

Dynamic Programming and Single Word Recognizers (Part 1)

- [Comparing Complete Utterances](#)
- [Comparing Complete Utterances](#)
- [Endpoint Detection](#)
- [Approaches to Sequence Alignment](#)
- [Alignment of Speech Vectors May Be Non-Bijective](#)
- [Time Warping](#)
- [Distance Measure between two Utterances](#)
- [The Minimal Editing Distance Problem](#)
- [Dynamic Programming](#)
- [The Dynamic Programming Matrix](#)
 - [Computing the Minimal Editing Distance](#)
- [Utterance Comparison by Dynamic Time Warping](#)
- [DTW-Steps](#)
 - [DTW-Applet](#)
- [Constraints for the DTW-Path](#)
- [Global Constraints for the DTW-Path](#)
- [Java Source-Code for DTW](#)
- [The DTW Searchspace](#)

Dynamic Programming and Single Word Recognizers (Part 2)

- [DTW with Beam Search](#)
- [The Principles of Building Speech Recognizers](#)
- [What are We Trying to Do now ?](#)
- [Isolated Word Recognition with Template Matching](#)

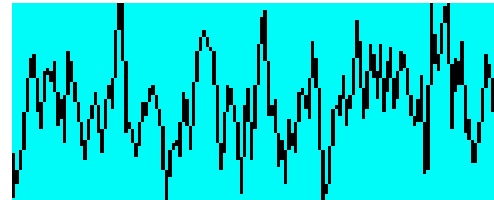
Comparing Complete Utterances

So far:

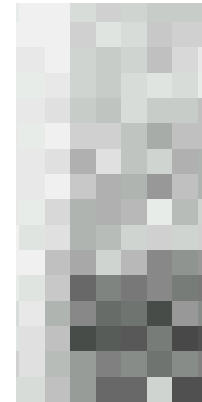
- record sound signal (ADC)
- compute frequency representation
- quantize/classify vectors

We now have:

- sequence of pattern vectors
- we want similarity between two such sequences



=>



Obviously: Order of vectors is important:



V
S.



Comparing Complete Utterances

Comparing speech vector sequences has to overcome three problems:

- speaking rate: if the speaker is speaking faster, we get fewer vectors in the same time
- changing speaking rate purposely: e.g. for disambiguation (said vs. sad)
- changing speaking rate non-purposely: speaking disfluencies



vs.



- so we have to find a way to decide which vector to compare to which
- impose some constraints (not every vector can be compared to every)

Endpoint Detection

When comparing two recorded utterances there might be:

- utterances are of different length
- one or both utterances can be preceded or followed by a period of (possibly non-voluntarily recorded) silence



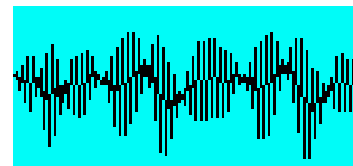
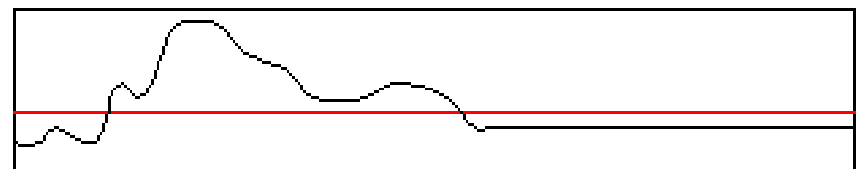
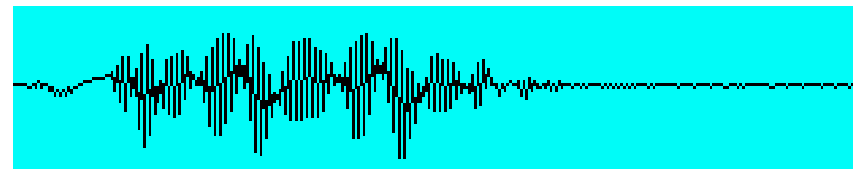
vs.



Also: we might not have any mechanism to signalize the recognizer when it should listen.

Typical Solution:

compute signal power: $p[i..j] = \sum_{k=i..j} s[k]^2$,
then apply threshold to detect speech

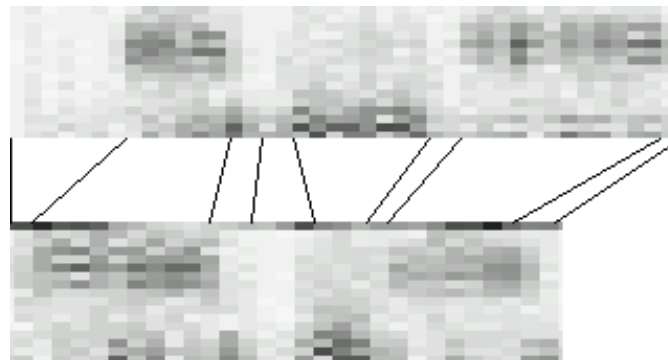


Approaches to Sequence Alignment

First idea:
normalize length and make linear alignment.



Linear alignment can handle the problem of different speaking rates.
But it can not handle the problem of varying speaking rates during the same utterance.

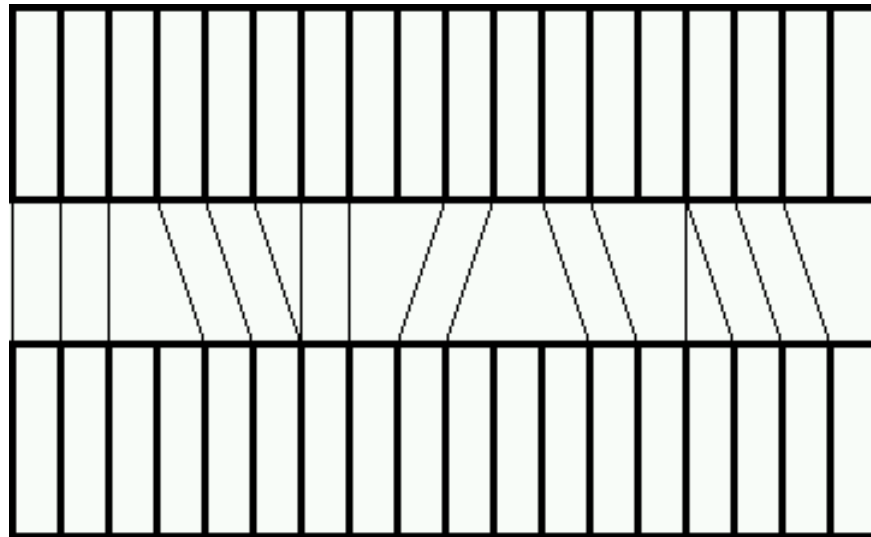


Alignment of Speech Vectors May Be Non-Bijective

Task:

given: two sequences x_1, x_2, \dots, x_n and y_1, y_2, \dots, y_m

wanted: alignment relation R (not function), where (i, j) is in R iff x_i is aligned with y_j .



It is possible that more than one x is aligned to the same y (or vice versa).

It is possible that more than an x or a y has no alignment partner at all.

Time Warping

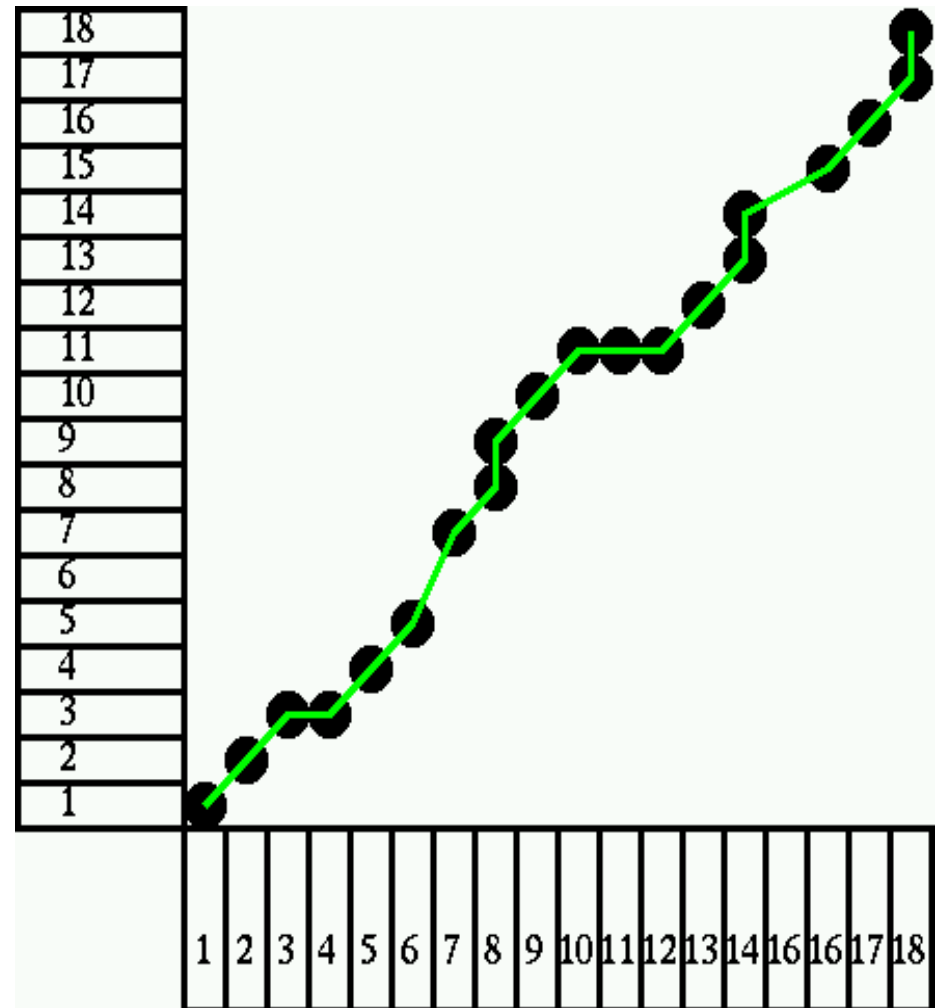
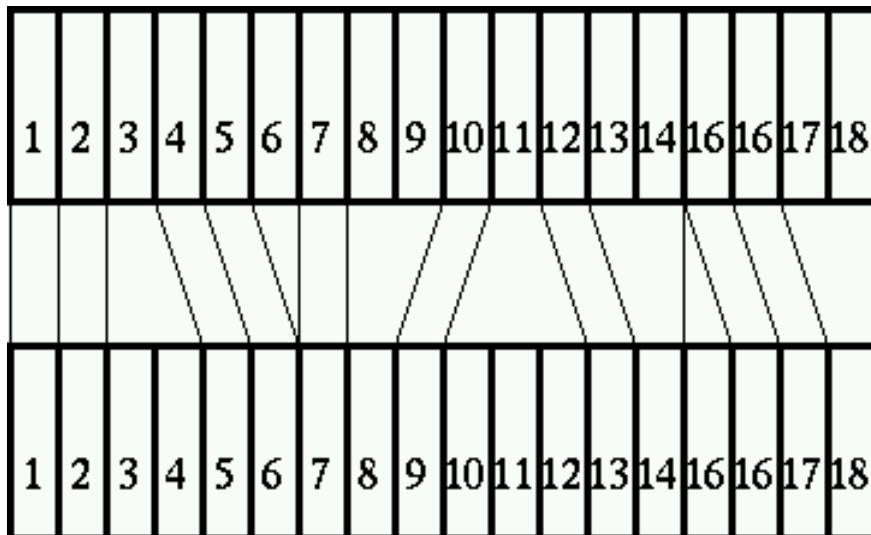
Task:

given: two sequences x_1, x_2, \dots, x_n and

y_1, y_2, \dots, y_m

wanted: alignment relation R (not function), where (i, j) is in R iff x_i is aligned with y_j .

We are looking for a common time-axis:



Distance Measure between two Utterances

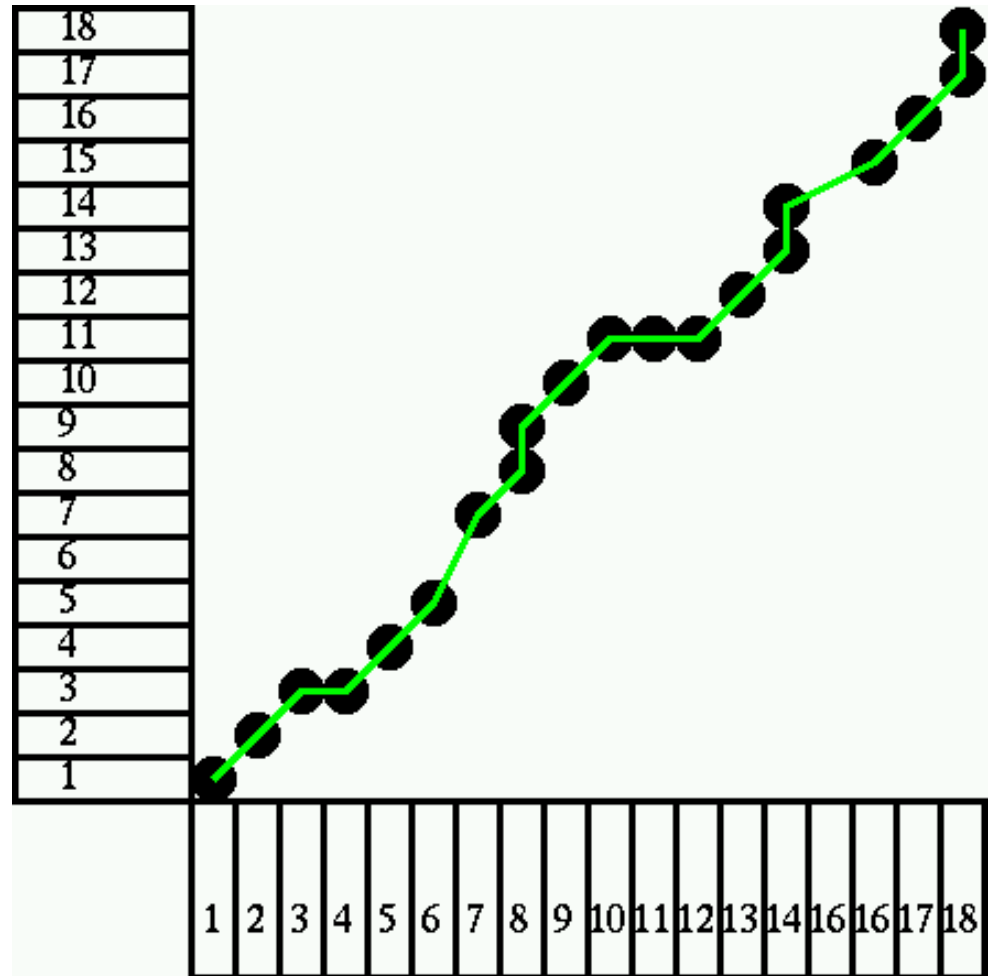
For a given **path** $R(i,j)$, the distance between x and y is the sum of all **local distances** $d(x_i, y_j)$.

In our example:

$$\begin{aligned} & d(x_1, y_1) + d(x_2, y_2) + d(x_3, y_3) + \\ & d(x_3, y_3) + d(x_5, y_4) + d(x_6, y_5) + \\ & d(x_7, y_7) + \dots \end{aligned}$$

Question:

How can we find a path that gives the minimal overall distance?



The Minimal Editing Distance Problem

given: two character sequences (words) x_1, x_2, \dots, x_n and y_1, y_2, \dots, y_m

wanted: the minimal number (and sequence) of editing steps that are needed to convert x to y

The editing cursor starts at x_0 , an editing step can be one of:

- delete the character x_i under the cursor
- insert a character x_i at the cursor position
- replace character x_i at the cursor position with y_j
- moving the cursor to the next character is no editing step, and we can't go back

Example: Convert $x = \text{"BAKERY"}$ to $y = \text{"BRAKES"}$: One possible solution:

- B = B, move cursor to next character
- **insert** character $y_2 = R$
- A = A, move cursor to next character
- K = K, move cursor to next character
- E = E, move cursor to next character
- **replace** character $x_5 = R$ with character $y_5 = S$
- **delete** character $x_6 = Y$ (sequence not necessarily unique)

Dynamic Programming

How can we find the minimal editing distance?

Greedy algorithm? Always perform the step that is currently the cheapest. If there are more than one cheapest step take any one of them.

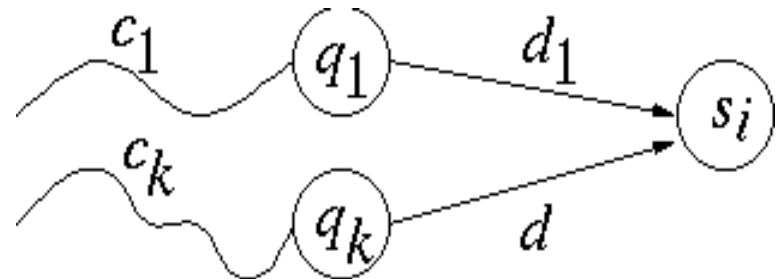
Obvious: can't guarantee to lead to the optimal solution.

Solution: Dynamic Programming (DP)

DP is frequently used in operations research, where consecutive decisions depend on each other and whose sequence must lead to optimal results.

The **key idea** of DP is:

If we would like to take our system into a state s_i , and we know the costs c_1, \dots, c_k for the optimal ways to get from the start to all states q_1, \dots, q_k from which we can go to s , then the optimal way to s goes over the state q_l where $l = \operatorname{argmin}_j c_j$



The Dynamic Programming Matrix

For finding the minimal editing distance from

x_1, x_2, \dots, x_n to y_1, y_2, \dots, y_m

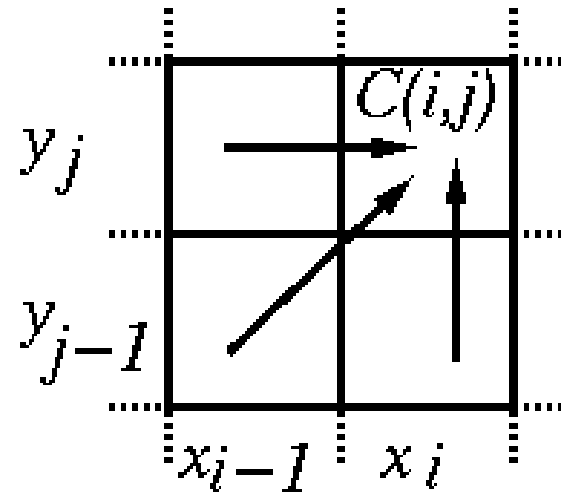
we can define an algorithm inductively.

Let $C(i, j)$ denote the minimal editing distance

from x_1, x_2, \dots, x_i to y_1, y_2, \dots, y_j .

Then we get:

- $C(0,0) = 0$ (no characters no editing)
- $C(i, j)$ is either (whichever is smallest):
 - $C(i-1, j-1)$ plus the cost for replacing x_i with y_j
 - or $C(i-1, j)$ plus the cost for deleting x_i
 - or $C(i, j-1)$ plus the cost for inserting y_j
- Usually:
 - the cost for deleting or inserting a character is 1
 - the cost for replacing x_i with y_j is 0 (if $x_i = y_j$) or 1 (else)
 - it might be useful to define other costs for special purposes
- Eventually:
 - remember for each state $(i-1, j-1)$ which one was the best predecessor (**backpointer**)
 - to find the sequence of editing steps backtrack the predecessor pointers from final state



Utterance Comparison by Dynamic Time Warping

How can we apply the DP algorithm for the minimal editing distance to the utterance comparison problem? **Differences and Questions:**

- What do editing steps correspond to?
- We "never" really get two identical vectors.
- We are dealing with continuous and not discrete signals here.

Answers:

- We can delete/insert/substitute vectors. Define cost for deleting/inserting as we wish, define cost for substituting = distance between vectors
- No two vectors are the same? So what.
- Continuous signals => so we get continuous distances (no big deal)

The DTW-Algorithm

Works like the minimal editing distance algorithm with minor modification:

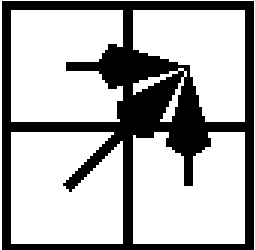
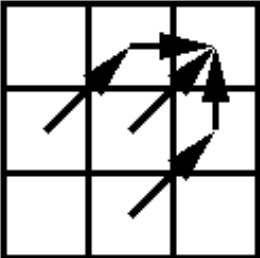
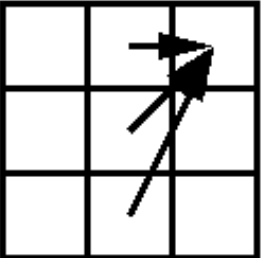
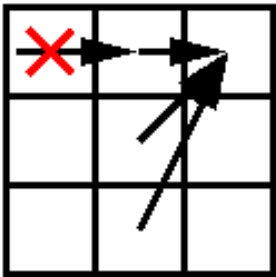
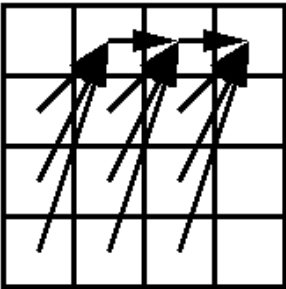
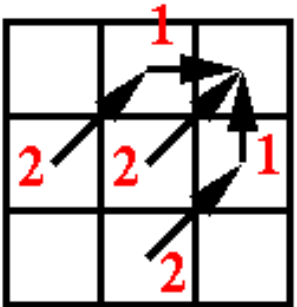
Allow different kinds of steps (different predecessors of a state).

Use vector-vector distance measure as cost function.

DTW-Steps

Many different warping steps are possible and have been used.
Examples:

General rule is:
Cumulative cost of destination =
best-of(cumulative cost of source + cost of step + distance in destination)

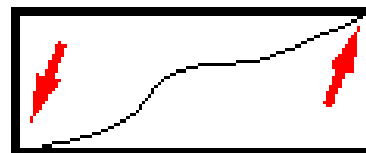
<p>symmetric (editing distance)</p> 		<p>Bakis</p> 
<p>Itakura</p> 		<p>weighted</p> 

Constraints for the DTW-Path

Different kinds of constraints:

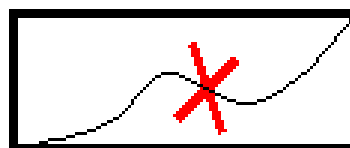
- endpoint constraints:

we want the path not to skip a part at the beginning or end of utterance



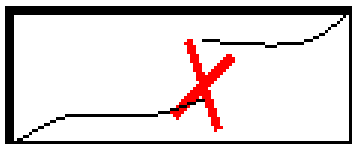
- monotonicity conditions:

we can't go back in time (for neither utterance)



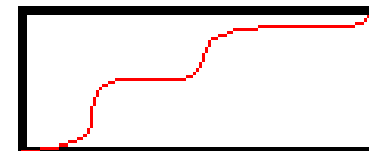
- local continuity:

no jumps etc.



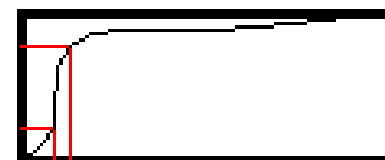
- global path constraints:

path should be close to diagonal

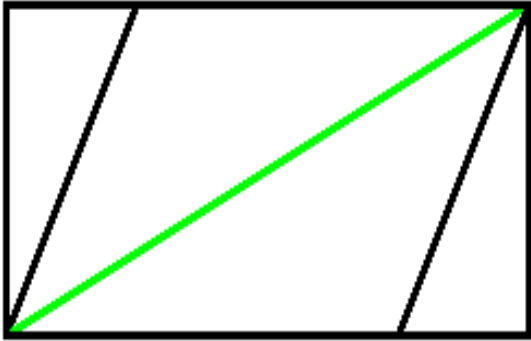
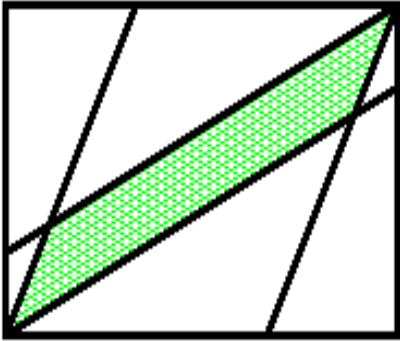
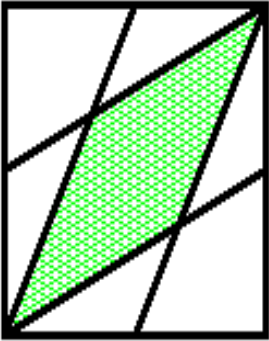
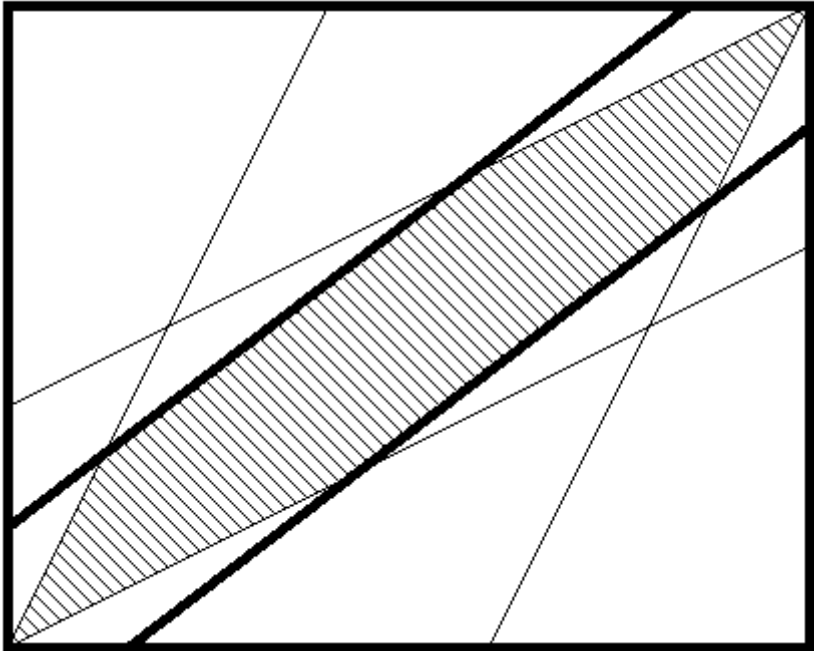
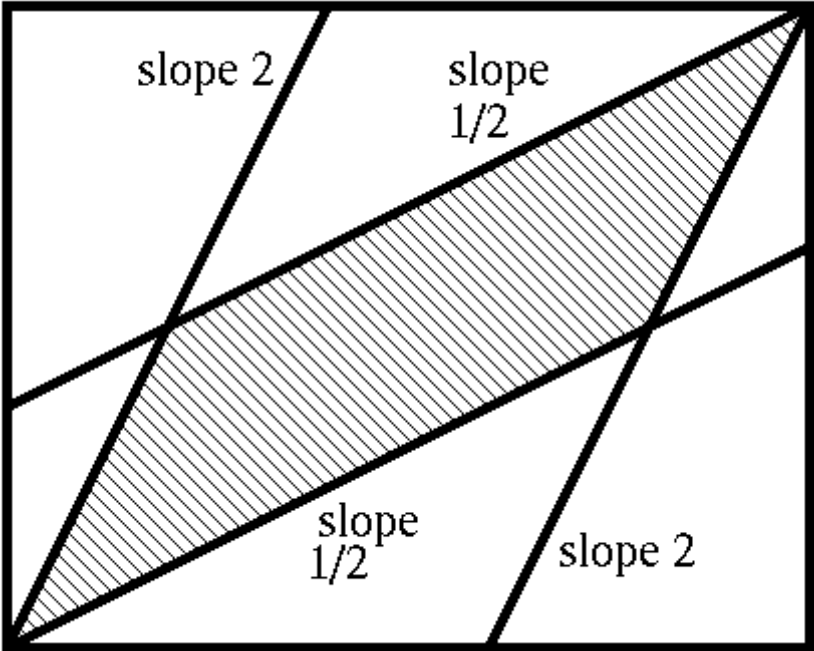


- slope weighting:

we believe the DTW-path should be somehow "smooth"



Global Constraints for the DTW-Path



<- only one path

Java Source-Code for DTW

```
public double dtw() { for (int r=0; r<rowN; r++) {  
  accu[0][r] = 9.9e99; accu[1][r] = 9.9e99; }  
  accu[0][0] = distance(0,0); for (int c=1; c<colN;  
  c++) { int curr = (c-1)%2, next = c%2; for (int r=0;  
  r<rowN; r++) { accu[next][r] = 9.9e99; double d =  
  distance(c,r); if (r>1) if (accu[curr][r-2] + d <  
  accu[next][r]) { accu[next][r] = accu[curr][r-2] +d;  
  back[c][r]=r-2; } if (r>0) if (accu[curr][r-1] + d <  
  accu[next][r]) { accu[next][r] = accu[curr][r-1] +d;  
  back[c][r]=r-1; } if (accu[curr][r ] + d <  
  accu[next][r]) { accu[next][r] = accu[curr][r ] +d;  
  back[c][r]=r; } } } return accu[curr][rowN-1]; }
```

The DTW Searchspace

Already suggested: restrict search space by window around diagonal.

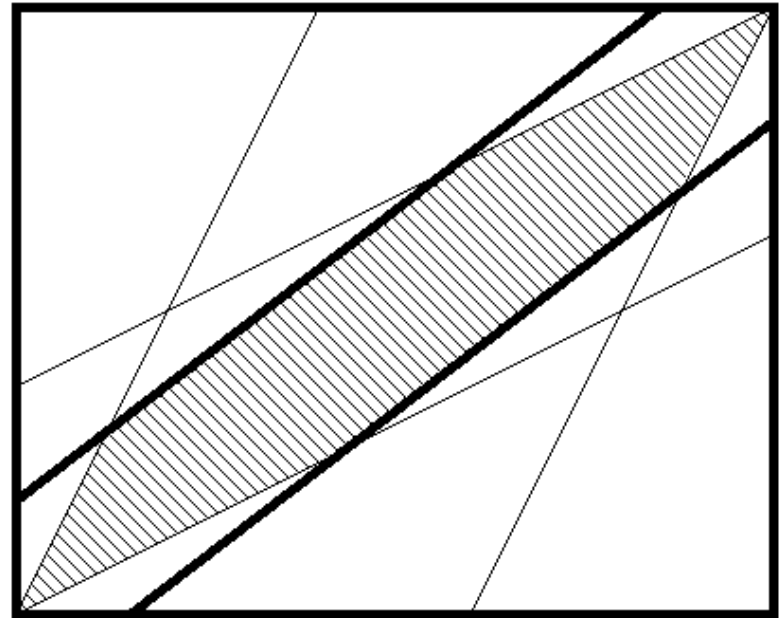
Caveats:

- silence period in one utterance can cause "edgy" path
- search area becomes too restricted when utterance durations differ a lot

Other reason (besides global path constraints) for restricting search space:

Save time: A window that has a constant width, reduces the search effort from $O(n^2)$ to $O(n)$

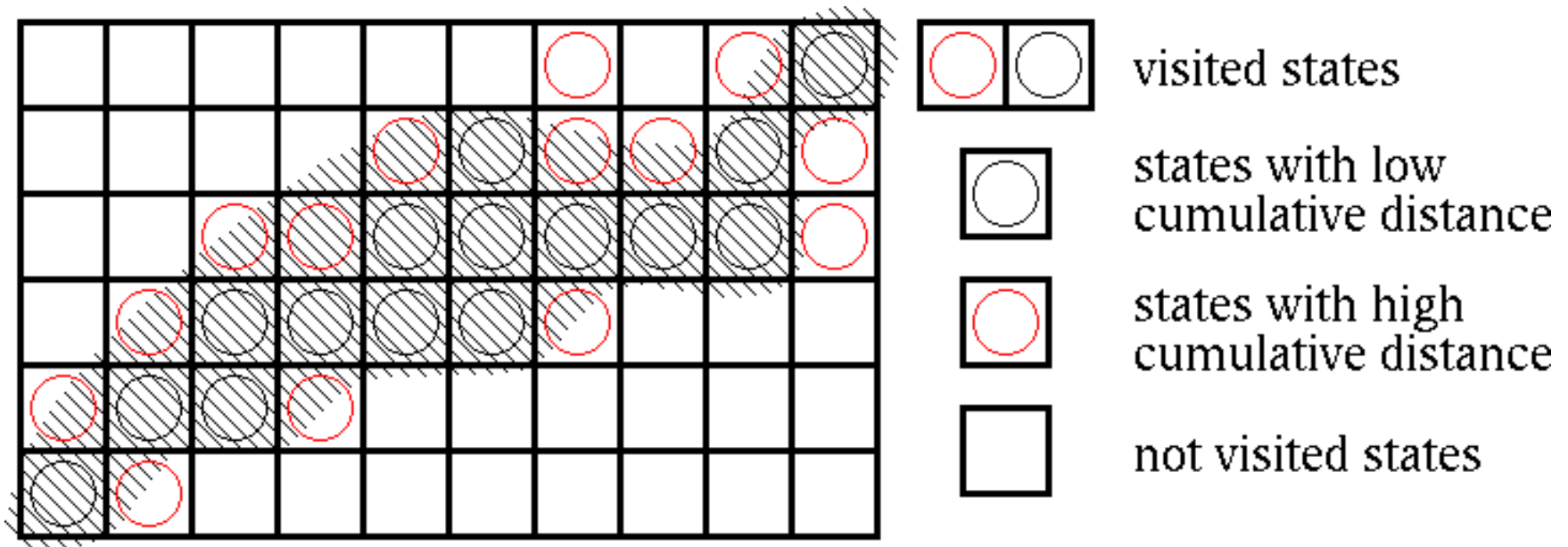
To overcome caveats of "diagonal window" restriction, use: **beamsearch**.



DTW with Beam Search

Idea:

do not consider steps to be possible out of states that have "too high" cumulative distances.



Approaches:

- "expand" only a fixed number of states per column of DTW-matrix
- expand only states that have a cumulative distance less than a factor (the beam) times the best distance so far

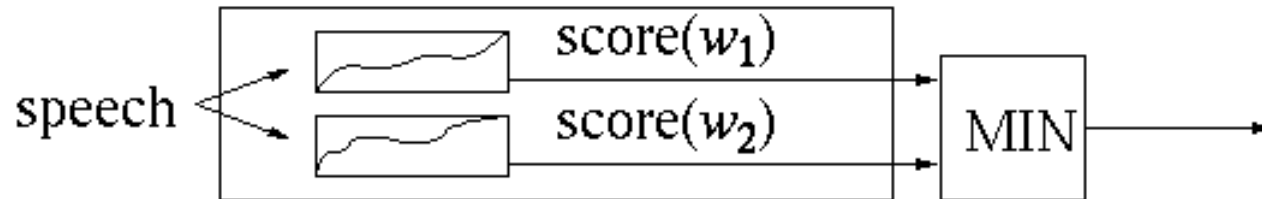
The Principles of Building Speech Recognizers

The following approach is generally considered to be good research habits:

- task specification (what must be recognized)
- data collection
- split up collected data into
 - train set** (used for estimating the recognizers parameters)
 - development set** (used for evaluating the recognizer during optimization)
 - evaluation set** (used only once and never again to report results)
- estimate parameters of recognizer with training data
- evaluate and optimize recognizer with development data
- run test on evaluation data and report results

What are We Trying to Do now?

We will build a first simple isolated-word recognizer using DTW. The recognizer will record speech and print the score for each of its reference patterns:



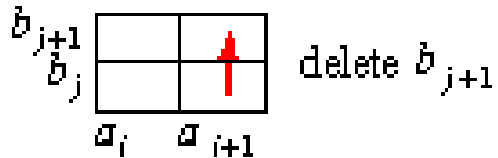
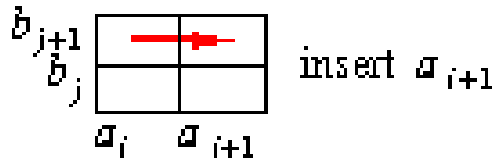
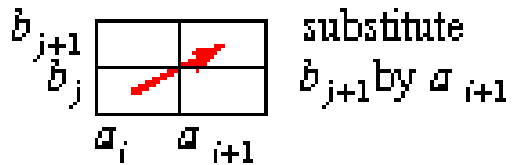
- build recognizer that can recognize two words w_1 and w_2
- collect training examples (one per word in demo, in real life: a lot more)
- skip the optimization phase (don't need development set)
- collect evaluation data (a few examples per word)
- run tests on evaluation data and report results

Isolated Word Recognition with Template Matching

- for each word in the vocabulary, store at least one reference pattern
- when multiple reference patterns are available, either use all of them or compute an average
- during recognition
 - record a spoken word
 - perform pattern matching with all stored patterns (or at least with those that can be used in the current context)
 - compute a DTW score for every vocabulary word (when using multiple references, compute one score out of many, e.g. average or max)
 - recognize the word with the best DTW score
- this approach works only for very small vocabularies and/or for speaker-dependent recognition

Computing the Minimal Editing Distance

Editing Steps:



Dynamic Programming Matrix:



In the applet You can see which editing steps were made (yellow lines). The number in a cell shows the actual costs to get to the specific field (insertion and deletion have the cost 1, substitution the cost 0 if the next characters are equal, else 1). At the end, a red line shows the optimal path.

[code](#)



