Pattern Matching

Gonzalo Navarro
Department of Computer Science
University of Chile

Abstract

An important subtask of the pattern discovery process is pattern matching, where the pattern sought is already known and we want to determine how often and where it occurs in a sequence. In this paper we review the most practical techniques to find patterns of different kinds. We show how regular expressions can be searched for with general techniques, and how simpler patterns can be dealt with more simply and efficiently. We consider exact as well as approximate pattern matching. Also, we cover both sequential searching, where the sequence cannot be preprocessed, and indexed searching, where we have a data structure built over the sequence to speed up the search.

1 Introduction

A central subproblem in pattern discovery is to determine how often a candidate pattern occurs, as well as possibly some information on its frequency distribution across the text. In general, a *pattern* will be a description of a (finite or infinite) set of strings, each string being a sequence of symbols. Usually, a good pattern must be as specific (that is, denote a small set of strings) and as frequent as possible. Hence, given a candidate pattern, it is usual to ask for its frequency, as well as to examine its occurrences looking for more specific instances of the pattern that are frequent enough.

In this paper we focus on *pattern matching*. Given a *known* pattern (which can be more general or more specific), we wish to count how many times it occurs in the text, and to point out its occurrence positions. The outcome of this search can be further processed by the pattern discovery machinery, possibly to come back with new searches for more specific patterns.

Since the search patterns of interest are in general complex, we leave out of this paper the search for exact strings, which are the most trivial search patterns. Of course strings can be searched for as particular cases of the complex patterns we consider in this paper, but no specific string matching techniques will be surveyed. These are easily found in most books of string matching or general algorithms [NR02, Gus97, CR94].

In general, we will focus on regular expression searching. More complex pattern searching (such as context-free languages [Mye95]) is less frequent and requires much more sophisticated techniques, so they will be out of our scope. On the other hand, we will show how some restricted types of regular expressions permit simpler and more efficient search algorithms. These simpler cases appear frequently in applications of computational biology, text editing, natural language processing, signal processing, etc. The kind of search patterns we consider in this paper follows.

Classes of characters: The simplest patterns we consider match a fixed-length string of the text, requiring that each character in the occurrence belongs to some subset of characters. We will represent this pattern as a sequence of sets of characters, each set within brackets. For example, `"[AB][CD]E"` is a pattern that matches strings of length 3. The first character of such strings must be `"A"` or `"B"`, the second `"C"` or `"D"`, and the third must by `"E"` (we omit the brackets for singleton sets). This encompasses simple strings, in particular. We may use lexicographic ranges to denote sets, for example `"[0-9]"` is a set matching any digit. Also `"."` is used to denote the set of all characters (to indicate the actual dot character, we use `"\."`).

Optional and repeatable characters: It is possible to indicate that some character in the pattern may or may not appear in its text occurrence. This character is said to be optional and will be followed by a questionmark, e.g., `"C?"`. Similarly, a character may be permitted to appear one or more times in its occurrence, which will be denoted as `"C+"`, and zero or more times will be denoted `"C*"`. For example, the pattern `"AB?C*D"` matches strings formed by `"A"`, then zero or one `"B"`, then zero or more `"C"`, and finally `"D"`.

Bounded length gaps: This matches any string within some length range, and is used to find a sequence of patterns within some distance range of each other. For compatibility with common biological usage, we will write `"x(a,b)"` to denote a string of length between a and b. For example `"AB x(2,5) C"` will match text areas where string `"AB"` appears followed by string `"C"`, with 2 to 5 characters in between.

Regular expressions: The most complex pattern we will consider are regular expressions. These are recursive structures built up from basic strings and union, concatenation, and repetition (zero or more times) of other regular expressions. The union is denoted with the infix operator `"|"`, concatenation by putting one expression after the other, and repetition with the postfix operator `"*"`. Highest precedence is given to the repetition, then to concatenation, and finally union, and this can be changed by using parentheses. For example, `"(AB|CD)*AFF*"` matches zero or more repetitions of either `"AB"` or `"CD"` (each repetition can use either string), then string `"AF"`, and finally zero or more repetitions of `"F"`. It is easy to see that regular expressions encompass all the other pattern specification techniques we have defined.

Network Expressions: These are a particular case of regular expressions, where the `"*"` operator is excluded. In most cases we do not give special search algorithms for these, but there will be some exceptions.

The above operations can be combined, and the methods we develop here easily permit their integration. For example, by permitting classes of characters and bounded length gaps, we obtain the expressivity of PROSITE patterns [HBFB99], of great interest in computational biology. Or we can have optional and repeatable classes of characters, so as to write `"[0-9]+\.?[0-9]*"` to find real numbers. Also, several other operators can be devised. The above are just some frequently used ones.

This paper is organized as follows. First, we will introduce automata as a general tool to deal with regular expression searching. We will show how regular expressions are converted to nondeterministic automata, how these can be made deterministic, and how automata are used for

searching. Some efficiency problems of this general approach will be pointed out. Then, we will present a technique called bit-parallelism to directly deal with the nondeterministic automata of the simpler patterns, by taking advantage of their regularity. Even for the most complex patterns, bit-parallelism will give us a space/time tradeoff. Then, we will explore some general speedup techniques that yield better average search times. We will then consider indexed searching, introducing the suffix trie data structure and the general way to traverse it given any automaton simulation. We will finally consider approximate pattern matching, which is very relevant to pattern discovery and orthogonal to the pattern complexity.

This paper aims at introducing the reader in the field, not at being exhaustive. We will give references for further reading at the appropriate points. Some good general references are [NR02, Gus97]. Also, some free software packages for exact and approximate complex pattern matching are *GNU grep* (`http://www.gnu.org/software/grep/grep.html`), *agrep* (`http://webglimpse.net/download.html`, top-level subdirectory `agrep/`), *nrgrep* (`http://www.dcc.uchile.cl/~gnavarro/pubcode`), and *nrep* (`http://www.cs.arizona.edu/people/gene/CODE/anrep.tar.Z`).

2 Formalization

Let our *text* $\mathcal{T}_{1...n}$ be a *string*, that is, a sequence of symbols (called *characters*) over a finite alphabet Σ of size σ. Some examples of alphabets are `"A", "C", "G", "T"` for DNA sequences, the 20 aminoacid letters for proteins, the set of letters and digits for natural language, the set of 128 pitch values for MIDI sequences, etc. $\mathcal{T}_{i...j}$ will denote the substring of $\mathcal{T}$ obtained by taking the i-th to the j-th characters. The empty string is denoted ε and xy is the concatenation of strings x and y, such that x is a *prefix* and y a *suffix* of xy.

Our searches will be made over text $\mathcal{T}$, which abstracts from a general collection of sequences by regarding them as a single sequence (some practical issues arise when handling multiple sequences, but these are easily dealt with).

In the most general case, the search pattern will be a *regular expression*. Regular expressions follow a simple syntax, and each regular expression $\mathcal{E}$ denotes a *language* $\mathcal{L}(\mathcal{E}) \subseteq \Sigma^*$, where Σ^* denotes the set of all strings over alphabet Σ. Regular expressions can be formed in the following ways:

- ε is a regular expression, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$.
- For any $c \in \Sigma$, c is a regular expression, $\mathcal{L}(c) = \{c\}$.
- If $\mathcal{E}_1$ and $\mathcal{E}_2$ are regular expressions, then $\mathcal{E}_1|\mathcal{E}_2$ is a regular expression, $\mathcal{L}(\mathcal{E}_1|\mathcal{E}_2) = \mathcal{L}(\mathcal{E}_1) \cup \mathcal{L}(\mathcal{E}_2)$.
- If $\mathcal{E}_1$ and $\mathcal{E}_2$ are regular expressions, then $\mathcal{E}_1 \cdot \mathcal{E}_2$ (or just $\mathcal{E}_1\mathcal{E}_2$) is a regular expression, $\mathcal{L}(\mathcal{E}_1 \cdot \mathcal{E}_2) = \mathcal{L}(\mathcal{E}_1) \cdot \mathcal{L}(\mathcal{E}_2)$. The operation over sets refers to all possible concatenations of one string from the left operand followed by one string from the right operand, that is, $X \cdot Y = \{xy,\ x \in X,\ y \in Y\}$.
- If $\mathcal{E}_1$ is a regular expression, then $\mathcal{E}_1*$ is a regular expression, $\mathcal{L}(\mathcal{E}_1*) = \mathcal{L}(\mathcal{E}_1)^*$. The latter operation over sets of strings, called "Kleene closure", gives all the strings formed by zero

or more concatenations of strings in the set. Formally, $X^* = \bigcup_{i \geq 0} X^i$, where $X^0 = \{\varepsilon\}$ and $X^{i+1} = X \cdot X^i$.

- If $\mathcal{E}_1$ is a regular expression, then $(\mathcal{E}_1)$ is a regular expression, $\mathcal{L}((\mathcal{E}_1)) = \mathcal{L}(\mathcal{E}_1)$.

As mentioned, the default precedence order is to read first the Kleene closures, then concatenations, and finally unions. Parentheses can be used to change that order.

Given a text $\mathcal{T}$ and a regular expression $\mathcal{E}$, the pattern matching problem is defined as: Find all the text positions that start an occurrence of $\mathcal{E}$, that is, compute the set

$$\{i,\ \exists j,\ \mathcal{T}_{i...j} \in \mathcal{L}(\mathcal{E})\}\ .$$

Alternatively, one may want all the final positions j of occurrences. Some applications require slightly different output. They may require finding *all* pairs (i, j), that is, initial and final positions of all occurrences. This might be problematic because some regular expressions may have a quadratic number of occurrences in $\mathcal{T}$. Because of this, some applications require finding maximal occurrences, minimal occurrences, or other variants [GOM98].

In this paper we concentrate on the simpler case of reporting initial or final occurrence positions. The other cases are dealt with by first finding initial/final positions (which is the computationally heaviest part), and then examining small text contexts around the positions found.

Observe that all the complex patterns we have defined in the Introduction are regular expressions. Simple strings $p_1 p_2 \ldots p_m$ can be written as $p_1 \cdot p_2 \cdot \ldots \cdot p_m$. Classes of characters $[p_1 p_2 \ldots p_r]$ can be written as $(p_1|p_2|\ldots|p_r)$. Optional characters $p?$ can be written as $(p|\varepsilon)$. Repeatable characters $p*$ can be written precisely as $p*$, and $p+$ as $pp*$. Finally, bounded length gaps like $x(2,5)$ can be written as $\Sigma\Sigma(\Sigma|\varepsilon)(\Sigma|\varepsilon)(\Sigma|\varepsilon)$, where Σ stands for the union ("|") of all the single characters of the alphabet.

3 Resorting to Automata

In this section we develop a complete solution to the pattern matching problem for regular expressions. This encompasses all the simpler types of patterns. At the end we will consider some efficiency problems of the proposed solution and motivate the next sections.

The general idea is to build an *automaton*, which is a finite-state machine that will be fed with the text characters and will signal all the text positions where an occurrence *finishes*. Used over the reverse pattern and text one gets initial occurrence positions. The general process is first to build a *nondeterministic finite automaton (NFA)* from the regular expression, then convert the NFA into a *deterministic finite automaton (DFA)*, and finally use the DFA to scan the text.

3.1 Nondeterministic Finite Automata

An NFA is a graph whose nodes are called *states* and its edges *transitions*. These leave from a *source* state and arrive at a *target* state, and are labeled by characters or by string ε. One state is designated as *initial* and zero or more as *final*. The NFA receives a string as input and *accepts* or *rejects* it. Essentially, if the string spells out the concatenation of labels of a path from the initial state to a final state, it will be accepted by the NFA, otherwise it will be rejected.

Let us describe acceptance and rejection more operationally. An *ε-transition* is a transition labeled by ε. The *ε-closure* of a state u, $E(u)$, is a set of states formed by u itself plus every state reachable from u via ε-transitions only. Given a string, start with the ε-closure of the initial NFA state being *active* and all the rest *inactive*. Now examine the string characters one by one. For each character, do as follows. If an active state is the source of a transition labeled by that character, then activate the target state. Now propagate activation to the ε-closures of those target states. These are the new active states. If, after having processed all the string characters, a final state is active, then accept the string, otherwise reject.

An NFA can be used to search the text as follows. Given regular expression $\mathcal{E}$, build the NFA for $\Sigma* \cdot \mathcal{E}$. This accepts all the strings *terminated* with some $y \in \mathcal{L}(\mathcal{E})$. Then, feed the NFA with $\mathcal{T}$. After processing the i-th character of $\mathcal{T}$, if the NFA accepts the string seen so far, then report an occurrence ending at text position i.

Given a regular expression $\mathcal{E}$, one can build an NFA that accepts exactly the strings in $\mathcal{L}(\mathcal{E})$. The best-known algorithm to do this is from Thompson [Tho68]. It proceeds recursively over the structure of the regular expression. Table 1 depicts the algorithm. An angle indicates the initial state and a double circle the final states. Figure 1 shows an example.

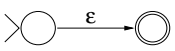
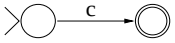
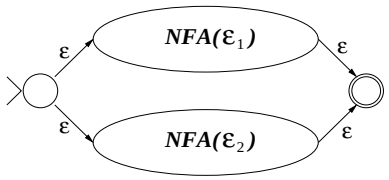
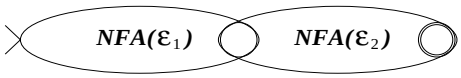
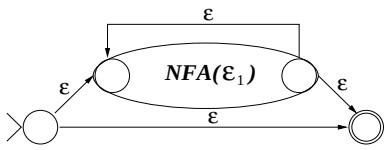
If $\mathcal{E} = \ldots$	then $NFA(\mathcal{E}) = \ldots$
ε	
$c \in \Sigma$	
$\mathcal{E}_1 \vert \mathcal{E}_2$	
$\mathcal{E}_1 \cdot \mathcal{E}_2$	
$\mathcal{E}_1*$	
$(\mathcal{E}_1)$	$NFA(\mathcal{E}_1)$

Table 1: Thompson's algorithm to build $NFA(\mathcal{E})$ from regular expression $\mathcal{E}$.

Thompson's NFA construction has several interesting properties: (i) if the regular expression is of length m (counting operators), then there are at most $2m$ states and $4m$ transitions; (ii) there is only one final state; (iii) no transition arrives at the initial or leaves the final state; (iv) at most two transitions leave from and arrive at each state; (v) it can be built in $O(m)$ time.

Based on those properties, Thompson [Tho68] gave an $O(mn)$ algorithm to run his NFA over

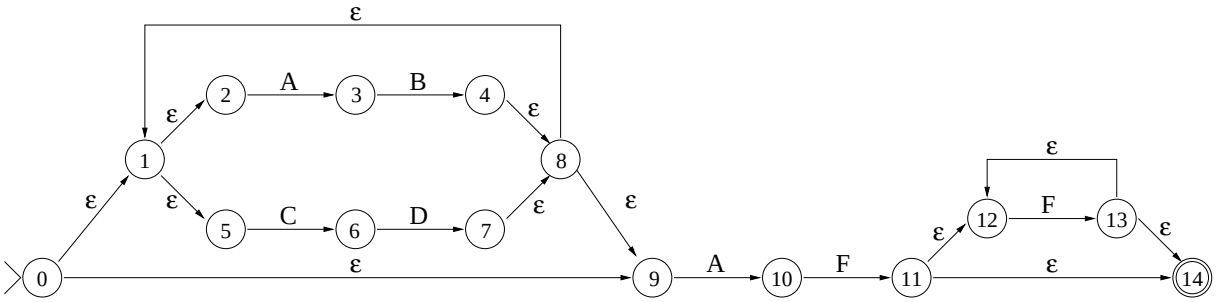


Figure 1: NFA resulting from applying Thompson's algorithm over the regular expression `"(AB|CD)*AFF*"`.

a text of length n. The idea is to maintain a boolean array indicating active/inactive states, and for each new text character, compute the newly active states and propagate through ε-transitions. The latter must be done with some care to ensure $O(m)$ cost per propagation (basically, we must not continue the propagation from a state that has already been activated.)

By running Thompson's algorithm for the NFA of regular expression $\Sigma* \cdot \mathcal{E}$ over the text $\mathcal{T}$, we immediately get an $O(mn)$ pattern matching algorithm for regular expression $\mathcal{E}$. We show next how to improve this search cost.

3.2 Deterministic Finite Automata

The efficiency problem of the above NFA simulation is due to the fact that several states can be simultaneously active. A DFA, on the other hand, has exactly one active state at a time, and hence gives us an $O(n)$ time search algorithm.

A DFA is a particular case of NFA such that: (i) there are no ε-transitions, and (ii) from each state and for each alphabet character there is exactly one leaving transition. It is easy to see that exactly one state can be active at a time in a DFA, since there is only exactly one transition to follow per text character. Hence processing each text character in $O(1)$ time is trivial.

To convert an NFA into a DFA [ASU86], we have to regard each possible subset of active NFA states as a single DFA state, and compute the transitions among them. Note that, if the NFA has $2m$ states, we have potentially 2^{2m} DFA states. To alleviate this, we build only the DFA states that can be reached from the initial state. For example, in Figure 1, every time state 0 is active, state 1 is active too. Hence no DFA states representing subsets that contain state 0 and do not contain state 1 are reachable.

The initial state of the DFA will be $U_0 = E(u_0)$, where u_0 is the initial NFA state. This is the first DFA state we build. Now, for each new DFA state we build, we compute its transitions for every $c \in \Sigma$. Given DFA state U and character c, the transition from U by c leads to state $U' = \bigcup_{u \in U, u \xrightarrow{c} v} E(v)$, where $u \xrightarrow{c} v$ means that the NFA has a transition from u to v labeled by c. If the new DFA state U' had not been already built, we recursively compute the σ transitions leaving it. The process continues until all the reachable states are built. Figure 2 shows the DFA for our example NFA.

As explained, the resulting DFA can be used to search the text for a regular expression in

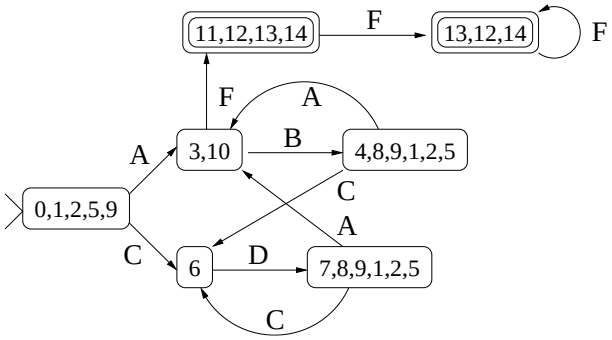


Figure 2: DFA resulting from determinizing Thompson's NFA for the regular expression `"(AB|CD)*AFF*"`. All the transitions not drawn lead to a *sink* state, corresponding to the empty subset of NFA states. All the transitions leaving from the sink state lead to itself.

$O(n)$ time. Observe, however, that $O(\sigma 2^{2m})$ space and preprocessing time may be necessary to build the DFA. Depending on the application, the worst case can be more or less probable. In our experience, this rarely happens in regular expressions used on natural language. On the other hand, many PROSITE patterns tend to produce large DFAs because of the gaps. It is possible to minimize the DFA so as to recognize the same language with the minimum number of states [ASU86]. For example, in Figure 2, states $\{4, 8, 9, 1, 2, 5\}$ and $\{7, 8, 9, 1, 2, 5\}$ could be merged. However, this is done over the original DFA and hence it does not reduce the necessary space nor preprocessing time.

3.3 Discussion

The classical techniques we have shown in this section provide a complete solution to the pattern matching problem. Either one can search in $O(mn)$ or in $O(\sigma 2^{2m} + n)$ worst case time. In many contexts, this turns out to be the best (or the only) known solution. However, in several cases we can do better: (i) for the simpler patterns considered in Section 1, resorting to automata may be an overkill, both in programming effort and in preprocessing cost; (ii) the space and construction time required by the DFA may be prohibitive; (iii) the text size may make a sequential scanning approach unfeasible. In the next sections we present techniques to address those concerns.

4 Bit Parallelism

In this section we consider simulating automata in their nondeterministic form rather than converting them to deterministic. This turns out to be especially interesting for the simpler patterns, because their NFAs are regular enough to to permit a very efficient simulation. Our central tool for this simulation is *bit-parallelism* [WM92, BYG92], a general technique to pack several small values into a single computer word, so as to update all them simultaneously with a few arithmetic and logical operations over the computer word. In our case, we mainly pack NFA states as bits of the computer word.

The exposition of bit-parallel techniques requires of some terminology for operations on *bit*

masks (sequences of bits). Just like binary numbers, bit masks are read right to left, the rightmost bit being the first. Given two bit masks M_1 and M_2 of the same length, "$M_1 \mid M_2$" is the bitwise *or* between M_1 and M_2, "M_1 & M_2" is the bitwise *and*, "$M_1 \wedge M_2$" is the bitwise *xor*, and "$\sim M_1$" negates all the bits in M_1. Also, "$M_1 << k$" shifts all the bits of M_1 to the left by k positions, discarding the k highest bits and entering zeros from the right. Similarly, "$M_1 >> k$" shifts the bits to the right and enters zeros from the left. For simplicity we assume that our bit masks fit in the computer word, of w bits, but we also indicate how to proceed otherwise.

4.1 Classes of Characters

Let us start with the simplest problem of handling classes of characters. Let $P = p_1 \ldots p_m$ be the search pattern, where $p_i \subseteq \Sigma$. The NFA that searches for $P =$ `"[AB][CD]E"` is given in Figure 3.

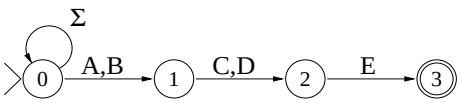


Figure 3: NFA that recognizes every string terminated in pattern `"[AB][CD]E"`.

The basic bit-parallel text search algorithm designed for simple strings [WM92, BYG92] extends naturally to classes of characters. Precompute a table B of bit masks, indexed by alphabet characters, so that the i-th bit of $B[c]$ is set if and only if $c \in p_i$. This mask is the only connection between the search phase and the pattern. To build B, set $B[c] \leftarrow 0$ for all $c \in \Sigma$, and then set $B[c] \leftarrow B[c] \mid (1 << (j-1))$ for every $c \in p_j$, for all j.

For the search, set up a bit mask D so that the i-th bit of D is set if and only if state i in the NFA is active. NFA state 0 is excluded from the representation because it is always active, so initially $D \leftarrow 0$. Now, upon reading text character c, update D as follows:

$$D \leftarrow ((D << 1) \mid 1) \;\&\; B[c] \;,$$

whose rationale is rather immediate. State i is active after reading text character c if and only state $i-1$ was previously active, and c belongs to p_i. The "$\mid 1$" fixes the fact that we do not represent the always-active NFA state 0. After reading text character c, we declare the end of an occurrence of P whenever the m-th bit of D is set, that is, $D \;\&\; (1 << (m-1)) \neq 0$

The resulting code has just a few lines and is extremely efficient. If $m \leq w$, it is $O(n)$ time and does most of the work using registers, so it is usually faster than a DFA based approach because it uses a smaller table (B), which can be cached better. If $m > w$ the simplest choice is to search for $P_{1\ldots w}$ and check every match for a complete occurrence. This is usually $O(n)$ on average. The best worst case time is $O(mn/w)$, obtained by using $\lceil m/w \rceil$ computer words to simulate the bit masks. Finally, the preprocessing is very light, $O(m\sigma/w)$ time.

4.2 Optional and Repeatable Characters

Figure 4 shows one possible NFA to search for $P =$ `"AB?C*D"`. From the three operators, we choose to deal only with `"?"` and `"+"`, treating `"C*"` as `"C?+"`. The example shows that it is possible to have a sequence of optional characters, so the ε-transitions could be followed transitively.

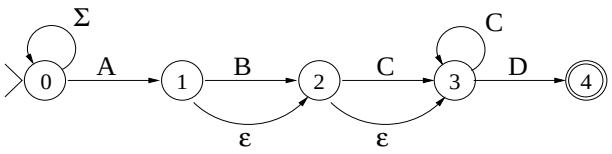


Figure 4: NFA that recognizes every text terminated in pattern `"AB?C*D"`.

Extending the basic bit-parallel approach to deal with repeatable characters is rather simple[1]. The self-loops can be simulated by setting up a bit mask R indicating which positions in P are repeatable (in our example $R = 0100$). Then, instead of the normal update to mask D, we do

$$D \leftarrow ((D << 1) \mid 1 \mid (D \;\&\; R)) \;\&\; B[c] \;,$$

where the extra element permits a repeatable state to stay active as long as we read a character matching its self-loop.

Dealing with optional characters (that is, ε-transitions), is a bit more complicated [NR02]. Let us consider *blocks* of contiguous optional characters in P. A bit mask I indicates the positions *preceding* the beginning of each block of optional characters, and F the positions where each block finishes. In our example, $I = 0001$ and $F = 0100$. We also use a mask A that indicates all the optional positions in P, $A = 0110$ in our example. To ensure that we propagate all the forward ε-transitions we must do, after processing D as usual,

$$D \leftarrow D \mid (A \;\&\; ((\sim ((D|F) - I)) \;\wedge\; (D|F))) \;.$$

The above formula requires some explanation. First, $(D|F) - I$ floods each block with 1's, from the beginning to just before the first active state in the block. Note that it could have been just $D - I$, except that we must consider blocks where there are no active states, so that the subtraction does not propagate to other blocks. By negating the result and *xor*-ing it with $D|F$, the effect is that the first active bit in each block of D floods all the block to the left, which is what we desired. The rest is just limiting the changes to occur inside blocks, and adding the flooded states to those already active in D.

The preprocessing and searching complexities are similar to the case of classes of characters.

4.3 Bounded Length Gaps

Bounded length gaps also permit a simple NFA simulation without converting it into a DFA. Note that, for example, `"AB x(2,5) C"` can be rewritten as `"AB...?.?.?C"` (Figure 5), and we have seen already how to simulate classes and optional characters. Actually, this case is more restricted than general optional characters, and a slightly simpler solution is possible [NR03, NR02].

4.4 Network and Regular Expressions

General regular expression NFAs are not regular enough to permit a simple bit-parallel simulation that avoids exponential preprocessing time and space. However, some regularities can be exploited

[1]This differs from the solution given in [NR02]. The simpler and faster version we present here is due to Heikki Hyyrö.

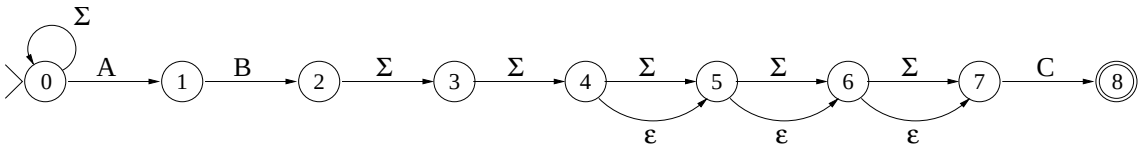


Figure 5: NFA that recognizes every text terminated in pattern `"AB x(2,5) C"`.

and a space/time tradeoff is possible [WM92, NR01].

Consider again Thompson's NFA of Figure 1. It is not hard to make the construction algorithm number the states so that all transitions labeled by characters in Σ go from state i to state $i+1$, for some i. This permits us dealing with all the character-labeled transitions in a way very similar to that of plain strings [WM92]. We identify each NFA state with a bit in a bit mask. Each different bit mask of active and inactive states actually identifies a DFA state. We use again table B so that the i-th bit of $B[c]$ is active whenever NFA state i is the target of a transition labeled by c. Note that, if this bit is set, we know that such transition can only come from state $i-1$. For all the ε-transitions, we precompute a large table E, which for every bit mask D of active and inactive states gives the ε-closure of states active in D, that is, states reachable from those in D via ε-transitions (including the states in D).

Once tables B and E are precomputed, we can easily simulate one step of the NFA upon reading text character c as follows:

$$D \quad \leftarrow \quad E[\ (D << 1)\ \&\ B[c]\ ]\ ,$$

where the argument of E takes care of the character-labeled transitions in a way very similar to that for simple strings, and table E performs all the propagations through ε-transitions.

For searching purposes, only table E must be modified, so as to include the ε-closure of the initial NFA state. Mask D is then initialized as $D \leftarrow E[0]$. It is not hard to build E in optimal time, that is, $O(2^h h/w)$, where h is the number of NFA states [NR02]. Thompson's construction ensures $h \leq 2m$, where m is the length of the regular expression. Hence the algorithm needs space $O(2^{2m}m/w)$ and time $O(2^{2m}m/w + nm/w)$. This is simply $O(2^{2m} + n)$ for $m \leq w$.

Note that, compared to the classical complexity, the $O(\sigma)$ factor that multiplied preprocessing time and space has disappeared. In exchange, we have now an $O(m/w)$ factor. The way to get rid of the alphabet dependence has been to take advantage of NFA regularities. Note, however, that the bit-parallel approach *always* needs $O(2^h)$ space and preprocessing time, while the classical approach does so only in the worst case.

On the other hand, bit-parallelism permits a simple space/time tradeoff related to table E. Assume that $O(2^h)$ is too large for our available space or preprocessing time. We can split E into two tables of size $O(2^{h/2})$ each, E_1 and E_2. Each table takes care of half of the NFA states, and tells which NFA state are reachable via ε-transitions from states of its half. Hence, if D is split into two halves as $D = D_1 : D_2$, then it holds $E[D] = E_1[D_1] \mid E_2[D_2]$. We now need two table accesses to compute the original E entry, but in exchange use about the square root of the size for the full E table. This can be generalized to splitting into k tables, at the cost of $O(kn)$ table accesses for searching. A few calculations show that, with $O(s)$ space available, we can search in $O(mn/\log s)$ time. If space is not a problem, then the optimum is to use $k = \log(mn)$, for an overall search time and space of $O(mn/\log n)$.

The same results have been achieved with other techniques [Mye92]. These are more complicated, although they remove the $O(m/w)$ multiplying factor we have attached to all the bit-parallel complexities.

4.5 Using Glushkov's NFA

We recall that Thompson's construction guarantees $h \leq 2m$. Since the scheme is finally exponential in h, it is interesting to notice that there are NFA constructions, such as Glushkov's [Glu61], that always produce exactly $h = m + 1$ states, where this time m is the number of alphabet characters in the regular expression (that is, operator symbols are not counted).

We do not have space to describe Glushkov's construction in this paper (see, e.g., [NR02]). We content ourselves by pointing out some of its properties: (i) there are exactly $m + 1$ states but $O(m^2)$ transitions; (ii) there are no ε-transitions; (iii) all transitions leading to a state are labeled by the same character. Figure 6 shows an example of this construction.

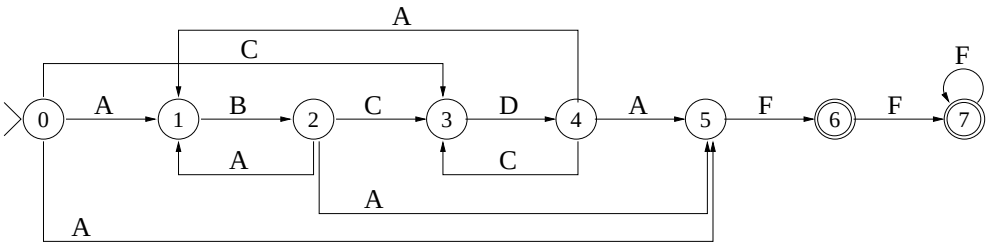


Figure 6: NFA resulting from applying Glushkov's algorithm over the regular expression `"(AB|CD)*AFF*"`.

The above properties permit a much more efficient bit-parallel simulation [NR01, NR02]. We use table $B[c]$ as before, and a large table T which for every bit mask D of $m + 1$ states tells which states are reachable from those of D in one transition (no matter by which character). Then, thanks to property (iii) above, the new D mask upon reading text character c is computed as

$$D \leftarrow T[D] \ \& \ B[c] \ .$$

Table T plays a role similar to E: It can be split to obtain time/space tradeoffs, and used to maintain active the initial state for searching.

Using Glushkov's NFA, the preprocessing time and space drops from $O(2^h)$ to $O(2^m)$, which is always better because $h \geq m + 1$ in Thompson's construction. Hence, Glushkov's construction should always be preferred over Thompson's for bit-parallel simulations. In other cases the $O(m)$ transitions guaranteed by Thompson's construction makes it preferable.

4.6 Discussion

In this section we have shown how bit-parallelism permits simulating an NFA directly without converting it into a DFA. Whether this can be done or not depends on the pattern features. We have shown how it can be carried out with the simpler search patterns we have defined. For the more complex case of regular expressions, we have shown how to obtain a simpler implementation and

a space/time tradeoff with little programming effort. Each new type of pattern we face requires designing a different bit-parallel algorithm using different tricks, but the reward may be much simpler programming, much faster preprocessing, and usually faster searching.

Having a lighter preprocessing is especially attractive in applications where there is a short text to search for many patterns, as it happens for example in computational biology when trying a new protein against a PROSITE database. In this case, most of the cost of a classical DFA approach is spent in pattern preprocessing.

5 General Speedup Techniques

Up to now, we have considered that all the search algorithms would process every text character, one by one. Thus, the best time we have obtained is $O(n)$. In simple string matching, *skipping* of text characters is used to obtain better average search times, reaching the average-case lower bound $O(n \log_\sigma(m)/m)$ [Yao79]. Although achieving this complexity is not always possible for complex patterns, the techniques used for strings can be extended and in many cases they improve search cost of complex patterns. We explore a couple of these techniques in this section.

5.1 Necessary Strings

A general technique to convert the search for pattern $\mathcal{E}$ into multiple string matching is as follows [Wat96]. First, compute ℓ_{min}, the minimum length of a string in $\mathcal{L}(\mathcal{E})$. Then, traverse all the NFA paths from the initial NFA state, obtaining all strings of length ℓ_{min} that can start a string in $\mathcal{L}(\mathcal{E})$. These strings will be called the *prefixes* of $\mathcal{E}$. Use a multiple string matching algorithm [NR02] to find the occurrences of all those prefixes in the text. For every such occurrence $\mathcal{T}_{i...i+\ell_{min}-1}$, use the NFA simulation from $\mathcal{T}_{i...}$ to determine whether the prefix starts a full occurrence of $\mathcal{E}$.

For our example regular expression `"(AB|CD)*AFF*"` we have $\ell_{min} = 2$. The prefixes of the expression is the set { `"AB"`, `"CD"`, `"AF"` }. Hence we can first search for those three strings and be sure that any occurrence of the regular expression starts with an occurrence of some of these strings.

The above technique can be generalized so that, instead of finding the set of prefixes of $\mathcal{E}$, we find some set of *necessary substrings* of $\mathcal{E}$, so that every occurrence of $\mathcal{E}$ must contain some necessary substring [Nav01b]. The verification process and the way to choose a good set of necessary strings is more complicated, but the reward may be considerable depending on the pattern.

In our example regular expression, `"AF"` turns out to be a necessary substring, so that any occurrence of the regular expression contains `"AF"`. It is faster to search for just this string than to search for the three possible prefixes.

5.2 Backward Suffix Matching

Another general and elegant way of speeding up the pattern search is inspired by the BDM algorithm [CCG+94, NR00]. Given an NFA that recognizes $\mathcal{E}$ (not $\Sigma* \cdot \mathcal{E}$), reverse all its transitions (that is, exchange sources with targets) and make the former initial state be the final state of the new NFA. Also, add a new initial NFA state and draw ε-transitions from it to all other NFA states. The result is an NFA that recognizes *reverse prefixes* of strings in $\mathcal{L}(\mathcal{E})$. Note that such NFA is active

as long as we have fed it with a reverse substring of a string in $\mathcal{L}(\mathcal{E})$. Figure 7 shows an example. We use the NFA to build an algorithm that finds all *initial* positions of $\mathcal{E}$ in $\mathcal{T}$.

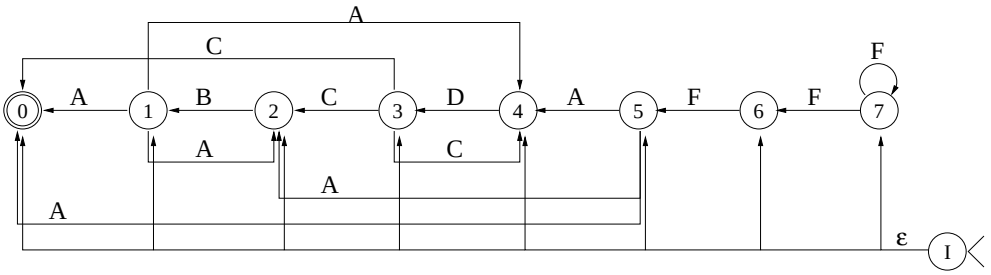


Figure 7: NFA resulting from applying Glushkov's algorithm over the regular expression `"(AB|CD)*AFF*"` and then modifying it to accept reverse prefixes of the expression.

Using again ℓ_{min}, proceed as follows. Slide a *window* of the form $\mathcal{T}_{i+1\ldots i+\ell_{min}}$ over the text, starting with $i = 0$ and finishing when $i + \ell_{min} > n$. The invariant is that any occurrence starting before position i has already been reported. For every window, read its characters *backwards* and feed the transformed NFA with them. Remember the window position $last$ that signals the last time the NFA recognized a reverse prefix. This means that $\mathcal{T}_{i+last\ldots i+\ell_{min}}$ is a prefix of a string in $\mathcal{L}(\mathcal{E})$. If at some point the transformed NFA runs out of active states, then the portion of the window read cannot be part of any occurrence of $\mathcal{E}$ because what we have read is not a substring of any string in $\mathcal{L}(\mathcal{E})$. At this point, we can safely advance the window to start at position $i + last$, which is the leftmost text position following i that has a chance of starting an occurrence.

If, on the other hand, we read all the window characters and the NFA is still active, there are two possibilities. The first is that the final NFA state is not active. In this case, the window is not a prefix of a string in $\mathcal{L}(\mathcal{E})$ and hence we advance the window as before. The second is that the final NFA state is active, and thus we have to check for an occurrence of $\mathcal{E}$ starting at position $i+1$. For this sake, we use the original NFA that recognizes $\mathcal{E}$ and feed it with $\mathcal{T}_{i+1\ldots}$ until either the NFA runs out of active states or it recognizes a string. In this latter case, we report an occurrence. In any case, after this verification, we advance the window to $i + last$.

Figure 8 illustrates the general process.

Bit-parallelism is an excellent tool to combine with BDM [NR00, NR00, NR01]. Given a bit-parallel simulation for $\mathcal{E}$ (not $\Sigma* \cdot \mathcal{E}$), we can apply it over the reversed regular expression, and initialize it with all states active (all 1's). This simulates an NFA that recognizes any reverse prefix of $\mathcal{E}$.

5.3 Discussion

In this section we have presented some techniques to speed up complex pattern searching. These derive from simple string matching and, in general, work better as the pattern is closer to a simple string. As a rule of thumb, patterns where ℓ_{min} is small or $\mathcal{L}(\mathcal{E})$ is very large, are unlikely to work well with these speedup techniques. It is possible to estimate with reasonable precision the expected search cost for a given pattern, and even use this estimation to choose the best necessary strings or subpattern to search for [Nav01b].

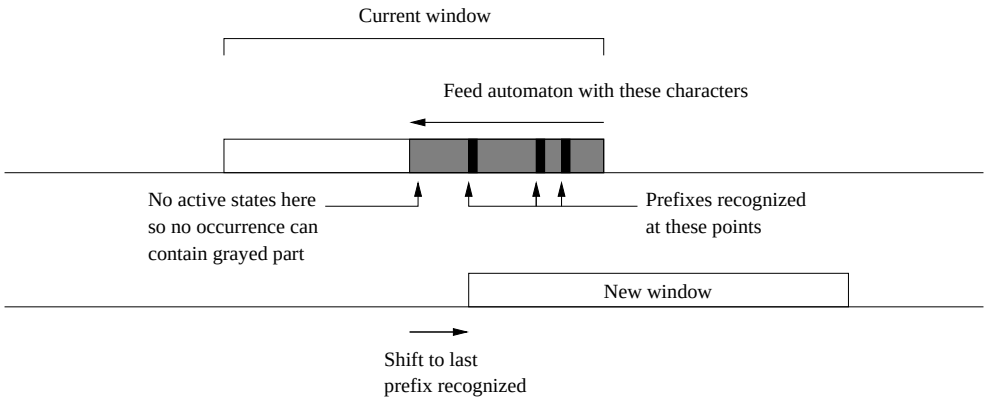


Figure 8: General BDM process. We show the case where the automaton runs out of active states before reaching the beginning of the window.

6 Indexed Searching

Up to now we have considered that the text is sequentially scanned for every query. In many practical cases the text is too large for that approach. In this case the text can be preprocessed to build an *index* on it, which is a data structure to accelerate searches. Indexed searching is interesting when (*i*) queries are much more frequent than changes to the text, so that the indexing cost is amortized; (*ii*) there is enough space to store the index; (*iii*) the text is large enough to justify the more complex indexed solution.

In this paper we introduce the most popular data structures for general text indexing: suffix tries, trees, and arrays. Most sequential search algorithms that work character by character can be easily adapted to work on these structures (hence, those of Section 5 are excluded). The way to adapt the algorithms is orthogonal to how they process each text character. For this reason, we will abstract from the way we simulate the NFA of the search pattern. We consider that we have to find all the *initial* occurrence positions of a regular expression $\mathcal{E}$, and we have already built an automaton that recognizes $\mathcal{E}$ (not $\Sigma* \cdot \mathcal{E}$). Again, finding final occurrence positions is easy once we know how to find initial positions.

6.1 Suffixes and Suffix Tries

Given a text $\mathcal{T}_{1...n}$, we consider its n suffixes of the form $\mathcal{T}_{i...n}$, for $i \in 1 \ldots n$. For example, the suffixes for text $\mathcal{T}$ = `"ABAFAAF"` are `"ABAFAAF"`, `"BAFAAF"`, `"AFAAF"`, `"FAAF"`, ..., `"F"`. No suffix is a prefix of another provided we append to $\mathcal{T}$ a special terminator character `"$"` which is smaller than any other character. The key observation is that every text substring is the prefix of some suffix, so searching can be done by running the automaton against every suffix of $\mathcal{T}$, until either (*i*) the automaton reaches a final state, in which case the suffix initial position starts an occurrence, or (*ii*) the automaton runs out of active states, in which case we can abandon that suffix.

For example, to find all the initial positions of the regular expression `"(AB|CD)*AFF*"` in the text given, we could run the NFA of Figure 1 (simulated somehow) over all the suffixes mentioned in the previous paragraph. For suffixes `"ABAFAAF"`, `"AFAAF"`, and `"AF"`, the NFA reaches its final

state after reading 4, 2, and 2 characters, respectively. Hence text positions 1, 3, and 6 are marked as initial occurrence positions. On the other hand, for suffixes `"BAFAAF"`, `"FAAF"`, `"AAF"`, and `"F"`, the NFA runs out of active states after processing 1, 1, 2, and 1 characters, respectively.

The *suffix trie* of a text $\mathcal{T}$ is a digital tree where we have inserted all the suffixes of $\mathcal{T}$. The path corresponding to each suffix is unrolled until a unique prefix of that suffix has been spelled out. At this point, a leaf pointing to the text position where the suffix begins is added to the path. Figure 9 shows an example.

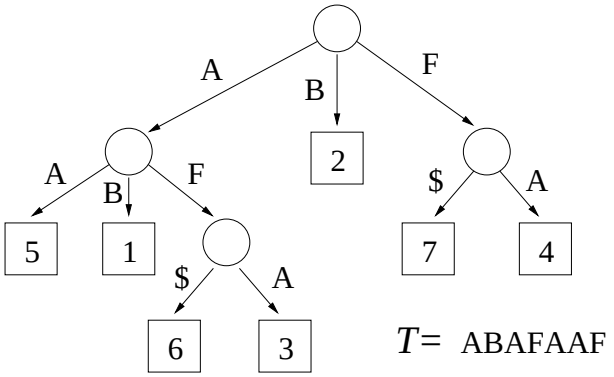


Figure 9: Suffix trie for text `"ABAFAAF"`. Leaves are squared nodes and internal nodes are circled.

The important property of the suffix trie is that all the suffixes sharing a prefix also share a path in the trie (that spells out the shared prefix). Therefore, by traversing the trie with the automaton we process in one shot all the prefixes shared by those suffixes. Next we precise the search mechanism.

6.2 Backtracking

We backtrack on the suffix trie, exploring all its branches. We start at the trie root with only the initial automaton state active (and its ε-closure). Every time we descend by character c to a child, we feed the automaton with c. We keep a stack of automata states so that, when we return from that child, we can recover the automaton state before we fed it with c, and thus can try another child. This backtracking must enter all trie nodes until:

(i) The automaton reaches a final state. In this case the path from the root to the current trie node spells out a string in $\mathcal{L}(\mathcal{E})$. This means that all the suffixes starting with the path read start an occurrence of $\mathcal{E}$. Therefore, we collect all the trie leaves that descend from the current node and report them as occurrences. Then we abandon the node.

(ii) The automaton runs out of active states. This means that the path from the root to the current node spells out a string that cannot start an occurrence of $\mathcal{E}$. Hence we abandon the current node immediately.

Note also that, if we reach a leaf labeled s at depth d before conditions (i) or (ii) have been met, we must go directly to $\mathcal{T}_{s+d...}$ and continue feeding the automaton with the text characters, until one of the two conditions is met. If condition (i) is met, we report position s.

Figure 10 shows which nodes in our example trie would be traversed by the backtracking. From the root we feed the automaton with `"A"` and it still has active nodes (3 and 10 in Figure 1). Then we try to descend again by `"A"`, but the automaton runs out of active states and hence leaf 5 is not reached. Instead of `"A"` we try `"B"`, and the automaton still has active states (4, 8, 9, 1, 2, and 5). But we have reached a leaf (pointing to text position 1 and at depth 2). So we go to $\mathcal{T}_{3...}$ and feed the automaton with the successive characters. After processing `"A"` and `"F"`, we reach a final state and hence report text position 1. Now we go back to depth 1 and descend by `"F"` instead of `"B"`. The automaton reaches a final state and therefore we report all the leaves in the subtree, that is, text positions 6 and 3. We go back to the root and try to descend by `"B"` and `"F"`. In both cases the automaton runs out of active states and we finish.

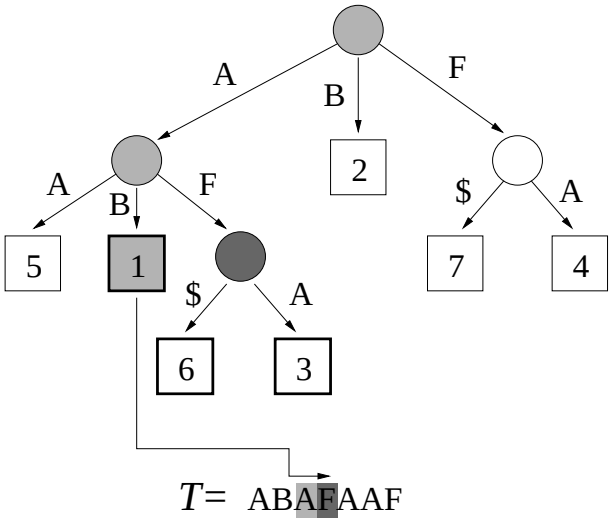


Figure 10: Backtracking for regular expression `"(AB|CD)*AFF*"` over the suffix trie for text `"ABAFAAF"`. Grayed nodes/characters are those traversed during the backtracking, and dark grayed ones indicate that the automaton reached a final state. Bold squares correspond to reported leaves.

It has been shown [BYG96] that the number of trie nodes processed by this search is $O(n^\lambda)$, where $0 \leq \lambda \leq 1$ depends on the regular expression. For most regular expressions $\lambda < 1$, and hence the search cost does not grow proportionally to the text size. In particular, $\lambda = 0$ when the automaton is acyclic. This includes, for example, classes of characters, optional characters, bounded length gaps, and network expressions. It is easy to see that there is a maximum string length ℓ_{max} that can match those patterns, and therefore we cannot enter the trie deeper than ℓ_{max}. Hence, no matter how large is the text, the maximum number of nodes that can be processed is $\sigma^{\ell_{max}}$. Although this is independent of n, it might be exponential in the pattern length.

6.3 Discussion

In this section we have addressed indexed searching. We have reexpressed the search problem as one of exploring all the text suffixes, and then introduced the suffix trie as a data structure that merges common prefixes of those suffixes. Hence the trie can be used to process characters from

many text suffixes at once.

One problem with the suffix trie is that, in the worst case, it might require $O(n^2)$ space. Although on average it is $O(n)$ size, the suffix trie still requires too much memory compared to the text, say 20 times the text size. A structure guaranteeing $O(n)$ worst-case space and construction time is the suffix tree [McC76, Gus97], which compresses unary suffix trie paths into single edges. Still, in practice suffix trees are as large as suffix tries.

A significantly smaller data structure (4 times the text size) that provides similar functionality is the suffix array [MM93], which is just an array of pointers to all text suffixes in lexicographical ordering. It can be regarded as an array with the leaves of the suffix trie. Every subtrie corresponds to a suffix array interval, which can be binary searched because the suffixes are lexicographically sorted. Hence, instead of stepping at a trie node, we step at a suffix array interval. Descending to a child involves two binary searches to determine the subinterval corresponding to the child. So the backtracking can be simulated over the suffix array with a penalty factor of $O(\log n)$ time. In practice this is a good compromise when no enough memory is available to build the suffix tree.

Suffix tries, trees, and arrays are also extremely useful for pattern discovery, for example to find repeated text strings. See [Gus97] for a thorough discussion.

7 Approximate Searching

Even when we have considered complex patterns, up to now we have required that the text occurrence matches *exactly* a string in the language specified by the pattern. In most pattern discovery applications, the text and/or the pattern may be imperfectly known, so that one would like to spot text occurrences that are *similar enough* to the pattern specification. The measure of similarity strongly depends on the application, and computing it for two strings may range from linear-time to NP-complete. In this section we focus on a very popular measure called *edit distance* (which is actually a measure of dissimilarity). The edit distance between two strings x and y is the minimum cost to convert x into y by means of a sequence of *edit operations* on x. The allowed edit operations are: inserting a character c (with cost $\delta(\varepsilon, c)$), deleting a character c (with cost $\delta(c, \varepsilon)$), and substituting character c by c' (with cost $\delta(c, c')$, so that $\delta(c, c) = 0$). Other operations such as transpositions, swaps, gap insertions, block moves, reversals, etc. will not be considered (see, e.g., [Gus97, SM97]).

The approximate search problem can therefore be stated as follows. Given a text $\mathcal{T}$, a regular expression $\mathcal{E}$, an edit distance between strings $d()$, and a distance threshold k, find all the text positions that start an approximate occurrence of $\mathcal{E}$ in $\mathcal{T}$, that is, compute the set

$$\{i,\ \exists j,\ \exists S \in \mathcal{L}(\mathcal{E}),\ d(\mathcal{T}_{i...j}, S) \leq k\}\ ,$$

where, again, we may wish to report final instead of initial occurrence positions.

The definition of edit distance encompasses several popular measures such as Levenshtein distance, Hamming distance, and indel distance (the dual of the Longest Common Subsequence similarity measure). In particular, Levenshtein distance results from using $\delta = 1$ for the three operations. It turns out to be a good approximation in many applications, and therefore it has received a lot of attention and very efficient algorithms have been developed for it [Nav01a]. We will first present

solutions for the Levenshtein distance and then will consider the more complex case of general edit distance, which is important in computational biology, for example.

7.1 Levenshtein Distance

Consider the NFA of Figure 11 (e.g., [BYN99]). It corresponds to the approximate search version of Figure 3. Actually, every time the top final state is reached, we have an exact occurrence (with Levenshtein distance zero). Let us now consider in which cases can the final state of the second row be activated. At some point in the path from state 0 to 3, we must have followed a vertical or diagonal transition. If we followed a vertical transition, it means that we have read a text character without advancing in the pattern. That is, we have deleted such text character to obtain a string S matching the pattern. If we followed a diagonal transition using label Σ, then we have read a text character and also advanced in the pattern, without checking that they matched. That is, we have substituted a text character by another to obtain S. Finally, if we followed a diagonal transition using label ε, then we have advanced a pattern position without advancing in the text. That is, we have inserted a character in the text to obtain S. Overall, the second final state is active whenever we find a text occurrence at Levenshtein distance at most 1 from a string S matching the pattern. In general, the final state at the $(k+1)$-th row is active whenever we have read a string at distance at most k from the pattern.

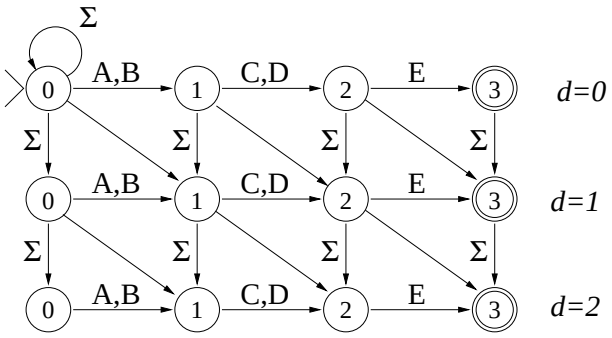


Figure 11: NFA that recognizes every string terminated in pattern `"[AB][CD]E"` within Levenshtein distance 2. Diagonal transitions are labeled by $\Sigma \cup \{\varepsilon\}$.

Therefore, we can use the $(k+1)$-rows NFA for searching purposes. Every time the final state of the bottom row is active, we declare the final position of a pattern occurrence within distance threshold k.

A nontrivial issue is how to simulate this NFA efficiently. Thompson's simulation [Tho68] can be used to obtain $O(kmn)$ search time. On the other hand, the NFA can be made deterministic to search in $O(n)$ time. However, the size of the DFA is $O(\min(3^m, (m\sigma)^k)$ [Ukk85], and this time the number of states is usually very large in practice.

A bit-parallel approach is usually a better choice [WM92]. Precompute table B exactly as in Section 4.1. Set up $k+1$ bit masks, $D_0 \ldots D_k$, each representing the active bits of each NFA row. Initialize them as $D_i = (1 << i) - 1$. For each new text character c, compute the new value for D_0

as in Section 4.1:

$$D'_0 \leftarrow ((D_0 << 1) \mid 1) \mathbin{\&} B[c] \ ,$$

and then, for $i \in 1 \ldots k$ in increasing order, compute the new value for D_i as:

$$D'_i \leftarrow ((D_i << 1) \mathbin{\&} B[c]) \mid D_{i-1} \mid (D_{i-1} << 1) \mid (D'_{i-1} << 1) \mid 1 \ ,$$

where, in addition to the normal transitions, we simulate vertical transitions by adding the active states in D_{i-1}, the diagonal transitions labeled by Σ by adding the active states in $D_{i-1} << 1$, and finally the diagonal transitions labeled by ε by adding the active states in $D'_{i-1} << 1$. Note that in the latter case we are referring to the *new* value of D_{i-1}, since ε-transitions propagate immediately. The extra "$\mid 1$" is necessary because we have not represented state 0. Finally, we declare the final position of an occurrence whenever $D_k \mathbin{\&} (1 << (m-1)) \neq 0$.

The above simulation works in $O(kn)$ time when $m \leq w$, and $O(kmn/w)$ time otherwise. It is quite efficient in practice for moderate size patterns. Although there exist more efficient simulations of this NFA [BYN99, Mye99], achieving $O(mn/w)$ time, the one we present here can be extended to more complex patterns, as we see next.

Assume that we are searching for a more complex pattern, for which we have a bit-parallel simulation algorithm that updates D upon text character c, $D \leftarrow f(D, c)$. In order to search for the same pattern with Levenshtein distance threshold k, we must set up $k+1$ copies of the NFA and draw the appropriate transitions among them. The vertical transitions are easy because they go from state j in row $i-1$ to state j in row i. The "diagonal" transitions are more complicated: If we can go by any character c from state j to state j', then our "diagonal" transitions must link state j of row $i-1$ to state j' in row i.

Let us compute $T[D] = \mid_{c \in \Sigma} f(D, c)$, that is, all states reachable from those in D by some character. (Actually T coincides with its definition in Glushkov's simulation of Section 4.4.) This may require exponential space for a general network or regular expression, but also can be as simple as $T[D] = (D << 1) \mid 1$, as it is the case for all the simpler types of patterns we have considered. Hence, the approximate search algorithm maintains bit masks $D_0 \ldots D_k$ and updates them upon reading text character c using

$$D'_0 \leftarrow f(D_0, c) \ ,$$

and, for every $i \in 1 \ldots k$, in increasing order,

$$D'_i \leftarrow f(D_i, c) \mid D_{i-1} \mid T[D_{i-1}] \mid T[D'_{i-1}] \ ,$$

signaling an occurrence whenever D_k has final states activated. The initialization is as follows: D_0 is initialized as in the basic algorithm and D_i is initialized as $D_i \leftarrow D_{i-1} \mid T[D_{i-1}]$. This corresponds to propagating by the diagonal ε-transitions the initially active states in D_0. As a concrete example for a complex pattern, $f(D, c) = T[D] \mathbin{\&} B[c]$ in Glushkov's simulation.

7.2 General Edit Distance

When the δ costs are general real values, the situation is significantly more complicated. A general approach that works well is to consider the search process as a graph distance problem. In the case of simple strings, this graph formulation boils down to the well-known dynamic programming matrix [Sel80].

Let us consider the approximate search for the pattern of Figure 4. Figure 12 illustrates how to transform the search for that NFA into a graph distance problem, with an example text $\mathcal{T}$ = "ACCED". The NFA has been rotated and replicated $n+1$ times. All the original NFA transitions have been preserved as graph edges representing insertions in the text. The ε-transitions, consequently, cost $\delta(\varepsilon, \varepsilon) = 0$, while those labeled by character c cost $\delta(\varepsilon, c)$. We have added horizontal graph edges representing deletions of text characters, so those at the j-th column cost $\delta(\mathcal{T}_j, \varepsilon)$. Finally, for every NFA transition labeled by character c between states i and i', we have added graph edges between those nodes in columns j and $j+1$, respectively, for every j. The cost of those edges is $\delta(\mathcal{T}_j, c)$. Note that some horizontal edge costs are actually due to the latter case applied to self-loops. In particular, all the first row edges cost zero because of the self-loops in the NFA.

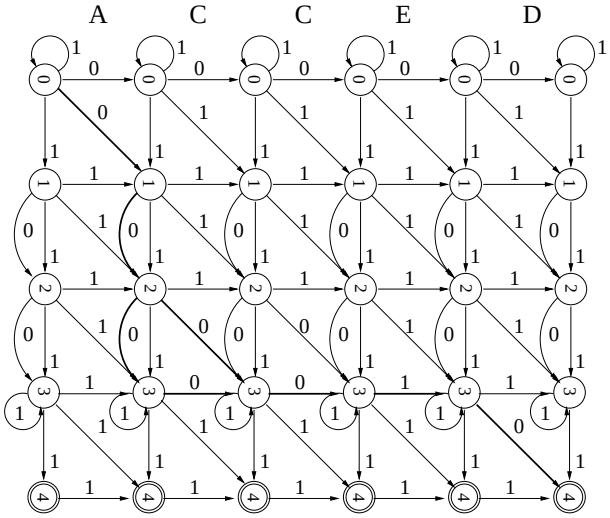


Figure 12: The approximate search for "AB?C*D" as a graph distance problem using Levenshtein costs. We show two paths of cost 1 from the upper left node to a final node in the last column.

Under this model, the edit distance between the pattern and $\mathcal{T}_{1...j}$ is the minimum graph distance between the upper left node and any node at column j corresponding to a final NFA state. If the NFA has an initial self-loop, what we get is actually the lowest distance between the pattern and $\mathcal{T}_{i...j}$, for any i (because the top horizontal edges cost zero). Hence, we report an approximate occurrence ending at text position j whenever the graph distance to any final NFA state at column j is $\leq k$.

In Figure 12 we have shown how the pattern occurs with threshold 1 at the end of the text $\mathcal{T}$ = "ACCED" (for simplicity we have used Levenshtein costs for the example). First, the "A" is matched at cost zero. Then we skip the optional "B" in the pattern, at cost zero. Then we match "C" by following a diagonal edge, again at cost zero (there are two graph paths to do this, corresponding to two NFA paths to obtain the first "C"). Then, since "C" is repeatable, the self-loop has produced a horizontal edge, which costs zero in that column. At this point, there are two equally good choices, both of which are expressed by the horizontal edge costing 1. One choice is to delete text character "E". The other is to take a third repetition of "C", and replace the "E" in the text by such third repetition of "C". In either case, we match "D" to finish.

Observe that it is possible to compute graph distance column by column, since there are no right-to-left edges in the graph. Therefore, we only require $O(m)$ space for the algorithm (assuming Thompson's construction and hence $O(m)$ edges per column), and never build the complete graph. For computing graph distances inside a given column, we can use any classical algorithm such as Dijkstra's to obtain overall $O(mn \log m)$ time complexity [CLR90]. If the pattern does not have backward transitions (such as self-loops and Kleene closures), then we can process the nodes of each column in topological ordering (that is, never process the target node of an edge before its source), and compute all graph distances in $O(mn)$ time. These patterns include network expressions. Actually, it is possible to obtain $O(mn)$ time even in the presence of backward transitions [MM89], but it is slower and more complex. In the case of integer costs, it is possible to further improve this result to $O(mn/\log_{k+2} n)$ time [WMM95].

7.3 Discussion

In this section we have considered approximate searching for all the different complex patterns considered in the rest of the paper. We have focused on sequential searching without skipping characters, but these techniques can be applied as well and are extremely important in practice.

The general technique developed in Section 6 can be immediately applied to approximate searching, since we can use any NFA simulation and this includes the NFAs developed in this section. The graph technique of Section 7.2 can also be adapted: It is enough to compute a new graph column every time we descend by a trie edge, and to stack the computed columns so as to be able to backtrack in the trie.

There is a potential efficiency problem, however, as the backtracking time is not only possibly exponential in the length of the pattern, but also surely exponential in k. For example, under Levenshtein distance, we examine at least all the trie nodes up to depth $k+1$, since no NFA can run out of active states before that depth. This means that, for long enough texts, we pay $O(\sigma^{k+1})$ time at the very least.

A technique that can be used to alleviate this exponential problem is *pattern partitioning*. The idea is that, if a string pattern matches the text within Levenshtein distance k, and we partition the string into j parts, for $1 \leq j \leq k+1$, then at least one of those parts will match inside the occurrence within distance $\lfloor k/j \rfloor$ [NBY00]. Therefore, we can search for the j pattern parts by using backtracking, and check for complete occurrences in the areas surrounding the occurrences of the parts. As j increases, the backtracking cost decreases (as it is exponential in k/j) but the verification cost increases, so there is an optimum j value. The idea can be extended to general edit distance and to more complex patterns, although it is less clear how to do so as the pattern gets more complicated. If only classes of characters, optional and repeatable characters, and bounded length gaps, are used, it is still possible to split the pattern into j consecutive parts, albeit how to find a good partition is less clear than for a simple string. For network and regular expressions, finding a partition into consecutive pieces is even harder, although some solutions have been proposed [Nav01b].

Pattern partitioning, especially the case $j = k+1$ (for Levenshtein distance), is also interesting for sequential approximate pattern matching, because it converts the problem into *exact* searching for $k+1$ pattern parts [Nav01a]. This can be combined with the techniques of Section 5.1 to get very efficient algorithms for complex patterns. This works better for a smaller k, as otherwise we

search for many short patterns and the number of candidate positions to verify increases.

Finally, for Levenshtein distance, the techniques developed in Section 5.2 can be immediately adapted to approximate searching, as we have expressed the search in terms of an NFA simulation. As explained in Section 5.3, however, a real improvement is not guaranteed. In particular, the speedup will not work well for high k thresholds, since we read at least $k+1$ characters per window, which is of length $\ell_{min} - k$.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BYG92] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[BYG96] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[BYN99] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[CCG+94] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecrocq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[Glu61] V.-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[GOM98] K. Guimaraes, P. Oliva, and G. Myers. Reporting exact and approximate regular expression matches. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM'98)*, LNCS 1448, pages 91–103, 1998.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[HBFB99] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Research*, 27:215–219, 1999.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.

[MM89] G. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51:7–37, 1989.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[Mye92] G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):430–448, 1992.

[Mye95] G. Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01a] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[Nav01b] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[NBY00] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[NR00] G. Navarro and M. Raffinot. Fast regular expression search. In *Proc. 3rd Workshop on Algorithm Engineering*, LNCS 1668, pages 199–213, 1000.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5(4), 2000. `http://www.jea.acm.org`.

[NR01] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proc. 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 1–12, 2001.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[NR03] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137, 1985.

[Wat96] B. Watson. A new regular grammar pattern matching algorithm. In *Proc 4th Annual European Symposium*, LNCS 1136, pages 364–377, 1996.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[WMM95] S. Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346–360, 1995.

[Yao79] A. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.