Dynamic Programming: When does Dynamic Programming Fail?[1]

1 When does Dynamic Programming Work?

We have seen some examples of problems that could be solved by dynamic programming. You have seen more examples in your problem sets (weekly, advanced, ungraded). Hopefully you see that there are ***two*** things required to make an efficient dynamic programming solution:

- **Recursive Structure.** Given an instance $\mathcal{I}$ of a problem, we should be able to break into smaller instances $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k$ such that (a) an "optimal" solution $\mathcal{S}$ to $\mathcal{I}$ can be used to obtain an "optimal" solution $\mathcal{S}_j$ for *some* $\mathcal{I}_j$, and (b) given "optimal" solutions $\mathcal{S}_1, \ldots, \mathcal{S}_k$ to the smaller instances, one can obtain an "optimal" solution to $\mathcal{I}$.

  The way we usually did this is to *imagine* an "optimal" solution $\mathcal{S}$, and then break this by arguing about whether it "contained" the "last" element or not. And then we argued that when it did, then the "remaining" items formed an optimal solution to a smaller sub-problem.

- **Small Number of Problems.** Given any instance $\mathcal{I}$, one obtains the smaller subproblems $\mathcal{I}_1, \ldots, \mathcal{I}_k$, and then for each $\mathcal{I}_j$, one obtains even smaller problems and so on and so forth. We ***must*** be able to control the number of such sub-problems seen. One way to do this is to *observe* that any small sub-instance seen somewhere in this "recursion tree" can be *parametrized* by a *few* parameters which ranges within certain "manageable" values.

  For example, in the knapsack problem these parameters were m and b, where m ranged from 0 to n, b ranged from 0 to B, and the instance parameterized by m, b only looked at the "first" m items and a knapsack of capacity b.

These two steps, once figured out, led us to the six-step approach to writing a dynamic programming solution precisely. The most important step was the *definition*; this involves a function parameterized by the parameters which governed each smaller subproblem. Equally important is the *recurrence* which rigorously states the recursive structure of the problem. If you go and investigate each and every dynamic programming problem you have solved (or will ever solve), you should see that these two features leaping out.

2 An Example where DP doesn't seem to help: Independent Sets in Graphs

Given an undirected graph $G = (V, E)$, a subset $I \subseteq V$ is said to be *independent* if **no** two $u, v \in I$ have an edge between them. The Independent Set problems takes input a graph and outputs the largest sized independent set.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

MAXIMUM INDEPENDENT SET
Input: Undirected graph $G = (V, E)$ with n vertices and m edges.
Output: Independent set I of the largest size/cardinality

Let the vertices of G be named $\{v_1, v_2, \ldots, v_n\}$ arbitrarily. Let us imagine S to be a largest sized independent set. As has been working for us well, let us consider whether S contains the last vertex $v_n \in V$. Two cases arise. Case 1: the vertex $v_n \notin S$. In that case, S must also be the largest sized independent set in $G - \{v_n\}$, the graph which has the vertex v_n deleted. What about Case 2 : the vertex $v_n \in S$. What can we say about the remaining solution $S - \{v_n\}$? It is indeed true that $S - \{v_n\}$ is an independent set of $G - \{v_n\}$. However, it *may not* be the **largest** independent set of $G - \{v_n\}$. The example below in Figure 1 illustrates this.

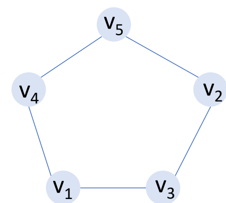

Figure 1: *The set $S = \{v_1, v_5\}$ is a largest independent set of G. The set $\{v_1\} = S - \{v_5\}$ is an independent set of $G - \{v_5\}$, but not the largest one. Rather, the* ***two*** *neighbors $\{v_2, v_4\}$ forms the largest independent set of $G - \{v_5\}$.*

Thus, we ***don't*** have the recursive structure in this way. This should remind you of the "mistake" in the first try for LIS in the previous lecture. And as in there, it doesn't mean the "death" of the DP approach, it is just that we probably have to shift our point of view.

And so we ask ourselves: can we find a smaller instance for which $S - \{v_n\}$ is indeed the largest independent set? The answer is yes. Consider the graph $G' = G - N_G(v_n)$ where $N_G(v_n) = v_n \cup \{u : (u, v_n) \in E\}$, that is, itself *plus* the neighbors of v_n in G. We now observe that any independent set of G' can be augmented with v_n to get an independent set of G which is one larger. Therefore, $S - \{v_n\}$ should be the largest independent set in G', or otherwise we could augment that to get a larger independent set in G. In the example in Figure 1, $\{v_1\}$ is a largest independent set in $G' = G - \{v_5, v_2, v_4\}$, that is, the single edge graph $\{v_1, v_3\}$.

Therefore, to find the largest independent set in G, we need to be told the largest independent set in $G_1 = G - \{v_n\}$ and the largest independent set in $G_2 = G - N(v_n)$. The base case is when the graph has only one vertex in which case the solution is the singleton. And thus, we have found our recursive structure. Yay!

More precisely, if we define $I(G)$ to be the size of the largest independent set in G, then we get

$$\text{For any vertex } v \in G, \quad I(G) = \max(I(G - v), 1 + I(G - N(v))) \tag{1}$$

However, there is a *big* snag, which is that the number of subproblems is not bounded. For instance to get the largest independent set in $G_1 = G - \{v_n\}$, we need to know the largest independent set in $G_1 - \{v_{n-1}\} = G - \{v_{n-1}, v_n\}$, but also in $G_1 - N(v_{n-1})$. Similarly, to get the largest independent set in G_2, assuming $v_{n-1} \in G_2$ (and not deleted in $N(v_1)$), we would need to know the largest independent set not only in $G_2 - \{v_{n-1}\}$, but also $G_2 - N(v_{n-1}) = G - (N(v_n) \cup N(v_{n-1}))$.

If you stare at it for a moment more , you will observe the *combinatorial explosion*: to get the largest independent set in G, we may have to know the largest independent set in $G - S$ for *most* subsets $S \subseteq V$. In other words, when we think how the instances/graphs are getting smaller, we observe they are being induced over subsets of vertices over which we don't a priori have control over structure. Said differently, we cannot see any small number of patterns governing the subproblems solved, and each subproblem seems to be indexed/parameterized by the subset itself. Therefore, in the bottom up approach, we would need to store the largest independent set in graphs induced by *all* subsets of vertices; a ludicrous proposition since at the same time we could just check all subsets and return the best!

> **Remark:** *Nevertheless, if someone puts a gun on your head and asks for the largest independent set, you could try the* memoization *approach to implement the recurrence* (1). *You may just get lucky, but don't count on it. There are better ways to practically compute the largest independent set, but none of them have guaranteed running time theorems.*

So, what was the point of all this? Two points. One, dynamic programming is not a panacea even when recursive structure exists. The *number* of smaller instances that need to be solved in all should be manageable. Till now, in the problems we saw, this occurred because if our solution contained a "last" element, then the remaining part of the solution did form a optimal solution of a not only smaller but "structured" sub-problem (like the "first" m items, or prefixes of strings). This is important – without it dynamic programming is either not straightforward, or just plain impossible to work with.

The second point, specific to independent set, is that if our instances *do have structure* in their neighborhood sets $N(v)$, then perhaps we can use the ideas above to get a good dynamic programming solution. Indeed, one example is given in the UGP as the "weighted interval packing" problem. Do you see why that is an (weighted) independent set problem? The other example is on trees which we describe next.

3 Independent Set on a Tree

Our final example of dynamic programming is the independent set problem on trees. To make things precise, we are focussing on *rooted* trees. Given a tree $T = (V, E)$ rooted at r, every vertex v which is not a root has a unique *parent* denoted as $p(v)$, and every vertex v which is not a leaf has children stored in the list $\mathsf{chld}(v)$. The picture below is an illustration.

> <u>MAXIMUM INDEPENDENT SET IN A TREE</u>
> **Input:** Rooted Tree T on n vertices with root r. Every vertex $v \in T$ has a weight $w(v)$.
> **Output:** Independent set I of the largest weight.

Note that any path is also a rooted tree; one can think of the whole path hanging from the last vertex. This gives an idea of ordering on the tree – starting from root, downwards. Using this we argue about the recursive structure as follows.

Consider the optimal independent set S in the tree T. We branch on two cases.

Case 1, the root $r \notin S$. In that case, as we argued before, S will be an optimal independent set in the graph $T - r$. Note, however, that $T - r$ is no longer a tree, and thus we don't quite have the same problem! However, $T - r$ breaks into a bunch of trees: $T_1, \ldots, T_k$ where k is the number of children of the root r. Suppose $S_j := S \cap T_j$ for all $1 \leq j \leq k$, that is, the part of S in the sub-tree T_j. Then we note that each S_j must be the optimal independent set in T_j. Intuitively (formal proof coming up), this is because what we select in T_j in our independent set doesn't affect what we select in some other $T_{j'}$.

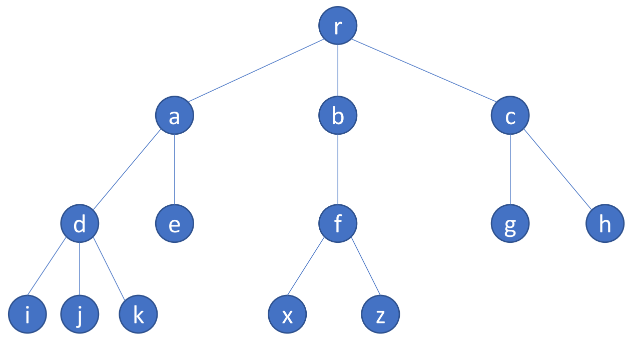

Figure 2: *r is the root of the tree.* $\mathsf{chld}(r) = \{a, b, c\}$ *and* $p(a) = r$*. Note that any vertex in the tree has a sub-tree rooted at that vertex.*

Case 2, the root $r \in S$. In this case we are sure that none of r's children can be in S. Now consider the "grandchildren" of the root r. That is, the union of $\mathsf{chld}(v_i)$ for all $v_i \in \mathsf{chld}(r)$. Let $U_1, \ldots, U_\ell$ be the trees rooted at these grandchildren. We see that $S - r$ must be partitioned into ℓ classes, and each of these classes must be optimal independent sets in the corresponding tree U_j. The base case are the leaves; in this case the optimal independent set is clearly the leaf itself.

We have obtained our recursive structure, and we are ready to state the dynamic program in our template.

- *Definition:* Given tree T and any vertex $v \in T$, we define $\mathsf{ISTree}(v)$ to be the weight of the maximum weight independent set in the tree T_v rooted at the vertex v. We are interested in $\mathsf{ISTree}(r)$.

- *Base Cases:* $\mathsf{ISTree}(\perp) = 0$ where $\perp$ is a null vertex; $\mathsf{ISTree}(\ell) = w(\ell)$ for every leaf ℓ.

- *Recursive Formulation:* For any non-leaf $v \in T$, let $\mathsf{chld}(v)$ be the set of its children and let $\mathsf{chld}^2(v)$ be the set of its grandchildren. Formally, $\mathsf{chld}^2(v) := \bigcup_{u \in \mathsf{chld}(v)} \mathsf{chld}(u)$.

$$\mathsf{ISTree}(v) = \max \left(\sum_{u \in \mathsf{chld}(v)} \mathsf{ISTree}(u),\ w(v) + \sum_{z \in \mathsf{chld}^2(v)} \mathsf{ISTree}(z) \right)$$

- *English Explanation:* Given above, along with the explanation for Independent Set.

- *Formal Proof:* To formally prove the above, it helps to introduce the notation of $\mathsf{Cand}(v)$ to be the set of all *independent sets* of the sub-tree T_v rooted at v.

$$\mathsf{ISTree}(v) = \max_{S \in \mathsf{Cand}(v)} w(S)$$

  where $w(S) := \sum_{x \in S} w(x)$.

  ($\leq$): Let $S \in \mathsf{Cand}(v)$ be an independent set in T_v with $w(S) = \mathsf{ISTree}(v)$.

  * *Case 1:* $v \notin S$. In this case, $S_u := S \cap T_u$ for every $u \in \mathsf{chld}(v)$ is an independent set in T_u. Thus, $w(S_u) \leq \mathsf{ISTree}(T_u)$ implying $w(S) = \sum_{u \in \mathsf{chld}(v)} w(S_u) \leq \sum_{u \in \mathsf{chld}(v)} \mathsf{ISTree}(u)$.

- * *Case 2:* $v \in S$. This implies $S \cap \mathsf{chld}(v) = \emptyset$. Let $S_z := S \cap T_z$ for every $z \in \mathsf{chld}^2(v)$. Since S_z is independent, $w(S_z) \leq \mathsf{ISTree}(z)$. Then, we get $w(S) = w(v) + \sum_{z \in \mathsf{chld}^2(v)} w(S_z) \leq w(v) + \sum_{z \in \mathsf{chld}^2(v)} \mathsf{ISTree}(z)$.

  In each case, $\mathsf{ISTree}(v)$ is less than one of the two things in the RHS.

  ($\geq$): Since there is no edge between two vertices in sub-trees of two *different* children of v, we get that the union of the independent sets in the trees rooted at the children of v must form an independent set in the tree rooted at v. Similarly, the union of the independent sets in the trees rooted at grandchildren and the vertex v is also an independent set in the tree rooted at v. Therefore, $\mathsf{ISTree}(v)$ is greater than both the terms in the RHS.

- *Pseudocode for computing value of Independent Set on a tree.*

1: **procedure** INDEPEDENTSETTREE(T, w):
2:     ▷ *Returns the maximum weight independent set*
3:     We assume we have access to the tree as layers. L_1 is the set of leaves, L_2 is the set of vertices with all children in L_1; L_3 is the set of vertices with all children in L_2, and so on and so forth. Let h be the number of layers.
4:     We also assume we have a data structure which stores $\mathsf{chld}(v)$ and $\mathsf{chld}^2(v)$ for all vertices v.
5:     Allocate space $I[v]$ for every vertex v. ▷ *$I[v]$ will contain $\mathsf{ISTree}(v)$.*
6:     **for** $1 \leq i \leq h$ **do**:
7:         **for** $v \in L_i$ **do**:
8:             $I[v] = \max\left(\sum_{u \in \mathsf{chld}(v)} I[u],\ w(v) + \sum_{z \in \mathsf{chld}^2(v)} I[z] \right)$
9:             ▷ *Note that $I[u]$ and $I[z]$ are defined since they appear in lower layers.*
10:     ▷ *$I[r]$ now contains the value $\mathsf{ISTree}(r)$*

- *Recovery.* We have left the recovery code for the independent set out of the above pseudocode (out of laziness) and describe it in words. We maintain a queue Q which initially contains r. At each step we pick the first vertex v of the Q and check if $I[v]$ equals the first summation in Line 8 or the second. In case of the first, we add $\mathsf{chld}(v)$ to Q; in case the second, we add v to S and add all the $\mathsf{chld}^2(v)$ to Q. Since whenever we add v to S we remove all its $\mathsf{chld}(v)$ from consideration, the returned set S is independent.

  It can be seen that at each step the following invariant remains true

$$w(S) + \sum_{u \in Q} I[u] = I[r]$$

  Since we end when the Q is empty, we end up with an independent set of weight $I[r]$.

- *Running time and space.* The above pseudocode take $O(n)$ space and $O(n)$ time, given the above data structures are in place. All these can also be done in $O(n)$ time.

Theorem 1. The maximum weight independent set in a tree can be found in $O(n)$ time and space.