1 Dynamic Programming

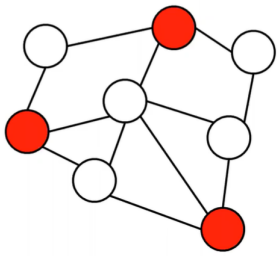
We continue our discussion on dynamic programming.

1.1 Problem: Maximum Independent Set of Trees

Definition 1.1: Independent Set

In an undirected graph G, an **independent set** is a subset of the vertices of G, no two of which are connected by an edge.

For example, the following is an independent set:



Problem Statement: Given an undirected graph G, compute the largest possible **size** of an independent set of G.

Call the result of the largest possible independent set $I(G)$; in the example above, $I(G) = 3$.

1.1.1 Simple Recursion

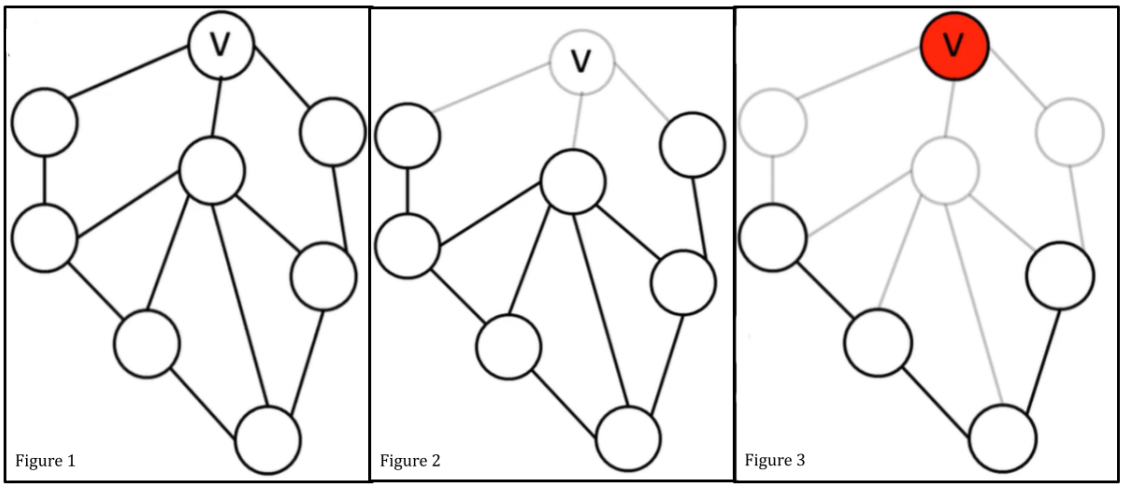


Figure 1 Figure 2 Figure 3

There is a reasonably simple recursion that can be used here. A simple question we can ask is: *is vertex v in the independent set?* We'll consider Figure 1 above as a baseline example.

- If v is not in the independent set, then the maximum independent set is an independent set of $G \setminus \{v\}$. In other words, we have
$$I(G) = I(G \setminus \{v\})$$
For example, Figure 2 shows what this could look like.

- If so, then the maximum independent set is v plus an independent set of $G \setminus N(v)$, where $N(v)$ denote the set of vertices which are neighbors of v. In other words, we have

$$I(G) = 1 + I(G \setminus N(v))$$

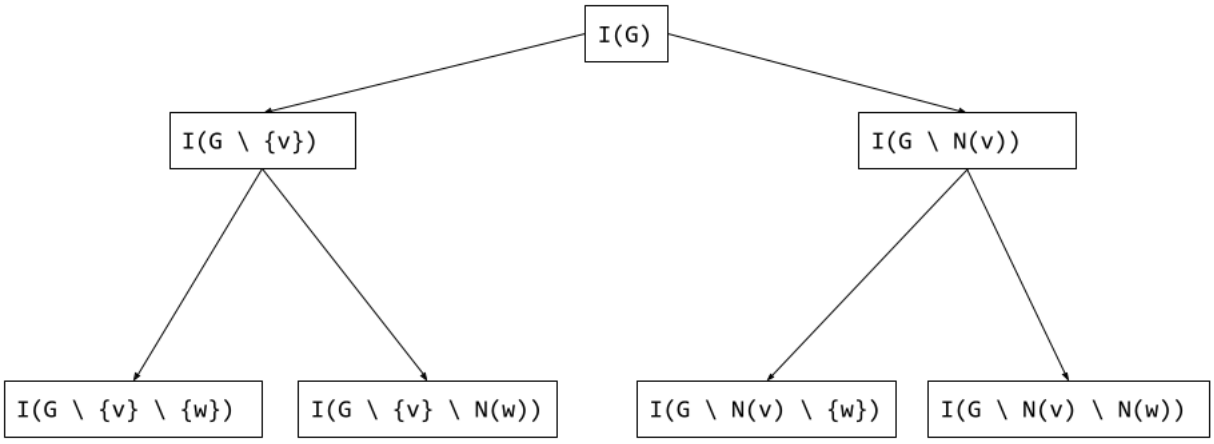
  For example, Figure 3 shows what this could look like.

Therefore, our recursion can be defined by

$$I(G) = \max(I(G \setminus \{v\}), 1 + I(G \setminus N(v)))$$

1.1.2 Subproblems

We now consider the subproblems generated by this recursion.



Here, it's not hard to see that, if we continued generating this recursion tree, we see very little subproblem reuse. In other words, we end up with *different subproblems* every time we get a new recursion. This is an issue for dynamic programming, as dynamic programming relies on subproblem reusability for efficiency.

In particular, for every subgraph G', we need subproblems $I(G')$. But, we'll end up with $2^{|V|}$ subproblems; each vertex can either be in G' or it can not be in G'. This further implies that the runtime of this algorithm will be exponential.

1.1.3 Hardness

Maximum Independent Set of any general graph is what is known as an **NP-Hard** problem. Basically, this means that people believe that there may well be no *efficient* algorithm for it (there is no sub-exponential runtime algorithm that solves this problem).

Instead, we now consider *special cases* of this problem where we can do better.

1.1.4 Independent Sets and Components

Lemma 1.1

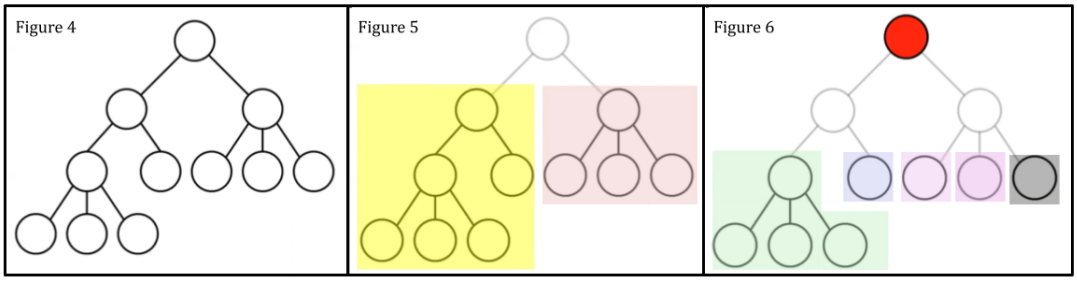
If G has connected components $C_1, C_2, \ldots, C_k$, then

$$I(G) = I(C_1) + I(C_2) + \cdots + I(C_k)$$

Proof. Since the components don't connect to each other, an independent set for G is exactly the union of an independent set for each of the C_i's. THen, we can pick the biggest set for each C_i. □

1.1.5 Independent Sets of Trees

One special case we can consider is independent sets of *trees*. The reason why we look at trees is because, when we remove vertices from trees, it splits the tree up into disconnected subgraphs. This is useful as we can consider the disconnected subgraphs separately.



Consider the above tree (Figure 4), and suppose we're focusing on the root of this tree r.

- If we do *not* include r in our maximum independent set, then we want to look at the maximum independent sets of $G \setminus \{r\}$. For this, see Figure 5.
- If we *do* include r in our maximum independent set, then we get r for free, remove all of the neighbors of r, and then we can consider the remaining subgraphs. For this, see Figure 6.

We note that the subproblems here are actually all *sub-trees* of the original tree. If we look at a subtree of a subtree, we note that this is simply just another subtree. Then, it follows that the set of all problems that we need to solve is just the set of all possible subtrees of the original tree.

1.1.6 Recursion of the Tree

We now consider several cases for our tree.

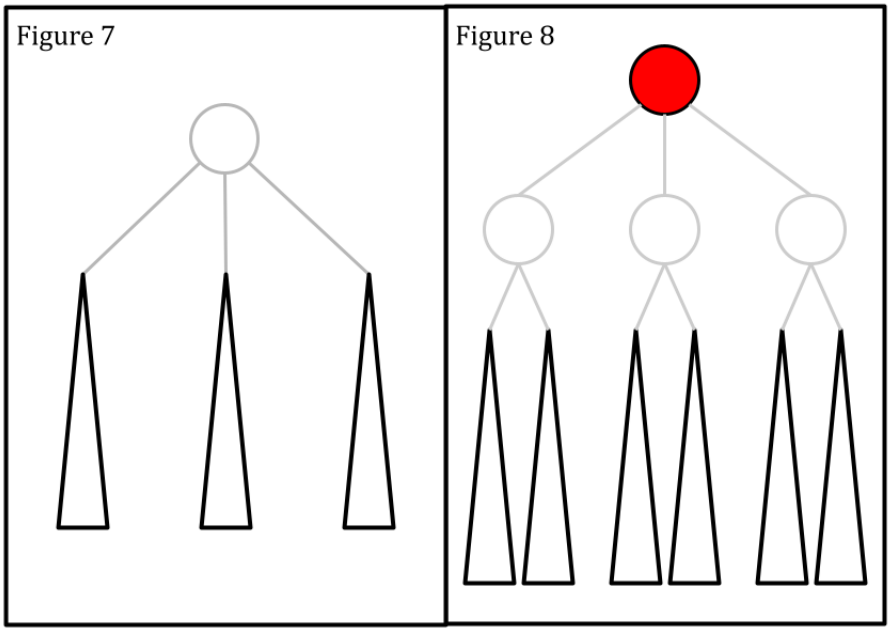
- Suppose the root is not used in the tree. Then, we end up with the subproblem of having to find the sum of the maximum independent sets of all of the children subtrees. That is,

  $$I(G) = \sum I(\text{Children's Subtrees})$$

  This is shown by Figure 7 below.
- Suppose the root is used. Then, we end up with the subproblem of having to find the sum of all of the maximum independent sets of all of the grandchildren subtrees. That is,

  $$I(G) = 1 + \sum I(\text{Grandchildren's Subtrees})$$

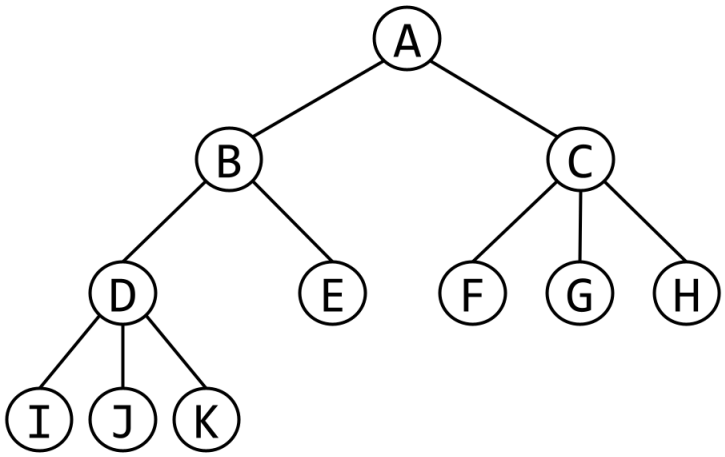
  This is shown by Figure 8 below.



Note: Triangles are arbitrary subtrees.

1.1.7 Example: Computing Maximum Independent Set of Trees

Consider the following tree:



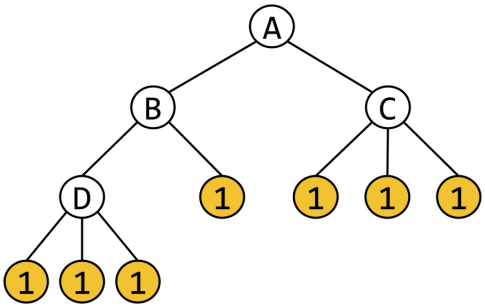
Compute the maximum independent set.

Remember, our subproblems are the subtrees. So, for each node, we associate that node to the subproblem that corresponds to what is the maximum independent set of that node's subtree. The formula that we have is that this will always be the maximum of either:

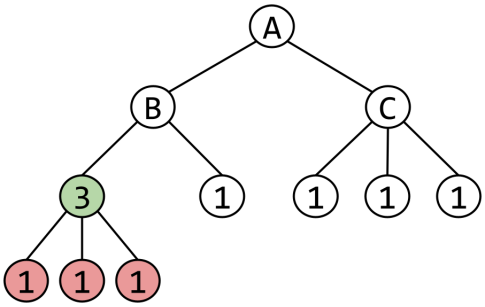
- The sum of the children's answers.
- One more than the sum of the grandchildren's answers.

So, we start at the leaves and make our way up.

- At the bottom level, each leaf (I, J, K, E, F, G, H) is going to have an answer of 1. If we just have that single node, the biggest independent set is going to be of size 1.

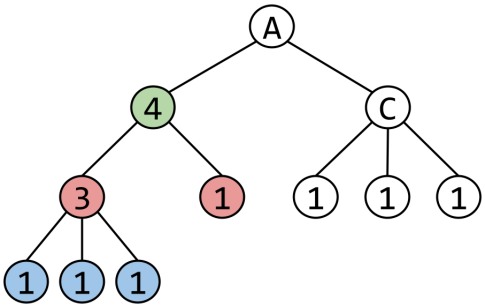


- Consider node D. We can either take the sum of the children $(1+1+1=3)$ or we can take one plus the sum of the grandchildren $(1+0=1)$. Clearly, the sum of the children is the best option, so D will be 3.



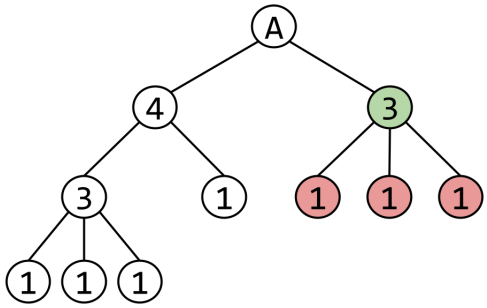
Note that the children nodes are *red* while node D is *green*.

- Consider node B. We can either take the sum of the children $(3+1=4)$ or we can take one plus the sum of the grandchildren $(1+(1+1+1)=4)$. Both options are good, so B will be 4.



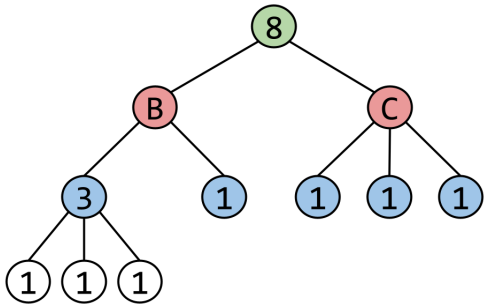
Note that the children nodes are *red*, the grandchildren nodes are *blue*, and node B is *green*.

- Consider node C. We can either take the sum of the children $(1+1+1=3)$ or we can take one plus the sum of the grandchildren $(1+0=1)$. Clearly, the sum of the children is the best option, so C will be 3.



Note that the children nodes are *red* and node C is *green*. Additionally, note that this is actually a subproblem that we've seen before.

- Consider node A. We can either take the sum of the children $(4 + 3 = 7)$ or we can take one plus the sum of the grandchildren $(1 + (3 + 1 + 1 + 1) = 8)$. Clearly, one plus the sum of the grandchildren is the best option, so A will be 8.



Note that the children nodes are *red*, the grandchildren nodes are *blue*, and node A is *green*.

Therefore, the maximum independent set of the *whole tree* is $\boxed{8}$.

1.2 Problem: Traveling Salesman Problem

In your job as a door-to-door vacuum salesperson, you need to plan a route that takes you through n different cities. In order to space things out, you do not want to get back to the start until you have visited all cities. You also want to do so with as little travel as possible.

Problem Statement: Given a weighted (undirected) graph G with n vertices, find a cycle that visits each vertex exactly once whose total weight is as small as possible.

1.2.1 Naive Algorithm

The naive algorithm is to try all possible paths and see which is the cheapest. There are $n!$ paths; in particular, if we assume that we have a complete graph where all edges are there, then:

- There are n possible options for the first city.
- There are $n - 1$ possible options for the second city.
- There are $n - 2$ possible options for the third city.
- ...

This gives us $n(n-1)(n-2)\dots(2)(1) = n!$. Therefore, the runtime of the naive algorithm is approximately $\mathcal{O}(n!)$.

1.2.2 Problem Difficulty

Like Maximum Independent Set, the Traveling Salesman Problem is widely considered to be a difficult problem. It is widely believed that there is no polynomial time algorithm for it. That being said, we note that there is an algorithm that beats the $n!$ naive algorithm.

1.2.3 Setup

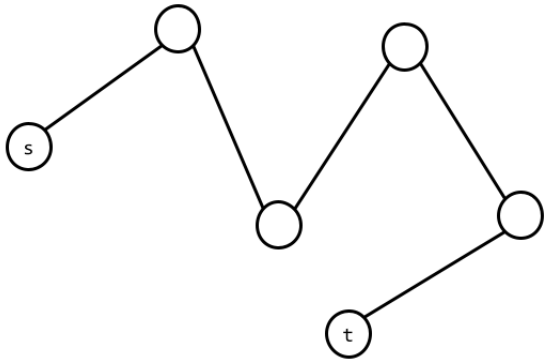
We need to find some notion of *partial solutions* for subproblems. In our case of cycles, we want to look for a path from vertex s to t such that, by adding one more vertex, we get a cycle. Therefore, the answer is given by

$$\text{Best}_{st}(G) = \ell(s, t)$$

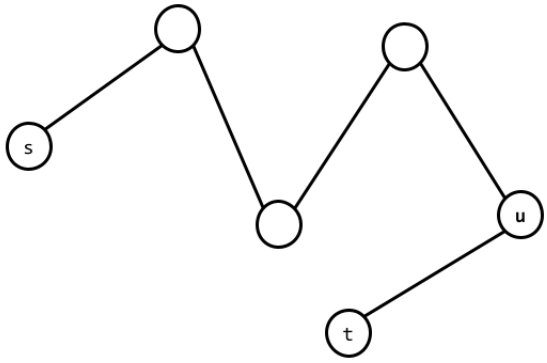
where $\text{Best}_{st}(G)$ is the best value of a path starting at vertex s and ending at vertex t that visits each vertex exactly once.

1.2.4 Recursion

Suppose we have this path from s to t.



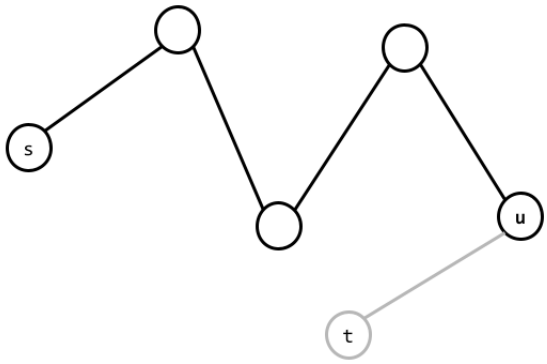
We need to find some recursion that relates the answer to this problem to some kind of subproblem. The standard way to do this is to ask ourselves: what happens if we undo the last step? In our case, our last step is some edge from u to t. That is:



The weight is, then, given by

$$\ell(u, t) + \text{Length of Rest of Path}$$

We want the best path from s to u that uses every vertex *except for* t.



Let's call this subproblem $\text{Best}_{su,-t}$; this is the minimum weight of the path from s to u that visits every vertex except for t exactly once. Thus, the formula is given by

$$\text{Best}_{st}(G) = \min_u \left(\text{Best}_{su,-t}(G) + \ell(u,t)\right)$$

For each t, we figure out what the best path is, add the length of the edge from u to t, and then we take the minimum over all the possible vertices u that have an edge to t. This will give us the best possible path length from s to t.

1.2.5 Recursion II

Now that we have a recursion that gives us the final answers in terms of subproblems, but we need a recursion that solves the subproblems. In other words, how do we solve for $\text{Best}_{su,-t}(G)$?

Again, we look at the edge from v to u. We need the best path that connects s to v that uses all vertices except for t and u. However, this makes the subproblems more complicated; all we're doing is avoiding more vertices.

Instead, we define $\text{Best}_{st,L}$ to be the best path length from s to t that uses exactly the vertices in L. That is, every vertex in L shows up in the path exactly once. The rest of the vertices not in L do not show up in the path.

The last edge is some $(v,t) \in E$ for some $v \in L$. Then, the cost of the path is given by

$$\text{Best}_{sv,L\setminus\{t\}} + \ell(v,t)$$

So, the full recursion is given by

$$\text{Best}_{st,L}(G) = \min_v \left(\text{Best}_{sv,L\setminus\{t\}}(G) + \ell(v,t)\right)$$

1.2.6 Runtime Analysis

Note that L can be any subset of the vertices; thus, there are 2^n possibilities. Additionally, s and t can both be any vertices; thus, there are n^2 possibilities. Thus, there are $n^2 2^n$ subproblems.

We also need to check this for every v (every vertex). This is because, for each subproblem, we need to find the minimum out of all vertices v. Thus, each subproblem takes $\mathcal{O}(n)$ time.

The final runtime is then given by $\mathcal{O}(n^3 2^n)$.