

# Distance Functions for Sequence Data and Time Series



---

Based on original slides by Prof. Dimitrios Gunopulos and Prof. Christos Faloutsos with some slides from tutorials by Prof. Eamonn Keogh and Dr. Michalis Vlachos. Excellent tutorials (and not only) about Time Series can be found there:  
<http://www.cs.ucr.edu/~eamonn/tutorials.html>



# Time Series Databases

---

- A time series is a sequence of real numbers, representing the measurements of a real variable at equal time intervals
  - Stock prices
  - Volume of sales over time
  - Daily temperature readings
  - ECG data
- A time series database is a large collection of time series

# Time Series Problems

(from a database perspective)

- The Similarity Problem

$$X = x_1, x_2, \dots, x_n \text{ and } Y = y_1, y_2, \dots, y_n$$

- Define and compute  $\text{Sim}(X, Y)$ 
  - E.g. do stocks X and Y have similar movements?
- Retrieve efficiently similar time series (Indexing for Similarity Queries)

# Metric Distances



---

- What properties should a similarity distance have?
- $D(A,B) = D(B,A)$  *Symmetry*
- $D(A,A) = 0$  *Constancy of Self-Similarity*
- $D(A,B) \geq 0$  *Positivity*
- $D(A,B) \leq D(A,C) + D(B,C)$  *Triangular Inequality*



# Euclidean Similarity Measure

---

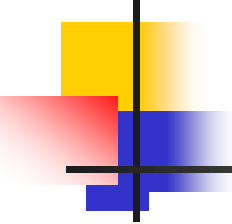
- View each sequence as a point in n-dimensional Euclidean space (n = length of each sequence)
- Define (dis-)similarity between sequences X and Y as

$$L_p = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}$$


p=1 Manhattan distance

p=2 Euclidean distance

# Subsequence matching in TS

- 
- We have a database of time series DB with time series  $S_1, S_2, \dots, S_N$  of potentially different lengths.
  - The user specifies a query  $Q$  of length  $\text{Len}(Q)$ .
  - We want to find quickly the time series  $S_i$  that have a segment that has distance (similarity) with the query less or equal to  $\varepsilon$ , i.e.,  
$$D ( S_i[k : k + \text{Len}(Q)-1], Q ) \leq \varepsilon.$$
  - One approach: Linear scan, read every time series for all possible offsets and compare against the query  $Q$ .
  - Another solution: use an index!😊

# ST-index

- 
- Assume also that there is a minimum length of each query  $w$ .

Approach:

- scan every sequence  $S_i$  and create segments of size  $w$ . i.e.,  $S[1, w]$ ,  $S[2, w+1]$ ,  $S[3, w+3]$ , etc..  
For  $S_i$ , we will create  $\text{Len}(S_i) - w + 1$  segments.
- Store every segment as an individual time series in an index (R-tree) by mapping it to a point.

**Problem with this approach: too many points are created.**

Observation: two consecutive segments differ in only up to 2 values! So, the mapped point will be close...

# ST-trails

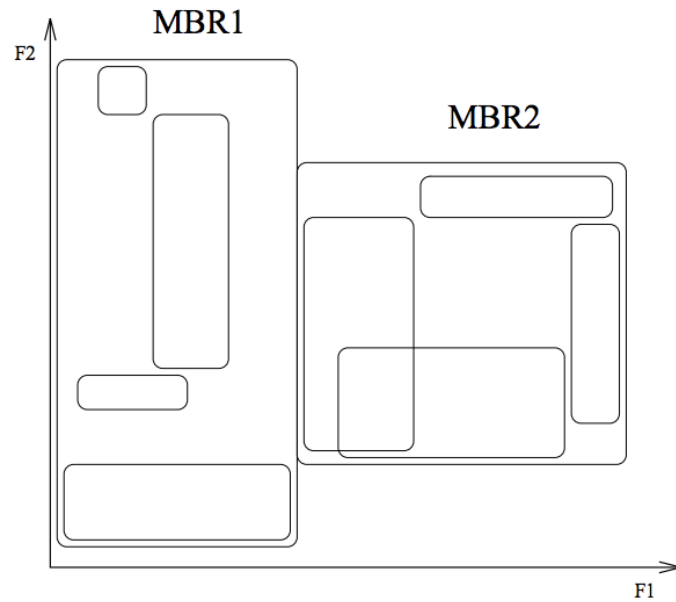
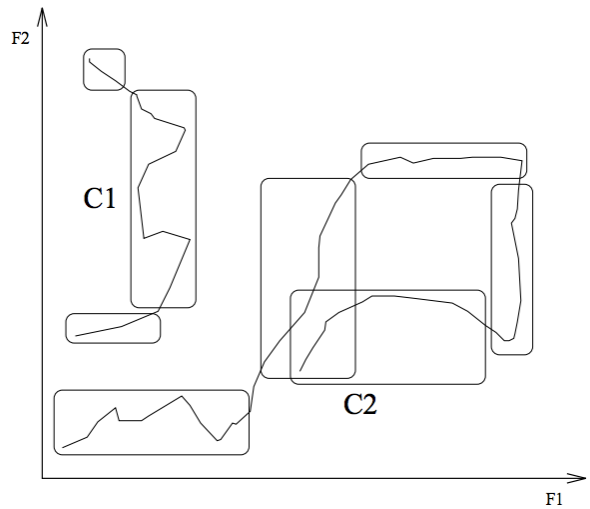


---

- Trail: The sequence of the points of a time series form a trail.
- Split the trail in parts and compute the MBR of each part.
- Store the MBR of the parts in the index! and use each MBR a representative of all the points inside!
- Instant of storing hundreds of points in the index we store a few MBRS!



# ST-index



# Query



---

- If the query length is  $w$ , just map the query to a point and run range query with radius  $\varepsilon$ , find candidate sub-trails and go and check the actual segments.
- What if the query is larger?
  - Break the query in segments of length  $w$ , run each segment individually and merge the results!

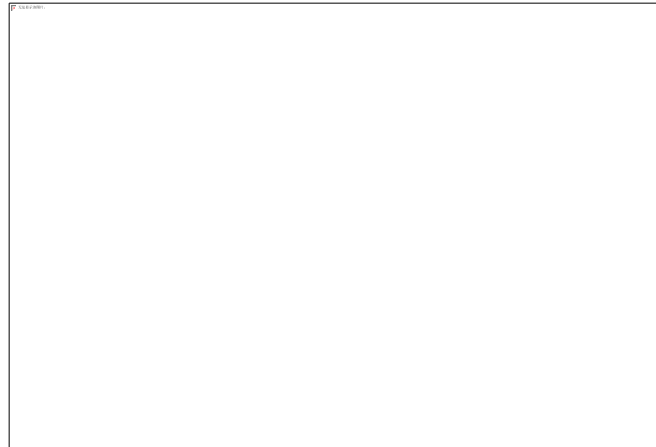
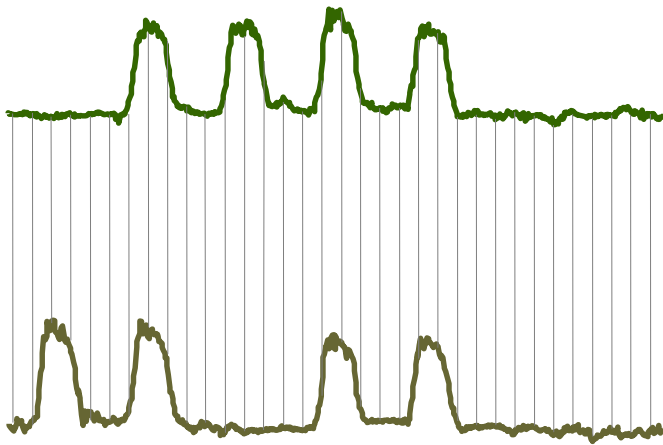
# Other distance functions



---

- Euclidian distance is easy to compute and index...
- But has a number of problems.
- Cannot handle changes in the speed.
- Cannot handle outliers
- ....

# Example



Euclidean distance vs DTW



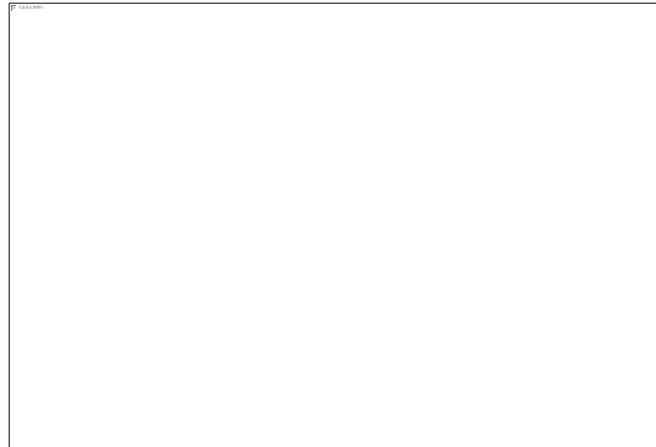
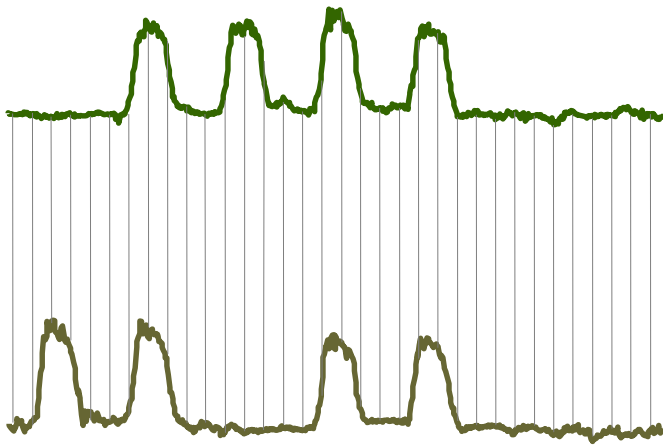
# Dynamic Time Warping

[Berndt, Clifford, 1994]

---

- Allows acceleration-deceleration of signals along the time dimension
- Basic idea
  - Consider  $X = x_1, x_2, \dots, x_n$ , and  $Y = y_1, y_2, \dots, y_m$
  - We are allowed to extend each sequence by repeating elements
  - Euclidean distance now calculated between the extended sequences  $X'$  and  $Y'$
  - Matrix  $M$ , where  $m_{ij} = d(x_i, y_j)$

# Example



Euclidean distance vs DTW



# Formulation

---

- Let  $D(i, j)$  refer to the dynamic time warping distance between the subsequences ( $L_1$  based distance)

$x_1, x_2, \dots, x_i$

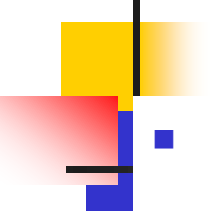
$y_1, y_2, \dots, y_j$

$$D(i, j) = |x_i - y_j| + \min \{ \begin{array}{l} D(i-1, j), \\ D(i-1, j-1), \\ D(i, j-1) \end{array} \}$$

How to solve it?

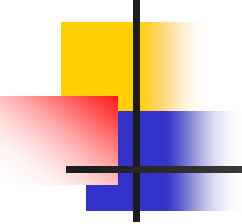
Dynamic programming!

# Dynamic Programming

- 
- Dynamic Programming is an algorithm design technique for **optimization problems**: often minimizing or maximizing.
  - **Like** divide and conquer, DP solves problems by combining solutions to subproblems.
  - **Unlike** divide and conquer, subproblems are not independent.
    - Subproblems may share subsubproblems,
    - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
  - DP reduces computation by
    - Solving subproblems in a bottom-up fashion.
    - Storing solution to a subproblem the first time it is solved.
    - Looking up the solution when subproblem is encountered again.
  - Key: determine structure of optimal solutions



# Steps in Dynamic Programming

- 
- 
1. Characterize structure of an optimal solution.
  2. Define value of optimal solution recursively.
  3. Compute optimal solution values either **top-down** with caching or **bottom-up** in a table.
  4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

# DTW variations



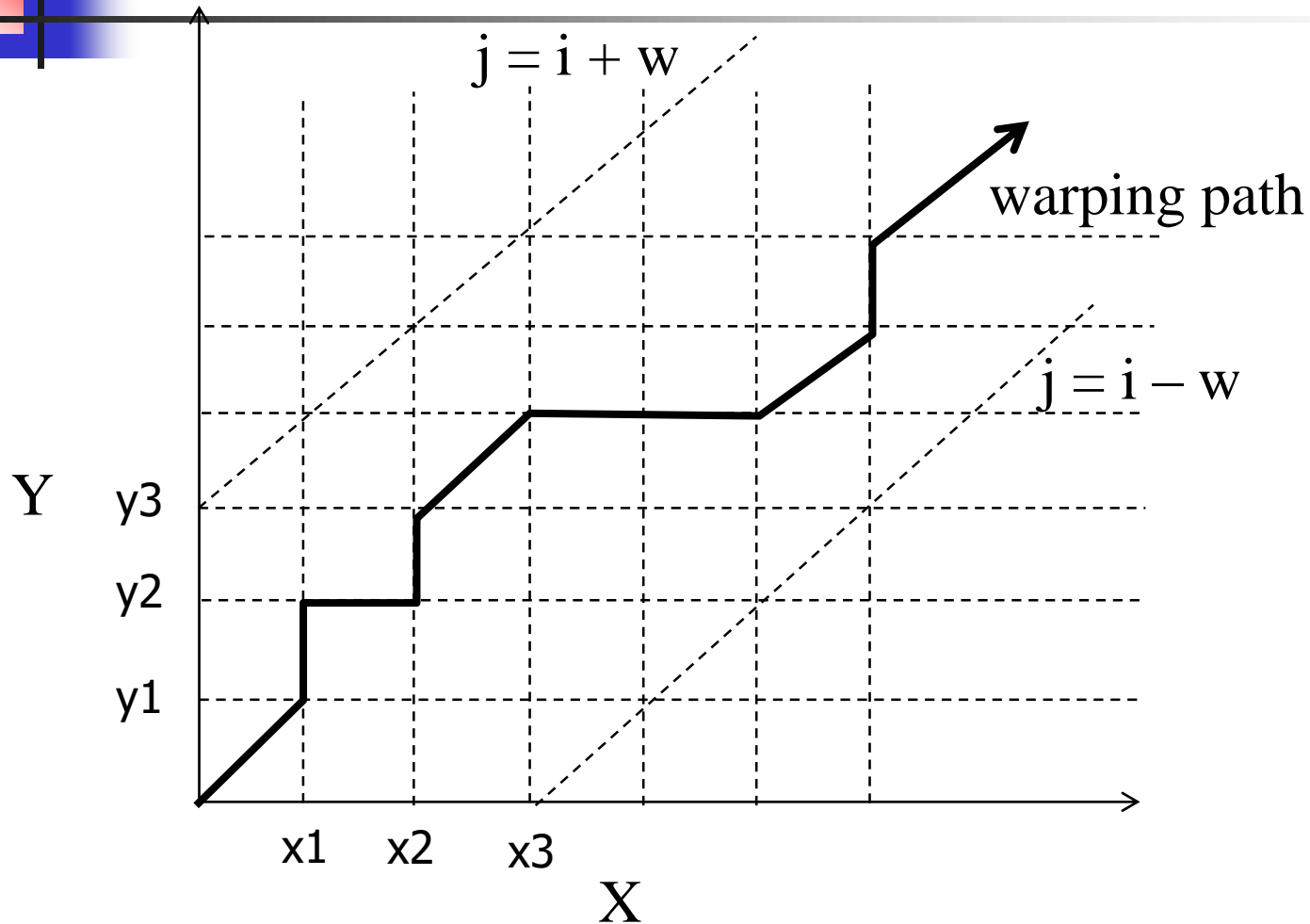
---

- If we use Euclidean distance to compute the difference between two elements  $x_i$  and  $y_j$  then we can use:

$$D(i, j) = (x_i - y_j)^2 + \min \{ \begin{array}{l} D(i - 1, j), \\ D(i - 1, j - 1), \\ D(i, j - 1) \end{array} \}$$

and take the square root of  $D(n, m)$  at the end.

# Dynamic Time Warping



# Restrictions on Warping Paths



---

- Monotonicity
  - Path should not go down or to the left
- Continuity
  - No elements may be skipped in a sequence
- Warping Window

$$|i - j| \leq w$$



# Solution by Dynamic Programming

---

- Basic implementation =  $O(nm)$  where  $n, m$  are the lengths of the sequences
  - will have to solve the problem for each  $(i, j)$  pair
- If warping window is specified, then  $O(nw)$ 
  - Only solve for the  $(i, j)$  pairs where  $|i - j| \leq w$

# How to speed up the calculation of DTW?

## What is Lower Bounding Measure ?

- a dimensionality reduction technique

## Why using Lower Bounding Measure ?

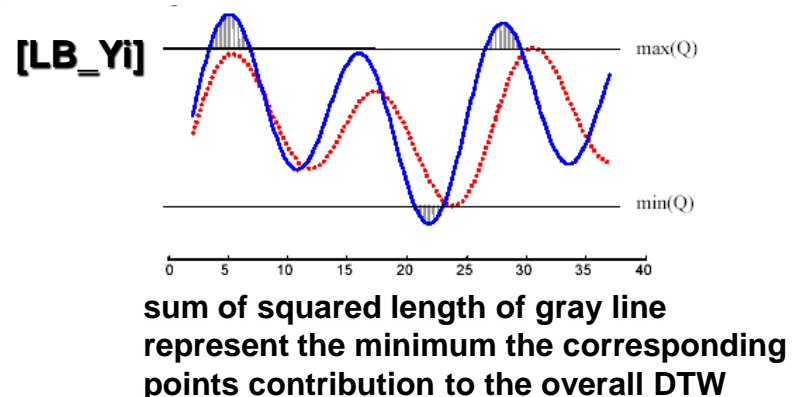
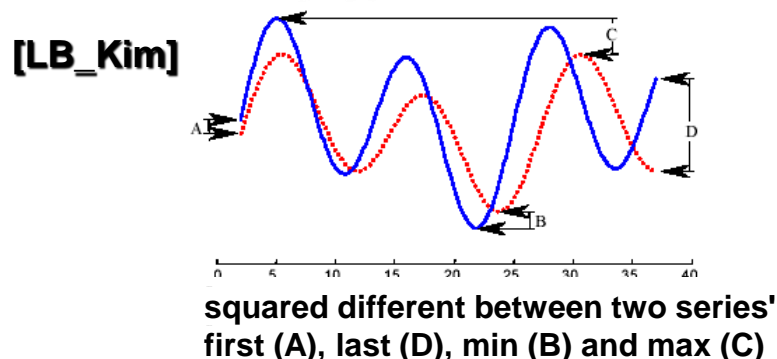
- DTW is either heavily I/O bound or very demanding in terms of CPU time.
- a fast lower bounding function can address this problem by erasing sequences that could not possibly be a best match.

## How to define a good Lower Bounding Measure ?

A good lower bounds function should basically match two criteria

- must be fast to compute
- must produces a relatively tight lower bounds which means that it can more tightly approximates the true DTW distance

## Two existing type of Lower bounding measure



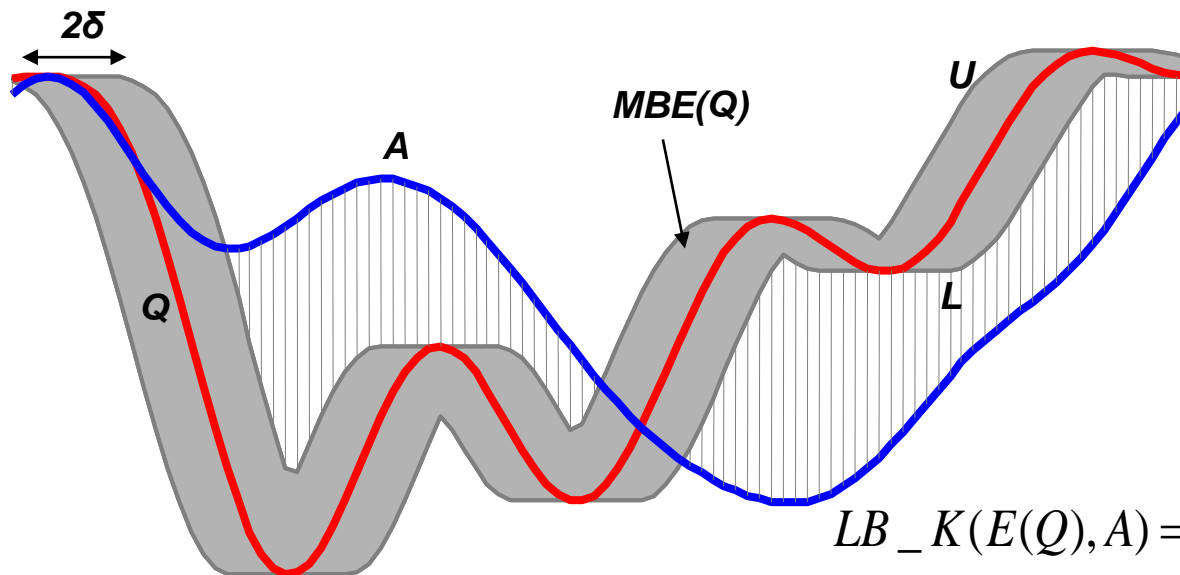
# Lower Bounding the Dynamic Time Warping

Recent approaches use the Minimum Bounding Envelope for bounding the **constrained** DTW

- Create a  $\delta$  Envelope of the query  $Q$  ( $U, L$ )
- Calculate distance between MBE of  $Q$  and any sequence  $A$
- One can show that:  $D(\text{MBE}(Q)_\delta, A) < \text{DTW}(Q, A)$
- $\delta$  is the constraint

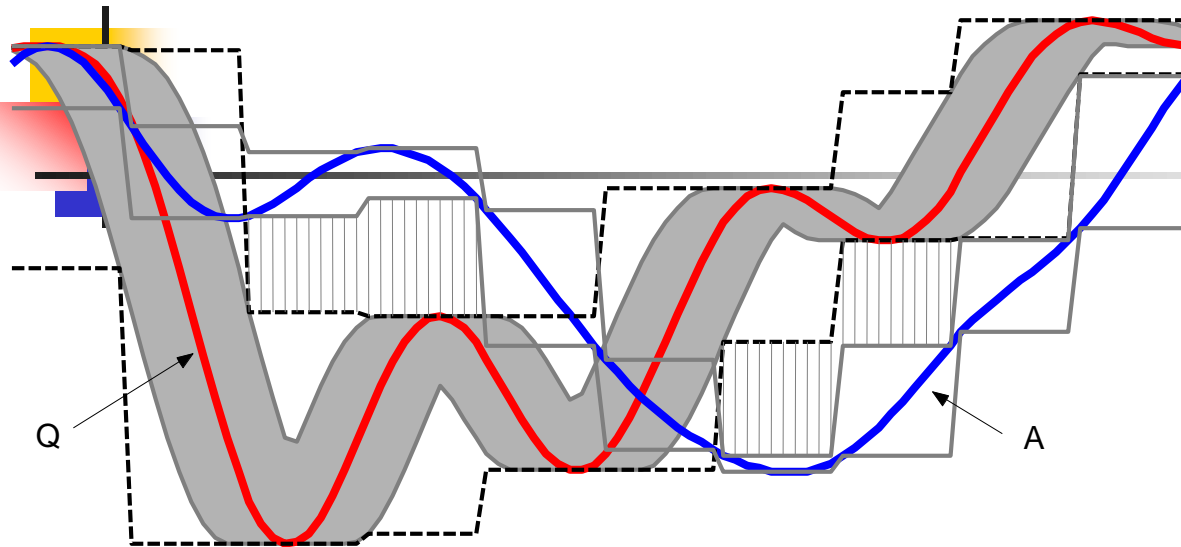
$$U[i] = \max_{-\delta \leq r \leq \delta} (Q[i + r])$$

$$L[i] = \min_{-\delta \leq r \leq \delta} (Q[i + r])$$

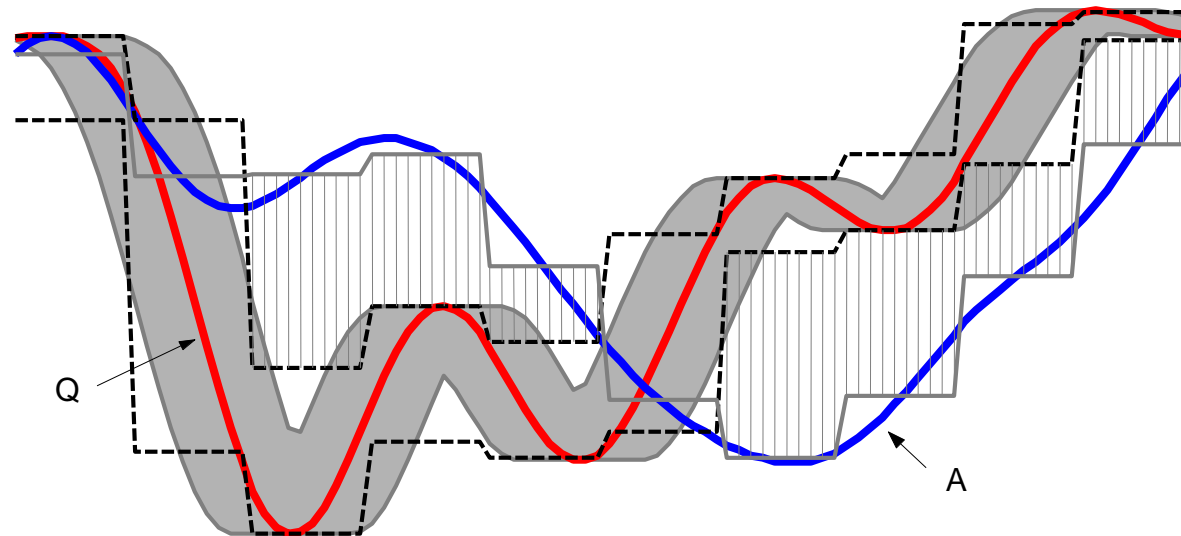


$$LB\_K(E(Q), A) = \sqrt[p]{\sum_{i=1}^N \begin{cases} |A[i] - U[i]|^p & \text{if } A[i] > U[i] \\ |A[i] - L[i]|^p & \text{if } A[i] < L[i] \\ 0 & \text{otherwise} \end{cases}}$$

# Lower Bounding the Dynamic Time Warping



*LB by Keogh  
approximate MBE and  
sequence using MBRs  
 $LB = 13.84$*

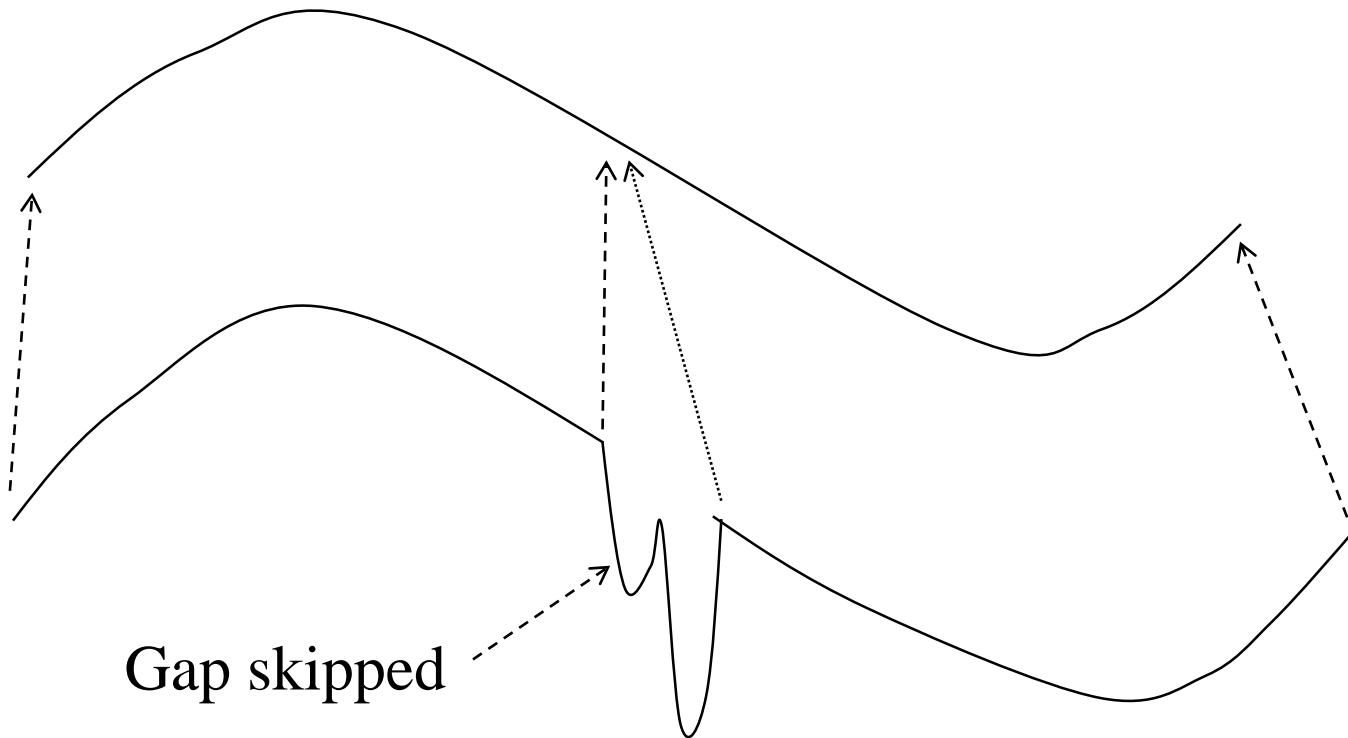


*LB by Zhu and  
Shasha approximate  
MBE and sequence  
using PAA  
 $LB = 25.41$*

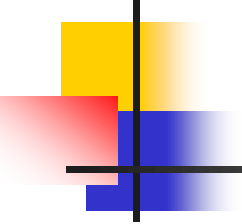


# Longest Common Subsequence Measures

(Allowing for Gaps in Sequences)



# Basic LCS Idea

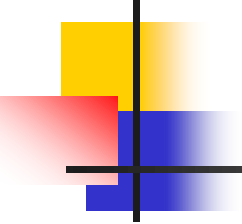


---

X = 3, **2**, **5**, **7**, 4, 8, **10**, 7  
Y = **2**, **5**, 4, **7**, 3, **10**, 8, 6  
LCS = **2, 5, 7, 10**

$$\text{Sim}(X,Y) = |\text{LCS}| \quad \text{or} \quad \text{Sim}(X,Y) = |\text{LCS}| / n$$

# LCSS distance for Time Series

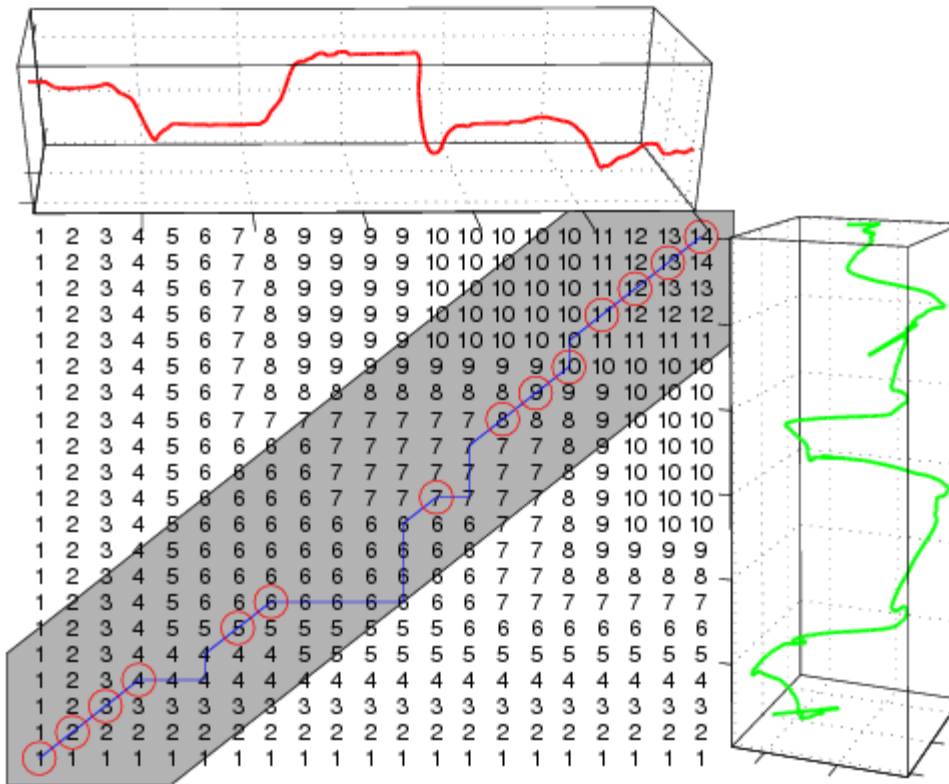


We can extend the idea of LCS to Time Series using a threshold  $\varepsilon$  for matching.

The distance for two subsequences:

$x_1, x_2, \dots, x_i$  and  $y_1, y_2, \dots, y_j$  is:

$$\text{LCSS}[i,j] = \begin{cases} 0, & \text{if } i=0 \text{ or } j=0 \\ 1+\text{LCSS}[i-1, j-1] & \text{if } |x_i - y_j| < \varepsilon \\ \max(\text{LCSS}[i-1, j], \text{LCSS}[i, j-1]) & \text{otherwise} \end{cases}$$



# Edit Distance for Strings



---

- Given two strings sequences we define the distance between the sequences as the minimum number of edit operation that need to be performed to transform one sequence to the other. Edit operation are insertion, deletion and substitution of single characters.
- This is also called **Levenshtein distance**

# Example: DNA



---

- S: a set of DNA sequences.
- Q: DNA sequence
  - with a small deviation from the database match.
    - within  $\delta |Q|$ , for  $\delta \leq 15\%$ .
  - can be large (up to 10,000 nucleotides).

# The Edit Distance [Levenshtein et al.1966]

- Measures how dissimilar two strings are.
- $ED(A, B)$  = minimum number of operations needed to transform  $A$  into  $B$ .
- Operations = [insertion, deletion, substitution].
- Example:

■  $A = ATC$

and  $B = ACTG$   
 $A = A - T C$

$B = A C T G$

$$ED(A, B) = 2$$


# Computing Edit distance

- We can compute edit distance for two strings  $x_1x_2x_3\dots x_i$  and  $y_1y_2y_3\dots y_j$  using the following function:

$$ED(i,j) = \begin{cases} ED(i-1, j-1) & \text{if } x_i = y_j \\ \min \begin{aligned} &(ED(i-1, j) + 1, \\ &ED(i, j-1) + 1, \\ &ED(i-1, j-1) + 1) \end{aligned} & \text{if } x_i \neq y_j \end{cases}$$



# DP algorithm



```
int LevenshteinDistance(char s[1..n], char t[1..m])
```

```
    int d[0..n, 0..m]
```

```
    for i from 0 to n d[i, 0] := i
```

```
    for j from 1 to m d[0, j] := j
```

```
    for i from 1 to n
```

```
        for j from 1 to m
```

```
            if s[i] = t[j] then cost := 0 else cost := 1
```

```
            d[i, j] := minimum(
```

```
                d[i-1, j] + 1, // deletion
```

```
                d[i, j-1] + 1, // insertion
```

```
                d[i-1, j-1] + cost // substitution )
```

```
    return d[n, m]
```

# The Edit Distance



Initialization:

---

		A	C	T	G
	0	1	2	3	4
A	1				
T	2				
C	3				

# The Edit Distance



First column:

- Match = 0
- In/del/sub = 1

		A	C	T	G
	0	1	2	3	4
A	1	0			
T	2	1			
C	3	2			

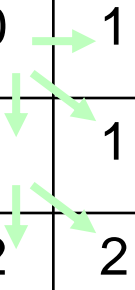
# The Edit Distance



■ Second column:

---

		A	C	T	G
	0	1	2	3	4
A	1	0	1		
T	2	1	1		
C	3	2	2		

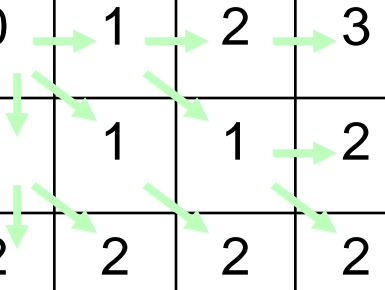


# The Edit Distance



■ Final Matrix:

		A	C	T	G
	0	1	2	3	4
A	1	0	1	2	3
T	2	1	1	1	2
C	3	2	2	2	2



# The Edit Distance

Alignment Path:

$A = A - T C$   
 $B = A C T G$

		A	C	T	G
	0	1	2	3	4
A	1	0	1	2	3
T	2	1	1	1	2
C	3	2	2	2	2

# The Edit Distance: Subsequence matching

## ■ Initialization:

		A	C	T	G
	0	0	0	0	0
A	1				
T	2				
C	3				

# The Edit Distance: Subsequence matching

■ Final Matrix:

		A	C	T	G
	0	0	0	0	0
A	1	0	1	1	1
T	2	1	1	1	2
C	3	2	2	2	2



# The Edit Distance: Subsequence matching

■ One path:

		A	C	T	G
	0	0	0	0	0
A	1	0	1	1	1
T	2	1	1	1	2
C	3	2	2	2	2

$A = A T C$

$B = A C T G$

# Smith-Waterman

[Smith&Waterman et al. 1981]

- Is a similarity measure used for local alignment:
  - Match can be a subsequence of the query sequence.
- Define three penalties:
  - match, mismatch, gap.
  - Scoring parameters are defined by the user.
- Example:
  - $A = \text{ATC}$  and  $B = \text{TATTCG}$
  - match = 2, mismatch = -1, gap = -1.

# Smith-Waterman



Initialization:

---

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0						
T	0						
C	0						
A	0						

# Smith-Waterman



■ First column:

---

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	-1					
T	0	2					
C	0	1					
A	0	0					

# Smith-Waterman



■ First column:

---

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	0					
T	0	2					
C	0	1					
A	0	0					

# Smith-Waterman

■ Second column:

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	0	2				
T	0	2	1				
C	0	1	1				
A	0	0	3				

# Smith-Waterman

■ Final Matrix:

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	0	2	1	0	0	0
T	0	2	1	2	3	2	1
C	0	1	1	1	2	5	4
A	0	0	3	2	1	4	4

# Smith-Waterman

- Detect highest value:

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	0	2	1	0	0	0
T	0	2	1	2	3	2	1
C	0	1	1	1	2	5	4
A	0	0	3	2	1	4	4



# Smith-Waterman

Alignment Path:

$A =$  A - T C A

$B =$  T A T T C G

		T	A	T	T	C	G
	0	0	0	0	0	0	0
A	0	0	2	1	0	0	0
T	0	2	1	2	3	2	1
C	0	1	1	1	2	5	4
A	0	0	3	2	1	4	4

# Metric properties and Indexing



---

- Edit distance is a metric! So, we can use metric indexes
- DTW and LCSS are not metrics... we have to use specialized indexes