Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 27

Dynamic Programmig V - 0/1 Knapsack and Maximal Independent Set

1 The Knapsack Problem

This is a classic problem, defined as the following:

We have n items, each with a value and a positive weight. The i-th item has weight w_i and value v_i. We have a knapsack that holds a maximum weight of W. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

Item	Weight	Value
A	6	25
B	3	13
C	4	15
D	2	8

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we saw last lecture, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

1.1 The 0-1 Knapsack Problem

We consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item A and item C, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension to the DP table). Let $K(x, j)$ be the maximum value that we can get with a knapsack of

capacity x considering only items at indices from $1, \cdots, j$. Consider the optimal solution for $K(x, j)$. There are two cases:

1. Item j is used in $K(x, j)$. Then, the remaining items that we choose to put in the knapsack must be the optimum solution for $K(x - w_j, j - 1)$. In this case, $K(x, j) = K(x - w_j, j - 1) + v_j$.
2. Item j is not used in $K(x, j)$. Then, $K(x, j)$ is the optimum solution for K(x, j - 1). In this case, $K(x, j) = K(x, j - 1)$.

So, our recurrence relation is: $K(x, j) = \max\{K(x - w_j, j - 1) + v_j, K(x, j - 1)\}$. Now, we're done: we simply calculate each entry up to $K(W, n)$, which gives us our final answer. Note that this also runs in $O(nW)$ time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do $O(1)$ work.

Algorithm 1: ZeroOneKnapsack(W, n, w, v)

```
for j = 1, ··· , n do
    K[0, j] ← 0
for x = 0, ··· , W do
    K[x, 0] ← 0
for x = 1, ··· , W do
    for j = 1, ··· , n do
        K[x, j] ← K[x, j − 1]
        if w_j <= x then
            K[x, j] = max{K[x − w_j, j − 1] + v_j, K[x, j]}
return K[W, n]
```

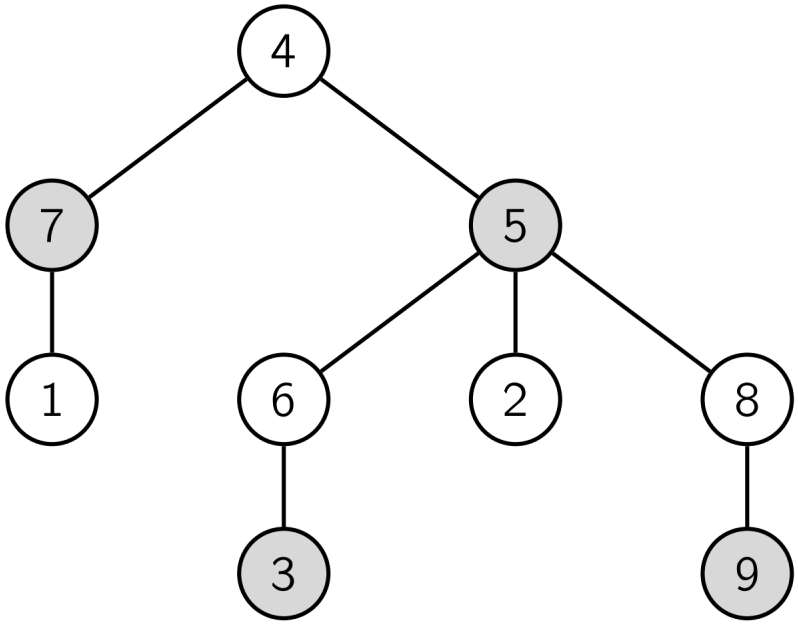
2 The Independent Set Problem

This problem is as follows:

Say that we have an undirected graph $G = (V, E)$. We call a subset $S \subseteq V$ of vertices "independent" if there are no edges between vertices in S. Let vertex i have weight w_i , and denote $w(S)$ as the sum of weights of vertices in S. Given G, find an independent set of maximum weight $arg\max_{S \subseteq V} w(S)$.

Actually, this problem is NP-hard for a general graph G. However, if our graph is a tree, then we can solve this problem in linear time. In the following figure, the maximum weight independent set is highlighted in blue.

Remark 1. *Dynamic programming is especially useful to keep in mind when you are solving a problem that involves trees. The tree structure often lends itself to dynamic programming solutions.*



As usual, the key question to ask is, "What should our subproblem(s) be?" Intuitively, if the problem has to do with trees, then subtrees often play an important role in identifying our subproblems. Let's pick any vertex r and designate it as the root. Denoting the subtree rooted at u as T_u, we define $A(u)$ to be the weight of the maximum weight independent set in T_u. How can we express $A(u)$ recursively? Letting S_u be the maximum weight independent set of T_u, there are two cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children v of u.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v A(v)$ for all grandchildren v of u.

To avoid solving the subproblem for trees rooted at grandchildren, we introduce $B(u)$ as the weight of the maximum weight independent set in $T_u \setminus \{u\}$. That is, $B(u) = \sum_v A(v)$ for all children v of u. Equivalently, we have the following cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children v of u.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v B(v)$ for all children v of u.

So, we can calculate the weight of the maximum weight independent set:

$$A(u) = \max \left\{ w(u) + \sum_{v \in Children(u)} B(u), \sum_{v \in Children(u)} A(v) \right\}$$

To create an algorithm out of this recurrence, we can compute the $A(u)$ and $B(u)$ values in a bottom-up manner (a post-order traversal on the tree), arriving at the answer, $A(r)$. This takes $O(|V|)$ time.

Algorithm 2: MaxWeightIndependentSet(G)

```
// G is a tree
r ← ArbitraryVertex(G)
T ← RootTreeAt(r)
procedure SolveSubtreeAt(u)
    if Children(T, u) = ∅ then
        A(u) ← w_u
        B(u) ← 0
        else
        for v ∈ Children(u) do
            SolveSubTreeAt(v)
        A(u) ← max{ Σ_{v∈Children(u)} A(v), w_u + Σ_{v∈Children(u)} B(v) }
        B(u) ← Σ_{v∈Children(u)} A(v)
end procedure
SolveSubtreeAt(r)
return A(r)
```