CSE 417
Dynamic Programming (pt 3)
More General Sub-problems

UNIVERSITY *of* WASHINGTON

Reminders

- **HW4 is due on Friday**
  - start last week

Dynamic Programming Review

- Apply the steps...
  1. Describe solution in terms of solution to *any* sub-problems
  2. Determine all the sub-problems you'll need to apply this recursively
  3. Solve every sub-problem (once only) in an appropriate order
- Key question:
  1. Can you solve the problem by combining solutions from sub-problems?
- Count sub-problems to determine running time
  - total is number of sub-problems times time per sub-problem

Review From Last Time: Consider Last Element of Opt Solution

- **Q**: How does the opt solution use the last element of the input?
  - construct a (small) set of solutions from sub-problems that <u>must</u> include the opt
- **Weighted Interval Scheduling**
  - opt value = max(opt with last interval, opt without last interval)
  - opt value without last interval is opt value on prefix of the data
- **Max Sub-array Sum**
  - change the problem to find opt interval ending at A[n-1]
  - again, only need opt values on prefixes of the data
- **Optimal Breakout Trades**
  - if sell on last day, max of choice where it starts — $O(n^2)$ worst case

Outline for Today

- Knapsack Problem
- All-Pairs Shortest Paths with Negative Weights
- Shortest Paths with Negative Weights
- Inference with Hidden Markov Models

Knapsack

> **Problem**: Given objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a weight limit W, find the subset of the items with total weight at most W that maximizes the total value.

- any subset $\{i_1, \ldots, ik\}$ such that $w_{i_1} + \ \ldots + w_{i_k} \leq W$
- chosen to maximize $v_{i_1} + \ \ldots + v_{i_k}$

Knapsack Example

- Consider these items with a weight limit of 11.

- Optimal value is 40
- Optimal solution is {3, 4}

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack

- Similar problems arise frequently in practice
  - can easily handle adding additional restrictions of various types
    - see HW6
  - another example of robustness to problem changes

Knapsack

> Brute force doesn't work: there are 2^n subsets to try

> Apply divide and conquer...

> **Q**: Is the last element in the optimal solution?
  - if no, then opt value is the same as on items 1, ..., n-1
  - if yes, then opt value = v_n + opt value on 1, ..., n-1

is this true?

Knapsack

- Brute force doesn't work: there are 2^n subsets to try
- Apply divide and conquer...
- **Q**: Is the last element in the optimal solution?
  - if no, then opt value is the same as on items 1, ..., n-1
  - if yes, then opt value = v_n + opt value on 1, ..., n-1

No! Could have w_n + (weight of opt on 1, ..., n-1) > W

Knapsack

- Brute force doesn't work: there are 2^n subsets to try
- Apply divide and conquer...
- **Q**: Is the last element in the optimal solution?
  - if no, then opt value is the same as on items 1, ..., n-1
  - if yes, then opt value = v_n + (opt value on 1, ..., n-1 with weight limit of $W - w_n$)

optimal substructure

Knapsack

- Brute force doesn't work: there are 2^n subsets to try
- Apply divide and conquer...
- **Q**: Is the last element in the optimal solution?
  - if no, then opt value is the same as on items 1, ..., n-1
  - if yes, then opt value = v_n + (opt value on 1, ..., n-1 with weight limit of $W - w_n$)

opt solution on 1, ..., n
must be optimal on 1, ..., n-1 with weight limit $W - w_n$

Knapsack

- Apply dynamic programming...
  1. Can find opt value for 1, ..., n and limit W using only
     (a) opt value for 1, ..., n-1 and limit W and
     (b) opt value for 1, ..., n-1 and limit W - w_n
  2. Need opt values sub-problems on 1, ..., j-1 and limit V with $j \le n$ and $V \le W$
  3. Solve each of these starting with V=0 or j=1
     - opt value for 1, ..., j and limit 0 = 0
     - opt value for 1 and limit V = v_1 if $w_1 \le V$ and 0 otherwise
     - opt value for 1, ..., j and limit V =
       max(opt value for 1, ..., j-1 and limit V,
       v_j + (opt value for 1, .., j-1 and limit V – w_j) if $w_j \le V$)

Knapsack Example

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

- Start by filling in first column and first row
  - $w_1 = 1$, so we get v_1 for any $W > 0$

Knapsack Example

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

> For {1, 2, 3} and 5: max of spot above (skipping 3) and 18 + spot for {1, 2} and 0

Knapsack Example

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

> For {1, 2, 3, 4} and 11: max of spot above (skipping 4) and
> 22 + spot for {1, 2, 3} and 5

Knapsack Example

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

> For {1, 2, 3, 4, 5} and 11: max of spot above (skipping 5) and
> 28 + spot for {1, 2, 3, 4} and 4

Knapsack Example

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

> Recovers the optimal value of 40

Knapsack

- One sub-problem for each prefix and weight limit, so nW total
- Running time is O(nW) since O(1) per table entry
- This is not efficient if W is large...
  - theory wants time polynomial in lg W (the number of bits used to store W)
  - this algorithm "pseudo-polynomial"
    - (polynomial if the numbers are written in unary, not binary)
  - still extremely useful in practice...

Knapsack

- Running time is O(nW)
- This is not efficient if W is large
  - should not actually expect an efficient algorithm because…
- Knapsack is an NP-complete problem
  - we do not believe any such problem has an efficient algorithm
  - that said, this is an “easy” NP-complete problem
    - our algorithm solves it when W is small
    - and you can efficiently approximate the solution for large W
  - more later…

Outline for Today

> Knapsack Problem
> All-Pairs Shortest Paths with Negative Weights
> Shortest Paths with Negative Weights
> Inference with Hidden Markov Models

All-Pairs Shortest Paths

> **Problem**: Given a weighted graph G on nodes 1 .. n, compute the lengths of the shortest paths between *all pairs* of nodes.
  - $\Theta(n^2)$ outputs
  - edge weights are allowed to be **negative**
  - BUT there can be no cycles with negative total weight

> We will see reasons to allow negative weight edges in the future...

> Here, we will discuss the Floyd-Warshall algorithm
  - names suggest some non-obvious ideas

All-Pairs Shortest Paths

> Usual approach will work for this problem...
  – i.e., consider the sub-problem with a node or edge removed

> However, result is nicest with a different choice of sub-problems

> Sub-problems will use the whole graph G but will return lengths of shortest paths that only use 1 .. k as *intermediate nodes*.

All-Pairs Shortest Paths

- Sub-problems will use the whole graph G but will return shortest paths that only use 1 .. k as *intermediate nodes* on the paths.

- When k = 0, no intermediate nodes are allowed...
  - only path from u to v is the edge (u,v) if it exists

- When k > 0, we can use the solution with 1 .. k-1 allowed...
  - **Q**: does the shortest path from u to v go through k?
  - If not, then shortest path is the same as with 1 .. k-1 allowed
  - If yes, then the shortest path looks like u ~> k ~> v...

All-Pairs Shortest Paths

- Sub-problems will use the whole graph G but will return shortest paths that only use 1 .. k as *intermediate nodes* on the paths.

- When k > 0, we can use the solution with 1 .. k-1 allowed
  - **Q**: does the shortest path from u to v go through k?
  - If not, then shortest path is the same as with 1 .. k-1 allowed
  - If yes, then the shortest path looks like u ~> k ~> v,
    where both the u ~> k part and the k ~> v part do not go through k
    - (a cycle could only increase the length since no negative cycles)
    - we already know the shortest u ~> k and k ~> v paths

All-Pairs Shortest Paths

- Sub-problems will use the whole graph G but will return shortest paths that only use 1 .. k as *intermediate nodes* on the paths.

- When k = 0, no intermediate nodes are allowed
  - only path from u to v is the edge (u,v) if it exists

- When k > 0, we can use the solution with 1 .. k-1 allowed
  - shortest path from u to v with 1 .. k allowed = min(
    shortest path from u to v with 1 .. k-1 allowed,
    shortest path from u to k with 1 .. k-1 allowed +
    shortest path from k to v with 1 .. k-1 allowed)

All-Pairs Shortest Paths Code

```
float[][] dist = /* new n x n table */

for (int u = 0; u < n; u++)
  for (int v = 0; v < n; v++)
    dist[u][v] = (u == v)  0 : length[u][v];  // infinity if no edge

for (int k = 0; k < n; k++)
  for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++)
      dist[u][v] = Math.min(dist[u][v],
                            dist[u][k] + dist[k][v]);
```

All-Pairs Shortest Paths

- Total running time is $O(n^3)$
- Using n calls to Dikjstra (with a binary heap) is $O(n\, m \log n)$
  - slower if $m = O(n^2)$
  - dynamic programming is saving *repeated work* across the n calls
- Can also compute with Strassen's algorithm
  - (i.e., fast matrix multiplication)
  - reduces the running time below $O(n^3)$
  - BUT only with some assumptions on the edge weights

Outline for Today

> Knapsack Problem
> All-Pairs Shortest Paths with Negative Weights
> Shortest Paths with Negative Weights
> Inference with Hidden Markov Models

Shortest Paths with Negative Edges

> **Problem**: Given a weighted graph G on nodes 1 .. n and a node s, compute the length of the shortest paths from s to other nodes.
  - edge weights are now allowed to be **negative**
  - BUT there can be no cycles with negative total weight
  - just asking for length of shortest path, but can get path itself in usual way

> We will discuss the Bellman-Ford algorithm
  - Bellman invented dynamic programming with this algorithm
  - again, names also suggest some non-obvious ideas

Shortest Paths with Negative Edges

- Usual approach will not work for this problem…
  - (i.e., considering the sub-problem with a node or edge removed)
    - (exercise: try it and see what goes wrong here that didn't with all-pairs paths)
  - need to think of a new type of sub-problem to use

- Sub-problems will be the whole graph G but will return lengths of shortest paths having at most k edges.
  - any edges can be used, but the paths can have at most k of them
  - shortest paths cannot have more than n-1 edges
    - otherwise, we would have a cycle
    - all cycles have non-negative cost

Shortest Paths with Negative Edges

- Sub-problems will be the whole graph G but will return lengths of shortest paths having at most k edges.

- When k = 0, there is no path to v unless v = s
  - so shortest path is infinite

- When k > 0, we can use the solution with at most k-1 edges
  - **Q**: does the shortest path use k edges?
  - If not, then shortest path is the same as with at most k-1 edges
  - If yes, then shortest path is s ~> u → v, where (u,v) is an edge
    - s ~> has k-1 edges

Shortest Paths with Negative Edges

- Sub-problems will be the whole graph G but will return lengths of shortest paths having at most k edges.

- When k = 0, there is no path to v unless v = s
  - so shortest path is infinite

- When k > 0, we can use the solution with at most k-1 edges
  - shortest path to v with at most k edges = min(
    shortest path to v with at most k-1 edges,
    min (shortest path to u with at most k-1 edges
    + length of (u,v)) **over all** edges (u,v))

Shortest Paths with Negative Edges

<u>formula</u>: shortest path to v with at most k edges = min(
shortest path to v with at most k-1 edges,
min (shortest path to u with at most k-1 edges
+ length of (u,v)) over all edges (u,v))

> Naive implementation:

```
for k = 1 .. n-1
    for each v in 1 .. n
        length_k[v] = length_{k-1}[v],
        for each u in 1 .. n
            length_k[v] = min(length_k[v],
                              length_{k-1}[u] + edgeLength[u][v])
```

$O(n^3)$ time… no better than Floyd-Warshall

Shortest Paths with Negative Edges

<u>formula</u>: shortest path to v with at most k edges = min(
shortest path to v with at most k-1 edges,
min (shortest path to u with at most k-1 edges
+ length of (u,v)) over all edges (u,v))

> Better implementation:

O(nm) time

for k = 1 .. n-1
  $length_k = length_{k-1}$
  for each (u,v) in E
    $length_k[v] = \min(length_k[v],$
      $length_{k-1}[u] + edgeLength[u][v])$

Shortest Paths with Negative Edges

> Only need to remember the length_{k-1} instead of whole table

> Other ways to improve the practical performance
  - see the textbook
  - worst case is still O(nm)

> Alternative solution: compute shortest path with exactly k edges
  - take minimum over all k at the end
  - that would allow you to find the min average edge cost instead

Outline for Today

> Knapsack Problem
> All-Pairs Shortest Paths with Negative Weights
> Shortest Paths with Negative Weights
> Inference with Hidden Markov Models

Inference with Hidden Markov Models (out of scope)

- **Definition**: a Markov chain is a model of a random process that starts a random state x_1 and transitions randomly between states $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$ according to fixed probabilities.
  - each x_i is in a fixed "state set", 1 .. n
  - probabilities p_{ij} of each transition, $x_i \rightarrow x_j$, depend only on x_i and x_j
    - i.e., it doesn't matter what states came before x_i ("Markov property")

Inference with Hidden Markov Models (out of scope)

> **Problem**: given a Markov chain with states 1 .. n, probabilities q_{ij} of outputting j when in state i, and specific outputs $y_1, \ldots, y_m$, find the sequence of states $x_1, \ldots, x_m$ that best explains the output.
  - i.e., maximize the probability that the chain goes through $x_1 \rightarrow \ldots \rightarrow x_m$ times the probability of those states producing those outputs
  - i.e., $p_{x_1}(p_{x_1 x_2} \cdots p_{x_{m-1} x_m})(q_{x_1 y_1} \cdots q_{x_m y_m})$

Inference with Hidden Markov Models (out of scope)

> Example: stock market trends
(adapted from "Spoken Language Processing", Ch. 8)

– Markov chain has states
for up, down, & sideways trends

> cannot go from up to down or vice versa

– Outputs are price changes

> Find the best explanation for

$$[\underbrace{-1\%, -1\%, -1\%}_{\text{best fits down}}, \underbrace{+10\%}_{\text{up}}]$$

must have sideways between…

	up	side	down
up	0.50	0.25	
side	0.50	0.50	0.50
down		0.25	0.50

	up	side	down
+10%	0.33		
+1%	0.33	0.50	0.10
-1%	0.33	0.50	0.70
-5%			0.20

Inference with Hidden Markov Models (out of scope)

- Many applications including...
  - telecommunications
    - used by cellular networks (Viterbi founded Qualcomm)
  - speech recognition (many)
    - e.g. (vastly simplified), determine intended sounds from actual sounds
      - includes not just similar sounds but likelihood they would appear next to each other
      - (outputs in frequency-domain... use FFT to compute them)
  - natural language processing
    - parsing
  - computational biology

Inference with Hidden Markov Models (out of scope)

- To compute states that are most likely given the outputs, apply dynamic programming…
- Start with the last output…
- For each ending state, want to determine the maximum probability over all sequences of states ending in that state
  - return the state with the largest probability as the last state of the solution

Inference with Hidden Markov Models (out of scope)

- For each state, want to determine the maximum probability over all sequences of states ending in that state.

- For each choice of x_m, find the maximum value of $p_{x_1}(p_{x_1x_2}\cdots p_{x_{m-1}x_m})(q_{x_1y_1}\cdots q_{x_my_m})$ over choices of x_1, .., x_{m-1}
  - this = $p_{x_1}(p_{x_1x_2}\cdots p_{x_{m-2}x_{m-1}})(q_{x_1y_1}\cdots q_{x_{m-1}y_{m-1}})p_{x_{m-1}x_m}q_{x_my_m}$
  - if we fix x_{m-1}, then maximizing the first part is the same problem applied to just y_1, ..., y_{m-1}
  - if we had the solutions for that, then we could compute this by taking the maximum over each choice of x_{m-1}

Inference with Hidden Markov Models (out of scope)

> Sub-problem for each prefix $y_1, \ldots, y_k$ of the outputs

> max over $x_1, .., x_{k-1}$ of $p_{x_1}(p_{x_1x_2} \cdots p_{x_{k-1}x_k})(q_{x_1y_1} \cdots q_{x_ky_k})$
> = max over x_{k-1} of $p_{x_{k-1}x_k} q_{x_ky_k}$ x
> (max over $x_1, \ldots, x_{k-2}$ of $p_{x_1}(p_{x_1x_2} \cdots p_{x_{k-2}x_{k-1}})(q_{x_1y_1} \cdots q_{x_{k-1}y_{k-1}}))$

> Fill in solutions for k = 1 directly from formulas

> Fill in k = 2 … m using the equation above

> Total running time is $O(n^2m)$ (… from double loop on states)

Inference with Hidden Markov Models (out of scope)

- This problem assumed we were given the Markov chain — only the states it went through were unknown
- You can also find the Markov model that best fits the data
- Like coordinate descent, usual approach is an iterative algorithm
- Each iteration requires two steps, one of which is <u>another</u> dynamic programming algorithm