COSC1285/2123: Algorithms & Analysis

Dynamic Programming

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 10

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 8.

Learning outcomes:

- Understand and be able to apply dynamic programming to solving problems.
- Examples:
  - Coin-row problem
  - Computing the edit distance
  - Knapsack
  - Transitive closure – Warshall's algorithm

Outline

Overview

Dynamic Programming

Dynamic Programming is a general algorithm approach for solving problems using the solutions of (overlapping) subproblems.

Dynamic Programming

Dynamic Programming is a general algorithm approach for solving problems using the solutions of (overlapping) subproblems.

Main idea:

1. Setup a recurrence relating a solution of larger instances to the solutions of smaller instances.
2. Solve smaller instances once.
3. Record solutions in a table.
4. Extract solutions to the initial instance from the table, i.e., use solutions of smaller instances to construct solutions of larger initial problem instance.

Dynamic Programming

Sounds similar? Divide-and-conquer?
Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?

Dynamic Programming

Sounds similar? Divide-and-conquer?
Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?
  - Divide-and-conquer algorithms are preferred when the sub-problems/instances are independent, e.g., Mergesort.

Sounds similar? Divide-and-conquer?
Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?
  - Divide-and-conquer algorithms are preferred when the sub-problems/instances are independent, e.g., Mergesort.
  - Dynamic programming approach better when the sub-problems/instances are dependent, i.e., the solution to a sub-problem may be needed multiple times. Hence saving solutions allow them to be reused rather than recomputed. Tradeoff space (more) for time (faster).

Dynamic Programming Approaches

Two basic approaches to dynamic programming.
For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

Dynamic Programming Approaches

Two basic approaches to dynamic programming.
For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

1. Top-Down
   - Start with a divide-and-conquer algorithm, and begin dividing recursively.
   - Only solve/recurse on a subproblem if the solution is not available in the table.
   - Save solutions to subproblems in a table.

Dynamic Programming Approaches

Two basic approaches to dynamic programming.
For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

1. Top-Down
   - Start with a divide-and-conquer algorithm, and begin dividing recursively.
   - Only solve/recurse on a subproblem if the solution is not available in the table.
   - Save solutions to subproblems in a table.
2. Bottom-Up
   - Solve all subproblems, and use solutions to subproblems to construct solutions to larger problems.

Dynamic Programming example: Coin-row Problem

Coin-row Problem

Given a row of n coins with positive integer values $c_1, c_2, \ldots, c_n$ (not necessarily distinct), pick up the maximum amount of money with the constraint that no two adjacent coins can be selected.

EXAMPLE: $5, 1, 2, 10, 6, 2$. What is the best selection?

Dynamic Programming example: Coin-row Problem

EXAMPLE: $5, 1, 2, 10, 6, 2$. What is the best selection?

Dynamic Programming example: Coin-row Problem

EXAMPLE: $5, 1, 2, 10, 6, 2$. What is the best selection?

Rough sketch of dynamic programming approach: Say I am considering whether to select the last coin (value 2) in my selection. I have two choices: select or not select the last coin (to maximise total value of coins selected):

Dynamic Programming example: Coin-row Problem

EXAMPLE: $5, 1, 2, 10, 6, 2$. What is the best selection?

Rough sketch of dynamic programming approach: Say I am considering whether to select the last coin (value 2) in my selection. I have two choices: select or not select the last coin (to maximise total value of coins selected):

- If I don't select the last coin:

Dynamic Programming example: Coin-row Problem

EXAMPLE: $5, 1, 2, 10, 6, 2$. What is the best selection?

Rough sketch of dynamic programming approach: Say I am considering whether to select the last coin (value 2) in my selection. I have two choices: select or not select the last coin (to maximise total value of coins selected):

- If I don't select the last coin:
- If I do select the last coin:

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i.

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i.

Then we can write recurence relationship:

$$F(n) = \max\{\text{total value of solution that selects } n\text{th coin,}$$
$$\text{total value of solution not does not select } n\text{th coin}\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i.

Then we can write recurence relationship:

$$F(n) = \max\{\text{total value of solution that selects } n\text{th coin}, \\ \text{total value of solution not does not select } n\text{th coin}\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Formally:

$$F(n) = \max\{c_n + \text{total value of solution to the } n-2 \text{ coin sub-problem}, \\ \text{total value of solution to the } n-1 \text{ coin sub-problem}\} \\ \text{for } n > 1,$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i.

Then we can write recurence relationship:

$$
\begin{aligned}
F(n) = \max\{c_n + & \text{total value of solution to the } n-2 \text{ coin sub-problem,} \\
& \text{total value of solution to the } n-1 \text{ coin sub-problem}\} \\
& \text{ for } n > 1,
\end{aligned}
$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i.

Then we can write recurence relationship:

$$
\begin{aligned}
F(n) = \max\{c_n + & \text{total value of solution to the } n-2 \text{ coin sub-problem},\\
& \text{total value of solution to the } n-1 \text{ coin sub-problem}\}\\
& \text{ for } n > 1,
\end{aligned}
$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Formally:

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$							

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0						

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5					

$F[0] = 0, F[1] = c_1 = 5$

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5				

$F[2] = \max(1 + 0, 5) = 5$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7			

$F[3] = \max(2 + 5, 5) = 7$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15		

$F[4] = \max(10 + 5, 7) = 15$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	

$F[5] = \max(6 + 7, 15) = 15$

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	**17**

$F[6] = \max(2 + 15, 15) = 17$

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$
$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	**17**

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.

Coin-row Problem (continued)

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.
- This is called a backtrace.

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.
- This is called a backtrace.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.
- This is called a backtrace.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
- Say $F(n-1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.
- This is called a backtrace.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
- Say $F(n-1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.
- Then backtracking from $F(n-1)$, we see how we got to $F(n-1)$ - either $c_{n-1} + F(n-3)$ or $F(n-2)$

- In order to retrieve the coins in the maximum subset, we must backtrack from the last coin.
- This is called a backtrace.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
- Say $F(n-1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.
- Then backtracking from $F(n-1)$, we see how we got to $F(n-1)$ - either $c_{n-1} + F(n-3)$ or $F(n-2)$
- Continue until we reach $F(0)$

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
-

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
-

index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17
- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.
- Finally, we have $F(2) = F(1)$, so best solution for $F(2)$ must be not to select c_2 but select $c_1 = 5$.
- The optimal solution is $\{c_1, c_4, c_6\}$.

- The previous value is the maximum of $c_n + F(n-2)$ or $F(n-1)$.
index (i)	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17
- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.
- Finally, we have $F(2) = F(1)$, so best solution for $F(2)$ must be not to select c_2 but select $c_1 = 5$.
- The optimal solution is $\{c_1, c_4, c_6\}$.

Question

What is the worst case time and space complexity of this solution?

Overview

Edit Distance

Consider the problem of comparing two sequences/strings. You may:

- Want to align them? Biology, speech recognition
- Want to compare them in the presence of noise? E.g., spell checking, plagiarism detection

Edit Distance

Consider the problem of comparing two sequences/strings. You may:

- Want to align them? Biology, speech recognition
- Want to compare them in the presence of noise? E.g., spell checking, plagiarism detection

Solution? Direct position by position comparison will not work.

Edit Distance

Consider the problem of comparing two sequences/strings. You may:

- Want to align them? Biology, speech recognition
- Want to compare them in the presence of noise? E.g., spell checking, plagiarism detection

Solution? Direct position by position comparison will not work.

One possible solution is the edit distance.

Edit Distance

Definition

The Levenshtein distance or edit distance of two strings/sequences S_1 and S_2 is the minimum number of point mutations required to change S_1 into S_2, where a point mutation is one of:

1. change a symbol,
2. insert a symbol, or
3. delete a symbol.

- A classic problem for dynamic programming solutions.

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the i th position of string S_1 with the i th position of string S_2.

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the ith position of string S_1 with the ith position of string S_2. Example:

$$
\begin{array}{ccccccccccc}
S_1 & = & A & T & A & T & A & T & A & \color{red}{T} \\
 & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
S_2 & = & \color{red}{T} & A & T & A & T & A & T & A
\end{array}
$$

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the ith position of string S_1 with the ith position of string S_2. Example:

$$\begin{array}{ccccccccccc} S_1 & = & A & T & A & T & A & T & A & T \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ S_2 & = & T & A & T & A & T & A & T & A \end{array}$$

- The Hamming distance $d(S_1, S_2) = 8$.

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the ith position of string S_1 with the ith position of string S_2. Example:

$$\begin{array}{ccccccccccc} S_1 & = & A & T & A & T & A & T & A & T \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ S_2 & = & T & A & T & A & T & A & T & A \end{array}$$

- The Hamming distance $d(S_1, S_2) = 8$.
- Because of its operations, the Edit Distance can allow shifts in order to maximise the matching. Example:

$$\begin{array}{cccccccccccc} S_1 & = & - & A & T & A & T & A & T & A & T \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ S_2 & = & T & A & T & A & T & A & T & A & - \end{array}$$

- The Edit distance $Ed(S_1, S_2) = 2$.

Edit Distance

Idea: Imagine we have two strings X, of length n, and Y, of length m.

Edit Distance

Idea: Imagine we have two strings X, of length n, and Y, of length m.

Let $M[n, m]$ represent the edit distance between the strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Edit Distance

Idea: Imagine we have two strings X, of length n, and Y, of length m.

Let $M[n, m]$ represent the edit distance between the strings $X[1 \ldots n]$ and $Y[1 \ldots m]$.

We essentially do a character by character comparison, but one that allows for shifts and gaps. Similar to coin-row problem, we want to reuse solutions to the edit distance between substrings of X and Y. But how?

Edit Distance

Idea: Imagine we have two strings X, of length n, and Y, of length m.

Let $M[n, m]$ represent the edit distance between the strings $X[1 \dots n]$ and $Y[1 \dots m]$.

We essentially do a character by character comparison, but one that allows for shifts and gaps. Similar to coin-row problem, we want to reuse solutions to the edit distance between substrings of X and Y. But how?

Lets study how to match $X[1 \dots n]$ and $Y[1 \dots m]$ and develop a recurrence relationship that will tell us how.

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters match, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m]$ = edit distance of matching the substrings $X[1 \dots n-1]$ and $Y[1 \dots m-1]$.

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters match, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m]$ = edit distance of matching the substrings $X[1 \dots n-1]$ and $Y[1 \dots m-1]$.
- Or, $M[n, m] = M[n-1, m-1]$.

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters match, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m]$ = edit distance of matching the substrings $X[1 \ldots n-1]$ and $Y[1 \ldots m-1]$.
- Or, $M[n, m] = M[n-1, m-1]$.
- E.g., matching strings

$$\begin{array}{cccc} X & = & a & b & c \\ Y & = & u & v & c \end{array}$$

Edit Distance

Case 2 ($X[n]! = Y[m]$): This means the last characters mismatch, and we need to apply one of the edit operations (delete, insert, change). Which one to choose, and which calculated edit distance to reuse?

- "Delete" $X[n]$ from X, at the cost of 1 unit to total edit distance. Hence $M[n, m] = 1 + M[n-1, m]$.

$$\begin{array}{lclll} X & = & a & b & d \\ Y & = & a & b & \end{array}$$

Edit Distance

Case 2 $(X[n]! = Y[m])$: This means the last characters mismatch, and we need to apply one of the edit operations (delete, insert, change). Which one to choose, and which calculated edit distance to reuse?

- "Delete" $X[n]$ from X, at the cost of 1 unit to total edit distance. Hence $M[n, m] = 1 + M[n-1, m]$.

$$\begin{array}{ccccc} X & = & a & b & d \\ Y & = & a & b & \end{array}$$

- "Insert" $Y[n]$ into X. Hence $M[n, m] = 1 + M[n, m-1]$.

$$\begin{array}{ccccc} X & = & a & b & \\ Y & = & a & b & d \end{array}$$

Edit Distance

Case 2 ($X[n]! = Y[m]$): This means the last characters mismatch, and we need to apply one of the edit operations (delete, insert, change). Which one to choose, and which calculated edit distance to reuse?

- "Delete" $X[n]$ from X, at the cost of 1 unit to total edit distance. Hence $M[n, m] = 1 + M[n-1, m]$.

$$\begin{array}{lclll} X & = & a & b & d \\ Y & = & a & b & \end{array}$$

- "Insert" $Y[n]$ into X. Hence $M[n, m] = 1 + M[n, m-1]$.

$$\begin{array}{lclll} X & = & a & b & \\ Y & = & a & b & d \end{array}$$

- "Change" either $X[n]$ or $Y[m]$ into the other, so the characters now match. Hence $M[n, m] = 1 + M[n-1, m-1]$.

$$\begin{array}{lclll} X & = & a & b & d \\ Y & = & a & b & e \end{array}$$

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \ldots n]$ and $Y[1 \ldots m]$.

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Computing the Edit Distance

$$M[n, m] = \begin{cases} M[n-1, m-1] & \text{if } X[n] = Y[m] \\ 1 + \min\left(M[n-1, m-1], M[n-1, m], M[n, m-1]\right) & \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Computing the Edit Distance

$$M[n, m] = \begin{cases} M[n-1, m-1] & \text{if } X[n] = Y[m] \\ 1 + \min\left(M[n-1, m-1], M[n-1, m], M[n, m-1]\right) & \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Recursive definition! Similar to row-coin problem, we build a dynamic programming table/matrix for $M[n, m]$.

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \ldots n]$ and $Y[1 \ldots m]$.

Computing the Edit Distance

$$M[n, m] = \begin{cases} M[n-1, m-1] & \text{if } X[n] = Y[m] \\ 1 + \min\left(M[n-1, m-1], M[n-1, m], M[n, m-1]\right) & \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Recursive definition! Similar to row-coin problem, we build a dynamic programming table/matrix for $M[n, m]$.

I.e., To compute $M[n, m]$, a table $M[1 \ldots n, 1 \ldots m]$ is computed and filled.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1									
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0								
n	2									
n	3									
u	4									
a	5									
l	6									

Compute Ed(`annual`, `annealing`).

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1							
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2						
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3					
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4				
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5			
n	2									
n	3									
u	4									
a	5									
l	6									

Compute *Ed*(`annual`, `annealing`).

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6		
n	2									
n	3									
u	4									
a	5									
l	6									

Compute *Ed*(`annual`, `annealing`).

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1								
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0							
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1						
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2					
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2	3				
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Example

And so on ...

Edit Distance – Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2	3	4	5	6	7
n	3	2	1	0	1	2	3	4	5	6
u	4	3	2	1	1	2	3	4	5	6
a	5	4	3	2	2	1	2	3	4	5
l	6	5	4	3	3	2	1	2	3	4

Compute $Ed(\texttt{annual}, \texttt{annealing})$.

Edit Distance – Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use backtrace

Edit Distance – Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use backtrace
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n). Repeat this process until traverse to (0,0) cell.

Edit Distance – Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use backtrace
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n). Repeat this process until traverse to (0,0) cell.
- This produces a path from (0,0) to (m,n) that is non-decreasing in edit distance.

Edit Distance – Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use backtrace
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n). Repeat this process until traverse to (0,0) cell.
- This produces a path from (0,0) to (m,n) that is non-decreasing in edit distance.
- May not be unique (i.e., several alignments/sequence of operations) lead to same edit distance. This means multiple backtraces.

Edit Distance – Backtrace

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2	3	4	5	6	7
n	3	2	1	0	1	2	3	4	5	6
u	4	3	2	1	1	2	3	4	5	6
a	5	4	3	2	2	1	2	3	4	5
l	6	5	4	3	3	2	1	2	3	4

Compute *Ed*(`annual`, `annealing`).

Edit Distance

- The worst-case and average-case time complexity for table construction is is $\Theta(nm)$ where $|S_1| = n$ and $|S_2| = m$. Backtrace complexity is $\Theta(m + n)$. The solution also uses $\Theta(nm)$ space.

Edit Distance

- The worst-case and average-case time complexity for table construction is is $\Theta(nm)$ where $|S_1| = n$ and $|S_2| = m$. Backtrace complexity is $\Theta(m + n)$. The solution also uses $\Theta(nm)$ space.
- DEMO: `http://www.let.rug.nl/kleiweg/lev/`
- The canonical reference on Edit Distance is: D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, New York, USA, 1997.

Overview

Knapsack Problem

Knapsack Problem

Given n items of known weights $w_1, \ldots, w_n$ and the values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

- Recall that the exact solution for all instances of this problem has been proven to be $\Omega(2^n)$.
- We can solve the problem using dynamic programming in "pseudo-polynomial" time.

DP Knapsack Problem - Sketch

- Consider an instance of the knapsack problem defined by the first i items, $1 \leq i \leq n$, with weights $w_1 \ldots w_i$, values $v_1 \ldots v_i$, and capacity j, $1 \leq j \leq W$.

DP Knapsack Problem - Sketch

- Consider an instance of the knapsack problem defined by the first i items, $1 \leq i \leq n$, with weights $w_1 \ldots w_i$, values $v_1 \ldots v_i$, and capacity j, $1 \leq j \leq W$.
- Let $V[i, j]$ be an optimal value to the subproblem instance of having the first i items and a knapsack capacity of j.

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories:

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories:
  - Among the subsets that do not include the i-th item, the value of the optimal subset is, by definition, $V[i-1, j]$

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories:
  - Among the subsets that do not include the i-th item, the value of the optimal subset is, by definition, $V[i-1, j]$
  - Among the subsets that do include the i-th item ($j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories:
  - Among the subsets that do not include the i·th item, the value of the optimal subset is, by definition, $V[i-1, j]$
  - Among the subsets that do include the i·th item ($j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.
- Whether we choose to include i·th item dependent on whether the i·th item can fit into knapsack and if so, which option leads to larger value ($V[i, j]$).

DP Knapsack Problem - Sketch

This leads to the following recursion:

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

Bottom-Up DP algorithm

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

For example the dynamic programming solution to the *coin row problem* and calculating the *edit distance*.

Given the following problem, how do we solve it using a Bottom-Up Dynamic Programming algorithm?

Bottom-Up DP algorithm

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

For example the dynamic programming solution to the *coin row problem* and calculating the *edit distance*.

Given the following problem, how do we solve it using a Bottom-Up Dynamic Programming algorithm?

Knapsack capacity $W = 6$.

i	1	2	3	4	5
weight(w_i)	3	2	1	4	5
value(v_i)	\$25	\$20	\$15	\$40	\$50

Bottom-Up DP algorithm

We record the solutions to each smaller problems in table.

$\downarrow i$ $\quad W \rightarrow$	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							**GOAL**

Bottom-Up DP algorithm

We record the solutions to each smaller problems in table.

$\downarrow i \quad W \rightarrow$	0	1	2	3	4	5	6
0							
1							
2							
3					?		
4							
5							**GOAL**

$V[4, 3] =$? stores the optimal value for a knapsack with capacity 4 of the first 3 items

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0							
$w_1 = 3$ $v_1 = 25$	1							
$w_2 = 2$ $v_2 = 20$	2							
$w_3 = 1$ $v_3 = 15$	3							
$w_4 = 4$ $v_4 = 40$	4							
$w_5 = 5$ $v_5 = 50$	5							

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0						
$w_2 = 2$ $v_2 = 20$	2	0						
$w_3 = 1$ $v_3 = 15$	3	0						
$w_4 = 4$ $v_4 = 40$	4	0						
$w_5 = 5$ $v_5 = 50$	5	0						

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0					
$w_2 = 2$ $v_2 = 20$	2	0						
$w_3 = 1$ $v_3 = 15$	3	0						
$w_4 = 4$ $v_4 = 40$	4	0						
$w_5 = 5$ $v_5 = 50$	5	0						

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0					
$w_2 = 2$ $v_2 = 20$	2	0	0					
$w_3 = 1$ $v_3 = 15$	3	0						
$w_4 = 4$ $v_4 = 40$	4	0						
$w_5 = 5$ $v_5 = 50$	5	0						

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0					
$w_2 = 2$ $v_2 = 20$	2	0	0					
$w_3 = 1$ $v_3 = 15$	3	0	15					
$w_4 = 4$ $v_4 = 40$	4	0						
$w_5 = 5$ $v_5 = 50$	5	0						

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0					
$w_2 = 2$ $v_2 = 20$	2	0	0					
$w_3 = 1$ $v_3 = 15$	3	0	15					
$w_4 = 4$ $v_4 = 40$	4	0	15					
$w_5 = 5$ $v_5 = 50$	5	0	15					

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0				
$w_2 = 2$ $v_2 = 20$	2	0	0					
$w_3 = 1$ $v_3 = 15$	3	0	15					
$w_4 = 4$ $v_4 = 40$	4	0	15					
$w_5 = 5$ $v_5 = 50$	5	0	15					

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0				
$w_2 = 2$ $v_2 = 20$	2	0	0	20				
$w_3 = 1$ $v_3 = 15$	3	0	15					
$w_4 = 4$ $v_4 = 40$	4	0	15					
$w_5 = 5$ $v_5 = 50$	5	0	15					

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0				
$w_2 = 2$ $v_2 = 20$	2	0	0	20				
$w_3 = 1$ $v_3 = 15$	3	0	15	20				
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20				
$w_3 = 1$ $v_3 = 15$	3	0	15	20				
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20				
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20	?			
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20	?			
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20				
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35			
$w_4 = 4$ $v_4 = 40$	4	0	15	20				
$w_5 = 5$ $v_5 = 50$	5	0	15	20				

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25			
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25			
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35			
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35			
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	?		
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	?		
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35			
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40		
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35			
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35			

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40		
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40		
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40		
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	

Bottom-Up DP algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0, \\ V[i-1,j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0.$$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm – Backtrace

How to find the set of items to include? Use backtrace, similar to edit distance and coin-row problem.

Bottom-Up DP algorithm – Backtrace

How to find the set of items to include? Use backtrace, similar to edit distance and coin-row problem.

1. From $V[n, W]$, trace back how we arrived at this table cell – either from $V[n-1, W]$ or $V[n-1, W-w_n]$.
2. Repeat this step until reach $V[0, 0]$.
3. Items that were included in the backtrace form the final solution for knapsack problem.

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Lets do the backtrace!

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $\quad$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm – Backtrace

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$ $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$ $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$ $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$ $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$ $v_5 = 50$	5	0	15	20	35	40	55	65

Question: In general, using the dynamic programming table, how can we tell if there is multiple optimal solutions to a Knapsack problem?

DP Knapsack Problem

- The complexity of constructing the dynamic table is $\Theta(nW)$ in time and space.
- The complexity of performing the backtrack to find the optimal subset is $\Theta(n + W)$.

NOTE : The running time of this algorithm is not a polynomial function of n; rather it is a polynomial function of n and W, the largest integer involved in defining the problem. Such algorithms are known as *pseudo-polynomial*. They are efficient when the values $\{w_i\}$ are small, but less practical as these values grow large.

DP Knapsack Problem – Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.

DP Knapsack Problem – Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.
- Bottom up dynamic programming approach avoids recomputation, but can compute many unnecessary solutions to sub-problems.

DP Knapsack Problem – Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.
- Bottom up dynamic programming approach avoids recomputation, but can compute many unnecessary solutions to sub-problems.
- Combine space saving of divide and conquer and speed up of bottom up approaches?

DP Knapsack Problem – Top-Down

ALGORITHM **MFKnapsack** (i, j)

/* Implement the memory function method (top-down) for the knapsack problem. */

/* INPUT : A non-negative integer i indicating the number of the first items being considered and a non-negative integer j indicating the knapsack capacity. */

/* OUTPUT : The value of an optimal, feasible subset of the first i items. */

/* NOTE: Requires global arrays $w[1 \dots n]$ and $v[1 \dots n]$ of weights and values of n items, and table $F[0 \dots n, 0 \dots W]$ initialized with -1s, except for row 0 and column 0 being all 0s. */

if $F[i, j] < 0$ **then**
    if $j < w[i]$ **then**
        $x =$ **MFKnapsack**$(i-1, j)$
    else
        $x = \max($ **MFKnapsack**$(i-1, j), v[i]+$ **MFKnapsack**$(i-1, j-w[i]))$
    end if
    $F[i, j] = x$
end if
return $F[i, j]$

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	-1	-1	-1	-1	-1	-1
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	-1

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	-1	-1	-1	-1	-1	-1
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	$\boxed{}$

if $F[i,j] < 0$ **then**
  if $j < w[i]$ **then**
    $x =$ **MFKnapsack**$(i-1, j)$
  else
    $x = \max($ **MFKnapsack**$(i-1,j), v[i]+$ **MFKnapsack**$(i-1, j-w[i]))$
  end if
  $F[i,j] = x$
end if

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	$\boxed{}$	-1	-1	-1	-1	$\boxed{}$
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	$\boxed{}$

DP Knapsack Problem – Top-Down

if $F[i, j] < 0$ **then**
  if $j < w[i]$ **then**
    $x =$ **MFKnapsack**$(i - 1, j)$
  else
    $x = \max($ **MFKnapsack**$(i - 1, j), v[i] +$ **MFKnapsack**$(i - 1, j - w[i]))$
  end if
  $F[i, j] = x$
end if

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0		-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0		-1	-1	-1	-1	
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	

DP Knapsack Problem – Top-Down

if $F[i, j] < 0$ **then**
  if $j < w[i]$ **then**
    $x =$ **MFKnapsack**$(i - 1, j)$
  else
    $x = \max($ **MFKnapsack**$(i - 1, j), v[i] +$ **MFKnapsack**$(i - 1, j - w[i]))$
  end if
  $F[i, j] = x$
end if

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0		-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0		-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0		-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0		-1	-1	-1	-1	
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	

DP Knapsack Problem – Top-Down

if $F[i, j] < 0$ **then**
  if $j < w[i]$ **then**
    x = **MFKnapsack**$(i - 1, j)$
  else
    $x = \max($ **MFKnapsack**$(i - 1, j), v[i] +$ **MFKnapsack**$(i - 1, j - w[i]))$
  end if
  $F[i, j] = x$
end if

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	□	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	□	□	-1	-1	-1	□
$w_4 = 4, v_4 = 40$	4	0	□	-1	-1	-1	-1	□
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	□

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$
$w_2 = 2, v_2 = 20$	2	0	$\square$	$\square$	-1	-1	$\square$	$\square$
$w_3 = 1, v_3 = 15$	3	0	$\square$	$\square$	-1	-1	-1	$\square$
$w_4 = 4, v_4 = 40$	4	0	$\square$	-1	-1	-1	-1	$\square$
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	$\square$

DP Knapsack Problem – Top-Down

if $F[i, j] < 0$ **then**
  if $j < w[i]$ **then**
    $x = $ **MFKnapsack**$(i - 1, j)$
  else
    $x = \max($ **MFKnapsack**$(i - 1, j), v[i]+$ **MFKnapsack**$(i - 1, j - w[i]))$
  end if
  $F[i, j] = x$
end if

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	□	□	□	□	□
$w_2 = 2, v_2 = 20$	2	0	□	□	–	–	□	□
$w_3 = 1, v_3 = 15$	3	0	□	□	–	–	–	□
$w_4 = 4, v_4 = 40$	4	0	□	–	–	–	–	□
$w_5 = 5, v_5 = 50$	5	0	–	–	–	–	–	□

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
		0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	–	–	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	–	–	–	60
$w_4 = 4, v_4 = 40$	4	0	15	–	–	–	–	60
$w_5 = 5, v_5 = 50$	5	0	–	–	–	–	–	**65**

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

In general, when to use top-down or bottom-up dynamic programming?

Top-down incurs additional space and time cost of maintaining recursion stack. Hence:

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

Top-down incurs additional space and time cost of maintaining recursion stack. Hence:

- Bottom-up: When the final problem instance requires most or all of the sub-problem instances to be solved, e.g., edit distance.
- Top-down: When the final problem instance only requires a subset of the sub-problem instances to be solved, e.g., possibly knapsack.

Overview

Transitive closure

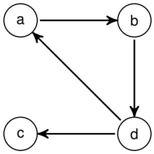
Definition

The transitive closure of a directed graph with n vertices is to determine if there is a path between any pair of vertices in the graph. Yes/True/1 if there is a path, No/False/0 if not. We want to determine for all pairs of vertices.

Transitive closure

Definition

The transitive closure of a directed graph with n vertices is to determine if there is a path between any pair of vertices in the graph. Yes/True/1 if there is a path, No/False/0 if not. We want to determine for all pairs of vertices.



(a) digraph

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) adjacency matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c) transitive closure

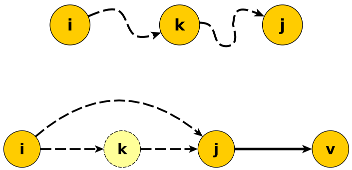
Why compute transitive closure?

- In spreadsheet software, when a cell is changed, what are the other cells whose computation directly or indirectly depend on it (i.e., might need to update also)?
- Critical communication systems: Is it possible for any nodes in this system to communicate with any other node?

Warshall's Algorithm – Sketch

Idea: Progressively use each vertex as an intermediate to join two paths.

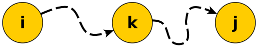
If we know there is a path between vertex *i* to intermediate node *k*, and a path from intermediate node *k* to *j*, then we know there is a path from *i* to *j*.

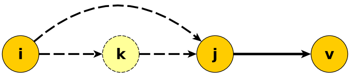


Warshall's Algorithm – Sketch

Idea: Progressively use each vertex as an intermediate to join two paths.

If we know there is a path between vertex *i* to intermediate node *k*, and a path from intermediate node *k* to *j*, then we know there is a path from *i* to *j*.





Warshall's algorithm progressively considers using each vertex as the intermediate, and when all the vertices have been considered as intermediates, we have the transitive closure of the graph.

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $R[i,j]$ in the i·th row and the j·th column is equal to 1 if there is a directed path from the i·th vertex to the j·th vertex.

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $R[i,j]$ in the i·th row and the j·th column is equal to 1 if there is a directed path from the i·th vertex to the j·th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $R[i,j]$ in the i-th row and the j-th column is equal to 1 if there is a directed path from the i-th vertex to the j-th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^{k}:

- If a path exists between *i* and *j* vertices, then it remains there, i.e., if $R[i,j]^{(k-1)} = 1$, it remains 1 ($R[i,j]^{(k)} = 1$).

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $R[i,j]$ in the i·th row and the j·th column is equal to 1 if there is a directed path from the i·th vertex to the j·th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^{k}:

- If a path exists between *i* and *j* vertices, then it remains there, i.e., if $R[i,j]^{(k-1)} = 1$, it remains 1 ($R[i,j]^{(k)} = 1$).
- If we can use vertex *k* as intermediate vertex to traverse from *i* to *j* vertices and we previously didn't know there was such a path, then update our transitivity information, i.e.,

Warshall's Algorithm – Sketch

Idea: Construct the transitive closure of a given digraph with *n* vertices through a series of *n*-by-*n* binary matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $R[i,j]$ in the i·th row and the j·th column is equal to 1 if there is a directed path from the i·th vertex to the j·th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^k:

- If a path exists between *i* and *j* vertices, then it remains there, i.e., if $R[i,j]^{(k-1)} = 1$, it remains 1 ($R[i,j]^{(k)} = 1$).
- If we can use vertex *k* as intermediate vertex to traverse from *i* to *j* vertices and we previously didn't know there was such a path, then update our transitivity information, i.e.,
  - if $R[i,j]^{(k-1)} = 0$, and $R[i,k]^{(k-1)} = 1$ and $R[k,j]^{(k-1)} = 1$, then change $R[i,j]^{(k)}$ to 1.

Warshall's Algorithm – Example

$$R^{(0)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(1)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(1)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & & & & \\ 2 & & & & \\ 3 & & & & \\ 4 & & & & \end{array}$$

$$R^{(1)} = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(1)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

$$R^{(2)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(2)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$
R^{(2)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}
$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$
R^{(2)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}
$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(2)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(3)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

$$R^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$R^{(4)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Example

$$T = R^{(4)} = \begin{array}{c|c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm – Pseudocode

ALGORITHM **Warshall** ($A[1 \ldots n, 1 \ldots n]$)

/* Compute the transitive closure of a graph using Warshall's algorithm. */

/* INPUT : The adjacency matrix A of a digraph with n vertices. */

/* OUTPUT : The transitive closure of the digraph. */

$R^{(0)} = A$
for $k = 1$ **to** n **do**
    for $i = 1$ **to** n **do**
        for $j = 1$ **to** n **do**
            $R^{(k)}[i, j] = R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $(R^{(k-1)}[k, j])$
        end for
    end for
end for
return $R^{(n)}$

Warshall's Algorithm

- The time efficiency of Warshall's algorithm is $\Theta(n^3)$ and uses $\Theta(n^2)$ space.
- For sparse graphs, the transitive closure can be calculated more efficiently using an adjacency list representation with a depth-first or breadth-first traversal.
- Using a DFS or BFS traversal with n vertices and m edges takes $\Theta(n + m)$ time. Doing it n times takes $\Theta(n^2 + nm)$ time.
- If the graph is sparse, i.e. $m \approx n$, the time efficiency is $\Theta(n^2)$.

Overview

Summary

- Described dynamic programming and how it is useful to solve problems that require solving sub-problems multiple times.
- Examples:
  - Coin-row problem
  - Computing the edit distance
  - Knapsack – Bottom up and top down
  - Transitive closure – Warshall's algorithm