Verifying Burrows Wheeler Transformation In Coq

Hanlin He, Sourav Dasgupta, Paul Parrot, Alan Padilla Chua

December 16, 2017

1 Motivation

The idea for this research project was given to our team by Dr. Kevin Hamlen. Our initial goal was to verify a full compression algorithm from top to bottom. However, given time constraints we narrowed our search to *bzip2*'s compression stack and saw that its core algorithm depended on the Burrows Wheeler Transformation [1]. BWT (Burrows Wheeler Transformation) is a block-sorting compression algorithm designed to improve the compressibility of a given string. BWT also has some important security properties related to integrity. It has a lossless property, meaning the forward transformation can be reversed to get back the original string. This means that there is no loss of data in either the forward or inverse transformations. We saw BWT to be of great potential to be verified in Coq. Its full verification could narrow the trusted code base of *bzip2*'s compression and allow future teams to tackle the full verification of *bzip2*.

2 Preliminaries

Our Coq implementation has been tested on the CoqIDE version 8.6.1 for Windows, Mac, and Linux.

3 Project Summary

3.1 Approach

For our implementation of BWT in Coq we decided to prove that the transformation is lossless; Meaning that for any string we apply the forward

BWT on, we can subsequently perform the inverse BWT on that result and get the original string back. In order to accomplish this and be able to reason about it, we implemented the forward BWT and inverse BWT in Coq. Before getting to these two implementations we need to discuss some more general, but still important, parts of our implementation.

Dr. Kevin Hamlen mentioned that proving theorems about lists could lead to complex issues. So instead of using lists, for our main implementation we chose to implement strings as mappings ($str : nat \rightarrow option\ nat$) and matrices as a mapping of mappings ($matrix : nat \rightarrow str$). It is worth noting that we did create a generic implementation of BWT using lists but did not attempt to prove any lemmas about it. Our proofs are solely related to the string mapping version of our Coq code.

Following is a description of our approach to implementing the forward BWT. Given a string S as input, we first convert the string into a *str* mapping s. Then we construct a matrix by right shifting s. This fills the matrix with each permutation of the original string. To avoid proving a sort function for our mapping, we then assume a sort function is given that sorts the matrix lexicographically. Finally, we return the last column of the sorted matrix as the result string.

Following is a description of our approach to implementing the inverse BWT. Given a string L, an index i, and a *sort* function as input, we first sort L lexicographically. Then we generate the index mapping π_L between the given string and the sorted string. We then recursively generate a sequence of indices as $\pi_L^k(i)$, each of which corresponds to the k-th character in the original string. Finally, we apply all the indices with the given string L to regenerate the original string before the forward BWT.

3.2 Difficulties

3.2.1 Sorting

The BWT algorithm requires two stable lexicographical sorts: matrix sort and string sort. Our main implementation of strings and matrices uses an abstract mapping of the strings rather than lists. This prohibited us from importing a Coq sorting library because it was unclear how our abstract mapping could use a sorting library meant for lists. To overcome this, we decided that instead of proving a sorting algorithm ourselves, we would assume either a sorting algorithm that satisfies a set of properties for our abstract mapping is given, or the sorted version of the string or matrix mapping is given.

3.2.2 Understanding the Mathematical Paper Proof of Burrows Wheeler Transform

Before we go into the paper proof, we would like to point out that the proof uses the following notations:

- Σ: Finite, non-empty symbol/alphabet set. Every string L is composed of a sequence of symbols which belong to Σ.
- λ_L: A mapping $\mathbb{N} \to \Sigma$, which given a natural number $i \leq length(L)$, will return the symbol α located at the ith position in L.
- π_L: A mapping $\mathbb{N} \to \mathbb{N}$, which maps an index $i \leq length(L)$ with the index of the symbol α in L that should have been in position i if the symbols in string L were stable sorted.
- $context_k(L)$: Means the k length prefix of string L.
- *conjugacy class of string* L: A sequence or collection of strings where each string w_i is a right-shift of w_{i-1}, with $w_0 = L$. Note that the right shift is a circular one, i.e., the last symbol in w_{i-1} goes to the beginning of w_i . We treat the conjugacy class of L as a matrix.

The preceding notations have been described very briefly for convenience. For full definitions, the reader is requested to refer [2].

The paper theorem follows:

The statement is a generic one which states that, given a matrix of k order context sorted conjugacy class of a string m , the string L composed of the last letters of each of the string in the sorted conjugacy class, the t-order context of w_i can be constructed as follows: starting with i , and recursively apply the π_L function, and take the letter in L , indexed at each recursive application of π_L. The recursion continues till k.

A more formal way of presenting the above stated notion (as mentioned in Lemma 3 of [2]) is as follows:

Theorem 1. *Let* $k \in \mathbb{N}$. *Let* $\bigcup_{i=1}^{s}[v_i] = \{w_1, \ldots, w_n\} \subseteq \Sigma^+$ *be a multiset built from conjugacy classes* $[v_i]$. *Let* $M = (w_1, \ldots, w_n)$ *satisfy* $\mathrm{context}_k(w_1) \leq \cdots \leq \mathrm{context}_k(w_n)$ *and let* $L = \mathrm{last}(w_1) \cdots \mathrm{last}(w_n)$ *be the sequence of the last symbols.*

Then

$$\mathrm{context}_k(w_i) = \lambda_L \pi_L(i) \cdot \lambda_L \pi_L^2(i) \cdots \lambda_L \pi_L^k(i)$$

where π_L^k *denotes the t-fold application of* π_L *and* $\lambda_L \pi_L^t(i) = \lambda_L\big(\pi_L^t(i)\big)$.

The proof is done using induction. While we do not present the entire proof in the paper, the core idea is briefly explained as follows: It is trivial to show that the above theorem holds for $k = 0$. As the induction hypothesis, it is assumed in the paper that the for any $t \leq k$, the theorem holds for $t - 1$. Using the hypothesis, each w_i is then right shifted and sorted by t-order context resulting in the sequence $(u_1, \ldots, u_n)$. Following this, it is argued that t-order context of u_i equals the t-order context of $w_{\pi_L}(i)$. Plugging this relation to the induction hypothesis yields the induction.

3.2.3 List of Necessary Implications for the Final Theorem

Interestingly, the proof mentioned in the previous section itself tells us how to compute the inverse of BWT. If we have the string L (which is the BWT transform of m) and π_L permutation (which can be constructed from L by sorting L), we can compute k order context of any w_i belonging to the conjugacy class of our original string m. Note that since m itself is one of the w_i's, if we start with the correct value of i, we can get the original string m. Also note that if we set the value of k to be $length(m)$, we get the entire m back.

4 Accomplishments

4.1 Theorem

Matrix is the theorem that we constructed from our implementation that we believe proves the losslessness of the BWT. This theorem states that given the last column L (and the sorted last column *L_sorted*), of the sorted right shift matrix, and an index i, we can perform the inverse BWT which will return the string at index i in the sorted right shift matrix. This theorem states a general solution whereby we can reconstruct any string from the conjugacy matrix given the forward BWT result string and the index at which that string resides within the matrix. Obviously this implies that with our solution we can also reconstruct the original string that was given so long as we know its index.

4.2 Working Functions (Definitions)

4.2.1 `bwt`

This function is our implementation of the forward BWT. It takes a string and a sort function as input. First, it creates a string mapping from the given string. Then it generates a matrix such that each row contains one right shifted permutation of the string mapping. Then it uses the sort function to lexicographically sort the rows. The last column, denoted L, is the output string. This implementation relies on several helper functions described below.

1. `string_to_map`
   This function takes a normal ascii string and returns it in a mapping form that is defined by us. This mapping makes it easier for us to generate a matrix of permutations for the string.

2. `map_to_conjugacy`
   This function returns a matrix such that each row contains a permutation of the original string. These permutations are all right shifted by one character and wrap around to the beginning of the row.

3. `map_to_string`
   This function takes our defined mapping form of a string and returns the corresponding string.

4.2.2 `inverse_bwt`

This function is our implementation of the inverse BWT. It simply applies `lambda` to each index in the string returned by `inverse_bwt'`; The result of which is the original string. This implementation relies on several helper functions described below.

1. `pi`
   This function takes a BWT string, and it's lexicographically sorted permutation, as input and returns a mapping. The returned mapping represents the index of each letter in the transformed string.

2. `inverse_bwt'`
   This function uses `pi` to generate the index mapping. It then recursively generates a sequence of indices; Each of which corresponds to one character in the original string.

3. `lambda`
   This function takes an input string *m* and index *i* and returns the character at position *i* in *m*.

5 Team Coordination

Table 1: Coordination

Task	Person
Presentation Slides	Alan, Paul, Hanlin, Sourav
Presentation	Alan, Hanlin
Coq Implementation	Hanlin, Sourav, Alan
Coq Proofs	Hanlin, Sourav, Paul
Report	Alan, Paul, Hanlin, Sourav

6 Future Work

We laid a solid foundation for a proper proof to be completed. However certain circumstances, described earlier in the report, prevented us from fully achieving our goal. For future work we propose the full completion of the theorem using our current implementation and potentially some modifications that will come from working on the proof. We believe that completion of this proof will allow future teams to focus on formally verifying the other sections of *bzip2*'s compression stack; And eventually leading to a complete, machine-verified, proof of *bzip2*.

References

[1] Wikipedia: bzip2. https://en.wikipedia.org/wiki/Bzip2, 2017. [Online; accessed December 16, 2017].

[2] Manfred Kufleitner. On bijective variants of the burrows-wheeler transform. *prague stringology conference*, pages 65–79, 2009.