A Unified View to Pattern Matching Problems*

Ricardo Baeza-Yates
Dept. de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120
Santiago, Chile
E-mail: rbaeza@dcc.uchile.cl

Abstract

We present a unified view to sequential algorithms for many pattern matching problems, using a bit-wise simulation of a non-deterministic finite automaton (NFA) built from the pattern which uses the text as input. This approach gives very fast practical algorithms which have good complexity for small patterns on a RAM machine with word length $O(\log n)$, where n is the size of the text. For generalized string matching the time complexity is $O(mn/\log n)$ which for small patterns is linear. For approximate string matching we show that the two main used approaches to the problem are variations of the NFA simulation. For this case we present a different simulation technique which gives a running time of $O(n)$ independently of the maximal number of errors allowed, k, for small patterns. This algorithm improves the best bit-wise or comparison based algorithms of running time $O(kn)$ and can be used as a basic block for algorithms with good average case behavior. We also formalize previous bit-wise simulation of general NFAs achieving $O(mn \log\log n/\log n)$ time.

*This work was partially funded by Fondecyt Chilean Grant 95-0622.

1 Introduction

Pattern matching is an important problem in many different areas. The solutions to this problem differ if the algorithm has to be on-line (that is, the text is not known in advance) or off-line (the text can be preprocessed). In this paper we are interested in the first kind of algorithms. The most simple case is exact matching, which can be generalized to allow don't care characters, wildcards or a maximum number of errors (mismatches, deletions or insertions). All these variations can be described using a regular expression. In this case, the most simple searching algorithm is to construct a NFA from the regular expression and then run the NFA using the text as input [Tho68]. If the size of the regular expression is m, the NFA is built in $O(m)$ time and the NFA simulation requires $O(nm)$ time, being n the size of the text. The algorithm at any stage keeps track of all the active states of the NFA, which in the worst case are $O(m)$.

An efficient way to speed up the NFA simulation is using a bit vector to keep track of all the NFA states. Assuming a RAM machine with word size $w \geq \log n$, we could expect to speed up the simulation at most to $O(mn/\log n)$ time. However, this is not always possible if we want limited extra space or the regular expression is arbitrary. NFA simulations of exact string matching are implicit in [BYG92], don't care characters in [Pin85, Abr87, BYG92, WM92], approximate string matching and general regular expressions in [WM92, WMM95]. The first cases achieve the $O(mn/\log n)$ running time, but for approximate pattern matching the running time is $O(kmn/\log n)$ where k is the maximum number of errors allowed, which could be $O(m)$. In all these cases, the simulation is done by just using bit-wise operations like shifts, and/ors, additions, etc; where comparisons are only used in the preprocessing phase (NFA construction).

In this paper we unify all the approaches already mentioned in basically three cases: linear NFAs for generalized string matching, approximate string matching NFAs and general NFAs. For the second case, we show that it is possible to achieve running time $O(mn\log k/\log n)$ where k is the maximal number of errors in a variation of the RAM model and $O(n)$ for small patterns. This should be compared with the previous best algorithms of $O(kn)$ [Ukk85a, LV88, GP90]. Also, in this case the number of states is $O(mk)$, but we reduce it to $O(m\log k)$ space by using the regularity of the NFA structure. For the generic case we formalize the approach presented in [WM92] to show that using $O(m)$ space, it is possible to achieve $O(mn\log m/\log n)$ running time. In all these cases we assume a finite alphabet, and the extension to arbitrary alphabets multiplies the complexity by an $O(\log m)$ factor. These results show that the comparison based model might not be the best for some problems and that we can use the intrinsic bit parallelism of the RAM model [BY92, WMM95], similarly to improvements achieved for sorting [FW93, AHNR95].

2 Preliminaries

There are several techniques to design pattern matching algorithms. Instead of using a NFA, we can convert it to a DFA, and run a DFA using the text as input. For a finite size alphabet, the running time is $O(n)$. However, the NFA to DFA conversion requires in some cases $O(2^m)$

time, where m is the size of the pattern. Therefore, even for small patterns of size $O(\log n)$ the total running time could be non-linear. More over, for some of problems studied here, in particular approximate string matching, the number of states of the DFA is exponential in the size of the NFA [Ukk85b].

There are ad-hoc comparison based algorithms for string matching with or without errors. Classic algorithms include Knuth-Morris-Pratt and Boyer-Moore for exact matching and dynamic programming for the case with errors. We refer the reader to [GBY91, Chapter 7] for more details. For string matching problems, non-comparison based algorithms include the use of matrix multiplication [FP74, Kar93], or bit-wise techniques as in this paper [Abr87, Der95, Wri94, WMM95]. The approach taken here tries to use the same technique for different problems.

We use a RAM machine with word size $w \geq \log_2 n$, for any text size n. We use the uniform cost model for a restricted set of operations including comparison, addition, subtraction, bitwise AND and OR, and unrestricted bit shifts (that is, the number of bits to shift is a parameter to the operation) [AHNR95]. That is, all operations on w bits take constant time, which is the case in the normal underlying hardware. This is the same assumption used for comparison-based algorithms in a RAM, so we can compare the time and space complexity. All these operations are in AC^0, that is, they can be implemented through constant-depth, polynomial-size circuits with unbounded fan-in.

In the following we use Σ as a finite size alphabet. This is the usual case in practice. Some common examples are ASCII (128 or 256), proteins (20) and DNA (4). If we need to handle arbitrary alphabets, we use the so called effective alphabet, which is the set of different symbols present in the pattern (which at most are m). For this we build a sorted table of size m where we map the effective alphabet to a given index, having an extra index for symbols not in the pattern. By using binary search in this table, we can handle arbitrary alphabets. This technique increases the searching time by a factor $O(\log_2 m)$ by using $O(m)$ extra space and $O(m \log m)$ preprocessing time. This can be reduced to a constant factor on the worst case by using perfect hashing, increasing the preprocessing time, but still depending only on m.

3 Generalized String Matching

In this section we unify several known results in a single problem and solution, where the NFA has size proportional to the length of the pattern. Let the pattern be a sequence of elements, each element being a:

- A set or class of characters, complemented or not, including Σ (don't care case).
- Any class, repeated zero or more times (Kleene or star closure), denoted by $*$ (class wildcards).

For example, the pattern `[Tt][^aeiou]`Σ`[0-9]`$*$ finds all sequences starting with T or t, not followed by a vowel, followed by any character and then a sequence of 0 or more digits.

Almost the same problem is considered in [WM92], where the wildcards were always of the form Σ^*. This problem includes exact string matching, generalized string matching [Abr87, BYG92], don't care characters [Pin85, MBY91, BYG92], wildcards [WM92], the followed-by problem (one string followed by another) [MBY91] and subsequence searching [BY91]. The NFA for this problem is very simple, consisting in $m+1$ states, where m is the number of non-starred elements. Figure 1 shows some examples, where we use P_i for the non-starred elements and S_i for the starred elements. We use M to denote the total length of the pattern.

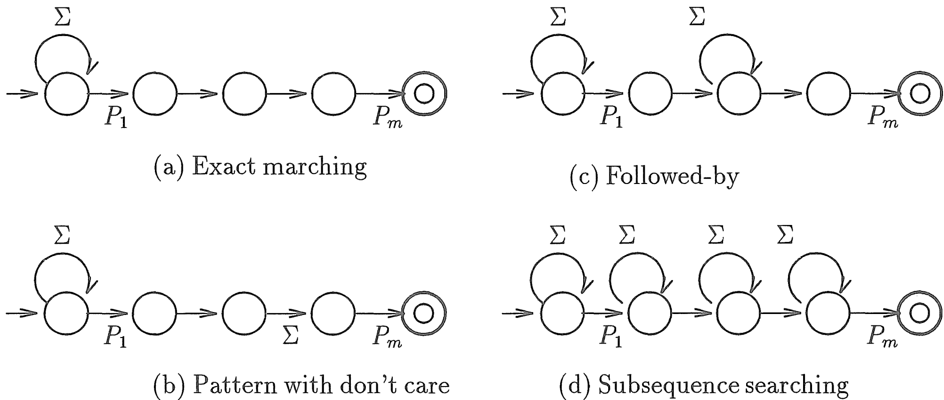

Figure 1: NFA examples of generalized string matching.

The NFA simulation is a generalization of the shift-or/and algorithm [BYG92, WM92]. Here we use the shift-and variant. Every state is one bit in a bit vector, using the enumeration induced by the automaton. So, a non-starred transition is simulated by shifting and and-ing the vector with the entry for the current symbol x in the text in a table T_1. T_1 has Σ entries, each with m bits. The entry $T_1[x]$ has a bit 1 in position i if $x \in P_i$. Similarly, the starred transitions are handled by just *and*-ing the vector with an entry in a similar table T_2, where a bit is set to 1 if $x \in S_i$. The final algorithm is shown in C-like pseudocode in Figure 2, where we apply a bit-wise *or* to both cases. Note that the initial state will always have a Σ transition to itself, to allow finding the pattern in all possible positions of the text. A match is reported if the $(m+1)$-th bit is a one.

If $m \geq w$, we use multiple words concatenating them, taken care of the inter-word transitions. On average it is better to apply the algorithm only on the first word, storing the second only when needed [WM92]. The preprocessing time and total space needed is $O((\Sigma + M)m/w) = O((\Sigma + M)m/\log n)$. The running time is $O(mn/w) = O(mn/\log n)$. For patterns with $m = O(\log n)$, the time complexity is linear.

4 Approximate String Matching

In this section we present fast NFA simulations for approximate string matching using a NFA of size $O(m^2)$. Approximate string matching is one of the main problems in classical string

```
T_1, T_2 <- Preprocess(pattern)
s <- 1; mask <- 10^m
for all characters x of the text
{
     s <- ((shift left s by 1) and T_1[x]) or (s and T_2[x]))
     if s and mask then Report match
}
```

Figure 2: Algorithm for Generalized String Matching.

algorithms. Given a text of length n, a pattern of length m, and a maximal number of errors allowed, k, we want to find all text positions where the pattern matches the text up to k errors. Errors can be be substituting, deleting or inserting a character. We distinguish the case of only mismatches (just substitutions), and the case where we also allow insertion and deletions. In both cases the NFA has $O(mk)$ states and is very regular. This regularity allows to use only $O(m \log k)$ bits to describe the automaton. We consider occurrences starting with a match and finishing as soon as there are at least $m - k$ matches.

4.1 String Matching with Mismatches

Consider the NFA for searching $p_1p_2p_3p_4$ with at most $k = 2$ mismatches, shown in Figure 3. Every row denotes the number of errors seen. The first 0, the second 1, and so on. Solid horizontal arrows represent matching a character. Solid diagonal arrows represent replacing the character of the text by the corresponding character in the pattern (that is, a mismatch). This NFA has clearly $O(mk)$ states and can be easily built in $O(mk)$ time given its highly regular structure.

This problem was considered in [BYG92], and here we summarize that solution. Instead of using one bit per state, we count the number of mismatches seen per column. Because we have at most k mismatches, we need just $B = \lceil \log(k+1) \rceil$ bits per counter. Then, the new value of the counter for column i is what the column $i-1$ had before plus zero if we had a match on that position or one otherwise. We can add these counters in parallel by concatenating them in a single bit vector and having an extra bit between counters to avoid trespassing carries from one counter to the next. This can be easily extended to classes of characters (mismatch implies $x \notin P_i$). The algorithm is shown in Figure 4 where $T_1[x]$ has a 1 in position i if $x \neq p_i$. The extra space and preprocessing time is $O((\Sigma + m) \log k / \log n)$. The running time is $O(mn \log k / \log n)$ which is $O(n)$ for patterns of size upto $m = O(\log n / \log \log n)$.

A different bit-wise simulation that requires $O(kn)$ time for small patterns is given by Dermouche [Der95]. We can combine the problem of mismatches and generalized patterns with wildcards, computing for every column the minimum of two values. That is, using the same notation as for generalized string matching, we have for every counter

$$C_i = \min(C_{i-1} + (text \notin P_{i-1}), C_i \text{ if } text \in S_i, k+1)$$

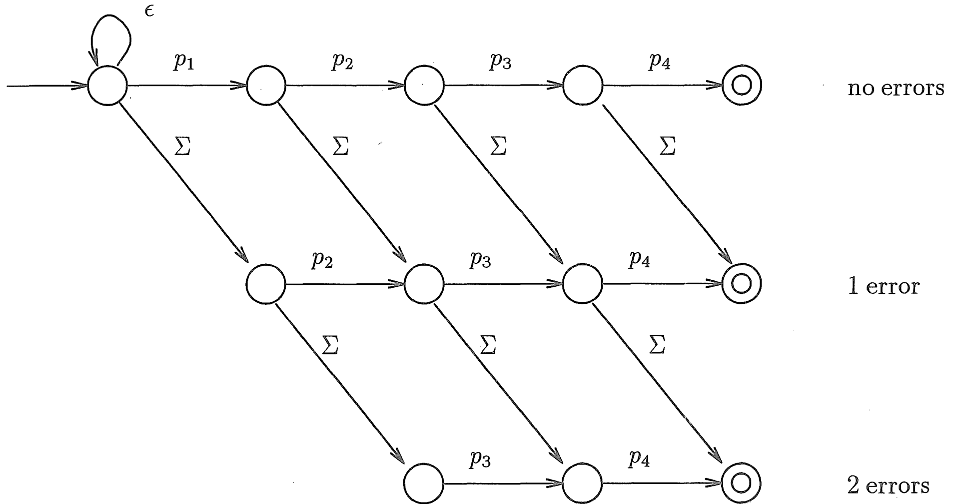

Figure 3: NFA for string matching with mismatches.

B, T_1, $mask_2 \leftarrow Preprocess(pattern,\ k)$
$s \leftarrow 0$; $mask_1 \leftarrow (01^B)^m$
for all characters x in the text
}
    $s \leftarrow$ ((shift left s by $(B+1)$) + $T_1[x]$) and $mask_1$
    if s and $mask_2$ then Report match
}

Figure 4: Algorithm for Generalized String Matching with Mismatches (no wildcards).

To achieve the same time complexity as before we need to do parallel mins. This can be done in $O(1)$ time by using several tricks in about ten machine instructions. With a different simulation technique we can have wildcards without using this parallel min technique, but the searching time increases [WM92].

This algorithm can also be used to count the number of mismatches between the pattern and all possible positions of the text. This is considered in [Abr87, Kos87] where a time complexity of $O(n\sqrt{m}\log m\sqrt{\log\log m})$ is given. Karloff [Kar93] improves this result to $O((n/\epsilon^2)\log^3 m)$ time for any $\epsilon > 0$, where ϵ is the maximal error tolerated (that is, the occurrences are also approximated). He also presents randomized algorithms that are better on average, but all these algorithms use matrix operations. Our algorithm requires in this case $O(mn\log m/\log n)$ which is better for patterns up to size $O(\log n(\log\log n)^2)$ and make no mistakes ($\epsilon = 0$).

4.2 String Matching with Errors

Consider now the NFA for searching $p_1p_2p_3p_4$ with at most $k = 2$ errors shown in Figure 5. As for the case of only mismatches, every row denotes the number of errors seen. The structure of the automaton is similar to the case of mismatches adding two additional transitions per state. Dashed diagonal arrows represent deleting a character of the pattern (empty transition), while dashed vertical arrows represent inserting a character. Let $s_{i,j}$ be true if the state on row i (number of errors) and column j (position in the pattern) is active. Then, after reading a new character, the new state values, s', are given by

$$s'_{i,j} = (s_{i,j-1} \text{ if } p_{j-1} = text) \mid s_{i-1,j} \mid s_{i-1,j-1} \mid s'_{i-1,j-1}$$

where the first term represents a match, the second when we insert a character, the third when we substitute a character and the last, when we delete a character. Note that because we can delete a character in the text at any time, we have to use the current value of s, that is s', instead of the previous value.

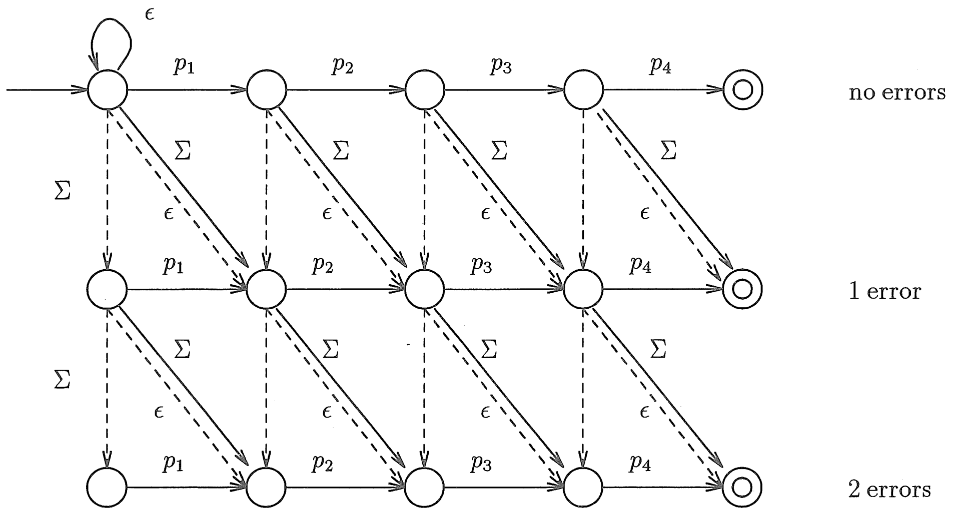

Figure 5: NFA for approximate string matching.

First, note that every row by itself matches the pattern. So, we can simulate every row using the technique shown for generalized string matching. By computing the dependences between rows, we obtain Wu & Manber's algorithm for approximate string matching[WM92]. In this case, the bit vectors R_k, being k the corresponding row, are updated to R'_k using the following formula

$$\vec{R'}_k = (shift(\vec{R}_k) \ \& \ T[text[j]]) \ or \ \vec{R}_{k-1} \ or \ shift(\vec{R}_{k-1} \ or \ \vec{R'}_{k-1})$$

where the terms are written in the same order as before. Initially $\vec{R}_k = 1^k0^{m-k}$ (k ones meaning up to k deletions). The main drawback is the dependency on R'_{k-1}, due to the

empty diagonal transitions, which does not allow to compute the updated values of R in parallel. The time complexity of this algorithm is $O(kmn/\log n)$ using $O((k+\Sigma)m/\log n)$ space for a RAM machine of word length $O(\log n)$.

Another possibility is to simulate the NFA by columns as for mismatches. Let define the value of a column as the smallest active state level per column (or equivalently, the smallest error valid in each column). Then, the state of the search are m numbers C_i on the range $0...k+1$. To update every column to C'_i after we read a new text character we use

$$C'_i \quad = \quad \min(\;\; C_{i-1} + (text[j] != patt[i-1]), \;\; C_i + 1, \;\; C'_{i-1} + 1 \;\;)$$

where the first term is either a match or a substitution, the second an insertion and the last one a deletion. Initially $C_0 = 0$ and $C'_0 = C_0$. Readers familiar with the problem will recognize immediately this solution as a variation of the well known dynamic programming approach to approximate string matching. The time complexity is $O(nm)$. Smarter versions of this approach run in worst-case time $O(kn)$. This simulation is also related to Ukkonen's automata approach [Ukk85b]. The main disadvantage of this approach is again the dependency on C'_{i-1} which does not allow a computation of C' in parallel.

The dependency on both cases is due to the empty transitions along the diagonals. The solution is to simulate the automaton using diagonals, such that each diagonal captures the ϵ-closure. This idea is used next to find a faster algorithm.

Suppose we use just the diagonals of length $k+1$ of the automaton. Let D_i be the highest row value (smallest error) active per diagonal. Then, D_1 is always 0 (starting point). The new values for D_i after we read a new text character are given by

$$D'_i = \min(D_i + 1, D_{i+1} + 1, g(D_{i-1}, text))$$

where $g(D_{i-1}, text)$ takes the value D_{i-1} if there is a position j such that $j \geq D_i$ and $p_{i+j} = text$; otherwise takes the value $k+1$. This later value is used if there are no active states in the diagonal. The first term of the min expression represents a substitution which follows the same diagonal. The second term represents the insertion of a character coming from the next diagonal above. Finally, the last term represents matching the character in previous diagonals above the current level (more than one when there are characters repeated in the pattern). This simulation has the advantage that can be computed in parallel for all i. The function g, considering that the alphabet is finite and that D_i can take only $k+2$ values, can be precomputed based solely in the pattern. So, we would like to use bit-wise parallelism to simulate the automaton. However, the drawback now is that although we can compute in parallel several minima, it is not possible to evaluate the function g in parallel (each argument having $O(\log k)$ bits) without using exponential space. However, implementing g by hardware (still in AC^0) and preprocessing the valid values of pairs $(j, text)$ in a table, the time complexity for this algorithm is $O(mn \log k/\log n)$ which is $O(n)$ for patterns of size up to $O(\log n/\log\log n)$.

A different bit-wise parallelism is considered by Wright [Wri94], which packs three diagonals of the dynamic programming matrix in one word (the diagonals perpendicular to ours), which also have the dependency problem. In a companion paper [BYN96], we show how to represent D_i using $O(k)$ bits in the usual RAM model and evaluating the recurrence

in $O(1)$ operations, obtaining the faster known algorithm for approximate string matching when $(m-k)(k+2) \leq w$. We can extend this problem to allow a generalized pattern with errors. For that we just include another function h that is equivalent to T_2, such that

$$R'_i = \min(R_i + 1, R_{i+1} + 1, g(R_{i-1}, text), h(R_i, text))$$

with $h(R_i, text)$ being R_i if $text \in S_i$ or $k+1$ otherwise.

5 Regular Expressions

In this section we address general automata, where the NFA size is proportional to the length of the regular expression to be searched. The standard algorithm [Tho68] requires $O(mn)$ time in the worst case. The first sublinear algorithm is given by Myers [Mye92] which uses a four russians approach. In [WM92] a bit-wise simulation for a generic NFA built using the standard Thompson's algorithm is given. By construction, all states either have one deterministic transition or two non-deterministic ϵ-transitions. So, the deterministic transitions are handled by shifting left by one bit the state vector and *and*-ing it with the entry of a table T_1 as in previous cases. Each entry in T_1 also cancels all states with non-deterministic transitions. To handle all the ϵ-transitions, they use tables to map one or two bytes of the new state vector into a new vector. In practice they use two bytes, which can be considered as a constant or $w/2$ for 32-bit words.

The above result can be understood better by using as a parameter the maximum space S to be used. Then, we can use blocks of size $\log S$ bits to divide the bit vector of size m, where m is the size of the NFA. Using the previous algorithm, the time complexity is $O(mn \log S/\log n)$. Using S of $O(\log n)$, which means a constant number of words, we get time complexity $O(mn \log\log n/\log n)$. Any polylog number of words only decreases the searching time on a constant. The complexity can be improved if a parallel mapping operation in blocks of bits is available (easily implemented by hardware and also in AC^0).

This algorithm can be improved in several ways. One of them is to reduce the number of ϵ-transitions based on states with out-degree greater than 1. This leads to an interesting optimization problem which is being studied. This algorithm can be extended to consider errors as is mentioned also in [WM92, WMM95] (see also [MM89]).

6 Concluding Remarks

In this paper we have unified several algorithms using a single technique: bit-wise simulation of NFAs. We have also shown that many algorithms for approximate string matching, including dynamic programming, can be seen as specialized bit-wise simulations. An additional result is the relevance of the machine model to be used. In an augmented RAM model, the new algorithm presented for string matching with errors can be a factor of $O(m/\log m)$ faster, which is another example where the traditional RAM comparison model does not seem to be the best choice.

Another issue is related to the effective use of the intrinsic bit-parallelism of the RAM model. Several new parallelism models which try to improve upon the classic PRAM model, do not address well the fact that many operations can be executed in constant time on $O(\log n)$ bits or that less memory is needed in practice. For example, our new algorithm can be considered as a parallel algorithm for m processors, each one only using $O(\log k)$ bits. Then, we would like to compare this algorithm using the fact that the operations are over small words and not over $O(\log n)$ size words.

There are several extensions to the basic NFA simulations shown here. They can be extended to more general cases and used as building blocks to handle other type of patterns or to devise fast expected time algorithms. We address briefly these ideas.

First, we can extend all the simulations to handle multiple patterns by simulating all NFAs at the same time (multiplying the time complexity by the number of patterns). For large patterns we can partition the automata in several words. For approximate string matching, another possibility is to divide the pattern in ℓ pieces of length m/ℓ where up to k/ℓ errors are allowed, searching all pieces at once and checking the whole pattern if a piece is found. This idea is a generalization of ideas presented in [WM92, BYP92, Mye94] and together with automata partition and other ideas is used in [BYN96] to design a fast expected time algorithm for approximate string matching.

Acknowledgements

We acknowledge the helpful discussions with Gaston Gonnet, Udi Manber and Gonzalo Navarro.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J on Computing*, 16:1039–1051, 1987.

[AHNR95] A. Anderson, T. Hagerup, S. Nilsson, and R. Rajeev. Sorting in linear time? In *STOC'95*, pages 427–436, Las Vegas, NE, 1995.

[BY91] R. Baeza-Yates. Searching subsequences (note). *Theoretical Computer Science*, 78:363–376, 1991.

[BY92] R. Baeza-Yates. Text retrieval: Theory and practice. In J. van Leeuwen, editor, *12th IFIP World Computer Congress, Volume I*, volume Algorithms, Software, Architecture, pages 465–476, Madrid, Spain, September 1992. Elsevier Science.

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

[BYN96] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Combinatorial Pattern Matching (CPM'96)*, Irvine, CA, Jun 1996.

[BYP92] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate pattern matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching*, Lecture Notes in Computer Science 644, pages 185–192, Tucson, AZ, April/May 1992. Springer Verlag.

[Der95] A Dermouche. A fast algorithm for string matching with mismatches. *Information Processing Letters*, 55(1):105–110, July 1995.

[FP74] M. Fischer and M. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings 7)*, volume 7, pages 113–125. American Mathematical Society, Providence, RI, 1974.

[FW93] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, Wokingham, UK, 1991. (second edition).

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.

[Kar93] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48:53–60, 1993.

[Kos87] S.R. Kosaraju. Efficient string matching. Manuscript, Johns Hopkins University, 1987.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *JCSS*, 37:63–78, 1988.

[MBY91] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, February 1991.

[MM89] E. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[Mye92] E. Myers. A four-russians algorithm for regular expression pattern matching. *JACM*, 39(2):430–448, 1992.

[Mye94] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

[Pin85] R. Pinter. Efficient string matching with don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 11–29. Springer-Verlag, 1985.

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

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

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

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

[Wri94] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.