COMP261 Tutorial Week 11

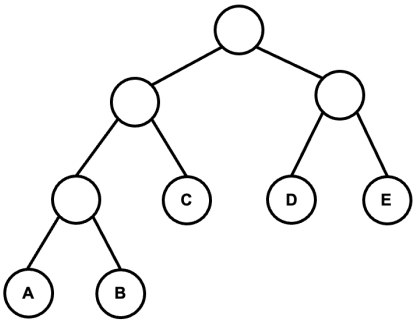
Data Encoding and Decoding

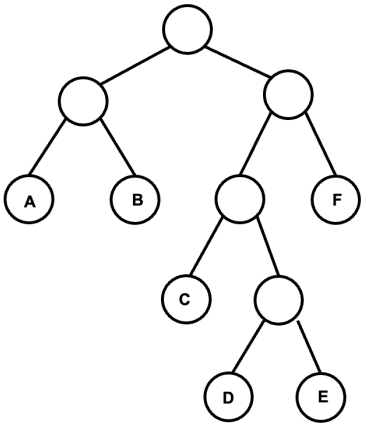
Huffman Encoding

1. Suppose the symbol probabilities for {A, B, C, D, E} are:
P(A) = 0.067, P(B) = 0.133, P(C) = 0.2, P(D) = 0.267, P(E) = 0.333

   (a) Please construct a Huffman tree for encoding the five characters. using these probabilities.
   (b) What kind of specific pattern that the symbols' probabilities need to follow to result in such a linear Huffman tree?

2. What kind of probability distributions can result in the following Huffman trees when using the Huffman encoding?





3. What kind of probability distribution can provide the most benefits in terms of compression rate when using the Huffman coding method?

Lempel-Ziv (LZ) Encoding

1. Please encode the following example string: "abababababcabcabc" by the Lempel-Ziv algorithm and write up all the tuples.

Question: what do you do when there are multiple occurrences of the same pattern? Did you consider the slide window size?

2.Please encode the following example string: "the quick brown fox jumps over a lazy dog" by the Lempel-Ziv algorithm and output all the tuples.

Question: Compared to the original ASCII code, how many bytes will be used in the LZ77's encoding result? What are the advantages and limitations of the LZ77 algorithm compared to other compression techniques?

3.(a) Decode the following sequence encoded using the LZ77 algorithm.:

[0, 0, 't'][0, 0, 'o'][0, 0, '_'][0, 0, 'b'][0, 0, 'e'][2, 1, 'o'][0, 0, 'r'][5, 1, 'r'][6, 1, 'n'][9, 1, 't'][10, 1, 't'][13, 4,'\n ']

(b) Decode the following sequence encoded using the LZ77 algorithm.:

[0, 0, 't'][0, 0, 'o'][0, 0, '_'][0, 0, 'b'][0, 0, 'e'][2, 1, 'o'][0, 0, 'r'][5, 1, 'r'][6, 1, 'n'][9, 1, 't'][10, 1, 't'][13, 1,'_ '][13, 2, '\n']

Question: After decoding, please check whether the above tuple sequences are correct encoding results for the text.

4. More questions:

--How does the LZ77 algorithm achieve compression by encoding phrases? Provide a step-by-step explanation.

--Discuss the trade-off between compression ratio and encoding complexity in the LZ77 algorithm.

String Search:

1. Go through the following example to understand how the Knuth-Morris-Pratt (KMP) Algorithm works on a real string. Consider the following text and string pattern S that we want to search for:
T: "ABCDABCDABABCDABCDABDE"
S: "ABCDABD"
Please construct the Partial Matching Table for the above example and walk through the text T to find a match of S. Refer to the partial matching table to determine how many characters we can shift the pattern without missing a potential match after a failed match.

2. Construct the Partial Matching Table for the following example

A	B	A	B	A	C	A	B	A	…
-1	0								…

(For Monday tutorials before learning the KMP method)

KMP string searching algorithm:

The idea of jumping more than one letter when a match fails in the Knuth-Morris-Pratt (KMP) Algorithm is to take advantage of the information stored in the partial matching table to avoid redundant comparisons.

When a mismatch occurs between the pattern and the text, instead of shifting the pattern by just one position to the right, the KMP Algorithm utilizes the information from the partial matching table to determine the maximum number of characters it can safely skip. This allows the algorithm to avoid unnecessary comparisons and improves its efficiency.

The value in the partial matching table at the current position of the pattern indicates the length of the longest proper suffix of the pattern that is also a prefix up to that position. By utilizing this information, the algorithm can determine how far it can shift the pattern to the right without missing any potential matches.

When a mismatch occurs, the algorithm checks the value in the partial matching table at the current position. This value represents the length of a prefix that matches the suffix up to the mismatched character. By shifting the pattern forward by this value, the algorithm effectively skips unnecessary comparisons.

An example:

Text: ABCDABCDABABCDABCDABDE
Pattern: ABCDABD
First Try:
ABCDABCDABABCDABCDABDE
ABCDABD
The next try will be at:
ABCDABCDABABCDABCDABDE
ABCDABD

Instead of shifting the pattern by just one position to the right, we can utilize the information from the partial matching table. Since the value is 2, we know that the first two characters of the pattern (AB) are a matching prefix and suffix. Therefore, we can safely skip these two characters and align the pattern with the next occurrence of the mismatched character in the text.

By jumping more than one letter based on the value from the partial matching table, the KMP Algorithm avoids unnecessary comparisons and efficiently progresses through the text, increasing its overall performance.

Partial Matching Table of KMP

The partial matching table is used to optimize the pattern matching process by avoiding unnecessary comparisons.

The partial matching table is constructed based on the pattern string. It helps determine how far the pattern can be shifted without missing any potential matches. Each entry in the table represents the length of the longest proper prefix of the pattern that is also a suffix of the pattern, up to that position.

To illustrate this, let's take an example pattern "abcdabca". We start by initializing the table as an array of zeros of the same length as the pattern: [0, 0, 0, 0, 0, 0, 0, 0]. Now, we iterate through the pattern from left to right, updating the table values based on the current character.

1. At position 0, there are no proper prefixes or suffixes, so the value remains 0.
2. At position 1, the only proper prefix is "a" with length 1, and it's not a suffix, so the value remains 0.
3. At position 2, the proper prefix is "ab" and the suffix is "ab", which have a length of 2. Therefore, the value is 2.
4. At position 3, the proper prefix is "abc" and the suffix is "c", which have a length of 0. Therefore, the value is 0.
5. At position 4, the proper prefix is "abcd" and the suffix is "d", which have a length of 0. Therefore, the value is 0.
6. At position 5, the proper prefix is "abcda" and the suffix is "a", which have a length of 1. Therefore, the value is 1.
7. At position 6, the proper prefix is "abcdab" and the suffix is "ab", which have a length of 2. Therefore, the value is 2.
8. At position 7, the proper prefix is "abcdabc" and the suffix is "abc", which have a length of 3. Therefore, the value is 3.

Thus, the final partial matching table would be: [0, 0, 2, 0, 0, 1, 2, 3].

The partial matching table is crucial in the KMP Algorithm because it allows the algorithm to skip unnecessary comparisons by shifting the pattern based on the information stored in the table. When a mismatch occurs between the pattern and the text, the algorithm uses the value in the partial matching table to determine the new starting position of the pattern for the next comparison, instead of shifting it by one position.