CS 165 Project 3: Data Compression

Sam Balana / Dallas Johnson
University of California, Irvine
E-mail: `sbalana@uci.edu` / `dallasj@uci.edu`

Introduction. This algorithm is based off of Ukkonen's Suffix Tree with modifications to allow for a sliding window. We selected this method since its O(1) search/insertion time is very time-efficient.

1. Compilation Instructions

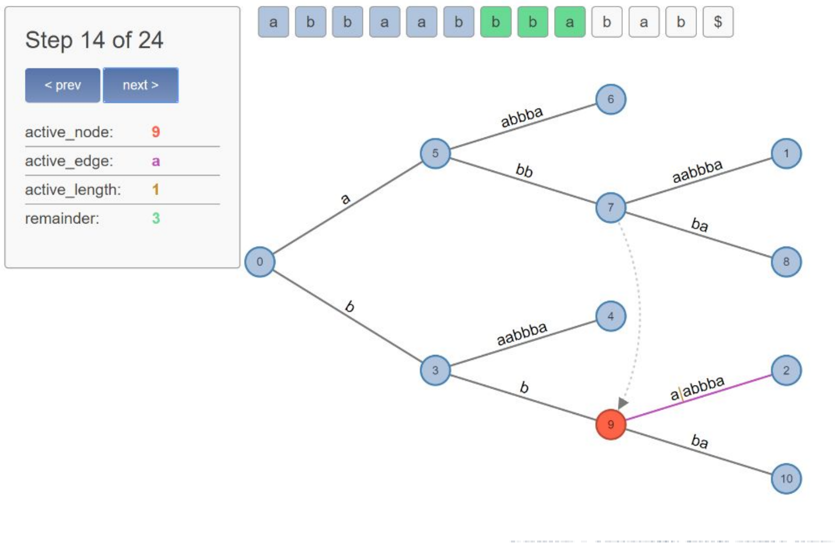
To compile our two projects, simply run the make command to use our Makefile. This will build both the LZ command and the EXPAND command.

2. Data Structures

To implement Ukkonen's algorithm, we used a suffix tree, a hash map, and a circular queue. For reconstruction, we used a similar circular queue.

1.1. Suffix Tree

A suffix tree contains all possible suffixes of a given string. Using a hashmap to store the tree allows us to attain O(1) lookups. Through careful manipulation of the tree, we can clear previous suffixes upon the exit of an entry from our sliding window. This allows us to do a single operation in O(1) time, taking just O(n) time to perform the compression for n characters.



Step 14 of 24

active_node: 9
active_edge: a
active_length: 1
remainder: 3

1.2. Hash Map

Hash maps allow O(1) lookup for key-value pairs using a hash function. By hashing the first letter of a match, we create a set of unique matches at each node to traverse as a tree.

1.3. Circular Queue

Using a circular queue allows us to avoid significant amounts of array shifting, which significantly improves our performance due to the large window sizes used. Our circular queue supports push and pop features with a static length. The circular queue used for expansion only supports a combined push/pop function and a head-based lookup function, as these functions are all that is needed for reconstruction.

3. Theoretical Worst/Average Cases

Due to the structure of our code, we will focus on the time complexity of three different parts of our algorithm. Our interpreter is completely separated from the rest of our code, so it is a good component to analyze, and our tokenizer and writer are separated enough that their time complexities are easily analyzable and important.

1.1. Interpreter

The interpreter reads characters from standard input until it is given the end-of-file (EOF) character. Assuming *n* characters before the EOF, it can do anything from roughly O(n) work when passed only string literals up to O(Ln) where L is our maximum match length in the case of all offsets. This time complexity is to be expected, since we need to write at least every character in the original file. In the worst case, this leads us to O(Ln) time, and in the average case, we will likely fall somewhere around O($\frac{Ln}{A}$), where A is our compression factor.

1.2. Tokenizer

As stated above, we have based our tokenizer on Ukkonen's algorithm. This allows us to achieve O(n) time complexity in this step, since each combined insertion/search/deletion takes constant time. It is prudent to note that this time complexity holds true in both the worst and average cases.

1.3. Writer

When time complexity is considered relative to the number of elements in this function's input vector, we have roughly O(n) time for n inputs. This makes sense, since all we are doing in this step is converting the input structures into binary code for compact storage, and we must do this for each element. This will also take O(n) time in the average case.

4. Observed Data

Since the "best" parameter values are rather ambiguous, we will assume this refers to the parameters with the highest compression factor. Below is a log of the most space-efficient parameter values for both files in the standard suite. These values were discovered by looping through all possible parameter values for each file.

File	N	L	S	Compression Rate
kennedy.xls				NA
book1				NA