COMP 1201 Algorithmics

Greedy Strategy

Hikmat Farhat

January 29, 2024

Introduction

- Greedy algorithms are algorithms that make **locally** optimal choices at each step.
- The hope is that such choices will lead to a **globally** optimal solution.
- That is not always the case but when it is, greedy algorithms are very efficient.
- We have already seen some examples of greedy algorithms: Dijkstra's, Prim's, Kruskal's.
- We will look at some more examples of greedy algorithms.

Job Scheduling

- Assume we have n jobs, each with weight w_i and length l_i, $1 \leq i \leq n$.
- The jobs share some resource (i.e. CPU) so they must be run sequentially.
- If we run the jobs in the order $1, 2, 3, \ldots$ then job i has completion time $c_i = \sum_{k=1}^{i} l_k$.
- our goal is to minimize the quantity $f = \sum_{k=1}^{n} w_k \cdot c_k$
- In particular, we would like to have a greedy algorithm that minimizes f.
- To do that we start by looking at special cases

Special case 1

- In this special case we assume that all jobs have the same weight then

$$\begin{aligned} f &= w(l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + \cdots + l_n)) \\ &= w(n \cdot l_1 + (n-1) \cdot l_2 + (n-2) \cdot l_3 + \ldots + 1 \cdot l_n) \end{aligned}$$

- Clearly f will be minimized if we choose $l_1 \leq l_2 \leq \ldots \leq l_n$
- This situation occurs often enough to warrant its own name:
- Shortest job (task) first.

Special case 2

- The second special case is when all jobs have the same length l then

$$f = l(w_1 + 2 \cdot w_2 + 3 \cdot w_3 + \ldots n \cdot w_n)$$

- Clearly, f will be minimized if $w_1 \geq w_2 \geq w_3 \geq \ldots \geq w_n$
- it is clear that f decreases by choosing the **largest** w or the **smallest** l first.
- Can we infer the general case from these two special cases?

The general case

- It can be shown that choosing, among the remaining tasks, the one with **largest** ratio w_i/l_i first, leads to the optimal solution.

---

input: An array A of n pairs (w_i, l_i)
for $i \leftarrow 1$ **to** n **do**
  $B[i] \leftarrow A(w_i/l_i, i)$;
$B \leftarrow$`sort(`B`)`;
foreach $(w/l, k) \in B$ **do**
  `run(`k`)`;

- The algorithm runs in $O(n \log n)$ time.

Interval Scheduling

- Consider a set of n intervals (s, e) where s and e are the starting and ending time respectively.
- We would like to choose a non-overlapping subset of those intervals such that the total number of selected intervals is maximum.
- For example, consider the intervals $\{(1, 5), (2, 7), (5, 8)\}$. The largest subset of non-overlapping intervals is $\{(1, 5), (5, 8)\}$.

We are looking for a greedy solution to this optimization problem. What property of the intervals should the greedy method select? There are many options:

1. shortest interval first
2. The interval with the smallest starting time
3. The interval with the smallest number of overlaps
4. etc...

Shortest iterval first

Figure: Shortest interval counterexample

Earliest starting time first

Figure: counterexample for earliest starting time first

Smallest overlap first

Figure: Smallest overlap first

Greedy Solution

The greedy solution consists of choose the next compatible interval with the smallest finishing time. We build a min-heap based on finishing times. Let I be the set of intervals and T the desired solution

$Q \leftarrow I$
$T \leftarrow \emptyset$
$last \leftarrow -1$
while $Q \neq \emptyset$ **do**
    $(s, f) \leftarrow$ `Delete-Min(Q)`
    if $s \geq last$ **then**
        $T \leftarrow T \cup \{(s, f)\}$
        $last \leftarrow f$

Fractional Knapsack

- Given n items having value $v_1, \dots, v_n$ and weights $w_1, \dots, w_n$ and a knapsack of size W
- We need to maximize

$$\sum_{i=1}^{n} x_i \cdot v_i$$

- Subject to the condition

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

- Where $0 \leq x_i \leq 1$ is a faction of item i that is used.

Greedy Solution

- A greedy solution is obtained by adding repeatedly items with biggest ration v/w until the next item does not "fit" in knpacksack so we add a fraction of it.

Symbol encoding

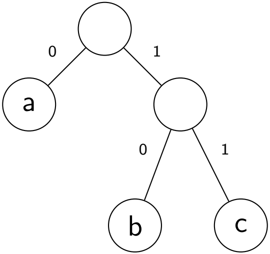
- Suppose we have an alphabet $S = \{s_1, \ldots, s_k\}$ of size k and we want to encode a message M consisting of n symbols from S.
- A trivial encoding is to use $n \cdot \lceil \log_2 k \rceil$ bits to encode the message.
- For example, $S = \{a, b, c\}$ and $M = abaabc$ then we can assign $a = 000, b = 001$ and $c = 010$ and thus
- $M = 000\ 001\ 000\ 000\ 001\ 010$ (space added for clarity)
- For a total of 18 bits.

Can we do better?

- Is it possible to encode the message using less than 18 bits?
- The answer is yes. We can use a variable length encoding.
- The trick is to use shorter codes for the most frequent symbols and longer codes for the least frequent symbols.
- But then we lose the "boundary" between symbols and creates ambiguity.
- Solution is to use **prefix** codes.
- This means there is no code that is a prefix of another code.
- For example if $a = 0$ and $b = 01$ then a is a prefix of b and this is not allowed.

Example prefix code

- Consider the previous example $S = \{a, b, c\}$ and $M = abaabc$ with $a = 0$, $b = 10$, $c = 11$.$M = 0\ 10\ 0\ 0\ 10\ 11$ for a total of 9 bits instead of 18.
- Note that prefix code can be represented by a binary tree.



Huffman Coding

- Formally, given an alphabet $S = \{s_1, \ldots, s_k\}$ with frequencies $f_1, \ldots, f_k$ we want to find a prefix code such that the average number of bits per symbol is minimized.
- Using the prefix tree representation, the average number of bits is given by

$$A = \sum_{i=1}^{k} f_i \cdot d(s_i)$$

  where $d(s_i)$ is the depth of the leaf corresponding to s_i.
- Hauffman coding is a greedy algorithm that constructs the prefix tree "bottom up".

- First a prefix code is equivalent to a binary tree so once we build the optimal binary tree then we "read off" the encoding.
- The basic idea of HC is to build the optimal tree recursively in a greedy manner
  1. The optimal tree T for k symbols is obtained by constructing the optimal tree T' for $k-1$ symbols where T' is the same as T except replacing the two nodes with the smallest frequencies in T, x and y by a single node having the sum of the frequencies: $f_w = f_x + f_y$

$$T' = T - \{x, y\} \cup \{w\}$$

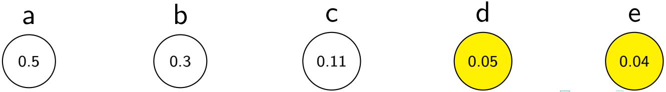
The optimal tree has the following properties:

- It is full. Suppose that y is a single child of node w. By replacing w by y we obtain a "better" tree
- For any two leaves x, y if $f_x > f_y$ then $d_x < d_y$. This can be shown by an exchange argument.

Note that there could be many optimal trees. Let x, y be the symbols with the least frequencies then there exists an optimal tree in which x, y are siblings.

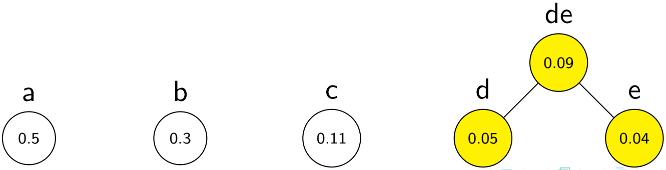
Example

(0.5,'a'), (0.3, 'b'), (0.11, 'c'), (0.05, 'd'),(0.04, 'e')



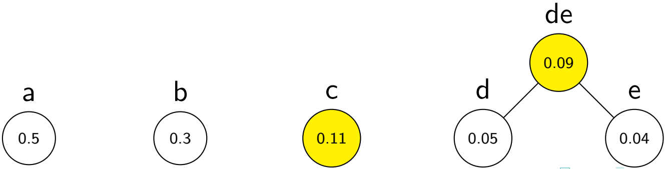
Example

(0.5,'a'), (0.3, 'b'), (0.11, 'c'), (0.09, 'de')



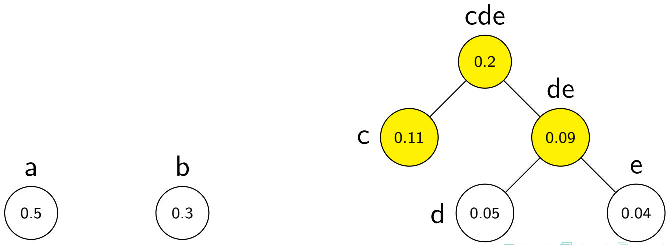
Example

(0.5,'a'), (0.3, 'b'), (0.11, 'c'), (0.09, 'de')



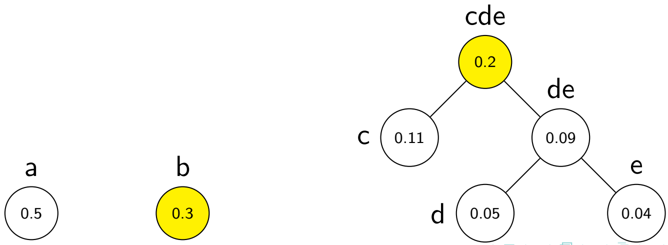
Example

(0.5,'a'), (0.3, 'b'), (0.2, 'cde')



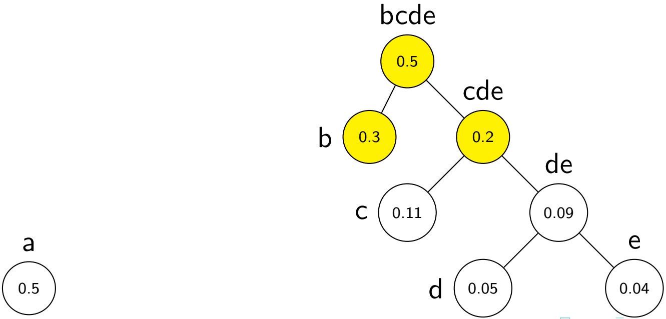
Example

(0.5,'a'), (0.3, 'b'), (0.2, 'cde')



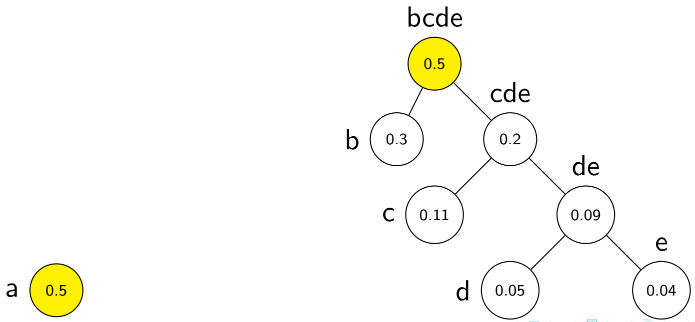
Example

(0.5,'a'), (0.5, 'bcde')



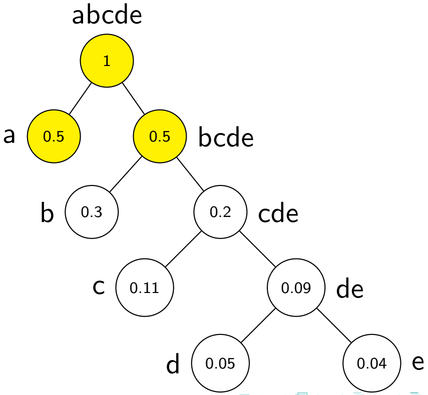
Example

(0.5,'a'), (0.5, 'bcde')



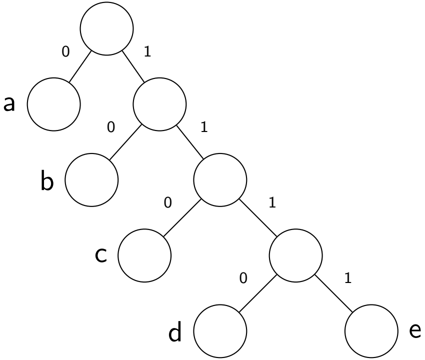
Example

(1,'abcde')



Example

a	0
b	10
c	110
d	1110
e	1111



Huffman coding: Python Code

```
import queue
class Node(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
    def children(self):
        return((self.left, self.right))
freq = [
    (25,'a'), (24, 'b'), (28, 'c'), (18, 'd'),(5, 'e')]

def create_tree(frequencies):
    p = queue.PriorityQueue()
    for value in frequencies:
        p.put(value)
    while p.qsize() > 1:
        l, r = p.get(), p.get()
```

```
# Recursively walk the tree down to the leaves,
#   assigning a code value to each symbol

def walk_tree(node, prefix=""):

    if isinstance(node[1],Node):
        l,r=node[1].children()
        walk_tree(l,prefix+"1")
        walk_tree(r,prefix+"0")
    else:
        code[node[1]]=prefix

node = create_tree(freq)
code={}
walk_tree(node)
```

$$
\begin{aligned}
A(T) &= \sum_{u \in S} f_u \cdot depth_T(u) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + f_x \cdot depth_T(x) + f_y \cdot depth_T(y) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + (f_x + f_y) \cdot (depth_{T'}(w) + 1) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + (f_w) \cdot (depth_{T'}(w) + 1) \\
&= \sum_{u \in S'} f_u \cdot depth_{T'}(u) + f_w \\
&= A(T') + f_w
\end{aligned}
$$