String Edit Distance
(and intro to dynamic programming)

Lecture #4

Computational Linguistics

CMPSCI 591N, Spring 2006

University of Massachusetts Amherst

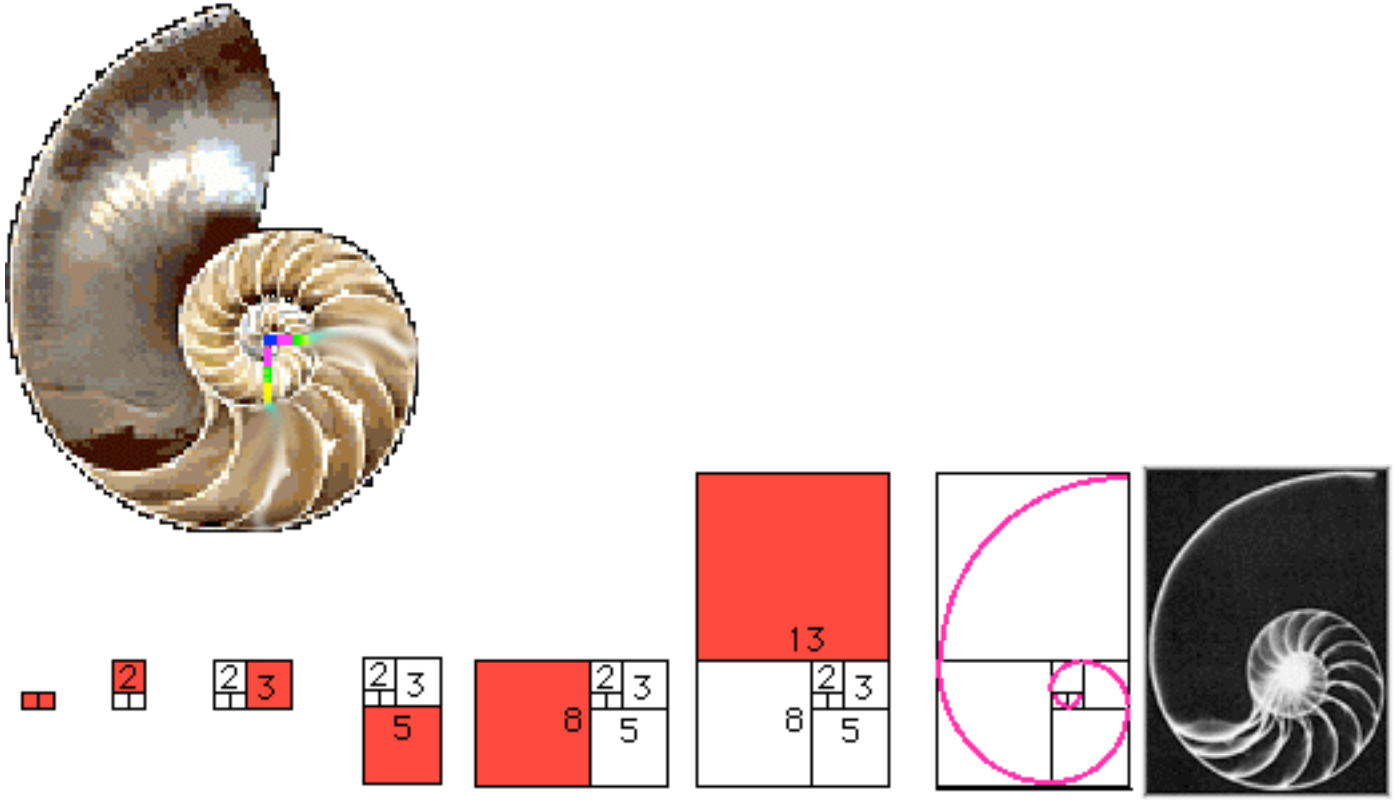
Andrew McCallum

Dynamic Programming

- (Not much to do with “programming” in the CS sense.)

- Dynamic programming is efficient in finding optimal solutions for cases with lots of **overlapping sub-problems**.
- It solves problems by *recombining solutions* to sub-problems, when the sub-problems themselves may share sub-sub-problems.

Fibonacci Numbers



1 1 2 3 5 8 13 21 34 ...

Andrew McCallum, UMass Amherst,
including material from William Cohen

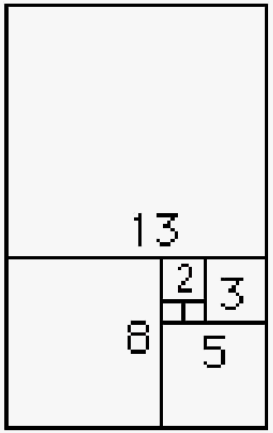
Fibonacci Numbers, $F_n = F_{n-2} + F_{n-1}$

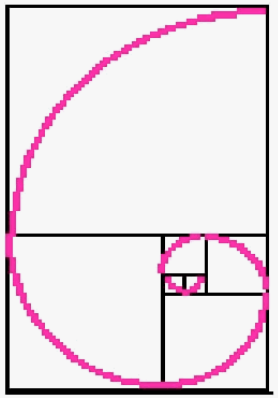
$F_1 = F_2 = 1$

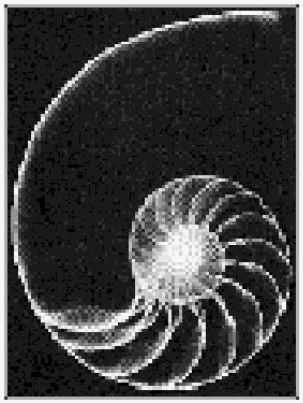
Golden Ratio

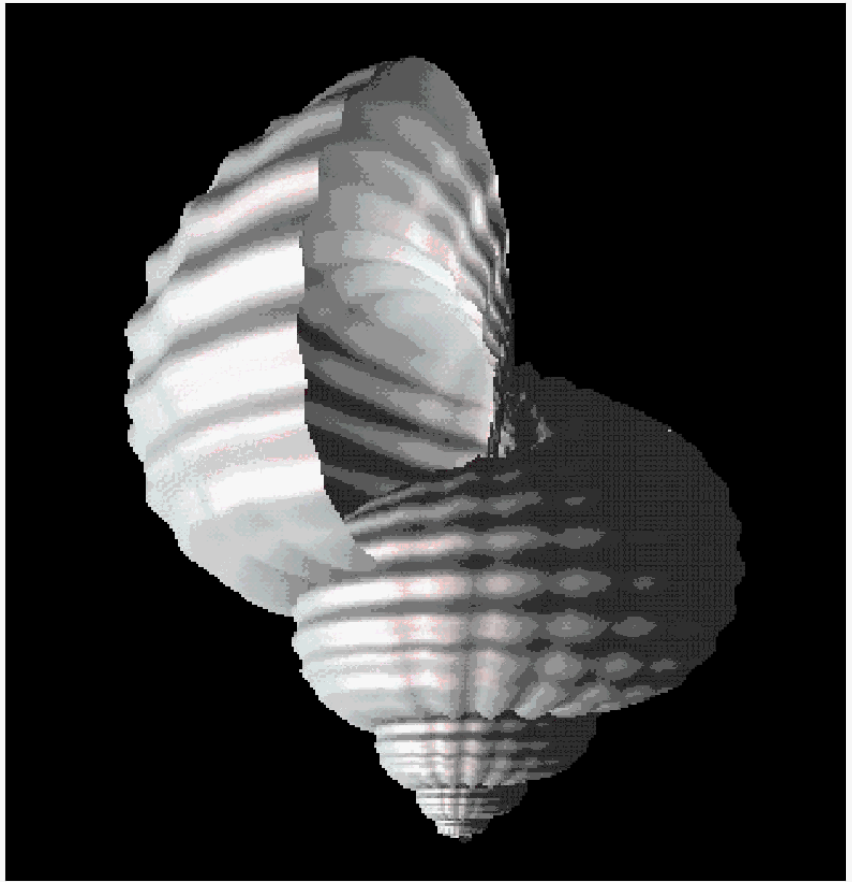
$$g = F_{n+1} / F_n = 1.62$$

and the Beauty of Life





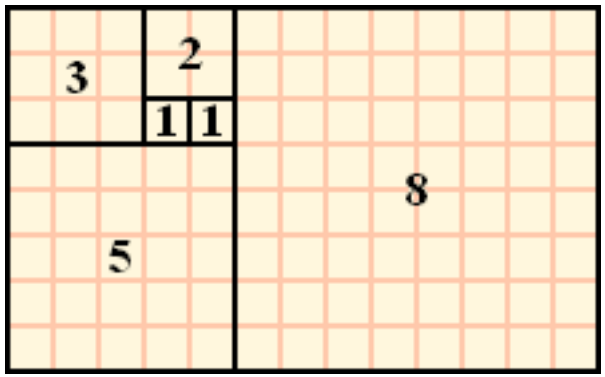




Andrew McCallum, UMass Amherst,
including material from William Cohe..

Calculating Fibonacci Numbers

F(n) = F(n-1) + F(n-2),
where F(0)=0, F(1)=1.



Non-Dynamic Programming implementation

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

For **`fib(8)`**, how many calls to function **`fib(n)`**?

Andrew McCallum, UMass Amherst,
including material from William Cohen

DP Example: Calculating Fibonacci Numbers

Dynamic Programming: avoid repeated calls by remembering function values already calculated.

```
table = {}
def fib(n):
    global table
    if table.has_key(n):
        return table[n]
    if n == 0 or n == 1:
        table[n] = n
        return n
    else:
        value = fib(n-1) + fib(n-2)
        table[n] = value
        return value
```

DP Example: Calculating Fibonacci Numbers

...or alternately, in a list instead of a dictionary...

```
def fib(n):
    table = [0] * (n+1)
    table[0] = 0
    table[1] = 1
    for i in range(2,n+1):
        table[i] = table[i-2] + table[i-1]
    return table[n]
```

We will see this pattern many more times in this course:

1. Create a table (of the right dimensions to describe our problem.
2. Fill the table, re-using solutions to previous sub-problems.

String Edit Distance

Given two strings (sequences) return the "distance" between the two strings as measured by...

...the *minimum* number of "character edit operations" needed to turn one sequence into the other.

`Andrew`

`Amdrewz`

1. substitute **`m`** to **`n`**
2. delete the **`z`**

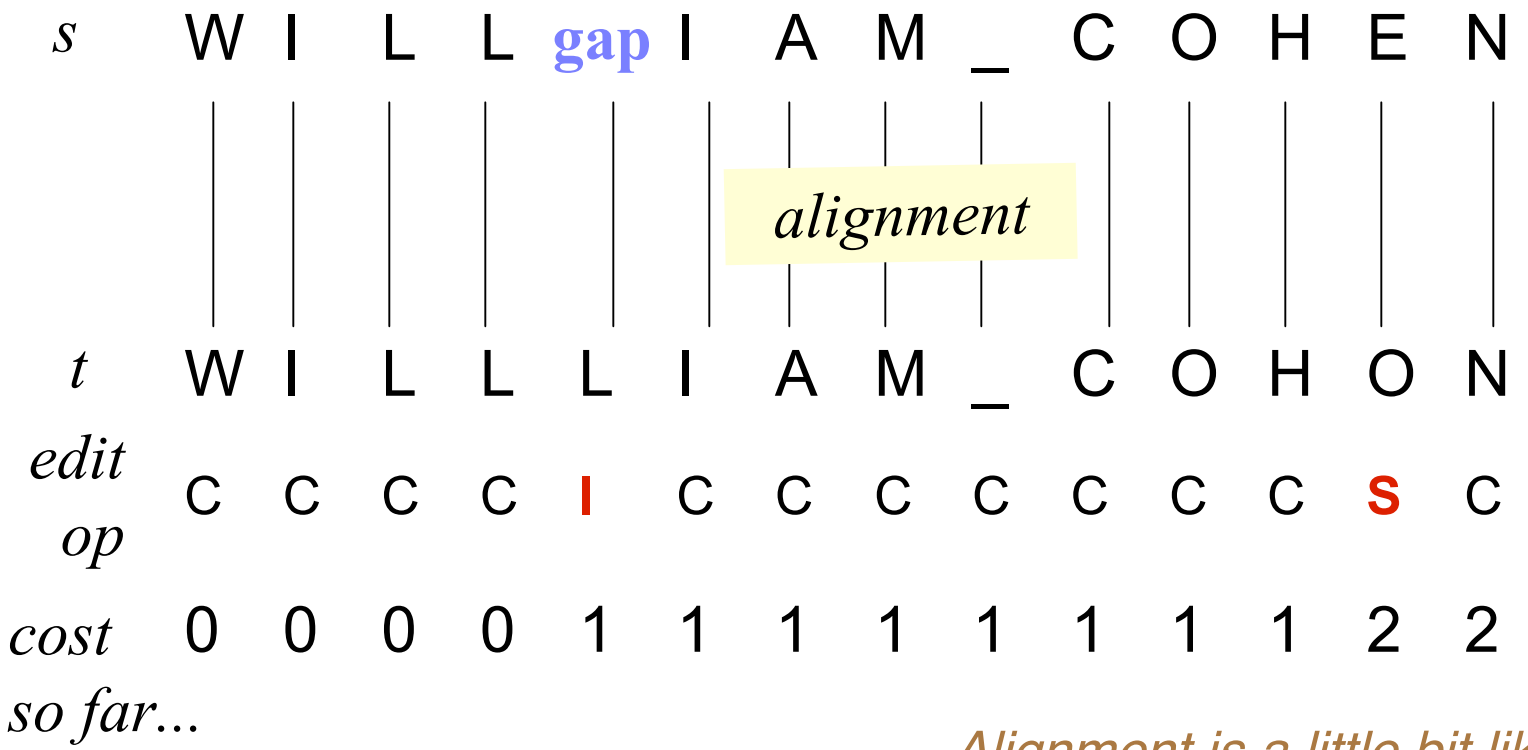
Distance = 2

String distance metrics: *Levenshtein*

- Given strings s and t
  - Distance is **shortest sequence of edit commands** that transform *s* to *t,* (or equivalently s to *t*).
  - Simple set of operations:
    - Copy character from *s* over to *t* (cost 0)
    - Delete a character in *s* (cost 1)
    - Insert a character in *t* (cost 1)
    - Substitute one character for another (cost 1)
- This is “Levenshtein distance”

Levenshtein distance - example

- distance("William Cohen", "Willliam Cohon")



Alignment is a little bit like a parse.

Finding the Minimum

What is the minimum number of operations for....?

```
Another fine day in the park
```

```
Anybody can see him pick the ball
```

Not so easy.... not so clear.

Not only are the strings, longer, but is isn't immediately obvious where the alignments should happen.

What if we consider all possible alignments by brute force? How many alignments are there?

Dynamic Program Table for String Edit

Measure distance between strings PARK

SPAKE

		P	A	R	K
S					
P					
A			c_{ij}		
K					
E					

c_{ij} = the number of edit operations needed to align PA with SPA.

Dynamic Programming to the Rescue!

How to take our big problem and chop it into building-block pieces.

- Given some partial solution, it isn't hard to figure out what a good next immediate step is.

- Partial solution =
  "This is the cost for aligning ***s*** up to position ***i*** with ***t*** up to position ***j***.
- Next step =
  "In order to align up to positions ***x*** in s and ***y*** in ***t***, should the last operation be a substitute, insert, or delete?

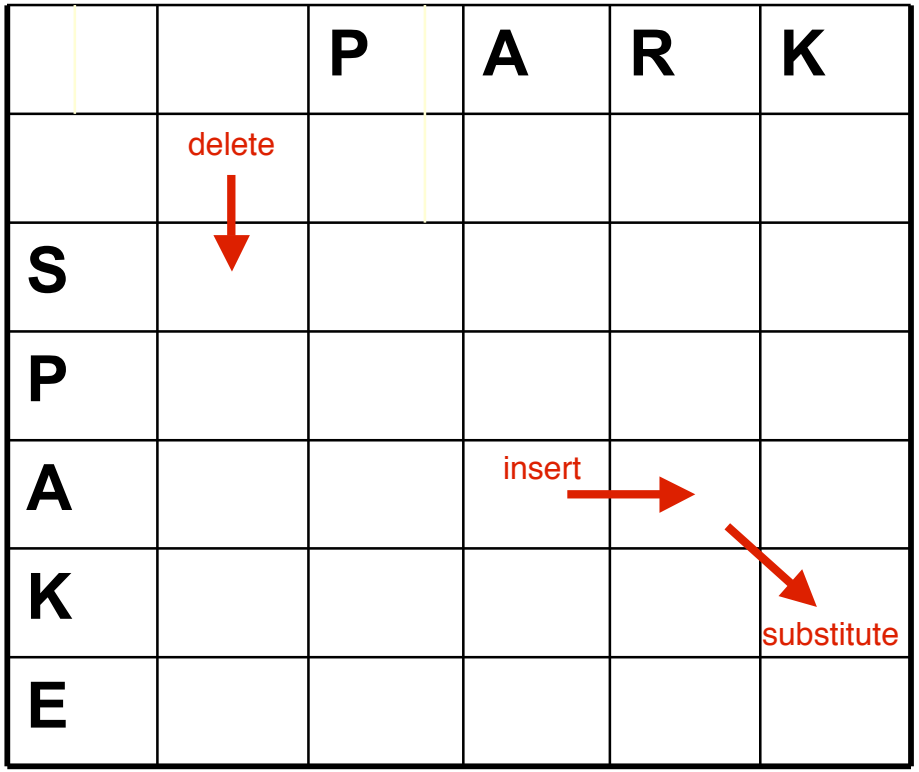
Dynamic Program Table for String Edit

Measure distance between strings

PARK

SPAKE

Edit operations for turning SPAKE into PARK



Andrew McCallum, UMass Amherst,
including material from William Cohen

Dynamic Program Table for String Edit

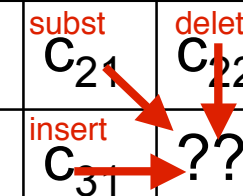
Measure distance between strings PARK

SPAKE

		P	A	R	K
	c_{00}	c_{02}	c_{03}	c_{04}	c_{05}
S	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}
P	c_{20}	c_{21}	c_{22}	c_{23}	c_{24}
A	c_{30}	c_{31}	???		
K					
E					

Dynamic Program Table for String Edit

		P	A	R	K
	c_{00}	c_{02}	c_{03}	c_{04}	c_{05}
S	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}
P	c_{20}	c_{21}	c_{22}	c_{23}	c_{24}
A	c_{30}	c_{31}	???		
K					
E					



D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} \text{D(i-1,j-1), if si=tj} & \textit{//copy} \\ \text{D(i-1,j-1)+1, if si!=tj} & \textit{//substitute} \\ \text{D(i-1,j)+1} & \textit{//insert} \\ \text{D(i,j-1)+1} & \textit{//delete} \end{cases}$$

Computing Levenshtein distance - 2

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1) + d(si,tj) & \textit{//subst/copy} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

(simplify by letting d(c,d)=0 if c=d, 1 else)

also let D(i,0)=i *(for i inserts)* and D(0,j)=j

Dynamic Program Table Initialized

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

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1)+d(si,tj) & //substitute \\ D(i-1,j)+1 & //insert \\ D(i,j-1)+1 & //delete \end{cases}$$

Dynamic Program Table ... filling in

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

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1)+d(si,tj) & \textit{//substitute} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

Dynamic Program Table ... filling in

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

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1)+d(si,tj) & \textit{//substitute} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

Dynamic Program Table ... filling in

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

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1)+d(si,tj) & \textit{//substitute} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

Dynamic Program Table ... filling in

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

Final cost of aligning all of both strings.

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1,j-1)+d(si,tj) & //substitute \\ D(i-1,j)+1 & //insert \\ D(i,j-1)+1 & //delete \end{cases}$$

DP String Edit Distance

```
def stredit (s1,s2):
    "Calculate Levenstein edit distance for strings s1 and s2."
    len1 = len(s1) # vertically
    len2 = len(s2) # horizontally
    # Allocate the table
    table = [None]*(len2+1)
    for i in range(len2+1): table[i] = [0]*(len1+1)
    # Initialize the table
    for i in range(1, len2+1): table[i][0] = i
    for i in range(1, len1+1): table[0][i] = i
    # Do dynamic programming
    for i in range(1,len2+1):
        for j in range(1,len1+1):
            if s1[j-1] == s2[i-1]:
                d = 0
            else:
                d = 1
            table[i][j] = min(table[i-1][j-1] + d,
                              table[i-1][j]+1,
                              table[i][j-1]+1)
```

Remebering the Alignment (trace)

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(si,tj) & \textit{//subst/copy} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

A *trace* indicates where the min value came from, and can be used to find edit operations and/or a best *alignment* (may be more than 1)

	C	O	H	E	N
M	1	2	3	4	5
C	1	2	3	4	5
C	2	3	3	4	5
O	3	2	3	4	5
H	4	3	2	3	4
N	5	4	3	3	3

Three Enhanced Variants

- Needleman-Munch
  - Variable costs
- Smith-Waterman
  - Find longest “soft matching” subsequence
- Affine Gap Distance
  - Make repeated deletions (insertions) cheaper

- (Implement one for homework?)

Needleman-Wunch distance

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(s_i,t_j) & \text{//subst/copy} \\ D(i-1,j) + G & \text{//insert} \\ D(i,j-1) + G & \text{//delete} \end{cases}$$

G = "gap cost"

d(c,d) is an arbitrary distance function on characters (e.g. related to typo frequencies, amino acid substitutibility, etc)

William Cohen

↓

Wukkuan Cigeb

Smith-Waterman distance

- Instead of looking at each sequence in its entirety, this compares segments of all possible lengths and chooses whichever maximize the similarity measure.

- For every cell the algorithm calculates all possible paths leading to it. These paths can be of any length and can contain insertions and deletions.

Smith-Waterman distance

$$D(i,j) = \min \begin{cases} \mathbf{0} & \textbf{//start over} \\ D(i-1,j-1) + d(si,tj) & \textit{//subst/copy} \\ D(i-1,j) + G & \textit{//insert} \\ D(i,j-1) + G & \textit{//delete} \end{cases}$$

$G = 1$

$d(c,c) = -2$

$d(c,d) = +1$

	C	O	H	E	N
M	0	0	0	0	0
C	**-2**	-1	0	0	0
C	**-2**	-1	0	0	0
O	-1	**-4**	-3	-2	-1
H	0	-3	**-6**	**-5**	-3
N	0	-2	-5	-5	**-7**

Example output from Python

		s	'	a	l	l	o	n	g	e	r
	0	1	2	3	4	5	6	7	8	9	10
l	1	0	0	0	0	* **0**	0	0	0	0	0
o	2	0	0	0	0	0	***-2**	-1	0	0	0
u	3	0	0	0	0	0	***-1**	-1	0	0	0
n	4	0	0	0	0	0	0	***-3**	-2	-1	0
g	5	0	0	0	0	0	0	-2	***-5**	-4	-3
e	6	0	0	0	0	0	0	-1	-4	***-7**	-6

(My implementation of HW#2, task choice #2. *-McCallum*)

Affine gap distances

- Smith-Waterman fails on some pairs that seem quite similar:

William W. Cohen

William W. ‘Don’t call me Dubya’ Cohen

Intuitively, single long insertions are “cheaper” than a lot of short insertions

Affine gap distances - 2

- Idea:
  - Current cost of a "gap" of *n* characters: *nG*
  - Make this cost: *A + (n-1)B,* where *A* is cost of "opening" a gap, and *B* is cost of "continuing" a gap.

Affine gap distances - 3

$$D(i,j) = \max \begin{cases} D(i\text{-}1,j\text{-}1) + d(si,tj) \\ IS(I\text{-}1,j\text{-}1) + d(si,tj) \\ IT(I\text{-}1,j\text{-}1) + d(si,tj) \end{cases}$$

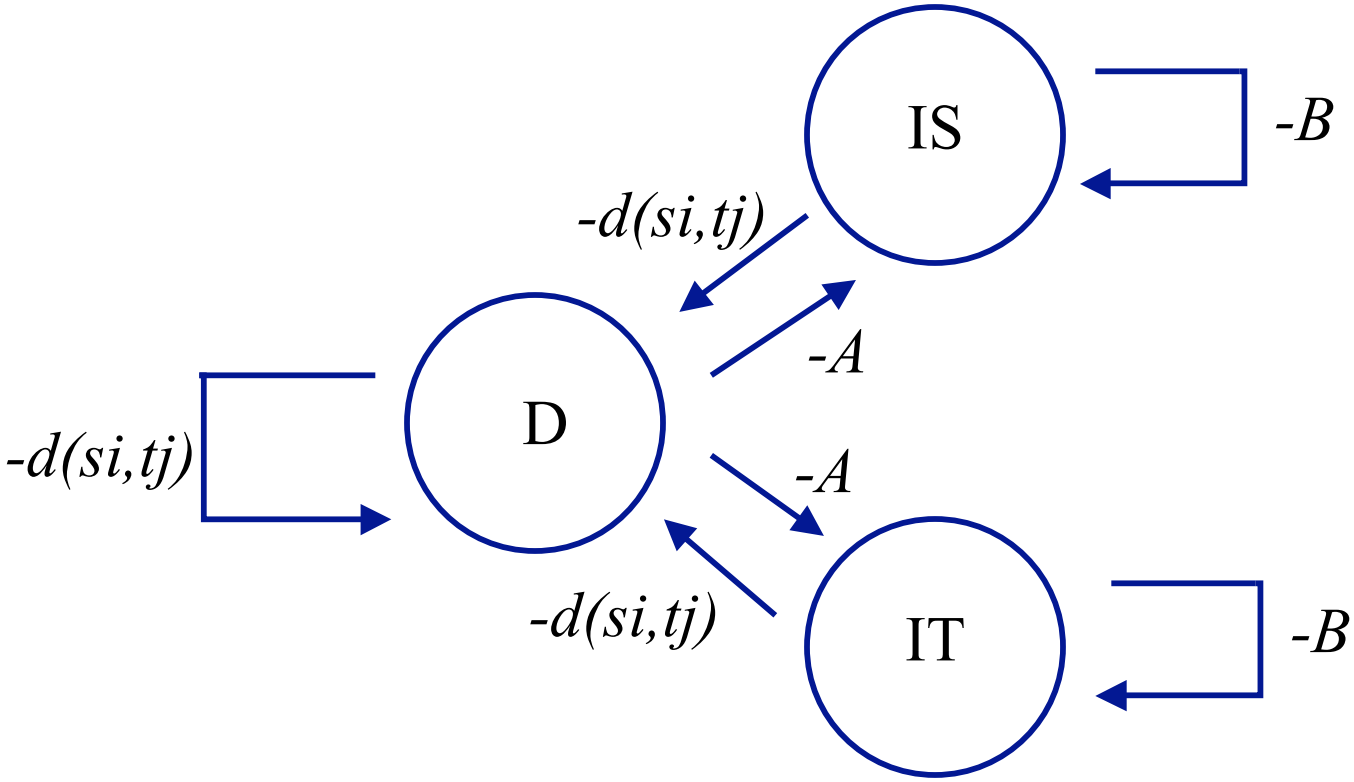
$$IS(i,j) = \max \begin{cases} D(i\text{-}1,j) - A \\ IS(i\text{-}1,j) - B \end{cases}$$

Best score in which si is aligned with a 'gap'

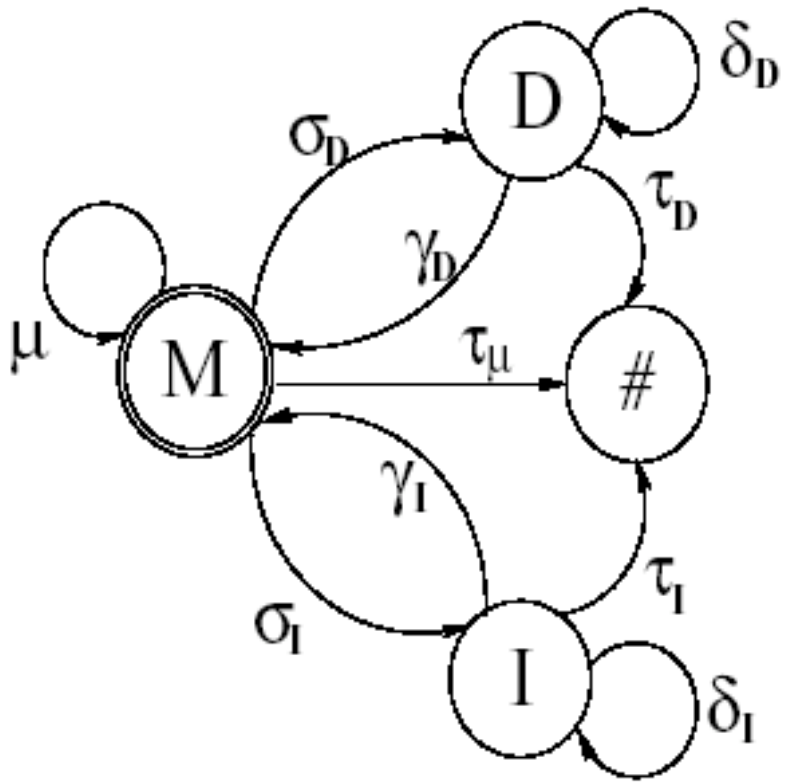
$$IT(i,j) = \max \begin{cases} D(i,j\text{-}1) - A \\ IT(i,j\text{-}1) - B \end{cases}$$

Best score in which tj is aligned with a 'gap'

Affine gap distances as automata



Generative version of affine gap automata (Bilenko&Mooney, TechReport 02)



HMM emits **pairs:** *(c,d)* in state *M*, pairs *(c,-)* in state *D*, and pairs *(-,d)* in state *I*.

For each state there is a **multinomial** distribution on pairs.

The HMM can trained with EM from a sample of pairs of **matched** strings *(s,t)*

E-step is forward-backward; M-step uses some *ad hoc* smoothing

Affine gap edit-distance learning: experiments results (Bilenko & Mooney)

Table 2: Sample duplicate records from the RESTAURANT database

name	address	city	phone	cuisine
Second Avenue Deli	156 2nd Ave. at 10th St.	New York	212/677-0606	Delicatessen
Second Avenue Deli	156 Second Ave.	New York City	212-677-0606	Delis

Table 3: Sample duplicate records from the MAILING database

first	last	street address	city
Tsy C	Dodgson	18 Lilammal Ave 3k1	Christina MT 59423
Tessy	Dodgeson	PO Box 3879	Christina MT 59428

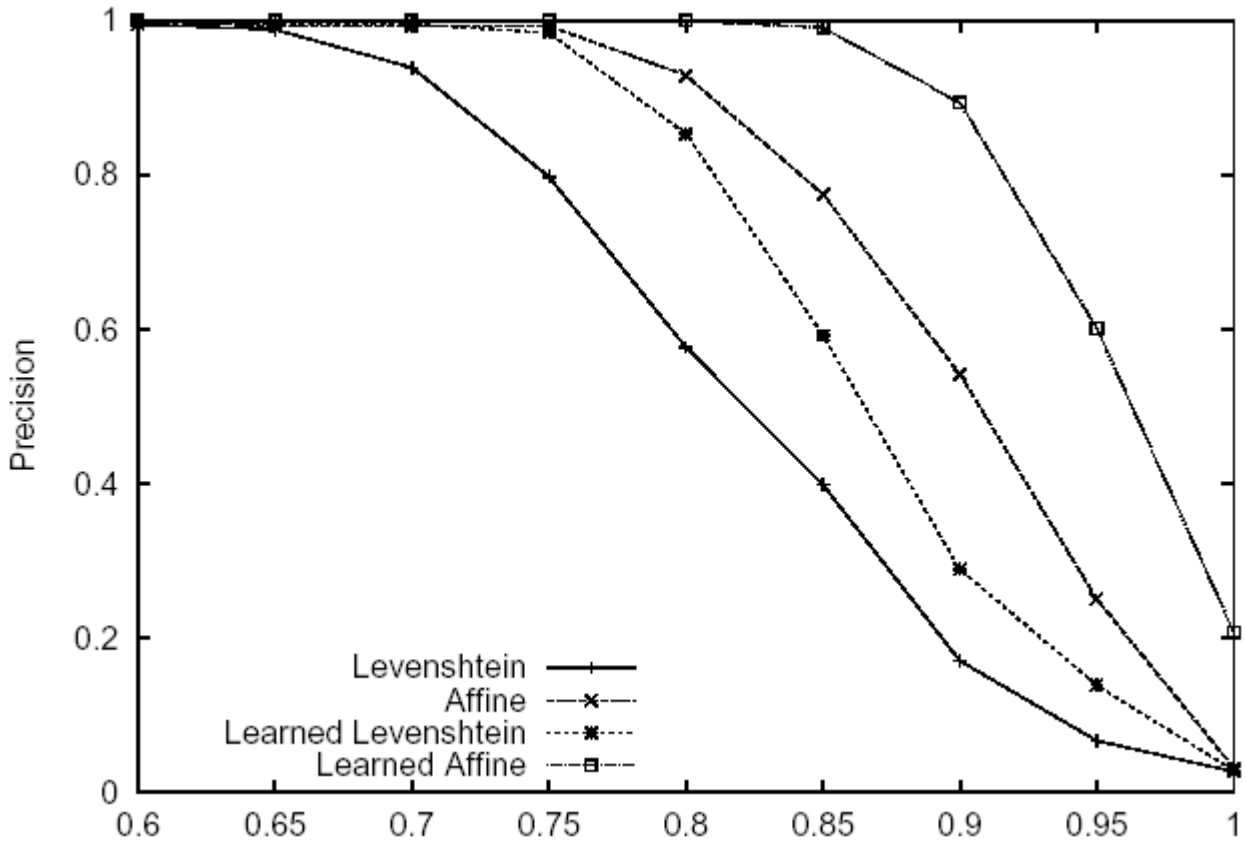
Experimental method: parse records into fields; append a few key fields together; sort by similarity; pick a threshold T and call all pairs with distance$(s,t) < T$ "duplicates"; picking T to maximize F-measure.

Affine gap edit-distance learning: experiments results (Bilenko & Mooney)

Distance metric	CORA title	RESTAURANT name
Levenshtein	0.870	0.843
Learned Levenshtein	0.902	**0.886**
Affine	0.917	0.883
Learned Affine	**0.971**	**0.967**

Distance met	RESTAURANT address	MAILING name	MAILING address
Levenshtein	0.950	0.867	0.878
Learned Leve	0.975	**0.899**	0.897
Affine	0.870	0.923	0.886
Learned Affi	**0.929**	**0.959**	0.892

Affine gap edit-distance learning: experiments results (Bilenko & Mooney)



Precision/recall for MAILING dataset duplicate detection

Affine gap distances – experiments
(from McCallum, Nigam,Ungar KDD2000)

- Goal is to match data like this:

Fahlman, Scott & Lebiere, Christian (1989). The cascade-correlation learning architecture. In Touretzky, D., editor, Advances in Neural Information Processing Systems (volume 2), (pp. 524-532), San Mateo, CA. Morgan Kaufmann.

Fahlman, S.E. and Lebiere, C., "The Cascade Correlation Learning Architecture," NIPS, Vol. 2, pp. 524-532, Morgan Kaufmann, 1990.

Fahlmann, S. E. and Lebiere, C. (1989). The cascade-correlation learning architecture. In Advances in Neural Information Processing Systems 2 (NIPS-2), Denver, Colorado.

Figure 2: Three sample citations to the same paper.

Homework #2

- The assignment
  - Start with my stredit.py code
  - Make some modifications
  - Write a little about your experiences
- Some possible modifications
  - Implement Needleman-Wunch, Smith-Waterman, or Affine Gap Distance.
  - Create a little spell-checker: if entered word isn't in the dictionary, return the dictionary word that is closest.
  - Change implementation to operate on sequences of *words* rather than characters... get an online translation dictionary, and find alignments between English & French or English & Russian!
  - Try to *learn* the parameters of the function from data. (Tough.)