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 $O(m(n+e))$ time and needs only $O(n)$ extra space. This improves all previous results in both time and space complexity.

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

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

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. [7,12,4,13,14,3]

Pattern matching on hypertext [5] 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 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 [6] 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 [8], which view a hypertext as a graph of files with no links inside (it is easy to transform any hypertext to that form, by cutting the node at its first reference). They solve the problem for an acyclic graph in $O(N + mE + R \log\log m)$ time (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 (included) 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. For the case of errors only in the pattern, they give an algorithm which is $O(m(n \log m + e))$ time and $O(mn)$ space on cyclic or acyclic graphs.

In this work we improve both in time and space the previous results for ap-

proximate pattern matching on hypertext. We present an algorithm which is $O(m(n+e))$ time and $O(n)$ space for cyclic or acyclic graphs. An early version of this work (which was $O(mk(n+e))$ time on cyclic graphs) was presented in [9].

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

$$C_{i,j} = \text{if } (a[i] = b[j]) \text{ then } C_{i-1,j-1} \\ \text{else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) \ ,$$

where in the minimization the term $C_{i-1,j}$ corresponds to deleting the current character of a, $C_{i,j-1}$ to inserting the current character of b into a, and $C_{i-1,j-1}$ to replacing the current character of a by that of b.

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). Notice that we are in fact computing the edit distance, not only determining whether or not it is $\leq k$. This is a feature of this algorithm that translates into our generalizations.

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, 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). The formula for a row-wise update keeping the old and new rows is

$$C'_j = \text{if } (patt_char = text[j]) \text{ then } C_{j-1} \\ \text{else } 1 + \min(C_j, Cj-1', C_{j-1}) \ . \tag{1}$$

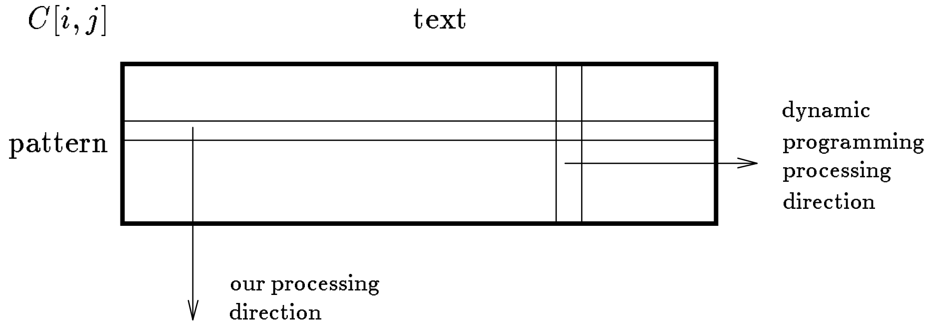

Fig. 1. The classical and our traversal of the dynamic programming matrix.

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 the 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.

```
Search (V,E,patt)
1.      for all v ∈ V, C_v ← 0.
2.      for i = 1 to m
3.          for all v ∈ V, C'_v ← f(v,i)
4.          for all v ∈ V, C_v ← C'_v
```

Fig. 2. First version of the algorithm for approximate string matching on hypertext.

To follow the classical formula of Eq. (1), the f function of the algorithm should be defined as

$$f(v,i) \;=\; \text{if } (patt[i] = t[v]) \text{ then } \min(\{C_u/(u,v) \in E\} \cup \{i\}) \\ \text{else } 1 + \min(C_v, \min_{u/(u,v)\in E} C'_u, \min_{u/(u,v)\in E} C_u) \;, \quad (2)$$

where the minima taken over empty sets yield an arbitrarily large value, and the mention of i stands for nodes with no arriving edges (which corresponds to the first column of the dynamic programming matrix). It is not hard to see that this algorithm takes $O(m(n+e))$ time and needs $O(n)$ extra space.

The problem is to ensure that C'_u have 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). 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 the *current* value of its predecessors in the graph. In a matrix, we solve this by computing the row from left to right, so that all the predecessors are already computed. But this is not the case of a graph with loops. We call this a "zero-time dependency" (using the metaphor that the matrix row represents the time). Figure 3 illustrates this case.

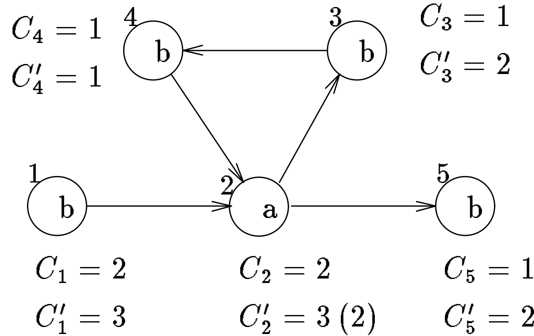

Fig. 3. A loop showing a zero-time dependency after processing the pattern `"bbbb"`. We show the C_v values before considering the last character, and the C'_v values after considering it. The number in parenthesis in C'_2 shows the change in the value that occurs when considering C'_4 again (after updating the loop).

Since an insertion which is propagated adds one error per step and we are interested in matches with up to k errors, we are only interested in the current values of the predecessors 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.

The problem can be solved 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

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

and we just change the f function to f':

$$\begin{aligned} f(v,i) &= \text{if } (patt[i] = t[v]) \text{ then } \min(\{C_u/(u,v) \in E\} \cup \{i\}) \\ &\quad\ \text{else } \min(del(v,i), \min_{u/(u,v)\in E} 1 + C_u) \\ del(v,i) &= \text{if } (pattern[i] = \sqcup) \text{ then } C_v \text{ else } 1 + C_v \ . \end{aligned}$$

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. It also improves 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. However, this technique can be improved as shown in the next section.

4 Improving the Algorithm for Cyclic Graphs

We present now a different technique to cope with cyclic graphs. The technique shown in the previous section simulates in fact a process of *insertion propagation*. To see this, consider a given node v. After updating C_v to C'_v without allowing insertions, we process k times the special character ("$\sqcup$"). This translates to the following update formula (iterated k times over all edges (u, v))

$$C'_v = \min(C_v, 1 + C_u)$$

This formula represents the insertion operations (which are propagated in the same row of the dynamic programming matrix, i.e. in "zero time"). This shows that we can, instead of using the special characters technique, avoid the use of the insertion operation in a first pass and then perform k passes propagating insertions. This is equivalent to the previous algorithm (although the previous one may be more intuitive).

As said, however, we can do better. We can keep track of which edges may need this update (i.e. those (u, v) where $C_v > C_u + 1$). A first list of candidates is

obtained by exhaustive search. Then, after the C_v value of a node v is reduced, all the edges leaving v may need the update too. As we show later, this reduces the complexity of the algorithm.

The new algorithm is shown in Figure 4. It is similar to the first version of Figure 2, except because of the propagation. The routine `Propagate` takes an edge (u, v) and determines if the edge needs insertion propagation. If it does, it changes the value of the target node v and recursively checks whether the change affects the edges with origin in v. The function g used is defined as

$$g(v,i) \;=\; \text{if } (patt[i] = t[v]) \text{ then } \min(\{C_u/(u,v) \in E\} \cup \{i\}) \quad \text{else } 1 + \min(C_v, \min_{u/(u,v)\in E} C_u) \;, \qquad (3)$$

where the insertion operation (and hence the zero-time dependency) has been eliminated. In fact, lines 3 to 5 aim at computing the f function correctly.

```
 Search (V,E,patt)
1.       for all v ∈ V, C_v ← 0.
2.       for i = 1 to m
3.           for all v ∈ V, C'_v ← g(v,i)
4.           for all v ∈ V, C_v ← C'_v
5.           for all (u,v) ∈ E, Propagate (u,v)

Propagate (u,v)
      if C_v > 1 + C_u
          C_v ← 1 + C_u
          for all z / (v,z) ∈ E
              Propagate (v,z)
```

Fig. 4. Final algorithm for approximate string matching on hypertext.

Notice that `Propagate` only works $O(1)$ time per edge whenever a reduction is made in its source node. That is, we perform a constant amount of work on all nodes of the form (u, v) each time C_u is reduced. This is the basis for the analysis that follows.

5 Analysis of the Algorithm

It is natural to wonder whether the algorithm of Figure 4 terminates at all in the propagation chain, especially in presence of loops. We prove not only that it terminates, but moreover, that it works $O(e)$ per character of the pattern. This is an amortized analysis over all graph edges. That is, although a given

edge can be updated $O(k)$ times for a given character of the pattern, the total amount of work when propagating insertions is $O(e)$ instead of $O(ke)$.

For each $(u,v) \in E$, let $D_{uv} = C_u$. That is, consider that the D value of an edge is the C value of its source node. We call C_u^i and D_{uv}^i the respective values after reading i characters of the pattern (assuming that they are correctly computed). Call G_i and F_i the sum of all D_{uv}^i, before and after propagating the insertions (line 5), respectively. We have that $F_0 = G_0 = 0$. Since for each unit of work done in the insertion propagation process we decrement a D_{uv}^i value, we cannot do more than $G_i - F_i$ operations in total. We prove now

(a) $F_i \;\leq\; F_{i-1} + e$

This is easily seen using the deletion operation of Eq. (2). For each $(u,v) \in E$, $D_{uv}^i = C_u^i \leq C_u^{i-1} + 1 = D_{uv}^{i-1} + 1$. Summing over all edges gives the result.

(b) $G_i \;\geq\; F_{i-1} - e$

To see this, we consider the three possible cases for the D_{uv} obtained without using insertions (the cases come from Eq. (3)). In all the three cases we prove that $D_{uv}^i \geq D_{uv}^{i-1} - 1$ for all $(u,v) \in E$. The result follows by simple summation over all the edges.

- $(match)$ $D_{uv}^i = C_u^i = C_z^{i-1}$ for some $(z,u) \in E$.

  Up to the character $i-1$ we correctly computed Eq. (2). Hence the insertion rule yields the relation $C_u^{i-1} \leq C_z^{i-1} + 1$. This shows that $C_u^i \geq C_u^{i-1} - 1$, and therefore $D_{uv}^i \geq D_{uv}^{i-1} - 1$. If u has no arriving edges, then $C_u^i \geq i - 1$ and $C_u^{i-1} \leq i - 1$ and the relation also holds.
- $(replacement)$ $D_{uv}^i = C_u^i = C_z^{i-1} + 1$ for some $(z,u) \in E$.

  Trivial once $(match)$ is proved, since we can even prove $D_{uv}^i \geq D_{uv}^{i-1}$.
- $(deletion)$ $D_{uv}^i = C_u^i = C_u^{i-1} + 1$

  Also immediate, since we obtain the stronger result $D_{uv}^i = D_{uv}^{i-1} + 1$.

We are ready for the analysis now. For each character of the pattern, the total amount of work due to insertion propagation is at most $G_i - F_i \;\leq\; 2e = O(e)$. We also add the e initial calls to the routine but the order does not change. Hence, for each pattern character we perform a normal iteration (lines 3-4) which costs $O(n+e)$ and the propagation of iterations which costs $O(e)$. The total cost is therefore $O(m(n+e))$.

As a final remark, notice that the depth of the stack in the `Propagate` routine cannot exceed m, since the value which is propagated is incremented in one at each new invocation and all the values are upper bounded by m. Hence, the space requirements remain the same.

In practical terms, the original algorithm of Figure 2 is still preferably on acyclic graphs (if the updates are performed on topological order), or on acyclic regions of the graph. However, the important result is that complexity does not change if we allow loops in the graph.

6 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 (beforehand, for example). Hence, we separate the first $m+k$ text characters of large nodes into nodes of one letter, therefore making sure that in what rests of the large node we can work independently of the global algorithm.

Therefore, if originally there are N nodes we end up with at most $\min(n, N(m+k)) = O(\min(n, Nm))$ nodes and $O(\min(e, E+Nm))$ edges after the separation of initial characters. The rest of the search on the whole text proceeds internally at each node at $O(kn)$ total cost.

Since our algorithm pays $O(m)$ per node and per edge of the graph, our search cost is $O(m(\min(n, mN) + \min(e, E+mN)) + kn)$. This is $O(kn)$ provided $N = O(nk/m^2)$ and $E = O(nk/m)$. This is the case of all but very fine-grained text graphs.

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

7 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. Previous algorithms achieved $O(m(n\log m+e))$ time and $O(mn)$ space [2], or $O(mk(n+e))$ time and $O(m)$ space [9]. We improved both previous algorithms to $O(m(n+e))$ time and $O(n)$ space, for cyclic and acyclic graphs. This is a complexity breakthrough over previous work.

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.

[4] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.

[5] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.

[6] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Karkäinen. Episode matching. In *Proc. CPM'97*, LNCS 1264, pages 12–27, 1997.

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

[8] 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.

[9] G. Navarro. Improved approximate pattern matching on hypertext. In *Proc. LATIN'98*, LNCS 1380, 1998. To appear.

[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] Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[13] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.

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