

Lecture 9: October 3

Lecturer: Micah Adler

Scribes: Yu Gu

9.1 Dynamic Programming

9.1.1 Knapsack Problem

Story:

A thief with a knapsack breaks into a store and finds n items, each of weight w_i and value v_i . His goal is to take items worth as much as possible, but the knapsack can only hold a total weight of W . The problem now is to find those items that maximize the value in the knapsack, while keeping the total weight $\leq W$.

Formal Problem Description:

Input: Set of n items, each item i has value v_i and weight w_i ($i \in \{1, 2, \dots, n\}$) and one knapsack of capacity W (weights, values and the capacity are integers).

Output: Subset S of the items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.

In the following we will only consider a simplified version of the knapsack problem by defining: $\forall i : v_i = w_i$

Example 1:

Input: Ordered set $\{6, 5, 5\}$, knapsack capacity $W = 10$

Output: Optimal: $\{5, 5\}$

To start with, we will just try to find the optimal value of the knapsack (not the subset of items that yields this value)

9.1.1.1 Greedy Algorithm

Example 1 (continued):

Input: Ordered set $\{6, 5, 5\}$, knapsack capacity $W = 10$

Output: $\{6\}$

Clearly the greedy approach is not optimal.

9.1.1.2 Divide and Conquer Algorithm

One way to solve this problem is to use a divide and conquer scheme.

Definition 9.1 $Knap(i, j)$: Optimal solution (value) using only items 1 to i (of the n available items) and knapsack capacity j

Example 1 (continued):

Input: Ordered set $\{6, 5, 5\}$, knapsack capacity $W = 10$

$Knap(2, 5) = 5$, $Knap(2, 8) = 6$, $Knap(1, 5) = 0$

Thus we divide the problem into:

$$\begin{aligned}
 i \geq 1 & : \text{Knap}(i, j) = \begin{cases} \max(\text{Knap}(i-1, j), \text{Knap}(i-1, j-w_i) + w_i) & : w_i \leq j \\ \text{Knap}(i-1, j) & : w_i > j \end{cases} \\
 i = 0 & : \text{Knap}(i, j) = 0
 \end{aligned}$$

When this algorithm is run for $\text{Knap}(n, W)$, it spans a tree:

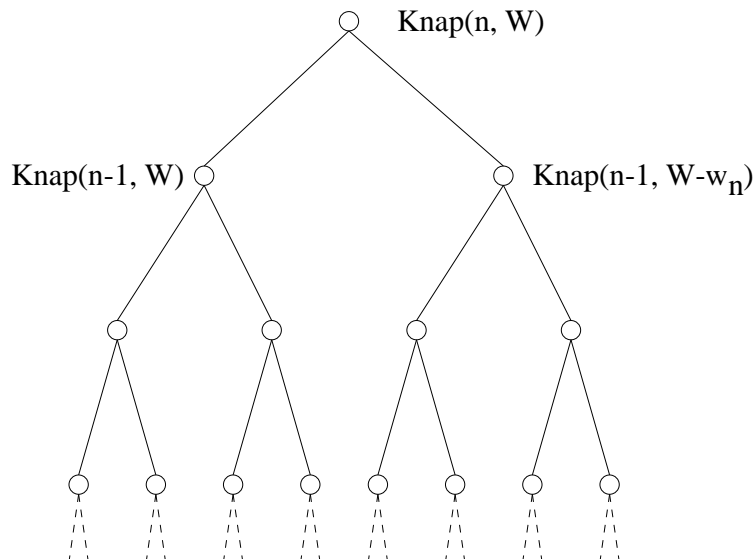


Figure 9.1: Tree of height $n + 1$ spanned by divide and conquer algorithm

The number of nodes in this tree is $O(2^n)$. This means that the algorithm will have an exponential running time. Since the height of the tree is n , we have 2^n leaf nodes. But there can be at most $W + 1$ unique subproblems at each level in the tree (j in $\text{Knap}(i, j)$ can only range from 0 to W and j is an integer). It turns out that many of the nodes overlap.

And there is a level in the tree, at which the number of generated subproblems (at that level) exceeds the possible number of knapsack problems ($W + 1$):

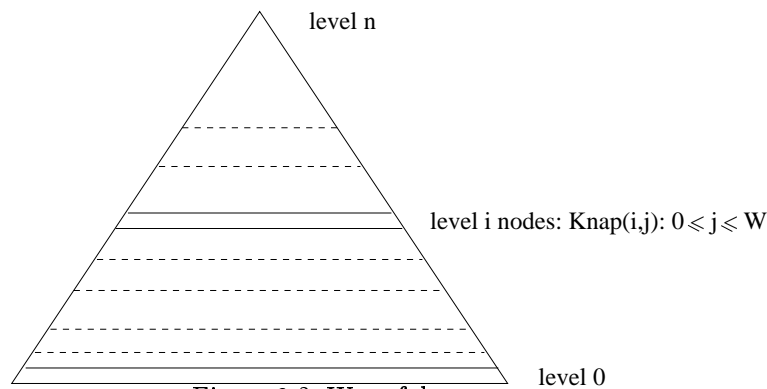


Figure 9.2: Wasteful tree structure

9.1.1.3 A Better Approach (Dynamic Programming)

We can improve our wasteful strategy by using a table representation which insures that we only consider each subproblem exactly once:

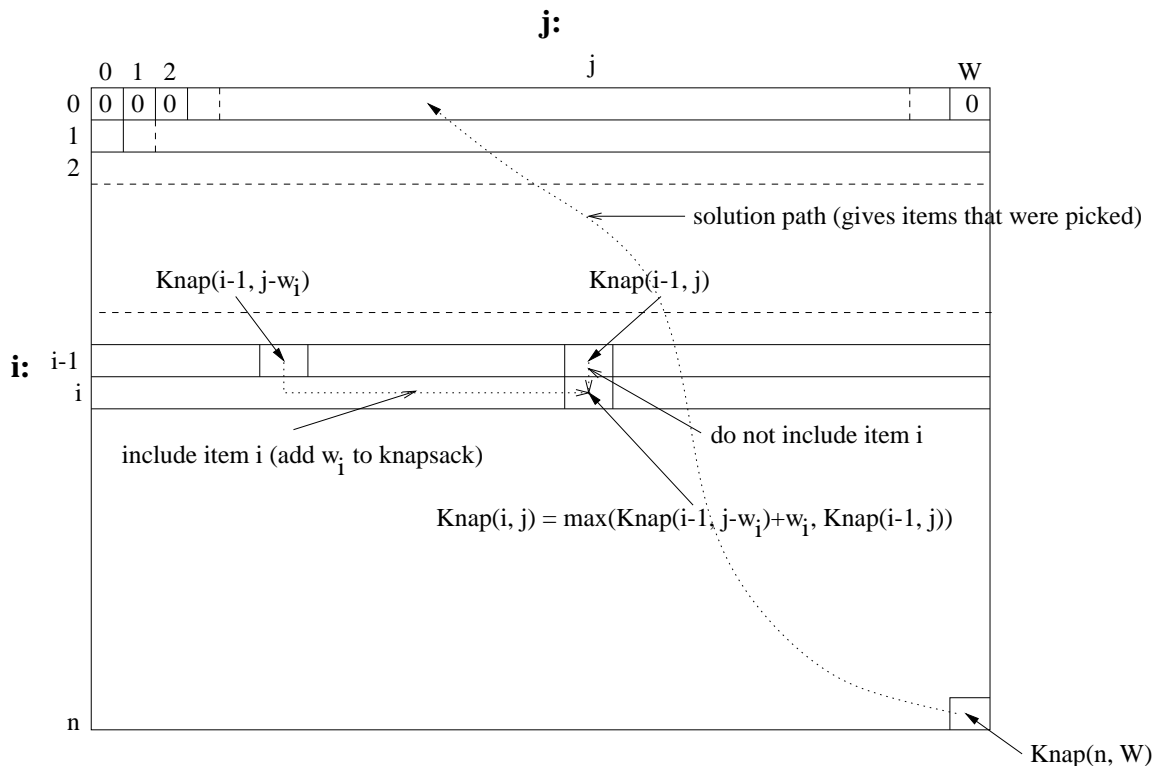


Figure 9.3: Table representation of the knapsack problem

The table is generated row by row from the top to the bottom. For each entry $Knap(i, j)$, the maximum of $Knap(i-1, j)$ and $Knap(i-1, j-w_i) + w_i$ is computed (see divide and conquer scheme). We also store a pointer, which of the entries in the table generated the current entry (current entry results from adding an item or from not adding an item).

The running time of this procedure can be easily calculated:

- each entry: $\Theta(1)$
- total time: $\Theta(n \cdot W)$ (table has $(n+1)(W+1)$ entries)

It seems as if the running time of the dynamic programming algorithm is polynomial, but the knapsack problem is known to be NP-complete. How can this be? NP-completeness is defined in terms of *bit*-complexity and therefore, our input length is not W , but rather $k \approx \log W$ (length of bit representation of W). Hence, the bit complexity of our algorithm is $O(n \cdot 2^k)$ (which is exponential). Therefore the algorithm is called a *pseudo-polynomial* algorithm.

The space complexity of the algorithm depends on whether we want to obtain the subset of items that yields an optimal solution or whether we are only interested in the optimal value (sum of the weights/values of the items in the knapsack):

- compute only optimal knapsack value: $\Theta(W)$
- also compute subset of items that yields optimal knapsack value: $\Theta(n \cdot W)$

This results from the fact that if we are only looking for the optimal value, we only have to keep 2 rows of the table in memory at each time. If we want the subset of items that produces the optimal value, we have to keep the whole table in memory. When the table is built, we start with the solution on the bottom right side and go backwards along the pointers that we stored for each entry to obtain the items that we have added.

9.2 When To Use Dynamic Programming?

Properties of a problem that indicate that a dynamic programming algorithm may be the right choice are:

- optimal substructure (the solution for a specific problem size is composed of optimal solutions of smaller size)
- overlap of subproblems (e.g. redundancy in above tree structure)

If Divide and Conquer is efficient, then we are done. Otherwise, we should find redundant subproblems. If there are some, we can try the Dynamic Programming method. Typically we use the table representation shown above.

9.3 Shortest Path Algorithms

There is a sort of problem domain called the Shortest Path. We use Shortest Path Algorithms as one example of Dynamic Programming Methods.

9.3.1 Catalog of the Problem Domain

Base case:

Input: Directed graph $G = (E, V)$, weight function $w : E \rightarrow \mathfrak{R}$.

Output: $\delta(u, v)$ = minimum over path weights of all paths from vertex u to vertex v ($u, v \in V$)
(a path weight is the sum of all weights of edges that lie on the path)

Single Source Shortest Paths:

Input: Source vertex $s \in V$

Output: $\forall v \in V$, compute $\delta(s, v)$

Single Pair Shortest Path:

Input: Source vertex $u \in V$, destination vertex $v \in V$

Output: $\delta(u, v)$

Algorithm: Use single source algorithm with $s = u$ and ignore everything but $\delta(u, v)$

Single Destination Shortest Paths:

Input: Destination vertex $d \in V$

Output: $\forall u \in V$, compute $\delta(u, d)$

Algorithm: Use single source algorithm with $s = d$ and reverse all edges in the graph

All-pairs Shortest-paths:

Input: Just base case input

Output: $\forall u, v \in V$, compute $\delta(u, v)$

Algorithm: Use single source S.P. for every vertex $u \in V$

A better algorithm uses dynamic programming, The Floyd-Warshall algorithm, which we will show later.

9.3.2 Single Source Shortest Path (Special Case)

Special case: All edges have weight = 1.

Using Breadth First Search from source S:

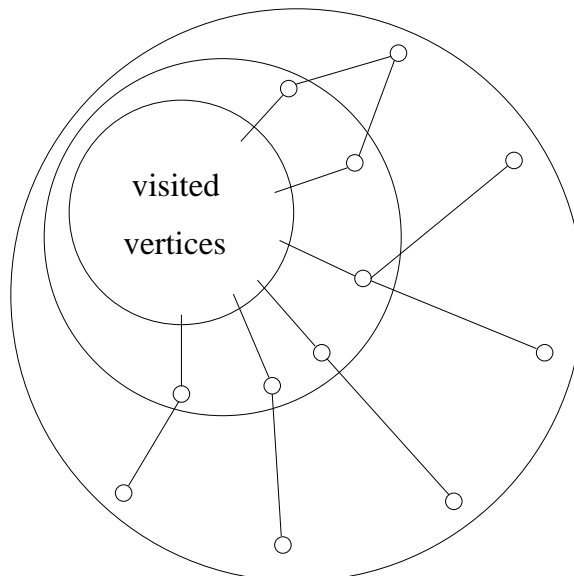


Figure 9.4: The graph is discovered “layer by layer”

$\delta(s, v) = \text{layer where } v \text{ is discovered}$

The visited vertices are expanded by adding new nodes to the outmost layer.

The running time is $O(|E|)$.

9.3.3 Floyd-Warshall Algorithm

Definition 9.2 $d_{ij}^{(k)}$: Length of shortest path from i to j where the only intermediate vertices used are v_1, v_2, \dots, v_k .

Example 2:

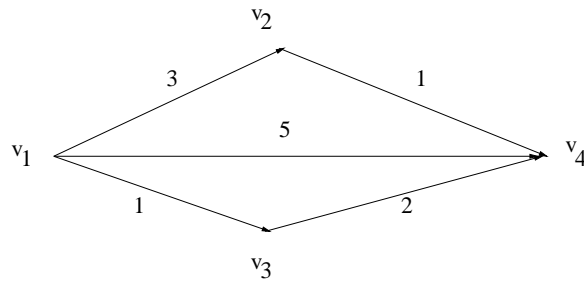


Figure 9.5: The graph is discovered “layer by layer”

The distances between v_1 and v_4 through different sets of intermediate vertices are as follows:

$$\begin{cases} d_{14}^{(1)} & = & 5, \\ d_{14}^{(2)} & = & 4, \\ d_{14}^{(3)} & = & 3, \end{cases}$$

Compute recursively by using the formula:

$$\begin{aligned} k \geq 1 & : d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \\ k = 0 & : d_{ij}^0 = w_{ij} \end{aligned}$$

where

$$w_{ij} = \begin{cases} w_{ij}, & \text{if } (i, j) \in E. \\ \infty, & \text{if } (i, j) \notin E. \end{cases}$$

Then the shortest paths will be in $d_{ij}^{|V|}$. A single step is shown in figure 9.6.

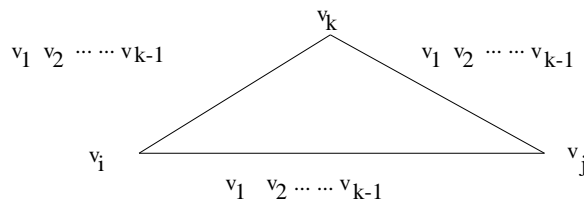


Figure 9.6: A single step to calculate the shortest path to v_j

This procedure can be represented by the following diagram. We can construct a three-dimensional table for (i, j, k) with the top level representing $k = 0$ and the final level representing the result.

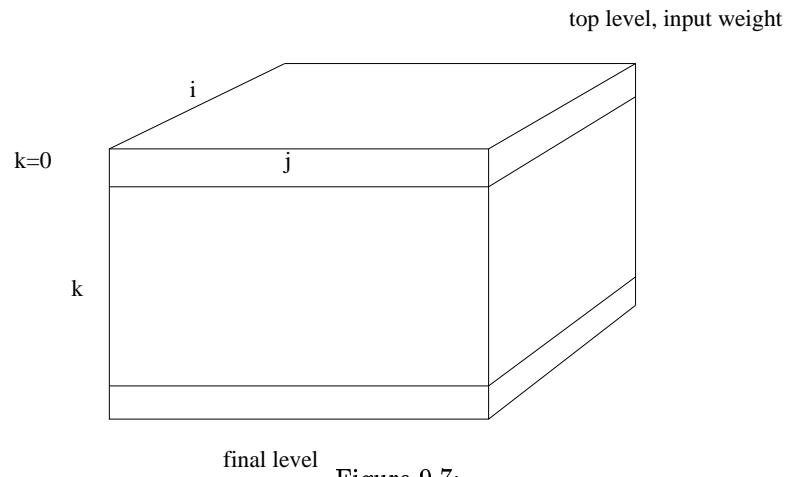


Figure 9.7:

Algorithm: Let's call the table D and $D^{(0)}$ represents layer 0.

1. $D^{(0)} = w$
2. for $k=1$ to $|V|$
3. for $i=1$ to $|V|$
4. for $j=1$ to $|V|$
5. $D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$
6. return $D^{(|V|)}$

Running time: $O(|V|^3)$

References

RD00 T. RATH and C. DANSONG, Scribe Notes for Lecture 8, UMASS course 611: Advanced Algorithms, Fall 2000.