TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

——————— * ———————

PROJECT II

Đề tài: ứng dụng thuật toán Burrows-Wheeler Transform vào nén dữ liệu

Giảng Viên hướng dẫn: TS. Trần Vĩnh Đức

Sinh viên thực hiện: Phạm Văn Nam - 20183599

Hà Nội – 6/2021

Mục lục

Danh mục hình ảnh

Chương 1: Thuật toán SACA-K

1.1. Giới Thiệu

Thuật toán SACA-K là một thuật toán có độ phức tạp O(n) dùng để xây dựng mảng hậu tố SA (suffix array) của một chuỗi đầu vào T[0..n-1]. Thuật toán SACA-K được cải tiến từ thuật toán SA-IS cũng là một thuật toán xây dựng mảng hậu tố SA. SACA-K đạt được tốc độ tốt hơn và không gian bộ nhớ tiêu thụ nhỏ hơn so với SA-IS. Để trình giải thuật của SACA-K, SA-IS sẽ được trình bày trước, sau đó là các cải tiến của SACA-K so với SA-IS.

1.2. Thuật toán SA-IS

1.2.1. Mã giả của thuật toán SA-IS

SA-IS(S,SA):

S is the input string on the alphabet Σ(S) has length | Σ(S) |;

SA is the output suffix array of S;

t: array [0..n − 1] of boolean;

S1: array [0..n1 − 1] of integer;

P1: array [0..n1 − 1] of integer;

B: array [0..|Σ(S)| − 1] of integer;

1. Scan S once to classify all the characters as L- or S-type into t;
2. Scan t once to find all the LMS-substrings in S into P1;
3. Induced sort all the LMS-substrings using P1 and B;
4. Name each LMS-substring in S by its bucket index to get a new shortened string S1;
5. **if** Each character in S1 is unique (|Σ(S1)| = |S1|)
6. **then**
7. Directly compute SA1 from S1;
8. **else**
9. SA-IS(S1, SA1);
10. Induce SA from SA1;
11. return;

1.2.2.Giải thích thuật toán

Dòng 1-4 mục đích là tạo ra một bài toán rút gọn. Sau đó bài toán rút gọn này được giải đệ quy ở dòng 5-9. Dòng 10 thực hiện đưa ra lời giải cuối cùng của bài toán đầu.

1.2.2.1. Bài toán rút gọn

Cho chuỗi S gồm n kí tự, được biểu diễn bằng mảng có chỉ số [0..n-1]. Giả sử S được kết thúc bằng 1 kí tự đặc biệt '$', là kí tự có thứ tự từ điển nhỏ nhất trong S. Kí hiệu *suf(S,i)* là hậu tố của S bắt đầu tại S[i]. Một hậu tố *suf(S,i)* sẽ thuộc kiểu S-type nếu *suf(S, i) < suf(S, i + 1)*, và sẽ thuộc L-type nếu *suf(S, i) > suf(S, i + 1)*. Hậu tố *suf(S,n-1)* chỉ chứa kí tự '$' được định nghĩa là thuộc kiểu S-type. Tương ứng, chúng ta sẽ phân loại kí tự S[i] thuộc kiểu S-type hoặc L-type nếu *suf(S,i)* thuộc kiểu S-type hoặc L-type.

Từ các định nghĩa này, có thể rút ra kết luận như sau về một kí tự S[i]:

- Nếu S[i] < S[i+1] thì S[i] là kiểu S-type
- Nếu S[i] > S[i+1] thì S[i] là kiểu L-type
- Nếu S[i] = S[i+1] thì S[i] cùng kiểu với S[i+1]
- S[n-1] là kiểu S-type

Một kí tự S[i], i ∈ [1, n − 1], được gọi là "leftmost S-type" (LMS) nếu S[i] là S-type và S[i-1] là L-type.

Theo như định nghĩa, mảng hậu tố SA sẽ chứa chỉ số của tất cả các hậu tố trong chuỗi S theo thứ tự từ điển. Như vậy, trong mảng SA, các con trỏ của tất cả các hậu tố có kí tự bắt đầu giống nhau thì phải xếp liên tục. Và chúng ta sẽ chia mảng SA ra thành các bucket, mỗi bucket là 1 phần của mảng SA, chứa con trỏ của tất cả các hậu tố có kí tự bắt đầu giống nhau. Do vậy, số lượng bucket trong mảng SA chính bằng với |Σ(S)|. Trong cùng 1 bucket, các hậu tố có cùng kiểu (S-type hoặc L-type) sẽ được phân cụm với nhau, và các hậu tố kiểu L-type sẽ được xếp trước, S-type sẽ xếp sau. Vì vậy mỗi bucket còn có thể chia ra làm 2 sub-bucket tương ứng với kiểu S-type và L-type.

<table>
<tr><td colspan="9">SA
0 n-1</td></tr>
<tr><td>$</td><td colspan="2">a</td><td colspan="2">…</td><td colspan="2">y</td><td colspan="2">z</td></tr>
<tr><td>S-type</td><td>L-type</td><td>S-type</td><td>L-type</td><td>S-type</td><td>L-type</td><td>S-type</td><td>L-type</td><td>S-type</td></tr>
</table>

Hình ảnh 1. Mô tả về bucket của mảng hậu tố SA

Một LMS-substring là một chuỗi con [i..j] của S, với S[i] và S[j] đều là kí tự LMS, và không có bất kì kí tự LMS nào khác nằm giữa S[i] và S[j]. Do vậy số lượng LMS-substring chính là số lượng của kí tự LMS, giả sử số lượng này là n1. Và mảng P1 sẽ lưu trữ chỉ số của tất cả các kí tự LMS này.

Về thứ tự của các LMS-substring được định nghĩa như sau, chúng ta so sánh tương ứng các kí tự của 2 chuỗi con từ trái qua phải. Với mỗi cặp kí tự, chúng ta sẽ so sánh về thứ tự từ điển trước, sau đó là kiểu của kí tự nếu 2 kí tự có cùng thứ tự từ điển, kiểu S-type có độ ưu tiên cao hơn L-type. Theo như định nghĩa này, thì 2 LMS-substring có thể có cùng thứ tự khi chúng giống về độ dài, kí tự và kiểu của các kí tự.

Giả sử chúng ta sắp xếp tất cả các LMS-substring vào các bucket theo đúng thứ tự từ điển. Sau đó chúng ta đặt tên cho mỗi phần tử của P1 theo chỉ số bucket của nó trong mảng SA để tạo ra mội chuỗi mới S1 có độ dài n1. Chúng ta có thể rút ra một vài nhận xét trên S1:

- Độ dài của S1 sẽ lớn nhất là bằng một nửa độ dài của S, có nghĩa là n1 <= n/2. Điều này là hiển nhiên do kí tự đầu tiên của S không thể là kí tự LMS, và không thể có 2 kí tự LMS liên tục trong S.
- Kí tự cuối cùng của S1 là kí tự duy nhất, và có thứ tự từ điển nhỏ nhất ở trong S1. Chứng minh: ta có S[n-1] là 1 kí tự LMS, và LMS-substring bắt đầu tại S[n-1] phải là chuỗi duy nhất và nhỏ nhất trong số tất cả các LMS-substring.
- Nếu S1[i] = S1[j] thì P1[i+1] – P1[i] = P1[j+1] – P1[j]. Chứng minh: từ định nghĩa về S1, ta có S[P1[i] .. P1[i+1]] = S1[P1[j] .. P1[j+1]] và t[P1[i] .. P1[i+1]] = t[P1[j] .. P1[j+1]]. Vì thế, hai LMS-substring bắt đầu tại S[P1[i]] và S[P1[j]] phải có độ dài bằng nhau.
- Thứ tự tương đối về mặt tử điển của 2 hậu tố bất kì *suf(S1, i)* và *suf(S1, j)* giống như thứ tự của *suf(S, P1[i])* và *suf(S, P1[j])*. Chứng minh:
    - Trường hợp 1: S1[i] != S1[j]. Điều này hiển nhiên đúng vì phải có một cặp kí tự trong 2 chuỗi con khác nhau về thứ tự từ điển hoặc kiểu.
    - Trường hợp 2: S1[i] = S1[j]. Trong trường hợp này thứ tự của *suf(S1, i)* và *suf(S1, j)* được quyết định bởi thứ tự của *suf(S1, i+1)* và *suf(S1, j+1)*. Điều tương tự lặp lại nếu S1[i+1] = S1[j+1],… , S1[i+k-1] = S1[j+k-1] đến khi S1[i+k] != S1[j+k]. Vì S1[i..i+k-1] = S1[j..j+k-1] nên P1[i+k] – P1[i] = P1[j+k] – P1[j], vì thế độ dài

của chuỗi con S[P1[i] .. P1[i+k]] và S[P1[j] .. P1[j+k]] là giống nhau. Do vậy việc sắp xếp S1[i .. i+k] và S1[j .. j+k] tương tự như việc sắp xếp S[P1[i] .. P1[i+k]] và S[P1[j] .. P1[j+k]].

- Điều này gợi ý rằng để sắp xếp tất cả các hậu tố LMS trong S, chúng ta có thể sắp xếp S1 để thay thế. Vì S1 ngắn hơn S ít nhất 1/2 nên việc tính toán trên S1 có thể được thực hiện bằng một nửa độ phức tạp đối với S. Gọi SA và SA1 lần lượt là mảng hậu tố của S và S1. Giả sử SA1 đã được tính toán, chúng ta có thể tính toán SA từ SA1.

1.2.2.2. Tính toán SA từ SA1

Thuật toán để tính toán SA từ SA1 trong thời gian tuyến tính:

1. Các con trỏ bucket trỏ vào phần cuối của mỗi bucket của SA (cũng chính là phần cuối của mỗi S-type bucket của SA), sau đó đặt tất cả các phần tử của SA1 vào tương ứng S-type bucket của SA mà không thay đổi thứ tự tương ứng của nó ở SA1.
2. Các con trỏ bucket trỏ vào phần đầu của mỗi bucket của SA (cũng chính là phần đầu của mỗi L-type bucket của SA), sau đó duyệt SA từ đầu đến cuối. Đối với mỗi SA[i], nếu S[SA[i] - 1] là L-type, đặt SA[i]-1 vào vị trí hiện tại của con trỏ bucket tương ứng, sau đó dịch con trỏ sang phải.
3. Các con trỏ bucket trỏ vào phần cuối của mỗi bucket của SA (cũng chính là phần cuối của mỗi S-type bucket của SA), sau đó duyệt SA từ cuối về đầu. Đối với mỗi SA[i], nếu S[SA[i] - 1] là S-type, đặt SA[i]-1 vào vị trí hiện tại của con trỏ bucket tương ứng, sau đó dịch con trỏ sang trái

Mỗi bước trong trên trong thuật toán đều có thể thực hiện với thời gian tuyến tính.

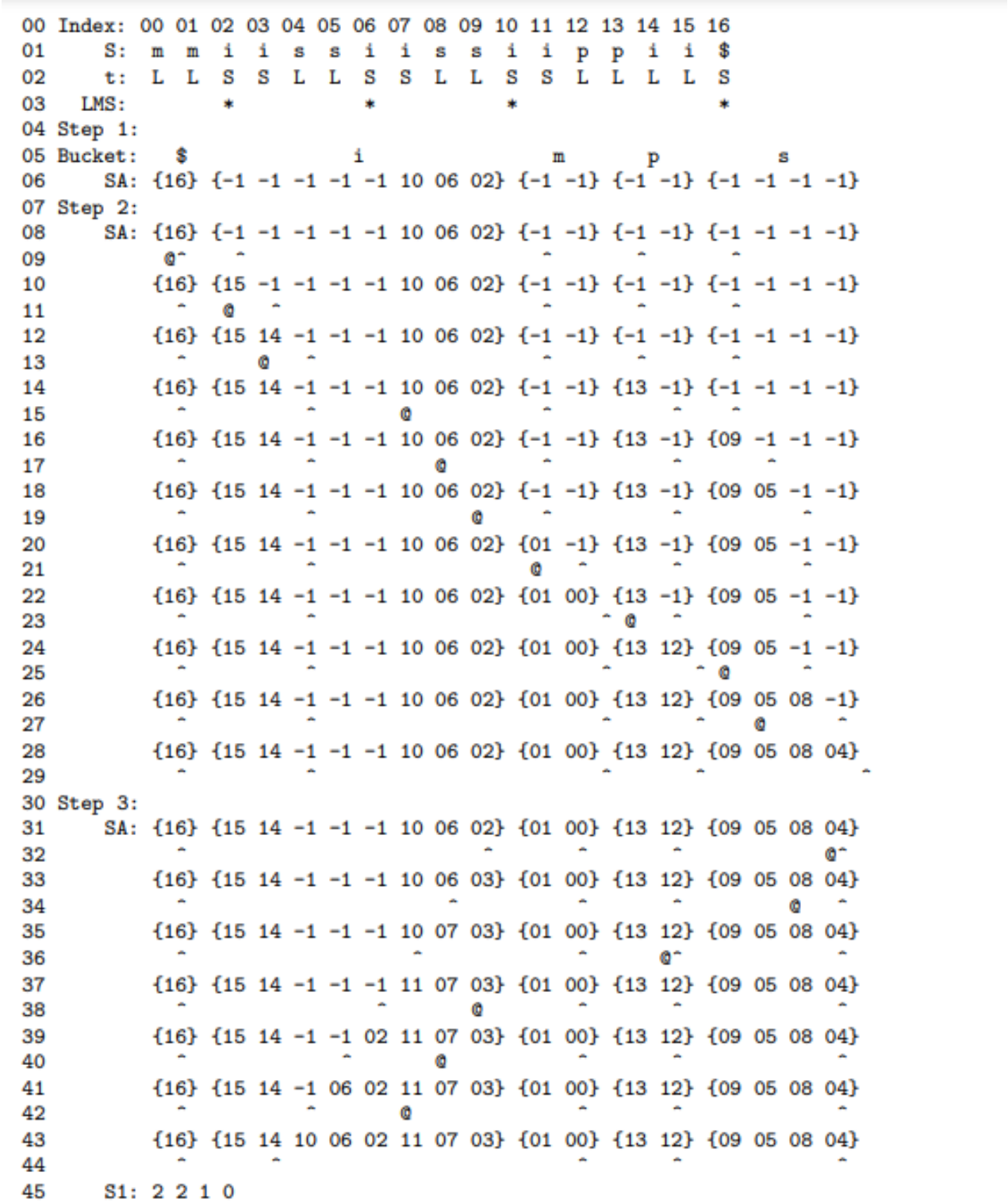
1.2.2.3. Ví dụ chạy thuật toán

Giả sử với chuỗi ban đầu là S = “mmiissiissiippii$”, với $ là kí tự duy nhất, và nhỏ nhất trong S. đầu tiên chúng ta sẽ duyệt S từ trái qua phải để thu được mảng t, các kí tự LMS sẽ được đánh dấu bằng kí tự * ở dòng dưới. Sau đó chúng ta sẽ chạy thuật toán qua từng bước (được mô tả ở hình 2):

1. Các hậu tố LMS là 2, 6, 10 và 16. Có 5 bucket lần lượt là $, i, m, p, s. Chúng ta khởi tạo SA với tất cả các giá trị là -1, sau đó duyệt từ trái qua phải để đặt lần lượt các hậu tố LMS vào cuối bucket tương ứng của nó. Việc đặt các hậu tố trong cùng 1 bucket của bước này chưa cần quan

trọng thứ tự, có thể đặt vào với bất kì thứ tự nào. Ví dụ ở bucket i, 3 hậu tố được đặt theo thứ tự là 10-6-2, chúng ta cũng có thể đặt la 2-6-10.

2. Trước tiên chúng ta sẽ đánh dấu phần đầu các bucket với kí tự '^', sau đó sẽ duyệt SA từ trái qua phải, con trỏ hiện tại sẽ được đánh dấu bằng kí tự '@'. Khi duyệt SA[0] = 16, chúng ta kiểm tra mảng t để biết S[15] = 'i' và thuộc kiểu L-type. Vì thế chúng ta đặt 15 vào bucket i, sau đó con trỏ của bucket i sẽ dịch sang phải một bước. Tiếp tục duyệt SA như vậy cho đến hết.
3. Trước tiên chúng ta sẽ đánh dấu phần cuối các bucket với kí tự '^', sau đó duyệt SA từ trái qua phải. Tại SA[16] = 4, chúng ta thấy S[3] = i là kiểu S-type. Vì thế chúng ta đặt 3 vào bucket i sau đó con trỏ của bucket i sẽ được dịch qua trái một bước. Tiếp tục duyệt SA như vậy cho đến hết.
4. Sau khi thực hiện xong 3 bước trên, ta thu được mảng SA như ở dòng 43. Sau đó chúng ta sẽ duyệt SA từ trái qua phải để đặt tên cho tất cả các LMS-substring. Các LMS-substring 2, 6, 10, 16 được đặt tên lần lượt là 2, 2, 1, 0. Và ta thu được chuỗi S1 ngắn hơn ở dòng 45.



Hình ảnh 2. Mô tả thuật toán để thu được bài rút gọn

Vì S1 có số lượng kí tự là 4, kích cỡ từ điển là 3 (0, 1, 2) nên để tính SA1 thì ta sẽ tiếp tục thực hiện các bước như trên. Sau một vài bước đệ quy, ta sẽ thu được mảng SA1 = [3, 2, 1, 0]. Sau đó ta thực hiện tính toán SA từ SA1 theo các bước (được mô tả ở hình 3):

1. Ta có S1 = [2, 2, 1, 0], tương ứng với các hậu tố LMS ở S là 2, 6, 10, 16. Nên ta đổi lại S1 thành [2, 6, 10, 16]. Có mảng SA1 = [3, 2, 1, 0] nên có thể suy ra thứ thự tương đối của 4 hậu tố LMS 2, 6, 10, 16 trong mảng hậu tố SA là 16, 10, 6, 2. Đặt 4 hậu tố này vào phần cuối bucket ta thu được mảng SA.
2. Đặt con trỏ lại về phần đầu các bucket, thực hiện duyệt từ trái qua phải. Khi duyệt SA[0] = 16, chúng ta kiểm tra mảng t để biết S[15] = 'i' và thuộc kiểu L-type. Vì thế chúng ta đặt 15 vào bucket i, sau đó con trỏ của bucket i sẽ dịch sang phải một bước. Tiếp tục duyệt SA như vậy cho đến hết.
3. Đặt con trỏ lại về phần cuối các bucket, thực hiện duyệt từ phải qua trái. Tại SA[16] = 4, chúng ta thấy S[3] = i là kiểu S-type. Vì thế chúng ta đặt 3 vào bucket i sau đó con trỏ của bucket i sẽ được dịch qua trái một bước. Tiếp tục duyệt SA như vậy cho đến hết.

Bucket	$	i								m		p		s			
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA	16						10	6	2								
SA	16	15	14				10	6	2	1	0	13	12	9	5	8	4
SA	16	15	14	10	6	2	11	7	3	1	0	13	12	9	5	8	4

Hình ảnh 3. Mô tả thuật toán tính toán SA từ SA1

1.3. Thuật toán SACA-K

1.3.1. Mã giả của thuật toán SACA-K

SACA-K(T, SA, K, n, level):

T: input string;

SA: suffix array of T;

K: alphabet size of T;

n: size of T;

level: recursion level;

Stage 1: induced sort the LMS-substrings of T.

1. **if** level = 0:
   then
2. Allocate an array of K integers for bkt;
3. Induced sort all the LMS-substrings of T, using bkt for bucket counters;
   else
4. Induced sort all the LMS-substrings of T, reusing the start or the end of each bucket as the bucket's counter;

SA is reused for storing T1 and SA1.

Stage 2: name the sorted LMS-substring of T

5. Compute the lexicographic names for the sorted LMS-substrings to produce T1;

Stage 3: sort recusively

6. **if** K1 = n1 (each character in T1 is unique)
   then
7. Directly compute SA(T1) from T1;

   else

8. SACA-K(T1, SA1, K1, n1, level + 1);

Stage 4: induced sort SA(T) from SA(T1).

9. **if** level = 0

then

10. Induced sort SA(T) from SA(T1), using bkt for bucket counters;
11. Free the space allocated for bkt;

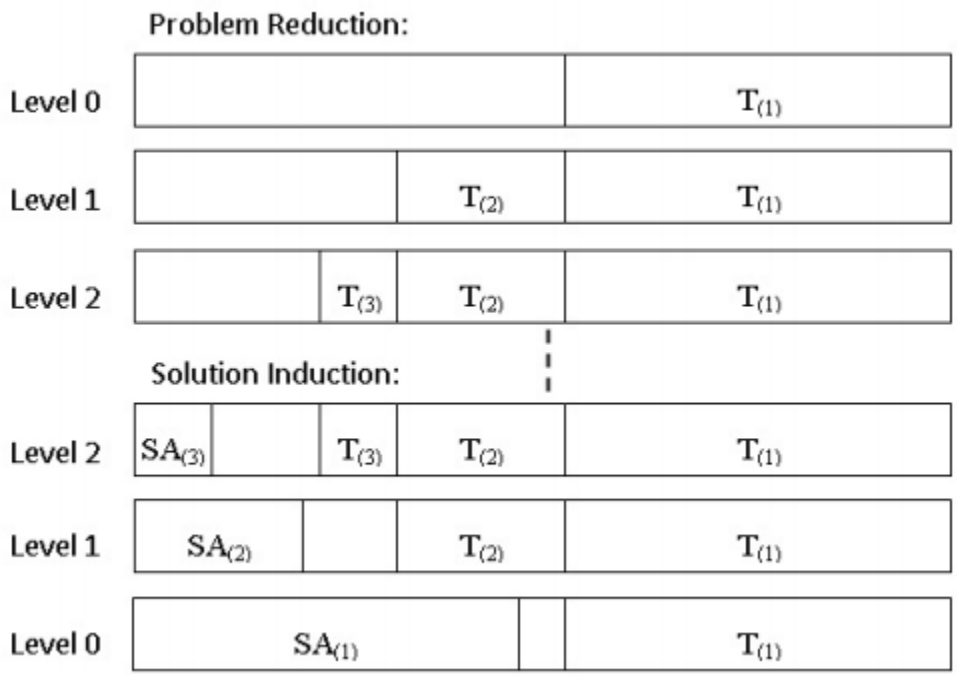
else

12. Induced sort SA(T) from SA(T1), reusing the start or the end of each bucket as the bucket's counter;
13. **return**;

1.3.2.Các cải tiến của thuật toán SACA-K so với SA-IS

Trong thiết kế của SA-IS, sử dụng mảng B để lưu chỉ số con trỏ của các bucket trong mảng SA, và sử dụng mảng t để lưu kiểu của các kí tự trong chuỗi ban đầu. Tuy nhiên SACA-K loại bỏ mảng t và chỉ sử dụng mảng B tại recursion level đầu tiên. Điều này giúp tiết kiệm một lượng bộ nhớ đáng kể so với SA-IS.

Mảng SA ban đầu được tái sử dụng để lưu trữ các chuỗi và mảng hậu tố của bài toán rút gọn ở các recursion level cao hơn. Điều này là khả thi do n1 <= 1/2 n.



Hình ảnh 4. Minh họa việc tái sử dụng mảng SA

Chương 2: Thuật toán Burrows-Wheeler Transform

2.1. Giới thiệu

Chuyển đổi Burrows-Wheeler (BWT) là thuật toán thực hiện sắp xếp lại các kí tự trong một chuỗi kí tự, có xu hướng phân cụm lại các kí tự tương tự nhau. Điều này rất có ích chơ việc nén. Quan trọng hơn là việc chuyển đổi có thể đảo ngược lại mà không cần lưu thêm bất kì dữ liệu nào ngoại trử chỉ số của kí tự đầu tiên. Do đó, BWT rất có ích trong việc cải thiện hiệu quả của các thuật toán nén.

2.2. Thuật toán BWT

Giả sử BWT mã hóa chuỗi T[0..n-1] trên một bảng chữ cái Σ(T) gồm |Σ(T)| kí tự. Các bước của chuyển đổi BWT thực hiện như sau:

1. Tính toán mảng hậu tố SA bằng thuật toán SACA-K
2. Thực hiện tính toán mảng BWT với công thức:

BWT[i] = T[(SA[i] – 1)%n]

Và thực hiện lưu lại chỉ số first_index (chỉ số của kí tự đầu tiên) như sau: first_index là chỉ số thỏa mãn điều kiện SA[first_index] = 0.

Chương 3: Thuật toán Move-To-Front

3.1. Giới thiệu

Phép biến đổi Move-To-Front (MTF) là một thuật toán mã hóa dữ liệu được thiết kế để cải tiến hiệu suất của các kỹ thuật mã hóa entropy nén. Nó là bước bổ sung trước bước nén dữ liệu, giúp việc nén đạt hiệu quả tốt hơn.

3.2. Thuật toán MTF

Các bước thực hiện thuật toán MTF với chuỗi kí tự đầu vào T[0..n-1]:

1. Xây dựng 1 bảng chữ cái ban đầu, ví dụ như là: “abcdefghijklmnopqrstuvwxyz”
2. Duyệt chuỗi kí tự đầu vào từ trái sang phải. Tại mỗi vị trí i sẽ thực hiện tìm vị trí của kí tự đó trong bảng chữ cái, sau đó thay kí tự T[i] bằng chỉ số của nó trong bảng chữ cái. Sau đó thực hiện đưa kí tự đó trong bảng chữ cái đẩy lên vị trí đầu bảng chữ cái.

3.3. Ví dụ chạy thuật toán MTF

Ví dụ mới chuỗi đầu vào T = “bananaaa”.

Ta xây dựng bảng chữ cái là “abcdefghijklmnopqrstuvwxyz”.

Các bước chạy của thuật toán:

Chuỗi T	Kết quả	Bảng chữ cái
bananaaa	1	abcdefghijklmnopqrstuvwxyz
bananaaa	1,1	bacdefghijklmnopqrstuvwxyz
bananaaa	1,1,13	abcdefghijklmnopqrstuvwxyz
bananaaa	1,1,13,1	nabcdefghijklmopqrstuvwxyz
bananaaa	1,1,13,1,1	anbcdefghijklmopqrstuvwxyz
bananaaa	1,1,13,1,1,1	nabcdefghijklmopqrstuvwxyz
bananaaa	1,1,13,1,1,1,0	anbcdefghijklmopqrstuvwxyz
bananaaa	1,1,13,1,1,1,0,0	anbcdefghijklmopqrstuvwxyz
Kết quả cuối cùng	1,1,13,1,1,1,0,0	anbcdefghijklmopqrstuvwxyz

Hình ảnh 5. Minh họa chạy thuật toán MTF

Chương 4: Kết quả thực hiện

4.1. Cài đặt chương trình

Chương trình minh họa được em cài đặt trên ngôn ngữ lập tình C++. Ý tưởng của chương trình:

1. Thực hiện đọc dữ liệu từ file đầu vào.
2. Thực hiện thuật toán SACA-K để tính toán mảng hậu tố SA.
3. Sau đó thực hiện chuyển đổi BWT từ mảng SA đã tính toán. Và lưu lại chỉ số first_index ra một file đầu ra.
4. Thực hiện biến đổi MTF. Ghi kết quả cuối cùng này ra file đầu ra.
5. Sau đó sử dụng một thư viện có sẵn để tiến hành mã hóa huffman file đầu ra để xem hiệu quả nén.

4.2. Kết quả thu được

Kết quả về thời gian và lượng RAM tiêu thụ với một số file:

Kích cỡ file	Thời gian encode (ms)	Lượng RAM tiêu thụ khi encode	Thời gian deocode (ms)	Lượng RAM tiêu thụ khi decode

168MB	129400	845.1MB	43198	1014MB
168MB	130274	845.1MB	39153	1014MB
173MB	180009	870.4MB	57349	1044.3MB
173MB	178572	870.4MB	59707	1044.3MB
197MB	203496	988.2MB	67530	1185.7MB
197MB	192907	988.2MB	64674	1185.7MB
98MB	57843	489.1MB	16199	586.8MB
98MB	54470	489.1MB	15922	586.8MB

Hình ảnh 6. Kết quả chạy chương trình

4.3. Kết luận

Qua kết quả chạy chương trình, em thấy rằng thuật toán SACA-K có thời gian chạy khá tốt, nhanh hơn đáng kể so với các thuật toán xây dựng mảng hậu tố khác, điều này giúp cho việc tính toán BWT trở nên dễ dàng hơn. Tuy nhiên em thấy lượng bộ nhớ tiêu thụ vẫn khá lớn. Nếu có thể cải thiện lượng bộ nhớ tiêu thụ thì thuật toán sẽ rất hiệu quả trong việc hỗ trợ nén file.

Tài liệu tham khảo

[1] Louza, F. A., Smyth, W., Manzini, G., & Telles, G. P. (2018). Lyndon array construction during Burrows–Wheeler inversion. *Journal of Discrete Algorithms*, *50*, 2–9. https://doi.org/10.1016/j.jda.2018.08.001

[2] Nong, G. (2013). Practical linear-time O (1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, *31*(3), 1–15. https://doi.org/10.1145/2493175.2493180

[3] Nong, G., Zhang, S., & Chan, W. H. (2009). Linear Suffix Array Construction by Almost Pure Induced-Sorting. *2009 Data Compression Conference*. Published. https://doi.org/10.1109/dcc.2009.42

[4] Thomas. (2014, June 14). Bàn về mảng hậu tố (Suffix Array). Retrieved June 14, 2014, from https://blogthuattoan.blogspot.com/2014/06/ban-ve-mang-hau-to-suffix-array.html

[5] Wikipedia contributors. (2021a, April 14). Move-to-front transform. Retrieved June 20, 2021, from https://en.wikipedia.org/wiki/Move-to-front_transform

[6] Wikipedia contributors. (2021b, June 10). Burrows–Wheeler transform. Retrieved June 20, 2021, from https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform

[7] Wikipedia contributors. (2021c, June 16). Huffman coding. Retrieved June 20, 2021, from https://en.wikipedia.org/wiki/Huffman_coding