CSE 421

Set Cover, Alg Design by Induction

Shayan Oveis Gharan
A Different Greedy Rule

Greedy 2: Iteratively, pick both endpoints of an uncovered edge.

Vertex cover = 6
Greedy 2: Pick Both endpoints of an uncovered edge

Greedy vertex cover = 16

OPT vertex cover = 8
**Greedy (2) gives 2-approximation**

**Thm:** Size of greedy (2) vertex cover is at most twice as big as size of optimal cover

**Pf:** Suppose Greedy (2) picks endpoints of edges $e_1, \ldots, e_k$. Since these edges do not touch, every valid cover must pick one vertex from each of these edges!

i.e., $OPT \geq k$.

But the size of greedy cover is $2k$. So, Greedy is a 2-approximation.
Set Cover

Given a number of sets on a ground set of $n$ 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
Greedy Gives $O(\log(n))$ approximation

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

**Pf:** Suppose $\text{OPT}=k$
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

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

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 50 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.
Strengthening Induction Hypothesis

We have seen examples on how to design algorithms by induction.

**Basic Idea:** A solution to every instance can be constructed from solutions of smaller instances.

In some cases it may help to strengthen the IH. High-level plan: Prove $P(n) \land Q(n)$ inductively.

**IH:** Assume $P(n - 1) \land Q(n - 1)$.

**IS:** You may use $Q(n - 1)$ to help you to prove $P(n)$. Remember you also have to prove $Q(n)$. 
Maximum Consecutive Subsequence

**Problem:** Given a sequence \( x_1, \ldots, x_n \) of integers (not necessarily positive),

**Goal:** Find a subsequence of consecutive elements s.t., the sum of its numbers is maximum.

\[
1 \quad -3 \quad 7 \quad -2 \quad -3 \quad 8 \quad -10 \quad 1 \quad -7
\]

**Applications:** Figuring out the highest interest rate period in stock market.
Brute Force Approach

Try all consecutive subsequences of the input sequence.

There are \( \binom{n}{2} = \Theta(n^2) \) such sequences.

We can compute the sum of numbers in each such sequence in \( O(n) \) steps.

So, the ALG runs in \( O(n^3) \).

With a clever loop we can do this in \( O(n^2) \).

But, can we solve in linear time?
First Attempt (Induction)

Suppose we can find the maximum-sum subsequence of $x_1, \ldots, x_{n-1}$. Say it is $x_i, \ldots, x_j$

- If $x_n < 0$ then it does not belong to the largest subsequence. So, we can output $x_i, \ldots, x_j$

- Suppose $x_n > 0$
  - If $j = n - 1$ then $x_i, \ldots, x_n$ is the maximum-sum subsequence.
  - If $j < n - 1$ there are two possibilities
    1) $x_i, \ldots, x_j$ is still the maximum-sum subsequence
    2) A sequence $x_k, \ldots, x_n$ is the maximum-sum subsequence

\[-3, \framebox{7}, -2, 1, -8, \framebox{6}, -2, 4\]

\[x_{n-1} \quad x_n\]
Second Attempt (Strengthening Ind Hyp)

**Stronger Ind Hypothesis:** Given $x_1, \ldots, x_{n-1}$ we can compute the maximum-sum subsequence, and the maximum-sum suffix subsequence.

Say $x_i, \ldots, x_j$ is the maximum-sum and $x_k, \ldots, x_{n-1}$ is the maximum-sum suffix subsequences.

- If $x_k + \cdots + x_{n-1} + x_n > x_i + \cdots + x_j$ then $x_k, \ldots, x_n$ will be the new maximum-sum subsequence
Are we done?
Updating Max Suffix Subsequence

Say $x_k, \ldots, x_{n-1}$ is the maximum-sum suffix subsequences of $x_1, \ldots, x_{n-1}$.

- If $x_k + \cdots + x_n \geq 0$ then, $x_k, \ldots, x_n$ is the new maximum-sum suffix subsequence.
- Otherwise, The new maximum-sum suffix is the empty string.
Initialize $S=0$ (Sum of numbers in Maximum Subseq)
Initialize $U=0$ (Sum of numbers in Maximum Suffix)
for (i=1 to n) {
  if ($x[i] + U > S$)
    $S = x[i] + U$
  if ($x[i] + U > 0$)
    $U = x[i] + U$
  else
    $U = 0$
}
Output $S$.

<table>
<thead>
<tr>
<th></th>
<th>-3</th>
<th>7</th>
<th>-2</th>
<th>1</th>
<th>-8</th>
<th>6</th>
<th>-2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S=0$</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$U=0$</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Pf of Correct: Maximum Sum Subseq

Ind Hypo: Suppose

- \( x_i, ..., x_j \) is the max-sum-subseq of \( x_1, ..., x_{n-1} \)
- \( x_k, ..., x_{n-1} \) is the max-suffix-sum-sub of \( x_1, ..., x_{n-1} \)

Ind Step: Suppose \( x_a, ..., x_b \) is the max-sum-subseq of \( x_1, ..., x_n \)

Case 1 \((b < n)\): \( x_a, ..., x_b \) is also the max-sum-subseq of \( x_1, ..., x_{n-1} \)

So, \( a = i, b = j \) and the algorithm correctly outputs OPT

Case 2 \((b = n)\): We must have \( x_a, ..., x_{b-1} \) is the max-suff-sum of \( x_1, ..., x_{n-1} \).

If not, then

\[ x_k + \cdots + x_{n-1} > x_a + \cdots + x_{n-1} \]

So, \( x_k + \cdots + x_n > x_a + \cdots + x_b \) which is a contradiction.

Therefore, \( a = k \) and the algorithm correctly outputs OPT

Special Cases (You don’t need to mention if follows from above):

- The max-suffix-sum is empty string
- There are multiple maximum sum subsequences.
Pf of Correct: Max-Sum Suff Subseq

Ind Hypo: Suppose

- \( x_i, ..., x_j \) is the max-sum-subseq of \( x_1, ..., x_{n-1} \)
- \( x_k, ..., x_{n-1} \) is the max-suffix-sum-sub of \( x_1, ..., x_{n-1} \)

Ind Step: Suppose \( x_a, ..., x_n \) is the max-sum-subseq of \( x_1, ..., x_n \)

Note that we may also have an empty sequence

Case 1 (OPT is empty): Then, we must have \( x_k + ... + x_n < 0 \). So the algorithm correctly finds max-suffix-sum subsequence.

Case 2 (\( x_a, ..., x_n \) is nonempty): We must have \( x_a + ... + x_n \geq 0 \). Also, \( x_a, ..., x_{n-1} \) must be the max-suffix-sum of \( x_1, ..., x_{n-1} \). If not, \( x_a + ... + x_{n-1} < x_k + ... + x_{n-1} \) which implies \( x_a + ... + x_n < x_k + ... + x_n \) which is a contradiction.

Therefore, \( a = k \). So, the algorithm correctly finds max-suffix-sum subsequence.
Summary

• Try to reduce an instance of size n to smaller instances
  • Never solve a problem twice

• Before designing an algorithm study properties of optimum solution

• If ordinary induction fails, you may need to strengthen the induction hypothesis