Column Time Warping with Neighborhood Distortion Cost

by
Brennan Mark Kucey

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented September 3, 2014
Commencement June 2015

AN ABSTRACT OF THE THESIS OF

Brennan Mark Kucey for the degree of Honors Baccalaureate of Science in Computer Science presented on September 3, 2014. Title: Column Time Warping with Neighborhood Distortion Cost.

Abstract approved: _____

Thomas Dietterich

The Cornell Laboratory of Ornithology coordinates the eBird Project in which volunteer bird watchers participate in a checklist program. Each time they go bird watching, they fill out a checklist of the number of birds of each species that they saw and upload it to a web site. This information has been used to fit models of the spatial distribution of each species of bird on a daily basis. One model pools data from many years; it provides a summary of the typical timing of bird migration each year. A second model describes the locations of the birds for each year separately. One important problem is to visualize, for each year, whether the birds are "ahead" or "behind" their typical migration timing. To do this, an algorithm was developed for "warping" the spatio-temporal distribution of the birds for a single year so that it matched the average spatio-temporal distribution. The algorithm only solves the problem approximately. The goal of this thesis was to understand the computation complexity of this time warping problem and to relate it to other known algorithms. Our analysis suggests, but does not prove, that the spatio-temporal time warping problem is computationally intractable (NP-Hard).

Key Words: Time Warping, Bird Migrations, Migration Analysis

Corresponding e-mail address: kuceyb@lifetime.oregonstate.edu

Column Time Warping with Neighborhood Distortion Cost

by
Brennan Mark Kucey

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Degree Science in Computer Science
(Honors Associate)

Presented September 3, 2014
Commencement June 2015

Honors Baccalaureate of Science in Computer Science project of Brennan Mark Kucey presented on September 3, 2014.

APPROVED:

Thomas Dietterich, Mentor, representing the School of Electrical Engineering and Computer Science

Glencora Borradaile, Committee Member, representing the School of Electrical Engineering and Computer Science

Alan Fern, Committee Member, representing the School of Electrical Engineering and Computer Science

Toni Doolen, Dean, University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, University Honors College. My signature below authorizes release of my project to any reader upon request.

Brennan Mark Kucey, Author

Acknowledgements

I would like to thank my mentor, Dr. Dietterich, for guiding me throughout the thesis process. This work would not have been possible without his kind and patient help.

I thank Dr. Borradaile and Dr. Fern for being on my committee and sharing their expertise.

Thanks to Liping Liu for sharing his work on the problem and clarifications.

Lastly, I would like to thank my family for their support during my time at Oregon State University and the thesis process.

This material is based upon work supported by the National Science Foundation under Grant No. 0832804. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Figures

List of Appendix Figures

1 Introduction

1.1 Background Motivation

An important part of conservation management is to understand the animals in the area of concern. Of the animals to observe in North America, some of the easiest species to detect are birds. Many birds migrate annually, making them easily detectable during that time. Observations of these annual migration patterns could produce insight into how human populations negatively affect neighboring animal populations.

The eBird project is a community of volunteer bird observers reporting their sightings to the eBird website. This provides some sparse data to work with, but much more is needed to track the annual migrations of birds in North America. Using the ebird data, the STEM modelling project, headed by Dr. Daniel Fink of Cornell Lab of Ornithology, interpolates these sightings to provide approximations of bird migrations in locations across the United States where no observations were made.

To observe these bird species over time, we can look at each species' annual migration. For each bird species, we can compare the annual migration data of one year with the data averaged over several years to see when and where the birds fall behind or advance ahead of their normal annual migration schedule. The map of the US can be divided up into cells. For each cell, the birds will start being observed at some point and cease being observed at some later point. For each cell in the map, they may appear sooner or later than in previous years. This restriction prevents us from using previously-developed time warping algorithms and forces us to create our own problem definition: the problem of Column Time Warping with Neighborhood Distortion Cost.

1.2 Thesis Scope

The scope of this thesis is to evaluate the tractability of the Column Dynamic Time Warping with Neighborhood Distortion Cost problem. Evaluating the tractability of the problem determines whether the problem is difficult enough to necessitate the use of heuristics (approximation algorithms). If the problem is found to be intractable (NP-HARD), then heuristics will be necessary to solve the problem within a reasonable amount of time. Alternatively, if the problem is found to be tractable (PTIME), then an exact algorithm will work.

1.3 Thesis Organization

To analyze the problem, the second section of this thesis formalizes the Column Dynamic Time Warping with Neighborhood Distortion Cost problem in detail. The third section gives evidence of the problem not being in P. The fourth section discusses the implications of this work in relation to the theoretical computer science field. The appendix shows an attempt at a PLANAR 3-SAT reduction for proving the problem to be NP-HARD.

2 Column Time Warping with Neighborhood Distortion Cost

2.1 Formulation from Bird Migration

Bird migrations for each species of concern are described in eBird using a series of observations with the following information: latitude, longitude, time, and bird count. To see where and when birds are behind or ahead of their normal migration schedule, we compare two different annual migration years with an alignment. To align a bird species' annual migration of one year, A, to another year, B, is to match every observation in A to an observation in B. A match between two observations represents the relationship that the birds observed in the observation in B are roughly the same birds as observed in the observation in A. These two observations happened at approximately the same stage of the birds' annual migration at different years. This is considered an alignment problem, and the terms "matching" and "alignment" are used interchangeably. The amount of precision of latitude and longitude makes it difficult to match two observations from different years because it is too specific for our sparse data. The odds of two observations at the same latitude, longitude pair is small.

To reduce the precision of observations, we produce a discretized representation of the annual migrations into a three dimensional array. The geographical map of interest is divided into a matrix of cells, taking place of latitude and longitude. The time component is represented as an integer day. Then, each cell has the percent chance of an observer spotting a bird of the species of interest.

The percentages are calculated as follows. First, a STEM model is created using the checklists (forms to fill out) recorded from bird observations. The STEM model takes as input latitude, longitude, and date, then outputs the percent chance that an observer at that time and location would observe a bird of a specified species. The date component discretizes time, leaving latitude and longitude for discretizing into cells of the geographic matrix. Second, a large set of points called the spatial random dataset (SRD) that was defined was spread across the geographic map for sampling of the STEM model. Each cell on a given day holds its respective percent chance of an observer within a cell seeing the specified bird species is calculated as the average of the STEM values of the SRD points within that cell.

The three dimensional representation of each species' annual bird migration appears similar to other alignment problems. Star Alignment, Tree Alignment, and Multiple Alignment could not be reduced to the Column Time Warp with Neighborhood Distortion Cost problem (Elias 2003). Three Dimensional Time Warping is a very similar problem (Mardziel 2004). The main principle that makes this problem different from other well documented alignment problems is that in any given cell, birds tend to enter at the same time each year. The importance of this alignment problem is to notice, in any given cell, the variations of these arrival times. This means that any observation within a cell can only be "matched" or aligned with other observations within the same cell. This allows for the alignment of birds within a cell at a specified time to birds from a previous year within the same cell to be interpreted as the birds being ahead of or behind their regular migration schedule. For example, within one cell, if day 144 of the year 2010 is aligned with day 146 of the year 2009, then the interpretation is in the year 2010 the birds within that cell on day 144 were two days ahead of the migration schedule of 2009. It is important to note that in this context the terms "align", "match", and "map" have the same meaning.

2.2 Definition

The problem of Column Time Warping with Neighborhood Distortion Cost concerns the alignment of two inputs (annual migration data), A and B, such that the alignment cost from elements of A to elements of B is minimized.

2.2.1 Input

A and B are two dimensional matrices of sequences with matching dimensions and lengths, $N \times M$ and L. All sequences of A and B have a length of L. These sequences have elements with real value numbers (the species observation probability). To access a single sequence, we use the notation $A_{(row,column)}$, where row is in N, column is in M. To access a specific element, we use the notation $A_{(row,column)}$[index], where index is within L. All elements of the sequences have real values. Mathematically,

$\forall_{(r,c,i) \in (N,M,L)} A_{(r,c)}[i] \in \mathbb{R}$, and likewise for B.

The dimensions N and M are for the latitude index and longitude index space. The L sequence length is the time component, holding the number of days considered for the annual migration. For example, $A_{(3,4)}[20] = 5\times10^{-4}$ means that on day 20 of the considered migration period, the bird observation probability in cell c (3,4) is 5×10^{-4}

2.2.2 Output

The output of this problem is an alignment function f. The function, similar to the input, is a two-dimensional matrix of sequences, matching the dimensions of inputs A and B. The sequences have elements of integer value. The notation for accessing of each sequence and element is the same as the inputs A and B. Mathematically,

$$\forall_{(r,c,i) \in (N,M,L)} f_{(r,c)}[i] \in \mathbb{Z}.$$

This function f represents the alignment of A onto B. The alignment of each pair of sequences $A_{(x,y)}$ and $B_{(x,y)}$ is represented with $f_{(x,y)}$.

Each element in $f_{(x,y)}$ represents the matching of its corresponding element in $A_{(x,y)}$ to an element in $B_{(x,y)}$. For example, $f_{(3,4)}[5] = 6$ means $A_{(3,4)}[5]$ is aligned to $B_{(3,4)}[6]$. According to Time Warping, the alignment of one element cannot reach further than its neighbors in the sequence, because warping is compression and expansion, without flipping (Gusfield 2007). In other words, during spring migration, the birds must continue flying North; they are not allowed to reverse direction. The Warping Restriction is, mathematically,

$$\forall i \in \left|f_{(x,y)}\right| \;\; f_{(x,y)}[i] \leq f_{(x,y)}[i+1].$$

As a running example, we will use the following example:

Sequence A										
A[0]		A[1]		A[2]		A[3]		A[4]		A[5]
5		6		5		5		12		5
↓	↙		↙		↙		↙			↓
5		5		6		10		5		5
B[0]		B[1]		B[2]		B[3]		B[4]		B[5]
Sequence B										

1 Example Sequence Alignment

2.2.3 Cost Functions

To rank the optimality of alignments, they are compared according to the negative sum of three cost functions associated with this problem. The first two are the matching cost and skipping cost associated with aligning two sequences, and the third is a neighborhood distortion cost. These costs are closely related to the costs of a paper by Keysers and Unger (2003). To calculate the total cost of an alignment, we sum the cost each element incurs according to these three cost functions. This is done by inspecting each element.

2.2.3.1 Matching Cost

The alignment of element $A_{(x,y)}$ to element $B_{(x,y)}$ incurs the cost

$$c_m\left(A_{(x,y)}, B_{(x,y)}\right) = \left|A_{(x,y)} - B_{(x,y)}\right|.$$

It is the absolute value of the elements' difference. When the elements have the same value, no cost is incurred. However, when they differ, this difference is counted towards the cost. This makes the optimal alignment lean towards matching elements of similar value.

For example, in the Example Sequence Alignment above, aligning A[2] with B[1] has no matching cost because they are both the same value. Matching A[4] to B[3] incurs a cost of 2.

2.2.3.2 Skipping cost

The skipping cost considers an alignment between two sequences represented with $f_{(x,y)}$. The skipping cost is

$$c_s\left(f_{(x,y)}\right) = \sum_{i\in(\|L\|-1)}\left|f_{(x,y)}[i+1] - f_{(x,y)}[i] - 1\right|.$$

Note that with the Warping Restriction

$$\forall i \in \left|f_{(x,y)}\right| \; f_{(x,y)}[i] \leq f_{(x,y)}[i+1]$$

the difference

$$f_{(x,y)}[i+1] - f_{(x,y)}[i] \geq 0$$

within the skipping cost will always be nonnegative.

This function penalizes deviations from the identity mapping function, $f_i(x)=x$. These deviations are sensed according to sequential neighbors. Using the Example Sequence Alignment on page 4, A[2] aligning to B[1] incurs no cost, because its sequential neighbors A[0] and A[2] both align to

the left also. A[0] incurs a cost of 1, because it differs with its neighbor A[1]'s alignment arrow by 1. The same applies to A[4].

2.2.3.3 Neighbor Distortion Cost

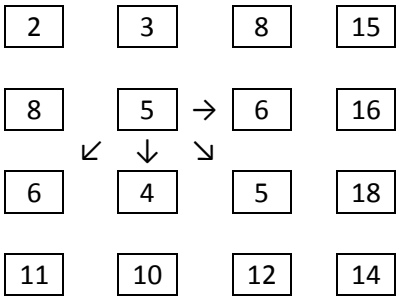
During migration, we do expect that the behavior of birds in spatially-neighboring cells should be similar. Hence, we do not expect that birds in one cell will be 5 days ahead of schedule, while birds in a neighboring cell will be 6 days behind. To constrain the matching, we introduce a neighbor distortion cost. This cost is also concerned with the alignment function, f. It takes the absolute difference of each element of f and its eight neighboring sequences. These differences are summed together for the total neighborhood distortion cost. Mathematically, with an element $f_{(x,y)}[d]$ and any of its nonsequential neighbors $f_{(x',y')}[d] | (x', y') \in \{1,0,-1\} \times \{1,0,-1\} \backslash \{(0,0)\}$, their distortion cost is

$$c_n\left(f_{(x,y)}[d], f_{(x',y')}[d]\right) = \left|f_{(x,y)}[d] - f_{(x'y')}[d]\right|.$$

Note that the cost between two neighboring elements will be counted once, not twice.

Ideally, each element of f would have the same alignment value as each of its eight nonsequential neighbors ($t_1 = t_2$). The differences between these neighbors are distortion from the ideal. This cost penalizes alignment functions for distortion between neighboring sequences.

For example, consider the Neighborhood Distortion Cost Example on page 6. The element of 5 is compared to 4 of its 8 neighbors. This is how the cost is calculated when summing the inspected distortion cost each element, one at a time. Note that to prevent double-counting, the distortion costs that 5 incurs with its other neighbors 8,2,3, and 8 will be added during the inspection of each of those neighbors.



2 Neighborhood Distortion Cost Example

2.2.4 Problem Statement

Now that the input, output, and cost functions have been described, we can now have a formal problem statement.

2.2.4.1 Instance

The problem starts out with a pair of three dimensional arrays, A and B, holding real values.

2.2.4.2 Solution

The answer to our problem is a mapping function, f, that maps each element of A to an element of B. It takes the form

$$f: (N \times M \times L) \mapsto (N \times M \times L)$$

That is, for each sequence A_{xy}, f assigns every element in A_{xy} an element in B_{xy} while following the warping constraint.

2.2.4.3 Measure

The cost of the assignments is the sum of the matching costs of every element of A to the elements in B, the skipping costs between these assignments and the sum of the neighbor distortion costs between these assignments. Mathematically, the cost is

$$c(\mathrm{A},\mathrm{B},\mathrm{f}) = \sum_{(\mathrm{x},\mathrm{y})\in N\times \mathrm{M}} c_s(\mathrm{f}_{xy}) + \sum_{(\mathrm{x},\mathrm{y},\mathrm{z})\in N\times M\times L} \left(\begin{array}{l} c_m(\mathrm{A}_{xy}[\mathrm{z}], \mathrm{B}_{xy}[\mathrm{f}_{xy}[\mathrm{z}]]) \\ + \sum_{(\mathrm{i},\mathrm{j},\mathrm{z})\in \left\{ \begin{array}{l} (\mathrm{x}-1,\mathrm{y}+1,\mathrm{z}),(\mathrm{x},\mathrm{y}+1,\mathrm{z}), \\ (\mathrm{x}+1,\mathrm{y}+1,\mathrm{z}),(\mathrm{x}+1,\mathrm{y},\mathrm{z}) \end{array} \right\} \cap \mathrm{N}\times\mathrm{M}\times\mathrm{z}} c_n(\mathrm{f}_{xy}[\mathrm{z}], \mathrm{f}_{ij}[\mathrm{z}]) \end{array} \right)$$

where c_m is the matching cost function, c_s is the skipping cost function, and c_n is the neighborhood distortion cost function.

2.2.4.4 Goal

Find f, the assignments for each element of A, to minimize the total cost.

2.3 Within NP

Given the problem definition, it can be shown that the Column Time Warp with Neighborhood Distortion Cost problem is in NP. An NP problem requires the solution be verifiable in polynomial time. The question of whether a mapping function f satisfies, for a give c',

$$c(A, B, f) \leq c'$$

is evaluated in polynomial time because the cost function iterates over the output once. This means that the Column Time Warp with Neighborhood Distortion Cost problem is in NP.

3 Counter Example Against P

Here we introduce an instance of the alignment problem that we claim cannot be optimized in P time. We align the elements of A onto the elements of B. To simplify the problem for illustration, a two dimensional version of the Column Time Warping with Neighborhood Distortion Cost problem will be presented. Instead of a two dimensional matrix of sequences, a one dimensional array of sequences will be used. The matching and skipping cost functions will remain unchanged because they are costs of aligning sequences. The neighborhood distortion cost will change to only include two nonsequential neighbors instead of eight. The difficulty of reducing the neighborhood distortion cost is present in both problem versions.

For the sake of the example, we use a different index system. We will use sequences of length ten. The indices will range from -4 to 5. This will help distinguish the globally optimal alignment (negative) from the locally optimal alignment (positive).

First we consider the construction of a problem instance, the half mine that discourages P algorithms. The half mine will be mirrored on its right side to create a full mine. Putting full mines next to each other will create a minefield problem instance. In the minefield problem instance, whenever an algorithm wants to inspect one element or sequence at a time, it will approach a full mine from the outside and continue inwards. This approach of starting from the outside and inspecting inwards and the symmetry of the full mine allows us to inspect the half mine knowing that every approach will start from the outside (the left) and continue inwards (to the right).

3.1 Half Mine

3.1.1 Construction of A

We construct each column of A as a sequence of zeroes with value 100 at index 0. This sequence has the appearance of 0,0,0,0,100,0,0,0,0,0. Remember, we are using a different index system for this example, so 0 is close to the middle. This sequence is repeated 10 times.

3.1.2 Construction of B

We construct each column of B as a sequence of zeroes with some values specified in the sparse format {index:value}. For example, {-3:100, 1:25, 2:100} evaluates to 0,100,0,0,0,25,100,0,0,0.

The matrix B contains the following sequences:

{-2:100,0:100}

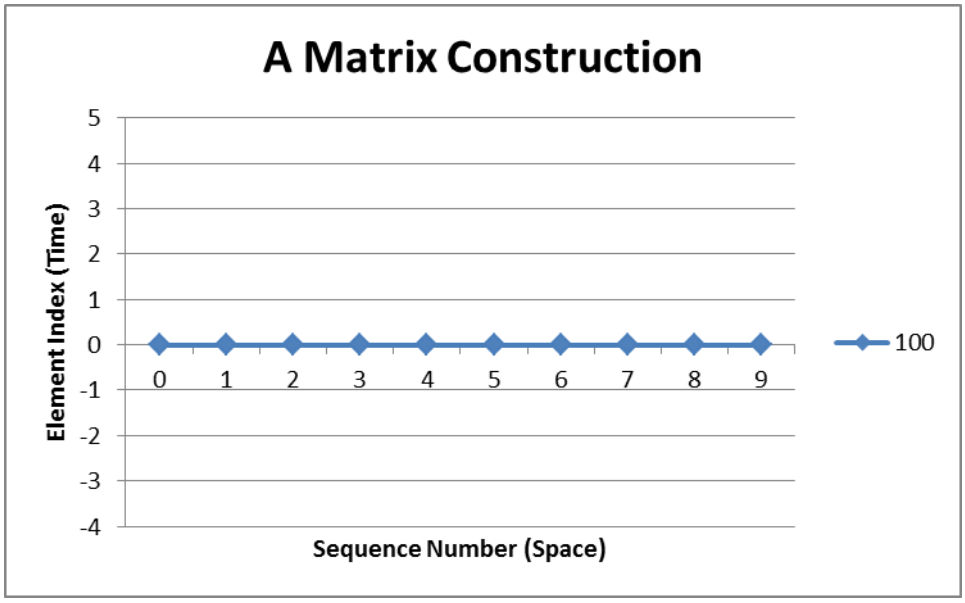
{-3:100,1:100}

{-3:100,1:25,2:100}

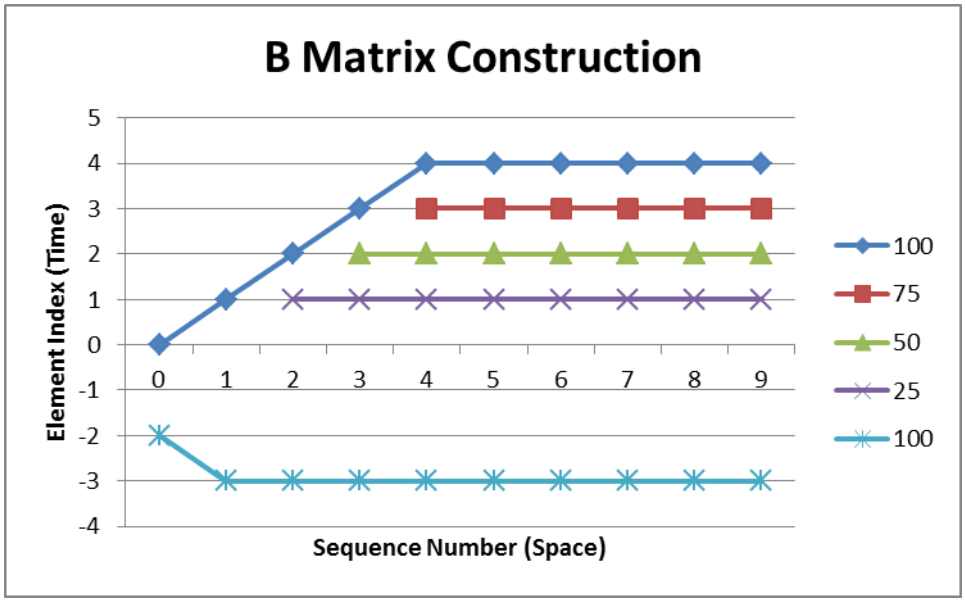
{-3:100,1:25,2:50,3:100}

{-3:100,1:25,2:50,3:75,4:100}.

The last sequence is repeated 5 times, but can be repeated more times to create a more drastic difference in alignment costs with different approaches. These constructions are visualized with the following graphs.



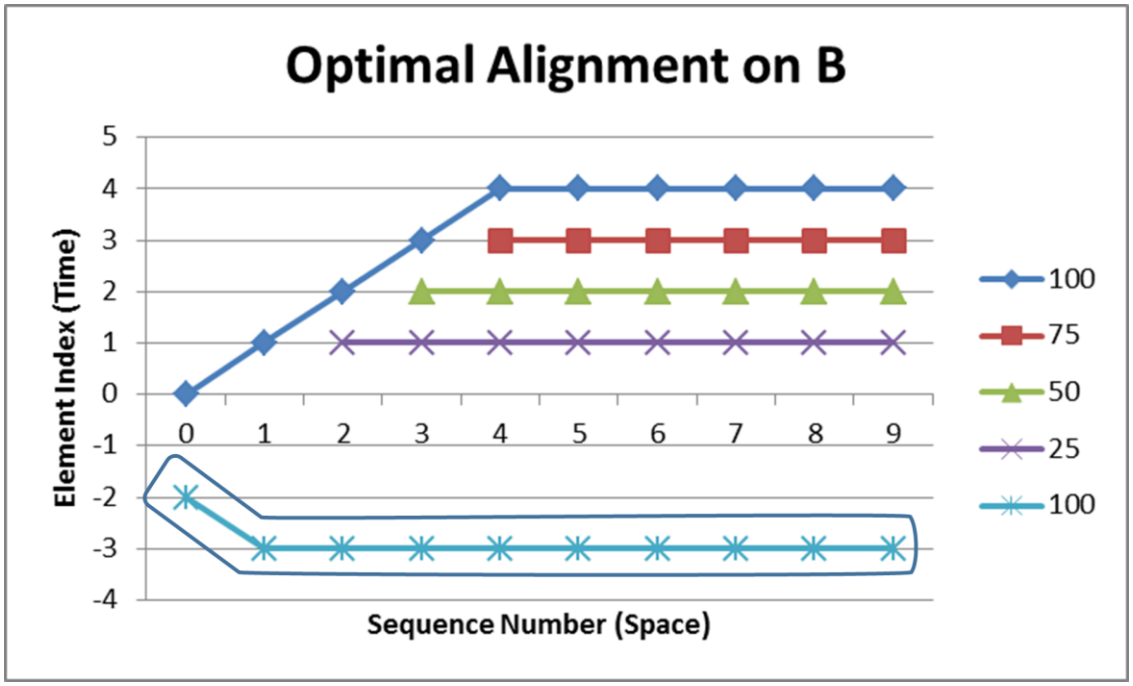
3 Half-Mine: A Matrix Construction



4 Half-Mine: B Matrix Construction

3.1.3 Optimal Alignment

The optimal alignment of A onto B is aligning the line within A (all $A_x[0]$) to the lower line segments in B. The matching costs have been set such that matching line segments is necessary, and the neighbor distortion costs are a secondary concern during alignment. However, since matching the line segments in A to one of the two lines in B is necessary for any local minimum in the alignment cost, the neighborhood distortion costs will determine which minimum alignment cost is optimal. To keep the neighborhood distortion costs minimal, aligning the A line to the lower B line will have each $A_c[0]$ aligned to $B_c[-3]$ ($\forall c \in \{1 \dots C\}\ f_c = -3$) except for $A_0[0]$ aligning with $B_0[-2]$ ($f_0[0] = -2$).



5 Half-Mine: Optimal Alignment

3.1.4 Ordering of Examples

The half mine is an array of sequences and can be seen as having two dimensions. Each $A_c[i]$ element is aligned, according to its corresponding $f_c[i]$, to a $B_c[i]$ element. The first dimension, i, is along the sequence, and the second dimension, c, is along the array. During some P time algorithms, each element in A will be visited to determine the best B element to match it to and record that in f. Some P time algorithms will take the approach to visiting each element by only travelling along one dimension at a time (besides divide and conquer). These dimensions and the different approaches within each dimension lead to the order of the examples.

The P time algorithm can approach the problem iterating over each sequence first, and the array second. This can be seen as assigning $f_0[5]$, then $f_0[4]$, etc. then $f_1[5]$, $f_1[4]$, etc. This is called smoothing, as it helps to smooth out the alignments to reduce the neighborhood distortion cost. During its visit to each sequence, we consider three different methods for creating the best alignment for that sequence. The first is to look at each element of the sequence, one at a time, and decide the best alignment for it; this is the Individual Smoothing method. The second is a slight modification of Dynamic Time Warping (Gusfield 2007), which we call the Column Smoothing method. Here a whole sequence alignment is done at once and the visiting order becomes f_0, f_1, etc. The last is the Multiple Smoothing method, were the alignment of the sequence considers all the possible alignments of the neighboring sequences in a lookahead.

In the second approach, a P time algorithm approaches the problem along the array dimension, visiting the first element of every sequence in order, then the second, etc. The f values are assigned in the order $f_0[0]$, $f_1[0]$, etc. This is very similar to Individual Smoothing. This approach is not considered in this paper.

Before the process of smoothing, some P time algorithms can try greedily creating the best alignment for each sequence, but not the problem as a whole in consideration of neighborhood distortion cost. Then it can begin smoothing to lower the neighborhood distortion cost through Smoothing. This method is Column DTW (Dynamic Time Warp). In cases where Column DTW is not used before Smoothing, then 0-Assignment will be used beforehand. 0-Assignment is $\underset{c,i}{\forall} f_c[i] = 0$.

The following table shows the order of the examples. The distinction between starting with a 0-Alignment and Column Time Warp is made between the first two examples. The distinction shows that starting with Column DTW is best, so 0-Assignment with Column Smoothing is not shown.

	0-Assignment	Column DTW
Individual Smoothing	1	2
Column Smoothing	X	3
Multiple Smoothing	4	4

Divide and conquer is a popular P time algorithm technique, but we will show an example of that technique not working for this problem.

3.1.5 0-Alignment to Individual Smoothing

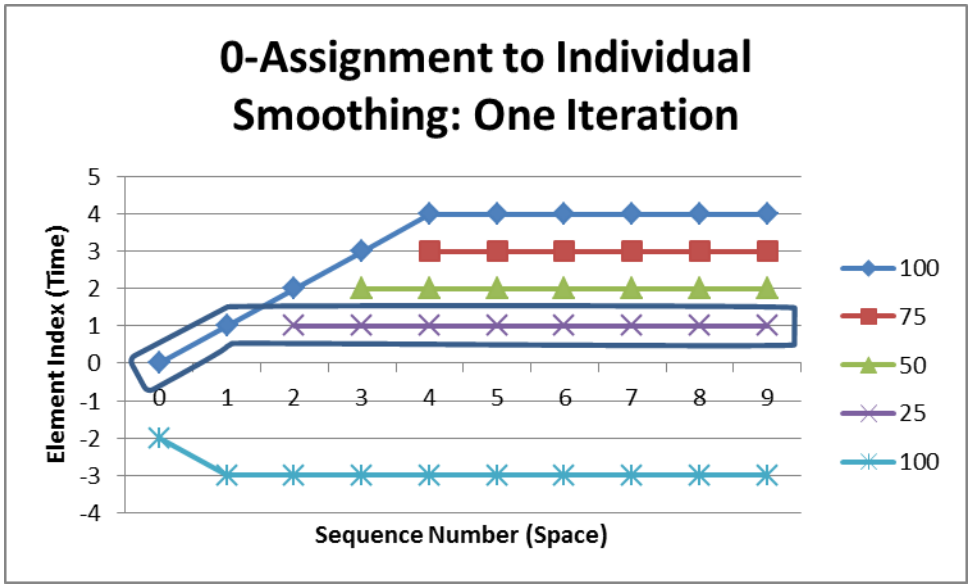
This algorithm starts with a 0-Assignment for f. Then it attempts to smooth out the neighborhood distortion costs. To do this, it visits every element of every sequence. During the visit to an element, it reconsider our assignment f_c with the options of moving our alignment either upwards or downwards ($f_c \pm 1$) while still constrained by the neighbors according to the Warping Restriction ($\forall i \in \|f_c\| \ f_c[i] \leq f_c[i+1]$). Only a change of one is considered, because it is the smallest nonzero number. Larger ranges of considered values are used in Column Smoothing. We reassign f_c with the lowest cost value according to the cost function $c(A, B, f)$. The pseudocode follows this paragraph. Please note that this pseudocode is meant for this reduced 2D problem version and can easily be modified to 3D with the inclusion of an N loop inside the C loop.

```
Smoothing(A,B,f){
      /* repeat smoothing until no further
         cost reductions can be made*/
      while(fPrev != f){
            /* analyze each column, in order */
            for(c=0; c<C; c++){
                  /* analyze each element of the sequence */
                  for(i=0; i<I; i++){
                        possible_alignments =
                              {f_m[l]-1, f_m[l],f_m[l]+1}
                                ∩ {N |(f_(m-1) < N) ∧ (N < f_(m+1))}
                        foreach p in possible_alignments{
                              /* make new possible alignment*/
                              fTry = f
                              fTry_c[l] = p
                              /* store alignment cost*/
                              p_cost = c(A,B,fTry)
                              add (p_cost,fTry) to possible_costs
                        }
                  }
                  f_m[l] = fTry of lowest p_cost in possible_costs
                  fPrev = f
            }
      }
}
```

6 Individual Smoothing Pseudocode

Smoothing from left to right, the horizontal line in A catches the front of our ramp in B. On the significant line in A, the first match considered is for $A_0[0]$. The matches considered are to $B_0[-1]$, $B_0[0]$, and $B_0[1]$. Note that $B_0[-2]$ would have been the optimal alignment, but it was not a match being considered, because it was out of range.

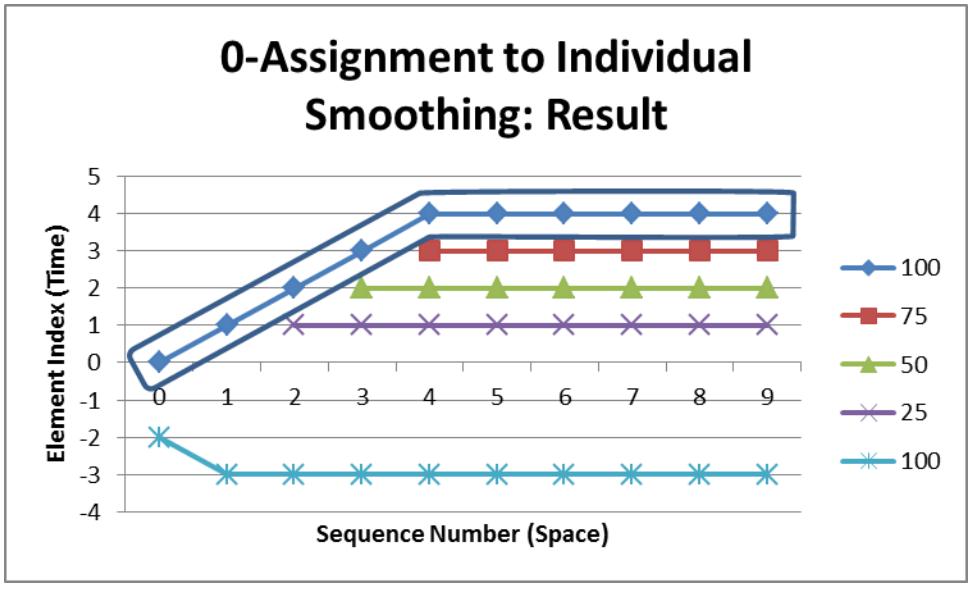
Now $A_0[0]$ has 0 displacement cost as it aligns with $B_0[0]$, beginning a cost decline into a suboptimal local minimum. On the next significant alignment, $A_1[0]$ considers $B_1[-1]$, $B_1[0]$, and $B_1[1]$ for matches. It has to choose $B_1[1]$ to reduce significant matching costs. As we progress across, each $A_c[0]$ chooses $B_c[1]$ to reduce significant matching costs as they are drawn along the matching cost gradient.



7 Half-Mine: 0-Assignment and Individual Smoothing After One Iteration

On the next iteration, $A_0[0]$ and $A_1[0]$ both maintain their alignments to avoid significant matching costs. $A_2[0]$, currently matched with $B_2[1]$, now considers and aligns with $B_2[2]$ to reduce significant matching costs. As we progress, each $A_c[0]$ matches with $B_c[2]$, furthering the A line alignment along the B gradient.

Upon further iterations, $A_3[0]$ matches with $B_3[3]$, and all line points after and including $A_4[0]$ align with their respective $B_c[4]$. The neighborhood distortion cost during the ramp up on $A_0[0]$ to $A_4[0]$ is greater than the neighborhood distortion cost of the optimal alignment. This is not the optimal alignment, and once the local minimum cost is reached, further iterations of smoothing bring no changes to the alignment.



8 Half-Mine: 0-Assignment to Individual Smoothing Results

3.1.6 Column DTW to Individual Smoothing

Now we consider individually time warping each A_c onto B_c, and then smoothing out the alignments afterwards. This is very similar to the previous problem. This approach first applies DTW to each sequence individually before smoothing. The pseudocode follows this paragraph. Please note the pseudocode is meant for this 2D example and can easily be modified for the 3D version with the inclusion of another loop along with the m for loop.

```
ColumnDTWwithIndividualSmoothing(A,B){
      /* greedily align each sequence */
      for(c=0; c<C; c++){
            f_c = DTW(A_c,B_c)
      }

      /* Apply Smoothing */
      f = smoothing(A,B,f)

      return f
}
```

9 Column DTW To Individual Smoothing Pseudocode

For this problem, Dynamic Time Warping is done differently. In traditional time warping, both elements in A and B can be skipped. In the Column Time Warping with Neighborhood Distortion Cost problem, only elements of B can be skipped, and the matching cost of every A element is calculated. In the example, sequence A is {5,6,5,5,12,5} and sequence B is {5,5,6,10,5,5}. The pseudocode following this paragraph is used.

```
/* Aligning two sequences X and Y */
DTW(X,Y){

      DTW = malloc(|X|,|Y|,{arrow,cost})

      //init border
      DTW[0][0].cost = 0
      /* cannot align any A element with nothing
      for n:1 to N
            DTW[n][0] = infinity
      /* the cumulative skip cost of skipping to B[m]*/
      for m:1 to M
            DTW[0][m].cost = m

      for n:1 to N{
            for m:1 to M{
                  DTW[n][m] = min{

             // skipping Y[m-1]
             (left, 1                    //skip cost
                    + DTW[n][m-1].cost  //previous cost
              ),

             // aligning X[n] to Y[m]
             (diag, DTW[n-1][m-1].cost  //previous cost
                    + cm(X[n],Y[m])      //matching cost
              ),

             // aligning X[n-1] and X[n] to Y[m]
             (up,   1                   //skip cost
                    + DTW[n-1][m]       //previous cost
                    + cm(X[n],BX[m])     //matching cost
              )
                  }

      }     }

      /* retrace for alignment */
      answer = {}
      while(n != 1 && m != 1){
            thisArrow = DTW[n][m].arrow
            answer.addToFront(thisArrow)
            if(thisArrow == up){
                  n = n-1
            }
            if(thisArrow == diag){
                  n = n-1
                  m = m-1
            }
            if(thisArrow == left){
                  m = m-1
      }     }
      return answer
}
```

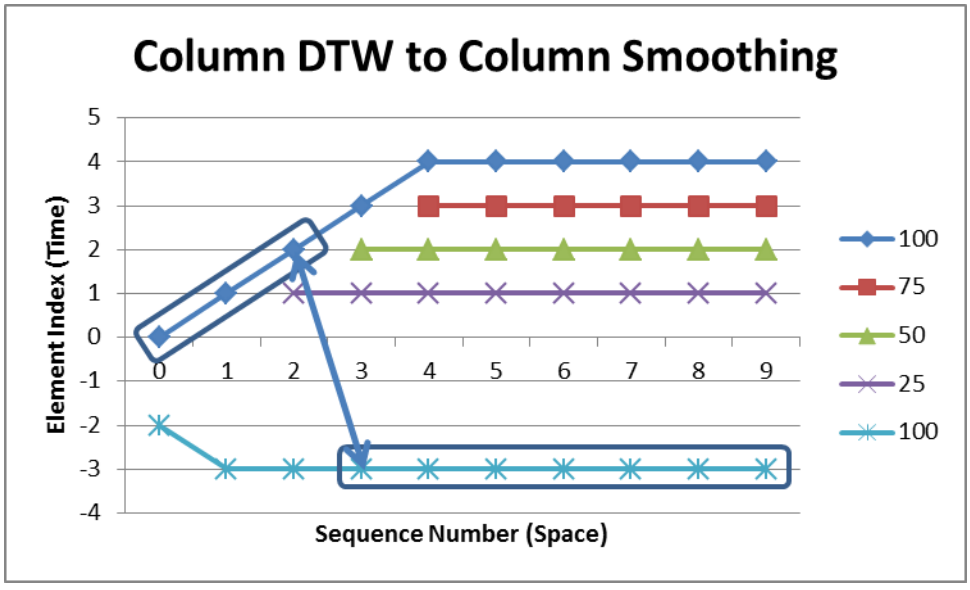
10 Column DTW Pseudocode

			Sequence B					
			5	5	6	10	5	5
		0	← 1	← 2	← 3	← 4	← 5	← 6
Sequence A	5	↑ ∞	↖ 0	↖ ← 1	↖ ← 2	↖ ← 3	↖ ← 4	↖ ← 5
	6	↑ ∞	↑ 2	↖ 1	↖ 1	← 2	← 3	← 4
	5	↑ ∞	↑ 3	↖ ↑ 2	↖ 2	← 3	↖ 2	↖ ← 3
	5	↑ ∞	↑ 4	↖ ↑ 3	↖ 3	← 4	↖ ↑ 3	↖ 2
	12	↑ ∞	↑ 12	↖ ↑ 11	↖ 9	↖ 5	← 6	← 7
	5	↑ ∞	↑ 13	↖ ↑ 12	↑ 11	↑ 11	↖ 5	↖ ← 6

11 Example Column Dynamic Time Warp

The Column DTW applied to the half mine will result in aligning each $A_c[0]$ to the closest line segment in B. The output alignment is the following: $A_0[0]$ to $B_0[0]$, $A_1[0]$ to $B_1[1]$, $A_2[0]$ to $B_2[2]$, $A_3[0]$ to $B_3[-3]$, $A_4[0]$ to $B_4[-3]$, and all other A[0] to B[-3].

When smoothing, no $A_c[0]$ can change its alignment to one above or one below because that could incur significant matching costs. This results in a local minimum of the alignment cost, but it is suboptimal.



12 Half-Mine: Column DTW to Individual Smoothing Results

3.1.7 Column DTW to Column Smoothing

Another approach to stop aligning error from Column DTW to Smoothing is to, during smoothing, consider new alignments for all the elements of a sequence. This reconsideration of aligning all elements in a sequence is effectively a DTW with modifications to account for the neighborhood distortion cost. This is described with the following pseudocode. Column Smoothing is done with DTW with minor edits to bring the skipping and neighborhood distortion costs into consideration. The function c_{nall} calculates the neighborhood distortion cost for one element. Note that this is for the 2D version of the problem, but minor edits will give the full 3D algorithm.

```
ColumnDTWtoColumnSmoothing(A,B){
      for(c=0; c<C; c++){
            f_c = DTW(A_c, B_c)
      }

      /* Column Smoothing */
      while(fPrev != f){
            for(c=0; c<C; c++){
                  f_c = contextDTW(A,B,f,c)
            }
            fPrev = f
      }
      return f
}
```

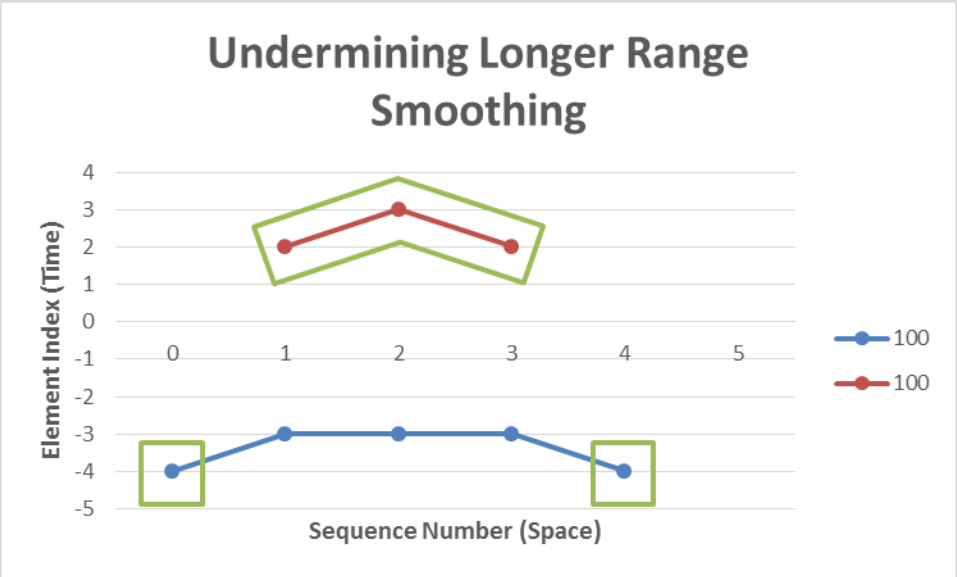
13 Column DTW to Column Smoothing Pseudocode

```
/* Align A_X with B_X with consideration to the current alignment f */
contextDTW(A,B,f,X){
      N = |A| ;  M = |A_N|
      DTW = malloc(N+1,M+1,{arrow,cost})
      /* init border */
      DTW[0][0].cost = 0
      for n:1 to N /* cannot align A elements with nothing */
            DTW[n][0].cost = infinity
      for m:1 to M /* skip cost of aligning A[1] to B[m]*/
            DTW[0][m].cost = m
      /* fill in the DTW table */
      for n:1 to N{
            for m:1 to M{
                  DTW[n][m] = min{

             // skipping Y[m-1]
             (left, 1 + DTW[n][m-1].cost ),

             // aligning A_X[n] to B_X[m]
             (diag,     DTW[n-1][m-1].cost
                        + c_m(A_X[n],B_X[m])
                        + c_nall(n,m,f,X)),

             // aligning A_X[n-1] and A_X[n] to B_X[m]
             (up,   1 + DTW[n-1][m]
                        + c_m(A_X[n],B_X[m])
                        + c_nall(n,m,f,X))
      }     }     }
      /* retrace for alignment */
      answer = {}
      while(n != 1 && m != 1){
            thisArrow = DTW[n][m].arrow
            answer.addToFront(thisArrow)
            if(thisArrow == up){
                  n = n-1
            }
            if(thisArrow == diag){
                  n = n-1
                  m = m-1
            }
            if(thisArrow == left){
                  m = m-1
      }     }
      return answer
}
/* calculate the neighborhood distortion cost */
C_nall(n,m,f,X){
      sum = 0
      for d: -1 to 1{
            if (d != 0) && (d + X >= 0) && (d + X < f.sequence_length){
                  sum += c_n(m-n, f_(d+X)[n])
      }     }
      return sum
}
```

14 Context DTW Pseudocode

The case for Column DTW to Column Smoothing will require the construction of a different problem instance. Consider a problem of aligning A and a new B. B will be constructed with the following sequences:

{-4:100},{-3:100, 2:100}, {-3:100, 3:100}, {-3:100, 2:100}, {-4:100}.



15 Half-Mine: Column DTW to Column Smoothing Results

With DTW, $A_0[0]$ and $A_4[0]$ align to $B_0[-4]$ and $B_4[-4]$, respectively. $A_1[0]$ and $A_3[0]$ will align with $B_1[2]$ and $B_3[2]$, respectively. $A_2[0]$ will align with $B_2[3]$. These alignments occur because they are the closest to A[0]. During the smoothing phase, we look at the elements in turn. $A_0[0]$ is stuck with its current alignment, because there are no other line segments to align with. $A_1[0]$ considers moving to $B_1[-3]$ only because that is the only other match with a line segment. However, since its neighbors $A_1[0]$ and $A_3[0]$ have alignment displacements of -4 and 2 (both have absolute values beyond 1 and have different polarity), the neighborhood distortion cost will remain constant regardless of a switch. $|2-A_1[0]|+|-4 – A_1[0]| = 6$ for both $f_1[0] = 1$ and -3. Therefore, $A_1[0]$ won't change. Next up for consideration is $A_2[0]$. Currently $f_2[0] =2$, and has neighbors $f_1[0]=1$ and $f_3[0]= 1$. The only other spot $A_2[0]$ can consider moving to is -3, but that will increase the neighborhood distortion cost, so the move is not made. The considerations have crossed the symmetry found in this alignment problem, so the smoothing process is continued with the remaining sequences and no realignments are made. The optimal alignment is to align with -4 for $A_0[0]$ and $A_4[0]$, and -3 for $A_1[0]$, $A_2[0]$, and $A_3[0]$.

3.1.8 Multiple Smoothing

One approach to stop the smoothing error is to use a lookahead during consideration of realigning each element. Considering the half mine problem, a lookahead of 3 would ensure that Column DTW to Multiple Smoothing would obtain the optimal alignment. However, the necessary lookahead is proportional to the problem size. Scaling the half mine to twice the dimensions would require twice the lookahead, so the necessary lookahead is O(n). The lookahead space would then become exponential and no longer leads to a P time algorithm. Following this

paragraph is the Column DTW to Multiple Smoothing pseudocode. The Multiple Smoothing is done with the function contextRDTW (context recursive dynamic time warp). It takes as input A, B, the current alignment, the column it is to work on, and the amount of lookahead space left. The All Alignments function produces every possible alignment between two sequences according to their lengths. Minor changes can make it suitable for 3D problems.

```
ColumnDTWtoMultipleSmoothing(A,B,lookahead){
      for(c=0; c<C; c++){
            f_c = DTW(A_c, B_c)
      }

      while(fPrev != f){
            for(c=0; c<C; c++){
                  f_c = contextRDTW(A,B,f,c,lookahead)
            }
            fPrev = f
      }
}
```

16 Column DTW to Multiple Smoothing Pseudocode

```
contextRDTW(A,B,f,c,lookahead){
      if(lookahead == 0){
            return contextDTW(A,B,f,c)
      }

      every_possible_alignment = allAlignmnts(||A_c||,||B_c||)
      minCost = infinity
      fMin = null
      for fNew in every_possible_alignment{
            possible_next_f = contextRDTW(A,B,fnew,c+1,lookahead-1)
            if( c(A,B,possible_next_f) < minCost){
                  fMin = possible_next_f
                  minCost = c(A,B,possible_next_f)
            }
      }
      return fMin
}
```

17 Recursive Context DTW

```
allAlignments(N,M){
      answers = {}

      //catch the 2X2 case
      if(N==2) && (M==2){
            return {left:up, diag, up:left}
      }

      //catch the wall
      if(N == 1){
            for(m=1; m<M; m++){
                  answer = left:answer
            }
            return answer
      }

      //catch other wall
      if(M == 1){
            for(n=1; n<N; n++){
                  answer = up:answer
            }
            return answer
      }

      //first align left
      foreach fNew in allAlignments(N,M-1){
            add left:fNew to answer
      }

      //first align diag
      foreach fNew in allAlignments(N-1,M-1){
            add diag:fNew to answer
      }

      //first align up
      foreach fNew in allAlignments(N-1,M){
            add up:fNew to answer
      }

      return answer
}
```

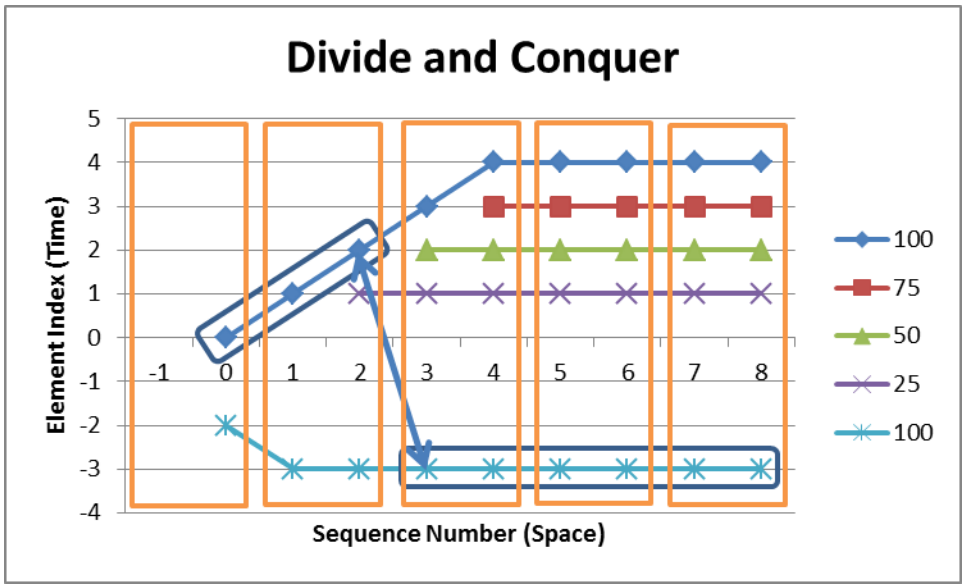
18 All Alignments Pseudocode

3.1.9 Divide and Conquer

The divide and conquer method is a common technique for dividing up an algorithm into smaller pieces and then combining them together to form an answer for the complete problem. With divide and conquer, splitting the problem up such that the sequences are divided won't work, as each sequence has the Warping Restriction. This restriction keeps elements in a subsequence of

A from aligning with B subsequences reserved for other A subsequences. This can keep the algorithm from finding the optimal alignment as this restricted alignment could be optimal.

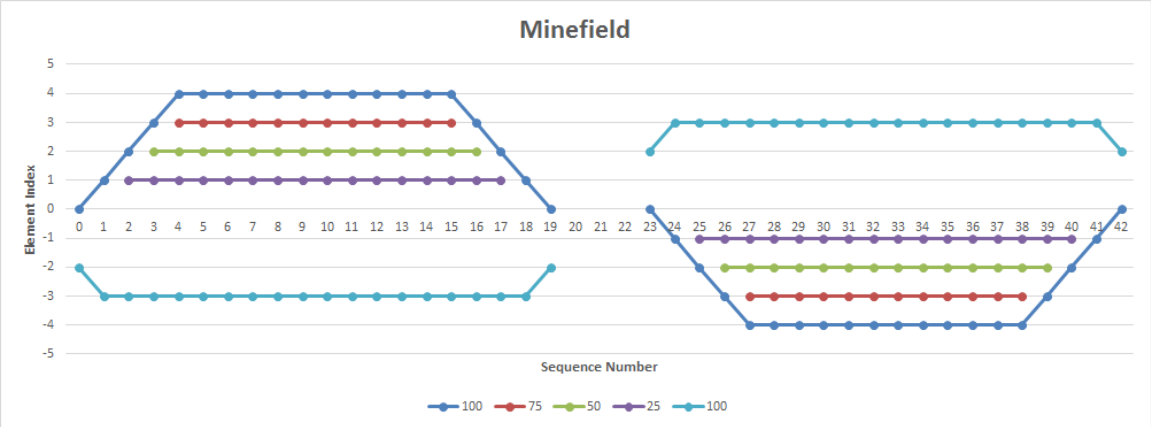
Dividing the set of sequences by dividing the array into smaller groups of sequences and then combining them together would be more feasible. However, consider dividing the problem into sequence pairs, aligning those pairs, and combining them together. With this approach applied to our half mine with an extra blank sequence pair at the left (we call these A_{-1} and B_{-1}) and take away one A and B sequence from the right (still resulting in 10 sequences in A to align with 10 sequences in B). Pairing off gives the pairs A_{-1} and A_0, A_1 and A_2, A_3 and A_4, and the remaining, identical pairs on the right. Using this approach, we would get the same alignment as the Column DTW to Individual Smoothing approach, which is suboptimal.



19 Half-Mine: Divide And Conquer Results

3.2 Minefield ensures no optimizations

Now we consider the creation of a 'mine' that undermines the mechanics of approaches to the problem to consistently produce suboptimal results. A mine is created with mirroring a half mine on the right side to create a full mine. A trivial number of mines can then be laid to the right, with two blank sequences in A and B separating these mines. This collection of mines can be referred to a minefield.



20 Minefield

3.2.1 Forcing Smoothing from Outside Mines Inwards

With one mine, doing a Column DTW to Column Smoothing with starting the smoothing from the inside (along the optimal alignment) could smooth the suboptimal alignments to the optimal with Column Smoothing However, with a trivial number of mines in a minefield, the inside of any mine cannot be determined before run time. This forces the smoothing process to start from outside the mines and move inward.

3.2.2 Unknowable Optimal Side

Looking at the single mine, one approach could be to align all elements of every sequence to $B_c[0]$, and then smooth outward. This would cause the smoothing to align $A_c[0]$ to the optimal alignment. In a minefield, with mines all centered on the same $A_c[0]$, but with some mines flipped over the $A_c[0]$ axis, some mines will have their optimal alignment on the top. Therefore, this technique cannot work for minefields.

4 Conclusion

This thesis has shown evidence of the Column Time Warp with Neighborhood Distortion Cost problem not residing within the set of P problems. This is not a proof, but it does suggest that if there is a polynomial-time algorithm, it will need to be based on principles other than iterative greedy scans or column wise divide-and-conquer. Only a full proof can definitively say which problem hardness class holds this problem. The question of whether or not all problems of NP are problems of P is still an open question in the computational community, so no full proof of this kind has ever been produced (Fortnow 2009). More work needs to be done to prove the Column Time Warp with Neighborhood Distortion Cost problem's relation to the class of P problems.

5 Bibliography

Elias, I. (2003). Settling the intractability of multiple alignment (pp. 352-363). Springer Berlin Heidelberg.

Keysers, D., & Unger, W. (2003). Elastic image matching is np-complete. Pattern Recognition Letters, 24(1-3), 2-3.

Fortnow, L. (2009, September). The status of the p versus np problem. Communications of the ACM, 52(9), 78-86.

Gusfield, D. (2007). Core string edits, alignments, and dynamic programming. In Algorithms on strings, trees, and sequences; computer science and computational biology (p. 216). New York, NY: Cambridge University Press.

Mardziel, P. (2004). Improved two-dimensional warping. Electronic Projects - Worcester Polytechnic Institute.

Appendix

6 Appendix: Planar 3-SAT Reduction Attempt

A popular proof of NP-COMPLETENESS is a reduction from 3-SAT (Garey 1979). This appendix shows an attempt at a reduction, but lacks an inversion gate to make the 3-SAT reduction possible. The reduction is from the Column Time Warp with Neighborhood Distortion Cost problem as defined in **Error! Reference source not found. Error! Reference source not found.** on page **Error! Bookmark not defined.**.

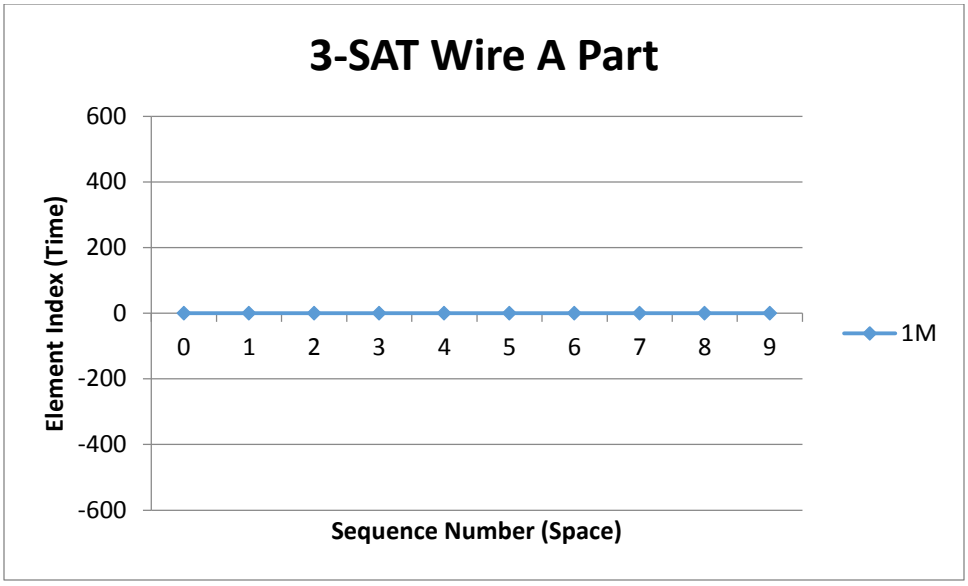
The reduction here will follow closely with the reduction shown in the paper by Keysers and Unger (Keysers 2003). The construction of a Planar 3-SAT problem instance within a Column Time Warping with Neighborhood Distortion Cost problem takes the approach of a bipartite graph, with one independent set for the variables, and the other set for the clauses. There is an edge between a variable node and an edge node if the variable is used within the clause. The edges are instantiated with wires to hold either a 1 state or a 0 state. These wires run between variable gates and clause gates. The only restriction on the placement of the independent sets is they cannot occupy the same sequences, leaving the two dimensions of the matrix to lay out the Planar 3-SAT. The wires and gates will be constructed with parts in the input A, and corresponding parts in input B. The state of each wire and gate is represented by the alignment of its part A in input A onto its part B in input B.

To describe each of the wires and gates, only two dimensions are shown at a time in the figures. The figures show a series of sequences along one dimension. Each wire and gate will incur a minimal alignment cost as part of its operation, so an account is made of the minimal alignment cost. An indicator of no satisfiability is when no alignment exists under the minimal alignment cost account. However, inspecting the alignment of each sequence (in polynomial time) is still necessary. There can be improper alignments under the minimal alignment cost account.

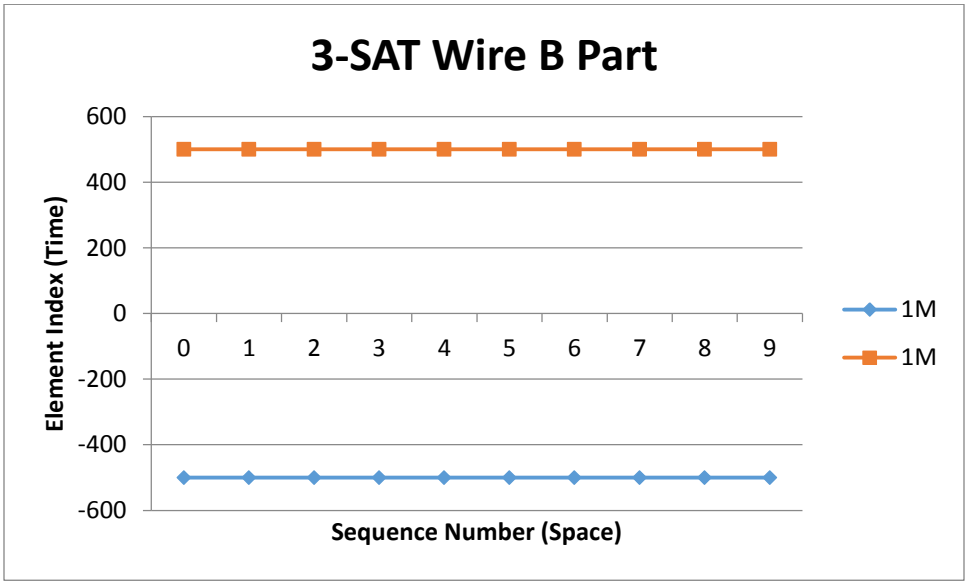
Wires

The wires are to transfer a signal, either a true or false value. This two state component depicts its signal according to the alignment of its part A onto its part B. The wire's part A and part B are depicted below. Unless specified, each element has a value of zero. Each sequence along the wire holds either a true state or a false state. The true state is when the 1 million valued element in part A aligns with the upper 1 million valued element in part B. False is when part A aligns downward. To reduce the matching cost, part A must align every sequences 1M element to one of the 1M elements in B, creating the bistability. To propagate this signal through the whole wire, each sequence in the wire must align the same as its neighbor. This is accomplished through the neighborhood distortion cost. Any difference in alignment between neighbors incurs a distortion cost, forcing the neighbors to have the same alignment, propagating the signal. While 1M is used for illustration, the real cost would be a function of the problem size. It would rather have a function value of $100*N^6$. The distance between the 1M elements in part B would also need to be a function of the problem size; $100*N^2$ would work.

While the wire is constructed, an account of the minimal alignment cost is taken. Both minimal alignments have the same cost, so for each sequence a constant amount is added to the minimal alignment cost account.

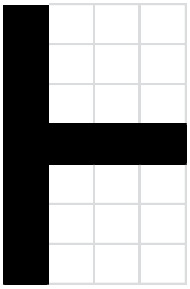


1 3-SAT Wire A Part



2 3-SAT Wire B Part

The wires are needed to carry the true/false value from the variable component to each of the clauses according to the bipartite graph. To carry the truth value to different clauses, the wire must fork. To accomplish this, the wire forks in a T shape. With the figures for parts A and B making the top of the T, another wire would stem out perpendicularly (see figure on next page). It travels along a different dimension of the matrix.



3 3-SAT Wire Fork

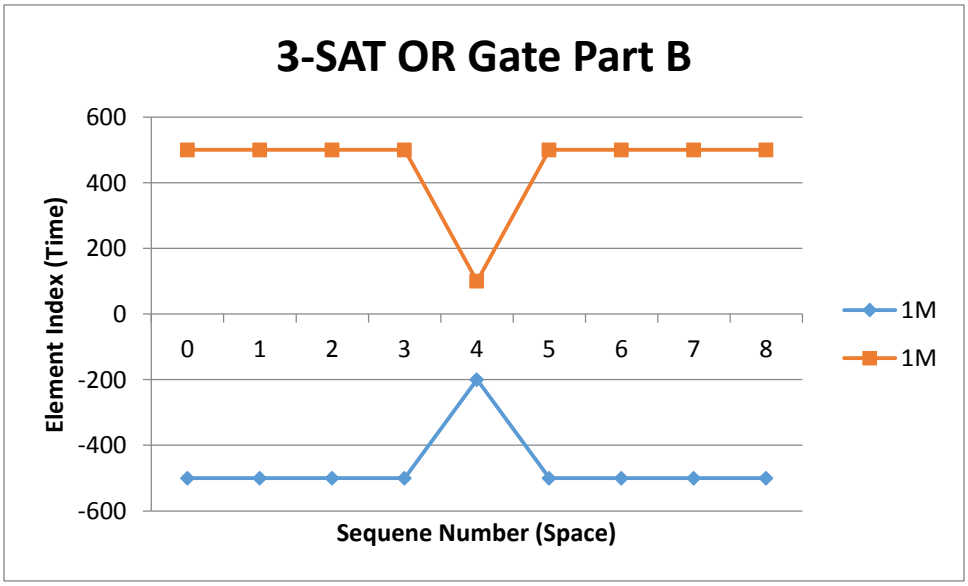
Each of the black boxes is a sequence used for the wire. The two dimensions of the T are the two dimensions of the matrix.

NOR Gates

NOR gates are used to construct the clause gates. Any digital logic can be designed with NOR gates. The output wire of a clause gate can be set to only allow for holding a true value by eliminating the series of 1M values that make false values possible in part B. This forces the clause gate to find a satisfactory assignment for its variables, or it will incur large distortion costs to signal the clause unsatisfied.

OR Gate

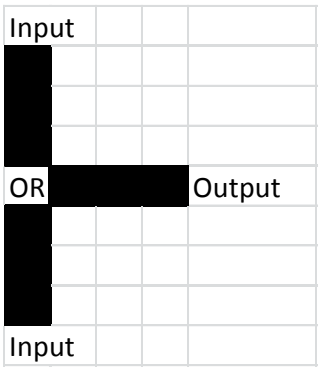
An OR gate gives a value of true when either of its inputs are true. Its construction can be thought of as a crimping the wire's part B, as depicted below. OR gates have the same part A as wires. The inputs are the wires at the side of the OR gate, and the output wire is perpendicular (not seen in figure). As with every other sequence in the wire, the middle sequence's part A has to align its 1M element either upwards or downwards for a true or false value, respectively.



4 3-SAT OR Gate Part B

There are four input signal combinations that can be represented as three: both positive, both negative, and conflicting. When both inputs are positive, the OR gate sequence aligns upwards to reduce neighborhood distortion cost. When both inputs are negative, the OR gate sequence aligns downwards to reduce alignment distortion cost. When the inputs conflict (one positive, one negative) the OR gate sequence will have equal distortion cost for either alignment, but to reduce skipping costs will align upwards.

The output of the OR sequence is another wire branching out perpendicular to the inputs (see figure below). The output wire is susceptible to distortion costs between the input wires, but that won't change the output wire's true/false value from matching the OR gate. When both inputs are the same, the output will match the inputs. When the inputs conflict, the output will have equal distortion cost from them for both its true and false alignments. However, the OR gate will be the tie breaker, making the output wire match its own value.



5 3-SAT OR Gate

Each of the black boxes and the OR represent a sequence used in the OR gate. The two dimensions of the figure are the two dimensions of the matrix that holds the sequences. The two inputs and OR sequence are depicted in the previous figure. The output runs perpendicularly to the inputs.

As with the wires, the OR gate incurs a minimal alignment cost. The highest alignment cost is when the inputs oppose each other and the output is one. This amount is added to the minimal alignment cost account, once each OR gate.

Inverter

This is the missing component of the Planar 3-SAT reduction. The nature of the distortion cost lends itself towards making the values more uniform, helping the wires propagate values and the OR gate function, but threatens the creation of an inverter.

The variable gates depend on the Inverter. If both the variable and inverted variable are used in the clauses, then an inverter connects the variable wires and inverts the signal for the inverted variable wires.