Lecture 17: Dynamic Programming II

Dynamic Programming Review

- Recursion where subproblems dependencies **overlap**
- "Recurse but reuse" (Top down: record and lookup subproblem solutions)
- "Careful brute force" (Bottom up: do each subproblem in order)

Dynamic Programming Steps (SR. BST)

1. Define **Subproblems** subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** Subproblems $x(i) = f(x(j), \ldots)$ for one or more $j < i$
   - State topological order to argue relations are acyclic and form a DAG
3. Identify **Base** Cases
   - State solutions for all reachable independent subproblems
4. Compute **Solution** from Subproblems
   - Compute subproblems via top-down memoized recursion or bottom-up
   - State how to compute solution from subproblems (possibly via parent pointers)
5. Analyze Running **Time**
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = W$ for all $x \in X$, then $|X| \times W$

Single Source Shortest Paths Revisited (Attempt 2)

1. **Subproblems**
   - Increase subproblems to add information to make acyclic!
   - $\boxed{x(v, k)\text{, minimum weight of any } k\text{-edge path from } s \text{ to } v \in V}$
2. **Relate**
   - $x(v, k) = \min\{x(u, k-1) + w(u, v) \mid (u, v) \in E\}$
   - Subproblems only depend on subproblems with strictly smaller k, so no cycles!
3. **Base**
   - $x(s, 0) = 0$ and $x(v, 0) = \infty$ for $v \neq s$ (no edges)
   - (draw subproblem graph)
4. **Solution**
   - Compute subproblems via top-down memoization or bottom up (draw graph)
   - Can keep track of parent pointers to subproblem that minimized recurrence
   - Unlike normal Bellman-Ford, parent pointers always form a path back to s!
   - If has finite shortest path, than $\delta(s, v) = \min\{x(v, k) \mid 0 \leq k < |V|\}$
   - Otherwise some $x(v, |V|) < \min\{x(v, k) \mid 0 \leq k < |V|\}$, so path contains a cycle
   - Claim: All cycles along a parent path have negative weight
   - Proof: If not, removing cycle is path with fewer edges with no greater weight
5. **Time**
   - # subproblems: $|V| \times (|V| + 1)$
   - Work for subproblem $x(v, k)$: $O(\deg_{\text{in}}(v))$

   $$\sum_{k=0}^{|V|} \sum_{v \in V} O(\deg_{\text{in}}(v)) = \sum_{k=0}^{|V|} O(|E|) = O(|V||E|)$$

   - Computing $\delta(s, v)$ takes $O(|V|)$ time per vertex, so $O(|V|^2)$ time in total
   - Running time $O(|V|(|V| + |E|))$. Can we make $O(|V||E|)$?
   - Only search on vertices reachable from s, then $|V| \leq |E| + 1 = O(|E|)$
   - Such vertices can be found via BFS or DFS in $O(|V| + |E|)$ time

This is just **Bellman-Ford**!

```
def bellman_ford_dp(Adj, w, s):
    # Return shortest paths and parent pointers, or a cycle with negative weight
    V = len(Adj)
    incoming = [[] for _ in range(V)]             # compute incoming adjacencies
    for v in range(V):
        for u in Adj[v]:
            incoming[u].append(v)
    x = [[float('inf')] * (V + 1) for _ in range(V)]
    parent = [[None] * (V + 1) for _ in range(V)]
    x[s][0] = 0                                   # base case
    for k in range(1, V + 1):                     # dynamic program
        for v in range(V):
            for u in incoming[v]:
                x_ = x[u][k - 1] + w(u, v)        # recurrence
                if x_ < x[v][k]:                  # minimization
                    x[v][k] = x_
                    parent[v][k] = u
    d, p = [float('inf')] * V, [None] * V         # shortest paths using < |V| edges
    for v in range(V):
        for k in range(V):
            if x[v][k] < d[v]:
                d[v] = x[v][k]
                p[v] = parent[v][k]
    for v in range(V):
        if x[v][V] < d[v]:                        # there is a negative cycle
            path = []
            u, k = v, V
            while u not in path:                  # construct path
                path.append(u)
                u = parent[u][k]
                k = k - 1
            i = 0
            while path[i] != u:                   # find start of cycle
                i = i + 1
            path = path[i:]                       # cut to cycle
            path.reverse()
            return path                           # return cycle
    return d, p                                   # return shortest paths
```

Rod Cutting

- Given a rod of length n and the value $v(i)$ of any rod piece of integral length i for $1 \leq i \leq n$, cut the rod to maximize the value of cut rod pieces.
- Example: $n = 7$, $v = (1, 5, 8, 9, 10, 16, 17)$ (one indexed)
- Solution: $v(2) + v(2) + v(3) = 5 + 5 + 8 = 18$
- Maximization problem on value of partition

1. **Subproblems**
   - $\boxed{x(i)\text{: maximum value obtainable by cutting rod of length } i}$
2. **Relate**
   - Left-most cut has some length **(Guess!)**
   - $x(i) = \max\{v(j) + x(i - j) \mid j \in \{1, \ldots, i\}\}$
   - (draw dependency graph)
   - Subproblems $x(i)$ only depend on strictly smaller i, so acyclic
3. **Base**
   - $x(0) = 0$ (length zero rod has no value!)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum value obtainable by cutting rod of length n is $x(n)$
   - Store choices to reconstruct cuts
   - If current rod length i and optimal choice is j, remainder is $i - j$
5. **Time**
   - # subproblems: n
   - work per subproblem: $O(i)$
   - $O(n^2)$ running time

```python
# recursive
x = {}
def cut_rod(w, v):
    if w < 1:   return 0                          # base case
    if w not in x:                                # check memo
        for piece in range(1, w + 1):             # try piece
            x_ = v[piece] + cut_rod(w - piece, v) # recurrence
            if (w not in x) or (x[w] < x_):       # update memo
                x[w] = x_
    return x[w]
```

```python
# iterative
def cut_rod(n, v):
    x = [0] * (n + 1)                             # base case
    for w in range(n + 1):                        # topological order
        for piece in range(1, w + 1):             # try piece
            x_ = v[piece] + x[w - piece]          # recurrence
            if x[w] < x_:                         # update memo
                x[w] = x_
    return x[n]
```

```python
# iterative with parent pointers
def cut_rod_pieces(n, v):
    x = [0] * (n + 1)                             # base case
    parent = [None] * (n + 1)                     # parent pointers
    for w in range(1, n + 1):                     # topological order
        for piece in range(1, w + 1):             # try piece
            x_ = v[piece] + x[w - piece]          # recurrence
            if x[w] < x_:                         # update memo
                x[w] = x_
                parent[w] = w - piece             # update parent
    w, pieces = n, []
    while parent[w] is not None:                  # walk back through parents
        piece = w - parent[w]
        pieces.append(piece)
        w = parent[w]
    return pieces
```