CS141: Intermediate Data Structures and Algorithms

Greedy Algorithms

Amr Magdy
Activity Selection Problem

- Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$ where each activity $i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$.
- An activity $a_i$ happens in the half-open time interval $[s_i, f_i)$.
Activity Selection Problem

- Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$ where each activity $i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$.
- An activity $a_i$ happens in the half-open time interval $[s_i, f_i)$.
- Activities compete on a single resource, e.g., CPU.
Activity Selection Problem

- Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$ where each activity $i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$.
- An activity $a_i$ happens in the half-open time interval $[s_i, f_i)$.
- Activities compete on a single resource, e.g., CPU
- Two activities are said to be compatible if they do not overlap.
Activity Selection Problem

- Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$ where each activity $i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$.
- An activity $a_i$ happens in the half-open time interval $[s_i, f_i)$.
- Activities compete on a single resource, e.g., CPU.
- Two activities are said to be compatible if they do not overlap.
- The problem is to find a maximum-size compatible subset, i.e., a one with the maximum number of activities.
Example

- $a_3[0,6)$
- $a_{10}[2,14)$
- $a_1[1,4)$
- $a_9[8,12)$
- $a_5[3,9)$
- $a_{11}[12,16)$
- $a_2[3,5)$
- $a_{11}[12,16]$
A Compatible Set

a3[0,6)

a10[2,14)

a1[1,4)  a9[8,12)

a5[3,9)

a4[5,7)  a8[8,11)

a2[3,5)  a7[6,10)  a11[12,16)

a6[5,9)
A Better Compatible Set

$\text{a3}[0,6)$

$\text{a10}[2,14)$

$\text{a1}[1,4)$

$\text{a9}[8,12)$

$\text{a5}[3,9)$

$\text{a4}[5,7)$

$\text{a8}[8,11)$

$\text{a2}[3,5)$

$\text{a7}[6,10)$

$\text{a11}[12,16)$

$\text{a6}[5,9)$
An Optimal Solution

a3[0,6)
a10[2,14)
a1[1,4)
a9[8,12)
a5[3,9)
a4[5,7)
a8[8,11)
a2[3,5)
a7[6,10)
a11[12,16)
a6[5,9)
Another Optimal Solution

- a3[0,6]
- a10[2,14]
- a1[1,4]
- a9[8,12]
- a5[3,9]
- a4[5,7]
- a8[8,11]
- a2[3,5]
- a7[6,10]
- a6[5,9]
- a11[12,16]
Activity Selection Problem

- Solution algorithm?
  - Brute force (naïve): all possible combinations → $O(2^n)$
  - Can we do better?
  - Divide line for D&C is not clear
Activity Selection Problem

Solution algorithm?
- Brute force (naïve): all possible combinations $\rightarrow O(2^n)$
- Can we do better?
- Divide line for D&C is not clear

Does the problem have optimal substructure?
- i.e., the optimal solution of a bigger problem has optimal solutions for subproblems
Activity Selection Problem

Does the problem have optimal substructure?
- i.e., the optimal solution of a bigger problem has optimal solutions for subproblems

Assume A is an optimal solution for S
- Is $A' = A - \{a_i\}$ an optimal solution for $S' = S - \{a_i \text{ and its incompatible activities}\}$?
- If $A'$ is not an optimal solution, then there an optimal solution $A''$ for $S'$ so that $|A''| > |A'|$
  - Then $B = A'' \cup \{a_i\}$ is a solution for $S$, $|B| = |A''| + 1, |A| = |A'| + 1$
  - Then $|B| > |A|$, i.e., $|A|$ is not an optimal solution, contradiction
  - Then $A'$ must be an optimal solution for $S'$
Activity Selection Problem

› Does the problem have optimal substructure?
  › i.e., the optimal solution of a bigger problem has optimal solutions for subproblems

› Assume A is an optimal solution for S
  › Is A’ = A-{a_i} an optimal solution for S’ = S-{a_i and its incompatible activities}? 
  › If A’ is not an optimal solution, then there an optimal solution A” for S’ so that |A”| > |A’|
  › Then B=A’ U {a_i} is a solution for S, |B|=|A”|+1, |A|=|A’|+1
  › Then |B| > |A|, i.e., |A| is not an optimal solution, contradiction
  › Then A’ must be an optimal solution for S’

› Proof by contradiction
  › Assume the opposite of your goal
  › Given that prove a contradiction, then your goal is proved
Activity Selection Problem

What does having optimal substructure means?
- We can solve smaller problems, then expand to larger
  - Similar to dynamic programming
Activity Selection Problem

- What does having optimal substructure means?
  - We can solve smaller problems, then expand to larger
    - Similar to dynamic programming
- Instead, can we a **greedy** choice?
  - i.e., take the best choice so far, reduce the problem size, and solve a subproblem later
Activity Selection Problem

- What does having optimal substructure means?
  - We can solve smaller problems, then expand to larger
    - Similar to dynamic programming

- Instead, can we a **greedy** choice?
  - i.e., take the best choice so far, reduce the problem size, and solve a subproblem later

- Greedy choices
  - Longest first
  - Shortest first
  - Earliest start first
  - Earliest finish first
  - ...?
Activity Selection Problem

- Greedy choice: earliest finish first
  - Why? It leaves as much resource as possible for other tasks
Activity Selection Problem

- Greedy choice: earliest finish first
  - Why? It leaves as much resource as possible for other tasks
- Solution:
  - Include earliest finish activity $a_m$ in solution A
  - Remove all $a_m$’s incompatible activities
  - Repeat for the remaining earliest finish activity
Activity Selection Problem: Greedy Solution

- $a_3[0,6)$
- $a_{10}[2,14)$
- $a_1[1,4)$
- $a_9[8,12)$
- $a_5[3,9)$
- $a_4[5,7)$
- $a_8[8,11)$
- $a_2[3,5)$
- $a_7[6,10)$
- $a_{11}[12,16)$
- $a_6[5,9)$
Activity Selection Problem: Greedy Solution

- \([a3[0,6)]\)
- \([a10[2,14)]\)
- \([a1[1,4)]\)
- \([a9[8,12)]\)
- \([a5[3,9)]\)
- \([a4[5,7)]\)
- \([a8[8,11)]\)
- \([a2[3,5)]\)
- \([a7[6,10)]\)
- \([a11[12,16)]\)
- \([a6[5,9)]\)
Activity Selection Problem: Greedy Solution

- a1[1,4)
- a9[8,12)
- a4[5,7)
- a8[8,11)
- a7[6,10)
- a11[12,16)
- a6[5,9)
Activity Selection Problem: Greedy Solution

- $a_1[1,4)$
- $a_4[5,7)$
- $a_6[5,9)$
- $a_7[6,10)$
- $a_8[8,11)$
- $a_9[8,12)$
- $a_{11}[12,16)$
Activity Selection Problem: Greedy Solution

- $a_1[1,4)$
- $a_4[5,7)$
- $a_8[8,11)$
- $a_9[8,12)$
- $a_{11}[12,16)$
Activity Selection Problem: Greedy Solution

- $a_1[1,4)$
- $a_9[8,12)$
- $a_{11}[12,16)$
- $a_{10}[10,11)$
Activity Selection Problem: Greedy Solution

a1[1,4)  
a4[5,7)   a8[8,11)  
a11[12,16]
Activity Selection Problem: Greedy Solution

- a1[1,4]
- a4[5,7]
- a8[8,11]
- a11[12,16]
Activity Selection Problem

- Pseudo code?
findMaxSet(Array a, int n)
{
- Sort “a” based on earliest finish time
- result \leftarrow \{\}
- for i = 1 to n
  validAi = true
  for j = 1 to result.size
    if (a[i] is incompatible with result[j])
      validAi = false
    if (validAi)
      result \leftarrow result U a[i]
- return result
}
Activity Selection Problem

- Is greedy choice is enough to get optimal solution?
Activity Selection Problem

- Is greedy choice is enough to get optimal solution?
- Greedy choice property
  - Prove that if \( a_m \) has the earliest finish time, it must be included in some optimal solution.
Activity Selection Problem

Is greedy choice is enough to get optimal solution?

Greedy choice property

Prove that if \( a_m \) has the earliest finish time, it must be included in some optimal solution.

Assume a set \( S \) and a solution set \( A \), where \( a_m \notin A \)

Let \( a_j \) is the activity with the earliest finish time in \( A \) (not in \( S \))

Compose another set \( A' = A - \{a_j\} \cup \{a_m\} \)

\( A' \) still have all activities disjoint (as \( a_m \) has the global earliest finish time and \( A \) activities are already disjoint), and \( |A'| = |A| \)

Then \( A' \) is an optimal solution

Then \( a_m \) is always included in an optimal solution
Elements of a Greedy Algorithm

1. Optimal Substructure
2. Greedy Choice Property
Greedy vs. Dynamic Programming

- Solving the bigger problem include
  One choice (greedy) vs Multiple possible choices
Greedy vs. Dynamic Programming

- Solving the bigger problem include
  
  One choice (greedy) vs Multiple possible choices

  One subproblem vs A lot of overlapping subproblems
Greedy vs. Dynamic Programming

- Solving the bigger problem include
  One choice (greedy) vs Multiple possible choices

  One subproblem A lot of overlapping subproblems

- Both have optimal substructure
Greedy vs. Dynamic Programming

- Solving the bigger problem include
  One choice (greedy) vs Multiple possible choices

  One subproblem  A lot of overlapping subproblems

- Both have optimal substructure

- Elements:

<table>
<thead>
<tr>
<th>Greedy</th>
<th>DM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal substructure</td>
<td>Optimal substructure</td>
</tr>
<tr>
<td>Greedy choice property</td>
<td>Overlapping subproblems</td>
</tr>
</tbody>
</table>
Knapsack Problem

item 1
10
$60

item 2
20
$100

item 3
45
$120

knapsack
50
Knapsack Problem

- 0-1 Knapsack: Each item either included or not
- Greedy choices:
  - Take the most valuable → Does not lead to optimal solution
  - Take the most valuable per unit → Works in this example
Knapsack Problem

- 0-1 Knapsack: Each item either included or not
- Greedy choices:
  - Take the most valuable → Does not lead to optimal solution
  - Take the most valuable per unit → Does not work
Knapsack Problem

- **Fractional Knapsack**: Part of items can be included.

  Item 1: $10, $60
  Item 2: $20, $100
  Item 3: $30, $120

  Knapsack: $50
Knapsack Problem

- Fractional Knapsack: Part of items can be included
- Greedy choices:
  - Take the most valuable → Does not lead to optimal solution
  - Take the most valuable per unit → Does work
Fractional Knapsack Problem

- Greedy choice property: take the most valuable per weight unit
Fractional Knapsack Problem

- Greedy choice property: take the most valuable per weight unit
- Proof of optimality:
  - Given the set $S$ ordered by the value-per-weight, taking as much as possible $x_j$ from the item $j$ with the highest value-per-weight will lead to an optimal solution $X$
  - Assume we have another optimal solution $X`$ where we take less amount of item $j$, say $x_j` < x_j$.
  - Since $x_j` < x_j$, there must be another item $k$ which was taken with a higher amount in $X`$, i.e., $x_k` > x_k$.
  - We create another solution $X``$ by doing the following changes in $X`$
    - Reduce the amount of item $k$ by a value $z$ and increase the amount of item $j$ by a value $z$
    - The value of the new solution $V`` = V` + z \frac{v_j}{w_j} - z \frac{v_k}{w_k}$
      $= V` + z (\frac{v_j}{w_j} - \frac{v_k}{w_k}) \rightarrow \frac{v_j}{w_j} - \frac{v_k}{w_k} \geq 0 \rightarrow V`` \geq V`$
Fractional Knapsack Problem

- Optimal substructure
Fractional Knapsack Problem

- Optimal substructure
  - Given the problem $S$ with an optimal solution $X$ with value $V$, we want to prove that the solution $X' = X - x_j$ is optimal to the problem $S' = S - \{j\}$ and the knapsack capacity $W' = W - x_j$

- Proof by contradiction
  - Assume that $X'$ is not optimal to $S'$
  - There is another solution $X''$ to $S'$ that has a higher total value $V'' > V'$
  - Then $X'' \cup \{x_j\}$ is a solution to $S$ with value $V'' + x_j > V' + x_j > V$
  - Contradiction as $V$ is the optimal value
Fractional Knapsack Problem

Fknapsack (W, S, v’s, w’s) {
    - Sort S based on vi/wi value
    - rw = W
    - result = { }  
    - for each si in S
        if(wi <= rw)
            result = result U si
            rw = rw-wi
        else
            result = result U rw/wi * si
            rw = 0
    - return result
}
## Huffman Codes

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (in thousands)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>
Huffman Codes

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (in thousands)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

Prefix Codes: No code is allowed to be a prefix of another code

Prefix codes give optimal data compression
Huffman Codes

<table>
<thead>
<tr>
<th>Frequency (in thousands)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length codeword</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
</tr>
</tbody>
</table>

- Prefix Codes: No code is allowed to be a prefix of another code
  - Prefix codes give optimal data compression
- Example: Message ‘JAVA’ a = “0”, j = “11”, v = “10”
  - Encoded message “110100” Decoding “110100”
Huffman Codes

Prefix Codes: No code is allowed to be a prefix of another code

Prefix codes give optimal data compression

Example: Message ‘JAVA’ a = “0”, j = “11”, v = “10”
Encoded message “110100” Decoding “110100”

In the table:
Encoding with fixed-length needs 300K bits
Encoding with variable-length needs 224K bits

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (in thousands)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

Huffman Codes

Fixed-length tree

Variable-length tree
Huffman Codes

We need an algorithm to build the optimal variable-length tree.

Fixed-length tree

Variable-length tree

We need an algorithm to build the optimal variable-length tree.
Huffman Codes: Tree Construction

**HUFFMAN**(C)

1. \( n = |C| \)
2. \( Q = C \)
3. **for** \( i = 1 \) **to** \( n - 1 \)
4. allocate a new node \( z \)
5. \( z.left = x = \text{EXTRACT-MIN}(Q) \)
6. \( z.right = y = \text{EXTRACT-MIN}(Q) \)
7. \( z.freq = x.freq + y.freq \)
8. \( \text{INSERT}(Q, z) \)
9. **return** \( \text{EXTRACT-MIN}(Q) \)  // return the root of the tree
Huffman Codes: Tree Construction

f:5  e:9  c:12  b:13  d:16  a:45
Huffman Codes: Tree Construction

c:12  b:13

14

0 1

f:5  e:9
d:16  a:45
Huffman Codes: Tree Construction

14
0
f:5
1
e:9
d:16
0
c:12
1
b:13
25
0
a:45
1
Huffman Codes: Tree Construction

```
25
  0 1
 c:12 b:13

30
  0 1
 14
  0 1
 f:5 e:9

a:45
```
Huffman Codes: Tree Construction

```plaintext
a: 45

55
  /  \
/    \
25    30
   / \\
  /   \\
 c: 12 b: 13 14
d: 16
   / \\
  /   \\
 f: 5 e: 9
```

Huffman Codes: Tree Construction
Huffman Codes

- Details of optimal substructure and greedy choice property in the text book
Book Readings and Credits

› Book Readings:
   › 16.1 – 16.3

› Credits to:
   › Prof. Ahmed Eldawy notes