Isolated Word Recognition Using Dynamic Time Warping

Haikya Corp., USA
and
Speech Systems Pvt. Ltd., Bangalore

Background

- Words are the basic units of speech humans most widely understand
- Speech recognition being a hard problem, we simplify it by focusing on these units in isolation

- Seems clumsy
- Yet, useful enough that it was used in one of the first practical commercial applications: *Dragon Dictate*
  - Users - had - to - speak - with – pauses – between - words
- Useful at least for people with impairments that interfered with their writing or typing ability
- The concept could also be useful in languages with no standard keyboard, or a complicated alphabet
  - *E.g.* Indian languages, or Chinese

ASR: A Simplified Problem

- How can we recognize speech?
  - *i.e.* convert spoken input into text?

- Computers do not “understand” anything, let alone speech
  - We have to reduce the speech recognition problem to a *mechanical* process that can be applied to the input speech
  - *i.e.* we need an *algorithm* that can be implemented as a computer program

- Constrain the problem to simplify it:
  - What if we know the input *vocabulary* beforehand, the set of words that may be spoken
  - What if we know that any given input consists exactly of one word from that vocabulary

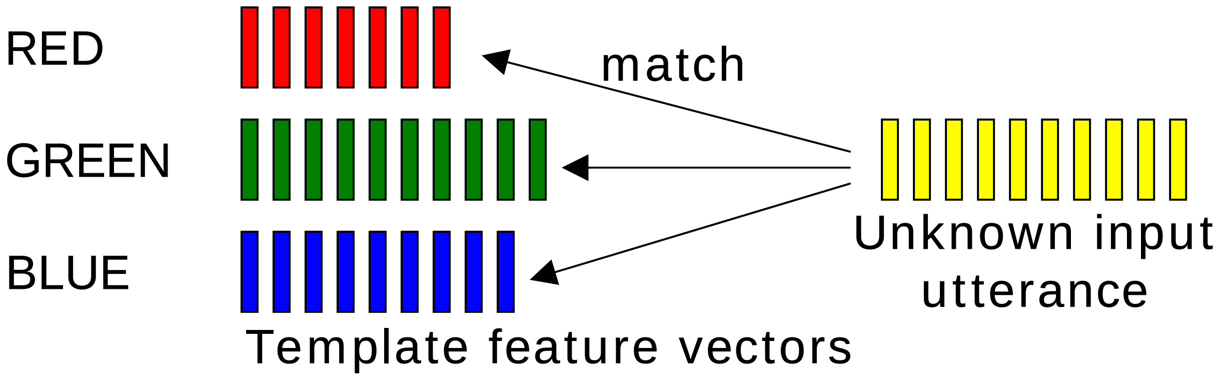
- Incidentally, a chunk of input speech to be decoded is usually called an *utterance*
- So, let us consider trying to decode utterances, each consisting of a *single* word from a *known* vocabulary

Template Matching

- The first successful solutions to this problem used a technique called *template matching*
  - Since the entire vocabulary is known, simply pre-record an example of each word
  - When an utterance is to be decoded, compare it to all the pre-recorded examples to determine the closest match
- The pre-recorded examples are the *templates*
- *Matching* is the process of comparing an input utterance to every template

- We have already noted that the raw speech stream has to be converted to *frames* of *feature vectors*
  - Thus, the matching has to occur between the feature vector sequences of the input utterance and each of the templates

Template Matching (contd.)

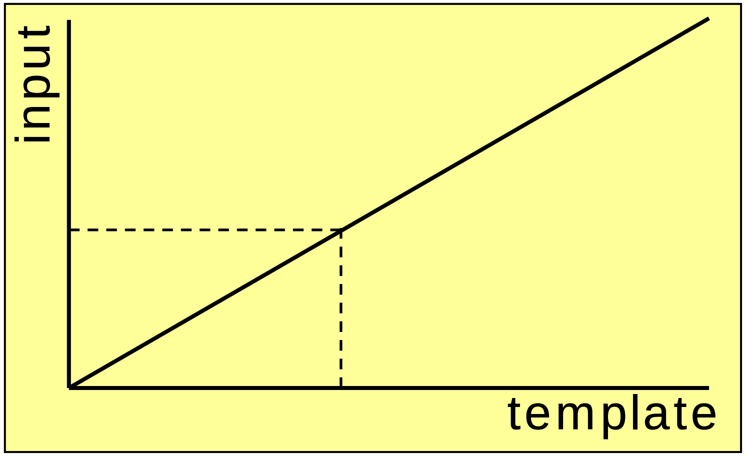


Template Matching Issues

- Speaking rates can be different from those in the template
  - => Number of feature vectors may be different
  - Cannot simply do a frame-by-frame comparison
- Speaking rate can even vary *within* a word
  - *E.g.* a syllable may be stressed and slowed down for emphasis
  - So, which frames do we match against which?
- As already noted, speaker differences cause feature vectors to differ
  - Even the same speaker will not be able to reproduce speech exactly
  - What comparison metric is appropriate, then?
- The beginning and end of speech may not be known precisely
  - How do we deal with silence regions surrounding the speech?

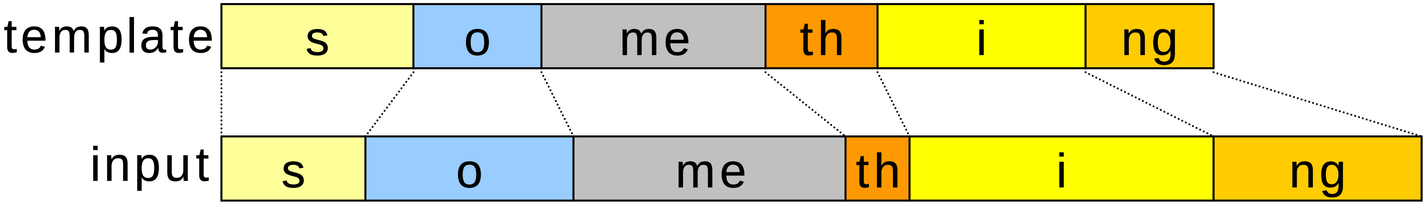
Template Matching Issues

- One possibility for dealing with different lengths of input and template: normalize the lengths



Template longer than input

- Obviously this does not deal with varying speech rates *within* an utterance; *e.g.* elongated or shrunk phonemes:



Template Matching: Dynamic Time Warp

- Clearly, it is necessary to *dynamically* adjust the durations of various *segments* within the input and template speech
  - How to optimally (and automatically!) *align* the template to the input speech to obtain the best possible match?
  - Brute force? Try all possible segmentations and alignments?
    - Computationally far too expensive!
    - Besides, precise segment boundaries may not even be known
- An efficient algorithm for finding an optimal alignment exists: *Dynamic Time Warping* or DTW
- DTW in speech recognition is essentially based on *dynamic programming* (DP)
  - DP is a well known optimization algorithm in Operations Research

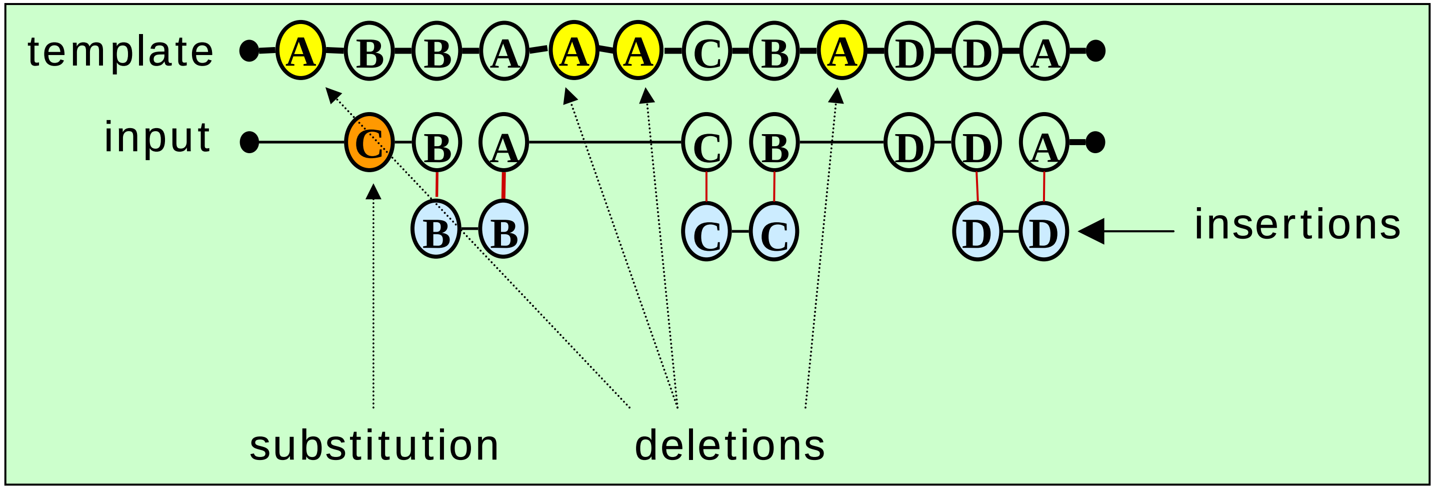
DP: String Matching Example

- First, consider a similar problem, but visually easier to grasp: Aligning two text strings to find how similar they are
  - *E.g.* spelling errors detected by google:
    - Type query string “Ravi Shanker” into google
    - Google responds: Did you mean “Ravi *Shankar*”?
    - How does it know this?

- “Ravi Shankar” is its template (among many others), and “Ravi Shanker” is the input
  - By measuring the dissimilarity between the two, it could guess that perhaps the template was the intended input

String Matching Example

- The example contained just a substitution error (*e* for *a*)
- In general, there can be three types of string errors:
  - *Substitution*: a template letter has been changed in the input
  - *Insertion*: a spurious letter is introduced in the input
  - *Deletion*: a template letter is missing from the input
- Hypothetical example:
  - Template: ABBAAACBADDA
  - Input: CBBBACCCBDDDDA

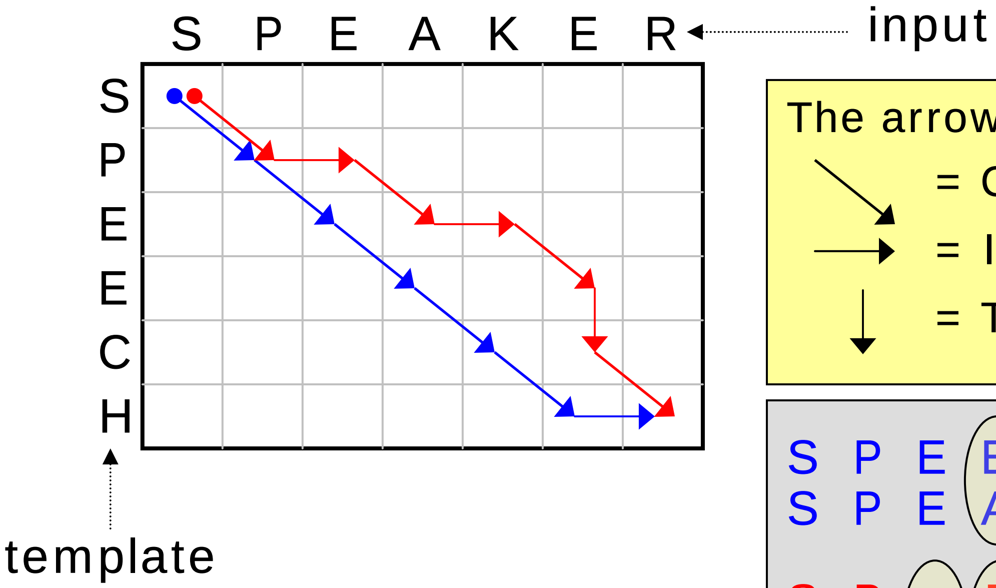


String Matching Problem

- Given two arbitrary strings, find the *minimum edit distance* between the two:
  - Edit distance = the *minimum number of editing operations* needed to convert one into the other
  - Editing operations: substitutions, insertions, deletions
  - Often, the distance is also called *cost*
- This minimum distance is a measure of the *dissimilarity* between the two strings
- Also called the *Levenshtein distance*
- Remember there is always Wikipedia to learn more!

String Edit Distance Computation

- Measuring edit distance is best visualized as a 2-D diagram of the template being *aligned* or *warped* to best match the input
  - Two possible alignments of template to input are shown, in blue and red

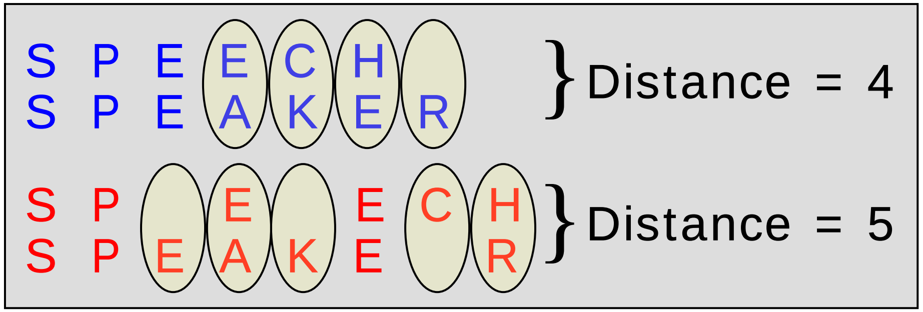


The arrow directions have specific meaning:

↘ = Correct or substituted

→ = Input character inserted

↓ = Template character deleted


S P E E C H
S P E A K E R
} Distance = 4

S P E E C H
S P E A K E R
} Distance = 5

- Wanted: *MINIMUM* edit distance: *i.e. inherent* dissimilarity
  - Over *all possible paths* from corner to corner

Minimum String Edit Distance

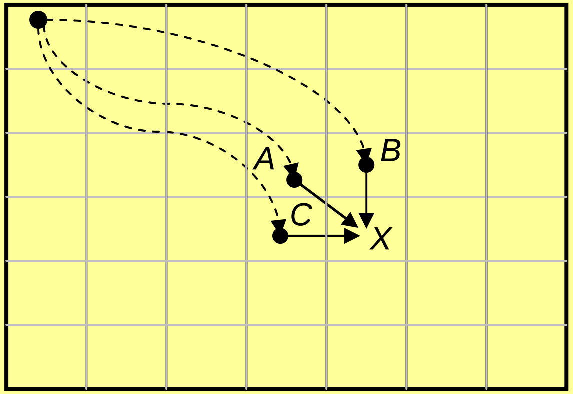
- This is an example of a *search* problem, since we need to search among all possible paths for the best one

- First possibility: Brute force search
  - Exhaustive search through all possible paths from top-left to bottom-right, and choose path with minimum cost
  - But, computationally intractable; exponentially many paths!
    - (*Exercise*: Exactly how many different paths are there?)
  - (A path is a connected sequence made up of the three types of arrows: diagonal, vertical and horizontal steps)

- Solution: *Dynamic Programming* (DP)
  - Find optimal (minimum cost) path by utilizing (re-using) optimal sub-paths
  - *Crucial to virtually all major speech recognition systems*

Minimum String Edit Distance: DP

- *Central idea*: formulate optimal path to any intermediate point X in the matrix *in terms of optimal paths of all its immediate predecessors*
  - Let M_X = Min. path cost from origin to any pt. X in matrix
  - Say, A, B and C are all of X's predecessors
  - Assume M_A, M_B and M_C are known (shown by dotted lines)
  - Then, $M_X = \min(M_A+AX, M_B+BX, M_C+CX)$
    - AX = edit distance for diagonal transition = 0 if the aligned letters are same, 1 if not)
    - BX = edit distance for vertical transition = 1 (deletion)
    - CX = edit distance for horizontal transition = 1 (insertion)



- Hence, start from the origin, and compute min. path cost for every matrix entry, proceeding from top-left to bottom right corner
  - Min. edit distance between the two strings = value at bottom right corner

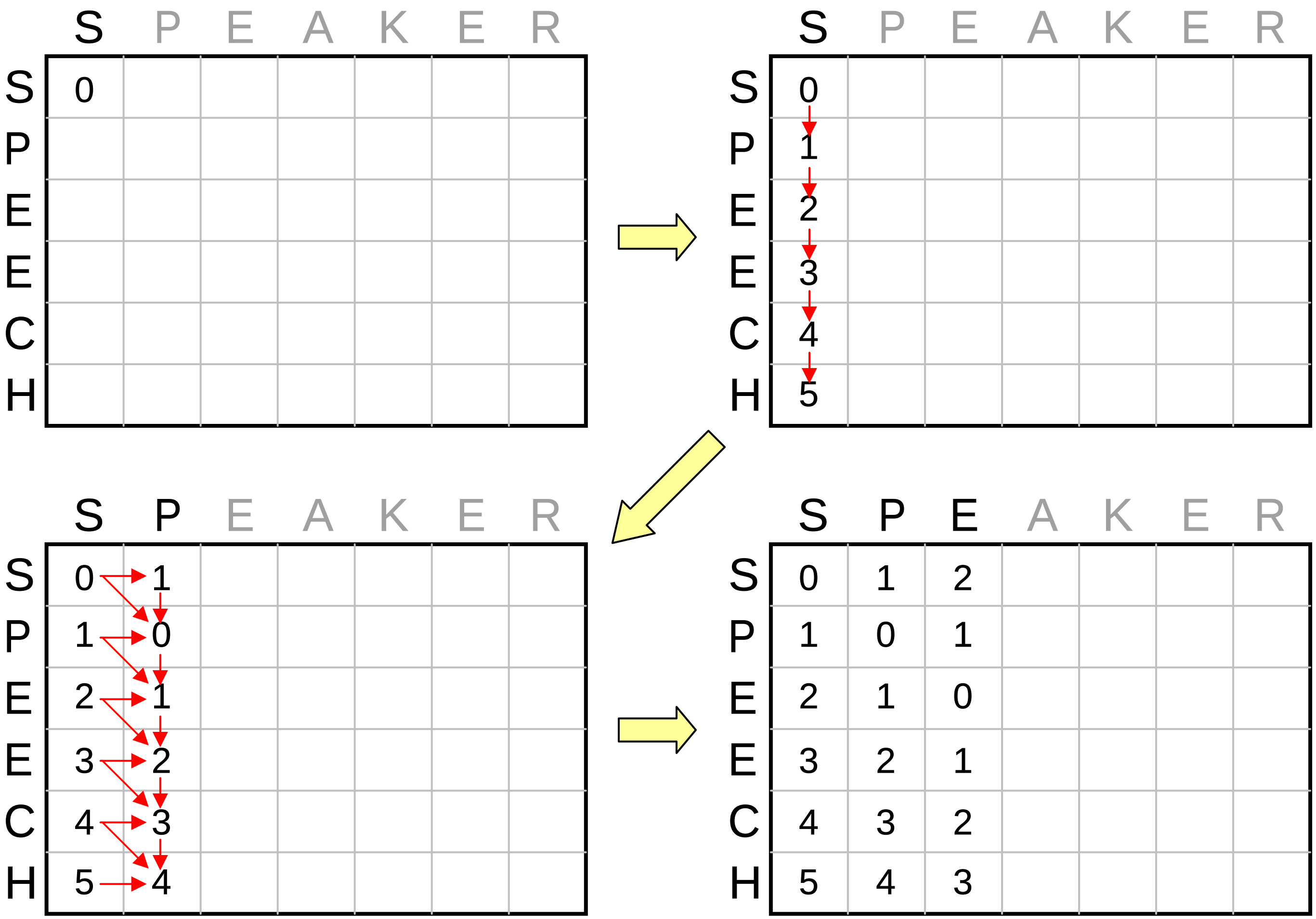
DP (Contd.)

- Although the matrix entries can be filled in any order (for any entry, we only need to ensure that all its predecessors are evaluated), it helps to proceed methodically, once column (*i.e.* one input character) at a time:
  - Consider each input character, one at a time
  - Fill out min. edit distance for that entire column before moving on to next input character
  - Forces us to examine every unit of input (in this case, every character) one at a time
    - Allows each input character to be processed *as it becomes available*

- Its usefulness will be seen later (*time synchronous search*)

DP Example

- □ First, initialize top left corner, aligning the first letters



	S	P	E	A	K	E	R
S	0						
P							
E							
E							
C							
H							

	S	P	E	A	K	E	R
S	0						
P	1						
E	2						
E	3						
C	4						
H	5						

	S	P	E	A	K	E	R
S	0	1					
P	1	0					
E	2	1					
E	3	2					
C	4	3					
H	5	4					

	S	P	E	A	K	E	R
S	0	1	2				
P	1	0	1				
E	2	1	0				
E	3	2	1				
C	4	3	2				
H	5	4	3				

DP Example (contd.)

	S	P	E	A	K	E	R
S	0	1	2	3			
P	1	0	1	2			
E	2	1	0	1			
E	3	2	1	1			
C	4	3	2	2			
H	5	4	3	3			

	S	P	E	A	K	E	R
S	0	1	2	3	4		
P	1	0	1	2	3		
E	2	1	0	1	2		
E	3	2	1	1	2		
C	4	3	2	2	2		
H	5	4	3	3	3		

	S	P	E	A	K	E	R
S	0	1	2	3	4	5	
P	1	0	1	2	3	4	
E	2	1	0	1	2	3	
E	3	2	1	1	2	2	
C	4	3	2	2	2	3	
H	5	4	3	3	3	3	

	S	P	E	A	K	E	R
S	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
E	2	1	0	1	2	3	4
E	3	2	1	1	2	2	3
C	4	3	2	2	2	3	3
H	5	4	3	3	3	3	4

- ☐ Min. edit distance (SPEECH, SPEAKER) = 4
  - One possible min. distance alignment is shown in blue

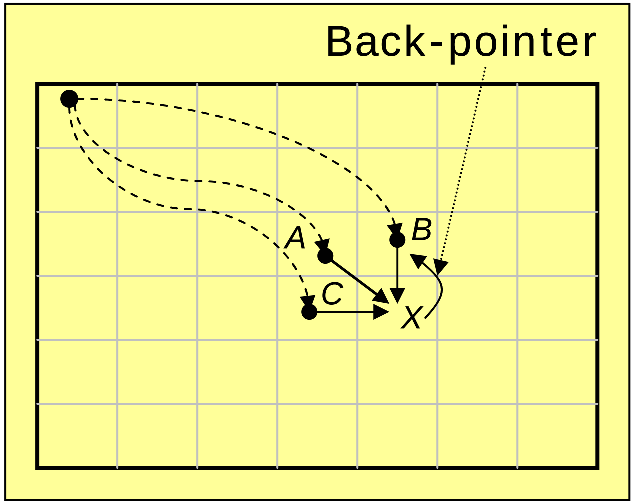
A Little Diversion: Algorithm Bug

- □ The above description and example has a small bug. What is it?
- □ *Hint*: Consider template and input: *AROUND* and *ROUND*
  - ■ What does the algorithm find as the minimum edit distance?
  - ■ What should it be?
- □ How can the algorithm be modified to fix the bug?

DP: Finding the Best Alignment

- The algorithm so far only finds the *cost*, not the alignment itself
- How do we find the actual path that minimizes edit distance?
  - There may be multiple such paths, any one path will suffice

- To determine the alignment, we modify the algorithm as follows
- Whenever a cell *X* is filled in, we maintain a *back-pointer* from *X* to its predecessor cell that led to the best score for *X*
  - Recall $M_X = \min(M_A + AX, M_B + BX, M_C + CX)$
  - So, if $M_B + BX$ happens to be the minimum we create a back-pointer *X->B*
  - If there are ties, break them arbitrarily
- Thus, every cell has a single back-pointer

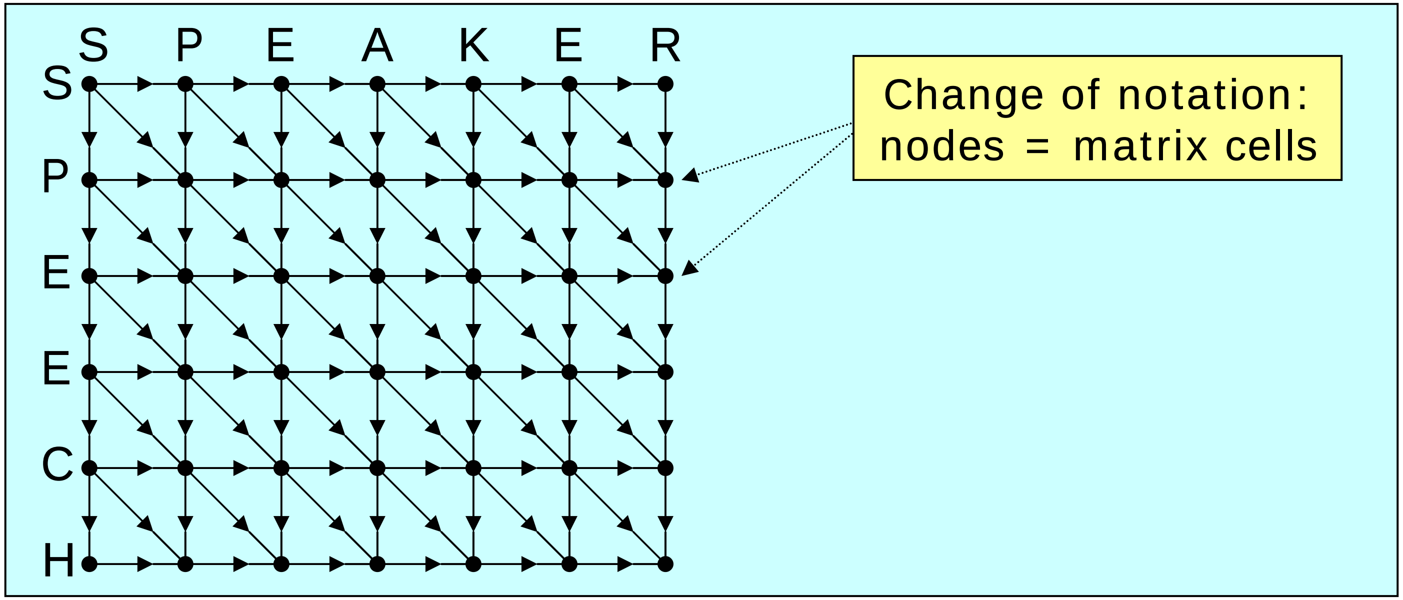


- At the end, we *trace back* from the final cell to the origin, using the back-pointers, to obtain the best alignment

Finding the Best Alignment: Example

DP Trellis

- The 2-D matrix, with all possible transitions filled in, is called the *search trellis*
- Search trellis for the previous example:



Change of notation:
nodes = matrix cells

- DP does *not* require that transitions be limited to the three types used in the example
- The primary requirement is that the optimal path be computable recursively, based on a node's predecessors' optimal sub-paths

DP Trellis (contd.)

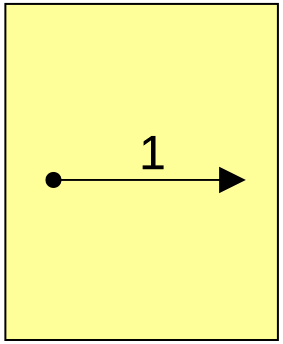
- *The search trellis is arguably one of the most crucial concepts in modern day decoders*!
  - We will encounter this again and again
- Just about any decoding problem is usually cast in terms of such a trellis
- It is then a matter of searching through the trellis for the best path

Elements of the Search Trellis

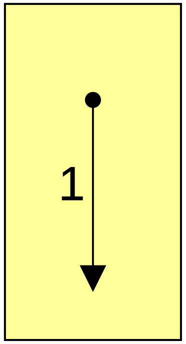
- Nodes represent the cells of the DP matrix
- Edges are the allowed transitions according to some *model* of the problem
  - In string matching we allow substitutions, insertions, and deletions
- Every edge optionally has an *edge cost* for taking that edge
- Every node optionally has a *local node cost* for aligning the particular input entry to the particular template entry
  - The node and edge costs used depend on the application and model
- The DP algorithm, at every node, maintains a *path cost* for the *best path* from the origin to that node
  - In string matching, this cost is the *substring* minimum edit distance
  - Path costs are computed by accumulating local node and edge costs according to the recursive formulation already seen (minimizing cost)
- One may also use a *similarity* measure, instead of *dissimilarity*
  - In this case DP algorithm should try to *maximize* the total path score

Edge and Node Costs for String Match

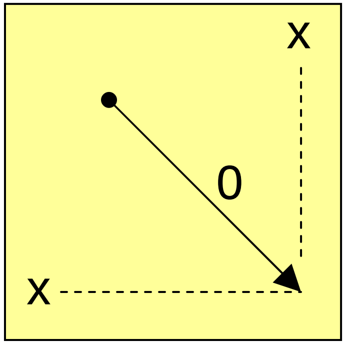
- □ Edge costs:



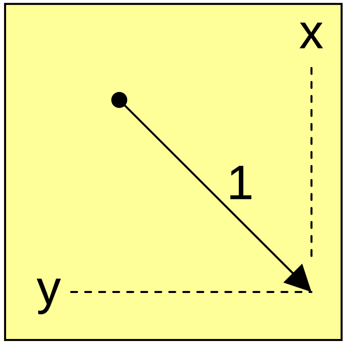
insertion



deletion



correct



substitution

- □ Local node costs: None

Computational Complexity of DP

- Computational cost ~

  No. of nodes x No. of edges entering each node
- For string matching, this is:

  String-length(template) x String-length(input) x 3
  - (Compare to exponential cost of brute force search!)

- Memory cost for string matching?
  - No of nodes (String-length(template) x String-length(input))?
  - Actually, we don't need to store the entire trellis if all we want is the min. edit distance (*i.e.* not the alignment; no back pointers)
  - Since each column depends only on the previous, we only need storage for 2 columns of the matrix
    - The current column being computed and the previous column
  - Actually, in most cases a column can be updated *in-place*
    - Memory requirement is reduced to just one column

Application of DP String Matching

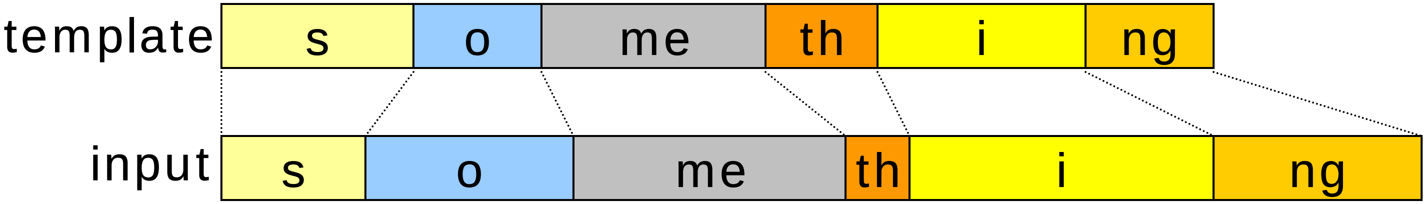
- How might google recognize “Ravi Shanker” as a mistyped version of “Ravi Shankar”?
- One hypothetical heuristic:
  - Google maintains a list of *highly popular* query strings
    - These are the templates; “Ravi Shankar” is one of them
  - When a user types in a query, it is string-matched to every template, using DP
  - If a template matches exactly, there is no spelling error
  - If a *single* template has one overall error (edit distance = 1), google can ask *Did you mean “…”*
  - Otherwise, do nothing
    - Either multiple templates have edit distance = 1, or
    - Minimum edit distance > 1

  - Here, we see the implicit introduction of a *confidence measure*
    - A decision based on the *value* of the min. edit distance (see later)

DTW: DP for Speech Template Matching

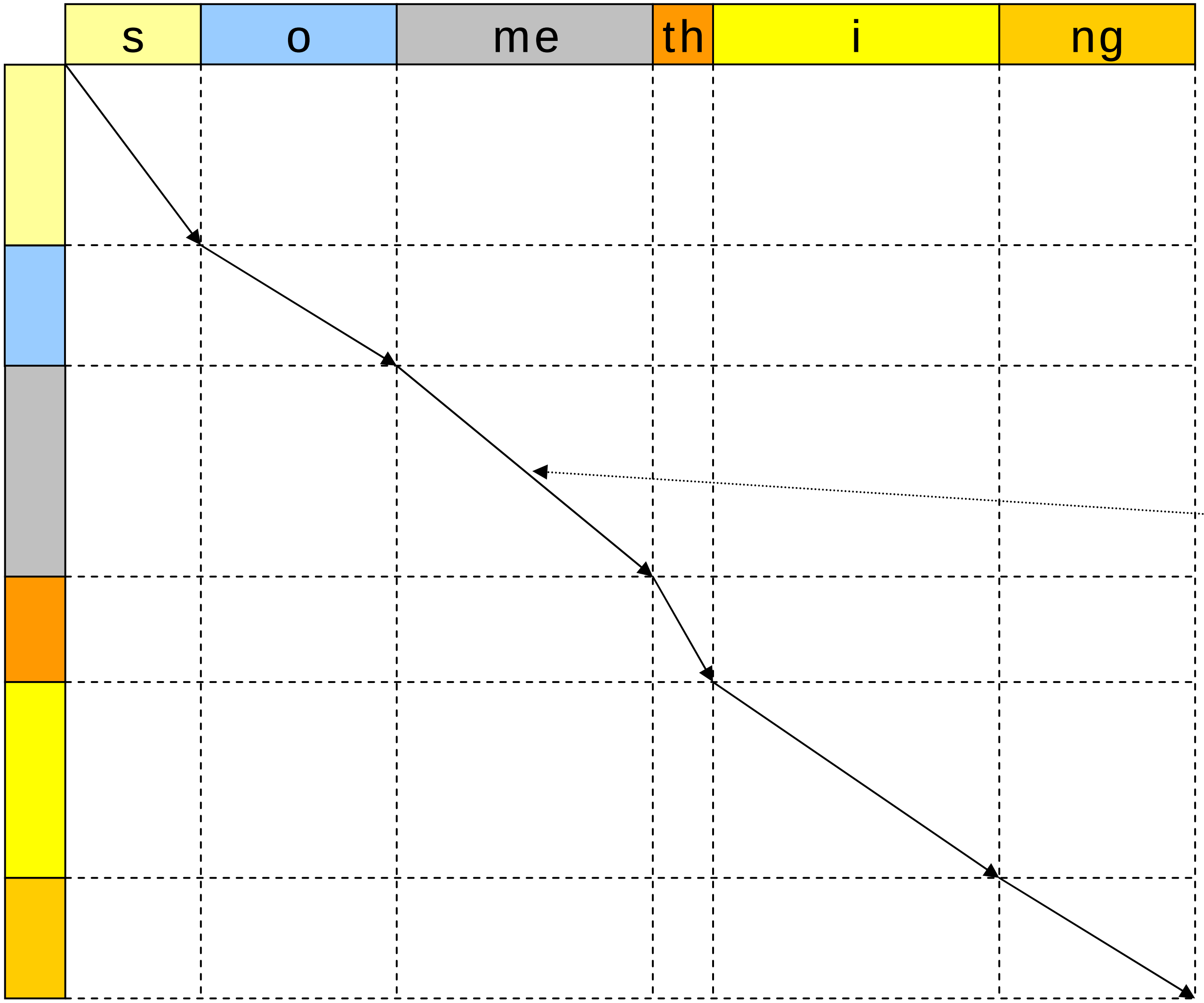
- Back to template matching for speech: *dynamic time warping*
  - Input and templates are sequences of feature vectors instead of letters

- Intuitive understanding of why DP-like algorithm might work to find a best alignment of a template to the input:
  - We need to search for a path that finds the following alignment:



- Consider the 2-D matrix of template-input frames of speech

DTW: DP for Speech Template Matching



Need to find something like this warped path

DTW: Adapting Concepts from DP

- Some concepts from string matching need to be adapted to this problem
  - What are the allowed set of transitions in the search trellis?
  - What are the edge and local node costs?

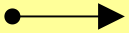
- Once these questions are answered, we can apply essentially the same DP algorithm to find a minimum cost match (path) through the search trellis

DTW: Determining Transitions

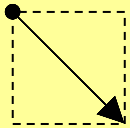
- Recall that transitions are derived from a conceptual model of how a template might have to be distorted to match the input
- The main modes of distortion are *stretching* and *shrinking* of speech segments, owing to different speaking rates
  - Of course, we do not know *a priori* what the distortion looks like
- Also, since speech signals are continuous valued instead of discrete, there is really no notion of insertion
  - Every input frame must be matched to *some* template frame
- For meaningful comparison of two different path costs, their lengths must be kept the same
  - So, every input frame is to be aligned to a template frame *exactly* once

DTW: Transitions

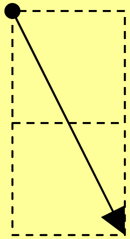
□ Typical transitions used in DTW for speech:



The next input frame aligns to the same template frame as the previous one. (Allows a template segment to be arbitrarily stretched to match some input segment)



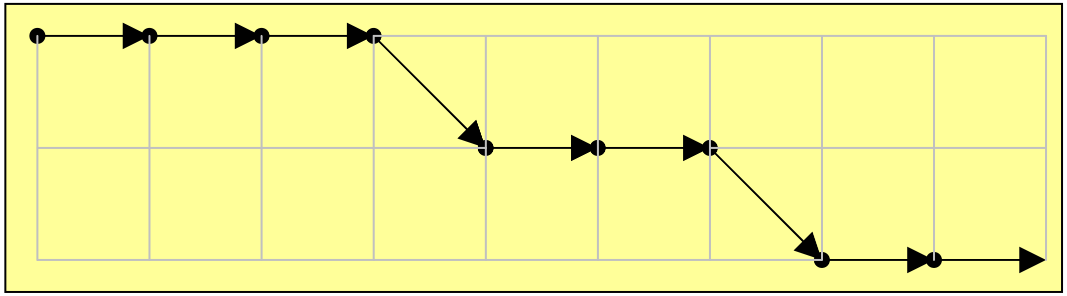
The next input frame aligns to the next template frame. No stretching or shrinking occurs in this region



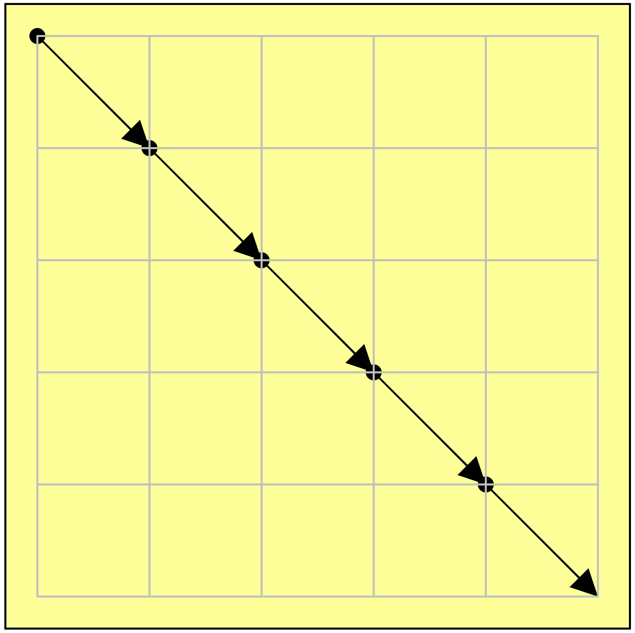
The next input frame skips the next template frame and aligns to the one after that. Allows a template segment to be shrunk (by at most ½) to match some input segment

□ Note that all transitions move one step to the right, ensuring that each input frame gets used exactly once along any path

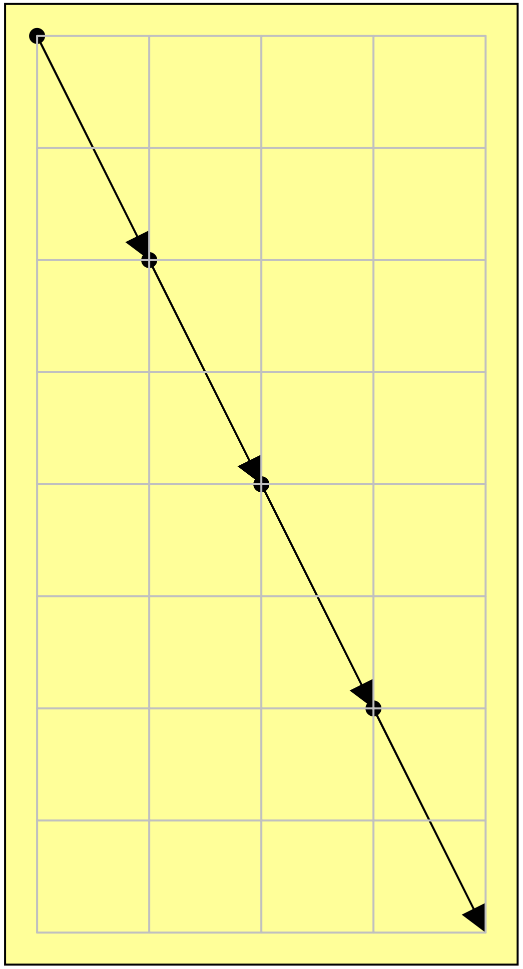
DTW: Use of Transition Types



Short template, long input



Approx. equal length template, input



Long template, short input

DTW: Other Transition Choices

- Other transition choices are possible:
  - Skipping more than one template frame (greater shrink rate)
  - Vertical transitions: the same input frame matches more than one template frame
    - This is less often used, as it can lead to different path lengths, making their costs not easily comparable

DTW: Local Edge and Node Costs

- □ Typically, there are no edge costs; any edge can be taken with no cost
- □ Local node costs measure the dissimilarity or distance between the respective input and template frames
- □ Since the frame content is a multi-dimensional feature-vector, what dissimilarity measure can we use?
- □ A simple measure is *Euclidean distance*; *i.e.* geometrically how far one point is from the other in the multi-dimensional vector space
  - For two vectors X = (x_1, x_2, x_3 … x_N), and Y = (y_1, y_2, y_3… y_N), the Euclidean distance between them is:

$$\boxed{\sqrt{\Sigma(x_i-y_i)^2},\ i = 1 \,..\, N}$$

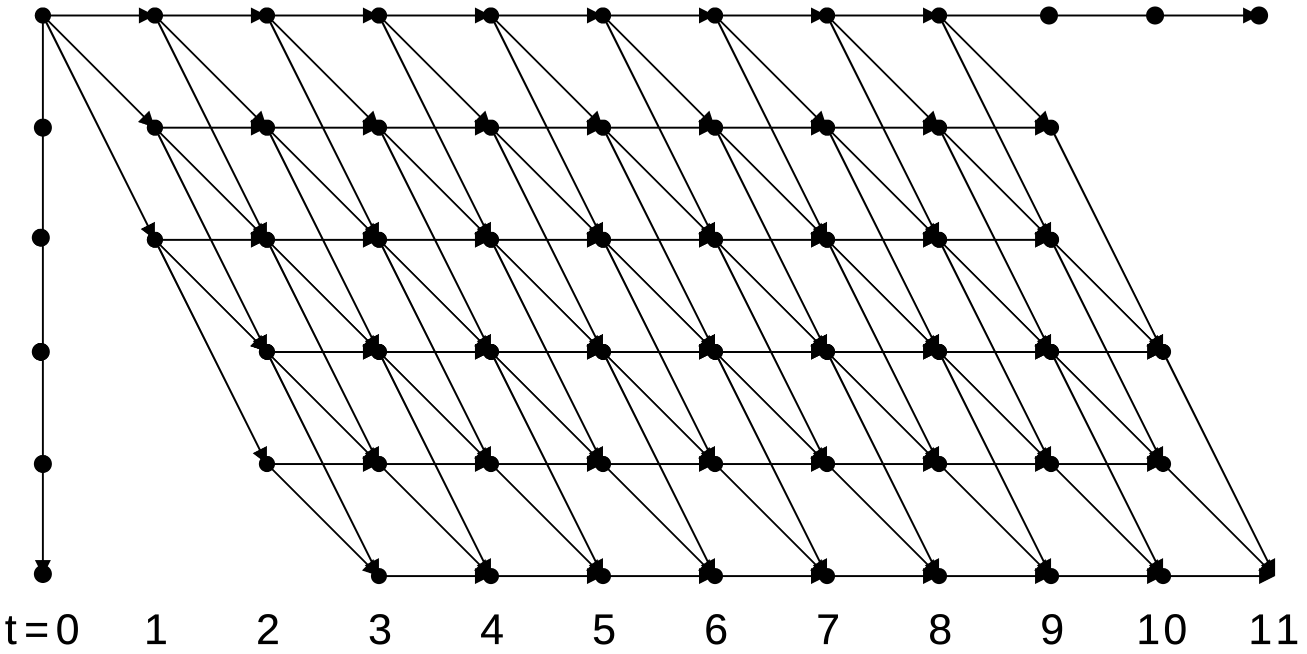
  - Thus, if X and Y are the same point, the Euclidean distance = 0
  - The farther apart X and Y are, the greater the distance

DTW: Local Edge and Node Costs

- Other distance measure could also be used:
  - Manhattan metric or the L1 norm: $\Sigma|A_i - B_i|$
  - Weighted Minkowski norms: $(\Sigma w_i|A_i - B_i|^n)^{1/n}$

DTW: Overall algorithm

- The transition structure and local edge and node costs are now defined
- The search trellis can be realized and the DP algorithm applied to search for the minimum cost path, as before
  - Example trellis using the transition types shown earlier:



DTW: Overall Algorithm (contd.)

- □ Let $P_{i,j}$ = the best path cost from origin to node [*i*,*j*] (i.e., where *i*-th template frame aligns with *j*-th input frame)
- □ Let $C_{i,j}$ = the local node cost of aligning template frame *i* to input frame *j* (Euclidean distance between the two vectors)
- □ Then, by the DP formulation:

$$\boxed{\begin{aligned} P_{i,j} &= \min\,(P_{i,j-1} + C_{i,j},\ P_{i-1,j-1} + C_{i,j},\ P_{i-2,j-1} + C_{i,j}) \\ &= \min\,(P_{i,j-1},\ P_{i-1,j-1},\ P_{i-2,j-1}) + C_{i,j} \end{aligned}}$$

  - ■ Remember edge costs are 0, otherwise they should be added to the costs

- □ If the template is *m* frames long and the input is *n* frames long, the best alignment of the two has the cost = $P_{m,n}$

- □ *Note*: Any path leading to an *internal* node (*i.e.* not *m*,*n*) is called a *partial path*

DTW: Computational Cost

- As with DP, the computational cost for the above DTW is proportional to:

  M x N x 3, where

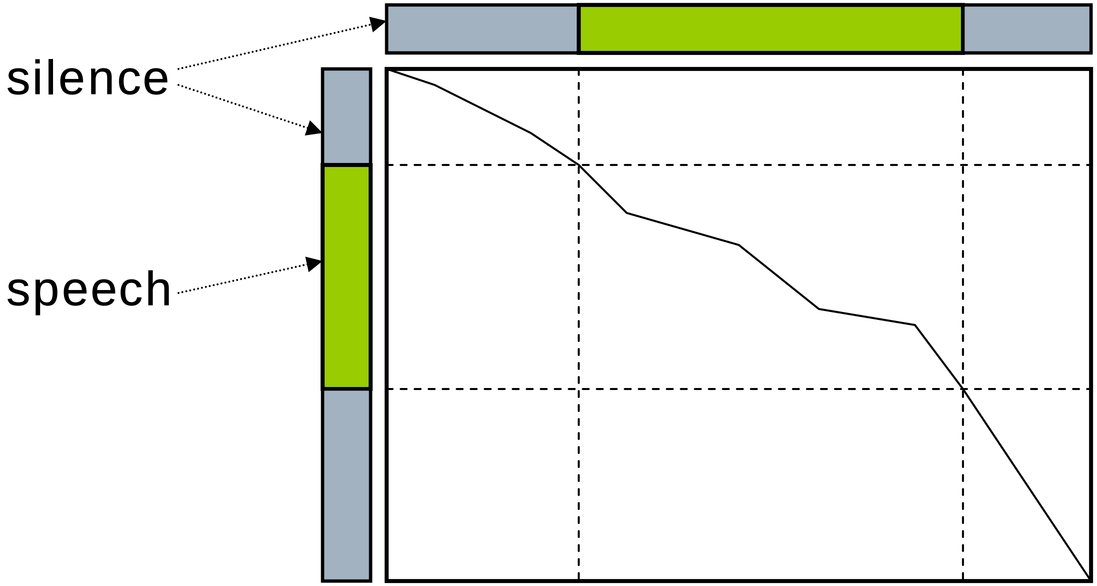
  M = No. of frames in the template

  N = No. of frames in the input

  3 is the number of incoming edges per node

Handling Surrounding Silence

- The DTW algorithm automatically handles any silence region surrounding the actual speech, within limits:



- But, the transition structure does not allow a region of the template to be shrunk by more than ½ !
  - Need to ensure silences included in recording are of generally consistent lengths, or allow other transitions to handle a greater “warp”

Isolated Word Recognition Using DTW

- We now have a method for measuring the best match of a template to the input speech

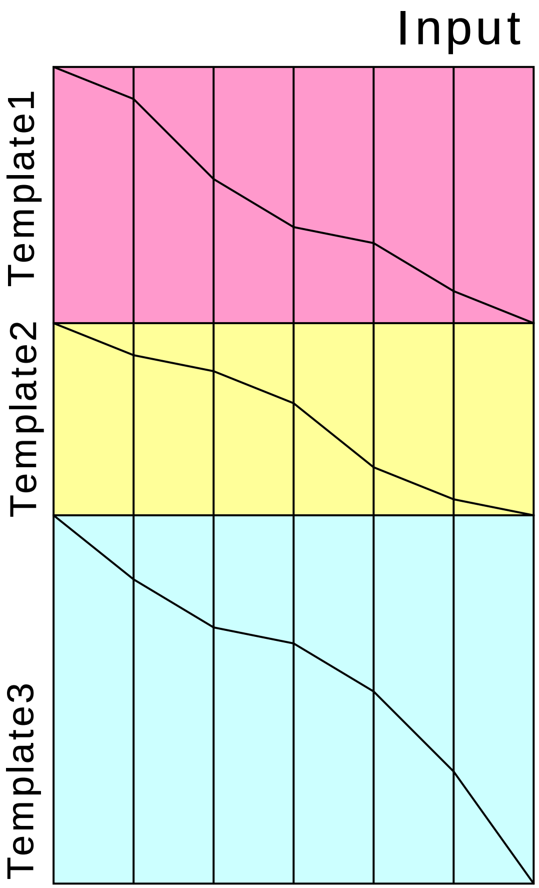
- How can we apply this to perform isolated word recognition?
  - For each word in the vocabulary, pre-record a spoken example (its template)
  - For a given input utterance, measure its minimum distance to each template using DTW
  - Choose the template that delivers the smallest distance

- As easy as that!
  - Could implement this on a cell phone for dialing

- Is there an efficient way to do this?

Time Synchronous Search

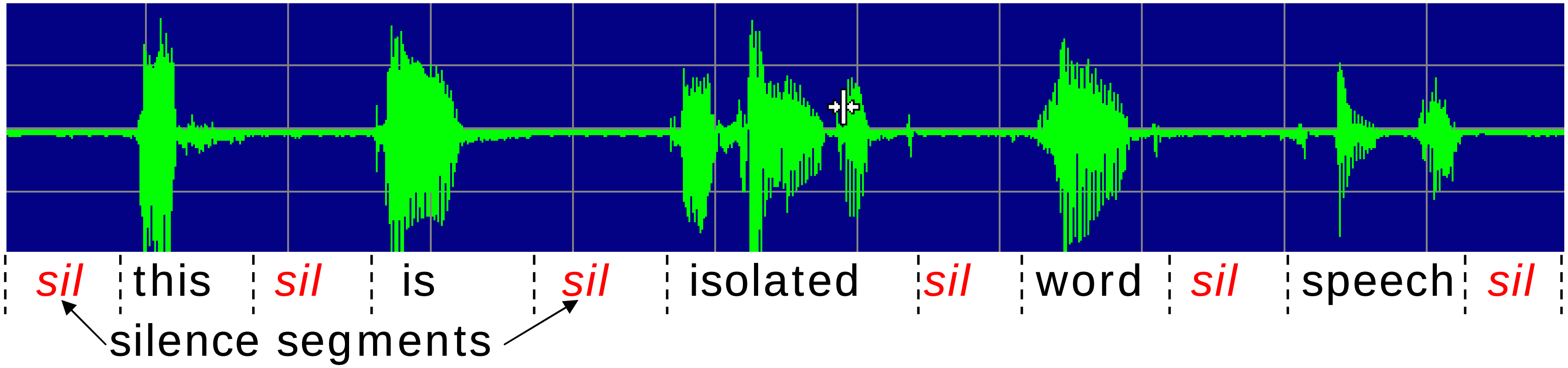
- Since input frames are processed sequentially, the input speech can be matched to all templates simultaneously
- The figure shows three such matches going on in parallel
- Essentially, every template match is started simultaneously and stepped through the input in lock-step fashion
  - Hence the term *time synchronous*
- Advantages
  - No need to store the entire input for matching with successive templates
  - Matching can proceed as the input comes in
  - Enables *pruning* for computational efficiency (as we will see later)
  - Other advantages in continuous speech recognition (will be seen later)



Example: Isolated Speech Based Dictation

- We could, in principle, almost build a large vocabulary dictation application using the techniques learned so far
  - Each word is spoken in isolation, *i.e.* silence after every word
  - Need a template for every word in vocabulary
  - Accuracy would probably be terrible
    - Many words have very similar acoustics in a large vocabulary system *e.g.* STAR/SCAR, MEAN/NEEM, DOOR/BORE
    - We need additional techniques for improving accuracy (later)
    - But, in principle, one can be built, except…
- How does such an application know when a word is spoken?
  - Explicit “click-to-speak”, “click-to-stop” button clicks from user, for every word?
    - Obviously extremely tedious
  - Need a speech/silence detector!

Speech-Silence Detection: Endpointer



sil | this | sil | is | sil | isolated | sil | word | sil | speech | sil

silence segments

- Without explicit signals from the user, can the system automatically detect pauses between words, and segment the speech stream into isolated words?
- Such a speech/silence detector is called an *endpointer*
  - Detects speech/silence boundaries (shown by dotted lines)
  - Words can be isolated by choosing speech segments between *midpoints* of successive silence segments
- Most speech applications use such an endpointer to relieve the user of having to indicate start and end of speech

A Simple Endpointing Scheme

- Based on silence segments having low signal amplitude
  - Usually called *energy-based* endpointing

- The raw audio samples stream is processed as a short sequence of *frames* (as for feature extraction)
- The signal *energy* in each frame is computed
  - Typically in decibels (dB): 10 *log* (Σx_i^2), where x_i are the sample values in the frame
- A pre-defined *threshold* is used to classify each frame as speech or silence
- The labels are *smoothed* to eliminate spurious labels due to noise
  - *E.g.* minimum silence and speech segment length limits may be imposed
  - A very short speech segment buried inside silence may be treated as silence

- This scheme works reasonably well under quiet background conditions

Isolated Speech Based Dictation (Again)

- With such an endpointer, we have all the tools to build a complete, isolated word recognition based dictation system, or any other application

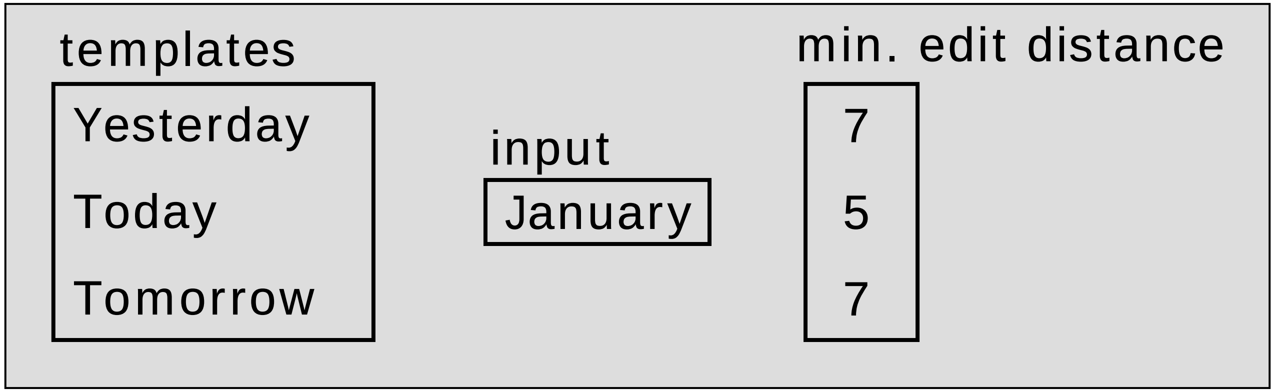
- However, as mentioned earlier, accuracy is a primary issue when going beyond simple, small vocabulary situations

Dealing with Recognition Errors

- Applications can use several approaches to deal with speech recognition errors
- Primary method: improve performance by using better models in place of simple templates
  - We will consider this later
- However, in addition to basic recognition, most systems also provide other, orthogonal mechanisms for applications to *deal* with errors
  - Confidence estimation
  - Alternative hypotheses generation (N-best lists)
- We now consider these two mechanisms, briefly

Confidence Scoring

- *Observation*: DP or DTW will *always* deliver a minimum cost path, *even if it makes no sense*
- Consider string matching:



templates

Yesterday

Today

Tomorrow

input

January

min. edit distance

7

5

7

- The template with minimum edit distance will be chosen, even though it is “obviously” incorrect
  - How can the application discover that it is “obviously” wrong?
- *Confidence scoring* is the problem of determining how confident one can be that the recognition is “correct”

Confidence Scoring for String Match

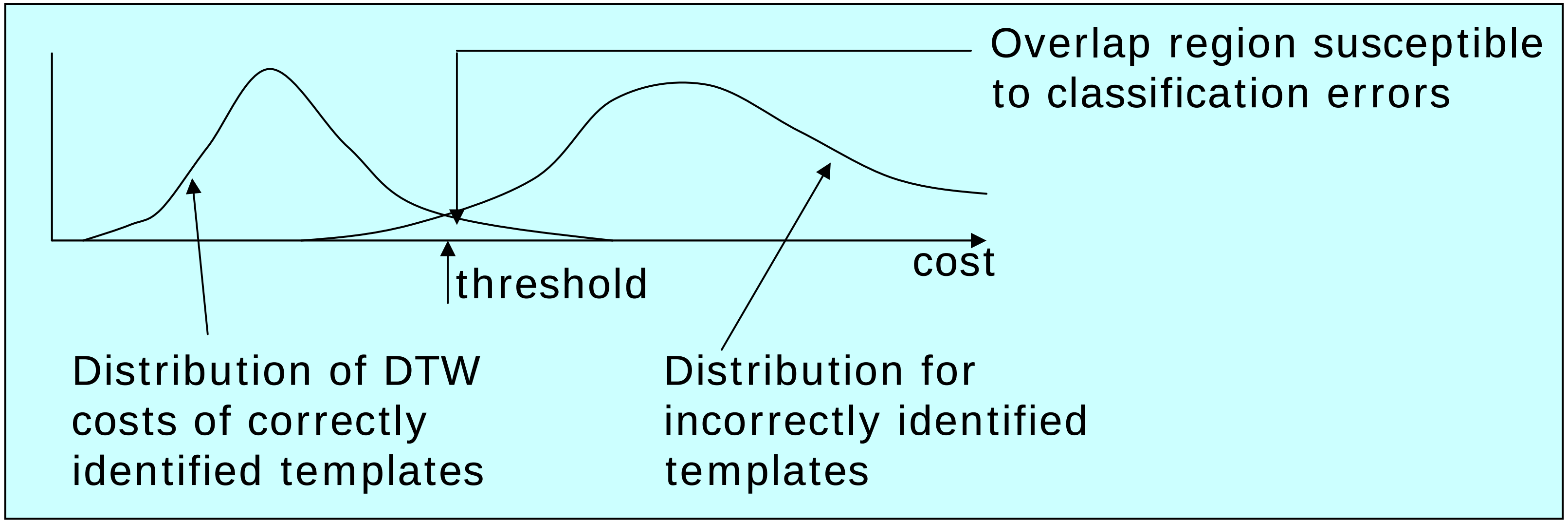
- A simple confidence scoring scheme: Accept the matched template string only if the cost <= some threshold
  - We encountered its use in the hypothetical google search string example!

- This treats all template strings equally, regardless of length
- Or: Accept if cost <= 1 + some fraction (*e.g.* 0.1) of template string length
  - Templates of 1-9 characters tolerate 1 error
  - Templates of 10-19 characters tolerate 2 errors, etc.
- Easy to think of other possibilities, depending on the application

- Confidence scoring is one of the more application-dependent functions in speech recognition

Confidence Scoring for DTW

- Can we use similar thresholding technique for template matching using DTW?
  - Unlike in string matching, the cost measures are not immediately, meaningfully “accessible” values
  - Need to know range of minimum cost when correctly matched and when incorrectly matched
    - If the ranges do not overlap, one could pick a threshold



Overlap region susceptible to classification errors

Distribution of DTW costs of correctly identified templates

Distribution for incorrectly identified templates

Confidence Scoring for DTW

- As with string matching, the DTW cost may have to be *normalized*
  - Use DTW cost / frame of input speech, instead of total DTW cost, before determining threshold
- Cost distributions and threshold have to be determined *empirically*, based on a sufficient collection of test data
- Unfortunately, confidence scores based on such distance measures are not very reliable
  - Too great an overlap between distribution of scores for correct and incorrect templates
  - We will see other, more reliable methods later on

N-best List Generation

- *Example*: Powerpoint catches spelling errors and offers several alternatives as possible corrections
- *Example*: In the isolated word dictation system, *Dragon Dictate*, one can select a recognized word and obtain alternatives
  - Useful if the original recognition was incorrect
- Basic idea: identifying not just the best match, but the top so many matches; *i.e.*, the *N-best list*
- Not hard to guess how this might be done, either for string matching or isolated word DTW!
  - (How?)

Improving Accuracy: Multiple Templates

- Problems with using a single exemplar as a template
  - New instances of a word can differ significantly from it
    - Makes template matching highly brittle
    - Works only with small vocabulary of very distinct words
    - Works poorly across different speakers
- What if we use multiple templates for each word to handle the variations?
  - Preferably collected from several speakers
- Template matching algorithm is easily modified
  - Simply match against *all* available templates and pick the best
- However, computational cost of matching increases linearly with the number of available templates
  - Remember matching each template cost ~ (Template length x Input length x 3)

Reducing Search Cost: Pruning

- Reducing search cost implies reducing the size of the lattice that has to be evaluated

- There are several ways to accomplish this
  - Reducing the complexity and size of the models (templates)
    - *E.g.* replacing the multiple templates for a word by a single, *average* one
  - Eliminating parts of the lattice from consideration altogether
    - This approach is called *search pruning*, or just *pruning*
  - We consider pruning first

- Basic consideration in pruning: *As long as the best cost path is not eliminated by pruning, we obtain the same result*

Pruning

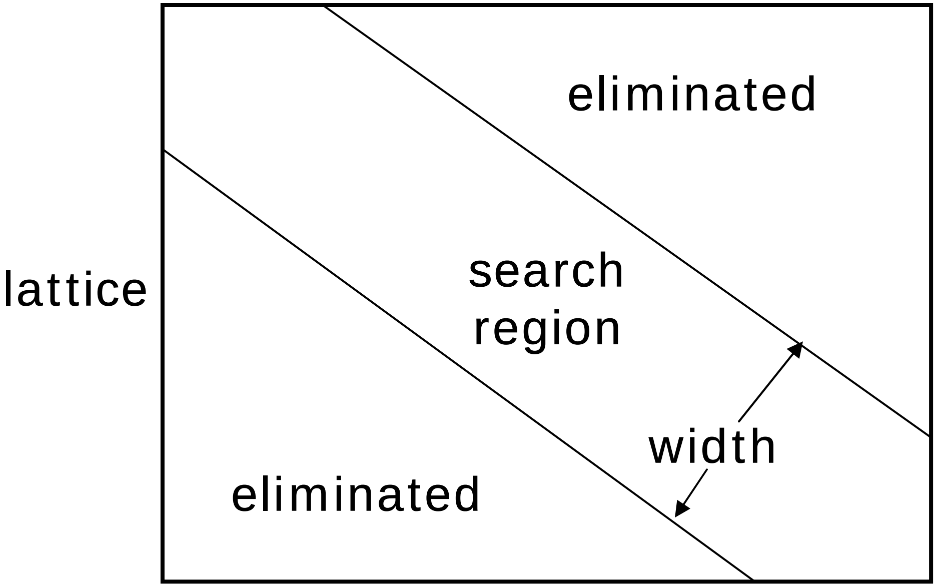
- Pruning is a *heuristic*: typically, there is a *threshold* on some measured quantity, and anything above or below is eliminated

- It is all about choosing the right measure, and the right threshold

- Let us see two different pruning methods:
  - Based on deviation from the diagonal path in the trellis
  - Based on path costs

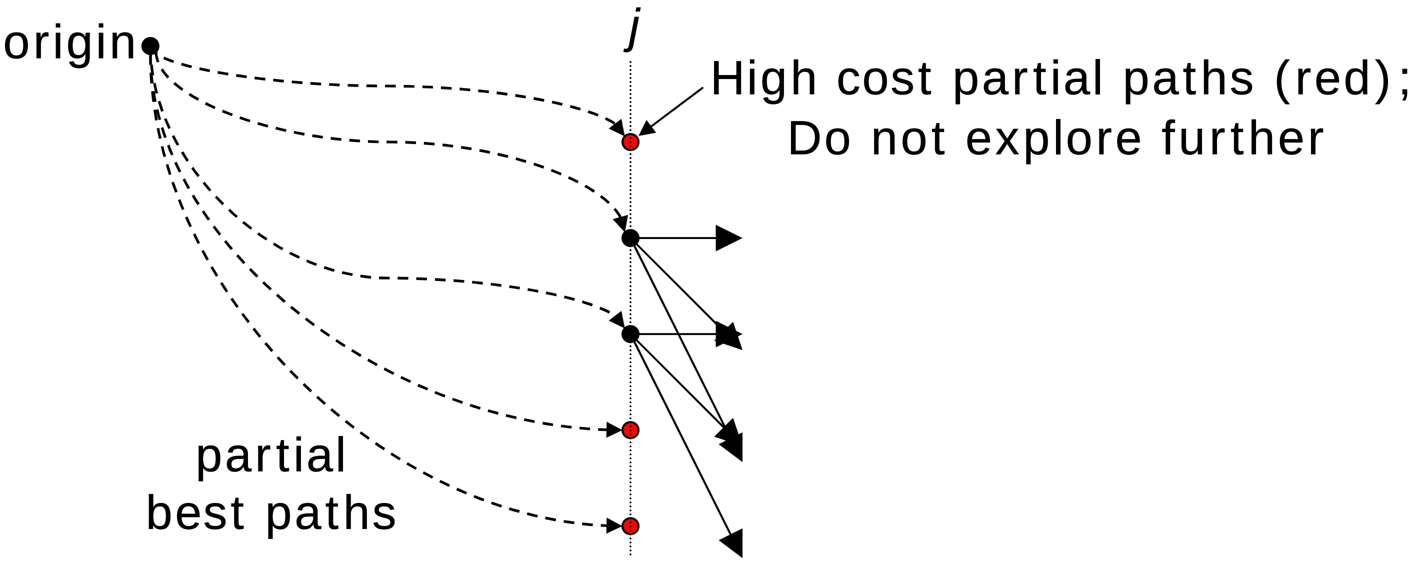
Pruning by Limiting Search Paths

- Assume that the speaking rates between the template and the input do not differ significantly
- There is no need to consider lattice nodes far off the diagonal
- If the search-space "width" is kept constant, cost of search is linear in utterance length instead of quadratic
- However, errors occur if the speaking rate assumption is violated
  - *i.e.* if the template needs to be *warped* more than allowed by the width



Pruning by Limiting Path Cost

- *Observation*: Partial paths that have “very high” costs will rarely recover to win
- Hence, poor partial paths can be eliminated from the search:
  - For each frame *j*, after computing all the trellis nodes path costs, determine which nodes have too high costs
  - Eliminate them from further exploration
  - (*Assumption*: In any frame, the best partial path has low cost)
- *Q*: How do we define “high cost”?



Pruning by Limiting Path Cost

- As with confidence scoring, one *could* define high path cost as a value worse than some fixed threshold
  - But, as already noted, absolute costs are unreliable indicators of correctness
  - Moreover, path costs keep increasing monotonically as search proceeds
    - Recall the path cost equation

$$\boxed{P_{i,j} = \min(P_{i,j-1},\ P_{i-1,j-1},\ P_{i-2,j-1}) + C_{i,j}}$$

- Fixed threshold will not work

Pruning: Beam Search

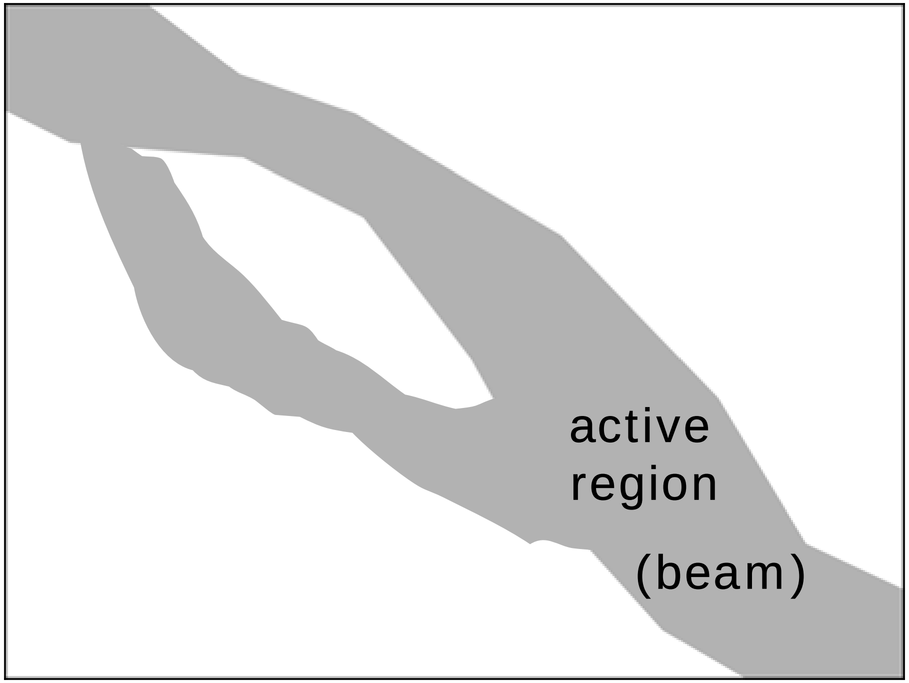
- *Solution*: In each frame *j*, set the pruning threshold by a fixed amount *T relative to the best cost in that frame*
  - *I.e.* if the best partial path cost achieved in the frame is *X*, prune away all nodes with partial path cost > *X*+*T*
  - Note that *time synchronous* search is very efficient for implementing the above

- Advantages:
  - Unreliability of absolute path costs is eliminated
  - Monotonic growth of path costs with time is also irrelevant

- Search that uses such pruning is called *beam search*
  - This is the most widely used search optimization strategy
- The relative threshold *T* is usually called *beam width* or just *beam*

Beam Search Visualization

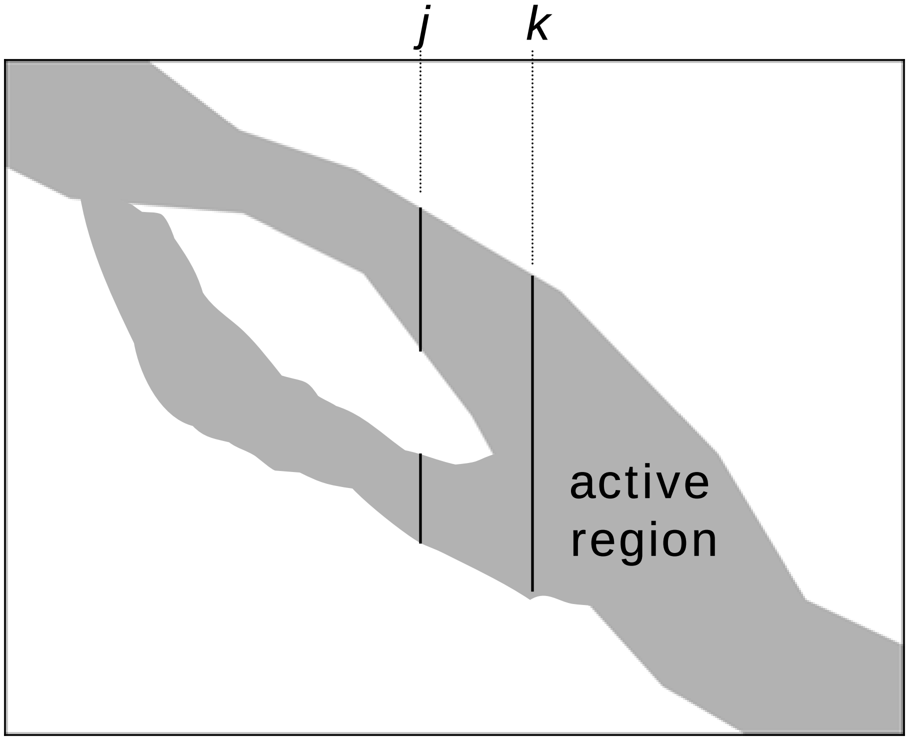
- □ The set of lattice nodes actually evaluated is the *active* set
- □ Here is a typical “map” of the *active region*, aka *beam* (confusingly)



- □ Presumably, the best path lies somewhere in the active region

Beam Search Efficiency

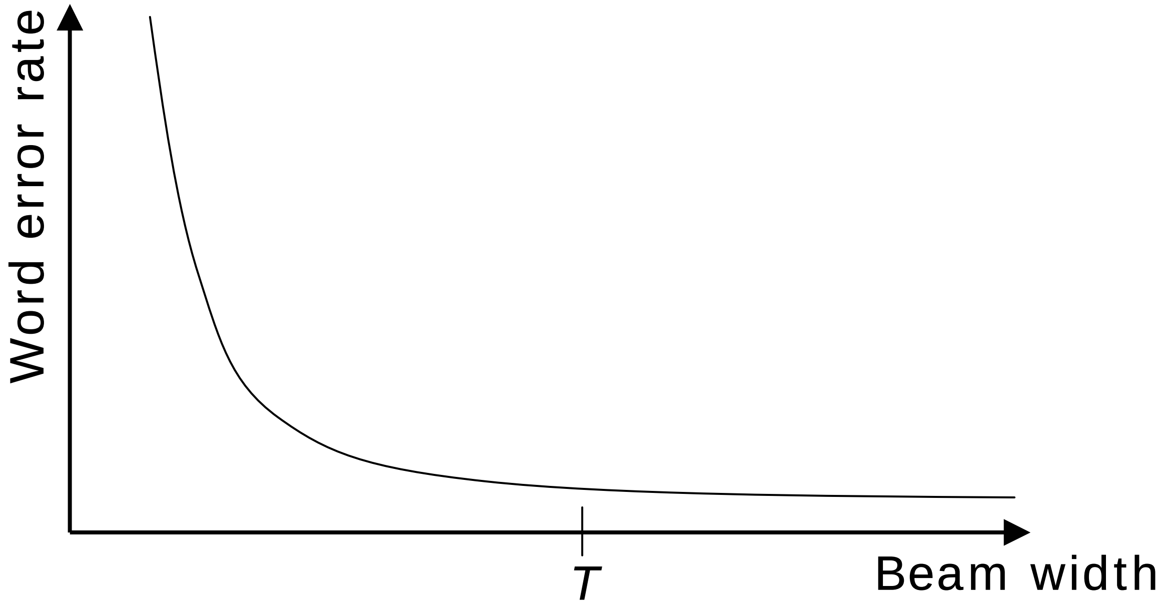
- Unlike the fixed width approach, the computation reduction with beam search is unpredictable
  - The set of *active nodes* at frames *j* and *k* is shown by the black lines
- However, since the active region can follow any *warping*, it is likely to be relatively more efficient than the fixed width approach



Determining the Optimal Beam Width

- Determining the optimal beam width to use is crucial
  - Using too *narrow* or *tight* a beam (too low *T*) can prune the best path and result in too high a match cost, and errors
  - Using too large a beam results in unnecessary computation in searching unlikely paths
  - One may also wish to set the beam to limit the computation (*e.g.* for real-time operation), regardless of recognition errors
- *Unfortunately, there is no mathematical solution to determining an optimal beam width*
- Common method: Try a wide range of beams on some test data until the desired operating point is found
  - Need to ensure that the test data are somehow representative of actual speech that will be encountered by the application
  - The operating point may be determined by some combination of recognition accuracy and computational efficiency

Determining the Optimal Beam Width



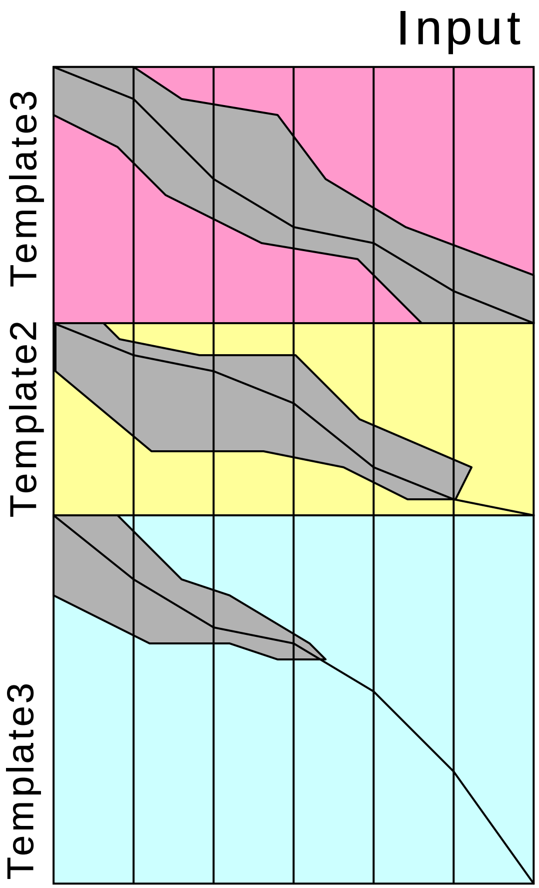
- □ Any value around the point marked *T* is a reasonable beam for minimizing *word error rate* (WER)
- □ A similar analysis may be performed based on average CPU usage (instead of WER)

Beam Search Applied to Recognition

- Thus far, we considered beam search to prune search paths within a single template
- However, its strength really becomes clear in actual recognition (*i.e.* time synchronous search through all templates simultaneously)
  - In each frame, the beam pruning threshold is determined from the *globally* best node in that frame (from all templates)
  - Pruning is performed globally, based on this threshold

Beam Search Applied to Recognition

- The advantage of simultaneous time-synchronous matching of multiple templates:
  - Beams can be globally applied to all templates
  - We use the best score of all template frames (trellis nodes at that instant) to determine the beam at any instant
  - Several templates may in fact exit early from contention
- In the ideal case, the computational cost will be independent of the number of templates
  - All competing templates will exit very early
  - Ideal cases don’t often occur



Pruning and Dynamic Trellis Allocation

- Since any form of pruning eliminates many trellis nodes from being expanded, there is no need to keep them in memory
  - Trellis nodes and associated data structures can be allocated *on demand* (*i.e.* whenever they become active)
  - This of course requires some book-keeping overhead

- May not make a big difference in small vocabulary systems
- But pruning is an essential part of all medium and large vocabulary systems
  - The search trellis structures in 20k word applications take up about 10MB with pruning
  - Without pruning, it would require perhaps 10 times as much!

Recognition Errors Due to Pruning

- Speech recognition invariably contains errors
- Major causes of errors:
  - Inadequate or inaccurate models
    - Templates may not be representative of all the variabilities in speech
  - Search errors
    - Even if the models are accurate, search may have failed because it found a *sub-optimal* path
- How can our DP/DTW algorithm find a sub-optimal path!?
- Because of pruning: it eliminates paths from consideration based on *local* information (the pruning threshold)
- Let *W* be the best cost word for some utterance, and *W*' the recognized word (with pruning)
  - In a *full* search, the path cost for *W* is better than for *W*'
  - But if *W* is not recognized when pruning is enabled, then we have a *pruning error* or *search error*

Measuring Search Errors

- How much of recognition errors is caused by search errors?
- We can estimate this from a sample test data, for which the correct answer is known, as follows:
  - For each utterance *j* in the test set, run recognition using pruning and note the best cost C_j' obtained for the result
  - For each utterance *j*, also match the *correct* word to the input *without* pruning, and note its cost C_j
  - If C_j is better than C_j' we have a pruning error or search error for utterance *j*
- Pruning errors can be reduced by lowering the pruning threshold (*i.e.* making it less aggressive)
- Note, however, this does not guarantee that the correct word is recognized!
  - The new pruning threshold may uncover other incorrect paths that perform better than the correct one

Summary So Far

- Dynamic programming for finding minimum cost paths
- Trellis as realization of DP, capturing the search dynamics
  - Essential components of trellis
- DP applied to string matching
- Adaptation of DP to template matching of speech
  - Dynamic Time Warping, to deal with varying rates of speech
- Isolated word speech recognition based on template matching
- Time synchronous search
- Isolated word recognition using automatic endpointing
- Dealing with errors using confidence estimation and N-best lists
- Improving recognition accuracy through multiple templates
- Beam search and beam pruning

A Footnote: Reversing Sense of "Cost"

- □ So far, we have a *cost* measure in DP and DTW, where higher values imply worse match
- □ We will also frequently use the opposite kind, where higher values imply a *better* match; *e.g.*:
  - The same cost function but with the sign changed (*i.e. negative* Euclidean distance (= $-\sqrt{\Sigma(x_i - y_i)^2}$; X and Y being vectors)
  - $-\Sigma(x_i - y_i)^2$; *i.e.* –ve Euclidean distance squared
- □ We may often use the generic term *score* to refer to such values
  - Higher scores imply better match, not surprisingly

DTW Using Scores

- How should DTW be changed when using scores *vs* costs?
- At least three points to consider:
  - Obviously, we need to *maximize* the total path score, rather than minimize it
  - Beam search has to be adjusted as follows: if the best partial path score achieved in a frame is X, prune away all nodes with partial path score $< X–T$ (instead of $> X+T$, where T is the beam pruning threshold)
  - Likewise, in confidence estimation, we accept paths with scores *above* the confidence threshold, in contrast to cost values *below* the threshold

Likelihood Functions for Scores

- Another common method is to use a *probabilistic* function, for the local node or edge “costs” in the trellis
  - Edges have transition probabilities
  - Nodes have *output* or *observation probabilities*
    - They provide the probability of the observed input
  - Again, the goal is to find the template with highest probability of matching the input

- Probability values as “costs” are also called *likelihoods*

Gaussian Distribution as Likelihood Function

- □ If x is an input feature vector and μ is a template vector of dimensionality N, the function:

$$f_X(x_1,\ldots,x_n) = \frac{1}{(2\pi)^{N/2}\,|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)\right)$$

  is the famous *multivariate Gaussian distribution*, where Σ is the *co-variance matrix* of the distribution

- □ It is one of the most commonly used probability distribution functions for acoustic models in speech recognition

- □ We will look at this in more detail in the next chapter

DTW Using Probabilistic Values

- As with scores (negative-cost) we need to maximize the total path likelihood, since higher likelihoods => better match
- However, the total likelihood for a path is the *product* of the local node and edge likelihoods, rather than the sum
  - One multiplies the individual probabilities to obtain a joint probability value
- As a result, beam pruning has to be modified as follows:
  - if the best partial path likelihood in a frame is X, prune away all nodes with partial path likelihood < XT (where T is the beam pruning threshold)
  - Obviously, T < 1

Log Likelihoods

- Sometimes, it is easier to use the *logarithm* of the likelihood function for scores, rather than likelihood function itself
- Such scores are usually called *log-likelihood* values
  - Using log-likelihoods, multiplication of likelihoods turns into addition of log-likelihoods, and exponentiation is eliminated
- Many speech recognizers operate in log-likelihood mode

Some Fun Exercises with Likelihoods

- How should the DTW algorithm be modified if we use log-likelihood values instead of likelihoods?

- Application of technique known as *scaling*:
  - When using cost or score (-ve cost) functions, show that adding some arbitrary constant value to all the partial path scores in any given frame does not change the outcome
    - The constant can be different for different input frames
  - When using likelihoods, show that *multiplying* partial path values by some positive constant does not change the outcome

- If the likelihood function is the multivariate Gaussian with identity covariance matrix (*i.e.* the Σ term disappears), show that using the log-likelihood function is equivalent to using the Euclidean distance squared cost function