

**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

-----🙞🙜🕮🙞🙜----- ****

**Project 1**

**Phép biến đổi BURROWS WHEELER**

**và ứng dụng nén file text Tiếng Anh dung lượng lớn.**

***Giảng viên hướng dẫn: TS. Trần Vĩnh Đức***

***Sinh viên: Nguyễn Hữu Kiêt***

***MSSV : 20183571***

**Hà Nội – 1/2021**

**Đề tài: Phép biến đổi BURROWS WHEELER**

**và ứng dụng nén file text Tiếng Anh dung lượng lớn**

**Người thực hiện báo cáo:**

**Nguyễn Hữu Kiệt**

**Hà Nội – 2021**

Mục lục:

[1. Mở đầu: 3](#_Toc61638702)

[2. Nội Dung: 4](#_Toc61638703)

[2.1. Phép biến đổi Burrows Wheeler (BWT) xuôi: 4](#_Toc61638704)

[2.1.1. Giới thiệu: 4](#_Toc61638705)

[2.1.2. Thuật toán: 4](#_Toc61638706)

[2.1.2. Giải thích: 4](#_Toc61638707)

[2.1.2. Mã giả: 5](#_Toc61638708)

[2.1.3. Tối ưu hóa BWT xuôi với mảng hậu tố (suffix array): 6](#_Toc61638709)

[2.2. Phép biến đổi Burrows Wheeler (BWT) ngược: 7](#_Toc61638710)

[2.2.1. Giới thiệu: 7](#_Toc61638711)

[2.2.2. Thuật toán: 7](#_Toc61638712)

[2.2.3. Giải thích: 8](#_Toc61638713)

[2.2.4. Mã giả: 8](#_Toc61638714)

[2.3. Xây dựng chương trình demo: 9](#_Toc61638715)

[2.3.1. Về nén xâu ký tự: 9](#_Toc61638716)

[2.3.2. Giải thuật BWT sử dụng: 9](#_Toc61638717)

[2.3.3. Tăng tốc độ nén xâu lớn: 10](#_Toc61638718)

[2.3.4. Kết quả thử nghiệm: 10](#_Toc61638719)

[2.3.5. Chương trình demo: 10](#_Toc61638720)

[3. Kết luận: 12](#_Toc61638721)

[Danh mục tài liệu tham khảo: 13](#_Toc61638722)

**Danh mục hình ảnh:**

Hình 1: Chương trình demo……………………………………………………… 11

**Danh mục từ viết tắt:**

BWT : Burrows Wheeler Transform.

**Link mã nguồn:** <https://github.com/KietNguyen10112000/Project1>

# 1. Mở đầu:

Nén dữ liệu (tiếng Anh: Data compression) là việc chuyển định dạng thông tin sử dụng ít bit hơn cách thể hiện ở dữ liệu gốc. Tùy theo dữ liệu có bị thay đổi trước và sau khi giải nén không, người ta chia nén thành hai loại: Nguyên vẹn (lossless) và bị mất dữ liệu (lossy). Nén mất dữ liệu giảm số lượng bit bằng cách xác định các thông tin không cần thiết và loại bỏ chúng.

Nén dữ liệu là cần thiết vì giảm được nguồn tài nguyên cũng như dung lượng lưu trữ hay băng thông đường truyền. Tuy nhiên, vì dữ liệu nén cần được giải nén nên sẽ đòi hỏi nhiều phần cứng và xử lý.

# 2. Nội Dung:

## 2.1. Phép biến đổi Burrows Wheeler (BWT) xuôi:

### 2.1.1. Giới thiệu:

Phép biến đổi Burrows Wheeler (gọi tắt là BWT) là một thuật toán để tổ chức lại dữ liệu text được sử dụng trong nén dữ liệu độ mất mát ít.

Một xâu text sau khi qua BWT sẽ trở thành một xâu có khả năng nén tốt hơn nhiều so với xâu ban đầu.

### 2.1.2. Thuật toán:

Bảng sau mô tả cách thực hiện BWT xuôi:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Transformation** | | | | |
| **1. Input** | **2. All rotations** | **3. Sort into lexical order** | **4. Take the last column** | **5. Output** |
| ^BANANA| | ^BANANA|  |^BANANA  A|^BANAN  NA|^BANA  ANA|^BAN  NANA|^BA  ANANA|^B  BANANA|^ | **A**NANA|^B  **A**NA|^BAN  **A**|^BANAN  **B**ANANA|^  **N**ANA|^BA  **N**A|^BANA  **^**BANANA|  **|**^BANANA | ANANA|^**B**  ANA|^BA**N**  A|^BANA**N**  BANANA|**^**  NANA|^B**A**  NA|^BAN**A**  ^BANANA**|**  |^BANAN**A** | BNN^AA|A |

BWT được hoàn thành bằng cách sắp xếp tất cả các xâu dịch trái (hoặc phải cho kết quả tương đương) xoay vòng theo thứ tự từ điển rồi lấy 1 kí tự cuối cùng cuối mỗi xâu.

### 2.1.2. Giải thích:

Xâu dịch trái xoay vòng: là xâu được dịch qua bên trái một số kí tự kể từ xâu ban đầu và các kí tự bị tràn ra ngoài sẽ được điền ngược lại vào đằng sau. Ví dụ:

Ta có xâu: BANANA^

Vậy ta sẽ có xâu dịch trái xoay vòng:

1 kí tự sẽ là: ANANA^B

2 kí tự sẽ là: NANA^BA

….

Xâu cuối: ^BANANA

Sắp xếp tất cả các xâu đó theo thứ tự từ điển rồi lấy kí tự cuối cùng mỗi xâu ta được xâu sau khi qua BWT.

Ta nhận thấy sau khi biến đổi các kí tự giống nhau sẽ được đẩy lại gần nhau. Tại sao?

Điều này dễ thấy bởi trong Tiếng Anh nói riêng và các ngôn ngữ khác nói riêng thì trong 1 văn bản các kí đôi sẽ được lặp đi lặp lại rất nhiều.

Đơn cử như trong Tiếng Anh, chỉ riêng đuôi “tion” trong 1 văn bản có thể xuất hiện rất nhiều lần, điều đó đồng nghĩa với việc các kí tự đôi đi kèm nhau như: “ti”, “io”, “on”… cũng xuất hiện nhiều lần.

Các kí tự đôi như vậy sau khi thực hiện BWT thì sẽ được đẩy lại gần nhau bởi khi thực hiện dịch trái xoay vòng ta đã tách các kí tự đôi ra, đẩy 1 kí tự lên đầu còn 1 kí tự xuống cuối, vì vậy mà các kí tự đôi như “ti” sẽ trở thành “i…t” và khi thực hiện sắp xếp các xâu bắt đầu bằng “i” sẽ được đẩy lại gần nhau đồng nghĩa với các kí tự “t” ở cuối cũng được đẩy lại gần nhau.

Từ đó, ta được 1 xâu mới có khả năng nén tốt hơn nhiều so với xâu cũ.

Vậy tại sao ta không lấy các kí tự đầu mỗi xâu, không phải lấy vậy sẽ được 1 xâu mới có khả năng nén tốt hơn rất rất nhiều sao?

Đơn giản vì nếu lấy kí tự đầu mỗi xâu thì không có í nghĩa gì (vì nó sẽ thành xâu kiểu “aaaaaaaaaaaaabbbbbbbbbbbcccccccccc…”, cùng với đó là ta không thể giải ngược được các xâu này để ra xâu ban đầu trong khi nếu lấy các kí tự cuối ta hoàn toàn có thể làm được (BWT ngược).

### 2.1.2. Mã giả:

**function** BWT (*string* s)

create a table, where the rows are all possible rotations of s

sort rows alphabetically

**return** (last column of the table)

Độ phức tạp tính toán: O(n2), với n là số lượng kí tự đầu vào của xâu string s.

Bộ nhớ chiếm dụng: O(n2).

### 2.1.3. Tối ưu hóa BWT xuôi với mảng hậu tố (suffix array):

Ta có công thức:

Trong đó:

+) BWT[i] là kí tự thứ i của xâu BWT

+) SA[i] là phần tử thứ i của mảng hậu tố SA

+) T là xâu gốc

+) $ là kí tự quy ước là kết thúc xâu

Từ đó bài toán biến đổi BWT xuôi được qui về bài toán tính mảng hậu tố.

Tính toán BWT khi sử dụng mảng hậu tố sẽ đem lại ta những lợi ích sau:

+) Độ phức tạp tính toán trên lý thuyết là có thể đạt O(nlog(n)) (thậm chí có thuật toán còn đạt tới O(n) nhưng trên thực tế thì những giải thuật O(nlog(n)) thường chậm hơn giải thuật O(n2) nếu không được tối ưu tốt (giải thích ở phần xây dụng ứng dụng demo)).

+) Bộ nhớ chiếm dụng: O(n). Điều này rất quan trọng vì khi nén nhưng xâu văn bản lớn (cỡ vài trăm mb thậm chí vài gb) sẽ gây ra tràn vùng nhớ heap khi chạy chương trình.

## 2.2. Phép biến đổi Burrows Wheeler (BWT) ngược:

### 2.2.1. Giới thiệu:

Phép biến đổi BWT ngược là thuật toán để giải ngược chuỗi dữ liệu đã được đi qua BWT xuôi.

### 2.2.2. Thuật toán:

Bảng sau mô tả thuật toán giải ngược xâu BWT:

|  |  |  |  |
| --- | --- | --- | --- |
| Input: BNN^AA|A | | | |
| **Add 1** | **Sort 1** | **Add 2** | **Sort 2** |
| B  N  N  ^  A  A  |  A | A  A  A  B  N  N  ^  | | BA  NA  NA  ^B  AN  AN  |^  A| | AN  AN  A|  BA  NA  NA  ^B  |^ |
| **Add 3** | **Sort 3** | **Add 4** | **Sort 4** |
| BAN  NAN  NA|  ^BA  ANA  ANA  |^B  A|^ | ANA  ANA  A|^  BAN  NAN  NA|  ^BA  |^B | BANA  NANA  NA|^  ^BAN  ANAN  ANA|  |^BA  A|^B | ANAN  ANA|  A|^B  BANA  NANA  NA|^  ^BAN  |^BA |
| **Add 5** | **Sort 5** | **Add 6** | **Sort 6** |
| BANAN  NANA|  NA|^B  ^BANA  ANANA  ANA|^  |^BAN  A|^BA | ANANA  ANA|^  A|^BA  BANAN  NANA|  NA|^B  ^BANA  |^BAN | BANANA  NANA|^  NA|^BA  ^BANAN  ANANA|  ANA|^B  |^BANA  A|^BAN | ANANA|  ANA|^B  A|^BAN  BANANA  NANA|^  NA|^BA  ^BANAN  |^BANA |

|  |  |  |  |
| --- | --- | --- | --- |
| **Add 7** | **Sort 7** | **Add 8** | **Sort 8** |
| BANANA|  NANA|^B  NA|^BAN  ^BANANA  ANANA|^  ANA|^BA  |^BANAN  A|^BANA | ANANA|^  ANA|^BA  A|^BANA  BANANA|  NANA|^B  NA|^BAN  ^BANANA  |^BANAN | BANANA|^  NANA|^BA  NA|^BANA  ^BANANA|  ANANA|^B  ANA|^BAN  |^BANANA  A|^BANAN | ANANA|^B  ANA|^BAN  A|^BANAN  BANANA|^  NANA|^BA  NA|^BANA  ^BANANA|  |^BANANA |
| **Output** | | | |
| ^BANANA| | | | |

### 2.2.3. Giải thích:

Khi thực hiện biến đổi xuôi chúng ta đã xếp chồng các xâu dịch trái xoay vòng, xong lấy cột cuối cùng của bảng nhận được, nên khi giải ngược chúng ta chỉ cần làm ngược lại là sẽ giải ngược được xâu BWT.

### 2.2.4. Mã giả:

**function** inverseBWT (*string* s)

create empty table

**repeat** length(s) **times**

// first insert creates first column

insert s as a column of table before first column of the table

sort rows of the table alphabetically

**return** (row that ends with the 'EOF' character)

Độ phức tạp tính toán: O(n2).

Bộ nhớ chiếm dụng: O(n2).

## 2.3. Xây dựng chương trình demo:

**Quy ước: xâu văn bản được nén không chứa kí tự ÿ , ta quy ước đó là kí tự kết thúc xâu và là điểm khởi đầu để giải ngược xâu BWT.**

Ngôn ngữ lập trình sử dụng: C++.

Môi trường phát triển: Microsoft Visual Studio Community 2019.

### 2.3.1. Về nén xâu ký tự:

Có rất nhiều thuật toán nén xâu kí tự hiệu quả như mã hóa Huffman, CCITT group 3 & 4 compression, LZW compression, … Nhưng trong chương trình demo này chúng ta sẽ dùng kĩ thuật nén thô sơ nhất có thể để đánh giá hiệu quả của biến đổi BWT trong nén dữ liệu.

Kĩ thuật nén sử dụng:

+) Nếu ta có một xâu dạng: “aaaaaaabbbbbbbbbbbbbcccccc” ta sẽ biến đổi nó thành “a7b13c6”.

+) Nếu trong xâu có các kĩ tự số 1, 2, … và có nhiều hơn một kí tự số kề nhau ta sẽ mã hóa một kí tự số bằng 4 bits nhị phân rồi gộp hai kí tự lại với nhau thành nhiều kí tự được bắt đầu bằng 0x03 và kết thúc bằng 0xFF để đảm bảo không nhầm với các kí tự chữ khác.

### 2.3.2. Giải thuật BWT sử dụng:

Chương trình demo sẽ sử dụng giải thuật có độ phức tạp trên lý thuyết là O(n2).

Lý do, sau khi đã triển khai một số giải thuật O(nlog(n)) tốc độ đạt được đều chậm hơn nhiều so với giải thuật O(n2).

Giải thích:

Giải thuật O(n2) sử dụng hàm so sánh hai xâu là:

*strcmp(const char\* s1, const char\* s2).*

Giải thuật O(nlog(n)) sử dụng hàm so sánh là hàm băm của một *const char\* s*, và một giải thuật O(nlog(n)) khác lại gọi hàm *std::sort()* của thư viện STL quá nhiều lần.

Lý do giải thuật O(n2) nhanh hơn hai giải thuật bên dưới bởi các xâu văn bản mà người bình thương viết lên thỉnh thoảng mới có những xâu trùng nhau liên tục trong một câu cả. Ví dụ như câu: “He is my father” thì khi so sánh 2 xâu “He is my father” và xâu “e is my fatherH” (xâu xoay vòng đầu tiên trong BWT), hàm *strcmp()* sẽ chạy trong thời gian 1 phép so sánh còn các giải thuật nêu trên sẽ phải tốn nhiều phép tính hơn, các giải thuật đó chủ yếu mang tính ổn định về thời gian hơn (vì khi so sánh hai xâu bất kì hàm băm đều tốn một khoảng O(log(n)) để biến đổi, hay số lượt gọi *std::sort()* cũng là hàm có thể tính theo n) nhưng trong trường hợp này chúng chạy chậm hơn.

Với nhưng xâu văn bản lớn các giải thuật O(nlog(n)) có thể sẽ chạy nhanh hơn vì khi văn bản lớn thì các câu bị lặp lại cũng trở nên nhiều hơn nên hàm *strcmp()* cũng cần nhiều phép tính để so sánh xong hai xâu hơn. Tuy nhiên, vấn đề tốc độ sẽ được xử lý với một số các kĩ thuật lập trình mà vẫn sẽ mang lại hiệu quả tốt khi chạy ở giải thuật O(n2).

Khi giải ngược xâu BWT ta sẽ cài đặt thuật toán O(n) nên không cần phải tối ưu tốc độ.

### 2.3.3. Tăng tốc độ nén xâu lớn:

Để tăng tốc độ nén các xâu quá lớn ta sẽ dùng kĩ thuật multithreading.

Khi xâu văn bản quá lớn, chúng ta có thể thực hiện chia xâu văn bản ra thành các đoạn nhỏ hơn và thực hiện BWT và nén độc lập với nhau.

Máy tính cá nhân bình thường có thể chạy song song tới 8 luồng cùng một lúc nên thời gian khi thực hiện nén song song cũng sẽ được tăng nên đến 8 lần.

Ví dụ khi nén một file văn bản 4mb, nếu chạy một luồng duy nhất sẽ tốn đến 20 giây ở giải thuật O(n2) và hơn 300 giây ở các giải thuật O(nlog(n)) thì khi thực hiện nén với 4 luồng song song chỉ mất cỡ 4 giây ở giải thuật O(n2) và cỡ 60 giây ở các giải thuật O(nlog(n)).

### 2.3.4. Kết quả thử nghiệm:

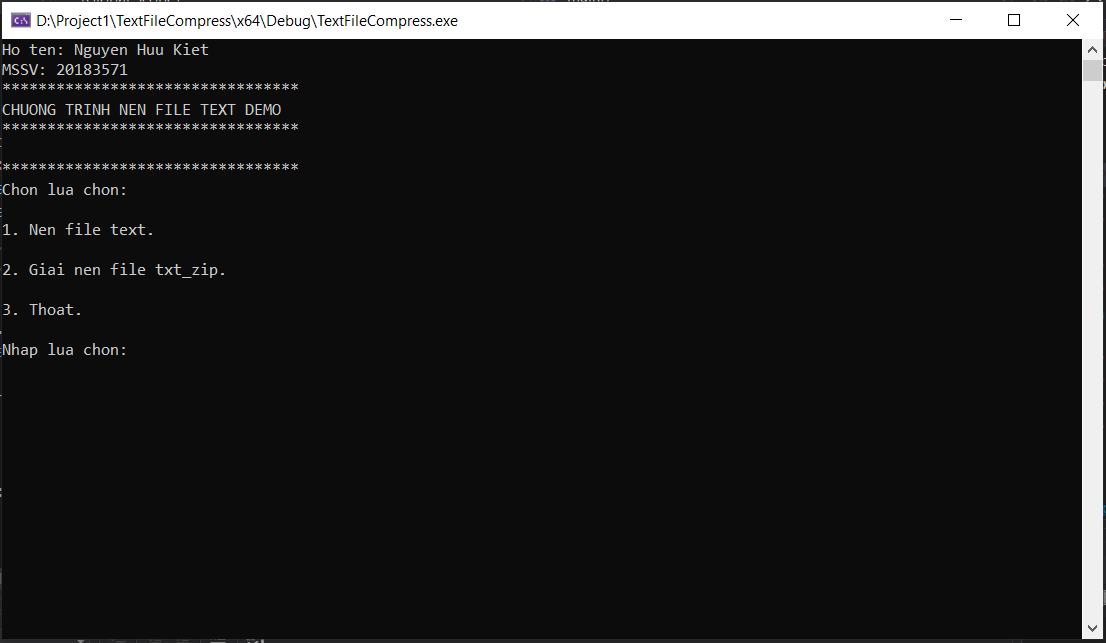
Khi thực hiện nén file văn bản, chương trình đã chạy đúng, có khả năng nén và giải nén không mất mát thông tin trong thời gian tương đối nhanh.

Dung lượng của file sau khi nén thường được giảm xuống còn một nửa so với dung lượng file ban đầu, tùy thuộc vào nội dung file.

### 2.3.5. Chương trình demo:

Mã nguồn của chương trình có thể được tìm thấy ở link sau:

<https://github.com/KietNguyen10112000/Project1>



Hình 1 Chương trình demo

# 3. Kết luận:

Mã hóa và nén thông tin là một trong các lĩnh vực nghiên cứu của ngành khoa học máy tính, mã hóa và nén thông tin mang lại cho ta rất nhiều lợi ích như giảm được lưu lượng mang, băng thông khi trao đổi thông tin, giảm được lưu lượng lưu trữ trên ổ đĩa, bảo mật và đảm bảo toàn vẹn thông tin,…

Kết quả đạt được, đã hiểu được phép biến đổi BWT và ứng dụng, đã biết cách tối ưu để giải thuật biến đổi BWT chạy hiệu quả hơn, đã biết cách sử dụng C++ cùng với môi trường phát triển Microsoft Visual Studio Community 2019.

Kết quả chương trình, chương trình chạy thành công tuy nhiên vẫn còn tương đối chậm, trong tương lai chương trình sẽ còn phải cải tiến nhiều để đạt hiệu quả tốt hơn.

# Danh mục tài liệu tham khảo:

1. <https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform>
2. <https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/>
3. <https://www.geeksforgeeks.org/inverting-burrows-wheeler-transform/>
4. <https://viblo.asia/p/lam-quen-voi-multithreading-trong-c-qm6RWQYXGeJE>
5. <https://vnoi.info/wiki/algo/data-structures/suffix-array.md>
6. <http://www.giaithuatlaptrinh.com/?p=488>
7. <http://blogthuattoan.blogspot.com/2014/06/ban-ve-mang-hau-to-suffix-array.html>