28

Fast Two Dimensional Pattern Matching

Mireille Régnier
INRIA, Rocquencourt

[summary by Pierre Nicodème]

e address the problem of finding a $m \times m$ pattern in a $n \times n$ text. We conjecture that, with a constant (i.e. $O(m^2)$) additional memory, this complexity lies between $[1+O(\frac{1}{m})]n^2$ and $[2+O(\frac{1}{m})]n^2$, which is different from the linear result in dimension 1. We provide an algorithm that achieves this goal with good average performance, that can be easily coded and that can be made alphabet independent.

1 Introduction and State of the Art

First algorithms for 2D pattern matching [4, 5, 9, 15] have been rather rough extensions to two dimensions of 1D pattern matching paradigms, that did not really used the specificity of 2D. Hence, all had an average case of n^2 at least. Additionally, most would need a $O(n)$ extra-space. The only exception is [15], but it needs a $O(n^2)$ extra-space. By dividing the text into subpieces, [9] reduces its extra-space to $O(m^2)$, but the price to pay is a substantial increasing of the average number of comparisons. Moreover, all are alphabet dependent.
Such performance sounds very poor when compared to 1D theoretical and practical results. For the average case, a $n\frac{\log m}{m}$ theoretical bound has been proved in [14], and it is achieved, for uniform distributions, by a rough algorithm in [10]. Boyer-Moore and its variants are sublinear for "non pathological" distributions and non-binary alphabets [3, 13]. Worst-case complexity relies between 1 and 2 for most practical algorithms [6], and theoretical worst case complexity was recently proved to be $1+O(\frac{1}{m})$ [7].
In [2], 2D specificity is used, and a first sublinear algorithm in the average is proposed. It drastically improves on previous ones, as average performance becomes $O(\frac{n^2}{m})$ with $O(m^2)$ extra-space. Additionally, the worst case does not depend on the pattern size, being $O(n^2)$, and it runs on-line.

2 The algorithm

2.1 Formalism

Worst-case complexity is related to the *maximal number of occurrences* of a searched pattern P in all possible texts, hence, to maximal coverings of the text by the pattern. As P can overlap with himself, these may not be tilings. This can be expressed as a function of a canonical decomposition of P. Let us say a few words on 1D complexity [8]. It is proved in [11] that a self-overlapping pattern can be written $p = u(vu)^m, m \geq 1, u \neq \epsilon$, where vu is *primitive*, i.e. not the power of any word w. The pattern p is *periodic* if and only if $m \geq 2$. As discussed in [12], the decomposition is not unique although at most one satisfies $m \geq 2$. Nevertheless, two coverings $t = u(vu)^*$ and $t = z(wz)^*$ cannot be interleaved. Hence, 1D worst case complexity is linear [7].
This property disappears in 2D pattern matching. Two coverings can be interleaved.

Definition 1 *An alignment of a pattern P with himself may be characterised by canonical coordinates: a 3-tuple (x, y, dir). One of the two occurrences, say P_r, has a left corner inside the area defined by the second one, P_l; x and y are the row and column indices so defined, and dir is a boolean set to $true$ (resp. $false$) when this is an upper (resp. lower) corner.*

We will refer to *dir* as the direction of the alignment, and use the terminology of *down* or *up* alignment.

Definition 2 *An alignment of a pattern P with himself is* **consistent** *if no mismatch occurs in the overlapping area.*

One also says that P self overlaps. When the overlapping area includes the center(s), P is periodic. We define here the repetitions in the pattern that will allow to speed up the searching process (compared to the naive search). We also define an order:

Definition 3 *Given two overlapping potential matching areas PM and PM' with canonical coordinates relative to the beginning of the text (x, y) and (x', y'), we define the order:*

$$PM \preceq PM' \Longleftrightarrow y < y' \text{ or } (\ y = y' \textit{ and } x \leq x'\ )\ .$$

Definition 4 *Two potential matching areas are said to be (fully) consistent in either one of the two cases:*

* *they do not overlap,*
* *the associated alignment is consistent.*

Definition 5 *A set of overlapping areas is said fully consistent if its elements are mutually consistent. Its left border is the column coordinate of its minimal elements.*

Remark that all column coordinates of an overlapping area range between the left border l and $l + 2m$. We also define a *canonical checking order* for all patterns in the text: say from left to right and top to bottom. This yields an interesting property that we will use:

Lemma 1 *Let PM' be a potential matching area. We define property $(P1)$ by:*

$(P1)$: *All checked positions are*

* *either matching positions of some smaller consistent overlapping potential matching areas.*
* *or mismatching positions of non consistent potential matching areas.*

Then, we will use an array MISMATCH(x, y, dir) of couples of integers (X, Y): (X, Y) is either $(0, 0)$ for a consistent alignment or the coordinates of the first mismatching position. This allows checking consistency in constant time. Additionally, this provides a criterium to discard a potential alignment of P with some potential matching area PM': namely, whenever some $PM \preceq PM'$ has been checked until mismatching position α interior to the overlapping area. This criterium is checked in *constant time* (procedure *Check* in our code). Remark that the preprocessing is trivially achieved by checking $(\sum_1^m i)^2$ characters, which is $O(m^4)$.

We are now ready to explain our algorithm. It relies on a division of the whole text in *slabs* of height m, delimited by rows km, $k \in \mathbb{N}$. These rows are said *primary* rows, and the characters in them *primary* characters (versus *secondary* rows and characters). Then, any potential matching area will intersect one primary row, and only one. This intersection will be any row p_i, $i = 1 \ldots m$ of the pattern. Hence, we proceed in two stages. First, a multi-string searching of strings p_i is performed on primary rows, defining a potential matching area. Any multi-string searching automaton [1] can be used. We name this automaton "automaton A", this stage *horizontal* or *primary* search. The second stage checks secondary characters. So far, this "slab method" does not significantly differ from [2] that provided a good average case: $O(\frac{n^2}{m})$. In order to ensure simultaneously an optimal worst case (or close to optimal) we delay secondary search until we know "enough information" to the right. And while performing it, we make use of all known information to the left. Our horizontal search is completed as follows: initialisation runs automaton A until some pattern is found, uniquely defining a potential matching area, *First*, with column coordinate *col*. The stationary stage consists of both checking the consistency of potential matching areas, and restarting the automaton A (which remains loaded), whenever the leftmost potential matching areas have been successfully checked or discarded. In the stationary stage, we run A to detect the next PM in the range $col, col + m - 1$. *ListCandidate* is maximal if and only if no more exists. Else, we check PM against every candidate from *ListCandidate*. If it is consistent with all of them, it is inserted to it. If not, a "Duel Strategy" using *Check* allows to kill a (strict) subset of $ListCandidate \cup \{PM\}$. Remark that whenever *First* is killed in this process, the left border shifts to the right and automaton A is restarted. Then, we start the effective secondary check, checking its minimal element, *First*, and avoiding to redo comparisons on positions already checked during the same effective secondary checking; this is done by maintaining a list of 2m-1 intervals of characters already checked. When it ends, we update *ListCandidate* using our *Check* procedure if last comparison was a mismatch and restart the horizontal search. Remark that all elements remaining in *ListCandidate* being consistent with the last checking, property $(P1)$ is still ensured. For a sake of clarity of the description and of the code, we assume now that all p_i are different. We delay after the discussion of worst case complexity the description of the general case.
Finally, we show how to avoid multiple comparisons in secondary search.

Lemma 2 *Let PM be a valid candidate, i.e. a potential matching area for which* Secondary Search *is called, and col its column coordinate. For any secondary row $i, 1 \le |i| \le m-1$, the largest column index of text positions scanned in a secondary search, j, satisfies:*

* *either $j < col$,*

* *or all positions between col and j are matching positions.*

Proof. From our *ListCandidate* construction, whenever *SecondarySearch* is called for two overlapping areas, they are consistent. Assume now $j \geq col$, and let PM_0 be the area for which $t[i, j]$ was scanned. If $t[i, j]$ was a mismatch, then PM would have been discarded from *ListCandidate* after this *SecondarySearch* call. Then, it was a match for PM_0, hence for PM. □

```
SlabSearch( (text, n × n), (pat, m × m) )
for( k ← m; k ≤ n; k ← k + m ) {                    /* k: current primary row index */
    q ← q0;                                          /* searching a fully consistent set */
    for( j ← 1; j ≤ n; j ← j + 1 ) {
        Repeat { q ← A(q, text[k, j]); i ← Output(q); } until (i ≠ 0);     /* i: found string index */
```

```
        Insert(ListCandidate,(i, j));
        while (ListCandidate ≠ ∅)
        {
          while (j < col + m & j ≤ n)                  /* col: column index of First */
          {
            q ← A(q, text[k, j]); i ← Output(q);
            if i ≠ 0 then {
              for (i', j') ∈ ListCandidate {           /* Check (i, j) consistency with ListCandidate */
                  Survive ← true;
                  (X, Y) ← MISMATCH(|i' − i|, j' − j, sign(i' − i)) ;
                  if X + Y ≠ 0 then                              /* kill one candidate */
                    if (Survive ← ¬(text([k − i' + X, j' + Y] == Pat[X, Y])) then
                      Delete(ListCandidate,(i', j'))      /* Mismatch on (i', j') in ListCandidate */
                      else break;   /* Mismatch on (i, j): stop checking ListCandidate, restart A */
                  };                                              /* end for */
              if Survive then Insert(ListCandidate,(i, j));          /* (i, j) consistency */
              };                                                    /* endif */
            j ← j + 1;
          }                                                      /* end second while */
        SecondaryCheck(First);
        Update(ListCandidate);
        }                           /* Secondary search end; restart ListCandidate search */
    }
}
```

3 2D-Complexity

We defer the precise study of 2D-complexity. This study will make an exhaustive analysis of plane covering and tiling by patterns not overlapping, or overlapping in different ways. Theoretical bounds will have to be derived from this analysis.

4 Alphabet Independency

As we said above, our algorithm depends on the alphabet only by the multistring searching automaton (and the preprocessing). It can be made alphabet independent in the following way [2]; choose any right to left automaton, represent it as a m-ary tree and map it to a binary tree in the classical way. A final state is attained in $(l + r)$ transitions, where l and r are the number of left and right branchings on the path to this state. Note that a left branching is associated to a match while a right one is a shift to another string. Consider now a step where one stopped on (l_0, r_0). The rough upper bound $l_0 + r_0 \leq 2m$ holds. Assume first the (strictly positive) shift s also is greater than 2. If no string was found, then 2 columns are definitely discarded, hence a cost $\frac{2m}{(m-1)s} \leq 1 + O(\frac{1}{m})$. This bound also holds when p_i is found but *SecondarySearch* not called later on the alignment so defined. Now, assume p_i is found and defines an alignment on which *SecondarySearch* is called later. Within a range of m, we have k steps with a cost $\sum_{i \geq 1}(l_i + r_i)$. By construction $\sum l_i \leq m$ and $\sum r_i \leq m + m$ (number of strings and potential number of fails). This finally yields a $4m$ bound, and a $O(1)$ amortised cost for *primary* characters. Last, we consider a shift $s = 1$ and assume a strict suffix *Suf* of some string is found. If $|Suf| \leq m - 2$, we reason

as above. Else, we have $r_0 + r_1 \leq m$, unless $Suf = a^*$. More generally, we either have $\sum r_i \leq m$ within a range of m or we are led to previous cases. The end of the proof for $p_i = a^{m-1}*$ is similar and we skip the technical details.
Hence, the difference to linear cost comes from slabs overlaps on secondary rows.

References

[1] A. V. Aho and M. Corasick. Efficient String Matching. *Communications of the ACM*, 18(6):333–340, 1975.

[2] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. In *SWAT'90*, volume 447 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 1990. Proc. Swedish Workshop on Algorithm Theory, Bergen, Norway. To appear in IPL.

[3] R. Baeza-Yates and M. Régnier. Average running time of Boyer-Moore-Horspool algorithm. *Theoretical Computer Science*, 92:19–31, 1992.

[4] T. Baker. A technique for extending rapid exact string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7:533–541, 1978.

[5] R. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6:168–170, 1977.

[6] R. Cole. Tight Bounds on the Complexity of the Boyer-Moore string matching algorithms. In *SODA'91*, pages 224–233. SIAM, 1991. Proc. 2-nd SIAM-ACM Symp. on Discrete Algorithms, San Francisco, USA.

[7] R. Cole and *al.* 1D pattern matching complexity, 1992. Preprint.

[8] L. Colussi, Z. Galil, and R. Giancarlo. On the exact Complexity of string matching. In *FOCS'90*, pages 135–143. IEEE, 1990. Proc. 31-st Annual IEEE Symposium on the Foundations of Computer Science.

[9] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[10] D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[11] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, Reading, Mass., 1983.

[12] M. Régnier. Knuth-Morris-Pratt algorithm: an analysis. In *MFCS'89*, volume 379 of *Lecture Notes in Computer Science*, pages 431–444. Springer-Verlag, 1989. Proc. Mathematical Foundations of Computer Science 89, Porubka, Poland.

[13] M. Régnier. A language approach to string searching evaluation. In *Proceedings of Combinatorial Pattern Matching '92, Tucson, Arizona*, pages 15–26. Springer-Verlag, 1992.

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

[15] R. F. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Communications of the ACM*, 32(9):1110–1120, 1989.