Approximate String Matching

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

From exact to approximate string matching

Often in applications we want to search a text for something that is similar to the pattern but not necessarily exactly the same.

This is the case, for example, when we want to search for a word in a text taking possible typos into account; or when we want to map a read in a genome taking into account sequencing errors.

To formalize this problem, we have to specify what does “similar” mean. This can be done by defining a distance measure for strings.

From exact to approximate string matching

There are several possible ways to define distances between strings. Let us start with the simplest notion, which is a measure of distance between two strings of the same length.

Given two strings S and T, both of length n, the Hamming distance between S and T is the number of positions i such that S[i]≠T[i]. The following two strings have Hamming distance 3.

TATGTTACAA
AATCTTACAC

Computing the Hamming distance between S and T requires (trivially) O(n) time.

The Hamming distance

Hamming distance is a metric on Σ^n, for any fixed alphabet Σ and string length n. Let us denote by $d_H(S,T)$ the Hamming distance between strings S and T, both in Σ^n.

- $d_H(S,T) \geq 0$ for any S,T
- $d_H(S,T) = d_H(T,S)$
- $d_H(S,T) = 0$ if and only if S=T
- $d_H(S,T) \leq d_H(S,U) + d_H(U,T)$. Indeed, for positions i s.t. $S[i] \neq T[i]$, it must be either $S[i] \neq U[i]$ or $U[i] \neq T[i]$ (or both).

```
S=TATGTTACAA
  |||x|||x||
U=TATCTTAGAA
  x||||||x|x
T=AATCTTACAC
```

$d_H(S,T)=3$; $d_H(S,U)=2$; $d_H(U,T)=3$

The k-mismatch problem

Reference: Chapters 9.1 and 9.4 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

The k-mismatch problem

IN: a text T of length n, a pattern P of length m<n, an integer k<m

OUT: all positions i in T such that $d_H(T[i..i+|P|-1],P) \leq k$

```
        7 10 13 16
T=AMBARABACCICCICCOCCO ; P=COCCO ; k=3
        xx  x
        COCCO
            x  x
           COCCO
               x
              COCCO
                 COCCO
```

Output: {7,10,13,16}

The k-mismatch problem

Theorem. Given a text T of length n, a pattern P of length m<n, and an integer k<m, the k-mismatch problem can be solved in O(kn) time, after an O(n+m)-time preprocessing.

Longest Common Extension queries

Problem: preprocess two strings S and T such that the following queries can be answered efficiently.

Query: given a position i in S and a position j in T, find the length of the longest common prefix (called Longest Common Extension) of S[i..|S|] and T[j..|T|], denoted by $LCE_{S,T}(i,j)$.

Example. Let S=abracadabra, T=bracco. Then:

$LCE_{S,T}(2,1)=4$

```
abracadabra
 bracco
```

$LCE_{S,T}(6,3)=1$

```
abracadabra
   bracco
```

$LCE_{S,T}(7,4)=0$

```
abracadabra
   bracco
```

Longest Common Extension queries

Theorem. Given a string S of length n and a string T of length m, LCE queries can be answered in O(1) time after an O(n+m)-time preprocessing.

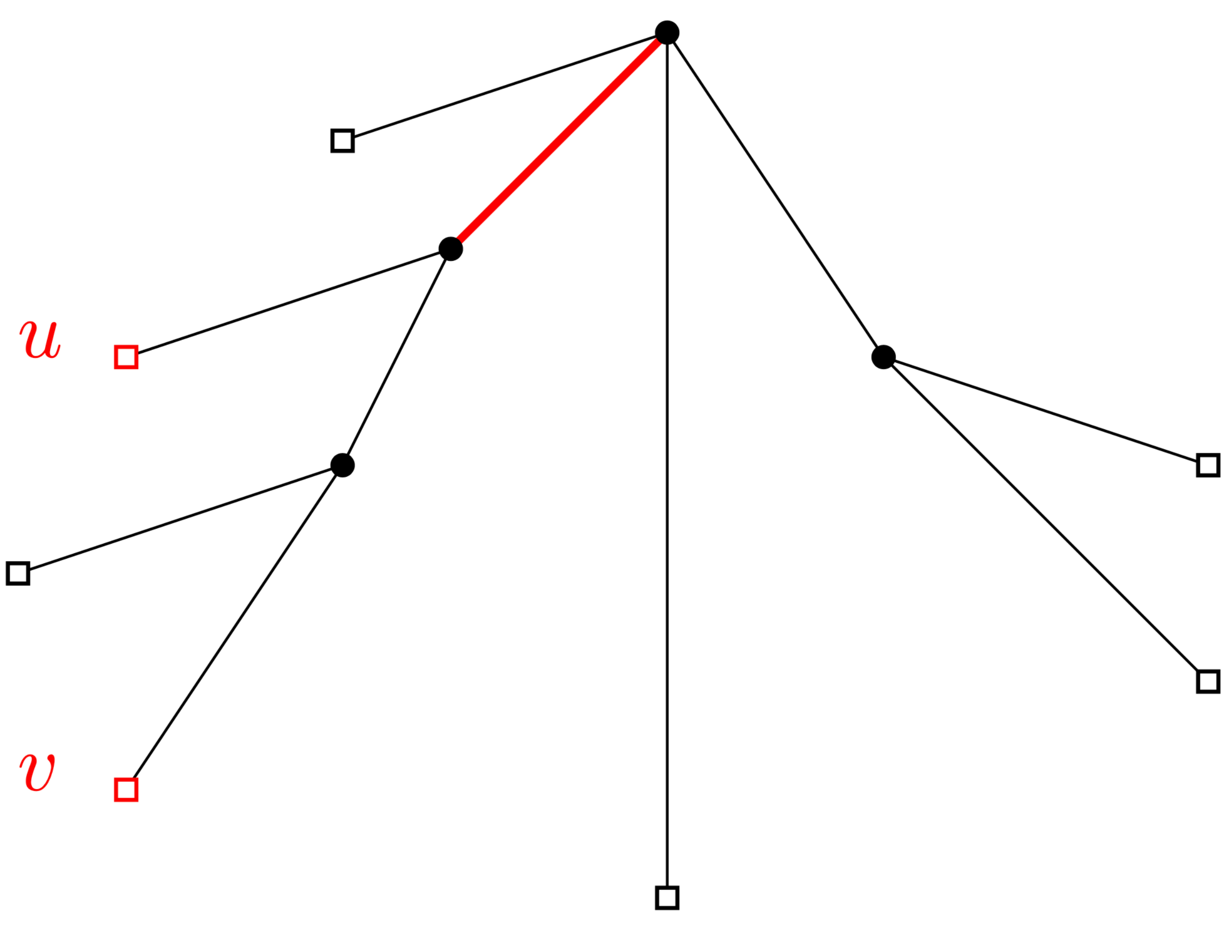
The preprocessing required is to build the generalised suffix tree of S and T, and then to preprocess it to allow constant-time LCA queries.

Then $LCE_{S,T}(i,j)$ is equal to the string depth of $LCA(u_i,v_j)$, where u_i is the leaf of the generalised suffix tree corresponding to $S[i..|S|]$, v_j is the leaf of the generalised suffix tree corresponding to $T[j..|T|]$.

Lowest Common Ancestor queries

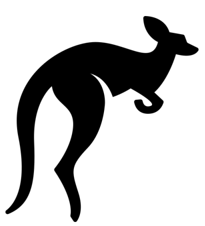
The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.

Theorem (Bender and Farach-Colton). Any tree of size O(N) can be preprocessed in O(N) time so that the LCA of any two nodes can be computed in O(1) time.



The kangaroo algorithm for k-mismatch

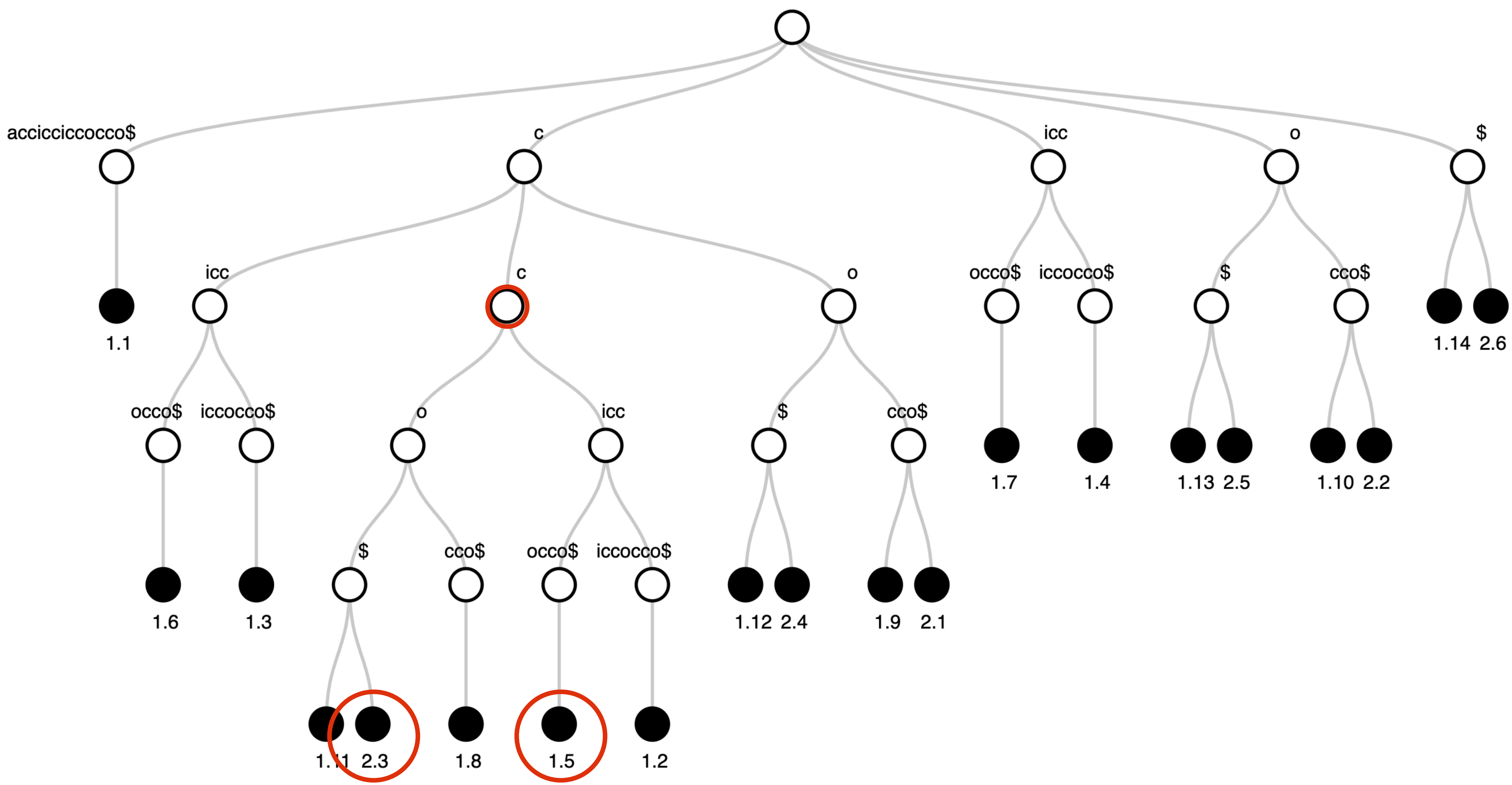
```
kMISMATCH(T,P,k)
    sol ← ∅;
    for all i=1,…,|T|
        count←0; match←0;
        while count≤k and match+count<|P|
            ext←LCE_{T,P}( match+count+i , match+count+1 );
            match←match + ext;
            if match+count=|P|
                sol.append(i);
            else
                count←count +1;
    return sol;
```



The k-mismatch problem

T=ACCICCICCOCCO ; P=COCCO ; k=3

$LCE_{T,P}(5,3)=2$



The edit distance problem

Reference: Chapters 11.2 and 11.3 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

The edit distance between two strings

The most widely used notion of distance between strings is a measure of distance between two strings of any lengths that focuses on transforming one string into the other with a series of edit operations on individual characters.

The permitted operations are deleting a character from the first string, (denoted by D) inserting a character in the first string (I), or replacing a character of the first string with another (R).

Given two strings S and T, both of length n, the edit distance between S and T is the minimum number of character deletions, insertions and replacements to transform S into T.

The following two strings have edit distance 5.

```
S=VINT NER
T= INTEREST
  D   IR RI
```

The edit distance between two strings

Note that there may be multiple sequences of edit operations of minimum length that transform S into T: the edit distance is just the minimum length.

```
S=VINT NER          S=VINT NE R          S=VINTNER
T= INTEREST         T= INTEREST          T= INT EREST
  D    IR RI          D    IR IR           D    D   III
```

The edit distance between two strings

Note that there may be multiple sequences of edit operations of minimum length that transform S into T: the edit distance is just the minimum length.

```
S=VINT NER          S=VINT NE R         S=VINTNER
T= INTEREST         T= INTEREST         T= INT EREST
  DMMMIRMRI           DMMMIRMIR           DMMMDMMIII
```

Using “M” to denote a non-operation “match” in addition to the symbols of the three edit operations “I”, “D”, “R”, a string over the alphabet {M,I,D,R} that describes a transformation of S into T is called an edit transcript of the two strings.

The edit distance problem

Given two strings S and T, the edit distance problem is to compute the edit distance between S and T together with an optimal edit transcript that describes a minimum-length transformation.

Theorem. The edit distance between a string S and a string T can be computed in $O(|S||T|)$ time and space.

This problem can be solved with a dynamic programming algorithm.

Computing edit distance: recursion

Given strings S of length n and T of length m, we denote by D(i,j) the edit distance between S[1..i] and T[1..j]. D(n,m) denotes the edit distance between the whole S and T.

Base conditions: D(i,0)=i (i deletions) and D(0,j)=j (j insertions).

Let d:[1,n]x[1,m]→{0,1} a function such that d(i,j)=1 if S[i]≠S[j], d(i,j)=0 otherwise. Then it holds the following

Recursion: D(i,j)=min{ D(i-1,j)+1 , D(i,j-1)+1 , D(i-1,j-1)+d(i,j) } for any i∈[1,n], j∈[1,m].

Computing edit distance: recursion

Lemma 1. For any i∈[1,n], j∈[1,m], D(i,j) is either D(i-1,j)+1, D(i,j-1)+1, or D(i-1,j-1)+d(i,j). There are no other possibilities.

Proof. Consider an optimal transcript for S[1..i] and T[1..j], and focus on the last operation. There are four cases.

1. The last operation is the insertion of T[j] at the end of the transformed S[1..i]. Then the transcript before the last symbol I gives the minimum number of operations to transform S[1..i] into T[1..j-1], and this number is precisely D(i,j-1). Adding 1 for the last insertion, we obtain that D(i,j)=D(i,j-1)+1.

```
S=VINT
T= INTE
  DMMMI
```

i=4,j=4, D(4,4)=2, D(4,3)=1

Computing edit distance: recursion

Lemma 1. For any i∈[1,n], j∈[1,m], D(i,j) is either D(i-1,j)+1, D(i,j-1)+1, or D(i-1,j-1)+d(i,j). There are no other possibilities.

Proof. Consider an optimal transcript for S[1..i] and T[1..j], and focus on the last operation. There are four cases.

2. The last operation is the deletion of S[i]. Then the transcript before the last symbol D gives the minimum number of operations to transform S[1..i-1] into T[1..j], and this number is precisely D(i-1,j). Adding 1 for the last deletion, we obtain that D(i,j)=D(i-1,j)+1.

```
S=VINTN
T= INT
  DMMMD
```

i=5,j=3, D(5,3)=2, D(4,3)=1

Computing edit distance: recursion

Lemma 1. For any i∈[1,n], j∈[1,m], D(i,j) is either D(i-1,j)+1, D(i,j-1)+1, or D(i-1,j-1)+d(i,j). There are no other possibilities.

Proof. Consider an optimal transcript for S[1..i] and T[1..j], and focus on the last operation. There are four cases.

3. The last operation is the replacement of S[i] with T[j]. Then the transcript before the last symbol R gives the minimum number of operations to transform S[1..i-1] into T[1..j-1], and this number is precisely D(i-1,j-1). Adding 1 for the last replacement, we obtain that D(i,j)=D(i-1,j-1)+1.

```
S=VINT N
T= INTER
  DMMMIR
i=5,j=5, D(5,5)=3, D(4,4)=2
```

Computing edit distance: recursion

Lemma 1. For any i∈[1,n], j∈[1,m], D(i,j) is either D(i-1,j)+1, D(i,j-1)+1, or D(i-1,j-1)+d(i,j). There are no other possibilities.

Proof. Consider an optimal transcript for S[1..i] and T[1..j], and focus on the last operation. There are four cases.

4. Finally, if the last symbol is the match S[i]=T[j]. Then D(i,j)=D(i-1,j-1).

```
S=VINT
T= INT
  DMMM
```

i=4,j=3, D(4,3)=1, D(3,2)=1

Computing edit distance: recursion

Lemma 2. For any $i \in [1,n]$, $j \in [1,m]$, $D(i,j) \leq \min\{ D(i-1,j)+1 , D(i,j-1)+1 , D(i-1,j-1)+d(i,j) \}$

Proof. With the same reasoning of the proof of Lemma 1, it suffices to show that for each case there exists a transformation achieving each of the three values specified in the lemma statement.

Computing edit distance: dynamic programming

The dynamic programming algorithm for computing edit distance consists in computing all values D(i,j) bottom-up, starting from the smallest possible i and j and storing the computed values in a dynamic programming table that has the letters of S at the columns and the letters of T at the rows (plus an extra row and column to account for i=0 and j=0).

Computing edit distance: dynamic programming

Column and row 0 are filled in using the base conditions.

		S	u	n	d	a	y
	0	1	2	3	4	5	6
S	1						
a	2						
t	3						
u	4						
r	5						
d	6						
a	7						
y	8						

Computing edit distance: dynamic programming

Column and row 0 are filled in using the base conditions. The other cells are filled in using the recursive relation.

		S	u	n	d	a	y
	0	1	2	3	4	5	6
S	1	0	1				
a	2	1					
t	3						
u	4						
r	5						
d	6						
a	7						
y	8						

Computing edit distance: dynamic programming

Column and row 0 are filled in using the base conditions. The other cells are filled in using the recursive relation. The result is in the bottom-right cell.

		S	u	n	d	a	y
	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
a	2	1	1	2	3	3	4
t	3	2	2	2	3	4	4
u	4	3	2	3	3	4	5
r	5	4	3	3	4	4	5
d	6	5	4	4	3	4	5
a	7	6	5	5	4	3	4
y	8	7	6	6	5	4	**3**

Computing edit distance: traceback

In order to reconstruct an optimal transcript, it suffices to store some pointers when computing the table: when computing D(i,j) we store a pointer from cell (i,j) to cell (i-1,j) if D(i,j)=D(i-1,j)+1; we store a pointer to cell (i,j-1) if D(i,j)=D(i,j-1)+1; we store a pointer to cell (i-1,j-1) if D(i,j)=D(i-1,j-1)+d(i,j).

We can then follow any pointer path from cell (m,n) to cell (0,0). This way we reconstruct a transcript backwards, writing an I every time we follow a vertical pointer, a D every time we follow a horizontal pointer, and a R or a M when we follow a diagonal pointer, depending on the value of function d.

Computing edit distance: dynamic programming

The optimal transcript highlighted in grey is MIIMRMMM, corresponding to the alignment

```
S  unday
Saturday
MIIMRMMM
```

		S	u	n	d	a	y
	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
a	2	1	1	2	3	3	4
t	3	2	2	2	3	4	4
u	4	3	2	3	3	4	5
r	5	4	3	3	4	4	5
d	6	5	4	4	3	4	5
a	7	6	5	5	4	3	4
y	8	7	6	6	5	4	**3**