Dynamic Programming

Dynamic Programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

Optimal Sub-structure

DP takes the advantage of the optimal sub-structure of a problem. A problem has an optimal sub-structure if the optimum answer to the problem contains optimum answer to smaller sub-problems. As an example, the shortest path from \( u \) to \( v \) is composed of some edge \((w, v)\) and the shortest path from \( u \) to \( w \) (a smaller problem to solve).

Review: Memoized Recursive and Bottom-up DP

Dynamic Programming can be framed as a memoized recursive algorithm or as a bottom-up iterative algorithm.

Memoized Recursive

To turn a normal recursive algorithm into a memoized recursive algorithm, we add a memo dictionary to store outputs for each input combination (subproblem) we’ve solved so far. To solve a subproblem that’s not in the memo, we just run our original recursive algorithm and then store the result in memo.

Our Fibonacci example from lecture:

```python
memo = {}
def fib(n):
    # Check if we’ve already solve the subproblem
    if n in memo: return memo[n]
    # Normal recursive algorithm
    if n <= 2: f = 1
    else: f = fib(n-1) + fib(n-2)
    # Remember our solution for later
    memo[n] = f
    return f
```

Analysis   Rather than attempting the complex analysis how the runtimes add up recursively, we instead separate out the problem into: (1) how many subproblems do we need to solve and (2) how long does each subproblem take to solve assuming constant lookup of its subproblems. This
makes sense, since we are just reordering the pieces of our recursive runtime to make them easier to analyze. Our overall runtime thus becomes:

\[ T = (\# \text{ subproblems}) \times (\text{time/subproblem}) \]

In our Fibonacci example, we thus get \(O(n)\) subproblems and \(O(1)\) to solve each subproblem when we ignore the recursive costs, giving us \(O(n)\) total.

**Bottom-Up**

Often times it makes sense to avoid the recursive overhead involved in a recursive algorithm and re-order how we solve the subproblems. Rather than letting the recursive algorithm solve the subproblems on an on-demand basis, we can choose to build up our subproblems from the base cases up, in such a way that every time we solve a subproblem any subproblems it refers to are already solved. Finding such an ordering is equivalent to topologically sorting the DAG defined by the dependencies of each subproblem.

Our Fibonacci example from lecture becomes:

```python
def fib_bottom_up(n):
    fib = {}
    for k in range(n):
        # Our original recursive solution. Notice that our "recursive" calls are now lookups
        if k <= 2: f = 1
        else: f = fib[k-1] + fib[k-2]
        fib[k] = f
    return fib[n]
```

**Analysis**  Our analysis is exactly the same as the memoized recursion, though in this case it’s much clearer from where the runtime arises.

**Trade-offs**

There are advantages and disadvantages to both types of DP.

**Advantages to Recursive Memoized:**

- Often much easier to understand
- Don’t have to determine an ordering, which might be hard to do manually in some cases

**Advantages to Bottom-Up:**

- Doesn’t have the overhead of recursion (we also avoid the issue of exceeding the maximum recursion depth, due to extremely deep recursions)
- Often easier to analyze runtime
Robotic Coin Collection

We have a grid of squares and coins on selected squares. A robot moves from square \((0, 0)\) at the bottom left to \((n, m)\) on the top right only moving one step up or to the right each time. If it sees a coin, it picks it up. (The number of columns is \(n + 1\) and the number of rows is \(m + 1\).)

**Goal**: Maximize number of coins picked by choosing an appropriate path.

![Figure 1: Robotic Coin Collection. \(n = 9\) and \(m = 5\). A maximum of 11 coins can be picked up in the optimal path shown. Picture from http://demonstrations.wolfram.com/PickingUpCoinsInAGrid/](http://demonstrations.wolfram.com/PickingUpCoinsInAGrid/)

[Sidenote: There are an exponential number of possible paths!]

**Algorithm using DP**

c\(_{ij}\) = 1 if there is a coin at \((i, j)\) else 0.

\(S(i, j)\): largest number of coins robot can pick up until and including \((i, j)\).

**Goal**: Maximize \(S(n, m)\).

The insight is that the robot can reach \((i, j)\) in only 2 different ways: From the left \((i - 1, j)\) square or from the bottom \((i, j - 1)\) square.

Therefore, we can write:

\[
S(i, j) = \max(S(i - 1, j), S(i, j - 1)) + c_{ij}, \ i \geq 1, \ j \geq 1 \\
S(0, 0) = 0 \\
S(i, 0) = S(i - 1, 0) + c_{i0}, \ i \geq 1 \\
S(0, j) = S(0, j - 1) + c_{0j}, \ j \geq 1
\]

**Analysis**  The total number of sub-problems here is \(O(mn)\). Each sub-problem takes \(O(1)\) time to compute, so the total runtime of this algorithm is \(O(mn)\).
Shortest Path with Dynamic Programming

The shortest path problem has an optimal sub-structure. Suppose \( s \leadsto u \leadsto v \) is a shortest path from \( s \) to \( v \). This implies that \( s \leadsto u \) is a shortest path from \( s \) to \( u \), and this can be proven by contradiction. If there is a shorter path between \( s \) and \( u \), we can replace \( s \leadsto u \) with the shorter path in \( s \leadsto u \leadsto v \), and this would yield a better path between \( s \) and \( v \). But we assumed that \( s \leadsto u \leadsto v \) is a shortest path between \( s \) and \( v \), so we have a contradiction.

Based on this optimal sub-structure, we can write down the recursive formulation of the single source shortest path problem as the following:

\[
\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \mid (u, v) \in E \}
\]

DAG

For a DAG, we can directly use memoized DP algorithm to solve this problem. The following is the Python code:

```python
class ShortestPathResult(object):
    def __init__(self):
        self.d = {}
        self.parent = {}

def shortest_path(graph, s):
    '''Single source shortest paths using DP on a DAG.

    Args:
    graph: weighted DAG.
    s: source
    
    '''
    result = ShortestPathResult()
    result.d[s] = 0
    result.parent[s] = None
    for v in graph.itervertices():
        sp_dp(graph, v, result)
    return result

def sp_dp(graph, v, result):
    '''Recursion on finding the shortest path to v.

    Args:
    graph: weighted DAG.
    v: a vertex in graph.
    result: for memoization and keeping track of the result.
    
    '''
    if v in result.d:
        return result.d[v]
    result.d[v] = float('inf')
    result.parent[v] = None
```
for u in graph.inverse_neighbors(v): # Theta(indegree(v))
    new_distance = sp_dp(graph, u, result) + graph.weight(u, v)
    if new_distance < result.d[v]:
        result.d[v] = new_distance
        result.parent[v] = u
return result.d[v]

The total running time of $DP = \text{number of subproblems} \times \text{time per subproblem (ignoring recursion)}$. In this case, the subproblem is represented by $\delta(s, v)$ which is parameterized by $v$ because $s$ is fixed. The number of possible values for $v$ is $|V|$, so there are $|V|$ subproblems. Each subproblem takes $\Theta(\text{indegree}(v) + 1)$ time. So the total time is $\Theta(\sum_{v \in V} \text{indegree}(v) + 1) = \Theta(|E| + |V|)$ by Handshaking Lemma.

For the bottom-up version, we need to topologically sort the vertices to find the right order to compute $\delta(s, v)$.

```python
def shortest_path_bottomup(graph, s):
    '''Bottom-up DP for finding single source shortest paths on a DAG.

    Args:
        graph: weighted DAG.
        s: source
    '''
    order = topological_sort(graph)
    result = ShortestPathResult()
    for v in graph.itervertices():
        result.d[v] = float('inf')
        result.parent[v] = None
    result.d[s] = 0
    for v in order:
        for w in graph.neighbors(v):
            new_distance = result.d[v] + graph.weight(v, w)
            if result.d[w] > new_distance:
                result.d[w] = new_distance
                result.parent[w] = v
    return result
```

Graph with Cycles

In order for DP to work, the subproblem dependency should be acyclic, otherwise there will be infinite loops. We can create more subproblems to remove the cyclic dependencies. Let $\delta_k(s, v)$ be the shortest $s \rightarrow v$ path using $\leq k$ edges. Then we can redefine the recurrence as the following:

$$\delta_k(s, v) = \min\{\delta_{k-1}(s, u) + w(u, v) | (u, v) \in E, \delta_{k-1}(s, v)\}$$

The base cases are:

$$\delta_0(s, v) = \infty \text{ for } v \neq s$$
$$\delta_k(s, s) = 0 \text{ for any } k$$
If there are no negative cycles, $\delta_{|V|-1}(s, v) = \delta(s, v)$.

We can visualize this as a graph transformation as well. Let $G = (V, E)$ be a directed graph with cycles. For every $v \in V$, make $|V|$ copies of $v$ as $v_0, v_1, \ldots, v_{|V|-1}$ in the new graph $G'$. For every edge $(u, v) \in E$, create an edge $(u_{k-1}, v_k)$ for $k = 1, \ldots, |V| - 1$ in $G'$.

![Figure 2: Transforming a cyclic graph into an acyclic graph.](image)

```python
def shortest_path_cycle(graph, s):
    '''Single source shortest paths using DP on a graph with cycles but no negative cycles.

    Args:
    graph: weighted graph with no negative cycles.
    s: source
    '''
    result = ShortestPathResult()
    num_vertices = graph.num_vertices()
    for i in range(num_vertices):
        result.d[(i, s)] = 0
        result.parent[(i, s)] = None
    for v in graph.itervertices():
        if v is not s:
            result.d[(0, v)] = float('inf')
    for v in graph.itervertices():
        sp_cycle_dp(graph, num_vertices - 1, v, result)
    d = {}
    parent = {}
    for v in graph.itervertices():
        d[v] = result.d[(num_vertices - 1, v)]
        parent[v] = result.parent[(num_vertices - 1, v)]
    result.d, result.parent = d, parent
    return result

def sp_cycle_dp(graph, k, v, result):
    '''Recursion on finding the shortest path to v with no more than k edges on a graph with cycles.

    Args:
    graph: weighted graph.
    k: kth level subproblem, i.e. finding paths with no more than k edges.
    v: a vertex in the graph.
    '''
```
result: for memoization and keeping track of the result.

if (k, v) in result.d:
    return result.d[(k, v)]
result.d[(k, v)] = float("inf")
result.parent[(k, v)] = None
for u in graph.inverse_neighbors(v):
    new_distance = sp_cycle_dp(graph, k - 1, u, result) + graph.weight(u, v)
    if new_distance < result.d[(k, v)]:
        result.d[(k, v)] = new_distance
        result.parent[(k, v)] = u
return result.d[(k, v)]

Crazy 8’s

In the game Crazy 8’s, we are given an input of a sequence of cards $C[0], \ldots, C[n - 1]$, e.g., $7\spadesuit, 7\heartsuit, K\spadesuit, K\clubsuit, 8\diamondsuit$. We want to find the longest subsequence of cards where consecutive cards must have the same value, same suit, or have one of the two cards be an eight. The longest such subsequence in the example is $7\spadesuit, K\spadesuit, K\spadesuit, 8\diamondsuit$.

To solve this, if the cards are stored in array $C$, we will to keep an auxiliary score array $S$ where $S[i]$ represents the length of the longest subsequence ending with card $C[i]$.

We start with $S[0] = 1$ since the longest subsequence ending with the first card is that card itself and has a length of 1. We iteratively calculate the next score $S[i]$ by scanning all previous scores and set $S[i]$ to be $S[k] + 1$ where $S[k]$ represents the length of the longest subsequence that card $C[i]$ can be appended to.

Analysis  For an input of $n$ cards, there are $O(n)$ subproblems: $S[0], \ldots, S[n - 1]$. Solving each subproblem requires iterating over all previous subproblems, for an $O(n)$ time per subproblem. Thus in total, our runtime is $O(n^2)$. 