Improved Approximate Pattern Matching on Hypertext *

Gonzalo Navarro

Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

Abstract. The problem of approximate pattern matching on hypertext is defined and solved by Amir et al. in $O(m(n \log m + e))$ time, where m is the length of the pattern, n is the total text size and e is the total number of edges. Their space complexity is $O(mn)$. We present a new algorithm which is $O(mk(n+e))$ time and needs only $O(n)$ extra space, where $k < m$ is the number of allowed errors in the pattern. If the graph is acyclic, our time complexity drops to $O(m(n+e))$, improving Amir's results.

1 Introduction

Approximate string matching problems appear in a number of important areas related to string processing: text searching, pattern recognition, computational biology, audio processing, etc.

The *edit distance* between two strings a and b, $ed(a,b)$, is defined as the minimum number of *edit operations* that must be carried out to make them equal. The allowed operations are insertion, deletion and substitution of characters in a or b. The problem of *approximate string matching* is defined as follows: given a *text* of length n, and a *pattern* of length m, both being sequences over an alphabet of size σ, and a maximum number of allowed errors $k < m$, find all segments (or "occurrences") in *text* whose edit distance to *pattern* is at most k. That is, report all text positions j such that there is a suffix x of $text[1..j]$ such that $ed(x, patt) \leq k$.

The classical solution is $O(mn)$ time and involves dynamic programming [11]. This solution is the most flexible to allow different distance functions. For the particular case of $ed()$, a number of algorithms have been presented to improve the worst case to $O(kn)$ or the average case, e.g. [8, 13, 5, 12, 4, 14, 15, 3]

Pattern matching on hypertext [6] has been considered only recently. The model is that the text forms a graph of N nodes and E edges, where a string is stored inside each node, and the edges indicate alternative texts that may follow the current node. The pattern is still a simple string of length m. It is also customary to transform this graph into one where there is exactly one character per node (by converting each node containing a text of length ℓ into a chain of

* This work has been supported in part by Fondecyt grants 1-950622 and 1-960881.

ℓ nodes). This graph has n nodes and e edges (note that n is the text size and $e = n - N + E$).

Approximate string matching over hypertext is not only motivated by the structure of the World-Wide-Web and the possibility to search sequences of elements across paths of references, but also because graphs model naturally complex processes. In [7] it is considered the possibility of using approximate string matching as a model for data mining, where the symbols are in fact events and sequences of interesting events (perhaps separated by uninteresting events) are sought. This corresponds to allowing only insertions into the pattern. A graph may be a functional description of a process (paths representing possible alternative sequences of events), and we may want to identify potentially dangerous sequences of events in the process under analysis.

The first attempt to define pattern matching on hypertext is due to Manber and Wu [9], which view a hypertext as a graph of files with no links inside (it is easy to transform any hypertext to that form, by ending the node at its first reference). They solve the problem for an acyclic graph in $O(N + mE + R \log\log m)$ (where R is the size of the answer).

Akutsu [1] solved the problem of *exact* pattern matching on a hypertext which has a tree structure in $O(n)$ time, while Park and Kim [10] extended this result to an $O(n + mE)$ algorithm for directed acyclic graphs and for graphs with cycles where no text node can match the pattern in two places.

Amir et al. [2] were the first in considering approximate string matching over hypertext. In this case they consider the graph with n nodes and e edges and want to report all nodes v where in the text graph there is a *suffix* x ending at node v such that $ed(x, patt) \leq k$. We say that x is a text suffix ending at v if there is a path in the graph ending at v such that the concatenation of all characters of the traversed nodes yields x.

Amir et al. prove that the problem is NP-Complete if the errors can occur in the text, and give an algorithm to solve the case of errors only in the pattern, which is $O(m(n \log m + e))$ time and $O(mn)$ space. Their algorithm can handle general graphs, not only acyclic ones.

We present a new algorithm for approximate pattern matching over hypertext graphs. For acyclic graphs, the algorithm is $O(m(n + e))$, which raises to $O(mk(n + e))$ for graphs with cycles. In both cases, our space complexity is $O(n)$, which is by far smaller than that of [2]. On the other hand, we improve their time complexity for a small number of errors, namely for $k = O(\log m)$ if $e = O(n)$ and for $kn = O(e \log m)$ otherwise. We also improve previous work in the case of acyclic graphs.

2 Rethinking the Classical Algorithm

The classical algorithm to solve the general approximate string matching problem [11] is defined in terms of a matrix $C[i, j]$. When used to compute edit distance between two strings a and b, we have that $C[i, j]$ is the edit distance between $a[1..i]$ and $b[1..j]$. Therefore $C[i, 0] = C[0, i] = i$ for all i, and the update

formula is

$$C[i,j] = (a[i] == b[j])\ ?\ C[i-1,j-1] : 1+\min(C[i-1,j], C[i,j-1], C[i-1,j-1])\,,$$

where in the minimization the term $C[i-1,j]$ corresponds to deleting the current character of the pattern, $C[i,j-1]$ to inserting the current text character into the pattern, and $C[i-1,j-1]$ to replacing the current character of the pattern by the current text character.

Now, if a turns out to be a short pattern of length m and b a long text of length n, and we want to search the approximate occurrences of the pattern into the text (i.e. text positions j such that the pattern occurs with at most k errors in a suffix of $text[1..j]$), almost the same algorithm can be applied. The only modification needed is to set $C[0,j] = 0$ for all j (so as to give each text position a chance to start a match).

The problem with a large text is space. In principle, we should store the $O(mn)$ size matrix C, which is prohibitively expensive. It is not hard to see, however, that to compute the column j of the matrix we only need to keep the column $j-1$. Therefore, it is enough to keep and "old" and a "new" column to do the job, at a total space complexity $O(m)$, which is very low. The time complexity does not change. For obvious reasons, the other alternative of computing the matrix row by row, keeping old and new rows at a space complexity of $O(n)$, has never been considered. However, this is what we propose if the text is a graph. See Figure 1.

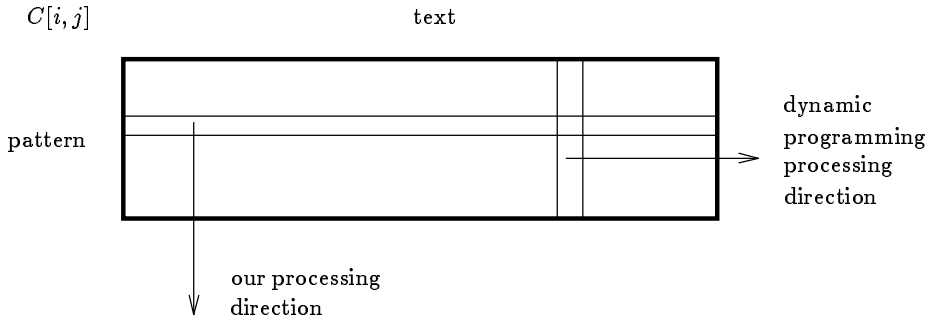

Fig. 1. The classical and our traversal of the dynamic programming matrix.

3 Applying the Algorithm to a Hypertext

Following [2], we first consider hypertexts where each node has just one character (it is easy to convert any hypertext to this form). Since the pattern keeps its

linear structure but the text does not, implementing the classical algorithm column-wise is difficult, because in a graph the notion of "advancing" in the text is not clear as in the linear version.

However, we take advantage of the fact that the pattern is still linear and apply the classical algorithm row-wise. That is, we perform m long iterations. At the end of iteration i, we have computed for every node v of the graph the best edit distance between $pattern[1..i]$ and any text suffix in the graph which ends at node v. We recall that x is a text suffix ending at v if there is a path in the graph ending at v such that the concatenation of all characters of the traversed nodes gives x. We denote by $t[v]$ the text character at node v.

The algorithm needs to keep a state per node, called $C[v]$. At each iteration new values for all $C[v]$, denoted $C'[v]$, are computed. This accounts for our $O(n)$ extra space. The pseudocode for the algorithm is presented in Figure 2.

```
for all v ∈ V, C[v] ← 0.
for i = 1 to m
   for all v ∈ V
       C'[v] ← min_{u/(u,v)∈E} f(u, v, patt[i])
   for all v ∈ V, C[v] ← C'[v]
```

Fig. 2. Our algorithm for approximate string matching on hypertext. The function $f()$ depends on the distance function used.

It is not hard to see that this algorithm takes $O(m(n+e))$ time and needs $O(n)$ extra space.

To follow the idea of the classical algorithm, the f function of the algorithm should be defined as

$$f(u,v,x) = (t[v] == x)\ ?\ C[u]\ :\ 1 + \min(C[u], C[v], C'[u])\ ,$$

the problem being to ensure that $C'[u]$ has been already computed. If the graph has no loops this is easily achieved by computing the new C' values in topological order (a topological sorting takes $O(n+e)$ time). This improves the previous result [2] both in time and space complexity.

However, this does not work in case of loops. The problem is that the insertion of the current text character into the pattern makes the current value of $C[v]$ to depend on its predecessors in the graph up to k nodes away. In a loop of length less than k, there seems not to be easy way to determine the proper place to start the computation of the values of the loop.

We solve the problem by not considering insertions in the f function. Instead, insertions are simulated by modifying the pattern. We take a new character $\sqcup$ that does not belong to the alphabet. This character can be deleted at zero cost, but replacing it costs the same as an insertion. We insert k such characters after each letter of the pattern. Therefore, if the algorithm would insert a text

character between two pattern characters, what it does now is to replace one of the $\sqcup$ characters. The others can be deleted at zero cost. We insert k special characters at each position to allow all the k insertions to occur at the same place, if necessary. Therefore, if the pattern is `aloha` and $k = 3$, we search for

$$\mathtt{a}\sqcup\sqcup\sqcup\mathtt{l}\sqcup\sqcup\sqcup\mathtt{o}\sqcup\sqcup\sqcup\mathtt{h}\sqcup\sqcup\sqcup\mathtt{a}\sqcup\sqcup\sqcup$$

and our new f function is

$$f(u, v, x) = (t[v] == x)\ ?\ C[u]\ :\ \min(1 + C[u], del(C[v], x))\ ,$$

where

$$del(C[v], x) = (x == \sqcup)\ ?\ C[v]\ :\ 1 + C[v]\ .$$

Since our pattern is now of length mk, the cost of the algorithm becomes $O(mk(n + e))$ when the graph has loops. This improves the previous result of [2] especially in space, since we need $O(n)$ extra space and they need $O(mn)$ extra space. We improve their $O(m(n \log m + e))$ time complexity for the case $k = O(\log m)$ if $e = O(n)$, and $kn = O(e \log m)$ otherwise.

4 Generalizations

We consider now the case where the text has a string at each node, instead of a single character. In this case we distinguish the total text size, n, from the number of nodes, N.

Since inside each node the text is linear, we can search at $O(kn)$ worst-case cost inside the node. The state of the search at character j of a node depends only on characters from $j - m - k + 1$ to j. Therefore, although the first $(m + k)$ text characters of each node still depend on the state of the global search (i.e. previous characters in the graph), the rest of the search at each node can be computed independently.

The final state of the search in node v has the information needed by the global search at the nodes that follow v in the graph. It is easy to modify the dynamic programming algorithm to keep count of the number of insertions performed in the pattern at each position. With that information it is possible to deduce which would be the state of the search at the end of node v for the global algorithm that uses the modified pattern.

Therefore, if there are N nodes we must perform at most $O(\min(n, N(m + k))) = O(\min(n, Nm))$ iterations of the global algorithm. The rest of the search on the whole text proceeds internally at each node at $O(kn)$ total cost.

Since our algorithm pays $O(mk)$ per node and per edge of the graph, our search cost is $O(mk(\min(n, mN) + E) + kn)$. This is $O(kn)$ provided $N = O(n/m^2)$ and $E = O(n/m)$.

The distance function can be easily modified to allow exact searching, or searching allowing only insertions (which is the case in data mining) or to give a particular edit cost to each operation.

5 Conclusions and Further Work

We have addressed the problem of approximate string matching when the text is a hypertext and the pattern is a string. The only previous algorithm is [2], which is $O(m(n \log m + e))$ time and $O(mn)$ space. We presented a new algorithm which in case of acyclic graphs is $O(m(n+e))$ and in case of graphs with loops is $O(mk(n+e))$ time. Our algorithm needs only $O(n)$ extra space.

The main problem that prevents an $O(m(n+e))$ time and $O(n)$ space algorithm is the combination of loops in the graph with operations that allow to insert text characters in the pattern. This situation creates circular dependencies that cannot be easily broken. We solved the problem by disallowing such operations and simulating them with a different, longer pattern. This solution is open to improvements and we are working at it.

References

1. T. Akutsu. A linear time pattern matching algorithm between a string and a tree. In *Proc. CPM'93*, pages 1–10, 1993.
2. A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. In *Proc. WADS'97*, LNCS 1272, pages 160–173, 1997.
3. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 1–23, 1996. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/cpm96.ps.gz`.
4. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, LNCS 644, pages 185–192, 1992.
5. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.
6. J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.
7. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Karkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, pages 12–27, 1997.
8. G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer Systems Science*, 37:63–78, 1988.
9. U. Manber and S. Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Proc. IAPR Workshop on Structural and Syntactic Pattern Recognition*, pages 22–33, Bern, Switzerland, 1992.
10. K. Park and D. Kim. String matching in hypertext. In *Proc. CPM'95*, pages 318–329, 1995.
11. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
12. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979, 1995.
13. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
14. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
15. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.