CSE 6140/ CX 4140:

Computational Science and Engineering ALGORITHMS

Instructor: Anne Benoit

Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina

Dynamic Programming: Adding a New Variable

Knapsack problem.

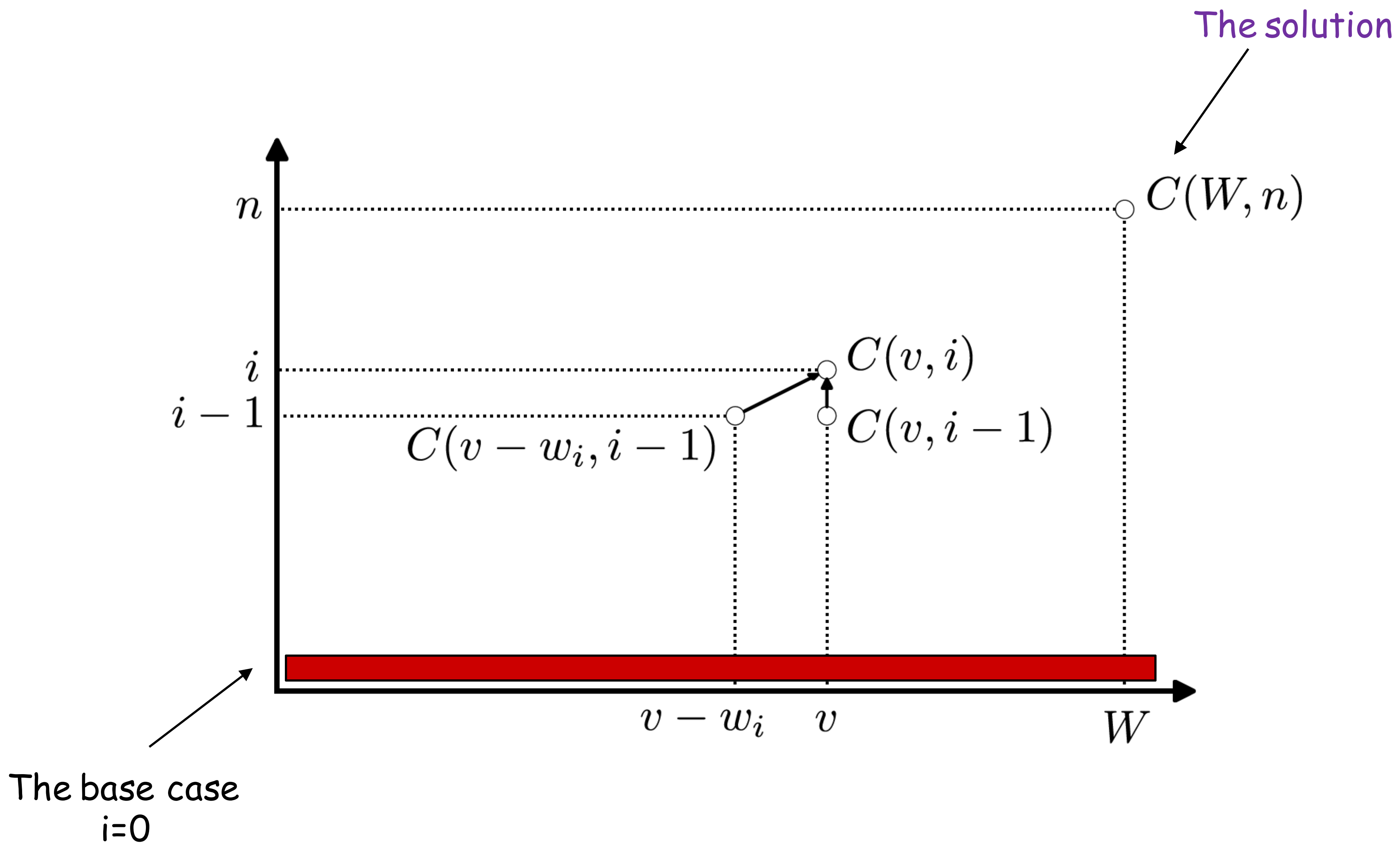
- n items: Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - Gains v_i, and has new remaining weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit $w - w_i$

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

Analyzing Precedence Constraints



Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, $w_1,\dots,w_N$, $v_1,\dots,v_N$

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

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

return M[n, W]
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 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

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

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

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
  - W is part of the input, and can be stored in $\log_2 W$ bits input size
- "Pseudo-polynomial": $\Theta(n\ 2^{\log W})$
- Decision version of Knapsack is NP-complete [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Space. $\Theta(n\ W)$?

- Remember we only need the entries of the matrix for M[i-1,*]
- Just save the row above the current one (not all rows)
- Better space complexity: $\Theta(W)$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array. Optimized space.

```
Input: n, W, $w_1,\dots,w_N$, $v_1,\dots,v_N$

for w = 0 to W
   Mprev[w] = 0

for i = 1 to n
   for w = 1 to W
      if ($w_i$ > w)
         Mcurr[w] = Mprev[w]
      else
         Mcurr[w] = max {Mprev[w], $v_i$ + Mprev[w-$w_i$ ]}
   Mprev = Mcurr

return Mcurr[W]
```

[CLRS 25.2]

ALL-PAIRS SHORTEST PATHS

All-Pairs Shortest Paths

- Given a directed graph G = (V, E), a weight function w: E->R,
  - n=|V|, m=|E|
- Assume no negative weight cycles (shortest path without cycles)

- Goal: create an n × n matrix of shortest-path distances $\delta(v_i, v_j)$
  - If no negative-weight edges, could run Dijkstra's algorithm with O(m lg n) once from each vertex in V:
    - O(nm lg n) with binary heap —> $O(n^3 \lg n)$ if dense, i.e. $m=O(n^2)$

  - We'll see how to do Floyd-Warshall Algorithm in $O(n^3)$ time with no fancy data structure, and allowing for negative-weight edges

All Pairs Shortest Path – Floyd-Warshall Algorithm

- Use optimal substructure of shortest paths: *Any subpath of a shortest path is a shortest path.*

- Given a path $p=(v_1, v_2,\ldots, v_m)$ in the graph, we will call the vertices v_k with index k in {2,…,m-1} the intermediate vertices of p.

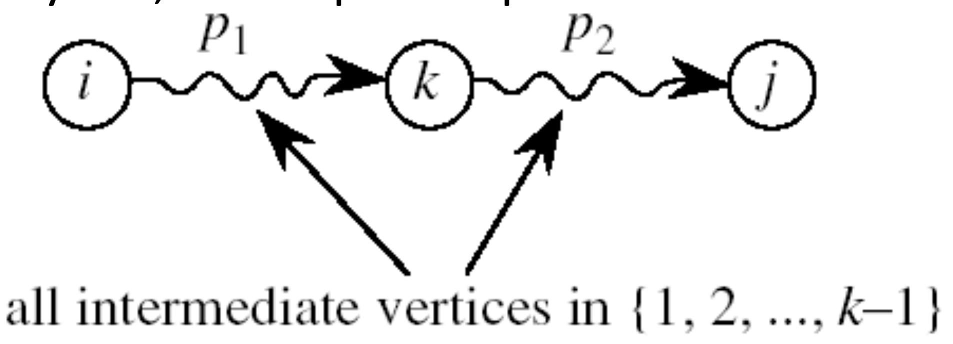
- Create a 3-dimensional table:
  - Let $d_{ij}^{(k)}$ –shortest path weight of any path from i to j where all intermediate vertices are from the set of nodes {1,2, …, k}.
  - Ultimately, we would like to know the values of $d_{ij}^{(n)}$ for each pair of nodes v_i and v_j.

Computing $d_{ij}^{(k)}$

- Base condition: $d_{ij}^{(0)}$ = ? (no intermediate vertices)
  - $d_{ij}^{(0)} = W_{ij}$.

$W_{ij} = 0$	if i=j
$W_{ij} = w(i,j)$	if i≠j and (i,j) in E
$W_{ij} = \infty$	if i≠j and (i,j) not in E

- For k>0:
  - Let p=<v_i, . . . , v_j> be a shortest path from vertex i to vertex j with all intermediate vertices in {1,2, …, k}.
  - Case 1: If k is *not* an intermediate vertex, then all intermediate vertices in p are in {1,2, …, k-1}, and must be an OPT solution with length $d_{ij}^{(k-1)}$ (shortest path for ij using only vertices upto k-1).
  - Case 2: If k *is* an intermediate vertex, then p is composed of 2 shortest subpaths with intermediate nodes drawn from {1,2, …, k-1}
    - no repeated vertices in SP, no cycles, hence p1 and p2 don't contain k
    - optimal substructure of SP
- Goal: $d_{ij}^{(n)}$



Recursive Formulation for $d_{ij}^{(k)}$

- We will use a weight matrix W defined by:

$W_{ij} = 0$	if $i=j$
$W_{ij} = w(i,j)$	if $i \neq j$ and (i,j) in E
$W_{ij} = \infty$	if $i \neq j$ and (i,j) not in E

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

- Draw precedence constraints…

Algorithm

```
FLOYD-WARSHALL$(W, n)$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ to $n$
    do for $i \leftarrow 1$ to $n$
        do for $j \leftarrow 1$ to $n$
            do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

- Running time = $O(n^3)$
- Memory required = $O(n^3)$?
  - we only need results for k-1, when computing results for k
  - We can make it use only $O(n^2)$ space

All-Pairs Shortest Path: Alternatives

- Idea: If the graph is <u>sparse</u> ($|E| \ll |V|^2$), it pays to run Dijkstra's algorithm once from each vertex.
  - $O(nm \log n)$ using binary heap, $O(n^2 \log n + nm)$ using Fibonacci heap.

- Floyd-Warshall still has advantages:
  - <u>Handles negative edges</u>
  - very simple implementation
  - no fancy data structures
  - small constant in big-O

Dynamic Programming Summary

Recipe.

- Characterize structure of problem: identify subproblems whose optimal solutions can be used to build an optimal solution to original pb. Conversely, given an optimal solution to original pb, identify subparts of the solution that are optimal solutions for some subproblems. <Solve more!>
- Write the recurrence and initial cases, know where the solution of pb is.
- Look at precedence constraints (draw a figure) and write the algorithm (iterative, or recursive with memos).
- Study the pb complexity (straightforward with iterative algo; don't forget the time to compute one subproblem).
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: Weighted interval scheduling.
- Multi-way choice: Sequence alignment.
- Dynamic programming over intervals: RNA secondary structure.
- Adding a new variable: Knapsack.

CSE 6140: NP-completeness

based partially on course slides from

Jennifer Welch, George Bebis, and Kevin Wayne

NP-Completeness

- So far we have seen a lot of good news!
  - Such-and-such a problem can be solved quickly (i.e., in close to linear time, or at least a time that is some small polynomial function of the input size)
- NP-completeness is a form of bad news!
  - Evidence that many important problems cannot be solved quickly.
- NP-complete problems really come up all the time!

Why should we care?

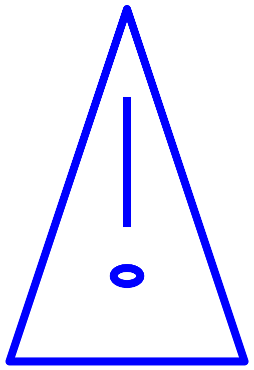
- Knowing that they are hard lets you stop beating your head against a wall trying to solve them…
  - **Restrict the problem:** find special restrictions/variants to the problem for which there is a polynomial time algorithm
  - **Use a heuristic:** come up with a method for solving a reasonable fraction of the common cases.
  - **Solve approximately:** come up with a method that finds solutions provably close to the optimal.
  - **Use an exponential time solution:** if you really have to solve the problem exactly and stop worrying about finding a better solution.

Optimization & Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no
- **Optimization problems**
  - Find a solution with the "best" value
- Optimization problems can be cast as decision problems that are easier to study
  - *E.g.:* Shortest path: G = unweighted directed graph
    - Find a path between u and v that uses the fewest edges
    - *Does a path exist from u to v consisting of at most k edges?*

The class P

- **Class P** consists of (decision) problems that are solvable in polynomial time
- Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$, for some constant k
- Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$
- What is n? Size of the input data!



Data size

- Depends on how data is encoded
  - Need to carefully define problem size to assess problem complexity: how many time steps are required to solve it?
- Integers: can be coded in binary: log size rather than linear one! Poly time assumes that elementary operations can be done in O(1) by a *Turing machine* (add, mult)
- Size of problem: number of memory locations, or bits

A first example: 2-partition

- 2-PARTITION: Given n positive integers $a_1, \ldots, a_n$, is there a subset I of {1, …, n} such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

- What is the size of input data?
  - log n ?
  - Need to encode n integers! Hence encode n in unary
  - What about the integers themselves?
  - Encode the a_i's in binary!

- The data size is therefore:
  1) $\sum_{1 \leq i \leq n} \log(a_i)$?
  2) $\log(n) + \max_{1 \leq i \leq n} \log(a_i)$?
  3) $n \times \max_{1 \leq i \leq n} \log(a_i)$
  4) $\log(n) + \sum_{1 \leq i \leq n} \log(a_i)$?

A first example: 2-partition

- 2-PARTITION: Given n positive integers $a_1, \ldots, a_n$, is there a subset I of {1, …, n} such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

- What is the size of input data?
  - log n ?
  - Need to encode n integers! Hence encode n in unary
  - What about the integers themselves?
  - Encode the a_i's in binary!

- The data size is therefore:
  1) $\sum_{1 \leq i \leq n} \log(a_i)$? YES
  2) $\log(n) + \max_{1 \leq i \leq n} \log(a_i)$? NO
  3) $n \times \max_{1 \leq i \leq n} \log(a_i)$? YES
  4) $\log(n) + \sum_{1 \leq i \leq n} \log(a_i)$? YES, same as 1

A first example: 2-partition

- 2-PARTITION: Given n positive integers $a_1, \ldots, a_n$, is there a subset I of {1, …, n} such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

- Solve the problem: use of DP!
- $c(i,T)$ is true if there is a subset of $\{a_1, \ldots, a_i\}$ of sum T
  - Original problem: $c(n, S/2)$ with $S = \sum_{1 \le i \le n} a_i$
  - Recurrence: $c(i,T) = c(i-1, T-a_i) \vee c(i-1, T)$
  - Base cases? $i=0$, $T < 0$

- Algorithm complexity?
  - $O(nS)$, while data of size $n \log(S)$, this is an exponential algorithm!
  - Pseudo-polynomial: poly if data encoded in unary

- Is this problem in P or not?

A second example: bipartite graphs

- BIPARTITE: Given a graph G, is G a bipartite graph?
- How do we encode a graph? What is the size of data?
  - Remember: n vertices and m edges
- Which of these are correct?
  1. n + m
  2. n + log(m)
  3. log(n) + log(m)
  4. n
- 1,2,4 are all correct, m is polynomial in n (at most n^2 edges)
- Need to enumerate all vertices to describe the pb instance, so 3 is not correct
- Greedy algorithm polynomial in n (good practice problem!), BIPARTITE is in P