NGUYỄN THÀNH NAM

HỆ THỐNG THÔNG TIN

2020 – 2022

HÀ NỘI – NĂM 2022

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

NGUYỄN THÀNH NAM

NGHIÊN CỨU PHƯƠNG PHÁP XÁC ĐỊNH MỨC ĐỘ TƯƠNG TỰ GIỮA CÁC MÃ NGUỒN DỰA VÀO CÂY CÚ PHÁP

LUẬN VĂN THẠC SĨ KỸ THUẬT

(*Theo định hướng ứng dụng*)

HÀ NỘI - NĂM 2022

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

NGUYỄN THÀNH NAM

NGHIÊN CỨU PHƯƠNG PHÁP XÁC ĐỊNH MỨC ĐỘ TƯƠNG TỰ GIỮA CÁC MÃ NGUỒN DỰA VÀO CÂY CÚ PHÁP

Chuyên ngành: HỆ THỐNG THÔNG TIN
Mã số: 8.48.01.04

LUẬN VĂN THẠC SĨ KỸ THUẬT
(*Theo định hướng ứng dụng*)

NGƯỜI HƯỚNG DẪN KHOA HỌC : TS. NGUYỄN DUY PHƯƠNG

HÀ NỘI - NĂM 2022

LỜI CAM ĐOAN

Tôi xin cam đoan luận văn đề tài "***Nghiên cứu phương pháp xác định mức độ tương tự giữa các mã nguồn dựa vào cây cú pháp***" là công trình nghiên cứu của cá nhân tôi.

Các kết quả tìm hiểu nêu trong Luận văn này là trung thực và không phải là sao chép toàn văn của bất kỳ công trình nào khác.

Hà Nội, ngày tháng năm 2022

Tác giả luận văn

Nguyễn Thành Nam

LỜI CẢM ƠN

Trong suốt thời gian học tập, nghiên cứu và hoàn thành luận văn, tôi đã nhận được rất nhiều sự giúp đỡ, động viên từ các thầy cô, gia đình và bạn bè tại Học viện Công nghệ Bưu chính Viễn Thông.

Lời đầu tiên, tôi xin bày tỏ sự cảm ơn đặc biệt tới TS Nguyễn Duy Phương, người thầy đã định hướng cho tôi trong việc lựa chọn đề tài, cũng như luôn đưa ra những nhận xét quý báu và trực tiếp hướng dẫn tôi trong suốt quá trình nghiên cứu và hoàn thành luận văn tốt nghiệp.

Tiếp theo, tôi xin gửi lời cảm ơn chân thành tới tất cả các thầy, các cô của Học viện Công nghệ Bưu chính Viễn thông đã giảng dạy và dìu dắt tôi trong trong thời gian tôi học tập, nghiên cứu tại Học viện.

Tôi cũng xin gửi lời cảm ơn tới gia đình và bạn bè, những người đã luôn ở bên cạnh động viên, ủng hộ, tạo điều kiện cho tôi hoàn thành luận văn này

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CÁC CHỮ VIẾT TẮT

Viết tắt	Tiếng Anh	Tiếng Việt
AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
LCS	Longest common subsequence	Tìm độ dài dãy con chung dài nhất
LLVM	Low Level Virtual Machine	Máy ảo cấp thấp
TF-IDF	Term Frequency – Inverse Document Frequency	Tần suất từ / Nghịch đảo tần suất tài liệu
ROC	Receiver Operating Characteristic	Đường cong đặc trưng hoạt động của bộ thu nhận
AST-CC	AST Code Comparison	So sánh cây AST mã nguồn

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH

MỞ ĐẦU

1. Lý do chọn đề tài

Trong những thập kỷ vừa qua, nhận loại đã chứng kiến sự phát triển vượt bậc của công nghệ thông tin. Sự phát triển của công nghệ thông tin cùng những công nghệ đột phá, đã đánh dấu con người bước vào một giai đoạn phát triển mới, giúp cho cuộc sống của con người ngày một hiện đại hơn, thông minh hơn. Tuy nhiên bên cạnh những tiện ích mà công nghệ mang lại, cũng tồn tại những ảnh hưởng tiêu cực đến cuộc sống chúng ta.

Công nghệ thông tin cùng với Internet ngày càng phát triển và phổ biến tạo điều kiện cho việc chia sẻ kiến thức được dễ dàng hơn. Theo thời gian, lượng tài liệu được tạo ra, lưu trữ và chia sẻ trên Internet ngày càng nhiều hơn. Tuy có nhiều thuận lợi cho việc tiếp cận kho tri thức mở, nhưng rõ ràng cũng đã nảy sinh nhiều vấn đề: các ý tưởng, các đoạn văn thậm chí toàn văn bản được sao chép một cách tràn lan mà không có chú thích nguồn hay tác giả. Điều này không chỉ ảnh hưởng đến quyền sở hữu trí tuệ, bản quyền của tác giả mà còn ảnh hưởng đến việc đánh giá chất lượng của các tài liệu báo cáo, tiểu luận, luận văn và trình độ của người tạo ra chúng, đặc biệt là trong môi trường giáo dục nói chung và đại học nói riêng.

Bản thân ngành công nghệ thông tin cũng đang chịu tác động bởi việc sao chép này. Việc sao chép mã nguồn là một vấn đề đang được lưu tâm tới bởi tính hai mặt của nó. Trong trường hợp mã nguồn được sử dụng lại nhằm mục đích tận dụng, kế thừa và giảm thiểu thời gian phát triển viết mới, kiểm thử, thì có thể coi phần nào đó là hữu dụng. Tuy nhiên trong nếu bị lạm dụng, việc sử dụng lại tràn lan và không kiểm soát các đoạn mã nguồn có thể dẫn tới việc khó khăn cho giai đoạn bảo trì và phát triển mới sau này.

Ngoài ra trong lĩnh vực giáo dục đào tạo, việc sao chép mã nguồn cũng có những ảnh hưởng nhất định tới chất lượng đào tạo nhân lực Công nghệ thông tin. Vấn đề này phổ biến đối với nhiều trường đại học trên thế giới, và các trường đại học tại Việt Nam cũng không phải là ngoại lệ. Các sinh viên sẽ sao chép mã nguồn của sinh viên khác sau đó sửa chữa đi một vài điểm để chuyển thành sản phẩm của mình.

So với việc sao chép các bài luận, các đề tài, nghiên cứu khoa học mà hình thức biểu hiện ở dưới dạng ngôn ngữ văn bản thông thường, việc sao chép mã nguồn lập trình có nhiều điểm khác biệt và đặc thù. Để giải quyết thực trạng này, trên thế giới đã có nhiều phần mềm

được phát triển để đánh giá sự tương tự giữa các bộ mã nguồn. Các phần mềm như vậy thực hiện nhiệm vụ khó khăn là giảm thiểu số lượng âm tính giả và dương tính giả cùng một lúc, tức là tìm ra số lượng bản sao cao nhất trong khi tránh những bản sao không đúng. Việc nghiên cứu các phương pháp để tối ưu hơn việc phát hiện việc sao chép mã nguồn là một bài toán thực sự thú vị và có nhiều ý nghĩa cũng như ứng dụng trong thực tiễn.

Như vậy có thể thấy, thực trạng sao chép mã nguồn xuất hiện trong cả ngành công nghiệp phát triển phần mềm cũng như trong công tác đào tạo sinh viên ngành công nghệ thông tin. Phạm vi nghiên cứu của đề tài này sẽ tập trung sâu hơn vào vấn đề sao chép mã nguồn trong công tác đào tạo sinh viên ngành công nghệ thông tin tại giảng đường.

1 CHƯƠNG 1: TỔNG QUAN VỀ BÀI TOÁN ĐÁNH GIÁ MỨC ĐỘ TƯƠNG TỰ GIỮA CÁC MÃ NGUỒN

1.1. Tổng quan về sao chép và sử dụng lại mã nguồn

1.1.1 Vấn đề sao chép và sử dụng lại mã nguồn

Cuộc cách mạng công nghiệp lần thứ tư được nhắc đến trong những năm gầy đây được trợ giúp rất lớn từ phát triển vượt bậc của lĩnh vực Công nghệ kỹ thuật số, Công nghệ thông tin. Trong đó, ngành công nghiệp phát triển mềm cũng đóng góp không nhỏ với việc liên tục nghiên cứu ra những công nghệ, ngôn ngữ lập trình, thư viện hay framework mới.

Lập trình viên trong thời đại ngày nay được tiếp cận với một lượng tài nguyên khổng lồ và gần như không giới hạn. Nếu như trước kia, các tài liệu của các ngôn ngữ lập trình hay sự hỗ trợ từ cộng đồng là hạn chế, thì ngày nay, những cộng đồng lập trình trên toàn thế giới (stackoverflow, github…) cho phép lập trình viên nhanh chóng học hỏi và tận dụng lại tri thức từ những người đi trước. Đôi khi một bài toán trước đây cần nhiều thời gian suy nghĩ, thì ngày nay có thể đã được lập trình viên đi trước chia sẻ công khai, chỉ cần sao chép về và đưa vào dự án của mình.

Tuy nhiên, việc này cũng có thể gây ra những hậu quả xấu trong giai đoạn bảo trì và nâng cấp phần mềm. Chẳng hạn như một lỗi được phát hiện trong một đoạn mã nguồn đã bị sao chép, tất cả các đoạn tương tự như nó sẽ phải được kiểm tra và sửa cho cùng một lỗi như vậy. Các đoạn mã nguồn trùng lặp sẽ làm tăng đáng kể công việc cần thực hiện khi cần bảo trì hoặc chỉnh sửa – nâng cấp mã nguồn. Điều này sinh ra nhiều nghiệp vụ trong phát triển phần mềm chẳng hạn như hiểu và chuyển giao chương trình, phân tích chất lượng mã nguồn (ít lặp hơn có nghĩa là mã nguồn chất lượng tốt hơn), kiểm tra tính duy nhất của bộ mã nguồn khi đăng ký bản quyền, phân tích cải tiến phần mềm, phát hiện các đoạn mã nguồn không tốt. Do vậy, việc phát hiện mức độ tương tự giữa các mã nguồn là một phần quan trọng và có giá trị lớn trong phân tích phần mềm.

Các sinh viên ngành Công nghệ thông tin trong thời đại mới cũng được thừa hưởng một môi trường sinh động và đa dạng hơn nhiều so với các thế hệ sinh viên trước. Giờ đây việc tiếp cận các ngôn ngữ lập trình mới, các thư viện/framework có độ trưởng thành cao được cộng

đồng chia sẻ, giúp cho sinh viên dù chưa có nhiều kinh nghiệm cũng có thể dễ dàng xây dựng nên một phần mềm tương đối hoàn chỉnh. Việc này rút ngắn thời gian tạo ra sản phẩm, tuy nhiên điều này sẽ làm giảm khả năng sáng tạo, đào sâu tìm tòi giải pháp của sinh viên. Trong khi đây là yếu tố rất quan trọng trong giai đoạn sinh viên công nghệ thông tin được đào tạo tại trường.

1.1.2 Những tác động của sao chép và sử dụng lại mã nguồn

Mặc dù việc sao chép và kế thừa lại mã nguồn thường có chủ đích và có thể mang lại giá trị ở một số mặt như: tiết kiệm thời gian, nguồn lực phát triển lại, giảm thiểu lỗi do các mã nguồn được sao chép đã chạy tốt và được kiểm chứng.

Bên cạnh những ưu điểm trình bày ở trên, việc sao chép, sử dụng lại các mã nguồn cũng có những nhược điểm nhất định cần quan tâm.

Đối với công tác đào tạo:

- Giảm khả năng sáng tạo của sinh viên: từ đó giảm chất lượng của công tác đào tạo.
- Kết quả đào tạo không khách quan: từ đó giáo viên khó đánh giá chất lượng kết quả đào tạo để đưa ra phương hướng, điều chỉnh cho nội dung giảng dạy của mình.

Đối với quy trình phát triển phần mềm:

- Tăng khả năng lỗi tiềm ẩn trong những đoạn mã nguồn giống nhau, từ đó gây khó khăn cho công tác bảo trì, phát triển mở rộng.
- Gây khó khăn cho công tác tối ưu mã nguồn, các đoạn mã nguồn tương tự nhau lặp đi lặp lại trong ứng dụng.

1.1.3 Ý nghĩa của việc đánh giá mức độ tương tự giữa các mã nguồn

Đối với công tác đào tạo:

- **Phát hiện sao chép:** Hỗ trợ cho giảng viên trong việc đưa ra mức độ tương tự nhau giữa các cặp mã nguồn, từ đó giảng viên có đánh giá và nhắc nhở sinh viên chủ động thực hiện bài tập của mình.

- **Phân tích và đánh giá:** Giúp giáo viên phân tích tính trùng lặp giữa các phương án của sinh viên khi triển khai thuật toán (trong trường hợp sinh viên vô tình có cùng ý tưởng).

Đối với quy trình phát triển phần mềm:

- **Phân tích và cải tiến hiệu năng của những đoạn mã nguồn tương tự:** Từ những đoạn mã nguồn tương tự, lập trình viên có thể tổng quát hóa để chuyển thành các thư viện dùng chung, việc này vừa giảm tải dung lượng mã nguồn, đồng thời đảm bảo tính tinh gọn và dễ bảo trì.
- **Tìm ra những đoạn mã nguồn không tốt:** Trong quá trình phát triển dự án, đôi khi lập trình viên tìm kiếm và sử dụng những đoạn mã nguồn được cộng đồng chia sẻ. Đôi khi những đoạn mã này tiềm ẩn nhiều rủi ro và cần tìm kiếm để thay thế. Việc sao chép chính xác được vị trí những đoạn mã nguồn tương tự nhau này là rất cần thiết.
- **Phát hiện sao chép mã nguồn mà không được phép:** Cùng với sự phát triển của Internet, mã nguồn mở, việc chia sẻ và tái sử dụng các đoạn code là vô cùng phổ biến. Điều này dẫn đến có khả năng việc sao chép chưa được sự cho phép của tác giả mã nguồn gốc. Việc xác định mức độ tương tương tự giữa các đoạn mã nguồn, cũng là cơ sở giúp phân định cho các cơ quan quản lý trong việc đảm bảo đăng ký bản quyền mã nguồn.

1.1.4 Giới thiệu các kiểu sao chép mã nguồn phổ biến

Khi một sinh viên sử dụng lại mã nguồn được viết bởi một người khác, sinh viên đó thường có thói quen sẽ điều chỉnh lại một chút nhằm mục đích tránh cho giáo viên có thể nhận biết sự sao chép này. Bằng cách vận dụng sự sáng tạo và kiến thức đã có về ngôn ngữ lập trình, sinh viên có rất nhiều cách khác nhau để tạo ra một phiên bản mới của mã nguồn mà mình vừa sao chép.

Dưới đây là danh sách một số kiểu sao chép mã nguồn phổ biến, được liệt kê theo thứ tự từ đơn giản đến phức tạp:

1. Thay đổi định dạng hoặc thêm/sửa các “comment code” (ghi chú).
2. Thay đổi tên của các hàm hoặc các biến.

3. Thay đổi thứ tự các toán hạng trong các biểu thức.

4. Thay đổi kiểu dữ liệu của các biến.

5. Thay một biểu thức bằng một biểu thức tương đương

6. Thêm các đoạn code dư thừa (deadcode)

7. Thay đổi thứ tự của các đoạn mã nguồn độc lập.

8. Thay đổi một vòng lặp bởi một vòng lặp khác

9. Thay đổi cấu trúc của câu lệnh lựa chọn

10. Thay thế việc gọi hàm mới bằng nội dung hàm

11. Kết hợp mã nguồn sao chép với mã nguồn của bản thân.

Dưới góc nhìn đào tạo, việc tìm hiểu và tham khảo mã nguồn của người khác là rất bình thường và không đáng phê phán (trong trường hợp bài tập này được giảng viên cho phép). Đây cũng là một cách để sinh viên học hỏi và ghi nhớ kiến thức một lần nữa.

Sau khi hiểu thuật toán, nếu sinh viên cố gắng tự viết lại giải thuật bằng ngôn ngữ của mình cũng là một lần học tập. Ví dụ cho việc này chính là kiểu sao chép số 11 được liệt kê trên đây. Trong khi đó các kiểu sao chép từ 1 đến số 10, phần nào đó thể hiện khía cạnh sinh viên cố tình sử dụng thủ thuật sau khi đã sao chép mã để tránh giáo viên phát hiện.

1.2. Đánh giá chương trình xác định mức độ tương đồng giữa các mã nguồn

Để xác định tính hiệu quả của một chương trình đánh giá mức độ tương đồng giữa các mã nguồn, người ta thường sử dụng khái niệm ROC (Receiver operating characteristic), còn gọi là đường cong đặc trưng hoạt động. Khái niệm đường cong ROC bắt nguồn từ một phần của lĩnh vực được gọi là thuyết phát hiện tín hiệu. Từ các tín hiệu nhận được, máy sẽ phân tích và vẽ đường cong ROC, để phân biệt tín hiệu của máy bay địch và tín hiệu nhiễu (noise) trong thế chiến thứ hai. Từ sau những năm 1970, ROC được dùng phổ biến để diễn dịch kết quả các test trong chẩn đoán y học [1].

Với đặc tính là một đồ thị được tạo ra bằng cách biểu diễn tỷ lệ dự báo Dương tính giả (true positive rate - TPR) và tỷ lệ dự báo Âm tính giả (false positive rate - FPR) tại các ngưỡng (threshold) khác nhau, ROC gợi ý cách so sánh hiệu suất của hai chương trình đánh giá độ tương đồng mã nguồn [2].

Trong phần trình bày tiếp theo dưới đây, chúng ta sẽ giới thiệu về khái niệm Ma trận nhầm lẫn và ứng dụng của đường cong ROC trong bài toán phân tích mức độ tương tự giữa các mã nguồn.

1.2.1 Khái niệm ma trận nhầm lẫn (Confusion Matrix)

Confusion Matrix hay còn gọi là Ma trận nhầm lẫn là bố cục bảng cụ thể cho phép hình dung hiệu suất của một hệ thống phân loại. Mỗi hàng trong ma trận sẽ biểu diễn thông tin cho một trường hợp của một lớp cụ thể. Với một ma trận nhầm lẫn, có thể dễ dàng nhận ra liệu hệ thống phân loại có gắn nhãn nhầm giữa lớp này với lớp khác hay không [3].

Hãy xem xét ví dụ thực tế dưới đây:

Vào một ngày nào đó, Bệnh viện A có 100 bệnh nhân đến khám bệnh, giả sử thực tế là trong 100 bệnh nhân có 60 người mắc bệnh, 40 người không có bệnh. Sau khi thăm khám, bệnh viện đưa ra kết quả:

- **Trong 60 người có bệnh thật gồm:**
  - 50 người chẩn đoán có bệnh
  - 10 người chẩn đoán không mắc bệnh.
- **Trong 40 người không có bệnh gồm:**
  - 25 người chẩn đoán không mắc bệnh
  - 15 người chẩn đoán là mắc bệnh.

Bảng 1.1: Kết quả ma trận nhầm lẫn sau khi chẩn đoán

	Dương tính (P)	**Âm tính (N)**
Dương tính	**TP (50)**	FP (15)
Âm tính	FN (10)	**TN (25)**

Từ ma trận cơ bản này, ta sẽ có một số thuật ngữ sau:

- Condition positive (P): Tổng số ca dương tính thực tế.
- Condition Negative (N: Tổng số ca âm tính thực tế.
- True positive (TP): Số các ca dự đoán dương tính đúng hay dương tính thật.
- True negative (TN): Số các ca dự đoán âm tính đúng hay âm tính thật.

- False positive (FP): Số các ca dự đoán dương tính sai hay dương tính giả.
- False negative (FN):: Số các ca dự đoán âm tính sai hay âm tính giả.

Để đánh giá kết quả khám bệnh của của bệnh viện trên, chúng ta có các chỉ số đánh giá quan trọng sau:

- Độ chính xác ACC (Accuracy)

$$ACC = \frac{\text{Số dự đoán đúng}}{\text{Tổng số mẫu}} = \frac{TP + TN}{P + N} = \frac{50 + 25}{100} = 0{,}75$$

- Độ nhạy TPR (True Positive Rate - Tỷ lệ dương tính thực)

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} = \frac{50}{50 + 10} = 0{,}833$$

- FPR (False Positive Rate - Tỷ lệ dương tính giả)

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} = \frac{15}{40} = 0{,}375$$

- Độ đặc hiệu (Độ đặc trưng)

$$\text{Độ đặc hiệu} = 1 - \text{FPR} = 1 - 0{,}375 = 0{,}625$$

1.2.2 Biểu diễn đường cong ROC

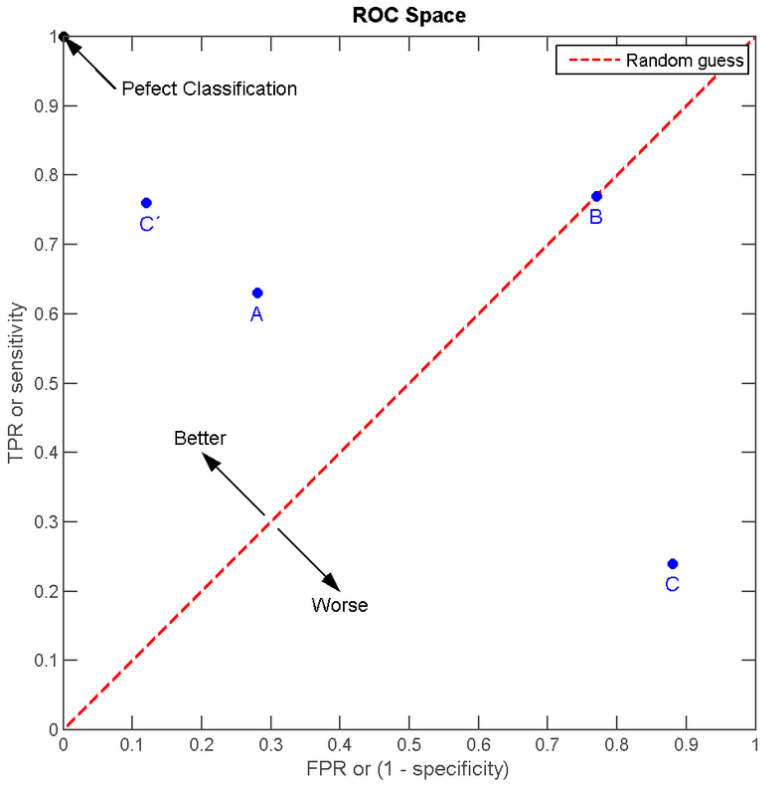
Khi biểu diễn 2 chỉ số TPR và FPR trên đồ thị với trục tung là TPR và trục hoành là FPR, chúng ta có khái niệm không gian ROC.

Xét ví dụ sau, biểu diễn 4 điểm A, B, C, D tương ứng cho 4 phép đánh giá có kết quả lần lượt như bảng dưới đây:

Bảng 1.2: So sánh kết quả các phép đánh giá khi chẩn đoán

	A	**B**	**C**	**D**
TP	63	77	24	76
FP	28	77	88	12
FN	37	23	76	24
TN	72	23	12	88
TPR	**0,63**	**0,77**	**0,24**	**0,76**
FPR	**0,28**	**0,77**	**0,88**	**0,12**
ACC	**0,68**	**0,50**	**0,18**	**0,82**

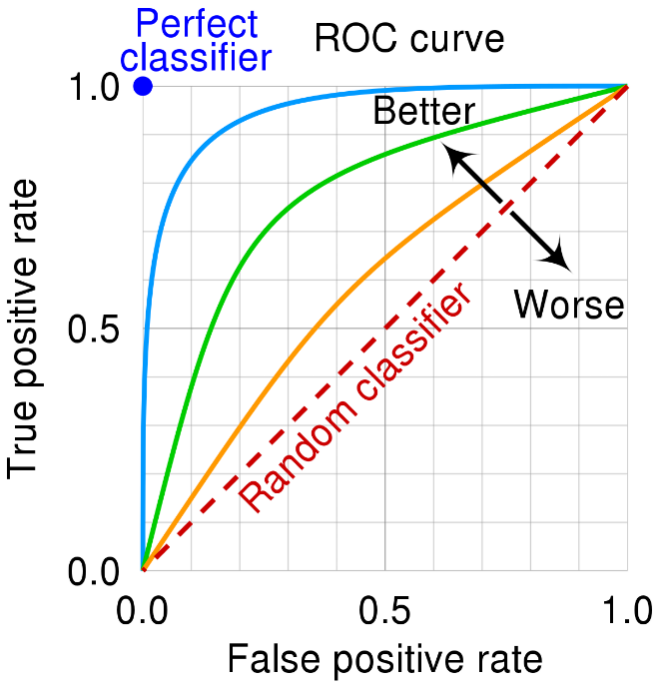
Theo đó kết quả thu được như sau:



Hình 1.1: Không gian ROC

Đường chéo nét đứt từ tọa độ (0,0) tới (1,1) đại diện cho phép dự đoán ngẫu nhiên, chia không gian ROC thành 2 phần theo 2 chiều hướng tốt hơn và xấu hơn. Tức là những phép dự đoán cho kết quả nằm phía trên đường chéo được đánh giá là có kết quả tốt hơn so với các phép dự đoán cho kết quả nằm dưới đường chéo, phép đo có kết quả càng gần với đỉnh trên bên trái (0,1) thì càng tiệm cận tới phép đo "hoàn hảo".

Tập hợp kết quả các lần dự đoán của một hệ thống dự đoán sẽ tạo thành một đường cong trong không gian ROC, thường được gọi là "đường cong ROC".



Hình 1.2: Đường cong ROC

Một số tính chất của đường cong ROC:

- Đường cong càng đi dọc theo biên trái và rồi đi dọc theo biên phía trên của không gian ROC, thì chứng tỏ kết quả kiểm tra càng chính xác.
- Đường cong càng tiến tới thành đường chéo 45 độ trong không gian ROC, thì độ chính xác của kiểm tra càng kém.
- Hệ số góc của đường thẳng tiếp tuyến tại một điểm cutpoint cho ta tỉ lệ likelihood ratio (LR) của giá trị cutpoint đó của bài kiểm tra.
- Diện tích phía dưới đường cong, giới hạn trong không gian ROC, là thước đo cho độ chính xác của bài kiểm tra, chẳng hạn: 1 là tối ưu, 0.5 là kém. Phần diện tích này có ý nghĩa là thước đo cho khả năng phân biệt (discrimination) tốt hay không.

Dựa vào tính chất trên, chúng ta có thể so sánh kết quả của những chương trình đánh giá mức độ tương tự bằng cách vẽ đường cong ROC tương ứng với mỗi chương trình, chương trình nào cho đường cong ROC nằm trên thì được đánh giá có khả năng dự đoán chính xác hơn.

1.3. Cây cú pháp trừu tượng (AST, abstract syntax tree)

1.3.1 Tổng quan về các công nghệ so sánh mã nguồn phổ biến

Trong giai đoạn đầu nghiên cứu về vấn đề sao chép, công nghệ chủ yếu để phát hiện ra một bản sao là dựa trên nội dung của văn bản. Phương pháp này rất đơn giản là xác định xem hai tệp tương tự bằng cách so sánh các giá trị được tính toán của các tệp. Nhưng giải pháp này chỉ có thể phát hiện khi sao chép mà không có sửa đổi. Với sự phát triển của công nghệ, nhiều nghiên cứu trong lĩnh vực này đã đề xuất rất nhiều thuật toán và phương án khác nhau. Đến nay có nhiều phần mềm, công cụ giúp phân tích và phát hiện trùng lặp trong mã nguồn, tuy nhiên về cơ bản đều thuộc một trong 4 trường phái sau:

Dựa trên phân tích văn bản thuần túy (string-based): Mã nguồn được chuyển thành các chuỗi liên tiếp ký tự. Hai mã nguồn được coi là tương tự nhau nếu chúng bao gồm các chuỗi ký tự giống nhau. Thuật toán chính được sử dụng trong công nghệ này là thuật toán tìm chuỗi con dài nhất (LCS). Nhưng rõ ràng thuật toán có những hạn chế, chỉ khi tất cả các chuỗi giống nhau, hai mã nguồn mới có thể được coi là tương tự.

Dựa trên phân tích chuỗi ký tự đại diện (token-based): Công nghệ này được cải tiến từ việc phân tích mã nguồn dựa trên văn bản thuần túy trình bày phía trên. Mã nguồn chương trình lúc này được tách từ thành chuỗi các token, sau đó thực hiện quét và kiểm tra các chuỗi token trùng nhau để xác định khả năng xảy ra lặp, sao chép mã nguồn. So với hướng tiếp cận trên, hướng tiếp cận này mạnh hơn trong việc phát hiện các đoạn mã nguồn lặp cho dù mã nguồn đã được sửa chữa đôi chút. Các công cụ phát hiện sao chép mã nguồn tiêu biểu theo trường phái này gồm: CP-Mine, CCFinder, Jplag. Tuy nhiên các công cụ này đều có nhược điểm không thể phát hiện khi mã nguồn có sự thay đổi như thay đổi tên hàm, biến, thứ tự các đoạn code độc lập, hay chèn thêm các đoạn code vô nghĩa.

Dựa trên cấu trúc cây (tree-based): Đây là công nghệ mới được đề cập và phát triển nhanh trong những năm gần đây. Việc so sánh các mã nguồn dựa trên cấu trúc của nó là hướng đi rất hợp lý, khi mà nó có thể bằng cách này hay cách khác để loại bỏ sự ảnh hưởng bởi những thao tác như: đổi tên hàm, tên biến, thứ tự code…

Do ưu điểm của phương pháp tiếp cận này, nên các thuật toán trong luận văn này tập trung vào việc phân tích mã nguồn dựa trên cấu trúc cây.

Dựa trên biểu đồ phụ thuộc (Program Dependence Graph - PDG): Một số nghiên cứu chỉ ra rằng các thủ thuật sao chép mã nguồn không làm thay đổi nhiều đối với biểu đồ phụ thuộc của mã nguồn gốc. Do đó việc tiếp cận PDG trong phân tích đánh giá mức độ tương tự giữa các mã nguồn là một ý tưởng rất thú vị.

Trong khuôn khổ của luận văn này, tác giả không trình bày về các giải thuật liên quan đến phương pháp tiếp cận này.

1.3.2 Khái niệm về cây cú pháp trừu tượng (AST, abstract syntax tree)

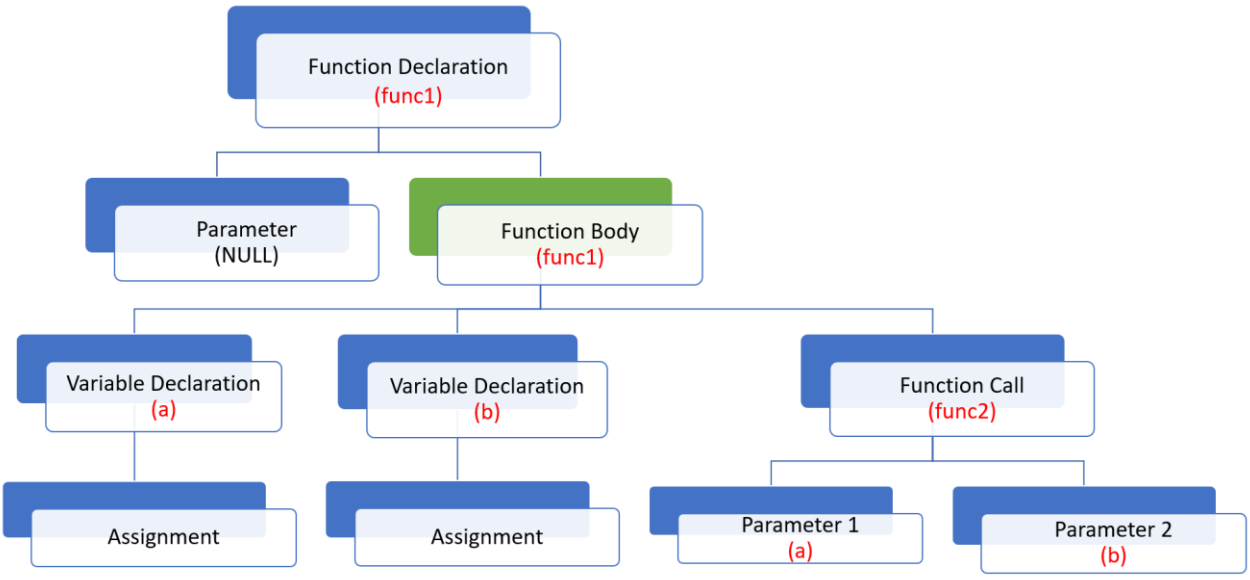
Trong ngành khoa học máy tính, cây cú pháp trừu tượng (AST, abstract syntax tree), là cấu trúc dữ liệu được sử dụng rộng rãi trong trình biên dịch, do thuộc tính đại diện cho cấu trúc của mã chương trình. AST thường là kết quả của giai đoạn phân tích cú pháp của trình biên dịch [11].

AST là cấu trúc cây mà các nút gốc của cây được gán nhãn bằng các toán tử và các nút là của cây là các toán hạng. Trong các công đoạn của chương trình dịch, cây AST này được dùng trong bộ phân tích cú pháp như là một trung gian giữa cây phân tích cú pháp (concrete syntax tree) và cấu trúc dữ liệu. Cây cú pháp trừu tượng khác với cây phân tích cú pháp là ở chỗ nó không chỉ quan tâm đến cú pháp mà còn quan tâm đến ngữ nghĩa của chương trình.

Hãy cùng xem xét ví dụ đoạn mã nguồn sau (ngôn ngữ C++):

```
void fun1(){
     int a = 2;
     int b = 5;
     fun2(a,                    b);
}
```

Về cơ bản đoạn mã nguồn này có cây AST như sau:



Hình 1.3: Cây cú pháp trừu tượng của mã nguồn hàm func

1.3.3 Các phần mềm sinh ra cây cú pháp AST

Mỗi ngôn ngữ lập trình có những công cụ complier khác nhau (ví dụ Java có Java Complier Complier – JavaCC hoặc JJTree, Javascript có JIT, C++ có GCC hoặc Clang). Tương ứng với mỗi công cụ này, sẽ có các công cụ xây dựng cây cú pháp AST khác nhau. Ngoài ra hiện nay trên internet đã có những công cụ trực tuyến cho phép xem trực tiếp cây cú pháp của một đoạn mã nguồn ngay trên trình duyệt web tại địa chỉ sau: https://astexplorer.net/. Công cụ này hỗ trợ nhiều ngôn ngữ lập trình như PHP, Python, Lua, Javascript.

Trong phạm vi luận văn này, do chỉ tập trung vào ngôn ngữ C++ nên chọn lựa trình biên dịch Clang trong bộ khung biên dịch LLVM để phục vụ cho việc xây dựng cây AST từ các mã nguồn viết bằng ngôn ngữ C++. Dưới đây là một số thông tin tổng quan về LLVM và Clang.

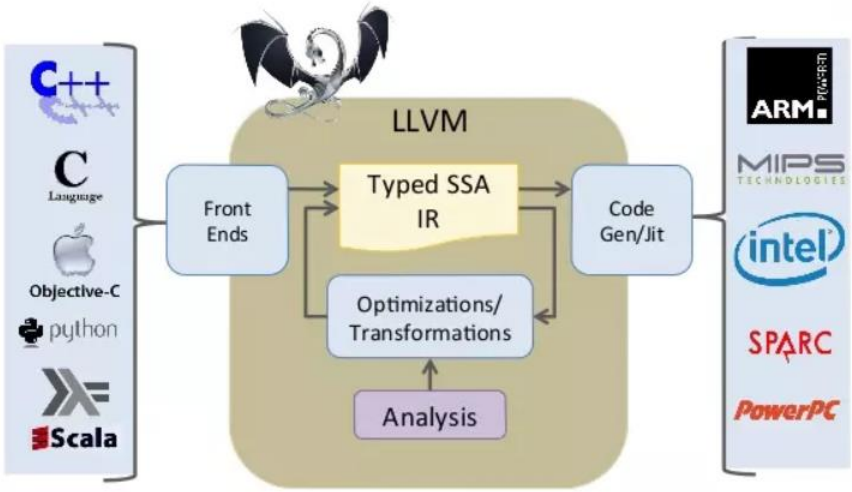
1.3.4 Tổng quan về LLVM (Low-Level Virtual Machine)

LLVM là một bộ khung biên dịch (compiler framework) được sử dụng để tối ưu hóa mã cho nhiều ngôn ngữ lập trình khác nhau. Nó cung cấp những công cụ mạnh mẽ để xây dựng phần front-end (parser, lexer) cũng như phần backend (phần chuyển phần code trung gian LLVM sang mã máy), cho các ngôn ngữ lập trình mới [12].

Xuất phát từ khái niệm Low-Level Virtual Machine - Máy ảo cấp thấp, LLVM thường được xem là một khung để tạo nên các trình biên dịch được thiết kế để hỗ trợ phân tích và chuyển đổi chương trình suốt đời, suốt đời cho các chương trình tùy ý, bằng cách cung cấp

thông tin cấp cao để chuyển đổi trình biên dịch tại thời điểm compile-time, link-time, run-time và idle time giữa các lần chạy. LLVM cung cấp những công cụ mạnh mẽ để xây dựng phần front-end (parser, lexer) cũng như phần backend (phần chuyển phần code trung gian LLVM sang mã máy), cho các ngôn ngữ lập trình mới.

LLVM đã được sử dụng để xây dựng nên nhiều trình biên dịch (compiler) của nhiều ngôn ngữ lập trình cấp cao phổ biến hiện nay, ví dụ như C, C++, Python, Java, Ruby, cũng như Objective-C và Swift.

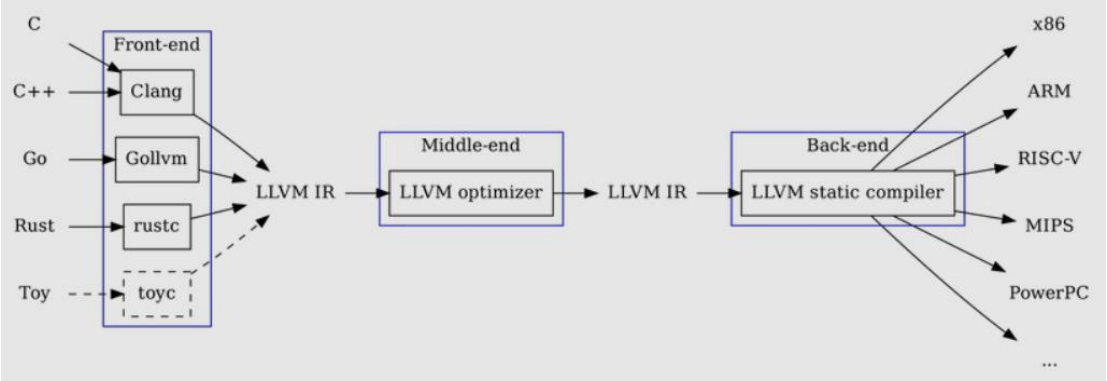


Hình 1.4: Kiến trúc nền tảng biên dịch LLVM

Như được thể hiện trong hình trên, một hệ sinh thái chương trình dịch LLVM gồm 3 phần:

- Front-end: nhận mã nguồn viết trên các ngôn ngữ lập trình và chịu trách nhiệm thực hiện 3 bước gồm Lexical analysis (phân tích từ tố) - đọc từng ký tự thành các token, syntax analysis (phân tích cú pháp) - Parser chuyển các token ở bước trước thành **AST** và semantic analysis (phân tích ngữ nghĩa) - kiểm tra các thông tin khác. Đầu ra của Front-end là mã trung gian.
- Middle-end: được biết đến với cái tên "LLVM Optimizer", sử dụng một ngôn ngữ lập trình cấp thấp gọi là intermediate representation (IR) làm mã trung gian. IR được sử dụng nhằm mục đích chuyển đổi giữa kết quả của phần Front-end sang Back-end để có thể tạo mã máy.

- Back-end: chịu trách nhiệm tạo ra mã máy từ mã trung gian cho từng kiến trúc CPU cụ thể.



Hình 1.5: Luồng xử lý từ frontend → middle-end → back-end

Nhờ thiết kế độc lập của 3 thành phần dạng module, LLVM giúp các nhà phát triển rất dễ hỗ trợ thêm ngôn ngữ front-end mới, cũng như hỗ trợ cho các kiến trúc CPU mới ở phía back-end, ngay cả những kiến trúc không tồn tại ở thời điểm ứng dụng ra đời. Cùng với nhiều công cụ mạnh mẽ khác, dự án LLVM mang đến sức mạnh, tốc độ, an toàn cùng với tiện lợi cho hầu hết các ngôn ngữ hiện đại ngày nay.

1.3.5 Tổng quan về Clang

Clang là một trình biên dịch (compiler front-end) sử dụng cơ sở hạ tầng trình biên dịch LLVM cho các ngôn ngữ lập trình C, C ++, Objective-C và Objective-C ++ [13].

Clang được thiết kế để hoạt động như một sự thay thế cho Bộ trình dịch GNU (GNU Compiler Collection - GCC). Clang được đóng góp và hỗ trợ bởi các tổ chức uy tín như Apple, Microsoft, Google, ARM, Sony, Intel và Advanced Micro Devices (AMD). Đây là phần mềm nguồn mở, với mã nguồn được phát hành theo Giấy phép của Đại học Illinois / NCSA.

Đặc điểm và mục tiêu của Clang:

- Đối với Người dùng cuối:
  - Biên dịch nhanh và sử dụng ít bộ nhớ
  - Chẩn đoán nhanh (ví dụ)
  - GCC tương thích
- Đối với Tiện ích và Ứng dụng:
  - Kiến trúc mô đun hóa

- Hỗ trợ đa dạng “client” (tái cấu trúc, phân tích tĩnh, tạo mã, v.v.)
- Cho phép tích hợp chặt chẽ với các IDE
- Sử dụng Giấy phép LLVM Apache 2

- Thiết kế nội bộ và Triển khai:
  - Trình biên dịch chất lượng áp dụng được áp dụng vào thực tiễn
  - Một cơ sở mã nguồn đơn giản và dễ hiểu, dễ tùy biến
  - Một trình phân tích cú pháp hợp nhất cho các ngôn ngữ C, Objective C, C ++ và Objective C ++
  - Tuân thủ C / C ++ / ObjC và các biến thể của chúng

Hình 1.6: Các pha của trình biên dịch CLang

Các cách sử dụng Clang

- Cài đặt và sử dụng trực tiếp Clang thông qua bộ thư viện mà Clang công khai trên trang chủ tại địa chỉ: https://clang.llvm.org/get_started.html. Lưu ý việc triển khai này phụ thuộc môi trường cài đặt của máy tính.
- Cài đặt và sử dụng gián tiếp thông qua API của bộ thư viện tích hợp LibClang [14] trên các ngôn ngữ lập trình phổ biến (Python). Đây là cách được triển khai trong nội dung luận văn này.

1.3.6 Sử dụng Clang với Python

Python là một ngôn ngữ lập trình phổ biến và được cộng đồng hỗ trợ nhiều trong các năm gần đây. Python là một ngôn ngữ lập trình bậc cao cho các mục đích lập trình đa năng, do Guido van Rossum tạo ra và lần đầu ra mắt vào năm 1991. Python được thiết kế với ưu điểm mạnh là dễ đọc, dễ học và dễ nhớ. Python có hình thức rất sáng sủa, cấu trúc rõ ràng, thuận tiện cho người mới học lập trình và là ngôn ngữ lập trình dễ học; được dùng rộng rãi trong phát triển trí tuệ nhân tạo.

Thông qua các API của thư viện LibClang trên Python, chúng ta có thể sử dụng một cách dễ dàng nhưng đảm bảo tương đối đầy đủ các thao tác đối với cây AST [4].

Hãy cùng xem xét cây AST tạo ra tương ứng với đoạn mã nguồn trong **phần 1.3.2** bởi LibClang, được thể hiện dưới dạng cây như sau:

```
C:\Users\Admin\AppData\Local\Programs\Python\Python37\python.exe D:/CaoHoc/LuanVan/source_code/CompareAST/ast_viewer.py
├──TRANSLATION_UNIT test_data_2/test_6.cpp  [line=0, col=0]
   └──FUNCTION_DECL fun1  [line=1, col=6]
      └──COMPOUND_STMT   [line=1, col=13]
         ├──DECL_STMT   [line=2, col=1]
         │  └──VAR_DECL a  [line=2, col=5]
         │     └──INTEGER_LITERAL   [line=2, col=9]
         ├──DECL_STMT   [line=3, col=1]
         │  └──VAR_DECL b  [line=3, col=5]
         │     └──INTEGER_LITERAL   [line=3, col=9]
         └──UNEXPOSED_EXPR   [line=4, col=1]
            ├──DECL_REF_EXPR   [line=4, col=1]
            │  └──OVERLOADED_DECL_REF fun2  [line=4, col=1]
            ├──DECL_REF_EXPR a  [line=4, col=6]
            └──DECL_REF_EXPR b  [line=4, col=9]
```

Hình 1.7: Hiển thị một cây AST sinh ra bởi mã nguồn

Một cách tổng quan, cây AST được tạo thành bởi 2 Class (lớp) rất linh hoạt: Decl và Stmt. Có nhiều lớp con của mỗi lớp, một số ví dụ:

- FunctionDecl — Đại diện cho một khai báo hoặc định nghĩa hàm.
- BinaryOperator — Biểu thức logic, toán tử… ví dụ "x + y" or "x <= y".
- CallExpr — Đại diện cho một lệnh gọi hàm, chẳng hạn như foo (x, 2)

Hầu hết các Class trong AST đều khá dễ hiểu, như ForStmt, IfStmt và ReturnStmt. Do tài liệu được viết rất khoa học và chi tiết nên một người mới có thể dễ dàng tiếp cận được AST sau khi thời gian tìm hiểu ngắn.

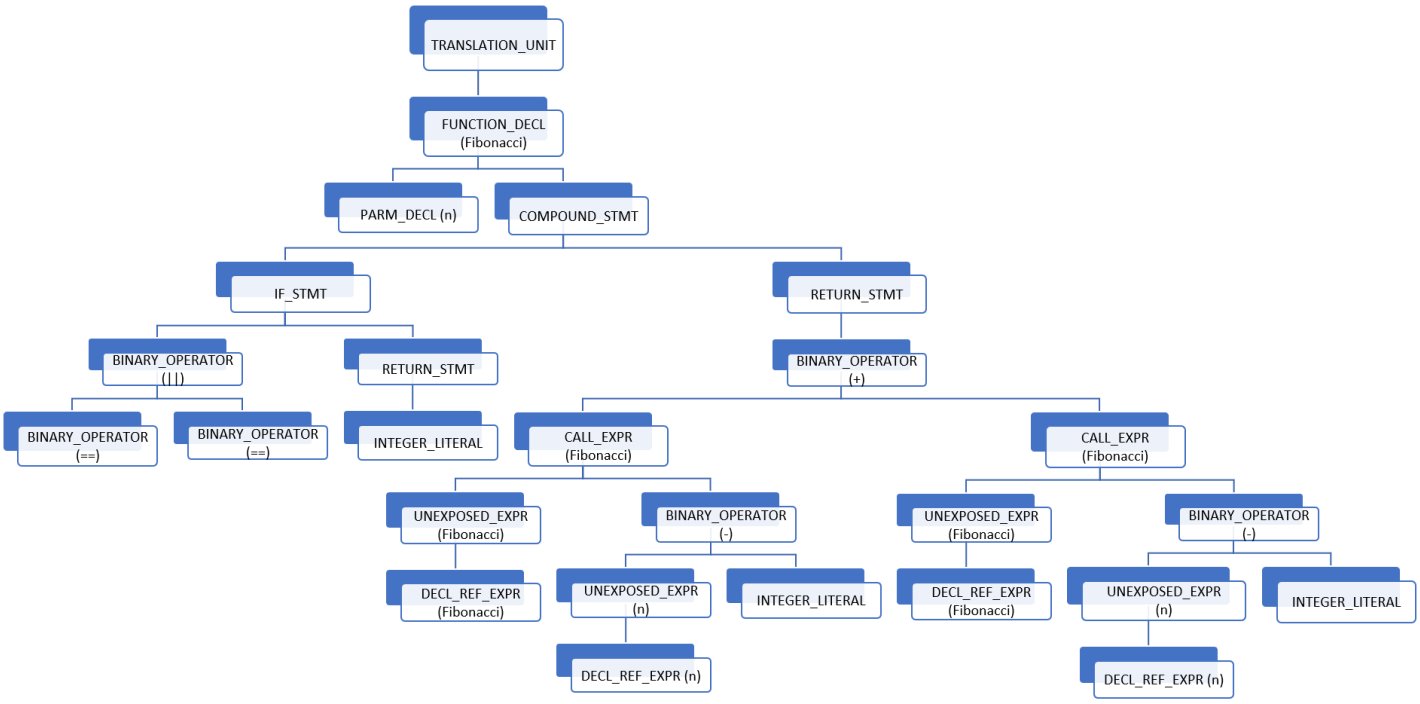
1.4. So sánh cây AST khi áp dụng các thủ thuật sao chép khác nhau

Khi sinh viên sao chép và sử dụng mã nguồn của người khác vào bài tập của mình, sinh viên thường có thêm bước thay đổi mã nguồn một chút nhằm mục đích che giấu hành vi sao chép này. Những sửa đổi này thường gây ra sự khác biệt giữa cây AST của mã nguồn gốc và mã nguồn được sửa đổi. Ở chương 2 đã chỉ ra một số phương pháp để đo mức độ tương tự nhau giữa các AST, trong chương này sẽ xem xét cây AST của mã đã sửa đổi tương ứng với AST của mã gốc như thế nào khi sử dụng các thủ thuật sao chép khác nhau.

Mã nguồn gốc để sử dụng cho mục này là bài tập xây dựng hàm tính chuỗi Fibonacci:

```
#include <stdio.h>
#include <conio.h>
int Fibonacci(int n){
  if (n == 1 || n == 2)
     return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Cây cú pháp trừu tượng sinh ra cho đoạn mã trên như sau:



Hình 1.8: Hiển thị một cây AST sinh ra bởi mã nguồn

1.4.1 Thay đổi định dạng hoặc thêm/sửa các "comment code"

```
#include <stdio.h>
#include <conio.h>
// Ngay 20/04/2022 - NamNT
// Ham Fibonacci dau vao la 1 so nguyen duong n
int Fibonacci(int n){
    if (n==1||n==2){
        return 1;
    }
    // Return noi dung can tim F(n) = F(n-1) + F(n-2)
    return Fibonacci(n - 1)+Fibonacci(n - 2);
}
```

Mã nguồn được bổ sung thêm ghi chú, thay đổi định dạng, thêm khoảng trắng

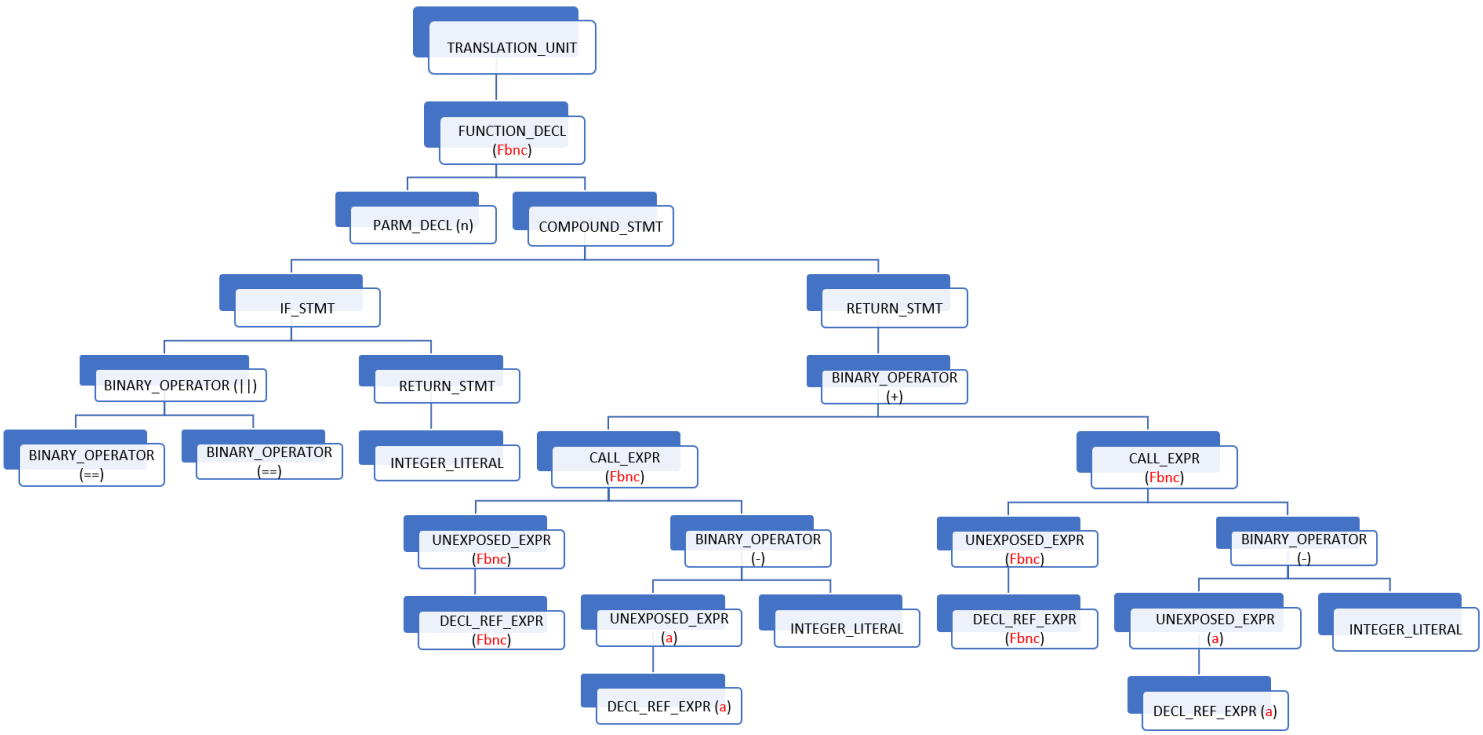
Do đặc tính của cây cú pháp trừu tượng AST, các khoảng trắng và ghi chú (comment code) bị loại bỏ khi trình biên dịch xây dựng cây AST, vì vậy việc thay đổi định dạng hoặc ghi

chú sẽ không làm ảnh hưởng tới cấu trúc của AST. Đây là thủ thuật phổ biến và dễ áp dụng nhất của sinh viên khi sao chép bài của nhau. Thông qua AST chúng ta hoàn toàn đã loại bỏ hết các điều chỉnh này. Cây AST mã nguồn mới không hề thay đổi so với cây AST mã nguồn gốc.

1.4.2 Đổi tên các định danh (hàm, tham số và biến)

```
#include <stdio.h>
#include <conio.h>
int Fbnc(int a){
    if (a == 1 || a == 2) return 1;
    return Fbnc(a - 1) + Fbnc(a - 2);
}
```

Đổi tên hàm Fibonacci() → Fbnc(), tham số đầu vào n → a



Hình 1.9: Cây AST sinh ra sau khi thực hiện đổi tên hàm và tham số đầu vào

Hình trên cho thấy việc thay đổi tên của các định danh cũng sẽ không ảnh hưởng gì đến cấu trúc và loại node của cây AST. Thay đổi các định danh chỉ làm thay đổi giá trị của node định danh chứ không làm thay đổi loại node và giữa node này với node cha và các node con (nếu có). Chúng ta sẽ vẫn có cùng một loại node, nhưng bây giờ với một giá trị khác.

Với thủ thuật sao chép này, khi so sánh cấu trúc AST, chúng ta chỉ cần bỏ qua giá trị tại các node, chỉ sử dụng loại node và cấu trúc AST để so sánh. Khi đó cấu trúc AST mã nguồn gốc và mã nguồn đã sửa đổi là tương đương nhau.

1.4.3 Thay đổi thứ tự các toán hạng trong biểu thức

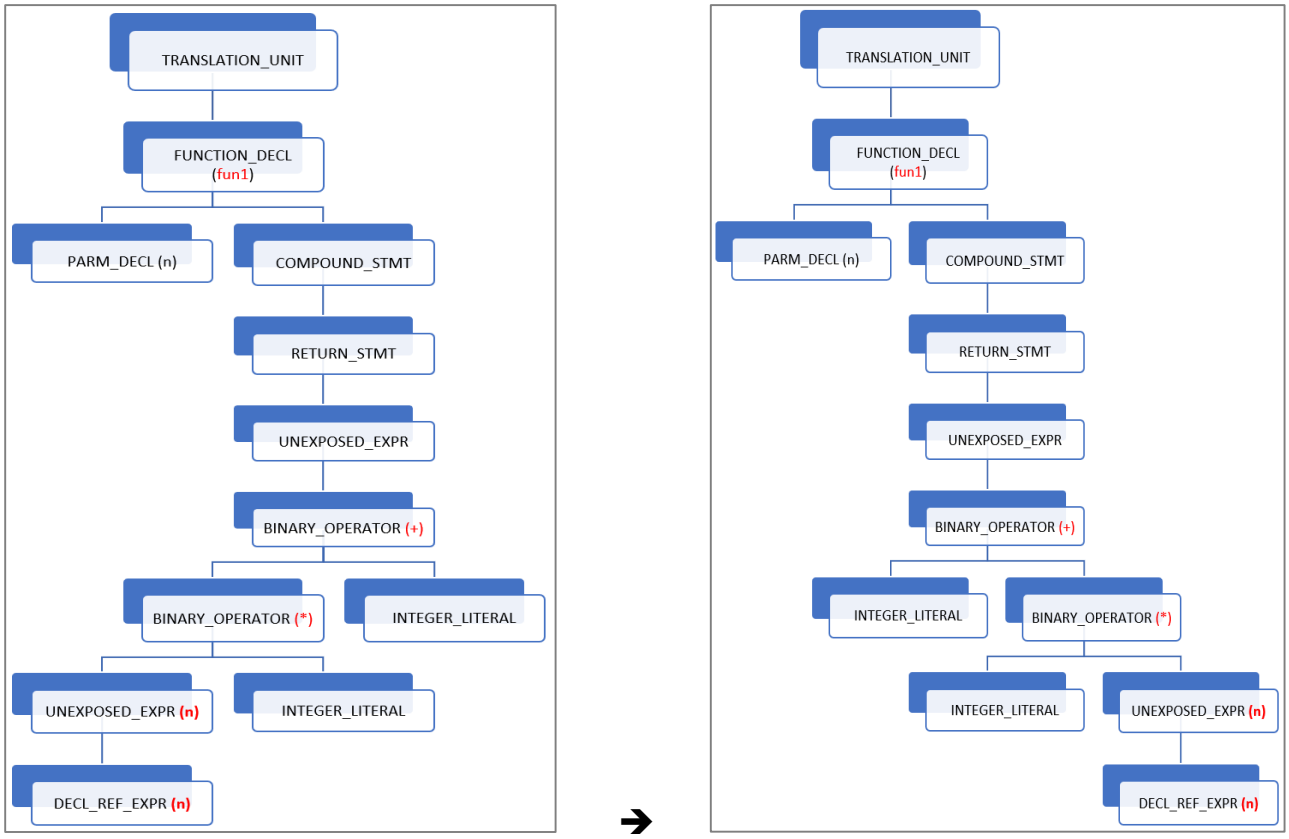
Một thủ thuật khác cũng tương đối dễ thực hiện và đảm bảo không làm thay đổi tính đúng đắn của chương trình là thay đổi thứ tự các toán hạng trong một biểu thức. Ví dụ thay đổi biểu thức trong mã nguồn sau:

```
void fun1(int n){
    return n * 8 + 3;
}
```

chuyển thành

```
void fun1(int n){
    return 3 + 8 * n;
}
```

Chúng ta thu được 2 cây AST tương ứng như sau:



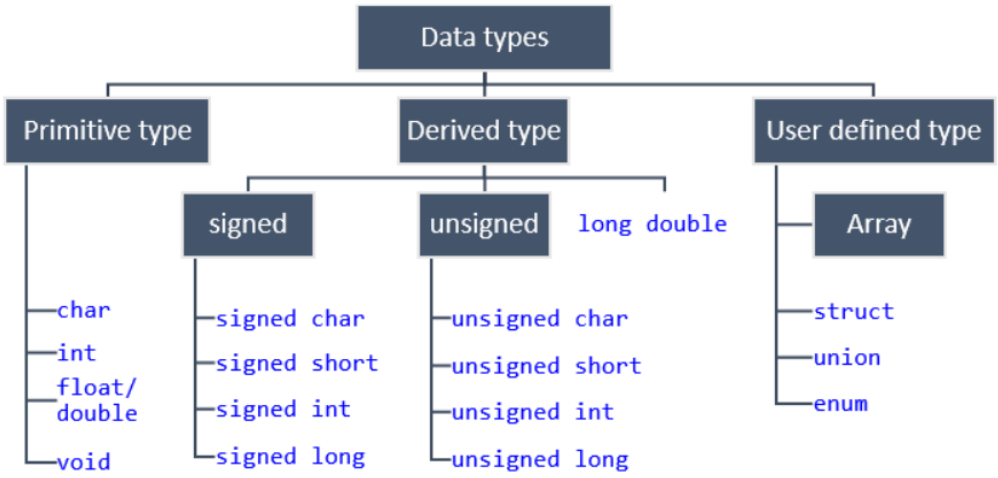
Hình 1.10: Cây AST sinh ra sau khi thay đổi thứ tự các toán hạng

Có thể thấy rằng con của hai cây AST bắt nguồn từ node BINARY_OPERATOR (+) đã được đổi vị trí cho nhau (node trái → node phải, node phải → node trái). Tương tự như vậy với node BINARY_OPERATOR (*).

Nghĩa là, phép biến đổi được áp dụng trong trường hợp này bao gồm một hoặc nhiều phép quay cây con, trong đó số lần quay phụ thuộc vào số toán hạng chuyển đổi vị trí. Thủ thuật này tạo ra một số khác biệt giữa các AST của hai mã nguồn, vì các phép biến đổi hoạt động trên các cây con thường có kích thước nhỏ.

1.4.4 Thay đổi kiểu dữ liệu (data types)

Một thủ thuật khác thường được áp dụng là thay đổi các kiểu dữ liệu trong cùng nhóm.



Hình 1.11: Các kiểu dữ liệu trong ngôn ngữ lập trình C++

Để đảm bảo tính logic của chương trình mà không cần sửa nhiều, cơ bản thủ thuật thay đổi kiểu dữ liệu tập trung vào các kiểu dữ liệu thuộc cùng nhóm (ví dụ cùng nhóm là số, nhóm chuỗi ký tự…). Việc thay đổi này hoàn toàn không làm thay đổi cấu trúc và type của node cây AST.

1.4.5 Thay thế giữa các biểu thức tương đương

Trong C++ có một số loại toán tử phổ biến như sau:

- **Arithmetic Operators** (toán tử số học): Là các Toán tử được sử dụng để thực hiện các phép tính toán học lên toán hạng. Ví dụ: (+, -, *, /, %,++,–). Toán tử số học được chia làm 2 loại:
    - **Unary Operators** (toán tử đơn ngôi): Là operator tác động tới một toán hạng duy nhất. Ví dụ: (++ , –)
    - **Binary Operators** (toán tử đa ngôi): Là các operator tác động đến hai toán hạng. Ví dụ: (+ , – , * , /)

- **Relational Operators (toán tử quy chiếu):** Relational operators được dùng để so sánh giá trị giữa hai toán hạng với nhau. Ví dụ như: để kiểm tra xem toán hạng này có bằng với toán hạng kia hay không, toán hạng này có lớn hơn toán hạng kia hay không,… Ví dụ (==, >= , <=).
- **Logical Operators** (toán tử logic): Được dùng để liên kết hai hoặc nhiều điều kiện/hạn chế, hoặc để bổ sung đánh giá điều kiện cần cân nhắc ban đầu, kết quả đầu ra là giá trị boolean true/false.
- **Bitwise Operators** (toán tử cho bit): được dùng để thực hiện thao tác cấp-bit lên các toán hạng.
- **Assignment Operators** (toán tử gán): Assignment operators được dùng để gán giá trị cho một variable (biến). Toán tử gán có thể kể đến: =, +=, -=, *=, /=.
- **Các Toán tử khác**: ví dụ sizeof, toán tử điều kiện, toán tử phẩy

Trong cây cú pháp AST, các loại toán tử này được biểu diễn tương ứng các loại node trong cây AST, các toán tử logic, toán tử quy chiếu,… được quy về chung loại BINARY_OPERATOR.

Biểu thức gốc	Biểu thức tương đương
n = n + 10;	n += 10;
n = n + 1;	n++;
n = n – 10;	n -= 10;
n = n * 10	n *= 10;

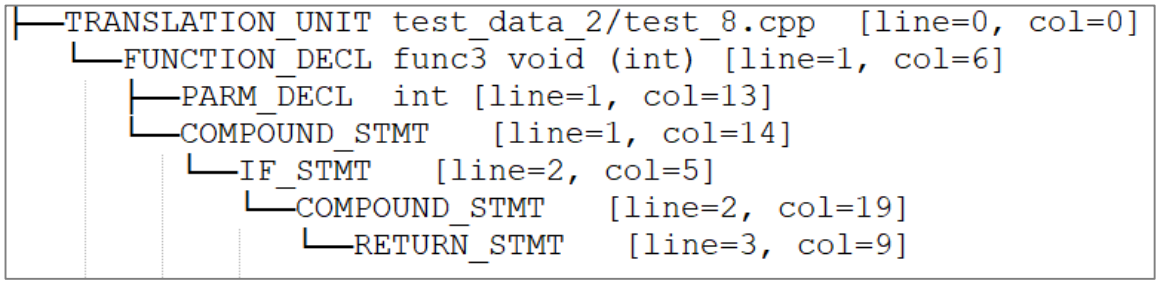
Bảng 1.3: Ví dụ thay thế các toán tử tương đương

1.4.6 Bổ sung các đoạn mã nguồn không có giá trị (dead-code)

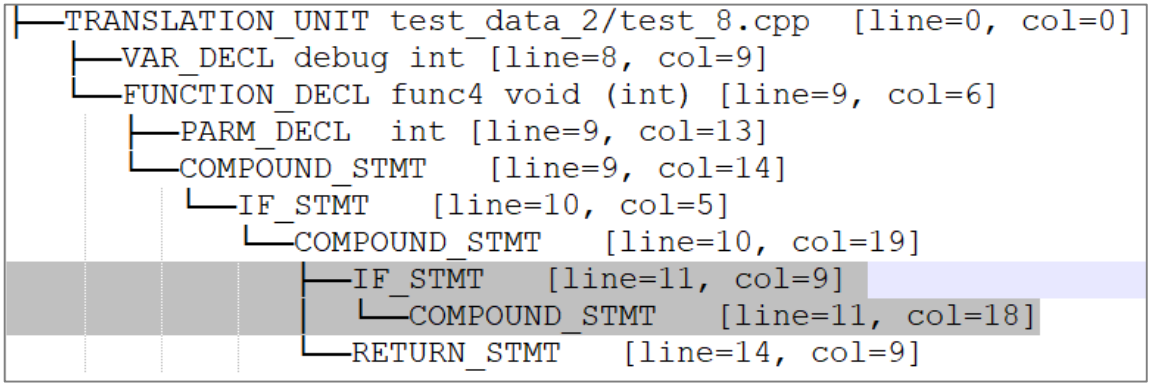
```
void func3(n){
    if ( n == 0 ) {
        return;
    }
}
```

chuyển thành

```
boolean debug = false ;
void func4(n){
    if ( n == 0 ) {
        if(debug){
            cout << "ERROR: ...";
        }
        return;
    }
}
```



Hình 1.12: Cây AST trước khi thay thế biểu thức tương đương



Hình 1.13: Cây AST sau khi thay thế biểu thức tương đương

Như vậy có thể thấy so với mã nguồn gốc, cây AST của mã nguồn sửa đổi đã bổ sung thêm một số node được bôi màu như ảnh trên, còn lại cấu trúc chính của cây cú pháp không thay đổi. Trong trường hợp này, để xác định sự giống nhau giữa hai cây AST, chúng ta cần tập trung vào các cây con mà chúng có điểm chung. Đối với thủ thuật sao chép này, số lượng khác biệt có thể nhận được giữa các AST của hai mã nguồn phụ thuộc rất nhiều vào loại và số lượng mã dư thừa đã được chèn vào. Ví dụ, việc chèn một câu lệnh thừa hoặc khai báo biến cục bộ sẽ tạo ra một số khác biệt, trong khi việc chèn một phương thức hoặc lớp dư thừa có thể tạo ra sự khác biệt lớn.

1.4.7 Thay đổi thứ tự của những đoạn mã nguồn độc lập

Trong một "khối mã nguồn" (code block), chúng ta có thể thay đổi thứ tự của các câu lệnh độc lập và các khai báo biến cục bộ mà không làm thay đổi bản chất của đoạn mã nguồn. interface- và enum là không quan trọng. Ví dụ, trong một khai báo lớp, thứ tự của các hàm khởi tạo và các thuộc tính, các phương thức là không quan trọng.

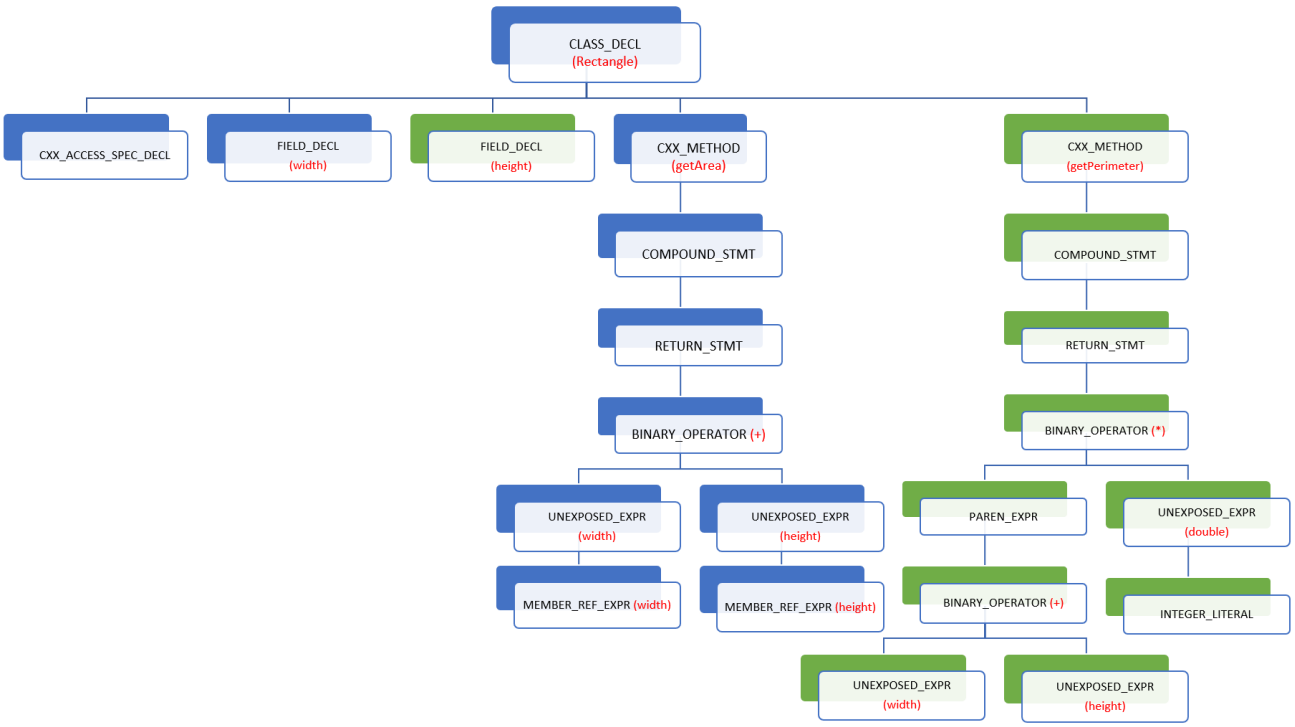
Hãy xem ví dụ dưới đây, chúng ta thay đổi thứ tự một thuộc tính và phương thức của lớp “Car” như sau:

```
class Rectangle {
    public:
        double width;
        double height;
        double getArea(){
            return width * height;
        }
        double getPerimeter(){
            return (width + height) * 2;
        }
};
```

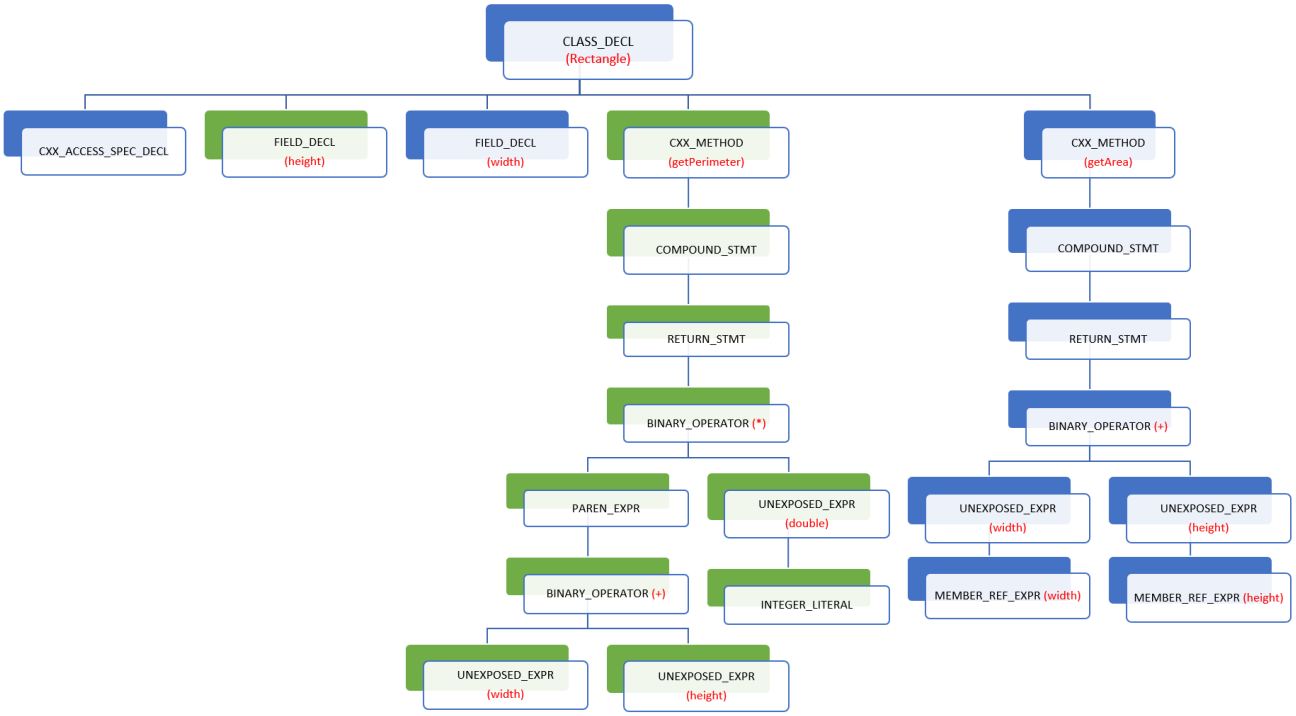
Sau khi thay đổi thứ tự các thuộc tính và phương thức, đoạn mã chuyển thành:

```
class Rectangle {
    public:
        double height;
        double width;
        double getPerimeter(){
            return (width + height) * 2;
        }
        double getArea(){
            return width * height;
        }
};
```

Cây cú pháp trừu tượng AST của đoạn mã nguồn gốc như sau:



Hình 1.14: Cây AST trước khi thay đổi thứ tự các đoạn mã nguồn



Hình 1.15: Cây AST sau khi thay đổi thứ tự các thuộc tính và phương thức của lớp

Trong đoạn mã khai báo Class Rectangle, chúng ta có khai báo các thuộc tính *width*, *height* và các phương thức *getArea()* và *getPerimeter()*. Nếu chúng ta thay đổi thứ tự của các khai báo này, chúng ta sẽ nhận được mã nguồn mới. So sánh 2 cây AST sinh ra bởi hai mã nguồn, cho thấy sự khác biệt duy nhất giữa những cây này là chúng tôi đã thay đổi thứ tự của các con của CLASS_DECL. Phép biến đổi đã được áp dụng là phép quay các cây con. Nếu chúng ta thay đổi thứ tự của các câu lệnh và khai báo biến cục bộ trong một khối, thì chúng ta sẽ áp dụng các phép quay. Mặt khác, nếu chúng ta có thể di chuyển một câu lệnh hoặc một khai báo biến cục bộ ra bên ngoài khối, thì chúng ta sử dụng một phép biến đổi trước tiên loại bỏ cây con của câu lệnh / khai báo biến cục bộ và sau đó chèn cây con này vào một chỗ khác trong AST.

Trong ví dụ trên, chỉ thứ tự của một số trường khai báo được thay đổi, nếu chúng ta thay đổi thứ tự của khai báo lớp hoặc khai báo phương thức, thì sự khác biệt giữa hai cây sẽ lớn hơn nhiều. Sự khác biệt giữa hai cây tùy thuộc vào loại cấu trúc được thay đổi vị trí.

1.4.8 Thay thế một câu lệnh lặp bằng một câu lệnh tương đương

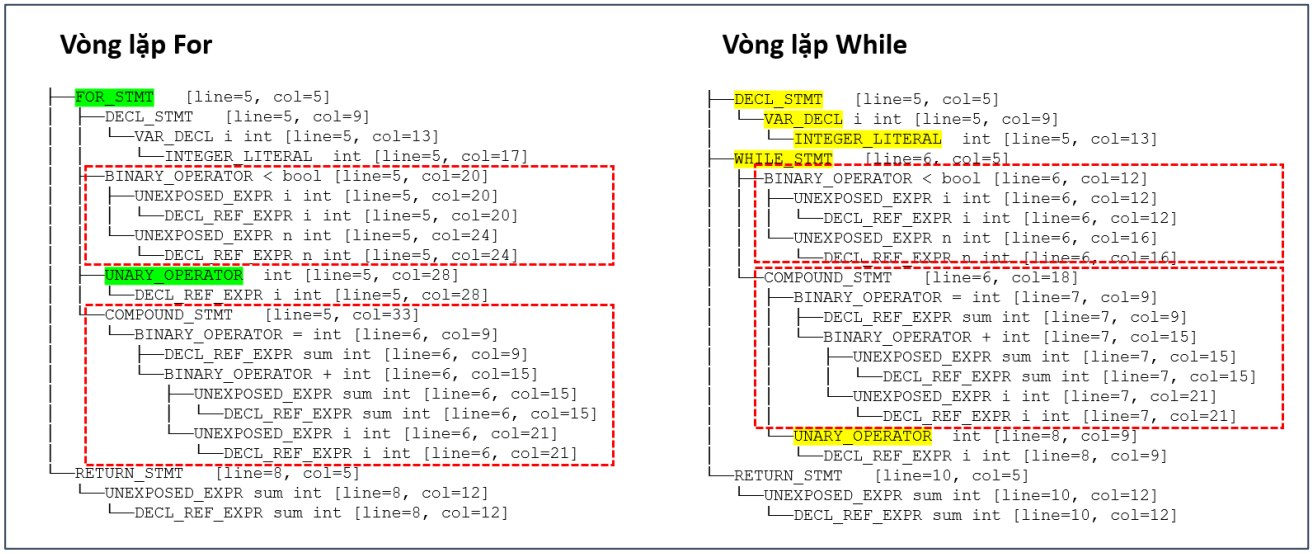
Vòng lặp *for* có thể được thay thế bằng vòng lặp *while (do-while hoặc while-do)* và ngược lại. Hãy xem xét mã trong ví dụ dưới đây:

```
int func1(int n){
    int sum = 0;
    for(int i = 0; i < n ; i++) {
        sum = sum + i;
    }
    return sum;
}
```

Thay thế vòng lặp *for* bằng vòng lặp *while* ta có:

```
int func1(int n){
    int sum = 0;
    int i = 0;
    while (i < n){
        sum = sum + i;
        i++;
    }
    return sum;
}
```

2 cây AST được sinh ra thông qua thư viện ast_viewer.py như sau:



Hình 1.16: Cây AST sau khi thay thế vòng lặp for bằng vòng lặp while

Kết quả cho thấy 2 cây AST có những sự thay đổi nhất định, nhưng về cơ bản là không nhiều, các nhóm node đại diện cho câu lệnh so sánh trong điều kiện dừng của for và while, câu lệnh bao gồm các toán tử toán hạng trong các vòng lặp cũng không thay đổi. Một điểm đáng

chú ý như khi chuyển sang vòng lặp while, cây AST có thêm một nhánh khai báo biến *i* ngay trên vòng lặp while, phép tự tăng biến đại diện vòng lặp là *i* cũng thay đổi vị trí.

Trong trường hợp thay mã nguồn gốc bằng do-while, kết quả cây AST sẽ cũng không cho nhiều khác biệt. Do đặc điểm thực hiện "do" trước "while" nên điều kiện so sánh cũng cần có sự thay đổi từ "i < n" thành "i < n-1". Trên cây AST thay thế WHILE_STMT bằng DO_STMT, thứ tự 2 nhánh nằm trong DO_STMT cũng ngược lại so với WHILE_STMT.

```
int func1(int n){
    int sum = 0;
    int i = 0;
    do {
        sum = sum + i;
        i++;
    }
    while (i < n-1);
    return sum;
}
```

1.4.9 Thay đổi các câu lệnh rẽ nhánh tương đương

Các câu lệnh *if* lồng nhau đôi khi có thể được thay thế bằng một chuỗi các câu lệnh *if*. Chúng ta cũng có thể sử dụng câu lệnh *if* thay vì câu lệnh *switch* hoặc ngược lại.

```
void main(){
    int day = 4;
    switch (day) {
          case 6:
                cout << "Today is Saturday";
                break;
          case 7:
                cout << "Today is Sunday";
                break;
          default:
                cout << "Looking forward to the Weekend";
    }
}
```

Rẽ nhánh với *switch-case*

```
void main(){
    int day = 4;
    if(day == 6){
        cout << "Today is Saturday";
        return;
    }
    if(day == 7){
        cout << "Today is Sunday";
        return;
    }

    cout << "Looking forward to the Weekend";
    return;
}
```

Rẽ nhánh với *if*

```
void main(){
    int day = 4;
    if(day == 6){
        cout << "Today is Saturday";
    }
    else if(day == 7){
        cout << "Today is Sunday";
    }
    else{
        cout << "Looking forward to the Weekend";
    }
}
```

Rẽ nhánh với *if-else*

Chúng ta có thể thấy rằng hai cây đầu tiên và hai cây cuối cùng không quá giống nhau, nhưng chúng có một số cây con giống hệt nhau (cây con màu xám). Sự khác biệt nhận được giữa hai AST bằng cách thay thế các câu lệnh lựa chọn bằng các câu lệnh lựa chọn khác phụ thuộc nhiều vào kích thước của các cây con có màu xám. Các cây con này càng lớn thì sự khác biệt càng lớn.

1.4.10 Thay thế các lệnh gọi hàm bằng chính nội dung của hàm

Hãy xem xét 2 đoạn mã nguồn dưới đây để hiểu rõ hơn về cơ chế sao chép này:

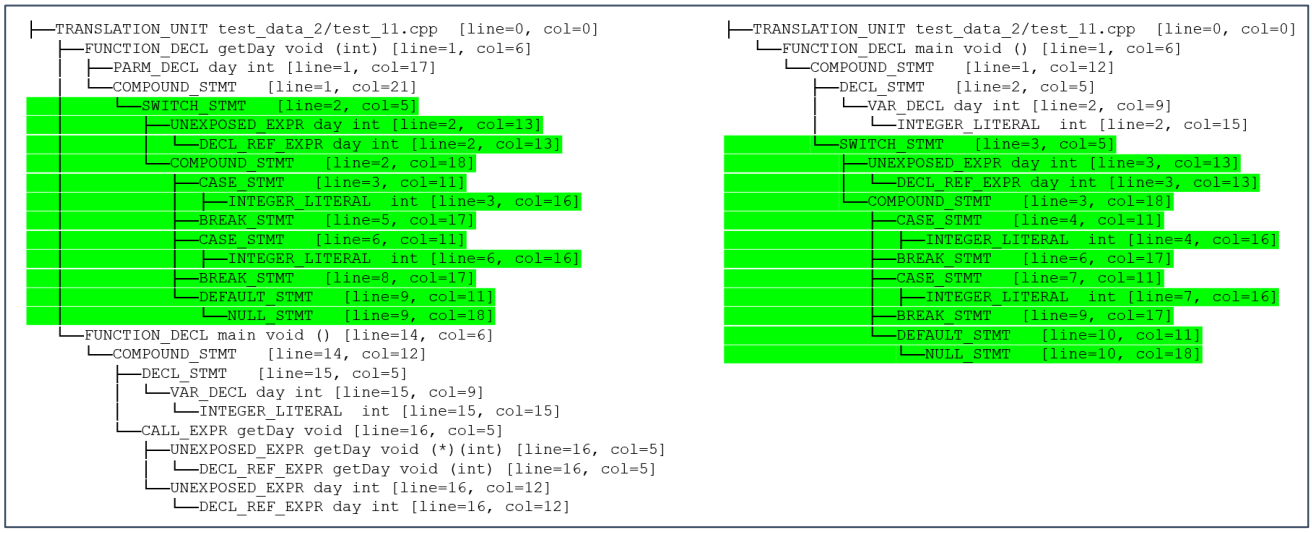
```
void getDay(int day){
    switch (day) {
          case 6:
                cout << "Today is Saturday";
                break;
          case 7:
                cout << "Today is Sunday";
                break;
          default:
                cout << "Looking forward to the Weekend";
    }
}
void main(){
    int day = 4;
    getDay(day);
}
```

Đoạn mã gốc, có lời gọi hàm getDay() trong hàm main()

```
void main(){
    int day = 4;
    if(day == 6){
        cout << "Today is Saturday";
    }
    else if(day == 7){
        cout << "Today is Sunday";
    }
    else{
        cout << "Looking forward to the Weekend";
    }
}
```

Đoạn mã đã thay thế gọi tên hàm bằng chính nội dung hàm

Các phép biến đổi có thể được sử dụng, phụ thuộc vào đoạn mã nguồn được thay thế. Trong ví dụ này, chúng ta có một hàm không trả về giá trị nào. Sự khác biệt nhận được giữa hai AST khi sử dụng chiến lược này phụ thuộc vào kích thước của phần thân phương pháp được di chuyển. Trong ví dụ này, phần nội dung của hàm không quá lớn, nhưng đối với các mã nguồn khác, có thể có phần nội dung của hàm lớn hơn nhiều.



Hình 1.17: Cây AST sau khi thay thế lời gọi hàm bằng nội dung hàm

1.4.11 Kết hợp đoạn mã nguồn sao chép với mã nguồn tự viết

Như đã phân tích ở phần mở đầu, thủ thuật sao chép này ngoài việc khó phát hiện, khó đánh giá mức độ tương tự, còn khó đánh giá rằng hành động này có cần phải ngăn chặn hay phản đối hay không. Việc sinh viên sử dụng lại mã nguồn một cách chủ động có vận dụng kiến thức của bản thân cũng giúp cho sinh viên hiểu bài hơn và có trải nghiệm sâu sắc với ngôn ngữ lập trình hơn nhiều so với việc sao chép thụ động. Do đó, việc đưa ra thuật toán để đánh giá mức độ tương tự của 2 mã nguồn khi sử dụng thủ thuật này là không cần thiết.

1.5. Đánh giá sự khác nhau giữa các cây AST với mỗi thủ thuật sao chép

Sự khác biệt giữa 2 cây AST	**Thủ thuật**
Không thay đổi	- Thay đổi định dạng hoặc thêm/sửa các “comment code” - Đổi tên các định danh (hàm, tham số và biến)
Thay đổi nhỏ	- Thay đổi thứ tự các toán hạng trong biểu thức - Thay đổi kiểu dữ liệu (data types) - Thay thế giữa các biểu thức tương đương
Có thể thay đổi nhiều	- Bổ sung các đoạn code không có giá trị (deadcode) - Thay đổi thứ tự của những đoạn mã nguồn độc lập - Thay thế một câu lệnh lặp bằng một câu lệnh tương đương

	- Thay đổi các câu lệnh rẽ nhánh tương đương - Thay thế các lệnh gọi hàm bằng chính nội dung của hàm - Kết hợp đoạn mã nguồn sao chép với mã nguồn của bản thân

Bảng 1.4: Sự khác biệt giữa 2 cây AST tương ứng với các thủ thuật sao chép khác nhau

1.6. Kết luận chương 1

Kết thúc chương 1, luận văn đã nghiên cứu, giới thiệu tổng quan về vấn đề sao chép và sử dụng lại mã nguồn, các phương pháp đánh giá mức độ tương tự giữa các mã nguồn, trình bày tổng quan về cây cú pháp trừu tượng AST và ứng dụng thư viện Clang với ngôn ngữ Python để tạo ra cây AST từ mã nguồn viết bằng ngôn ngữ C/C++.

Trong chương này, luận văn cũng đã phân tích và đánh giá sự thay đổi trên cây AST tương ứng với các cách sao chép khác nhau. Sự thay đổi giữa các cây AST đi từ ít đến nhiều, đơn giản đến phức tạp khi mà phương thức sao chép mã nguồn trở nên phức tạp hơn.

Các vấn đề nghiên cứu ở chương 1, gợi ý về việc cần có bước tiền xử lý cây AST để tinh gọn và tối ưu trước khi được sử dụng. Sau bước tiền xử lý này, việc đánh giá độ tương đồng giữa 2 mã nguồn tương đương với việc đánh giá độ tương đồng giữa 2 cây AST.

Trong chương tiếp theo, luận văn sẽ nghiên cứu các phương pháp đánh giá mức độ tương tự giữa các mã nguồn dựa trên dữ liệu từ cây AST.

2 CHƯƠNG 2: PHƯƠNG PHÁP ĐÁNH GIÁ ĐỘ TƯƠNG TỰ GIỮA CÁC MÃ NGUỒN

2.1. Tiền xử lý cây AST trước khi thực hiện đánh giá

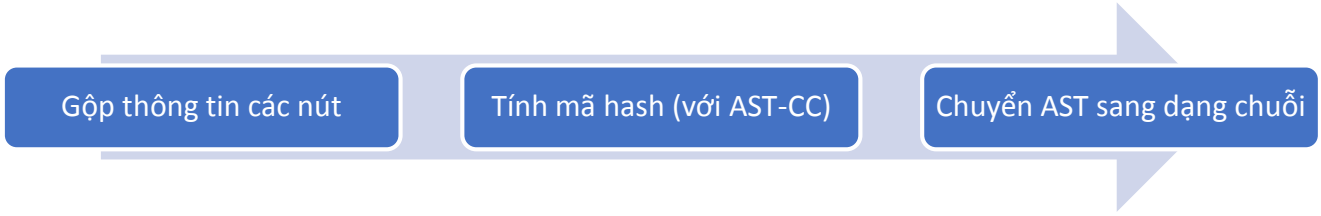
Hình 2.1: Các bước thực hiện so sánh giữa 2 mã nguồn

Có 2 lý do chính cần thực hiện việc tiền xử lý cây AST trước khi so sánh, đánh giá:

- Cây AST gốc do trình biên dịch (cụ thể trong trường hợp này là Clang) sinh ra chứa nhiều thông tin hơn mức cần thiết nếu chỉ để phục vụ cho mục đích so sánh.
- Cây AST gốc phức tạo và nhiều dữ liệu, nhiều nút, nhiều cấp, việc so sánh trực tiếp trên cấu trúc cây AST tốn nhiều tài nguyên và có hiệu năng không cao.

Ý tưởng ở đây là mục tiêu của công đoạn tiền xử lý là chuyển từ cấu trúc cây thành chuỗi (sequence) trong khi vẫn giữ được các thông tin quan trọng có ý nghĩa trong việc đánh giá mức độ tương đồng giữa 2 mã nguồn.

Công đoạn tiền xử lý cây AST cơ bản gồm 3 bước:



Hình 2.2: Các bước thực hiện tiền xử lý cây AST

Lưu ý: Riêng đối với giải thuật AST-CC có thêm bước tính toán mã hash tại mỗi nút (sẽ được trình bày chi tiết trong phần 2.3.4).

2.1.1 Gộp thông tin tại các nút

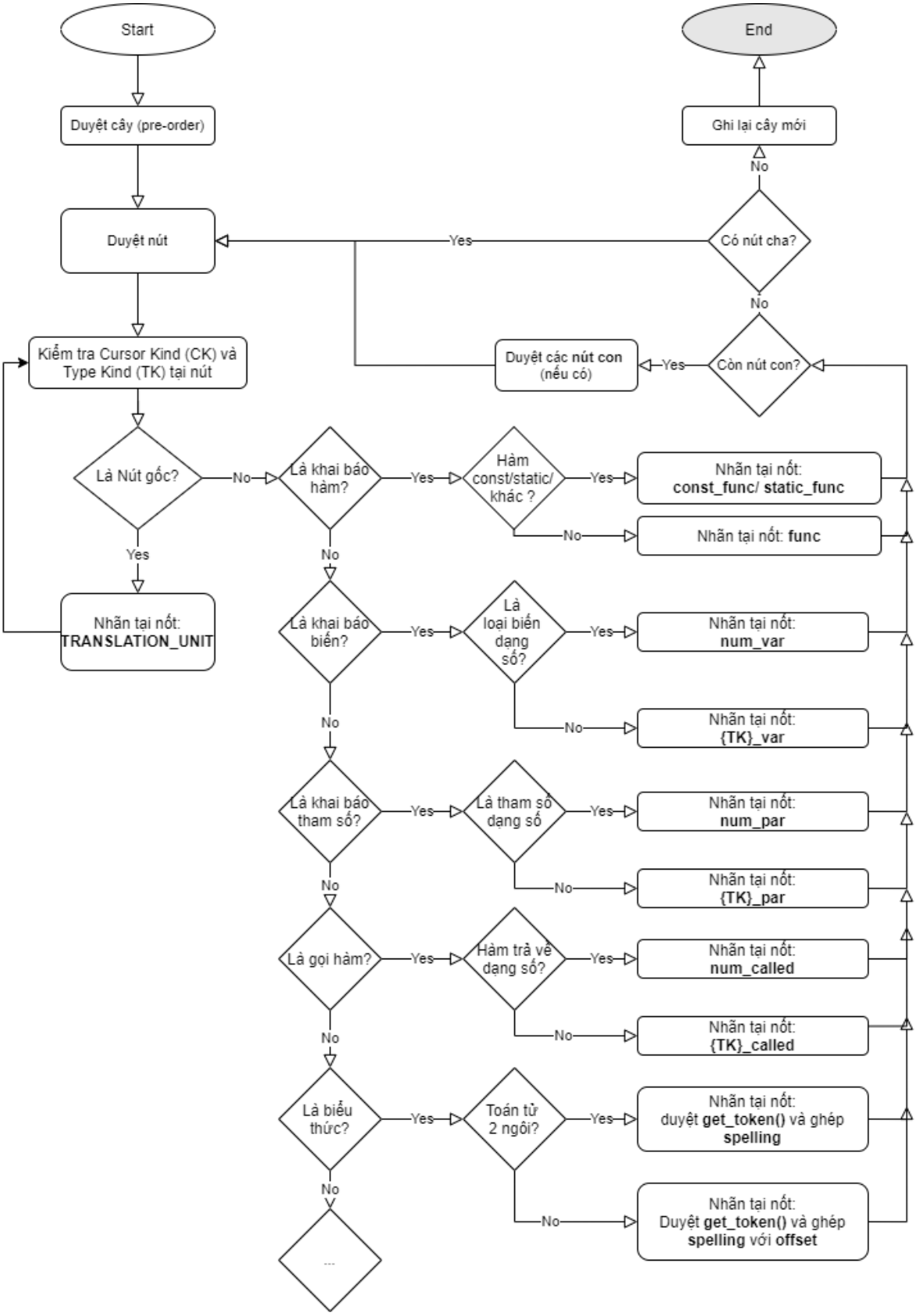
Đối với cây AST do Clang sinh ra, 2 thông tin quan trọng tại mỗi nút gồm:

- Loại nút (Cursor kind): Để biểu diễn thông tin về loại biểu thức/câu lệnh tại nút. Ví dụ là câu lệnh khai báo hàm/biến/lớp, là câu lệnh điều kiện rẽ nhánh if/switch; là toán tử 2 ngôi(cộng/trừ/nhân/chia), 1 ngôi, vòng lặp (for/while) …
- Loại kiểu (Type kind): Để biểu diễn loại của kiểu dữ liệu khai báo trong trường hợp nút là loại khai báo. Ví dụ: int/double/string/char…

Giả thuật chuyển từ cấu trúc cây thành chuỗi sẽ cần phải giữ lại 2 thông tin này nhằm đảm bảo việc so sánh đạt hiệu quả cao nhất.

Một số lưu ý khi thực hiện gộp:

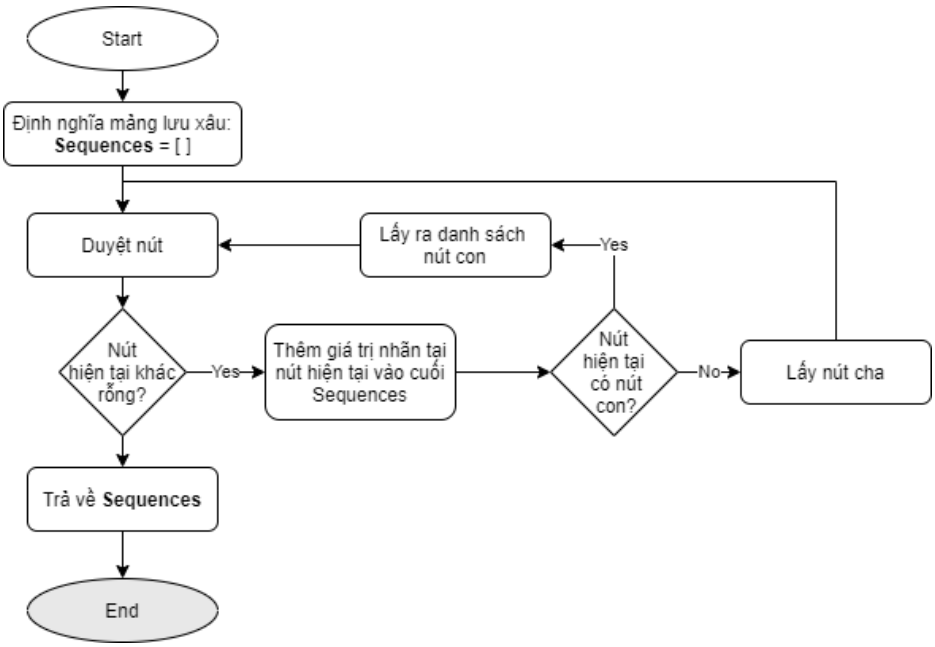
- Các loại kiểu dữ liệu cùng là dạng số (short, int, long, float, double, ushort, uint, ulong, …) được gộp lại dưới cùng từ khóa “num”.
- Các định nghĩa hàm được đặc trưng bởi hậu tố “func”
- Các khai báo biến được đặc trưng bởi hậu tố “var”
- Các tham chiếu được đặc trưng bởi hậu tố “used”
- Các lời gọi hàm/biến được đặc trưng bởi hậu tố “called”
- Các giá trị tĩnh được đặc trưng bởi “num_literal”
- Các từ khóa của ngôn ngữ lập trình được giữ nguyên “this”, “throw”, “new”, “delete”
- Các đoạn ép kiểu (casting) được đặc trưng bằng tiền tố “cast”


Start
Duyệt cây (pre-order)
Duyệt nút
Kiểm tra Cursor Kind (CK) và Type Kind (TK) tại nút
Là Nút gốc?
No
Yes
Nhãn tại nốt: TRANSLATION_UNIT
Là khai báo hàm?
Yes
No
Hàm const/static/ khác ?
Yes
Nhãn tại nốt: const_func/ static_func
No
Nhãn tại nốt: func
Là khai báo biến?
Yes
No
Là loại biến dạng số?
Yes
Nhãn tại nốt: num_var
No
Nhãn tại nốt: {TK}_var
Là khai báo tham số?
Yes
No
Là tham số dạng số
Yes
Nhãn tại nốt: num_par
No
Nhãn tại nốt: {TK}_par
Là gọi hàm?
Yes
No
Hàm trả về dạng số?
Yes
Nhãn tại nốt: num_called
No
Nhãn tại nốt: {TK}_called
Là biểu thức?
Yes
No
Toán tử 2 ngôi?
Yes
Nhãn tại nốt: duyệt get_token() và ghép spelling
No
Nhãn tại nốt: Duyệt get_token() và ghép spelling với offset
...
Còn nút con?
Yes
Duyệt các nút con (nếu có)
Có nút cha?
Yes
No
Ghi lại cây mới
No
End

Hình 2.3: Tóm tắt giải thuật tiền xử lý cây AST

2.1.2 Chuyển AST sang dạng chuỗi (xâu)

Cây AST sau khi được tiền xử lý được duyệt đệ quy để ghép vào một mảng 1 chiều lưu các chuỗi giá trị tại các nút.



Hình 2.4: Giải thuật chuyển cây AST thành chuỗi

2.2. Các phương pháp đánh giá mức độ tương tự giữa các mã nguồn

2.2.1 Tìm xâu con chung dài nhất (LCS - Longest common subsequence)

2.2.1.1 Lý thuyết về LCS

Bài toán tìm xâu con chung dài nhất (LCS) [5] được sử dụng khi chúng ta muốn tìm xâu con dài nhất giữa hai hoặc nhiều xâu (trong phạm vi nghiên cứu này, chỉ xét đến LCS giữa hai chuỗi). Bài toán xâu con chung dài nhất là một trong những bài toán khoa học máy tính cổ điển, là cơ sở của các chương trình so sánh dữ liệu như diff trên môi trường Unix. LCS còn có các ứng dụng trong ngôn ngữ học tính toán và tin sinh học. LCS cũng được sử dụng rộng rãi bởi các hệ thống quản lý phiên bản như Git để điều chỉnh nhiều thay đổi được thực hiện cho một bộ sưu tập tệp được kiểm soát sửa đổi.

Phát biểu bài toán: Cho hai xâu ký tự $A[a_1, a_2, \dots, a_m]$; $B[b_1, b_2, \dots, b_n]$ có độ dài lần lượt là m và n. Tìm xâu con chung dài nhất của hai xâu này.

Ví dụ: Ví dụ, hãy xem xét 2 xâu (ABCD) và (ACBAD). Chúng có 5 chuỗi con chung có độ dài bằng 2: (AB), (AC), (AD), (BD) và (CD); 2 chuỗi con chung có độ dài bằng 3: (ABD) và (ACD); và không còn chuỗi con chung nào khác có độ dài lớn hơn nữa. Vì vậy (ABD) và (ACD) là hai dãy con chung dài nhất của hai chuỗi ban đầu.

Phương pháp xử lý

Áp dụng nguyên lý quy hoạch động ta có thể giải quyết bài toán LCS. Gọi $L\ [i,j]$ là độ dài xâu con chung dài nhất của hai xâu $A[a_1, a_2, \dots, a_i]\ \forall i \le m; B\left[b_1, b_2, \dots, b_j\right]\ \forall j \le n$, điều này có nghĩa là $L[i,j]$ là độ dài xâu con chung dài nhất của i ký tự đầu của xâu A và j ký tự đầu của xâu B. Do đó $L[m,n]$ chính là độ dài xâu con chung dài nhất của A và B.

$L\ =\ (l_1, l_2, \dots, l_k)$ là xâu con chung dài nhất của A và B.

- Nếu $a_m =\ b_n$ thì $l_k =\ a_m =\ b_n$ và L_{k-1} là LCS của A_{m-1} và B_{n-1}.
- Nếu $a_m \ne b_n$ và $l_k \ne a_m$ thì L là LCS của A_{m-1} và B_n.
- Nếu $a_m \ne b_n$ và $l_k \ne b_n$ thì L là LCS của A_m và B_{n-1}.

Công thức truy hồi

Trường hợp i = 0 hoặc j = 0 thì độ dài xâu con chung dài nhất của A và B là 0, nghĩa là L[0, j] = L[i, 0] = 0.

Trường hợp xét $a_i = b_j$ thì ta thêm $l_h = a_i = a_j$ vào L và độ dài xâu con chung dài nhất L[i, j] sẽ bằng độ dài xâu con chung dài nhất của A_{i-1} và B_{j-1} là L[i-1,j-1] cộng thêm 1.

Ngược lại nếu $a_i \ne b_j$ ta phải tìm LCS của A_{i-1} và B và LCS của A và B_{j-1} là L[i-1,j] và L[i,j-1]. Xâu nào lớn hơn sẽ là LCS của A và B.

Ta có : $L[i,j] = \begin{cases} 0 & \forall\ i = 0, j = 0 \\ L[i-1, j-1] + 1 & \forall\ a_i = b_j \\ \max\{L[i-1, j], L[i, j-1]\} & \forall\ a_i \ne b_j \end{cases}$

2.2.1.2 Phương pháp LCS

Mô tả: Bắt đầu từ vị trí L[n][m] và dừng lại khi L[i][j] == 0.

- Bắt đầu duyệt từ $i = n\ , j = m$. Phần tử thứ i của xâu a sẽ là $a[i-1]$, thứ j của xâu b sẽ là $b[j-1]$
- Nếu $a[i-1] == b[j-1]$ ta sẽ lưu lại con chung $a[i-1]$ và giảm i và j đi 1 đơn vị

- Nếu $a[i-1]! = b[j-1]$ có 2 trường hợp:
  - Nếu $\boldsymbol{L[i-1][j] \geq L[i][j-1]}$ thì giảm i đi 1 đơn vị
  - Ngược lại giảm **j** đi 1 đơn vị

Mã giả:

```
function LCS(A[1..m],B[1..n])
   L = array(0..m,0..n)
   for i:= 0..m
      L[i,0] = 0
   for j:= 0..n
      L[0,j] = 0
   for i:= 1..m
      for j:= 1..n
         if A[i] = B[j]
            L[i,j]:= L[i-1,j-1] + 1
         else
            L[i,j]:= max(L[i,j-1],L[i-1,j])
  return L[m,n]
```

2.2.1.3 Ứng dụng LCS để so sánh 2 cây AST

Từ ý nghĩa LCS đã nêu ở phần trước, chúng ta có thể thấy 2 cây AST tương đồng khi xâu tương ứng sẽ có xâu LCS có kích thước lớn và tỉ lệ phủ cao. Để sử dụng LCS trên hai cây AST, chúng ta cần chuyển 2 cây này về **dạng chuỗi** nhưng vẫn phản ánh cấu trúc của hai cây. Điều này có thể đạt được bằng cách duyệt cây AST theo thứ tự xuôi (pre-order) hay ngược (post-order).

Từ xâu con chung dài nhất tìm được, ta xây dựng công thức hàm tính mức độ tương tự giữa 2 cây AST. Gọi 2 cây AST lần lượt là $T_1 = (V_1, E_1)$ và $T_2 = (V_2, E_2)$ tương ứng với 2 cặp mã nguồn cần so sánh là p và q. Khi đó hàm $d(p,q)$ để đo mức độ tương tự giữa p và q được định nghĩa như sau:

$$d(p,q) = \frac{|V_1| + |V_2| - 2 * LCS(p,q)}{|V_1| + |V_2|}$$

Trong đó:

- $d(p,q)$ là hàm đo mức độ tương tự, có giá trị từ 0 → 1
- $|V_1|$ là số lượng đỉnh của cây T_1, $|V_2|$ là số lượng đỉnh của cây T_2
- $LCS(p,q)$ là độ dài xâu con dài nhất của 2 cây AST

2.2.1.4 Đánh giá hiệu quả sử dụng LCS

Để sử dụng phương pháp LCS, chúng ta cần thực hiện duyệt qua từng AST. Sau đó tìm xâu con dài nhất giữa hai xâu tương ứng 2 cây AST.

Một vấn đề ở đây là thứ tự của các đoạn mã nguồn độc lập có thể đã bị thay đổi ở một trong các cây AST. Nếu thứ tự của các khai báo như lớp, phương thức, v.v. đã được thay đổi, thì chúng ta có thể nhận được tỉ lệ tương đồng giữa hai cây thấp. Điều này là do thực tế là LCS là một phương pháp được sử dụng để tìm sự liên kết giữa hai chuỗi đã được sắp xếp, trong khi chuỗi của chúng ta chứa các chuỗi con không có thứ tự làm cho công việc đối chiếu giữa 2 cây AST trở nên phức tạp hơn.

Để sử dụng phương pháp này, chúng ta cần tìm sự liên kết giữa mỗi khai báo lớp trong AST đầu tiên và mỗi khai báo lớp trong AST thứ hai. Bằng cách đó, chúng ta có thể tìm thấy các khai báo lớp có thể được căn chỉnh với nhau tốt nhất. Hơn nữa, để tìm sự liên kết giữa hai khai báo lớp, chúng ta cần tìm sự liên kết giữa mỗi khai báo phương thức trong lớp đầu tiên và mỗi khai báo phương thức trong lớp thứ hai. Chúng ta có thể tiếp tục nếu chúng ta muốn tìm sự liên kết tốt nhất giữa khai báo phương thức khởi tạo và khai báo trường.

Một thuật toán phát hiện sao chép mã nguồn sử dụng phương pháp tương tự như LCS là Sim [6]. Thuật toán này được thực hiện cho danh sách chương trình được viết bằng C và nó tìm thấy sự liên kết tối ưu giữa hai chuỗi mã thông báo. Ngoài ra, đối với thuật toán này, cần phải tìm sự liên kết tốt nhất giữa các chức năng khác nhau trong hai danh sách.

Ưu điểm chính của LCS là nó có thể tìm thấy điểm tương đồng giữa các cây con mà nhãn của các nút gốc khác nhau, chẳng hạn như cây con của các vòng lặp khác nhau hoặc cây con của các câu lệnh lựa chọn khác nhau. Ví dụ các thủ thuật thay thế câu lệnh lặp bằng các câu lệnh tương đương trình bày ở mục 2.1.8 có thể được phát hiện tốt qua thông qua LCS.
Khi nói đến việc căn chỉnh các câu lệnh và / hoặc khai báo biến cục bộ của hai khối, LCS sẽ tìm thấy sự liên kết tối ưu giữa các xâu của hai cây con bắt nguồn từ các nút thể hiện code block COMPOUND_STMT. Sự liên kết này là sự liên kết tối ưu giữa hai chuỗi có thứ tự, nhưng nó không nhất thiết phải là sự liên kết tốt nhất giữa hai cây con. Khi sử dụng phương pháp này, điều quan trọng là phải quyết định nút nào trong số các nút được căn chỉnh trong hai chuỗi là một phần liên kết giữa hai cây con.

2.2.2 TF-IDF và Độ tương tự Cosin

2.2.2.1 Lý thuyết về TF-IDF

TF-IDF (viết tắt của Term Frequency – Inverse Document Frequency) [7] là một phương pháp thống kê thường được sử dụng trong truy xuất thông tin và khai phá dữ liệu văn bản để đánh giá mức độ quan trọng của một cụm từ đối với một tài liệu cụ thể trong một tập hợp bao gồm nhiều tài liệu. Khái niệm này đã xuất hiện từ rất sớm trong các lĩnh vực nghiên cứu khác nhau, chẳng hạn như ngôn ngữ học và cấu trúc thông tin, nhờ vào khả năng hỗ trợ xử lý nhiều tập tài liệu với số lượng lớn trong một khoảng thời gian ngắn.

Các máy tìm kiếm thường sử dụng các biến số khác nhau của thuật toán TF-IDF như là một phần trong cơ chế xếp hạng. Bằng cách gán cho các tài liệu một mức điểm số về độ liên quan (relevance score), chúng có thể đưa ra các kết quả tìm kiếm thích hợp chỉ trong phần triệu giây.

Ví dụ, TF-IDF từ lâu đã là một phần trong cơ chế xếp hạng của Google. Google sử dụng TF-IDF để xác định xem những cụm từ nào có liên quan (hoặc không liên quan) về mặt chủ đề bằng cách phân tích tần suất một cụm từ xuất hiện trên một trang (term frequency – TF) và tần suất ước tính xuất hiện trên một trang trung bình, trong một tập hợp lớn hơn bao gồm nhiều tài liệu (inverse document frequency – IDF).

Như tên gọi, TF-IDF gồm 2 thành phần là TF và IDF. Sau đây là trình bày chi tiết về 2 thành phần này:

TF là gì?

TF: Term Frequency (Tần suất xuất hiện của từ) là số lần từ xuất hiện trong văn bản. Vì các văn bản có thể có độ dài ngắn khác nhau nên một số từ có thể xuất hiện nhiều lần trong một văn bản dài hơn là một văn bản ngắn. Như vậy, term frequency thường được chia cho độ dài văn bản(tổng số từ trong một văn bản). Công thức TF:

$$tf(t,d) = \frac{f(t,d)}{\max\{f(w,d)\ \forall w\ \in d\}}$$

Trong đó:

- $tf(t,d)$ tần suất xuất hiện của từ t trong văn bản $\boldsymbol{d}$
- $f(t,d)$ Số lần xuất hiện của từ t trong văn bản $\boldsymbol{d}$

- $\max\{f(w,d)\ \forall w\ \in d\}$ Số lần xuất hiện của từ có số lần xuất hiện nhiều nhất trong văn bản ***d***

IDF là gì?

IDF: Inverse Document Frequency (Nghịch đảo tần suất của văn bản), giúp đánh giá tầm quan trọng của một từ . Khi sử dụng TF, ta nhận thấy rằng tất cả các từ được coi có độ quan trọng bằng nhau. Tuy nhiên thực tế một số từ như "Từ nối": và, nhưng, tuy nhiên, vì thế, vì vậy, …; giới từ: ở, trong, trên, …; từ chỉ định: ấy, đó, nhỉ, thường xuất hiện rất nhiều lần nhưng độ quan trọng là không cao. Vì vậy để kết quả tìm kiếm chính xác, chúng ta cần giảm độ quan trọng của những từ này xuống. Công thức tính IDF như sau:

$$idf(t,D) = log\frac{|D|}{|\{d \in D : t \in D\}|}$$

Trong đó:

- $idf(t,d)$ giá trị idf của từ t trong tập văn bản
- |D| tổng số văn bản trong tập D
- |{d ∈ D : t ∈ d}| thể hiện số văn bản trong tập D có chứa từ t.

Cơ số logarit trong công thức này không thay đổi giá trị idf của từ mà chỉ thu hẹp khoảng giá trị của từ đó. Vì thay đổi cơ số sẽ dẫn đến việc giá trị của các từ thay đổi bởi một số nhất định và tỷ lệ giữa các trọng lượng với nhau sẽ không thay đổi, (nói cách khác, thay đổi cơ số sẽ không ảnh hưởng đến tỷ lệ giữa các giá trị IDF). Việc sử dụng logarit nhằm giúp giá trị TF-IDF của một từ nhỏ hơn, do chúng ta có công thức tính TF-IDF của một từ trong 1 văn bản là tích của TF và IDF của từ đó.

$$tfidf(t,d,D) = tf(t,d) * idf(t,D)$$

Những từ có giá trị TF-IDF cao là những từ xuất hiện nhiều trong văn bản này, và xuất hiện ít trong các văn bản khác. Việc này giúp lọc ra những từ phổ biến và giữ lại những từ có giá trị cao (từ khoá của văn bản đó).

Ví dụ: chúng ta có 2 văn bản sau "*The car is driven on the road*" và "*The truck is driven on the highway*" Sử dụng công thức ở trên chúng ta có bảng tính giá trị TF-IDF như sau:

Word	**TF**		**IDF**	**TF*IDF**	
	A	**B**		**A**	**B**
The	1/7	1/7	$\log(2/2) = 0$	0	0

Car	1/7	0	$\log(2/1) = 0{,}3$	**0,043**	0
Truck	0	1/7	$\log(2/1) = 0{,}3$	0	**0,043**
Is	1/7	1/7	$\log(2/2) = 0$	0	0
Driven	1/7	1/7	$\log(2/2) = 0$	0	0
On	1/7	1/7	$\log(2/2) = 0$	0	0
The	1/7	1/7	$\log(2/2) = 0$	0	0
Road	1/7	0	$\log(2/1) = 0{,}3$	**0,043**	0
Highway	0	1/7	$\log(2/1) = 0{,}3$	0	**0,043**

Bảng 2.1: Bảng tính toán giá trị TF-IDF của 2 xâu A và B

Từ ví dụ trên, có thể thấy các từ dừng không có nhiều ý nghĩa trong việc phân biệt 2 văn bản như "is", "on", "the" đều có trọng số bằng 0. Trong chiều ngược lại, các từ khóa quan trọng như "car", "truck", "road", "highway" có trọng số khác 0.

2.2.2.2 Phương pháp TF-IDF

```
In [94]: def computeTF(wordDict, bow):
             tfDict = {}
             bowCount = len(bow)
             for word, count in wordDict.items():
                 tfDict[word] = count/float(bowCount)
             return tfDict
```

Tính giá trị TF

```
In [98]: def computeIDF(docList):
             import math
             idfDict = {}
             N = len(docList)

             idfDict = dict.fromkeys(docList[0].keys(), 0)
             for doc in docList:
                 for word, val in doc.items():
                     if val > 0:
                         idfDict[word] += 1

             for word, val in idfDict.items():
                 idfDict[word] = math.log10(N / float(val))

             return idfDict
```

Tính giá trị IDF

```
In [100]: def computeTFIDF(tfBow, idfs):
              tfidf = {}
              for word, val in tfBow.items():
                  tfidf[word] = val*idfs[word]
              return tfidf
```

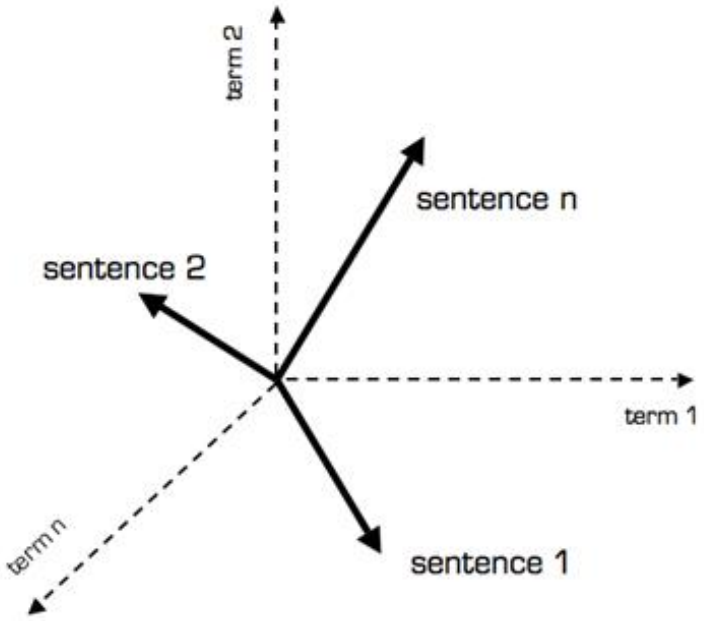
Tính giá trị TF-IDF

2.2.2.3 Lý thuyết về Mô hình không gian vector (Vector space model)

Mô hình không gian vector là một mô hình đại số biểu diễn các văn bản dưới dạng véc-tơ, các phần tử của véc-tơ này thể hiện mức độ quan trọng của một từ và cả sự xuất hiện hay

không xuất hiện của nó trong một tài liệu. Mỗi thành phần trong véc-tơ chỉ đến một từ tương ứng trong danh sách mục từ chính. Không gian véc-tơ có kích thước bằng số mục từ trong danh sách mục từ chính. Mỗi phần tử là độ quan trọng của mục từ tương ứng trong câu.

Mô hình này biểu diễn văn bản như những điểm trong không gian Euclid n-chiều, mỗi chiều tương ứng với một từ trong tập hợp các từ. Phần tử thứ i, là di của véc-tơ văn bản cho biết số lần mà từ thứ i xuất hiện trong văn bản. Sự tương đồng của hai văn bản được định nghĩa là khoảng cách giữa các điểm, hoặc là góc giữa những véc-tơ trong không gian. Mô hình không gian vector và những biến thể của nó hiện nay vẫn là cách phổ biến để biểu diễn văn bản trong Khai phá dữ liệu và Truy vấn thông tin.



Hình 2.5: Biểu diễn các văn bản trên mô hình không gian véc-tơ

Nguyên tắc chung để biểu diễn văn bản dưới dạng các véc-tơ là tách thành các từ/cụm từ trong văn bản đi kèm theo trọng số của các từ/cụm từ này. Có nhiều phương pháp để đánh trọng số các từ/cụm từ trong văn bản, nhưng TF-IDF là phương pháp phổ biến để đánh giá và xếp hạng.

2.2.2.4 Lý thuyết về Độ tương tự Cosin (Cosine similarity)

Độ tương tự Cosin là một cách đo ***độ tương tự*** giữa 2 vectơ khác 0 trong một không gian véc-tơ đã được chuẩn hóa. Độ tương tự này được định nghĩa bằng giá trị cosine của góc

giữa hai véc-tơ, và cũng là tích vô hướng của cùng các véc-tơ đơn vị để cả hai đều có chiều dài 1 [15]. Giá trị cosin của 0° là 1, và bé hơn 1 với bất kỳ góc nào trong khoảng các radian (0, π].

Sau khi đã biểu diễn 2 văn bản dưới dạng véc-tơ thì có thể sử dụng phép đo Độ tương tự Cosin để đánh giá mức tương đồng giữa 2 văn bản này.

Công thức Độ tương tự Cosin được tính như sau:

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q},\vec{d}}{\overrightarrow{|q|} * \overrightarrow{|d|}} = \frac{\vec{q}}{\overrightarrow{|q|}} * \frac{\vec{d}}{\overrightarrow{|d|}} = \frac{\sum_{i=1}^{|v|} q_i * d_i}{\sqrt{\sum_{i=1}^{|v|} q_i^2} * \sqrt{\sum_{i=1}^{|v|} d_i^2}}$$

Trong đó:

- q_i là trọng số TF-IDF của từ thứ i trong tài liệu thứ nhất
- d_i là trọng số TF-IDF của từ i trong tài liệu thứ hai
- $\cos(\vec{q},\vec{d})$ là Độ tương tự Cosin giữa $\vec{q}$ và $\vec{d}$, luôn nằm trong khoảng [0,1].

Đặc điểm:

- Nếu độ tương tự Cosin của 2 văn bản A, B bằng 1, nghĩa là góc hợp bởi 2 véc-tơ biểu diễn là 0 độ. Tức là 2 văn bản A, B giống nhau.
- Nếu độ tương tự Cosin càng tiến dần về 0 thì A, B càng khác nhau.

2.2.2.5 Ứng dụng TF-IDF và Độ tương tự Cosin để so sánh 2 cây AST

Sau bước tiền xử lý ở mục 1.3.4, cây AST của mã nguồn đã được chuyển thành dạng chuỗi, tức là 2 văn bản, với nội dung là nhãn của các node của cây AST. Để so sánh độ tương đồng giữa 2 văn bản này, chúng ta sử dụng độ tương tự TF-IDF và Độ tương tự Cosin để so sánh mức độ tương đồng. Nếu độ tương tự cosin cho kết quả càng gần 1, càng cho thấy 2 cây AST có mức tương đồng cao.

2.2.2.6 Đánh giá hiệu quả sử dụng TF-IDF và Độ tương tự Cosin

Khi áp dụng TF-IDF và độ tương tự Cosin vào đánh giá độ tương tự giữa 2 cây AST, có một số đặc điểm như sau:

- Đánh giá độ tương tự dựa trên tần suất và độ quan trọng của giá trị tại các nút
- Ít bị ảnh hưởng bởi sự thay đổi thứ tự các đoạn mã nguồn (trong khi phương pháp LCS trình bày ở mục 1.3.5 bị ảnh hưởng nhiều)
- Có thể gây ra tỉ lệ dương tính giả cao nếu đặt n-gram không hợp lý.

2.2.3 AST-CC (AST Code Comparison)

2.2.3.1 Tổng quan về AST-CC

Vào năm 2015, tại Hội nghị quốc tế lần thứ 10 về Máy tính, Truyền thông và Ứng dụng Băng thông rộng và Không dây, nhóm tác giả Jingling Zhao, Kunfeng Xia, Yilun Fu đến từ Đại học Bưu chính Viễn thông Bắc Kinh, Bắc Kinh; Phòng thí nghiệm Kỹ thuật Quốc gia về An ninh Mạng Di động; Viện nghiên cứu điện lực Trung Quốc, Bắc Kinh; đã giới thiệu một thuật toán phát hiện sao chép mã nguồn đáng tin cậy dựa trên cây AST [8].

Phương pháp này được nhóm tác giả đặt tên là AST-CC (AST Code Comparison).

Ý tưởng của phương pháp như sau:

Sau khi có được cấu trúc cây cú pháp của mã nguồn, việc cần làm là so sánh cây cú pháp. Tuy nhiên trong thực tế, các cây cú pháp đều khá phức tạp và có kích thước lớn nên việc so sánh trực tiếp giữa các cây cú pháp là khó và hiệu quả sẽ thấp. Vì vậy, nhóm tác giả sử dụng giá trị băm của cây cú pháp và tính toán giá trị dựa trên loại của mỗi nút và các cây con của cây AST. Sau đó so sánh các giá trị băm này. Nhờ đó, độ khó so sánh được giảm bớt và thông tin của cây cú pháp không bị mất.

Hình 2.6: Ý tưởng của phương pháp AST-CC

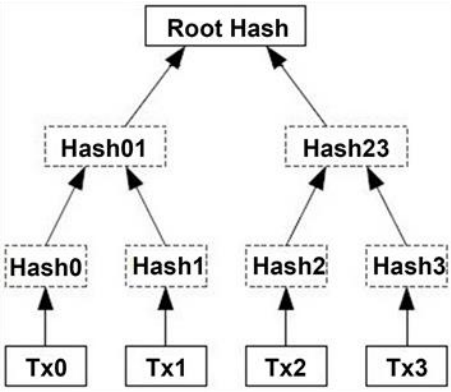
2.2.3.2 Các bước thực hiện AST-CC

Các bước thực hiện AST-CC có 3 bước chính:

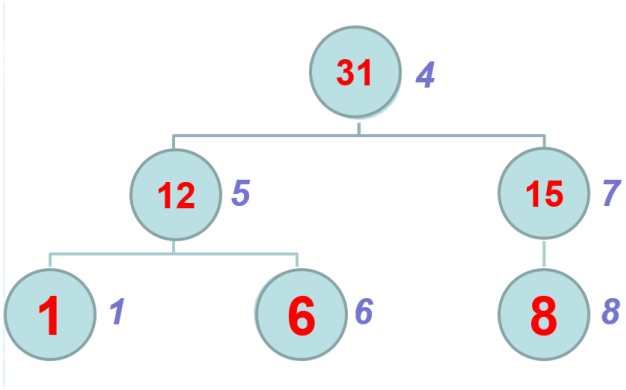
- Đọc thông tin của mã nguồn cần kiểm tra từ HashListArray tại vị trí có chỉ số bằng với ngưỡng (tức là số lượng nút con bằng giá trị ngưỡng), sau đó so sánh với mã gốc.
- Duyệt qua các mảng của SuspectHashListArray và OriginalHashListArray lưu trữ thông tin của mã cần kiểm tra và mã gốc tương ứng. Sau đó, sắp xếp hai mảng này và đảm bảo rằng chuỗi của các nút được lưu trữ theo số nút và giá trị băm.

- So sánh giá trị băm của các cây con có cùng số nút con. Nếu chúng có cùng một giá trị băm, thì lưu trữ vị trí của chúng và tăng số lượng nút trùng thêm 1.

Vì các cây con có số nút con khác nhau đại diện cho các cấu trúc khác nhau, nên chỉ các cặp cây con có cùng số nút con mới cần so sánh. Do đó chúng ta có thể giảm số lần so sánh và độ phức tạp của thuật toán.



Hình 2.7: Mô phỏng việc tính toán mã hash tại các nút của cây AST



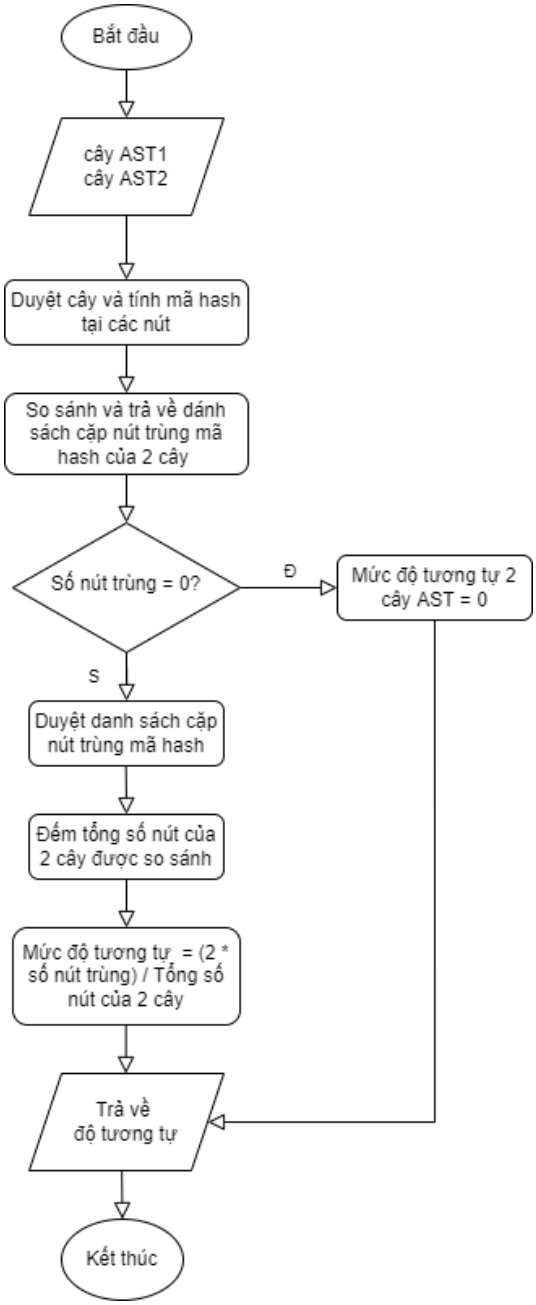
Hình 2.8: Ví dụ minh họa việc mã hash tại các nút được cộng dồn

Công thức tính độ tương tự giữa 2 cây AST sử dụng phương pháp AST-CC được tính như sau:

$$(p, q) = \frac{2 * \mathrm{m}}{\mathrm{n}_1 + \mathrm{n}_2}$$

Trong đó:

- m là số nút có trùng mã hash giữa 2 AST
- n_1 là số nút của cây AST thứ nhất
- n_2 là số nút của cây AST thứ hai



Hình 2.9: Mô tả giải thuật của phương pháp AST-CC

2.3. Kết luận chương 2

Kết thúc chương 2, luận văn đã trình bày được về tổng quan về sự thay đổi giữa các cây AST khi áp dụng các thủ thuật sao chép mã nguồn khác nhau, áp dụng các kỹ thuật LCS, TF-IDF, AST-CC trong việc so sánh đánh giá các mã nguồn. Trong chương tiếp theo luận văn tập trung xây dựng, thử nghiệm hệ thống đánh giá mức độ tương tự giữa các mã nguồn theo 3 kỹ thuật đã trình bày ở chương 2.

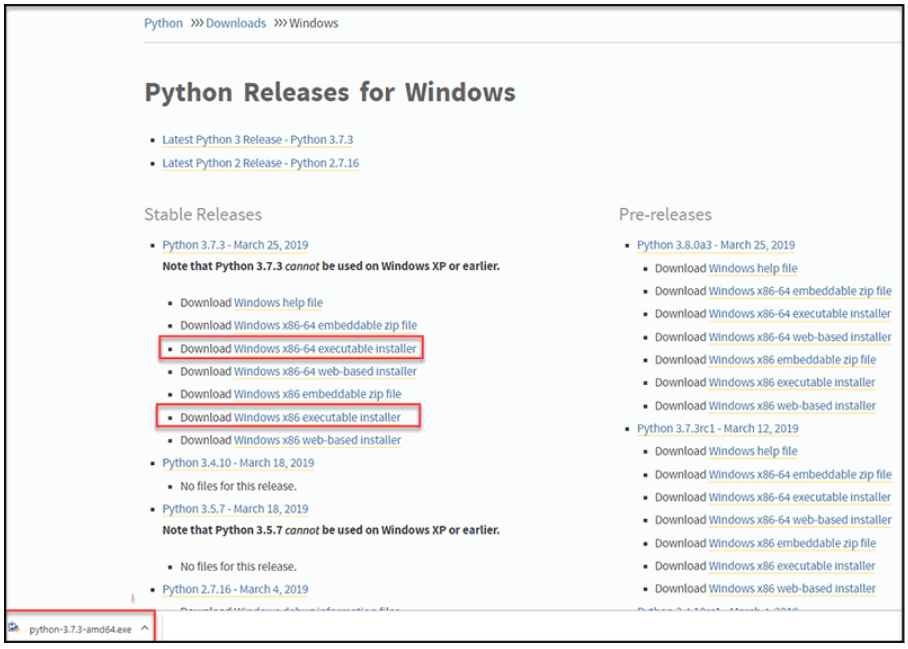
3 CHƯƠNG 3: THỰC NGHIỆM VÀ ĐÁNH GIÁ

3.1. Cài đặt hệ thống để so sánh độ tương tự giữa các mã nguồn

3.1.1 Cài đặt Python

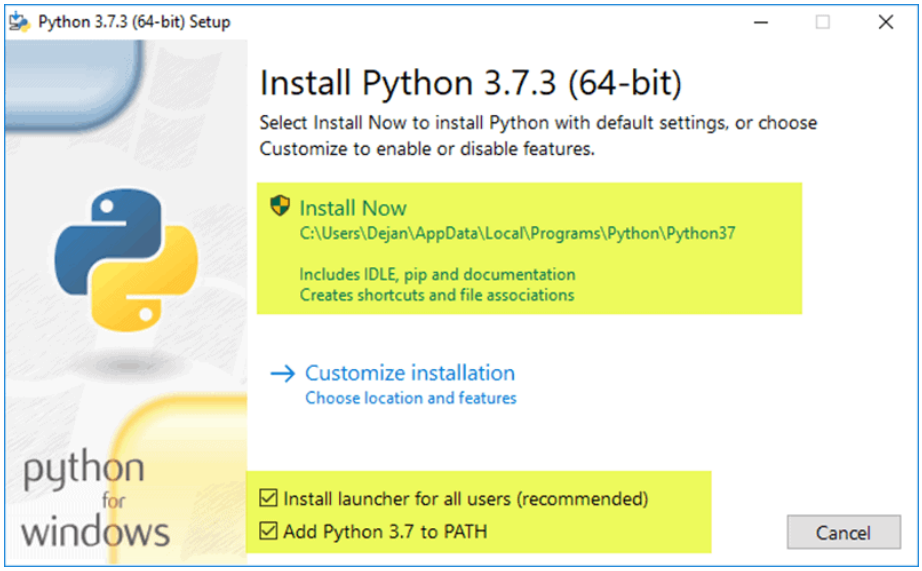
Bước 1: Tải về bộ cài đặt Python 3 cho hệ điều hành windows

Với đặc thù của bài toán, chúng ta chọn phiên bản Python được đánh giá ổn định là Python 3.7



Hình 3.1: Các gói cài đặt Python 3 dành cho Hệ điều hành Windows

Bước 2: Cài đặt Python 3 trên Hệ điều hành Windows



Hình 3.2: Các gói cài đặt Python 3 dành cho Hệ điều hành Windows

Bước 3: Kiểm tra phiên bản Python đã cài đặt thành công

Mở cửa sổ terminal và gõ dòng lệnh sau: `python --version`

Terminal trả về đúng phiên bản vừa cài đặt là thành công (ví dụ Python 3.7.3)

Trong trường hợp có lỗi, vui lòng kiểm tra lại đường dẫn PATH của Windows đã có sẵn đường dẫn đến thư mục cài đặt Python hay chưa.

```
$ python --version
Python 3.7.3
```

Bước 4: Kiểm tra thư viện pip (Trình quản lý các gói thư viện của Python) đã được cài đặt hay chưa. Trong trường hợp thành công trả về như sau:

```
$ pip --version
pip 22.0.4 from c:\users\admin\appdata\local\programs\python\python37\lib\site-
packages\pip (python 3.7)
```

3.1.2 Cài đặt thư viện Clang

Tài liệu hướng dẫn cài đặt thư viện Clang trên nền ngôn ngữ lập trình Python tại đây: https://pypi.org/project/clang/

Bước 1: Mở terminal trên máy cần cài đặt

Bước 2: Gõ lệnh cài đặt clang:

```
pip install clang
```

Theo dõi kết quả trên terminal trả về thành công:

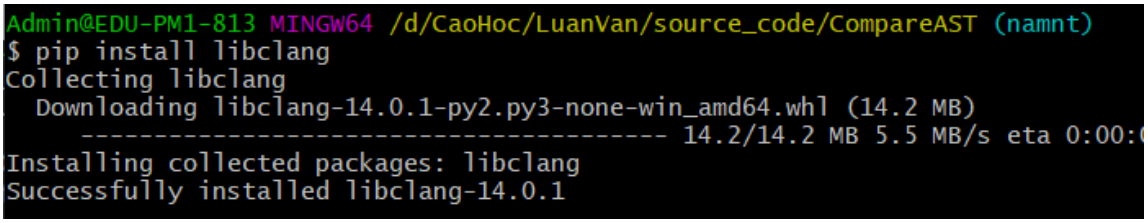
```
Admin@EDU-PM1-813 MINGW64 /d/CaoHoc/LuanVan/source_code/CompareAST (namnt)
$ pip install clang
Collecting clang
  Downloading clang-14.0-py3-none-any.whl (31 kB)
Installing collected packages: clang
Successfully installed clang-14.0
```

Hình 3.3: Kết quả khi chạy lệnh cài đặt thư viện clang

Bước 3: Gõ lệnh cài đặt libclang

```
pip install libclang
```

Theo dõi kết quả trên terminal trả về thành công:

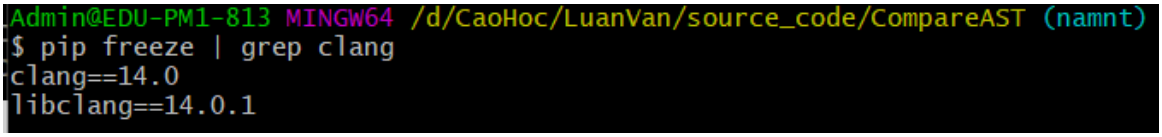


Hình 3.4: Kết quả khi chạy lệnh cài đặt thư viện libclang

Bước 4: Kiểm tra kết quả cài đặt:

```
pip freeze | grep clang
```

Theo dõi kết quả trên terminal trả về thành công:



Hình 3.5: Kết quả kiểm tra cài đặt thư viện libclang

3.2. Xây dựng công cụ so sánh mã nguồn với 3 thuật toán LCS, TF-IDF, AST-CC

Công cụ so sánh mã nguồn gồm các script thực hiện các nhiệm vụ độc lập như sau:

Bảng 3.1: Bảng danh sách các script thực hiện so sánh mức tương tự giữa các mã nguồn

STT	Tên file	Ý nghĩa
1	ast_viewer.py	Hiển thị cây AST của một mã nguồn trên terminal
2	reduceAST.py	Tiền xử lý AST của một mã nguồn trước khi so sánh
3	similarity.py	Script gọi hàm so sánh 2 mã nguồn và trả về mức độ tương tự (lựa chọn giữa 3 kỹ thuật LCS, TF-IDF, AST-CC)
4	lcs.py	Script thực hiện so sánh 2 trực tiếp 2 mã nguồn và trả về mức độ tương tự sử dụng phương pháp LCS
5	tf_idf.py	Script thực hiện so sánh 2 trực tiếp 2 mã nguồn và trả về mức độ tương tự sử dụng phương pháp TF-IDF
6	ast_cc.py	Script thực hiện so sánh 2 trực tiếp 2 mã nguồn và trả về mức độ tương tự sử dụng phương pháp AST-CC
7	compare_two_code.py	Giao diện cho phép so sánh trực tiếp 2 mã nguồn, hiển thị chi tiết mức độ tương tự, các phần mã nguồn được xác định tương tự (lựa chọn giữa 3 kỹ thuật LCS, TF-IDF, AST-CC)

8	main.py	Hàm chính chạy so sánh giữa 1 tập dữ liệu là N file mã nguồn trong thư mục test_data. Kết quả trả về là một bảng NxN, trong đó cho thấy mức độ tương tự giữa các cặp file mã nguồn.

3.2.1 Mã nguồn hiển thị cây AST

```
import clang.cindex
import argparse

function_calls = []  # List of AST node objects that are function calls
function_declarations = []  # List of AST node objects that are fucntion
declarations

tree_line_arr = []

# Traverse the AST tree
def trimClangNodeName(nodeName):
    ret = str(nodeName)
    ret = ret.split(".")[1]
    return ret

def parse_binary_op(cursor):
    # assert cursor.kind == clang.cindex.CursorKind.BINARY_OPERATOR
    children_list = [i for i in cursor.get_children()]
    assert len(children_list) == 2
    left_offset = len([i for i in children_list[0].get_tokens()])
    op = [i for i in cursor.get_tokens()][left_offset].spelling
    return op


def printASTNode(node, level, is_last_child):
    global tree_line_arr
    tree_line_str = ""
    for i in range(0, level-1):
        tree_line_str += '    '

    if is_last_child:
        prefix_str = '└──'
    else:
        prefix_str = '├──'

    if node.kind == clang.cindex.CursorKind.BINARY_OPERATOR or node.kind ==
clang.cindex.CursorKind.COMPOUND_ASSIGNMENT_OPERATOR:
        binaryOp = parse_binary_op(node)
        tree_line_str += (f'{prefix_str}{node.kind.name} {binaryOp}
{node.type.spelling} {node.type.kind} [line={node.location.line},
col={node.location.column}]')
    else:
        tree_line_str += (f'{prefix_str}{node.kind.name} {node.spelling}
{node.type.spelling} {node.type.kind} [line={node.location.line},
col={node.location.column}]')
    tree_line_arr.append(tree_line_str)
```

```
    # Xu ly ky tu
    idx = 0;
    tree_line_qty_of_char = len(tree_line_arr)
    for tmp_line in reversed(tree_line_arr):
        if idx == 0:
            idx += 1
            continue

        tmp_line_str = list(tmp_line)
        tmp_line_char_pos = 3 * (level - 1)
        tmp_line_char = tmp_line_str[tmp_line_char_pos];
        if(tmp_line_char != " " or tmp_line_char == "├" or tmp_line_char == "└"):
             break

        tmp_line_str[tmp_line_char_pos] = "│"
        tree_line_arr[tree_line_qty_of_char - 1 - idx] = "".join(tmp_line_str)
        idx += 1

def traverseAST(node, level, is_last_child):
    list_parent_level = []
    if node is not None:
        level = level + 1
        if level == 1 or (node.location.file and node.location.file.name ==
file_path):
            printASTNode(node, level, is_last_child)
        # Recurse for children of this node
        count_children = len(list(node.get_children()))
        j = 0
        for childNode in node.get_children():
            list_parent_level.append(level)
            j = j + 1
            traverseAST(childNode, level, j == count_children)
        level = level - 1

def previewAST(file_path):
    # Tell clang.cindex where libclang.dylib is
    # clang.cindex.Config.set_library_path("/usr/lib/llvm-6.0/lib/")
    index = clang.cindex.Index.create()
    # Generate AST from filepath passed in the command line
    tu = index.parse(file_path, ['-x', 'c++', '-std=c++11', '-
D__CODE_GENERATOR__'])

    root = tu.cursor  # Get the root of the AST
    traverseAST(root, 0, False)
    # print result
    print("\n".join(tree_line_arr))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input_file', type=str,
default='test_data_2/test_1.cpp', help="input file C++")
    args = parser.parse_args()
    file_path = args.input_file
    previewAST(file_path)
```

3.2.2 Mã nguồn so sánh 2 cây AST theo phương pháp LCS

```
from reduceAST import CustomNode, default_tree_size

def hashListClassify(node, size):
    hashList = [[] for _ in range(size + 1)]

    def traverseHash(sub_node):
        if sub_node is not None:
            n_node = default_tree_size(sub_node, CustomNode.get_children)
            if n_node > 0:
                hashList[n_node].append(sub_node)
            for childNode in CustomNode.get_children(sub_node):
                traverseHash(childNode)
    traverseHash(node)
    return hashList

def compare_from_parent(hashList_1, hashList_2, threshold=1):
    n = min(len(hashList_1), len(hashList_2))
    pair = []
    ls = []
    for n_sub_node in range(n - 1, threshold - 1, -1):
        L1 = hashList_1[n_sub_node]
        L2 = hashList_2[n_sub_node]
        L1 = sorted(L1, key=lambda x: x.hashnode)
        L2 = sorted(L2, key=lambda x: x.hashnode)
        i = 0
        j = 0
        while i < len(L1) and j < len(L2):
            P1 = CustomNode.get_parent(L1[i])
            P2 = CustomNode.get_parent(L2[j])
            if P1 and P1 in ls:
                ls.append(L1[i])
                i += 1
                continue
            elif P2 and P2 in ls:
                ls.append(L2[j])
                j += 1
                continue
            if L1[i].hashnode > L2[j].hashnode:
                j += 1
            elif L1[i].hashnode < L2[j].hashnode:
                i += 1
            else:
                pair.append([L1[i], L2[j]])
                ls.append(L1[i])
                ls.append(L2[j])
                i += 1
                j += 1
    return pair

def compare(hashList_1, hashList_2, threshold=1):
    n = min(len(hashList_1), len(hashList_2))
    pair = []
    for n_sub_node in range(threshold, n):
        L1 = hashList_1[n_sub_node]
        L2 = hashList_2[n_sub_node]
```

```
        L1 = sorted(L1, key=lambda x: x.hashnode)
        L2 = sorted(L2, key=lambda x: x.hashnode)
        i = 0
        j = 0
        while i < len(L1) and j < len(L2):
            P1 = CustomNode.get_parent(L1[i])
            P2 = CustomNode.get_parent(L2[j])
            if P1 and P2 and P1.hashnode == P2.hashnode:
                i += 1
                j += 1
                continue
            if L1[i].hashnode > L2[j].hashnode:
                j += 1
            elif L1[i].hashnode < L2[j].hashnode:
                i += 1
            else:
                pair.append([L1[i], L2[j]])
                i += 1
                j += 1

    return pair


def ast_cc_similarity(ast1, ast2):
    size_1 = default_tree_size(ast1, CustomNode.get_children)
    size_2 = default_tree_size(ast2, CustomNode.get_children)

    hashList1 = hashListClassify(ast1, size_1)
    hashList2 = hashListClassify(ast2, size_2)

    res = compare(hashList1, hashList2, threshold=2)
    total = 0
    for r in res:
        total += default_tree_size(r[0], CustomNode.get_children)

    return 2 * total / (size_1 + size_2)


def ast_cc_compare(ast1, ast2):
    size_1 = default_tree_size(ast1, CustomNode.get_children)
    size_2 = default_tree_size(ast2, CustomNode.get_children)

    hashList1 = hashListClassify(ast1, size_1)
    hashList2 = hashListClassify(ast2, size_2)

    res = compare_from_parent(hashList1, hashList2, threshold=2)
    total = 0
    if not res:
        print('There aren\'t common')
        return 0
    print(f'There are {len(res)} reports')
    print('---------AST1--------- | ---------AST2--------- | ---No. Node---')
    for r in res:
        n_node = default_tree_size(r[0], CustomNode.get_children)
        print('(%3d, %3d), (%3d, %3d) | (%3d, %3d), (%3d, %3d) |   %5d'
              % (r[0].start[0], r[0].start[1], r[0].end[0], r[0].end[1],
                 r[1].start[0], r[1].start[1], r[1].end[0], r[1].end[1],
```

```
                n_node))
        total += n_node

    return 2 * total / (size_1 + size_2)
```

3.2.3 Mã nguồn so sánh 2 cây AST theo phương pháp TF-IDF

```
# TF-IDF
from sklearn.metrics.pairwise import linear_kernel
from sklearn.feature_extraction.text import TfidfVectorizer

def identityFunction(file):
    return file

def tf_idf_similarity(seq1, seq2, ngram_range=(1, 2)):
    docs = [seq1, seq2]

    VOCAB_LIMIT = 2000  # Can be increased if efficency is not an issue
    vectorizer = TfidfVectorizer(
        analyzer='word',
        tokenizer=identityFunction,
        preprocessor=identityFunction,
        # Consider unigrams and bigrams only
        ngram_range=ngram_range,
        sublinear_tf=True,  # (1+log(tf)) instead of just tf
        max_features=VOCAB_LIMIT,
        encoding="utf-8",
        decode_error="ignore",
        stop_words=None,
        lowercase=False,
        norm="l2"  # Each row will be unit normalized
    )
    S = vectorizer.fit_transform(docs)

    tfm = linear_kernel(S, S)
    return tfm[0][1]
```

3.2.4 Mã nguồn so sánh 2 cây AST theo phương pháp AST-CC

```
from reduceAST import CustomNode, default_tree_size

def hashListClassify(node, size):
    hashList = [[] for _ in range(size + 1)]

    def traverseHash(sub_node):
        if sub_node is not None:
            n_node = default_tree_size(sub_node, CustomNode.get_children)
            if n_node > 0:
                hashList[n_node].append(sub_node)
            for childNode in CustomNode.get_children(sub_node):
                traverseHash(childNode)
    traverseHash(node)
    return hashList

def compare_from_parent(hashList_1, hashList_2, threshold=1):
    n = min(len(hashList_1), len(hashList_2))
    pair = []
    ls = []
```

```
    for n_sub_node in range(n - 1, threshold - 1, -1):
        L1 = hashList_1[n_sub_node]
        L2 = hashList_2[n_sub_node]
        L1 = sorted(L1, key=lambda x: x.hashnode)
        L2 = sorted(L2, key=lambda x: x.hashnode)
        i = 0
        j = 0
        while i < len(L1) and j < len(L2):
            P1 = CustomNode.get_parent(L1[i])
            P2 = CustomNode.get_parent(L2[j])
            if P1 and P1 in ls:
                ls.append(L1[i])
                i += 1
                continue
            elif P2 and P2 in ls:
                ls.append(L2[j])
                j += 1
                continue
            if L1[i].hashnode > L2[j].hashnode:
                j += 1
            elif L1[i].hashnode < L2[j].hashnode:
                i += 1
            else:
                pair.append([L1[i], L2[j]])
                ls.append(L1[i])
                ls.append(L2[j])
                i += 1
                j += 1
    return pair

def compare(hashList_1, hashList_2, threshold=1):
    n = min(len(hashList_1), len(hashList_2))
    pair = []
    for n_sub_node in range(threshold, n):
        L1 = hashList_1[n_sub_node]
        L2 = hashList_2[n_sub_node]
        L1 = sorted(L1, key=lambda x: x.hashnode)
        L2 = sorted(L2, key=lambda x: x.hashnode)
        i = 0
        j = 0
        while i < len(L1) and j < len(L2):
            P1 = CustomNode.get_parent(L1[i])
            P2 = CustomNode.get_parent(L2[j])
            if P1 and P2 and P1.hashnode == P2.hashnode:
                i += 1
                j += 1
                continue
            if L1[i].hashnode > L2[j].hashnode:
                j += 1
            elif L1[i].hashnode < L2[j].hashnode:
                i += 1
            else:
                pair.append([L1[i], L2[j]])
                i += 1
                j += 1
    return pair
```

```
def ast_cc_similarity(ast1, ast2):
    size_1 = default_tree_size(ast1, CustomNode.get_children)
    size_2 = default_tree_size(ast2, CustomNode.get_children)

    hashList1 = hashListClassify(ast1, size_1)
    hashList2 = hashListClassify(ast2, size_2)

    res = compare(hashList1, hashList2, threshold=2)
    total = 0
    for r in res:
        total += default_tree_size(r[0], CustomNode.get_children)

    return 2 * total / (size_1 + size_2)

def ast_cc_compare(ast1, ast2):
    size_1 = default_tree_size(ast1, CustomNode.get_children)
    size_2 = default_tree_size(ast2, CustomNode.get_children)

    hashList1 = hashListClassify(ast1, size_1)
    hashList2 = hashListClassify(ast2, size_2)

    res = compare_from_parent(hashList1, hashList2, threshold=2)
    total = 0
    if not res:
        print('There aren\'t common')
        return 0
    print(f'There are {len(res)} reports')
    print('---------AST1--------- | ---------AST2--------- | ---No. Node---')
    for r in res:
        n_node = default_tree_size(r[0], CustomNode.get_children)
        print('(%3d, %3d), (%3d, %3d) | (%3d, %3d), (%3d, %3d) |   %5d'
              % (r[0].start[0], r[0].start[1], r[0].end[0], r[0].end[1],
                 r[1].start[0], r[1].start[1], r[1].end[0], r[1].end[1],
                 n_node))
        total += n_node

    return 2 * total / (size_1 + size_2)
```

3.3. Thực nghiệm hệ thống với dữ liệu đầu vào đã gán nhãn (phân loại)

Dữ liệu đầu vào đã phân loại gồm 16 mã nguồn gồm 10 mã nguồn đã được gán nhãn (phân loại) theo thủ thuật sao chép và 6 mã nguồn không liên quan để đảm bảo tránh trường hợp dương tính giả (False Positive) [10]

Name	Date modified	Type	Size
00-original.cpp	3/27/2022 4:06 PM	C++ source file	3 KB
01-variables-and-types-changed.cpp	3/27/2022 4:06 PM	C++ source file	4 KB
02-order-changed.cpp	4/16/2022 4:34 PM	C++ source file	3 KB
03-dead-code.cpp	3/27/2022 4:06 PM	C++ source file	3 KB
04-moderately-plagiarized.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
05-heavily-plagiarized.cpp	3/27/2022 4:06 PM	C++ source file	3 KB
06-different-approach.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
07-another-approach.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
08-somewhat-related-dfs.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
09-somewhat-related-bfs.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
10-unrelated-quicksort.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
11-unrelated-mergesort.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
12-unrelated-fibonacci-recursive.cpp	3/27/2022 4:06 PM	C++ source file	1 KB
13-unrelated-fibonacci-matrix.cpp	3/27/2022 4:06 PM	C++ source file	2 KB
14-unrelated-fibonacci-memoization.cpp	3/27/2022 4:06 PM	C++ source file	1 KB
15-unrelated-fibonacci-dynamic.cpp	3/27/2022 4:06 PM	C++ source file	1 KB

Hình 3.6: Bộ mã nguồn đã gán nhãn

3.3.1 Sử dụng phương pháp LCS

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0.76	0.86	0.61	0.91	0.26	0.31	0.39	0.38	0.43	0.4	0.22	0.27	0.26	0.25
2	1	1	0.76	0.86	0.61	0.91	0.26	0.31	0.39	0.38	0.43	0.4	0.22	0.27	0.26	0.25
3	0.76	0.76	1	0.67	0.52	0.74	0.26	0.3	0.39	0.35	0.41	0.4	0.2	0.28	0.27	0.25
4	0.86	0.86	0.67	1	0.55	0.78	0.21	0.26	0.34	0.33	0.39	0.41	0.27	0.26	0.24	0.21
5	0.61	0.61	0.52	0.55	1	0.61	0.26	0.34	0.35	0.32	0.43	0.4	0.22	0.22	0.29	0.31
6	0.91	0.91	0.74	0.78	0.61	1	0.3	0.34	0.42	0.39	0.45	0.41	0.22	0.26	0.28	0.28
7	0.26	0.26	0.26	0.21	0.26	0.3	1	0.53	0.34	0.39	0.23	0.17	0.39	0.13	0.26	0.4
8	0.31	0.31	0.3	0.26	0.34	0.34	0.53	1	0.38	0.35	0.29	0.25	0.37	0.18	0.33	0.34
9	0.39	0.39	0.39	0.34	0.35	0.42	0.34	0.38	1	0.82	0.37	0.33	0.27	0.23	0.3	0.35
10	0.38	0.38	0.35	0.33	0.32	0.39	0.39	0.35	0.82	1	0.34	0.3	0.25	0.23	0.29	0.34
11	0.43	0.43	0.41	0.39	0.43	0.45	0.23	0.29	0.37	0.34	1	0.58	0.31	0.26	0.36	0.32
12	0.4	0.4	0.4	0.41	0.4	0.41	0.17	0.25	0.33	0.3	0.58	1	0.2	0.28	0.3	0.31
13	0.22	0.22	0.2	0.27	0.22	0.22	0.39	0.37	0.27	0.25	0.31	0.2	1	0.2	0.41	0.4
14	0.27	0.27	0.28	0.26	0.22	0.26	0.13	0.18	0.23	0.23	0.26	0.28	0.2	1	0.29	0.23
15	0.26	0.26	0.27	0.24	0.29	0.28	0.26	0.33	0.3	0.29	0.36	0.3	0.41	0.29	1	0.51
16	0.25	0.25	0.25	0.21	0.31	0.28	0.4	0.34	0.35	0.34	0.32	0.31	0.4	0.23	0.51	1

Hình 3.7: Kết quả phương pháp LCS so sánh giữa 16 mã nguồn đã gán nhãn

3.3.2 *Sử dụng phương pháp TF-IDF*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0.94	0.87	0.69	0.97	0.38	0.32	0.43	0.42	0.42	0.45	0.23	0.3	0.22	0.26
2	1	1	0.94	0.87	0.69	0.97	0.38	0.32	0.43	0.42	0.42	0.45	0.23	0.3	0.22	0.26
3	0.94	0.94	1	0.85	0.68	0.92	0.38	0.33	0.44	0.42	0.42	0.44	0.23	0.3	0.22	0.25
4	0.87	0.87	0.85	1	0.65	0.85	0.37	0.32	0.42	0.39	0.53	0.47	0.43	0.33	0.31	0.3
5	0.69	0.69	0.68	0.65	1	0.68	0.37	0.33	0.42	0.37	0.34	0.42	0.23	0.23	0.21	0.34
6	0.97	0.97	0.92	0.85	0.68	1	0.39	0.33	0.43	0.41	0.41	0.45	0.22	0.31	0.22	0.26
7	0.38	0.38	0.38	0.37	0.37	0.39	1	0.63	0.34	0.35	0.26	0.3	0.32	0.19	0.19	0.25
8	0.32	0.32	0.33	0.32	0.33	0.33	0.63	1	0.29	0.29	0.26	0.31	0.29	0.17	0.19	0.26
9	0.43	0.43	0.44	0.42	0.42	0.43	0.34	0.29	1	0.8	0.33	0.34	0.2	0.26	0.19	0.29
10	0.42	0.42	0.42	0.39	0.37	0.41	0.35	0.29	0.8	1	0.3	0.31	0.23	0.27	0.19	0.31
11	0.42	0.42	0.42	0.53	0.34	0.41	0.26	0.26	0.33	0.3	1	0.66	0.36	0.36	0.37	0.33
12	0.45	0.45	0.44	0.47	0.42	0.45	0.3	0.31	0.34	0.31	0.66	1	0.34	0.37	0.34	0.45
13	0.23	0.23	0.23	0.43	0.23	0.22	0.32	0.29	0.2	0.23	0.36	0.34	1	0.23	0.37	0.29
14	0.3	0.3	0.3	0.33	0.23	0.31	0.19	0.17	0.26	0.27	0.36	0.37	0.23	1	0.39	0.39
15	0.22	0.22	0.22	0.31	0.21	0.22	0.19	0.19	0.19	0.19	0.37	0.34	0.37	0.39	1	0.41
16	0.26	0.26	0.25	0.3	0.34	0.26	0.25	0.26	0.29	0.31	0.33	0.45	0.29	0.39	0.41	1

Hình 3.8: Kết quả phương pháp TF-IDF so sánh giữa 16 mã nguồn đã gán nhãn

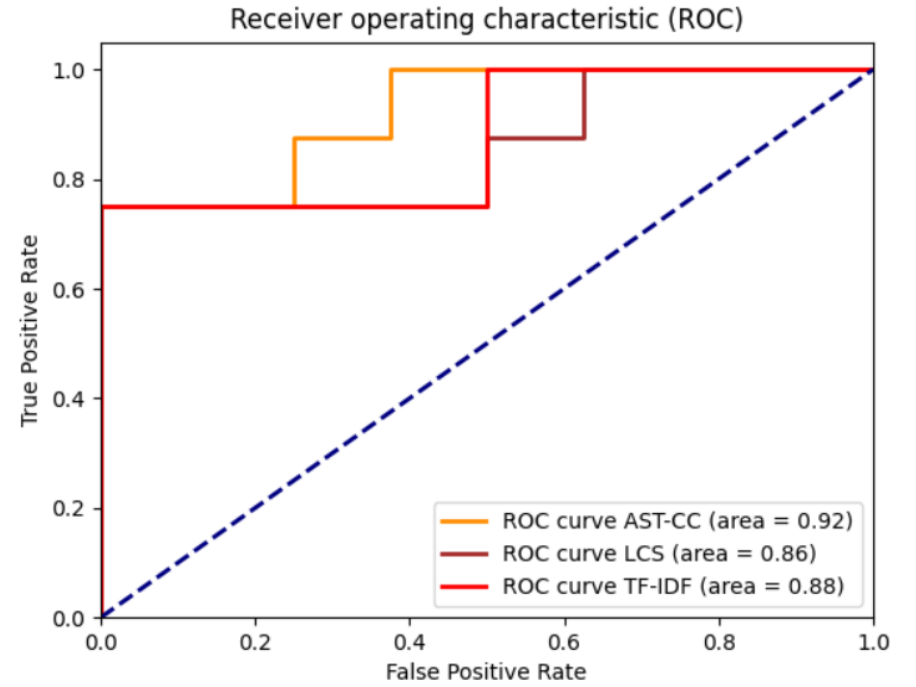
3.3.3 *Sử dụng phương pháp AST-CC*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	1	0.82	0.29	0.44	0.1	0.13	0.12	0.1	0.18	0.22	0.08	0.05	0.05	0.05
2	1	1	1	0.82	0.29	0.44	0.1	0.13	0.12	0.1	0.18	0.22	0.08	0.05	0.05	0.05
3	1	1	1	0.82	0.29	0.44	0.1	0.13	0.12	0.1	0.18	0.22	0.08	0.05	0.05	0.05
4	0.82	0.82	0.82	1	0.23	0.37	0.08	0.1	0.1	0.08	0.15	0.24	0.21	0.08	0.09	0.1
5	0.29	0.29	0.29	0.23	1	0.28	0.11	0.14	0.13	0.11	0.17	0.21	0.09	0.03	0.03	0.08
6	0.44	0.44	0.44	0.37	0.28	1	0.14	0.14	0.13	0.11	0.17	0.22	0.09	0.06	0.06	0.06
7	0.1	0.1	0.1	0.08	0.11	0.14	1	0.23	0.13	0.11	0.08	0.09	0.23	0.05	0.07	0.14
8	0.13	0.13	0.13	0.1	0.14	0.14	0.23	1	0.13	0.09	0.12	0.12	0.17	0.04	0.06	0.1
9	0.12	0.12	0.12	0.1	0.13	0.13	0.13	0.13	1	0.33	0.08	0.09	0.09	0.04	0.05	0.08
10	0.1	0.1	0.1	0.08	0.11	0.11	0.11	0.09	0.33	1	0.04	0.04	0.1	0.04	0.05	0.09
11	0.18	0.18	0.18	0.15	0.17	0.17	0.08	0.12	0.08	0.04	1	0.31	0.14	0.07	0.16	0.16
12	0.22	0.22	0.22	0.24	0.21	0.22	0.09	0.12	0.09	0.04	0.31	1	0.09	0.06	0.09	0.14
13	0.08	0.08	0.08	0.21	0.09	0.09	0.23	0.17	0.09	0.1	0.14	0.09	1	0.1	0.2	0.24
14	0.05	0.05	0.05	0.08	0.03	0.06	0.05	0.04	0.04	0.04	0.07	0.06	0.1	1	0.05	0.12
15	0.05	0.05	0.05	0.09	0.03	0.06	0.07	0.06	0.05	0.05	0.16	0.09	0.2	0.05	1	0.15
16	0.05	0.05	0.05	0.1	0.08	0.06	0.14	0.1	0.08	0.09	0.16	0.14	0.24	0.12	0.15	1

Hình 3.9: Kết quả phương pháp AST-CC so sánh giữa 16 mã nguồn đã gán nhãn

3.3.4 Nhận xét, đánh giá

Dựa trên kết quả dữ liệu đã gán nhãn đầu vào, đường cong ROC được vẽ bởi 3 phương pháp LCS, TF-IDF và AST-CC được biểu diễn như sau:



Hình 3.10: Kết quả phương pháp LCS so sánh giữa mã nguồn chưa phân loại

Đồ thị trên không gian ROC cho thấy theo lý thuyết, đường ROC tạo bởi AST-CC có độ chính xác cao nhất. Tuy nhiên do dữ liệu thực nghiệm dược gán nhãn còn quá ít, nên đánh giá này mang tính chủ quan và cần được kiểm thử với tập dữ liệu lớn hơn.

Kết quả đánh giá từ phần thực nghiệm từ dữ liệu đã phân loại (đã gán nhãn) cho thấy một số nhận xét như sau:

- Với các thủ thuật thông thường như thay đổi format mã nguồn, thay đổi tên hàm, tên biến, thay đổi kiểu biến. Cả 3 kỹ thuật hoạt động tốt, trả về độ tương tự cao
- Với thủ thuật thay đổi thứ tự mã nguồn, bổ sung mã nguồn dư thừa (dead-code)
    - LCS không còn nhạy, trả về độ tương tự thấp (âm tính giả).
    - TF-IDF và ASC-CC vẫn thể hiện độ nhạy cao, tính đáp ứng tương đối tốt, khi trả về độ tương tự cao (trên 0,8)
- Với việc triển khai mã nguồn theo hướng tiếp cận mới:

- LCS và TF-IDF trả về độ tương tự cao
- AST-CC trong khi đó trả về độ tương tự thấp

- Khi so sánh với nhóm mã nguồn hoàn toàn không liên quan từ 10 cho đến 16:
  - LCS và TF-IDF cho thấy khả năng gây ra dương tính giả cao.
  - AST-CC vẫn hoạt động tốt khi cho mức độ tương tự thấp.

3.4. Thực nghiệm hệ thống với dữ liệu đầu vào chưa phân loại

Dữ liệu đầu vào chưa phân loại gồm 49 mã nguồn là các bài làm của sinh viên trên hệ thống *code.ptit.edu.vn* cho bài tập sau:

Một xâu nhị phân độ dài *n* được gọi là thuận nghịch hay đối xứng nếu đảo ngược xâu nhị phân đó ta vẫn nhận được chính nó. Cho số tự nhiên *n* (n nhập từ bàn phím). Hãy viết chương trình liệt kê tất cả các xâu nhị phân thuận nghịch có độ dài *n*. Hai phần tử khác nhau của xâu thuận nghịch được ghi cách nhau một khoảng trống. Ví dụ với n = 4 ta tìm được 4 xâu nhị phân thuận nghịch như dưới đây.

0 0 0 0
0 1 1 0
1 0 0 1
1 1 1 1

Ví dụ

Input	Output
4	0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 1

Bài tập thực hành Sinh nhị phân trên code.ptit.edu.vn

Một số bài làm của các sinh viên như sau:

```
#include<stdio.h>

int nextBitString(int a[], int n){
    int k;
    for (k = n; k >= 1; k--){
        if (a[k] == 0) break;
    }
    for (int i = k; i <= n; i++){
        a[i] = 1 - a[i];
    }
    for (k = n; k >= 1; k--){
        if (a[k] == 0) break;
    }
    if (k == 0) return 1;
    else return 0;
}

void printfBitString(int a[], int n){
    for (int i = 1; i <= n; i++) printf("%d ", a[i]);
    printf("\n");
}

int thng(int a[], int n){
      for (int i = 1; i <= n/2 + 1; i++){
            if(a[i] != a[n - i + 1]) return 0;
      }
      return 1;
}

int main(){
    int n;
    scanf("%d", &n);
    int a[n];
    for (int i = 1; i <= n; i++) a[i] = 0;
    int last = 0;
    while (!last){
      if(thng(a, n)){
             printfBitString(a, n);
            }
            last = nextBitString(a, n);
    }
    printfBitString(a, n);
    return 0;
}
```

Bài làm số 1

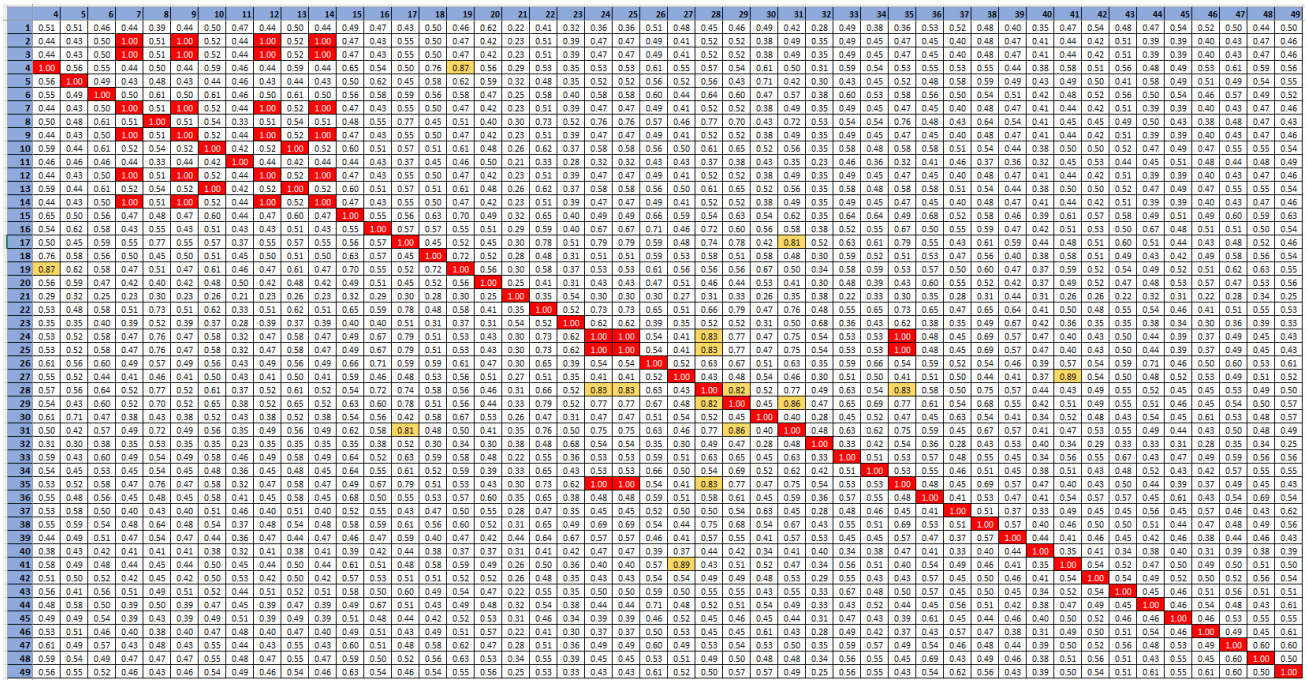
```
#include<stdio.h>

int nextBitString(int a[], int n){
    int k;
    for (k = n; k >= 1; k--){
        if (a[k] == 0) break;
    }
    for (int i = k; i <= n; i++){
        a[i] = 1 - a[i];
    }
    for (k = n; k >= 1; k--){
        if (a[k] == 0) break;
    }
    if (k == 0) return 1;
    else return 0;
}

void printfBitString(int a[], int n){
    for (int i = 1; i <= n; i++) printf("%d ", a[i]);
    printf("\n");
}

int thng(int a[], int n){
      for (int i = 1; i <= n/2 + 1; i++){
            if(a[i] != a[n - i + 1]) return 0;
      }
      return 1;
}

int main(){
    int n;
    scanf("%d", &n);
    int a[n];
    for (int i = 1; i <= n; i++) a[i] = 0;
    int last = 0;
    while (!last){
      if(thng(a, n)){
              printfBitString(a, n);
            }
            last = nextBitString(a, n);
    }
    printfBitString(a, n);
    return 0;
}
```

Bài làm của số 4

```
#include<stdio.h>
#include<string.h>
int t,k,i,j,ok;
int arr[1005];
void in(){
      for(i=1;i<=k;i++)
      printf("%d ",arr[i]);
      printf("\n");
}
int thuannghich(){
      int d=1,c=k;
      while(d<c){
            if(arr[d]!=arr[c])
            return 0;
            d++;  c--;
      }
      return 1;
}
void sinh(){
      int i=k;
      while(i>0 && arr[i]==1) i--;
            if(i<=0){
                  ok=1;
            }
            else{
                  arr[i]=1;
                  for(j=i+1;j<=k;j++){
                        arr[j]=0;
                  }
            }
}
int main(){
      scanf("%d",&k);
            ok=0;
            for(i=1;i<=k;i++)
            arr[i]=0;
            while(!ok){
            if(thuannghich()==1)
            in();
            sinh();
            }
}
```

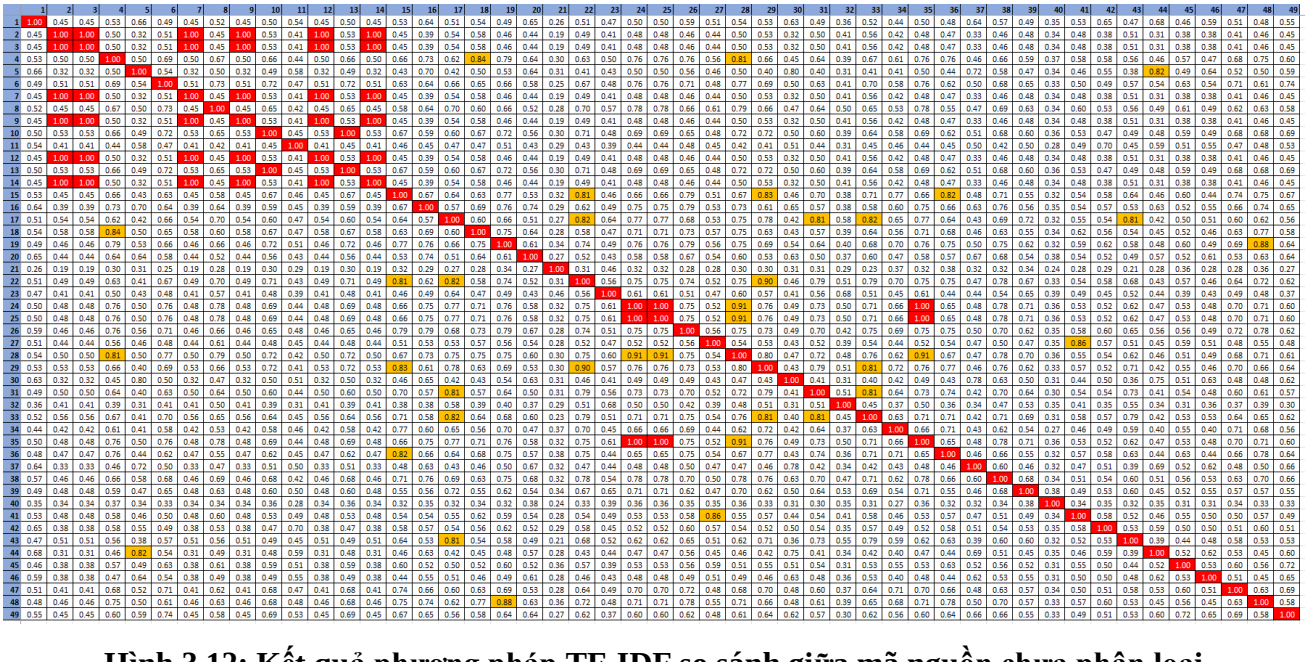
Bài làm của số 18

3.4.1 *Sử dụng phương pháp LCS*



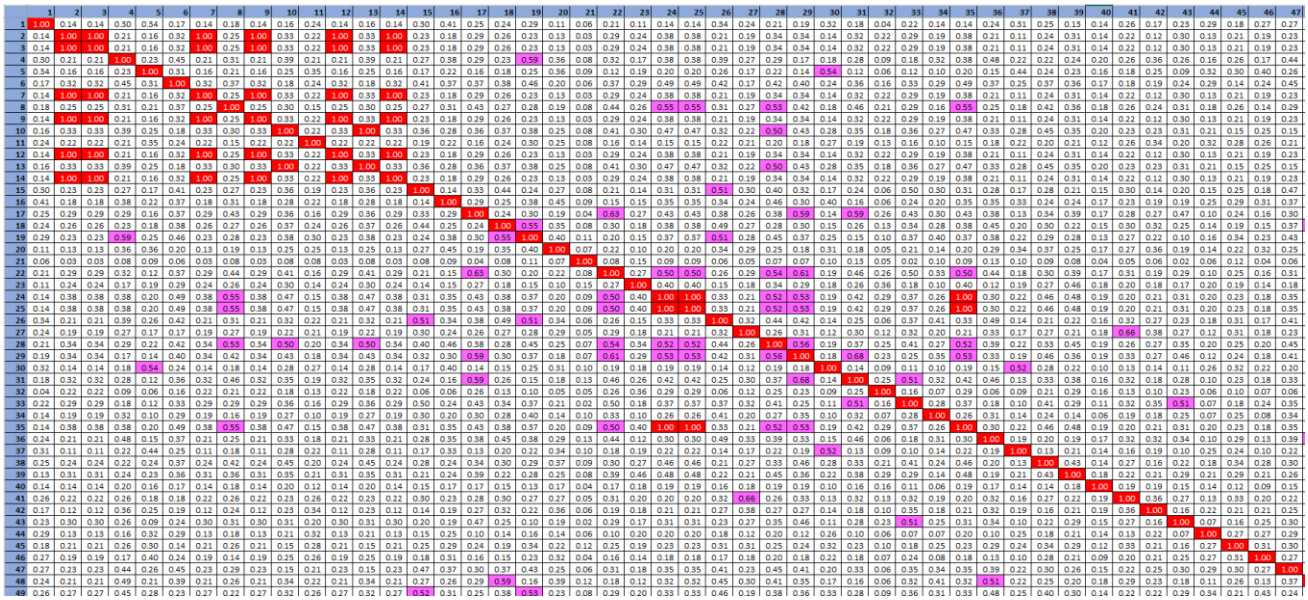
Hình 3.11: Kết quả phương pháp LCS so sánh giữa mã nguồn chưa phân loại

3.4.2 *Sử dụng phương pháp TF-IDF*



Hình 3.12: Kết quả phương pháp TF-IDF so sánh giữa mã nguồn chưa phân loại

3.4.3 *Sử dụng phương pháp AST-CC*



Hình 3.13: Kết quả phương pháp AST-CC so sánh giữa mã nguồn chưa phân loại

3.4.4 *Nhận xét, đánh giá*

Kết hợp kết quả đánh giá từ phần thực nghiệm từ dữ liệu chưa phân loại và việc kiểm tra nội dung các cặp file mã nguồn có độ tương tự cao (từ 0.8 trở lên) cho thấy như sau:

- 3 kỹ thuật trả về tốt với các thủ thuật sao chép thông thường mà sinh viên hay áp dụng.
- LCS và TF-IDF có xác suất trả về độ dương tính giả cao
- AST-CC ít bị nhạy bởi việc trùng lặp các đoạn mã nguồn vô tình giữa các sinh viên. AST-CC có xác suất cho dương tính giả thấp. Thể hiện ở việc không có cặp mã nguồn nào cho khoảng trống mức độ tương tự từ 0.8 đến 1.0.

Kết quả này là phù hợp khi LCS và TF-IDF chủ yếu dựa vào mức độ trùng lặp giữa 2 chuỗi biểu diễn 2 cây AST mà quên đi sự tương đồng ở cấp độ cấu trúc giữa 2 mã nguồn. Nhược điểm này được giải quyết tốt hơn đối với kỹ thuật AST-CC. Tuy nhiên AST-CC lại cho thấy rất nhạy cảm với việc thêm các mã nguồn dư thừa. Để giải quyết điều này, ở khâu tiền xử lý cây AST cần làm tốt hơn bằng cách loại bỏ các đoạn mã nguồn dư thừa.

Nói chung 3 kỹ thuật cho thấy các ưu điểm và nhược điểm khác nhau, đối với bài toán tính toán mức độ trùng lặp giữa các bài tập cho sinh viên thực hiện trên hệ thống *code.ptit.edu.vn*, việc cần hạn chế dương tính giả là phù hợp do tính chất nhạy cảm của việc

đánh giá. Do đó đề xuất áp dụng kỹ thuật AST-CC để đánh giá mức độ tương đồng giữa các giải thuật của sinh viên nộp trên hệ thống, cụ thể như sau:

Hình 3.14: Đề xuất áp dụng AST-CC trên hệ thống code.ptit.edu.vn

3.5. Kết luận chương 3

Kết thúc chương 3, luận văn đã thực hiện được một số nội dung sau:

- Cài đặt Python và cài đặt thư viện Clang
- Viết mã nguồn sử dụng các kỹ thuật LCS, TF-IDF, AST-CC
- Thực nghiệm trên dữ liệu đầu vào đã gãn nhán
- Thực nghiệm trên dữ liệu đầu vào chưa gãn nhán
- So sánh - đánh giá kết quả giữa các kỹ thuật
- Đề xuất phương án phù hợp để áp dụng trên hệ thống *code.ptit.edu.vn*

KẾT LUẬN

Luận văn này đã nghiên cứu, thực hiện và đạt được một số kết quả như sau:

- Nghiên cứu tổng quan về vấn đề sao chép mã nguồn, trình bày các thủ thuật sao chép mã nguồn phổ biến, các phương pháp đánh giá mức độ tương tự giữa các mã nguồn.

- Nghiên cứu tổng quan về vấn đề cây cú pháp trừu tượng AST và ứng dụng thư viện CLang kết hợp với ngôn ngữ lập trình Python để phân tích mã nguồn viết trên ngôn ngữ C/C++.

- Nghiên cứu và đánh giá những điểm khác và giống nhau giữa các cây AST trên mã nguồn gốc và mã nguồn cần đánh giá tương ứng với các thủ thuật sao chép khác nhau.

- Nghiên cứu và đề xuất phương pháp tiền xử lý cây AST để giảm độ phức tạp, tăng hiệu năng trước khi thực hiện so sánh mã nguồn.

- Nghiên cứu và ứng dụng phương pháp tìm chuỗi con chung dài nhất (LCS) vào việc so sánh độ tương tự giữa hai mã nguồn.

- Nghiên cứu và ứng dụng phương pháp TF-IDF vào việc so sánh độ tương tự giữa hai mã nguồn.

- Nghiên cứu và ứng dụng kỹ thuật AST-CC vào việc so sánh độ tương tự giữa hai mã nguồn.

- Nghiên cứu, xây dựng, thử nghiệm và đánh giá khi áp dụng các kỹ thuật LCS, TF-IDF, AST-CC đối với dữ liệu trích xuất từ bài làm của các sinh viên trên hệ thống *code.ptit.edu.vn*;

Trong tương lai, luận văn có thể được tiếp tục nghiên cứu theo hướng xử lý loại bỏ các đoạn mã nguồn dư thừa trên cây AST trước khi thực hiện việc so sánh để tăng độ chính xác, có thể mở rộng nghiên cứu theo hướng biểu diễn mã nguồn theo cấu trúc đồ thị phụ thuộc [9]. Ngoài ra có thể xây dựng công cụ đóng gói và cho phép tích hợp trực tiếp trên hệ thống code.ptit.edu.vn và các hệ thống Lab thực hành để đưa ra khuyến nghị, cảnh báo cho Giảng viên và Sinh viên khi sử dụng để ôn tập, kiểm tra.

DANH MỤC CÁC TÀI LIỆU THAM KHẢO

Tiếng Việt

[1] Nguyễn Ngọc Rạng, "Ứng dụng đường cong ROC trong nghiên cứu y học", p. 6, 2017

Tiếng Anh

[2] T. Fawcett, "ROC Graphs: Notes and Practical Considerations for Researchers", p. 6, 2004

[3] R. Kohavi and F. Provost, "Glossary of terms, Machine Learning", p. 8, 1998

[4] Doug Gregor. "Libclang: Thinking Beyond the Compiler", p. 16, 2010

[5] T. Cormen, C. Leiserson, R. Rivest and C. Stein, "Introduction to Algorithms, The MIT Press, 2nd edition", p. 35, 2001

[6] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs", p. 38, 1999

[7] Oscar Karnalim, "TF-IDF Inspired Detection for Cross-Language Source Code Plagiarism and Collusion", p. 39, 2020

[8] Jingling Zhao and Kunfeng Xia and Yilun Fu and Baojiang Cui, "An AST-based Code Plagiarism Detection Algorithm", p. 44, 2015

[9] Thomas Schaper, "Using Program Dependency Graphs for plagiarism detection in Python", p. 67, 2018

Website

[10] https://github.com/scriptographers/UnPlag/tree/master/models/cpp/demo-data

[11] https://vi.wikipedia.org/wiki/ Cây_cú_pháp_trừu_tượng

[12] https://www.llvm.org/

[13] https://clang.llvm.org/

[14] https://pypi.org/project/libclang/

[15] https://www.sciencedirect.com/topics/computer-science/cosine-similarity