeed, it is equal to P). We report the occurrence, and shift the window by one position.

3.3 Combining Shift-Or and BDM: the BNDM Algorithm

We describe in this section the BNDM pattern matching algorithm [22]. This algorithm, a combination of Shift-Or and BDM, has all the advantages of the bit-parallel forward scan algorithm, and in addition it is able to skip some text characters like BDM.

Instead of making the automaton of Figure 2 deterministic, BNDM simulates it using bit-parallelism. The bit-parallel simulation works as follows. Just as for Shift-And, we keep the state of the search using m bits of a computer word $D = d_m \ldots d_1$. Each time we position the window in the text we initialize $D = 1^m$ (this corresponds to the ε-transitions) and scan the window backward. For each new text character read in the window we update D. If we run out of 1's in D then there cannot be a match and we suspend the scanning and shift the window. If, on the other hand, we can perform m iterations, then we report the match.

We use a table B which for each character c stores a bit mask. This mask sets the bits corresponding to the positions where the reversed pattern has the character c (just as in the Shift-And algorithm). The formula to update D is

$$D' \;\longleftarrow\; (D \;\&\; B[t_j]) \;<<\; 1$$

BNDM is not only faster than Shift-Or and BDM (for about $5 \leq m \leq 100$), but it can accommodate all the extensions mentioned in Section 2. In particular, it can easily deal with classes of characters by just altering the preprocessing, and it is by far the fastest algorithm to search for this type of patterns [22, 23].

Note that this type of search is called "backward" scanning because the text characters inside the window are read backwards. However, the search progresses from left to right in the text as the window is shifted. There have been other (few) attempts to skip characters under a Shift-Or approach, for example [10].

3.4 Regular Expression Searching

Bit-parallelism has been successfully used to deal with regular expressions. Shift-Or was extended in two ways [30, 24, 25] to deal with this case, first using the Thompson [28] and later Glushkov's [5] constructions of NFAs from the regular expression. Figure 4 shows both constructions for the pattern "`abcd(d|`ε`)(e|f)de`".

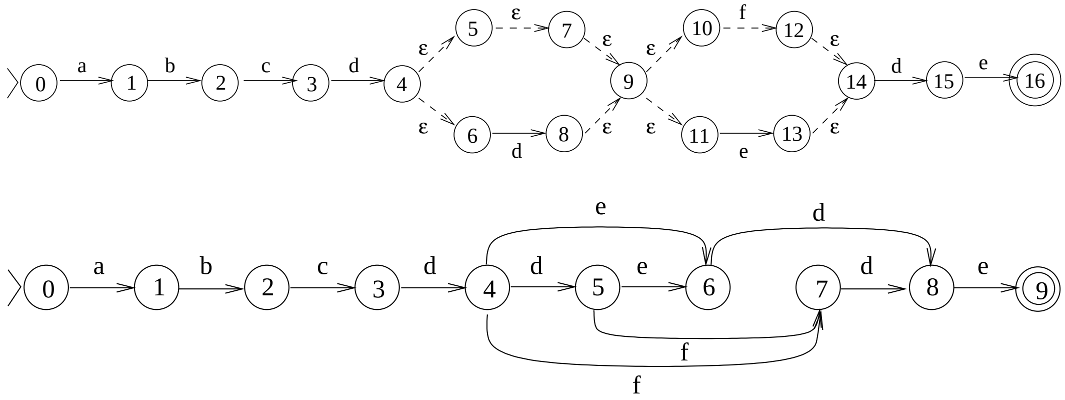

Figure 4: Thompson's (top) and Glushkov's (bottom) resulting NFAs for the regular expression "`abcd(d|`ε`)(e|f)de`".

Given a regular expression with m *positions* (each character/class defines a new position), Thompson's construction produces an automaton of up to $2m$ states. Its advantage is that the states of the resulting automaton can be arranged in a bit mask so that all the transitions move forward except the ε-transitions. This is used [30] for a bit-parallel simulation which moves the bits forward (as for the simple Shift-Or) and then applies all the moves corresponding to ε-transitions. For this sake, a table E mapping from bit masks to bit masks is precomputed, so that $E[x]$ yields a bit mask where x has been expanded with all the ε-moves. The code for a transition is therefore

$$\begin{array}{rcl} D & \longleftarrow & ((D << 1) \mid 0^{s-1}1) \ \& \ B[t_j] \\ D & \longleftarrow & E[D] \end{array}$$

We have used s as the number of states in the Thompson automaton, where $m < s \leq 2m$. The main problem is that the E table has 2^s entries. This is handled by splitting the argument horizontally, so for example if $s = 32$ then two tables E_1 and E_2 can be created which receive half masks and deliver full masks with the ε-expansion of only the bits of their half (of course the ε-transitions can go to the other half, this is why they deliver full masks). In this way the amount of memory required is largely reduced at the cost of two operations to build the real E value. This takes advantage of the fact that, if a bit mask x is split in two halves $x = yz$, then $E[yz] = E[y0^{|z|}] \mid E[0^{|y|}z]$.

Glushkov's construction has the advantage of producing an automaton with exactly $m+1$ states, which can as low as half the states generated by Thompson's construction. On the other hand, the structure of arrows is not regular and the trick of forward shift plus ε-moves cannot be used. Instead, it has another property: all the arrows leading to a given state are labeled by the same character or class. This property has been used recently [25] to provide a space-economical bit parallel simulation where the code for a transition is:

$$D \quad \longleftarrow \quad T[D] \ \& \ B[t_j]$$

where T is a table that receives a bit map D of states and delivers another bit map of states reachable from states in D, no matter by which characters. The ε-transitions do not have to be dealt with because Glushkov's construction does not produce them. The T table can be horizontally partitioned as well.

It has been shown [25] that a bit-parallel implementation of Glushkov's construction is faster than one of Thompson's, which should be clear since in general the tables obtained are much smaller. An interesting improvement, possible thanks to bit parallelism, is that classes of characters can be dealt with the normal mechanism used for simple patterns, without generating one state for each alternative.

On the other hand, a deterministic automaton (DFA) can be built from the nondeterministic one. It is not hard to see that indeed the previous constructions simulate a DFA, since each state of the DFA can be seen as a set of states of the NFA, and each possible set of states of the NFA is represented by a bitmask. Normally the DFA takes less space because only the reachable combinations are generated and stored, while for direct access to the tables we need to store in the bit-parallel simulations all the possible combinations of active and inactive states. On the other hand, bit-parallelism permits extending regular expressions with classes of characters and other features (e.g. approximate searching), which is difficult otherwise. Furthermore, Glushkov's construction permits not storing a table of *states* $\times$ *characters*, of worst case size $O(2^m\sigma)$ in the case of a DFA, but just the table T of size $O(2^m)$. Finally, in case of space problems the technique of splitting the bitmasks can be applied.

Therefore, we use the bit-parallel simulation of Glushkov's automaton for *nrgrep*. After the update operation and we check whether a final state of D is reached (this means just an *and* operation with the mask of final states). Describing Glushkov's NFA construction algorithm [5] is outside the scope of this paper, but it takes $O(m^2)$ time. The result of the construction can be represented as a table $B[c]$, which yields the states reached by character c (no matter from where), and a table $Follow[i]$, which yields the bitmask of states activated from state i, no matter by which character. From $Follow$, the deterministic version T can be built in $O(2^m)$ worst case time with the following procedure:

```
$T[0] \longleftarrow 0$
for $i \in 0 \ldots m$
    for $j \in 0 \ldots 2^i - 1$
        $T[2^i + j] \longleftarrow Follow[i] \mid T[j]$
```

A backward search algorithm for regular expressions is also possible [24, 25] and in some cases the search is much faster than a forward search. The idea is as follows. First, we compute the length of the shortest path from the initial to a final state (using a simple graph algorithm). This will be the length of the window in order not to lose any occurrence. Second, we reverse all the arrows of the automaton, make all the states initial, and take as the only final state the original initial state. The resulting automaton will have active states as long as we have read a reverse factor of a string matching the regular expression, and will reach its final state when we read in particular a reverse prefix. Figure 5 illustrates the result.

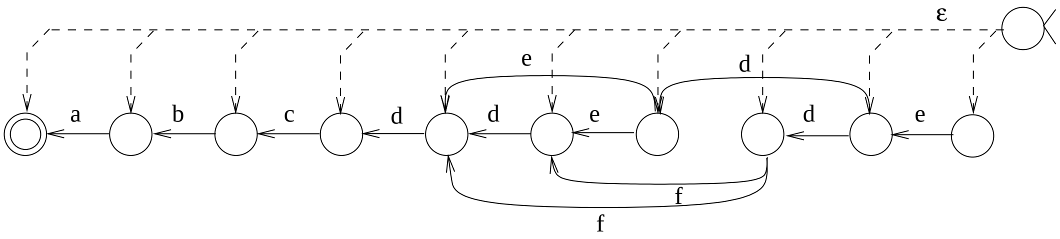

Figure 5: An automaton recognizing reverse prefixes of "abcd(d|ε)(e|f)de", based on the Glushkov construction of Figure 4.

We apply the same BNDM technique of reading backwards the text window. If the automaton runs out of active states, then no factor of an occurrence of the pattern is present in the window and we can shift the window, aligning its beginning to one position after the one that caused the mismatch. If, on the other hand, we reach the beginning of the window in the backward scan, we *cannot* guarantee that an occurrence has been found. When searching for a simple string, the only way to reach the window beginning is to have read the whole pattern. Regular expressions, on the other hand, can have occurrences of different length, and all we know is that we have matched a factor. There are in fact two choices.

- The final state of the automaton is not active; which means that we have not read a prefix of an occurrence. In this case we shift the window by one position and resume the scanning.
- The final state of the automaton is active. Since we have found a pattern prefix, we have to perform a forward verification starting at the window initial position until either we find an occurrence or the automaton runs out of active states.

So we need, apart from the reversed automaton, also the normal automaton (without initial self-loop, as in Figure 4) for the verification of potential occurrences.

An extra complication comes from the fact that the NFA with reverse arrows does not have the property that all arrows leading to a state are labeled by the same character. Rather, all the arrows *leaving* a state are labeled by the same character. Hence the simulation can be done as follows

$$D \quad \longleftarrow \quad T^R[D \;\&\; B[t_j]]$$

where T^R corresponds to the reverse arrows but B is that of the forward automaton [25].

3.5 Approximate String Matching

Approximate searching means finding the text substrings that can be converted into the pattern by performing at most k "operations" on them. Permitting a limited number k of such *differences* (also called *errors*) is an essential tool to recover from typing, spelling and OCR (optical character recognition) errors. Despite that approximate pattern matching can be reduced to a problem of regular expression searching, the regular expression grows exponentially with the number of allowed errors (or differences).

A first design decision is what should be taken as an error. Based on existing surveys [14, 19], we have chosen the following four types of errors: insertion of characters, deletion of characters, replacement of a character by another character, and exchange of adjacent characters (transposition). These errors are symmetric in the sense that one can consider that they occur in the pattern or in the text and the result is the same. Traditionally, only the first three errors have been permitted because transposition, despite being recognized as a very important source of errors, is harder to handle. However, the problem is known to grow very fast in complexity as k increases, and since a transposition can only be simulated with two errors of the other kind (i.e. an insertion and a deletion), we would need to double k in order to obtain a similar result. One of the algorithmic contributions of *nrgrep* is a bit-parallel algorithm for permitting the transpositions together with the other types of errors. This permits us searching with smaller k values and hence obtain faster searching with similar (or better) retrieval results.

Approximate searching is characterized by the fact that no known search algorithm is the best in all cases [19]. From the wealth of existing solutions, we have selected those that adapt best to our goal of flexibility and uniformity. Three main ideas can be used.

3.5.1 Forward Searching

The most basic idea that is well suited to bit parallelism [30] is to have $k+1$ similar automata, representing the state of the search when zero to k errors have occurred. Apart from the normal arrows inside each automaton, there are arrows going from automaton i to $i+1$ corresponding to the different errors. The original approach [30] did not consider transpositions, which have been dealt with later [16].

Figure 6 shows an example with $k=2$. Let us first focus on the big nodes and solid/dashed lines. Apart from the normal forward arrows we have three types of arrows that lead from each row to the next one (i.e. increment the number of errors): vertical arrows, which represent the insertion of characters in the pattern (since they advance in the text but not in the pattern); diagonal arrows, which represent replacement of the current text character by a pattern character (since they advance in the text and in the pattern); and dashed diagonal arrows (ε-transitions), which represent deletion of characters in the pattern (since they advance in the pattern without consuming any text input). The remaining arrows (the dotted ones) represent transpositions, which permit reading the next two pattern characters in the wrong order and move to the next row. This is achieved by means of "temporary" states, which we have drawn as smaller circles.

If we disregard the transpositions, there are different ways to simulate this automaton in $O(1)$ time when it fits in a computer word [3, 18], but no bit parallel solution has been presented to account for the transpositions. This is one of our contributions and is explained later in the paper. We extend a simpler bit parallel simulation [30], which takes $O(k)$ time per text character as long as $m = O(w)$. The technique stores each row in a machine word, $R_0 \ldots R_k$, just as we did for Shift-And in Section 3.1. The bitmasks R_i are initialized to $0^{m-i}1^i$ to account for i possible initial deletions in the pattern. The update procedure to produce R' upon reading text character t_j is as

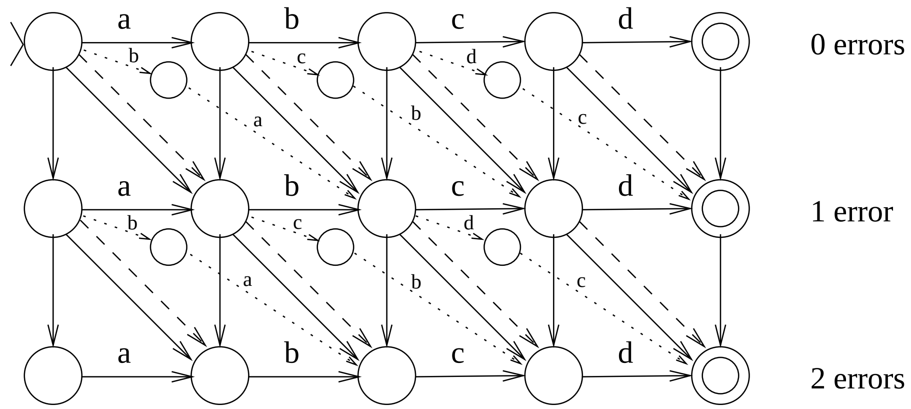

Figure 6: A nondeterministic automaton accepting the pattern "abcd" with at most 2 errors. Unlabeled solid arrows match any character, while the dashed (not dotted) lines are ε-transitions.

follows:

```
$R'_0 \longleftarrow ((R_0 << 1) \mid 0^{m-1}1) \ \& \ B[t_j]$
for $i \in 1 \dots k$ do
    $R'_i \longleftarrow ((R_i << 1) \ \& \ B[t_j]) \mid R_{i-1} \mid (R_{i-1} << 1) \mid (R'_{i-1} << 1)$
```

where of course many coding optimizations are possible (and are done in *nrgrep*) but make the code less clear. In particular, using the complemented version of the representation (as in Shift-Or) is a bit faster.

The rationale of the procedure is as follows. R_0 has the same update formula as for the Shift-And algorithm. For the others, the update formula is the *or* of four possible facts. The first one corresponds to the normal forward arrows (note that there is no initial self-loop for them, only for R_0). The second one brings 1's (state activations) from the upper row at the same position, which corresponds to a vertical arrow, i.e. an insertion. The third one brings 1's from the upper row at the *previous* positions (this is obtained with the left shift), corresponding to a diagonal arrow, i.e. a replacement. The fourth one is similar but it works on the *newer* value of the previous row (R'_{i-1} instead of R_{i-1}), and hence it corresponds to an ε-transition, i.e. a deletion.

A match is detected when $R_k \ \& \ 10^{m-1}$ is not zero. It is not hard to show that whenever the final state of R_i is active, the final state of R_k is active too, so it suffices to consider R_k as the only final state.

3.5.2 Backward Searching

Backward searching can be easily adapted from the forward searching automaton following the same techniques used for exact searching [22, 23]. That is, we build the automaton of Figure 6 on the reverse pattern, consider all the states as initial ones, and consider as the only final state the first node of the last row. This will recognize all the reverse prefixes of P allowing at most k errors, and will have active states as long as some factor of P has been seen (with at most k errors).

Some observations are of interest. First, note that we will never shift the window before examining at least $k+1$ characters (since we cannot make k errors before that). Second, the length of the window has to be that of the shortest possible match, which, because of deletions in the pattern, is of length $m - k$. Third, just as it happens with regular expressions, the fact that we arrive to the beginning of the window with some active states in the automaton does not immediately mean that we have an occurrence, so we have to check the text for a complete occurrence starting at the window beginning.

It has been shown [19] that this algorithm takes time $O(k(k + \log_\sigma m)/(m - k))$ for $m \leq w$.

3.5.3 Splitting into $k + 1$ Subpatterns

A well known property [30, 19] establishes that, under the model of insertions, deletions, and replacements, if the pattern is cut in $k + 1$ contiguous pieces, then at least one of the pieces occurs unchanged inside any occurrences with k errors or less. This is easily verified because each operation can alter at most one piece. So the technique consists of performing a multipattern searching for the pieces without errors, and checking the text surrounding the occurrences of each piece for a complete approximate occurrence of the whole pattern. This leads to the fastest algorithms for low error levels [20, 19].

The property is not true if we add the transposition, because this operation can alter two contiguous pieces at the same time. Much better than splitting the pattern in $2k + 1$ pieces is to split it in $k + 1$ pieces and leave one unused character between each pair of pieces [19]. Under this partition a transposition can alter at most one piece.

We are now confronted with a multipattern search problem. This can be solved with a very simple modification of the single pattern backward search algorithm [22, 23]. Consider the pattern "abracadabra" searched with two errors. We split it in "abr", "cad" and "bra". Figure 7 depicts the automaton used for backward searching of the three pieces. This is implemented with the same bit parallel mechanism as for a single pattern, except that (1) there are more final states; and (2) an extra bit mask is necessary to avoid propagating 1's by the missing arrows. This extra bit mask is implemented at no cost by removing the corresponding 1's from the B mask during the preprocessing. Note that for this to work we need that all the pieces have the same length.

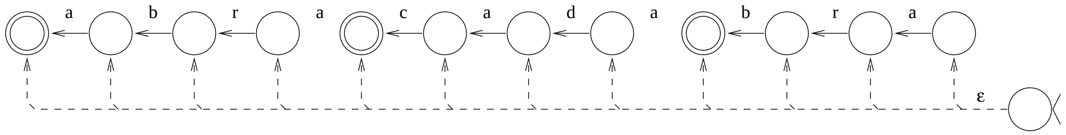

Figure 7: An automaton recognizing reverse prefixes of selected pieces of "abracadabra".

4 Searching for Simple Patterns

Nrgrep directly uses the BNDM algorithm when it searches for simple patterns. However, some modifications are necessary to convert the pattern matching algorithm into a search software.

4.1 Record Oriented Output

The first issue to consider is what will we report from the results of the search. Printing the text positions i where a match occurs is normally of little help for the user. Printing the text portion that matched (i.e. the occurrence) does not help much either, because this is equal to the pattern (at least if no classes of characters are used). We have followed *agrep*'s philosophy: the most useful way to present the result is to print a *context* of the text portion that matched the pattern.

This context is defined as follows. The text is considered to be a sequence of *records*. A user-defined *record delimiter* determines the text positions where a new record starts. The text areas until the first record separator and after the last record separators are considered records as well. When a pattern is found in the text, the whole record where the occurrence lies is printed. If the occurrence overlaps with or contains a record delimiter then it is not considered a pattern occurrence.

The record delimiter is by default the newline character, but the user can specify any other simple pattern as a record delimiter. For example, the string `"^From "` can be used to delimit e-mail messages in a mail archive, therefore being able to retrieve complete e-mails that contain a given string. The system permits to specify whether the record delimiter should be contained in the next or in the previous record. Moreover, *nrgrep* permits to specify an extra record separator when the records are printed (e.g. add another newline).

It should be clear by now that searching for longer strings is faster than for shorter ones. Since record delimiters tend to be short strings, it is not a good idea to delimit the text records first and then search for the pattern inside each record. Rather, we prefer to search for the pattern in the text without any consideration for record separators and, when the pattern is found, search for the next and previous record delimiters. At this time we may determine that the occurrence overlaps a record delimiter and discard it. Note that we have to be able to search for record delimiters forward and backward. We use the same BNDM algorithm to search for record delimiters.

There are some *nrgrep* options, however, that make it necessary a record-wise traversal: printing record numbers or printing records that do *not* contain the pattern. In this case we advance in the file by delimiting the records first and then searching for the pattern inside each record.

4.2 Text Buffering

Text buffering is necessary to cope with large files and to achieve optimum performance. For example, in *nrgrep* the buffer size is set to 64 Kb because it fits well the cache size of many machines, but this default can be overriden by the user. To avoid complex interactions between record limits and buffer limits, we discard the last incomplete record each time we read a new buffer from disk. The "discarded" partial record is moved to the beginning of the buffer before reading more text at the next buffer loading. If a record is larger than the buffer size then an artificial record delimiter is inserted to correct the situation (and a warning message is printed). Note that this also requires the ability to search for the record delimiter in backward direction. This technique works well unless the record size is large compared to the buffer size, in which case the user should enlarge the buffer size using the appropriate option.

4.3 Contexts

Another capability of *nrgrep* is context specification. This means that the pattern occurrence has to be surrounded by certain characters in order to be considered as such. For example, one may specify that the pattern should match as a whole word (i.e. surrounded by separators), or a whole record (i.e. surrounded by record delimiters). However, it is not just a matter of adding the context strings at the ends of the pattern because, for example, a word may be in the beginning of a record and hence the separator may be absent. We solve this by checking each occurrence found to determine that the required string is present before/after the occurrence or that we reached the beginning/end of the record.

This seems trivial for a simple pattern because its length is fixed, but for more complex patterns (such as regular expressions) where there may be many different occurrences in the same text area, we need a way to discard possible occurrences and still check for other ones that are basically in the same place. For example, the search for `"a*ba*"` as a whole word should match in the text `"aaa aabaa aaa"`. Despite that `"b"` alone is an occurrence of the pattern that does not fit the whole word criterion, the occurrence can be extended to another one that does. We return to this issue later.

4.4 Subpattern Filter

The BNDM algorithm is designed for the case $m \leq w$. Otherwise, we have in principle to simulate the algorithm using many computer words. However, as shown in [22, 23], it is much faster to prune the pattern to its first w characters, search for that subpattern, and try to extend its occurrences to an occurrence of the complete pattern. This is because, for reasonably large w, the probability of finding a pattern of length w is low enough to make the cost of unnecessary verifications negligible. On the other hand, the benefit of a possible shift of length $m > w$ would be cancelled by the need to update $\lceil m/w \rceil$ computer words per text character read.

Hence we select a contiguous subpattern of w characters (or classes, remember that a class needs also one bit, the same as a character) and search for it. Its occurrences are verified with the complete pattern prior to checking records and contexts.

The main point is which part of the pattern to search for. In the abstract algorithms of [22, 23], any part is equally good or bad because a uniformly distributed model is assumed. In practice, different characters have different probabilities, and some pattern positions may have classes of characters, whose probability is the sum of those of the individual characters. This in fact is farther reaching than the problem of the limit w in the length of the pattern: even in a short pattern we may prefer not to include a part of the pattern in the fast scanning part. This is discussed in detail in the next subsection.

4.5 Selecting the Optimal Scanning Subpattern

Let us consider the pattern `"hello...a"`. Pattern positions 6 to 8 match all the alphabet, which means that the search with the nondeterministic automaton inside the window will examine at least four window positions (even in a text window like "xxxxxxxxx") and will shift at most by 6, so the average number of comparisons per character is at the very best 2/3. If we take the subpattern `"hello"` then we can have an average much closer to 1/5 operations per text character.

We have designed a general algorithm that, under the assumption that the text characters are independent, finds the best search subpattern in $O(m^3)$ worst case time (although in practice it is closer to $O(m^2 \log m)$). This is a modest overhead in most practical text search scenarios. The algorithm is tailored to the BDM/BNDM search technique and works as follows.

First, we build an array $prob[1 \ldots m]$, which stores the sum of the probabilities of the characters participating in the class of each pattern position. *Nrgrep* stores an array of English letter probabilities, but this can be tailored to other purposes and the final scheme is robust with respect to changes in those probabilities from one language to another. The construction of *prob* takes $O(m\sigma)$ time in the worst case.

Second, we build an array $pprob[1 \ldots m, 1 \ldots m]$, where $pprob[i, \ell]$ stores the probability of matching the subpattern $P_{i \ldots i+\ell-1}$. This is computed in $O(m^2)$ time by dynamic programming, as follows

$$pprob[i,0] \quad \longleftarrow \quad 1 \quad , \quad pprob[i,\ell+1] \quad \longleftarrow \quad prob[i] \times pprob[i+1,\ell]$$

for increasing ℓ values.

Third, we build an array $mprob[1 \ldots m, 1 \ldots m, 1 \ldots m]$, where $mprob[i, j, \ell]$ gives the probability of matching any pattern substring of length ℓ in $P_{i \ldots j-1}$. This is computed in $O(m^3)$ time by dynamic programming using the formulas

$$\begin{array}{rrcl} & mprob[i,j,0] & \longleftarrow & 1 \\ (\ell > 0) & mprob[i,i+\ell-1,\ell] & \longleftarrow & 0 \\ (\ell > 0, j-i \geq \ell) & mprob[i,j,\ell] & \longleftarrow & 1-(1-pprob[i,\ell])(1-mprob[i+1,j,\ell]) \end{array}$$

for decreasing i values. Note that we have used the formula for the union of independent events $Pr(A \cup B) = 1-(1-Pr(A))(1-Pr(B))$.

Finally, the average cost per character associated to a subpattern $P_{i \ldots j}$ is computed with the following rationale. With probability 1 we inspect one window character. If *any* pattern position in the range $i \ldots j$ matches the window character read, then we read a second character (recall Section 3.2). If *any* consecutive pair of pattern positions in $i \ldots j$ matches the two window characters read, then we read a third character, and so on. This is what we have computed in *mprob*. The expected number of window characters to read is therefore

$$pcost[i,j] \quad \longleftarrow \quad mprob[i,j,0] + mprob[i,j,1] + \ldots + mprob[i,j,j-i] \tag{1}$$

In the BDM/BNDM algorithm, as soon as the window suffix read ceases to be found in the pattern, we shift the window to the position following the character that caused the mismatch. A simplified computation considers that the above $pcost[i, j]$ (which is an average) can be used as a fixed value, and therefore we approximate the real average number of operations per text character as

$$ops[i,j] \quad \longleftarrow \quad \frac{pcost[i,j]}{j-i-pcost[i,j]+1}$$

which is the average number of characters inspected divided by the shift obtained. Once this is obtained we select the (i, j) pair that minimizes the work to do. We also avoid considering cases where $j - i > w$. The total time needed to obtain this has been $O(m^3)$.

The amount of work is reduced by noting that we can check the ranges in increasing order of i values, and therefore we do not need the first coordinate of *mprob* (which can be independently computed for each i). Moreover, we start by considering maximum j, since in practice longer

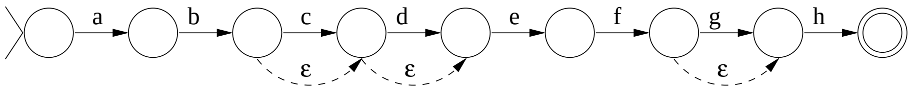

Figure 8: A nondeterministic automaton accepting the pattern "abc?d?efg?h".

subpatterns tend to be better than shorter ones. We keep the best value found up to now and avoid considering ranges (i, j) which cannot be better than the current best solution even for $pcost[i, j] = 1$ (note that since i is tried in ascending order and j in descending order, the whole pattern is tried first). This reduces the cost to $O(m^2 \log m)$ in practice.

As a result of this procedure we not only obtain the best subpattern to search for (under a simplified cost model) but also a hint of how many operations per character will we perform. If this number is larger than 1, then it is faster and safer to switch to plain Shift-Or. This is precisely what *nrgrep* does.

5 Searching for Extended Patterns

As explained in Section 2, we have considered optional and repeatable (classes of) characters as the features allowed in our extended patterns. Each of these features is treated in a different way and all are integrated in an automaton which is more general than that of Figure 1. Over this automaton we later apply the general forward and backward search machinery.

5.1 Optional Characters

Let us consider the pattern "abc?d?efg?h". A nondeterministic automaton accepting that pattern is drawn in Figure 8.

The figure is chosen so as to show that multiple consecutive optional characters could exist. This outrules the simplest solution (which works when that does not happen): one could set up a bit mask O with ones in the optional positions (in our example, $O = 01001100$), and let the ones in previous states of D propagate to them. Hence, after the normal update to D, we could perform

$$D \longleftarrow D \mid ((D << 1) \; \& \; O)$$

For example, this works if we have read "abcdef" ($D = 00100000$) and the next text character is "h", since the above operation would convert D to 01100000 before operating it against $B[\text{"h"}] = 10000000$. However, it does not work if the text is "abefgh", where both consecutive optional characters have been omitted.

A general solution needs to propagate each active state in D so as to flood all the states ahead it that correspond to optional characters. In our example, we would like that when D is 00000010 (and in general whenever its second bit is active), it becomes 00001110 after the flooding.

This is achieved with three masks, A, I and F, marking different aspects of the states related to optional characters. More specifically, the i-th bit of A is set if this position in P is optional; that of I is set if this is the position of the first optional character of a block (of consecutive optional characters); and that of F is set if this is the position after the last optional character of a block.

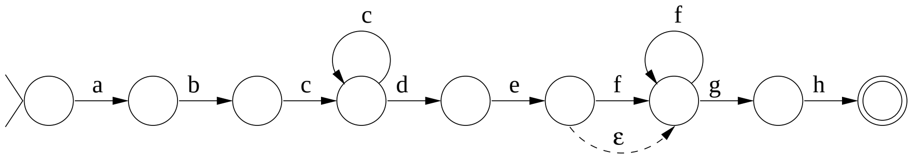

Figure 9: A nondeterministic automaton accepting the pattern `"abc+def*gh"`.

In our example, $A = 01001100$, $I = 01000100$ and $F = 10010000$. After performing the normal transition on D, we do as follows

$$\begin{aligned} Df &\longleftarrow D \mid F \\ D &\longleftarrow D \mid (A \,\&\, ((\sim (Df - I)) \,\hat{}\, Df)) \end{aligned}$$

whose rationale is as follows. The first line adds a 1 at the positions following optional blocks in D. In the second line we add some active states to D. Since the states to add are *and*-ed with A, let us just consider what happens inside a specific optional block. The effect that we want is that the first 1 (counting from the right) floods all the block bits to the left of it. We subtract I from Df, which is equivalent to subtracting 1 at each block. This subtraction cannot propagate its effect outside the block because there is a 1 (coming from "$|F$" in Df) after the highest bit of the block. The effect of the subtraction is that all the bits until the first 1 (counting from the right) are reversed (e.g. $1000000 - 1 = 0111111$), and the rest are unchanged. In general, $b_x b_{x-1} \ldots b_{x-y} 10^z - 1 = b_x b_{x-1} \ldots b_{x-y} 01^z$. When this is reversed by the "$\sim$" operation we get $\sim b_x \ \sim b_{x-1} \ldots \ \sim b_{x-y} 10^z$. Finally, when this is *xor*-ed with the same $Df = b_x b_{x-1} \ldots b_{x-y} 10^z$ we get $1^{x-y+1} 0^{z+1}$.

This is precisely the effect we wanted: the last 1 flooded all the bits to the left. That 1 itself has been converted to zero, however, but it is restored when the result is *or*-ed with the original D. This works even if the last active state in the optional block is the leftmost bit of the block. Note that it is necessary to *and* with A at the end to filter out the bits of F that survive the process whenever the block is not all zeros. On the other hand, it is necessary to *or* Df with F because a block of all zeros would propagate the "$-$" operation outside its limits.

Note that we could have a border problem if there are optional characters at the beginning of the pattern. As seen later, however, this cannot happen when we select the best subpattern for fast scanning, but it has to be dealt with when verifying the whole pattern.

5.2 Repeatable Characters

There are two kinds of repeatable characters, marked in the syntax by `"*"` (zero or more repetitions) and `"+"` (one or more repetitions). Each of them can be simulated using the other since $a+ = aa*$ and $a* = a+?$. For involved technical reasons (that are related, for example, to the ability to build easily the masks for the reversed patterns and border conditions for the verification) we preferred the second choice, despite that it uses one more bit than necessary for the `"*"` operation. Figure 9 shows the automaton for `"abc+def*gh"`.

The bit-parallel simulation of this automaton is more straightforward than for the optional characters. We just need to have a mask $S[c]$ that for each character c tells which pattern positions can remain active when we read character c. In the above example, $S[”\texttt{c}”] = 00000100$ and $S[”\texttt{f}”] = 00100000$. The ε-transition is handled with the mechanism for optional characters. Therefore a complete simulation step permitting optional and repeatable characters is as follows.

$$\begin{array}{rcl} D & \longleftarrow & ((D << 1) \mid 0^{m-1}1) \;\&\; B[t_j]) \;\mid\; (D \;\&\; S[t_j]) \\ Df & \longleftarrow & D \mid F \\ D & \longleftarrow & D \mid (A \;\&\; ((\sim (Df - I)) \,\hat{}\, Df)) \end{array}$$

5.3 Forward and Backward Search

Our aim is to extend the approach used for simple patterns to patterns containing optional symbols. Forward scanning is immediate once we learn how to simulate the different automata using bit parallelism. We just have to add an initial self-loop to enable text scanning (this is already done in the last formula). We detect the final positions of occurrences and then check the surrounding record and context conditions.

Backward searching needs, in principle, just to obtain an automaton that recognizes reverse factors of the pattern. This is obtained by building exactly the same automaton of the forward scan (without initial self-loop) on the reverse pattern, and letting all the states be initial (i.e. initializing D with all active states). However, there are some problems to deal with, all of them deriving from the fact that the occurrences have variable length now.

Since the occurrences do not have a fixed length, we have to compute the minimum length of a possible match of the pattern (e.g. 7 in the example `"abc+def*gh"`) and use this value as the width of the search window in order not to lose any potential occurrence. As before, we set up an automaton that recognizes all the reverse factors of the automaton and use it to traverse the window backward. Because the occurrences are not all of the same length, the fact that we arrive to the beginning of the window does not immediately imply that the pattern is present. For example, a 7-length text window could be `"cdefffg"`. Despite that this is a factor of the pattern and therefore we would reach the window beginning, no pattern occurrence starts in the beginning of the window.

Therefore, each time we arrive to the beginning of the window we have to check that the initial state is active and then run a forward verification from the window beginning on, until either we find a match (i.e. the last automaton state is activated) or we determine that no match can start at the window position under consideration (i.e. the automaton runs out of active states). The automaton used for this forward verification is the same as for forward scanning, except that the initial self-loop is absent. However, as we see next, verification is in fact a little more complicated and we mix it with the rest of verifications that are needed on every occurrence (surrounding record, context, etc.).

5.4 Verifying Occurrences

In fact, the verification is a little different since, as we see in the next subsection, we select a subpattern for the scanning (as before, this is necessary if the pattern has more than w characters/classes, but can also be convenient on shorter patterns). Say that $P = P_1 S P_2$, where S is the subpattern that has been selected for fast text scanning. Each time the backward search determines that a given text position is a potential start point for an occurrence of S, we obtain the surrounding record and check, from the candidate text position, the occurrence of SP_2 in forward direction and P_1 in backward direction (do not confuse forward/backward scanning with forward/backward verification!).

When we had a simple pattern, this just needed to check that each text character belonged to the corresponding pattern class. Since the occurrences have variable length, we need to use pre-built automata for SP_2 and for the reversed version of P_1. These automata do not have the initial self-loop. However, this time we need to use a multi-word bit-parallel simulation, since the patterns could be longer than the computer word.

Note also that, under this scenario, the forward scanning also needs verification. In this case we find a guaranteed end position of S in the text (not a candidate one as for backward searching). Hence we check, from that final position, P_2 in forward direction and P_1S in backward direction. Note that we need to check S again because the automaton cannot tell where is the beginning of S, and there could be various beginning positions.

A final complication is introduced by record limits and context conditions. Since we require that a valid occurrence lies totally inside a record, we should submit for verification the smallest possible occurrence. For example, the pattern "`b[ab]*cde?`" has many occurrences in the text record "`bbbcdee`", and we should report "`bcd`" in order to guarantee that no valid occurrence will be missed. However, it is possible that the context conditions require the presence of certain strings immediately preceding or following the occurrence. For example, if the context condition tells that the occurrence should begin a record, then the minimal occurrence "`bcd`" would not qualify, while "`bbbcde`" would do.

Fortunately, context conditions about the initial and final positions of occurrences are independent, and hence we can check them separately. So we treat the forward and backward parts of the verification separately. For each one, we traverse the text and find all the positions that represent the beginning (or ending) of occurrences and submit them to the context checking mechanism until one is accepted, the record ends, or the automaton runs out of active states.

5.5 Selecting a Good Search Subpattern

As before, we would like to select the best subpattern for text scanning, since we have anyway to check the potential occurrences. We want to apply an algorithm similar to that for simple patterns. However, this time the problem is more complicated because there are more arrows in the automaton.

We compute *prob* as before, adding another array $sprob[1 \ldots m]$, which is the probability of staying at state i via the $S[c]$ array. The major complication is that a factor of length ℓ starting at pattern position i does not necessarily finish at pattern position $i + \ell - 1$. Therefore, we fill an array $pprob[1 \ldots m, 1 \ldots m, 1 \ldots L]$, where L is the minimum length of an occurrence of P (at most m) and $pprob[i, j, \ell]$ is the sum of the probabilities of all the factors of length ℓ that start at position i and do not reach after pattern position j. In our example "`abc+def*gh`", $pprob[3, 6, 4]$ should account for "`ccccc`", "`ccccd`", "`cccde`", "`ccdef`" and "`cdeff`".

This is computed as

$$\begin{array}{rlcl} & pprob[i,i-1,\ell] & \longleftarrow & 0 \\ (i \le j) & pprob[i,j,0] & \longleftarrow & 1 \\ (i \le j \wedge \ell > 0) & pprob[i,j,\ell] & \longleftarrow & prob[i] \times pprob[i+1,j,\ell-1] \\ & & & +\ sprob[i] \times pprob[i,j,\ell-1] \\ & & & +\ (\text{if } i\text{–th bit of } A \text{ is set})\ pprob[i+1,j,\ell] \end{array}$$

which is filled for decreasing i and increasing ℓ in $O(m^3)$ time. Note that we are simplifying the computation of probabilities, since we are computing the probability of a set of factors as the sum of the individual probabilities, which is only true if the factors are disjoint.

Similarly to $pprob[i,j,\ell]$ we have $mprob[i,j,\ell]$ as the probability of any factor of length ℓ starting at position i or later and not surpassing position j. This is trivially computed from *pprob* as for simple patterns.

Finally, we compute the average cost per character as before. We consider subpatterns from length $\min(m,w)$ until a length that is so short that we will always prefer the best solution found up to now. For each subpattern considered we compute the expected cost per window as the sum of the probabilities of the subpatterns of each length, i.e.

$$pcost[i,j] \quad \longleftarrow \quad mprob[i,j,0] + mprob[i,j,1] + \ldots + mprob[i,j,j-i]$$

and later obtain the cost per character as $pcost[i,j]/(\ell - pcost[i,j]+1)$, where ℓ is the minimum length of an ocurrence of the interval (i,j).

As for simple patterns, we fill *pprob* and *mprob* in lazy form, together with the computation of the best factor. This makes the expected cost of the algorithm closer to $O(m^2 \log m)$ than to the worst case $O(m^3)$ (really $O(m^2 \min(m,w))$).

Note that it is not possible (because it is not optimal) to select a subpattern that starts or ends with optional characters or with "*". If it happens that the best subpattern gives one or more operations per text character, we switch to forward searching and select the first $\min(m,w)$ characters of the pattern (excluding initial and final "?"'s or "*"'s).

In particular, note that it is possible that the scanning subpattern selected has not any optional or repeatable character. In this case we use the scanning algorithm for simple patterns, despite that at verification time we use the checking algorithm of extended patterns.

6 Searching for Regular Expressions

The most complex patterns that *nrgrep* can search for are regular expressions. For this sake, we use the technique explained in Section 3.4 [24, 25], both for forward and backward searching. However, some aspects need to be dealt with in a real software.

6.1 Space Problems

The first problem is space, since the T table needs $O(2^m)$ entries and this can be unmanageable for long patterns. Despite that we expect that the patterns are not very long in typical text searching, some reasonable solution has to be provided when this is not the case.

We permit the user to specify the amount of memory that can be used for the table, and split the bitmasks in as many parts as needed to meet the space requirements. Since we do not search for masks longer than w bits, it is in fact unlikely that the text scanning part needs more than 3 or 4 table accesses per text character. Attending to the most common alternatives, we developed separate code for the cases of 1 and 2 tables, which permits much faster scanning since register usage is enabled.

6.2 Subpattern Filtering

As for simple and extended patterns, we select the best subpattern for the scanning phase, and check all the potential occurrences for complete occurrences and for record and context conditions. This means, according to Section 3.4, that we need a forward and a backward automaton for the selected subpattern, and that we also need forward and backward verification automata to check for the complete occurrence. An exception is when, given the result of selecting the best factor, we prefer to use forward scanning, in which case only that automaton is needed (but the two verification automata are still necessary). All the complications addressed for extended patterns are present on regular expressions as well, namely, those derived from the fact that the occurrences may have different lengths.

It may also happen that, after selecting the search subpattern, it turns out to be just a simple or an extended pattern, in which case the search is handled by the appropriate search algorithm, which should be faster. The verification is handled as a general regular expression. Note that heuristics like those of *Gnu Grep*, which tries to find a literal string inside the regular expression in order to use it as a filter, are no more than particular cases of our general optimization method. This makes our approach much smoother than others. For example *Gnu Grep* will be much faster to search for "c+(aaaaa|bbbbb)c+" (where the strings "caaaaac" and/or "cbbbbbc" can be used to filter the search) than for "c+[ab][ab][ab][ab][ab]c+", while the difference should not be as large.

Regarding optimal use of space (and also because accessing smaller tables is faster) we use a state remapping function. When a subset of the states of the automaton is selected for scanning we build a new automaton with only the necessary states. This reduces the size of the tables.

What is left is to explain how we select the best factor to search for. This is much more complex on regular expressions than on extended patterns.

6.3 Selecting an Optimal Necessary Factor

The first nontrivial task on a regular expression is to determine what is a *necessary factor*, which is defined as a subexpression that has to match inside every occurrence of the whole expression. For example, "fgh" is not a necessary factor of "ab(cde|fghi)jk", but "cd|fgh" is. Note that any range of states is a necessary factor in a simple or extended pattern.

Determining necessary subexpressions is complicated if we try to do it using just the automaton graph. We rather make use of the syntax tree of the regular expression as well. The main trouble is caused by the "|" operator, which forces us to choose one necessary factor from each branch. We simplify the problem by fixing the minimum occurrence length and searching for a necessary factor of that length on each side. In our previous example, we could select "cd|fg" or "de|hi", for example, but not "cd|fgh". However, we are able to take the whole construction, namely "cde|fghi".

The procedure is recursive and aims at finding the best necessary factor of minimum length ℓ provided we already know (we consider later the computation of these data)

- $wlens[0 \ldots m]$, where $wlens[i]$ is the minimum length of a path from i to a final state;
- $mark, markf[0 \ldots m, 0 \ldots L]$, where $mark[i, \ell]$ is the bitmask of all the states reachable in ℓ steps from state i, and $markf$ sets the corresponding final states;
- $cost[0 \ldots m, 0 \ldots L]$, where $cost[i, \ell]$ is the average cost per character if we search for all the paths of length ℓ leaving from state i.

The method starts by analyzing the root operator of the syntax tree. Depending on the type of operand, we do as follows.

Concatenation: choose the best factor (minimum cost) among both sides of the expression. Note that factors starting in the left side can continue to the right side, but we are deciding about where the initial positions are.

Union: choose the best factor from each side and take the union of the states. The average cost is the maximum over the two sides (not the sum, since the cost is related to the average number of characters to inspect).

One or more repetition $(+)$**:** the minimum is one repetition, so ignore the node and treat the subtree.

Zero or more repetitions $(?, *)$**:** the subtree cannot contain a necessary factor since it does not need to appear in the occurrences. Nothing can be found starting inside it. We assign a high cost to the factors starting inside the subexpression to make sure that it is not chosen in a concatenation, and that a union containing it will be equally undesirable.

Simple character or class (tree leaf): there is only one possible initial position, so choose it unless its $wlens$ value is smaller than ℓ.

The procedure delivers a set of selected states and the proper initial and final states for the selected subautomaton. It also delivers a reasonable approximation of the average cost per character.

The rest of the work is to obtain the input for this procedure. The ideas are similar as for extended patterns, but the techniques need to make heavier use of graph traversal and are more expensive. For example, just computing $wlens$ and $L = wlens[initial\ state]$, i.e. shortest paths from any state to a final state, we need a Floyd-like algorithm that takes $O(Lm^3/w) = O(\min(m^3, m^4/w))$ time.

Arrow probabilities $prob$ are loaded as before, but $pprob[i, \ell]$ now gives the total probability of all the paths of length ℓ leaving state i, and it includes the probability of reaching i from a previous state. In Glushkov's construction, all the arrows reaching a state have the same label, so *pprob* is computed for increasing ℓ using the formulas

$$\begin{array}{rrcl} & pprob[i,0] & \longleftarrow & 1 \\ & pprob[i,1] & \longleftarrow & prob[i] \\ (\ell \geq 2) & pprob[i,\ell] & \longleftarrow & prob[i] \times \displaystyle\sum_{j,(i,j)\in NFA} pprob[j,\ell-1] \end{array}$$

This takes $O(Lm^2) = O(\min(m^3, m^2w))$ time. At the same time we compute the $mark, markf$ masks, at a total cost of $O(\min(m^3, m^4/w))$:

$$\begin{array}{lrcl} & mark[i,0], markf[i,0] & \longleftarrow & 0^{m+1} \\ & mark[i,1], markf[i,1] & \longleftarrow & 0^{m-i}10^{i-1} \\ (\ell \geq 2) & mark[i,\ell] & \longleftarrow & mark[i,\ell-1] \cup \displaystyle\bigcup_{j,(i,j)\in NFA} mark[j,\ell-1] \\ (\ell \geq 2) & markf[i,\ell] & \longleftarrow & \displaystyle\bigcup_{j,(i,j)\in NFA} markf[j,\ell-1] \end{array}$$

for increasing ℓ values. Once *pprob* is computed, we build $mprob[0\ldots m, 0\ldots L, 0\ldots L]$, so that $mprob[i,\ell,\ell']$ is the probability of any path of length ℓ' inside the area $mark[i,\ell]$. This is computed in $O(L^2m^2) = O(\min(m^4, m^2w^2))$ time with the formulas:

$$\begin{array}{rrcl} (\ell' > \ell) & mprob[i,\ell,\ell'] & \longleftarrow & 0 \\ & mprob[i,\ell,0] & \longleftarrow & 1 \\ (0 < \ell' \leq \ell) & mprob[i,\ell,\ell'] & \longleftarrow & 1 - \left(1 - pprob[i,\ell']\right) \displaystyle\prod_{j,(i,j)\in NFA} \left(1 - mprob[j,\ell-1,\ell']\right) \end{array}$$

Finally, the search cost is computed as always using *mprob*, in $O(L^2m) = O(\min(m^3, mw^2))$ time.

Again, it is possible that the scanning subpattern selected is in fact a simpler type of pattern, such as a simple or extended one. In this case we use the appropriate scanning procedure, which should be faster.

Another nontrivial possibility is that even the best necessary factor is too bad (i.e. it has a high predicted cost per character). In this case we select from the initial state of the automaton a subset of it that fits in the computer word (i.e. at most w states), intended for forward scanning. Since all the occurrences found with this automaton will have to be checked, we would like to select the subautomaton that finds the least possible spurious matches. If $m \leq w$ then we use the whole automaton, otherwise we try to minimize the probability of getting outside the set of selected states. To compute this, we consider that the probability of reaching a state is inversely proportional to its shortest path from the initial state. Hence we add states farther and farther from the root until we have w states. All this takes $O(m^2)$ time.

This probabilistic assumption is of course simplistic, but an optimal solution is quite hard and at this point we know that the search will be costly anyway.

6.4 Verification

A final nontrivial problem is how to determine which states should be present in the forward and backward verification automata once the best scanning subautomaton is selected. Since we have chosen a necessary factor, we know for sure that the automaton is "cut" in two parts by the necessary factor. If we have chosen a backward scanning, then the verifications start from the *initial* states of the scanning automaton, otherwise they start from its *final* states.

Figure 10 illustrates these automata for the pattern "abcd(d|ε)(e|f)de", where we have selected "d(d|ε)(e|f)" as our scanning subexpression. This corresponds to the states $\{3, 4, 5, 6, 7\}$ of the original automaton. The selected arrows and states are in boldface in the figure (note that arrows leaving from the selected subautomaton are not selected). Depending on whether the selected subautomaton is searched with backward or forward scanning, we know the text position where its initial or final states, respectively, were reached. Hence, in backward scanning the verification starts from the initial states of the subautomaton, while it starts from the final states in case of forward scanning. We need two full automata: the original one and one with the arrows reversed.

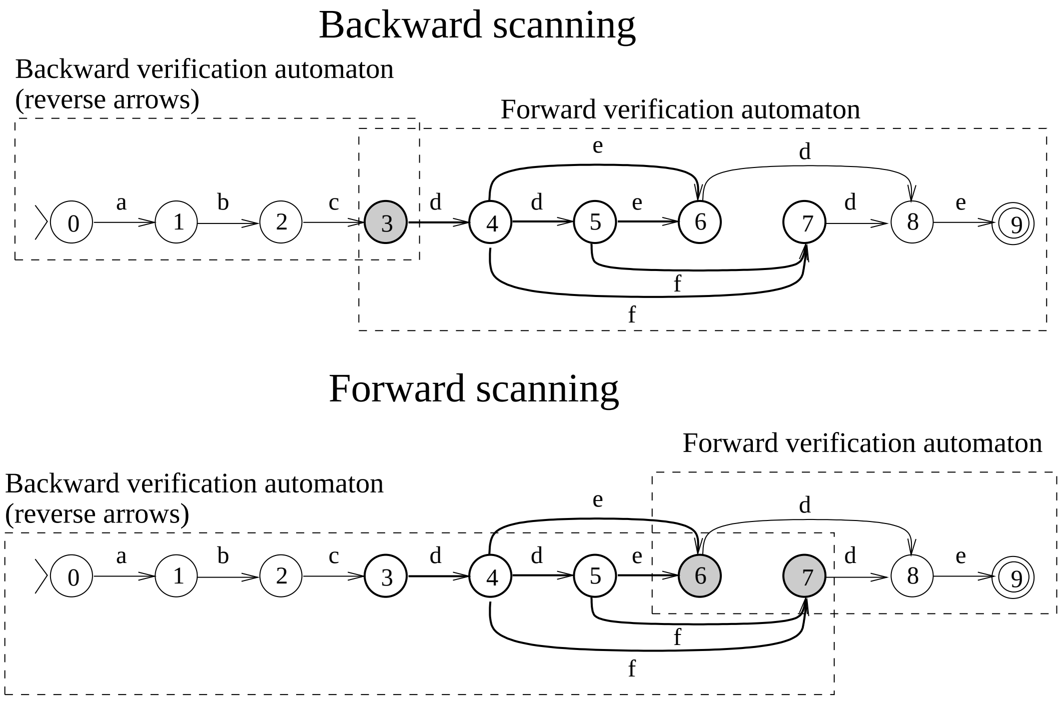
Backward scanning
Backward verification automaton
(reverse arrows)
Forward verification automaton
0
1
2
3
4
5
6
7
8
9
a
b
c
d
d
e
e
d
d
e
f
f
Forward scanning
Forward verification automaton
Backward verification automaton
(reverse arrows)

Figure 10: Forward and backward verification automata corresponding to forward and backward scanning of the automaton for "abcd(d|ε)(e|f)de". The shaded states are the initial ones for verification. In practice we use the complete automaton because "cutting-off" the exact verification subautomaton is too complicated.

There is, however, a complication when verifying a regular expression in this scheme. Assume the search pattern is $AXB|CYD$, for strings A, B, C, D, X and Y. The algorithm could select $X|Y$ as the scanning subpattern. Now, in the text AXD, the backward verification for $A|C$ would find A and the forward verification for $B|D$ would find D, and therefore an occurrence would be incorrectly triggered. Worse than that, it could be the case that $X = Y$, so there is no hope in distinguishing what to check based on the initial states of the scanning automaton.

The problem, which does not appear in simpler types of patterns, is that there is a *set* of initial states for the verification. The backward and forward verifications tell us that *some* state of the set can be extended to a complete occurrence, but there is no guarantee that there is a single state in that set that can be extended in both directions. To overcome this problem we check the initial states one by one, instead of using a set of initial states and doing just one verification.

7 Approximate Pattern Matching

All the previous algorithms permit specifying a flexible search pattern, but they do not allow any difference between the pattern specified and its occurrence in the text. We now consider the problem of permitting at most k insertions, deletions, replacements or transpositions in the pattern.

We let the user to specify that only a subset of the four allowed errors are permitted. However, designing one different search algorithm for each subset was impractical and against the spirit of a uniform software. So we have considered that our criterion of permitting these four types of errors would be the most commonly preferred in practice and have a unique scanning phase under this model (as all the previous algorithms, we have a scanning and a verification phase). Only at the verification phase, which is hopefully executed a few times, we take care of only applying the permitted operations. Note that we cannot miss any potential match because we scan with the most permissive error model. We also wrote in *nrgrep* specialized code for $k = 1$ and $k = 2$, which are the most common cases and using a fixed k permits better register usage.

7.1 Simple Patterns

We make use of the three techniques described in Section 3.5, choosing the one that promises to be the best.

7.1.1 Forward and Backward Searching

In forward searching, $k + 1$ similar rows corresponding to the pattern are used. There exists a bit-parallel algorithm to simulate the automaton in case of k insertions, deletions and replacements [30], but despite that the automaton that incorporates transpositions has been depicted [16] (see Figure 6), no bit parallel formula for the complete operation has been shown. We do that now.

We store each row in a machine word, $R_0 \ldots R_k$, just as for the base technique. The temporary states are stored as $T_1 \ldots T_k$. The bitmasks R_i are initialized to $0^{m-i}1^i$ as before, while all the T_i are initialized to 0^m. The update procedure to produce R' and T' upon reading text character t_j is as follows:

```
$R'_0 \longleftarrow ((R_0 << 1) \mid 0^{m-1}1) \;\; \& \;\; B[t_j]$
for $i \in 1 \ldots k$ do
    $R'_i \longleftarrow ((R_i << 1) \;\; \& \;\; B[t_j]) \mid R_{i-1} \mid (R_{i-1} << 1) \mid (R'_{i-1} << 1)$
                  $\mid (T_i \;\; \& \;\; (B[t_j] << 1))$
    $T'_i \longleftarrow (R_{i-1} << 2) \;\; \& \;\; B[t_j]$
```

The rationale of the procedure is as follows. The first three lines are as for the base technique without transpositions. The second line of the formula for R'_i corresponds to transpositions. Note that the new T' value is computed accordingly to the *old* R value, so it is 2 text positions behind. Once we compute the new bitmasks R' for text character t_j, we take those old R masks that were not updated with t_j, shift them in *two* positions (aligning them for the position t_{j+1} and killing the states that do not match t_j). This is equivalent to processing two characters: $\Sigma\ t_j$. At the next iteration (t_{j+1}), we shift left the mask $B[t_{j+1}]$ and kill the states of T that do not match. The net effect is that, at iteration $j+1$, we are *or*-ing R'_i with

$$\begin{aligned} T_i(j+1)\ \&\ (B[t_{j+1}] << 1) &= (R_{i-1}(j) << 2)\ \&\ B[t_j])\ \&\ (B[t_{j+1}] << 1) \\ &= (((R_{i-1}(j) << 1)\ \&\ B[t_{j+1}]) << 1)\ \&\ B[t_j] \end{aligned}$$

which corresponds to two forward transitions with the characters $t_{j+1}t_j$. If those characters matched the text, then we permit the activation of R'_i.

A match is detected as before, when $R_k\ \&\ 10^{m-1}$ is not zero. Since we may cut w characters from the pattern, the context conditions, and the possibility that the user really wanted to permit only some types of errors, we have to check each match found by the automaton before reporting it. We detail later the verification procedure.

Backward searching is adapted from this technique exactly as it is done from the basic algorithm without transpositions. A subtle point is that we cannot consider the automaton dead until both the R and the T masks run out of active states, since T can awake a dead R.

As before, we select the best subpattern to search for, which is of length at most w. The algorithm to select the best subpattern has to account for the fact that we are allowing errors. A good approximation is obtained by considering that any factor will be alive for k turns, and then adding the normal expected number of window characters to read until the factor does not match. So we add k to $cost[i,j]$ in Eq. (1). Now it is more possible than before (for large k/m) that the final cost per character is greater than one, in which case we prefer forward scanning.

7.1.2 Splitting into $k+1$ Subpatterns

This technique can be directly adapted from previous work, taking care of leaving a hole between each pair of pieces. For the checking of complete occurrences in candidate text areas we use the general verification engine (we have a different preprocessing for the case of each of the $k+1$ pieces matching). Note that it makes no sense to design a forward search algorithm for this case: if the average cost per character is more than one, this means that the probability of finding a subpattern is so high that we will pay too much time verifying spurious matches, in which case the whole method does not work.

The main difficulty that remains is how to find the best set of $k+1$ equal length pieces to search for. We start by computing *prob* and *pprob* as in Section 4.5. The only difference is that now we are interested only in lengths up to $L = \lfloor (m-k)/(k+1) \rfloor$, which is the maximum length of a piece (the $m-k$ comes out because there are k unused characters in the partition[2]). This cuts down the cost to compute these vectors to $O(m^2/k)$.

[2] The numerator is in fact converted into m if no transpositions are permitted. Despite that the scanning phase will allow transpositions anyway, the possible matches missed by converting the numerator to m all include transpositions, which by hypothesis are not permitted.

Now, we compute $pcost[1 \ldots m, 1 \ldots L]$, where $pcost[i, \ell]$ gives the average cost per window when searching for the factor $P_{i \ldots i+\ell-1}$. For this sake, we compute for each i value the matrix $mprob[1 \ldots L, 1 \ldots L]$, where $mprob[\ell, r]$ is the probability of any factor of length r in $P_{i \ldots i+\ell-1}$. This is computed for each i in $O(m^2/k^2)$ time as

$$\begin{array}{rrl} (\ell < r) & mprob[\ell, r] & \longleftarrow 0 \\ & mprob[\ell, 0] & \longleftarrow 1 \\ (\ell \geq r > 0) & mprob[\ell, r] & \longleftarrow 1 - (1 - pprob[i + \ell - r, r])(1 - mprob[\ell - 1, r]) \\ & pcost[i, \ell] & \longleftarrow \displaystyle\sum_{r=0}^{L} mprob[\ell, r] \end{array}$$

All this is computed for every i, for increasing ℓ, in $O(m^3/k^2)$ total time. The justification for the previous formula is similar to that in Section 4.5. Now, the most interesting part is $mbest[1 \ldots m, 1 \ldots k+1]$, where $mbest[i, s]$ is the expected cost per character of the best s pattern pieces starting at pattern position i. The strings selected have the same length. Together with $mbest$ we have $ibest[i, s]$, which tells where must the first string start in order to obtain $mbest$.

The maximum length for the pieces is L. We try all the lengths from L to 1, until we determine that even in the best case we cannot improve our current best cost. For each possible piece length ℓ we compute the whole $mbest$ and $ibest$, as follows

$$\begin{array}{rrl} & mbest[i, 0] & \longleftarrow 0 \\ (i > m - s\ell - (s-1) \wedge s > 0) & mbest[i, s] & \longleftarrow 1 \\ (i \leq m - s\ell - (s-1) \wedge s > 0) & mbest[i, s] & \longleftarrow \min(mbest[i+1, s], \\ & & \qquad 1 - (1 - cost)(1 - mbest[i + \ell + 1, s - 1])) \\ & \text{where } cost & = \min(1, pcost[i, \ell]/(\ell - pcost[i, \ell] + 1)) \end{array}$$

which is computed for decreasing i. The rationale is that $mbest[i, s]$ can choose whether to start the first piece immediately or not. The second case is easy since $mbest[i+1, s]$ is already computed, and $ibest[i, s]$ is made equal to $ibest[i+1, s]$. In the first case, we have that the cost of the first piece is $cost$ and the other $s - 1$ pieces are chosen in the best way from $P_{i+\ell+1 \ldots m}$ according to $mbest[i + \ell + 1, s - 1]$. In this case $ibest[i, s] = i$. Note that as the total cost to search for the k pieces we could take the maximum cost, but we obtained better results by using the model of the probability of the union of independent events.

All this costs in the worst case $O(m^2)$, but normally the longest pieces are the best and the cost becomes $O(mk)$. At the end we have the best length ℓ for the pieces, the expected cost per character in $mbest[1, k+1]$ and the optimal initial positions for the pieces in $i_0 = ibest[1, k+1]$, $i_1 = ibest[i_0 + \ell + 1, k]$, $i_2 = ibest[i_1 + \ell + 1, k - 1]$, and so on. Finally, if $mbest[1, k+1] \geq 1$ (actually larger than a smaller number) we know that we reach the window beginning with probability high enough and therefore the whole scheme will not work well. In this case we have to choose among forward or backward searching.

Summarizing the costs, we pay $O(m^3/k^2 + m^2)$ in the worst case and $O(m^3/k^2 + km)$ in practice. Normally k is small so the cost is close to $O(m^3)$ in all cases.

7.1.3 Verification

Finally, we explain the verification procedure. This is necessary because of the context conditions, of the possibly restricted set of edit operations, and because in some cases we are not sure that there is actually a match. Verification can be called from the forward scanning (in which case, in general, we have partitioned the pattern in $P = P_1SP_2$ and know that at a given text position a match of S ends); from the backward scanning (where we have partitioned the pattern in the same way and know that at a given text position the match of a factor of S begins); or from the partitioning into $k+1$ pieces (in which case we have partitioned $P = P_0S_1P_1 \ldots S_{k+1}P_{k+1}$ and know that at a given text position the exact occurrence of a given S_i begins).

In general, all we know about the possible match of P around text position j is that, if we partition $P = P_LP_R$ (a partition that we know), then there should be a match of P_L ending at T_j and a match of P_R starting at T_{j+1}, and that the total number of errors should not exceed k. In all the cases, we have precomputed forward automata corresponding to P_L and P_R (the first one is reversed because the verification goes backward). In forward and backward scanning there is just one choice for P_L and P_R, while for partitioning into $k+1$ pieces there are $k+1$ choices (all of them are precomputed).

We run the forward and backward verification from text character j, in both directions. Fortunately, the context conditions that make a match valid or invalid can be checked at each extreme separately. So we go backward and, among all the positions where a legal match of P_L can begin, we choose the one with minimum error level k_L. Then we go forward looking for legal occurrences of P_R with $k - k_L$ errors. If we find one, then we report an occurrence (and the whole record is reported). In fact we take advantage of each match found during the traversal: if we are looking the pattern with k errors and find a legal endpoint with $k' \leq k$, we still continue searching for better occurrences, but now allowing just $k'-1$ errors. This saves time because the verification automaton needs just to use $(k'-1)+1$ rows.

A complicated condition can arise because of transpositions: the optimal solution may involve transposing T_j with T_{j+1}, an operation that we are not permitting because we chose to split the verification there. We solve this in a rather simple but effective way: if we cannot find an occurrence in a candidate area, we give it a second chance after transposing both characters in the text.

7.2 Extended Patterns

The treatment for extended patterns is quite a combination of the extension from simple to extended patterns without errors and the use of $k+1$ rows or the partition into $k+1$ pieces in order to permit k errors. However, there are a few complications that deserve mention.

A first one is that the propagation of active states due to optional and repeatable characters does not mix well with transpositions (for example, try to draw an automaton that finds `"abc?de"` with one transposition in the text `"...adbe..."`). We solved the problem in a rather practical way. Instead of trying to simulate a complex automaton, we have two parallel automata. The R masks are used in the normal way without permitting transpositions (but permitting the other errors and the optional and repeatable characters), and they are always one character behind the current text position. Each T mask is obtained from the R mask of the previous row by processing the last two text characters in reverse order, and its result is used for the R mask of the same row in the next iteration. The code to process text position j is follows (the initial values are as always)

```
$R'_0 \longleftarrow ((R_0 << 1) \mid 0^{m-1}1) \;\&\; B[t_{j-1}]$
for $i \in 1 \ldots k$ do
    $R'_i \longleftarrow ((R_i << 1) \;\&\; B[t_{j-1}]) \mid (R_i \;\&\; S[t_{j-1}])$
                 $\mid R_{i-1} \mid (R_{i-1} << 1) \mid (R'_{i-1} << 1) \mid T_i$
    $Df \longleftarrow R'_i \mid F$
    $R'_i \longleftarrow R'_i \mid (A \;\&\; ((\sim (Df - I)) \;\hat{}\; Df))$
    $T'_i \longleftarrow ((R_{i-1} << 1) \mid 0^{m-1}1) \;\&\; B[t_j] \mid (R_{i-1} \;\&\; S[t_j])$
    $Df \longleftarrow T'_i \mid F$
    $T'_i \longleftarrow T'_i \mid (A \;\&\; ((\sim (Df - I)) \;\hat{}\; Df))$
    $T'_i \longleftarrow ((T'_i << 1) \mid 0^{m-1}1) \;\&\; B[t_{j-1}] \mid (T'_i \;\&\; S[t_{j-1}])$
```

where the A, I and F masks are defined in Section 5.

A second complication is that subpattern optimization changes. For the forward and backward automata we use the same technique for searching without errors but we add k to the number of processed window positions for any subpattern. So it remains to be explained how is the optimization for the partitioning into $k+1$ subpatterns.

We have *prob* and *sprob* computed in $O(m)$ time as for normal extended patterns. A first condition is that the $k+1$ pieces must be disjoint, so we compute $reach[1 \ldots m, 1 \ldots L']$, where $L' = \lfloor (L-k)/(k+1) \rfloor$ is the maximum length of a piece as before (L is the minimum length of a pattern occurrence as in Section 5) and $reach[i, \ell]$ gives the last pattern position reachable in ℓ text characters from i. This is computed in $O(m^2)$ time as $reach[i, 0] = i$ and, for increasing $\ell > 0$,

```
$t \longleftarrow reach[i, \ell - 1] + 1$
while ($t < m \;\wedge\; (A \;\&\; 0^{m-t}10^{t-1} \neq 0^m)$)    $t \longleftarrow t + 1$
$reach[i, \ell] \longleftarrow t$
```

We then compute *pprob*, *mprob* and *pcost* exactly as in Section 5.5, in $O(m^3)$ time. Finally, the selection of the best set of $k+1$ subpatterns is done with *mbest* and *ibest* just as in Section 7.1.2, the only difference being that *reach* is used to determine which is the first unused position if pattern position i is chosen and a minimum piece length ℓ has to be reserved for it. The total process takes $O(m^3)$ time.

7.3 Regular Expressions

Finally, *nrgrep* permits searching for regular expressions allowing errors. The same mechanisms used for simple and extended patterns are used, namely using $k+1$ replicas of the search automaton and splitting the pattern into $k+1$ disjoint pieces. Both adaptations present important complications with respect to their simpler counterparts.

7.3.1 Forward and Backward Search

One problem in regular expressions is that the concept of "forward" does not immediately mean one shift to the right. Approximate searching with $k+1$ copies of the NFA of the regular expression implies being able to move "forward" from row i to row $i+1$, as Figure 11 shows.

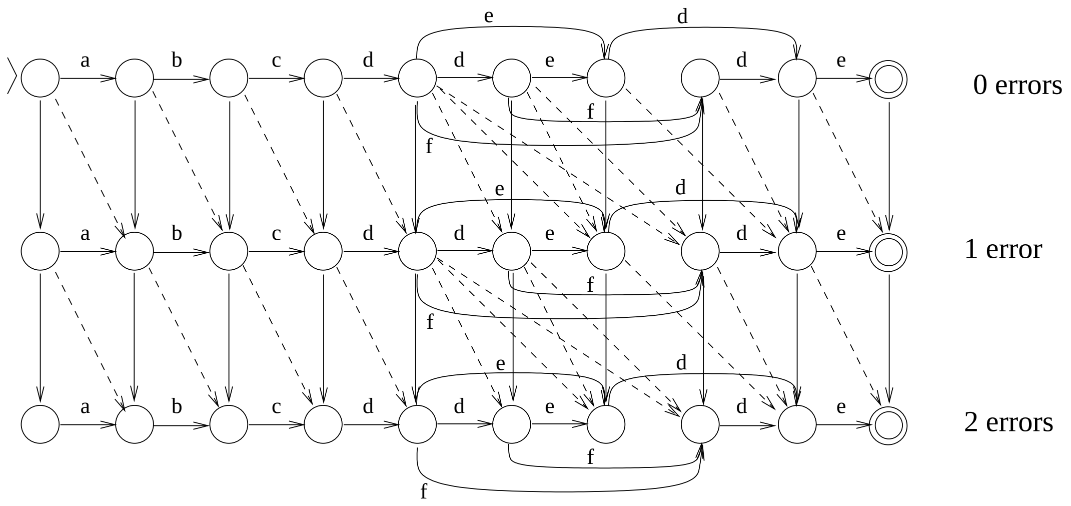

Figure 11: Glushkov's resulting NFAs for the search of the regular expression `"abcd(d|`ε`)(e|f)de"` with two insertions, deletions or replacements. To simplify the plot, the dashed lines represent deletions and replacements (i.e. they move by $\Sigma \cup \{\varepsilon\}$), while the vertical lines represent insertions (i.e. they move by Σ).

For this sake, we use our table T, which for each state D[3] gives the bit mask of all the states reachable from D in one step. The update formula upon reading a new text character t_j is therefore

```
$R'_0 \longleftarrow T[R_0]$ & $B[t_j]$
$oldR_0 \longleftarrow R_0$
for $i \in 1 \ldots k$ do
    $R'_i \longleftarrow (T[R_i]$ & $B[t_j]) \mid R_{i-1} \mid T[R_{i-1} \mid R'_{i-1}] \mid (T[T[oldR_{i-1}]$ & $B[t_j]]$ & $B[t_{j-1}])$
    $oldR_{i-1} \longleftarrow R_{i-1}$
```

The rationale is as follows. R'_0 is computed according to the simple formula for regular expression searching without errors. For $i > 0$, R'_i permits arrows coming from matching the current character ($T[R_i]$ & $B[t_j]$), "vertical" arrows representing insertions in the pattern (R_{i-1}) and "diagonal" arrows representing replacements (from R_{i-1}) and deletions (from R'_{i-1}), which are joined in $T[R_{i-1}|R'_{i-1}]$. Finally, transpositions are arranged in a rather simple way. In $oldR$ we store the value of R two positions in the past (note the way we update it to avoid having two arrays, $oldR$ and $oldoldR$), and each time we permit from that state the processing of the last two text character in reverse order.

Forward scanning, backward scanning, and the verification of occurrences are carried out in the normal way using this update technique. The technique to select the best necessary factor is unaltered except that we add k to the number of characters that are scanned inside every text window.

[3] Recall that in the deterministic simulation, each bit mask of active NFA states D is identified as a state.

7.3.2 Splitting into $k+1$ Subexpressions

If we can select $k+1$ *necessary* factors of the regular expression as done in Section 6.3 for selecting one necessary factor, then we can be sure that at least one of them will appear unaltered in any occurrence with k errors or less. As before, in order to include the transpositions we need to ensure that one character is left between consecutive necessary factors.

We first consider how, once the subexpressions have been selected, can we perform the multipattern search. Each subexpression has a set of initial and final states. We reverse all the arrows and convert the formerly initial states of all the subexpressions into final states. Since we search for any reverse factor of the regular expression, all the states are made initial.

We set the window length to the shortest path from an initial to a final state. At each window position, we read the text characters backward and feed the transformed automaton. Each time we arrive to a final state we know that the prefix of a necessary subexpression has appeared. If we happen to be at the beginning of the window then we check for the whole pattern as before. We keep masks with the final states of each necessary factor in order to determine, from the current mask of active states, which of them matched (recall that a different verification is triggered for each).

Figure 12 illustrates a possible selection of two necessary factors (corresponding to $k=1$) for our running example "abcd(d|ε)(e|f)de". These are "abc" and "(d|ε)(e|f)de", where the first "d" has been left to separate both necessary factors. Their minimum length is 3, so this is the window length to use.

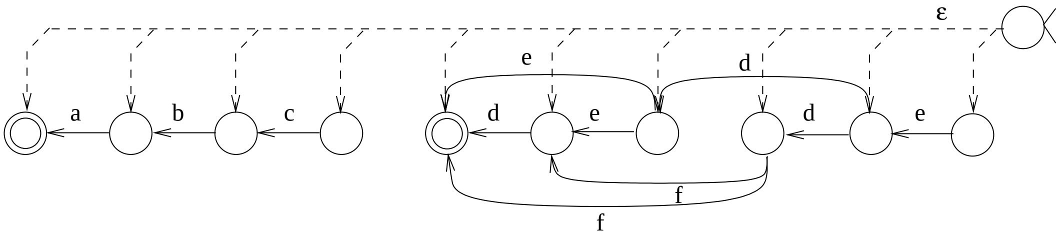

Figure 12: An automaton recognizing reverse prefixes of two necessary factors of "abcd(d|ε)(e|f)de", based on the Glushkov construction of Figure 5. Note that there are no transitions among subexpressions.

The difficult part is how to select $k+1$ necessary *and disjoint* factors from a regular expression. Disjoint means that the subautomata do not share states (this ensures that there is a character separating them, which is necessary for transpositions). Moreover, we want the best set, and we want that all them have the same minimum path length from an initial to a final state.

We believe that an algorithm finding the optimal choice has a time cost which grows exponentially with k. We have therefore made the following simplification. Each node s is assigned a number I_s which corresponds to the length of the shortest path reaching it from the initial state. We do not permit picking arbitrary necessary factors, but only those subautomata formed by all the states s that have numbers $i \leq I_s \leq i+\ell$, for some i and ℓ. This ℓ is the minimum window length to search using that subautomata, and should be the same for all the necessary factors chosen. Moreover, every arrow leaving out of the chosen subautomaton is dropped. Finally, note that if the i values corresponding to any pair of subautomata chosen differ by more than ℓ then we are sure that they are disjoint.

Figure 13 illustrates this numbering for our running example. The partition obtained in Figure 12 corresponds to choosing the ranges $[0,3]$ and $[4,7]$ as the sets of states (this corresponds to the sets $\{0,1,2,3\}$ and $\{4,5,6,7,8,9\}$). Of course the method does not permit us to choose all the legal sets of subautomata. In this example, the necessary subautomaton $\{6,7,8\}$ cannot be picked because it includes some, but not all, states numbered 5.

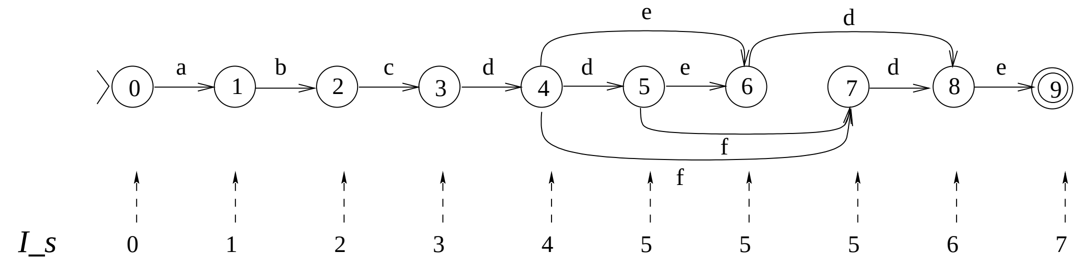

Figure 13: Numbering Glushkov's NFA states for the regular expression `"abcd(d|ε)(e|f)de"`.

Numbering the states is easily done by a closure process starting from the initial state in $O(Lm^2/w)$ time, where L is the minimum length of a string matching the whole regular expression. As before, $L' = \lfloor (L-k)/(k+1) \rfloor$ is the maximum possible length of a string matching a necessary factor.

To determine the best factors, we first compute *prob*, *pprob*, *mprob* and *cost* exactly as in Section 6.3. The only difference is that now we are only interested in starting from sets of initial states of the same I_s number (there are L possible choices) and we are interested in analyzing factors of length no more than $L' = O(L/k)$. This lowers the costs to compute the previous arrays to $O((L'm)^2) = O((Lm/k)^2)$. We also make sure that we do not select subautomata that need more than w bits to be represented.

Once we have the expected cost of choosing any possible I_s value and any possible minimum factor length ℓ, we can apply the optimization algorithm of Section 7.1.2, since we have in fact "linearized" the problem: thanks to our simplification, the optimization problem is similar to when we had a single pattern and needed to extract the best set of $k+1$ disjoint factors from it, of a single length ℓ. This takes $O(L^2)$ in the worst case but should in practice be closer to $O(kL)$.

Hence the total optimization algorithm needs $O(m^4)$ time at worst.

8 A Pattern Matching Software

Finally, we are in position to describe the software *nrgrep*. *Nrgrep* has been developed in ANSI C and tested on Linux and Solaris platforms. Its source code is publicly available under a *Gnu* license [4]. We discuss now its main aspects.

[4] From `http://www.dcc.uchile.cl/~gnavarro/pubcode/`.

8.1 Usage and Options

Nrgrep receives in the command line a pattern and a list of files to search. The syntax of the pattern is given in Section 2. If no options are given, *nrgrep* searches the files in the order given and prints all the lines where it finds the pattern. If more than one file is given then the file name is printed prior to the lines that have matched inside it, if there is one. If no file names are given, it receives the text from the standard input.

The default behavior of *nrgrep* can be modified with a set of possible options, which prefixed by the minus sign can precede the pattern specification. Most of them are inspired in *agrep*:

`i`: the search is case insensitive;

`w`: only whole words matching the pattern are accepted;

`x`: only whole records (e.g. lines) matching the pattern are accepted;

`c`: just counts the matches, does not print them;

`l`: outputs the names of the files containing matches, but not the matches themselves;

`G`: outputs the whole contents of the files containing matches;

`h`: does not output file names;

`n`: prints records preceded by their record number;

`v`: reverse matching, reports the records that do *not* match.

`d` $< delim >$: sets the record delimiter to $< delim >$, which is `"\n#"` by default. A `"#"` at the end of the delimiter makes it appear as a part of the previous record (default is next), so by default the end of line is the record delimiter and is considered as a part of the previous record;

`b` $< bufsize >$: sets the buffer size, default is 64 Kb. This affects the efficiency and the possibility of cutting very long records that do not fit in a buffer;

`s` $< sep >$: prints the string $< sep >$ between each pair of records output;

`k` $< err >$ [`idst`]: allows up to $< err >$ errors in the matches. If `idst` is not present the errors permitted are (i)nsertions, (d)eletions, (s)ubstitutions and (t)ranspositions, otherwise a subset of them can be specified, e.g. `"-k 3ids"`;

`L`: takes the pattern literally (no special characters);

`H`: explains the usage and exits.

We now discuss briefly how each of these options are implemented: `-i` is easily carried out by using classes of characters; `-w` and `-x` are handled using context conditions; `-c`, `-l`, `-G`, `-h`, `-b` and `-s` are easily handled, but some changes are necessary such as stopping the search when the first match is found for `-l` and `-G`; `-n` and `-v` are more complicated because they force all the records to be processed, so we first search for the record delimiters and then search for the pattern inside the records (normally we search for the pattern and find record delimiters around occurrences only); `-d` is arranged by just changing the record delimiter (it has to be a simple pattern, but classes or characters are permitted); `-k` switches to approximate searching and the `[idst]` flags are considered at verification time only; and `-L` avoids the normal parsing process and considers the pattern as a simple string.

Of course it is permitted to combine the options in a single string preceded by the minus sign or as a sequence of strings, each preceded by the minus sign. Some combinations, however, make no sense and are automatically overriden (a warning message is issued): filenames are not printed if the text comes by the standard input; `-c` and `-G` are not compatible and `-c` dominates; `-n` and `-l` are not compatible and `-l` dominates; `-l` and `-G` are not compatible and `-G` dominates; and `-G` is ignored when working on the standard input (since `-G` works by printing the whole file from the shell).

8.2 Parsing the Pattern

One important aspect that we have not discussed is how is the parsing done. In principle we just have to parse a regular expression. However, our parser module carries out some other tasks, such as

Parsing our extended syntax: some of our operations, such as "?" and classes of characters, are not part of the classical regular expression syntax. The result of the parsing is a syntax tree which is not discarded, since as we have seen it is useful later for preprocessing regular expressions.

Map the pattern to bit mask positions: the parser determines the number of states required by the pattern and assigns the bit positions corresponding to each part of the pattern specification.

Determine type of subexpression: given the whole pattern or a subset of its states, the parser is able to determine which type of expression is involved (simple, extended, or regular expression).

Algebraic simplification: the parser performs some algebraic simplification on the pattern to optimize the search and reduce the number of bits needed for it.

The parsing phase operates in a top-down way. It first tries to parse an *or* ("|") of subexpressions. Each of them is parsed as a concatenation of subexpressions, each of which is a "single expression" finished with a sequence of "*", "?" or "+" terminators, and the single expression is either a single symbol, a class of characters or a top-level expression in parentheses. Apart from the syntax tree, the parser produces a mapping from positions of the pattern strings to leaves of the syntax tree, meaning that the character or class described at that pattern position must be loaded at that tree leaf.

The second step is the algebraic simplification, which proceeds bottom-up and is able to enforce the following rules at any level

1. If $[C_1]$ and $[C_2]$ are classes of characters then $[C_1] \mid [C_2] \longrightarrow [C_1C_2]$.

2. If $[C]$ is a class, then $[C] \mid \varepsilon,\ \varepsilon \mid [C] \longrightarrow [C]?$.

3. If E is any subexpression, then $E \cdot \varepsilon,\ \varepsilon \cdot E \longrightarrow E$.

4. If E is any subexpression, then $E**$, $E?*$, $E+*$, $E*?$, $E+?$, $E**+$, $E?+$ $\longrightarrow$ $E*$, and $E??$ $\longrightarrow$ $E?$, $E++$ $\longrightarrow$ $E+$.

5. $\varepsilon \mid \varepsilon, \varepsilon*,\ \varepsilon?,\ \varepsilon+ \ \longrightarrow\ \varepsilon$.

6. A subexpression which can match the empty string and appears at the beginning or at the end of the pattern is replaced by ε, except when we need to match whole words or records.

To collapse classes of characters in a single node (first rule) we traverse the obtained mapping from pattern positions to tree leaves, find those mapping to the leaves that store $[C_1]$ and $[C_2]$ and make all them point to the new leaf that represents $[C_1C_2]$.

More simplifications are possible, but they are more complicated and we chose to stop here for the current version of *nrgrep*. Some interesting simplifications that may reduce the number of bits required to represent the pattern are $xwz|uwv \longrightarrow (x|u)w(z|v)$, $EE* \longrightarrow E+$ and $E|E \longrightarrow E$ for arbitrarily complex E (right now we do that just for leaves).

After the simplification is done we assign positions in the bit mask corresponding to each character/class in the pattern. This is easily done by traversing the leaves left to right and assigning one bit to each leaf of the tree except to those storing ε instead of a class of characters. Note that this works as expected on simple and extended patterns as well. For several purposes (including Glushkov's NFA construction algorithm) we need to store which are the bit positions used inside every subtree, which of these correspond to initial and final states, and whether the empty string matches the subexpression. All this is easily done in the same recursive traversal over the tree.

Once we have determined the bits that correspond to each tree leaf and the mapping from pattern positions to tree leaves, we build a table of bit masks B, such that $B[c]$ tells which pattern positions are activated by the character c. This B table can be directly used for simple and extended patterns, and it is the basis to build the DFA transitions of regular expressions.

Finally, the parser is able to tell which is the type of the pattern that it has processed. This can be different from what the syntax used to express the pattern may suggest, for example "a(b|c)dε*e(f|g)" is the simple pattern "a[bc]de[fg]", which is discovered by the simplification procedure. This is in our general spirit of not being mislead by simple problems presented in a complicated way, which motivated us to select optimum subpatterns to search for. As a secondary effect of simplifications, the number of bits required to represent the pattern may be reduced.

A second reason to be able to determine the real type of pattern is that the scanning subpattern selected can be simpler than the whole pattern and hence it can admit a faster search algorithm. For this sake we permit determining the type not only of the whole pattern but also of a selected set of positions.

The algorithm for determining the type of pattern or subpattern starts by assuming a simple pattern until it finds evidence of an extended pattern or a regular expression. The algorithm enters recursively into the syntax tree of the pattern avoding entering subexpressions where no selected state is involved. Among those that have to be entered in order to reach the selected states, it answers "simple" for the leaves of the tree (characters, classes and ε), and in concatenations it takes the most complex type among the two sides. When reaching an internal node "?", "*" or "+" it assumes "extended" if the subtree was just a single character, otherwise it assumes "regular expression". The latter is always assumed when an *or* ("|") that could not be simplified is found.

8.3 Software Structure

Nrgrep is implemented as a set of modules, which permits easy enrichment with new types of patterns. Each type of pattern (simple, extended and regular expression) has two modules to deal with the exact and the approximate case, which makes six modules: `simple`, `esimple`, `extended`, `eextended`, `regular`, `eregular` (the prefix "`e`" stands for allowing (e)rrors). The other modules are:

`basics,options,except,memio`: basic definitions, exception handling and memory and I/O management functions.

`bitmasks`: handles operations on simple and multi-word bit masks.

`buffer`: implements the buffering mechanism to read files.

`record`: takes care of context conditions and record management.

`parser`: performs the parsing.

`search`: template that exports the preprocessing and search functions which are implemented in the six modules described (`simple`, etc).

`shell`: interacts with the user and provides the main program.

The template `search` facilitates the management of different search algorithms for different pattern types. Moreover, we remind that the scanning procedure for a pattern of a given type can use a subpattern whose type is simpler, and hence a different scanning function is used. This is easily handled with the template.

9 Some Experimental Results

We present now some experiments comparing the performance of *nrgrep* version 1.1 against that of its best known competitors, *Gnu grep* version 2.4 and *agrep* version 3.0.

Agrep [29] uses for simple strings a very efficient modification of the Boyer-Moore-Horspool algorithm [12]. For classes of characters and wild cards (denoted "`#`" and equivalent to "`.*`") it uses an extension of the Shift-Or algorithm explained in Section 3.1, in all cases based on forward scanning. For regular expressions it uses a forward scanning with the bit parallel simulation of Thompson's automaton, as explained in Section 3.4. Finally, for approximate searching of simple strings it tries to use partitioning into $k+1$ pieces and a multipattern search algorithm based on Boyer-Moore, but if this does not promise to yield good results it prefers a bit-parallel forward scanning with the automaton of $k+1$ rows (these techniques were explained in Section 3.5). For approximate searching of more complex patterns *agrep* uses only this last technique.

Gnu Grep cannot search for approximate patterns, but it permits all the extended patterns and regular expressions. Simple strings are searched for with a Boyer-Moore-Gosper search algorithm (similar to Horspool). All the other complex patterns are searched for with a lazy deterministic automaton, i.e. a DFA whose states are built as needed (using forward scanning). To speed up the search for complex patterns, *grep* tries to extract their longest necessary string, which is used as a filter and searched for as a simple string. In fact, *grep* is able to extract a necessary *set* of strings, i.e. such that one of the strings in the set has to appear in every match. This set is searched for as a filter using a Commentz-Walter like algorithm [7], which is a kind of multipattern Boyer-Moore. As we will see, this extension makes *grep* very powerful and closer to our goal of a smooth degradation in efficiency as the pattern gets more complex.

The experiments were carried out over 100 Mb of English text extracted from Wall Street Journal articles of 1987, which are part of the TREC collection [11]. Two different machines were used: SUN is a Sun UltraSparc-1 of 167 MHz with 64 Mb RAM running Solaris 2.6, and INTEL is an i686 of 550 MHz with 64 Mb RAM running Linux Red Hat 6.2 (kernel 2.2.14-5.0). Both are 32-bit machines ($w = 32$).

To illustrate the complexity of the code, we show the sizes of the sources and executables in Table 1. The source size is obtained by summing up the sizes of all the `".c"` and `".h"` files. The size of the executables is computed after running Unix's `"strip"` command on them. As can be seen, *nrgrep* is in general a simpler software.

Software	Source size	Executable size (SUN)	Executable size (INTEL)
agrep v. 3.0	412.49 Kb	152.79 Kb	136.26 Kb
grep v. 2.4	472.65 Kb	80.91 Kb	73.83 Kb
nrgrep v 1.1	281.52 Kb	92.50 Kb	89.59 Kb

Table 1: Sizes of the different softwares under comparison.

The experiments were repeated for 100 different patterns of each kind. To minimize the interference of I/O times in our measures, we had the text in a local disk, we considered only user times and we asked the programs to show the count of matching records only. The records were the lines of the file. The same patterns were searched for on the same text for *grep*, *agrep* and *nrgrep*.

Our first experiment shows the search times for simple strings of lengths 5 to 30. Those strings were randomly selected from the same text starting at word beginnings and taking care of including only letters, digits and spaces. We also tested how the `"-i"` (case insensitive search) and `"-w"` (match whole words) affected the performance, but the differences were negligible. We also made this experiment allowing 10% and 20% of errors (i.e. $k = \lfloor 0.1m \rfloor$ or $k = \lfloor 0.2m \rfloor$). In the case of errors *grep* is excluded from the comparison because it does not permit approximate searching.

As Figure 14 shows, *nrgrep* is competitive against the others, more or less depending on the machine and the pattern length. It works better on the INTEL architecture and on moderate length patterns rather than on very short ones. It is interesting to notice that in our experiments *grep* performed better than *agrep*.

When searching allowing errors, *nrgrep* is slower or faster than *agrep* depending on the case. With low error levels (10%) they are quite close, except for $m = 20$, where for some reason *agrep* performs consistently bad on SUN. With moderate error levels (20%) the picture is more complicated and each of them is better for different pattern lengths. This is a point of very volatile behavior because 20% happens to be very close to the limit where splitting into $k + 1$ pieces ceases to be a good choice, and a small error estimating the probability of a match produces dramatic changes in the performance. Depending on each pattern, a different search technique is used.

On the other hand, the search permitting transpositions is consistently worse than the one not permitting them. This is not only because when splitting into $k+1$ pieces they may have to be shorter to allow one free space among them, but also because even when they can have the same length the subpattern selection process has more options to find the best pieces if no space has to be left among them.

The behavior of *nrgrep*, however, is not as erratic as it seems to be. The non-monotonic behavior can be explained in terms of splitting into $k+1$ pieces. As m grows, the length of the pieces that can be searched for grows, tending to the real $m/(k+1)$ value from below. For example, for $k =$ 20% of m, we have in the case of transpositions to search for 2 pieces of length 2 when $m = 5$. For $m = 10$ we have to search for 3 pieces of length 2, which is estimated to produce too many verifications and then another method is used. For $m = 15$, however, we have to search for 4 pieces of length 3, which has much lower probability of occurrence and recommends again splitting into $k+1$ pieces. The differences between searching using or not transpositions is explained in the same way. For $m = 5$, we have to search for 2 pieces of length 2 no matter whether transpositions are permitted. But for $m = 10$ we can search for 3 pieces of length 3 if no transpositions are permitted, which yields much better search time for that case. We remark that permitting transpositions has the potential of yielding approximate searching of better quality, and hence obtain the same (or better) results using a smaller k.

Our second experiment aims at evaluating the performance when searching for simple patterns that include classes of characters. We have selected strings as before from the text and have replaced some random positions with a class of characters. The classes have been: upper and lower case versions of the letter replaced (called "case" in the experiments), all the letters (`[a-zA-Z]`, called "letters" in the experiment) and all the characters (`"."`, called "all" in the experiments). We have considered pattern lengths of 10 and 20 and an increasing number of positions converted into classes. We show also the case of length 15 and $k = 2$ errors (excluding *grep*).

As Figure 15 shows, *nrgrep* deals much more efficiently with classes of characters than its competitors, worsening slowly as there are more or bigger classes to consider. *Agrep* yields always the same search time (coming from a Shift-Or like algorithm). *Grep*, on the other hand, worsens progressively as the number of classes grows but is independent on how big the classes are, and it worsens much faster than *nrgrep*. We have included the case of zero classes basically to show the effect of the change of *agrep*'s search algorithm.

Our third experiment deals with extended patterns. We have selected strings as before from the text and have added an increasing number of operators `"?"`, `"*"` or `"+"` to it, at random positions (avoiding the first and last ones). For this sake we had to use the `"-E"` option of *grep*, which deals with regular expressions, and convert $E+$ into $EE*$ for *agrep*. Moreover, we found no way to express the `"?"` in *agrep*, since even allowing regular expressions, it did not permit specifying the ε string or the empty set ($a? = (a|\varepsilon) = (a|\emptyset*)$). We also show how *agrep* and *nrgrep* perform to search for extended patterns allowing errors. Because of *agrep*'s limitations, we chose $m = 8$ and $k = 1$ errors (no transpositions permitted) for this test.

As Figure 16 shows, *agrep* has a constant search time coming again from Shift-Or like searching (plus some size limitations on the pattern). Both *grep* and *nrgrep* improve as the problem becomes

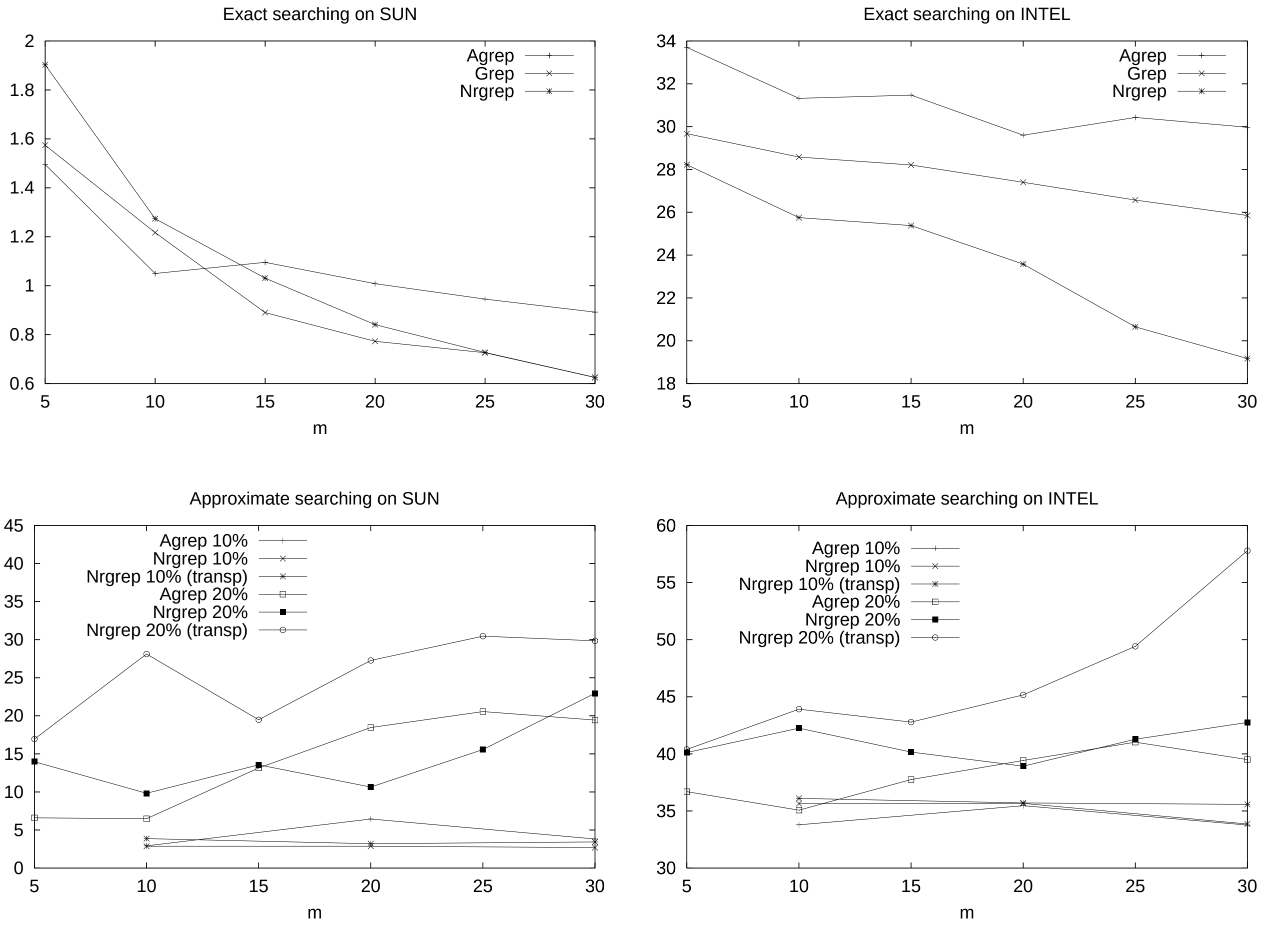

Figure 14: Search times on 100 Mb for simple strings using exact and approximate searching.

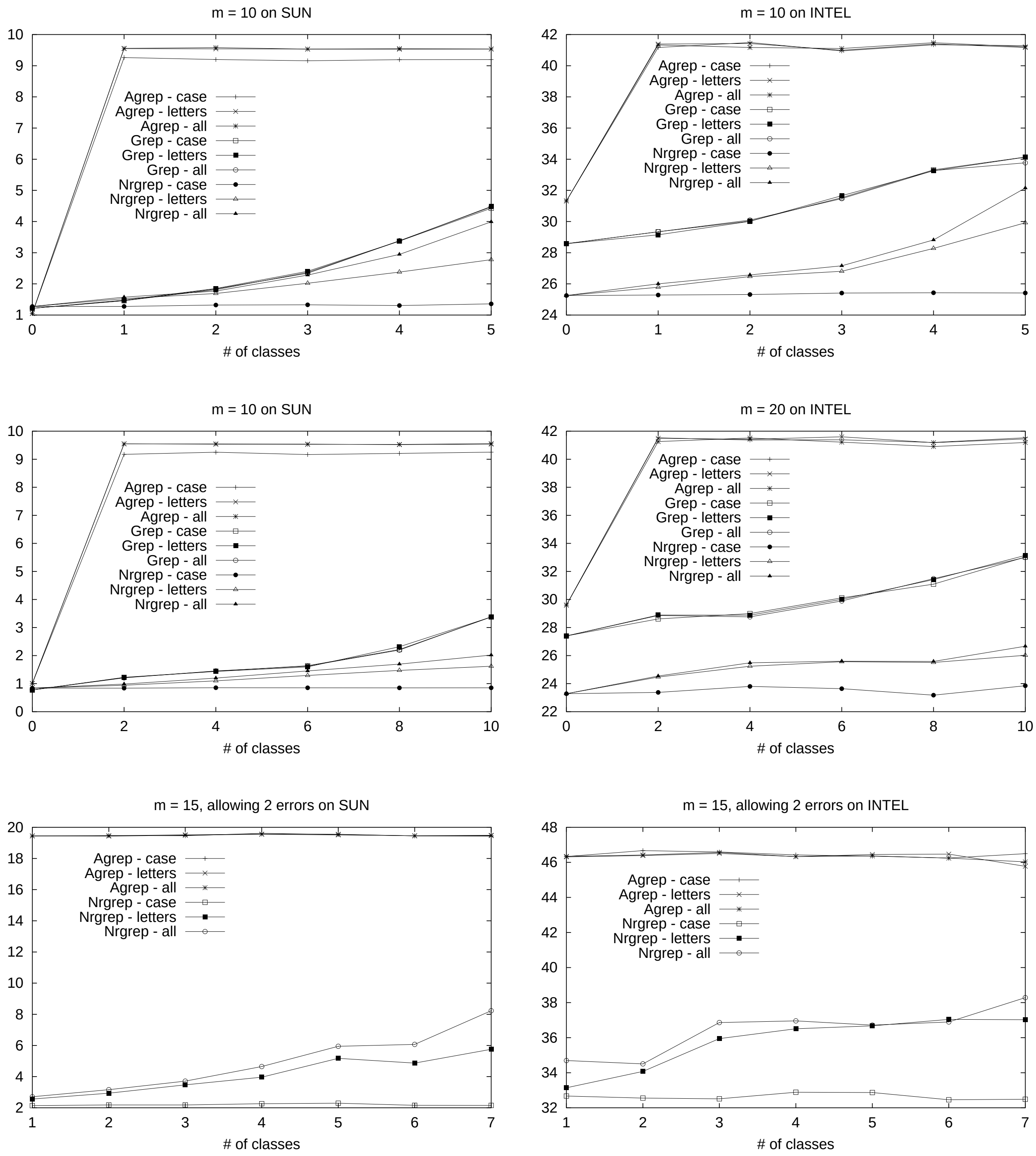

Figure 15: Exact and approximate search times on 100 Mb for simple patterns with a varying number of classes of characters.

simpler, but *nrgrep* is consistently faster. Note that the problem is necessarily easier with the "+" operator because the minimum length of the pattern is not reduced as the number of operators grows. We have again included the case of zero operators to show how *agrep* jumps.

With respect to approximate searching, *nrgrep* is consistently faster than *agrep*, whose cost is a kind of worst case which is reached when the minimum pattern length becomes 4. This is indeed a very difficult case for approximate searching and *nrgrep* wisely chooses to do forward scanning on that case.

Our fourth experiment considers regular expressions. It is not easy to define what is a "random" regular expression, so we have tested 9 complex patterns that we considered interesting to illustrate the different alternatives for the efficiency. These have been searched for with zero, one and two errors (no transpositions permitted). Table 2 shows the patterns selected and some aspects that explain the efficiency problems to search for them.

No.	Pattern	Size	Minimum	% of lines that match		
		(# chars)	length ℓ	exactly	1 error	2 errors
1	`American|Canadian`	16	8	1.245	1.561	1.872
2	`American|Canadian|Mexican`	23	7	1.288	1.604	2.134
3	`Amer[a-z]*can`	8	7	0.990	1.309	1.693
4	`Amer[a-z]*can|Can[a-z]*ian`	15	6	1.245	1.731	8.733
5	`Ame(i|(r|i)*)can`	9	6	0.990	1.312	2.262
6	`Am[a-z]*ri[a-z]*an`	8	6	0.991	1.422	3.756
7	`(Am|Ca)(er|na)(ic|di)an`	14	8	1.245	1.561	1.905
8	`American#*policy`	15	14	0.002	0.003	0.008
9	`A(mer|i)+can#*p(oli|cy)`	15	8	0.007	0.013	0.164

Table 2: The regular expressions searched for (written with the syntax of *nrgrep*). Note that some are indeed extended patterns.

Table 3 shows the results. These are more complex to interpret than in the previous cases, and there are important differences in the behavior of the same code depending on the machines.

For example, *grep* performed consistently well on INTEL, while it showed wide differences on SUN. We believe that this comes from the fact that in some cases *grep* cannot find a suitable set of filtering strings (patterns 1, 2, 4 and 7). In those cases the time corresponds to that of a forward DFA. On the INTEL machine, however, the search times are always good.

Another example is *agrep*, which has basically two different times on SUN and always the same time on INTEL. Despite that *agrep* uses always forward scanning with a bit-parallel automaton, it takes on the SUN half the time when the pattern is very short (up to 12 positions), while it is slower for longer patterns. This difference seems to come from the number of computer words needed for the simulation, but this seems not to be important on the INTEL machine. The approximate search using *agrep* (which works only on the shorter patterns) scales in time accordingly to the number of computer words used, although there is a large constant factor added in the INTEL machine.

Nrgrep performs well on exact searching. All the patterns yield fast search times, in general better than those of *grep* on INTEL and worse on SUN. The exception is where *grep* does not use filtering in the SUN machine, and *nrgrep* becomes much faster. When errors are permitted the search times of *nrgrep* vary depending on its ability to filter the search. The main aspects that

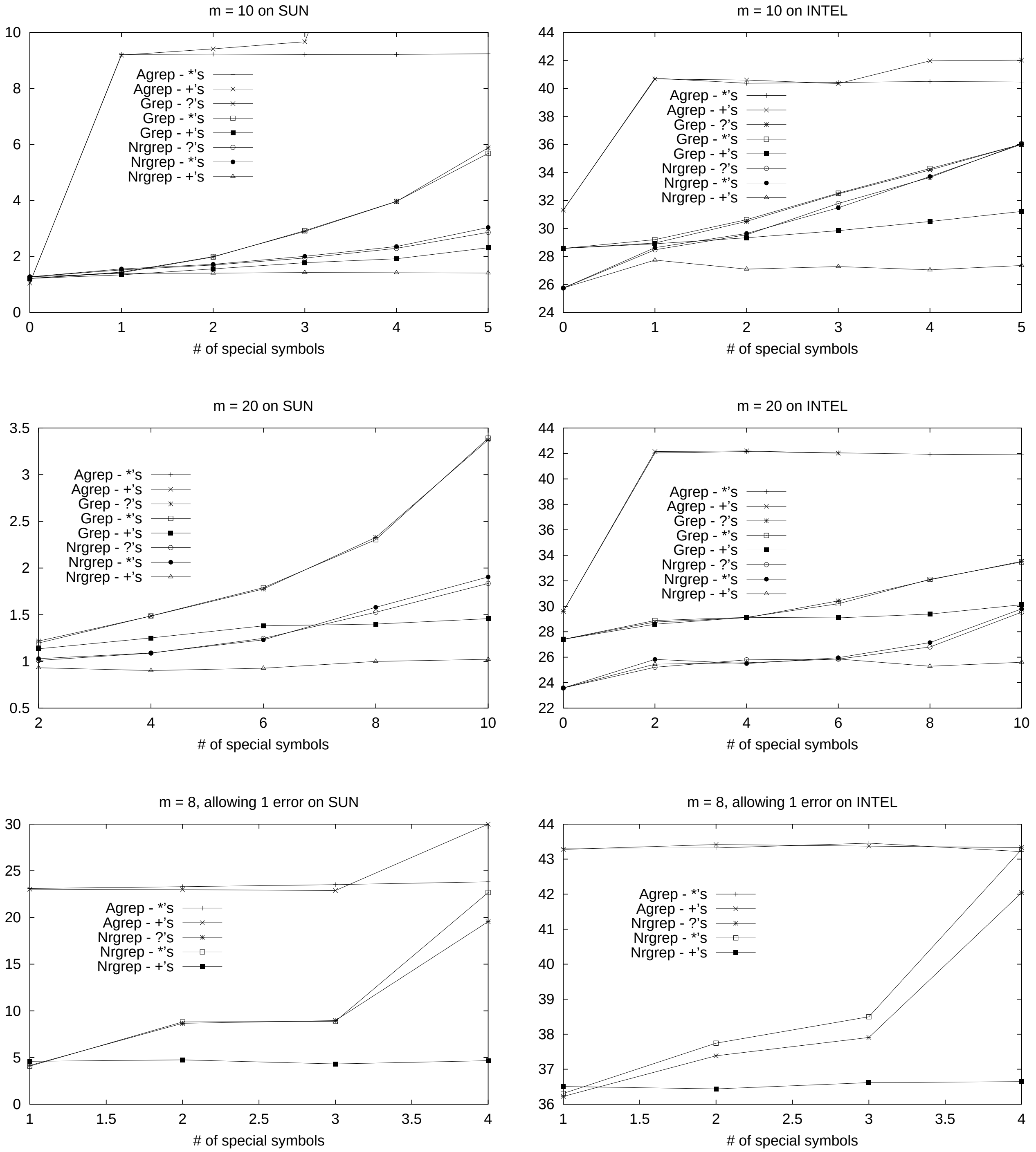

Figure 16: Exact and approximate search times on 100 Mb for extended patterns with a varying number of operators. Where, on SUN, *Agrep* is out of bounds, it takes nearly 18 seconds.

affect the search time are the frequency of the pattern and its size. When compared to *agrep*, *nrgrep* performs always better on exact searching and better or similarly on approximate searching.

SUN							
No.	*grep*	*agrep*			*nrgrep*		
	0 errors	0 errors	1 error	2 errors	0 errors	1 error	2 errors
1	8.13	18.46			2.31	4.78	52.40
2	8.14	18.12			2.41	12.34	58.81
3	1.57	9.27	23.64	32.95	2.42	3.49	33.06
4	7.74	18.07			3.84	13.28	29.42
5	2.01	9.46	22.93	32.75	2.41	10.08	18.83
6	2.39	9.41	23.13	33.03	3.18	25.03	31.30
7	9.00	18.23			2.04	3.60	18.99
8	1.69	18.39			1.76	3.33	5.24
9	2.84	18.54			3.04	8.27	18.16

INTEL							
No.	*grep*	*agrep*			*nrgrep*		
	0 errors	0 errors	1 error	2 errors	0 errors	1 error	2 errors
1	34.87	42.43			25.16	38.33	55.82
2	34.41	42.46			28.78	42.38	58.18
3	30.86	40.12	43.34	44.86	22.90	32.48	48.04
4	35.61	41.81			33.72	45.06	50.52
5	33.81	40.34	43.43	44.78	26.01	39.78	43.84
6	34.95	39.96	43.31	45.88	27.29	43.28	58.21
7	35.27	41.13			23.87	36.51	44.23
8	28.70	42.10			23.33	32.53	35.81
9	34.18	41.59			27.61	39.44	43.42

Table 3: Search times for the selected regular expressions. There are empty cells because *agrep* severely restricts the lengths of the complex patterns that can be approximately searched for.

The general conclusions from the experiments are that *nrgrep* is, for exact searching, competitive against *agrep* and *grep*, while it is in general superior (sometimes by far) when searching for classes of characters and extended patterns, exactly or allowing errors. When it comes to search for regular expressions, *nrgrep* is in general, but not always, faster than *grep* and *agrep*.

One final word about *nrgrep*'s smoothness is worthwhile. The reader may get the impression that *nrgrep*'s behavior is not as smooth as promised because it takes very different times for different patterns, for example on regular expressions, while *agrep*'s behavior is much more predictable. The point is that some of these patterns are indeed much simpler than others, and *nrgrep* is much faster to search for the simpler ones. *Agrep*, on the other hand, does not distinguish between simple and complicated cases. *Grep* does a much better job but it does not deal with the complex area of approximate searching. Something similar happens on other cases: *nrgrep* had the highest variance when searching for patterns where classes were inserted at random points. This is because this random process does produce a high variance in the complexity of the patterns: it is much simpler to search for the pattern when all the classes are in one extreme (then cutting it out from the scanning subpattern) than when they are uniformly spread. *Nrgrep*'s behavior simply mimics the variance in its input, precisely because it takes time proportional to the real complexity of the search problem.

10 Conclusions

We have presented *nrgrep*, a fast online pattern matching tool especially well suited for complex pattern searching on natural language text. *Nrgrep* is now at version 1.1, and publicly available under a *Gnu* license. Our belief is that it can be a successful new member of the *grep* family. The *Free Software Foundation* has shown interest in making *nrgrep* an important part of a new release of *Gnu grep*, and we are currently defining de details.

The most important improvements of *nrgrep* over the other members of the *grep* family are:

Efficiency: *nrgrep* is similar to the others when searching for simple strings (a sequence of single characters) and some regular expressions, and generally much faster for all the other types of patterns.

Uniformity: our search model is uniform, based on a single concept. This translates into smooth variations in the search time as the pattern gets more complex, and into an absence of obscure restrictions present in *agrep*.

Extended patterns: we introduce a class of patterns which is intermediate between simple patterns and regular expressions and develop efficient search algorithms for it.

Error model: we include character transpositions in the error model.

Optimization: we find the optimal subpattern to scan the text and check the potential occurrences for the complete pattern.

Some possible extensions and improvements have been left for future work. The first one is a better computation of the matching probability, which has a direct impact on the ability of *nrgrep* for choosing the right search method (currently it normally succeeds, but not always). A second one is a better algebraic optimization of the regular expressions, which has also an impact on the ability to correctly compute the matching probabilities. Finally, we would also like to be able to combine exact and approximate searching as *agrep* does, where parts of the pattern accept errors and others do not. It is not hard to do this by using bit masks that control the propagation of errors.

Acknowledgements

Most of the algorithmic work preceding *nrgrep* was done in collaboration with Mathieu Raffinot, who unfortunately had no time to participate in this software.

References

[1] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476, September 1992.

[2] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[3] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[5] G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[6] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[7] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, LNCS v. 6, pages 118–132, 1979.

[8] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[9] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[10] N. El-Mabrouk and M. Crochemore. Boyer-moore strategy to efficient approximate string matching. In *7th International Symposium on Combinatorial Pattern Matching (CPM'96)*, pages 24–38, 1996.

[11] 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.

[12] R. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[13] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *Siam Journal on Computing*, 6(1):323–350, 1977.

[14] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[15] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.

[16] B. Melichar. String matching with k differences by finite automata. In *Proc. International Congress on Pattern Recognition (ICPR'96)*, pages 256–260. IEEE CS Press, 1996.

[17] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 2000. To appear. Earlier versions in *SIGIR'98* and *SPIRE'98*.

[18] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[19] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 2000. To appear.

[20] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.

[21] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Kluwer Information Retrieval Journal*, 3(1):49–77, 2000.

[22] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *9th International Symposium on Combinatorial Pattern Matching (CPM'98)*, LNCS 1448, pages 14–33, 1998.

[23] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. Technical Report TR/DCC-98-4, Dept. of Computer Science, Univ. of Chile, 1998.

[24] G. Navarro and M. Raffinot. Fast regular expression searching. In *3rd International Workshop on Algorithm Engineering (WAE'99)*, LNCS 1668, pages 198–212, 1999.

[25] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *5th International Workshop on Algorithm Engineering (WAE'01)*, LNCS, 2001. To appear.

[26] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings.* Cambridge University Press, 2001. To appear.

[27] M. Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In *Proc. 4th South American Workshop on String Processing (WSP'97)*, pages 149–165. Carleton University Press, 1997.

[28] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[29] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, 1992.

[30] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.