CSE 421

Set Cover, Dynamic Programming

Shayan Oveis Gharan
Set Cover

Given a number of sets on a ground set of elements,

**Goal**: choose minimum number of sets that cover all.

  e.g., a company wants to hire employees with certain skills.
Set Cover

Given a number of sets on a ground set of elements,

Goal: choose minimum number of sets that cover all.

Set cover = 4
A Greedy Algorithm

**Strategy:** Pick the set that maximizes # new elements covered
A Greedy Algorithm

**Strategy**: Pick the set that maximizes # new elements covered
A Greedy Algorithm

Strategy: Pick the set that maximizes # new elements covered
A Greedy Algorithm

Strategy: Pick the set that maximizes # new elements covered
A Greedy Algorithm

Strategy: Pick the set that maximizes \# new elements covered

Thm: Greedy has $\ln n$ approximation ratio
A Tight Example for Greedy
A Tight Example for Greedy
A Tight Example for Greedy
A Tight Example for Greedy
A Tight Example for Greedy
A Tight Example for Greedy

Greedy = 5

OPT = 2
Thm: If the best solution has $k$ sets, greedy finds at most $k \ln(n)$ sets.

Pf: Suppose $OPT = k$
There is set that covers $1/k$ fraction of remaining elements, since there are $k$ sets that cover all remaining elements.
So in each step, algorithm will cover $1/k$ fraction of remaining elements.

#elements uncovered after $t$ steps

$$\leq n \left(1 - \frac{1}{k}\right)^t \leq e^{-t/k}$$

So after $t = k \ln n$ steps, # uncovered elements $< 1$. 
Approximation Alg Summary

• To design approximation Alg, always find a way to lower bound OPT

• The best known approximation Alg for vertex cover is the greedy.
  – It has been open for 40 years to obtain a polynomial time algorithm with approximation ratio better than 2

• The best known approximation Alg for set cover is the greedy.
  – It is NP-Complete to obtain better than ln n approximation ratio for set cover.
Dynamic Programming
Algorithmic Paradigm

**Greedy:**  Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer:**  Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.**  Break up a problem into a series of **overlapping** sub-problems, and build up solutions to larger and larger sub-problems. Memorize the answers to obtain polynomial time ALG.
Dynamic Programming History

**Bellman.** Pioneered the systematic study of dynamic programming in the 1950s.

**Etymology.**

Dynamic programming = planning over time.

Secretary of Defense was hostile to mathematical research. Bellman sought an impressive name to avoid confrontation.

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"
Dynamic Programming Applications

Areas:

- Bioinformatics
- Control Theory
- Information Theory
- Operations Research
- Computer Science: Theory, Graphics, AI, …

Some famous DP algorithms

- Viterbi for hidden Markov Model
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
Dynamic Programming

• Give a solution of a problem using smaller (overlapping) sub-problems where the parameters of all sub-problems are determined in-advance

• Useful when the same subproblems show up again and again in the solution.
Weighted Interval Scheduling
Interval Scheduling

- Job $j$ starts at $s(j)$ and finishes at $f(j)$ and has weight $w_j$
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
Unweighted Interval Scheduling: Review

**Recall**: Greedy algorithm works if all weights are 1:
- Consider jobs in ascending order of finishing time
- Add job to a subset if it is compatible with prev added jobs.

**OBS**: Greedy ALG fails spectacularly (no approximation ratio) if arbitrary weights are allowed:

![Diagram showing the difference between scheduling by finish time and by weight when weights are 1 and 1000]
Weighted Job Scheduling by Induction

Suppose 1, ..., \( n \) are all jobs. Let us use induction:

**IH (strong ind):** Suppose we can compute the optimum job scheduling for \(< n\) jobs.

**IS: Goal:** For any \( n \) jobs we can compute OPT.

**Case 1:** Job \( n \) is not in OPT.
-- Then, just return OPT of 1, ..., \( n - 1 \).

**Case 2:** Job \( n \) is in OPT.
-- Then, delete all jobs not compatible with \( n \) and recurse.

Q: Are we done?
A: No, How many subproblems are there? Potentially \( 2^n \) all possible subsets of jobs.
IS: For jobs 1,…,n we want to compute OPT

**Sorting Idea:** Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

**Case 1:** Suppose OPT has job n.
- So, all jobs $i$ that are not compatible with $n$ are not OPT
- Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with $n$.
- Then, we just need to find OPT of $1, \ldots, p(n)$
**Sorting to reduce Subproblems**

IS: For jobs 1,…,n we want to compute OPT

**Sorting Idea:** Label jobs by finishing time \( f(1) \leq \cdots \leq f(n) \)

**Case 1:** Suppose OPT has job n.
- So, all jobs i that are not compatible with n are not OPT
- Let \( p(n) = \) largest index \( i < n \) such that job i is compatible with n.
- Then, we just need to find OPT of 1, …, \( p(n) \)

**Case 2:** OPT does not select job n.
- Then, OPT is just the optimum 1, …, \( n - 1 \)

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form 1, …, \( i \) for some \( i \)
So, at most \( n \) possible subproblems.
Weighted Job Scheduling by Induction

**Sorting Idea:** Label jobs by finishing time \( f(1) \leq \cdots \leq f(n) \)

Let \( \text{OPT}(j) \) denote the OPT solution of \( 1, \ldots, j \)

To solve \( \text{OPT}(j) \):

**Case 1:** \( \text{OPT}(j) \) has job \( j \).
  - So, all jobs \( i \) that are not compatible with \( j \) are not \( \text{OPT}(j) \)
  - Let \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
  - So \( \text{OPT}(j) = \text{OPT}(p(j)) \cup \{j\} \).

**Case 2:** \( \text{OPT}(j) \) does not select job \( j \).
  - Then, \( \text{OPT}(j) = \text{OPT}(j - 1) \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + \text{OPT}(p(j)), \text{OPT}(j - 1) \right) & \text{otherwise}
\end{cases}
\]
Input: $n$, $s(1), \ldots, s(n)$ and $f(1), \ldots, f(n)$ and $w_1, \ldots, w_n$.

Sort jobs by finish times so that $f(1) \leq f(2) \leq \cdots f(n)$.

Compute $p(1), p(2), \ldots, p(n)$

Compute-Opt$(j)$ {
    if $(j = 0)$
        return 0
    else
        return max($w_j + \text{Compute-Opt}(p(j))$, Compute-Opt$(j-1)$)
}
Even though we have only $n$ subproblems, we do not store the solution to the subproblems.

So, we may re-solve the same problem many many times.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
Algorithm with Memorization

**Memoization.** Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

**Input:** n, s(1), ..., s(n) and f(1), ..., f(n) and w₁, ..., wₙ.

Sort jobs by finish times so that f(1) ≤ f(2) ≤ … f(n).

Compute p(1), p(2), ..., p(n)

for j = 1 to n
    M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    return M[j]
}
Bottom up Dynamic Programming

You can also avoid recursion
• recursion may be easier conceptually when you use induction

Input: $n$, $s(1), \ldots, s(n)$ and $f(1), \ldots, f(n)$ and $w_1, \ldots, w_n$.

Sort jobs by finish times so that $f(1) \leq f(2) \leq \cdots f(n)$.

Compute $p(1), p(2), \ldots, p(n)$

Iterative-Compute-Opt {
    $M[0] = 0$
    for $j = 1$ to $n$
        $M[j] = \max(w_j + M[p(j)], M[j-1])$
}

Output $M[n]$

Claim: $M[j]$ is value of OPT($j$)
Timing: Easy. Main loop is $O(n)$; sorting is $O(n \log n)$
Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$. 

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>p(j)</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$. 

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>p(j)</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

<table>
<thead>
<tr>
<th>j</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>OPT($j$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
p(j) = largest index $i < j$ such that job $i$ is compatible with $j$. 

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>p(j)</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j.$
Knapsack Problem
Knapsack Problem

Given n objects and a "knapsack."
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.

Ex: OPT is { 3, 4 } with value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Ex: {$5, 2, 1$} achieves only value $= 35 \Rightarrow$ greedy not optimal.

Greedy: repeatedly add item with maximum ratio $v_i / w_i$. 