Dynamic Programming

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Overview

Space and Time Tradeoff

Dynamic Programming

Memory Functions

Space and Time Tradeoff

In algorithm design, space and time trade-offs are a well-known issue for both theoreticians and practitioners of computing.

- Trading space for time is much more prevalent.

Two principal varieties of trading space for time in algorithm design:

- **input enhancement** - preprocess the problem's input, in whole or in part, and store the additional information obtained in order to accelerate solving the problem afterward
- **prestructuring** - uses extra space to facilitate a faster and/or more flexible access to the data

Overview

Space and Time Tradeoff

Dynamic Programming

Memory Functions

Dynamic Programming

- ”invented” by Richard Bellman in 1950s as a general method for optimizing multistage decision processes (e.g. optimization problems)

  Principle of optimality: An optimal solution to any of its instances must be made up of optimal solutions to its subinstances
- general algorithm design technique for solving problems defined by or formulated as a recurrences with **overlapping subinstances**
- suggests solving each of the smaller subproblems only once and recording the results in a table from w/c a solution to the original problem can be obtained
- allows solving recursive problems with a highly- overlapping subproblem structure *efficiently*

Recall: Computing the n^{th} Fibonacci Number

The Fibonacci numbers $F(n)$ are the elements of the sequence above defined by the recurrence: For $n \geq 2$, $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$

ALGORITHM $F(n)$
//Computes nth Fibonacci number recursively by using its defn
//Input: A nonnegative integer n
//Output: The n^{th} Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

Remark. The algorithm is inefficient. Replicated computation is done.

Dynamic Programming: Array-based Methods

ALGORITHM $Fib(n)$
//Computes the n^{th} Fibonacci number iteratively using defn
//Input: A nonnegative integer n
//Output: The n^{th} Fibonacci number
$F[0] \leftarrow 0$; $F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

What is the time efficiency of the above algorithm? $\Theta(n)$

What is the space efficiency? $\Theta(n)$

Dynamic Programming: Array-based Methods

ALGORITHM $Fib(n)$
//Computes the n^{th} Fibonacci number iteratively using defn
//Input: A nonnegative integer n
//Output: The n^{th} Fibonacci number
$F[0] \leftarrow 0$; $F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

What is the time efficiency of the above algorithm? $\Theta(n)$

What is the space efficiency? $\Theta(n)$ (*can be reduced to* $\Theta(1)$ *by storing the new term in* $F[0]$ *and* $F[1]$ *simultaneously.*)

Dynamic Programming

Main Idea

- identify the subproblems
- set up a recurrence relating a solution to a larger instance to solutions of smaller instances
- determine an ordering for the subproblems
- implement the recurrence by solving the subproblems in order and once; keep results that will be needed at any given point by recording solutions (e.g. in a table)
- extract solution to the initial instance from the table

Computing a Binomial Coefficient

Binomial Coefficient $C(n,k)$ or $\binom{n}{k}$ where $0 \le k \le n$

- number of combinations (subsets) of k elements from an $n-$element set
- coefficients of the binomial formula

Property of Binomial Coefficients:

$$\begin{aligned} C(n,0) &= 1 \\ C(n,n) &= 1 \\ C(n,k) &= C(n-1,k-1) + C(n-1,k) \quad \text{for } n > k > 0 \end{aligned}$$

	0	1	2	$\ldots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
k	1					1
$n-1$	1				$C(n-1,k-1)$	$C(n-1,k)$
n	1					$C(n,k)$

Table. Computing $C(n,k)$ by the dynamic programming algorithm

Computing a Binomial Coefficient

ALGORITHM *Binomial*(n, k)
//Computes $C(n, k)$ by the dynamic programming algorithm
//Input: A pair of nonnegative integers $n \geq k \geq 0$
//Output: The value of $C(n, k)$
for $i \leftarrow 0$ to n **do**
  for $j \leftarrow 0$ to $\min(i, k)$ **do**
    if $j = 0$ or $j = i$ $C[i, j] \leftarrow 1$
    else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
return $C[n, k]$

- What is the time efficiency of the above algorithm? $\Theta(nk)$

Revisiting the Knapsack Problem

Problem: Given n items of known weights

$$w_1, \ w_2, \ w_3, \ \ldots, \ w_n$$

with corresponding values

$$v_1, \ v_2, \ v_3, \ \ldots, \ v_n$$

and a knapsack capacity W, find the most valuable subset of the items that fit into the knapsack.

- **brute force**: generate all subsets to determine optimal solution

Revisiting the Knapsack Problem

Problem: Given n items of known weights

$$w_1, \ w_2, \ w_3, \ \ldots, \ w_n$$

with corresponding values

$$v_1, \ v_2, \ v_3, \ \ldots, \ v_n$$

and a knapsack capacity W, find the most valuable subset of the items that fit into the knapsack.

- **brute force**: generate all subsets to determine optimal solution
- **reduction**: transform to a linear programming problem

The Knapsack Problem

Dynamic Programming Approach

Consider the subproblem defined by the first i items $(1 \leq i \leq n)$ with weights $w_1, w_2, \ldots, w_i$ and values $v_1, v_2, \ldots, v_i$, and a knapsack capacity j $(1 \leq j \leq W)$.

Define $V[i, j]$ as value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j

- What is the value of an optimal subset?

The Knapsack Problem: DP Approach

Divide all the subsets of the first i items that fit into the knapsack of capacity j:

(i) subsets that do not include the i^{th} item
value of an optimal subset is: $V[i-1, j]$

(ii) subsets that do include the i^{th} item
value of an optimal subset is: $v_i + V[i-1, j-w_i]$

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

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

		0	$\dots$	$j - w_i$	$\dots$	j	$\dots$	W
	0	0	$\dots$	0	$\dots$	0	$\dots$	0
w_1, v_1	1	0						
$\vdots$	$\vdots$	$\vdots$						
	$i-1$	0	$\dots$	$V[i-1, j-w_i]$	$\dots$	$V[i-1, j]$		
w_i, v_i	i	0				$V[i, j]$		
$\vdots$	$\vdots$	$\vdots$						
w_n, v_n	n	0						**GOAL**

Table. Solving the knapsack problem by the dynamic programming approach

The Knapsack Problem: DP Approach

ALGORITHM *Knapsack*(n, W)
//Input: A pair of nonnegative integers n and K
//Output: Value of the optimal feasible subset of first n items
for $i \leftarrow 0$ to n **do**
  for $j \leftarrow 0$ to W **do**
    if $j = 0$ or $i = 0$ $V[i, j] \leftarrow 0$
    else if $j - w_i < 0$ $V[i, j] \leftarrow V[i-1, j]$
    else $V[i, j] \leftarrow \max\{V[i-1, j], v_i + V[i-1, j-w_i]\}$
return $V[n, W]$

- What is the time efficiency and space efficiency of the algorithm? $\Theta(nW)$
- What is the time needed to find composition of an optimal solution? $O(n)$.

The Knapsack Problem

Example 1

Consider the instance of the Knapsack problem given by the following data:

item	weight	value	
1	2	\$12	
2	1	\$10	capacity $W = 5$
3	3	\$20	
4	2	\$15	

Use the dynamic programming algorithm to solve the knapsack problem.

Answer: The maximal value is $V[4, 5] = 37$.

Example 1: Solution

	i	capacity j: 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

To find composition of an optimal subset, backtrace the computations:

- $V[4,5] > V[3,5]$ so item 4 has to be included (item left: 3; W: 3u)
- $V[3,3] = V[2,3]$ so item 3 is not included (item left: 2; W: 3u)
- $V[2,3] > V[1,3]$ so item 2 has to be included (item left: 1; W: 2u)
- $V[1,2] > V[0,2]$ so item 1 has to be included (item left: 0; W: 0u)

Overview

Space and Time Tradeoff

Dynamic Programming

Memory Functions

Memory Functions

We proceed with a method that combines the strength of the top-down and bottom-up approaches. Such method exists and is based on using **memory functions**.

Memoization

- general technique that attempts to relieve the potential inefficiency of recursion by using basic idea of dynamic programming
- adds a table indexed by possible inputs to recursive function
- *Idea:* checks whether value of function for requested input is already stored: if it is, value is returned; if not, calls function recursively, then add value to the table for future reference

ALGORITHM $MFKnapsack(i, j)$

//Input: A nonnegative integer i and j

//Output: Value of the optimal feasible subset of first i items

//Note: Uses as global variables input arrays $Weights[1...n]$, $Values[1...n]$ and table

// $V[0...n, 0...W]$ initialized with -1's except for row 0 and column 0 with 0's

if $V[i, j] < 0$

  if $j < Weights[i]$

    $value \leftarrow MFKnapsack(i - 1, j)$

  else

    $value \leftarrow \max\{MFKnapsack(i - 1, j), Values[i] + MFKnapsack(i - 1, j\text{-}Weights[i])\}$

  $V[i, j] \leftarrow value$

return $V[i, j]$

The Knapsack Problem

Example 2

Apply the memory function method to the instance considered in the previous example.

item	weight	value	
1	2	\$12	
2	1	\$10	capacity $W = 5$
3	3	\$20	
4	2	\$15	

Example 2: Solution

- $V[4,5]$ is called first which followed by only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) computations (recursive calls).
- One nontrivial entry, $V[1,2]$, is retrieved rather than recomputed.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Following the same procedure in the previous example, the composition of an optimal subset are items 1, 2, and 4.

End of Lecture