A Bipartite Matching Approach to Approximate String Comparison and Search

Samuel R. Buss* Peter N. Yianilos†

Abstract

Approximate string comparison and search is an important part of applications that range from natural language to the interpretation of DNA. This paper presents a bipartite weighted graph matching approach to these problems, based on the authors' linear time matching algorithms‡. Our approach's tolerance to permutation of symbols or blocks, distinguishes it from the widely used edit distance and finite state machine methods. A close relationship with the earlier related 'proximity comparison' method is established.

Under the linear cost model, a simple $O(1)$ time per position online algorithm is presented for comparing two strings given a fixed alignment. Heuristics are given for optimal alignment. In the approximate string search problem, one string advances in a fixed direction relative to the other with each time step. We introduce a new online algorithm for this setting which dynamically maintains an optimal bipartite weighted matching.

We discuss the application of our algorithms to natural language text search, including prefilters to improve efficiency, and the use of polygraphic symbols to improve search quality. Our approach is used in the LIKEIT text search utility now under development. Its overall design and objectives are summarized.

Keywords: *Approximate String Comparison, Approximate String Search, Text Search, Sequence Comparison, Bipartite Quasi-Convex Matching, Distance Metric, Natural Language Processing.*

*Supported in part by NSF grant DMS-9503247. Department of Mathematics, University of California, San Diego, La Jolla, CA 92093-0112. Email: `sbuss@ucsd.edu`.

†NEC Research Institute, 4 Independence Way, Princeton, NJ 08540 and Department of Computer Science, Princeton University, Princeton, NJ 08544. Email: `pny@research.nj.nec.com`.

‡Earlier related works [4, 3] of the authors may be obtained by anonymous ftp from `euclid.ucsd.edu`, directory `pub/sbuss/research`, filenames `quasiconvex.ps` and `quasiconvex.Ccode.ps`.

1 Introduction

Search and the implied process of comparison, are fundamental notions of both theoretical and applied computer science. The usual case is that comparison is straightforward, and attention rests mainly on search. Given finite strings over a finite alphabet Σ, one might for example define comparison to be a test for equality, and the well known dictionary problem arises. That is: organize a set of strings for effective storage and retrieval of exact matches. Given a short string α and a much longer one β, one might also strive to efficiently locate occurrences of α in β. This is sometimes called the string matching problem [8].

When the comparison method is generalized to allow inexact match, the resulting ideas, algorithms, and data structures become somewhat more complicated. Despite the added complexity, this research direction is important because the resulting algorithms are frequently of greater practical value. This is the case because almost all naturally occurring strings are the result of a generative process which includes error and ambiguity. Examples include natural language text, the speech signal, naturally occurring DNA — and just about any other sequence which corresponds to measurement of a natural process.

Perhaps the most natural generalized comparison method relaxes the requirement of exact equality to admit a bounded number of errors. Each such error is typically restricted to be either an insertion, deletion, substitution, or sometimes a transposition of adjacent symbols. Given a query string, it is then possible to build a finite state machine (FSM) to detect it, or any match within the error bounds, within a second string. We will refer to this approach generalizations of it as FSM methods. The recent work of [10, 12] demonstrates that text can be scanned at very high speeds within this framework for comparison.

Another well known approach to generalized string comparison computes edit-distance (ED), which measures the least costly transformation of one string into another using some set of primitive operations. The most common choices of primitive operations are insert, delete, substitute, and sometimes adjacent transposition. A simple dynamic program computes this distance in quadratic time, i.e. proportional to the product of string lengths. See [11] for a discussion of these and related algorithms which we will refer to as ED methods.

Both the FSM and ED approaches rather strictly enforce temporal ordering. In most applications this is to some extent desirable. We observe however that similar strings from natural language exhibit strong local temporal agreement but frequently include global ordering violations. More concretely, words or entire clauses might be rearranged with the result nevertheless "similar" to the original. For example, the strings "ABCD" and "DCBA" are maximally distant with respect to edit distance[§]. To transform the first into the second, 'A', 'B', and 'C' are first deleted, and then 'C', 'B', 'A' are inserted. This is not disturbing at the word level, but if each symbol is replaced by a word, significant divergence from human similarity judgment becomes apparent.

This paper presents a formal basis for string comparison and search which is entirely distinct from both the FSM and ED paradigms. It represents a computationally affordable solution to the problem of preserving an emphasis on local temporal ordering, while allowing global permutation. Our framework reduces string comparison to multiple instances of the bipartite weighted matching problem, or as it is sometimes called, the assignment problem. The bipartite graph's two parts correspond to the strings being compared. The nodes are symbols or, more generally, features present at each string position. Edges connect symbols in one string with those in another and are

[§] We assume only insertion and deletion operators for simplicity

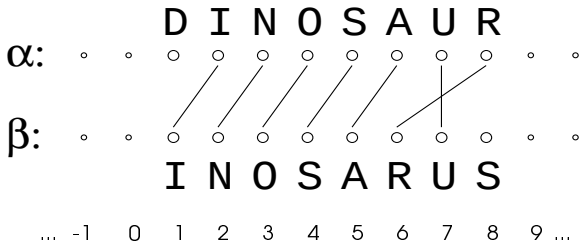

Figure 1: String Comparison by Bipartite Weighted Matching

given costs which vary with the magnitude of positional displacement. Local temporal ordering is emphasized by defining features which consist of the polygrams of varying lengths found at each string position. The overall solution is then superposition of assignment problems for each distinct polygram. The sections that follow describe our method in greater detail.

The "proximity comparison" method is a related approach used in several commercial spelling error correctors and information retrieval products. We show that this earlier method may be viewed as an approximation to the assignment problem we solve exactly. Its principal virtue is the algorithm's simplicity and resulting speed. However proximity comparison requires two passes over the strings being compared while we demonstrate that exact solution is possible with only one. In both cases $O(1)$ time is spent processing each symbol. Thus the algorithms of this paper essentially supplant the proximity comparison approach. Polygrams and other context sensitive feature symbols were used in proximity comparison based applications, but for practical reasons, polygram length was generally limited to two. The "Friendly Finder" program[¶] is an example of a proximity comparison based database retrieval system based entirely on 1 and 2-grams and their positional relationship. More recently Damashek and Huffman [5, 6] have developed a large scale text retrieval system based entirely on polygrams.

It is convenient to imagine two strings α and β arranged on top of one another as in Figure 1. In this example the graph's nodes consist of unigrams only, i.e., single letters. In the 'linear-cost' model the matching's cost is determined by simply adding the horizontal displacements of each graph edge, and imposing a penalty based on the number of unmatched nodes. It is evident from the figure that a lower cost matching would result if α were shifted left by one position. This illustrates that the cost of an optimal matching is a function of the string's relative 'alignment'. In some applications fixed alignment is assumed while others require that alignment also be optimized. In the case of string search, the long β string is imagined to advance leftward with each time step. Then problem then is to maintain an optimal matching dynamically as time advances and the alignment changes. We'll describe this problem in greater detail later and describe an algorithm which is somewhat more efficient than the obvious solution which finds an entirely new matching for each step.

Figure 1 illustrates that each edge in the matching connects identical symbols. Thus, for a given alignment, the overall matching problem may be decomposed into a union of subproblems – one for each alphabet member. We will show that each of these may be further decomposed into 'levels'. The level decomposition is trivially produced in one-pass using what amounts to the "parenthesis nesting algorithm". Finally, for each level, a simple one-pass $O(1)$ time per node algorithm finds the optimal matching. Since the combined sizes of all subproblems equals the size of the original

[¶]First published in 1987 by Proximity Technology Inc.

problem, the original problem is solved in one-pass, using $O(1)$ time per symbol.

Finally we discuss the text search application and techniques which improve processing speed and search quality. Polygraphic symbols of variable length are employed at each node to increase the system's sensitivity to local ordering changes. Before searching for the optimal matching and alignment, several prefilter stages are employed to effectively rule out most candidates. The LIKEIT system now under development combines these techniques. In practice a database record may be located given almost any query which remotely resembles it. The system is character and polygram oriented and does not make any effort to extract words from the query or the database. It is perhaps best described as an approximate phrase retrieval system which tolerates re-ordering while considering word proximity.

2 Theoretical Foundations

2.1 Bipartite Matching for String Comparison

We start with the mathematical definitions of strings, of bipartite matchings and of the cost of a matching. In spite of the mathematical abstraction of the definitions we present, the underlying idea is quite simple; namely, given a *query string* α and a *database string* β we wish to find an optimal one-to-one assignment of symbols occurring in α to occurrences of the same symbols in β. Here 'optimality' is measured in terms of the distance from a symbol in α to its assigned symbol in β. The 'distance' corresponding to each such symbol assignment depends on our choice of alignment for α relative to β. For this reason, our definition of a string will specify not just its contents, but a position along the integer number line. The notion of an optimal assignment is then well-defined since a position is implicit in the definition of each string. Later we will consider a higher level optimization problem, namely that of varying the relative position (alignment) of the strings so as to optimize (minimize) the assignment cost.

A string is intuitively a sequence of characters over a finite alphabet Σ; formally, a string α has a domain $[k,m] = \{k, k+1, k+2, \ldots, m\}$ and is a mapping $\alpha : [k,m] \to \Sigma$. We denote by $\alpha(i)$ the symbol at position i of α. Notice that only when a string's domain is $[1,m]$ is $\alpha(i)$ really the i-th symbol. Allowing domains more general than $[1,m]$ makes it easier to describe the process of searching for an optimal alignment, and the process of searching for approximate occurrences of α in β.

Let α be a string with domain $[k,m]$. If $n \in \mathbb{Z}$, then we let $\alpha \ll n$ denote the string β with domain $[k-n, m-n]$ such that $\beta(i) = \alpha(i+n)$. The length, $|\alpha|$, of α is equal to $m-k+1$, the number of symbols in α. We use $\alpha, \beta, \gamma, \ldots$ as meta-symbols for strings and $\sigma, \tau, \ldots$ as meta-symbols for symbols from Σ.

Definition Let α and β be strings. A (bipartite) matching of α into β is a partial one-to-one mapping $\pi : dom(\alpha) \to dom(\beta)$ such that $\beta(\pi(i)) = \alpha(i)$, for all $i \in dom(\pi)$. We denote by $|\pi|$ the number of values where π is defined.

A *cost function* $c(\cdot)$ is a mapping from $\mathbb{N} \cup \{\perp\}$ to $\mathbb{R}$. The *two-sided cost* of a matching π under a cost function c is equal to

$$\left(\frac{|\alpha| + |\beta|}{2} - |\pi|\right) c(\perp) \; + \sum_{i \in dom(\alpha)} c(|\pi(i) - i|)$$

The *one-sided cost* of a matching π from α into β is equal to

$$(|\alpha| - |\pi|)c(\perp) \; + \sum_{i \in dom(\alpha)} c(|\pi(i) - i|).$$

The two-sided cost is more useful in the *string comparison* problem where one wants a notion of the distance between α and β; on the other hand, the one-sided cost is more useful for the *string search* problem where one wants to know to what extent α is contained within β. Note that if α and β are equal length, then the one-sided and two-sided costs are identical.

Definition We write $dist(\alpha, \beta)$ to denote the optimal two-sided cost of a matching from α to β.

The function *dist* enjoys the property of being a metric:

Theorem 1 *Let the cost function c be non-decreasing and satisfy $c(0) = 0$ and $c(i) + c(j) \geq c(i + j)$ for all $i, j > 0$. Then $dist(\cdot, \cdot)$ is a metric. That is, $dist(\alpha, \alpha) = 0$, $dist(\alpha, \beta) = dist(\beta, \alpha)$ and $dist(\alpha, \beta) + dist(\beta, \gamma) \leq dist(\alpha, \gamma)$, for all α, β and γ.*

Proof The properties $dist(\alpha, \alpha) = 0$ and $dist(\alpha, \beta) = dist(\beta, \alpha)$ are immediate from the definition. For the triangle inequality, let let π_1 and π_2 be a minimum cost matchings of α and β and of β and γ, respectively. Form their composition $\pi_3 = \pi_1 \circ \pi_2$. It will suffice to show that the cost of π_3 is less than or equal to the sum of the costs of π_1 and π_2. For this, it is convenient to reexpress the cost of π_3 as

$$\sum_{i \in dom(\pi_3)} c(|\pi_3(i) - i|) \; + \sum_{i \in dom(\alpha) \setminus dom(\pi_3)} \frac{1}{2} c(\perp) \; + \sum_{i \in dom(\gamma) \setminus ran(\pi_3)} \frac{1}{2} c(\perp),$$

and to think of the costs of π_1 and π_2 being expressed in similar form. Now consider the five following kinds of values $i \in dom(\alpha)$ and $k \in dom(\gamma)$ that contribute to the cost of π_3:

$i \notin dom(\pi_1)$*:* Such an i contributes $\frac{1}{2}c(\perp)$ to cost of π_1 and, since it is not in $dom(\pi_3)$, contributes the same to the cost of π_3.

$i \in dom(\pi_1) \setminus dom(\pi_3)$*:* Such an i contributes $\frac{1}{2}c(\perp)$ to the cost of π_3, and since $\pi_1(i) \notin dom(\pi_2)$, $\pi_1(i)$ contributes $\frac{1}{2}c(\perp)$ to the cost of π_2.

$k \notin ran(\pi_2)$*:* Such a k contributes $\frac{1}{2}c(\perp)$ to the cost of π_3 and the same to the cost of π_2.

$k \in ran(\pi_2) \setminus ran(\pi_3)$*:* Such a k contributes $\frac{1}{2}c(\perp)$ to the cost of π_3, and since $\pi_2^{-1}(k)$ is not in $ran(\pi_1)$, $\pi_2^{-1}(k)$ contributes $\frac{1}{2}c(\perp)$ to the cost of $\pi - 1$.

$i \in dom(\pi_3)$*:* Then $\pi_3 : i \mapsto \pi_2(\pi_1(i))$ contributes $c(|\pi_2(\pi_1(i)) - i|)$ to the cost of π_3. Corresponding to this, there is a contribution of $c(|\pi_1(i) - i|)$ to the cost of π_1 and a contribution of $c(|\pi_2(\pi_1(i)) - \pi_1(i)|)$ to the cost of π_2. And by the fact that c satisfies the triangle inequality,

$$c(|\pi_2(\pi_1(i)) - i|) \; \leq \; c(|\pi_2(\pi_1(i)) - \pi_1(i)|) + c(|\pi_1(i) - i|).$$

In every case the contribution to the cost of π_3 is less than or equal to the sum of contributions to π_1 and π_2. □

There are a large number of enhancements that can be made to the definitions of cost and distance. For example, it is possible to assign every character a real-valued *weight* and then weight the costs proportionally to the weight of the character. It is also possible to more heavily weight characters near the beginning of α, etc. These enhancements can be very useful in practice to improve the perceived quality of the optimal assignment; however, they make little difference to the algorithms described in this paper, so we shall use just the simple notion of cost as defined above.

In [4] the authors have developed linear time and near-linear time algorithms for finding minimum cost bipartite matchings when the cost function is concave down. Although space does not permit us to review the details of this algorithm, we shall review a few keys aspects of the algorithm which are needed in the case when the cost function is linear (i.e., when $c(i) = c_0 \cdot i$ for some constant $c_0 > 0$).

For simplicity assume that α and β are both strings with domain $[1, n]$.[‖] Let σ be a symbol which occurs at position i in α, i.e., $\alpha(i) = \sigma$. Then $level_\alpha(i)$ is equal to the number of σ's that occur in $\alpha[i, i-1]$ minus the number that occur $\beta[1, i-1]$. Similarly, if $\tau = \beta(j)$, then $level_\beta(j)$ is equal to the number of τ's in $\alpha[1, i]$ minus the number in $\beta[1, i-1]$ minus 1. A simple, but important, fact about optimal matchings is:

Proposition 2 ([1, Lemma 1] or [4, Lemma 4]) *Let c be concave down and assume $c(\perp) > c(i)$ for all i. There is an optimal cost matching π such that, for all i, $level_\alpha(i) = level_\beta(\pi(i))$ whenever $\pi(i)$ is defined.*

Proposition 2 provides the first step towards an algorithm for finding an optimal matching. This first step is to separately consider each symbol σ and to calculate the level of each occurrence of σ in both α and β; for each level ℓ, we can separately find an optimal matching on the σ's at level ℓ. Taking the union of the these matchings gives an optimal matching for the original strings. This allows us to reduce the problem of finding the optimal bipartite matching between α and β to the following problem:

Definition Let $i_1 < i_2 < \cdots < i_q$ be positions of a symbol σ in α and $j_1 < j_2 < \cdots < j_r$ be positions of σ in β. Also suppose r equals either $q-1$ or q, and that $i_a \leq j_a$ and $j_a < i_{a+1}$ for all a. (For example, these occurrences of σ are all the occurrences of σ at a given level.) The *alternating matching problem* for these occurrences of σ is to find a maximal partial mapping π from $\{i_1, \ldots, i_q\}$ into $\{j_1, \ldots, j_r\}$ which minimizes the total cost

$$\sum_{a \in dom(\pi)} c(|\pi(i_a) - i_a|).$$

Note that $dom(\pi)$ must contain r many points. When $r = q$, the alternating matching problem is said to be *balanced*, otherwise it is *unbalanced*.

There is a dual formulation of this alternating matching problem where the first occurrence of σ in β precedes the first occurrence in α. In this case, r is equal to either q or to $q+1$, and $j_a < i_a$ and $i_a \leq j_{a+1}$ for all a.

[‖]This simplifying assumption holds without any loss of generality, since we can always pad strings with new symbols which do not occur in the other string, so the strings then have the same domain. Then a common shifting makes both strings begin with position 1.

A complete solution to the alternating matching problem, and thereby the matching problem, for concave-down cost functions was given by the authors in [4]. A particularly simple, yet useful, special case of this is when the cost function is linear. In this case, the following theorem explains how the algorithm of [4] is simplified.

Theorem 3 *Let the cost function be linear (and $c(\perp)$ arbitrary). Let an instance of the alternating matching problem be given as above. Then the minimum cost matching π is one of the following:*

(a) *If $r = q$, then $\pi(i_a) = j_a$ for all i_a's.*

(b) *If $r = q + 1$, then there is some $1 \leq b \leq r$ such that $\pi(i_a) = j_a$ for all $a < b$, such that $\pi(i_b)$ is undefined, and such that $\pi(i_a) = j_{a-1}$ for all $a > b$.*

(c) *Dually to (b), if $r = q - 1$, then there is some $1 \leq b \leq r$ such that $\pi(i_a) = j_a$ for all $a < b$ and such that $\pi(i_a) = j_{a+1}$ for all $a \geq b$.*

Part (a) is a simple special case of the well-known Skier and Skis problem, see [7, 9]. Part (b) is the generalization to unbalanced matching.

Proof Recall the notion of "jumper" from [4]. First suppose $r = q$. Since the cost function is linear, there is never any need to put a jumper, assuming ties are broken in favor of not adding a jumper. Therefore the optimal matching is the matching $\pi(i_a) = j_a$ for all a.

Now suppose $r = q + 1$. Following [4], this unbalanced problem is reduced to a balanced quasiconvex tour by adding a new phantom occurrence of σ at j_r; the cost function is modified so that an edge $\pi : i_a \mapsto j_r$ contributes zero to the total cost (i.e., the new phantom node j_r has distance zero from all other nodes). As before, ties may be broken in favor of not using jumpers. Therefore, by the methods of [4] the optimal matching π has some i_b such that $\pi(i_b) = j_r$ and since there are no other jumpers, $\pi(i_a) = j_a$ for $a < b$ and $\pi(i_a) = j_{a-1}$ for $a > b$.

The case (c) is dual to (b) and essentially the same proof works. □

In [4] we proved that there is a linear time algorithm to solve the alternating matching problem of Theorem 3 (since a linear cost function clearly satisfies the *weak crossover condition*; the algorithm, as explained there, made two passes over the words from left to right. For the special case of a linear cost function, there is a much simpler online algorithm which uses $O(1)$ time per symbol:

Theorem 4 *Let $i_1, \ldots, i_r$ and $j_1, \ldots, j_q$ be an instance of the alternating matching problem and assume w.l.o.g. that $i_1 \leq j_1$. There is an online algorithm, which scans the values $i_1 < j_1 < i_2 < j_2 < \ldots$ in sequential order using constant time per value, which finds the optimal matching described in Theorem 3.*

Proof W.l.o.g., we are in the setting of part (b) of Theorem 3 with $r = q + 1$ and we are trying to find the value b so that the matching π with $\pi(i_b)$ undefined is minimal cost. Since the cost function c is linear, we may assume w.l.o.g. that $c(i) = i$. Let $c_s = j_s - i_s$ and $d_s = i_{s+1} - j_s$. The cost of the matching π which omits i_b from its domain is clearly equal to

$$\sum_{s=1}^{b-1} c_s + \sum_{s=b}^{q} d_s \quad = \quad \sum_{s=1}^{q} d_s + \left(\sum_{s=1}^{b-1} c_s - \sum_{s=1}^{b-1} d_s \right).$$

Let $m_b = \sum_{s=1}^{b-1} c_s - \sum_{s=1}^{b-1} d_s$, since $\sum_{s=1}^{q} d_s$ is constant, it will suffice to find a value b which minimizes m_b. Now the algorithm is very simple: As $i_1, j_1, \ldots, i_r$ are being read sequentially, the value of m_b is updated with a single addition or subtraction per node; the minimum value of m_b is remembered along with corresponding value of b. At the end of the scan, the stored value for b which minimizes m_b describes an optimal cost matching and m_b gives its total cost.

Note the algorithm uses on a constant amount of space, since it is not necessary to store all the m values, only the current values of m_b and b plus the minimal value for m_b seen so far and its associated b need to be kept. □

2.2 Realignment

The previous section considered the problem of finding the minimum cost matching between two strings α and β; will the positions of the two strings were held fixed. Frequently, however, it is desirable to allow the strings to be shifted relative to one another to find a better matching. An example of this would be matching the string $\alpha =$ "SOUR" against $\beta =$ "DINOSAUR". If the strings are aligned so that both α has domain $[1,4]$ and β has domain $[1,8]$, then the total cost will be $c(4) + c(2) + c(4) + c(4)$. However, by realigning the strings and shifting β leftward by 4 (thereby letting it have domain $[-3,4]$, the total cost is reduced to $c(0) + c(2) + c(0) + c(0)$.

We call the problem of finding the optimal shift of β the *realignment* problem. We give below two heuristics for finding a good realignment. The first, based on 'center-of-gravity', is quick to implement but does not always produce an optimal free realignment. The second, based on a median calculation always produces an optimal free realignment, but is more time-consuming to compute.

Let α and β be fixed strings; we define C_n to equal $dist(\alpha, \beta \ll n)$. The problem is find a n which minimizes C_n and to find a corresponding optimal cost matching π. The general idea is to iteratively calculate an optimal match at a given alignment, then to calculate a "free realignment" which improves the cost of the match, and then to repeat the process. More formally, we define a free realignment as follows:

Definition Let α and β be strings, let $n \in \mathbb{N}$, and let $\pi : dom(\alpha) \to dom(\beta \ll n)$ be a partial matching. A *free realignment* of α and $\beta \ll n$ under π is specified by an integer parameter m: the cost of the free realignment is equal to

$$D(m) \;=\; \sum_{i \in dom(\pi)} c(|(\pi(i) - m) - i|).$$

The general approach we take to realignment is based on the following algorithm:

Algorithm *Realign*(α, β, n)
    Set $n = 0$
    Loop
        Set π = *optimal matching of* α *and* $\beta \ll n$.
        Let $m = FreeRealign(\pi)$
        If $m = 0$,
            Exit loop
        Set $n = n + m$
    Endloop

Here *FreeRealign*(π) is a function that computes a free realignment.

We suggest two possible methods of computing *FreeRealign*: the first is based on a 'center-of-gravity' calculation: the *center of gravity*, $CoG(\pi)$, of π is defined to equal

$$CoG(\pi) \;=\; \frac{1}{|\pi|} \cdot \sum_{i \in dom(\pi)} (\pi(i) - i),$$

rounded to nearest integer. Then one can use $FreeRealign(\pi) = CoG(\pi)$. The CoG function is easy to compute, is fast, and has worked well in the LIKEIT system described in section 3.4. However, it is not guaranteed to always yield the best possible free realignment: an example of this is when the string $\alpha =$ "ABC" is compared to the string $\beta =$ "A - - - BC" where the dashes represent a long string of blanks; in this case, the CoG free realignment would reposition β so that α is placed at the two-thirds mark of β. However, the optimal free realignment, i.e., the one with minimum cost, would be to have the end of α positioned at the end of β so that the substrings "BC" are perfectly aligned.

Our second free realignment algorithm is based on the median value (rather than the mean) of the differences $\pi(i) - i$. The median p of a multiset X of integers is defined as usual; if X has an even number of members, then there are two "middle" elements $p_1 \leq p_2$ of X and every p, $p_1 \leq p \leq p_2$ is considered to be a median value for X. Let $Median(\pi)$ be defined to equal the (i.e., any) median of the multiset $\{pi(i) - i : i \in dom(\pi)\}$. Then the following theorem shows that $FreeRealign(\pi) = Median(\pi)$ is an optimal free realignment.

Theorem 5 *FreeRealign*$(\pi) = Median(\pi)$ *gives a minimum cost free realignment of* α *and* β *under* π.

Proof Recall π is a matching of α and $\beta \ll n$. W.l.o.g., let $c(i) = i$ for all $i \geq 0$. Consider the difference between $D(m)$ and $D(m+1)$:

$$D(m+1) - D(m) \;=\; \sum_{i \in dom(pi)} \left(|\pi(i) - m - i + 1| - |\pi(i) - m - i|\right).$$

This is easily seen to be equal to the number of i such that $\pi(i) - i \geq m$ minus the number of i such that $\pi(i) - i < m$. So $D(m+1) > D(m)$ if and only if m is greater than the (or every) median of $\{\pi(i) - i\}$ and conversely, $D(m+1) < D(m)$ if m is less than every median of $\{\pi(i) - i\}$. From this, it follows that $D(m)$ is minimized when m is any median of the multiset $\{\pi(i) - i\}$. □

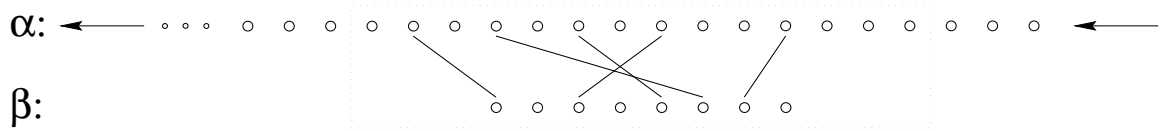

Figure 2: In Approximate String Search an optimal Bipartite Weighted Matching is maintained within a fixed window as text slides by a stationary query.

Unfortunately, although the use *Median* for free realignment always provides an optimal *free* realignment, this does not mean that *Realign* must converge to an alignment which is even locally optimal. For example, suppose that $\alpha =$"BCA" with domain $[1,3]$, that $\beta =$"ACBXA" with domain $[1,5]$, that $n = 0$, and that π is the optimal matching for this alignment with $\pi(1) = 3$, $\pi(2) = 2$ and $\pi(3) = 1$. The (only) median value for this π is $m = 0$. However, the alignment is not locally optimal, since there is a matching π' of α and $\beta \ll 1$ which has lower cost, assuming $2c(2) + c(0) > 3c(1)$; namely, $\pi'(1) = 2$, $\pi'(2) = 1$ and $\pi'(3) = 4$.

For this reason, the above realignment remains merely a useful heuristic, which works well in practice, but may fail to produce an optimal alignment in some rare situations. Both the *CoG* function and the *Median* function are computable in linear time; but, in practice, the *CoG* is slightly simpler and quicker to compute and seems to yield very good results. Therefore, the *CoG*-based free realignment is used in the prefilter stages of the LIKEIT system, while the *Median* approach when computing the final matching.

2.3 Bipartite Matching for String Search

Our development above began by considering the problem of approximate string comparison given a fixed alignment of the query string relative to a database string. We then discussed optimization of this alignment. But in both cases the setting was *record oriented*, i.e., it was assumed that α and β are complete records such as words, names, etc. In this section we consider the different but related problem of approximate string search using our bipartite matching outlook. Here, one of the two strings is generally much longer, e.g., a paragraph or an article, and represents the text to be searched. The shorter string is thought of as the human generated query. The text is imagined to be in motion passing by the query as depicted in Figure 2.

Our earlier record oriented approach could in principle be applied in this setting but several problems exist. First, the local search heuristics introduced cannot be relied upon to produce a global optimum alignment, so one would be led to recompute the entire matching for each alignment. Second, under the linear cost function, the agreement between our mathematical notion of similarity, and human judgment, appears to break down substantially. To see this consider that varying the position of a distant matching symbol affects the score just as much as nearby variations. If one is searching for keywords without regard to their position, this may be reasonable behavior. But if the query represents a phrase or collection of words which are expected to occur in close proximity, then those parts of the query which match only to distance locations, can easily dominate the overall matching score. This in part motivated the authors to consider non-linear quasi-convex cost functions in their earlier work.

The approach we take in this paper is to use linear costs, but restrict the matching process to a moving window in the text indicated with a dotted line in Figure 2. That is, matching edges

are allowed only within the window. For each position of the text relative to the query, a different bipartite matching may exist. The problem we face then consists of efficiently maintaining an optimal matching as the text slides by.

As before, the problem separates immediately by alphabet symbol and we may therefore focus on the subproblem corresponding to a single symbol. We will again use the notion of a level decomposition but now must from time to time adjust it as the text slides by. Focusing on a single level, it is important to realize that the optimal matching may itself change as the text moves; before the need arises to recompute levels. We think of these two eventualities as events for which an alarm is set sometime in the future, i.e. the events consisting of the recomputation of levels, and of the optimal matching for a level. Until one of these alarms sounds, it is easy to see that each matching edge will grow either shorter or longer by one with each time step. An edge will never change growth direction between alarms because such a change would correspond to a releveling event. Hence in-between alarms, the overall matching cost will change by a constant amount per time step given by the total of each edge change.

The algorithms we describe work by computing for each symbol subproblem, the cost of the optimal matching, the number of time steps before an alarm event, and the change in matching cost per time step. The cost changes for each symbol are summed to form a single value. So until an alarm sounds for some symbol, processing a time step consists of adding this single value to adjust the cost of the optimal matching. A trivial outer loop manages the alarms and updates the cost change variables.

In the worst case, alarms might sound on nearly every time step and our algorithm is no better than the direct approach in which one merely recomputes the optimal matching cost for at every time step. But in some problem domains such as natural language text, the alphabet may be defined so that considerable savings result. For example one can define the alphabet to consist of, say, 2-grams or 3-grams but exclude 1-grams. Doing this makes the problem considerable more sparse and many subproblems will *sleep* for many time steps before any work is needed. So if w is the window's size, our algorithms might perform $O(w)$ work per time step. One might hope to find and algorithm with instead does $O(1)$ or $O(\log w)$ work per step, perhaps on an amortized basis. This represents an interesting area for future work. Failing such a result, one might analyze algorithms like the one we present under the assumption of some distribution of strings.

Since the outer level processing is rather straightforward, we confine our discussion to the matching subproblem for a single symbol which we will denote σ. Our algorithm represents the occurrences of σ in β as a linked list which is formed before search begins. The occurrences in α are also maintained as a linked list but this list is sometimes updated. An entry is added to the tail when σ is shifted onto (becomes the) right-end of α during a time step. The list's head is deleted when σ is shifted out of the window. Assuming a constant number of symbols are present at each string position the maintenance of these lists is clearly $O(1)$ cost per step. No lists need to be formed for alphabet symbols in α which do not occur in β.

Time steps are denoted t and numbered from zero. We denote by s the offset of the windows right edge from the first position of α. The list entries for α consist of an integer giving the time step during which the particular σ occurrence was shifted onto α. The list entries for β consist of the position of each σ occurrence in the natural coordinate system in which the first position is zero. At each time step the β lists are effectively renumbered by adding $t - s$ to each element before it is used.

We will show that a single scan through the two lists associated with σ finds the optimal

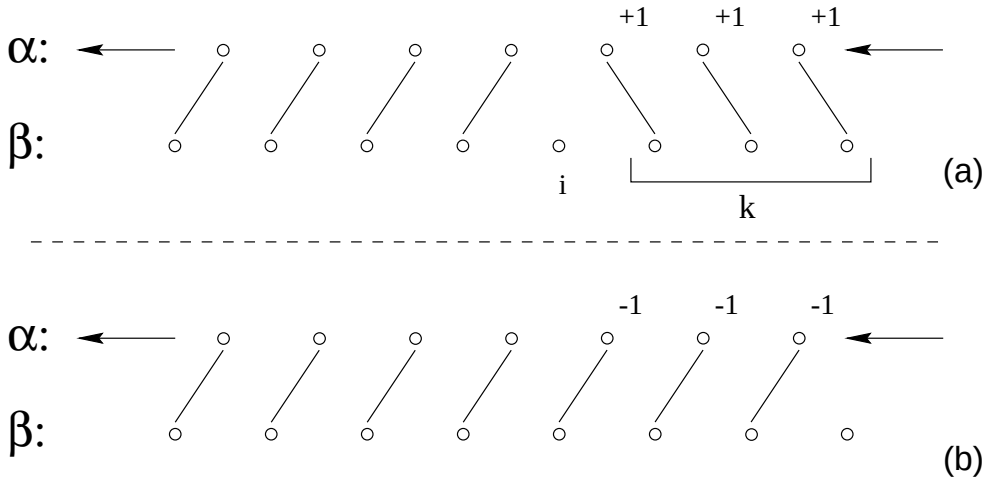

Figure 3: The cost of leaving the ith element unmatched minus the cost of leaving the last element unmatched, increases by $2k$ with each time step.

matching, its cost, and the two alarm intervals corresponding to leveling and rematching as described above. This scan is performed when an earlier alarm sounds, and also when σ either appears on the right, or vanishes from the left of the window.

The first part of the process consists of merging the two lists to produce a single sorted list. During the merge it is a straightforward matter to split the list into sublists, one for each level. Also, during the merge, it is easy to set the leveling alarm by focusing on positions in the merged list where a β list element is immediately followed by an α list element. The minimum gap between such pairs gives the value of the leveling alarm and may be trivially maintained during the scan.** We have seen in an earlier section that the optimal matching and its cost may be found for each level in an online fashion, so we have only to show that the rematching alarm can be set as well.

Before turning to this we remark that an $O(1)$ or $O(\log w)$ per step algorithm will need to either leave leveling behind, or maintain the levels using non-trivial data structures. To see this consider that a block of say 10 σ occurrence in α moving over a block of 10 σ occurrences in β will generate a leveling event at every time step. Moreover a quadratic number of matching edge changes are generated despite the fact that the canonical skis and skiers matching might have been used; avoiding all leveling alarms, matching edge changes, and rematching alarms. This is of course only true for examples like the one above in which the number of σ occurrence in α exactly equals the number in β. Still though it may provide intuition leading to an improved algorithm for the general case.

Focusing now on an individual level, recall that it must consist of elements which alternate between α and β. If level is of even length, then all elements are matched and no rematching is required until the leveling itself changes. If the level is of odd length, then exactly one element will be left unmatched in an optimal matching. Since we are processing the level online, we don't know in advance whether it is of odd or even length. But this poses no real problem since we may assume that it is odd until we reach the end and perhaps find out otherwise.

Assume without generality that the situation is as in Figure 3, i.e. the level begins with a β

**During the merge ties are broken in favor of β.

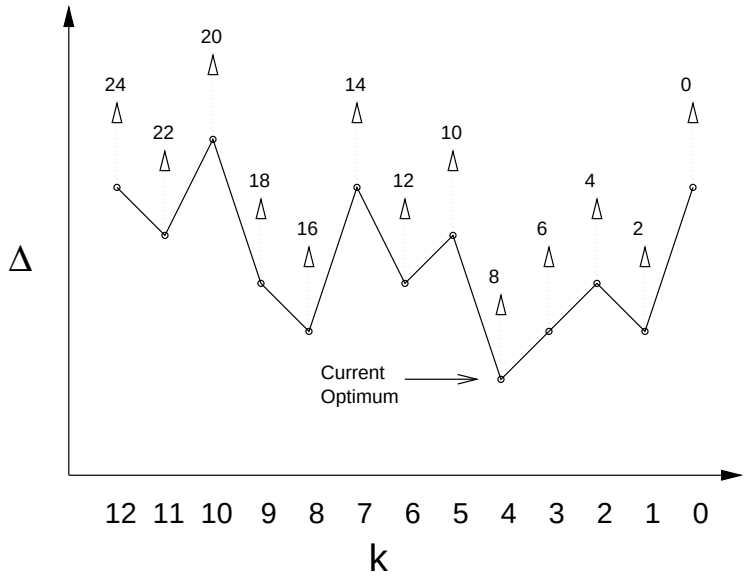

Figure 4: Graph of the difference computed in Figure 3. Minima correspond to optimal matchings, and as time passes the Δ values increase at a rate which increases linearly from right to left.

element, that the level is of odd length, and that β contains the unmatched element. Part (a) of this figure depicts the matching in which element i of β is left unmatched. Part (b) shows the matching in which the last element is excluded. As α slides to the left, the edges to the left of i in both parts of the figure, decrease in cost by 1. However the k edges to the right of i behave differently. In part (a) they grow more costly by 1 while in (b) their cost declines by 1. So the difference between the cost of these two matchings, i.e. that of part (a) minus that of part (b), increases by 2 with each time step. Of course this is true only until elements *cross*, but by that time a leveling alarm will have sounded.

For each position i we denote this difference $\Delta(i)$. The optimal matching corresponds to a minimum of this function's graph. We now imagine animating the graph with respect to time using the $2k$ per step rule derived above. Each value $\Delta(i)$ may be expanded to become a function of time: $\Delta(i,t) = \Delta(i) + 2kt$ where k denotes the level's length minus i. It is apparent that the as we move from left to right through a level, the rate of increase in Δ declines. This situation is illustrated in Figure 4. So as time passes, the values of i which represent minima will in general change. Notice that the the absolute vertical location of this graph does not affect the identification of an optimal matching since the minima are unchanged by vertical translation. Also observe that the differences $\Delta(i+1) - \Delta(i)$ are easily computed online from local differences within the level list. So up to vertical translation, this graph may be built online starting from an arbitrary value and accumulating $\Delta(i+1) - \Delta(i)$ values.

We will show that as the graph is built from left to right, we can easily and in constant time update the value of variable *alarm* which gives the number of time steps until the graph has a new global minimum. This corresponds of course to a change in the optimal matching and is the "rematching" alarm we require to complete our algorithm for string search. Two observations are all that is needed to understand the short algorithm that follows. First, if during left to right graph construction, a new global minimum is encountered, then no earlier point can ever replace it since all

earlier points are increasing in value at a greater rate. Second, if the current point is greater than or equal to the current global minimum, then the number of time steps that will pass before the current point becomes smaller than the current global minimum, is exactly $(\Delta(i) - \Delta_{\min})/(2(i - i_{\min}))$. The following algorithm results:

Algorithm MatchAlarm
  Set $\Delta_{min} = \infty$
  For $i = 0, 1, \ldots ...$
    If $\Delta(i) < \Delta_{min}$
      Set $\Delta_{min} = \Delta(i), i_{min} = i$, $alarm = \infty$
    Else
      $t = (\Delta(i) - \Delta_{min})/(2(i - i_{min}))$
      If $t < alarm$
        Set $alarm = t$

2.4 Relationship to Proximity Matching

In this section, we give a new characterization of what we here call the *proximity metric* on strings. This metric and various generalizations, developed primarily by the second author, has been reported in [16, 13, 15, 2] and has been used extensively for natural language applications, especially spelling correction, by Proximity Technologies and a large number of word processing software publishers, and by Franklin Electronic Publishers in their hand-held solid-state electronic books. Other product implementations include the PF474 VLSI circuit [14], the "Friendly Finder" database retrieval system, the "Proximity Scan" subroutine library, and the "Clean Mail" program for eliminating duplicates in mailing lists. We are proposing the bipartite graph matching approach to string matching as an improvement over the proximity metric for string matching. The proximity metric has a fast, efficient implementation and has proven to work well for many natural language applications. However, we argue below that our new bipartite graph matching approach provides a qualitative improvement over the proximity matching and, as we have already seen, the bipartite graph based string matching can be performed efficiently in linear time, so its performance rivals that of the proximity metrics.

Space does not allow a complete treatment of the proximity metric, but we do include its full definition and proofs of the main new results. For the rest of this section, we let α and β have equal length and both have domain $[1, n]$.

Definition We let $\alpha[i, j]$ denote the substring of α from position i to position j.

For each symbol σ, let $Common_\sigma(\alpha, \beta)$ equal the minimum of the number of occurrences of σ in α and the number in β. Let

$$Common(\alpha, \beta) \;=\; \sum_{\sigma \in \Sigma} Common_\sigma(\alpha, \beta)$$

which is the number of (occurrences of) symbols common to both α and β. The *proximity similarity* of α and β is defined to equal

$$sim^\theta(\alpha, \beta) \;=\; \sum_{i=1}^{n} Common(\alpha[1, i], \beta[1, i]) + \sum_{i=1}^{n} Common(\alpha[i, n], \beta[i, n]).$$

Note that $Common(\alpha[1,i],\beta[1,i])$ is at most i. Therefore, $sim^\theta(\alpha,\beta)$ is at most $n(n+1)$. Finally, define the *proximity distance* to equal

$$dist^\theta(\alpha,\beta) \;=\; n(n+1) - sim^\theta(\alpha,\beta).$$

It is clear that $dist^\theta(\alpha,\beta)$ has value 0 if and only if $\alpha=\beta$, and has value $n(n+1)$ if and only if α and β have no common symbols. Traditionally, the proximity metric has been a scaled version of sim^θ; namely, $\theta(\alpha,\beta) = sim^\theta(\alpha,\beta)/(n(n+1))$. The next theorem is proved in [13], and we omit its proof here:

Theorem 6 *Fix $n \geq 0$. Then $dist(\alpha,\beta)$ is a metric.*

It has been known that $dist^\theta$ acts somewhat like a matching-based string comparison method with linear cost function; it also has been noted that the $dist^\theta$ metric acts counter-intuitively when there are unequal numbers of a given symbol in α and β. These phenomena are both explained by the following characterization of $dist^\theta$.

Before giving the characterization of $dist^\theta$, we give a slightly revamped definition of the *dist* function; which is basically a reformalization of the discussion in section 2.1 where the algorithm finding an optimal cost matching was implemented to first split symbols into alternating tours at each level and then to find an optimal matching for each tour independently. We shall work with the cost function $c(i)=i$ and $c(\perp)=(n+1)$. Let $\sigma\in\Sigma$ be a symbol and $\ell\in\mathbb{N}$ be a level number: a (σ,ℓ)-matching π on α and β is a maximal matching which maps (positions of) occurrences of σ at level ℓ in α to (positions of) occurrences of σ at level ℓ in β. (I.e., π is injective is either total or onto.) The cost of such a π is equal to $\sum_{i\in dom(\pi)} c(|\pi(i)-i|)$. Recall that the level ℓ occurrences of σ in α and β form an alternating tour and that Theorem 3 describes the ways in which minimum cost (σ,ℓ)-matchings can be formed. If case (a) of Theorem 3 holds, then we say α and β are (σ,ℓ)*-balanced*; otherwise, there are unequal numbers of occurrences of σ at level ℓ in α and β and the strings are said to be (σ,ℓ)*-unbalanced*. We define $dist_{\sigma,\ell}(\alpha,\beta)$ to be the minimum two-sided cost of (σ,ℓ)-matchings on α and β. When there are no occurrences of σ at level ℓ, then $dist_{\sigma,\ell}(\alpha,\beta)=0$. Proposition 2 tells us that the bipartite graph string matching distance can be defined as

$$dist(\alpha,\beta) \;=\; \sum_{\sigma\in\Sigma}\sum_{\ell\in\mathbb{N}} dist_{\sigma,\ell}(\alpha,\beta).$$

Now to give a similar characterization of $dist^\theta$, let $dist^\theta_{\sigma,\ell}$ be defined by:

$$dist^\theta_{\sigma,\ell} \;=\; \begin{cases} dist_{\sigma,\ell} & \text{if } \alpha,\ \beta \text{ are } (\sigma,\ell)\text{-balanced} \\ \frac{n+1}{2} & \text{otherwise} \end{cases}$$

Note that in the case where α and β are (σ,ℓ)-unbalanced, $dist^\theta(\alpha,\beta)$ is just a constant, independent of the positions of level ℓ occurrences of σ. It may seem slightly counterintuitive that the constant is $\frac{n+1}{2}$ instead of $n+1$, since if α and β differ by exactly one character substitution then $dist^\theta(\alpha,\beta)=n+1$. However, in this case, there are two values of (σ,ℓ) for which there is an unbalanced tour; one for the character in α and one for the substituted character in β. Each of there two unbalanced tours contributes $\frac{n+1}{2}$ as a $dist^\theta_{\sigma,\ell}$ which is a net contribution of $n+1$.[††]

[††]Another way to think about why $\frac{n+1}{2}$ is used, is that $dist^\theta_{\sigma,\ell}$ is intended to be a variation of a two-sided cost.

Theorem 7 $dist^{\theta}(\alpha,\beta) \;=\; \sum_{\sigma\in\Sigma}\sum_{\ell\in\mathbb{N}} dist^{\theta}_{\sigma,\ell}(\alpha,\beta)\,.$

Proof Let $RHS(\alpha,\beta)$ be a shorthand notation for the double summation in Theorem 7. We shall prove Theorem 7 by induction on $Common(\alpha,\beta)$. The base case is when α and β have no common symbols. In this case, $dist^{\theta}(\alpha,\beta) = n(n+1)$. Also, there are exactly $2n$ distinct pairs (σ,ℓ) such that σ occurs (once) in at level ℓ in one of the strings. Thus $RHS(\alpha,\beta)$ is also equal to $n(n+1)$.

For the induction step, pick a symbol σ and a level ℓ such that σ occurs in both α and β at level ℓ. Let $i_1 < i_2 < \cdots < i_q$, and $j_1 < j_2 < \cdots < j_r$, be the positions where σ appears in α, and in β respectively, at level ℓ. Without loss of generality, $i_a \le j_a$ and $j_a < i_{a+1}$ for all a. Let Δ_α and Δ_β be new symbols and define α^* to be obtained from α by replacing the σ at position i_1 with Δ_α and define β^* similarly by replacing σ at position j_1 with Δ_β. By the induction hypothesis, $dist^{\theta}(\alpha^*,\beta^*)$ is equal to $RHS(\alpha^*,\beta^*)$, so it will suffice to show that

$$dist^{\theta}(\alpha^*,\beta^*) - dist^{\theta}(\alpha,\beta) \;=\; RHS(\alpha^*,\beta^*) - RHS(\alpha,\beta).$$

To prove this, first suppose the strings are (σ,ℓ)-balanced with $q = r$. Consider the difference between $RHS(\alpha^*,\beta^*)$ and $RHS(\alpha,\beta)$; in the former, the unmatched symbols Δ_α and Δ_β contribute $n+1$ to the distance, and in the latter, Theorem 3(a) tells us that the occurrences of σ are positions i_1 and j_1 are matched to each other in the optimal (σ,ℓ)-matching, so they contribute $j_1 - i_1$ to the distance. Therefore,

$$RHS(\alpha^*,\beta^*) - RHS(\alpha,\beta) \;=\; (n+1) - (j_1 - i_1).$$

Now consider the difference between $dist^{\theta}(\alpha^*,\beta^*)$ and $dist^{\theta}(\alpha,\beta)$. For the former, the unmatched symbols provide nothing to the "$Common$" values. For the latter, the two occurrences of σ increase $Common(\alpha[1,i],\beta[1,i])$ by 1 for all $i \ge j_1$ and they increase $Common(\alpha[i,n],\beta[i,n])$ by 1 for all $i \le i_1$. Therefore,

$$dist^{\theta}(\alpha^*,\beta^*) - dist^{\theta}(\alpha,\beta) \;=\; (n - j_1 + 1) + (i_1) \;=\; n + 1 - (j_1 - i_1).$$

as desired.

Now suppose the strings are (σ,ℓ)-unbalanced with $q = r+1$. Consider the difference between $RHS(\alpha^*,\beta^*)$ and $RHS(\alpha,\beta)$; in α and β, the (σ,ℓ) tour was unbalanced and contributed $dist^{\theta}_{\sigma,\ell}(\alpha,\beta)$; however, α^* and β^* are not only (σ,ℓ)-unbalanced but are also $(\Delta_\alpha,0)$- and $(\Delta_\beta,-1)$-unbalanced. Each of these two contribute an extra $(n+1)/2$ to the distance, so

$$RHS(\alpha^*,\beta^*) - RHS(\alpha,\beta) \;=\; (n+1).$$

Now consider the difference between $dist^{\theta}(\alpha^*,\beta^*)$ and $dist^{\theta}(\alpha,\beta)$. For the former, the unmatched symbols provide nothing to the "$Common$" values. For the latter, the two occurrences of σ increase $Common(\alpha[1,i],\beta[1,i])$ by 1 for all $i \ge j_1$ and they increase $Common(\alpha[i,n],\beta[i,n])$ by 1 for all $i \le j_1$. Therefore,

$$dist^{\theta}(\alpha^*,\beta^*) - dist^{\theta}(\alpha,\beta) \;=\; (n - j_1 + 1) + (j_1) \;=\; n + 1.$$

as desired. □

The string matching algorithms based on bipartite graph matchings now are seen to present several advantages over the proximity matchings. The principal two advantages are: (1) The proximity matching approach is based on counting common occurrences of symbol and does not give any assignment (i.e., matching) of symbols in one string to symbols in the other string. The bipartite graph approach to string matching does give such an assignment. It can be potentially very useful to have such as assignment; firstly, since it shows explicitly correspondences between the strings, and secondly since the assignment can be used in a post-processing phase to further evaluate the correspondence between the matched strings. (2) The second advantage is the proximity matching seems to have serious flaws when symbols do not occur equal numbers of times in both strings. For an example consider the strings $\alpha =$"ABA" and $\beta =$"BAB" and $\gamma =$"ABB". Here we have

$$dist^{\theta}(\alpha,\beta) = 6 \qquad dist(\alpha,\beta) = 6$$
$$dist^{\theta}(\alpha,\gamma) = 6 \qquad dist(\alpha,\gamma) = 4$$

(To normalize these distances to the range $[0,1]$, they should be divided by 12; e.g., $dist(\alpha,\gamma) = 4$ reflects the fact that α and γ differ in exactly $1/3$ $(= 4/12)$ of their symbols.) But now one clearly feels that γ is more similar to α than β is. Therefore, the fact that $dist^{\theta}(\alpha,\beta) = dist^{\theta}(\alpha,\gamma)$ is undesirable, but $dist(\alpha,\gamma) < dist(\alpha,\beta)$ is as desired.

See the conclusion for a discussion of the relative merits of the bipartite graph approach to string matching and of the edit-distance (ED) approach to string matching.

3 Natural language search

3.1 Polygraphs

In the bipartite approach to string matching discussed above, a single string comparison is decomposed into a linear superposition of matching problems; one for each alphabet symbol. In this framework, even extensive permutation of a string can result in a new string which is very close to the original, as measured by the minimum cost of a matching between the new string and the original string. For natural language applications, this behavior presents problems because perceptual distance increases rapidly with local permutational rearrangement, while the cost of an optimal matching does not. Fortunately, there is a simple way to augment the string matching algorithm discussed above, to overcome these problems; namely, to add additional polygraphic symbols to the strings before performing the comparison.

The polygraphic symbols, called *polygrams*, consist of a sequence of ordinary symbols. That is, a n-gram, is a sequence of m-symbols from the underlying alphabet Σ. Before comparing the strings α and β, we preprocess them by adding, at each character position, new symbols for the n-grams which end at that position of the string. To give a concrete example, LIKEIT system has used to the following method of preprocessing a string α with domain $[1,m]$: for each value $i = 2,\ldots,m$ and each j such that $2 \le j \le \min(i,6)$, let $G_{i,j}$ be the j-gram comprising symbols $\alpha(i-j+1)\cdots\alpha(i)$; then the polygrams $G_{i,2}\cdots G_{i,\min(i,6)}$ are inserted into α immediately following symbol i of α. A similar preprocessing is applied to β. The number 6 has been chosen empirically as it works well for natural language text applications. For other applications, e.g., DNA sequence analysis, it might be more appropriate to use polygrams of different lengths.

This preprocessing makes α and β nearly six times as long, so one might fear that it would increase the runtimes by a factor of six. However, in practice, the preprocessing greatly improves the perceived quality of the matches and greatly reduces the run time of the search process. The reason for the paradoxical reduction of the runtime of the search is that the prefilterings discussed in the next section, will do a much more effective filtering thereby reducing greatly the number of actual string comparisons which must be performed. Note that insertion of polygraphic characters can be done efficiently, by a finite state machine scanning from left to right only.

The LIKEIT system also gives different weights to polygrams than to single symbols; namely, a n-gram receives a weight of n, so the weight of a polygram is proportional to its length.

It should be noted that when polygrams are used, the optimal matching may not respect overlapping polygrams. That is to say, even though two polygrams overlap in α, they may be matched to widely divergent polygrams in β. As an example consider matching the strings $\alpha =$"BART" and $\beta =$"BARCHART". The initial 3-gram "BAR" in α would match the initial 3-gram of β; whereas the overlapping 3-gram "ART" in α would match to the final 3-gram of β.

Polygrams are only one kind of extra symbols that can be inserted into strings before matching occurs. Other natural text search applications have used the addition of symbols encoding phonetic information into the strings; in this approach, phonetic information is inserted into both α and β using an algorithmic method (that is, the phonetic information is all inserted algorithmically, there is no use of a dictionary).

3.2 Prefilters

Our matching approach may be thought of as a crude model for human similarity judgment. Even though the corresponding are linear time and very simple, it is in practice important to consider even simpler models and algorithms in order to increase search speed. Perhaps the simplest model consists of comparing the set of polygrams which occur in both strings, along with their frequencies. That is, ignore position entirely. This may be thought of as a crude projection of the matching model so as to use a cost function which assumes a constant value for all matching edges independent of edge length. It corresponds to the use of frequencies only in [5, 6]. Other model simplifications involve approximations we have already seen. Examples are the use of free realignment only, the CoG heuristic, and approximate solutions to each level's matching problem. Another simplification consists of dealing fewer polygram sizes. These simplified algorithms form *prefilters* which are applied to the database in order to limit the search to a subset which is then processed by later prefilters and ultimately by the final algorithm. As records are considered by a prefilter, they are added to a heap of bounded size. The next prefilter reads from this heap and writes to a new one. For particular domains, transaction logs or query models may be used to intelligently set the size of these heaps so that some estimate of the probability of error (failing to pass-on the correct record) is acceptable.

3.3 Alphabet Element Relationships

Our framework allows for matching edges only between identical characters. This in some sense corresponds to a discrete metric on Σ, i.e. a distance function assuming zero only upon equality, and unity otherwise. In many languages this is awkward and in some it may be an unacceptable simplification. The simplest current practical technique known to the authors consists of mapping each alphabet element to a string. The mapping then attempts to capture linguistic relationships.

3.4 The LIKEIT Text Search Engine: Summary Preview

The ideas of this paper and the authors' earlier work have lead to the development of LIKEIT; a new text search engine for the Web [17]. The weighted matching approach represents a single conceptual framework that deals with variations in word spelling, form, spacing, ordering, and proximity within the text.

The system's objective is to provide an easy to administer, distributed search service offering a great deal of error tolerance, and a very simple user interface. The Web user's query is communicated to one or more a LIKEIT search servers and an HTML document is returned containing the most similar information found. The optimal matching is visualized for the user by highlighting selected characters in the returned text. Relationships between servers may be established to form a distributed network search service which can include explicit hierarchy when desired.

When a database of roughly 50,000 bibliographic citations is searched for the query PROBLMOFOPTIMLDICTIONRY the title OPTIMAL BOUNDS ON THE DICTIONARY PROBLEM is returned first. The search requires a little over two seconds on a 133MHz Pentium processor based system. In a more realistic example, a search for "SAMBUSS" finds papers by the first author as intended. Keyword-based systems such as GLIMPSE and WAIS cannot by design. We remark that the algorithms of this paper might then find application in construction of front-end spelling correctors for keyword-based systems such as these.

4 Conclusions and Future Work

There are several interesting directions for future work. Our algorithms for approximate string search might be improved to provide constant or nearly constant time bounds per time step. Strategies for polygraphic indexing or hashing might be explored so that prefilters can more rapidly limit the search's range. General techniques for dealing with relationships between elements of Σ while retaining computational efficiency might be developed. The alphabet symbol weights and other parameters might be in some way learned by the system. Finally, other application areas such as DNA analysis might be explored.

To conclude, we review again some of the relative advantages and disadvantages between the bipartite graph string matching (BGSM) algorithms discussed in this paper and prior edit distance (ED) algorithms. First, our BGSM algorithms have the advantage of being online and taking linear time; in contrast, ED algorithms are based on dynamic programming and require quadratic time in general (but they can be made linear time if only a limited number of edit operations are allowed). The BGSM algorithms give a better quality match when the strings being compared contain substantial differences; whereas the ED approach gives a higher quality matching when the two strings differ by only a few edit operations (e.g., three or fewer). This is primarily due to the fact that the BGSM algorithms match distinct alphabet symbols independently. However, with use of polygraphic symbols, this disadvantage of the BGSM algorithms can be largely overcome. The ED distance algorithms have been widely used in text search applications. A precursor to the BGSM algorithms, the proximity matching, has also been widely used. The BGSM algorithm provides a qualitative improvement over the proximity matching; and is being deployed as a general purpose tool in the LIKEIT text search package.

References

[1] A. Aggarwal, A. Bar-Noy, S. Khuller, D. Kravets, and B. Schieber, *Efficient minimum cost matching using quadrangle inequality*, in Proceedings of the 33th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 1992, pp. 583–592.

[2] S. R. Buss, *Neighborhood metrics on n-dimensional blocks of characters*, Tech. Rep. 00218-86, Mathematical Sciences Research Institute, Berkeley, September 1985.

[3] S. R. Buss, K. G. Kanzelberger, D. Robinson, and P. N. Yianilos, *Solving the minimum-cost matching problem for quasi-convex tours: An efficient ANSI C implementation*, Tech. Rep. CS94-370, U.C. San Diego, 1994.

[4] S. R. Buss and P. N. Yianilos, *Linear and $O(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours.* To appear in *Siam J. Comput.*. An extended abstract of this paper appeared in *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994, pp. 65-76., 199?

[5] M. Damashek, *Gauging similarity with n-grams: Language-independent categorization of text*, Science, 267 (1995), pp. 843–848.

[6] S. Huffman and M. Damashek, *Acquaintance: A novel vector-space n-gram technique for document categorization*, in Proceedings, Text REtrieval Conference (TREC-3), Washington, D.C., 1995, NIST, pp. 305–310.

[7] R. M. Karp and S.-Y. R. Li, *Two special cases of the assignment problem*, Discrete Mathematics, 13 (1975), pp. 129–142.

[8] D. E. Knuth, J. James H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing, 6 (1977), pp. 323–350.

[9] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976.

[10] U. Manber and S. Wu, *GLIMPSE: A tool to search through entire file systems*, in Proceedings of the Winter 1994 USENIX Conference, 1994, pp. 23–32.

[11] D. Sankoff and J. B. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.

[12] S. Wu and U. Manber, *Fast test searching allowing errors*, Communications of the ACM, 35 (1993), pp. 83–91.

[13] P. N. Yianilos, *The definition, computation and application of symbol string similarity functions*, Master's thesis, Emory University, 1978.

[14] ——, *A dedicated comparator matches symbol strings fast and intelligently*, Electronics Magazine, (1983).

[15] P. N. Yianilos and S. R. Buss, *Associative memory circuit system and method, continuation-in-part.* U.S. Patent #4490811, December 1984.

[16] P. N. Yianilos, R. A. Harbort, and S. R. Buss, *The application of a pattern matching algorithm to searching medical record text*, in IEEE Symposium on Computer Applications in Medical Care, 1978, pp. 308–313.

[17] P. N. Yianilos and K. G. Kanzelberger, *The LikeIt distributed Web search system.* Manuscript, in preparation, 1995.