6 N − 1

String approximate pattern-matching
文字列近似照合

Yves Lepage
ATR 音声翻訳通信研究所
619-02 京都府相楽郡精華町光台２丁目２番地
`lepage@itl.atr.co.jp`

Abstract

We present an algorithm for approximate pattern-matching that has been shown to be asymptotically faster than *agrep* on average on an NLP task: finding all words in a dictionary at a certain distance of a given word.

Introduction

Approximate matching is searching for those lines in a file that contain a substring at a distance less than or equal to a certain threshold k from a given pattern p (of length m). This distance is the minimum number of character insertions, deletions or substitutions necessary to transform the substring into the pattern.

For example, when looking for **analogy** with a threshold of 2, only the first and third lines of the following file are output.

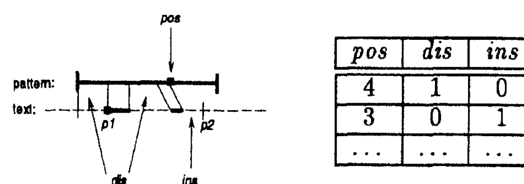
`analogous`	dist(**analog**, **analogy**) = 1
`explanation`	(insert `y`)
`neuroanatomy`	dist(**anatomy**, **analogy**) = 2
	(substitute `l` to `t` and `g` to `m`)

Recently, Wu and Manber [Wu & Manber 92] proposed a practical implementation of the Baeza-Yates and Gonnet method, which exhibits a behaviour of $O(nk\lceil\frac{m}{w}\rceil)$, where w is in fact some constant. *agrep* is considered the fastest practical such algorithm.

1 Common subsequences

On the contrary to existing methods which, basically, improve the computation of a standard distance matrix, our method tries to detect the longest common subsequence between the pattern and the text (the two problems are dual). Any candidate longest subsequence may be characterised by just three integers: the last position matched in the pattern (*pos*), the past distance up to that point (sum of the maximums of the lengths of the corresponding non-matching parts of the pattern and the text, *dis*), and the length of a gap ahead (*ins*).

Hence, the core data of our algorithm is a table containing arrays of three integers, each array representing a possible candidate match being built.



pos	*dis*	*ins*
4	1	0
3	0	1
...	...	...

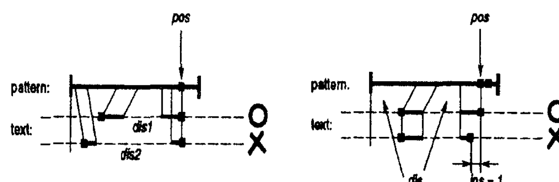
At a given position in the text, if the next character read from the text belongs to the pattern, a new candidate may be computed over all compatible candidates, by taking the minimal past distance possible (Ⓒ in the algorithm).

In any case, the next character may be considered as a gap added ahead. Hence, for all candidates, this gap, represented by *ins*, is incremented (Ⓐ and Ⓑ).

From this sketch of the algorithm, one could conclude that the set of candidates will always grow. Fortunately, some constraints may be applied so as to reduce it.

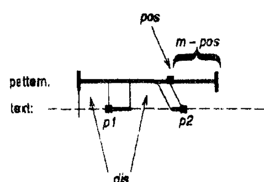
2 Constraints

Global constraints Firstly, for the same position in the pattern and the text, and without any gap ahead ($ins = 0$), we may eliminate a candidate with a bigger distance currently computed, as it will never become better than the one with the smallest distance. This is the Min constraint (on the left below, the candidate with *dis2* is eliminated in favor of the one with *dis1*).



Secondly, introducing a gap ahead, would be penalising, if the next character read from the text matches exactly the next character in the pattern. Hence, this sort of candidate is eliminated. This is the Continuity constraint: $p[pos + 1] = c$ and $ins = 0$ (on the right above, the candidate marked with a circle is kept, as the one marked with a cross is not introduced).

Local constraints Obviously, we can discard any candidate for which the sum of the past distance and the gap ahead exceeds the threshold. This is the Candidate constraint: $dis + ins \leq k$.



We may finally remark that it is possible to verify that a candidate is a solution even before matching the pattern entirely. It suffices that the remaining part of the pattern be less than what is left to fill the threshold: $m - pos < k - dis$. Whenever the Solution constraint is fulfilled, the line may be output: it contains a solution.

3 The algorithm

The previous remarks are incorporated in the following sketch of the algorithm. Depending on whether the next character c read from the text belongs to the pattern p, different constraints are checked in different orders so as to eliminate candidates (Ⓐ, Ⓑ and Ⓒ). Only in Ⓒ can a line be output when the Solution constraint is verified.

```
func match(character c)
   if c ∉ p then
      for each candidate (pos, dis, ins) do
Ⓐ        Candidate for (pos, dis, ins+1)
   else -- c ∈ p
      for each candidate (pos, dis, ins) do
Ⓑ        Continuity/Candidate for (pos, dis, ins+1)
      for each position π of c in p do
Ⓒ        Min/Solution/Candidate for (π, min_{π>pos} dis, 0)
   end if
end func match
```

4 Performance

The algorithm was tested on the task of matching any word, with all possible thresholds (from 1 to the length of the word minus 1), extracted from a 25 000 word long dictionary against that dictionary.

As the performance of the algorithm depends on the number of candidates, we measured the size of the candidate table during execution. Over all runs, the maximal size ever reached was 21. In comparison, the average size is extremely low: 0.41, and this explains why our algorithm competes well with the state-of-the-art algorithm *agrep*.

Although, over all runs, *agrep* is faster in 59% of the cases, our algorithm exhibits a much better asymptotic behaviour. For cases with a threshold bigger than 3, *agrep* is only faster in 6% of the cases! We measured the runtimes differences Δt between *agrep* and our algorithm. Of course, both algorithms deliver exactly the same solutions for all runs.

threshold	$k \leq 3$	$k > 3$	all k's
percentage of cases	47%	53%	100%
agrep faster	79%	6%	59%
mean Δt (in ms)	-68	95	18
std. dev. Δt (in ms)	±59	±71	±105

When drawing the runtimes of both algorithms one against the other, by pattern lengths and by thresholds (on average), it appears clearly that *agrep* is only faster for small values. The fact that these cases are more frequent on the whole of the runs, explains why *agrep* was found faster in 59% of the cases.

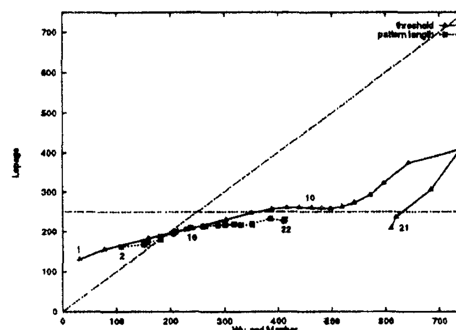


Figure 1: Our algorithm against *agrep* (times in ms, for all pattern lengths and thresholds, on average).

The threshold/pattern length ratio shows even better that the behaviour of our algorithm is more promising. When this ratio grows, our algorithm remains below or around 250 ms, whereas *agrep* needs more and more time. For ratios greater than 0.4, our algorithm is faster.

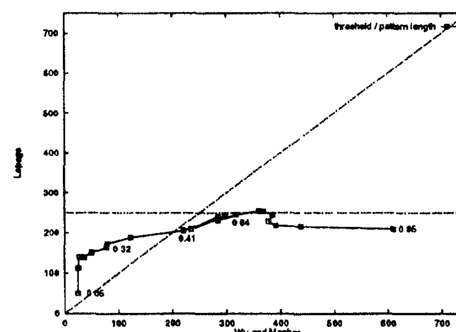


Figure 2: Our algorithm against *agrep* (times in ms, threshold/pattern length ratio, on average).

Conclusion

We presented an algorithm which competes well with *agrep*. It has been shown to be faster for thresholds greater than 3, and also for threshold/length ratios greater than 0.4. This algorithm makes retrieval of similar sentences for example-based NLP systems envisageable, as, typically, sentences have more than 10 words, and thresholds need to be larger than 5.

References

[Wu & Manber 92] Sun Wu & Udi Manber
Fast Text Searching Allowing Errors
Communications of the ACM, Vol. 35, No. 10, October 1992, pp. 83-91.