Algorithms on Strings, Trees, and Sequences

Part III
Chapter 11.
Core String Edits, Alignments, and Dynamic Programming

컴퓨터소프트웨어학부 심승현

Contents

- Edit distance between two strings
- String alignment
- Dynamic programming calculation of edit distance
- Edit graphs
- Weighted edit distance
- String similarity
- Local alignment
- Gaps

Summary : Chapter 10

▶ The first fact of biological sequence analysis

In biomolecular sequences, (e.g. DNA, RNA, amino acid sequences)
High sequence similarity usually implies significant functional or structural similarity

- Redundancy, and similarity is the important feature

Summary : Chapter 10

- The first fact of biological sequence analysis

> In biomolecular sequences, (e.g. DNA, RNA, amino acid sequences)
> High sequence similarity usually implies significant functional or structural similarity

- The converse of first fact is not true, i.e. the high similarity of function or structure doesn’t necessarily mean high sequence similarity.

Chapter 11.

Core String Edits, Alignments, and Dynamic Programming

Introduction

- Chapter 11 is about the inexact matching, alignment problems, and some other relative techniques.

Introduction

- Chapter 11 is about the inexact matching, alignment problems, and some other relative techniques.
  - Inexact matching is to find strings, or sequences that approximately matches to another strings or sequences
  - Alignment problem is to align some strings, to find mutual similarity or redundance

The Edit Distance Between Two Strings

- Edit distance between two strings means “ How far is from one string to other string? “

  Measures the converting step from one string to the other string

The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : ‘credit’
- S_2 : ‘greedy’

R	M	I	M	M	R	D
c	r		e	d	i	t
g	r	e	e	d	y	

The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'

R	M	I	M	M	R	D
c	r		e	d	i	t
g	r	e	e	d	y	

→ This is called edit transcript !

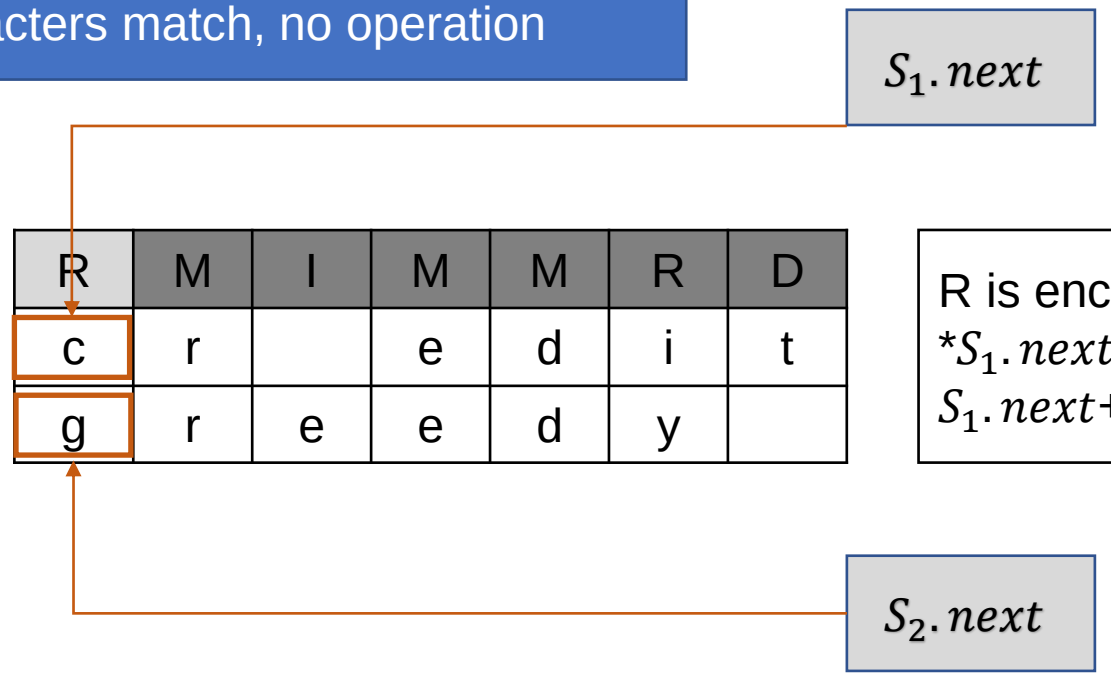
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



R is encountered
$*S_1.next = *S_2.next$
$S_1.next{+}{+}, S_2.next{+}{+}$

- These two pointers are pointing S_1 and S_2 each, we aligned them as we can see in the picture, but S_1 and S_2 do not include space we can see in the picture actually.

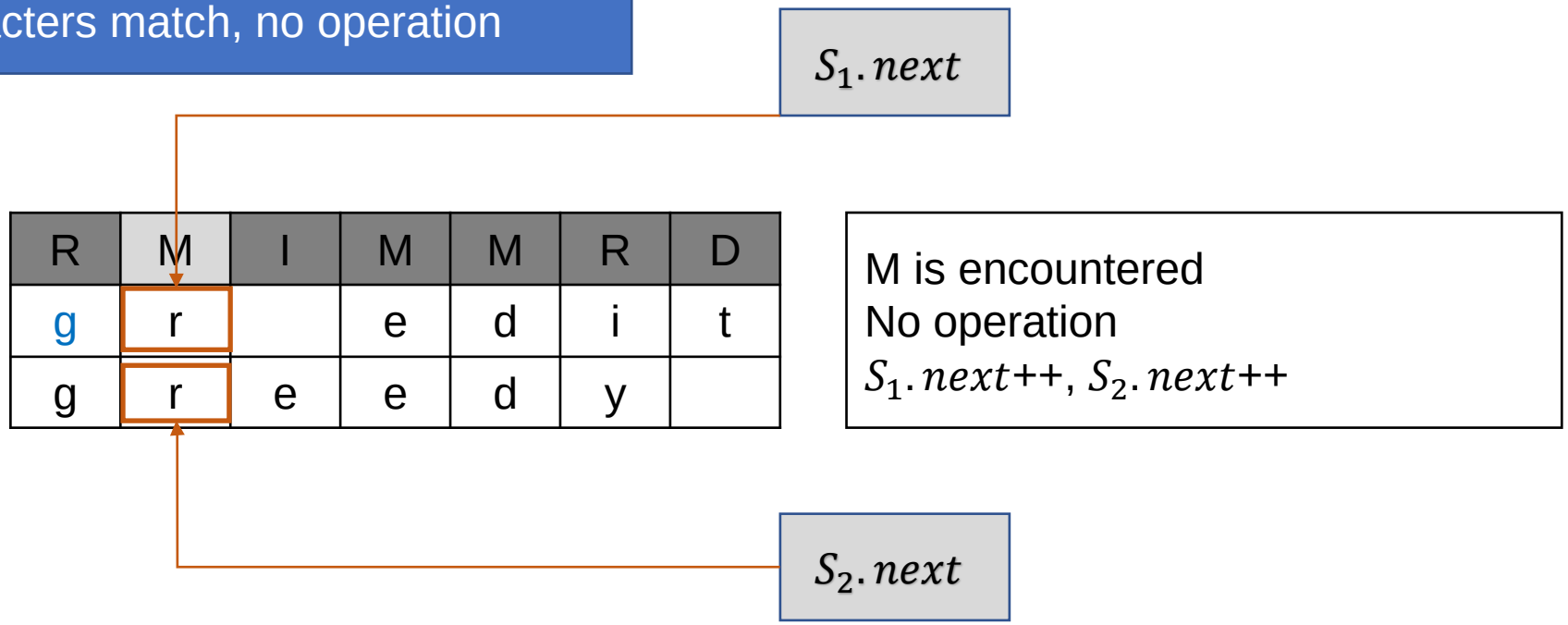
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



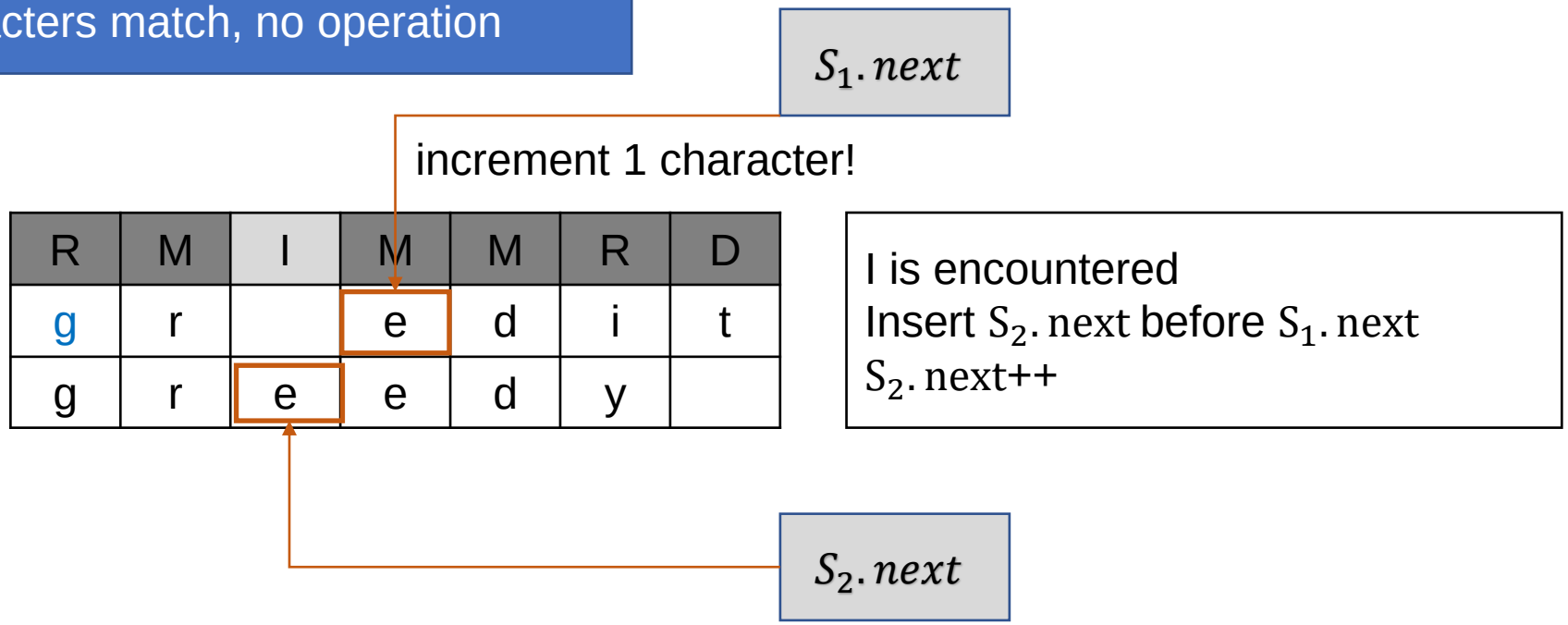
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



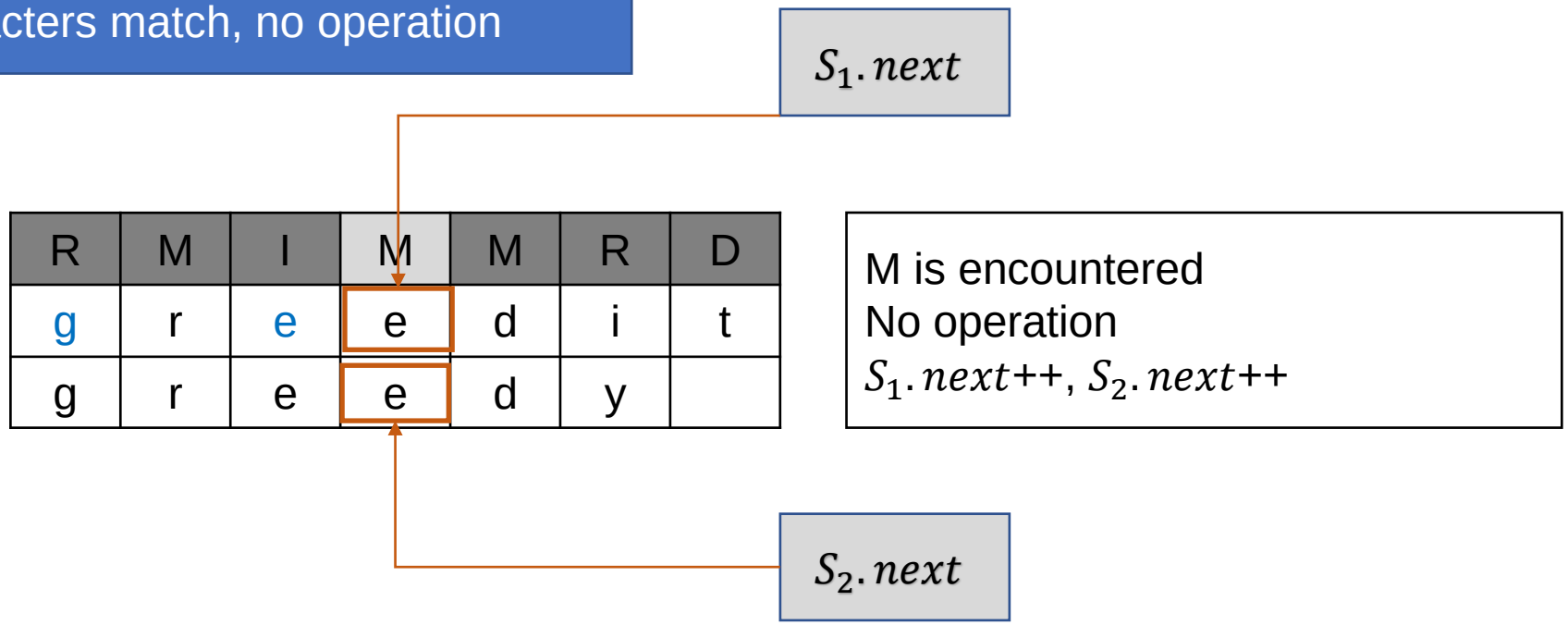
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



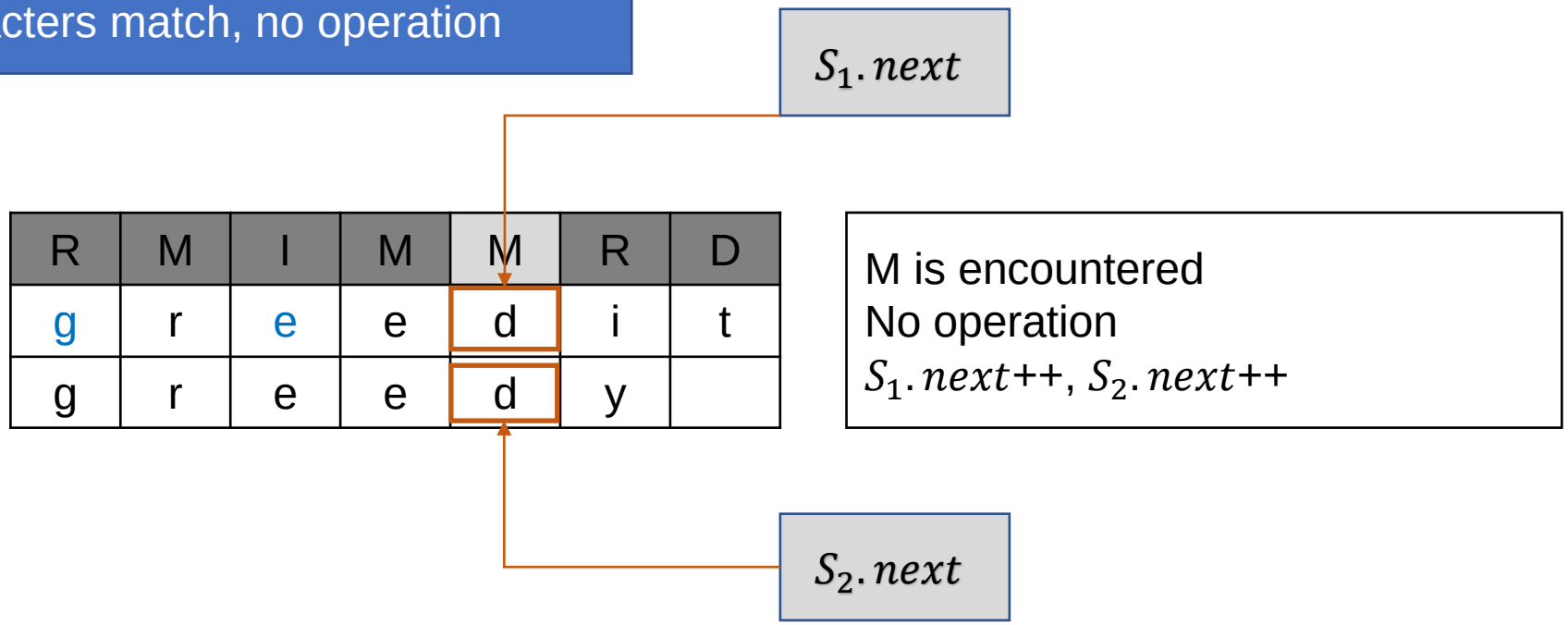
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'

R	M	I	M	M	R	D
g	r	e	e	d	i	t
g	r	e	e	d	y	

$S_1.next$

$S_2.next$

R is encountered
$*S_1.next = *S_2.next$
$S_1.next{+}{+}, S_2.next{+}{+}$

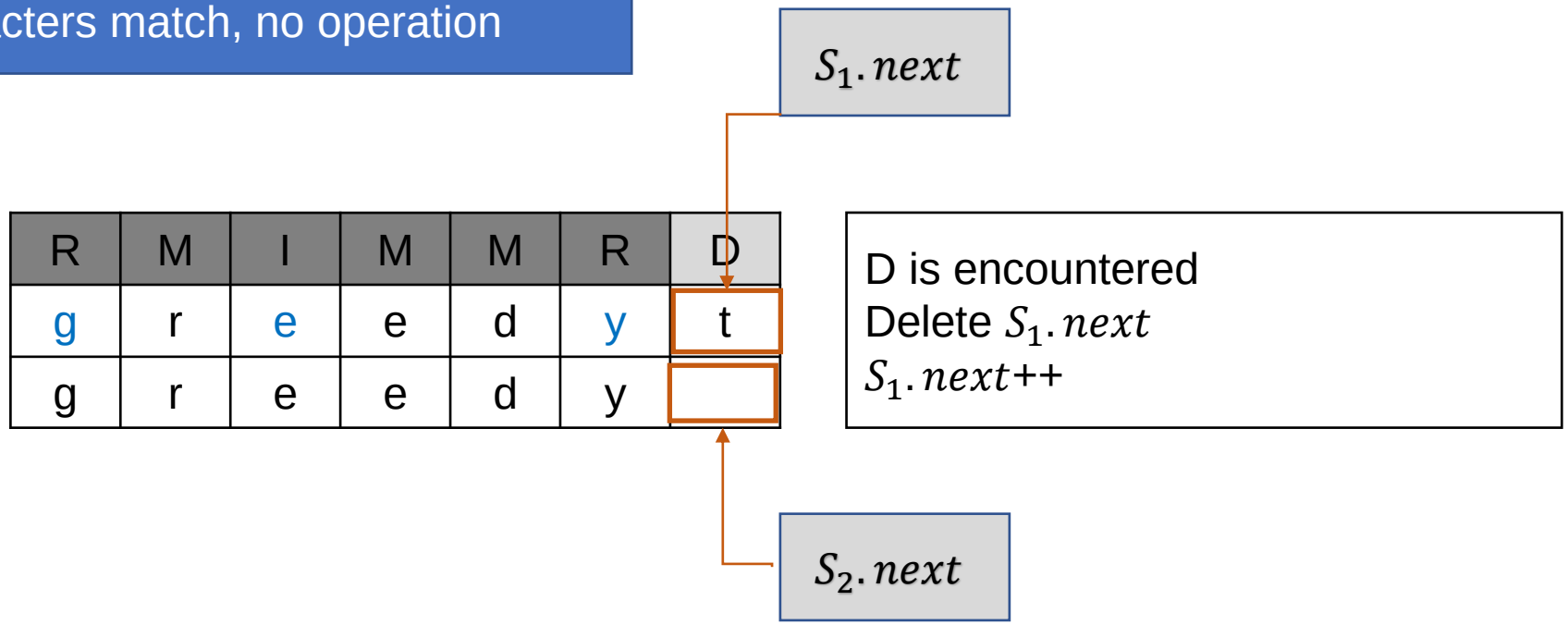
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



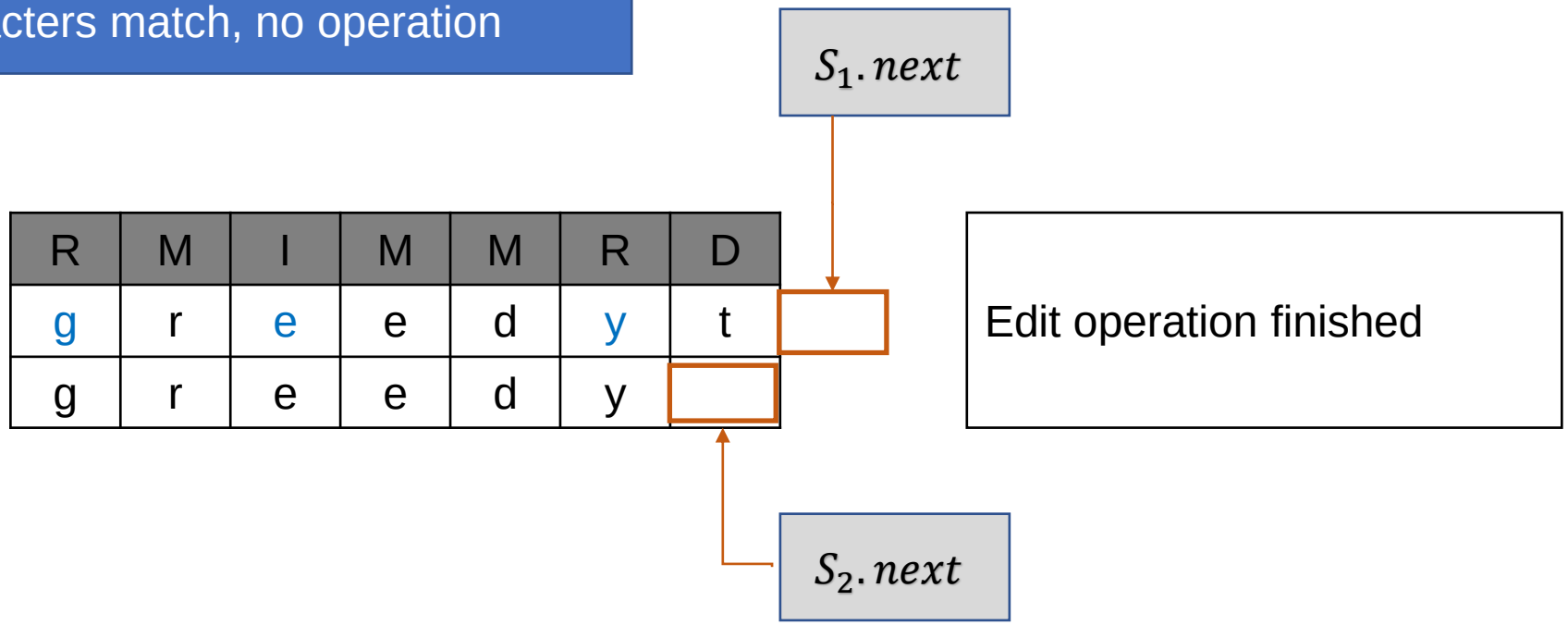
The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'



The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'

R	M	I	M	M	R	D
c	r		e	d	i	t
g	r	e	e	d	y	

→ This is called edit transcript !

- Edit distance between two strings is the minimum number of edit operations

We define optimal transcript as an edit transcript that uses the minimum number of edit operations

The Edit Distance Between Two Strings

There are 4 kinds of edit operation

I : Insert character of S_2 to S_1
D : Delete character in S_1
R : Replace character in S_1 with character in S_2
M : If two characters match, no operation

< Example >

- S_1 : 'credit'
- S_2 : 'greedy'

R	M	I	M	M	R	D
c	r		e	d	i	t
g	r	e	e	d	y	

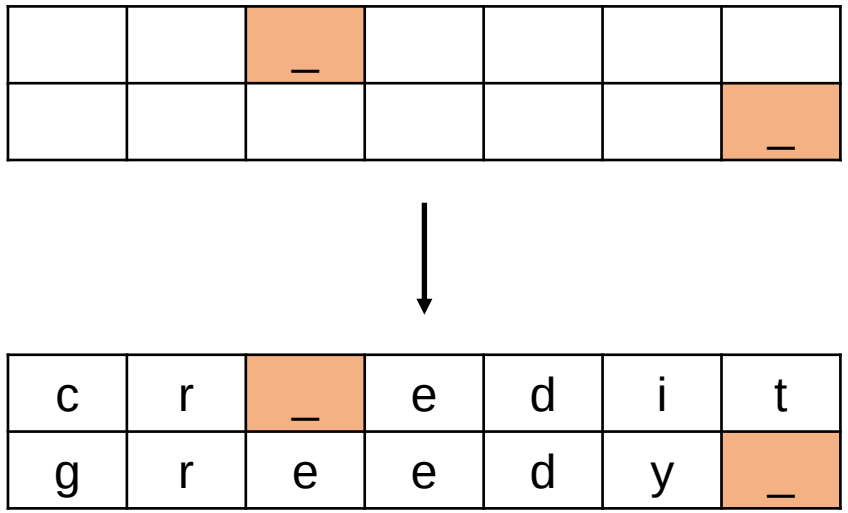
→ This is called edit transcript !

- There are various ways aligning these strings, there can be multiple optimal transcripts

  We call them cooptimal transcripts

String Alignments

- Given two string S_1 and S_2,
  A (global) alignment of S_1 and S_2 is derived by first inserting spaces or dashes, then put strings that each characters in string opposite the other string's characters



- c opposite g
- e and t opposite space

Comparing Edit Transcript and String Alignments

- Mathematically, they are identical

- In terms of model,
  Edit transcript focuses on mutational events, that S_1 transforms into S_2
  Alignment only shows just a relationship between two strings.

Dynamic Programming Calculation of Edit Distance

- Computing edit distance of two strings
  - Dynamic programming approach, to divide a problem and solve through them
  - Specific application to edit distance problem

Dynamic Programming Approach

- The recurrence relation
- Tabular computation of edit distance
- The traceback

Dynamic Programming Approach

- The recurrence relation
- Tabular computation of edit distance
- The traceback

Before we start, we need to define..

Given two strings S_1 and S_2,
$D_{(i,j)}$ is defined to be the edit distance of $S_1[1..i]$ and $S_2[1..j]$

Dynamic Programming Approach : The Recurrence Relation

- Base condition
- Recurrence relation

Dynamic Programming Approach : The Recurrence Relation

▶ Base condition

$$D_{(i,0)} = i$$
$$D_{(0,j)} = \text{j}$$

$D_{(6,0)}$
S_1 = ‘abcdef’

	D	D	D	D	D	D
S_1	a	b	c	d	e	f
S_2	_	_	_	_	_	_

▶ We need one delete for every character in S_1
In this case, $D_{(6,0)}$ is 6

Dynamic Programming Approach : The Recurrence Relation

▶ Recurrence relation

$$D_{(i,j)} = \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

Dynamic Programming Approach : The Recurrence Relation

▶ Recurrence relation

$$D_{(i,j)} = \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

This relation is built on these two lemmas,

Lemma 11.3.1 : $D_{(i,j)}$ is one of $[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

Lemma 11.3.2 : $D_{(i,j)} \leq \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.1

$D_{(i,j)}$ is one of $[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

$\mathrm{i} = 4, j = 3$, symbol = D

...	...	...	...	...	...	...
S_1	...	...	...	...	...	...
S_2	...	...	...	...	...	...

D is encountered
Delete $S_1.next$
$S_1.next$++

When D is encountered, we move only S_1's pointer

So maybe the previous step was deriving $D_{(3,3)}$.
We have edit operation D, so $D_{(4,3)}$ is
$D_{(4,3)} = D_{(3,3)} + 1$

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.1

$D_{(i,j)}$ is one of $[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

$i = 4, j = 3$, symbol = I

...	...	...	...	...	...	...
S_1	...	...	...	...	...	...
S_2	...	...	...	...	...	...

I is encountered
Insert $S_2.\text{next}$ before $S_1.\text{next}$
$S_2.\text{next}$++

When I is encountered, we move only S_2's pointer

So maybe the previous step was deriving $D_{(4,2)}$.
We have edit operation I, so $D_{(4,3)}$ is
$D_{(4,3)} = D_{(4,2)} + 1$

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.1

$D_{(i,j)}$ is one of $[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

$i = 4, j = 3$, symbol = M

...	...	...	...	...	...	...
S_1	...	...	...	...	...	...
S_2	...	...	...	...	...	...

M is encountered
No operation
$S_1.next{+}{+}$, $S_2.next{+}{+}$

When M is encountered, we move pointer of S_1 and S_2

So maybe the previous step was deriving $D_{(3,2)}$.
We have no operation here, so $D_{(4,3)}$ is
$D_{(4,3)} = D_{(3,2)}$

$S_1(i) = S_2(j)$ when M encountered,
so it is case of $t_{(i,j)}$ = 0 above

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.1

$D_{(i,j)}$ is one of $[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

When R is encountered, we move pointer of S_1 and S_2

So maybe the previous step was deriving $D_{(3,2)}$.
We have no operation here, so $D_{(4,3)}$ is
$D_{(4,3)} = D_{(3,2)} + 1$

$S_1(i) \neq S_2(j)$ when R encountered,
so it is case of $t_{(i,j)}$ = 1 above

$i = 4, j = 3$, symbol = R

...	...	...	...	...	...	...
S_1	...	...	...	...	...	...
S_2	...	...	...	...	...	...

R is encountered
$*S_1.next = *S_2.next$
$S_1.next{+}{+}$, $S_2.next{+}{+}$

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.1

$$D_{(i,j)} \text{ is one of } [D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

In edit transcript, we only had these four edit operations, so

$$D_{(i,j)} \text{ is one of } [D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.2

$$D_{(i,j)} \leq \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

Index	...	...	i-1	i	...	j-1	j	...
S_1	...	...	$S_1(i-1)$	$S_1(i)$	...	$S_1(j-1)$	$S_1(j)$	...
S_2			$S_2(i-1)$	$S_2(i)$		$S_2(j-1)$	$S_2(j)$	

▶ Calculating $D_{(i,j)}$, we can first calculate $D_{(i-1,j)}$, then delete $S_1(i)$ with result if the last symbol is D
or, calculate $D_{(i,j-1)}$, then insert $S_2(j)$ with result if the last symbol is I
or, calculate $D_{(i-1,j-1)}$, then insert $S_1(i)$ and $S_2(j)$,
edit distance is 0 when the last symbol is M, 1 when the last symbol is R

Dynamic Programming Approach : The Recurrence Relation

▶ Lemma 11.3.2

$$D_{(i,j)} \leq \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

Index	...	...	i-1	i	...	j-1	j	...
S_1	...		$S_1(i-1)$	$S_1(i)$		$S_1(j-1)$	$S_1(j)$	..
S_2								

and $D_{(a,b)}$ must be the edit distance between $S_1[1..a]$ and $S_2[1..b]$

$$D_{(i,j)} = \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

▶ Calculating $D_{(i,j)}$, we can first calculate $D_{(i-1,j)}$, then delete $S_1(i)$ with result if the last symbol is D
or, calculate $D_{(i,j-1)}$, then insert $S_2(j)$ with result if the last symbol is I
or, calculate $D_{(i-1,j-1)}$, then insert $S_1(i)$ and $S_2(j)$,
edit distance is 0 when the last symbol is M, 1 when the last symbol is R

Dynamic Programming Approach : The Recurrence Relation

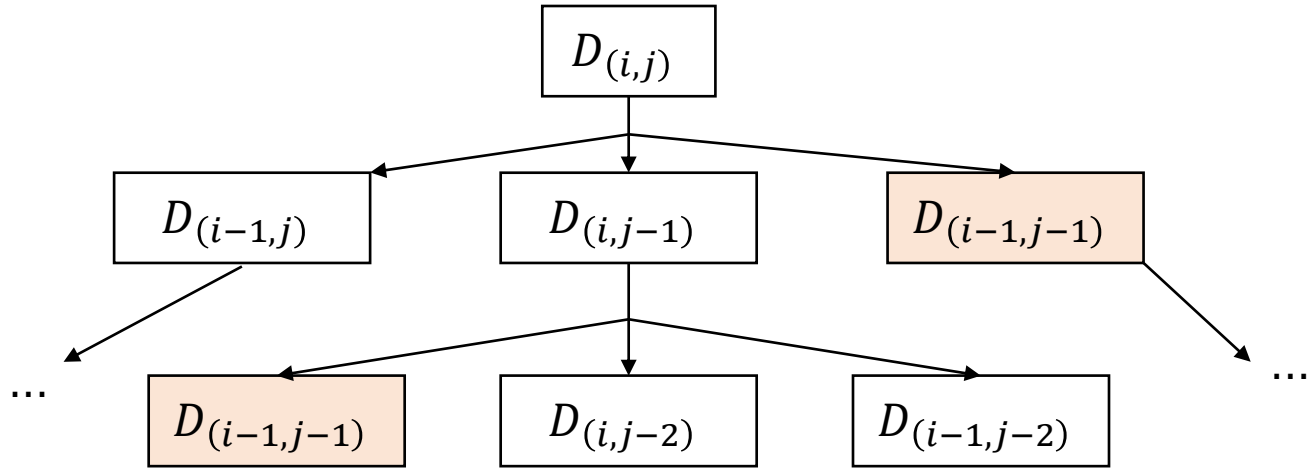
▶ So, we proved this recurrence relation

$$D_{(i,j)} = \min[D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}]$$

※ $t_{(i,j)}$ = 0 if $S_1(i) = S_2(j)$, otherwise 1

Dynamic Programming Approach : Tabular Computation

▶ Recurrence relation we obtained before has exploding time complexity



※ We calculate the same value already called before …

This problem arises since this method is top-down approach
So, let's see bottom-up approach !

Dynamic Programming Approach : Tabular Computation

- Build a table, and fill in the table successively,

Dynamic Programming Approach : Tabular Computation

- Base condition

$$D_{(i,0)} = i$$
$$D_{(0,j)} = \text{j}$$

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1						
r	2	2						
e	3	3						
d	4	4						
i	5	5						
t	6	6						

- Orange are indexes.
- These values (sky) are derived by the base condition.

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$$D_{(i,0)} = i$$
$$D_{(0,j)} = \text{j}$$

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1						
r	2	2						
e	3	3						
d	4	4						
i	5	5						
t	6	6						

▶ g and c is mismatch, $t_{(1,1)}$=1

▶ min value for $D_{(1,1)}$ is one of

$$D_{(0,0)} + 1 = 1$$
$$D_{(0,1)} + 1 = 2$$
$$D_{(1,0)} + 1 = 2$$

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$$D_{(i,0)} = i$$
$$D_{(0,j)} = \text{j}$$

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1					
r	2	2						
e	3	3						
d	4	4						
i	5	5						
t	6	6						

▶ g and c is mismatch, $t_{(1,1)}$=1

▶ min value for $D_{(1,1)}$ is one of

$$D_{(0,0)} + 1 = 1$$
$$D_{(0,1)} + 1 = 2$$
$$D_{(1,0)} + 1 = 2$$

▶ So $D_{(1,1)} = D_{(0,0)} + 1 = 1$

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$D_{(i,0)} = i$
$D_{(0,j)} =$ j

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2				
r	2	2	2	1				
e	3	3	3					
d	4	4	4					
i	5	5	5					
t	6	6	6					

▶ r and r is match, $t_{(2,2)}$= 0

▶ min value for $D_{(2,2)}$ is one of

$$D_{(1,1)} + 0 = 1$$
$$D_{(0,2)} + 1 = 3$$
$$D_{(2,0)} + 1 = 3$$

▶ So $D_{(2,2)} = D_{(1,1)} + 1 = 1$

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$$D_{(i,0)} = i$$
$$D_{(0,j)} = \text{j}$$

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2				
r	2	2	2	1				
e	3	3	3	2				
d	4	4	4					
i	5	5	5					
t	6	6	6					

▶ e and r is match, $t_{(3,2)}$= 1

▶ min value for $D_{(3,2)}$ is one of

$$D_{(2,1)} + 1 = 3$$
$$D_{(2,2)} + 1 = 2$$
$$D_{(3,1)} + 1 = 4$$

▶ So $D_{(3,2)} = D_{(2,2)} + 1 = 2$

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$D_{(i,0)} = i$
$D_{(0,j)} =$ j

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

▶ Table is filled

Dynamic Programming Approach : Tabular Computation

▶ Base condition

$D_{(i,0)} = i$
$D_{(0,j)} = \text{j}$

S_1 : ‘credit’
S_2 : ‘greedy’

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

▶ When we fill a cell (i, j)

▷ check if $S_1(i) = S_2(j)$

▷ compare three cells

constant number of operation!

▶ We had $n * m$ cells to fill,
so the time analysis should be

$O(nm)$

Dynamic Programming Approach : Traceback

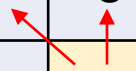
$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

▶ Key idea : Set pointers for every cell,
represents there the value came from.

i.g. which value of
$D_{(i-1,j)} + 1, D_{(i,j-1)} + 1, D_{(i-1,j-1)} + t_{(i,j)}$
is selected

Dynamic Programming Approach : Traceback

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4



▶ Key idea : Set pointers for every cell,
represents there the value came from.

i.g. which value of
$D_{(i-1,j)}+1, D_{(i,j-1)}+1, D_{(i-1,j-1)}+t_{(i,j)}$
is selected

For example, calculating $D_{(6,6)}$,
we compared $D_{(5,6)}+1, D_{(6,5)}+1, D_{(5,5)}+1$
which has value of 4, 5, 4.

$D_{(5,6)}+1$ and $D_{(5,5)}+1$ are chosen,
so we set the pointer to $D_{(5,6)}$ and $D_{(5,5)}$.

Dynamic Programming Approach : Traceback

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

- Likewise, we fill the pointer
- With these pointers, we can find optimal alignments

← : deletion

↑ : insertion

↖ : match or replace

Dynamic Programming Approach : Traceback

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

For example, we will find optimal alignments of $S_1[1..6]$ and $S_2[1..4]$.
Trace back from $(6,4)$, following pointers.

Dynamic Programming Approach : Traceback

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5	6
r	2	2	2	1	2	3	4	5
e	3	3	3	2	1	2	3	4
d	4	4	4	3	2	2	2	3
i	5	5	5	4	3	3	3	3
t	6	6	6	5	4	4	4	4

For example, we will find optimal alignments of $S_1[1..6]$ and $S_2[1..4]$.
Trace back from $(6,4)$, following pointers.

▶ The optimal alignments are

g	r	e	e	_	_
c	r	e	d	i	t

g	r	e	_	e	_
c	r	e	d	i	t

g	r	e	_	_	e
c	r	e	d	i	t

and we can find optimal edit transcripts since :

← : D

↑ : I

↖ : M or R

Dynamic Programming Approach : Traceback

$D_{(i,j)}$		S_2	g	r	e	e	...	y
	Idx	0	1	2	3	4	...	n
S_1	0							
c	1							
r	2							
e	3							
d	4							
...	...							
t	m							

- The worst case in terms of time complexity is

  move along the edges of the table,
  that will have m+n number of tracking required

So, the optimal edit transcript can be found in

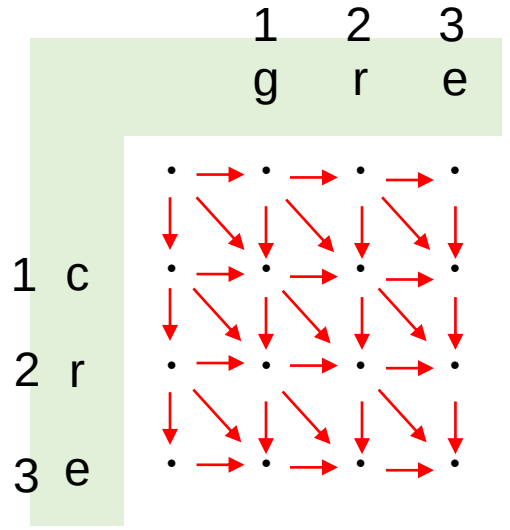
$$O(m+n)$$

Edit Graphs

Given two strings
S_1 : ‘cre’
S_2 : ‘gre’

		g	r	e
	0	1	2	3
c	1	1	2	3
r	2	2	1	2
e	3	3	2	1

Dynamic programming table

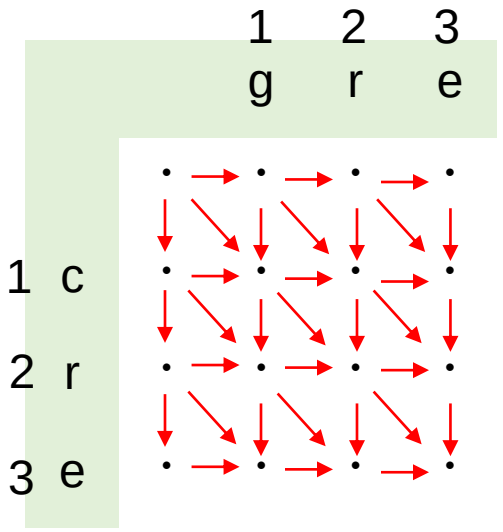


Edit graph

Edit Graphs

Definition

Given two strings S_1 and S_2 of lengths n, m
a weighted graph has $(n+1) * (m+1)$ nodes
each labeled with a distinct pair (i, j)
their edge weights depend on the
specific string problem



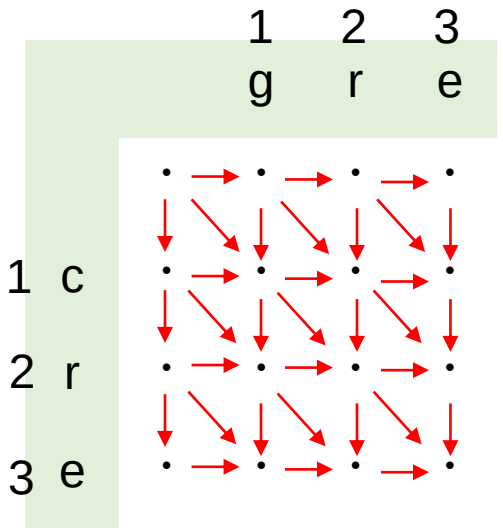
Edit graph

Edit Graphs

- So, we can translate finding optimal transcript to shortest path problem in edit graph.

Theorem 11.4.1

An edit transcript for S_1, S_2 has the minimum number of edit operations if and only if corresponds to a shortest path from (0,0) to (n,m) in the edit graph.



Edit graph

Weighted Edit Distance

- Set weights for each edit operations

S_1 : ‘variable’
S_2 : ‘aristotle’

v	a	r	i	_	a	b	_	l	e
_	a	r	i	s	t	o	t	l	e

Weighted Edit Distance

▶ Set weights for each edit operations

S_1 : ‘variable’
S_2 : ‘aristotle’

v	a	r	i	_	a	b	_	l	e
_	a	r	i	s	t	o	t	l	e

If we set mismatch weights r, match weights e, space weights d,

this alignment will have $2\mathrm{r} + 5e + 3d$ as a weight

Weighted Edit Distance

▶ Base condition

$$D_{(i,0)} = i * d$$
$$D_{(0,j)} = j * d$$

※ r for mismatch, e for match, d for space

▶ General recurrence

$$D_{(i,j)} = \min[D_{(i,j-1)} + d, D_{(i-1,j)} + d, D_{(i,-1\ j-1)} + t_{(i,j)}$$

※ if $S_1(i) = S_2(i)$, $t_{(i,j)} = e$, otherwise r

Weighted Edit Distance

▶ Base condition

$$D_{(i,0)} = i * d$$
$$D_{(0,j)} = j * d$$

If the last characters in an alignment match, $D_{(i,j)} = D_{(i,-1\ j-1)} + e$
if mismatch, $D_{(i,j)} = D_{(i,-1\ j-1)} + r$
$D_{(i,j-1)} + d, D_{(i-1,j)} + d$ corresponds for insertion and deletion, opposite space.

▶ General recurrence

$$D_{(i,j)} = \min[D_{(i,j-1)} + d, D_{(i-1,j)} + d, D_{(i,-1\ j-1)} + t_{(i,j)}$$

※ if $S_1(i) = S_2(i)$, $t_{(i,j)} = e$, otherwise r

Weighted Edit Distance : Alphabet Weight Edit Distance

▶ Alphabet weight edit distance is to put weight on

not the type of the edit operation, but focus on which character is inserted, deleted, or replaced.

String similarity

▶ To measure relevance of two strings

Definition

Let $\sum$ be the alphabet used for strings S_1 and S_2, and let $\sum$' be $\sum$ + '_' (_ denotes space)

Then, for any two characters x, y in $\sum$', $s(x, y)$ denotes the value obtained by aligning character x against character y

S_1	a	x
S_2	b	y

$\sum$ = {a, b, x, y}
$\sum$' = {a, b, x, y, _}

String similarity

Definition

For a given alignment A of S_1 and S_2,
Let S'_1 and S'_2 denote the strings including spaces and let l denote the length of S'_1 and S'_2 in A,
the value of alignment A is defined as

$$\sum_{i=1}^{l} s(S'_1(i), S'_2(i))$$

S_1 : ‘aabbxxyy’
S_2 : ‘abbxyyy’

S'_1 : ‘aabbxxyy’
S'_2 : ‘a_bbxyyy’

String similarity

▶ Given pairwise scoring matrix,

	a	b	x	y	_
a	$w_{(a,a)}$	$w_{(a,b)}$	$w_{(a,x)}$	$w_{(a,y)}$	$w_{(a,\_)}$
b		$w_{(b,b)}$	$w_{(b,x)}$	$w_{(b,y)}$	$w_{(b,\_)}$
x			$w_{(x,x)}$	$w_{(x,y)}$	$w_{(x,\_)}$
y				$w_{(y,y)}$	$w_{(y,\_)}$
_					$w_{(\_,\_)}$

S_1	a	x
S_2	b	y

$\sum$ = {a, b, x, y}
$\sum$' = {a, b, x, y, _}

▶ Then the alignment at the right side of the page will have a value of $w_{(a,b)} + w_{(x,y)}$

String similarity

Definition

For a given alignment A of S_1 and S_2,
Let S'_1 and S'_2 denote the strings including spaces
and let l denote the length of S'_1 and S'_2 in A,
the value of alignment A is defined as

$$\sum_{i=1}^{l} s(S'_1(i), S'_2(i))$$

S_1 : 'aabbxxyy'
S_2 : 'abbxyyy'

S'_1 : 'aabbxxyy'
S'_2 : 'a_bbxyyy'

▶ Then the alignment in this page will have a value of $w_{(a,a)} + w_{(a,\_)} + 2 * (w_{(b,b)} + w_{(y,y)}) + w_{(x,x)} + w_{(x,y)}$

String similarity

Definition

Given a pairwise scoring matrix over ∑',
the similarity of two strings S_1, S_2 is
the value of the alignment A of S_1 and S_2,
that maximizes total alignment value.

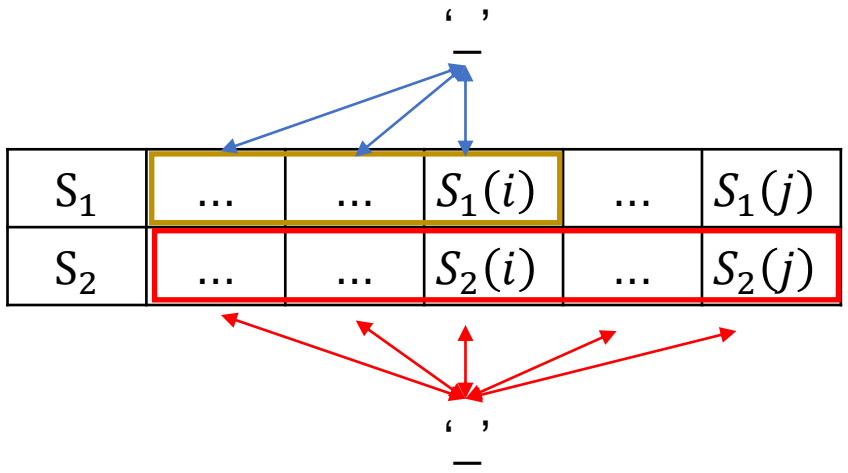
※ It's called optimal alignment value of S_1 and S_2

String similarity

▶ Base condition

$$V_{(0,j)} = \sum_{k=1}^{j} s(\_, S_2(k))$$

$$V_{(i,0)} = \sum_{k=1}^{i} s(S_1(k), \_)$$



$$V_{(0,j)} = \sum_{k=1}^{j} s(\_, S_2(k))$$

$$V_{(i,0)} = \sum_{k=1}^{i} s(S_1(k), \_)$$

- General Recurrence

$$V(i,j) = \max[$$
$$V(i-1,j-1) + s\big(S_1(i), S_2(j)\big),$$
$$V(i-1,j) + s(S_1(i), \_),$$
$$V(i,j-1) + s\big(\_, S_2(j)\big)]$$

- If $S_1(i)$ opposite $S_2(j)$, it means $i-1, j-1$ was in previous step.
  by definition, $V(i-1,j-1)$ is the maximum value by scoring matrix,
  so we can add $s\big(S_1(i), S_2(j)\big)$ to $V(i-1,j-1)$.

$V(i-1,j-1)$

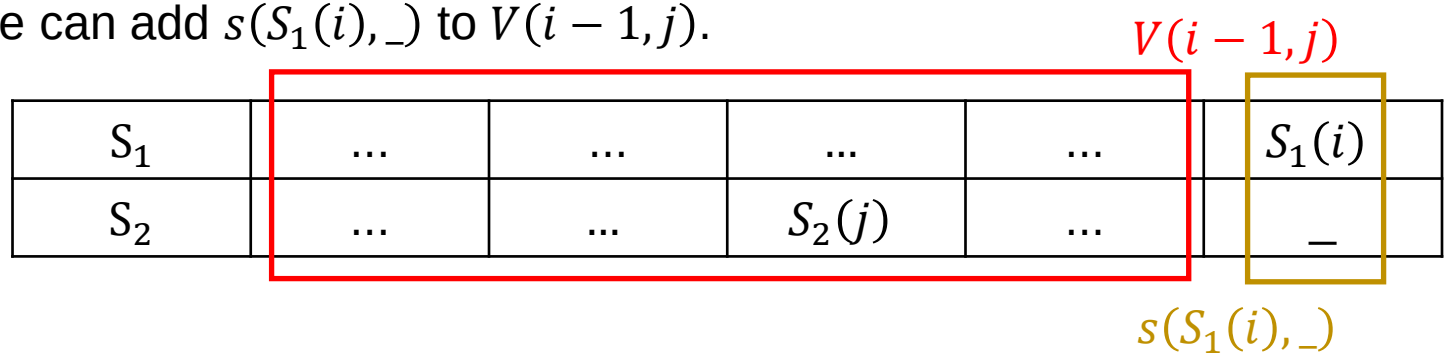
S_1	...	...	$S_1(i-1)$	...	$S_1(i)$
S_2	...	$S_2(j-1)$	...	...	$S_2(j)$

$s\big(S_1(i), S_2(j)\big)$

- General Recurrence

$$V(i,j) = \max[$$
$$V(i-1,j-1) + s\big(S_1(i), S_2(j)\big),$$
$$V(i-1,j) + s(S_1(i), \_),$$
$$V(i,j-1) + s\big(\_, S_2(j)\big)]$$

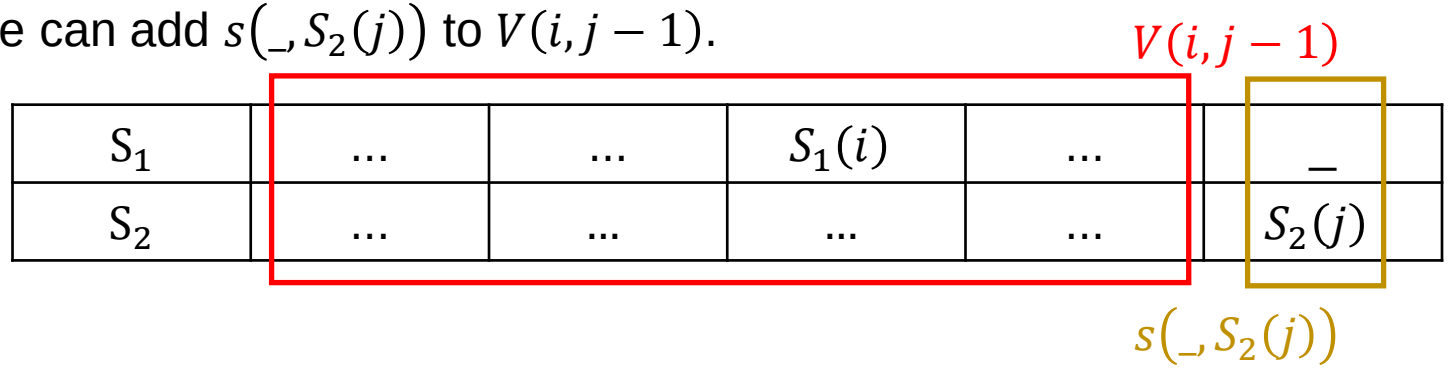
- If $S_1(i)$ opposite space, it means $i-1, j$ was in previous step.
  by definition, $V(i-1,j)$ is the maximum value by scoring matrix,
  so we can add $s(S_1(i), \_)$ to $V(i-1,j)$.



- General Recurrence

$$V(i,j) = \max[$$
$$V(i-1,j-1) + s\big(S_1(i), S_2(j)\big),$$
$$V(i-1,j) + s(S_1(i), \_),$$
$$V(i,j-1) + s\big(\_, S_2(j)\big)]$$

- If $S_2(j)$ opposite space, it means $i, j-1$ was in previous step.
  by definition, $V(i,j-1)$ is the maximum value by scoring matrix,
  so we can add $s\big(\_, S_2(j)\big)$ to $V(i,j-1)$.



String similarity : Longest subsequence problem

Definition

In a string S, a subsequence is defined as
a subset of the characters of S
arranged in their original relative order.

- Given string S for its length n,

  for some $\mathrm{k} \leq n$ and indices $i_1, i_2, i_3 \ \ldots i_k$ $(i_1 < i_2 < i_3 < \cdots < i_k)$
  subsequence specified by this list of indices is

$$S(i_1)S(i_2)S(i_3) \ldots S(i_k)$$

String similarity : Longest subsequence problem

Definition

In a string S, a subsequence is defined as
a subset of the characters of S
arranged in their original relative order.

- Given string S for its length n,

  for some $\mathrm{k} \leq n$ and indices $i_1, i_2, i_3 \ \ldots i_k$ $(i_1 < i_2 < i_3 < \cdots < i_k)$
  subsequence specified by this list of indices is

$$S(i_1)S(i_2)S(i_3) \ldots S(i_k)$$

S : ‘algorithm’

‘agtm’, ‘aorim’, ‘arm’ are subsequences

String similarity : Longest subsequence problem

Definition

Given two strings, a common subsequence is a subsequence that appears in both of them.

The longest common subsequence is to find a longest common subsequence for two strings.

String similarity : Longest subsequence problem

Theorem 11.6.1

With a scoring scheme (corresponds to what we called pairwise scoring matrix before),
match for one score, mismatch or space for zero score
will lead to alignment that has a longest common subsequence

- This kind of scoring will make an alignment that has maximum number of matches, regardless of the number of mismatch or space

  This kind of property directly means longest common subsequence

String similarity : Longest subsequence problem

- Time analysis for longest subsequence problem

In dynamic approach in longest subsequence problem,
We fill in the table,
which means the number of subsequence

$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	0	0	0	0	0	0
c	1	0						
r	2	0						
e	3	0						
d	4	0						
i	5	0						
t	6	0						

String similarity : Longest subsequence problem

- Time analysis for longest subsequence problem

In dynamic approach in longest subsequence problem,
We fill in the table,
which means the number of subsequence

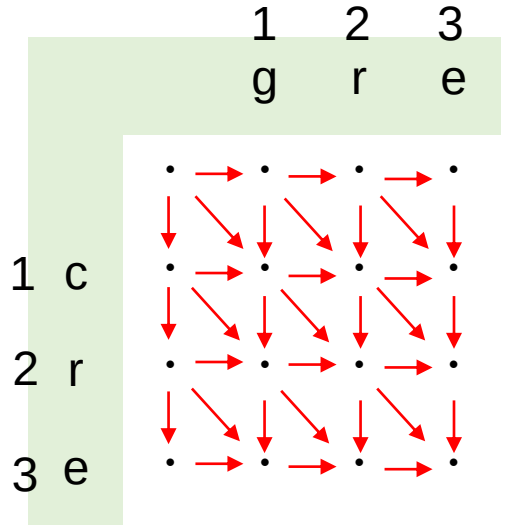
$D_{(i,j)}$		S_2	g	r	e	e	d	y
	Idx	0	1	2	3	4	5	6
S_1	0	0	0	0	0	0	0	0
c	1	0	0	0	0	0	0	0
r	2	0	0	1	1	1	1	1
e	3	0	0	1	2	2	2	2
d	4	0	0	1	2	2	3	3
i	5	0	0	1	2	2	3	3
t	6	0	0	1	2	2	3	3

Filling the table, we have m*n cells
with a constant number of
comparisons and arithmetic operations,

Time complexity obtained is
$O(nm)$

String similarity : Alignment Graphs for Similarity

- Alignment graph is same as what we did in edit graph, where the weights of each nodes are up to the specific pair of characters (or character against space)



Our goal is to find path to (n,m) with maximum score,

there are $3 * (n-1) * (m-1) + (n+m+1)$ nodes

$3nm - 2n - 2m + 4$ nodes, which is $O(nm)$

End-space Free Variant

- To set a weight of every spaces at the end, or the beginning to zero

_	_	c	a	c	_	d	b	d	b
l	t	c	a	b	b	d	b	a	_

weight = 0

End-space Free Variant

- Usually, mismatch or space corresponds to contribution of negative value, but end-space free alignment is free of having space at the edge of a string.

_	_	_	a	c	_	d	b	d	b
l	t	c	a	b	b	d	b	a	a

weight = 0

- As a result, this kind of alignment encourages with one string suffix to other,

End-space Free Variant

- Usually, mismatch or space corresponds to contribution of negative value, but end-space free alignment is free of having space at the edge of a string.

b	a	a	a	c	_	_	_	_	_
l	t	c	a	b	b	d	b	a	a

weight = 0

- As a result, this kind of alignment encourages with one string suffix to other, or prefix to other,

End-space Free Variant

- Usually, mismatch or space corresponds to contribution of negative value,
  but end-space free alignment is free of having space at the edge of a string.

_	_	_	a	c	_	_	_	_	_
l	t	c	a	b	b	d	b	a	a

weight = 0

- As a result, this kind of alignment encourages with one string suffix to other,
  or prefix to other,
  or interior to other.

End-space Free Variant

- Computing similarity for free end-space alignments

- Base condition

$$V_{(0,j)} = 0$$
$$V_{(i,0)} = 0$$

- General Recurrence

$$V(i,j) = \max[$$
$$V(i-1,j-1) + s\big(S_1(i), S_2(j)\big),$$
$$V(i-1,j) + s(S_1(i), \_),$$
$$V(i,j-1) + s\big(\_, S_2(j)\big)]$$

Same method to computing the value of optimal alignment,
differs in base condition to make free of the spaces at the beginning or end.