Online Dictionary Matching with Variable-Length Gaps

Tuukka Haapasalo[1], Panu Silvasti[1], Seppo Sippu[2], and Eljas Soisalon-Soininen[1]

[1] Aalto University School of Science
{thaapasa,psilvast,ess}@cs.hut.fi
[2] University of Helsinki
sippu@cs.helsinki.fi

Abstract. The string-matching problem with wildcards is considered in the context of online matching of multiple patterns. Our patterns are strings of characters in the input alphabet and of variable-length gaps, where the width of a gap may vary between two integer bounds or from an integer lower bound to infinity. Our algorithm is based on locating "keywords" of the patterns in the input text, that is, maximal substrings of the patterns that contain only input characters. Matches of prefixes of patterns are collected from the keyword matches, and when a prefix constituting a complete pattern is found, a match is reported. In collecting these partial matches we avoid locating those keyword occurrences that cannot participate in any prefix of a pattern found thus far. Our experiments show that our algorithm scales up well, when the number of patterns increases.

1 Introduction

String-pattern matching with wildcards has been considered in various contexts and for various types of wildcards in the pattern and sometimes also in the text [2–11, 13–17]. The simplest approach is to use the single-character wildcard, denoted "`.`" in `grep` patterns, to denote a character that can be used in any position of the string pattern and matches any character of the input alphabet Σ [4, 8, 15]. Generalizations of this are the various ways in which "variable-length gaps" in the patterns are allowed [2–4, 8, 11, 13, 14, 16]. Typically, a lower and upper bound is given on the number of single-character wildcards allowed between two alphabet characters in a pattern, such as "`.{`l`,`h`}`" in `grep` patterns. A special case is that any number of wildcards is allowed, called the *arbitrary-length wildcard*, denoted "`.*`" in `grep`, that matches any string in Σ^* [10, 13].

The above-mentioned solutions, except the ones by Kucherov and Rusinowitch [10] and by Zhang et al. [17], are for the single-pattern problem, that is, the text is matched against a single pattern. These algorithms [10, 17] are exceptions, because they take as input—besides the text—a set of patterns, but they are restricted to handle arbitrary-length wildcards only, and, moreover, they only find the first occurrence of any of the patterns.

P.M. Pardalos and S. Rebennack (Eds.): SEA 2011, LNCS 6630, pp. 76–87, 2011.

In this article we present a new algorithm that finds all occurrences of all patterns in a given pattern set, a "dictionary", in an online fashion. The patterns are strings over characters in the input alphabet Σ and over variable-length gaps, where the gaps can be specified as "`.`" (single-character wildcard), "`.{`l`,`h`}`" (gap of length l to h), or "`.*`" (gap of length 0 to ∞).

Our online algorithm performs a single left-to-right scan of the text and reports each pattern occurrence once its end position is reached, but at most one occurrence for each pattern at each character position. Each matched occurrence is identified by the pattern and its last element position in the document [2]. Because of the variable-length gaps, there can be more than one, actually an exponential number of occurrences of the same pattern at the same element position, but we avoid this possible explosion by recognizing only one occurrence in such situations.

We use the classic Aho–Corasick pattern-matching automaton (PMA) [1] constructed from the set of all keywords that appear in the patterns. A similar approach for solving the single-pattern matching problem was previously used by Pinter [15] allowing single-character wildcards in the pattern and by Bille et al. [2] allowing variable-length gaps with fixed lower and upper bounds.

Our new algorithm matches sequences of keywords that form prefixes of patterns with prescribed gaps between them. We thus record partial matches of the patterns in the form of matches of prefixes of patterns, and when a matched prefix extends up to the last keyword of the pattern, we have a true match. An important feature in our algorithm is that we use a *dynamic output function* for the PMA constructed from the keywords.

The problem definition is given Sec. 2, and our algorithm is presented in detail in Sec. 3. The complexity is analyzed in Sec. 4, based on the estimation of the number of pattern prefix occurrences in terms of the properties of the pattern set only. Experimental results, including comparisons with `grep` and `nrgrep`, are reported in Sec. 5.

2 Patterns with Gaps and Wildcards

Assume that we are given a string T of length $|T| = n$ (called the *text*) over a character alphabet Σ, whose size is assumed to be bounded, and a finite set D (called a *dictionary*) of nonempty strings (called *patterns*) P_i over characters in input alphabet Σ and over variable-length gaps. Here the *gaps* are specified as "`.{`l`,`h`}`", denoting a gap of length l to h, where l and h are natural numbers with $l \leq h$ or l is a natural number and $h = \infty$. The gap "`.{1,1}`" can also be denoted as "`.`" (the single-character wildcard or the don't-care character), and the gap "`.{0,`∞`}`" as "`.*`" (the arbitrary-length wildcard).

Patterns are decomposed into keywords and gaps: the *keywords* are maximal substrings in Σ^+ of patterns. If the pattern ends at a gap, then we assume that the last keyword of the pattern is the empty string ϵ. Each pattern is considered to begin with a gap, which thus may be ϵ. For example, the pattern "`.*ab.{1,3}c.*.d..`" consists of four gaps, namely "`.*`", "`.{1,3}`", "`.*.`" (i.e., "`.{1,`∞`}`"), and "`..`" (i.e., "`.{2,2}`"), and of four keywords, namely `ab`, `c`, `d`,

and ϵ. This pattern matches with, say, the input text `eeeabeecedeee`, while the pattern "`ab.{1,3}c.*.d..`" does not.

Our task is to determine all occurrences of all patterns $P_i \in D$ in text T. Like Bille et al. [2], we report a pattern occurrence by a pair of a pattern number and the character position in T of the last character of the occurrence. Because variable-length gaps are allowed, the same pattern may have many occurrences that end at the same character position; all these occurrences are reported as a single occurrence.

We number the patterns and their gaps and keywords consecutively, so that the ith pattern P_i can be represented as

$$P_i = \mathit{gap}(i,1)\mathit{keyword}(i,1)\ldots\mathit{gap}(i,m_i)\mathit{keyword}(i,m_i),$$

where $\mathit{gap}(i,j)$ denotes the jth gap and $\mathit{keyword}(i,j)$ denotes the jth keyword of pattern P_i.

For pattern P_i, we denote by $\mathit{mingap}(i,j)$ and $\mathit{maxgap}(i,j)$, respectively, the minimum and maximum lengths of strings in Σ^* that can be matched by $\mathit{gap}(i,j)$. The length of the jth keyword of pattern P_i is denoted by $\mathit{length}(i,j)$. We also assume that $\#\mathit{keywords}(i)$ gives m_i, the number of keywords in pattern P_i. For example, if the pattern "`.*ab.{1,3}c.*.d..`" is the ith pattern, we have

$\#\mathit{keywords}(i) = 4$,
$\mathit{mingap}(i,1) = 0$, $\mathit{maxgap}(i,1) = \infty$, $\mathit{length}(i,1) = 2$,
$\mathit{mingap}(i,2) = 1$, $\mathit{maxgap}(i,2) = 3$, $\mathit{length}(i,2) = 1$,
$\mathit{mingap}(i,3) = 1$, $\mathit{maxgap}(i,3) = \infty$, $\mathit{length}(i,3) = 1$,
$\mathit{mingap}(i,4) = 2$, $\mathit{maxgap}(i,4) = 2$, $\mathit{length}(i,4) = 0$.

3 The Matching Algorithm

For the set of all keywords in the patterns, we construct an Aho–Corasick pattern-matching automaton with a dynamically changing output function. This function is represented by sets *current-output*(q) containing *output tuples* of the form (i,j,b,e), where $q = \mathit{state}(\mathit{keyword}(i,j))$, the state reached from the initial state upon reading the jth keyword of pattern P_i, and b and e are the earliest and latest character positions in text T at which some partial match of pattern P_i up to and including the jth keyword can possibly be found. The latest possible character position e may be ∞, meaning the end of the text.

The current character position, i.e., the number of characters scanned from the input text is maintained in a global variable *character-count*. Tuples (i,j,b,e) are inserted into *current-output*(q) only at the point when the variable *character-count* has reached the value b, so that tuples (i,j,b,e) are stored and often denoted as triples (i,j,e). The function $\mathit{state}(\mathit{keyword}(i,j))$, defined from pairs (i,j) to state numbers q, is implemented as an array of $\#D$ elements, where each element is an array of $\#\mathit{keywords}(i)$ elements, each containing a state number.

The operating cycle of the PMA is given as Alg. 1. The procedure call *scan-next*(*character*) returns the next character from the input text. The functions

goto and *fail* are the goto and fail functions of the standard Aho–Corasick PMA, so that $goto(state(y), a) = state(ya)$, where ya is a prefix of some keyword and a is in Σ, and that $fail(state(uv)) = state(v)$, where uv is a prefix of some keyword and v is the longest proper suffix of uv such that v is also a prefix of some keyword.

Algorithm 1. Operating cycle of the PMA with dynamic output sets

```
initialize-output()
state ← initial-state
character-count ← 0
scan-next(character)
while character was found do
   character-count ← character-count + 1
   distribute-output()
   while goto(state, character) = fail do
      state ← fail(state)
   end while
   state ← goto(state, character)
   traverse-output-path(state)
   scan-next(character)
end while
```

The function *output-fail*(q) used in the procedure *traverse-output-path* (Alg. 4) to traverse the *output path* for state q is defined by: $output\text{-}fail(q) = fail^k(q)$, where k is the greatest integer less than or equal to the length of $string(q)$ such that for all $m = 1, \ldots, k-1$, $string(fail^m(q))$ is not a keyword. Here $string(q)$ is the unique string y with $state(y) = q$, and $fail^m$ denotes the *fail* function applied m times. Thus, the output path for state q includes, besides q, those states q' in the fail path from q for which $string(q')$ is a keyword; for such states q' the dynamically changing current output can sometimes be nonempty.

The initial current output tuples, as well as all subsequently generated output tuples, are inserted through a set called *pending-output* to sets *current-output*(q). Let

$$maxdist = \max\{mingap(i, j) + length(i, j) \mid i \geq 1, j \geq 1\}.$$

The set *pending-output* is implemented as an array of *maxdist* elements such that for any character position b in the input text the element

$$pending\text{-}output(b \bmod maxdist)$$

contains an unordered set of tuples (i, j, e), called *pending output tuples*. The first pending output tuples $(i, 1, e)$, with $e = maxgap(i, 1) + length(i, 1)$, are inserted, before starting the first operating cycle, into *pending-output*$(b \bmod maxdist)$, where $b = mingap(i, 1) + length(i, 1)$ (see Alg. 2). At the beginning of the operating cycle, when *character-count* has reached b, all tuples (i, j, e) from the set *pending-output*$(b \bmod maxdist)$ are distributed into the sets *current-output*(q), $q = state(keyword(i, j))$ (see Alg. 3).

When visiting state q, the set *current-output*(q) of the PMA is checked for possible matches of keywords in the procedure call *traverse-output-path*(q) (see Alg. 4). If this set contains a tuple (i, j, e), where *character-count* $\leq e$, then a match of the jth keyword of pattern P_i is obtained. Now if the jth keyword is the last one in pattern P_i, then a match of the entire pattern P_i is obtained. Otherwise, an output tuple $(i, j+1, e')$ for the $(j+1)$st keyword of pattern P_i is inserted into the set *pending-output*(b' mod *maxdist*), where

$b' =$ *character-count* + *mingap*$(i, j+1)$ + *length*$(i, j+1)$, and
$e' =$ *character-count* + *maxgap*$(i, j+1)$ + *length*$(i, j+1)$.

Here $e' = \infty$ if *maxgap*$(i, j+1) = \infty$.

Algorithm 2. Procedure *initialize-output*()

```
for all b = 0, ..., maxdist − 1 do
    pending-output(b) ← ∅
end for
for all patterns P_i do
    b ← mingap(i, 1) + length(i, 1)
    e ← maxgap(i, 1) + length(i, 1)
    insert (i, 1, e) into the set pending-output(b mod maxdist)
end for
for all states q do
    current-output(q) ← ∅
end for
```

Algorithm 3. Procedure *distribute-output*()

```
b ← character-count
for all (i, j, e) ∈ pending-output(b mod maxdist) do
    q ← state(keyword(i, j))
    insert (i, j, e) into the list current-output(q)
end for
pending-output(b mod maxdist) ← ∅
```

The collection of the sets *current-output*(q), for states q, is implemented as an array indexed by state numbers q, where each element *current-output*(q) is an unordered doubly-linked list of elements (i, j, e), each representing a current output tuple (i, j, b, e) for some character position $b \leq$ *character-count*. The doubly-linked structure makes it easy to delete outdated elements, that is, elements with $e <$ *character-count*, and insert new elements from *pending-output*.

We also note that for each pair (i, j) (representing a single keyword occurrence in the dictionary) it is sufficient to store only one output tuple (i, j, e), namely the one with the greatest e determined thus far. To accomplish this we maintain an array of vectors, one for each pattern P_i, where the vector for P_i is indexed by j and the entry for (i, j) contains a pointer to the tuple (i, j, e) in the doubly-linked list. We assume that the insertion into *current-output*(q) in Alg. 3 first

Algorithm 4. Procedure *traverse-output-path*(*state*)

```
q ← state
traversed ← false
while not traversed do
  for all elements (i, j, e) in the list current-output(q) do
    if e < character-count then
      delete (i, j, e) from the list current-output(q)
    else if j = #keywords(i) then
      report a match of pattern P_i at position character-count in text T
    else
      b' ← character-count + mingap(i, j + 1) + length(i, j + 1)
      e' ← character-count + maxgap(i, j + 1) + length(i, j + 1)
      insert (i, j + 1, e') into pending-output(b' mod maxdist)
    end if
  end for
  if q = initial-state then
    traversed ← true
  else
    q ← output-fail(q)
  end if
end while
```

checks from this array of vectors whether or not a tuple (i, j, e') already exists, and if so, replaces e' with the greater of e and e'.

4 Complexity

The main concern in the complexity analysis is the question of how many steps are performed for each scanned input character. For each new character the procedure *traverse-output-path* (Alg. 4) is executed, and thus we need to analyze how many times the outer **while** loop and the inner **for** loop are then performed within a *traverse-output-path* call. The number of iterations of the **while** loop is the length of the output path for the current state q. The maximum length of this path is the maximum number of different keywords that all are suffixes of *string*(q) for a given state q, which implies the bound for the maximal number of performed iterations.

Additionally, within each iteration of the **while** loop the **for** loop is performed for all triples (i, j, e) that belong to *current-output*(q). Because for any pair (i, j) at most one output tuple (i, j, e) exists in *current-output*(q) for $q =$ *state*(*keyword*(i, j)) at any time, this implies that the number of iterations performed in the **for** loop for state q is bounded by the number of different occurrences of keywords equal to *keyword*$(i, j) =$ *string*(q). However, not all these occurrences have been inserted into *current-output*(q), but only those for which all preceding keyword occurrences of the pattern have been recognized.

For any two strings w_1 and w_2 composed of keywords and gaps as defined in Sec. 2, we define that w_1 is a suffix of w_2, if there are instances w_1' and w_2' of w_1 and w_2, respectively, such that w_1' is a suffix of w_2'. The instance of string w

is defined such that each gap in w is replaced by any string in Σ^* such that the gap rules are obeyed. For keyword instance (i, j) we define set $S_{i,j}$ to contain all keyword instances (i', j') where the prefix of pattern $P_{i'}$ ending with keyword instance (i', j') is a suffix of the prefix of pattern P_i ending with keyword instance (i, j).

For any two tuples (i_1, j_1, e_1) and (i_2, j_2, e_2) in *current-output*(q), either the pattern prefix ending with (i_1, j_1) is a suffix of the pattern prefix ending with (i_2, j_2), or vice versa. Thus we can conclude that the number of iterations performed in the **for** loop for q is at most $\max\{|S_{i,j}| \mid (i, j, e) \in \textit{current-output}(q)\}$. This implies further that the number of operations per input character induced by the procedure *traverse-output-path* is bounded above by the maximum size, denoted k, of the sets $S_{i,j}$, where (i, j) is any keyword instance in the dictionary D. It is clear from the matching algorithm that all other work done also has the time bound $O(kn)$, where n is the length of the input text. An upper bound for k is the number of keyword instances in the dictionary, but k is usually much less.

A better upper bound for k, instead of simply taking the number of keyword instances in D, is obtained as follows. For keyword set W denote by *pocc*(W) the number of occurrences of keywords in W in the dictionary D. Further denote by *closure*(w), for a single keyword w, the set of keywords in D that contains w and all suffixes of w that are also keywords. Then $\max\{\textit{pocc}(\textit{closure}(w)) \mid w \text{ is a keyword in } D\}$ is an upper bound of k.

The preprocessing time, that is, the time spent on the construction of the PMA with its associated functions and arrays, is linear in the size of dictionary D (i.e., the sum of the sizes of the patterns in D).

In terms of the occurrences of pattern prefixes in the text it is easy to derive, for processing the text, the time complexity bound $O(Kn+\textit{occ}(\textit{pattern-prefixes}))$, where K denotes the maximum number of suffixes of a keyword that are also keywords, and *occ*(*pattern-prefixes*) denotes the number of occurrences in the text of pattern prefixes ending with a keyword.

5 Experimental Results

We have implemented a slightly modified version of the algorithm of Sec. 3 in C++. The modifications are concerned with minor details of the organization of the current and pending output sets and with the deletion of expired output tuples. We observe that after seeing the jth keyword of pattern P_i that is followed by a gap of unlimited length, we may also consider as expired all output tuples (i, j', e) with $j' < j$. Also, we did not use the array of vectors indexed by pairs (i, j) and containing pointers to output tuples (i, j, e) (see the end of Sec. 3), but allowed the current output set for *state*(*keyword*(i, j)) to contain many tuples (i, j, e).

We have run tests with a 1.2 MB input text file (the text of the book *Moby Dick* by H. Melville) using pattern files with varying number of patterns and varying number of *segments* delimited by an unlimited gap "`.*`". Let m be the length of a pattern and s the number of segments in it. We generated the patterns by taking from the input text s pieces of length m/s that are relatively

close to each other (so that the entire pattern is taken from an $8m$-character substring of the input), and by catenating these pieces together by appending a "`.*`" gap in between them. In addition, we replaced a portion of the characters in the segments with wildcards, and we converted some wildcard substrings to randomly chosen (but matching) limited variable-length gaps. Each formed pattern thus matches at least once. Partial examples of generated patterns include:

```
.omple.. .irc.* f...l.nc....n.* r.shed.to...e.* .s ..e eager
i.ot.{2,19}e.e.{0,6}.*.{0,2}cia..y.Capta.*.{1,4}ge .{2,22}eyo
```

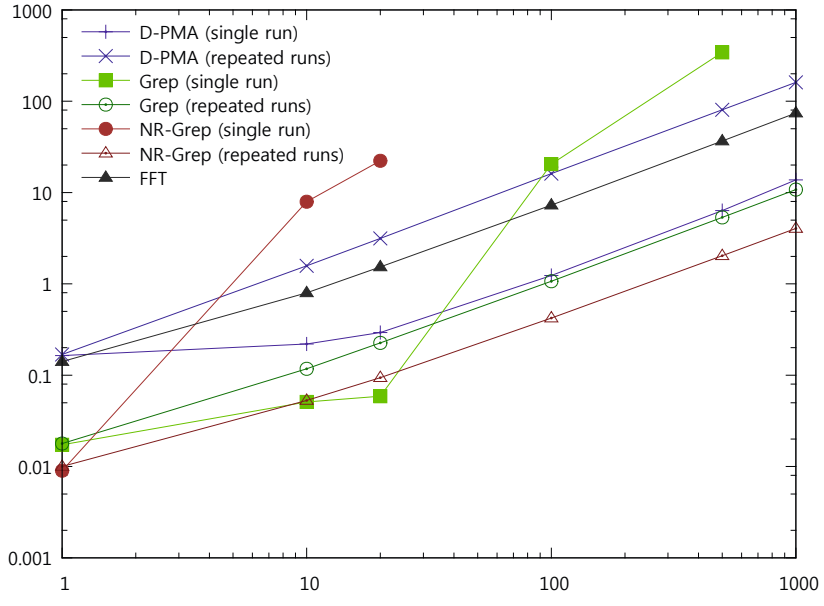
We used patterns with an average length of 80 characters when the patterns did not contain limited variable-length gaps, and 100 characters when they did (five gaps on average). The length of the variable-length gaps we used was not very high, varying on average from close to zero to around ten or twenty. (We expect the speed of our algorithm to be independent of the (minimum) length of a gap (*mingap*), while the difference of the maximum and minimum lengths of a gap does matter; *mingap*s only affect the size of the array *pending-output*.) Finally, we made 75 % of the patterns non-matching by appending a character that does not appear in the input text, at the end of the pattern; although further tests revealed that this does not affect the run time of our algorithm much. Each matching pattern usually has only one occurrence in the input text. We generated workloads with 1, 3, and 5 segments; with a total of 1, 10, 20, 100, 500, and 1000 patterns in each workload. The workloads were generated additively, so that the smaller workloads are subsets of the larger workloads.

We used the following programs in our test runs (cf. Fig. 1): (1) D-PMA, our dynamic pattern-matching algorithm; (2) FFT, the wildcard matcher based on fast Fourier transform [5] (this can only be used when the patterns do not contain arbitrary- or variable-length gaps); (3) Grep, the standard Linux command-line tool `grep`; we use the extended regular expression syntax with the `-E` parameter so that variable-length gaps can be expressed; (4) NR-Grep, the `nrgrep` Linux command-line tool by Navarro [12] (which can only handle fixed-length and arbitrary-length gaps).

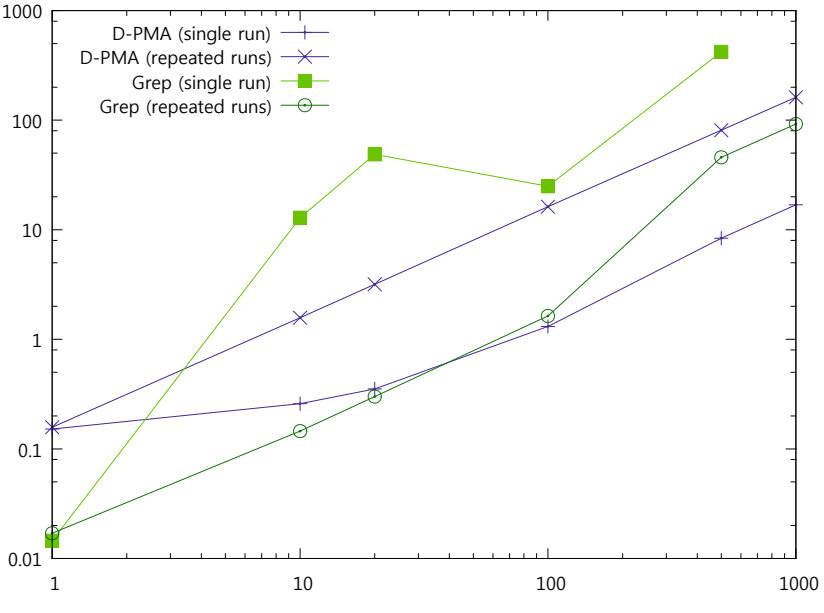
Fig. 1 shows the results of three of the test runs. The values are averages of six test runs with a standard deviation of mostly less than a percent when there are more than 20 patterns; with only a few patterns there is some small variance. The figures and further tests confirmed that `grep` performs much worse for variable-length gaps than for fixed-length gaps. On the contrary, our D-PMA algorithm has about the same performance with and without variable-length gaps.

The variants *single run* and *repeated runs* refer to how the programs were run. With *repeated runs*, each pattern of the workload was processed separately, running the program once for each pattern. This is the only way to make `grep` and `nrgrep` find occurrences individually for every pattern; in this case `grep` and `nrgrep` solve the *filtering problem* for the dictionary, that is, find the first occurrences for each pattern, if any.

With *single run*, all the patterns of a workload were fed to the program at once. With `grep`, we gave the patterns in a file with the `-f` parameter, and with `nrgrep`, we catenated the patterns together, inserting the union operator

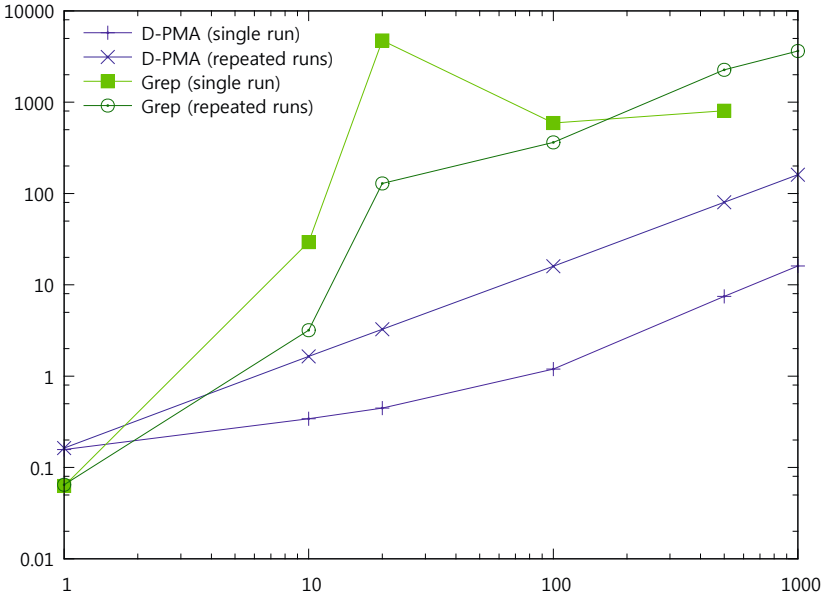


(a) Patterns with only fixed-length gaps



(b) Patterns with limited fixed- and variable-length gaps

Fig. 1. Matching times in seconds for dictionaries of increasing numbers of patterns. (NR-Grep cannot handle limited variable-length gaps).



(c) Patterns with fixed- and variable-length gaps, and four unlimited gaps

Fig. 1. *Continued*

"|" in between the pattern instances, and enclosing the patterns themselves in parentheses. In this case both `grep` and `nrgrep` only solve the *language-recognition problem* for the dictionary, that is, determine whether some pattern in the dictionary has a match; they thus stop processing the input as soon as the first match has been found. This can be seen from Figs. 1(b) and 1(c): searching for 100 patterns is faster than searching for 20 patterns, because then the first match of some pattern is found earlier.

All our tests were run on a computer with a 64-bit 2.40 GHz Intel Core 2 Quad Q6600 processor, 4 GB of main memory, and 8 MB of on-chip cache, running Fedora 14 Linux 2.6.35. The test programs were compiled with the GNU C++ compiler (`g++`) 4.5.1.

When run with a single run, both `grep` and `nrgrep` fail when there are too many patterns to process: `grep` could not complete any workload with 1000 patterns (out of memory); and `nrgrep` could not complete any workload with more than 20 patterns, but rather failed due to a possible overflow bug. Furthermore, `nrgrep` could not be run with the test workloads that included limited variable-length gaps, because `nrgrep` does not support them.

The results clearly show that our algorithm outperforms `grep` and also `nrgrep`, except when `nrgrep` was applied repeatedly (offline) for patterns with fixed-length gaps only. In that case `nrgrep` was about three times faster than our algorithm. Our algorithm scales very well to the number of patterns, for instance, for 500 patterns the online single run was ten times faster than 500 individual

runs. Moreover, we emphasize that our algorithm solves the genuine dictionary-matching problem, finding all occurrences for all the patterns, while `grep` and `nrgrep` do not. In addition, our algorithm can process multiple patterns efficiently in an online fashion, with a single pass over the input text, making it the only viable option if the input is given in a data stream that cannot be stored for reprocessing. In solving the filtering problem, our algorithm was slightly faster than when solving the dictionary-matching problem with the same pattern set and input text.

6 Conclusion

We have presented a new algorithm for string matching when patterns may contain variable-length gaps and all occurrences of a (possibly large) set of patterns are to be located. Moreover, our assumption is that pattern occurrences should be found online in a given input text, so that they are reported once their end positions have been recognized during a single scan of the text. Our solution is an extension of the Aho–Corasick algorithm [1], using the same approach as Pinter [15] or Bille et al. [2] in the sense that keywords, the maximal strings without wildcards occurring in the patterns, are matched using the Aho–Corasick pattern-matching automaton (PMA) for multiple-pattern matching.

An important feature in our algorithm is that we avoid locating keyword occurrences that at the current character position cannot take part in any complete pattern occurrence. The idea is to dynamically update the output function of the Aho–Corasick PMA. Whenever we have recognized a pattern prefix up to the end of a keyword, output tuples for the next keyword of the pattern will be inserted. In this way we get an algorithm whose complexity is not dominated by the number of all keyword occurrences in the patterns. This claim is confirmed by our experiments, which show that our algorithm outperforms `grep` and scales very well to the number of patterns in the dictionary.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. of the ACM 18, 333–340 (1975)
2. Bille, P., Li Gørtz, I., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 385–394. Springer, Heidelberg (2010)
3. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proc. of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010), pp. 1297–1308 (2010)
4. Chen, G., Wu, X., Zhu, X., Arslan, A.N., He, Y.: Efficient string matching with wildcards and length constraints. Knowl. Inf. Syst. 10, 399–419 (2006)
5. Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Inform. Process. Letters 101, 53–54 (2007)

6. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. of the 36th Annual ACM Symposium on Theory of Computing, pp. 90–100 (2004)
7. Fischer, M., Paterson, M.: String matching and other products. In: Proc. of the 7th SIAM-AMS Complexity of Computation, pp. 113–125 (1974)
8. He, D., Wu, X., Zhu, X.: SAIL-APPROX: an efficient on-line algorithm for approximate pattern matching with wildcards and length constraints. In: Proc. of the IEEE Internat. Conf. on Bioinformatics and Biomedicine, BIBM 2007, pp. 151–158 (2007)
9. Kalai, A.: Efficient pattern-matching with don't cares. In: Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 655–656 (2002)
10. Kucherov, G., Rusinowitch, M.: Matching a set of strings with variable length don't cares. Theor. Comput. Sci. 178, 129–154 (1997)
11. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Biol. 12, 1065–1082 (2005)
12. Navarro, G.: NR-grep: a fast and flexible pattern-matching tool. Soft. Pract. Exper. 31, 1265–1312 (2001)
13. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings. Cambridge University Press, Cambridge (2002)
14. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. J. Comput. Biol. 10, 903–923 (2003)
15. Pinter, R.Y.: Efficient string matching. Combinatorial Algorithms on Words. NATO Advanced Science Institute Series F: Computer and System Sciences, vol. 12, pp. 11–29 (1985)
16. Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: Finding patterns with variable length gaps or don't cares. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 146–155. Springer, Heidelberg (2006)
17. Zhang, M., Zhang, Y., Hu, L.: A faster algorithm for matching a set of patterns with variable length don't cares. Inform. Process. Letters 110, 216–220 (2010)