A Partial Deterministic Automaton for Approximate String Matching

Gonzalo Navarro[1,2]

[1] Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl
[2] This work has been supported in part by Fondecyt grants 1-950622 and 1-960881.

Abstract. One of the simplest approaches to approximate string matching is to consider the associated non-deterministic finite automaton and make it deterministic. Besides automaton generation, the search time is $O(n)$ in the worst case, where n is the text size. This solution is mentioned in the classical literature but has not been further pursued, due to the large number of automaton states that may be generated.

We study the idea of generating the deterministic automaton on the fly. That is, we only generate the states that are actually reached when the text is traversed. We show that this limits drastically the number of states actually generated. Moreover, the algorithm is competitive, being the fastest one for intermediate error ratios and pattern lengths.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc.

The problem is defined as follows: given a *text* of length n and a *pattern* of length m (both sequences over an alphabet Σ of size σ), and given 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 replacing, deleting or inserting a character. We call $\alpha = k/m$ the *error ratio*. We are interested in the on-line problem, i.e. the text is not known in advance and therefore only the pattern can be preprocessed.

The classical solution to this problem is $O(mn)$ running time [16, 17]. It maintains a set of $m+1$ values C_i ($i \in 0..m$), which at any moment represent the amount of errors if $pattern[1..i]$ is matched against the best suffix of the already traversed text. A match is reported whenever $C_m \leq k$. C_i is initialized at i, and the new C'_i values are computed in $O(m)$ time from the C_i current values using the formula

$$C'_i = \text{if } (pattern[i] == text\_char) \text{ then } C_{i-1} \text{ else } 1 + \min(C_{i-1}, C'_{i-1}, C_i) \quad (1)$$

where it is assumed that C_0 is always 0.

Further improvements in the last years achieved $O(kn)$ time in the average case [23, 9] or even in the worst case [22, 9, 13, 14] by taking advantage of the

properties of the dynamic programming matrix. In the same trend is [6], with average time complexity $O(kn/\sqrt{\sigma})$.

Especially interesting to this work is [23], which defined "active" C_i values. Active values are those from 1 to the last i having $C_i \leq k$. The outcome of the algorithm depends only on the active values. It was conjectured (and later proved in [6, 4]) that the average number of active values is $O(k)$. By working only on the active values, $O(kn)$ average time is achieved.

Other approaches attempt to filter the text, reducing the area in which dynamic programming needs to be used [21, 20, 19, 7, 8, 11, 5, 18]. These algorithms achieve sublinear expected time in many cases ($O(kn \log_\sigma m/m)$ is a typical figure) for moderate k/m ratios, but the filtration is not effective for larger ratios.

Yet other approaches use bit-parallelism [1, 27] to reduce the number of operations [24, 28]. In [28], $O(kn/w)$ average time is achieved, where w is the number of bits in the computer word.

A radically different approach to the problem is to look at it as a regular grammar matching problem. Consider the NFA (non-deterministic finite automaton) for searching `patt` with at most $k = 2$ errors shown in Figure 1. Every row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. At each iteration, a new text character is considered and the automaton changes its states. Horizontal arrows represent matching a character (they can only be followed if the corresponding match occurs), vertical arrows represent inserting a character in the pattern, solid diagonal arrows represent replacing a character, and dashed diagonal arrows represent deleting a character of the pattern (they are ϵ-transitions[3], since we delete the character from the pattern without advancing in the text). Finally, the loop transition at the initial state allows to consider any character as a potential starting point of a match. We say that a state is *active* if the automaton can reach that state after processing the text up to now. The automaton accepts a character (as the end of a match) whenever a rightmost state is active. This NFA has $(m + 1)(k + 1)$ states. Initially, the active states at row i ($i \in 0..k$) are at columns from 0 to i, to represent the deletion of the first i characters of the pattern.

If we look at the columns of this automaton, we see that once a state is active, all subsequent states in the same column are active too, and therefore it is possible to define a column by the minimum row active at that column. The values obtained are precisely the C_i already mentioned. Therefore, simulating this NFA by columns leads to the classical algorithm.

Although we have cited [23] to mention the idea of active values, the central point of the paper is to make this automaton deterministic. Once this is done, the search time is $O(n)$. However, the number of states of the DFA (deterministic finite automaton) can be huge. Although the regularities make the total number of states far below the maximum $O\left(2^{(m+1)(k+1)}\right)$, the bounds are still discouraging. In [23], a bound of $O(\min(3^m, (2\sigma m)^k m))$ is obtained. In [15] it

[3] ϵ-transitions are those that the NFA may follow without reading any input.

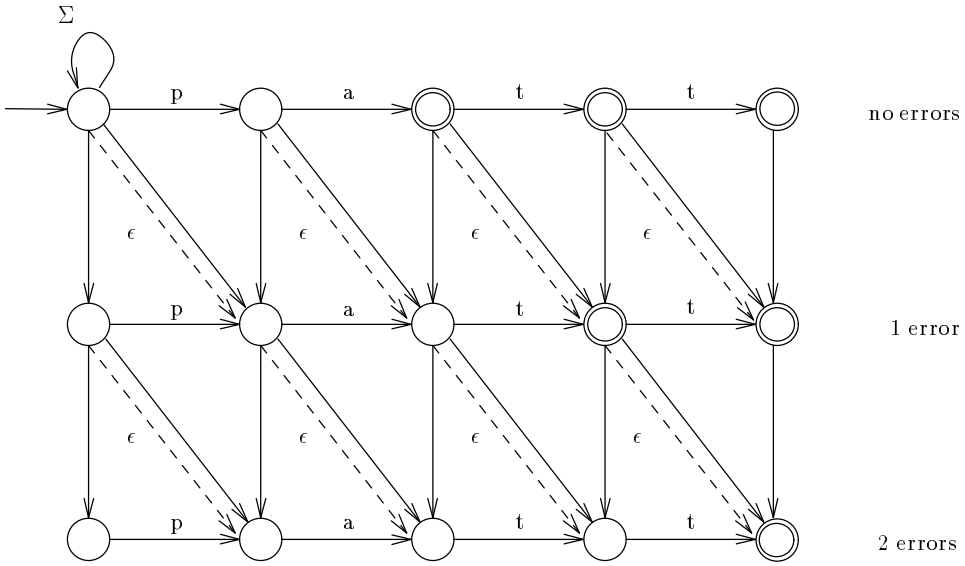

Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character.

is shown that $O((k+2)^{m-k}(k+1)!)$ is also an upper bound. For this reason, it has been attempted to simulate the non-deterministic automaton rather than the deterministic one.

In [26], this automaton is simulated by rows (using bit-parallelism) and $O(kmn/w)$ time complexity is achieved. In [4, 3] it is simulated by diagonals (which allows higher parallelism) and $O(\sqrt{km/w}\ n)$ time is achieved for moderate error ratios.

We return to the idea of building the DFA. However, we observe that most of the states of the DFA are never reached throughout the search. Therefore, instead of building the automaton beforehand and then using it to search the text, we have a partially built automaton. This partial automaton has only the states and transitions that have been reached. As we search, if we find a transition which has not been computed yet, we compute it before proceeding. This is as efficient as building the automaton completely, but the number of states can be much smaller.

The idea of on-the-fly construction of DFAs is not new. For instance, it has been mentioned before, although not exploited, for the more restricted case of string matching allowing character replacements (no deletions nor insertions) [2]. It has also been used for the general problem in [12], where it was implemented on a lazy functional language. They arrive to similar conclusions about performance, although we include more algorithms in the comparison and our test suite is much larger and has no hidden performance factors that could be included in a funcional language implementation.

We explain the algorithm and show experimentally that it outperforms all

the others for intermediate error ratios. This is because for very low error ratios the filtration techniques inspect less than n characters, while for very high error ratios we generate a huge number of states anyway. We also show how the partial automaton grows with the text size.

2 The Partial DFA Algorithm

We begin with some terminology. A *deterministic finite automaton* or *DFA* is a set of *states* connected by *transitions*. Transitions are arrows among states labeled with symbols drawn from an alphabet Σ. There is exactly one transition leaving every state for each alphabet symbol. One of the states is *initial* and some states are *final*. The automaton is run over a text beginning at its initial state. Given each text character, it follows the appropriate transition and reaches a new state. We say that the automaton *accepts* a text position whenever it is at a final state just after that position is read.

A *partial* DFA is a DFA where some transitions are missing. A missing transition means that we still have not computed to which state it should go.

A *configuration* represents the state of the search at a given moment. It can be represented as the set of active or inactive states of the NFA, or as the current (active) values of the the dynamic programming algorithm. Each possible configuration corresponds to a state of the DFA.

Our DFA will have one state per "known" configuration. Only the configurations (states) which are actually reached in the processing of the text will be present.

We begin with a partial DFA with just one state and all missing transitions. The state corresponds to the initial configuration ($C_i = i$). We traverse the text exactly as if we had a complete DFA. The difference is that, whenever we must follow a missing transition, we compute it. That is, we take the current state (we store the configuration that corresponds to each state) and perform an $O(m)$ step of the classical algorithm. This gives us the configuration of a new state. We search the configuration among the known states. If it already exists, we put the previously missing transition pointing to that state. Otherwise, we must first create a new state (with all missing transitions leaving it).

The advantage of such construction is that, although the DFA of a pattern can be very large, only a small portion of the states may be actually reached along the text. Of course, the larger the text, the more states will be generated, but this larger text will compensate for the effort of generating the automaton. Note that in natural language some substrings never appear, no matter how long the text.

The only disadvantage is that once the complete DFA is generated, the configuration to which each state corresponds needs not be stored, while in the partial DFA we need to keep those configurations all the time to be able to generate new transitions and states. This extra space turns out to be about a 25% extra per *generated* state, which is not too much, especially because much less states are generated in the partial DFA (as we show in the experimental section).

On the other hand, the complete DFA algorithm needs to keep all configurations to generate the DFA, and only then can free their space. Therefore, at some point it demands strictly more memory than what the partial DFA algorithm demands along the whole search.

Our algorithm can thus be sketched as follows

```
Search (text, n, pat, m, k)
   Aut ← initial state (configuration C_i = i)
   state ← initial state
   ∀c ∈ Σ, transition(Aut, state, c) ← unknown
   for (i = 0; i < n; i++)
       { nstate ← transition(Aut, state, text[i])
         if (nstate == unknown)
             { nconf ← perform_step(conf(Aut, state), text[i])
               nstate ← state in Aut corresponding to nconf
               if (nstate not found)
                   { nstate ← new state
                     Aut ← Aut ∪ {nstate}
                     ∀c ∈ Σ, transition(Aut, nstate, c) ← unknown
                   }
               transition(Aut, state, text[i]) ← nstate
             }
         state ← nstate
         if (state is final) report match
       }
```

The Σ in the algorithm does not stand for the complete alphabet, but only for those symbols appearing in the pattern, plus one that represents "any symbol not in the pattern". The alphabet is then mapped to the interval $[0..p]$ where $p \leq m$, and therefore creating a new state costs $O(m)$. The text characters are mapped in $O(1)$ time by using a global $O(\sigma)$ size table.

Now we point out some details of the algorithm.

We represent configurations as the current C_i values of the classical dynamic programming algorithm. Therefore the configurations are $O(m)$ size. The action *perform_step* of the algorithm simply makes a step of the classical algorithm on the given configuration, and therefore it is $O(m)$ time. Since we work only on active values, our representation for a column is $[a, c_1, ..., c_a]$, where a is the position of the last active value. As conjectured in [23] and proved in [6, 4], the average number of active values is $O(k)$. However, this is probably not true if we compute each configuration only once.

Once *nconf* has been computed, it is necessary to know whether it corresponds to a state which is already present in the automaton. Therefore, we must search in a set of "known" configurations. This search can be done in time proportional to the length of the searched configuration (i.e. $O(m)$) in the worst case. The data structure to achieve this is a trie on the $a + 1$ "digits" of the configuration.

The structure of this trie is quite particular. The root has $m-k$ subtries (one per possible a value). The subtrie number i has height i. Every non-root node which is not a leaf has at most 3 children. This is because the difference between two consecutive values in the column is -1, 0 or 1 (recall Eq. (1)). This fact, already noted in [23], allows to save a lot of space in the trie implementation. This structure is used in [12].

However, we found that a simple hashing turns out to be the most efficient data structure in practice. We take the hash function over the $a+1$ "digits" of the configuration. Collisions are resolved with a linked list. If the table size and the hash function are well chosen, the average cost is $O(m)$ (to evaluate the hash function), at much less space consumption.

3 Analysis

We call s and t the total number of states and transitions, respectively, in the complete automaton. This is an upper bound to those actually generated (s' and t', respectively). As Ukkonen has shown in [23], the number of states can be upper bounded by $s = O(\min(3^m, (2\sigma m)^k m))$ (the first one comes from the property mentioned before about tries, the second one considers all possible strings at distance $\leq k$ from the pattern). Since we can map Σ to $O(m)$ functionally different symbols, we replace σ by $min(\sigma, m)$ (this is also noted in [15]). In [15], an alternative bound is given by $s = O((k+2)^{m-k}(k+1)!)$. This is clear by looking at the approach of [4], where the NFA is represented by diagonals with $m-k$ numbers in the range $0..k+1$ (the non-full diagonals were discarded in that paper and correspond to the $(k+1)!$ present here).

The number of transitions is therefore $t = O(s \min(m, \sigma))$. We have $s' = O(\min(s, n))$ and $t' = O(\min(t, n))$, since each text character can create at most one state and transition. The space needed by our algorithm is $O(s' \min(m, \sigma) + t') = O(s' \min(m, \sigma))$ in the worst case, i.e. $O(\min(s, n) \min(m, \sigma))$. The algorithm is linear time except for the generation of the states and new transitions, each one costing $O(m)$. This makes the total cost of the algorithm $O(n + s' \min(m, \sigma) + t'm) = O(n + t'm) = O(n + m \min(t, n))$.

It is extremely difficult to set up a correct probabilistic model to compute the number of states that are generated on average after reading n random symbols. We make the simplifying assumption that each new character produces a random transition. This assumption is pessimistic since some transitions are much more probable than others.

The probability of a given transition *not* being generated at a particular text position is $(1 - 1/t)$. Therefore, the average number of generated states after reading n text characters is

$$t' = t\left(1 - \left(1 - \frac{1}{t}\right)^n\right) = t\left(1 - e^{-n/t}\right) + O(1/t)$$

and therefore the average time complexity is $O(n + mt(1 - e^{-n/t}))$. Notice that

$(1-e^{-n/t})$ is the factor by which building our partial DFA is more efficient than building the full DFA. We have three cases now

$[n = o(t)]$ In this case $e^{-n/t} = 1 - n/t + O((n/t)^2)$, which makes the total time cost $O(n + mn(1 + o(1))) = O(mn)$. That is as bad as plain dynamic programming. However, in this case the partial DFA is asymptotically better than the full DFA (i.e. partial/full $= 1 - e^{-n/t} = o(1)$).

$[n = \omega(t)]$ In this case we have that the time is $O(n+mt) = o(mn)$, and therefore we improve dynamic programming. If $n = \Omega(mt)$, then the algorithm is $O(n)$.

$[n = \Theta(t)]$ We have that $(1 - e^{-n/t}) = \Theta(1)$ (i.e. constant) and therefore the total time is $\Theta(mn)$, although the constant is smaller. For instance, if $n = t$ it is $\approx 0.63mn$.

This shows that the algorithm is competitive when the text is large compared to the size of the automaton, which is intuitively clear. However, this analysis is pessimistic and the results are much better in practice, as well as the efficiency ratio among partial and full DFA. We show this in the next section.

4 Experiments

We present some experimental results collected on English text. We first show the growth of the automata and then we compare our algorithm against the others.

The tests were run on a Sun SparcServer-1000 running SunOS 5.5, with 128 Mb of RAM (this implies $w = 32$). We tested English text from the TREC collection [10], selecting the patterns randomly from the same text, at the beginning of non-stopwords. The text was converted to lower-case and all separators to a single space (respecting lines). This filtering is to mimic classical Information Retrieval scenarios. We used 10 Mb of filtered text.

Each data point was obtained by averaging over 10 trials.

4.1 Automaton Growth

We experimentally show in Figure 2 which is the growth of the partial and complete DFAs as the error level increases and as the traversed text increases.

As it can be seen, the size of the automaton grows slowly with the text size after a sharp start. Even after processing 10 Mb of natural language, the sizes are less than 20% of the complete automata (except for very small k where the complete automata are very small anyway). Memory limitations prevented us to compute automata with more than 500,000 states. For $m = 20$ we could not generate the complete automaton past $k = 6$ and for $m = 30$ past $k = 5$. For those larger patterns our improvement is more dramatic, since the partial DFAs are completely manageable up to $k = 10$ or 11, being hundreds of times smaller. For $m = 30$ we could not even compute the partial automata past $k = 18$.

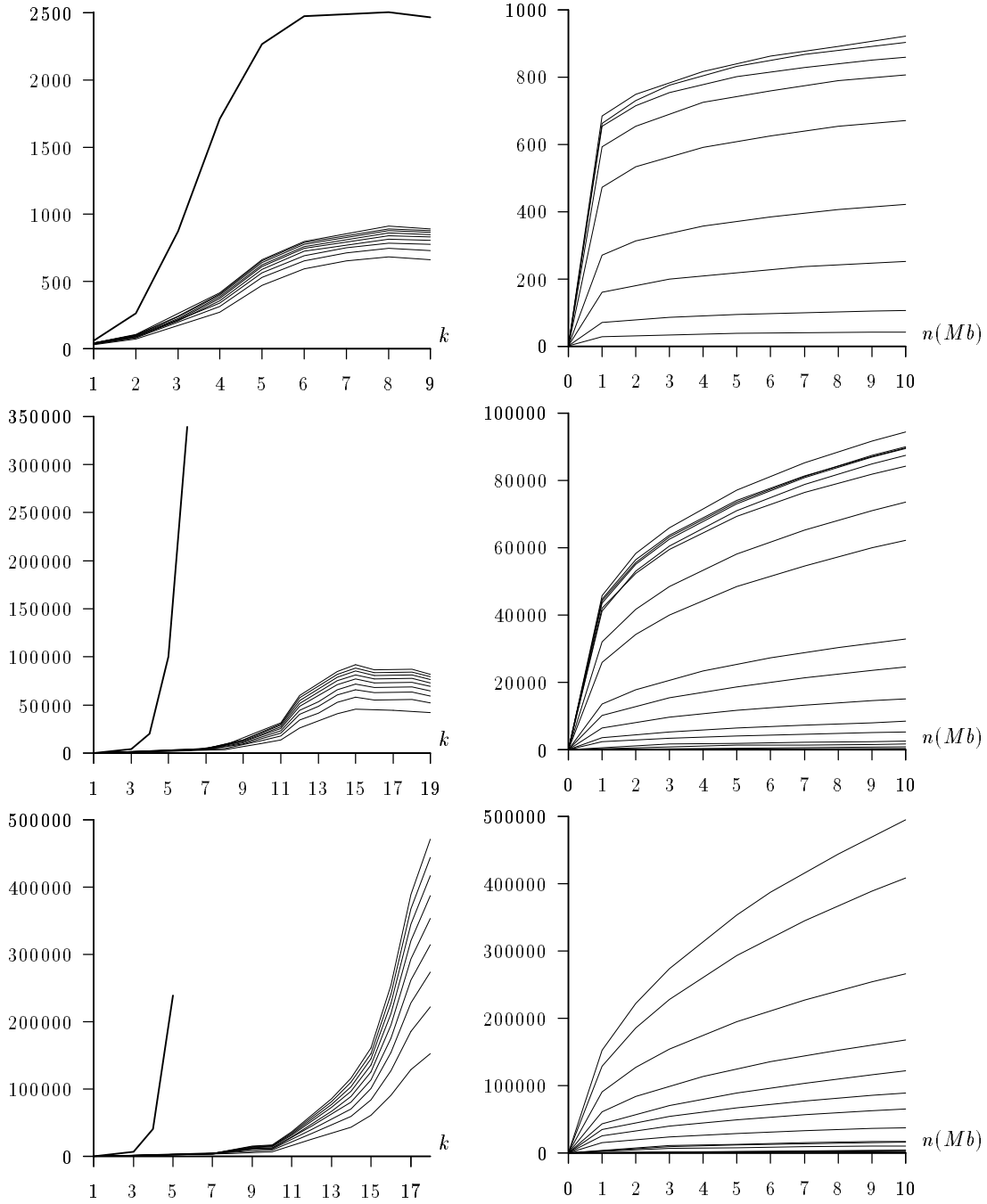

Fig. 2. Number of states of the partial and complete DFAs. The rows are for $m = 10$, 20 and 30, respectively. The left column shows the growth in terms of k (for n=1 to 10 Mb, from lower to upper, and the thick line for the full automaton). The right column shows the growth in terms of n (for $k = 1$ to $m - 1$, in general from lower to upper).

4.2 Comparison Against Other Algorithms

We experimentally compare our algorithm against the fastest previous algorithms we are aware of. These are (in alphabetical order)

Agrep [25] is a widely distributed approximate search software, that implements a hybrid algorithm. It is limited, although not inherently, to $m \leq 29$ and $k \leq 8$, so it is only included in the test for small patterns. Because of its match reporting policy and its options, it is hard to compare fairly with the other algorithms, but we include it as a reference point.

Baeza-Yates/Navarro [4] parallelizes the NFA by diagonals using bits of the computer word. The code is from the authors.

Baeza-Yates/Perleberg [5] is a filter based on partitioning the pattern plus exact search of the pieces. The code is ours.

Chang-Lampe [6] is the algorithm kn.clp, which computes only the places where the value of the dynamic programming matrix does not change along each column. The code is from the author.

Jokinen-Tarhio-Ukkonen [11] is a filter based on moving a window over the text and keeping a counter of text characters of the window that are present in the pattern. The code is ours.

Ours is our partial DFA algorithm.

Sutinen-Tarhio [19] is a filtration algorithm based on finding portions (q-grams) of the pattern in the text. The method is limited to $\alpha < 1/2$, and the implementation to $k \leq w/2 - 3$. The code is from the authors. We use $s = 2$ (number of samples to match) and maximal q (length of the q-grams), as suggested in [19].

Ukkonen [23] is the standard dynamic programming algorithm, modified to work only on active columns. The code is ours.

Wu-Manber-Myers [28] applies a Four Russians technique to the dynamic programming matrix, storing the states of the automaton in computer words. The code is from the authors, and is used with $r = 5$ as suggested in [28] (r is related with the size of the Four Russians tables).

Figure 3 shows the results. As it can be seen, our algorithm is the most efficient for intermediate error ratios, especially in not very short patterns. That is, from the point where filtration algorithms cease to work (since they are faster than linear) to the point where the number of states of the automaton grow too much. Observe that, as m grows, that point is reached for smaller α value, which allows to predict that the area where the method is the best will shrink and disappear for larger patterns.

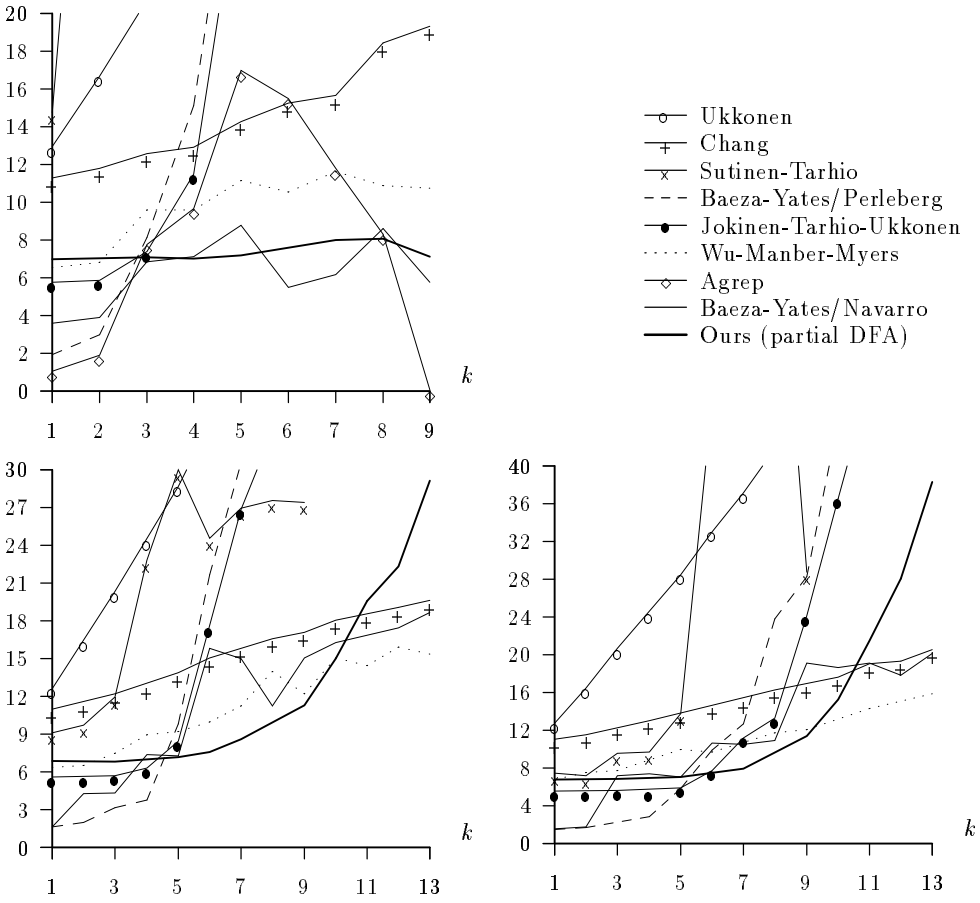

Fig. 3. Comparison among algorithms on 10 Mb of English text (seconds of user time). The plots are for $m = 10$, 20 and 30.

5 Conclusions and Future Work

We have explored the use of a deterministic automaton for approximate string matching. The main drawback of the method, namely the huge number of states of the automaton, is partially alleviated by building only the part of the automaton that is really needed in the search of that particular text. We have shown experimentally that the method outperforms all the others for intermediate error ratios and pattern lengths, and studied the growth of the partial automaton as the traversed text grows, finding that the partial DFA is much smaller and more manageable than the complete one.

Future work regarding this issue involves better handling of the number of states. For instance, some states may have been generated and never used again since a long time ago, and could be deallocated (and later re-generated if needed). Another alternative is to stop using the automaton in the interesting parts of the text (i.e. when the last active column approaches m). This would reduce drastically the number of states at the cost of using the classical algorithm in some portions of the text.

We are also studying how to estimate accurately the correct size of the hash table beforehand, or to use extendible hashing techniques. Finally, we can approach multipattern approximate search by making a single DFA for all the patterns. However, the number of states would grow even more than for a single pattern.

Acknowledgements

We thank Ricardo Baeza-Yates and an anonymous referee for their suggestions to improve this paper. We thank Stefan Kurtz for his prompt help on his PhD. thesis.

References

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

2. R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.

3. R. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proc. WSP'96*, pages 47–63, 1996. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/wsp96.2.ps.gz`.

4. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. Hirschberg and G. Myers, editors, *Proc. CPM'96*, pages 1–23, 1996. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/cpm96.ps.gz`.

5. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In Z. Galil A. Apostolico, M. Crochemore and U. Manber, editors, *Proc. CPM'92*, pages 185–192. Springer-Verlag, 1992. Lecture Notes in Computer Science 644.

6. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, 1992. Lecture Notes in Computer Science 644.

7. W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, Oct/Nov 1994.

8. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, pages 259–273. Springer-Verlag, 1994. Lecture Notes in Computer Science 807.

9. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal of Computing*, 19(6):989–999, 1990.

10. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

11. P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996.

12. S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, July 1995.

13. G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer Systems Science*, 37:63–78, 1988.

14. G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

15. B. Melichar. String matching with k differences by finite automata. In *Proc. of ICPR'96*, pages 256–260. IEEE CS Press, 1996.

16. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

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

18. F. Shi. Fast approximate string matching with q-blocks sequences. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proc. WSP'96*, pages 257–271, 1996.

19. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In P. Spirakis, editor, *Proc. ESA'95*. Springer-Verlag, 1995. Lecture Notes in Computer Science 979.

20. T. Takaoka. Approximate pattern matching with samples. In *Proc. ISAAC'94*, pages 234–242. Springer-Verlag, 1994. Lecture Notes in Computer Science 834.

21. J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. In J. Gilbert and R. Karlsson, editors, *SWAT'90*, pages 348–359. Springer-Verlag, 1990. Lecture Notes in Computer Science 447.

22. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

23. Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

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

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

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

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

28. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class