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 configuration
• 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 results from 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 $\{a_1, a_9, a_{11}\}$ 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 first activity (activity 1)
- Remove all activities that conflict with 1
- Recurse on the remaining 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.

Greedy

- Goal: build a solution in steps, never making a “mistake” --- maintain the invariant that the partial solution so far is always extendible to an optimal solution.

- Choosing activity 1 for the first job maximizes the set of remaining (possible, non-conflicting) jobs.
Correctness (optimality)

- **Greedy choice property:**
  First choice is consistent with some opt’l soln
- **Optimal substructure property:**
  After this first choice, to solve the entire problem optimally, it is enough to solve the remaining sub-problem optimally.

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 \( f_i \leq f_k \)
    - \( B \) has the same number of activities as \( A \) thus, \( B \) is optimal

Optimal Substructure

- After the greedy choice of the first activity, 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. Formally:

  Remaining sub-problem is \( S' = \{i \in S: s_i \geq f_1\} \).

\[ A' \text{ is an optimal solution for } S' \]
\[ \text{if and only if} \]
\[ A' \cup \{1\} \text{ is an optimal solution for } S. \]
Optimal Substructure

Remaining sub-problem is \( S' = \{ i \text{ in } S : s_i \geq f_i \} \).

**Claim.** \( A' \) is an optimal solution for \( S' \)

if and only if

\[ A' \cup \{1\} \text{ is an optimal solution for } S. \]

**Proof.** (\( \Rightarrow \) direction)

Let \( A' \) be any optimal solution for \( S' \).

Then \( A' \cup \{1\} \) is a solution for \( S \). (why?)

If it is not optimal for \( S \), then (by greedy choice)

there is a larger solution \( B' \cup \{1\} \) for \( S \). But then \( B' \)

is a solution for \( S' \) (why?), and \( B' \) is larger than \( A' \).

(We leave the \(<= \) direction as an exercise.) QED

---

**Claim.** *Greedy is optimal*

**Proof.**

\[
\begin{align*}
\text{greedy}(S) & = \{1\} \cup \text{greedy}(S') & \text{- defn of greedy} \\
& = \{1\} \cup \text{opt}(S') & \text{- induction on } |S| \\
& = \text{opt}(S) & \text{- optimal substructure}
\end{align*}
\]

base case:

\[
\text{greedy(\{}\{} = \{}\{} = \text{opt(\{}\})
\]
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

<table>
<thead>
<tr>
<th>Item 1</th>
<th>Item 2</th>
<th>Item 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₁=10</td>
<td>w₂=20</td>
<td>w₃=30</td>
</tr>
<tr>
<td>b₁=$60</td>
<td>b₂=$100</td>
<td>b₃=$120</td>
</tr>
<tr>
<td>$6/pound</td>
<td>$5/pound</td>
<td>$4/pound</td>
</tr>
</tbody>
</table>

W=50

$240

Remark: sort v by value = benefit/weight

Remark: select and remove the highest value (high)

Remark: a is how much item high we took

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
```

35

36
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 property holds

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

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

- While $x_j < \min(w_j, W)$:
- For some small $d > 0$, increase $x_j$ by $d$, decrease some $x_i$ by $d$.
- Increase in total benefit, $d \left(b_j/w_j - b_i/w_i\right)$, is non-negative.

- Stop when $x_j = \min(w_j, W)$.

- Yields equally good solution with $x_j = \min(w_j, W)$. 
Optimal substructure property

- Let $x_j = \min(w_j, W)$ be the greedy choice for the first step.
- $(S, W)$ is the original problem,
- $(S', W')$ is the sub-problem, where $S' = \{2, 3, \ldots , n\}$, $W' = W - x_j$

\[
\text{items: } 1 \quad 2 \quad 3 \quad \ldots \quad n \\
\text{soln for } (S', W'):\quad - \quad x_2 \quad x_3 \quad \ldots \quad x_n \\
\text{soln for } (S, W) : \quad x_j \quad ? \quad ? \quad \ldots \quad ?
\]

$x_2, x_3, \ldots x_n$ is an optimal solution to $(S', W')$ if and only if $x_1, x_2, x_3, \ldots x_n$ is an optimal solution to $(S, W)$

Proof. Left as an exercise!

Huffman codes
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 “jaaa”, 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’)

43 44
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: 01011011010000101001011011010
- text: ABRACADABRA (11 bytes=88 bits)
Data Compression

• **Problem**: We want the encoded text as short as possible

• **Example**: ABRACADABRA
  0101101101000010100101101101029 bits

```
D
B
C
R
0 1
0 0
0 1
0 1
```

Data Compression

• **Example2**: ABRACADABRA
  0010110001000110010110024 bits

```
B
R
A
D
0 1
0 0
0 1
0 1
```
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

$$\min_{\text{binary trees } T} \sum_{c \in \Sigma} f(c)d_T(c)$$

Huffman Encoding: Example

<table>
<thead>
<tr>
<th>Step 0</th>
<th>ABRACADABRA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>character frequency</td>
</tr>
<tr>
<td></td>
<td>5 2 2 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Huffman Encoding: Example

Step 2

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

Step 3

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

Step 4

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

Step 5

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

A B R A C A D A B R A
0 1 0 0 1 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0  (23 bits)

Another Example

<table>
<thead>
<tr>
<th>Step</th>
<th>ABRACADABRA</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 0</td>
<td>character frequency</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Step 1</td>
<td>5 A 2 B 2 R 2 C 1 D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2</td>
<td>5 A 2 B 2 R 4 C 1 D 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

Step 3

Step 4

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)  # Remark: transforms list t in a heap in linear time
    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]  # Remark: returns the tree represented a nested tuple

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

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)