

CMPSCI 611: Advanced Algorithms

Lecture 8: Dynamic Programming

Andrew McGregor

Last Compiled: October 1, 2009

Outline

Dynamic Programming

Shortest Paths

Knapsack Warmup

Problem

- ▶ Input: n items each with value w_i and a capacity W
- ▶ Output: Subset S of elements such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximized.

Knapsack Warmup

Problem

- ▶ Input: n items each with value w_i and a capacity W
- ▶ Output: Subset S of elements such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximized.

Example

Consider input $\{6, 5, 5\}$ and $W = 10$.

Knapsack Warmup

Problem

- ▶ Input: n items each with value w_i and a capacity W
- ▶ Output: Subset S of elements such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximized.

Example

Consider input $\{6, 5, 5\}$ and $W = 10$. Optimal is 10. (Note that greedy doesn't work.)

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j .

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j . Define $\text{knap}(i, j) = -\infty$ if $j < 0$

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j . Define $\text{knap}(i, j) = -\infty$ if $j < 0$

To compute $\text{knap}(i, j)$:

- ▶ If $i = 0$: $\text{knap}(i, j) = 0$
- ▶ Otherwise:
 - ▶ Compute $\text{knap}(i - 1, j)$ and $\text{knap}(i - 1, j - w_i)$
 - ▶ $\text{knap}(i, j) = \max(\text{knap}(i - 1, j), \text{knap}(i - 1, j - w_i) + w_i)$

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j . Define $\text{knap}(i, j) = -\infty$ if $j < 0$

To compute $\text{knap}(i, j)$:

- ▶ If $i = 0$: $\text{knap}(i, j) = 0$
- ▶ Otherwise:
 - ▶ Compute $\text{knap}(i - 1, j)$ and $\text{knap}(i - 1, j - w_i)$
 - ▶ $\text{knap}(i, j) = \max(\text{knap}(i - 1, j), \text{knap}(i - 1, j - w_i) + w_i)$

Claim

The above recursive algorithm will return $\text{knap}(n, W)$ correctly.

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j . Define $\text{knap}(i, j) = -\infty$ if $j < 0$

To compute $\text{knap}(i, j)$:

- ▶ If $i = 0$: $\text{knap}(i, j) = 0$
- ▶ Otherwise:
 - ▶ Compute $\text{knap}(i - 1, j)$ and $\text{knap}(i - 1, j - w_i)$
 - ▶ $\text{knap}(i, j) = \max(\text{knap}(i - 1, j), \text{knap}(i - 1, j - w_i) + w_i)$

Claim

The above recursive algorithm will return $\text{knap}(n, W)$ correctly.

But it's very inefficient...

Try something like divide and conquer...

Definition

Let $\text{knap}(i, j)$ be the optimal solution obtained by using only first i items and capacity j . Define $\text{knap}(i, j) = -\infty$ if $j < 0$

To compute $\text{knap}(i, j)$:

- ▶ If $i = 0$: $\text{knap}(i, j) = 0$
- ▶ Otherwise:
 - ▶ Compute $\text{knap}(i - 1, j)$ and $\text{knap}(i - 1, j - w_i)$
 - ▶ $\text{knap}(i, j) = \max(\text{knap}(i - 1, j), \text{knap}(i - 1, j - w_i) + w_i)$

Claim

The above recursive algorithm will return $\text{knap}(n, W)$ correctly.

But it's very inefficient... Why?

Dynamic Programming Table

Construct a $(n + 1) \times (W + 1)$ table K where $K_{i,j} = \text{knap}(i, j)$:

Dynamic Programming Table

Construct a $(n + 1) \times (W + 1)$ table K where $K_{i,j} = \text{knap}(i, j)$:

- ▶ Fill in “0” for each entry of first row
- ▶ To fill in i -th row use entries of $(i - 1)$ -th row:

$$K_{i,j} = \begin{cases} \max(K_{i-1,j}, K_{i-1,j-w_i} + w_i) & \text{if } j \geq w_i \\ K_{i-1,j} & \text{if } j < w_i \end{cases}$$

Dynamic Programming Table

Construct a $(n + 1) \times (W + 1)$ table K where $K_{i,j} = \text{knap}(i, j)$:

- ▶ Fill in “0” for each entry of first row
- ▶ To fill in i -th row use entries of $(i - 1)$ -th row:

$$K_{i,j} = \begin{cases} \max(K_{i-1,j}, K_{i-1,j-w_i} + w_i) & \text{if } j \geq w_i \\ K_{i-1,j} & \text{if } j < w_i \end{cases}$$

Claim

Running time is $O(nW)$ and space required is $O(W)$.

Dynamic Programming Table

Construct a $(n + 1) \times (W + 1)$ table K where $K_{i,j} = \text{knap}(i, j)$:

- ▶ Fill in “0” for each entry of first row
- ▶ To fill in i -th row use entries of $(i - 1)$ -th row:

$$K_{i,j} = \begin{cases} \max(K_{i-1,j}, K_{i-1,j-w_i} + w_i) & \text{if } j \geq w_i \\ K_{i-1,j} & \text{if } j < w_i \end{cases}$$

Claim

Running time is $O(nW)$ and space required is $O(W)$.

Easy to tweak algorithm to find S and not just $\sum_{i \in S} w_i$

Dynamic Programming Table

Construct a $(n + 1) \times (W + 1)$ table K where $K_{i,j} = \text{knap}(i, j)$:

- ▶ Fill in “0” for each entry of first row
- ▶ To fill in i -th row use entries of $(i - 1)$ -th row:

$$K_{i,j} = \begin{cases} \max(K_{i-1,j}, K_{i-1,j-w_i} + w_i) & \text{if } j \geq w_i \\ K_{i-1,j} & \text{if } j < w_i \end{cases}$$

Claim

Running time is $O(nW)$ and space required is $O(W)$.

Easy to tweak algorithm to find S and not just $\sum_{i \in S} w_i$

Actually Knapsack is NP-complete, have we proved that $P = NP$?

When to use dynamic programming. . .

- ▶ *Optimal Substructure*: The solution to the problem can be found using solutions to smaller sub-problems.

When to use dynamic programming. . .

- ▶ *Optimal Substructure*: The solution to the problem can be found using solutions to smaller sub-problems.
- ▶ *Overlap of Sub-Problems*: By taking advantage of the fact that many identical sub-problems are created, a dynamic programming algorithm may be more efficient than a divide and conquer algorithm.

Outline

Dynamic Programming

Shortest Paths

Shortest Paths

Let $G = (V, E)$ be a directed graph with weights $w : E \rightarrow \mathbb{R}^+$.

Definition

For path $p = (v_1, \dots, v_k)$ be a path, define

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) .$$

The *shortest path* between u and v is

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$$

if there is a path from u to v and ∞ otherwise.

Floyd-Warshall Warm-Up

Problem: Find $\delta(u, v)$ for all $u, v \in V$.

Floyd-Warshall Warm-Up

Problem: Find $\delta(u, v)$ for all $u, v \in V$.

- ▶ Define sub-problems by limiting the set of intermediate nodes

Floyd-Warshall Warm-Up

Problem: Find $\delta(u, v)$ for all $u, v \in V$.

- ▶ Define sub-problems by limiting the set of intermediate nodes
- ▶ Let $d_{ij}^{(k)}$ = length of shortest path from i to j for which all intermediate vertices are in $\{v_1, \dots, v_k\}$

Floyd-Warshall Warm-Up

Problem: Find $\delta(u, v)$ for all $u, v \in V$.

- ▶ Define sub-problems by limiting the set of intermediate nodes
- ▶ Let $d_{ij}^{(k)}$ = length of shortest path from i to j for which all intermediate vertices are in $\{v_1, \dots, v_k\}$
- ▶ Easy: $d_{ij}^{(0)} = w(i, j)$ if $(i, j) \in E$ and $d_{ij}^{(0)} = \infty$ otherwise

Floyd-Warshall Warm-Up

Problem: Find $\delta(u, v)$ for all $u, v \in V$.

- ▶ Define sub-problems by limiting the set of intermediate nodes
- ▶ Let $d_{ij}^{(k)}$ = length of shortest path from i to j for which all intermediate vertices are in $\{v_1, \dots, v_k\}$
- ▶ Easy: $d_{ij}^{(0)} = w(i, j)$ if $(i, j) \in E$ and $d_{ij}^{(0)} = \infty$ otherwise
- ▶ For $k \geq 1$:

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

Floyd-Warshall Algorithm

Algorithm

1. Let $d_{ij}^{(0)} = w(i, j)$ if $(i, j) \in E$ and $d_{ij}^{(0)} = \infty$ otherwise.
2. For $k = 1$ to n :
 - 2.1 For $i, j \in [n]$: let

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

3. Return $d_{ij}^{(n)}$

Floyd-Warshall Algorithm

Algorithm

1. Let $d_{ij}^{(0)} = w(i, j)$ if $(i, j) \in E$ and $d_{ij}^{(0)} = \infty$ otherwise.
2. For $k = 1$ to n :
 - 2.1 For $i, j \in [n]$: let

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

3. Return $d_{ij}^{(n)}$

Running Time: $\Theta(n^3)$ where $n = |V|$

Dijkstra's Warm-Up

Problem: Given $s \in V$, find $\delta(s, v)$ for all $v \in V$.

Dijkstra's Warm-Up

Problem: Given $s \in V$, find $\delta(s, v)$ for all $v \in V$.

Dijkstra's algorithm solves problem if all edges are non-negative:

- ▶ Maintains array $(d[v] : v \in V)$ and we'll prove the invariant:

$$d[v] \geq \delta(s, v)$$

- ▶ Maintains a set of processed vertices R and unprocessed vertices Q . We'll prove that for all $v \in R$:

$$d[v] = \delta(s, v)$$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}$, $Q = V - \{s\}$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}$, $Q = V - \{s\}$
3. While $|Q| \geq 1$:

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}$, $Q = V - \{s\}$
3. While $|Q| \geq 1$:
 - 3.1 Pick $u = \operatorname{argmin}_{v \in Q} d[v]$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}, Q = V - \{s\}$
3. While $|Q| \geq 1$:
 - 3.1 Pick $u = \operatorname{argmin}_{v \in Q} d[v]$
 - 3.2 $R = R + u, Q = Q - u$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}$, $Q = V - \{s\}$

3. While $|Q| \geq 1$:

- 3.1 Pick $u = \operatorname{argmin}_{v \in Q} d[v]$

- 3.2 $R = R + u$, $Q = Q - u$

- 3.3 For each v with $(u, v) \in E$:

$$d[v] = \min(d[u] + w(u, v), d[v])$$

Dijkstra's Algorithm

Algorithm

1. $d[s] = 0$ and for $s \neq v$:

$$d[v] = w(s, v) \text{ if } (s, v) \in E \text{ and } \infty \text{ otherwise}$$

2. $R = \{s\}$, $Q = V - \{s\}$

3. While $|Q| \geq 1$:

- 3.1 Pick $u = \operatorname{argmin}_{v \in Q} d[v]$

- 3.2 $R = R + u$, $Q = Q - u$

- 3.3 For each v with $(u, v) \in E$:

$$d[v] = \min(d[u] + w(u, v), d[v])$$

Running Time: $O(|V|^2)$ for simple implementation but can be improved to $O(|E| + |V| \log |V|)$ using Fibonacci heaps.

For Next Time...

- ▶ Finish reading up to 4.4 of the notes.