Dynamic Programming: From DAGs to Edit Distance

CSC Lecture

Dynamic Programming

Overview

- Goal: solve optimization problems where a large problem can be broken into smaller problems of the same general form. The smaller problems should be meaningful on their own, not just arbitrary fragments.

Parts of a DP Problem

- Optimal substructure: an optimal solution to the whole problem can be assembled from optimal solutions to smaller subproblems. This is what lets a recurrence describe the global optimum correctly.
- Overlapping subproblems: the same smaller problem appears again and again inside different larger problems. This is what makes storing previous answers worthwhile.

Value Functions

- Core idea: compute a **value function** over subproblems rather than repeatedly recomputing full solutions from scratch. The value function records exactly the information needed for future decisions.

How to Solve a DP Problem

- Strategy:
  1. Define the subproblems precisely.
  2. Identify which subproblems depend on which others.
  3. Order the subproblems so dependencies are solved first.
  4. Evaluate each subproblem once and reuse the result everywhere.

DP as a Problem DAG

Two Distinct Objects

- **Input structure**: the object given by the problem statement, such as a graph, a string, or a sequence of items. This is the data we start from.
- **Problem DAG**: the graph whose nodes are subproblems and whose edges represent dependency. This is the structure induced by the recurrence, not necessarily the structure handed to us in the input.

Input and DAG Are NOT the Same

- Important distinction: sometimes the input structure and the problem DAG are the same object, but often they are not. Keeping these two views separate prevents confusion later.

DP Recipe

General Pattern I

- Define a value function: assign to each subproblem the optimal value we care about, such as minimum cost, maximum profit, or number of ways. The value function is the mathematical heart of the DP.
- Build the dependency graph: draw an edge from subproblem *x* to subproblem *y* when solving *x* requires the value of *y*. If this dependency graph is acyclic, then a one-pass evaluation order exists.

General Pattern II

- Solve by: computing a topological ordering of the subproblems and evaluating each node exactly once in that order. This is the bottom-up version of the recurrence.
- Main takeaway: DP is not a bag of tricks. It is systematic evaluation on a dependency DAG.

Example 1: Shortest Path in a DAG

Problem

- Input: a directed acyclic graph $G = (V, E)$, a weight on each edge, and a designated target node t.
- Goal: compute the shortest-path distance from every node v to the target t. This gives us one value for every possible starting point.
- Why this is a good first example: the graph is already visible, the choices are real, and the Bellman recurrence is easy to read directly from the picture.

Direction Matters

Hard vs Easy

- Hard viewpoint: if we begin from arbitrary source nodes and try to reason forward, it is less clear what information should already be known. The dependencies feel backwards.
- Easy viewpoint: if we define distance **to the target**, then the base case is immediate: $d(t) = 0$.
- Propagate backward: every other node can be solved from its outgoing neighbors because any path from v to t must first choose one outgoing edge. This lines up perfectly with dynamic programming.

Value Function

Definition

- *d(v)* = the shortest distance from node *v* to the target *t*. This is the cost-to-go from state *v*.

Recurrence

- *d(v) = min over edges (v -> u) of [w(v,u) + d(u)]* because any path from *v* to *t* must first take some outgoing edge *(v,u)*, pay its weight, and then follow an optimal path from *u* onward.
- Base case: *d(t) = 0*. The target is already at the destination, so no further cost is needed.

Subproblem Structure

Key Observation

- One subproblem per node: each node *v* asks one question, namely the optimal remaining distance from *v* to *t*.
- Dependencies follow graph edges: computing *d(v)* requires the values of successor nodes *u* reachable by edges *v -> u*.

Critical Property

- Problem DAG = Input DAG: in this example, the dependency structure is literally the same DAG we were given. That makes the DP interpretation unusually transparent.

Algorithm

Steps

1. Topologically sort the nodes of the DAG. This gives an order in which every edge points forward.
2. Process the nodes in reverse topological order. That ensures every successor value needed by *d(v)* has already been computed.
3. Compute each node once using the recurrence. Store both the optimal value and, if desired, the best outgoing edge for path reconstruction.

DP Properties

Why This Works

- Optimal substructure: once the first edge *(v,u)* is chosen, the rest of the path must itself be an optimal path from *u* to *t*. Otherwise we could improve the overall solution.
- Overlapping subproblems: several different nodes may lead into the same successor region of the graph, so the same suffix path information is reused many times.

Complexity

- Time: $O(|V| + |E|)$ because each node and edge is examined only a constant number of times after the topological sort.
- Space: $O(|V|)$ for the value table and optional predecessor or choice information.

Takeaway

Cleanest DP Example

- The subproblem DAG is explicit: we can literally point to the dependency graph on the board.
- There is almost no abstraction gap: the recurrence, the graph, and the algorithm all describe the same object in slightly different language.
- This makes the example ideal for introducing the general DP viewpoint before moving to more implicit state spaces.

Transition

What If No DAG is Given?

- Sometimes the input does not come with an obvious dependency graph. We still use the same DP ideas, but we must define the subproblems ourselves.
- In those cases: the recurrence induces the problem DAG even if we never explicitly draw every node and edge in code.
- That is the next conceptual step: move from an explicit problem DAG to an implicit one.

Example 2: Edit Distance

Problem

- We are given two strings *A* and *B*. The task is to transform *A* into *B* using the fewest edits possible.
- Allowed operations: insert a character, delete a character, or substitute one character for another.
- Goal: minimize the total number of operations. This makes the problem a natural candidate for a min-type value function.