CS2223: Algorithms, D-Term 2008

Homework 4 Written Solutions

Prepared by Andrey Sklyar with some help from Yaobin Tang

Worcester Polytechnic Institute
4/8/2008

Table of Contents

Table of Figures

Problem 1 – Packing Coffee

Choosing a Selection Criterion

Greedy Selection by Weight:

(1) (T5:50, T4:40, T3:10): 60+40+2.2*10=122.

Greedy Selection by Cost:

(2) (T3:30, T5:50, T4:20): 66+60+1*20=146.

Greedy Selection by Cost per Pound:

(3) (T3:30, T1:10, T2:20, T5:40): 66+20+30+1.2*40=164.

Best of the three selection criteria :

(4) Cost per pound produces the best result. As we will show in the proof of correctness of the algorithm below, a greedy approach based on cost-per-pound yields optimal solutions in general, not just in this example.

Pseudo Code (part a)

High level description:

Take as much of the most expensive coffee as you can until you fill up the truck.

This will require:
Determining the most expensive coffee will use the priority queue.
Need to keep track of either how much room is left or how full the truck is.

Input:

W: Maximum number of coffee pounds that the delivery van can carry.
n: Number of different types of coffee that can be shipped.
weight: a 1-dimensional array of size n of integers with
cost_per_pound: a 1-dimensional array of size n of reals
such that for each i, 1 ≤ i ≤ n:
weight[i] = maximum number of pounds of coffee type Ti that can be shipped
cost_per_pound[i] = cost of one pound of type Ti coffee

Output:

ship: 1-dimensional array of size n of reals such that for each i, 1 ≤ i ≤ n:
ship[i]= amount in pounds of coffee type Ti included in the shipping
satisfying:
ship[i] ≤ weight[i],
$\Sigma^{n}_{i=1}$ ship[i]≤ W, and
$\Sigma^{n}_{i=1}$ ship[i]*cost_per_pound[i] is as large as possible

Algorithm Pseudo Code:

```
ship[] represents the fraction of coffee pounds to load in the van.
Set ship[i] = 0 for 1 < i <= n (no coffee loaded yet)

Initialize a priority queue PQ of size n for storing the coffee types, with the cost_per_pound[i] as the
keys.  The most expensive coffee type will be kept at the front of the queue.

For each type of coffee t, 1 < t <= n
    Insert t into PQ with key value cost_per_pound[t]
EndFor

Extract the most expensive coffee type from PQ and store it in a variable t
While weight[t] <= W and PQ is not empty
    Set ship[t] = weight[t]
    Decrease the van's capacity W by weight[t]
    Extract the most expensive coffee type from PQ and store it in a variable t
EndWhile

Set ship[t] = W, the remaining room in the truck (that is less than the total weight of type t coffee that
we have)
EndAlgorithm
```

More Java-Like Algorithm Pseudo Code:

```
real ship[n];   //fraction of coffee pounds to ship

for (i = 1; i <= n+1; i++)
            ship[i] = 0;         // initially there is no coffee in the van

// Initialize a priority queue PQ of size n for storing the coffee types, with
// the weights as the keys, with the most expensive coffee on top.
CoffeePriorityQueue pq = new CoffeePriorityQueue (n);
for (int t = 1; t <= n+1; i++)
            pq.insert(t, cost_per_pound[t]);   // add the coffee types, with weights as keys

int t = pq.extractMax();  // get the most expensive coffee
while(weight[t] <= W && !(pq.isEmpty())) {
            ship[t] = weight[t]
            W -= weight[t]
            int t = pq.extractMax();
}

ship[t] = W
// the remaining room in the truck (that is less than the total weight of type t coffee that we have)
}
```

Alternative Using Sorting

```
ship[] represents the fraction of coffee pounds to load in the van.
Set ship[i] = 0 for 1 < i <= n (no coffee loaded yet)

type[] represents the coffee types.
Initialize type[i] = i for 1 < i <=n

Sort type[] in decreasing order by cost_per_pound[i]

Let t be the current coffee type to load
t = 1
While weight[t] <= W and PQ is not empty
            Set ship[t] = weight[t]
            Decrease the van's capacity W by weight[t]
            t = t+1
EndWhile

Set ship[t] = W, the remaining room in the truck (that is less than the total weight of type t coffee that
we have)
EndAlgorithm
```

Proof of Correctness (part b)

Intuition

This algorithm puts all the most expensive coffees into the truck. If we do anything else, we will decrease the value of the payload.

Proof

Consider a solution ship[] that does not have as much of the most expensive coffees as it can. We can always improve such a shipment by replacing an amount of a cheaper type of coffee by the same amount of more expensive coffee. If the cheaper coffee type has a price c per pound, and the more expensive coffee has a price e per pound, replacing w pounds of c priced coffee with w pounds of e priced coffee will make the payload e*w – c*w = (e-c)*w more expensive. Since e > c, this improves the payload. Therefore, any payload that does not have as much of the most expensive coffees as it can cannot be as good as one that does, thus proving the optimality of the algorithm.

Complexity Analysis (part c)

The analysis is parameterized on n, the number of coffee types.

Priority Queue

Initializing the shipment array takes O(n).

Performing n insertions a priority queue takes O(n log n), since each insertion takes O(log n) and insertions are performed n times.

The while loop can iterate at most n times (until we run out of coffee types). Setting the shipment weight and decreasing the capacity both take constant time. Extracting the maximum element out of a priority queue, on the other hand, takes O(log n) time. Since this extraction is done inside the while loop, the whole loop has complexity O(n log n).

The final assignment takes constant time.

The total running time of the algorithm is then:

T(n) = [Initialize Ship] + [Initialize PQ] + [Fill Truck] + [Final assign.] = O(n) + O(n log n) + O(n log n) + O(1)
T(n) = O(n log n)

Sorting

An alternate way to solve this problem would be to use a sorted array of the coffee types instead of a priority queue. Using Merge Sort (to studied in class soon) or Heap Sort, the sort can be done in O(n log n). Then the packing stage becomes O(n), since looking at an element in an array takes constant time. This leads to a total running time of:

T(n) = [Initialize Ship] + [Initialize Types] + [Sort Types] + [Fill Truck] + [Final assign.]
T(n) = O(n) + O(n) + O(n log n) + O(n) + O(1)
T(n) = O(n log n)

This is exactly the same complexity as using a priority queue.

Problem 2 – Huffman Codes

Generating an Encoding (part a)

The algorithm for generating Huffman codes repeatedly merges the two least frequently occurring letters in the alphabet together into one meta-letter. After repeating this merge n-1 times, where n is the number of letters, all that is left is one large meta-letter. This process is represented in the table below.

Figure 1: Making Meta-Letters

🟥 Lowest Frequencies (will be merged)

Intial Configuration					
alphabet	a	e	i	o	u
frequency	0.5	0.25	0.125	0.0625	0.0625

Step 1				
alphabet	a	e	i	ou
frequency	0.5	0.25	0.125	0.125

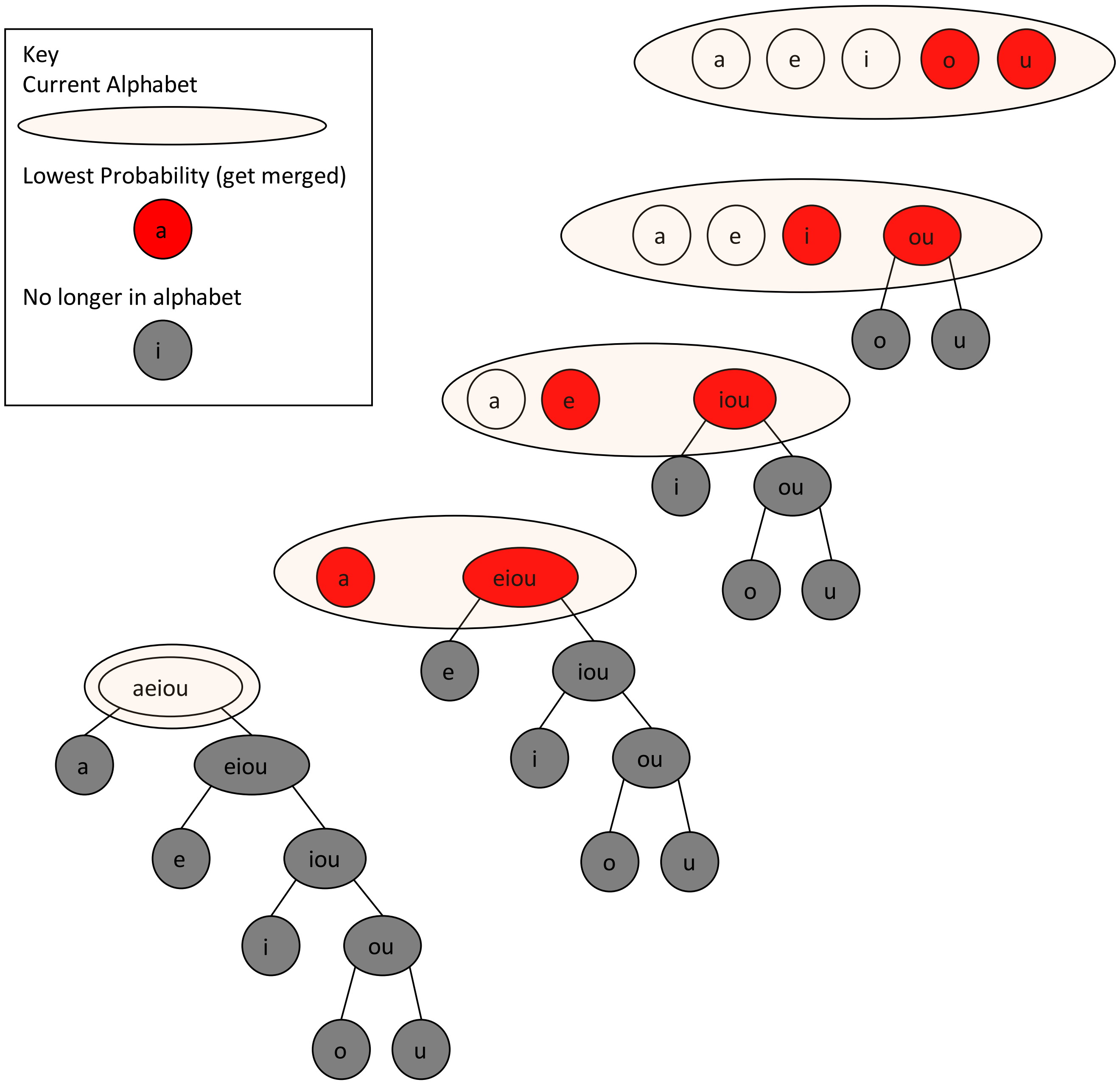
Step 2			
alphabet	a	e	iou
frequency	0.5	0.25	0.25

Step 3		
alphabet	a	eiou
frequency	0.5	0.5

Step 4	
alphabet	aeiou
frequency	1

This merging process builds a tree, beginning with all nodes being in their own separate trees. As nodes get merged, they form small trees, which get merged to form bigger trees. The final tree that results provides the Huffman Encoding for the letters in the given alphabet.

Figure 2: Building Huffman Trees



To get the code of each letter in the alphabet, traverse the tree from the root to the leaves. Each time you go left, put down a 1. Each time you go right, put down a 0. For the above graph, the resulting encoding is:

a	e	i	o	u
0	10	110	1110	1111

Figure 3: Alphabet Encoding

Since the encodings traverse the tree all the way to the leaves, there is no possible way that any encoding could be the prefix of any other encoding. The next section provides an example of this decoding process.

A Particular Encoding (part b)

The encoding of "auuoi" using the above tree is

a	u	u	o	i
0	1111	1111	1110	110

Figure 4: Encoding of a Particular Word

This will actually just come in one long string:
0111111111110110

In order to reconstruct the original word from this long string, start at the beginning of the string and at the beginning of the Huffman Encoding Tree. For each number (0 or 1) you read from the encoded string, take that branch in the tree. When you hit a leaf in the tree, you know you found a letter! Write it down and start back at the root with the next number on the input string.

Figure 5: Decoding Huffman Codes

Input String	Accumulated Encoding	Letter
0	0	a
1	1	
1	11	
1	111	
1	1111	u
1	1	
1	11	
1	111	
1	1111	u
1	1	
1	11	
1	111	
0	1110	o
1	1	
1	11	
0	110	i

Encoded Length (part c)

The length of all of the encoded versions of a letter L is going to be the length of the encoded letter times the total number of its occurrences in the string. The number of occurrences can be calculated using the frequency.

In a string of length m, with each letter having frequency F[i] and encoded length E[i], the number of bits used to write those letters is m*F[i]*E[i]. The total number of bits used is then the sum over all the letters. If there are n letters, then

$$TotalLetterLength(m, f, e) = m * f * e$$

$$TotalLength(m, F[\,], E[\,]) = \sum_{i=1}^{n} TotalLetterLength(m, F[i], E[i]) = \sum_{i=1}^{n} m * F[i] * E[i]$$

This can be simplified one step further to get:

$$TotalLength(m, F[\,], E[\,]) = m * \sum_{i=1}^{n} F[i] * E[i]$$

Figure 6: Encoded Lengths

Plain Text Length	Letter	Frequency	Encoding	Encoding Length	Encoded Text Letter Length	ASCII Encoding*
1000000	a	0.5	0	1	500000	3500000
1000000	e	0.25	10	2	500000	1750000
1000000	i	0.125	110	3	375000	875000
1000000	o	0.0625	1110	4	250000	437500
1000000	u	0.0625	1111	4	250000	437500
				Total Length	1875000	7000000
					Compression	0.2678

* 7 bits per Letter