Lecture 14: Greedy Algorithms
CSCI 700 - Algorithms I

Andrew Rosenberg
Last Time

- Dynamic Programming
Today

- Greedy Algorithms
Recap of Optimization.

- Find the greatest or smallest solution to some problem.
Recap of Optimization.

- Find the greatest or smallest solution to some problem.
- There are many valid solutions.
- Find the best solution – Efficiently.
Recap of Optimization.

- Find the greatest or smallest solution to some problem.
- There are many valid solutions.
- Find the best solution – Efficiently.
- Search over the space of (partial) solutions.
What is the shortest path between two points?
Shortest paths

- What is the shortest path between two points?

A line, but what if there are constraints?
What is the shortest path between two points?
You have a knapsack of a fixed size, $k$.

There is a set of $n$ available items with associated sizes $s_i$ and values $v_i$ where $i \in \{1..n\}$

You want to take the greatest value of items.
Knapsack problems

- You have a knapsack of a fixed size, $k$.
- There is a set of $n$ available items with associated sizes $s_i$ and values $v_i$ where $i \in \{1..n\}$
- You want to take the greatest value of items.
- **Fractional** Knapsack problem
- What’s the optimal solution?
You have a knapsack of a fixed size, \( k \).

There is a set of \( n \) available items with associated sizes \( s_i \) and values \( v_i \) where \( i \in \{1..n\} \).

You want to take the greatest value of items.

0-1 Knapsack problem

You must take all or none of each item.
Queuing

- Imagine a bank.
- There are a finite number of resources (tellers) and a set of jobs (customers) waiting to access them.
- How do we keep the average wait time to a minimum?

Operations Research
Greedy algorithms search for **global optima**, by making decisions towards **local optima**.
Define Greedy Algorithm

- Greedy algorithms search for global optima, by making decisions towards local optima.
- Components of a greedy strategy.
  - Determine the **optimal substructure** of the problem
  - Construct a recursive solution that covers the search space.
  - Prove that at each stage of the recursion, one of the optimal choices is the greedy choice. I.e. The greedy choice is always a safe choice.
  - Show that all but one of the subproblems constructed by making the greedy choice are empty. – There is only one step following the greedy choice.
  - Modify the recursive solution to implement the greedy strategy.
  - *Convert the recursive algorithm into an iterative algorithm.
Define a space of subproblems.

Let $S_{ij}$ be a subproblem.

- For example, travel from city $i$ to city $j$.

Let $A_{ij}$ be an optimal solution to subproblem $S_{ij}$.

- Say, the shortest route between city $i$ and city $j$.

if $A_{ij}$ includes state $k$ – travel through city $k$ – then the solutions $A_{ik}$ to $S_{ik}$ and $A_{kj}$ to $S_{kj}$ must both be optimal.
Define a space of subproblems.

Let $S_{ij}$ be a subproblem.

For example, travel from city $i$ to city $j$.

Let $A_{ij}$ be an optimal solution to subproblem $S_{ij}$.

Say, the shortest route between city $i$ and city $j$.

if $A_{ij}$ includes state $k$ – travel through city $k$ – then the solutions $A_{ik}$ to $S_{ik}$ and $A_{kj}$ to $S_{kj}$ must both be optimal.

Prove it.
Activity Selection

- Scheduling problem
- We have a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ proposed activities.
- Each activity $a_i$ has a start time, $s_i$, and end time, $e_i$, where $0 \leq s_i < f_i < \infty$.
- We say that two activities are **compatible** if they don’t overlap.
- **Problem:** Identify the largest set of compatible activities.
Given the set of activities below, identify the largest set of compatible activities.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>(s_i)</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>(f_i)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
Let $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ be the subset of activities that start after activity $a_i$ ends and ends before $a_j$ starts.

Let $A_{ij}$ be an optimal solution of $S_{ij}$ a largest possible subset of activities that can be drawn from $S_{ij}$.

**Problem:** Find the largest possible set $S_{0,n+1}$.  

Recursive Coverage of Activity Selection

- $A_{ij}$ be an optimal solution of $S_{ij}$.
- $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
Recursive Coverage of Activity Selection

- Let $A_{ij}$ be an optimal solution of $S_{ij}$.
- $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- But we don’t know what $k$ is.
- $A_{ij}$ be an optimal solution of $S_{ij}$.
- $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- But we don’t know what $k$ is.
- Try them all.
- $|A_{ij}| = \max_{i < k < j} |A_{ik}| + |A_{kj}| + 1$
Theorem

Consider a subproblem $S_{ij}$ and let $a_m$ be the activity in $S_{ij}$ with the earliest finishing time.

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

Then

- $a_m$ is a member of a maximum subset of compatible activities of $S_{ij}$ – if it’s not it can be swapped for the activity in $A_{ij}$ that ends earliest.

- The subproblem $S_{im}$ is empty, so choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem. – nothing can fit between $i$ and $m$.

- Who cares?
Theorem

Consider a subproblem $S_{ij}$ and let $a_m$ be the activity in $S_{ij}$ with the earliest finishing time.

$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

Then

- $a_m$ is a member of a maximum subset of compatible activities of $S_{ij}$ – if it’s not it can be swapped for the activity in $A_{ij}$ that ends earliest.

- The subproblem $S_{im}$ is empty, so choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem. – nothing can fit between $i$ and $m$.

Who cares?

This shows that we don’t have to search the whole space – limiting the subspace.
Theorem

Consider a subproblem \( S_{ij} \) and let \( a_m \) be the activity in \( S_{ij} \) with the earliest finishing time.

\[
f_m = \min\{f_k : a_k \in S_{ij}\}
\]

Then

- \( a_m \) is a member of a maximum subset of compatible activities of \( S_{ij} \) – if it’s not it can be swapped for the activity in \( A_{ij} \) that ends earliest.

- The subproblem \( S_{im} \) is empty, so choosing \( a_m \) leaves \( S_{mj} \) as the only nonempty subproblem. – nothing can fit between \( i \) and \( m \).

Who cares?

This shows that we don’t have to search the whole space – limiting the subspace.

Pick the item \( a_m \) with the earliest finishing time \( f_m \) at each step.
Activity Selection Solution

- Sort A by finishing time.
- Include $a_1$ in the solution.
- Include the compatible activity with the smallest finishing time in the solution.
- Repeat
GreedyActivitySelection(A)

\[ A \leftarrow \text{SortByFinishingTime}(A) \]
\[ n \leftarrow A.\text{size} \]
\[ S \leftarrow \{A[1]\} \]
\[ \text{for } m \leftarrow 2 \text{ to } n \text{ do} \]
\[ \quad \text{if } s_m \geq f_i \text{ then} \]
\[ \quad \quad S \leftarrow S \cup \{A[i]\} \]
\[ \quad \quad i \leftarrow m \]
\[ \quad \text{end if} \]
\[ \text{end for} \]
Fractional Knapsack Problem

- You have a knapsack of a fixed size, \( k \).
- There is a set of \( n \) available items with associated sizes \( s_i \) and values \( v_i \) where \( i \in \{1..n\} \).
- You want to take the greatest value of items whose size sums to less than \( k \).
- **Fractional** Knapsack problem
- You can take as much or as little of each item as you like.
Optimal substructure.

Consider the most valuable load that has size $k$.

If we remove $s$ of item $j$ from the load, then the remaining load must be the most valuable with size $k - s$ that can be taken from the n-1 original items and the remaining $s_j - s$ units of item $j$. 
What is the greedy choice here?
What is the greedy choice here?
- Take as much as possible of the most valuable item available remaining.
- Define value as cost per unit size.
Can we prove this?
Can we prove this?

Let \( i \) be the most valuable item. Assume there exists an optimal solution \( S \) with value \( V \).

Assume that \( S \) does not contain as much of \( i \) as it could have.

Then there exists some size \( s \) of \( i \) that could have been included in \( S \), but rather, \( S \) includes \( s \) units of some other item \( j \).

However, \( v_i > v_j \). Thus \( s \cdot v_i > s \cdot v_j \).

Therefore, if we replace \( j \) by \( i \) in \( S \), this new solution \( S' \) has value \( V' = V - s \cdot v_j + s \cdot v_i \).

\( V' > V \), which is a contradiction since \( S \) is optimal. Therefore \( i \) must be in \( S \).
Therefore the Fractional Knapsack Problem can be solved in \(\Theta(n \log n)\).

The items are sorted by value, then the largest set is identified.
The 0-1 Knapsack Problem is identical to the Fractional Knapsack problem with one constraint.

- If an item is selected, either all of it or none of it is included in the solution.
This problem has a similar optimal substructure.

Consider the most valuable load that has size $k$.

If we remove item $j$ from the load, then the remaining load must be the most valuable with size $k - s_j$ that can be taken from the $n-1$ remaining items.
This problem has a similar optimal substructure.

Consider the most valuable load that has size $k$.

If we remove item $j$ from the load, then the remaining load must be the most valuable with size $k - s_j$ that can be taken from the n-1 remaining items.

However, identifying item $j$ is not so trivial.
Does the greedy strategy work for the 0-1 knapsack problem?
Does the greedy strategy work for the 0-1 knapsack problem?
Does the greedy strategy work for the 0-1 knapsack problem?
Does the greedy strategy work for the 0-1 knapsack problem?
Discussion Problem
Next time
  • Huffman Coding
For Next Class
  • Read 16.3