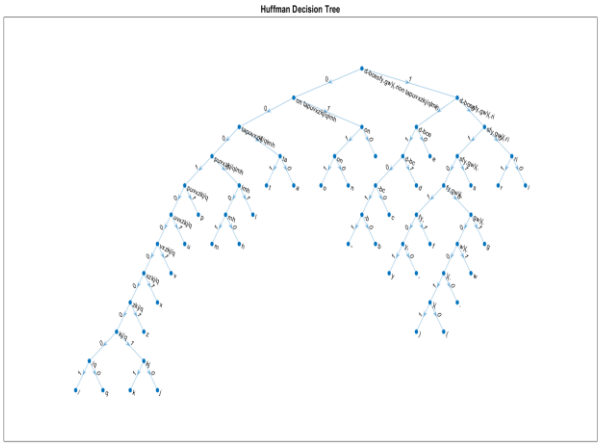

PROJECT 1 CODE DOCUMENTATION

Huffman Algorithm

ABSTRACT

Code snippets are given below which are composed of one main MATLAB file, and 8 different MATLAB functions. The documentation for each function is right before it with function summary and function details.

Ahmed Wael

Information Theory and Coding CIE425

Table of Contents :

Main

1 - Open the Textfile and save its content in a char array then close it

```
fileID = fopen('Huff.txt','r');
text = fscanf(fileID,'%c');
fclose(fileID);
```

2 - Get the probability of each symbol and convert the char array into index array (ex : replace each 'a' with 1)

```
[probabilities,text_mapped] = get_prob_indicies(text);
```

3 - Calculate the Entropy of the file

```
entropy = calc_entropy(probabilities);
```

4 - Calculate the number of bits needed using run length algorithm and calculate its efficiency

```
fixed_length = get_fixed_length( length(probabilities) );
efficiency_fixed_length  =  ( entropy / fixed_length ) *100
```

```
efficiency_fixed_length =

    70.9502
```

5 - Create a table with three data types (probabilities - alphabet - indicies) to link them into one data structure

```
table = alphabet_prob_table(probabilities);
```

6 - Construct the decision Tree and then calculate the avergae number of bits needed and return the dictionary containing each symbol with its index and its Huffman code

```
[tree,dict,avg_huffman] = decision_tree(table);
```

7 - Encode the file using the dictionary

```
encoded = huff_encoder(text_mapped,dict);
```

8 - Decode the encoded file using the dictionary

```
decoded = huff_decoder(encoded,dict);
```

9 - Convert the index array back into char array

```
decoded_symbol = huff_decoder_symbol(decoded,dict);
```

10 - Calculate the Huffman Algorithm Efficiency

```
efficiency_huffman  =  ( entropy / avg_huffman ) *100
```

```
efficiency_huffman =

    99.5477
```

Published with MATLAB® R2018b

get_prob_indicies Function

- get_prob_indicies Function Summary
- get_prob_indicies Function Details

```
function [prob,text_mapped] = get_prob_indicies(text)
```

get_prob_indicies Function Summary

- Inputs :

1. The char array Text file

- Outputs :

1. A corresponding array to the text file array but mapped into double values
2. Array of probabilities for each symbol

get_prob_indicies Function Details

1. Convert the text file into its corresponding ASCII code using the build-in function*double*
2. Pre-allocate the probabilities array with the length of the alphabet used (**33**) with the following mapping :

- 1:26 -> a:z
- 27 -> space character
- 28 -> (character
- 29 ->) character
- 30 -> . character
- 31 -> , character
- 32 -> / character
- 33 -> - character

3. Pre-allocate the mapping array with the length of the text file.

4. loop over the text file and do the following :

- Get the remainder of the current ASCII symbol and**96** which correspond to 'a' -1.
- If the remainder less than or equal 26 , then increase the corresponding index in the probability array by one. If not , compare it with corresponding ASCII code for the remaining symbols ().,/- , then do the same.
- Map this value into the same index in the mapping array.

5. Transpose the index array for convinence in subsequent functions.

6. Get the probability by dividing the probability array by the length of the text file. This can be done as MATLAB supports broadcasting.

```
text_to_ascii = double(text);
prob = zeros(1,33);
text_mapped= zeros(1,length(text));

for i = 1: length(text_to_ascii)
    alphabet = mod( text_to_ascii(i) , 96 );
    if alphabet <= 26
        prob(alphabet) = prob(alphabet) +1;
        text_mapped(i) = alphabet;

    elseif alphabet == 32
        prob(27) = prob(27) +1;
        text_mapped(i) = 27;

    elseif alphabet == 40
        prob(28) = prob(28) +1;
        text_mapped(i) =28;

    elseif alphabet == 41
        prob(29) = prob(29) + 1;
        text_mapped(i) = 29;

    elseif alphabet == 46
        prob(30) = prob(30) +1;
        text_mapped(i) =30;
```

```
    elseif alphabet == 44
        prob(31) = prob(31) +1 ;
        text_mapped(i) = 31;

    elseif alphabet == 47
        prob(32) = prob(32) +1;
        text_mapped(i) = 32;

    elseif alphabet == 45
        prob(33) = prob(33) +1;
        text_mapped(i) = 33;
    end
end

text_mapped = text_mapped';
prob = prob /length(text_to_ascii);
```

```
end
```

Published with MATLAB® R2018b

calc_entropy Function

- calc_entropy Function Summary
- calc_entropy Function Details

```
function [entropy] = calc_entropy(probability)
```

calc_entropy Function Summary

- Inputs :

1. The probability array

- Outputs :

1. The Theoritical Entorpy

calc_entropy Function Details

1. The Entropy formula is , so it's implemented using vectorization instead of a loop in order to increase performance.

```
i = 1:length(probability);
entropy = sum ( - ( probability(i) .* log2(probability(i))));
```

```
end
```

Published with MATLAB® R2018b

get_fixed_length Function

- get_fixed_length Function Summary
- get_fixed_length Function Details

```
function [bits_per_symbol] = get_fixed_length(vector_length)
```

get_fixed_length Function Summary

- Inputs :

1. The length of the probability vector

- Outputs :

1. The bits per symbol needed using a fixed length algorithm

get_fixed_length Function Details

convert the lenght of the vector into a binary format and take the length of it.

```
bits_per_symbol =  length(dec2bin( vector_length) ) ;
```

```
end
```

Published with MATLAB® R2018b

alphabet_prob_table Function

- alphabet_prob_table Function Summary
- alphabet_prob_table Function Details

```
function [Table] = alphabet_prob_table(prob)
```

alphabet_prob_table Function Summary

- Inputs :

1. The probability vector

- Outputs :

1. Table with three data types (probabilities - alphabet - indicies) to link them into one data structure
2. Array of probabilities for each symbol

alphabet_prob_table Function Details

1. Create a character array of the alphabet.
2. Convert the character array into cell array in order to separate the symbols from each other using the built-in function *num2cell* .
3. Create a indicies array with values equal to the length of the probability array.
4. transpose all the vectors for convience and later use.
5. Construct the table by using the command *table* .

```
alphabet = [ 'a':'z' ' ().,/-'];
alphabet = num2cell(alphabet)';
indices = [1:length(prob)]';
prob = prob';
Table = table(prob, alphabet,indices);
```

```
end
```

Published with MATLAB® R2018b

decision_tree Function

- decision_tree Function Summary
- decision_tree Function Details (Construct the tree)
- decision_tree Function Details (Construct the dictionary)
- decision_tree Function Details (Calculate the averege bits per symbol)

```
function [tree,dict,avg] = decision_tree(table)
```

decision_tree Function Summary

- Inputs :

1. The table structure obtained from **alphabet_prob_table** function containing three data types (probabilities - alphabet - indicies).
2. ITEM2

- Outputs :

1. The construced tree with all the nodes, edges and weights **plotted** .
2. Dictionary containing the alphabet, the huffman code and the index.
3. The average number of bits needed in huffman code.

decision_tree Function Details (Construct the tree)

1. Copy the table into another variable in order to save its content for later use when constructing the dictionary.
2. Create a graph with directed edges using the built-in function *digraph*
3. Constuct the weight vector for the nodes.
4. Loop over the symbols in a descending order and do the following :

- Sort the table by rowing in a desecnding order.
- Get the first and second least probabilities by accessing their corresponding index in the table.
- Sum the two least probabilities and assign the value into a variable.
- Construct the two children nodes by getting the corresponding symbol from the sorted table entry.
- Create a 1*2 string array containing both children nodes after converting each one of them seperately into string ,otherwise they will be treated as one symbol **(x= 'c' , y = 'v' , then MATLAB will evaluate z = [x y] as 'cv' , while it needed here as "c", "v".)**
- **The magic happens here** Construct the parent node by first concatenating the two children nodes as characheres, then convert the whole character array into string array **This is needed because for example 'cv' is stored as a 1*2 char array while "cv" is stored as 1*1 string array, which is needed in this case**.
- Add a new edge to the tree by passing the parent and the children nodes with the weight vector for each node (* The smallest one is assigned 0 and the second smallest one is assigned 1*).
- Delete the last entry of the table and store the sum of the two least probabilities in index before the last.
- Store the parent node symbol as a cell in the index before the last in the table using the built-in function *cellstr*.

- plot the tree with the weights on each edges.
- Note that the nodes are unique so the tree will be constructed correctly. This is a manipulation of the graph as tree is a special case of a graph with special properties. These properties are forced here, therefore the graph behave **exactly** as a tree.

```
sorted_table =table;
tree = digraph();
w = [1;0];
for last_symbol_index = length(table.prob):-1:2

    sorted_table = sortrows(sorted_table,{'prob'},{'descend'});

    first_two_least = sorted_table.prob(last_symbol_index);
    second_two_least = sorted_table.prob(last_symbol_index-1);
    sum_two_least = first_two_least + second_two_least;

    first_child_node = cell2mat(sorted_table.alphabet(last_symbol_index));
    second_child_node = cell2mat(sorted_table.alphabet(last_symbol_index-1));
    children_nodes =[convertCharsToStrings(first_child_node)  convertCharsToStrings( (second_child_node)) ];

    parent_node = convertCharsToStrings( [ (first_child_node) (second_child_node)] );

    tree = addedge(tree,(parent_node),children_nodes,w );

    sorted_table(last_symbol_index,:) = [];
    sorted_table.prob(last_symbol_index-1) = sum_two_least;
    sorted_table.alphabet(last_symbol_index-1) = cellstr(parent_node);
```

```
end

% plot(tree,'EdgeLabel',tree.Edges.Weight);
% title('Huffman Decision Tree');
```

decision_tree Function Details (Construct the dictionary)

1. Create an empty cell array and name it dict.
2. Assign the table alphabet to the first column in the dict.
3. Initialize a flag and name it found_symbol_flag. This will be used in the following loop.
4. Loop for a number of times equal to the number of alphabet symbols and do the following :

- Initialize a flag variable each iteration with the name of first_time_flag which will be used to indicate if this is the first time to find this symbol. This will prevent putting a false 0 or 1 at the beginning of the code.
- Initialize temp_weight = 0 each iteration.
- Store the current alphabet symbol in a variable alphabet_symbol.
- Loop through symbols and compare the symbol character by character using a for loop.
- If a match is found, raise the found_symbol_flag and break from the nested loop. Then, if the flag = 1, concatenate the temp_weight array. *restart the state of found_symbol_flag. *Assign the weight vector to the second column with the corresponding row. *Assign the table indicies to the third column with the corresponding row.

- convert the dictionary data type into a table in order to deal with the names of each column.
- Assign {'alphabet','code','index'} to the variables names

```
dict = {};
dict(:,1) = table.alphabet;
found_symbol_flag = 0;

for i = 1: length(table.prob)
    first_time_flag = 0;
    temp_weight=0;

    alphabet_symbol = cell2mat(table.alphabet(i));

    for j = length( tree.Edges.Weight) : -1 :1
        current_symbol = cell2mat(tree.Edges.EndNodes(j,2));

        for k = 1:length(current_symbol)
            char_in_symbol = current_symbol(k);
            if alphabet_symbol ==char_in_symbol
                found_symbol_flag = 1;
                break;
            end
        end
        if found_symbol_flag ==1
            if first_time_flag == 0
                temp_weight= tree.Edges.Weight(j);
                first_time_flag =1;
            else
                temp_weight_temp =tree.Edges.Weight(j); %%lol
                temp_weight = [ temp_weight temp_weight_temp];

            end
        end
        found_symbol_flag =0;
    end

    dict(i,2) = {temp_weight};
    dict(i,3) = {table.indices(i)};


end
dict = cell2table(dict);
dict.Properties.VariableNames = {'alphabet','code','index'};
```

decision_tree Function Details (Calculate the averege bits per symbol)

- Sum over the probability of each symbol with its corresponding code in the dictionary

```
avg=0;
for i = 1:length(table.prob)
    avg = avg+ table.prob(i)*length(cell2mat( dict.code(i)));
```

```
end
```

```
end
```

Published with MATLAB® R2018b

huff_encoder Function

- huff_encoder Function Summary
- huff_encoder Function Details

```
function [encoded] = huff_encoder(text_mapped,dict)
```

huff_encoder Function Summary

- Inputs :

1. The text file mapped array obtained from**get_prob_indicies** function.
2. The dictionary containing the symbols,the codes,and the index for each code obtained from**decision** function.

- Outputs :

1. The encoded sequence using the Huffman code.

huff_encoder Function Details

1. Encode by assigning the value of each element in the mapped text vector to the corresponding code in the dictionary.
2. converting the encoded cell array into a matrix**(i.e. the first cell array is { [0,1,1] } and it should be converted into [0 0 0] as three different elements)** .

```
encoded = dict.code(text_mapped);
encoded = cell2mat(encoded');
encoded = encoded';
```

```
end
```

Published with MATLAB® R2018b

huff_decoder Function

- huff_decoder Function Summary
- huff_decoder Function Details

```
function [decoded] = huff_decoder(encoded,dict)
```

huff_decoder Function Summary

- Inputs :

1. Encoded sequence
2. The dictionary containing the symbols,the codes,and the index for each code obtained from**decision** function.

- Outputs :

1. The decoded sequence using the Huffman code.

huff_decoder Function Details

1. Initialize the end and start indicies to be 1.
2. Create a flag to indicate if a code for a symbol is found.
3. Iterate over the encoded sequence until the end index is equal to the and do the following :

- Save the values of the encoded sequence from the start index to the end index in a temporary variable.
- Loop for a number of times equal the number of the symbols and do the following :

- Create a compare variable which contains the code of each symbol for each iteration**(i.e. if j = 1 then compare will be [0;1;1;0] for example which corresponds for the huffman code for 'a')** . This can be done be calling the built-in function*cell2mat* and transponsing it for convience.
- If the values of the temporary variable and the compare variable are equal, concatenate the decoded array with corresponding index**(i.e. if 'a' is found, write 1)** .
- Raise the found symbol flag and break from the loop as the following iterations are unnecessary.
- Increase the end index by one , and increase the start index by end +1 if a the flag is raised**(i.e. the flag = 1 for start = 2 and end = 6, then we need to have start = 7 and end =8)** .

```
start_index=1;
end_index = 1;
decoded = [];
found_symbol_flag=0;

while(end_index <= length(encoded))
    temp_code = encoded(start_index:end_index);
    for j = 1:length(dict.code)
        compare = dict.code(j);
        compare = cell2mat(compare');
        compare = compare';
        if ( isequal ( temp_code, compare ) )
            new = (dict.index(j));
            decoded = [ decoded ; new];
            found_symbol_flag =1;
            break;
        end
    end
    if found_symbol_flag ==1
        start_index = end_index +1;
    end
    end_index = end_index +1;

    found_symbol_flag =0;
end
```

```
end
```

Published with MATLAB® R2018b

huff_decoder_symbol Function

- huff_decoder_symbol Function Summary
- huff_decoder_symbol Function Details

```
function [decoded_symbol] = huff_decoder_symbol(decoded,dict)
```

huff_decoder_symbol Function Summary

- Inputs :

1. Decoded sequence
2. The dictionary containing the symbols,the codes,and the index for each code obtained from**decision** function.

- Outputs :

1. The decoded sequence using the Huffman code but converted back into char array.

huff_decoder_symbol Function Details

1. Find the mapped value for each symbol by accessing the decoded vector.
2. Find the corresponding character by using accessing the dicitonary element of this mapped value
3. Convert the cell into vector using *cell2mat* and transpose it.

```
index_of_symbol = 1 : length(decoded);
decoded_symbol = cell2mat(dict.alphabet( decoded(index_of_symbol)) )';
```

Published with MATLAB® R2018b