More Dynamic Programming

Lecture 10
February 22, 2011

Part I

All Pairs Shortest Paths
Shortest Path Problems

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.
- Find shortest paths for all pairs of nodes.

Single-Source Shortest Paths

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.

**Dijkstra’s algorithm** for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

**Bellman-Ford algorithm** for arbitrary edge lengths. Running time: $O(nm)$. 
All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

- Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.
- Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
- Arbitrary edge lengths: $O(n^2 m)$. Can we do better?

Shortest Paths and Recursion

- Can we compute the shortest path distance from $s$ to $t$ recursively?
- What are the smaller sub-problems?

Lemma

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

$\bullet$ $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source \( s \).

\( \text{OPT}(v, k) \): shortest path distance from \( s \) to \( v \) using at most \( k \) edges.

Note: \( \text{dist}(s, v) = \text{OPT}(v, n - 1) \)

Recursion for \( \text{OPT}(v, k) \):

\[
\text{OPT}(v, k) = \min_{u \in V}(\text{OPT}(u, k - 1) + c(u, v)).
\]

Base case: \( \text{OPT}(v, 1) = c(s, v) \) if \((s, v) \in E\) otherwise \( \infty \)

Leads to Bellman-Ford algorithm — see textbook.

\( \text{OPT}(v, k) \) values are also of independent interest: shortest paths with at most \( k \) hops

All-Pairs: recursion on index of intermediate nodes

- Number vertices arbitrarily as \( v_1, v_2, \ldots, v_n \)
- \( \text{dist}(i, j, k) \): shortest path distance between \( v_i \) and \( v_j \) among all paths in which the largest index of an intermediate node is at most \( k \)

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5 
\end{align*}
\]
**Floyd-Warshall Algorithm**

for All-Pairs Shortest Paths

Check if $G$ has a negative cycle using Bellman-Ford in $O(mn)$ time

If there is a negative cycle return

\[
\text{for } i = 1 \text{ to } n \text{ do }
\text{   for } j = 1 \text{ to } n \text{ do }
\text{     } dist(i, j, 0) = c(i, j) \quad (\text{* } c(i, j) = \infty \text{ if } (i, j) \text{ not edge, 0 if } i = j \text{ *})
\]

\[
\text{for } k = 1 \text{ to } n \text{ do }
\text{   for } i = 1 \text{ to } n \text{ do }
\text{     for } j = 1 \text{ to } n \text{ do }
\text{       } dist(i, j, k) = \min(dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1))
\]

**Correctness:** Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

**Running Time:** $\Theta(n^3)$, **Space:** $\Theta(n^3)$.
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

\[
\begin{align*}
\text{for } & \ i = 1 \ \text{to} \ n \ \text{do} \\
& \quad \text{for } \ j = 1 \ \text{to} \ n \ \text{do} \\
& \qquad \text{dist}(i, j, 0) = c(i, j) \quad (* \ c(i, j) = \infty \text{ if } (i, j) \text{ not edge, } 0 \text{ if } i = j \ *)
\end{align*}
\]

\[
\begin{align*}
\text{for } & \ k = 1 \ \text{to} \ n \ \text{do} \\
& \quad \text{for } \ i = 1 \ \text{to} \ n \ \text{do} \\
& \qquad \text{for } \ j = 1 \ \text{to} \ n \ \text{do} \\
& \qquad \quad \text{dist}(i, j, k) = \min(\text{dist}(i, j, k - 1), \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1))
\end{align*}
\]

\[
\begin{align*}
\text{for } & \ i = 1 \ \text{to} \ n \ \text{do} \\
& \quad \text{if } (\text{dist}(i, i, n) < 0) \ \text{then} \\
& \qquad \text{Output that there is a negative length cycle in } G
\end{align*}
\]

Correctness: exercise

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

- Create a \( n \times n \) array Next that stores the next vertex on shortest path for each pair of vertices
- With array Next, for any pair of given vertices \( i, j \) can compute a shortest path in \( O(n) \) time.
Floyd-Warshall Algorithm
Finding the Paths

for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $\text{dist}(i, j, 0) = c(i, j)$  \(\text{(*) } c(i, j) = \infty \text{ if } (i, j) \text{ not edge, } 0 \text{ if } i = j \text{ *)}\)
        $\text{Next}(i, j) = -1$

for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            if $(\text{dist}(i, j, k - 1) > \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1))$ then
                $\text{dist}(i, j, k) = \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1)$
                $\text{Next}(i, j) = k$

for $i = 1$ to $n$ do
    if $(\text{dist}(i, i, n) < 0)$ then
        Output that there is a negative length cycle in $G$

**Exercise:** Given $\text{Next}$ array and any two vertices $i, j$ describe an $O(n)$ algorithm to find a $i-j$ shortest path.

### Summary of results on shortest paths

<table>
<thead>
<tr>
<th></th>
<th>Single vertex</th>
<th>All Pairs Shortest Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td>Dijkstra $O(n \log n + m)$</td>
<td>$n$ * Dijkstra $O(n^2 \log n + nm)$</td>
</tr>
<tr>
<td>Edges cost might be negative, But no negative cycles</td>
<td>Bellman Ford $O(nm)$</td>
<td>$n$ * Bellman Ford $O(n^2 m) = O(n^4)$</td>
</tr>
<tr>
<td>No negative cycles</td>
<td>Floyd-Warshall $O(n^3)$</td>
<td>$n$ * Floyd-Warshall $O(n^3)$</td>
</tr>
</tbody>
</table>
Knapsack Problem

**Input**  Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W$, $w_i$, $v_i$ are all positive integers.

**Goal**  Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.
Knapsack Example

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If $W = 11$, the best is $\{3, 4\}$ giving value 40.

Special Case

When $v_i = w_i$, the Knapsack problem is called the Subset Sum Problem.

Greedy Approach

- Pick objects with greatest value
  - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

- Pick objects with smallest weight
  - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$

- Pick objects with largest $v_i/w_i$ ratio
  - Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

  Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$. 
Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

Observation

Consider an optimal solution $\mathcal{O}$ for $1, \ldots, i$

Case item $i \notin \mathcal{O}$: $\mathcal{O}$ is an optimal solution to items $1$ to $i - 1$.

Case item $i \in \mathcal{O}$: Then $\mathcal{O} - \{i\}$ is an optimum solution for items 1 to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items 1 to $i$ in knapsack of size $w$

Goal: compute $\text{Opt}(n, W)$

Dynamic Programming Solution

Definition

Let $\text{Opt}(i, w)$ be the optimal way of picking items from 1 to $i$, with total weight not exceeding $w$

$$\text{Opt}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{Opt}(i - 1, w) & \text{if } w_i > w \\
\max \left\{ \begin{array}{l}
\text{Opt}(i - 1, w) \\
\text{Opt}(i - 1, w - w_i) + v_i
\end{array} \right\} & \text{otherwise}
\end{cases}$$
An Iterative Algorithm

for $w = 0$ to $W$ do
  $M[0, w] = 0$

for $i = 1$ to $n$ do
  for $w = 1$ to $W$ do
    if ($w_i > w$) then
      $M[i, w] = M[i - 1, w]$
    else
      $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$

Running Time

• Time taken is $O(nW)$
• Input has size $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$; so running time not polynomial but “pseudo-polynomial”!

Knapsack Algorithm and Polynomial time

Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$

Running time of dynamic programming algorithm: $O(nW)$

Not a polynomial time algorithm.

Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$.

Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.

Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if **numbers** in input are of size polynomial in the **combinatorial size** of problem.

Knapsack is NP-hard if numbers are not polynomial in $n$. 
Traveling Salesman Problem

Input: A graph $G = (V, E)$ with non-negative edge costs/lengths, $c(e)$ for edge $e$.

Goal: Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is NP-Hard.
Example: optimal tour for cities of a country (which one?)

An Exponential Time Algorithm

How many different tours are there? \( n! \)

Stirling’s formula: \( n! \approx \sqrt{n}(n/e)^n \) which is \( \Theta(2^{cn \log n}) \) for some constant \( c > 1 \)

Can we do better? Can we get a \( 2^{O(n)} \) time algorithm?
Towards a Recursive Solution

*Order vertices as* \( v_1, v_2, \ldots, v_n \)

*OPT*(*S*): optimum TSP tour for the vertices *S* \( \subseteq \mathcal{V} \) in the graph restricted to *S*. Want *OPT*(*V*).

Can we compute *OPT*(*S*) recursively?

- Say *v* \( \in \mathcal{S} \). What are the two neighbors of *v* in optimum tour in *S*?
- If *u*, *w* are neighbors of *v* in an optimum tour of *S* then removing *v* gives an optimum path from *u* to *w* visiting all nodes in *S* \( - \{v\}\).

Path from *u* to *w* is not a recursive subproblem! Need to find a more general problem to allow recursion.

A More General Problem: TSP Path

**Input** A graph \( G = (V, E) \) with non-negative edge costs/lengths \( c(e) \) for edge *e* and two nodes *s*, *t*

**Goal** Find a path from *s* to *t* of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

- *OPT*(*u*, *v*, *S*): optimum TSP Path from *u* to *v* in the graph restricted to *S* (here *u*, *v* \( \in \mathcal{S} \)).
What is the next node in the optimum path from $u$ to $v$? Suppose it is $w$. Then what is $OPT(u, v, S)$?

$$OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$$

We do not know $w$! So try all possibilities for $w$.

### A Recursive Solution

$$OPT(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + OPT(w, v, S - \{u\}) \right)$$

What are the subproblems for the original problem $OPT(s, t, V)$? $OPT(u, v, S)$ for $u, v \in S, S \subseteq V$.

How many subproblems?

- number of distinct subsets $S$ of $V$ is at most $2^n$
- number of pairs of nodes in a set $S$ is at most $n^2$
- hence number of subproblems is $O(n^2 2^n)$

**Exercise:** Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Disadvantage of dynamic programming solution: memory!
Dynamic Programming = Smart Recursion + Memoization

- How to come up with the recursion?
- How to recognize that dynamic programming may apply?

Some Tips

- Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
- Problems involving trees: recursion based on subtrees.
- More generally:
  - Problem admits a natural recursive divide and conquer
  - If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
  - If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.
Examples

- Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
- Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
- Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
- Independent Set in a graph: break graph into two graphs. What is the interaction? Very high!
- Knapsack: Split items into two sets of half each. What is the interaction?