Trường Đại học Khoa học tự nhiên - ĐHQG HCM

Khoa Công nghệ thông tin

BÁO CÁO ĐỒ ÁN

Cấu trúc dữ liệu và giải thuật

Tìm hiểu thuật toán Burrows–Wheeler Transform

Ngày 05 tháng 01 năm 2025

Lớp 23CTT3

23120222 - Lê Thành Công

23120232 - Lê Thượng Đế

23120262 - Tống Dương Thái Hoà

23120264 - Nguyễn Phúc Hoàng

Giảng viên hướng dẫn

Ths. Văn Chí Nam

Thầy Trần Hoàng Quân

Mục lục

1 Giới thiệu

1.1 Chủ đề

Burrows-Wheeler Transform (BWT) là một thuật toán quan trọng trong lĩnh vực Khoa học Máy tính, được phát minh bởi Michael Burrows và David Wheeler vào năm 1994. Với khả năng tổ chức các ký tự giống nhau lại gần nhau BWT đóng vai trò chủ chốt trong các ứng dụng như nén dữ liệu (ví dụ: bzip2), tìm kiếm mẫu, và xử lý chuỗi.

1.2 Mục tiêu

Dự án có các mục tiêu cụ thể như sau:

- Hiểu rõ nguyên lý hoạt động của thuật toán BWT.
- Chuyển đổi chuỗi giữa dạng thường và dạng BWT.
- Thực hiện tìm kiếm mẫu trong đoạn văn dựa trên dạng BWT.
- Triển khai và đánh giá ứng dụng của BWT trong nén dữ liệu.

1.3 Phương pháp nghiên cứu

- **Nghiên cứu lý thuyết:** Tìm hiểu các tài liệu, bài báo và sách giáo khoa liên quan đến thuật toán BWT.
- **Thực nghiệm lập trình:** Xây dựng các thuật toán và kiểm tra trên các tập dữ liệu mẫu.
- **Phân tích hiệu suất:** Đánh giá độ phức tạp về thời gian và không gian của các thuật toán.
- **Ứng dụng mở rộng:** Thực hiện nén dữ liệu sử dụng BWT.

1.4 Thành tựu đạt được

- Hiểu rõ cách hoạt động của thuật toán BWT và cách đảo ngược nó.
- Triển khai thành công các chức năng chính:
  - Chuyển đổi chuỗi sang dạng BWT và ngược lại.
  - Tìm kiếm mẫu trong đoạn văn sử dụng BWT.
- Thực hiện thành công chức năng nén dữ liệu sử dụng BWT.
- Viết báo cáo chi tiết và trình bày kết quả thông qua video minh họa.

1.5 Lời cảm ơn

Chúng tôi xin gửi lời cảm ơn chân thành đến thầy cô và các bạn/anh/chị đã đồng hành cùng chúng tôi trong suốt quá trình thực hiện dự án này. Chúng tôi đặc biệt cảm kích sự hỗ trợ từ **Ths. Văn Chí Nam** và **Thầy Trần Hoàng Quân** với những hướng dẫn tận tình và kiến thức thầy đã truyền đạt góp phần giúp chúng tôi vượt qua những khó khăn và thử thách trong suốt quá trình thực hiện dự án. Đây không chỉ là cơ hội để chúng tôi nâng cao kiến thức chuyên môn mà còn là dịp để học hỏi những bài học quý giá. Với kiến thức và kinh nghiệm còn hạn hẹp, trong quá trình thực hiện sẽ không tránh khỏi các sai sót. Rất mong nhận được sự quan tâm và góp ý xây dựng đến từ thầy và các bạn đồng học để nhóm rút kinh nghiệm và hoàn thiện đồ án tốt hơn.

2 Phân tích thuật toán

2.1 Chuyển đổi một chuỗi S thành dạng BWT

2.1.1 Ý tưởng

Quá trình chuyển đổi một chuỗi s sang dạng BWT dựa trên việc tạo ra tất cả các vòng xoay của chuỗi, sắp xếp chúng theo thứ tự từ điển, và sau đó lấy các ký tự cuối cùng từ mỗi vòng xoay. Ký tự `$` (kết thúc chuỗi) được thêm vào để xác định điểm kết thúc.

2.1.2 Các bước thực hiện

Bước 1: Thêm ký tự `$` vào cuối chuỗi s để đánh dấu điểm kết thúc.

Bước 2: Sinh tất cả các vòng xoay của chuỗi s bằng cách di chuyển ký tự đầu tiên đến cuối chuỗi.

Bước 3: Sắp xếp các vòng xoay theo thứ tự từ điển.

Bước 4: Tạo BWT bằng cách lấy ký tự cuối cùng của mỗi vòng xoay đã sắp xếp.

2.1.3 Ví dụ minh họa

Giả sử chuỗi $s =$ `apple`. Các bước chuyển đổi sang dạng BWT được trình bày trong bảng dưới đây:

1. Đầu vào	2. Tất cả vòng xoay	3. Sắp xếp theo thứ tự từ điển	4. Lấy ký tự ở cột cuối	5. Đầu ra
apple$	apple$	$apple	e	e$lppa
	pple$a	apple$	$	
	ple$ap	e$appl	l	
	le$app	le$app	p	
	e$appl	ple$ap	p	
	$apple	pple$a	a	

2.2 Đảo ngược chuỗi dạng BWT L để lấy lại chuỗi gốc

Một khi dữ liệu đã được chuyển đổi thành dạng BWT, một trong những bài toán quan trọng là làm thế nào để đảo ngược quá trình này và phục hồi lại chuỗi ban đầu. Phép đảo ngược BWT có thể thực hiện mà không cần phải lưu trữ toàn bộ các vòng xoay của chuỗi, mà chỉ cần sử dụng thông tin từ các ký tự cuối cùng của mỗi vòng xoay đã được sắp xếp.

2.2.1 Ý tưởng

Để đảo ngược phép biến đổi BWT và phục hồi lại chuỗi ban đầu, ta thực hiện theo các bước sau:

Bước 1: Bắt đầu với chuỗi L (dạng BWT), là chuỗi các ký tự cuối cùng của các vòng xoay đã được sắp xếp.

Bước 2: Tạo một danh sách các cặp ký tự và vị trí của chúng trong chuỗi L, sau đó sắp xếp lại theo thứ tự từ điển của ký tự.

Bước 3: Xây dựng chuỗi gốc bằng cách lần lượt lấy ký tự ở vị trí được chỉ định bởi các cặp trong danh sách đã sắp xếp, cho đến khi gặp ký tự kết thúc ($).

2.2.2 Ví dụ minh hoạ

Dưới đây là các bước cụ thể để đảo ngược phép biến đổi BWT cho chuỗi $L =$ `e$lppa`, với ví dụ từ câu hỏi trước:

Bước 1 - Khởi tạo: Ta bắt đầu với chuỗi $L =$ `e$lppa`.

Bước 2 - Tạo cặp ký tự và chỉ số: Tạo danh sách các cặp ký tự và chỉ số của chúng trong chuỗi L:

$$\text{couple} = \{(e,0), (\$,1), (l,2), (p,3), (p,4), (a,5)\}$$

Bước 3 - Sắp xếp các cặp: Sắp xếp các cặp ký tự theo thứ tự từ điển:

$$\text{sorted\_couple} = \{(\$,1), (a,5), (e,0), (l,2), (p,3), (p,4)\}$$

Bước 4 - Khôi phục chuỗi gốc: Dùng các cặp đã sắp xếp và chỉ số vị trí để khôi phục chuỗi ban đầu. Bắt đầu với ký tự $ và lần lượt lấy ký tự tiếp theo tại vị trí chỉ định: Ví dụ: Khởi tạo chuỗi decoding bằng rỗng

Lần lặp thứ	Giá trị decode	Mô tả
1	""	Tạo chuỗi decoding
2	"a"	Bắt đầu từ vị trí của '$' là 0, giá trị cặp của '$' là 1, cộng vào chuỗi decoding giá trị sorted_couple[1]
3	"ap"	Tiếp tục từ vị trí của 'a' là 1, giá trị cặp của 'a' là 5, cộng vào chuỗi decoding giá trị sorted_couple[5]
4	"app"	Tiếp tục từ vị trí của 'p' là 5, giá trị cặp của 'p' là 4, cộng vào chuỗi decoding giá trị sorted_couple[4]
5	"appl"	Tiếp tục từ vị trí của 'p' là 4, giá trị cặp của 'p' là 3, cộng vào chuỗi decoding giá trị sorted_couple[3]
6	"apple"	Tiếp tục từ vị trí của 'l' là 3, giá trị cặp của 'l' là 2, cộng vào chuỗi decoding giá trị sorted_couple[2]
7	"apple"	Tiếp tục từ vị trí của 'e' là 2, giá trị cặp của 'l' là 0, nhận thấy sorted_couple[0] bằng '$' thì kết thúc quá trình decode.

Bước 5 - Kết quả: Sau các bước trên, ta phục hồi lại chuỗi ban đầu $s =$ `apple`.

2.3 Phân tích độ phức tạp của việc xây dựng ma trận BWT

Ma trận BWT có thể được xây dựng từ một chuỗi ký tự s có độ dài n. Quy trình này bao gồm việc tạo ra tất cả các phép quay cyclic của chuỗi s, sau đó sắp xếp các phép quay này (luôn đảm bảo rằng tuỳ vào cấu trúc của từng chuỗi sẽ luôn chọn được hàm sort tương ứng để độ phức tạp luôn là $O(n \log n)$ với k phụ thuộc vào cách cài đặt thuật toán, ở trong phần lập trình, nhóm chúng tôi lựa chọn hàm `sort` có sẵn của thư viện `algorithm`), và sau đó trích xuất cột cuối cùng của các phép quay đã sắp xếp để tạo ra BWT. Độ phức tạp thời gian và không gian của việc xây dựng ma trận BWT phụ thuộc vào thuật toán sử dụng.

Dưới đây là sự phân tích chi tiết về độ phức tạp thời gian và không gian trong việc xây dựng ma trận BWT:

2.3.1 Cách Tiếp Cận Đơn Giản (Naive Approach)

Cách tiếp cận đơn giản nhất để xây dựng ma trận BWT là tạo ra tất cả các phép quay cyclic của chuỗi s, sau đó sắp xếp chúng và trích xuất cột cuối cùng.

Độ Phức Tạp Thời Gian:

- **Tạo Các Phép Quay Cyclic:** Để tạo các phép quay cyclic của chuỗi s, chúng ta cần tạo ra tất cả các phép quay của chuỗi này. Mỗi phép quay có độ dài n, và có tổng cộng n phép quay, vậy tổng thời gian để tạo ra các phép quay cyclic là $O(n^2)$ (vì mỗi phép quay có độ dài n, và có n phép quay).
- **Sắp Xếp Các Phép Quay:** Sau khi tạo ra tất cả các phép quay, chúng ta cần sắp xếp chúng theo thứ tự từ điển. Để sắp xếp n chuỗi có độ dài n, chúng ta cần sử dụng thuật toán sắp xếp dựa trên so sánh. Độ phức tạp thời gian của sắp xếp là $O(n \log n)$, và vì mỗi phép quay có độ dài n, việc so sánh chúng sẽ mất $O(n)$ thời gian cho mỗi cặp phép quay. Do đó, tổng thời gian sắp xếp là $O(n^2 \log n)$.
- **Tạo Ma Trận BWT:** Cuối cùng, sau khi sắp xếp các phép quay, cột cuối cùng của các phép quay đã sắp xếp sẽ là ma trận BWT. Trích xuất cột này có độ phức tạp thời gian $O(n)$.

Vì vậy, độ phức tạp thời gian tổng thể của phương pháp này là:

$$O(n^2) \text{ (tạo các phép quay)} + O(n^2 \log n) \text{ (sắp xếp các phép quay)} = O(n^2 \log n)$$

Độ Phức Tạp Không Gian:

- Để lưu trữ tất cả các phép quay cyclic của chuỗi s, chúng ta cần $O(n^2)$ không gian, vì mỗi phép quay có độ dài n, và có n phép quay.
- Ngoài ra, chúng ta cũng cần không gian để lưu trữ ma trận BWT, điều này cần $O(n)$ không gian.

Vì vậy, tổng độ phức tạp không gian là $O(n^2)$.

2.3.2 Cách Tiếp Cận Hiệu Quả (Sử Dụng Mảng hậu tố (Suffix Array) hoặc Cây hậu tố (Suffix Tree))

Một phương pháp hiệu quả hơn để xây dựng BWT là sử dụng **mảng hậu tố** (Suffix Array) hoặc **cây hậu tố** (Suffix Tree). Phương pháp này không cần tạo ra tất cả các phép quay cyclic mà chỉ cần làm việc với các hậu tố của chuỗi s, chính vì thế việc xây dựng mảng hậu tố tương tự như việc sắp xếp các chuỗi phép quay.

Độ Phức Tạp Thời Gian:

- **Xây Dựng mảng hậu tố:** Mảng hậu tố là một dãy các chỉ số chỉ ra vị trí của các hậu tố của chuỗi s sau khi được sắp xếp. mảng hậu tố có thể được xây dựng trong thời gian tuyến tính $O(n \log n)$ bằng cách sử dụng các thuật toán như thuật toán sắp xếp theo độ dài của hậu tố hoặc thuật toán sắp xếp bước nhảy (Induced Sorting) [1] [2].
- **Tạo Ma Trận BWT:** Khi đã có mảng hậu tố, BWT có thể được xây dựng bằng cách duyệt qua các chỉ số trong mảng hậu tố và lấy ký tự trước mỗi hậu tố. Vì vậy, thời gian để trích xuất cột cuối cùng của mảng hậu tố và tạo ma trận BWT là $O(n)$.

Do đó, tổng độ phức tạp thời gian để xây dựng BWT là:

$$O(n \log n) \text{ (xây dựng mảng hậu tố)} + O(n) \text{ (tạo BWT từ mảng hậu tố)} = O(n \log n)$$

Độ Phức Tạp Không Gian:

- Để lưu trữ mảng hậu tố, chúng ta cần $O(n)$ không gian, vì mảng hậu tố chỉ chứa n chỉ số.
- Ngoài ra, chúng ta cần $O(n)$ không gian để lưu trữ ma trận BWT.

Do đó, tổng độ phức tạp không gian là $O(n)$.

2.3.3 Tóm Tắt:

- **Trường Hợp Tốt Nhất (Best Case):** Đối với cách tiếp cận hiệu quả sử dụng mảng hậu tố hoặc cây hậu tố, độ phức tạp thời gian và không gian là:

$$O(n \log n) \text{ (thời gian)} \quad \text{và} \quad O(n) \text{ (không gian)}$$

- **Trường Hợp Tệ Nhất (Worst Case):** Đối với cách tiếp cận đơn giản, độ phức tạp thời gian và không gian là:

$$O(n^2 \log n) \text{ (thời gian)} \quad \text{và} \quad O(n^2) \text{ (không gian)}$$

2.3.4 Kết Luận:

Cài đặt thuật toán BWT dựa trên **mảng hậu tố** hoặc **cây hậu tố** có độ phức tạp thời gian và không gian tốt hơn rất nhiều so với cách tiếp cận đơn giản, đặc biệt đối với các chuỗi có độ dài lớn. Trong thực tế, phương pháp sử dụng mảng hậu tố hoặc cây hậu tố là sự lựa chọn tối ưu vì hiệu suất vượt trội của chúng đối với các chuỗi dài và nhóm chúng tôi đã áp dụng cách cài đặt này vào tính năng **Tìm kiếm mẫu** để tối ưu việc quá trình tìm kiếm.

2.4 Mối liên hệ giữa BWT, mảng hậu tố và cây hậu tố

Có một mối liên hệ chặt chẽ giữa Biến đổi Burrows-Wheeler (BWT) và các mảng hậu tố (suffix arrays) cũng như cây hậu tố (suffix trees). BWT có thể được xem như một biểu diễn thứ tự của các hậu tố trong một chuỗi khi chúng được sắp xếp theo thứ tự từ điển.

2.4.1 Mối liên hệ với mảng hậu tố:

- BWT có thể được suy ra trực tiếp từ mảng hậu tố của một chuỗi. [3]
- Mảng hậu tố của một chuỗi T là một mảng SA, trong đó $SA[i]$ chứa vị trí bắt đầu của hậu tố nhỏ nhất thứ i theo thứ tự từ điển của T[4].
- BWT L của T có thể được định nghĩa thông qua SA như sau[1]:
  - $L[i] = T[SA[i] - 1]$ nếu $SA[i] > 0$
  - $L[i] = \$$ nếu $SA[i] = 0$
- Bản chất, BWT được tạo ra bằng cách lấy các ký tự ngay trước các hậu tố đã sắp xếp trong mảng hậu tố. Nếu hậu tố bắt đầu từ đầu chuỗi, ký tự đặc biệt "$"sẽ được sử dụng [4].

2.4.2 Mối liên hệ với cây hậu tố:

- Cây hậu tố là các cấu trúc dữ liệu biểu diễn tất cả các hậu tố của một chuỗi một cách gọn gàng.
- Mảng hậu tố có thể dễ dàng được xây dựng từ cây hậu tố bằng cách duyệt theo chiều sâu (DFS) và liệt kê các vị trí bắt đầu của các hậu tố được tìm thấy tại các nút lá[4].
- Do đó, BWT cũng có thể được suy ra gián tiếp từ cây hậu tố bằng cách đầu tiên thu được mảng hậu tố.

2.4.3 Nhận định chính:

BWT, mảng hậu tố và cây hậu tố đều biểu diễn các hậu tố của một chuỗi theo thứ tự từ điển. BWT đạt được điều này bằng cách hoán vị các ký tự của chuỗi dựa trên ngữ cảnh trước đó của chúng, trong khi mảng hậu tố và cây hậu tố lưu trữ trực tiếp các vị trí bắt đầu của các hậu tố đã sắp xếp.

2.4.4 Ứng dụng trong tìm kiếm mẫu:

- Mối liên hệ với mảng hậu tố và cây hậu tố làm cho BWT hữu ích trong việc tìm kiếm mẫu, vì các cấu trúc này cho phép thực hiện các thao tác tìm kiếm hiệu quả trong văn bản.
- Tính chất sắp xếp của BWT cho phép tìm kiếm mẫu nhanh bằng các kỹ thuật như tìm kiếm ngược (backward search) với ánh xạ LF, khai thác mối quan hệ giữa cột đầu tiên (F) và cột cuối cùng (L) của ma trận BWT [4].

Mối liên hệ chặt chẽ giữa BWT, mảng hậu tố và cây hậu tố nhấn mạnh cách mà BWT mã hóa thông tin một cách thông minh về thứ tự các hậu tố trong một chuỗi, cho phép nén dữ liệu hiệu quả và khả năng tìm kiếm mẫu nhanh chóng.

2.5 Bài toán Tìm kiếm mẫu (Pattern Searching)

Thuật toán tìm kiếm mẫu dựa trên Burrows-Wheeler Transform (BWT) sử dụng kỹ thuật *Backward Search* kết hợp với *LF-Mapping*. Với thuật toán này, ta có thể tìm kiếm mẫu p trong chuỗi gốc s của L mà không cần biết trực tiếp chuỗi s. Dưới đây là giải thích chi tiết và các bước thực hiện:

2.5.1 Tổng quan thuật toán

Khi một chuỗi s được biến đổi thành Burrows-Wheeler Transform (BWT), ta thu được một chuỗi L với các đặc tính:

- **Cột đầu tiên:** Là các ký tự của s đã được sắp xếp theo thứ tự từ điển.
- **Cột cuối cùng:** Là các ký tự của BWT.

Với BWT, ta không cần khôi phục chuỗi gốc s để tìm kiếm mẫu. Thay vào đó, thuật toán sử dụng các bảng dữ liệu hỗ trợ để duyệt ngược từ mẫu p.

2.5.2 Các bước thực hiện

Bước 1 - Xây dựng bảng dữ liệu:[5]

- **Mảng tích lũy (Cumulate):** Mảng này xác định vị trí bắt đầu của mỗi ký tự trong cột đầu tiên của ma trận BWT đã sắp xếp. Mảng tích lũy giúp thu hẹp phạm vi tìm kiếm trong BWT một cách nhanh chóng.

- **Bảng tần suất (Occurrence Table - *occ*):** Bảng này ghi nhận số lần xuất hiện của mỗi ký tự tại từng vị trí trong cột cuối. Nó hỗ trợ việc xác định số lần xuất hiện và vị trí của một ký tự trong phạm vi tìm kiếm.
- **Mảng Suffix Array:** Mảng này ánh xạ từng vị trí trong cột cuối của BWT đến vị trí tương ứng trong chuỗi gốc s. Điều này giúp xác định chính xác vị trí của mẫu trong chuỗi gốc s.

Bước 2 - Duyệt ngược chuỗi mẫu p:[5]

- Xét từng ký tự của mẫu p từ phải sang trái, ký tự hiện tại gọi là c. Đặt $B = 0$ và $E = \text{length}(s)$ (đại diện cho cột cuối cùng của BWT).
- Sử dụng bảng *occ* để cập nhật phạm vi tìm kiếm $[B, E]$ cho ký tự hiện tại c:
  - Nếu $occ[c][B] = occ[c][E+1]$, nghĩa là không có ký tự c nào trong khoảng $[B, E]$, ta dừng tìm kiếm vì không tìm thấy p trong chuỗi gốc.
  - Ngược lại, nếu có các ký tự c trong phạm vi $[B, E]$, ta biết được số ký tự c trong đoạn này và bao nhiêu ký tự c đã xuất hiện trước vị trí B.
  - Cập nhật phạm vi tìm kiếm:

    $B = \text{cumulate}[c] + occ[c][B]$: Cập nhật B thành vị trí bắt đầu tìm kiếm mới, là vị trí bắt đầu của ký tự c thứ $occ[c][B]$ trong cột đầu.

    $E = \text{cumulate}[c] + occ[c][E+1] - 1$: Cập nhật E thành vị trí kết thúc tìm kiếm mới, là vị trí kết thúc của ký tự c thứ $occ[c][E+1] - 1$ trong cột đầu.

Bước 3 - Kết luận: Khi duyệt hết mẫu p, phạm vi $[B, E]$ còn lại xác định các vị trí trong mảng Suffix Array, từ đó ta có thể ánh xạ ngược về chuỗi gốc s.

2.5.3 Ví dụ minh họa

- **Chuỗi gốc:** `banana$`
- **BWT:** `annb$aa`
- **Mẫu:** `ana`

Bảng dữ liệu hỗ trợ

Ký tự	Cumulate
$	0
a	1
b	4
n	5

Bảng 1: Mảng tích lũy (Cumulate)

Các bước tìm kiếm mẫu `ana`

Bước 1: Ký tự cuối cùng của mẫu là a, $B = 0, E = 6$.

- Dựa vào mảng cumulate[a] và ma trận occ:

$$B = \text{cumulate}[a] + \text{occ}[a][0] = 1, \quad E = \text{cumulate}[a] + \text{occ}[a][7] - 1 = 3.$$

Vị trí	Cột đầu	Cột cuối	$	a	b	n	Suffix Array
0	$	a	0	0	0	0	6
1	a	n	0	1	0	0	5
2	a	n	0	1	0	1	3
3	a	b	0	1	0	2	1
4	b	$	0	1	1	2	0
5	n	a	1	1	1	2	4
6	n	a	1	3	1	2	2
7			1	3	1	2	

Bảng 2: Bảng tần suất (*occ*) với cột đầu và mảng Suffix Array.

- Phạm vi thu hẹp: $[B, E] = [1, 3]$.

Bước 2: Ký tự tiếp theo là n.

- Dựa vào cumulate$[n]$ và occ:

$$B = \text{cumulate}[n] + \text{occ}[n][1] = 5, \quad E = \text{cumulate}[n] + \text{occ}[n][4] - 1 = 6.$$

- Phạm vi thu hẹp: $[B, E] = [5, 6]$.

Bước 3: Ký tự tiếp theo là a.

- Dựa vào cumulate$[a]$ và occ:

$$B = \text{cumulate}[a] + \text{occ}[a][5] = 2, \quad E = \text{cumulate}[a] + \text{occ}[a][7] - 1 = 3.$$

- Phạm vi thu hẹp: $[B, E] = [2, 3]$.

Bước 4: Kết quả: Phạm vi cuối cùng xác định các vị trí **Suffix Array** ứng với mẫu **ana**, trong trường hợp này là **1, 3** tương ứng với "b**ana**na" và "ban**ana**".

2.5.4 Phân tích độ phức tạp thuật toán

Gọi n là kích thước của chuỗi gốc s, k là số ký tự khác nhau của chuỗi s và p là độ dài của chuỗi cần tìm kiếm.

Độ phức tạp thời gian

- **Xây dựng bảng dữ liệu:**
  - **Mảng tích lũy (Cumulate):** Duyệt qua các ký tự trong BWT và tính vị trí bắt đầu của từng ký tự, có độ phức tạp $O(n)$.
  - **Bảng tần suất (*occ*):** Xây dựng bảng *occ* với ma trận $(n + 1) \times k$, độ phức tạp $O(n \cdot k)$.
  - **Mảng Suffix Array:** Sắp xếp các chuỗi con, có độ phức tạp $O(n \log n)$.
- **Duyệt ngược chuỗi mẫu p:**
  - Với mỗi ký tự c trong mẫu p, sử dụng bảng *occ* và mảng *cumulate* để cập nhật phạm vi tìm kiếm $[B, E]$. Thao tác này có độ phức tạp $O(1)$.
  - Duyệt qua toàn bộ mẫu p, độ phức tạp là $O(p)$.

Kết luận: Tổng độ phức tạp thời gian của thuật toán là $O(n \log n + p)$. Tuy nhiên, khi tìm vị trí một chuỗi q trên cùng chuỗi gốc s, chỉ mất $O(|q|)$.

Độ phức tạp không gian

Chương trình có sử dụng thêm phần không gian cho việc xây dựng bảng dữ liệu:

- **Mảng tích lũy (Cumulate):** Cần $O(k)$ không gian.
- **Bảng tần suất (*occ*):** Cần không gian $O(n \cdot k)$.
- **Mảng Suffix Array:** Cần $O(n)$ không gian.

Kết luận: Tổng độ phức tạp không gian của thuật toán là $O(n \cdot k)$.

2.6 Ứng dụng của BWT

Bên cạnh việc tìm kiếm mẫu, Biến đổi Burrows-Wheeler Transform (BWT) còn có nhiều ứng dụng đáng chú ý khác, chủ yếu xuất phát từ khả năng **nhóm các ký tự tương tự lại với nhau**. Đặc tính này mang lại lợi ích trong các lĩnh vực như:

1. **Nén dữ liệu:** BWT đóng vai trò như một bước tiền xử lý cho nhiều thuật toán nén. Bằng cách nhóm các ký tự tương tự lại với nhau, nó tăng cường hiệu quả của các kỹ thuật như:
   - **Biến đổi Move-to-front (MTF):** MTF mã hóa một ký tự theo vị trí của nó trong danh sách, chuyển nó lên đầu sau khi mã hóa. Các chuỗi ký tự giống nhau trong BWT dẫn đến các chuỗi 0 trong kết quả của MTF, điều này giúp nén dữ liệu hiệu quả. [3]
   - **Mã hóa độ dài chuỗi (RLE):** RLE thay thế các dãy ký tự lặp lại bằng một ký tự duy nhất và số lần xuất hiện của nó. Việc nhóm các ký tự trong BWT làm tăng hiệu quả của RLE. [1]
   - **Mã hóa Huffman:** Mã hóa Huffman gán các mã ngắn hơn cho các ký tự xuất hiện thường xuyên. Việc nhóm các ký tự trong BWT làm tăng tần suất của một số ký tự, giúp mã hóa Huffman hiệu quả hơn.[3]

   Công cụ nén ‘bzip2‘ là một ví dụ nổi bật về việc sử dụng BWT cho nén dữ liệu.
2. **Nén dữ liệu gen:** Trong sinh học tin học, BWT được sử dụng để nén các cơ sở dữ liệu gen lớn. Tính chất lặp lại của các chuỗi DNA khiến chúng rất phù hợp với nén dữ liệu dựa trên BWT. Ví dụ, một nghiên cứu của Cox và cộng sự đã chứng minh rằng BWT, kết hợp với kỹ thuật nén giai đoạn hai gọi là "same-as-previous encoding", đạt tỷ lệ nén 94% trên một cơ sở dữ liệu gen.[1]
3. **Căn chỉnh chuỗi:** BWT rất quan trọng trong việc căn chỉnh hàng triệu đoạn DNA ngắn vào một bộ gen tham chiếu trong giải trình tự thế hệ tiếp theo. Các chương trình căn chỉnh dựa trên BWT giảm yêu cầu bộ nhớ so với các phương pháp truyền thống dựa trên băm.[1]
4. **Nén ảnh:** Tương tự như nén văn bản, BWT cũng có thể được áp dụng cho dữ liệu hình ảnh. Bằng cách biến đổi dữ liệu hình ảnh để tạo ra các chuỗi giá trị tương tự nhau, các kỹ thuật nén tiếp theo sẽ trở nên hiệu quả hơn. Nghiên cứu đã chỉ ra rằng các pipeline nén hình ảnh dựa trên BWT có thể vượt trội hơn các thuật toán chuẩn như Lossless JPEG và JPEG 2000. [1]

Nguyên lý đằng sau những ứng dụng này nằm ở khả năng của BWT trong việc **biến đổi dữ liệu thành một dạng làm lộ ra và làm nổi bật các sự dư thừa**. Dữ liệu đã được biến đổi sau đó sẽ trở nên dễ dàng nén hoặc xử lý hơn bằng cách sử dụng các thuật toán khác.

3 Ứng dụng Nén dữ liệu với Huffman Tree

Cây Huffman là một thuật toán nén dữ liệu không mất thông tin, nó xây dựng một bảng mã hóa với các ký tự có tần suất xuất hiện cao hơn sẽ được mã hóa bằng mã ngắn hơn. Ngược lại, những ký tự có tần suất xuất hiện thấp sẽ có mã dài hơn. Cây Huffman được sử dụng để giảm kích thước của dữ liệu bằng cách tối ưu hóa việc mã hóa các ký tự trong dữ liệu.

3.1 Các bước thực hiện:

Bước 1 - Run-Length Encoding (RLE): Đầu vào của quá trình này là một chuỗi ký tự (ví dụ: chuỗi BWT - Burrows-Wheeler Transform). Mục đích của RLE là thay thế các chuỗi con giống nhau (ví dụ, một chuỗi nhiều ký tự giống nhau liên tiếp) bằng ký tự đó và số lần lặp lại của nó.
Ví dụ: Dữ liệu đầu vào là `aaaaaabb`. Sau khi mã hóa RLE: `a5 b2`

Bước 2 - Xây dựng Cây Huffman: Cây Huffman được xây dựng từ các đoạn chuỗi được mã hóa bằng RLE (tức là từ các "run"). Đầu tiên, ta tạo một bảng tần suất cho các chuỗi này và sau đó dùng thuật toán Huffman để xây dựng cây. Cây này có các nhánh chứa các "run"ký tự, và độ dài của các mã được gán sẽ phụ thuộc vào tần suất xuất hiện của chúng.
Các bước xây dựng Cây Huffman:

- Gộp hai nút có tần số nhỏ nhất thành một nút cha mới.
- Nút cha này sẽ có tần số bằng tổng tần số của hai nút con.
- Tiếp tục lặp đến khi chỉ còn lại một nút duy nhất (nút gốc của cây).

Bước 3 - Mã hóa Huffman: Sau khi cây Huffman đã được xây dựng, ta sẽ mã hóa chuỗi RLE thành các chuỗi mã nhị phân bằng cách duyệt cây Huffman:
Gán bit cho nhánh cây:

- Đi sang trái, gán 0.
- Đi sang phải, gán 1.

Duyệt cây để tìm mã cho từng ký tự:

- Mỗi lá của cây (node không có con) đại diện cho một ký tự hoặc chuỗi ("run").
- Mã Huffman của một ký tự là chuỗi các bit được hình thành từ gốc đến lá tương ứng.

Mỗi chuỗi "run"được thay thế bằng mã nhị phân tương ứng với nó trong cây Huffman.

Bước 4 - Lưu trữ cây Huffman: Sau khi mã hóa dữ liệu, cây Huffman cũng được lưu trữ vào tệp để có thể giải mã sau này. Cây Huffman được lưu trữ dạng nhị phân (lưu thông qua DFS).

Bước 5 - Giải nén: Giải nén sẽ tiến hành các bước ngược lại:

(a) Đọc dữ liệu đã mã hóa từ tệp.

(b) Giải mã theo cây Huffman.

(c) Giải mã RLE để phục hồi lại chuỗi ban đầu.

(d) Giải mã Burrows-Wheeler Transform (BWT) để lấy lại dữ liệu gốc.

3.2 Ví dụ minh họa:

Giả sử chúng ta có chuỗi dữ liệu sau:

input: `abcabcabcababb`

Bước 1 - Burrows-Wheeler Transform (BWT): BWT của chuỗi có là:

`bcbcc$baaaaabbb`

Bước 2 - Run-Length Encoding (RLE): Mã hóa RLE cho chuỗi BWT:

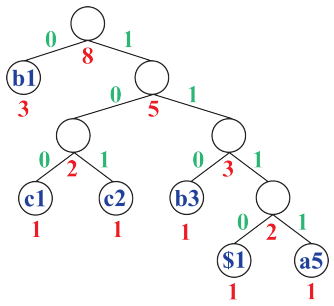
`b1 c1 b1 c2 $1 b1 a5 b3`

Bước 3 - Xây dựng Cây Huffman: Cây Huffman được xây dựng từ các chuỗi mã hóa bằng RLE:

- Tạo bảng tần số xuất hiện: b1 - 3, c1 - 1, c2 - 1, $1 - 1, a5 - 1, b3 - 1.
- Xây dựng cây Huffman:

[c1](1), [c2](1), [$1](1), [a5](1), [b3](1), [b1](3)
[$1](1), [a5](1), [b3](1), [c1 + c2](2), [b1](3)
[b3](1), [$1 + a5](2), [c1 + c2](2), [b1](3)
[c1 + c2](2), [b3 + ($1 + a5)](3), [b1](3)
[b1](3), [(c1 + c2) + b3 + ($1 + a5)](5),
[b1 + (c1 + c2) + b3 + ($1 + a5)](8),

Cây Huffman sau khi xây dựng xong:



Màu đỏ: Tần suất xuất hiện.
Màu xanh dương: Chuỗi "run".
Màu xanh lá: Mã nhị phân tương ứng.

Bước 4 - Mã hóa Huffman: Mã hóa các chuỗi trên thành mã nhị phân.

- **b1:** 0
- **c1:** 100
- **c2:** 101
- **b3:** 110
- **$1:** 1110
- **a5:** 1111

Bước 5 - Lưu trữ Cây Huffman: Cây Huffman được lưu vào tệp theo thứ tự RLE tương ứng:
b1{0} c1{100} b1{0} c2{101} $1{1110} b1{0} a5{1111} b3{110}
Dữ liệu nén được lưu trữ dưới mã nhị phân: `01000101111001111110`

3.3 Đánh giá độ phức tạp:

- **Run-Length Encoding (RLE):** Độ phức tạp thời gian là $O(n)$, với n là chiều dài của chuỗi đầu vào. Ta phải quét qua chuỗi một lần để tạo các đoạn run.
- **Xây dựng Cây Huffman:** Độ phức tạp là $O(n \log n)$, trong đó n là số lượng các đoạn chuỗi run (với mỗi đoạn tương ứng với một ký tự trong tệp).
- **Mã hóa Huffman:** Sau khi xây dựng cây, độ phức tạp của việc mã hóa là $O(n)$, với n là chiều dài của chuỗi sau khi đã mã hóa bằng RLE.
- **Giải nén:** Quá trình giải nén có độ phức tạp tương đương với nén, $O(n)$, vì phải duyệt qua chuỗi mã hóa để giải mã theo cây Huffman và sau đó giải mã RLE.

Tổng thể, quá trình nén và giải nén có độ phức tạp chính là $O(n \log n)$, trong đó n là kích thước của dữ liệu ban đầu.

3.4 Ưu điểm và nhược điểm:

Ưu điểm

- Tối ưu hóa không gian lưu trữ, đặc biệt là đối với các dữ liệu có tần suất lặp lại cao.
- Cây Huffman đảm bảo rằng dữ liệu nén đạt hiệu quả cao.

Nhược điểm

- Quá trình nén có thể tốn thời gian với dữ liệu rất lớn.
- Cây Huffman phải được lưu trữ cùng với dữ liệu, làm tăng thêm kích thước tệp nén.

Thông qua phương pháp này, chúng ta có thể nén dữ liệu một cách hiệu quả, sử dụng kết hợp giữa các kỹ thuật như RLE và cây Huffman.

4 Tổ chức dự án và các ghi chú

4.1 Tổ chức dự án

Trong dự án này, chúng tôi đã sắp xếp các tệp mã của mình bằng cách sử dụng quy chuẩn thống nhất chung như sau: ***Các tập tin trong thư mục code/***

- **include/*.hpp**: Những tệp tiêu đề ở đây sẽ khai báo hàm cho từng loại chương trình chính.
- **algorithm/*.cpp**: Những tệp nguồn ở đây sẽ định nghĩa cho từng chức năng cần thiết.
- **main.cpp**: Tệp đặc biệt dùng để liên kết và xử lý chương trình.
- ***.txt**: Các tệp .txt từ yêu cầu; Lưu ý rằng các file này phải được đặt cùng cấp với file .exe.

4.2 Hướng các biên dịch và chạy chương trình

Để compile file dùng để chạy command, đồ án sử dụng g++ với cú pháp sau: Mô tả từng bước:

Bước 1: Di chuyển vào thư mục tổng (thư mục code/)

Bước 2: Nhập lệnh sau vào terminal hoặc command prompt:

g++ algorithm/*.cpp main.cpp -o main.exe

Trong cú pháp trên:

- **algorithm/*.cpp**: Compile các file cpp trong thư mục algorithm - Các file hàm thuật toán về BWT được yêu cầu.
- **main.cpp**: Compile file main.cpp dùng cho việc hỗ trợ liên kết và xử lý chương trình.
- **main.exe**: Kết quả quá trình compile các file trên sẽ được viết vào file **main.exe**.

Ở đây người dùng có thể sử dụng tên file khác để đặt cho file exe dùng để chạy chương trình.

Bước 3: Chạy chương trình bằng lệnh **./main.exe** cùng 3 loại lệnh đã được yêu cầu và 2 loại lệnh tự phát triển, gồm:

- `<yourprogram.exe> -c input.txt output.txt [--bwt]`:
  input.txt: Tệp văn bản chứa các chuỗi cần chuyển đổi. Mỗi chuỗi nằm trên một dòng;
  output.txt: Tệp văn bản chứa chuỗi đã chuyển đổi, mỗi chuỗi nằm trên một dòng;
  --bwt: Nếu tham số này được cung cấp, chương trình sẽ chuyển đổi chuỗi từ dạng thông thường sang dạng BWT. Nếu không, chương trình chuyển từ dạng BWT về dạng thông thường.

- `<yourprogram>.exe -p paragraph.txt patterns.txt output.txt`:
  paragraph.txt: Tệp văn bản chứa đoạn văn, tất cả nội dung nằm trên một dòng;
  patterns.txt: Tệp văn bản chứa các mẫu cần tìm kiếm, mỗi mẫu trên một dòng;
  output.txt: Tệp văn bản chứa kết quả tìm kiếm.

- `main.exe -cprs input.txt compressed.bin`:
  input.txt: Tệp ban đầu cần nén;
  compressed.bin: Tệp sau khi nén.

- `main.exe -decprs compressed.bin output.txt`:
  compressed.bin: Tệp đã nén;
  output.txt: Tệp kết quả sau khi giải nén.

Lưu ý: Nên biên dịch bằng C++17 trở lên vì có sử dụng một số cú pháp của C++17 và C++20.

Tài liệu

[1] wikipedia. *Burrows-Wheeler Transform*. https://en.wikipedia.org/wiki/Burrows-Wheeler_transform. Độ phức tạp về thời gian khi dùng mảng/ cây tiền tố; Ứng dụng RLE, Nén dữ liệu Gen; Căn chỉnh chuỗi; Nén ảnh. Truy cập: 03-01-2025.

[2] helsinki. *Induced Sorting*. https://www.cs.helsinki.fi/u/tpkarkka/opetus/11s/spa/lecture11.pdf. Thuật toán sắp xếp Induced. Truy cập: 04-01-2025.

[3] CMU School of Computer Science. *Burrows-Wheeler Transform CMSC 423*. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf. Biến đổi MTF; Mã hoá Huffman. Slide 18. Truy cập: 03-01-2025.

[4] CMU School of Computer Science. *Burrows-Wheeler Transform CMSC 423*. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf. Tham khảo mối quan hệ giữa BWT và Suffix Tree. Slide 15. Truy cập: 03-01-2025.

[5] CMU School of Computer Science. *Burrows-Wheeler Transform CMSC 423*. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf. Bài toán tìm kiếm mẫu. Slide 11. Truy cập: 03-01-2025.

[6] CMU School of Computer Science. *Burrows-Wheeler Transform CMSC 423*. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf. Lấy ví dụ tham khảo. Slide 6. Truy cập: 03-01-2025.

[7] geeksforgeeks. *Burrows – Wheeler Data Transform Algorithm*. https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/. Tham khảo cách convert từ chuỗi thường sang dạng BWT. Truy cập: 02-01-2025.

[8] geeksforgeeks. *Inverting the Burrows – Wheeler Transform*. https://www.geeksforgeeks.org/inverting-burrows-wheeler-transform/. Tham khảo cách đảo ngược từ dạng BWT về chuỗi thường. Truy cập: 02-01-2025.

[9] Niema Moshiri. *Advanced Data Structures: Burrows-Wheeler Transform (BWT)*. https://www.youtube.com/watch?v=Lc-ACiJIrnM&t=219s&ab_channel=NiemaMoshiri. Tham khảo cách hoạt động của BWT. Truy cập: 02-01-2025.