Block Edit Models for Approximate String Matching

Daniel P. Lopresti
Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
Princeton, NJ 08540

Andrew Tomkins
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

In this paper we examine the concept of string block edit distance, where 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 [15, 14] has found widespread application in fields ranging from molecular biology to bird song classification [12]. A great deal of research has been devoted to this area, and numerous algorithms have been proposed for computing edit distance efficiently (*e.g.*, [9, 13, 3, 8, 2, 4]).

In a previous paper [10], 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 linear process: the motion 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.

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. Figure 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 Figure 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*.

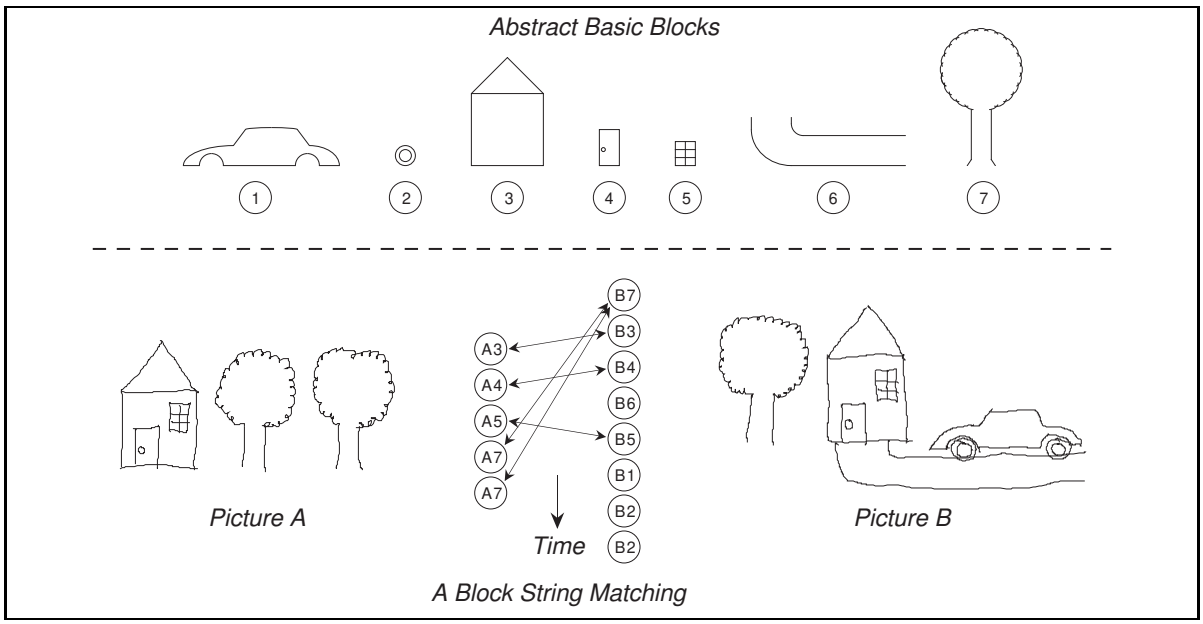

Figure 1: Approximate string matching applied to hand-drawn pictorial data.

Likewise, in genetic sequence alignment, some biologists suggest that comparisons based on simple edit distance may fail to account for certain common evolutionary processes [7]:

> 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. [pg. 96]

So-called "local" schemes, such as the well-known BLAST algorithm [1], can determine subregions of similarity within two longer sequences. However, to the best of our knowledge, there presently exists no fully automatic technique for correlating the relationships between the multiple local matches returned.

Manual inspection of a "dot matrix" plot appears to be the most popular approach for addressing this problem today.[1] As shown in Figure 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 i^{th} symbol of A is the same as the j^{th} 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.

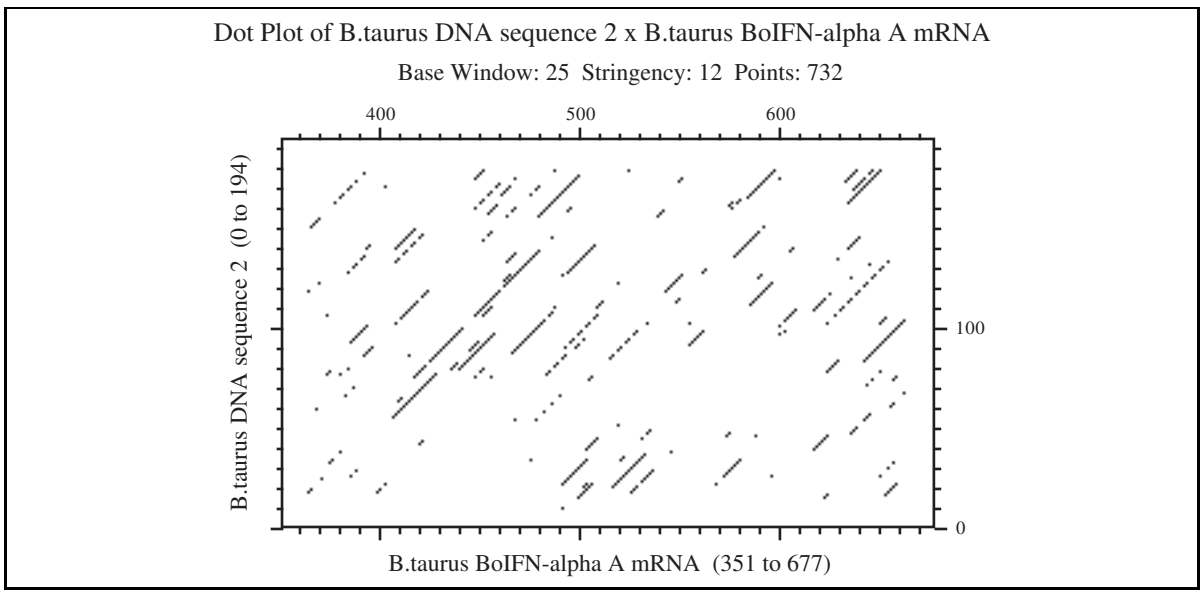

Figure 2: Dot matrix plot of two short genetic sequences.

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 examples. We prove that certain variants 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 a *trace*. 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	∅	x	
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	
k	∅	i	c	k		d	r	a	w	∅		f	l	a	x	

[1] Also from [7]: "Dot matrix analysis is the only currently available tool that deals sensibly with this phenomenon." [pg. 96]

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

As a rule, the arrows in a trace are not allowed to cross.[2] Moreover, the character-to-character correspondence is determined on an individual basis, with no regard to higher-level structure. Consider now a trace comparing the strings "hello world" and "world hello":

h	e	l	l	o		w	o	r	l	d	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	w	o	r	l	d		h	e	l	l	o

The "cost" of this alignment is five deletions and five 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.*, [15]) produces a trace 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.

Figure 3 presents a *block trace* 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 traces such as this.

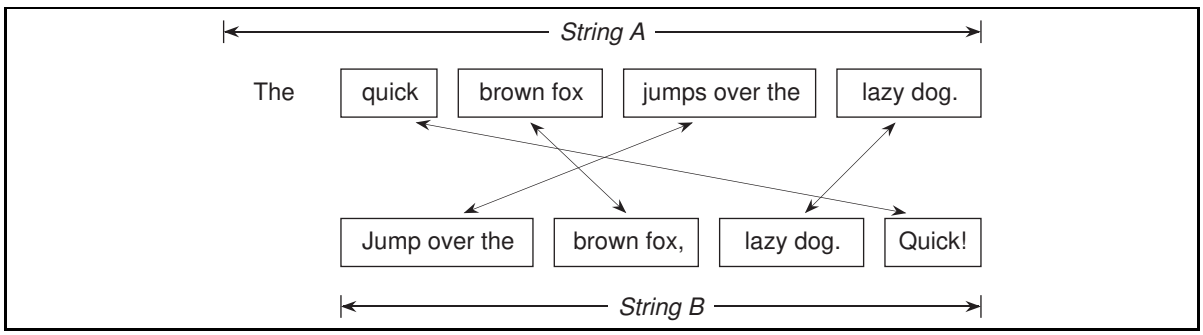

Figure 3: Example of a block trace.

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

2.1 Substring Families

Assume we have a finite alphabet Σ. Let $A = [a_1 a_2 \ldots a_m]$ and $B = [b_1 b_2 \ldots b_n]$ 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

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

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, Figure 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. Figure 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 Figure 3 one of the substring families is a disjoint cover, while the other 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:

$$dist : \{i, j \mid 1 \leq i \leq j \leq |A|\} \times \{k, l \mid 1 \leq k \leq l \leq |B|\} \rightarrow \mathcal{R}$$

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, so the generality of the cost function does not affect the difficulty of the problem.[3]

The *block edit distance* $\mathcal{B}$ between two strings A and B is defined as the best way to choose 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.[4] The correspondence between blocks is given by a permutation $\sigma \in S(t)$ from the permutation group on t elements. We impose the additional restriction that if

[3] 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.

[4] The distance between two blocks could be augmented with information about how far apart they are in the original A and B strings. The algorithms we shall present can be trivially extended to allow for this.

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

Table 1: Summary of the results presented in this paper.

$i \neq j$ 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. More formally,

$$\mathcal{B}(A,B) \equiv \min_{t} \min_{A|_t, B|_t} \min_{\sigma \in S(t)} \left\{ \sum_{i=1}^{t} dist\left(A^{(i)}, B^{(\sigma(i))}\right) \right\} \tag{1}$$

Equation 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.

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 a later section, 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 $\text{JOB}_t \in T$, a length $l(t) \in Z^+$, a release time $r(t) \in Z_0^+$, and a deadline $d(t) \in Z^+$.
>
> **Question:** Is there a one-processor schedule for T that satisfies the release time constraints and meets all the deadlines?

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

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

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, $D = \max\{d(t)\}$. 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 $\sum_i l(i) = D$. 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 $D - \sum_i l(i)$ additional jobs with length 1, release time 0, and deadline D to meet the constraint without changing the problem. Figure 4 depicts the two strings.

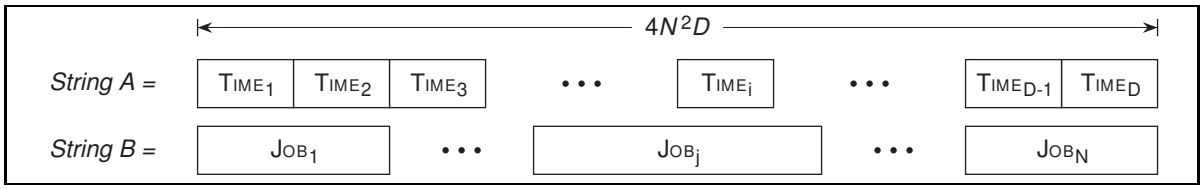

Figure 4: Strings A and B for the NP-completeness reduction.

Each of the time-step blocks in string A is a filled-in copy of the template shown in Figure 5. We need some new notation for referencing these substrings. We will write $A[\text{TIME}_i]$ to refer to the i^{th} time-step of A, and $A[\text{TIME}_i, \text{CHUNK}_j]$ to refer to the j^{th} "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 $r(j) \leq i$), and #(0) otherwise. Similarly, the second component is #(j) if JOB_j may end at time-step i (*i.e.*, if and only if $d(j) \geq i$), 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 Figure 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 i^{th} 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 k^{th} 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.

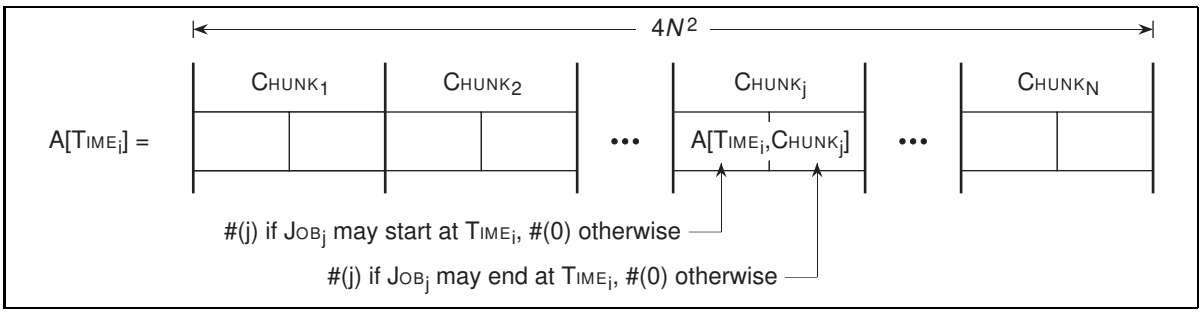

Figure 5: Template of a time-step block.

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:

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

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

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

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

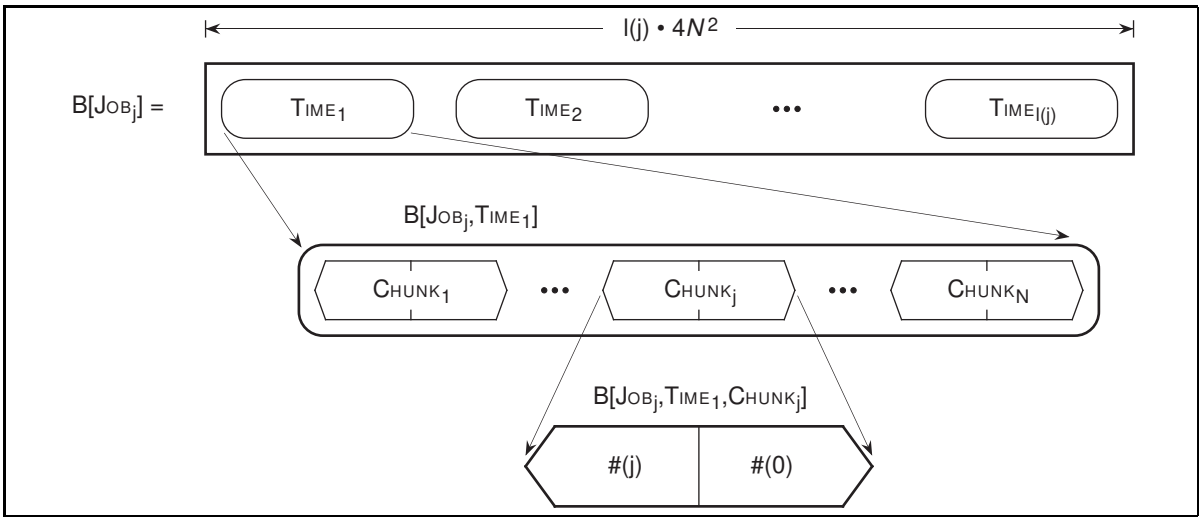

Figure 6: Template of a job block.

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. $r(j) \leq i_1$ and $d(j) \geq i_2$

This particular cost function is formulated so that a zero-cost matching, if one exists, yields a solution to the uniprocessor scheduling problem. We now prove a series of lemmas that will allow us to equate block string matches with job schedules.

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 $i \geq 0$. Any substring t of length $2N$ will overlap at most two such blocks, say s_1 and s_2. Create $s = s_1 \mid\mid s_2$. 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 $1 \leq j \leq N$.*
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 $r(j) \leq i_1$ 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 $i_2 - i_1 + 1 = l(j)$, $r(j) \leq i_1$, and $d(j) \geq i_2$.*

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 Equation 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-C$\overline{\text{D}}$, and C$\overline{\text{D}}$-$\overline{\text{C}}$D 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-$\overline{\text{C}}$D 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 C$\overline{\text{D}}$-$\overline{\text{C}}$D 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 $C\overline{D}$-$C\overline{D}$ and $\overline{C}D$-$\overline{C}D$ block edit distance problems are NP-complete.*

Proof. For the C$\overline{\text{D}}$-C$\overline{\text{D}}$ model, we can adapt the reduction fairly simply. In this case, neither string must be disjoint, so time-steps and jobs can be re-used more than once. We change the distance measure so that a valid match between JOB_i 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 $\overline{\text{C}}$D-$\overline{\text{C}}$D 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. Hence if a matching with distance $-N$ can be found, it must correspond to a schedule. If no such match can be found, then no schedule exists. This completes the proof of the theorem. □

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 $1 \leq i \leq j \leq m$:

$$W^1(i,j) \equiv \min_{k \leq l} \{dist(a_i \ldots a_j, b_k \ldots b_l)\} \tag{5}$$

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. 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.

Consider the diagram shown in Figure 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 Figure 7 that W^1 induces a complete interval graph, a well-studied class for which most known problems have efficient solutions [11, 6]. 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.

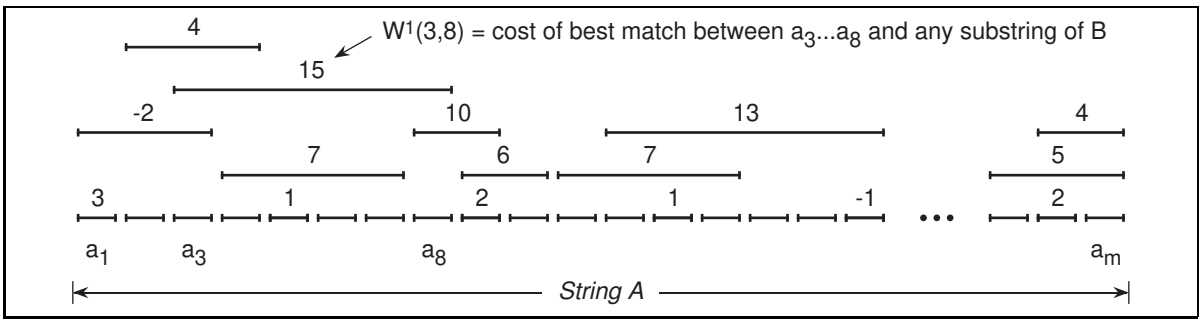

Figure 7: Possible string matches viewed as intervals.

We define $\mathcal{M}(i)$ 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 $\mathcal{M}$ for $i = 1, 2, \ldots, m$ (recall that $|A| = m$), our final answer is $\mathcal{B}(A,B) = \mathcal{M}(m)$.

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 $\mathcal{M}$ using the following recurrence:

Algorithm CD-$\overline{\text{CD}}$

$$\mathcal{M}(i) = \min_{j<i} \Big\{ \mathcal{M}(j) + W^1(j+1, i) \Big\} \tag{6}$$

In this recurrence, $\mathcal{M}(i)$ 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. By dynamic programming, the value of $\mathcal{M}(i)$ can be computed in time $O(i)$ given all previous $\mathcal{M}(j < i)$. Thus, the total time to compute $\mathcal{M}(m)$ is $O(m^2) + T(W)$.

5.2 $\overline{\text{CD}}$-$\overline{\text{CD}}$ 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 Equation 6, $W^{0,1}$ allows sections of A to be "skipped" whenever it is advantageous to do so. The recurrence is:

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

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

5.3 C$\overline{\text{D}}$-$\overline{\text{CD}}$ 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 Equation 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:

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

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

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

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

$$\text{Algorithm C}\overline{\text{D}}\text{-}\overline{\text{CD}}\mathcal{M}(i) = \min_{j<i}\left\{\min_{k\in[j,i-1]} \mathcal{M}(k) + W^+(j+1,i) + \sum_{k\in[j+2,i]} W^*(k,i)\right\} \tag{11}$$

Intuitively, the recurrence can be understood as follows: we require at least one interval in B to cover a_i, yielding the W^+ term. We can also include other intervals ending at a_i, but only if they lower the overall cost, giving us the W^* term. Finally, we combine this with any previous best matching ending somewhere in the range $[j, i-1]$ (recall that the substring family of A must be a cover), which explains the $\mathcal{M}$ term.

Note that this can still be computed in $O(m^2)$ time, despite the additional minimization and summation. If the outer minimization is taken in reverse order (*i.e.*,

$j = (i-1), \ldots, 1)$, then for fixed i the value $\min_{k \in [j-1, i-1]} \mathcal{M}(k)$ can be computed from $\min_{k \in [j, i-1]} \mathcal{M}(k)$ in constant time:

$$\min_{k \in [j-1, i-1]} \mathcal{M}(k) = \min \left\{ \mathcal{M}(j-1), \min_{k \in [j, i-1]} \mathcal{M}(k) \right\} \tag{12}$$

A similar "trick" applies in the case of the summation. Hence, the running time remains $O(m^2) + T(W)$.

5.4 $\overline{\text{CD}}$-$\overline{\text{CD}}$ Block Edit Distance

Finally, we give a recurrence to solve the problem when neither substring family is constrained. We can extend the ideas of the previous subsection to solve this problem:

$$\text{Algorithm } \overline{\text{CD}}\text{-}\overline{\text{CD}} \quad \mathcal{M}(i) = \min_{j<i} \left\{ \mathcal{M}(i-1) + W^*(j+1, i) + \sum_{k \in [j+2, i]} W^*(k, i) \right\} \tag{13}$$

The $\mathcal{M}(i-1)$ term incorporates the best shorter matching (one which does not include a_i), and the use of W^* frees the substring family of A from having to be a cover. Since any new interval added to the matching at step i must overlap a_i, and no interval in the best so far may contain a_i, we maintain the invariant that no block pairing is used more than once.

Again, this 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.*, Equation 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, $T(W^1) = O(m^3 n^3)$. 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 $O(mn)$. This saves 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 $a_i \ldots a_k$ as well, for $i \leq k \leq n$. Hence, only $O(m)$ such tables need be built to compute W^1, saving another $O(m)$. Thus, $T(W^1) = O(m^2 n)$.

All of the other versions of W can be computed from W^1 and the tables used to build it in less time than this. Hence, in general, $T(W) = O(m^2 n)$. Since this dominates the time needed to evaluate the recurrences presented earlier, the full time complexity in all cases is $O(m^2 n)$.

6 Conclusions

In this paper we have examined the concept of string block edit distance, where 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.

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 an algorithm for computing W in $O(m^2 n)$ time – it would be interesting to know whether this result 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.

Acknowledgements

The genetic sequence dot matrix plot (Figure 2) was generated using "Dotty Plotter," a program written by Don Gilbert.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] W. I. Chang and E. L. Lawler. Approximate string matching in sublinear expected time. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 116–124, 1990.

[3] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

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

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

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

[7] M. Gribskov and J. Devereux. *Sequence Analysis Primer.* Stockton Press, 1991.

[8] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[9] R. J. Lipton and D. P. Lopresti. A systolic array for rapid string comparison. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 363–376. Computer Science Press, 1985.

[10] Daniel Lopresti and Andrew Tomkins. On the searchability of electronic ink. In *Proceedings of the Fourth International Workshop on Frontiers in Handwriting Recognition*, pages 156–165, Taipei, Taiwan, December 1994.

[11] F. S. Roberts. *Graph Theory and Its Applications to Problems of Society.* SIAM, Philadelphia, PA, 1978.

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

[13] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[14] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proceedings of the 7th ACM Symposium on Theory of Computing.* Association for Computing Machinery, 1975.

[15] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, 1974.