Dynamic Programming (3)

By: Aminul Islam

Based on Chapter 3 of Foundations of Algorithms

Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem

Example of 0-1 Knapsack problem (Brute force sol.)

item (i)	1	2	3	4
value (v)	100	20	60	40
weight (w)	3	2	4	1

- Assume Knapsack's max weight capacity is $W = 5$
- How to fill the knapsack with items such that the value is maximum?

items picked= item 1 and item 4

value= 140

weight= 4

Order of complexity= $\Theta(2^n)$

1	2	3	4	W	V
0	0	0	0	0	0
0	0	0	1	1	40
0	0	1	0	4	60
0	0	1	1	5	100
0	1	0	0	2	20
0	1	0	1	3	60
0	1	1	0	6	80
0	1	1	1	7	120
1	0	0	0	3	100
1	0	0	1	4	140
1	0	1	0	7	160
1	0	1	1	8	200
1	1	0	0	5	120
1	1	0	1	6	160
1	1	1	0	9	180
1	1	1	1	10	220

* 1 means item is picked and 0 means item is not picked

Here, we went through all possible solutions/cases and picked the one that maximizes the value. This is known as Brute force algorithm.

Optimization Problems

Optimization problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

Principle of Optimality in DP

- Many optimization problems can be solved using DP
  - For example, 0-1 Knapsack Problem, Shortest path problem in a digraph
- However, not all optimization problems can be solved using DP
- The "principle of optimality" must apply in the problem!

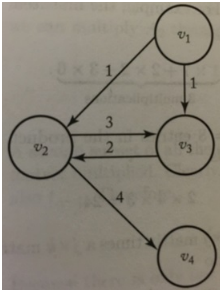
“Principle of Optimality”

- The principle of optimality is applied in a problem if an optimal solution for a problem instance includes optimal solutions for all sub-problems
- If the principle holds, we can provide a recursive solution and obtain optimal solutions from smaller ones

Example: when the principle holds

Shortest path from a node to another node in a graph

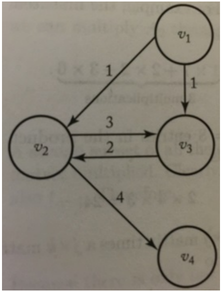
- Suppose V_k is a vertex in an optimal path from V_i to V_j
- Then the sub-path from V_i to V_k and from V_k to V_j are also optimal
- Therefore, the principle of optimality holds!



Example: when the principle of optimality does not hold

Longest simple path (no repeated vertex) from a vertex to another vertex in a graph

- Consider longest path from V_1 to V_4
- The answer is: V_1, V_3, V_2, V_4
- However, sub-path $V_1 V_3$ is not the longest possible!
- In fact, $V_1 V_2 V_3$ is the longest simple path from V_1 to V_3
- Principle of optimality does not hold in this case!



Steps in Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed values.

Example of 0-1 Knapsack Problem (DP solution)

item (`i`)	1	2	3	4
value (`val[]`)	100	20	60	40
weight (`wt[]`)	3	2	4	1

- Assume Knapsack's max weight capacity is $W = 5$
- How to fill the knapsack with items such that the value is maximum?

Algorithm: DP solution to 0-1 Knapsack Problem

```
KnapSack (int W, int n, int val[], int wt[] )
{
    int i, w;
    int V[n+1,W+1];
    for (w=0; w<=W; w++){
     V[0,w]=0; }
    for (i=0; i<=n; i++){
     V[i,0]=0; }
    for (i=1; i<=n; i++){
     for (w=1; w<=W; w++){
      if (wt[i]<=w){
       V[i,w] = max(V[i-1,w], val[i]+V[i-1, w-wt[i]]) }
      else {
       V[i,w] = V[i-1,w] }
     }
    }
    return V[n,W];
    // Add the algorithm on the next slide here for optimal sol.
}
```

Constructing the Optimal Solution

```
w=W;
for (i = n downto 1){
   if (V[i,w] != V[i-1,w]){
    output i ;
    w = w - wt[i] ;
   }
}
```

Example of 0-1 Knapsack problem (DP sol.)

item (i)	1	2	3	4
value (val[])	100	20	60	40
weight (wt[])	3	2	4	1

```
if (wt[i] <= w)
  V[i,w] = max(V[i-1,w], val[i] + V[i-1, w-wt[i]])
else
  V[i,w] = V[i-1,w]
```

$\underline{\texttt{V[i-1,w]}}$, $\underline{\texttt{val[i] + V[i-1, w-wt[i]]}}$

V[i,w]	w=0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

W = 5

n = 4

wt[i] <= w

FalseTrue

Time complexity is: