Dynamic Programming

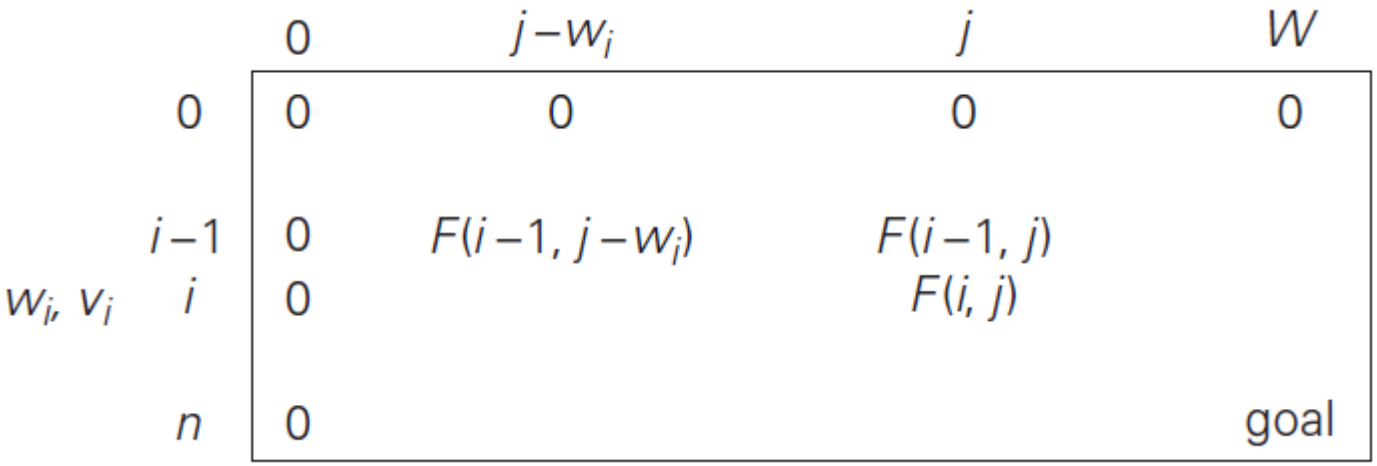
knapsack Problem

- n items of known weight $w_1, w_2, \dots, w_n$ with values $v_1, v_2, \dots, v_n$
- A knapsack of capacity W
- Find most valuable subset of items that fit in the knapsack

- $F(i,j)$: $maximum\ value\ using\ fisrt\ i\ items\ to\ fit\ in\ capacity\ of\ j \leq W$
- $F(i-1,j)$: $does\ not\ include\ ith\ item$
- $F(i-1,j-w_i)+v_i$: $includes\ the\ ith\ item$

- $F(0,j)=0$
- $F(i,0)=0$
- $F(i,j)=\begin{cases} F(i-1,j) & \rightarrow j < w_i \\ \max\{f(i-1,j), F(i-1,j-w_i)+v_i) \end{cases}$

knapsack Problem



Time efficiency: $\theta(nW)$
Space efficiency: $\theta(nW)$
Optimal path: $O(n+W)$

knapsack Problem

- $F(0, j) = 0$
- $F(i, 0) = 0$
- $F(i, j) = \begin{cases} F(i-1, j) & \rightarrow j < w_i \\ \max\{f(i-1, j), F(i-1, j-w_i) + v_i) \end{cases}$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

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

knapsack Problem

- $F(0,j) = 0$
- $F(i,0) = 0$
- $F(i,j) = \begin{cases} F(i-1,j) & \rightarrow j < w_i \\ \max\{f(i-1,j), F(i-1,j-w_i) + v_i) \end{cases}$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

		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	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**

Memory Functions

- Problems with overlapping subproblems:
  - Direct top-down approach
    Solving subproblems more than once
  - Classic dynamic programming
    Bottom-up, solve all smaller subproblems
  - Memory functions
    Top-down, solve only the subproblems that are necessary
    Keep the results in memory
    Time efficiency does not change
    Run time could be better

knapsack Problem: Memory Function

- n items of known weight $w_1, w_2, \dots, w_n$ with values $v_1, v_2, \dots, v_n$
- A knapsack of capacity W
- Find most valuable subset of items that fit in the knapsack

- $F(i,j)$: $maximum\ value\ using\ fisrt\ i\ items\ to\ fit\ in\ capacity\ of\ j \leq W$
- $F(i-1,j)$: $does\ not\ include\ ith\ item$
- $F(i-1,j-w_i)+v_i$: $includes\ the\ ith\ item$

- $F(0,j)=0$
- $F(i,0)=0$
- $F(i,j)=\begin{cases} F(i-1,j) & \rightarrow j < w_i \\ \max\{f(i-1,j), F(i-1,j-w_i)+v_i) \end{cases}$

knapsack Problem: Memory Function

- $F(0,j) = 0$
- $F(i,0) = 0$
- $F(i,j) = \begin{cases} F(i-1,j) & \rightarrow j < w_i \\ \max\{f(i-1,j), F(i-1,j-w_i) + v_i) \end{cases}$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

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

knapsack Problem: Memory Function

- $F(0,j) = 0$
- $F(i,0) = 0$
- $F(i,j) = \begin{cases} F(i-1,j) & \rightarrow j < w_i \\ \max\{f(i-1,j), F(i-1,j-w_i) + v_i) \end{cases}$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

		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**

Discussion

1. a. What does dynamic programming have in common with divide-and-conquer?

   b. What is a principal difference between the two techniques?

Discussion

1. a. What does dynamic programming have in common with divide-and-conquer?

   b. What is a principal difference between the two techniques?

1. a. Both techniques are based on dividing a problem's instance into smaller instances of the same problem.

   b. Typically, divide-and-conquer divides an instance into smaller instances with no intersection whereas dynamic programming deals with problems in which smaller instances overlap. Consequently, divide-and-conquer algorithms do not explicitly store solutions to smaller instances and dynamic programming algorithms do.

Discussion

9. *Shortest path counting* A chess rook can move horizontally or vertically to any square in the same row or in the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner [Gar78], p.10

   (a) by a dynamic programming algorithm.

   (b) by using elementary combinatorics.

Discussion

9. a. With no loss of generality, we can assume that the rook is initially located in the lower left corner of a chessboard, whose rows and columns are numbered from 1 to 8. Let $P(i,j)$ be the number of the rook's shortest paths from square (1,1) to square (i,j) in the ith row and the jth column, where $1 \le i,j \le 8$. Any such path will be composed of vertical and horizontal moves directed toward the goal. Obviously, $P(i,1) = P(1,j) = 1$ for any $1 \le i,j \le 8$. In general, any shortest path to square (i,j) will be reached either from square $(i,j-1)$ or from square $(i-1,j)$ square. Hence, we have the following recurrence

$$\begin{aligned} P(i,j) &= P(i,j-1) + P(i-1,j) \quad \text{for } 1 < i,j \le 8, \\ P(i,1) &= P(1,j) = 1 \quad \text{for } 1 \le i,j \le 8. \end{aligned}$$

Using this recurrence, we can compute the values of $P(i,j)$ for each square $(i.j)$ of the board. This can be done either row by row, or column by column, or diagonal by diagonal. (One can also take advantage of the board's symmetry to make the computations only for the squares either on and above or on and below the board's main diagonal.) The results are given in the diagram below:

Discussion

1	8	36	120	330	792	1716	3432
1	7	28	84	210	462	924	1716
1	6	21	56	126	252	462	792
1	5	15	35	70	126	210	330
1	4	10	20	35	56	84	120
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

Discussion

b. Any shortest path from square (1,1) to square (8,8) can be thought of as 14 consecutive moves to adjacent squares, seven of which being up while the other seven being to the right. For example, the shortest path composed of the vertical move from (1,1) to (8,1) followed by the horizontal move from (8,1) to (8,8) corresponds to the following sequence of 14 one-square moves:

$$(u, u, u, u, u, u, u, r, r, r, r, r, r, r),$$

where u and r stand for a move up and to the right, respectively. Hence, the total number of distinct shortest paths is equal to the number of different ways to choose seven u-positions among the total of 14 possible positions, which is equal to $C(14, 7)$.

Discussion

1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

b. How many different optimal subsets does the instance of part (a) have?

c. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

Discussion

1. a.

		capacity j						
	i	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

The maximal value of a feasible subset is $V[5, 6] = 65$. The optimal subset is {item 3, item 5}.

b.-c. The instance has a unique optimal subset in view of the following general property: An instance of the knapsack problem has a unique optimal solution if and only if the algorithm for obtaining an optimal subset, which retraces backward the computation of $V[n, W]$, encounters no equality between $V[i-1, j]$ and $v_i + V[i-1, j-w_i]$ during its operation.

Discussion

5. Apply the memory function method to the instance of the knapsack problem given in Problem 1. Indicate the entries of the dynamic programming table that are: (i) never computed by the memory function method on this instance; (ii) retrieved without a recomputation.

1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

Discussion

5. Apply the memory function method to the instance of the knapsack problem given in Problem 1. Indicate the entries of the dynamic programming table that are: (i) never computed by the memory function method on this instance; (ii) retrieved without a recomputation.

In the table below, the cells marked by a minus indicate the ones for which no entry is computed for the instance in question; the only nontrivial entry that is retrieved without recomputation is $(2, 1)$.

		capacity j						
	i	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	-	-	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

Discussion

4. a. True or false: A sequence of values in a row of the dynamic programming table for an instance of the knapsack problem is always nondecreasing.

   b. True or false: A sequence of values in a column of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?

Discussion

4. a. True or false: A sequence of values in a row of the dynamic programming table for an instance of the knapsack problem is always nondecreasing.

   b. True or false: A sequence of values in a column of the dynamic programming table for an instance of the knapsack problem is always nondecreasing?

a. $V[i, j-1] \leq V[i, j]$ for $1 \leq j \leq W$ is true because it simply means that the maximal value of a subset that fits into a knapsack of capacity $j-1$ cannot exceed the maximal value of a subset that fits into a knapsack of capacity j.

b. $V[i-1, j] \leq V[i, j]$ for $1 \leq i \leq n$ is true because it simply means that the maximal value of a subset of the first $i-1$ items that fits into a knapsack of capacity j cannot exceed the maximal value of a subset of the first i items that fits into a knapsack of the same capacity j.