1 Dynamic programming

Dynamic programming is a general strategy for solving problems, much like divide and conquer is a general strategy. And, like divide and conquer, dynamic programming[1] builds up a final solution by combining elements of intermediate solutions.

Recall that divide and conquer (i) partitions a problem into subproblems (with the same properties as the original problem), (ii) recursively solves them, and then (iii) combines their solutions to form a solution of the original problem.

In contrast, a dynamic programming solution (i) defines the value of an optimal solution in terms of overlapping subproblems (with the same properties as the original problem; this is the *optimal substructure* property again), (ii) computes the optimal value by recursively solving its subproblems while (iii) storing results of solved subproblems for later use, and finally (iv) reconstructing the optimal solution from the stored information.

Not every problem can be solved using dynamic programming. For instance, sorting a list of numbers cannot be expressed in terms of overlapping subproblems (i.e., sorting numbers does not have optimal substructure), since every subarray could contain different values in different orders. But, many problems do have overlapping subproblems, and in a dynamic programming approach, once we have solved a subproblem once, we can store or *memoize* its result for future use. In this way, as the algorithm progresses, it can simply look up the answer to a previously solved subproblem rather than resolving it from scratch, and this can lead to dramatic speedups in running time over "brute force" approaches that repeatedly resolve the same subproblems.

1.1 Counting paths in a DAG

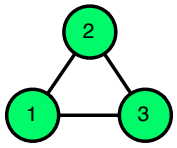
As a first illustration of the dynamic programming approach, we will consider counting the number of paths in a directed acyclic graph (also called a "DAG"). We'll spend much more time later in the class on graph algorithms, and so we'll defer most of the terminology and concepts until later. Here's what we need to understand how to use dynamic programming to count the number of paths between some pair of nodes i and j in a DAG.

1.1.1 Directed acyclic graphs and paths

A *graph* G is defined as a pair of sets $G = (V, E)$, where V is a set of *vertices* or *nodes* and $E : V \times V$ is a set of pairwise *edges* or *arcs*. Often, we say that the number of nodes is $n = |V|$ and the number

[1] The name "dynamic programming" comes from a time when "programming" meant tabulation rather than writing computer code. A more modern and interpretable name would be something like "dynamic tabulation," but it's hard to change a name this late in the game.

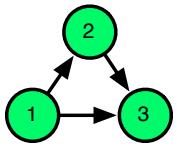
of edges is $m = |E|$.



An undirected graph representing a triangle: $V = \{1, 2, 3\}$ and $E = \{(1, 2), (1, 3), (2, 3)\}$

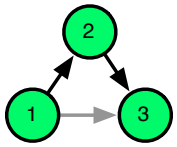
In an *undirected* graph, edges are *undirected*, meaning that the edge (i, j) and the edge (j, i) both represent the same connection between nodes i and j.

In a *directed* graph, the edge set E may contain both (i, j) and (j, i), which are sometimes denoted $(i \to j)$ and $(j \to i)$ to illustrate their directionality.



A DAG representing a triangle: $V = \{1, 2, 3\}$ and $E = \{(1 \to 2), (1 \to 3), (2 \to 3)\}$

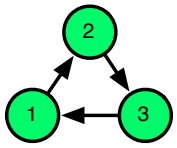
A *path* is a sequence of edges $\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_\ell]$ such that $\forall_k \sigma_k \in E$ and each consecutive pair of edges has the form $\sigma_k, \sigma_{k+1} = (i, j), (a, b)$ where $j = a$. That is, the endpoint of σ_a is the origin of σ_{a+1}.



A path from node 1 to node 3 on the DAG triangle: $\sigma = [(1 \to 2), (2 \to 3)]$

How many paths are there from node 1 to node 3? There are two: $[(1 \to 2), (2 \to 3)]$ and $[(1, 3)]$. Notably, to be a well-defined or non-trivial path, we require that $|\sigma| > 0$, i.e., that the path includes at least one edge.

A *cycle* is a path σ such that there exists some $\sigma_x, \sigma_y = (i, j), (a, b)$ where $i = b$. That is, the origin of σ_a is the endpoint of σ_b.



A cycle from node 1 to node 1 on a triangle: $\sigma = [(1 \to 2), (2 \to 3), (3 \to 1)]$

A *directed acyclic graph* (DAG) is a direct graph with no cycles.

1.1.2 Counting paths

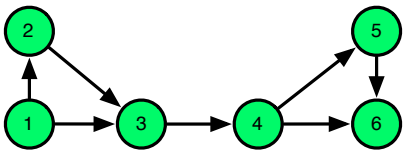
Given a DAG $G = (V, E)$, and a pair of nodes i, j, how many paths are there from i to j?

We can solve this problem using dynamic programming. Suppose we knew the number of paths from i to j was X, and let the set $s(j)$ denote the set of nodes x such that $(x \to j) \in E$, i.e., x is a neighbor of j. Each of the X paths to j must pass through some particular neighbor of j, i.e., each path pass though a node $x \in s(j)$. Thus, the total number of paths X must equal the number of paths from i to $x_1 \in s(j)$ plus the number of paths from i to $x_2 \in s(j)$, etc. That is, X is the sum of the number of paths that start at i and terminate at some node x that neighbors j.

Mathematically, we can say that $X_{i,j}$ counts the number of paths from i to j, and, if we let ℓ index the nodes $s(j)$ that point to j, we can define $X_{i,j}$ recursively:

$$X_{i,j} = \sum_{\ell \in s(j)} X_{i,\ell} \ . \tag{1}$$

Now, we can recursively compute the number of paths $X_{i,j}$, and store the intermediate values for later use in a table to make things faster. Consider the following DAG as a concrete example, where we want to count the number of paths from node 1 to node 6:



Count the paths from node 1 to node 6

We can keep track of the optimal solutions to the subproblems using a simple 1-dimensional table (a.k.a., an array) of the number of paths $X_{1,i}$, which we will use to build up the optimal value of $X_{1,6}$. Using Eq. (1) to write down the solution to $X_{1,i}$ in terms of other elements in the table, and then building up, we obtain

node i	6	5	4	3	2	1
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	$X_{1,3}$	$X_{1,2} + 1$	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	$X_{1,3}$	2	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	$X_{1,4}$	2	2	1	0
$X_{1,i}$	$X_{1,5} + X_{1,4}$	2	2	2	1	0
$X_{1,i}$	4	2	2	2	1	0

where the base case occurs for all nodes that are directly connected to node 1. Hence, the number of paths from 1 to 6 in this simple example is 4.

1.2 The 0-1 Knapsack problem

Another problem that is amenable to the dynamic programming approach is the 0-1 Knapsack problem. In this problem, you are faced with a set of n *indivisible* items S, where each item $i \in S$ has both a *value* v_i and a weight w_i. Your task is to select a subset of items $T \subseteq S$ such that the total value of the items $\sum_{i \in T} v_i$ is maximized and the total weight of the items does not exceed a threshold W, i.e., $\sum_{i \in T} w_i \leq W$.

A *brute force* approach to solving this problem would consider all 2^n possible choices, in which for each of the n items, we either try to include it or exclude it, and then evaluate whether this particular set of choices satisfies our weight limit W and maximizes the total value. Obviously, this approach is infeasible for any reasonable value of n, and so we must try a more efficient approach.

To be continued...

2 On your own

1. Read Chapter 15