Chapter 10

More Dynamic Programming

CS 473: Fundamental Algorithms, Spring 2011
February 22, 2011

10.1 All Pairs Shortest Paths

10.1.0.1 Shortest Path Problems

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.

10.1.0.2 Single-Source Shortest Paths

Single-Source Shortest Path Problems

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: $O(nm)$.

10.1.0.3 All-Pairs Shortest Paths

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?

10.1.0.4 Shortest Paths and Recursion

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

Lemma 10.1.1 *Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:*

- *$s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i*

Sub-problem idea: paths of fewer hops/edges

10.1.0.5 Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source s.

$OPT(v, k)$: shortest path distance from s to v using at most k edges.
Note: $dist(s, v) = OPT(v, n - 1)$
Recursion for $OPT(v, k)$:

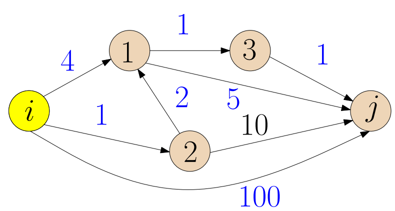
$$OPT(v, k) = \min_{u \in V}(OPT(u, k - 1) + c(u, v)).$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
Leads to Bellman-Ford algorithm — see text book.
$OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops

10.1.0.6 All-Pairs: recursion on index of intermediate nodes

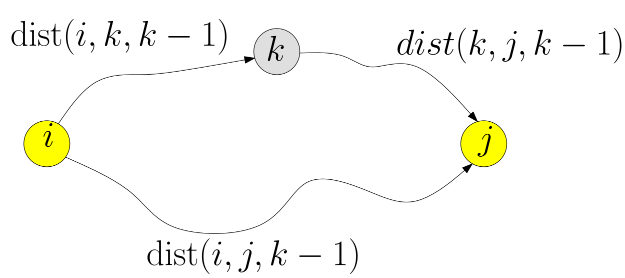
- Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

- $dist(i, j, k)$: 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



$$\begin{aligned} dist(i, j, 0) &= 100 \\ dist(i, j, 1) &= 9 \\ dist(i, j, 2) &= 8 \\ dist(i, j, 3) &= 5 \end{aligned}$$

10.1.0.7 All-Pairs: recursion on index of intermediate nodes



$$dist(i, j, k) = \min(dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1))$$

Base case: $dist(i, j, 0) = c(i, j)$ if $(i, j) \in E$, otherwise ∞

Correctness: If $i \to j$ shortest path goes through k then k occurs only once on the path — otherwise there is a negative length cycle.

10.1.1 Floyd-Warshall Algorithm

10.1.1.1 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

for i = 1 to n do
   for j = 1 to n do
    dist(i, j, 0) = c(i, j) (* c(i, j) = ∞ if (i, j) not edge, 0 if i = j *)

for k = 1 to n do
   for i = 1 to n do
    for j = 1 to n do
        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:* $\Theta(n^3)$.

10.1.2 Floyd-Warshall Algorithm

10.1.2.1 for All-Pairs Shortest Paths

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

```
for i = 1 to n do
   for j = 1 to n do
    dist(i, j, 0) = c(i, j) (* c(i, j) = ∞ if (i, j) not edge, 0 if i = j *)

for k = 1 to n do
   for i = 1 to n do
    for j = 1 to n do
        dist(i, j, k) = min(dist(i, j, k − 1), dist(i, k, k − 1) + dist(k, j, k − 1))

for i = 1 to n do
   if (dist(i, i, n − 1) < 0) then
    Output that there is a negative length cycle in G
```

Correctness: exercise

10.1.2.2 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.

10.1.3 Floyd-Warshall Algorithm

10.1.3.1 Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
      dist(i, j, 0) = c(i, j)  (* c(i, j) = ∞ if (i, j) not edge, 0 if i = j *)
      Next(i, j) = −1
for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
          if (dist(i, j, k − 1) > dist(i, k, k − 1) + dist(k, j, k − 1)) then
              dist(i, j, k) = dist(i, k, k − 1) + dist(k, j, k − 1)
              Next(i, j) = k

for i = 1 to n do
    if (dist(i, i, n − 1) < 0) then
      Output that there is a negative length cycle in G
```

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

10.1.3.2 Summary of results on shortest paths

Single vertex		
No negative edges	Dijkstra	$O(n \log n + m)$
Edges cost might be negative But no negative cycles	Bellman Ford	$O(nm)$

All Pairs Shortest Paths

No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
No negative cycles	Floyd-Warshall	$O(n^3)$

10.2 Knapsack

10.2.0.3 Knapsack Problem

Input Given a Knapsack of capacity W lbs. and n objects with ith 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.

10.2.0.4 Knapsack Example

Example 10.2.1

Item	*1*	*2*	*3*	*4*	*5*
Value	*1*	*6*	*18*	*22*	*28*
Weight	*1*	*2*	*5*	*6*	*7*

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.*

10.2.0.5 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.

10.2.0.6 Towards a Recursive Solution

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

Observation 10.2.2 *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* $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

Opt(i, w): optimum profit for items 1 to i in knapsack of size w
Goal: compute Opt(n, W)

10.2.0.7 Dynamic Programming Solution

Definition 10.2.3 *Let* Opt(i, w) *be the optimal way of picking items from* 1 *to* i*, with total weight not exceeding* w

$$\text{Opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{Opt}(i-1, w) & \text{if } w_i > w \\ \max \begin{cases} \text{Opt}(i-1, w) \\ \text{Opt}(i-1, w - w_i) + v_i \end{cases} & \text{otherwise} \end{cases}$$

10.2.0.8 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"!

10.2.0.9 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.

10.3 Traveling Salesman Problem

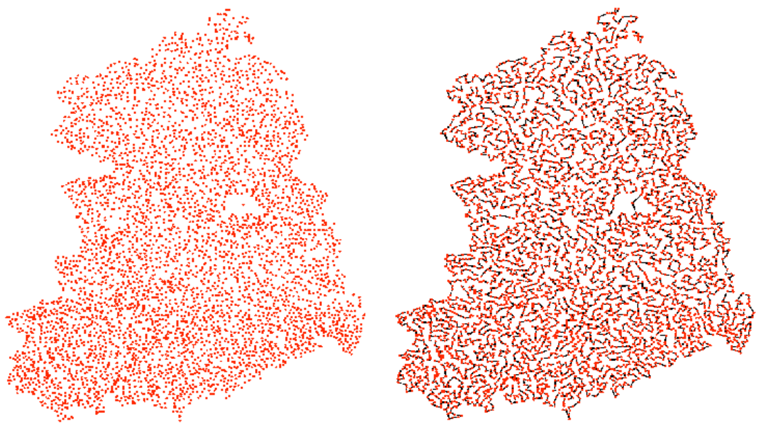
10.3.0.10 Traveling Salesman Problem

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.

10.3.0.11 Example: optimal tour for cities of a country (which one?)



10.3.0.12 An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ 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?

10.3.0.13 Towards a Recursive Solution

- Order vertices as $v_1, v_2, \ldots, v_n$
- $OPT(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to S. Want $OPT(V)$.

Can we compute $OPT(S)$ recursively?

- Say $v \in S$. 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.

10.3.0.14 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 $u, v \in S$).

10.3.1 A More General Problem: TSP Path

10.3.1.1 Continued...

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.

10.3.1.2 A Recursive Solution

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

10.3.1.3 Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

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