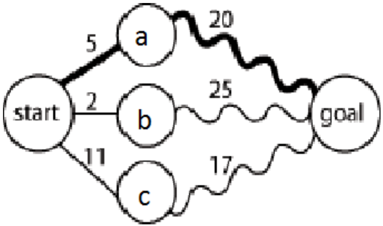

THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Algorithms and Data Structures

Lecture 21 Dynamic Programming

Jiamou Liu
The University of Auckland



Optimisation

Recall

An optimisation problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Examples of Optimisation Problem

- Shortest Path
- Minimum Spanning Tree
- Knapsack Problem
- Sorting

Optimisation

Recall

An optimisation problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Examples of Optimisation Problem

- Shortest Path
- Minimum Spanning Tree
- Knapsack Problem
- Sorting (Reformulated): Arrange a collection of n numbers into a sequence

  $$a_1, a_2, \ldots, a_n$$

  where the length of the longest increasing subsequence is maximized.

Sorting VS Shortest Path

Sorting VS ShortestPath

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]
  - Sorting: In a sorted array, any subarray is also sorted
  - SP: In a shortest path, any segment is also a shortest path

Sorting VS ShortestPath

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]

  - Sorting: In a sorted array, any subarray is also sorted

  - SP: In a shortest path, any segment is also a shortest path

  $\Rightarrow$ both problems can be solved by "division".

Sorting VS ShortestPath

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]

  \- Sorting: In a sorted array, any subarray is also sorted

  \- SP: In a shortest path, any segment is also a shortest path

  ⇒ both problems can be solved by "division".

- Difference: Suppose we divide the problem into subproblems

  \- Sorting: The subproblems are completely independent.

  Hence a top-down algorithm is suitable

  ⇒ Divide and Conquer

  \- SP: The subproblems overlap.

  Hence a bottom-up algorithm is suitable

  ⇒ Dynamic Programming

Dynamic Programming

Dynamic Programming

- Dynamic programming is a method for solving complex optimisation problems by breaking them down into simpler subproblems.
- It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure.
- Dynamic programming solves the subproblems from small to large to avoid duplication

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

3. Write a recurrence for larger subproblems

   $$d_{k+1}(u) = \min\{d_k(u), \min\{d_k(v) + w(v,u) \mid (v,u) \in E\}\}$$

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

3. Write a recurrence for larger subproblems

   $d_{k+1}(u) = \min\{d_k(u), \min\{d_k(v) + w(v,u) \mid (v,u) \in E\}\}$

4. Fill the table of partial solutions in a bottom-up way

   Start from d_0, then compute $d_1, d_2, \ldots, d_{n-1}$

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $f_0(i, j) = w(v_i, v_j)$.

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $f_0(i, j) = w(v_i, v_j)$.

3. Write a recurrence for larger subproblems

   $$f_{k+1}(i, j) = \min\{f_k(i, j), f_k(i, k+1) + f_k(k+1, j)\}$$

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \le k \le n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $$f_0(i, j) = w(v_i, v_j).$$

3. Write a recurrence for larger subproblems

   $$f_{k+1}(i, j) = \min\{f_k(i, j), f_k(i, k + 1) + f_k(k + 1, j)\}$$

4. Fill the table of partial solutions in a bottom-up way

   Start from f_1, then compute $f_2, f_3, \ldots, f_{n-1}$

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A sequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

An increasing subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A longest increasing subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

Another longest increasing subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

Reformulate as a Graph Question

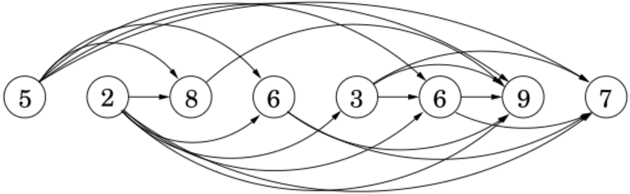
$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

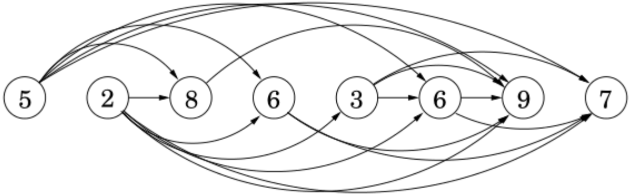
Reformulate as a Graph Question



Create an edge $(a[i], a[j])$ if $i < j$ and $a[i] < a[j]$.
Compute the longest path in this graph.

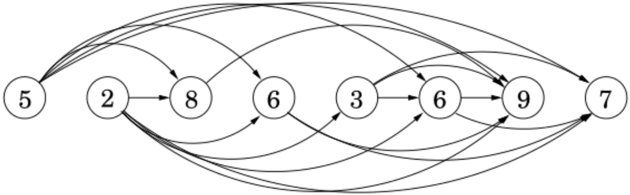
Example: Longest Increasing Subsequence

Divide into Subproblems



Example: Longest Increasing Subsequence

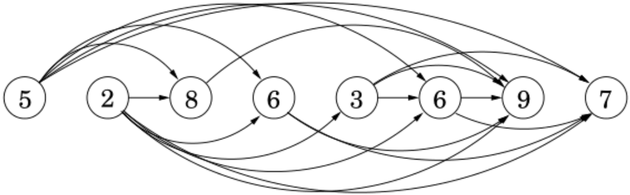
Divide into Subproblems



Let $L(i)$ denote the length of the longest path that ends at $a[i]$.

Example: Longest Increasing Subsequence

Divide into Subproblems

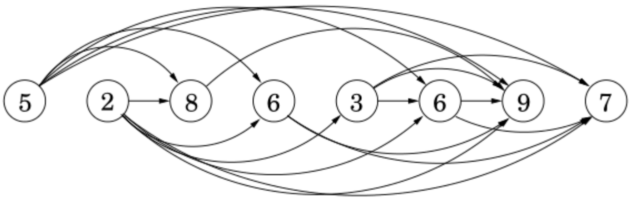


Let $L(i)$ denote the length of the longest path that ends at $a[i]$.

Then the length of the longest increasing subsequence is:

$$\max\{L(i) \mid 1 \leq i \leq n\}$$

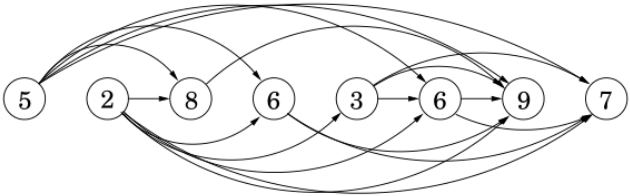
Example: Longest Increasing Subsequence



Base Case: The Smallest Subproblem

Recurrence for Larger Subproblems

Example: Longest Increasing Subsequence

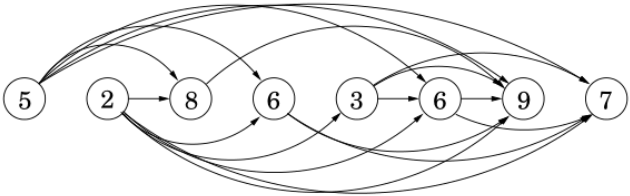


Base Case: The Smallest Subproblem

$L(1) = 1$

Recurrence for Larger Subproblems

Example: Longest Increasing Subsequence



Base Case: The Smallest Subproblem

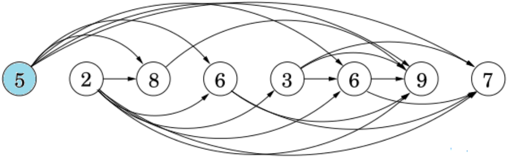
$L(1) = 1$

Recurrence for Larger Subproblems

$L(i+1) = 1$ if indegree$(a[i+1]) = 0$
$L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

Example: Longest Increasing Subsequence

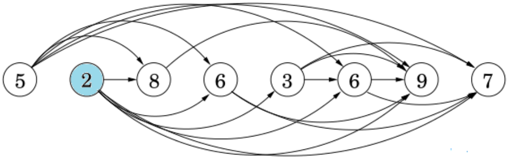
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1							

Example: Longest Increasing Subsequence

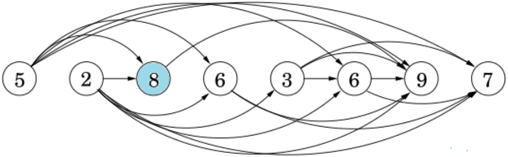
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1						

Example: Longest Increasing Subsequence

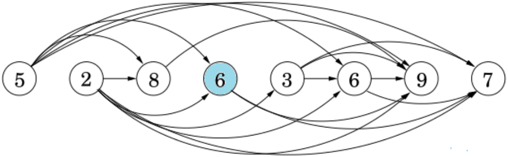
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2					

Example: Longest Increasing Subsequence

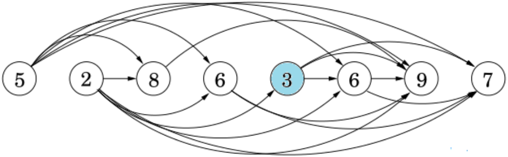
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2				

Example: Longest Increasing Subsequence

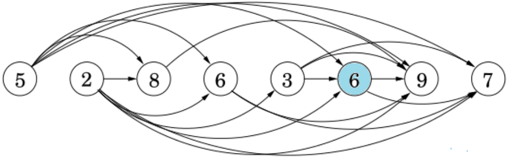
Filling the Table



$L(1)$	$L(2)$	$L(3)$	$L(4)$	$L(5)$	$L(6)$	$L(7)$	$L(8)$
1	1	2	2	2			

Example: Longest Increasing Subsequence

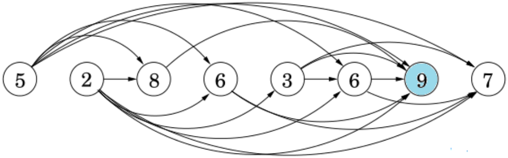
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2	3		

Example: Longest Increasing Subsequence

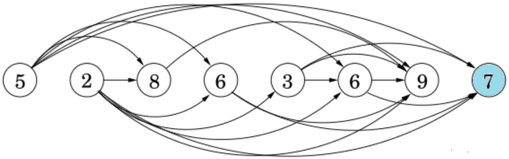
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2	3	4	

Example: Longest Increasing Subsequence

Filling the Table



$L(1)$	$L(2)$	$L(3)$	$L(4)$	$L(5)$	$L(6)$	$L(7)$	$L(8)$
1	1	2	2	2	3	4	4

Example: Longest Increasing Subsequence

LIS($a[1..n]$)

INPUT: An integer array a of length n
OUTPUT: The length of the longest increasing subsequence in a
Create an integer array $L[1..n]$
Set every $L[i]$ to 1
for $i = 2..n$ **do**
    for $j = 1..i-1$ **do**
        if $a[j] < a[i]$ **then**
            $L[i] \leftarrow \max\{L[i], a[j] + 1\}$
Return $\max\{L[i] \mid 1 \leq i \leq n\}$

Note: Can you extend this algorithm so that it also finds the longest increasing subsequence?

Example: Longest Increasing Subsequence

Summary

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

3. Write a recurrence for larger subproblems

   $L(i+1) = 1$ if $\text{indegree}(a[i+1]) = 0$

   $L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

3. Write a recurrence for larger subproblems

   $L(i+1) = 1$ if $\text{indegree}(a[i+1]) = 0$

   $L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

4. Fill the table of partial solutions in a bottom-up way

   Start from $L(1)$, then compute $L(2), \ldots, L(n)$

Edit Distance

Motivation

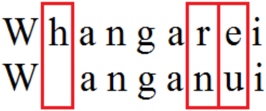
There are words that look similar and words that look different:

Whangarei   Wanganui   Auckland

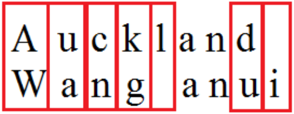
Can we formalize this notion of similarity and disimilarity?
Can we define a distance measure on words?

Edit Distance

The edit distance of two words is the smallest number of edits, that are insertion, deletion, and replacement of letters, needed to transform from one word to another.



edit distance=3



edit distance = 7

Edit Distance

Edit Distance Problem

Given two words $a[1..m]$ and $b[1..n]$, compute the edit distance of them.

Note:

- Different ways of aligning the words result in different number of edits
- The edit distance problem asks for the best way to align the words.

S	—	N	O	W	Y
S	U	N	N	—	Y

edit distance: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

edit distance: 5

Observation

Suppose we would like to find the best alignment for the following:

x = Whangarei

y = Wanganui

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

x = Whangarei

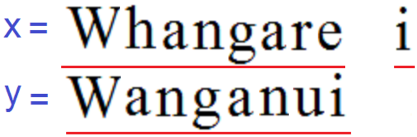
y = Wanganui

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 1: The last letter of x aligns with a blank.



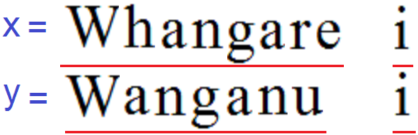
We need to then align Whangare and Wanganui

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 2: The last letter of x aligns with the last letter of y

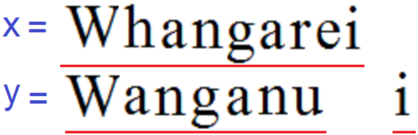


We need to then align Whangare and Wanganu

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 3: The last letter of y aligns with a blank.



We need to then align Whangarei and Wanganu

Edit Distance

Divide into Subproblems

Suppose we want to compute the edit distance of two words

$$x[1..m] \quad \text{and} \quad y[1..n]$$

Let $E(i, j)$ be the edit distance of

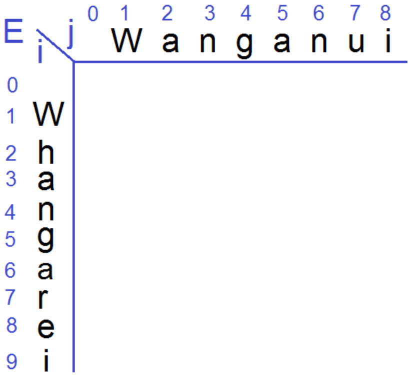
$$x[1..i] \quad \text{and} \quad y[1..j]$$

We would like to find $E(m, n)$

Edit Distance

Base Case: The Smallest Subproblem

$i = 0$ or $j = 0$



Edit Distance

Base Case: The Smallest Subproblem

$i = 0$ or $j = 0$

E	j	0	1	2	3	4	5	6	7	8
	i		W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

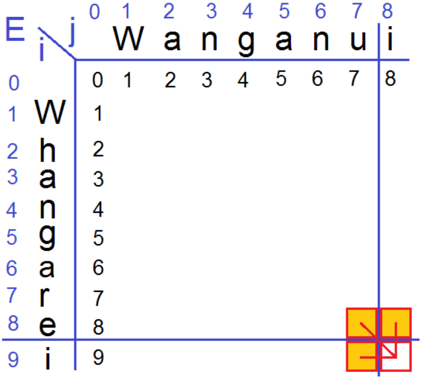
Edit Distance

Recurrence for Larger Subproblem

E (i \ j)		0	1 W	2 a	3 n	4 g	5 a	6 n	7 u	8 i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								$\boxed{}$

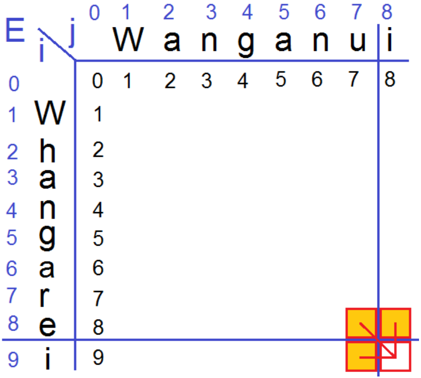
Edit Distance

Recurrence for Larger Subproblem



Edit Distance

Recurrence for Larger Subproblem



$E(i+1, j+1) = \min\{E(i, j+1) + 1, E[i+1, j] + 1, E[i, j] + k\}$
where $k = 1$ if $a[i+1] \neq b[j+1]$, $k = 0$ if $a[i+1] = b[j+1]$.

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0										
1	W									
2	h									
3	a									
4	n									
5	g									
6	a									
7	r									
8	e									
9	i									

Edit Distance

Filling the Table

		0	1	2	3	4	5	6	7	8
E	i \ j		W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1 W	2 a	3 n	4 g	5 a	6 n	7 u	8 i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1 W	2 a	3 n	4 g	5 a	6 n	7 u	8 i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4	3	2	1	2	3	3	4	5
5	g	5	4	3	2	1	2	3	4	5
6	a	6	5	4	3	2	1	2	3	4
7	r	7	6	5	4	3	2	2	3	4
8	e	8	7	6	5	4	3	3	3	4
9	i	9	8	7	6	5	4	4	4	3

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4	3	2	1	2	3	3	4	5
5	g	5	4	3	2	1	2	3	4	5
6	a	6	5	4	3	2	1	2	3	4
7	r	7	6	5	4	3	2	2	3	4
8	e	8	7	6	5	4	3	3	3	4
9	i	9	8	7	6	5	4	4	4	$\boxed{3}$

Edit Distance

EditDistance($a[1..m], b[1..n]$)

INPUT: Two words represented by two char arrays a, b
OUTPUT: The edit distance between a and b
Create an empty 2-dim array $E[1..m][1..n]$
for $i = 0..m$ **do**
    $E[i][0] \leftarrow i$
for $j = 0..n$ **do**
    $E[0][j] \leftarrow j$
for $i = 1..m$ **do**
    for $j = 1..n$ **do**
        $k \leftarrow (a[i] \neq b[j])$
        $E[i][j] \leftarrow \min\{E[i-1][j]+1, E[i][j-1]+1, E[i-1][j-1]+k\}$
return $E[m][n]$

Edit Distance

Summary

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

3. Write a recurrence for larger subproblems

   $$E(i+1, j+1) = \max\{E(i+1, j)+1, E(i, j+1)+1, E(i, j)+k\}$$

   where $k = 0$ if $a[i+1] = b[j+1]$ and $k = 1$ otherwise.

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

3. Write a recurrence for larger subproblems

   $$E(i+1, j+1) = \max\{E(i+1, j)+1, E(i, j+1)+1, E(i, j)+k\}$$

   where $k = 0$ if $a[i+1] = b[j+1]$ and $k = 1$ otherwise.

4. Fill the table of partial solutions in a bottom-up way

   Start from $E(0, j)$, then compute $E(1, j), E(2, j), \ldots, E(m, j)$


THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND