ELSEVIER

Theoretical Computer Science 181 (1997) 159–179

Theoretical Computer Science

Block edit models for approximate string matching[1]

Daniel Lopresti*, Andrew Tomkins

Matsushita Information Technology Laboratory, Panasonic Technologies, Inc., Two Research Way, Princeton, NJ 08540, USA

Abstract

In this paper we examine *string block edit distance*, in which two strings A and B are compared by extracting collections of substrings and placing them into correspondence. This model accounts for certain phenomena encountered in important real-world applications, including pen computing and molecular biology. The basic problem admits a family of variations depending on whether the strings must be matched in their entireties, and whether overlap is permitted. We show that several variants are NP-complete, and give polynomial-time algorithms for solving the remainder.

1. Introduction

The edit distance model for string comparison [13, 18, 26] has found widespread application in fields ranging from molecular biology to bird song classification [20]. A great deal of research has been devoted to this area, and numerous algorithms have been proposed for computing edit distance efficiently (e.g., [2–4, 12, 14, 25]). For a recent survey, see [23].

In a previous paper [15], we introduced a new application of edit distance in the realm of pen computing. *Approximate ink matching*, or AIM, is the concept of matching handwritten/drawn queries against an existing ink database. While ink is an expressive two-dimensional medium, its creation, when viewed in the temporal domain, is an inherently one-dimensional process: the path of a stylus tip against a writing surface. Ink can be treated as a string by taking pen input from a digitizing tablet and segmenting it into strokes, extracting a standard set of features (e.g., stroke length, total angle traversed), and clustering the resulting vectors into a smaller number of basic stroke types. It then becomes possible to compare strings over this "ink" alphabet using approximate string matching techniques.

For handwritten text (English and Japanese, cursive and printed), our empirical studies indicate that this approach, which is writer-dependent, performs quite well.

* Corresponding author. E-mail: dpl@research.panasonic.com

[1] Originally presented at the *Second South American Workshop on String Processing*, Valparaiso, Chile, April 1995.

PII S0304-3975(96)00268-X

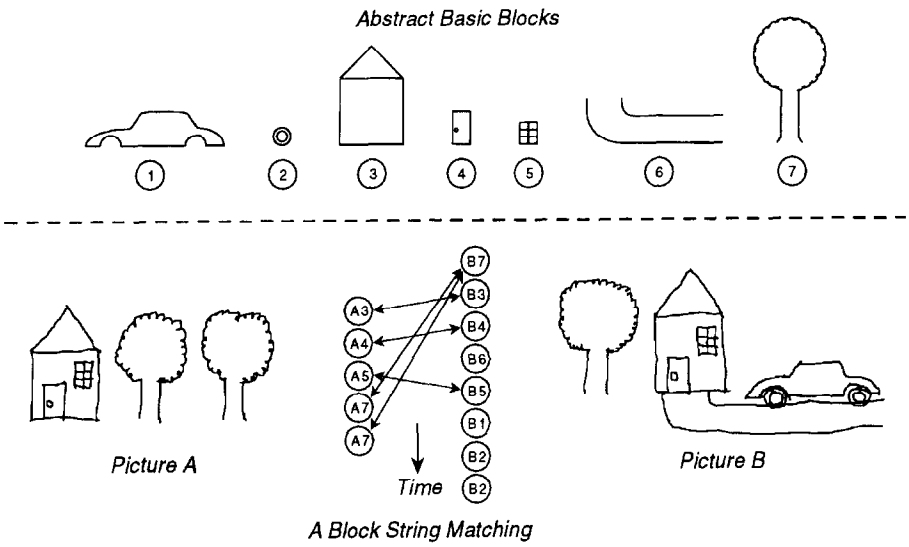

Fig. 1. Approximate string matching applied to hand-drawn pictorial data.

However, the situation becomes more complicated for pictorial data. Certain substructures within a larger image can correspond stroke-for-stroke, but these basic "blocks" may have been drawn by the user in an otherwise arbitrary order. Fig. 1 demonstrates this; the two trees in Picture A are drawn last, while the tree in Picture B is drawn first. Moreover, if the goal is to search a database, the best match may be imprecise in the sense that certain elements are omitted or repeated. This phenomenon is also illustrated in Fig. 1. Intuitively, we judge the two pictures to be quite similar, even though Picture A has an extra tree and is missing the car and driveway. Existing string matching algorithms are not flexible enough to capture these forms of *block motion*.

Likewise, in genetic sequence alignment, some biologists suggest that comparisons based on simple edit distance may fail to account for certain common evolutionary processes [8]:

> "Global dynamic programming alignments of such rearranged sequences yield unpredictable, evolutionarily confusing results. ... Global alignment methods are generally incapable of dealing with intrasequence rearrangements, yet this phenomenon is quite common among mosaic and repetitive sequence proteins." [p. 96].

Manual inspection of a "dot matrix" plot appears to be the most popular approach for addressing this problem today.[2] As shown in Fig. 2, to compare two sequences A and B, a table of size $|A| \times |B|$ is built and a dot placed at the (i,j)th entry if the ith

[2] Also from [8]: "Dot matrix analysis is the only currently available tool that deals sensibly with this phenomenon." [p. 96]

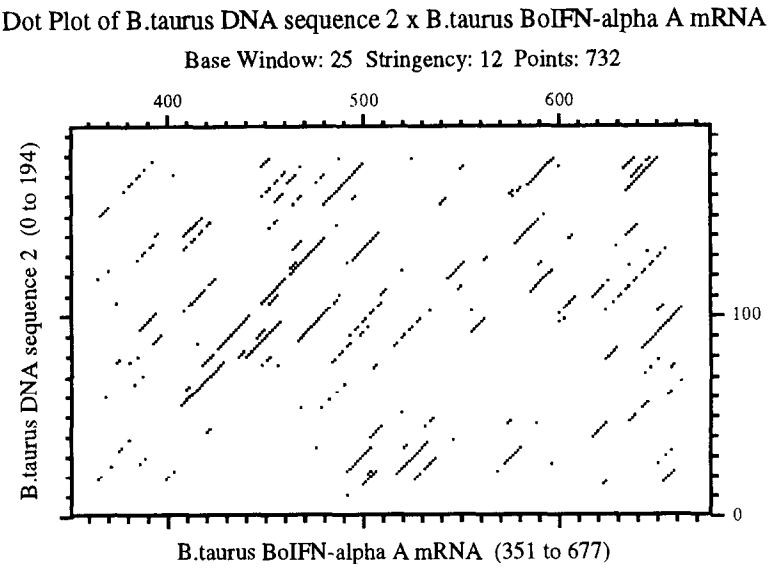

Fig. 2. Dot matrix plot of two short genetic sequences.

symbol of A is the same as the jth symbol of B. To reduce noise, a minimum number of exact matches within a window centered around the location in question can be required before a dot is placed there. In our example, the window is 25 nucleotides and must contain at least 12 matches. The resulting plot is then examined visually for interesting similarities.

Tichy has examined a special case of the problem where the blocks themselves must match exactly [24]. However, this is of limited value in applications where the flexibility of allowing the blocks to be edited to create a better correspondence is important.

In another approach, a number of researchers have addressed the problem of finding good local alignments, or sets of alignments, that avoid unconserved regions [1, 6, 9, 21]. Smith and Waterman [22] and later Waterman and Eggert [27] showed how to locate the best local alignment, and then how to iterate the process to determine the next-best non-overlapping alignment. Others have extended this approach, resulting in better time and space complexities [10, 11].

All of this earlier work shares the same strategy: find the optimal alignment, then find the best alignment that does not overlap this, then find the best alignment that does not overlap the first two, etc. This approach yields a series of local alignments with costs $p_1, p_2, \ldots, p_k$ such that no other series of alignments with costs $q_1, q_2, \ldots, q_k$ is better, under lexicographic order. That is, $p_1 \leqslant q_1$, and, if they are equal, $p_2 \leqslant q_2$, etc.

In this paper, we describe a family of models for the string block edit problem. These formalize in a succinct and rigorous way the notions illustrated in the preceding discussion: blocks can be moved and matched freely, while individual characters within blocks can be edited in the traditional way. This work can be distinguished from

previous efforts in that it focuses on finding a set of alignments that is optimal under a different criterion: the L_1 norm, or *sum-of-costs*. That is, it determines a set of block alignments with costs $\{p_1, p_2, \ldots, p_k\}$ such that for any other set of alignments with costs $\{q_1, q_2, \ldots, q_k\}$, we have $\sum_i p_i \leqslant \sum_i q_i$. We prove that certain variants of the problem are NP-complete, and give polynomial-time algorithms for the remainder. We conclude the paper by suggesting some directions for further research.

2. Block edit models

Standard edit distance allows the relationship between two strings to be expressed graphically by means of an *alignment*. An example showing how "quick brown fox" can be mapped into "kick draw flax" is:

q	u	i	c	k		b	r	o	w	n		f	o	ε	x
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
k	ε	i	c	k		d	r	a	w	ε		f	l	a	x

The special symbol 'ε' is used to represent the absence of a character. A transformation from a character into ε is considered a *deletion* (e.g., u → ε), from ε into a character an *insertion* (e.g., ε → a), and from one character into another, different character a *substitution* (e.g., q → k).

As a rule, the arrows in an alignment are not allowed to cross.[3] Moreover, the character-to-character correspondence is determined on an individual basis, with no regard to higher-level structure. Consider now an alignment comparing the strings "hello world" and "world hello":

h	e	l	l	o		w	o	r	l	d	ε	ε	ε	ε	ε	ε
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
ε	ε	ε	ε	ε	ε	w	o	r	l	d		h	e	l	l	o

The "cost" of this alignment is six deletions and six insertions. By overlooking the higher-level structure, the motion of the word "hello" from the beginning of the string to the end, traditional edit distance (e.g., [26]) produces an alignment that seems to miss the true relationship between the two strings. There is no obvious way of taking the result returned by simple edit distance and using it to generate a more representative block matching.

Fig. 3 presents a *block alignment* relating the strings "The quick brown fox jumps over the lazy dog." and "Jump over the brown fox, lazy dog. Quick!" This captures both the low-level notion of approximate string matching (e.g., the close similarity between the blocks "jumps over the" and "Jump over the"), as well as the higher-level concept of block motion. We seek algorithms capable of producing alignments such as this.

We now give a more formal definition of a string block edit model.

[3] This is dictated by the model and enforced by the dynamic programming algorithm used to perform the computation.

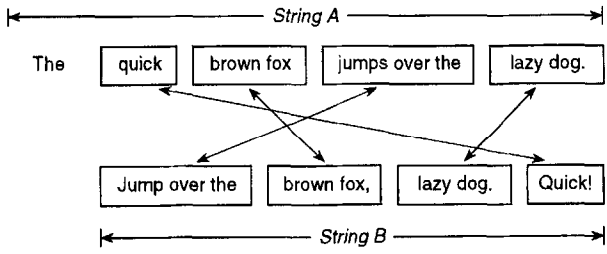

Fig. 3. Example of a block alignment.

2.1. Substring families

Assume we have a finite alphabet Σ. Let $A=[a_1a_2\ldots a_m]$ and $B=[b_1b_2\ldots b_n]$ be strings over the alphabet, $a_i, b_j \in \Sigma$. We say that a *t-block substring family of A*, $A|_t$, is a multiset containing t substrings of A, some of which may be identical. In the following, we will write $A|_t=\{A^{(1)},\ldots,A^{(t)}\}$, with the understanding that the $A^{(i)}$'s need not be distinct. A corresponding t-block substring family of B, $B|_t$, is a multiset of t substrings of B.

If the substrings in $A|_t$ do not overlap, we say the family is *disjoint*. If each character of A is contained in some substring, we say the family represents a *cover* of A. Thus, Fig. 1 shows a mapping between substring families such that $A|_5$, on the left, is a disjoint cover, and $B|_5$, on the right, is neither disjoint nor a cover. Fig. 2 illustrates that substantial overlap can occur between candidate substrings of genetic sequences, hence there is an argument for preferring substring families that are not necessarily disjoint in this case. Finally, in Fig. 3 the lower substring family is a disjoint cover, while the upper is disjoint but not a cover.

In general, we may require that either or both of the substring families be disjoint and/or a cover. Each possible combination of constraints represents a particular block edit model. For succinctness, we introduce the following notation:

C must be a cover,
$\overline{\text{C}}$ need not be a cover,
D must be disjoint,
$\overline{\text{D}}$ need not be disjoint.

To refer to the model in which the first substring family must be a disjoint cover, and the second substring family is unconstrained, we write CD-$\overline{\text{CD}}$. (Note: from a computational standpoint, by symmetry $\overline{\text{CD}}$-CD is exactly the same problem.)

2.2. Block edit distance

Before defining block edit distance, we require an underlying function *dist* that returns the cost of corresponding a substring of A with a substring of B:

$$dist : \{i,j \mid 1 \leqslant i \leqslant j \leqslant |A|\} \times \{k,l \mid 1 \leqslant k \leqslant l \leqslant |B|\} \rightarrow \Re$$

Table 1
Summary of the results presented in this paper

	$\overline{\mathrm{CD}}$	$\overline{\mathrm{C}}\mathrm{D}$	$\mathrm{C}\overline{\mathrm{D}}$	CD
$\overline{\mathrm{CD}}$	$O(m^2n^2)$ Section 5			
$\overline{\mathrm{C}}\mathrm{D}$	$O(m^2n)$ Section 5	NP-complete Section 4		
$\mathrm{C}\overline{\mathrm{D}}$	$O(m^2n^2)$ Section 5	NP-complete Section 4	NP-complete Section 4	
CD	$O(m^2n)$ Section 5	NP-complete Section 4	NP-complete Section 4	NP-complete Section 3

In practice, it is natural to assume that *dist* is traditional string edit distance, but any cost function could be used. The algorithms we give work for arbitrary measures, and the reductions work for bi-valued measures as well as for string edit distance, so the generality of the cost function does not affect the difficulty of the problem.[4]

The *block edit distance* $\mathscr{B}$ between two strings A and B is determined by finding the best way to choose t-block substring families of A and B and correspond each member of $A|_t$ with some member of $B|_t$. For each pairing, a cost is assessed based on the distance between the two substrings plus a constant per-block cost, c_{block}. The correspondence between blocks is given by a permutation $\sigma \in S_t$ from the symmetric group on t elements. More formally,

$$\mathscr{B}(A,B) \equiv \min_{t} \min_{A|_t, B|_t} \min_{\sigma \in S(t)} \left\{ t \cdot c_{\text{block}} + \sum_{i=1}^{t} dist(A^{(i)}, B^{(\sigma(i))}) \right\} \tag{1}$$

Eq. (1) does not specify whether the particular substring families must be covers, disjoint, or both. In this paper, we examine the various cases, show which are hard, and give algorithms for those that are solvable in polynomial time. Table 1 summarizes our results.

Before proceeding, however, we take the opportunity to clarify an important point. In our analyses, we impose the restriction that if $i \neq j$ and $A^{(i)} = A^{(j)}$, then $B^{(\sigma(i))} \neq B^{(\sigma(j))}$. That is, a particular pair of blocks cannot be placed into correspondence more than once. This allows us to keep the measure from diverging if a negative-cost pairing exists and the substring families do not have to be disjoint. In the event of negative-cost pairings, it may be helpful to think of the dual problem, maximizing similarity, as opposed to minimizing distance.

[4] As per common usage, we refer to *dist* as a "distance" when in fact it is more general than this: it need not be symmetric, can take on negative values, and does not have to obey the triangle inequality. A *bi-valued* measure is one that takes on only two values; for instance, just 0 and 1.

3. CD-CD block edit distance is NP-complete

In this section we show that if both substring families must be disjoint covers, the block edit distance problem is NP-complete. In Section 4, we extend the same reduction to the other hard versions of the problem.

Theorem 1. *CD-CD block edit distance is* NP*-complete.*

Proof. Membership in NP is trivial. We must show that the problem is NP-hard.

The reduction is from uniprocessor scheduling. From Garey and Johnson [5]:

Sequencing with release times and deadlines

Instance: Set T of jobs and, for each $\text{JOB}_j \in T$, a length $l(j) \in Z^+$, a release time $r(j) \in Z_0^+$, and a deadline $d(j) \in Z^+$.

Question: Is there a one-processor schedule for T that schedules no job before its release time and completes each job by its deadline?

We take the string alphabet to be $\Sigma = \{0, 1\}$. Assume that the number of jobs in the scheduling problem is $N = |T|$. For $n \in \{1, \ldots, N\}$ we define the string $\#(j)$ to be

$$\#(j) = \overbrace{0\ldots0}^{N-j}\overbrace{1\ldots1}^{j}\overbrace{0\ldots0}^{N}$$

Thus, for all j, $|\#(j)| = 2N$, and $\#(0) = 0^{2N}$.

We must now specify two strings and a cost function as input to the block edit distance algorithm. String A will represent time, and string B will represent the jobs. Let D be the latest deadline, $D = \max_j\{d(j)\}$. Strings A and B will have length $4N^2D$. Note that since the scheduling problem is NP-hard in the strong sense, we can assume that the size of the input is $O(D)$, so these strings are polynomial-sized.

We assume without loss of generality that $\sum_j l(j) = D$. That is, if all jobs are scheduled in time, then all units of time through the final deadline will be used. If this is not the case, we can add to the list of jobs $D - \sum_j l(j)$ additional jobs with length 1, release time 0, and deadline D to meet the constraint without changing the problem. Fig. 4 depicts the two strings.

Each of the time-step blocks in string A is a filled-in copy of the template shown in Fig. 5. We need some new notation for referencing these substrings. We will write $A[\text{TIME}_i]$ to refer to the ith time-step of A, and $A[\text{TIME}_i, \text{CHUNK}_j]$ to refer to the jth

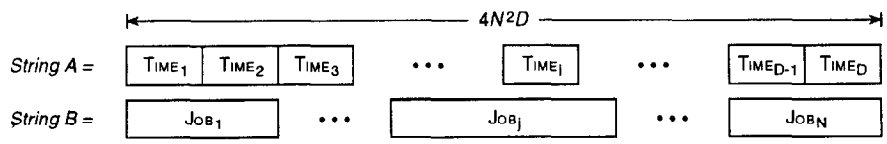

Fig. 4. Strings A and B for the NP-completeness reduction.

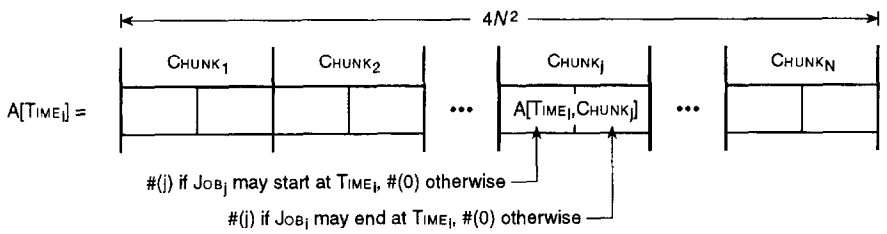

Fig. 5. Template of a time-step block.

"chunk" of $4N$ characters in substring $A[\text{TIME}_i]$. As the figure shows, $A[\text{TIME}_i, \text{CHUNK}_j]$ is made up of two components, each of length $2N$. The first is $\#(j)$ if JOB_j may start at time-step i (i.e., if and only if $r(j) \leqslant i$), and $\#(0)$ otherwise. Similarly, the second component is $\#(j)$ if JOB_j may end at time-step i (i.e., if and only if $d(j) \geqslant i$), and $\#(0)$ otherwise. Each time-step block represents a string of $4N^2$ characters.

At this point, we have completely specified string A. We now turn to the structure of string B by specifying the job blocks in Fig. 4. Following the notation introduced previously, we will write $B[\text{JOB}_j]$ to refer to the entire block JOB_j. Each such block is a string consisting of $l(j) \cdot 4N^2$ characters.

Within $B[\text{JOB}_j]$, each of the $l(j)$ substrings of $4N^2$ characters corresponds to a time-step, so we will write $B[\text{JOB}_j, \text{TIME}_i]$ to refer to the ith group of $4N^2$ characters within $B[\text{JOB}_j]$. Finally, these $4N^2$ characters are broken into N "chunks" of $4N$ characters, each of which corresponds to a particular task. We will refer to the kth chunk within time-step i in job j as $B[\text{JOB}_j, \text{TIME}_i, \text{CHUNK}_k]$. Within $B[\text{JOB}_j, \text{TIME}_i]$, all chunks except those numbered j will consist of $4N$ 0's.

We now give the procedure for assigning substrings to each of the chunks. As in the construction of string A, the first and second groups of $2N$ characters are used to hold information about starting and ending a job, respectively:

$$\text{start}(j,i) = \begin{cases} \#(j) & \text{if } i = 1 \\ \#(0) & \text{otherwise} \end{cases} \tag{2}$$

$$\text{end}(j,i) = \begin{cases} \#(j) & \text{if } i = l(j) \\ \#(0) & \text{otherwise} \end{cases} \tag{3}$$

$$B[\text{JOB}_j, \text{TIME}_i, \text{CHUNK}_k] = \begin{cases} \#(0) \,||\, \#(0) & \text{if } j \neq k \\ \text{start}(j,i) \,||\, \text{end}(j,i) & \text{otherwise} \end{cases} \tag{4}$$

$B[\text{JOB}_j, \text{TIME}_1, \text{CHUNK}_j]$ has the effect of constraining job j to begin at or after its release time, while $B[\text{JOB}_j, \text{TIME}_{l(j)}, \text{CHUNK}_j]$ constrains it to end at or before its deadline. This is depicted in Fig. 6.

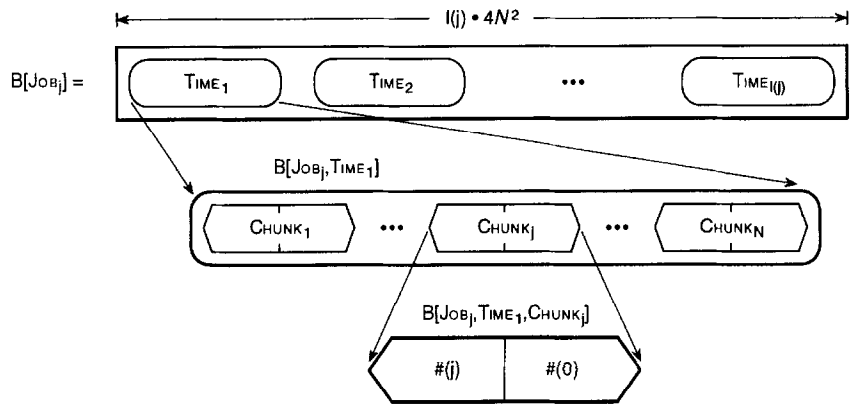

Fig. 6. Template of a job block.

This completes the specification of both strings. All that remains is to give the cost function, *dist*. This function returns 1 for all pairs of substrings with the following exception:

$dist(s_1, s_2) = 0$ if and only if there exist indices i_1, i_2, and j such that the following conditions hold :

1. $s_1 = A[\text{TIME}_{i_1}] \cdots A[\text{TIME}_{i_2}]$
2. $s_2 = B[\text{JOB}_j]$
3. $i_2 - i_1 + 1 = l(j)$
4. $r(j) \leqslant i_1$ and $d(j) \geqslant i_2$

For the purposes of the reduction, we set $c_{\text{block}} = 0$ in Eq. (1).

This particular cost function is formulated so that a zero-cost matching, if one exists, yields a solution to the uniprocessor scheduling problem. The function checks to see whether the two strings passed to it correspond to a block of time and a job, respectively. One might worry that s_1 and s_2 could be identical to $A[\text{TIME}_{i_1}] \cdots A[\text{TIME}_{i_2}]$ and $B[\text{JOB}_j]$, but come from entirely different parts of A and B, or that their sources could violate the time-step and job boundaries. We now show that this cannot happen: the distance function returns 0 for s_1 and s_2 if and only if they are actually drawn from the appropriate parts of A and B.

Lemma 1. *The substring #(j), $j > 0$, occurs in strings A and B only at 2N block boundaries.*

Proof. Break both strings into blocks of length $2N$. By their construction, each block consists of #(i) for $i \geqslant 0$. Any substring t of length $2N$ will overlap at most two such blocks, say s_1 and s_2. Create $s = s_1 \,||\, s_2$. For t to equal #(j) for some $j > 0$, there must exist a 1–0 transition in s that corresponds to positions N and $N + 1$ in t. But such transitions occur in at most two places in s: at positions N and $N + 1$ (i.e., s_1),

and at positions $3N$ and $3N+1$ (i.e., s_2). That is, t can equal $\#(j)$ if and only if $t = s_1$ or $t = s_2$.

Thus, the substring $\#(j)$ occurs only at $2N$ block boundaries for $j > 0$. This completes the proof of Lemma 1. □

At this point we require some additional notation. Consider two substrings, s_1 drawn from A and s_2 drawn from B. Let s_i^0 be the first $4N^2$ characters of s_i, and s_i^1 be the last $4N^2$ characters of s_i. Our goal is to show that the distance function defined earlier will never return 0 for substrings that are not taken from appropriate locations in A and B. We do so by presenting two sets of definitions and accompanying lemmas. The first defines a syntactic property between two strings and shows that no other substrings can fulfill the zero-cost conditions of *dist*. The second makes precise the notion of "appropriate location" and relates it to the syntactic property.

Definition 1. Substrings s_1 and s_2 have the *match property* if the following conditions hold:

1. They both have length $l(j) \cdot 4N^2$ for some $1 \leqslant j \leqslant N$.
2. The $(2j-1)$th block of $2N$ characters in s_1^0 and s_2^0 are both $\#(j)$.
3. The $2j$th block of $2N$ characters in s_1^1 and s_2^1 are both $\#(j)$.

We now show that any two substrings with distance 0 must have the match property.

Lemma 2. *If* $dist(s_1, s_2) = 0$, *then substrings* s_1 *and* s_2 *have the match property.*

Proof. By the definition of *dist* and the construction of strings A and B, the length condition for the match property is clearly satisfied.

We now examine the $(2j-1)$th block of $2N$ characters in s_1^0 and s_2^0. For the time-step substring, s_1, this will be the first $2N$ characters of $A[\text{TIME}_{i_1}, \text{CHUNK}_j]$, and assuming that JOB_j may start at time-step i_1 (we know that $r(j) \leqslant i_1$ from the definition of *dist*), this will be $\#(j)$. Likewise, for the job substring, s_2, this will be the first $2N$ characters of $B[\text{JOB}_j, \text{TIME}_1, \text{CHUNK}_j]$, which by definition is also $\#(j)$.

Next, we examine the $2j$th block of $2N$ characters in s_1^1 and s_2^1. For the time-step substring, this corresponds to the last $2N$ characters of $A[\text{TIME}_{i_2}, \text{CHUNK}_j]$, which, assuming JOB_j can terminate at time-step i_2 (again, this is true from the definition of *dist*), is $\#(j)$. For the job substring, this block corresponds to the last $2N$ characters of $B[\text{JOB}_j, \text{TIME}_{l(j)}, \text{CHUNK}_j]$, which is also $\#(j)$.

This completes the proof of Lemma 2. □

Finally, we must guarantee that no spurious matches can occur.

Definition 2. A set of indices i_1, i_2, and j, and the substrings they induce,

$$s_1 = A[\text{TIME}_{i_1}] \cdots A[\text{TIME}_{i_2}]$$

and

$$s_2 = B[\text{JOB}_j],$$

are *valid* if $i_2 - i_1 + 1 = l(j)$, $r(j) \leqslant i_1$, and $d(j) \geqslant i_2$.

Note that s_1 and s_2 are not considered valid if either is taken from a different position in A or B, even if the resulting substrings are identical. Validity is a property of the indices into A and B.

Lemma 3. *If substrings s_1 and s_2 have the match property, then they are valid.*

Proof. We must show that substrings of A and B will match only if they represent a particular job and a feasible time-slot for the job. The match property requires that the string $\#(j)$ appear four times between the two substrings, for some j in the range $[1,N]$. By Lemma 1, we can conclude that any erroneous matches could come about only as a result of $\#(j)$'s placed during the construction of A and B, and not from "random" patterns appearing in the strings by coincidence.

Thus, s_1 and s_2 must begin and end on $2N$ block boundaries within A and B, respectively. Further, since s_1 and s_2 have the match property, the string $\#(j)$ must appear as the $(2j-1)$th block of $2N$ characters at the beginning of s_1, and as the $2j$th block of $2N$ characters in the last $4N^2$ characters. This forces substring s_1 to be aligned on a $4N^2$ boundary, so it must indeed represent a legal sequence of time-steps. In this case, the details of the construction of A guarantee that job j can be scheduled during this time-slot and meet its release and deadline constraints.

The proof for substring s_2 is immediate, since $\#(j)$ must appear exactly twice, at specific locations, by the match property. By the construction of B, this can only occur if s_2 represents job j. This completes the proof of Lemma 3. □

We can now prove the primary lemma that leads directly to our theorem.

Lemma 4. *The uniprocessor scheduling problem has a solution if and only if the corresponding string block edit problem has a matching that is a zero-cost disjoint cover.*

Proof. If the scheduling problem is solvable, then by the definition of *dist* this will yield a zero-cost block matching. That the matching must be disjoint is clear (otherwise two jobs will have been scheduled for the same time-step). The fact that it is a cover follows from our earlier assumption that the total duration of the jobs consumes all time-steps up to the latest deadline.

Assume now that a matching exists that is a zero-cost disjoint cover. Since the cost function returns only 0 or 1, by Eq. (1) the cost for each pair of blocks must be 0. Hence, by Lemma 2, all of the pairings have the match property. Applying Lemma 3, this means they correspond to valid substrings and therefore represent an assignment

of jobs to time-slots that satisfies the constraints of the scheduling problem. By the construction of string B, all of the jobs are scheduled. This completes the proof of the lemma. □

With Lemma 4, we have completed the proof of Theorem 1, showing that CD-CD block edit distance is NP-complete. □

4. NP-completeness of other models

In this section, we show that essentially the same reduction works for the other hard models listed in Table 1.

Theorem 2. *The CD-$\overline{\text{CD}}$, CD-$\overline{\text{CD}}$, and $\overline{\text{CD}}$-$\overline{\text{CD}}$ block edit distance problems are* NP-*complete.*

Proof. As before, membership in NP is obvious, so we need only demonstrate how the reduction can be applied to these models.

Theorem 1 states that the problem is hard if both substring families must be disjoint covers. The same proof can be used if one substring family need not be a cover. Recall that string A represents time-steps. Clearly a block matching that does not use all of the available time, but still schedules all of the jobs in a valid way, is just as difficult to achieve. This shows that CD-$\overline{\text{CD}}$ is NP-complete.

Likewise, the problem remains difficult if one substring family need not be disjoint. For this variant we use the same reduction, but do not require the substring family chosen from B to be disjoint. Thus, all jobs must be matched (i.e., B must still be a cover) to distinct units of time (i.e., A must be disjoint), but jobs can also be re-used to help cover all of the time-steps. Again, the original reduction need not be changed. This shows that CD-C$\overline{\text{D}}$ is NP-complete.

Combining these two observations, if the time string need not be covered, and the job string need not be disjoint, the resulting schedule will still be valid, so the reduction holds. This shows that $\overline{\text{CD}}$-$\overline{\text{CD}}$ is NP-complete, completing the proof of the theorem. □

To finish the last two hard entries in Table 1, we must make minor changes to the cost function.

Theorem 3. *The $\overline{\text{CD}}$-$\overline{\text{CD}}$ and $\overline{\text{CD}}$-$\overline{\text{CD}}$ block edit distance problems are* NP-*complete.*

Proof. For the $\overline{\text{CD}}$-$\overline{\text{CD}}$ model, we can adapt the reduction fairly simply. In this case, neither string must be disjoint, so time-steps and jobs can be used more than once. We change the distance measure so that a valid match between JOB_j and a particular sequence of time-steps has cost 1, and all other substring pairings have cost ∞. Then if

a schedule exists, a string matching can be constructed with total distance N, otherwise no such match can be found.

The proof for the $\overline{\text{CD}}$-$\overline{\text{CD}}$ model is similar. If neither string must be covered, the problem makes sense only if negative distances are allowed (otherwise the best match would always return empty substring families for both strings). We modify the distance measure so that a valid match has cost -1. Since both substring families must be disjoint, no time-step or job can be re-used. In other words, each JOB_j can be matched at most once, so the minimal attainable distance is $-N$. Hence, if a matching with distance $-N$ can be found, it must correspond to a schedule. If there is no such match, then no schedule exists. This completes the proof of the theorem. □

We have shown that, for certain models, block edit distance is hard to compute. Our proofs relied on a carefully chosen underlying distance measure. It seems likely, however, that block editing would be most often used in conjunction with standard string edit distance. One might hope that, while the problem is hard as formulated above, it becomes tractable when restricted in this way. Unfortunately, this is not the case:

Theorem 4. *The CD-CD block edit distance problem is* NP*-complete when* dist *is standard string edit distance.*

Proof. See Appendix A. □

We describe extensions for the other hard versions of the problem in the appendix as well.

5. Polynomial-time algorithms for block editing

We now present a family of polynomial-time algorithms to compute block edit distance when at least one of the substring families is unconstrained.

Say that B is the string whose substring family need not be disjoint or a cover. For the discussion that follows, it will be convenient to assume we have an array W^1 defined as below for $1 \leqslant i \leqslant j \leqslant m$:

$$W^1(i,j) \equiv c_{\text{block}} + \min_{k \leqslant l} \{dist(a_i \ldots a_j, b_k \ldots b_l)\}. \tag{5}$$

That is, $W^1(i,j)$ gives the value of the best possible match between $a_i \ldots a_j$ and any substring of B, plus the per-block cost c_{block}. Since portions of B can be re-used, and it need not be covered, the information in W^1 is sufficient to perform the needed calculations for the CD-$\overline{\text{CD}}$ problem; we will define similar matrices W for the other problems in their respective subsections. We write $T(W)$ to mean the time required to compute a matrix W, and shall discuss later how W can be computed more efficiently than the naive implementation when *dist* is standard edit distance.

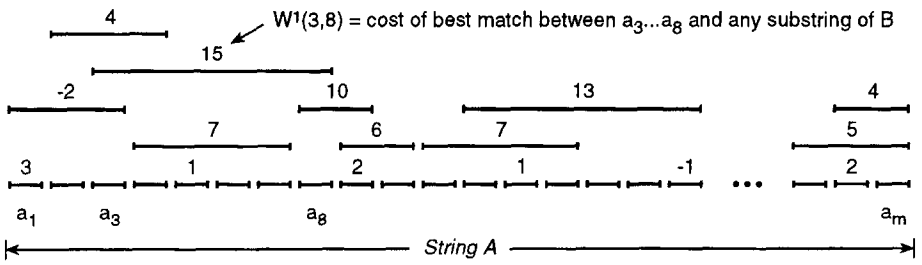

Fig. 7. Possible string matches viewed as intervals.

Consider the diagram shown in Fig. 7. Each of the intervals $a_i \ldots a_j$ in the figure represents a substring of A, and is labelled with $W^1(i,j)$. Note that $W^1(i,i)$ represents the best match between the single character a_i and any interval of B. As before, a substring family of A is a multiset of substrings (i.e., intervals). If the intervals do not overlap, the family is disjoint; if the union of the intervals is the entire line, the family is a cover. Enforcing or relaxing these constraints (all relative to string A) results in different versions of the block edit distance problem.

It is clear from Fig. 7 that W^1 induces a complete interval graph, a well-studied class for which most known problems have efficient solutions [7, 19]. We now present a series of dynamic programming recurrences for the variants of block edit distance that admit poly-time solutions, based on choosing intervals in a way that satisfies certain constraints.

We define $\mathscr{M}(i)$ to be the best block match between $a_1 \ldots a_i$ and B for the particular model we are interested in. Once we have computed $\mathscr{M}$ for $i = 1, 2, \ldots, m$ (recall that $|A| = m$), our final answer is $\mathscr{B}(A,B) = \mathscr{M}(m)$.

5.1. CD-$\overline{\text{CD}}$ block edit distance

We begin with the CD-$\overline{\text{CD}}$ block edit distance problem, in which the substring family of A must be both disjoint and a cover. We can compute $\mathscr{M}$ using the following recurrence:

Algorithm CD-$\overline{\text{CD}}$: $$\mathscr{M}(i) = \min_{j<i}\{\mathscr{M}(j) + W^1(j+1,i)\} \tag{6}$$

In this recurrence, $\mathscr{M}(i)$ allows the best match in B corresponding to $a_{j+1} \ldots a_i$ to be added to the optimal solution for $a_1 \ldots a_j$ for all possible "cuts" in the string, j. It is easy to show this satisfies the requirement that the substring family for A be a disjoint cover. Since each addition of an element from W^1 corresponds to a new block pairing, the $t \cdot c_{\text{block}}$ term in the definition of block edit distance (Eq. (1)) is incorporated in the recurrence. By dynamic programming, the value of $\mathscr{M}(i)$ can be computed in time $O(i)$ given all previous $\mathscr{M}(j < i)$. Thus, the total time to compute $\mathscr{M}(m)$ is $O(m^2) + T(W^1)$.

5.2. $\overline{CD}$-$\overline{CD}$ block edit distance

We now address the problem where the substring family must be disjoint, but need not cover A. For this case we define another W matrix:

$$W^{0,1}(i,j) \equiv \min\{W^1(i,j), 0\}. \tag{7}$$

Used in place of W^1 in Eq. (6), $W^{0,1}$ allows sections of A to be "skipped" whenever it is advantageous to do so. The recurrence is:

Algorithm $\overline{\text{CD}}$-$\overline{\text{CD}}$: $$\mathcal{M}(i) = \min_{j<i}\{\mathcal{M}(j) + W^{0,1}(j+1,i)\} \tag{8}$$

The time bound is exactly the same as for Eq. (6), namely $O(m^2) + T(W^1)$.

5.3. $\overline{CD}$-$\overline{CD}$ block edit distance

Next, we consider the variant in which the substring family of A must be a cover, but need not be disjoint. Recall that block edit distance as defined in Eq. (1) does not allow the same block pairing to be used more than once. Here we see why this should be so; otherwise the block edit distance between two strings A and B could be $-\infty$ (if a negative-cost pairing exists). We require a version of W that allows the substring in A to match one or more intervals in B:

$$W^+(i,j) \equiv \begin{cases} W^1(i,j) & \text{if } W^1(i,j) > 0, \\ \sum\limits_{k \leqslant l} \min\{c_{\text{block}} + dist(a_i \ldots a_j, b_k \ldots b_l), 0\} & \text{otherwise.} \end{cases} \tag{9}$$

Similarly, we define W^* to represent the cost of zero or more matches:

$$W^*(i,j) \equiv \min\{W^+(i,j), 0\} \tag{10}$$

We can now use the following recurrence to allow overlapping intervals:

Algorithm $\overline{\text{CD}}$-$\overline{\text{CD}}$: $$\mathcal{M}(i) = \min_{j<i}\left\{\mathcal{M}(j) + \min_{k\in[0,j+1]} W^+(k,i) + \sum_{\substack{a\in[0,i] \\ b\in[j+1,i]}} W^*(a,b)\right\} \tag{11}$$

Intuitively, the recurrence can be understood as follows: we must cover the string A through a_i. We choose optimally some intermediate point a_j and cover through it (the $\mathcal{M}(j)$ term). Then we cover the region between $j+1$ and i, possibly overlapping some previous blocks (the W^+ term). Finally, we may add additional matchings that end between $j+1$ and i if they lower the overall cost (the W^* term).

Note that this can be computed in $O(m^2) + T(W^+)$ time, despite the additional minimization and summation. First, we build a table $T_1(b) = \sum_{0\leqslant a\leqslant b} W^*(a,b)$. This takes time $O(m^2)$, as each element of W^* is added to an element of T_1 exactly

once, and there are $O(m^2)$ such elements. We then create another table, T_2, containing $\min_{k\in[0,j+1]} W^+(k,i)$ for all $j \in [0, i-1]$. This can be done in time $O(m)$, since given the value $T_2(j)$, the value $T_2(j+1)$ can be determined in constant time. Finally, we construct a third table, T_3, for the sum $\sum W^*(a,b)$ over all values of j beginning at $i-1$ and working backwards to 0, adding subsequent elements of T_1 at each step. Using T_2 and T_3, $\mathcal{M}(i)$ can be computed in $O(m)$ time for each i. Hence, the running time is $O(m^2) + T(W^+)$.

5.4. $\overline{CD}$-$\overline{CD}$ block edit distance

Finally, we give a recurrence to solve the problem when neither substring family is constrained. In this case, every negative-cost pairing between blocks of A and B is added to the matching; these are exactly the non-zero elements of W^*, so the equation is:

$$\textbf{Algorithm } \overline{\text{CD}}\text{-}\overline{\text{CD}}\text{:} \qquad \mathcal{M}(i) = \sum_{j\leqslant i} W^*(j,i) \tag{12}$$

Again, the recurrence can be evaluated in time $O(m^2) + T(W^+)$.

5.5. Time complexity

As we indicated, each of the recurrences requires $O(m^2)$ time, where $m = |A|$. However, they all depend on having a matrix W, so the full time bound is $O(m^2) + T(W)$. If we build W^1 according to its definition (i.e., Eq. (5)), for example, we must fill in $O(m^2)$ entries by comparing $O(n^2)$ values, each of which can take time $O(mn)$ to compute when *dist* is standard string edit distance. Thus, naively, $T(W^1) = O(m^3n^3)$.

There is, however, a well-known modification of the basic dynamic programming algorithm that allows the best match in B for a fixed substring in A to be found in time $O(mn)$. Let $d_{i,j} = dist(a_1 \ldots a_i, b_1 \ldots b_j)$ be standard string edit distance [18, 26]. The initial conditions are

$$\begin{aligned} &d_{0,0} = 0, \\ &d_{i,0} = d_{i-1,0} + c_{\text{del}}(a_i), \quad 1 \leqslant i \leqslant m; \\ &d_{0,j} = d_{0,j-1} + c_{\text{ins}}(b_j), \quad 1 \leqslant j \leqslant n, \end{aligned} \tag{13}$$

and the main dynamic programming recurrence is

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + c_{\text{del}}(a_i), \\ d_{i,j-1} + c_{\text{ins}}(b_j), \\ d_{i-1,j-1} + c_{\text{sub}}(a_i, b_j), \end{cases} \qquad 1 \leqslant i \leqslant m, \ 1 \leqslant j \leqslant n \tag{14}$$

The time required to evaluate the recurrence is $O(mn)$.

For the modification, the initial edit distance along the entire length of B is made 0 (allowing the match to start anywhere), and the final row of the table is searched for its smallest value (allowing the match to end anywhere). The initial conditions become:

$$\begin{aligned} d_{0,0} &= 0, \\ d_{i,0} &= d_{i-1,0} + c_{\text{del}}(a_i), \quad 1 \leqslant i \leqslant m, \\ d_{0,j} &= 0. \end{aligned} \tag{15}$$

The inner-loop recurrence (i.e., Eq. (14)) remains the same. Using this formulation, we obviate the need to try all possible starting and ending positions for the block in B, saving a factor of $O(n^2)$ over the naive approach.

Furthermore, a property of this computation is that the table generated for matching $a_i \ldots a_n$ to B contains information about the best substring matches for $a_i \ldots a_k$ for $i \leqslant k < n$ as well, since these correspond to intermediate rows. Hence, only $O(m)$ such tables need be built to compute W^1, saving another $O(m)$. The c_{block} term can be added to all the entries in the final table in $O(m^2)$ time once these values have been determined. Thus, $T(W^1) = O(m^2 n)$.

The case for W^+ is only somewhat more complicated. The construction above allows us to find the best match in B for a fixed substring in A; to compute W^+, however, we require not only this but also the sum of all other negative-cost matchings. To do this, we apply the same "trick" as before with B in place of A: instead of building a table comparing B to a suffix of A, we build a series of n tables comparing each suffix of B to the suffix of A. This allows us to look up the cost of the matching between the substrings $a_i \ldots a_j$ and $b_k \ldots b_l$ by locating the table comparing $a_i \ldots a_m$ to $b_k \ldots b_n$ and reading the $(j - i, l - k)$th entry. The time required to construct the tables (and hence W^+) is $O(m^2 n^2)$.

Finally, we note that it is possible to extend the computations in other interesting ways by changing the definitions of the various W matrices appropriately. For instance, since each element in W^1 represents a specific block pairing, we could incorporate an additional cost that depends on the distance between the two blocks in their respective strings (favoring pairings that come from the same general locations in A and B, for example).

6. Conclusions

In this paper we have examined the concept of string block edit distance, in which two strings A and B are compared by extracting collections of substrings and placing them into correspondence. This model seems to account for certain phenomena encountered in important real-world applications, including pen computing and molecular biology. Experimentally, we have confirmed that such a framework does indeed facilitate the matching of hand-drawn sketches – these results are described elsewhere [16, 17].

As we demonstrated, the basic problem admits a family of variations depending on whether the strings must be matched in their entireties, and whether overlap is

permitted. The problem is NP-complete if both substring families are constrained in any way, and solvable in time $O(m^2) + T(W)$ otherwise. We gave algorithms for computing W in $O(m^2n)$ or $O(m^2n^2)$ time according to the specific variant – it would be interesting to know whether these results can be improved.

Another open question concerns the existence of approximation algorithms for the more difficult versions of the problem (especially if such algorithms also overcome the bottleneck of having to compute W exactly). While the recurrences presented in Section 5 for the poly-time cases are not guaranteed to return a disjoint cover for string B, this does not exclude the possibility under some scenarios. It may be instructive to attempt to characterize just when these additional constraints can be satisfied.

Finally, although we have given polynomial algorithms for some of the problems, the order of growth is sufficiently large that pruning of the search space is required for all but the smallest instances. In many cases, it should be possible to incorporate application-specific knowledge to improve run-time performance.

Acknowledgements

The genetic sequence dot matrix plot (Fig. 2) was generated using "Dotty Plotter," a program written by Don Gilbert. The authors would like to thank Professors Ricardo Baeza-Yates and Udi Manber as well as the anonymous referees for their many helpful comments on an earlier draft of this paper.

Appendix. Block editing under standard string edit distance

The algorithms presented in Section 5 will work regardless of the choice of cost functions. We need to show, however, that the same hardness results hold when the underlying distance *dist* is restricted to be standard string edit distance. We begin by restating the theorem:

Theorem 4. *The CD-CD block edit distance problem is* NP*-complete when* dist *is standard string edit distance.*

Proof. We extend the reduction given in Section 3 by introducing a new character ω into the alphabet. The original proof converted an instance of uniprocessor scheduling into an instance of block edit distance with two strings $A, B \in \Sigma^*$. We will post-process these strings to yield two new strings $A' \in \Sigma^*$ and $B' \in (\Sigma \cup \{\omega\})^*$ such that the distance function on underlying blocks can be replaced by standard string edit distance.

In the original reduction, the nature of the distance function required that blocks be of a particular size, so the per-block cost c_{block} was not important (and hence set equal to 0). Here, however, we assume that we may choose a particular positive value for c_{block}. This assumption is a reasonable one, because when $c_{\text{block}} = 0$ the problem

becomes trivial,[5] and when $c_{\text{block}} = \infty$ it becomes standard string edit distance. For the single-character editing operations, we assume that deletions, insertions, and substitutions all have cost c_{char}, while the cost of a perfect match (substituting a character for itself) is 0.

Assume we are given an instance of the uniprocessor scheduling problem. As in Section 3, we convert this into a block matching problem. We then use a "trick" to guarantee that the algorithm cannot leave two adjacent jobs (i.e., substrings of B) together as a single block, thereby saving c_{block} in the total cost.

First, we slightly more than double the number of jobs so that our new string B' consists of $2n+1$ jobs, the even-numbered ones corresponding to the original jobs and the odd-numbered ones having the constraint that they must be scheduled in order, before any even-numbered job, for their deadlines to be met. Each odd-numbered job takes one time unit, and the ith odd-numbered job must be finished by time i. The deadlines of all the original jobs are delayed by $n+1$ time units.

Thus, B' starts with the form $B'[\text{JOB}_1]B'[\text{JOB}_2]\ldots B'[\text{JOB}_{2n+1}]$, and any valid schedule must rearrange B' into the form $B'[\text{JOB}_1]\ B'[\text{JOB}_3]\ldots B'[\text{JOB}_{2n+1}]\ B'[\text{JOB}_{2\sigma(1)}]\ B'[\text{JOB}_{2\sigma(2)}]$ $\ldots\ B'[\text{JOB}_{2\sigma(n)}]$ for some permutation, σ. We assert that it is impossible to find a satisfying schedule that saves c_{block} by keeping two adjacent jobs together. Let i be even. If jobs i and $i+1$ are adjacent in the final schedule, then an even-numbered job is scheduled before an odd-numbered job. If jobs $i-1$ and i are adjacent, then either i is scheduled before some other odd-numbered job, or the last odd-numbered job is numbered less than $2n+1$.

We now consider the structure of the strings generated in the original reduction. As Fig. 6 illustrates, the only non-zero elements in string B occur at specific locations; namely, the first half of $B[\text{JOB}_j, \text{TIME}_1, \text{CHUNK}_j]$ and the last half of $B[\text{JOB}_j, \text{TIME}_{l(j)}, \text{CHUNK}_j]$. Hence, for each job in string B there are only two non-zero half-chunks (where a half-chunk is a string of length $2N$), so B contains $2(2n+1)$ non-zero half-chunks and all the rest are zero-filled. For B', we replace every zero in the zero-filled half-chunks with ω.

String A' is identical to string A from the original reduction. Note that ω does not appear anywhere in A'.

We now replace the original distance function with string edit distance and demonstrate that the reduction still works. If a schedule exists satisfying the constraints, it will be possible to place every non-ω character in B' into correspondence with the same character in A'. Since ω is equally distant from all characters in A', it does not matter which ones it is mapped to. Thus, if an optimal schedule exists, the block edit distance will be c_{char} times the number of ω's in B' (note that any block matching must pay at least this amount because each ω must either be deleted or aligned with *some* character in A') plus c_{block} times the number of blocks.

[5] In the absence of a per-block penalty, when the underlying distance measure is string edit distance, the algorithm could simply make each individual character a "block" and place all matching characters in both strings into correspondence.

We choose c_{block} to be smaller than $c_{\text{char}}/(2n+1)$. Assume an optimal schedule exists. Then any matching that does not break B' into $2n+1$ or more blocks must place at least one pair of non-distinct non-ω characters into correspondence (by the analysis of Section 3), or must delete some ω and therefore perform at least one compensating insertion. It must therefore pay at least c_{char} plus c_{char} times the number of ω's in B', which is strictly greater than the block edit distance of the optimal schedule.

Therefore we can solve the scheduling decision problem by determining whether the block edit distance is equal to $(2n+1)\ c_{\text{block}}$ plus c_{char} times the number of ω's in B. □

Extensions for the other hard models of Section 4 follow directly.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers and D.J. Lipman, Basic local alignment search tool, *J. Mol. Biol.* **215** (1990) 403–410.

[2] W.I. Chang and E.L. Lawler, Approximate string matching in sublinear expected time, in: *Proc. Symp. on Foundations of Computer Science* (1990) 116–124.

[3] Z. Galil and R. Giancarlo, Data Structures and algorithms for approximate string matching, *J. Complexity* **4** (1988) 33–72.

[4] Z. Galil and K. Park, An improved algorithm for approximate string matching, *SIAM J. Comput.* **19** (6) (1990) 989–999.

[5] M.R. Garey and D.S. Johnson, *A Guide to the Theory of* NP-*Completeness* (W.H. Freeman, New York, 1979).

[6] W.B. Goad and M.I. Kanehisa, Pattern recognition in nucleic acid sequences I: A general method for finding local homologies and symmetries, *Nucleic Acids Res.* **10** (1982) 247–263.

[7] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs* (Academic Press, New York, 1980).

[8] M. Gribskov and J. Devereux, *Sequence Analysis Primer* (Stockton Press, New York, NY, 1991).

[9] J.D. Hall and E.W. Myers, A software tool for finding locally optimal alignments in protein and nucleic acid sequence, *CABIOS* **4** (1988) 35–40.

[10] X. Huang, R.C. Hardison and W. Miller, A space-efficient algorithm for local similarities, *CABIOS* **6** (1990) 373–381.

[11] X. Huang and W. Miller, A time-efficient, linear-space local similarity algorithm, *Adv. Appl. Math.* **12** (1991) 337–357.

[12] G.M. Landau and U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms* **10** (1989) 157–169.

[13] V.I. Levenshtein, Binay codes capable of correcting deletions, insertions, and reversals, *Cybernet. and Control Theory* **10** (8) (1966) 707–710.

[14] R.J. Lipton and D.P. Lopresti, A systolic array for rapid string comparison, in: H. Fuchs, ed., *Proc. 1985 Chapel Hill Conf. on Very Large Scale Integration* (Computer Science Press, Rockville, MD, 1985) 363–376.

[15] D. Lopresti and A. Tomkins, On the searchability of electronic ink, in: *Proc. 4th Internat. Workshop on Frontiers in Handwriting Recognition*, Taipei, Taiwan (1994) 156–165.

[16] D. Lopresti and A. Tomkins, Temporal domain matching of hand-drawn pictorial queries, in: M.L. Simner, C.G. Leedham and A.J.W.M. Thomassen, eds., *Handwriting and Drawing Research: Basic and Applied Issues* (IOS Press, Amsterdam, 1996) 387–401.

[17] D. Lopresti, A. Tomkins and J. Zhou, Algorithms for matching hand-drawn sketches, in: *Proc. 5th Internat. Workshop on Frontiers in Handwriting Recognition*, Colchester, England (1996) 233–238.

[18] S.B. Needleman and C.D. Wunsch, A general method applicable to the search for similarities in the amino-acid sequences of two proteins, *J. Mol. Biol.* **48** (1970) 443–453.

[19] F.S. Roberts, *Graph Theory and its Applications to Problems of Society* (SIAM, Philadelphia, PA, 1978).

[20] D. Sankoff and J.B. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* (Addison-Wesley, Reading, MA, 1983).
[21] P.H. Sellers, Pattern recognition in genetic sequences by mismatch density, *Bull. Math. Biol.* **46** (1984) 652–686.
[22] T.F. Smith and M.S. Waterman, Identification of common molecular sequences, *J. Mol. Biol.* **147** (1981) 195–197.
[23] G.A. Stephen, *String Searching Algorithms* (World Scientific, Singapore, 1994).
[24] W.F. Tichy, The string-to-string correction problem with block moves, *ACM Trans. Comput. Systems* **2** (4) (1984) 309–321.
[25] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100–118.
[26] R.A. Wagner and M.J. Fischer, The string-to-string correction problem, *J. Assoc. Comput. Machinery* **21** (1974) 168–173.
[27] M.S. Waterman and M. Eggert, A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons, *J. Mol. Biol.* **197** (1987) 723–728.