

CSE 6140/ CX 4140

Computational Science and Engineering
ALGORITHMS

Dynamic Programming 3

Instructor: Xiuwei Zhang

Assistant Professor

School of Computational Science and Engineering

Dynamic Programming: problems visited

Coin changing problem

$$z(T,i) = \min \{ \begin{array}{l} z(T, i-1) \quad (\text{i-th coin not used}), \\ z(T-v_i, i) + 1 \quad (\text{i-th coin used at least once}) \end{array} \}$$

$$z(T,0) = +\infty \text{ if } T > 0 \quad (\text{no more coins})$$

$$z(0,i) = 0 \quad (\text{we are done})$$

$$z(T,i) = +\infty \text{ if } T < 0 \quad (\text{too much change given})$$

Weighted Interval Scheduling

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise} \end{cases}$$

Dynamic Programming: problems visited

Longest Common Subsequence

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise} \end{cases}$$

Base cases: $c[i, j] = 0$ if $i=0$ or $j=0$

Sequence Alignment

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Back-tracing

The recurrence relation:

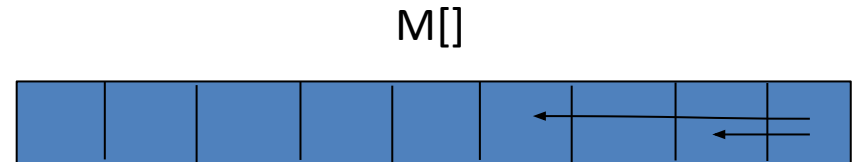
Calculate the optimal score

Back-tracing:

Find the actual solution (path) which gives the optimal score

Back-tracing: weighted interval scheduling

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise} \end{cases}$$



Run M-Compute-Opt(n)

Run Find-Solution(n)

```
Find-Solution(j) {  
    if (j = 0)  
        output nothing  
    else if ( $v_j + M[p(j)] > M[j-1]$ )  
        print j  
        Find-Solution(p(j))  
    else  
        Find-Solution(j-1)  
}
```

LCS bottom-up approach

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

		j	0	1	2	...	n-1	n
			Yj	B	D	...	A	B
i								
0	Xi							
1	A							
...	...							
m-1	C							
m	B							

Allocate array $c[m+1,n+1]$

Back-tracing: LCS

```
Run Find-Solution(m,n)

Find-Solution(i, j) {
  if (i = 0 or j = 0)
    return
  else
    if ( $x_i = y_j$ )
      print  $x_i$ 
      Find-Solution(i-1, j-1)
    else
      if ( $c[i-1, j] > c[i, j-1]$ )
        Find-Solution(i-1, j)
      else
        Find-Solution(i, j-1)
}
```

Note the order of characters in the output string. Since we will print characters with larger indices first, we should either print from right to left or reverse the final string if we print from left to right.

Coming next

Knapsack problem

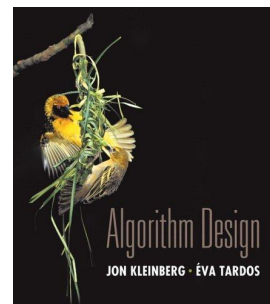
All-pairs shortest path

RNA secondary structure

KNAPSACK PROBLEM [KT 6.4]

Adapted from Slides by
Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

And Bistra Dilkina, Anne Benoit



Knapsack problem

- Given n items and a “knapsack.”
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has weight capacity of W .
- Goal: pack knapsack so as to maximize total value.

Ex. { 1, 2, 5 } has value 35 (and weight 10).

Ex. { 3, 4 } has value 40 (and weight 11).

Ex. { 3, 5 } has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit $W = 11$)

Greedy by value. Repeatedly add item with maximum v_i .

Greedy by weight. Repeatedly add item with minimum w_i .

Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i .

Observation. None of greedy algorithms is optimal.

Knapsack problem: dynamic programming

Subproblem?

Dynamic programming: first attempt

Def. $OPT(i)$ = max-profit subset of items $1, \dots, i$.

Goal. $OPT(n)$.

Case 1. $OPT(i)$ does not select item i .

OPT selects best of $\{ 1, 2, \dots, i - 1 \}$.

← optimal substructure property
(proof via exchange argument)

Case 2. $OPT(i)$ selects item i .

Selecting item i does not immediately imply that we will have to reject other items.

Without knowing what other items were selected before i , we don't even know if we have enough room for i .

Conclusion. Need more subproblems!

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max-profit subset of items $1, \dots, i$ with **weight limit** w .


Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i .  possibly because $w_i > w$

- $OPT(i, w)$ selects best of $\{ 1, 2, \dots, i - 1 \}$ using weight limit w .

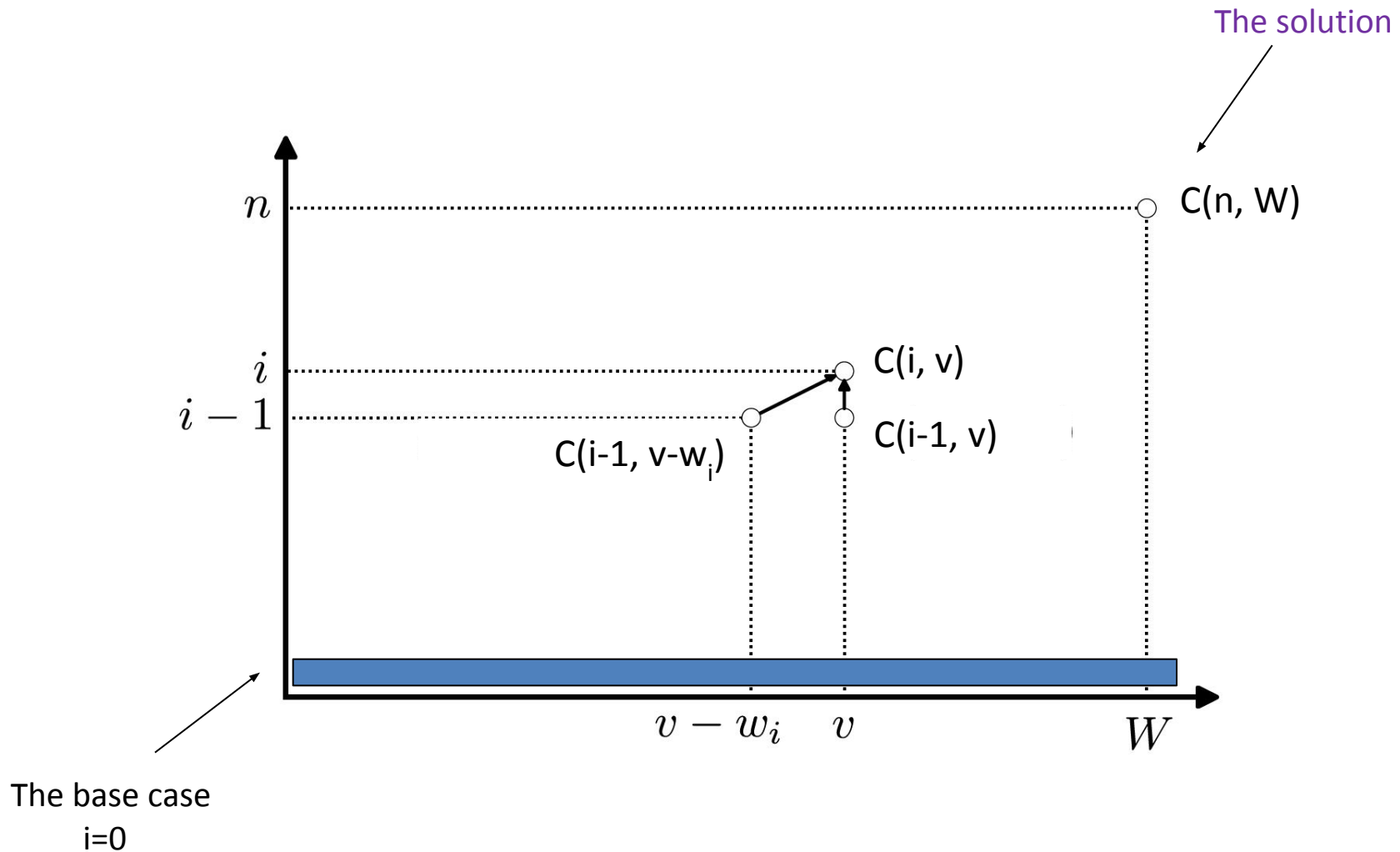
Case 2. $OPT(i, w)$ selects item i .

- Collect value v_i .
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of $\{ 1, 2, \dots, i - 1 \}$ using this new weight limit.

 optimal substructure property
(proof via exchange argument)

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

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 to n											
		0	1	2	3	4	5	6	7	8	9	10	11
i: 1 to n	\varnothing	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

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

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

Knapsack Problem: Running Time

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

Space. $\Theta(n W)$?

- We can keep only 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  $W$ 
     $M_{\text{prev}}[w] = 0$ 

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

return  $M_{\text{curr}}[W]$ 
```

Knapsack Problem: Running Time

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

Is this a polynomial algorithm?

- Not polynomial in input size!
 - W is part of the input
- "Pseudo-polynomial"
- 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. [\[Section 11.8\]](#)

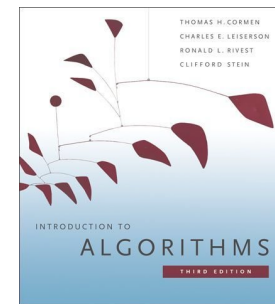
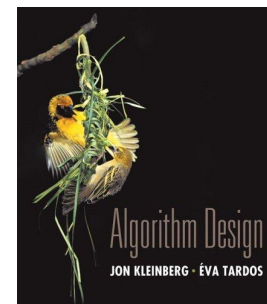
Similar to the coin changing problem.

ALL-PAIRS SHORTEST PATHS

[CLRS 25.2]

Adapted from Slides by
Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

And Bistra Dilkina, Anne Benoit



All-Pairs Shortest Paths

- Given a directed graph $G = (V, E)$, a weight function $w: E \rightarrow \mathbb{R}$,
 - $n = |V|$, $m = |E|$
- Assume **no negative weight cycles** (shortest path without cycles)
- Goal: create an $n \times n$ 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 $\rightarrow O(n^3 \lg n)$ 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

- Optimal substructure?
- Use optimal substructure of shortest paths: *Any subpath of a shortest path is a shortest path.*
- Given a path $p=(v_1, v_2, \dots, v_m)$ in the graph, we will call the vertices v_k with index k in $\{2, \dots, m-1\}$ 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 $\{1, 2, \dots, k\}$.
 - 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)

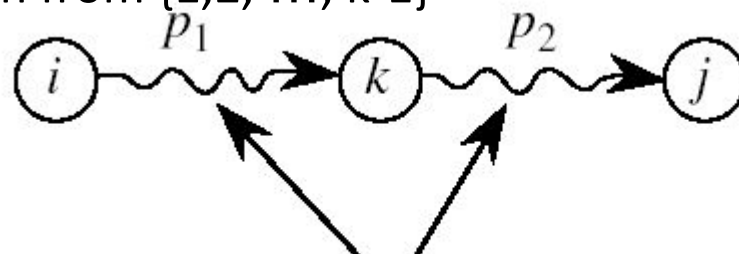
- $d_{ij}^{(0)} = W_{ij}$

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

- For $k > 0$:

- Let $p = \langle v_i, \dots, v_j \rangle$ be a shortest path from vertex i to vertex j with all intermediate vertices in $\{1, 2, \dots, k\}$.
- Case 1: If k is *not* an intermediate vertex, then all intermediate vertices in p are in $\{1, 2, \dots, k-1\}$, and must be an OPT solution with length $d_{ij}^{(k-1)}$ (shortest path for i to j 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 $\{1, 2, \dots, k-1\}$

- no repeated vertices in SP, hence p_1 and p_2 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:

$$W_{ij} = 0 \quad \text{if } i=j$$

$$W_{ij} = w(i,j) \quad \text{if } i \neq j \text{ and } (i,j) \text{ in } E$$

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

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

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 $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

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 sparse ($|E| \ll |V|^2$), 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:
 - Handles negative edges
 - very simple implementation
 - no fancy data structures
 - small constant in big-O

Back-tracing

We have all values of $d_{ij}^{(k)}$. For every pair i, j , we start from $d_{ij}^{(n)}$

```
function FindPath(i,j,k):
```

```
    if  $d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)}$ 
```

```
        print k
```

```
        FindPath(i,k,k-1)
```

```
        FindPath(k,j,k-1)
```

```
    else
```

```
        FindPath(i,j,k-1)
```

```
run FindPath(k)
```