Greedy

Chapters 5 of Dasgupta et al.

Outline

• Activity selection
• Fractional knapsack
• Huffman encoding
• Later:
  – Dijkstra (single source shortest path)
  – Prim and Kruskal (minimum spanning tree)
Optimization problems

- A class of problems in which we are asked to find a set (or a sequence) of “items” that satisfy some constraints and simultaneously optimize (i.e., maximize or minimize) some objective function

Greedy method

- Typically applied to optimization problems, that is, problems that involve searching through a set of configurations to find one that minimizes/maximizes an objective function defined on these configurations
- Greedy strategy: at each step of the optimization procedure, choose the configuration which seems the best between all of those possible
Searching for the global minimum

### Greedy method

- There are problems for which the globally optimal solution can be found by making a series of locally optimal (greedy) choices
  - Make whatever choice seems **best** at the moment and then solve the sub-problem arising after the choice is made
  - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems
- The greedy strategy does **not** always lead to the global optimal solution
Searching for the global minimum

Elements of greedy strategy

- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property: a globally optimal solution can be reached by making a locally optimal choice
  - Optimal substructure: optimal solution to the problem consists of optimal solutions to sub-problems
Activity selection

(aka, “task scheduling”)

Activity Selection

• **Input**: A set of activities $S = \{a_1, \ldots, a_n\}$
• Each activity has start time and a finish time $a_i = (s_i, f_i)$
• Two activities are *conflicting* if and only if their interval overlap
• **Output**: a maximum-size subset of non-conflicting activities
Activity Selection

- Here are a set of start and finish times

<table>
<thead>
<tr>
<th>i</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>$s_i$</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>

- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_4, a_9, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$
"Greedy" Strategies

1. Longest first
2. Shortest first
3. Early start first
4. Early finish first
5. None of the above
Early Finish Greedy strategy

- Sort the activities by finish time
- Schedule the first activity
- Then, schedule the next activity (in sorted list) which starts after previous activity finishes (first non-conflicting task)
- Repeat until no more activities
Activity selection in Python

```python
def greedy_activity_selection(A):
    A.sort(key=itemgetter(1))  # Remark: sort A by finish time
    result = [A[0]]  # Remark: first activity in the solution
    i = 0
    for j in range(1, len(A)):
        if A[j][0] >= A[i][1]:  # Remark: start[j] >= finish[i]
            result.append(A[j])
            i = j
    return result
```

Time complexity? $O(n \log n)$ to sort, the rest is linear.

Why it is Greedy?

• Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled

• The greedy choice is the one that maximizes the amount of unscheduled time remaining
Correctness (optimality)

- We will show that
  - the problem has the optimal substructure property
  - the algorithm satisfies the greedy-choice property
- Thus, the algorithm always finds the optimal solution

Greedy-Choice Property

- We want to show there is an optimal solution that begins with a greedy choice (i.e., with the first activity, which has the earliest finish time)
Greedy-Choice Property

• Suppose $A \subseteq S$ is an optimal solution
  – Order the activities in $A$ by finish time
    Let $k$ be the first activity in $A$
      • If $k = 1$, the schedule $A$ begins with a greedy choice
      • If $k \neq 1$, show that there is another optimal solution $B$ that begins with the greedy choice (activity 1)
  – Let $B = (A - \{k\}) \cup \{1\}$
    • Activities in $B$ are non-conflicting because activities in $A$ are non-conflicting, $k$ is the first activity to finish and $f_1 \leq f_k$
    • $B$ has the same number of activities as $A$ thus, $B$ is optimal

Optimal Substructure

• Once the greedy choice of the first activity is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in $S$ that are compatible with the first activity
  – Optimal Substructure: if $A$ is optimal to $S$ and $A$ contains task $I$, then $A' = A - \{I\}$ is optimal to $S' = \{i$ in $S: s_i \geq f_I\}$
  – Why? If we could find a solution $B'$ to $S'$ with more activities than $A'$, adding activity 1 to $B'$ would yield a solution $B$ to $S$ with more activities than $A$ contradicting the optimality of $A$
Optimal Substructure

• After each greedy choice is made, we are left with an optimization problem of the same form as the original problem.

• By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.

Fractional Knapsack
Fractional Knapsack

- Given a set $S$ of $n$ items, such that each item $i$ has a positive benefit $b_i$ and a positive weight $w_i$; the size of the knapsack $W$
- The problem is to find the amount $x_i$ of each item $i$ which maximizes the total benefit

$$\sum_i b_i \left( \frac{x_i}{w_i} \right)$$

under the condition that $0 \leq x_i \leq w_i$ and $\sum_i x_i \leq W$

Fractional Knapsack - Example
Fractional Knapsack in Python

```python
def fractional_knapsack(S, W):
    v = []
    for item in S:
        value = float(item[1]) / float(item[0])
        v.append((value, item[1], item[2]))
    v.sort(key=itemgetter(0))
    w, result = 0, []
    while w < W:
        high = v[-1]
        v.pop()
        a = min(high[1], W-w)
        w += a
        result.append((a, high[2]))
    return result
```

Remark: sort \( v \) by value = benefit/weight

Remark: select and remove the highest value (\( \text{high} \))

Remark: \( a \) is how much item \( \text{high} \) we took

Fractional Knapsack

- Time complexity is \( O(n \log n) \)
- Fact: Greedy strategy is optimal for the fractional knapsack problem
- Proof: We will show that the problem has the optimal substructure and the algorithm satisfies the greedy-choice property
Greedy Choice

Items (sorted by $b_i/w_i$)  1  2  3  ...  $j$  ...  $n$

"Optimal" solution:  $x_1$  $x_2$  $x_3$  ...  $x_j$  ...  $x_n$

Greedy solution:  $x_1'$  $x_2'$  $x_3'$  ...  $x_j'$  ...  $x_n'$

• We want to prove that taking as much as possible from item $i$ is optimal
• Let us assume that is not the case, and in the optimal solution $x_i < x_i'$
• Because we taking more of item $i$ in the greedy solution, we have to decrease the quantity taken of some other item $j$
• Therefore, in the greedy solution $x_j$ is decreased by $(x_i' - x_i)$
• In the greedy solution, we gain $(x_i' - x_i) b_i / w_i$, and we lose $(x_i' - x_i) b_j / w_j$

$$\frac{b_i}{w_i} \geq \frac{b_j}{w_j} \quad \text{True, since } x_i \text{ had the best benefit/weight ratio}$$

Optimal substructure

Items:  1  2  3  ...  $j$  ...  $n$

Solution $U$:  $x_1$  $x_2$  $x_3$  ...  $x_j$  ...  $x_n$

Solution $U'$:  0  $x_2'$  $x_3'$  ...  $x_j'$  ...  $x_n'$

• $(S,W)$ is the original problem, assume $U$ is optimal for $(S,W)$
• $S'$ is the sub-problem \( \{2,3,\ldots,n\} \)
• $U$ contains the greedy choice $x_i$
• Prove that $U'$ is optimal for $(S',W-x_i)$
  where $x_i' = x_i$ for all $i > 1$
• By contradiction: if $U''$ was not optimal, then $U'''$ exists such that

$$\sum_{1 \leq i \leq n} x_i' (b_i / w_i) > \sum_{2 \leq i \leq n} x_i (b_i / w_i)$$

$$\sum_{1 \leq i \leq n} x_i (b_i / w_i) = \sum_{2 \leq i \leq n} x_i (b_i / w_i) + x_i (b_i / w_i) < \sum_{2 \leq i \leq n} x_i' (b_i / w_i) + x_i (b_i / w_i)$$

which means that $U$ was not optimal for $(S,W)$ → contradiction
Data Compression

• Text files are usually stored by representing each character with an 8-bit ASCII code
• The ASCII encoding is an example of fixed-length encoding, where each character is represented with the same number of bits
• In order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others
Data Compression

• **Variable-length** encoding uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters

• Huffman coding (section 5.2)

File Compression: Example

• An example
  – text: “java”
  – encoding: a = “0”, j = “11”, v = “10”
  – encoded text: 110100 (6 bits)

• How to decode in the case of ambiguity?
  – encoding: a = “0”, j = “11”, v = “00”
  – encoded text: 110000 (6 bits)
  – could be “java”, or “jvv”, or “jaaaa”, or …
Encoding

• To prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule: no code is a prefix of another

• Example
  – a = “0”, j = “11”, v = “10” satisfies the prefix rule
  – a = “0”, j = “11”, v= “00” does not satisfy the prefix rule (the code of ‘a’ is a prefix of the codes of ‘v’)

Trie

• We use an encoding trie to satisfy this prefix rule
  – the characters are stored at the external nodes
  – a left child (edge) means 0
  – a right child (edge) means 1
Example of Decoding

- encoded text: 
  01011011010001010010111011010
- text: ABRACADABRA (11 bytes=88 bits)

Data Compression

- Problem: We want the encoded text as short as possible
- Example: ABRACADABRA
  01011011010001010010111011010 29 bits
Data Compression

- Example 2: ABRACADABRA
  001011000100001100101100 24 bits

Optimization problem

- Given a character $c$ in the alphabet $\Sigma$
  - let $f(c)$ be the frequency of $c$ in the file
  - let $d_T(c)$ be the depth of $c$ in the tree = the length of the codeword

- We want to minimize the number of bits required to encode the file, that is

$$\min_{\text{binary trees } T \text{ with } |\Sigma| \text{ leaves } c \in \Sigma} \sum f(c)d_T(c)$$
Huffman Encoding: Example

**Step 0**

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

**Step 1**

```
ABRACADABRA
```

```
5 2 2 1 1
```

Huffman Encoding: Example

**Step 2**

```
5
A
```

```
2
B
```

```
2
R
```

```
2
C
```

```
1
D
```

**Step 3**

```
5
A
```

```
4
B
```

```
2
R
```

```
2
C
```

```
1
D
```

Stefano Lonardi, UCR
Huffman Encoding: Example

Step 4

Step 5

Final Huffman Trie

A B R A C A D A B R A
0 100 101 0 110 0 111 0 100 101 0 (23 bits)
Another Example

<table>
<thead>
<tr>
<th>Step 0</th>
<th>ABRACADABRA character frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Step 1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Another Example

<table>
<thead>
<tr>
<th>Step 3</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 4

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Stefano Lonardi, UCR
Another Example

Step 5

Step 6

Final Trie

A B R A C   A D   A B R A
0 10 110 0 1110 0 1111 0 10 110 0 (23 bits)
Priority queue

- Use a priority queue for storing the nodes
- Priority queue is a queue ordered by priority (heap)
- For our application, priority = frequency
- If there are $k$ elements in the queue:
  - Extracting the lowest priority is $O(\log k)$
  - Inserting takes $O(\log k)$

Huffman algorithm in Python

```python
def makeHuffTree(t):
    heapq.heapify(t)
    while len(t) > 1:
        L, R = heapq.heappop(t), heapq.heappop(t)
        parent = (L[0] + R[0], L, R)
        heapq.heappush(t, parent)
    return t[0]

def printHuffTree(t, prefix = ''):
    if len(t) == 2:
        print(t[1], prefix)
    else:
        printHuffTree(t[1], prefix + '0')
        printHuffTree(t[2], prefix + '1')
```

Remark: transforms list $t$ in a heap in linear time
Remark: returns the tree represented a nested tuple
Huffman Algorithm

- Running time for a text of length \( n \) with \( k \) distinct characters: \( O(n + k \log k) \)
- If we assume \( k \) to be a constant (i.e., not a function of \( n \)) then the algorithm runs in \( O(n) \) time
- Fact: Using a Huffman encoding trie, the encoded text has minimal length
- Proof: omitted

Greedy method: summary

- Task scheduling
- Fractional knapsack
- Huffman encoding (section 5.2)

- Other greedy algorithms that will be covered later: Prim (section 5.1.5), Kruskal (section 5.1.3) and Dijkstra (section 4.4)