Dynamic Programming

CSC 349

Christopher Siu

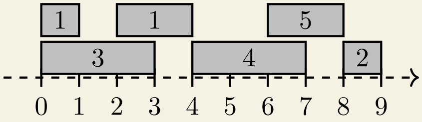
Weighted Activity Selection

Consider the following problem:

- An **activity** is a triple $a_i = (s_i, f_i, w_i)$ of a start time s_i, a finish time f_i, and a weight w_i, taking place during the time $[s_i, f_i)$.
- Given a set of activities, find a maximum weight subset of non-conflicting activities.

Example

Given $\{(2,4,1), (0,3,3), (0,1,1), (4,7,4), (6,8,5), (8,9,2)\}$:



...the maximum weight subset is $\{(0,3,3), (6,8,5), (8,9,2)\}$.

Weighted Activity Selection

NaïveWeightedSelection$(S = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence S of n weighted activities, where $a_i = (s_i, f_i, w_i)$, sorted in increasing order by finish time

Output: The max weight of a subset of non-conflicting activities

if $n = 1$ **then**
  return w_0
else
  let a_j be the last activity such that $f_j \leq s_{n-1}$
  let x be NaïveWeightedSelection$((a_0, a_1, \ldots, a_j))$
  let y be NaïveWeightedSelection$((a_0, a_1, \ldots, a_{n-2}))$
  return $\max\{x + w_{n-1}, y\}$, where $x = 0$ if a_j does not exist

Weighted Activity Selection

Example

The complexity of NAÏVEWEIGHTEDSELECTION is given by:

$$\begin{aligned} T(n) &= a \cdot T(m) + O(g(n)) \\ &= 2 \cdot T(n-1) + O(\log n) \\ &\approx O(2^n \log n) \end{aligned}$$

- NAÏVEWEIGHTEDSELECTION recomputes many subsolutions multiple times.

Definition

Memoization optimizes a function by caching its return values.

Memoization

$\textsc{MemoWeightedSelection}(S = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence S of n weighted activities, where $a_i = (s_i, f_i, w_i)$, sorted in increasing order by finish time

Output: The max weight of a subset of non-conflicting activities

if $T(n)$ is defined **then**
  return $T(n)$
if $n = 1$ **then**
  let $T(n)$ be w_0
else
  let a_j be the last activity such that $f_j \leq s_{n-1}$
  let x be $\textsc{MemoWeightedSelection}((a_0, a_1, \ldots, a_j))$
  let y be $\textsc{MemoWeightedSelection}((a_0, a_1, \ldots, a_{n-2}))$
  let $T(n)$ be $\max\{x + w_{n-1}, y\}$, $x = 0$ if a_j does not exist
return $T(n)$

Dynamic Programming

Definition

A **dynamic programming** algorithm is one that:

1. Solves subproblems, starting with the smallest.
2. Caches subsolutions in a table.
3. Uses cached subsolutions to construct larger solutions.

- Like a divide and conquer approach, dynamic programming identifies and solves subproblems.
- Unlike a divide and conquer approach, dynamic programming efficiently solves interdependent subproblems.

Dynamic Programming

A dynamic programming algorithm populates a table and is characterized by four components:

- **Definition:** A precise, English description of each cell's contents in the table
- **Base Cases:** Rules for populating one or more initial cells in the table
- **Formula:** Rules for populating general cells in the table using the contents of previous cells
- **Solution:** Rules for finding the solution to the problem within the table

Dynamic Programming

$\textsc{WeightedSelection}(S = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence S of n weighted activities, where $a_i = (s_i, f_i, w_i)$, sorted in increasing order by finish time

Output: The max weight of a subset of non-conflicting activities

- *Definition:* Let $T(i)$ be the maximum weight of non-conflicting activities, drawing from activities $(a_0, a_1, \ldots, a_i)$.
- *Base Cases:* $T(0) = w_0$
- *Formula:* $T(i) = \max\{T(j) + w_i, T(i-1)\}$, where a_j is the last activity such that $f_j \leq s_i$ and $T(j) = 0$ if a_j does not exist
- *Solution:* $T(n-1)$

Dynamic Programming

Example

Given $\{(2,4,1),(0,3,3),(0,1,1),(4,7,4),(6,8,5),(8,9,2)\}$:

T	a_0	a_1	a_2	a_3	a_4	a_5
	1	**3**	3	7	**8**	**10**

- Sorting S has complexity $O(n \log n)$.
- The table T has (n) cells.
- Populating a single cell has complexity $O(\log n)$.

WeightedSelection has complexity $O(n \log n)$.

- Note that WeightedSelection has *space* complexity $O(n)$.
- Given its definition, base cases, formula, and solution, the pseudocode for a dynamic programming algorithm is trivial.

String Problems

Alphabets and Strings

Definition

An **alphabet** is a finite set of symbols.

Definition

A **string** is a finite sequence of symbols from an alphabet.

- The **empty string** is the string containing no symbols.
- By convention, parentheses and commas are omitted.

Example

Given the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$:

- $\mathtt{acbabcac}$ is a string over this alphabet.
- $\alpha\kappa\beta\alpha\beta\kappa\alpha\kappa$ is *not* a string over this alphabet.

Longest Common Substrings

Consider the following problem:

- Given two strings, find their longest common substring.

Definition

A string x is a **substring** of a string y if and only if there exist strings u and v such that $y = uxv$.

Example

Given $x = \texttt{caccba}$ and $y = \texttt{acbabcac}$:

$$\texttt{caccba} = u\texttt{cba}v, \quad u = \texttt{cac}, \quad v = \emptyset$$
$$\texttt{acbabcac} = u\texttt{cba}v, \quad u = \texttt{a}, \quad v = \texttt{bcac}$$

...one of their longest common substrings is $\texttt{cba}$.

Longest Common Substrings

$\textsc{LongestSubstring}(x = x_0 x_1 \ldots x_{n-1}, y = y_0 y_1 \ldots y_{m-1})$

Input: A string x of length n and a string y of length m
Output: The length of their longest common substring

- *Definition:* Let $T(i, j)$ be the length of the longest common substring ending with x_i and y_j.
- *Base Cases:* $T(i, -1) = 0$, $T(-1, j) = 0$
- *Formula:* $T(i,j) = \begin{cases} T(i-1, j-1) + 1 & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}$
- *Solution:* $\max\{T\}$

Longest Common Substrings

Example

Given $x =$ `caccba` and $y =$ `acbabcac`:

T		a	c	b	a	b	c	a	c
	0	0	0	0	0	0	0	0	0
c	0	0	1	0	0	0	**1**	0	1
a	0	1	0	0	1	0	0	**2**	0
c	0	0	2	0	0	0	1	0	**3**
c	0	0	**1**	0	0	0	1	0	1
b	0	0	0	**2**	0	1	0	0	0
a	0	1	0	0	**3**	0	0	1	0

- The table T has $(n+1) \times (m+1)$ cells.
- Populating a single cell has complexity $O(1)$.

LongestSubstring has complexity $O(nm)$.

Longest Common Subsequences

Consider the following problem:

- Given two strings, find their longest common subsequence.

Definition

A string x is a **subsequence** of a string y if and only if there exist strings u_i such that $y = u_1 x_1 u_2 \ldots u_k x_k u_{k+1}$.

Example

Given $x = \texttt{caccba}$ and $y = \texttt{acbabcac}$:

$$\texttt{caccba} = \texttt{cacc}\ldots$$

$$\texttt{acbabcac} = \ldots\texttt{c}\ldots\texttt{a}\ldots\texttt{c}\ldots\texttt{c}$$

...one of their longest common subsequences is cacc.

Longest Common Subsequences

$\textsc{LongestSubsequence}(x = x_0 x_1 \ldots x_{n-1}, y = y_0 y_1 \ldots y_{m-1})$

Input: A string x of length n and a string y of length m
Output: The length of their longest common subsequence

- *Definition:* Let $T(i,j)$ be the length of the longest common subsequence drawing from $x_0 x_1 \ldots x_i$ and $y_0 y_1 \ldots y_j$.
- *Base Cases:* $T(i,-1) = 0$, $T(-1,j) = 0$
- *Formula:* $T(i,j) = \begin{cases} T(i-1,j-1)+1 & \text{if } x_i = y_j \\ \max\{T(i-1,j), T(i,j-1)\} & \text{if } x_i \neq y_j \end{cases}$
- *Solution:* $T(n-1,m-1)$

Longest Common Subsequences

Example

Given $x =$ `caccba` and $y =$ `acbabcac`:

T		a	c	b	a	b	c	a	c
	0	**0**	0	0	0	0	0	0	0
c	0	0	**1**	**1**	1	1	1	1	1
a	0	1	1	1	**2**	**2**	2	2	2
c	0	1	2	2	2	2	**3**	**3**	3
c	0	1	2	2	2	2	3	3	**4**
b	0	1	2	3	3	3	3	3	**4**
a	0	1	2	3	4	4	4	4	**4**

- The table T has $(n+1) \times (m+1)$ cells.
- Populating a single cell has complexity $O(1)$.

LongestSubsequence has complexity $O(nm)$.

Levenshtein Distance

Consider the following problem:

- Given two strings, find their Levenshtein distance.

Definition

The **Levenshtein distance** between two strings x and y is the minimum deletions, insertions, substitutions to transform x into y.

Example

Given $x = \texttt{caccba}$ and $y = \texttt{acbabcac}$:

1. Insert a: $\boxed{\texttt{a}}\texttt{caccba}$.
2. Insert b: $\texttt{ac}\boxed{\texttt{b}}\texttt{accba}$.
3. Substitute b: $\texttt{acba}\boxed{\texttt{b}}\texttt{cba}$.
4. Delete b: $\texttt{acbabc}\boxed{}\texttt{a}$.
5. Insert c: $\texttt{acbabca}\boxed{\texttt{c}}$.

...the Levenshtein distance between them is 5.

Levenshtein Distance

$\textsc{LevenshteinDistance}(x = x_0 x_1 \dots x_{n-1}, y = y_0 y_1 \dots y_{m-1})$

Input: A string x of length n and a string y of length m
Output: The minimum number of edits to transform x into y

- *Definition:* Let $T(i,j)$ be the minimum number of edits to transform $x_0 x_1 \dots x_i$ into $y_0 y_1 \dots y_j$.
- *Base Cases:* $T(i,-1) = i+1,\ T(-1,j) = j+1$
- *Formula:* $T(i,j) = \min \left\{ \begin{array}{l} T(i-1,j)+1 \\ T(i,j-1)+1 \\ T(i-1,j-1) + \begin{cases} 0 & \text{if } x_i = y_j \\ 1 & \text{if } x_i \neq y_j \end{cases} \end{array} \right\}$
- *Solution:* $T(n-1,m-1)$

Levenshtein Distance

Example

Given $x = \texttt{caccba}$ and $y = \texttt{acbabcac}$:

T		a	c	b	a	b	c	a	c
	0	**1**	2	3	4	5	6	7	8
c	1	1	**1**	**2**	3	4	5	6	7
a	2	1	2	2	**2**	3	4	5	6
c	3	2	1	2	3	**3**	3	4	5
c	4	3	2	2	3	4	**3**	4	4
b	5	4	3	2	3	3	**4**	4	5
a	6	5	4	3	2	3	4	**4**	**5**

- The table T has $(n+1) \times (m+1)$ cells.
- Populating a single cell has complexity $O(1)$.

LevenshteinDistance has complexity $O(nm)$.

The Knapsack Problem

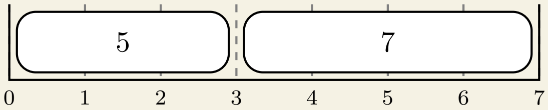
The Knapsack Problem

Consider the following problem:

- An **item** is a pair $a_i = (w_i, v_i)$ of a weight w_i and a value v_i.
- Given a set of items and a weight capacity W, find quantities $x_i \in \{0, 1\}$ such that $\sum x_i \cdot w_i \leq W$ and $\sum x_i \cdot v_i$ is maximized.

Example

Given $S = \{(3,5), (3,1), (2,4), (4,7)\}$ and $W = 7$:



...the maximizing quantities are 1, 0, 0, and 1.

The Knapsack Problem

$\textsc{Knapsack}(S = \{a_0, a_1, \ldots, a_{n-1}\}, W)$

Input: A finite set S of n items, where $a_i = (w_i, v_i)$, a natural W

Output: The maximum value of a knapsack with capacity W, where items may not be repeated

- *Definition:* Let $T(i, j)$ be the maximum value of a knapsack with capacity j drawing from $\{a_0, a_1, \ldots a_i\}$.
- *Base Cases:* $T(i, 0) = 0$, $T(-1, j) = 0$
- *Formula:* $T(i,j) = \begin{cases} T(i-1, j) & \text{if } w_i > j \\ \max\left\{\begin{array}{l} T(i-1, j) \\ T(i-1, j-w_i) + v_i \end{array}\right\} & \text{if } w_i \leq j \end{cases}$
- *Solution:* $T(n-1, W)$

The Knapsack Problem

Example

Given $S = \{(3,5), (3,1), (2,4), (4,7)\}$ and $W = 7$:

T	0	1	2	3	4	5	6	7
	$\mathbf{0}$	0	0	0	0	0	0	0
$(3,5)$	0	0	0	$\mathbf{5}$	5	5	5	5
$(3,1)$	0	0	0	$\mathbf{5}$	5	5	6	6
$(2,4)$	0	0	4	$\mathbf{5}$	5	9	9	9
$(4,7)$	0	0	4	5	7	9	11	$\mathbf{12}$

- The table T has $(n+1) \times (W+1)$ cells.
- Populating a single cell has complexity $O(1)$.

KNAPSACK has **pseudo-polynomial** complexity $O(nW)$.

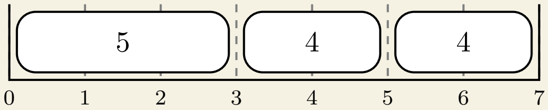
The Unbounded Knapsack Problem

Consider the following problem:

- An **item** is a pair $a_i = (w_i, v_i)$ of a weight w_i and a value v_i.
- Given a set of items and a weight capacity W, find quantities $x_i \in \mathbb{N}$ such that $\sum x_i \cdot w_i \leq W$ and $\sum x_i \cdot v_i$ is maximized.

Example

Given $S = \{(3,5), (3,1), (2,4), (4,7)\}$ and $W = 7$:



...the maximizing quantities are 1, 0, 2, and 0.

The Unbounded Knapsack Problem

$\textsc{UnboundedKnapsack}(S = \{a_0, a_1, \ldots, a_{n-1}\}, W)$

Input: A finite set S of n items, where $a_i = (w_i, v_i)$, a natural W

Output: The maximum value of a knapsack with capacity W, where items may be repeated

- *Definition:* Let $T(i, j)$ be the maximum value of a knapsack with capacity j drawing from $\{a_0, a_1, \ldots a_i\}$.
- *Base Cases:* $T(i, 0) = 0$, $T(-1, j) = 0$
- *Formula:* $T(i,j) = \begin{cases} T(i-1, j) & \text{if } w_i > j \\ \max\begin{Bmatrix} T(i-1, j) \\ T(i, j-w_i) + v_i \end{Bmatrix} & \text{if } w_i \leq j \end{cases}$
- *Solution:* $T(n-1, W)$

The Unbounded Knapsack Problem

Example

Given $S = \{(3,5), (3,1), (2,4), (4,7)\}$ and $W = 7$:

T	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
$(3,5)$	**0**	0	0	**5**	5	5	10	10
$(3,1)$	0	0	0	**5**	5	5	10	10
$(2,4)$	0	0	4	**5**	8	**9**	12	**13**
$(4,7)$	0	0	4	5	8	9	12	**13**

- The table T has $(n+1) \times (W+1)$ cells.
- Populating a single cell has complexity $O(1)$.

UnboundedKnapsack has complexity $O(nW)$.

The Traveling Salesperson Problem

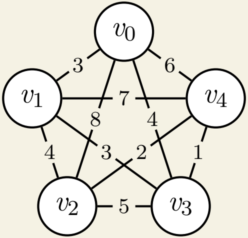
The Traveling Salesperson Problem

Recall the following problem:

- Given a complete, weighted graph, find the Hamiltonian cycle of minimum weight.

Example

Given:



...the Hamiltonian cycle of minimum weight is $(v_0, v_1, v_2, v_4, v_3, v_0)$.

The Held-Karp Algorithm

$\textsc{TravelingSalesperson}(G = (V, E))$

Input: A complete, weighted graph G

Output: The minimum weight of a Hamiltonian cycle in G

- *Definition:* Let $s \in V$ be any vertex and $\mathcal{P}(V - \{s\})$ be the subsets of $V - \{s\}$, sorted in ascending order by cardinality. Let $T(i, j)$ be the minimum weight of a Hamiltonian path starting at s, passing through $S_i \in \mathcal{P}(V - \{s\})$, and ending at $v_j \notin S_i$.
- *Base Cases:* $T(0, j) = w_{sj}$
- *Formula:* $T(i, j) = \min_{v_l \in S_i} \left\{ T(k, l) + w_{lj} \;\middle|\; S_k = S_i - \{v_l\} \right\}$
- *Solution:* $\min_{v_l \in V - \{s\}} \left\{ T(k, l) + w_{ls} \;\middle|\; S_k = V - \{s, v_l\} \right\}$

The Held-Karp Algorithm

Example *(cont.)*

Given $G = (\{v_0, v_1, v_2, v_3, v_4\}, E)$, let $s = v_0$:

T	v_1	v_2	v_3	v_4
$S_0 = \emptyset$	**3**	8	4	6
$S_1 = \{v_1\}$		**7**	6	10
$S_2 = \{v_2\}$	12		13	10
$S_3 = \{v_3\}$	7	9		5
$S_4 = \{v_4\}$	13	8	7	
$S_5 = \{v_1, v_2\}$			12	**9**
$S_6 = \{v_1, v_3\}$		11		7
$S_7 = \{v_1, v_4\}$		12	11	
$\vdots$		*(cont.)*		

The Held-Karp Algorithm

Example *(cont.)*

Given $G = (\{v_0, v_1, v_2, v_3, v_4\}, E)$, let $s = v_0$:

T	v_1	v_2	v_3	v_4
$\vdots$		*(cont.)*		
$S_8 = \{v_2, v_3\}$	13			11
$S_9 = \{v_2, v_4\}$	12		11	
$S_{10} = \{v_3, v_4\}$	10	7		
$S_{11} = \{v_1, v_2, v_3\}$				13
$S_{12} = \{v_1, v_2, v_4\}$			**10**	
$S_{13} = \{v_1, v_3, v_4\}$		9		
$S_{14} = \{v_2, v_3, v_4\}$	11			
$S_{15} = \{v_1, v_2, v_3, v_4\}$				

The Held-Karp Algorithm

Example *(cont.)*

Given $G = (\{v_0, v_1, v_2, v_3, v_4\}, E)$, let $s = v_0$:

T	v_1	v_2	v_3	v_4
$\vdots$		$\vdots$		
(sol'n)	14	17	**14**	19

- Generating $\mathcal{P}(V - \{s\})$ in order has complexity $O(2^{|V|})$.
- The table T has $\left(2^{|V|-1}\right) \times (|V| - 1)$ cells.
- Populating a single cell has complexity $O(|V|)$.
- Finding the solution has complexity $O(|V|)$.

TravelingSalesperson has complexity $O\left(|V|^2 \cdot 2^{|V|}\right)$.