Dynamic Programming: When does the idea not work?[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.

  For example, in the knapsack problem, the "optimal" solution was the subset S of items. If the "last" (which, for this problem, can be picked arbitrarily) item n is not in the subset, then the smaller problem looks at only the first $n-1$ elements with the same knapsack. If it is in the set, then we look at the smaller problem with the first $n-1$ elements with a knapsack whose capacity has been decreased by the weight of the last element. If one is given the best solution to both these problems, then one could obtain the best solution to the original problem by taking the best of the solution given by the first problem, and the solution given by the second problem *added* with the last item.

- **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 these two features leaping out.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 22nd Apr, 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!

2 DP Fail: No (Apparent) Recursive Structure

Let me now mention a generalization of the knapsack problem where the above recursive structure which worked for knapsack fails. The setting is similar: the input contains n items and a knapsack of capacity B. Each item has a weight w_i and a solution is a subset $S \subseteq \{1, \ldots, n\}$ with $\sum_{i \in S} w_i \leq B$. However, the profits are not as simple. Rather, for any subset S of items, there is a profit $p(S)$ which is not necessarily a linear function. That is, unlike the original knapsack problem, $p(S)$ is *not* equal to $\sum_{j \in S} p_j$.

For an illustration, consider the following 4 items: (1) peanut butter, (2) milk, (3) organic milk, and (4) jelly. Suppose they cost (and let this be the weight), \$3, \$4, \$6, and \$2, respectively. You have come to the market with a budget of \$9. Now, *individually* these items give you happiness of 2, 4, 5, and 3 units, respectively. So, if you bought peanut-butter and that was the only thing you bought, you get a happiness of 2 unit. And if you bought jelly and that was the only thing you bought, you get a happiness of 4 units. However, if you bought peanut-butter *and jelly*, then it could be your happiness is 7 units[2]. In other words, $p(\{1\}) = 2$ and $p(\{4\}) = 4$, but $p(\{1, 4\}) = 7 > p(\{1\}) + p(\{4\})$. In Econ parlance, the items peanut butter and jelly are complements. Similarly, it could be that if you bought both milk *and* organic milk, your happiness is not $4 + 5 = 9$, but perhaps 0 (too much milk and milk spoils fast). In Econ parlance, these two items are substitutes. For completeness in this illustration, let's say all other subsets are additive: so if you bought milk and jelly, the total happiness is say 7 units. The question is what's the best set of items to buy that fits the budget.

Suppose we try to do the same thought experiment to solve the problem with this complicated profit function. Once again, we can imagine the optimal subset S. We ask ourselves, if item 4, jelly, is in S or not. If not, then we can piggy back and solve the problem on items $\{1, 2, 3\}$ with budget unchanged. That is a smaller problem. However, the issue comes up when $4 \in S$. If the profits were additive, we would piggy back on to the items $\{1, 2, 3\}$ with budget reduced to \$9 − \$2 = \$7. And suppose we did solve this and the optimum solution to this smaller problem was $\{1, 2\}$ giving a total profit of 6 units of happiness. Now when we try to go back to the original problem and *add* jelly back, the total profit is *not* 6, the answer to the smaller problem, $+3$ which is the profit of jelly. Rather, it is more than that since items 1 and 4 are complements. In short, given the *value* of a solution to the smaller subproblem, we *cannot* figure out the *value* of the solution to the bigger problem. And if we can't do that, we can't compare between the two subproblems: one without jelly, and one with.

In short, since the objective function was not a decoupled function of the various choices (unlike sum of individual profits/multiplication of individual profits), the choices made for the "nth" can't be decoupled when you try and solve the smaller problem recursively. Contrapositively, if you are trying to solve an optimization problem with DP, it is often instructive to see whether one can write the objective as a decoupled (the technical parlance is separable) function of the various decision choices you need to make. For instance, going back to the illustration, we would take the 4 items and break into two *groups*: $G_1 = \{1, 4\}$ and $G_2 = \{2, 3\}$. Note that by assumption *across* groups the profit is additive. That is if we buy $\{1, 3, 4\}$, that is, peanut butter, milk, and jelly, then the profit would be 7 for the PBJ plus 4 for the milk. Mathematically, $p(S) = p(S \cap G_1) + p(S \cap G_2)$. More generally, if there were n items divided into k groups and each group of items (hopefully containing 2-3 items) had possible complements/supplements, but across groups things were independent, then we would say $p(S) = p(S \cap G_1) + \cdots + p(S \cap G_k)$. Once we do that, DP is back again in the picture. We ask ourselves, what subset of group G_k does the optimum solution pick? If the group is small, then this number of choices is small too. And once we have that, then we can piggy back to a smaller subproblem with fewer number of groups, and a modified budget. Let me stop here.

[2] maybe you have bread lying around in the house for a PB&J

3 DP Fail: Too many subproblems

Let's now look at a problem where the profit function is additive, but the issue is that there are too many subproblems. 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.

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.

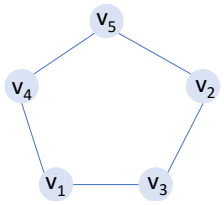

Figure 1: *The set $S = \{v_1, v_5\}$ is a largest independent set of G. The set $\{v_1\} = S - \{v_5\}$ is an independent set of $G - \{v_5\}$, but not the largest one. Rather, the **two** neighbors $\{v_2, v_4\}$ forms the largest independent set of $G - \{v_5\}$.*

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.

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.

4 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.

MAXIMUM INDEPENDENT SET IN A TREE
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.

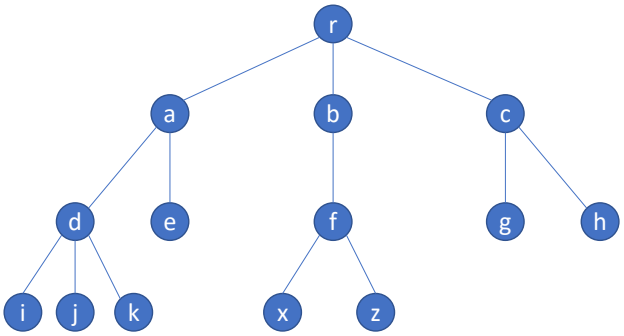

Figure 2: *r is the root of the tree.* $\mathsf{chld}(r) = \{a, b, c\}$ *and* $p(a) = r$. *Note that any vertex in the tree has a sub-tree rooted at that vertex.*

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 $T_{j'}$.

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.

$$\mathsf{ISTree}(v) = \max_{S \in \mathsf{Cand}(v)} \underbrace{w(S)}_{\sum_{x \in S} w(x)}$$

($\leq$): 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 chld(v) and chld^2(v) for all
        vertices v.
5:      Allocate space I[v] for every vertex v. ▷ I[v] will contain ISTree(v).
6:      for 1 ≤ i ≤ h do:
7:          for v ∈ L_i do:
8:              I[v] = max( Σ_{u∈chld(v)} I[u], w(v) + Σ_{z∈chld^2(v)} I[z] )
9:              ▷ Note that I[u] and I[z] are defined since they appear in lower layers.
10:     ▷ I[r] now contains the value 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.