LECTURE 20
DYNAMIC PROGRAMMING II:
LONGEST PATH IN A DAG &
SHORTEST PATH

Admin

→ PS 6 due in 1 week (only one grace day)
→ Cookie Challenge 30% submissions of completed code 48 hours in advance (Tuesday)

Review - Dynamic Programming concepts
  - Typically an optimization problem, can be solved recursively
  - Recursive calls generate repeated (same) subproblems
  - Subproblem dependence graph (DAG ⇒ no cycles)
    \[ V = \text{set of subproblems} \]
    \[ E = \{ x \rightarrow y : \text{solving} \ x \ \text{requires solution to} \ y \} \]
  - Two approaches
    (i) "top-down" (recursive) using memoization to reuse subproblem solutions [DFS on graph]
    (ii) "bottom-up" sorts \( V \) in reverse topological order, solve subproblems in that order, saving solutions in array for reuse.
  - If solving a subproblem is linear in number of recursive calls it makes (\# of outgoing edges in graph)
    then running time is typically \( O(V + E) \)
Today

- Longest path in a DAG
- Shortest path

**Longest Path in a DAG**

Note: The general longest-path-in-a-graph problem lacks optimal substructure and so can't be solved with DP. (Not necessarily a DAG)

Note: Longest path should be simple, with no repeated vertices.

Longest A→D path is A→B→C→D. But longest A→B path is not A→B (by substructure); rather, it is A→C→B.

But a DAG does have optimal substructure for longest paths.

Imagine we want the longest (greatest weight) s→t path.

d[T[v]] = weight of longest s→v path
d[T[s]] = 0 in DAG

The very last step of longest path is max(d[T[v]] + 4, d[T[D]] + 1) for the two edges incident on t.
This leads to subproblems corresponding to \( d[V] \) for \( V \in V \).

Recall that DFS can be used to sort a DAG into a topologically sorted order.

![Directed Acyclic Graph (DAG)](image)

Progressing left-to-right in sorted order solves each subproblem before it is needed for a larger subproblem.

Note that this is different from the subproblem dependency graph, of the previous lecture; in fact, the arrows would need to be reversed to create such a dependency graph.

Given: Weighted DAG \( G = (V, E, w) \) & starting vertex \( s \in V \)
Find: largest weight path in \( G \), starting from \( s \)

Topologically sort \( G \) for each vertex \( v \in V \) in sorted order before \( s \):

\[
d[s] \leftarrow -\infty
\]

\[
d[s] \leftarrow 0
\]

for each vertex \( v \in V \) in sorted order after \( s \):

\[
d[v] \leftarrow \max_{(u,v) \in E} \{ d[u] + w(u,v) \}
\]

return \( \max_{v \in V} d[v] \) returns weight of longest (acyclic) path in DAG

\[\Theta(V + E)\]

worst case

\[\Theta(V^2)\]

examines edges

\[\Theta(E)\]
Shortest Paths

Single source shortest paths

Imagine that we didn’t know about Dijkstra, DA6-SP, and Bellman-Ford, but we know about dynamic programming.

Given: Directed graph with weights \( G(V, E, w) \) and source vertex \( s \in V \)

Find: Set of shortest path lengths \( \delta(s, v) \) for \( v \in V \)

To get started, imagine only a particular destination \( v \in V \). Then generalize.

Step 0: What are we enumerating over and optimizing on?
- Enumerate over all simple \( s \rightarrow v \) paths
- Optimization selects least weight among paths

Step 1: Break that enumeration into set of nested subproblems linked through recursion.
- One of the edges incident into \( v \) must be on the shortest \( s \rightarrow v \) path. Take the "best of these" edges incident on \( v \). Enumerates

\[
\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \mid (u, v) \in E \}
\]
Step 2: Examine the relationships between the subproblems expressed by the recursion. Is it a valid recursion (Is the subproblem dependency graph a DAG)?

(example (i))

\[
\begin{array}{c}
S \\
B \\
D \\
A \\
C \\
Y \\
\end{array}
\]

(example (ii))

\[
\begin{array}{c}
S \\
B \\
D \\
A \\
C \\
Y \\
\end{array}
\]

Not a DAG-
The subproblem graph is not a DAG, and so the recursion can't run.
\[
\delta(s, c) \rightarrow \delta(s, a) \rightarrow \delta(s, b) \leftarrow \delta(s, d) \leftarrow
\]

The code would have an infinite loop.

\[\checkmark \quad \text{Will work for DAGs}\]

\[\times \quad \text{(for now on cyclic graphs)}\]

Step 3: What to memoize or store for reuse?

- Here (on left), reuse \(\delta(s, u)\) \(u \in V\)
Step 4: Write algorithm.

```python
memo = {}
def d(v):
    if v in memo: return memo[v]
    if v == s: return 0
    ans = float("inf")
    for "all edges (u,v) ∈ E incident on v" in E:
        ans = min(ans, d(u) + w(u,v))
    memo[v] = ans
    return ans
```

Running Time:
In worst case runs through every vertex and edge once $\Theta(V+E)$

Effectively this algorithm runs like DAG-SP, without being as explicit. Bottom-up version would be more explicit.
Can we somehow "repair" the cyclic case?

The issue was that subproblems depended upon each other, and so "neither" could be solved.

One solution to this issue is to specify a "round" to each subproblem, effectively multiplying the number of subproblems.

Here we let

\[ \delta_k(s, v) = \text{shortest s} \to v \text{ path using } \leq k \text{ edges} \]

which replaces our recursion with

\[
\delta_k(s, v) = \min \left\{ \delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E \right\}
\]

\[ \delta_0(s, v) = \infty \quad \forall v \neq s \in V \quad \text{Base Case} \]

\[ \delta_k(s, s) = 0 \quad \forall k \geq 0 \]

Our goal: \[ \delta(s, v) = \delta_{|V|-1}(s, v) \]

\{ If no neg-weight cycles (but can be cycles) \}
Step 2': How does this repair our subproblem graph?

Example (ii)

Before:

Now:

\[ \delta_k(s,v) \text{ for } 0 \leq k \leq |V|-1 \text{ and } v \in V \]

Step 3': What to memoize, store, reuse?

- \[ \delta_k(s,v) \text{ for } 0 \leq k \leq |V|-1 \text{ and } v \in V \]
Step 4': Write Algorithm → See R20

Running Time Analysis

(a) # of subproblems (# of unique recursive calls):
   \[ |V| \cdot (|V|) \]
   \[ \Rightarrow \Theta(|V|^2) \]
   \[ S_k(s, V) \quad \uparrow \]
   \[ 0 \leq k \leq |V| - 1 \]
   \[ |V| \text{ values of } k \]

(b) Time per subproblem: proportional to
   number of edges incident on current vertex
   \[ \Rightarrow O(|V|) \]

(c) (# of subproblems) \cdot (time/subprob) = \Theta(|V|^3)

This is too pessimistic.

For each of \(|V|\) values of \(k\), we have to carry out an operation "for each edge incident on each vertex". The term in quotes is \(|E|\), the number of edges.

\[ \Theta(|V| \cdot |E|) = \Theta(V \cdot E) \quad \text{\color{red}{Bellman-Ford}} \]