Assignment no. 5

**Title:** Huffman encoding concurrently on many core GPU

**Theory:**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See this for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

* Build a Huffman Tree from input characters.
* Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

1. Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
2. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of the frequency field is used to compare two nodes in a min heap. Initially, the least frequent character is at root)
3. Extract two nodes with the minimum frequency from the min heap.
4. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
5. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

**Code:**

from heapq import heappush, heappop, heapify

from collections import defaultdict

import multiprocessing

cnt1=0

cnt3=0

f = open('demofile3.txt', 'w+')

def encode(symb2freq):

"""Huffman encode the given dict mapping symbols to weights"""

heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]

heapify(heap)

while len(heap) > 1:

lo = heappop(heap)

hi = heappop(heap)

for pair in lo[1:]:

pair[1] = '0' + pair[1]

for pair in hi[1:]:

pair[1] = '1' + pair[1]

heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

cnt1=len(heap)

return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

txt = "this is an example for huffman encoding eeeeerfdkb"

symb2freq = defaultdict(int)

for ch in txt:

symb2freq[ch] += 1

# in Python 3.1+:

# symb2freq = collections.Counter(txt)

def write\_log(x,y):

#with open("demofile3.txt",'w',encoding = 'utf-8') as f:

#f = open('demofile3.txt', 'w')

#f.write(huff)

cnt2=0

for p in x:

if(cnt2<=cnt1):

f.write('%s\t%s\t%s\n' % (p, symb2freq[p], y[cnt2]))

cnt2+=1

f.close()

print(he)

#print(huff)

def main():

huff = encode(symb2freq)

for p in symb2freq:s

p1 = multiprocessing.Process(target=encode, args=(p))

p1.start()

for p in huff:

p2 = multiprocessing.Process(target=write\_log, args=(p[0],p[1]))

p2.start()

# starting process 2

# wait until process 1 is finished

p1.join()

# wait until process 2 is finished

p2.join()

print ('Symbol\tWeight\tHuffman Code')

for p in huff:

print ('%s\t%s\t%s' % (p[0], symb2freq[p[0]], p[1]))

if \_\_name\_\_== "\_\_main\_\_":

main()

**Output:**

