BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT THÀNH PHỐ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KỲ

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

TÌM HIỂU THUẬT TOÁN, CẤU TRÚC DỮ LIỆU ĐỂ ĐÁNH GIÁ MỨC ĐỘ GIỐNG NHAU CỦA HAI VĂN BẢN

Học kỳ: 1/2021-2022

Mã HP: DASA230179_21_1_08

GVHD:TS.Huỳnh Xuân Phụng

Nhóm SVTH:

Họ và tên	MSSV
Lê Thị Kim Lệ	20110248
Phùng Thị Thùy Trang	20110313
Hoàng Trần Nguyễn	20110686
Nguyễn Thị Cẩm Nguyên	20110315

Tp. Hồ Chí Minh, tháng 12 năm 2021

NHẬN XÉT CỦA GIÁO VIÊN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Điểm: ……..

KÝ TÊN:

MỤC LỤC

MỞ ĐẦU

1. Lý do chọn đề tài

Ngày nay mạng Internet phát triển rộng rãi nên khối lượng thông tin được chia sẻ trên Internet là vô cùng to lớn. Đó là những thông tin, kiến thức trong tất cả các lĩnh vực y tế, giáo dục, khoa học,thể thao, âm nhạc….Để cho con người khai thác thông tin một cách hiệu quả và khoa học thì việc đo lường sự giống nhau giữa hai văn bản, tài liệu là vô cùng quan trọng trong các lĩnh vực tìm kiếm, phân loại và quản lý thông tin trên mạng theo nội dung. Đặc biệt việc so sánh mức độ giống nhau giữa hai văn bản là vô cùng cần thiết trong nghành giáo dục để tránh trường hợp sao chép luận văn, báo cáo, các công trình nghiên cứu khoa học… nhằm đảm bảo chất lượng học sinh, sinh viên.Vì vậy việc xây dựng một công cụ để kiểm tra sự trùng lặp, giống nhau giữa hai văn bản là vô cùng hữu ích nên nhóm chúng em quyết định chọn đề tài "Tìm hiểu thuật toán, cấu trúc dữ liệu để đánh giá mức độ giống nhau của hai văn bản".

2. Mục tiêu nghiên cứu

Xây dựng được thuật toán so sánh, đánh giá sự giống nhau của hai văn bản

Xây dựng được cấu trúc dữ liệu phù hợp với thuật toán so sánh độ giống nhau giữa hai văn bản

3. Nhiệm vụ của đề tài

Tìm hiểu các phương pháp đánh giá mức độ giống nhau giữa hai văn bản

Tìm hiểu các giải thuật đánh giá mức độ giống nhau giữa hai văn bản

Ứng dụng kiến thức môn học cấu trúc dữ liệu và giải thuật vào lựa chọn các cấu trúc dữ liệu phù hợp với đề tài

4. Đối tượng và phạm vi nghiên cứu

4.1.Đối tượng nghiên cứu

Đối tượng nghiên cứu của đề tài là các cấu trúc dữ liệu và cơ sở lý thuyết các phương pháp, các thuật toán đánh giá sự giống nhau của hai văn bản

4.2.Phạm vi nghiên cứu

Đề tài tập trung nghiên cứu các kiến thức có liên quan đến các cấu trúc dữ liệu như BTree, vecto, spare matrix và các giải thuật so sánh độ giống nhau hai văn bản như levenstein

5.Phương pháp nghiên cứu

Phương pháp nghiên cứu chủ yếu là tham khảo các tài liệu, bài viết, sách giáo trình liên quan tới cấu trúc dữ liệu,phương pháp và giải thuật so sánh sự giống nhau giữa hai văn bản

Tìm hiểu trên các diễn đàn lập trình các hướng giải quyết cho vấn đề phát sinh trong quá trình thực hiện đồ án.

6. Các hướng đánh giá trước đó

Để đánh giá sự giống nhau của hai văn bản trước đó có các công cụ như Plagiarism Checker Software, Turnitin,... Nhưng những hệ thống này chỉ cho phép phát hiện sự trùng lặp của dữ liệu có trong tên miền gốc, việc mở rộng cơ sở dữ liệu mẫu theo yêu cầu người sử dụng trở nên khó khăn và tốn chi phí rất cao. Vì thế, cần tiếp tục nghiên cứu để tìm kiếm các giải pháp tốt hơn.

7. Kết quả dự kiến đạt được

Đạt được mục tiêu đạt ra của đề tài để khi nhập vào hai văn bản tiếng anh vào file text thì chương trình sẽ in ra phần trăm tỉ lệ giống nhau của hai văn bản

NỘI DUNG

CHƯƠNG 1: CƠ SỞ LÝ THUYẾT VÀ PHƯƠNG PHÁP NGHIÊN CỨU

1.1. Độ giống nhau của văn bản

1.1.1. Định nghĩa

Phát biểu bài toán tính độ tương đồng văn bản như sau: xét một tài liệu d gồm có n câu: d = s1 , s2 ,… sn . Mục tiêu của bài toán là tìm ra một giá trị của hàm S(si , sj) với S (0,1), và i, j = 1,… n. Hàm S(si , sj) được gọi là độ đo tương đồng giữa 2 văn bản si và sj . Giá trị càng cao thì sự giống nhau về nghĩa của 2 văn bản càng nhiều .

Ví dụ, xét 2 câu "Tôi là nam" và "Tôi là nữ", bằng trực giác có thể thấy rằng 2 câu trên có sự tương đồng khá cao. Độ tương đồng ngữ nghĩa là một giá trị tin cậy phản ánh mối quan hệ ngữ nghĩa giữa 2 câu. Trên thực tế, khó có thể lấy một giá trị chính xác bởi vì ngữ nghĩa chỉ được hiểu đầy đủ trong một ngữ cảnh cụ thể.

1.1.2. Các phương pháp đánh giá độ giống nhau của hai văn bản

Các phương pháp đánh giá độ giống nhau của hai văn bản có thể được chia thành như sau:

- Đánh giá dựa trên chuỗi ký tự
- Đánh giá dựa trên từ vựng
- Đánh giá dựa trên ngữ nghĩa

Trong bài báo cáo tập trung nghiên cứu về phương pháp đánh giá độ giống nhau của hai văn bản dựa trên chuỗi ký tự, cụ thể là thuật toán Levenshtein.

1.2. Stopword

Trong điện toán và xử lý ngôn ngữ tự nhiên, từ dừng (tiếng Anh: stopword) là các từ được lọc ra trước hoặc sau quá trình xử lý dữ liệu văn bản. Mặc dù từ dừng thường coi là các từ phổ biến trong một ngôn ngữ, tuy nhiên vẫn chưa các một danh sách chung (toàn cục) các từ dừng được dùng trong tất cả công cụ xử lý ngôn ngữ tự nhiên, và thực vậy không phải tất cả công cụ đều có danh sách này. Một số công cụ tránh xóa từ dùng để hỗ trợ việc tìm kiếm cụm từ.

Từ dừng là những từ ngữ bị các công cụ tìm kiếm bỏ qua hoàn toàn. Những từ này không có liên quan gì đến ý nghĩa của bài viết và cũng không được các công cụ tìm kiếm ghi nhận.

Một bài viết thường có đến 25% từ dừng có thể kể đến như: of, a, or, many, bởi, bị cả,....

Ưu và nhược điểm:

Một trong những điều đầu tiên mà chúng tôi tự hỏi mình là ưu và nhược điểm của bất kỳ nhiệm vụ nào chúng tôi thực hiện. Hãy xem xét một số ưu và nhược điểm của việc loại bỏ từ dừng trong NLP.

Ưu điểm:

* Các từ dừng thường bị xóa khỏi văn bản trước khi đào tạo mô hình học sâu và học máy vì các từ dừng xuất hiện rất nhiều, do đó cung cấp rất ít hoặc không có thông tin duy nhất có thể được sử dụng để phân loại hoặc phân cụm.

* Khi loại bỏ các từ dừng, kích thước tập dữ liệu giảm và thời gian đào tạo mô hình cũng giảm mà không ảnh hưởng lớn đến độ chính xác của mô hình.

* Loại bỏ từ khóa có khả năng giúp cải thiện hiệu suất, vì có ít hơn và chỉ còn lại các mã thông báo quan trọng. Do đó, độ chính xác phân loại có thể được cải thiện

Khuyết điểm:

Việc lựa chọn và loại bỏ các từ dừng không đúng cách có thể thay đổi ý nghĩa của văn bản của chúng ta. Vì vậy, chúng ta phải cẩn thận trong việc lựa chọn từ dừng của mình.

1.3. Giải thuật

1.3.1. Khoảng cách Levenshtein

1.3.1.1. Khái niệm

Khoảng cách Levenshtein thể hiện khoảng cách khác biệt giữa 2 chuỗi ký tự. Khoảng cách Levenshtein giữa chuỗi S và chuỗi T là số bước ít nhất biến chuỗi S thành chuỗi T thông qua 3 phép biến đổi là

- xoá 1 ký tự.
- thêm 1 ký tự.
- thay ký tự này bằng ký tự khác.

Khoảng cách này được đặt theo tên Vladimir Levenshtein, người đã đề ra khái niệm này vào năm 1965. Nó được sử dụng trong việc tính toán sự giống và khác nhau giữa 2 chuỗi, như chương trình kiểm tra lỗi chính tả của winword spellchecker.

1.3.1.2. Thuật toán

Để tính toán Khoảng cách Levenshtein, ta sử dụng thuật toán quy hoạch động, tính toán trên mảng 2 chiều (n+1)*(m+1), với n, m là độ dài của chuỗi cần tính. Sau đây là đoạn mã *(S, T là chuỗi cần tính khoảng cách, n, m là độ dài của chuỗi S, T)*:

```
int LevenshteinDistance(char s[1..m], char t[1..n])
// d is a table with m+1 rows and n+1 columns
declare int d[0..m, 0..n]

for i from 0 to m
  d[i, 0]:= i
for j from 0 to n
  d[0, j]:= j

for i from 1 to m
  for j from 1 to n
  {
 if s[i] = t[j] then cost:= 0
 else cost:= 1
 d[i, j]:= minimum(
  d[i-1, j] + 1, // trường hợp xoá
  d[i, j-1] + 1, // trường hợp thêm
  d[i-1, j-1] + cost  // trường hợp thay thế
 )
  }
```

```
return d[m, n]
```

1.3.1.2. Tính độ giống nhau của hai chuỗi ký tự dựa vào Khoảng cách Levenshtein

Độ giống nhau của hai chuỗi ký tự được tính theo công thức sau:

$$P = 1 - (\frac{LevenshteinDistance(a,b)}{maxlenght(a,b)})$$

Trong đó: a,b là hai chuỗi ký tự;

Maxlenght: độ dài chuỗi lớn hơn trong hai chuỗi a,b.

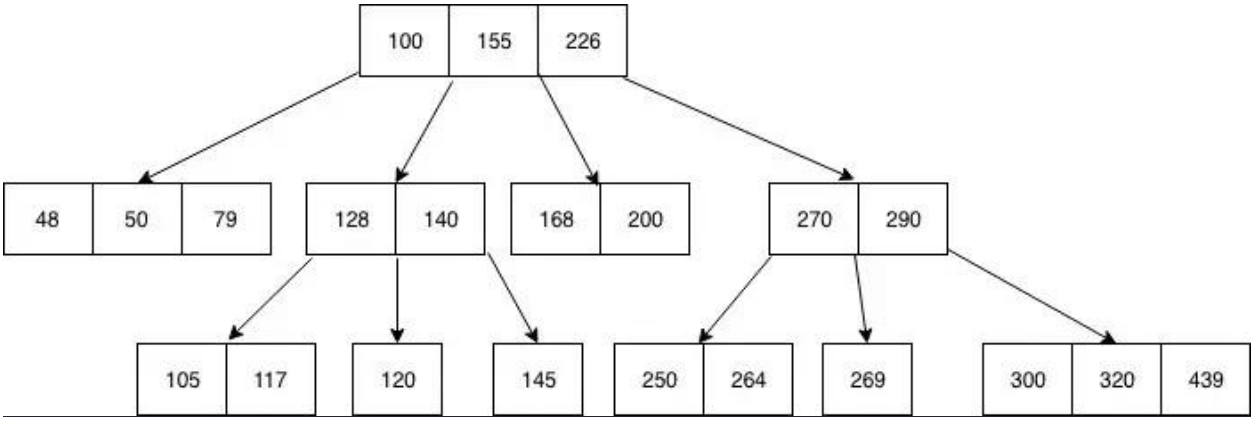
1.4. Cấu trúc dữ liệu

1.4.1. B-tree

1.4.1.1. Khái niệm

B-tree là một loại cây tìm kiếm tự cân bằng đặc biệt, trong đó mỗi nút có thể chứa nhiều hơn một khóa và có thể có nhiều hơn hai nút con. Nó là một dạng tổng quát của cây tìm kiếm nhị phân.

Ví dụ minh họa :



1.4.1.2. Lý do sử dụng

Thuật toán đòi hỏi ít thời gian hơn trong việc truy cập phương tiện lưu trữ như ổ cứng , giảm thiểu việc truy cập vào đĩa.

B-tree có thể lưu trữ nhiều khóa trong một nút và có thể có nhiều nút con .Điều này làm giảm chiều cao đáng kể cho phép truy cập đĩa nhanh hơn so với các cây khác.

1.4.1.3. Thuộc tính của cây B-tree

Đối với mỗi nút x, các khóa được lưu trữ theo thứ tự tăng dần.

Trong mỗi nút, có một giá trị kiểu boolean là x.leaf sẽ có giá trị là true nếu x là nút lá.

Nếu n là bậc của cây, mỗi nút bên trong có thể chứa nhiều nhất n-1 khóa cùng với một con trỏ tới mỗi nút con.

Mỗi nút ngoại trừ nút gốc có thể có nhiều nhất n nút con và tối thiểu là n/2 nút con.

Tất cả các lá đều có cùng độ sâu (tức là chiều cao h của cây).

Nút gốc có ít nhất 2 nút con và chứa tối thiểu 1 khoá.

Nếu n≥1, thì với bất kỳ cây B-tree với n khóa có chiều cao h và bậc nhỏ nhất là t≥2,h≥ logt(n+1)2

1.4.1.4. Thao tác trên cây B-tree

1.4.1.4.1. Thao tác tìm kiếm

Tìm kiếm phần tử là hình thức tổng quát của việc tìm kiếm phần tử trong cây tìm kiếm nhị phân. Các bước được thực hiện như sau.

- Bắt đầu từ nút gốc, so sánh k với khóa đầu tiên của nút. Nếu k = khóa đầu tiên của nút, trả về nút và chỉ số của nó.
- Nếu k.leaf có giá trị là true, trả về NULL (tức là không tìm thấy).
- Nếu k < khóa đầu tiên của nút gốc, ta sẽ tìm kiếm đệ quy cho nút con bên trái của khóa này.
- Nếu có nhiều hơn một khóa trong nút hiện tại và k > khóa đầu tiên, ta sẽ so sánh k với khóa tiếp theo trong nút. Nếu k < khóa tiếp theo, ta sẽ tìm kiếm nút con bên trái của khóa này (tức là k nằm giữa khóa đầu tiên và khóa thứ hai). Nếu không, ta sẽ tìm kiếm nút con bên phải của khóa.
- Lặp lại các bước từ 1 đến 4 cho đến khi đạt được nút lá.

1.4.1.4.2. Thao tác chèn vào cây

Việc chèn một phần tử trên cây B-tree bao gồm hai sự kiện: tìm kiếm nút thích hợp để chèn phần tử và tách nút nếu được yêu cầu. Hoạt động chèn luôn diễn ra theo phương pháp từ dưới lên.

Thao tác chèn được thực hiện như sau:

- Nếu cây trống rỗng, ta sẽ tạo một nút gốc và chèn khóa.
- Cập nhật số lượng khóa được cho phép trong nút.
- Tìm kiếm nút thích hợp để chèn.
- Nếu nút đã đầy, ta sẽ làm theo các bước bên dưới.
- Chèn các phần tử theo thứ tự tăng dần.
- Có những phần tử lớn hơn so với giới hạn của nó. Vì vậy, ta sẽ tách ra ở phần giữa.

- Đẩy khóa giữa lên trên và đặt khóa bên trái làm khóa của nút con bên trái và khóa bên phải làm nút con bên phải.
- Nếu nút chưa đầy, ta sẽ làm theo bước bên dưới.
- Chèn nút theo thứ tự tăng dần.

1.4.1.5. Độ phức tạp của cây B-tree

- Độ phức tạp về thời gian của trường hợp xấu nhất: Θ(log n)
- Độ phức tạp thời gian của trường hợp trung bình: Θ(log n)
- Độ phức tạp về thời gian của trường hợp tốt nhất: Θ(log n)
- Độ phức tạp không gian của trường hợp trung bình: Θ(n)
- Độ phức tạp không gian của trường hợp xấu nhất: Θ(n)

1.4.2. Mảng

1.4.2.1. Khái niệm

Mảng là một loại cấu trúc dữ liệu trong ngôn ngữ lập trình C/C++, nó lưu trữ một tập hợp tuần tự các phần tử cùng kiểu với độ dài cố định. Mảng thường được sử dụng để lưu trữ tập hợp dữ liệu, nhưng nó cũng hữu dụng khi dùng để lưu trữ một tập hợp biến có cùng kiểu.

1.4.2.2. Các thao tác với mảng

1.4.2.2.1. Mảng 1 chiều

Khai báo mảng

Để khai báo một mảng trong ngôn ngữ C/C++, bạn xác định kiểu của phần tử và số lượng các phần tử được yêu cầu bởi biến đó như sau:

```
Kieu Ten_mang [ Kich_co_mang ];
```

Trong đó Kich_co_mang phải là một số nguyen lớn hơn 0.

Khởi tạo mảng

Mảng được khởi tạo như sau

```
int array[] = {num1, num2, num3, num4, …};
```

Truy cập các phần tử trong mảng

Một mảng được truy cập bởi cách đánh chỉ số trong tên của mảng. Dưới đây là một cách truy cập một giá trị của mảng:

```
int num = array[2];
```

1.4.2.2.2. Mảng 2 chiều

Khai báo mảng

Một mảng hai chiều về bản chất là danh sách của các mảng một chiều. Để khai báo một mảng hai chiều integer với kích cỡ x, y có thể viết như sau

```
array [ x ][ y ];
```

Khởi tạo mảng hai chiều

Các mảng đa chiều có thể được khởi tạo bởi xác định các giá trị trong dấu móc vuông cho mỗi hang

Ví dụ :

```
int array[3][4] = {
 {0, 1, 2, 3} , /* khoi tao gia tri cho hang ma co chi muc la 0 */
 {4, 5, 6, 7} , /* khoi tao gia tri cho hang ma co chi muc la 1 */
 {8, 9, 10, 11} /* khoi tao gia tri cho hang ma co chi muc la 2 */
};
```

Truy cập các phần tử của mảng hai chiều

Các phần tử mảng hai chiều được truy cập bởi sử dụng các chỉ số, ví dụ chỉ số hàng và chỉ số cột. Ví dụ:

```
int val = a[2][3];
```

1.4.3. Vecto

1.4.3.1. Khái niệm

Không giống như array (mảng), chỉ một số giá trị nhất định có thể được lưu trữ dưới một tên biến duy nhất. Vector trong C++ giống dynamic array (mảng đ3.3ộng) nhưng có khả năng tự động thay đổi kích thước khi một phần tử được chèn hoặc xóa tùy thuộc vào nhu cầu của tác vụ được thực thi, với việc lưu trữ của chúng sẽ được vùng chứa

tự động xử lý. Các phần tử vector được đặt trong contiguous storage (bộ nhớ liền kề) để chúng có thể được truy cập và duyệt qua bằng cách sử dụng iterator.

1.4.3.2. Lý do sử dụng

Không cần thiết phải cấp phát hoặc thu hồi vùng nhớ

Nếu bạn thêm một phần tử vào vecto đã đầy thì nó sẽ tự động tăng kích thước của nó.

Xác định được số lượng các phần tử bạn lưu trong đó.

CHƯƠNG 2: VẬN DỤNG GIẢI QUYẾT VẤN ĐỀ

2.1. Môi trường xây dựng đồ án

2.1.1. Ngôn ngữ lập trình

Ngôn ngữ C/C++

2.1.2. Môi trường lập trình

Visual Studio 2019

2.1.3. Thư viện sử dụng

Sử dụng các thư viện chuẩn của ngôn ngữ C/C++:

Thư viện nhập xuất chuẩn: <iostream>

Thư viện làm việc với file: <fstream>

Thư viện string: <string>

Thư viện vector: <vector>

2.1.4. Môi trường chạy minh họa kết quả Simulate

Các phần đánh giá độ phức tạp của thuật toán cũng như cấu trúc dữ liệu bằng Simulate được thể hiện trong các phần code trong link Githup thứ hai.

Mỗi quá trình đánh giá gồm hai giai đoạn:

+ GD1: Thực hiện chạy Simulate trên các phần code C++ cho đầu ra là file chứa dữ liệu

+ GD2: Thực hiện đánh giá kết quả thông qua mô phỏng đồ thị trong phần code Python Simulate

Các cặp dữ liệu đánh giá được ký hiệu từ P1 tới P30, dung lượng các file được ghi trong bảng sau:

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
File 1 size(kb)	1	3	3	9	20	30	49	58	68	73	81	99	112	124	130
File 2 size(kb)	3	3	9	20	30	49	58	68	73	81	99	112	124	130	137

P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30

137	143	154	177	207	221	312	331	434	538	548	711	766	887	1085
143	154	177	207	221	312	331	434	538	548	711	766	887	1085	2153

2.2. Thực hiện đồ án

2.2.1. Cấu trúc dữ liệu

2.2.1.1. Cấu trúc chứa từ

Xây dựng struct "word" để chứa nội dung của từng từ trong câu, đồng thời chứa vị trí của từ trong văn bản (thuộc câu nào).

```
struct word
{
  int belong;
  string value;
  bool isstopwword = false;
  void Init(string name, int num)
  {
        value = name;
        belong = num;
  }
}
```

Cấu trúc word

```
word* NewWord(string name, int num)
{
        word* temp = new word;
        temp->Init(name, num);
        return temp;
}
```

Hàm khởi tạo cho 1 word

Trong đó:

- Int belong: vị trí của từ trong văn bản (thuộc câu nào).
- string value: nội dung của từ.
- bool isstopword: đánh dấu từ có phải là một stopword hay không.
- void Init(): Hàm khởi tạo 1 word với nội dung và vị trí truyền vào

Độ phức tạp về thời gian đọc cũng như truy xuất: O(1).

2.2.1.2. Cấu trúc chứa câu

Tương tự với từ, sử dụng struct để xây dựng 1 cấu trúc câu bao gồm 1 vecto chứa các từ trong câu, và vị trí của câu trong văn bản.

```
struct sentence
{
        int index = 0;
        vector<word> key;
        int count = 0;
        void AddWord(word* x)
        {
                key.push_back(*x);
                count++;
    }

}
```

Cấu trúc sentence

Trong đó:

- int index: là vị trí của câu trong văn bản
- vector<word> key: lưu chuỗi các word trong câu. Ở đây, việc sử dụng vecto để lưu trữ mà không phải là mảng bởi ta chưa biết trước được kích thước của câu, vecto hoàn toàn phù hợp với việc này và cách truy xuất cũng đơn giản như với mảng
- int count: là số word trong câu.
- void AddWord(): hàm thêm 1 word vào câu.

Độ phức tạp về thời gian đọc cũng như truy xuất: O(1).

2.2.1.3. Cấu trúc chứa văn bản

Đối với cấu trúc văn bản, về cơ bản sẽ được tổ chức thành vecto các câu. Bên cạnh đó, để tạo sự thuận lợi cho việc tìm ra độ giống nhau của hai văn bản cũng như tiết kiệm thời gian cho thuật toán, nhóm đã tổ chức hai văn bản cần xét sự giống nhau thành hai kiểu.

Văn bản với dung lượng ít hơn (des): Được tổ chức theo như ban đầu với vecto các câu.

Văn bản với dung lượng lớn hơn (sour) : Bên cạnh việc tổ chức thành vecto các câu, các từ trong văn bản sẽ được tổ chức vào cây Btree, mỗi phần tử tại mỗi Node của Btree là nội dung của từ, đồng thời là một vecto chứa các vị trí của từ đó trong văn bản.

```
struct text
{
  BTree* tree = new BTree(3);
  vector<sentence> sen;
  int count = 0;
  bool issour;
  void AddSentence(sentence*& s)
  {
          int index;
          s->index = count;
          sen.push_back(*s);
          if (issour == true)
      for (word i : s->key)
           {
            if(i.isstopwword==false)
                    {
                     tree->insert(i);
                    }
           }
          count++;
  }
```

Cấu trúc text

Trong đó:

- BTree* tree: cấu trúc BTree lưu các từ đối với văn bản sour, có tác dụng như từ điển phục vụ cho việc tìm kiếm.
- vector<sentence> sen: vector chứa các câu trong văn bản.
- Int count: chứa số câu trong văn bản
- bool issour: đánh dấu văn bản dài hơn(sour).
- void AddSentence(): thêm 1 câu vào cấu trúc text.

Độ phức tạp về thời gian đọc cũng như truy xuất: O(log n).

Đối với việc tổ chức các từ trong văn bản dưới dạng như từ điển tìm kiếm và lưu trữ vị trí xuất hiện của từng từ, nhóm đã nghiên cứu và đánh giá bốn loại cấu trúc dữ liệu: BTree, Trie và Binary Search Tree và AVL Tree đưa đến kết luận sử dụng BTree. Sau đây là những đánh giá mà nhóm có được:

* So sánh với BTree các cấu trúc dữ liệu khác:

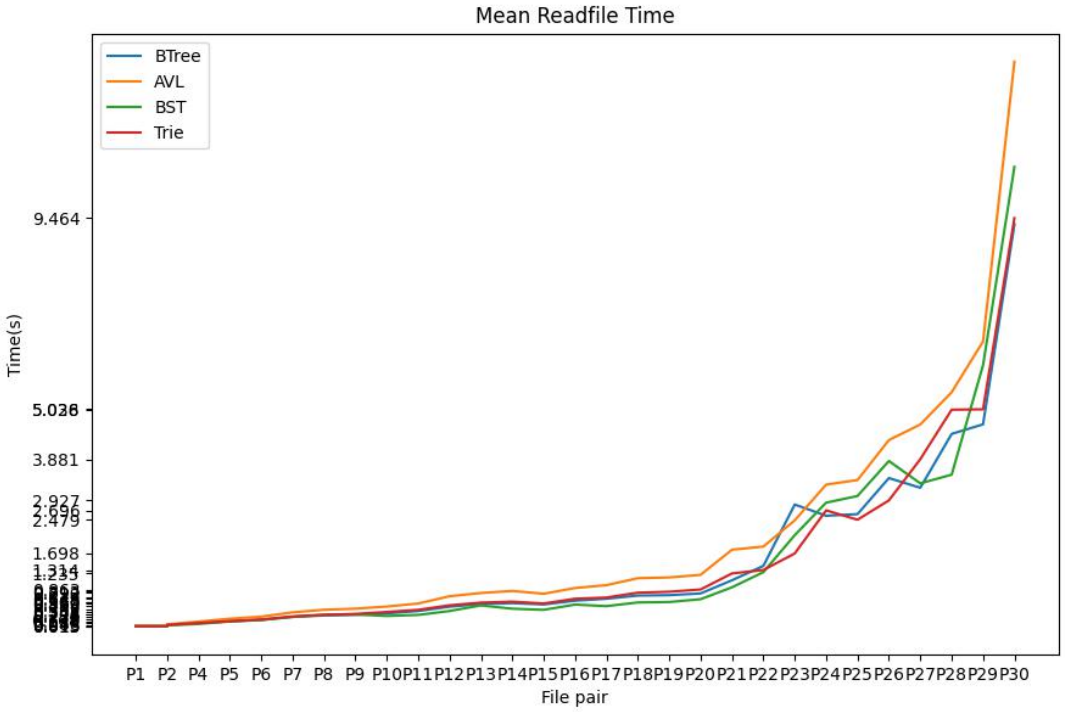
	Độ phức tạp (on average)			Ưu điểm	Nhược điểm
	Không gian	Thời gian			
		Insert	Search		
BTree	O(n)	O(log n)	O(log n)	Giảm đáng kể thời gian truy cập vì yếu tố phân nhánh cao và giảm chiều cao của cây.	Khó cài đặt và sửa chữa dữ liệu
Binary Search Tree	O(n)	O(h)	O(h)	Phân cấp dữ liệu cao	Độ phức tạp tăng khi chiều cao cây quá lớn
AVL Tree	O(n)	O(log n)	O(log n)	Dễ cài đặt, tính hiệu cân bằng cao	Cần thực hiện thao tác cân bằng sau khi chèn
Trie	O(n*k)	O(n)	O(k)	Cài đặt đơn giản. Phù hợp với cấu trúc từ điển	Tốn bộ nhớ khi mỗi node đều phải cấp phát vùng nhớ cho N node con nhất định

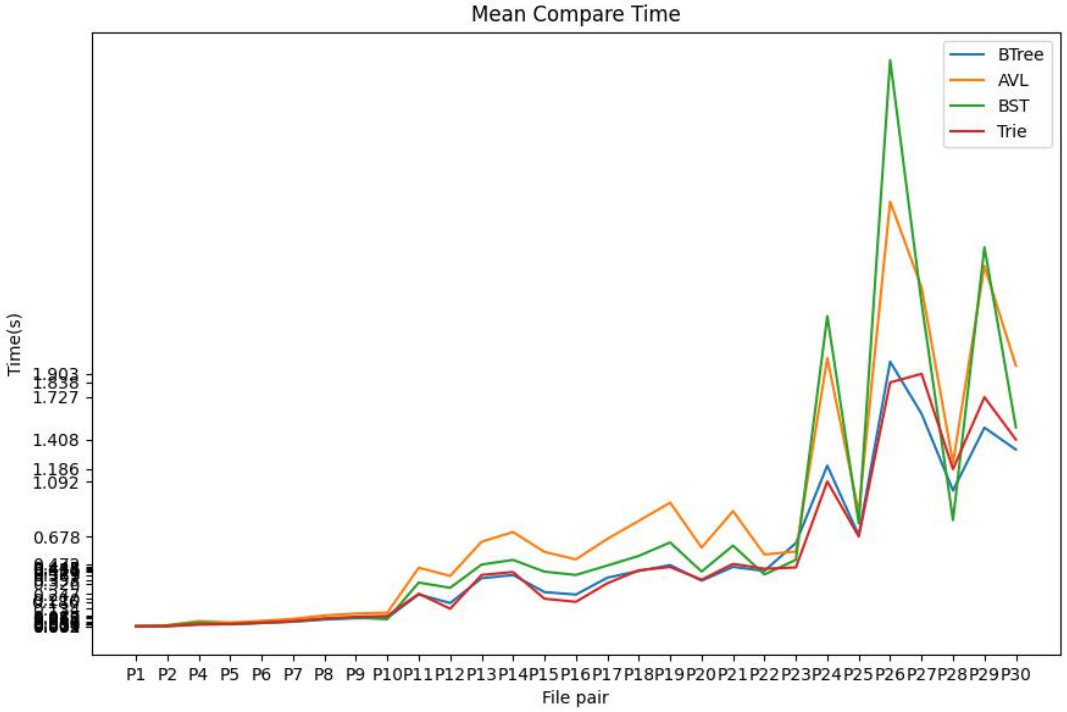
**n: số từ*

**k: độ dài trung bình của các từ*

**h: chiều cao của cây*

Qua việc thực hiện chạy simulate các cặp file văn bản, nhóm đã có được biểu đồ biểu diễn thời gian đọc dữ liệu cũng như thời gian thực hiện thuật toán so sánh của nhóm của bốn loại cấu trúc dữ liệu:


Mean Readfile Time
BTree
AVL
BST
Trie
Time(s)
9.464
3.881
2.927
2.479
1.698
P1 P2 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22 P23 P24 P25 P26 P27 P28 P29 P30
File pair


Mean Compare Time
BTree
AVL
BST
Trie
Time(s)
1.903
1.727
1.408
1.186
1.092
0.678
P1 P2 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22 P23 P24 P25 P26 P27 P28 P29 P30
File pair

Des file size (kb)	Sour file size (kb)	BTree			AVL Tree			Binary Search Tree			Trie		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
1	3	0.011	0.0135	0.018	0.015	0.0192	0.029	0.008	0.0097	0.012	0.012	0.015	0.025
130	137	0.505	0.5178	0.534	0.759	0.7645	0.775	0.369	0.3917	0.437	0.526	0.5393	0.545
207	221	0.739	0.7715	0.991	1.2	1.2033	1.207	0.613	0.6383	0.697	0.836	0.635	0.898
546	711	2.61	3.4438	4.699	4.129	4.3248	4.884	3.665	3.8395	5.512	2.805	2.9272	3.15
1085	2153	7.994	9.3234	13.31	13.026	13.0884	13.218	10.44	10.650	10.84	8.957	9.4642	10.6

Thời gian đọc sour file của các Cấu trúc dữ liệu (s)

Des file size (kb)	Sour file size (kb)	BTree			AVL Tree			Binary Search Tree			Trie		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
1	3	0.001	0.0017	0.003	0.003	0.0042	0.001	0.002	0.003	0.004	0.0	0.001	0.005
130	137	0.257	0.2596	0.262	0.559	0.5621	0.571	0.389	0.4134	0.462	0.208	0.2096	0.212
207	221	0.325	0.3454	0.506	0.592	0.5946	0.6	0.39	0.4143	0.46	0.349	0.353	0.369
546	711	1.618	1.9944	2.936	3.041	3.1994	3.55	3.677	4.2649	5.068	1.709	1.8385	2.084
1085	2153	1.171	1.3328	2.366	1.955	1.9647	1.976	1.348	1.1499	1.74	1.318	1.408	1.688

Thời gian chạy thuật toán so sánh (s)

Qua bảng số liệu ta có thể thấy, thời gian đọc dữ liệu của BTree nhìn chung, ít hơn BST và Trie.

Hơn nữa,

So với cấu trúc Trie, mỗi node của Trie chỉ chứa một ký tự liên quan đến các ký tự phía sau của một từ nên có phần khó khăn khi muốn lưu trữ tất cả các vị trí xuất hiện của các từ.

So với cấu trúc BST, thì BTree có sự tiết kiệm về không gian và thời gian hơn vì có sự giới hạn về chiều cao của cây (Chiều cao tối đa của cây B là $\log_M N$ (M là thứ tự của cây). Ngược lại, chiều cao tối đa của cây nhị phân là $\log_2 N$).

Vì vậy, với tiêu chí tối ưu và phù hợp với yêu cầu của thuật toán, nhóm đã chọn BTree để tổ chức các từ trong văn bản.

***Cấu trúc Btree**

Cấu trúc Bword: Là các phần tử trong 1 Node của Btree, chứa nội dung của từ và vị trí từ trong văn bản

```
struct Bword
{
  vector<int> belong;
  string value;
  void Init(string name, int num)
        {
              value = name;
              belong.push_back(num);
        }
}
```

Cấu trúc Bword

```
Bword* NewBWord(word in)
{
  Bword* temp = new Bword;
  temp->Init(in.value, in.belong);
        return temp;
}
```

Hàm khởi tạo cho Bword

Class BTreeNode: Là cấu trúc 1 Node của Btree, chứa các thuộc tính và các thao tác với Node trên cây Btree.

```
class BTreeNode
{
	int t;	// Minimum degree (defines the range for number of keys)
	BTreeNode** C; // An array of child pointers
	int n;	// Current number of keys
	bool leaf; // Is true when node is leaf. Otherwise false
public:
	Bword* keys;
	BTreeNode(int _t, bool _leaf); // Constructor

	// A utility function to insert a new key in the subtree rooted with
	// this node. The assumption is, the node must be non-full when this
	// function is called
	void insertNonFull(word k);

	// A utility function to split the child y of this node. i is index of y in
	// child array C[]. The Child y must be full when this function is called
	void splitChild(int i, BTreeNode* y);
	void traverse();
	// A function to search a key in the subtree rooted with this node.
	BTreeNode* search(string k, int& index);
	friend class BTree;
};
```

Hàm dựng của Node: Nhận 2 giá trị truyền vào là t1 liên quan đến khởi tạo số phần tử trong Node và số Node con và biến kiểu bool leaf1 đánh dấu Node lá.

```
	BTreeNode::BTreeNode(int t1, bool leaf1)
```

```
{
        // Copy the given minimum degree and leaf property
        t = t1;
        leaf = leaf1;

        // Allocate memory for maximum number of possible keys
        // and child pointers
        keys = new Bword[2 * t - 1];
        C = new BTreeNode * [2 * t];
        // Initialize the number of keys as 0
        n = 0;
}
```

Hàm search giá trị: Tìm kiếm một giá trị trong 1 Node, nếu tìm thấy, trả về Node và giá trị index của phần tử trong Node, ngược lại sẽ tiếp tục tìm trong Node con thích hợp

```
BTreeNode* BTreeNode::search(string k, int& index)
{
        // Find the first key greater than or equal to k
        int i = 0;
        while (i < n)
        {
                if (k.compare(keys[i].value) > 0)
                        i++;
                else
                        break;
        }
        if (i < n)
                // If the found key is equal to k, return this node
                if (k.compare(keys[i].value) == 0)
```

```
                {
                        index = i;
                        return this;
                }
        // If key is not found here and this is a leaf node
        if (leaf == true)
                return NULL;
        // Go to the appropriate child
        return C[i]->search(k, index);
```

Class BTree: là cấu trúc cây của BTree, chứa các Node và các thao tác trên cây Btree.

```
class BTree
{
        BTreeNode* root; // Pointer to root node
        int t; // Minimum degree
public:
        // Constructor (Initializes tree as empty)
        BTree(int _t)
        {
                root = NULL; t = _t;
        }
        // function to traverse the tree
        void traverse()
        {
                if (root != NULL) root->traverse();
        }
        // function to search a key in this tree
        BTreeNode* search(string k, int& index)
        {
```

```
                return (root == NULL) ? NULL : root->search(k, index);
        }
        // The main function that inserts a new key in this B-Tree
        void insert(word k);
};
```

Thao tác insert trên cây BTree: Kiểm tra giá trị được chèn đã tồn tại trên Btree chưa, nếu có thì chỉ thêm vị trí của từ cần chèn vào phần tử đã có trước đó, nếu chưa thì tiếp tục thực hiện thao tác chèn.

```
void BTree::insert(word k)
{
        // If tree is empty

        if (root == NULL)
        {
                // Allocate memory for root
          root = new BTreeNode(t, true);
          root->keys[0] = *NewBWord(k); // Insert key
          root->n = 1; // Update number of keys in root
        }
        else // If tree is not empty
        {
          int index;
          BTreeNode* temp = search(k.value, index);
          if (temp != NULL)
          {
            temp->keys[index].belong.push_back(k.belong);
                return;
```

```
        }
                // If root is full, then tree grows in height
        if (root->n == 2 * t - 1)
        {
                // Allocate memory for new root
                BTreeNode* s = new BTreeNode(t, false);

                // Make old root as child of new root
                s->C[0] = root;

                // Split the old root and move 1 key to the new root
                s->splitChild(0, root);

                // New root has two children now. Decide which of the
                // two children is going to have new key
                int i = 0;
                if (s->keys[0].value.compare(k.value) < 0)
                        i++;

                s->C[i]->insertNonFull(k);

                // Change root
                root = s;
        }
        else // If root is not full, call insertNonFull for root
                root->insertNonFull(k);
    }
}
```

2.2.2.2.4. Các cấu trúc xử lý tối ưu dữ liệu cho thuật toán Levenshtein

Các khối Block

Các khối Block đại diện cho một nhóm câu liên tục có cùng một tính chất. Trong thuật toán, đó là một trong các tính chất: câu không tìm được câu tương đồng, câu có độ tương đồng cao, câu có độ tương đồng trung bình.

BlockX đại diện cho các khối Block của văn bản des.

BlockY đại diện cho các khối Block của văn bản sour.

Tổ chức dữ liệu đầu vào thuật toán Levenshtein theo các Block thay vì theo các câu sẽ tối ưu được đáng kể độ phức tạp của thuật toán.

Cấu trúc Block

Cấu trúc BlockY

```
typedef struct Blockx
{
  bool match = false;
  int index;
  int first = -1;
  int last = first;
  float value = 0;
  int count = 0;
  float similar = 0;
  BlockY y;
  void Add(int X, float Value)
  {
          if (count == 0)
          {
            first = X;
            last = first;
            value = Value;
          }
          else
          {
            last = X;
            value += Value;
          }
          similar += 1 - Value;
          count++;
```

Cấu trúc BlockY

```
typedef struct Blocky
{
        long index;
        int first = -1;
        int last = first;
        int count = 0;
        bool match = false;
        void Add(int Y)
        {
                if (count == 0)
                {
                        first = Y;
                        last = first;
                }
                else
                        last = Y;
                count++;
                return;
        }
} *BlockY;
```

```
  }
}*BlockX;
```

Trong đó:

- bool match: Đánh dấu Block rỗng hay không.
- long index: Tọa độ của Block
- int first, last: Giá trị đầu vs cuối của Block
- float value: Chứa tổng độ khác nhau giữa hai Block (Đối với BlockX khác rỗng);
-float similar: Chứa tổng độ giống nhau giữa hai Block (Đối với Block khác rỗng);

Cấu trúc Hash- Cấu trúc chính xử lý việc phân định Block cho các văn bản

struct Hash
{
int count = 0;
int width;
int depth;
unordered_map<long,Dif> map;
vector<BlockX> t;
vector<BlockY> v;
BlockX nowX;
BlockY nowY;
	Trong đó:
	- unordered_map<long,Dif> map: Sử dụng cấu trúc Hashtable có sẵn trong thư viện C++, chứa tọa độ, cũng như độ khác nhau của các cặp Block khác Block rỗng.
	
	* Lý do sử dụng unordered_map:
	+ Dữ liệu vào không cần thứ tự
	+ Độ phức tạp insert trung bình là

bool iswait = false;
int lastx;
int lasty;
int isempty = 1;
float sum = 0;
void Getnode(int x, int y, float value)(…)
void Dataprocess()(…)
void CreateHashmap()(…)
float Get(int x, int y)
{
long key = x * width + y;
if (map[key].value == -1)
{
if (x == 0 && y == 0)
return 0;
else if (x == 0)
return v[y - 1]->count;
else if (y == 0)
return t[x - 1]->count;
else
return v[y - 1]->count;
}
else
return map[key].value;
}
float Getx(int x)
{
return t[x - 1]->count;
	O(1)
	+ Chỉ cần việc truy xuất dữ liệu, không cần việc xuất theo thứ tự
	
	- vector<BlockX> t: vector chứa các Block của văn bản des
	- vector<BlockY> v: vector chứa các Block của văn bản sour
	
	- BlockX nowX và - BlockY nowY: Các khối Block đang xét hiện tại.
	
	- int width, depth: lưu trữ các kích thước của các vector Block (cũng là kích thước của mảng Levenshtein khi chạy qui hoạch động).
	
	- Các hàm Getnode, DataProcess, CreateHashmap liên quan đến thao tác phân định Block, sẽ được trình bày rõ hơn ở phần giải thuật.
	
	- Các hàm truy xuất giá trị từ map: Get(x,y): Trả về giá trị độ khác nhau giữa hai Block truyền vào, nếu không tồn tại giá trị

}
float Gety(int y)
{
return v[y - 1]->count;
}
	trong map, sẽ trả về độ khác nhau lớn nhất (chính bằng số phần tử trong Block tương ứng)

2.2.2. Giải thuật

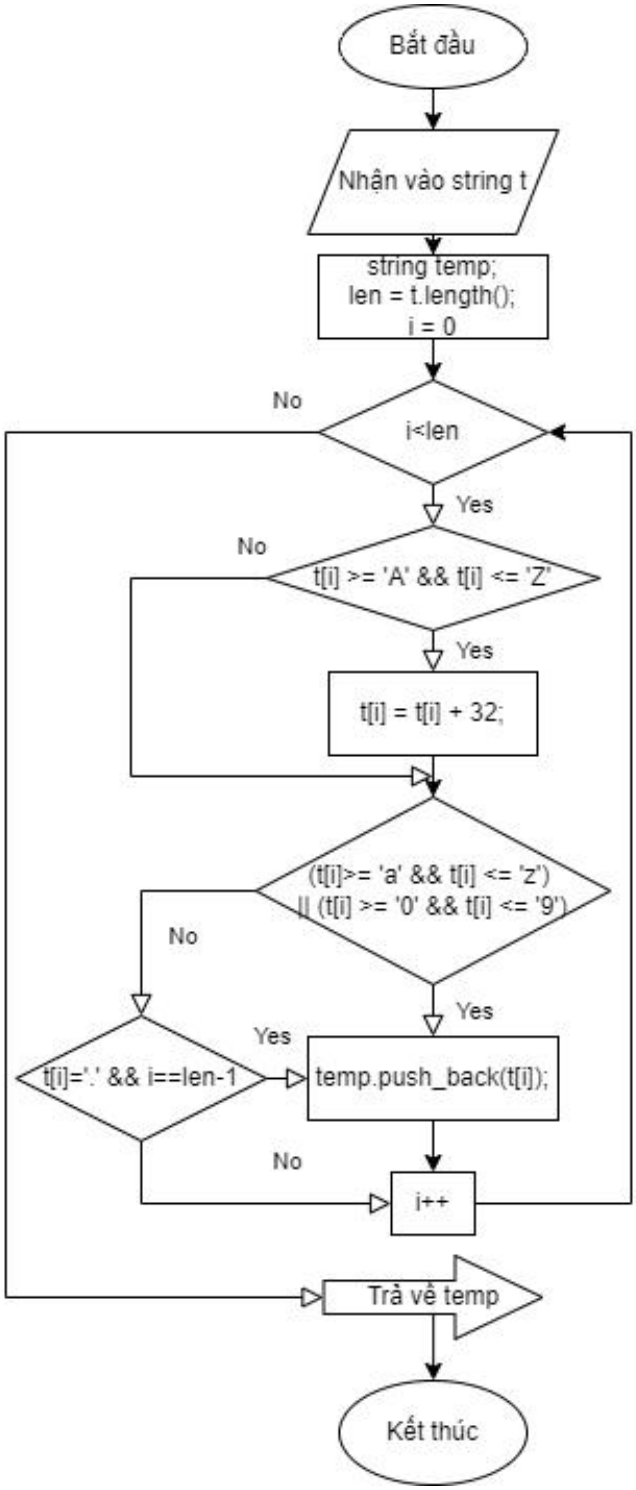
Ý tưởng thực hiện chung: Dựa vào thuật toán Levenshtein, xem mỗi câu trong văn bản là mỗi kí tự trong chuỗi, lập mảng hai chiều, dùng qui hoạch động để tìm ra độ giống nhau của hai văn bản.

Vấn đề được đặt ra là tính toán mức độ giống nhau của hai câu văn (tương tự với việc so sánh hai kí tự trong thuật toán Levenshtein trên một chuỗi) sao cho tối ưu nhất. Khi văn bản trở nên lớn hơn, số câu văn trong văn bản rất nhiều (vài nghìn câu) thì có thể tối ưu các câu trong văn bản thành cấp dữ liệu khác hiệu quả hơn không.

Dưới đây là chi tiết các thuật toán mà nhóm đã sử dụng để giải quyết bài toán

2.2.2.1. Giải thuật phân loại, đọc và xử lý dữ liệu từ file lên cấu trúc dữ liệu

2.2.2.1.1. Thao tác xử lý kí tự trên từng từ



Với mục đích đảm bảo sự chính xác, thuận tiện trong quá trình so sánh các chuỗi, thao tác này có tác dụng loại bỏ đi những ký tự không phải là chữ số, đồng thời chuyển các ký tự hoa sang thường nhằm tạo sự đồng bộ trong quá trình so sánh.

Các bước thực hiện như lưu đồ bên

Input: Một từ được đọc vào từ file

Output: Một string đã được xử lý, tuy nhiên vẫn giữ dấu chấm cuối từ (nếu có) nhằm mục đích phân biệt các câu.

Thuật toán kết thúc khi toàn bộ các ký tự của chuỗi được duyệt qua.

Độ phức tạp về thời gian: O(m) với m là độ dài của từ.

2.2.2.1.2. Cập nhật từ điển stopword

Việc đánh dấu từ nào là một stopword giúp cho việc tìm kiếm trở nên dễ dàng và chính xác hơn bởi khi tìm kiếm sẽ loại bỏ qua được các từ này, chỉ tập trung tìm kiếm các từ có nghĩa khác.

Các bước thực hiện:

Bước 1: Mở file txt chứa các stopword thông dụng (76 từ)

Bước 2: Đọc từng stopword vào mảng.

Bước 3: Xây dựng hàm look_up(), sử dụng binary_search trên mảng đó để xác định xem một từ cho trước có phải là một stopword hay không.

```
string* loadwords(const string& filename)
{
	string* arr = new string[76];
	ifstream file(filename);
	string data;
	int i = 0;
	while (getline(file, data))
	{
		arr[i++] = data;
	}
	file.close();
	return arr;
}
bool lookup(string* words, const string& word)
{
	return binary_search(words, words + 76, word);
}
string* dictionary = loadwords("stopword.txt");
```

2.2.2.1.3. Phân loại file đầu vào

Như đã trình bày ở phần cấu trúc dữ liệu, hai file đầu vào sẽ được tổ chức thành hai loại là des và sour

Thực hiện các bước tính toán để biết được file nào có dung lượng lớn hơn (có số từ và câu lớn hơn) và file nào có dung lượng nhỏ hơn.

Đối với trường hợp hai file có cùng kích thước thì việc phân loại hay không sẽ không ảnh hưởng nhiều tới thời gian thực hiện thuật toán so sánh, tuy nhiên, đối với những file có dung lượng khác nhau lớn, việc phân chia này mang lại hiệu quả cao trong việc thực hiện thuật toán so sánh phía sau.

Code thực hiện việc tính toán dung lượng hai file như sau:

```
long Getsize(const string name)
{
	long l, m;
	ifstream file(name, ios::in | ios::binary);
	l = file.tellg();
	file.seekg(0, ios::end);
	m = file.tellg();
	file.close();
	return m;
}
```

Input: Tên file/ Đường dẫn tới vị trí của file

Output: Một số nguyên cho biết dung lượng (byte) của file.

2.2.2.1.4. Đọc file vào cấu trúc dữ liệu

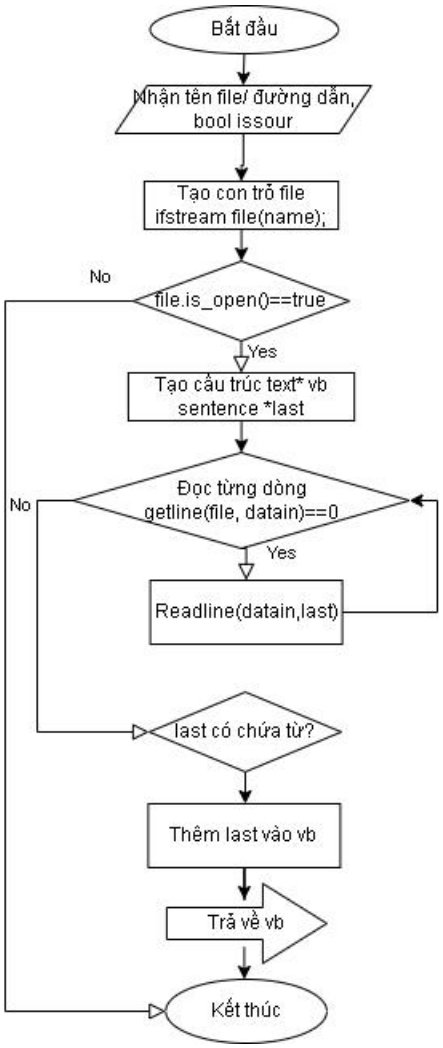
Bằng việc xác định file lớn hơn và nhỏ hơn thông qua hàm Getsize ở trên, file lớn hơn sẽ được đánh dấu issour=true, khi đó, dữ liệu khi được add vào sẽ tự động thêm vào cấu trúc BTree.

Thực hiện đọc dữ liệu theo sơ đồ giải thuật bên dưới:

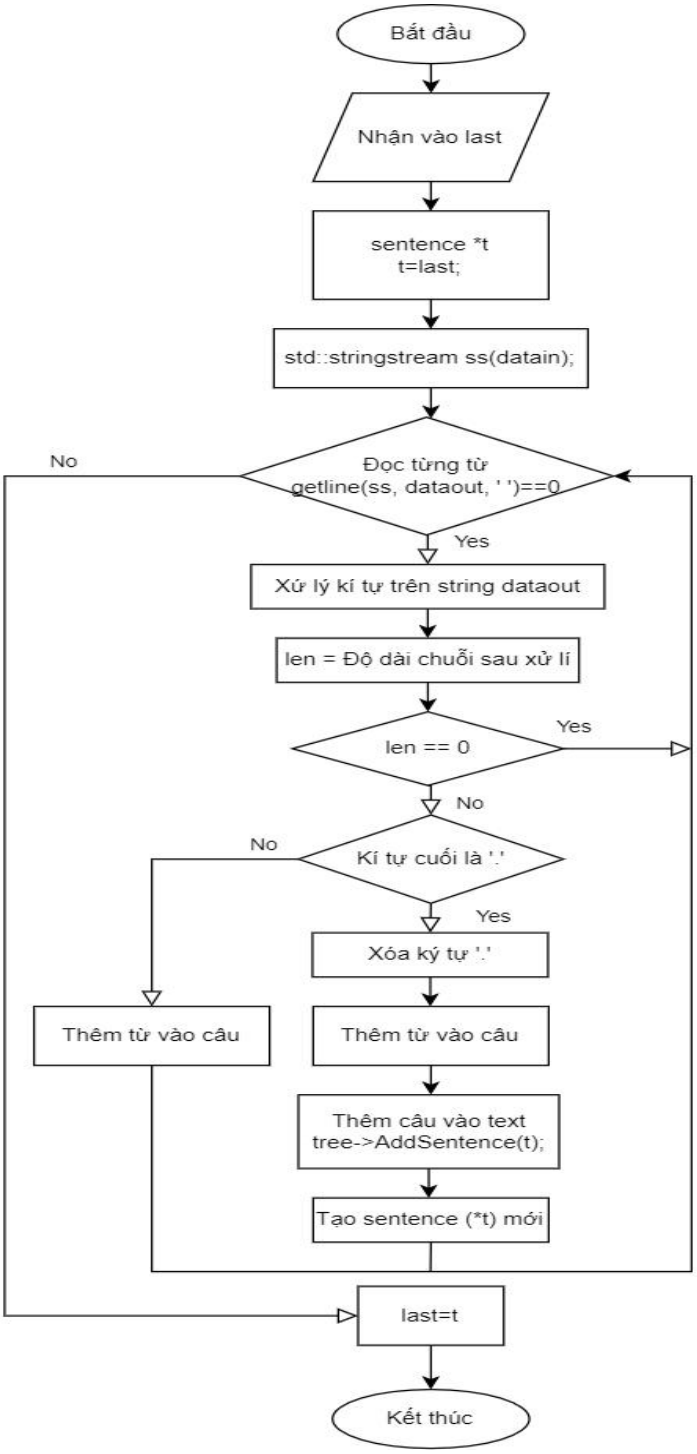
Input: Tên file/ Đường dẫn file và giá trị bool cho biết file des hay sour.

Output: Cấu trúc text chứa toàn bộ thông tin trong văn bản.

Thuật toán sẽ kết thúc khi dữ liệu trong văn bản được đọc hết.



Hàm CreateText



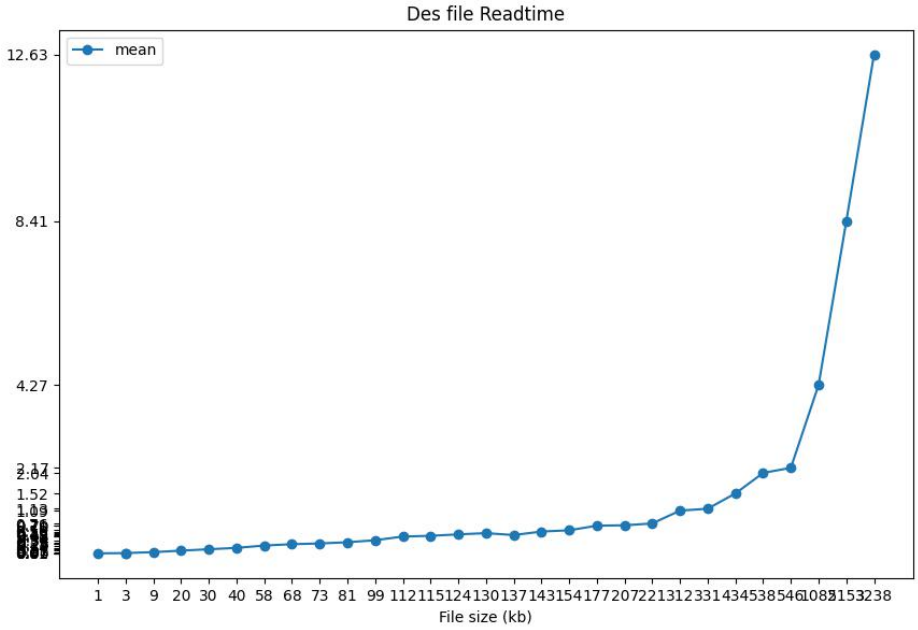
Hàm ReadLine

Đánh giá tính hiệu quả của thuật toán:

Thông qua chạy Simulate với 30 cặp file, nhóm thu được dữ liệu sau:

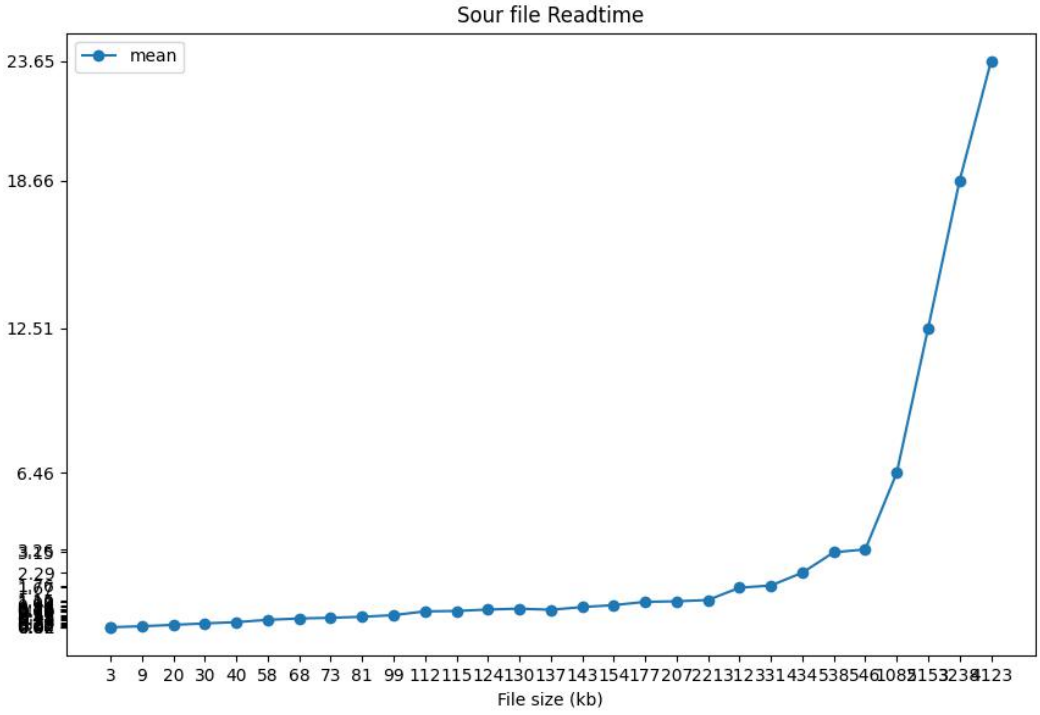
Đối với file des:

Dung lượng file des (kb)	Min	Mean	Max
1	0.004	0.0045	0.005
58	0.200	0.201	0.205
112	0.428	0.430	0.433
130	0.504	0.561	0.568
207	0.71	0.714	0.724
434	1.504	1.52	1.56
546	2.08	2.16	2.27
1085	4.249	4.268	4.311
2153	8.385	8.413	8.525
3238	12.595	12.63	12.74



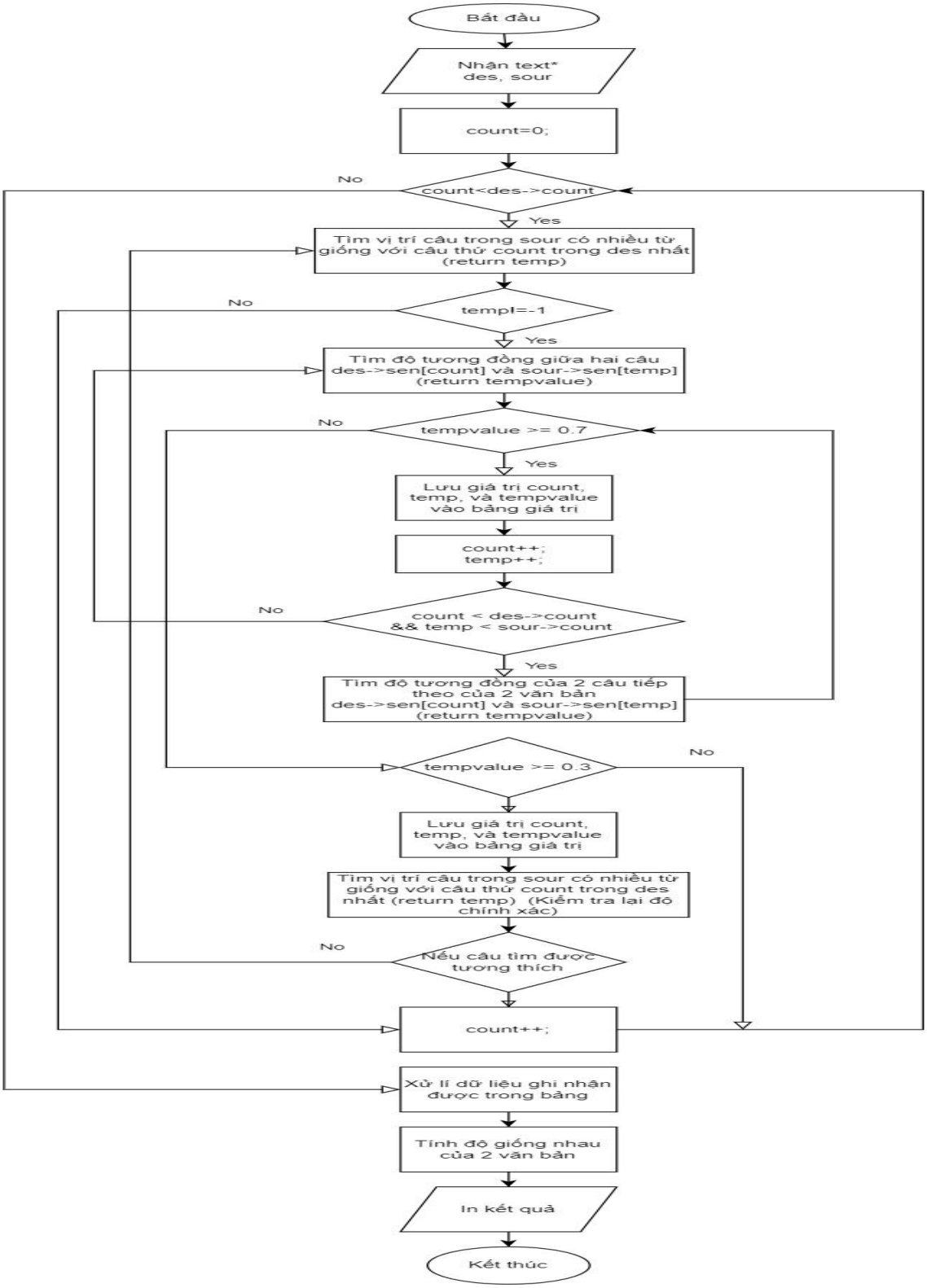
Đối với file sour

Dung lượng file (kb)	Min	Mean	Max
3	0.016	0.017	0.022
58	0.323	0.326	0.332
112	0.667	0.679	0.713
130	0.786	0.791	0.814
207	1.089	1.09	1.157
434	2.256	2.293	2.398
546	3.115	3.145	3.265
1085	6.315	6.45	3.392
2153	12.471	12.5	12.56
3238	18.61	18.67	18.73



2.2.2.2. Giải thuật so sánh độ giống nhau của hai văn bản

2.2.2.2.1. Giải thuật so sánh lớn



Giải thuật được thực hiện theo lưu đồ trên.

Input: Hai cấu trúc văn bản des và sour được xây dựng trước đó

Output: Kết qua ra màn hình độ giống nhau của hai văn bản

Phân tích thuật toán:

Đầu tiên là lấy câu văn đầu tiên trong văn bản des, thông qua việc tìm kiếm trên cây BTree (được tạo từ các từ trong văn bản dài hơn), sẽ tìm được câu văn trong sour có độ tương thích lớn nhất với câu đã lấy ra (*Hàm Findthefirst, được trình bày ở mục sau)*. Sau đó, áp dụng thuật toán Levenshtein vào tính toán độ giống nhau của 2 câu văn đó (*Hàm FindSimilarity, được trình bày ở mục sau*). Kết quả cho ra số thực x (x>=0 && x<=1), x dần về 1 ứng với việc 2 câu văn càng giống nhau, ngược lại thì 2 câu càng khác nhau.

Nếu kết quả cho ra x>0.7, tức là độ tương thích của hai câu là trên 70% thì ghi nhận kết quả (1-x) và hai giá trị vị trí của hai câu (sử dụng như hai tọa độ) vào một bảng giá trị để sử dụng trong việc tính toán "Big Levenshtein" (ở cấp độ so sánh 2 văn bản) và ta tiếp tục tính toán độ giống nhau của các cặp câu tương ứng tiếp theo. (*Cấu trúc bảng lưu giá trị được trình bày ở mục 2.2.2.2.4)*

Nếu kết quả cho ra 0.7>x>0.3, tức là độ tương thích của 2 câu không nhiều, nhưng cũng không ít thì thuật toán lại quay lại bước tìm kiếm trên cây BTree để tìm ra câu trong văn bản sour tương đồng nhất với câu đang xét, nếu kết quả cho ra trùng khớp với cặp câu đang xét thì ghi nhận kết quả trên, và xét câu tiếp theo của văn bản des.

Nếu kết quả cho ra x<0.3, tức độ tương đồng của 2 câu là không nhiều, thì coi như hai câu hoàn toàn khác nhau, x sẽ không được ghi nhận, và chuyển đến xét câu tiếp theo.

Thuật toán kết thúc khi tất cả các câu văn của văn bản ngắn hơn đều được duyệt.

Các cặp câu không tìm được câu tương đồng, x sẽ qui ước bằng 1 (tương tự với trường hợp cost=1 khi hai kí tự khác nhau trong thuật toán Levenshtein với hai chuỗi).

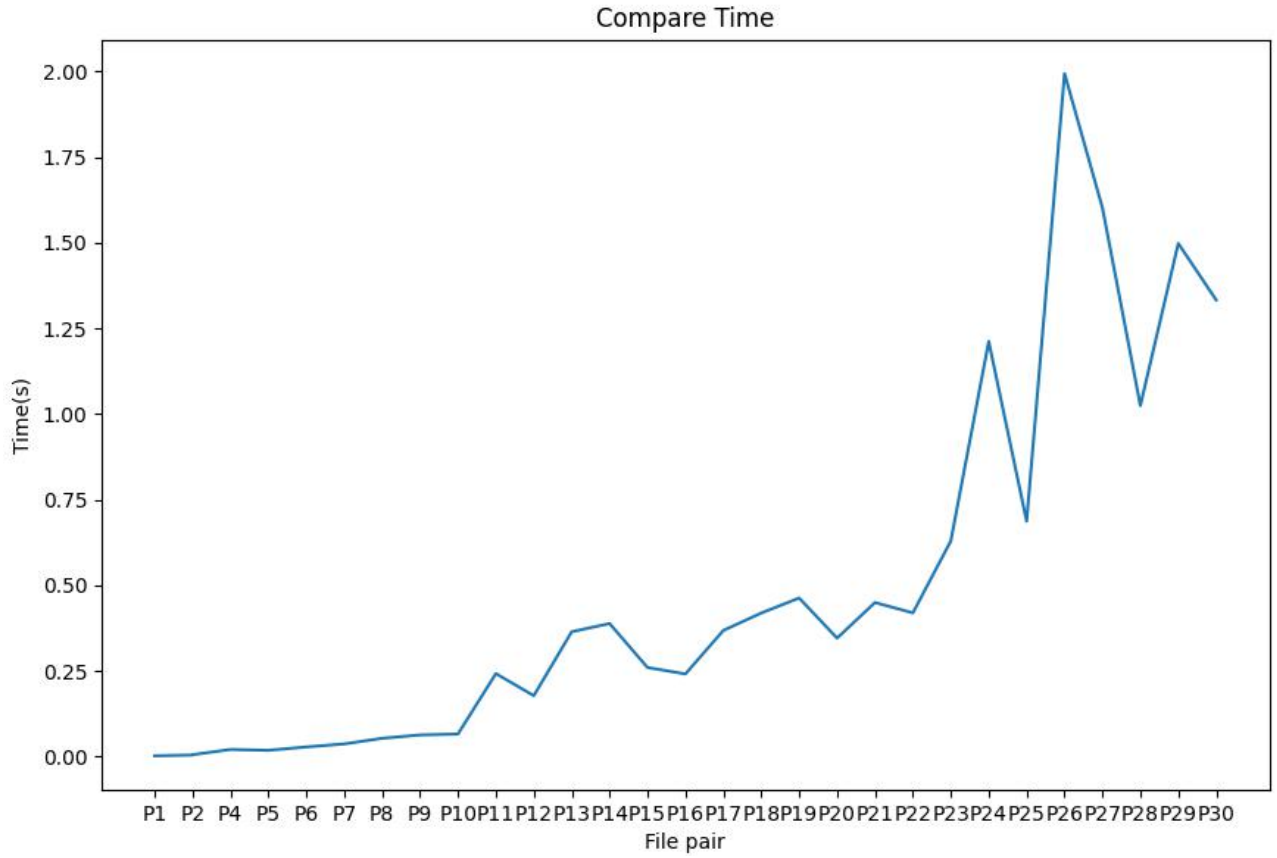
Cuối cùng là áp dụng "Big Levenshtein" với các giá trị cost khi so sánh 2 câu văn là các giá trị x tìm được trong thuật toán được trình bày ở trên.

Đánh giá hiệu quả của thuật toán:

Thuật toán sẽ nhanh chóng tìm và đánh giá được các cặp câu có độ tương đồng cao. Đồng thời, với thao tác duyệt liên tục các cặp câu liên tiếp khi phát hiện câu có độ tương đồng cao giúp giảm thời gian tìm kiếm các cặp câu xuống tới mức đáng kể.

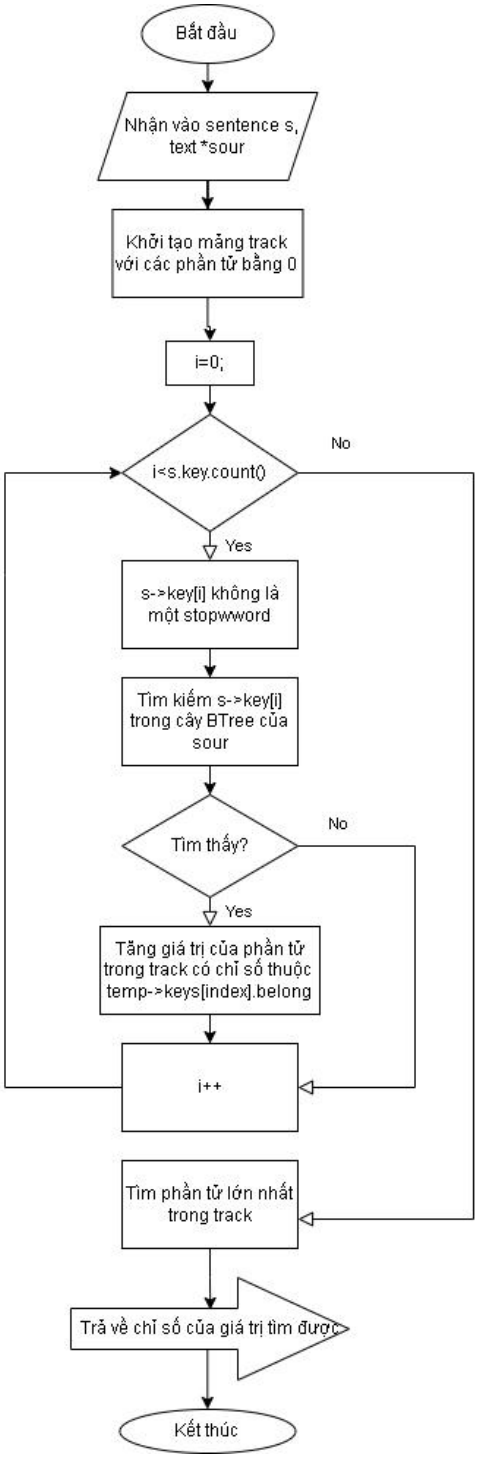
Sự phân chia tìm kiếm các cặp câu tương đồng dựa vào văn bản des (có số câu ít hơn) làm gốc giúp giảm độ phức tạp trong quá trình tìm kiếm.

Thông qua chạy simulate với 30 cặp file, mỗi cặp 10 lần, nhóm thu được dữ liệu sau:

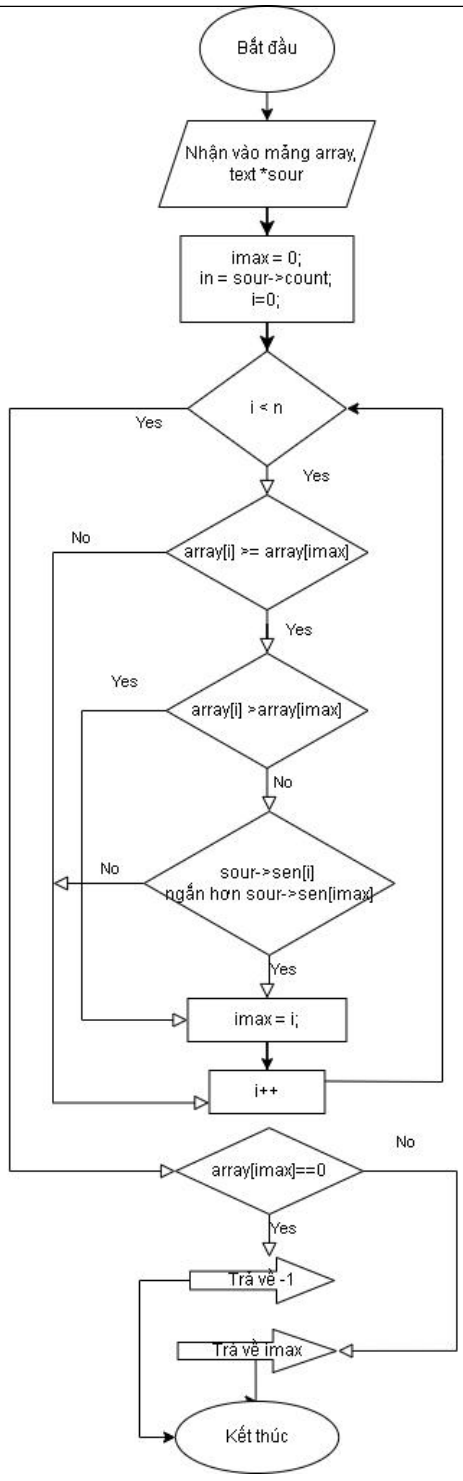


First file size (kb)	Second file size (kb)	Min	Mean	Max
1	3	0.001	0.0017	0.003
58	68	0.051	0.0528	0.055
112	124	0.354	0.364	0.403
130	137	0.257	0.2596	0.262
207	221	0.325	0.234	0.506
434	538	0.95	1.212	1.882
546	711	1.618	1.9944	2.936
1085	2153	1.171	1.3328	2.366

2.2.2.2.2. Giải thuật tìm kiếm câu có nhiều từ tương đồng với câu được xét (void Findthefirst)



Thuật toán FindtheFirst



Thuật toán tìm phần tử lớn nhất

Phân tích giải thuật Findthefirst:

Input: Một câu (sentence) cần xét từ văn bản des, text *sour

Output: Một số nguyên chỉ vị trí của câu đã tìm được trong văn bản sour, nếu không tìm được câu phù hợp, trả về -1.

Bắt đầu, chương trình tạo một mảng track với số phần tử bằng với số câu trong văn bản sour, sau đó lần lượt xét các từ của sentence đã cho.

Nếu từ đang xét không phải là một stopword thì sẽ tìm kiếm sự tồn tại của từ đó trong BTree của sour text bằng hàm search của cấu trúc BTree. Nếu chỉ số trả về không âm, ta lần lượt tăng giá trị của các phần tử mảng track có chỉ số thuộc vector chỉ vị trí xuất hiện của từ tìm được (được lưu tại mỗi key của Node BTree). Ở đây, không duyệt sự xuất hiện của các stopword bởi stopword xuất hiện ở hầu hết các câu văn, việc tránh đi các stopword vừa giúp giảm được số lần tìm kiếm vừa làm tăng sự chính xác trong quá trình tìm khi chỉ duyệt trên những từ vựng có nghĩa.

Như vậy, sau khi duyệt qua các từ trong câu đã cho, ta thu được mảng track chứa số lượng từ tương đồng với câu đã cho của từng câu trong text sour.

ĐPT:

Phân tích giải thuật tìm phần tử tương thích nhất trong track:

Input: Mảng track

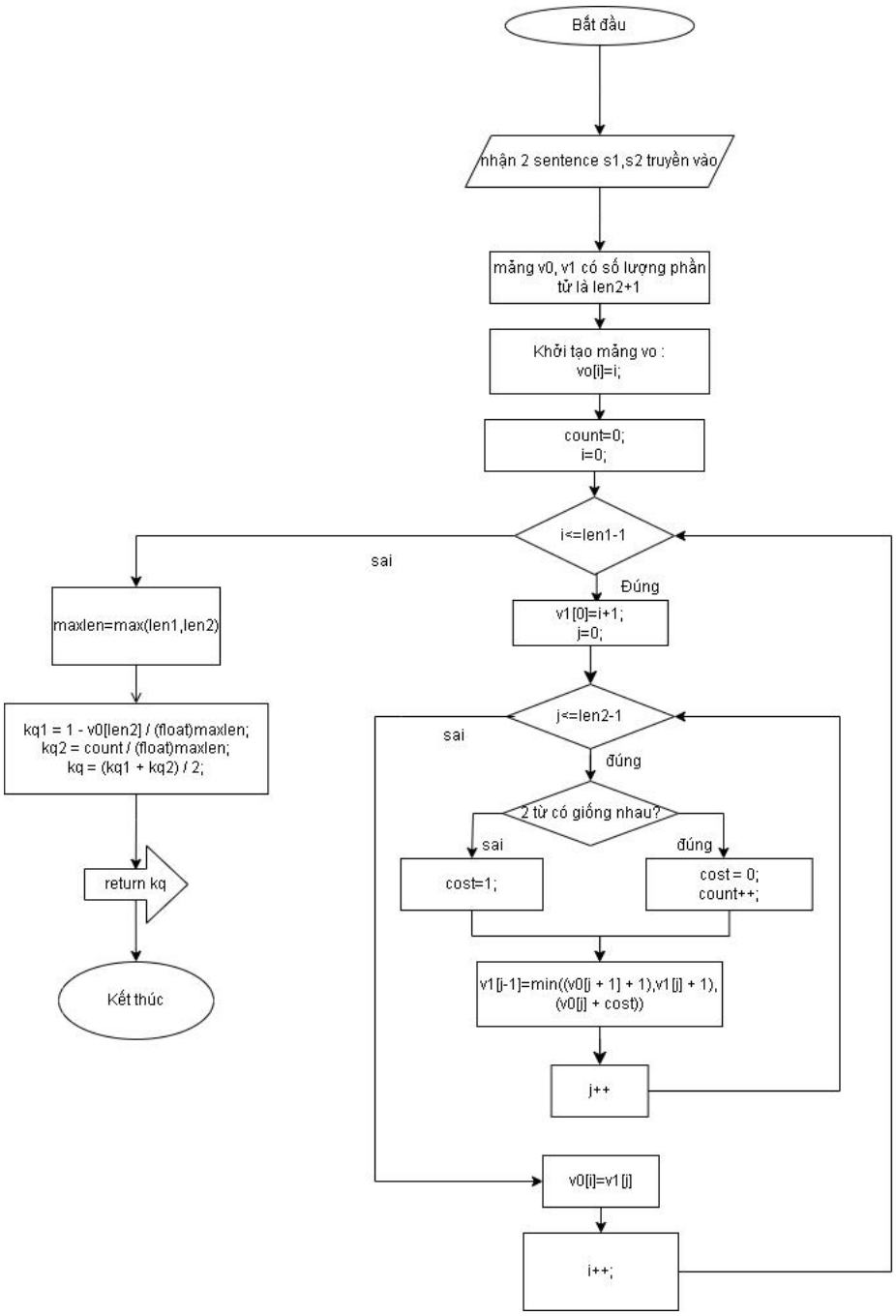
Output: vị trí của câu phù hợp nhất (giá trị max)

Tương tự với các giải thuật tìm giá trị lớn nhất, ở đây, khi có nhiều câu có cùng giá trị lớn nhất, thuật toán sẽ ưu tiên chọn câu có độ dài ngắn nhất làm giá trị trả về.

Như vậy, tóm lại về mức độ hiệu quả của thuật toán, Findthefirst sẽ tránh được việc phải duyệt tuần tự cả văn bản sour để tìm ra câu văn tương đồng

2.2.2.2.3. Giải thuật ứng dụng thuật toán Levenshtein vào xác định chính xác độ tương đồng giữa hai câu.

Giải thuật được thể hiện qua lưu đồ sau:



Phân tích giải thuật:

Input: Hai câu (sentence) được truyền vào từ thuật toán Compare chung

Output: Giá trị độ giống nhau của 2 câu x (x>=0 && x<=1)

Đánh giá mức độ hiệu quả của thuật toán:

Đánh giá về thuật toán levenstein: Có thể xem thuật toán Levenshtein đánh giá mức độ giống nhau của hai chuỗi về phương diện khoảng cách giữa các từ. Điều này gây ra nhược điểm khi áp dụng vào so sánh hai câu văn khi ví dụ 2 câu văn chỉ là sự đổi vế câu cho nhau, hay nói cách khác là nội dung giống nhau nhưng thứ tự của các từ là khác nhau, levenshtein sẽ cho ra kết quả là hai câu này khác nhau hoàn toàn

Thuật toán được trình bày trên lưu đồ sẽ giúp khắc phục được hạn chế trên, lồng ghép bên cạnh các bước truyền thống của levenshtein, nhóm đã thực hiện đếm số lượng từ giống nhau của 2 câu. Như vậy, sau khi hoàn thành các bước quy hoạch động, ta sẽ có được một là levenshtein distance, hai là lượng nội dung giống nhau của hai câu.

Tiến hành chuyển từ lenvenshtein distance sang độ giống nhau của hai câu, đồng thời tính độ giống nhau về mặt nội dung bằng cách lấy lượng từ giống nhau đo được chia cho độ dài câu văn dài hơn. (Hai kết quả này sẽ giống nhau nếu trường hợp các từ giống nhau trong hai câu không có sự khác nhau về vị trí).

Lấy trung bình cộng của hai kết quả trên, ta thu được độ giống nhau của hai câu.

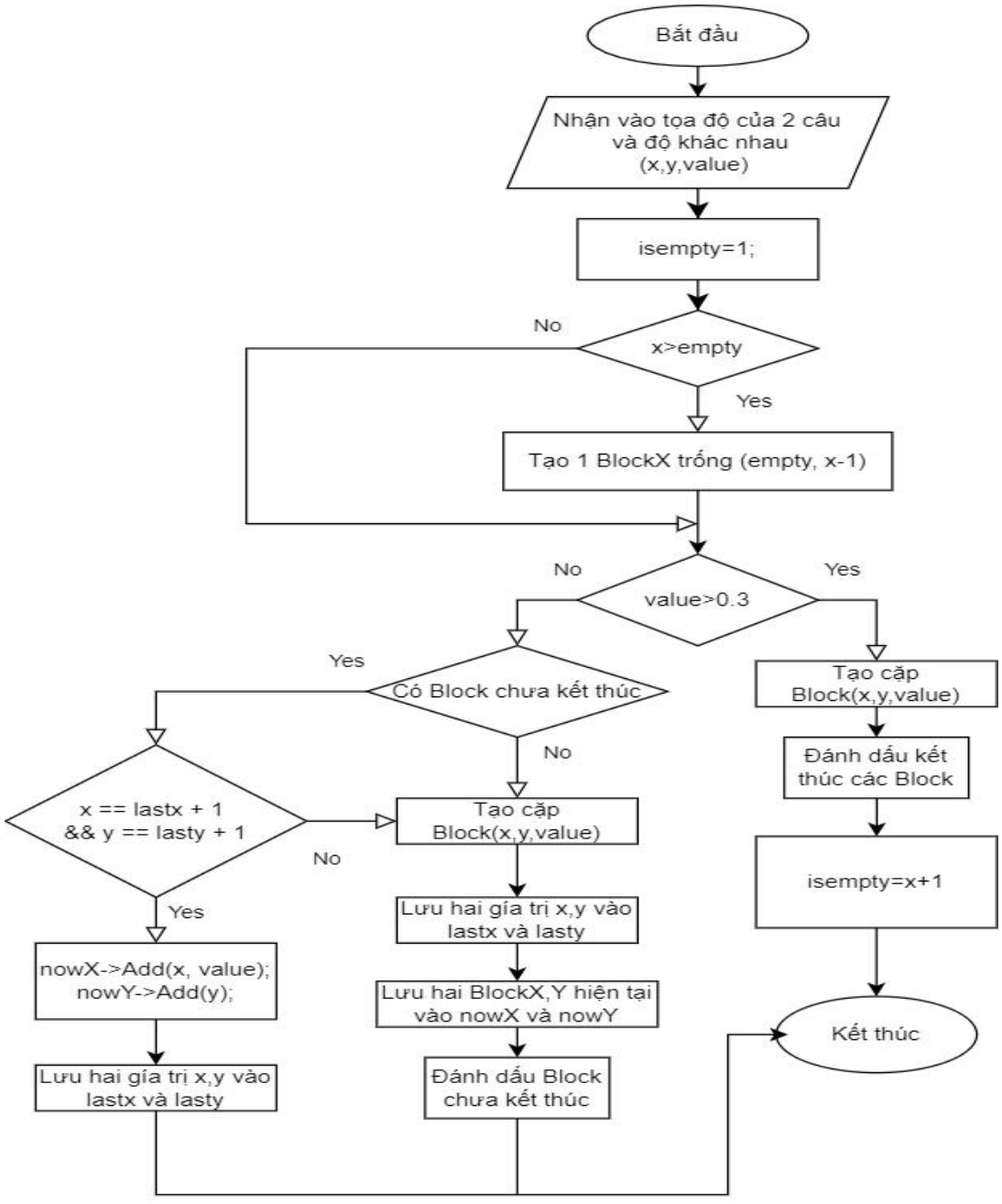
Độ phức tạp của thuật toán: O(mxn) với m,n lần lượt là độ dài của hai câu.

2.2.2.2.4. Giải thuật tối ưu hóa cấu trúc dữ liệu đầu vào cho Thuật toán Levenshtein cấp độ hai văn bản.

Như đã trình bày ở các phần trên, việc phân định cấu trúc văn bản từ các câu sang các khối Block sẽ tối ưu được đáng kể độ phức tạp của Thuật toán Levenshtein khi hoạt động trên các khối Block thay vì trên các câu văn.

Sau đây là các giải thuật thực hiện:

Hàm Getnode() trong cấu trúc Hash:




```
Tạo 1 block trống (first, last)

Block q = new Block;

q->Add(first, 1);

q->Add(last, 1);

q->count = q->last - q->first + 1;

vector.push_back(q);
Kết thúc các Block hiện có
```

```
Tạo cặp block (x,y,value)

BlockX m = new Blockx;
BlockY n=new Blocky;

m->Add(x, value);

n->Add(y);

m->y=n;

t.push_back(m);
v.push_back(n);
```

Giải thuật phân định BlockX,BlockY theo dữ liệu truyền từ hàm Compare vào.

Phân tích thuật toán:

Input: Giá trị vị trí của hai câu trong hai văn bản có độ tương đồng trung bình trở lên giữa des, sour và độ khác nhau giữa chúng.

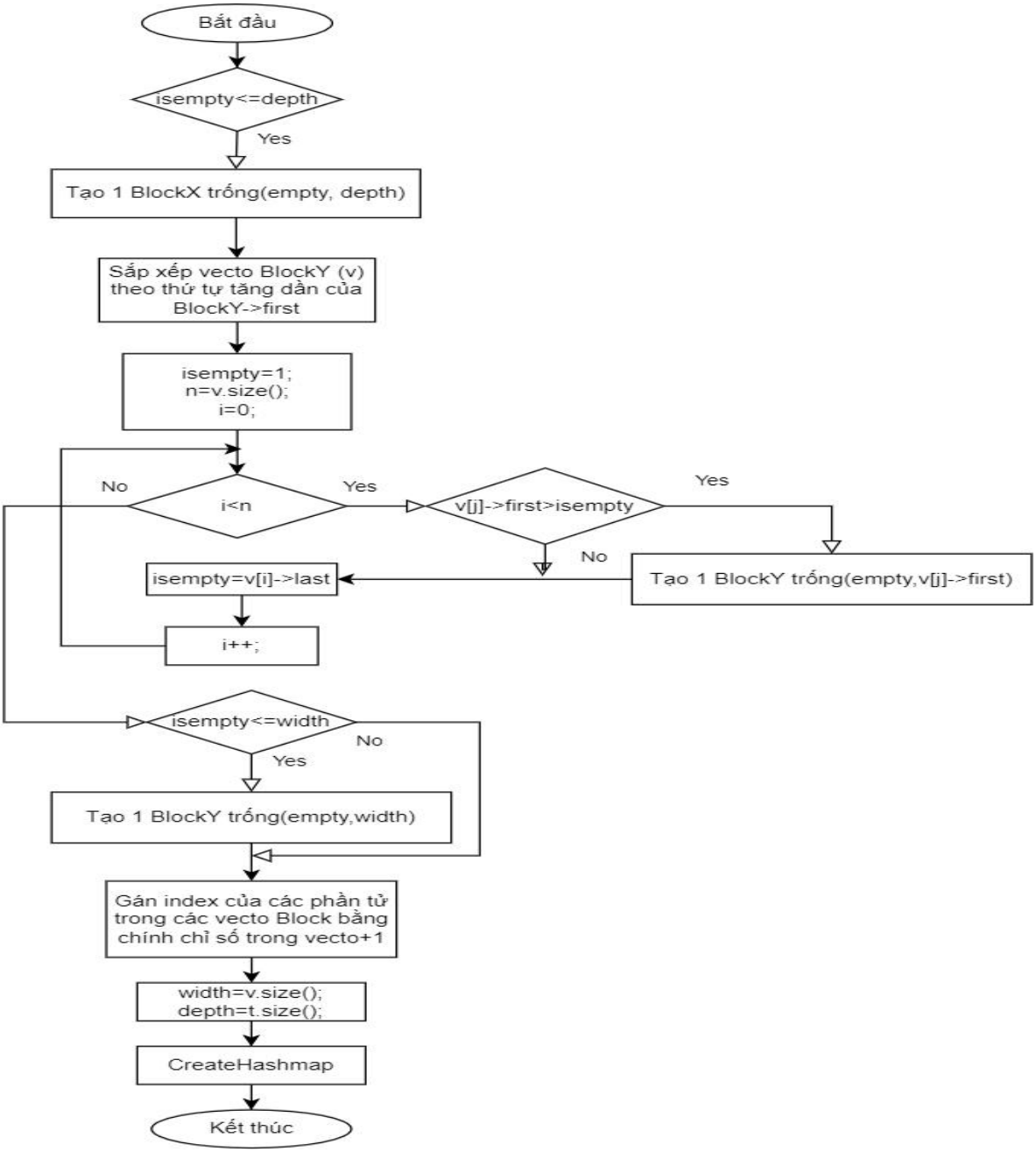
Output: Các phân định BlockX, BlockY được hình thành dần sau mối lần thêm dữ liệu. Thuật toán kết thúc với BlockX, Y được phân định tùy theo giá trị truyền vào.

Đánh giá mức độ hiệu quả của thuật toán:

Độ phức tạp của thuật toán: O(1)

Thuật toán tạo bước tiền đề cho bước xử lý dữ liệu phía sau (void Dataprocess), hạn chế việc sử dụng một không gian lưu trữ phụ so với chờ việc đọc hết toàn bộ dữ liệu rồi mới xử lý.

Hàm DataProcess trong cấu trúc Hash:Hoàn thiện việc phân định Block



Phân tích thuật toán:

Input: Nhận các vecto BlockX,Y, các Block nowX,nowY từ kết quả xử lý của hàm Getnode;

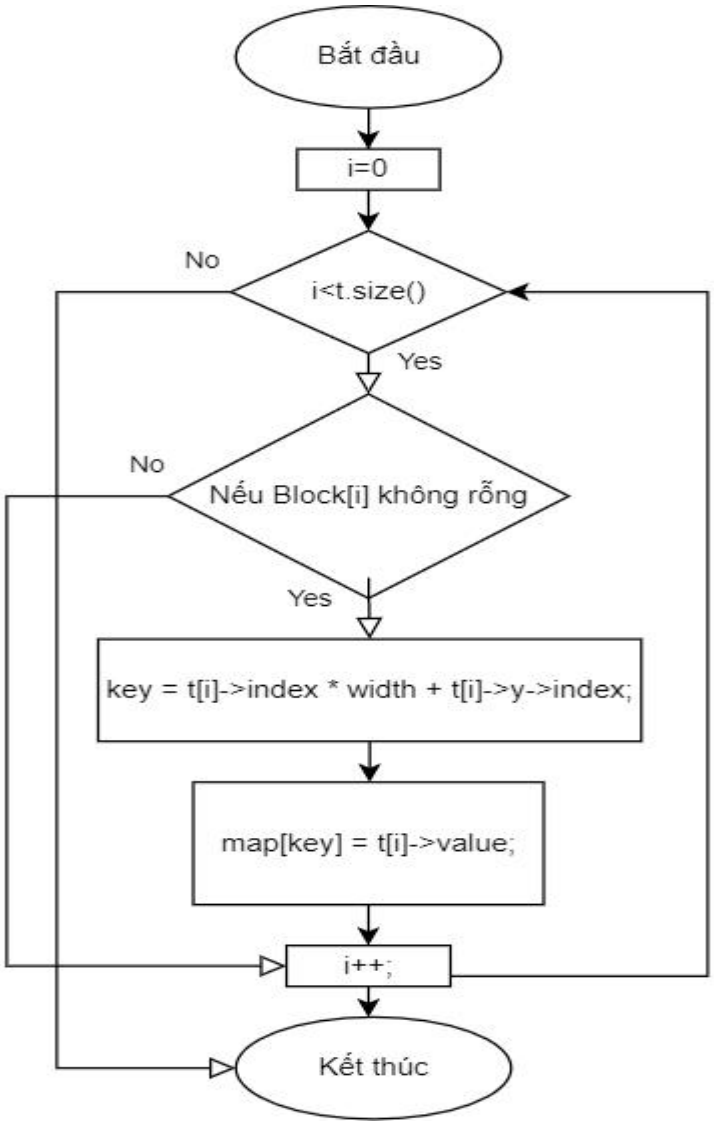
Output: Các vector BlockX,Y hoàn thiện chứa đầy đủ các Block phân định được trong hai văn bản.

Thuật toán dừng lại khi hoàn tất việc phân định Block cho hai văn bản

Đánh giá mức độ hiệu quả của thuật toán:

Thuật toán hoàn thiện và bổ sung các phần cần thiết cho việc phân định Block : hoàn thiện các Block , sắp xếp lại các BlockY theo thứ tự, gán chỉ số cho các BlockX,Y (được sử dụng như tọa độ trong mảng qui hoạch động trong thuật toán Levenshtein), chuẩn bị đầy đủ các dữ liệu để tạo bảng hash có thể truy xuất dữ liệu dễ dàng.

Hàm CreateHashmap trong cấu trúc Hash: Đưa dữ liệu Block xử lý được vào cấu trúc có thể truy xuất dễ dàng.



Phân tích thuật toán:

Input: Sử dụng mảng vector BlockX từ xử lý của các hàm trên.

Output: Bảng hash chứa key đặc trưng cho bộ dữ liệu tọa độ x,y và value là độ khác nhau giữa hai Block.

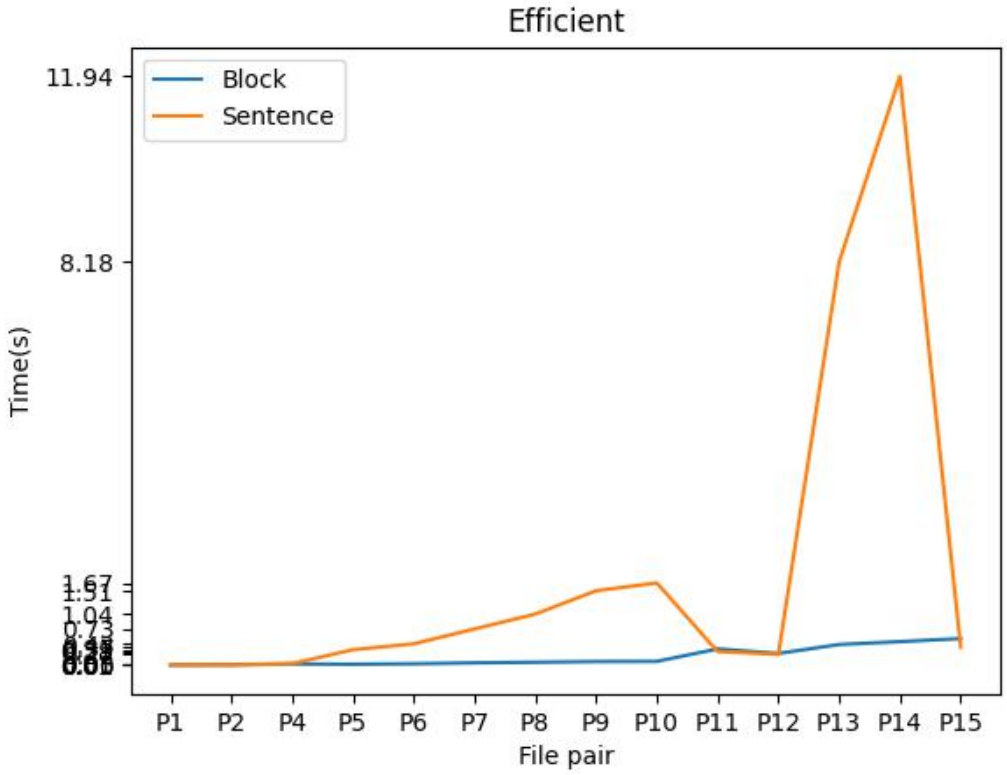
Thuật toán dừng khi duyệt qua hết các phần tử trong vecto BlockX;

Đánh giá mức độ hiệu quả của thuật toán:

Thuật toán sử dụng việc biến đổi tọa độ của mảng hai chiều sang tọa độ mảng một chiều, làm cho giá trị key là duy nhất đối với hai cặp số x,y khác nhau bất kỳ.

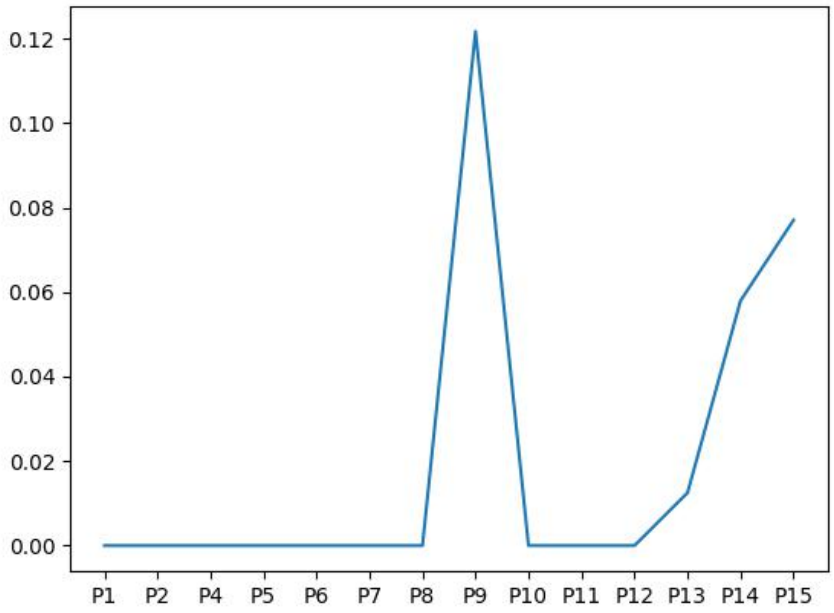
Simulate đánh giá mức độ hiệu quả thuật toán so với khi đầu vào thuật toán Levenshtein cuối là câu

Tiến hành đánh giá trên 15 cặp file 1-15, mỗi cặp file thực hiện 10 lần, t thu được kết quả như sơ đồ sau:



Thời gian thực hiện thuật toán so sánh trung bình

Với việc thay đổi cấu trúc dữ liệu đầu vào thành các khối Block, thì kết quả vẫn tương đương so với việc dùng cấu trúc câu và thời gian thực thi giảm xuống nhiều lần.



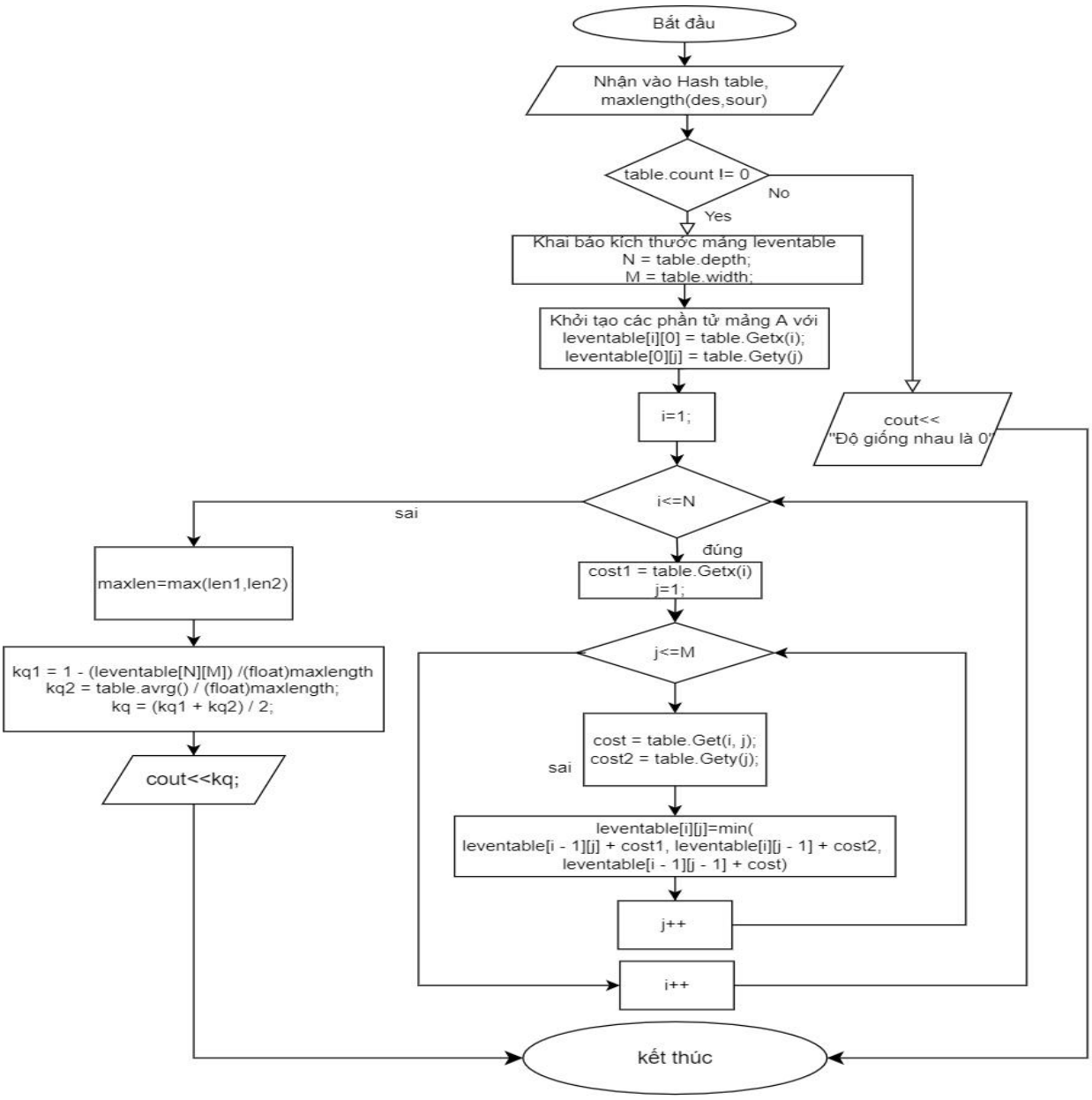
Mức chênh lệch về kết quả đầu ra

Mức chênh lệch về kết quả của hai kiểu dữ liệu đầu vàoỞ đây, kết quả đầu ra của thuật toán có sự chênh lệch nhẹ vì trong quá trình tính toán Levenshtein theo Block trong một số trường hợp, nhất là trường hợp thay thế Block trong quá trình tính toán, có thể gây ra một số chênh lệch nhưng không đáng kể.

2.2.2.2.5. Giải thuật "BigLevenshtein" cho cấp độ hai văn bản.

Việc phân định loại cấu trúc dữ liệu đầu vào là các Block đã giảm đáng kể độ phức tạp của thuật toán Levenshtein, tuy nhiên vì các Block có kích thước khác nhau, nên cần điều chỉnh các bước qui hoạch động của Levenshtein để tạo nên sự tương đương về kết quả so sánh cuối cùng.

Giải thuật được thực hiện theo lưu đồ sau:



Trong đó, các thao tác

Get(x,y): lấy giá trị độ khác nhau giữa hai Block (tương tự với việc xác định giá trị biến cost trong thuật toán Levenshtein gốc so sánh hai chuỗi).

Getx(x)/ Gety(y) : trả về kích thước (count) của Block, tương đương với việc thực hiện count bước biến đổi so với Levenshtein theo cấu trúc dữ liệu câu.

Phân tích thuật toán:

Input: Cấu trúc Hash chứa bảng giá trị đã xử lý và số câu lớn hơn của hai văn bản.

Output: Kết quả so sánh độ giống nhau của hai văn bản.

Đánh giá hiệu quả của thuật toán:

Thuật toán cho ra kết quả tương đương với các công cụ so sánh hiện nay.

Khảo sát trên 3 công cụ so sánh online hiện nay:

Cặp file	Plagiarim	countwordfree	60tool	Project
P1	1%	0%	0%	0%
P2	22%	1.30%	21.15%	14.2%
P3	0%	0.00%	0%	0%
P4	0%	0.00%	0%	0%
P5	68%	67.41%	67.35%	75.33%
P6	75%	74.89%	74.82%	84.26%
P7	68%	68.12%	81.91%	75.42%
P8	100%	85.78%	100.00%	86.44%
P9	100%	70.91%	79.86%	71.33%
P10	100%	89.86%	100.00%	87.94%
P11	0%	0.00%	0.00%	0%
P12	0%	0.00%	0.00%	0%
P13	0%	0.00%	0.01%	0.5%
P14	0%	0.1%	0.08%	0%
P15	0%	0.3%	0.02%	0%

2.3. Cài đặt và kiểm thử

Quá trình chạy chương trình và kiểm thử các trường hợp cho thấy:

Về kết quả thực hiện: Cho kết quả chính xác 70-80% so với các công cụ so sánh hiện có.

Về thời gian thực hiện: Tương đối thấp, ở mức 0.1-5s tùy vào dung lượng của văn bản.

Một số kết quả minh họa:

Văn bản 1	Văn bản 2	Kết quả so sánh
The National Steering Committee for Natural Disaster Prevention and Control directed localities and ministries to prepare for the typhoon Rai **amid the COVID-19 pandemic in its No 26/CĐ-QG urgent message issued on Thursday.** **The Committee for Search and Rescue in provinces and** cities are required to closely monitor the typhoon's developments in order to instruct vessels to find safe shelter and take measures to ensure the safety of people. .	The National Steering Committee for Natural Disaster Prevention and Control directed localities and ministries to prepare for the typhoon Rai. Cities are required to closely monitor the typhoon's developments in order to instruct vessels to find safe shelter	**57.40%**

KẾT LUẬN

1. KẾ HOẠCH THỰC HIỆN

Bảng 3.1 Kế hoạch thực hiện đồ án

STT	Thời gian	Công việc
1	Từ 23/10/2021 đến 30/10/2021	Tìm hiểu, phân tích yêu cầu đề tài và nguồn code tham khảo
2	Từ 31/10/2021 đến 14/11/2021	Tiếp tục tìm hiểu và bắt đầu viết code cho chương trình
3	Từ 15/11/2021 đến 01/12/2021	Viết code và tinh chỉnh kết quả chương trình
4	Từ 2/12/2021 đến 10/12/2021	Hoàn thanh và viết báo cáo
5	Từ ngày 11/12/2021 tới 18/12/2021	Hoàn thành báo cáo

2. KẾT QUẢ ĐẠT ĐƯỢC

- Hiểu được những kiến thức nền tảng về cấu trúc dữ liệu và giải thuật
- Áp dụng được các kiến thức đã tìm hiểu cũng như kiến thức nền từ các môn đã được học trong trường.
- Tìm hiểu được cách xây dựng một chương trình có thể so sánh được độ giống nhau của 2 văn bản.

3. ƯU ĐIỂM VÀ HẠN CHẾ

3.1. Ưu điểm

- Xây dựng được một chương trình đơn giản dễ sử dụng trong việc so sanh 2 văn bản

3.2. Hạn chế

- Nhóm chưa tinh chỉnh được độ chính xác của thuật toán
- Dù nhóm đặt ra nhiều lỗi có thể xảy ra và tìm cách khắc phục, nhưng không thể tránh khỏi thiếu sót, hạn chế chưa được tìm ra

4. KHÓ KHĂN GẶP PHẢI

4.1. Công nghệ

Bảng 3.2 Khó khăn về công nghệ

STT	Khó khăn	Cách khắc phục
1	Có một số thư viện, hàm, phương thức chưa được học được sử dụng trong source code	Sử dụng mạng Internet, các trang diễn đàn về công nghệ để tìm hiểu
2	Quá trình chỉnh sửa, thêm, xóa source thường xảy ra lỗi giữa các các hàm .	Đọc và nghiên cứu kĩ các hàm, phương thức của chúng.

4.2. Quá trình thực hiện

Bảng 3.2 Khó khăn trong qui trình thực hiện đồ án

STT	Khó khăn	Cách khắc phục
1	Do dịch bệnh nên không thể gặp mặt trực tiếp và bàn bạc cụ thể về đồ án	Sử dụng các trang mạng xã hội như Facebook, Zalo,...Các nền tảng họp trực tuyến như Google meet
2	Lịch trình thực hiện đồ án đôi khi bị trễ do lý do khách quan	Tập trung và dành nhiều thời gian hơn cho việc hoàn thành đồ án

5. KINH NGHIỆM ĐẠT ĐƯỢC

Sau khi hoàn thiện đồ án, các thành viên trong nhóm rút ra nhiều bài học để thực hiện các dự án sau tốt hơn:

- Chú ý tới việc quản lý thời gian khi làm đồ án, thường xuyên liên lạc, nhắc nhở nhau hoàn thiện công việc

- Lưu trữ, sao chép code ở nhiều nơi để khi xảy ra sự cố không cần phải xây dựng lại từ đầu.

5. HƯỚNG CẢI TIẾN

Thuật toán Levenshtein có thể được thực hiện với mức độ sử dụng không gian ít hơn nếu thay mảng hai chiều thành hai mảng một chiều.

Trong thời gian tới, nhóm sẽ cải tiến thuật toán theo hướng trên để giảm đi độ phức tạp về không gian.

DANH MỤC TÀI LIỆU THAM KHẢO

1.tek4.vn.https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat?fbclid=IwAR1sz3M5j4mjWC9AIjnK9YiguYlp5tpp0qlrmxCtAGWqnyQaHuVh5rjRZjA

2.Wikipedia https://vi.m.wikipedia.org/wiki/M%E1%BA%A3ng_(c%E1%BA%A5u_tr%C3%BAc_d%E1%BB%AF_li%E1%BB%87u)?fbclid=IwAR18J_y-w5aGlGuqbVCAHMZ56fLlsWjATfHmvvACvyDQm-cqPdzggSDnC14

3. Ứng dụng khoảng cách Levenshtein để so sánh mức độ giống nhau của hai văn bản_Trần Ngọc Chiến . https://text.123docz.net/document/4836509-ung-dung-khoang-cach-levenstein-de-so-sanh-muc-do-giong-nhau-cua-hai-van-ban.htm

LINK GITHUB ĐỀ TÀI

Link project

https://github.com/phungthithuytrang/CTDL-CUOIKI-DETAI6-NHOM2?fbclid=IwAR1_YVATsXI0GmGtuULehz5ilzwm27A5zuPUOKiShS_ERU5fCfDSijIg-cQ

Link chạy simulate

https://l.facebook.com/l.php?u=https%3A%2F%2Fdrive.google.com%2Ffile%2Fd%2F1YRI6R1oLYn0xWC8xraOLIYeb58NV7W8l%2Fview%3Fusp%3Dsharing%26fbclid%3DIwAR28UoVWpVOy4dzXm4LFSibyzB1Fild1fFi9CCB3yeWPltM4n0PfF5shGRU&h=AT0bF84U4IFAffTAN3nS55ne3ig2CyPBK10O2BP0dqylRsG9fhQVtd6nfCFPnTiNT9aGX9iLuFOVncKDLRYO8tPHjk93jqkecYff9WOKHkuARSpz1SyBiFBNA29-Gh0Smg1ZJg

Link update project 7/1/2022.

Project update về cơ bản về thuật toán và cấu trúc dữ liệu đều giống với file trước. Điểm khác biệt ở đây là nhóm đã cải thiện thành công thuật toán Levenstein theo phần hướng phát triển đã nêu ở trên.

Dưới đây là link Github:

https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Fphungthithuytrang%2FFinal-CTDL-CUOIKI.git%3Ffbclid%3DIwAR0Z5PiYpBka3THQLk57MogwdyTp7xrCQLksnS2TX_x1x0Bf8d4b5J__1Jg&h=AT0bF84U4IFAffTAN3nS55ne3ig2CyPBK10O2BP0dqylRsG9fhQVtd6nfCFPnTiNT9aGX9iLuFOVncKDLRYO8tPHjk93jqkecYff9WOKHkuARSpz1SyBiFBNA29-Gh0Smg1ZJg