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 that 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.

Shortest Path with Dynamic Programming

The shortest path problem has an optimal sub-structure. Supose $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path from s to v. This implies that $s \rightsquigarrow u$ 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 $s \rightsquigarrow u$ with the shorter path in $s \rightsquigarrow u \rightsquigarrow v$, and this would yield a better path between s and v. But we assumed that $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path between s and v, so 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:

$$\delta(s, v) = \min\{\delta(s, u) + w(u, v)|(u, v) \in E\}$$

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
```

```
15          result.parent[s] = None
16      for v in graph.itervertices():
17          sp_dp(graph, v, result)
18      return result
19
20  def sp_dp(graph, v, result):
21      '''Recursion on finding the shortest path to v.
22
23      Args:
24          graph: weighted DAG.
25          v: a vertex in graph.
26          result: for memoization and keeping track of the result.
27      '''
28      if v in result.d:
29          return result.d[v]
30      result.d[v] = float('inf')
31      result.parent[v] = None
32      for u in graph.inverse_neighbors(v): # Theta(indegree(v))
33          new_distance = sp_dp(graph, u, result) + graph.weight(u, v)
34          if new_distance < result.d[v]:
35              result.d[v] = new_distance
36              result.parent[v] = u
37      return result.d[v]
```

The total running time of DP = number of subproblems × 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 $\Theta(\texttt{indegree}(v) + 1)$ time. So the total time is $\Theta(\sum_{v \in V} \texttt{indegree}(v) + 1) = \Theta(E + V)$ by Handshaking Lemma.

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

```
1  def shortest_path_bottomup(graph, s):
2      '''Bottom-up DP for finding single source shortest paths on a DAG.
3
4      Args:
5          graph: weighted DAG.
6          s: source
7      '''
8      order = topological_sort(graph)
9      result = ShortestPathResult()
10     for v in graph.itervertices():
11         result.d[v] = float('inf')
12         result.parent[v] = None
13     result.d[s] = 0
14     for v in order:
15         for w in graph.neighbors(v):
16             new_distance = result.d[v] + graph.weight(v, w)
17             if result.d[w] > new_distance:
18                 result.d[w] = new_distance
```

```
19                  result.parent[w] = v
20          return result
```

Graph with Cycles

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

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

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, $\delta(s, v) = \delta_{|V|-1}(s, v)$ because the maximum possible number of edges of a simple path is $|V| - 1$.

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 1: Transforming a cyclic graph into an acylic graph.

In the following Python implementation, we do not transform the graph. We just use the tuple (k, v) as the key in the dictionaries for memoization.

```python
1   def shortest_path_cycle(graph, s):
2       '''Single source shortest paths using DP on a graph with cycles but no
3       negative cycles.
4
5       Args:
6           graph: weighted graph with no negative cycles.
7           s: source
8       '''
9       result = ShortestPathResult()
10      num_vertices = graph.num_vertices()
11      for i in range(num_vertices):
12          result.d[(i, s)] = 0
13          result.parent[(i, s)] = None
14
```

```
15      for v in graph.itervertices():
16          if v is not s:
17              result.d[(0, v)] = float('inf')
18      for v in graph.itervertices():
19          sp_cycle_dp(graph, num_vertices - 1, v, result)
20
21      d = {}
22      parent = {}
23      for v in graph.itervertices():
24          d[v] = result.d[(num_vertices - 1, v)]
25          parent[v] = result.parent[(num_vertices - 1, v)]
26      result.d, result.parent = d, parent
27      return result
28
29  def sp_cycle_dp(graph, k, v, result):
30      '''Recursion on finding the shortest path to v with no more than k edges
31      on a graph with cycles.
32
33      Args:
34          graph: weighted graph.
35          k: kth level subproblem, i.e. finding paths with no more than k edges.
36          v: a vertex in the graph.
37          result: for memoization and keeping track of the result.
38      '''
39      if (k, v) in result.d:
40          return result.d[(k, v)]
41      result.d[(k, v)] = float('inf')
42      result.parent[(k, v)] = None
43      for u in graph.inverse_neighbors(v):
44          new_distance = sp_cycle_dp(graph, k - 1, u, result) + graph.weight(u, v)
45          if new_distance < result.d[(k, v)]:
46              result.d[(k, v)] = new_distance
47              result.parent[(k, v)] = u
48      return result.d[(k, v)]
```

The subproblem is parameterized by two variables k and v. The number of values k can take is $|V|$, and the number of values v can take is $|V|$ as well. Time per subproblem is the same as before: $\Theta(\texttt{indegree}(v)+1)$. To total time is $\Theta\big(V\sum_{v\in V}\texttt{indegree}(v)+1\big) = \Theta(VE)$. Note that this is the same running time as Bellmand-Ford algorithm, and you should observe the similarities between the two algorithms.

Crazy 8's

See the previous year's lecture notes (slides 14 - 20): http://courses.csail.mit.edu/6.006/spring11/lectures/lec18.pdf

In the game Crazy 8's, given an input of a sequence of cards $C[0], \ldots, C[n-1]$, e.g., $7\clubsuit, 7\heartsuit, K\clubsuit, K\spadesuit, 8\heartsuit$, we want to find the longest "trick subsequence" of cards where consecutive cards must have the

same value, same suit, or contains at least one eight. The longeset such subsequence in the example is $7\clubsuit, K\clubsuit, K\spadesuit, 8\heartsuit$.

If the cards are stored in array C, we want 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.

6.006 Introduction to Algorithms
Fall 2011