ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC VÀ KỸ THUẬT THÔNG TIN

HUỲNH THANH SANG

LÊ VŨ KHÁNH THẢO

NGUYỄN ĐÌNH HUY

BÁO CÁO ĐỒ ÁN HỌC PHẦN

NHẬP MÔN BẢO ĐẢM VÀ AN NINH THÔNG TIN

TÌM HIỂU PHÁT HIỆN LỖI PHẦN MỀM BẰNG ĐỒ THỊ

RESEARCH ON SOFTWARE BUG DETECTION USING GRAPH

TP. HỒ CHÍ MINH, 2025

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC VÀ KỸ THUẬT THÔNG TIN

HUỲNH THANH SANG – 23521341

LÊ VŨ KHÁNH THẢO – 23521469

NGUYỄN ĐÌNH HUY – 23520626

BÁO CÁO ĐỒ ÁN HỌC PHẦN

NHẬP MÔN BẢO ĐẢM VÀ AN NINH THÔNG TIN

TÌM HIỂU PHÁT HIỆN LỖI PHẦN MỀM BẰNG ĐỒ THỊ

RESEARCH ON SOFTWARE BUG DETECTION USING GRAPH

GIẢNG VIÊN HƯỚNG DẪN

NGUYỄN TẤN CẦM

TP. HỒ CHÍ MINH, 2025

THÔNG TIN BÁO CÁO ĐỒ ÁN HỌC PHẦN

Ngày báo cáo đồ án học phần được xác định là ngày 20 tháng 5 năm 2025 theo quyết định của giảng viên học phần IE105 - Nhập môn bảo đảm và an ninh thông tin lớp IE105.P21.

LỜI CẢM ƠN

Nhóm chúng em xin bày tỏ lòng biết ơn sâu sắc đến thầy Nguyễn Tấn Cầm, giảng viên học phần IE105 - "Nhập môn bảo đảm và an ninh thông tin", lớp IE105.P21 thuộc khoa Khoa học và Kỹ thuật thông tin. Nhờ sự hướng dẫn của thầy, chúng em đã có được những kiến thức và kỹ năng nền tảng để thực hiện đề tài "Tìm hiểu phát hiện lỗi phần mềm bằng đồ thị". Mặc dù vậy, do còn hạn chế về kinh nghiệm và kiến thức chuyên môn, nhóm em nhận thấy vẫn còn nhiều thiếu sót trong quá trình nghiên cứu, đánh giá và trình bày đề tài. Chúng em rất mong nhận được sự góp ý, chỉ dẫn từ quý thầy cô bộ môn để đề tài này được hoàn thiện hơn nữa. Xin chân thành cảm ơn.

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

TÓM TẮT ĐỀ TÀI ĐỒ ÁN

Trong quá trình phát triển phần mềm, lập trình viên thường phải đối mặt với các lỗi - *bug* khác nhau. Các lỗi này đa dạng từ logic phát triển đơn giản đến xung đột thư viện sử dụng, hay phức tạp hơn là các lỗi ẩn bên trong quá trình hoạt động nhưng khó phát hiện. Một trong số các lỗi ẩn này có thể kể đến như các trường hợp dữ liệu đặc biệt mà logic hoạt động không thể nắm bắt - *handle* được, lỗi liên quan đến tắc nghẽn luồng hoạt động - *deadlock*, lỗi xử lý biểu thức toán học, … .

Đa dạng phương pháp có thể được sử dụng như phân tích mã nguồn tĩnh, phân tích mã nguồn động, suy luận bất biến, ...Trong đó, quá trình phân tích mã nguồn tĩnh là phương pháp thông dụng được cả lập trình viên và kiểm thử viên, điều này có thể được lý giải bởi vì việc phân tích mã nguồn tĩnh chỉ cần thực hiện kiểm tra logic mã nguồn thủ công mà không cần kiểm chứng thực thi. Điều này giúp cho lập trình viên và kiểm thử viên có thể kiểm tra được phần lớn lỗi thông dụng như NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException, ... Ngoài phân tích tĩnh, việc phân tích mã nguồn động bằng thực thi trình biên dịch cũng giúp tiếp tục loại bỏ các lỗi khác. Tuy nhiên, có một số lỗi lại không thể xác định bằng hai phương pháp kể trên, đặc biệt chính là các lỗi liên quan đến lời gọi cặp hàm, lỗi rò rỉ bộ nhớ, lỗi liên quan đến môi trường, ...

Do đó, về tổng quan nhìn nhận bối cảnh kỹ thuật hiện tại, lỗi phần mềm không chỉ xuất hiện đa dạng mà cũng có số lượng công cụ phong phú để phát hiện, điều này dẫn tới việc không thể phán định một phương pháp bất kỳ có khả năng phát hiện toàn bộ lỗi trong một mã nguồn chương trình. Vì vậy, nhóm tác giả đối với đề tài này quyết định lựa chọn áp dụng thử nghiệm các công cụ hỗ trợ kết hợp phương pháp phân tích đồ thị lời gọi kèm suy luận bất biến nhằm xác định và cục bộ hóa phạm vi của các lỗi bao gồm lỗi phát hiện bởi trình biên dịch, lỗi bộ nhớ, lỗi xử lý đa luồng, lỗi tranh chấp dữ liệu và lỗi vi phạm bất biến cặp hàm.

Chương 1. GIỚI THIỆU

1.1. Tổng quan

Chương này sẽ giới thiệu các phần thông tin cơ bản của đề tài, làm cơ sở để xây dựng bối cảnh, vấn đề và động lực của phương pháp.

1.1.1. Tầm quan trọng của việc phát hiện lỗi

Trong bối cảnh xây dựng và phát triển phần mềm, việc phát hiện các lỗi kỹ thuật của phần mềm là chuỗi các bước cần thiết để nhanh chóng khắc phục các gián đoạn hoạt động của phần mềm. Điều này rõ ràng giúp tiết kiệm chi phí triển khai khi phát hiện và sửa lỗi ở giai đoạn đầu của quá trình thiết kế xây dựng. Nếu một lỗi sai ban đầu không được phát hiện và sửa đổi, điều kiện này có thể là một nguy cơ tiềm ẩn của chuỗi lỗi - *chain of bugs* ở giai đoạn kiểm thử, vận hành sau này. Do đó, phát hiện các yếu tố bất ổn và lỗi thao tác kỹ thuật sẽ giảm đi thời gian phát triển dự kiến và nâng cao chất lượng phần mềm.

1.1.2. Đồ thị lời gọi trong phát tích cấu trúc và hành vi chương trình

Đồ thị lời gọi - *call graph* là một mô hình đồ thị có thể biểu diễn mối quan hệ giữa các hàm hoặc thủ tục trong chương trình. Mối quan hệ này được hiểu chính là lời gọi - *call* hoặc *invocation*. Bên trong đồ thị sẽ có hình thức biểu diễn (mô hình hóa) dưới dạng một đồ thị toán học chứa các đỉnh - *node* hoặc *vertice* để biểu diễn một hàm và các cạnh - *edge* để biểu diễn một lời gọi. [1]

Trong thực thi chương trình - *program execution*, đồ thị lời gọi giúp lập trình viên hình dung được cấu trúc điều khiển tổng thể của chương trình và mối quan hệ phụ thuộc giữa các thành phần. Dựa vào thông tin thực thi của chương trình mà đồ thị lời gọi sẽ xác định luồng thực thi, theo dõi dòng dữ liệu và xác định các điểm ảnh hưởng qua lại giữa các hàm. Do đó, việc sử dụng đồ thị lời gọi để thực hiện phân tích hành vi tĩnh - *static*

analysis mà không cần thực thi chương trình cũng là một trong số các phương pháp dò tìm lỗi phổ biến.

1.1.3. Suy luận bất biến trong khả năng phát hiện lỗi

Bất biến - *invariant* là một khái niệm có thể được phát biểu ở nhiều lĩnh vực trải dài từ toán học, vật lý, âm nhạc, ... với nhiều cách khác nhau. Tuy nhiên, các phát biểu ở nhiều lĩnh vực đều đảm bảo một tính chất cốt lõi của bất biến, đó chính là tính đúng đắn không đổi ở bất kể tác động. Nhìn nhận ở góc độ về phạm trù khoa học máy tính, điều này đồng nghĩa với các điều kiện, mệnh đề, khai báo hay lời gọi hàm được xem là các bất biến tại vị trí cụ thể của chính nó trong chương trình mặc kệ con đường thực thi nào dẫn đến nó.

Từ nhìn nhận bên trên, việc thực hiện suy luận bất biến - *invariant inference* chính là quá trình tự động hoặc bán tự động để tìm ra các bất biến khả năng - *likely invariants* trong chương trình bằng cách phân tích mã nguồn hoặc theo dõi hành vi thực thi [2]. Điều này có tác dụng gì đối với việc phân tích lỗi? Việc suy luận bất biến sẽ đảm bảo tìm kiếm được các bất biến chung của chương trình, ví dụ như tần suất một lời gọi hàm thỏa mãn ở một giá trị số nào đó trong thực thi chương trình, hoặc một cặp hàm phải xuất hiện cùng nhau trong thực thi chương trình như `malloc()` và `free()` trong ngôn ngữ C. Từ các bất biến này, lập trình viên có thể xác định các ràng buộc rõ ràng của chương trình để tìm kiếm các thực thi vi phạm các bất biến đề ra.

1.2. Vấn đề và động lực

Phần này sẽ trình bày các vấn đề cản trở của bối cảnh hiện tại và động lực để thử nghiệm các phương pháp được trình bày ở phần trước đó.

1.2.1. Thách thức trong việc tìm kiếm và sửa các lỗi không gây crash

Về mặt lý thuyết, các lỗi phần mềm trong quá trình xây dựng thiết kế chương trình được chia thành hai phạm trù hoạt động và tần suất [3] [4]. Ở phạm trù hoạt động, lỗi có thể xảy ra với chương trình sẽ khiến chương trình rơi vào một trong hai trạng thái là bình

thường - *normal* hay *non-crashing và* tê liệt - *crashing*. Ở phạm trù tần suất, các lỗi có thể xuất hiện ở mức độ từ hiếm gặp đến thường xuyên. Đặc biệt, ở các lỗi không gây ra *crashing*, chúng thường khó phát hiện hơn do điều kiện để xảy ra lỗi rất hiếm dựa trên miền dữ liệu đầu vào. Đồng thời, không giống như các lỗi gây crash - *crashing bugs* thường dẫn đến lỗi hệ thống, dừng chương trình hoặc sinh thông báo lỗi, các lỗi không gây crash - *non-crashing bugs* thường âm thầm gây ra hành vi sai lệch (như tính toán sai, trạng thái sai, kết quả đầu ra không đúng), khiến chúng khó bị phát hiện trong quá trình kiểm thử thông thường.

1.2.2. Sự cần thiết của các kỹ thuật phân tích động và tĩnh để hỗ trợ tìm kiếm lỗi

Dựa trên các thách thức phát hiện lỗi, rất nhiều kỹ thuật được áp dụng trong xử lý. Có thể kể đến kỹ thuật phân tích tĩnh - *static analysis* [5] [6] [7], kỹ thuật phân tích động - *dynamic analysis* [8] [9] [10], kỹ thuật khai thác đồ thị lời gọi - *call graph mining* [1] [3] [4] [11] [12], suy luận bất biến chung - *general invariant inference* [2][13] [14] [15] [16] [17] [18], học máy - *machine learning* [19] [20] [21] [22], ...

Trong số các kỹ thuật được nêu tên, kỹ thuật phân tích tĩnh và kỹ thuật phân tích động là hai kỹ thuật phổ biến nhất để tìm kiếm lỗi. Đối với kỹ thuật phân tích tĩnh, kỹ thuật này phân tích chương trình mà không thực sự thực thi các chương trình [9], lập trình viên không cần phải thực thi chương trình mà chỉ cần thực hiện quét thông tin trên đoạn mã nguồn - *code* hoặc các thông tin liên quan để thực hiện tìm kiếm lỗi. Bên cạnh kỹ thuật phân tích tĩnh, kỹ thuật phân tích động cũng có nhiều ưu điểm đáng chú ý. Kỹ thuật phân tích động được thể hiện thông qua việc thực thi chương trình để trực tiếp xem xét hoạt động của chương trình nhằm phát hiện lỗi [9]. Ngoài việc phân tích thực thi hành vi chương trình, kỹ thuật phân tích động cũng tập trung vào các yếu tố như dấu vết thực thi - *execution trace*, lời gọi ngăn xếp - *call stack* tại thời điểm thực thi để đánh giá và tìm kiếm lỗi.

Nhờ vào các kỹ thuật hỗ trợ ở trên, quy mô và độ phức tạp chương trình ngày càng mở rộng nhờ các công cụ áp dụng. Chúng giúp giảm thiểu chi phí và thời gian phát triển phần mềm, đồng thời cũng là nền tảng cho học máy và các phương pháp mới hơn xuất hiện.

1.2.3. Ưu điểm của việc sử dụng suy luận bất biến lên đồ thị lời gọi để phát hiện lỗi so với các phương pháp khác

Mặc dù việc áp dụng các kỹ thuật phân tích tĩnh, kỹ thuật phân tích động, ... cũng đủ đáp ứng cho tuyệt đại đa số nhu cầu tìm lỗi của lập trình viên, nhưng bản thân các kỹ thuật cũng có hạn chế bên trong.

Đối với kỹ thuật phân tích tĩnh, kỹ thuật này có nét tương tự với việc tìm kiếm lỗi thủ công và tốn rất nhiều thời gian. Do đó, rất nhiều kỹ thuật khác được sử dụng kèm với phân tích tĩnh để thực hiện như tích hợp LLM [6], suy luận linh hoạt - *dynamic reasoni*ng [9], ...

Còn đối với kỹ thuật phân tích động, chúng cũng có hạn chế khi thiếu đi độ nhạy bối cảnh thực thi - *execution context sensitivity* do chi phí trình bày ở thời gian chạy - *runtime*. Nghĩa là lúc này, lập trình viên phải thực hiện tìm kiếm thủ công với các thông tin cung cấp khi thực thi hoặc sử dụng các công cụ như Breadcrumbs để phân tích. Đối với công cụ Breadcrumbs, nó có thể giúp tăng độ nhạy bối cảnh so với mức bình thường khoảng 10% đến 20% để giảm thiểu chi phí thu thập thông tin ở thời gian thực, nhưng đổi lại cũng có trường hợp dẫn đến phạm vi tìm kiếm vượt ngoài khoảng cho phép [8].

Trong một số trường hợp lỗi đặc biệt như lỗi vi phạm bất biến, lỗi không đúng yêu cầu thiết kế thì cả kỹ thuật phân tích động và kỹ thuật phân tích tĩnh cũng không xác định được các lỗi cụ thể này mà chỉ dừng lại ở các lỗi thông dụng như cú pháp, hành vi, ... Đối với phương pháp suy luận bất biến lên đồ thị lời gọi, kỹ thuật này được đề cập ở [14] [15] chính là sự kết hợp giữa việc phân tích tĩnh đồ thị lời gọi và suy luận bất biến

để khai thác lỗi. Kỹ thuật này cho phép mở ra khả năng "học và ghi nhớ" hành vi "bình thường" của chương trình thực thi. Ngoài ra, các bất biến này có thể được thay đổi tùy thuộc vào ngưỡng thỏa mãn hành vi - *threshold* mà lập trình viên đề ra, do đó với việc thay đổi *threshold* của kỹ thuật, sẽ cho ra số lượng lỗi khác nhau [14]. Khi một bất biến có xảy ra vi phạm, điều này có thể được kỹ thuật phát hiện là sự bất thường tiềm ẩn ngay trước khi chương trình thực thi. Không giống các kỹ thuật kiểm thử dựa trên kiểm tra kết quả, phương pháp này phát hiện lỗi dựa trên sự sai lệch hành vi nội tại, ví dụ như một lời gọi hàm bất thường, một tần suất gọi thay đổi đáng kể.

Ngoài cách vận dụng suy luận bất biến lên đồ thị lời gọi, còn có một số kỹ thuật khác áp dụng lên đồ thị lời gọi chương trình để thực hiện tìm kiếm các lỗi tương tự như khai thác đồ thị - *graph mining* [1] [3] [4] [11] [12] , mạng lưới neuron đồ thị - *graph neural network* [23].

1.3. Mục tiêu đề ra của báo cáo:

Dựa vào vấn đề rào cản của việc tìm kiếm lỗi trong không gian phần mềm, các kỹ thuật đã được nghiên cứu sử dụng và ưu điểm của phương pháp được đề cập, nhóm tác giả nhìn nhận rằng các mô hình phát hiện lỗi hiện tại vẫn còn sự thiếu sót và chưa liền mạch để giúp nhận diện đầy đủ các lỗi có thể xảy ra của một chương trình thực thi. Do đó, thông qua việc làm rõ và xây dựng một mô hình con thực thi tuần tự đối với phương pháp áp dụng suy luận bất biến lên đồ thị lời gọi, nhóm tác giả muốn đặt mục tiêu xây dựng thử nghiệm mô hình với bộ biên dịch Clang, bộ công cụ phân tích thực thi Valgrind và phương pháp được đề cập để lắp ráp thành một đường ống - *pipeline* tự động hóa hoàn chỉnh hỗ trợ phát hiện lỗi trước và sau khi thực thi, đặc biệt sẽ tập trung vào các lỗi có thể phát hiện bởi trình biên dịch, các lỗi về bộ nhớ, đa luồng, tranh chấp dữ liệu và lỗi thiếu sót nội bộ cặp hàm. Đồng thời, nhóm cũng mong muốn trình bày một cách chi tiết các khái niệm, kỹ thuật liên quan để hỗ trợ quá trình xây dựng thử nghiệm.

1.4. Phạm vi tìm hiểu của báo cáo:

Báo cáo này tập trung vào việc giải quyết xem xét phương pháp ở mức độ hàm, cụ thể hơn là tập trung vào lời gọi giữa các hàm trong chương trình. Đồng thời cũng xem xét kết hợp với việc sử dụng bộ biên dịch Clang làm cơ sở cho kỹ thuật phân tích tĩnh và sử dụng Valgrind là cơ sở cho kỹ thuật phân tích động để xây dựng đồ thị lời gọi với bộ máy ảo bậc thấp LLVM nhằm phản ánh chính xác hành vi thực thi thực tế. Các thử nghiệm của báo cáo này tập trung vào tính đơn giản, dễ thực hiện với bộ dữ liệu thử nghiệm mẫu đơn giản nhằm kiểm tra hiệu quả của mô hình phát hiện lỗi, do đó bản thân việc thử nghiệm chỉ giới hạn trong quá trình kiểm thử bán tự động mà không mở rộng sang kiểm thử tự động.

Chương 2. CƠ SỞ LÝ THUYẾT

Chương này sẽ trình bày cơ sở lý thuyết cho nội dung các phương pháp ở các chương sau.

2.1. Đồ thị lời gọi - Call graph

Đồ thị lời gọi là một trong những cách biểu diễn những lời gọi nào có thể xảy ra khi thực thi chương trình.

2.1.1. Định nghĩa đồ thị lời gọi

Đồ thị lời gọi là hình thức biểu diễn dữ liệu hữu ích cho các chương trình điều khiển và luồng dữ liệu nghiên cứu giao tiếp giữa các hàm (tức là cách các hàm trao đổi thông tin). Nó chứa tất cả các mối quan hệ giữa các hàm trong một chương trình và có thể chứa thông tin bổ trợ liên quan đến dữ liệu trong mỗi hàm và dữ liệu toàn cục được chia sẻ giữa các hàm.

Về mặt định nghĩa toán học, đồ thị lời gọi có thể biểu diễn thành đồ thị có hướng với ký hiệu $G = (V, E)$ với V là tập các đỉnh đại diện cho hàm và E là tập các lời gọi từ đỉnh hàm nguồn đến đỉnh hàm đích. Với mỗi hàm P_i thuộc tập hàm V và P, mỗi đỉnh V_i là tương ứng một-một với mỗi hàm P_i và tập vector lời gọi có đỉnh hàm tương ứng. [24]

2.1.2. Đồ thị lời gọi tĩnh - Static call graph

Đồ thị lời gọi tĩnh là sự trừu tượng hóa của một chương trình máy tính biểu diễn tất cả lời gọi hàm hay phương thức có thể xảy ra. Nó được xây dựng bằng cách phân tích mã nguồn mà không cần phải chạy chương trình và không thể hiện thứ tự thực thi chương trình - *flow-insensitive*.

Vì xem xét tất cả các lời gọi có thể xảy ra nên đồ thị lời gọi tĩnh có thể bao gồm những lời gọi không bao giờ xảy ra như lời gọi phương thức đa hình, thực tế chỉ có một phương

thức được gọi nhưng trong đồ thị biểu diễn tất cả lời gọi từ các lớp con ghi đè lại phương thức đa hình đó.

2.1.3. Đồ thị lời gọi động - Dynamic call graph

Đồ thị lời gọi động biểu diễn luồng điều khiển của một chương trình khi nó được thực thi. Nó hiển thị trình tự các lệnh gọi hàm hay phương thức được thực hiện trong quá trình thực thi chương trình, cùng với tùy chọn biểu thị các tham số được truyền cho từng hàm.

Vì nó thực sự thực thi chương trình nên sẽ không có lời gọi không bao giờ xảy ra nào nhưng đồ thị cũng chỉ chứa những lời gọi với những đầu vào cụ thể, những lời gọi chưa được thực thi sẽ không được ghi lại. Ngoài ra, đồ thị lời gọi động cũng chiếm một phần bộ nhớ cũng như tài nguyên của bộ xử lý để ghi lại các lời gọi.

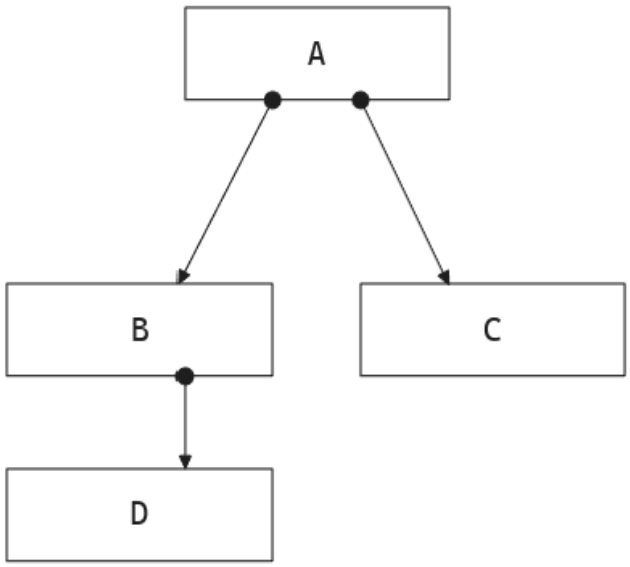
2.1.4. Phân tích nội thủ tục - Intraprocedural analysis

Phân tích nội thủ tục là loại phân tích chỉ xem xét trong phạm vi một hàm đơn lẻ, sử dụng thông tin cục bộ để tính toán luồng điều khiển và dữ liệu bên trong hàm đó. Nó không xem xét các hàm khác hoặc cách các hàm tương tác với nhau. Vì chỉ tập trung vào một hàm, phân tích nội thủ tục thường đơn giản và chi phí tính toán thấp vì không cần theo dõi lời gọi ngoài hàm. Do không xem xét các lời gọi hàm ngoài - *external calling*, nó có thể bỏ lỡ thông tin quan trọng về luồng điều khiển và sự phụ thuộc dữ liệu giữa các hàm. Trong việc phân tích đồ thị lời gọi, phân tích nội thủ tục có thể giúp xác định các lời gọi hàm trực tiếp trong một hàm và hỗ trợ xây dựng đồ thị lời gọi.

2.1.5. Phân tích liên thủ tục - Interprocedural analysis

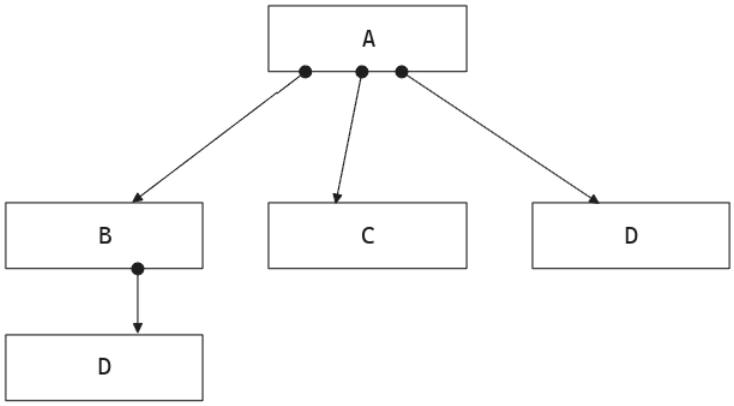
Phân tích liên thủ tục so với phân tích nội thủ tục mở rộng phạm vi ra toàn bộ chương trình, xem xét các mối quan hệ giữa các hàm, bao gồm lời gọi, trả về và truyền tham số, giúp xây dựng và tinh chỉnh đồ thị lời gọi chính xác hơn nhưng đồng thời phức tạp và tốn kém về mặt tính toán hơn.

Phân tích liên thủ tục có độ chính xác cao hơn vì nó có thể thu thập thông tin chính xác hơn về luồng điều khiển và sự phụ thuộc dữ liệu, đặc biệt là trong các chương trình phức tạp với nhiều lời gọi hàm. Nhưng việc triển khai thường phức tạp hơn so với phân tích nội thủ tục. Trong phân tích đồ thị lời gọi, phân tích liên thủ tập có thể xác định các lời gọi hàm gián tiếp. Ví dụ, nếu ta có hàm A gọi đến hàm B và C, hàm B thì gọi đến D. Với phân tích liên thủ tục ở mức một, ta sẽ thu được đồ thị lời gọi phân tích như sau:



Hình 1: Đồ thị lời gọi ở mức phân tích liên thủ tục một.

Với phân tích liên thủ tục ở mức hai, ta sẽ thu được đồ thị lời gọi phân tích như sau:



Hình 2: Đồ thị lời gọi ở mức phân tích liên thủ tục hai.

2.2. Lỗi - Bug

Phần này trình bày về các loại lỗi trong quá trình phân tích thiết kế phần mềm.

2.2.1. Định nghĩa lỗi

Lỗi là các sai sót, khuyết điểm hoặc lỗi trong phần mềm máy tính dẫn đến kết quả không mong muốn hoặc không lường trước được. Chúng có thể xuất hiện theo nhiều cách khác nhau, bao gồm hành vi không mong muốn, hệ thống bị sập hoặc đóng băng hoặc đầu ra không đúng và không đủ. [25]

Trong tài liệu [4], lỗi trong quá trình thực thi chương trình biểu hiện bằng cách tạo ra một số kết quả khác với kết quả đã chỉ định hoặc dẫn đến một số hành vi thời gian chạy không mong muốn như sự cố hoặc chạy không kết thúc.

2.2.2. Phân loại lỗi

Theo tài liệu [4], lỗi có thể được chia ra thành ba loại và sáu kiểu lỗi:

Lỗi loại hành vi

Loại lỗi này gồm hai lỗi chính là lỗi gây tê liệt - *crashing bug*, lỗi không gây tê liệt - *non-crashing bug*.

Lỗi gây tê liệt là loại lỗi dẫn tới sự kết thúc của chương trình. Thông thường các ngoại lệ của một ngôn ngữ lập trình sẽ thuộc về lỗi này, ví dụ như NullPointerException, ArithmeticException, IndexOutOfBoundException, ... Các lỗi này có thể được tìm thấy thông qua dấu vết ngăn xếp - *stack trace*.

Lỗi không gây tê liệt trái lại với lỗi gây tê liệt sẽ khó tìm thấy hơn và thường sẽ được tập trung vào các phương pháp tìm kiếm hơn.

Lỗi loại tần suất

Loại lỗi này gồm hai lỗi là lỗi thỉnh thoảng - *occasional bug* và lỗi thường xuyên - *non-occasional bug*.

Lỗi thường xuyên là loại lỗi dễ xảy ra, dễ thực hiện lại - *reproduce*. Các lỗi này có thể kể đến như lỗi cú pháp, lỗi logic đơn giản, … dễ dàng kiểm tra do có thể viết các test case để kiểm tra và xác nhận việc sửa lỗi và thường liên quan đến lỗi logic hoặc lỗi cú pháp.

Lỗi thỉnh thoảng thường khó phát hiện hơn so với lỗi thường xuyên, có thể phụ thuộc vào sự kết hợp phức tạp của các yếu tố, chẳng hạn như trạng thái hệ thống, thời gian, dữ liệu đầu vào cụ thể, hoặc thậm chí là sự tương tác với các phần mềm khác.

Lỗi loại tác động

Loại lỗi này gồm lỗi tác động cấu trúc - *structure-affecting bug* và lỗi tác động tần suất gọi - *call frequency-affecting bug*.

Lỗi tác động cấu trúc không chỉ gây ra hành vi sai hoặc sập ứng dụng, mà còn có thể làm suy thoái cấu trúc chương trình. Những lỗi cấu trúc này dẫn đến khó bảo trì, dễ xuất hiện lỗi lan truyền và tăng chi phí phát triển dài hạn. Một ví dụ có thể kể đến như lỗi tăng phụ thuộc liên kết cặp - *coupling, coupling* là mức độ phụ thuộc giữa các lớp, hàm hoặc

thành phần con - *module*, các thể hiện của lỗi này có thể kể đến như sử dụng biến toàn cục, truyền tham số không cần thiết làm tăng *coupling* khiến các *module* liên quan quá nhiều đến nhau.

Lỗi tác động tần suất gọi sẽ không làm thay đổi cấu trúc đồ thị lời gọi nhưng lại thay đổi đi số lần thực thi hàm so với kỳ vọng. Những lỗi này thường khó phát hiện bằng phân tích cấu trúc đơn thuần vì đường đi lời gọi hàm vẫn giống nhau, chỉ có tần suất gọi thay đổi. Lỗi này ảnh hưởng đến hiệu năng khi tăng tần suất gọi có thể gây giảm hiệu năng nghiêm trọng, hoặc ngược lại bỏ sót xử lý khi gọi quá ít. Làm giảm độ tin cậy khi dữ liệu không được xử lý đúng số lần, dẫn đến mất mát hoặc trùng lặp kết quả.

2.3. Bất biến - Invariant

Phần này sẽ trình bày lý thuyết về bất biến và suy luận với bất biến.

2.3.1. Định nghĩa bất biến

Bất biến - *invariant* là một điều kiện hoặc mệnh đề luôn đúng tại một vị trí cụ thể trong chương trình, bất kể con đường thực thi nào dẫn đến nó. Chúng giúp lập trình viên xác định các giả định mà chương trình phụ thuộc vào, từ đó bảo vệ họ khỏi việc vô tình vi phạm các giả định này khi thực hiện thay đổi. [2] [17]

2.3.2. Bất biến khả năng

Bất biến khả năng là những tính chất của chương trình mà được suy luận là đúng dựa trên quan sát hành vi của chương trình trong quá trình thực thi. Chúng phản ánh những mối quan hệ hoặc điều kiện “gần như luôn đúng” trong thực tế thực thi, và thường được dùng để hỗ trợ kiểm thử, gỡ lỗi, hay tăng cường tính an toàn của chương trình.

2.3.3. Suy luận bất biến trong chương trình và đồ thị lời gọi

Suy luận bất biến là quá trình tự động xác định các bất biến của chương trình. Kỹ thuật này nhằm tìm ra các bất biến có khả năng đúng từ chương trình đó. Các quy tắc hoặc mẫu hành vi thường xuyên xuất hiện trong đồ thị lời gọi như sau:

Chuỗi lời gọi cố định

Mẫu đường đi thực thi hàm $A \rightarrow B \rightarrow C$ xuất hiện hầu hết trong các lần chạy, dùng để suy ra luồng nghiệp vụ.

Bất biến thứ tự

Nếu B được gọi, thì A phải được gọi trước đó trong cùng ngữ cảnh.

Bất biến tần suất

Tỉ lệ số lần gọi hàm X trên n số lần gọi hàm Y nằm trong một ngưỡng ổn định mẫu (ví dụ X ≈ 2Y).

Bất biến vắng mặt

Hàm Z không bao giờ có cạnh - lời gọi đến một hàm Y hoặc hàm X mẫu.

Quan hệ tham số và lời gọi

Giá trị trả về của một hàm X luôn là tham số đầu vào của hàm Y trong cùng một chuỗi lời gọi.

Ngoài việc sử dụng các quy tắc hoặc mẫu hành vi để xác định, một số phương pháp khác có thể như sử dụng phân tích thống kê, sử dụng học máy, phân tích mẫu đồ thị con.

Chương 3. PHƯƠNG PHÁP TỐI GIẢN ĐỒ THỊ LỜI GỌI

3.1. Tổng quan

Chương này tập trung vào tìm hiểu các phương pháp tối giản đồ thị lời gọi nhằm hỗ trợ quá trình suy luận trên đồ thị.

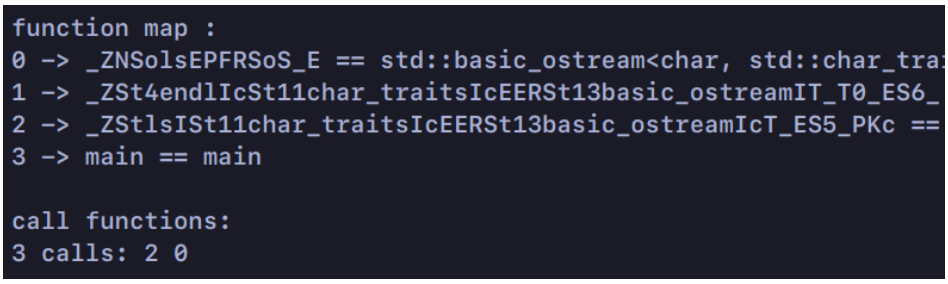
3.1.1. Sự cần thiết của việc tối giản đồ thị lời gọi

Khi thực hiện xây dựng đồ thị lời gọi từ ngôn ngữ lập trình và công cụ, đa phần chỉ hỗ trợ việc xuất ra hoàn chỉnh đồ thị mà không quan tâm tới độ phức tạp thông tin. Nếu không thực hiện tối giản cả về cấu trúc thông tin lẫn kích thước thông tin, chi phí để thực hiện phân tích sẽ vô cùng lớn, ảnh hưởng đến cả thời gian chạy, bộ nhớ tiêu thụ và độ trễ phản hồi của thông tin hệ thống. Ví dụ đối với công cụ dòng lệnh `opt` của LLVM, khi thực hiện trích xuất tệp bitcode (tệp bitcode là tệp trung gian của công cụ Clang dùng để lưu trữ thông tin chương trình không phụ thuộc vào một nền tảng) của một chương trình C hoặc C++, ta sẽ thu được thông tin về đồ thị lời gọi mẫu như sau:

```
Call graph node <<null function>><<0x55a86934df90>>  #uses=0
  CS<None> calls function 'main'
  CS<None> calls function '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'
  CS<None> calls function '_ZNSolsEPFRSoS_E'
  CS<None> calls function '_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_'

Call graph node for function: '_ZNSolsEPFRSoS_E'<<0x55a86934a430>>  #uses=2
  CS<None> calls external node

Call graph node for function: '_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_'<<0x55a86934a4b0>>  #uses=1
  CS<None> calls external node

Call graph node for function: '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'<<0x55a86934a380>>  #uses=2
  CS<None> calls external node

Call graph node for function: 'main'<<0x55a8693487d0>>  #uses=1
  CS<0x55a86934e540> calls function '_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc'
  CS<0x55a86934e600> calls function '_ZNSolsEPFRSoS_E
```

Hình 3: Thông tin đồ thị lời gọi trích xuất từ tệp bitcode của lệnh opt.

Từ Hình 3 có thể thấy thông tin về đồ thị lời gọi được trình bày đầy đủ nhưng lại gây khó khăn cho việc đọc hiểu, vì vậy cần thực hiện một số thao tác như ánh xạ và lưu trữ nhằm tối giản thông tin. Thông qua Hình 3, nhóm tác giả thực hiện tối giản thông tin từ đồ thị lời gọi thành mẫu thông tin như sau:



Hình 4: Thông tin đồ thị lời gọi được tối giản từ Hình 3.

Sau khi thực hiện tối giản thông tin, đồ thị sẽ trở nên gọn gàng và dễ thao tác hơn với kiểu dữ liệu số và cấu trúc dữ liệu của C/C++. Các phần thông tin bị cắt đi sau dấu "==" chỉ là phần biến đổi - *demangle* các tên bị mã hóa trong tệp bitcode, do đó có thể tùy ý thêm vào hoặc không tùy theo yêu cầu.

3.1.2. Các phương pháp tối giản và ưu nhược điểm

Phương pháp Total Reduction

Gộp tất cả các đỉnh đại diện cho cùng một hàm trong toàn bộ đồ thị thành 1 đỉnh duy nhất. Mỗi cạnh chỉ biểu diễn sự tồn tại của mối quan hệ gọi hàm, không mang thông tin về tần suất. Phương pháp này đơn giản dễ thực hiện, phù hợp đối với các phân tích cấu trúc cơ bản nhưng đổi lại làm mất thông tin về tần suất gọi, có thể không phát hiện các lỗi liên quan đến tần suất gọi - *call frequency affecting bugs* [11].

Phương pháp Total reduction with edge weight

Thực hiện tương tự phương pháp Total Reduction nhưng mỗi cạnh trong đồ thị được gắn thêm trọng số đại diện cho tần số gọi giữa hai hàm trong nhiều lần chạy. Điều này giúp giữ lại thông tin của hành vi, mở rộng hỗ trợ của Total Reduction đối với lỗi liên quan đến tần suất gọi, cân bằng giữa giảm thiểu độ phức tạp và giữ lại thông tin quan trọng. Ngược lại, phương pháp này sẽ tăng xử lý số liệu thống kê, điều này có thể không cần thiết nếu yêu cầu phát hiện lỗi không quá chú trọng vào tần suất.

Phương pháp Zero-one-many Reduction

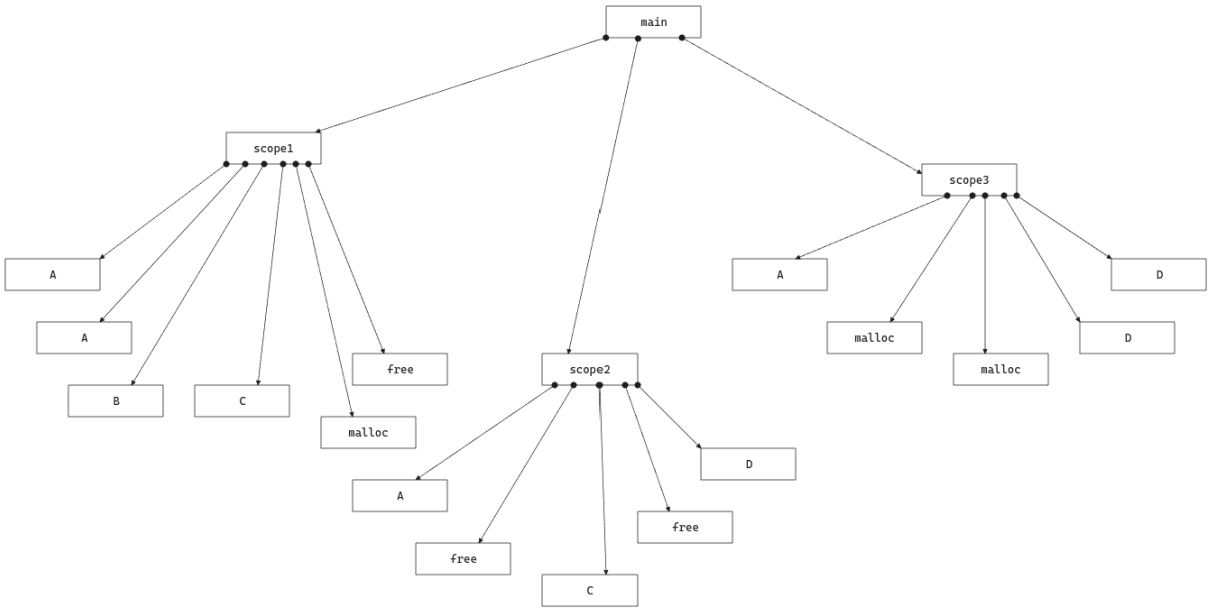
Thực hiện việc trừu tượng hóa tần suất thành ba mức: 0 đối với không gọi, 1 đối với gọi 1 lần, N đối với gọi nhiều hơn 1 lần. Phương pháp này có thể thay thế cho phương pháp Total reduction with edge weight do đơn giản hóa biểu diễn nhưng vẫn giữ được ý nghĩa hành vi tổng quát, hỗ trợ đối với suy luận bất biến định tính - *qualitative invariant*. Tuy nhiên, cách trừu tượng này lại không phù hợp trong trường hợp cần sự chi tiết trong thông tin và đồng thời cũng cho ra đồ thị có sự tối giản hoàn toàn.

3.2. Các kỹ thuật tối giản đồ thị lời gọi

Phần này sẽ tập trung vào ba phương pháp được sử dụng phổ biến là tối giản toàn cục - *total reduction*, tối giản toàn cục với trọng số lời gọi - *total reduction with edge weight* và tối giản 0-1-N - *zero-one-many reduction*. Mỗi phương pháp sẽ có ưu nhược điểm khác nhau tùy thuộc vào mục đích sử dụng.

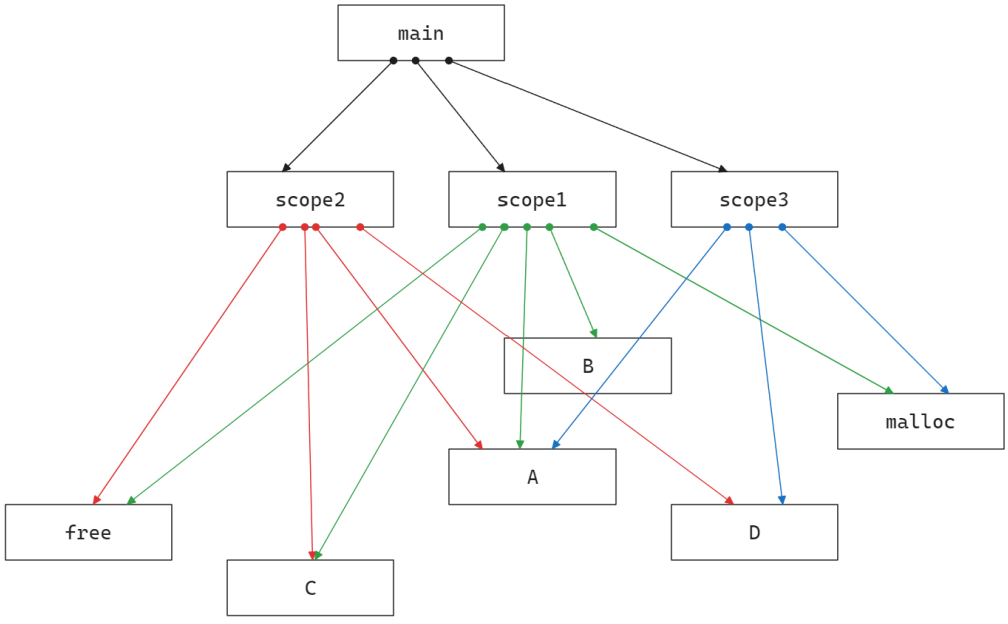
3.2.1. Tối giản toàn cục - Total reduction

Tối giản toàn cục tập trung vào việc tối giản thông tin đồ thị ở mức tối đa mà không quan tâm đến cấu trúc nội tại của đồ thị. Khi này, mỗi hàm sẽ chỉ xuất hiện một lần duy nhất trong đồ thị và được biểu thị bằng một đỉnh, một cạnh sẽ được nối giữa hai đỉnh nếu có lời gọi giữa hai hàm tương ứng. Lấy ví dụ, ta có một đồ thị lời gọi mẫu như sau:



Hình 5: Đồ thị lời gọi mẫu.

Nếu thực hiện phương pháp tối giản toàn cục, đồ thị lời gọi sẽ có hình thái như sau:



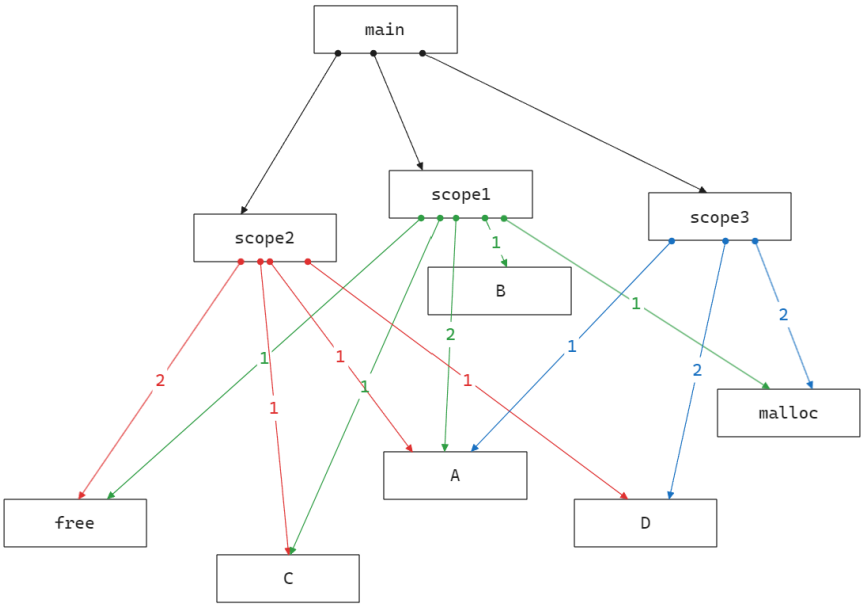
Hình 6: Đồ thị lời gọi sau khi áp dụng phương pháp tối giản toàn cục.

Đối với phương pháp tối giản toàn cục, chúng có khả năng giảm đáng kể kích thước đồ thị. Trong thí nghiệm mẫu của [26], nhóm tác giả này đã áp dụng thực hiện phương pháp này lên đồ thị lời gọi mẫu có 22 đỉnh 21 cạnh và cho ra kết quả ấn tượng khi đồ thị lời gọi chỉ còn lại 6 đỉnh 6 cạnh. Tuy nhiên, đánh đổi lại, cấu trúc sẽ bị mất đi thông tin và thay đổi. Điều này lại gây khó khăn trong việc truy vấn hoặc khai thác mẫu đồ thị con - *subgraph*.

Do đó, kỹ thuật tối giản toàn cục chỉ phù hợp khi cần xử lý lượng lớn dữ liệu và không yêu cầu thông tin chi tiết, phân tích cấu trúc tổng quát giữa các hàm, không phân tích hành vi chi tiết.

3.2.2. Tối giản toàn cục với trọng số lời gọi - Total reduction with edge weight

Kỹ thuật này là sự mở rộng từ kỹ thuật tối giản toàn cục khi bổ sung thêm tần suất gọi - ở đây chính là trọng số vào các cạnh. Lấy lại ví dụ ở trên, đối với kỹ thuật này ta sẽ thu được kết quả như sau:



Hình 7: Đồ thị lời gọi sau khi áp dụng tối giản toàn cục với trọng số lời gọi.

Kỹ thuật này giảm thiểu các rủi ro về vấn đề không phát hiện các lỗi tần suất mà vẫn đảm bảo cấu trúc thông tin. Tuy nhiên, kỹ thuật này lại không được sử dụng phổ biến do phải bổ sung thêm chi phí thống kê.

3.2.3. Tối giản 0-1-N - Zero-One-Many reduction

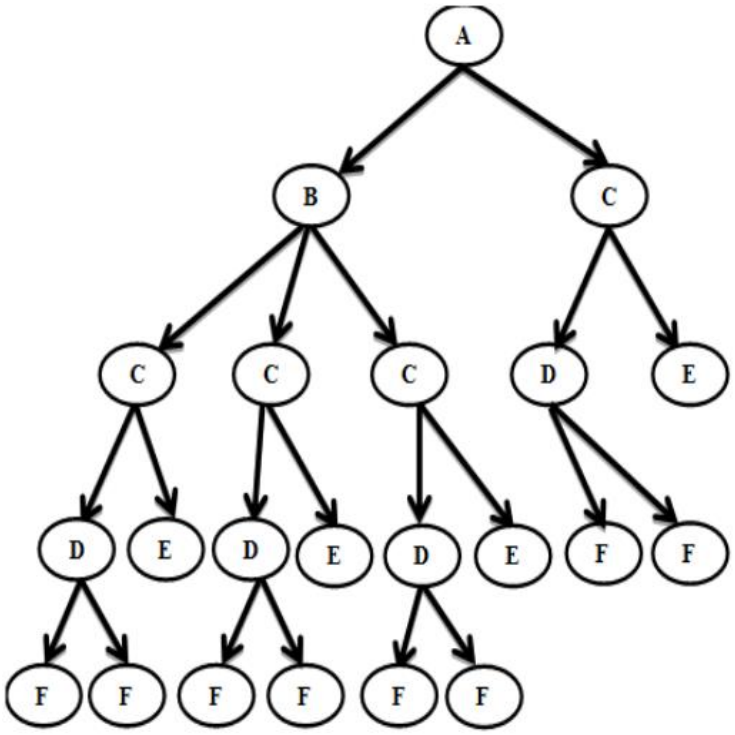
Kỹ thuật tối giản 0-1-N là một kỹ thuật có nét tương tự với kỹ thuật tối giản toàn cục, tuy nhiên trái với việc loại bỏ toàn bộ để giữ lại duy nhất thông tin, kỹ thuật tối giản 0-1-N thực hiện việc trừu tượng hóa tần suất thành ba mức. Trong đó đối với mức N sẽ phụ thuộc vào yêu cầu tối giản. Lấy ví dụ của [12], kỹ thuật này sẽ định sẵn mức N = 2, nghĩa là các lời gọi nào có tần suất vượt quá N sẽ bị loại bỏ và chỉ giữ lại đúng N lời gọi trên đồ thị. Điều này giúp giảm tải việc phân tích thống kê nhưng lại không hoàn toàn tối giản đồ thị, phù hợp hơn đối với việc sử dụng kèm phương pháp khai thác đồ thị - *graph mining*.

3.3. So sánh các kỹ thuật

Phần này thực hiện so sánh giữa kỹ thuật tối giản toàn cục - *total reduction* và kỹ thuật tối giản 0-1-N - *Zero-one-many reduction*. Kỹ thuật tối giản toàn cục với trọng số lời gọi chỉ là phần mở rộng của kỹ thuật tối giản toàn cục, do đó không cần thiết phải đưa vào so sánh.

3.3.1. Hiệu quả tối giản đồ thị

Theo thí nghiệm mô phỏng của [26] với mẫu đồ thị 22 đỉnh 21 cạnh như sau:



Hình 8: Đồ thị lời gọi mẫu từ thử nghiệm.

Kết quả được nhóm tác giả đưa ra như sau:

Phương pháp tối giản	**Số lượng đỉnh**	**Số lượng cạnh**	**Ảnh hưởng lên đồ thị**
Đồ thị mẫu	22	21	
Tối giản toàn cục	6	6	Mất thông tin và thay đổi cấu trúc
Tối giản 0-1-N	15	14	Không tối giản hoàn toàn Mất thông tin nhưng đảm bảo cấu trúc

Bảng 1: So sánh giữa hai kỹ thuật tối giản đồ thị lời gọi.

Kết quả so sánh cho thấy tính hiệu quả cao của phương pháp tối giản toàn cục lên đồ thị mẫu, nhưng đồng thời cũng tỉ lệ thuận với mức độ ảnh hưởng lên đồ thị. Ngược lại, phương pháp tối giản 0-1-N chỉ cho ra kết quả tối giản ở mức vừa phải nhưng đủ đáp ứng thông tin và cấu trúc để thực hiện phân tích sâu hơn.

3.3.2. Ảnh hưởng đến khả năng phát hiện lỗi

Hiện tại, các nghiên cứu liên quan đến lĩnh vực này vẫn chưa hoàn toàn đầy đủ để thu thập và phân tích sâu ảnh hưởng của các kỹ thuật lên khả năng phát hiện lỗi. Đồng thời, các kỹ thuật này chỉ được xem là bước trung gian đầu vào cho việc phân tích bằng các kỹ thuật khác như suy luận bất biến, khai thác đồ thị, học máy, LLM, ... Do đó, ở thời điểm hiện tại nhóm tác giả nhận định rằng ảnh hưởng của các kỹ thuật này lên khả năng phát hiện chỉ dừng lại xem xét ở khả năng khai thác thu thập đầy đủ thông tin mà vẫn giữ nguyên được cấu trúc về đồ thị lời gọi. Ở tiêu chí này, kỹ thuật tối giản 0-1-N có phần nhỉnh hơn so với kỹ thuật tối giản toàn cục và tối giản toàn cục với trọng số lời gọi do có thể bảo toàn được cấu trúc đồ thị lời gọi.

Chương 4. PHƯƠNG PHÁP PHÁT HIỆN LỖI CẶP HÀM DỰA TRÊN ĐỒ THỊ LỜI GỌI VÀ SUY LUẬN BẤT BIẾN

4.1. Tổng quan

Chương này sẽ trình bày chi tiết về phương pháp phát hiện lỗi cặp hàm dựa trên đồ thị lời gọi đã tối giản và suy luận bất biến.

4.2. Nguyên tắc hoạt động tuần tự

Phần này sẽ trình bày đầy đủ các bước của phương pháp.

4.2.1. Thu thập và tiền xử lý đồ thị lời gọi

Đây là quá trình đầu tiên trong việc phát hiện lỗi, ở bước này, tùy thuộc vào ngôn ngữ lập trình được lựa chọn sẽ có nhiều phương pháp thực hiện khác nhau. Nhóm tác giả thực hiện lựa chọn thử nghiệm trên hai ngôn ngữ C và C++ vì tính đơn giản và được xây dựng sẵn bộ công cụ hỗ trợ opt từ LLVM [14] [15]. Sau khi xác định được ngôn ngữ và bộ công cụ đi kèm, lúc này ta có thể lựa chọn thêm bash script để thực hiện tự động hóa quy trình xây dựng. Với đồ thị lời gọi thu thập được, việc tiền xử lý đồ thị lời gọi sẽ bao gồm một số công đoạn như ánh xạ, loại bỏ thông tin thừa, thay đổi thông tin nhằm phù hợp với nhu cầu ở các giai đoạn sau.

4.2.2. Tối giản đồ thị lời gọi

Sau khi đã thực hiện tiền xử lý đồ thị lời gọi, ta sẽ tiếp tục thực hiện tối giản đồ thị lời gọi dựa theo các phương pháp đã đề cập ở phần lý thuyết. Ở đây, nhóm tác giả thực hiện lựa chọn theo phương pháp tối giản 0-1-N với N = 1 để đảm bảo tính đơn giản, dễ thực hiện đối với thử nghiệm. Việc sử dụng N = 1 cho phép tối giản thông tin tối đa với các lời gọi có tần suất cao mà vẫn đảm bảo toàn bộ cấu trúc thông tin của lời gọi.

4.2.3. Suy luận các bất biến khả năng

Ở bước này, thực hiện suy luận dựa trên việc xác định các cặp hàm (X,Y) và bản thân các hàm X, Y bất kỳ để tính số lần một cặp hàm hoặc hàm xuất hiện - *support* và độ tin cậy của cặp hàm đã xác định - *confidence* (*confidence* có thể hiểu đơn giản chính là xác suất mà hàm Y xuất hiện khi hàm X xuất hiện) [14]. Công thức tính *support* và *confidence* được tính như sau:

$$support(X) = \sum(số\ lần\ X\ xuất\ hiện)$$

$$support(X,Y) = \sum(số\ lần\ X\ và\ Y\ xuất\ hiện\ cùng\ nhau)$$

$$confidence(\{X,Y\},\{X\}) = \frac{support(X,Y)}{support(X)}$$

Sau khi tính toán các giá trị cần thiết, thực hiện so sánh và giữ lại các cặp hàm thỏa mãn giá trị ngưỡng - *threshold* của *support* và *confidence*. Hai giá trị ngưỡng này có thể tùy ý thay đổi phụ thuộc vào yêu cầu tần suất lỗi xuất hiện là bao nhiêu.

4.2.4. Phân tích thống kê các bất biến có vi phạm

Dựa vào các cặp hàm (X,Y) thỏa mãn yêu cầu, ta thực hiện duyệt lại đồ thị lời gọi gốc ban đầu và kiểm tra nếu có xuất hiện trường hợp một hàm thành phần Y trong chương trình hoặc khu vực thực thi được gọi nhưng lại không đi kèm với hàm thành phần X. Nếu xuất hiện trường hợp này thì trả về lỗi có thể xảy ra tại cặp hàm và vị trí có khả năng lỗi đó.

Chương 5. ĐÁNH GIÁ THỐNG KÊ THỬ NGHIỆM

Chương này sẽ thực hiện mô tả việc áp dụng quy trình phát hiện lỗi với hai công cụ Clang, Valgrind và phương pháp được đề cập lên bộ dữ liệu mẫu và thực hiện thống kê kết quả trả về.

5.1. Thiết lập thử nghiệm

Thử nghiệm này được thực hiện trên cấu hình AMD Ryzen™ 3 4300U, 8GB RAM và nền tảng Fedora Linux 40 với các công cụ hỗ trợ như Github, VS Code, Clang, Valgrind, LLVM. Các ngôn ngữ lập trình được đưa vào sử dụng bao gồm Python, C, C++ và Bash Shell.

5.1.1. Bộ dữ liệu thử nghiệm

Bộ dữ liệu thử nghiệm - *dataset* được sử dụng bao gồm 30 chương trình C++ với số lượng phương phức, lời gọi khác nhau không được gắn nhãn lỗi. Đây là tập hợp từ nhiều nguồn khác nhau bao gồm bộ dữ liệu C++ của Alexandru Jercan về bài nộp lập trình thi đấu. mẫu chương trình từ bài giảng LLVM của Edmund Wong, Irene Huang, Taiyue Liu và giáo sư Lin Tan. Ngoài một số nguồn dữ liệu này, nhóm tác giả cũng lấy ý tưởng thêm từ các mã nguồn thực tế thường sử dụng cho khóa học nhập môn lập trình và hệ điều hành để hoàn thiện. Về cụ thể chi tiết, các chương trình có lỗi sẽ nằm tập trung rải rác trải đều, trong đó một số chương trình kiểm thử sẽ có số lượng lỗi bên trong từ 1 đến tối đa 30 lỗi chưa được xác định.

5.1.2. Lỗi sử dụng trong thử nghiệm

Trong thử nghiệm này sẽ sử dụng đa dạng lỗi từ các lỗi cú pháp đơn giản đến các lỗi nghiêm trọng như lỗi liên quan đến tranh chấp dữ liệu, lỗi rò rỉ bộ nhớ. Ngoài ra lỗi liên quan đến cặp hàm cũng được đem vào thử nghiệm để đánh giá hiệu quả khi thực hiện phân tích liên thủ tục kèm theo phương pháp suy luận bất biến lên đồ thị lời gọi. Phân bố lỗi trong thử nghiệm sẽ là ngẫu nhiên và việc áp dụng quy trình vào kiểm tra lỗi với

các tham số khác nhau sẽ cho ra số lượng lỗi khác nhau. Điều này cũng dẫn tới việc nên thực hiện kiểm tra các hành vi không xác định - *undefined behaviour*, các trường hợp dương tính giả - *false-positive* (nghĩa là các trường hợp bị xem là lỗi nhưng lại không có lỗi trên thực tế).

5.1.3. Quy trình thử nghiệm

Phần này sẽ trình bày đầy đủ các bước thực hiện để sử dụng phối hợp các công cụ trong việc tìm kiếm lỗi.

5.1.3.1. Kiểm tra lỗi từ biên dịch thực thi

Ở bước này, ta thực hiện truyền tệp chương trình C++ cần kiểm tra vào công cụ giao diện dòng lệnh Clang. Nếu có lỗi, Clang sẽ trả về trực tiếp trên dòng lệnh hoặc có thể điều hướng vào một tệp đầu ra và không thực hiện biên dịch. Ở bước này sẽ là quyết định quá trình tiếp theo, nếu phát hiện ra lỗi từ lúc biên dịch thì toàn bộ quy trình dừng lại. Cú pháp như sau:

```
clang++ -g <testfilename>.cpp -o <testfilename>
```

Nếu muốn chuyển hướng lỗi vào tệp đầu ra, ta thực hiện bổ sung thêm `&>> output.out` vào đuôi cú pháp.

5.1.3.2. Kiểm tra lỗi bộ nhớ bằng Valgrind

Sau khi vượt qua kiểm tra từ biên dịch thực thi, lúc này ta sẽ có tệp biên dịch chương trình, sử dụng công cụ memcheck của Valgrind, ta có thể thực hiện kiểm tra các lỗi liên quan đến bộ nhớ (lưu ý rằng, đối với một số sai sót trong thao tác bộ nhớ như cấp phát, thu hồi sẽ bị không bị memcheck chú trọng mà chỉ để thông báo xem như một cảnh báo - *warning*). Cú pháp thực thi như sau:

```
valgrind --tool=memcheck <testfilename> <arguments>
```

Việc thực hiện chuyển hướng đầu ra tương tự như phần 5.1.3.1. Nếu muốn kiểm tra kỹ hơn, Valgrind hỗ trợ kiểm tra lỗi rò rỉ chi tiết bằng tùy chọn trước tệp biên dịch thực thi `--leak-check=full`.

5.1.3.3. Kiểm tra lỗi xử lý đa luồng bằng Valgrind

Từ tệp biên dịch thực thi của chương trình, ta tiếp tục thực thi kiểm tra đa luồng với Valgrind bằng cú pháp sau:

```
valgrind --tool=helgrind <testfilename> <arguments>
```

Nếu có xảy ra các khả năng tranh chấp thì Valgrind sẽ thông báo và báo lỗi.

5.1.3.4. Kiểm tra lỗi tranh chấp dữ liệu bằng Valgrind

Sau khi kiểm tra lỗi bằng công cụ helgrind, Valgrind sẽ hỗ trợ tiếp tục kiểm tra lỗi tranh chấp dữ liệu từ chương trình với cú pháp sau:

```
valgrind --tool=drd <testfilename> <arguments>
```

5.1.3.5. Kiểm tra lỗi bằng đồ thị lời gọi và suy luận bất biến

Sau khi đã thực hiện kiểm tra lỗi với các công cụ kể trên, ta sẽ sử dụng công cụ Clang để sinh ra tệp bitcode chương trình như sau:

```
clang++ -emit-llvm -c <testfilename>.cpp -o <testfilename>.bc
```

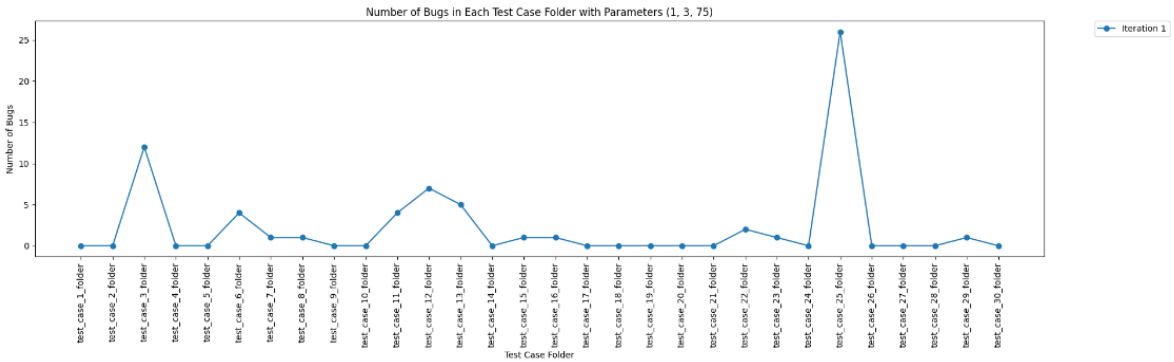
Từ tệp bitcode thu được, ta sẽ xây dựng chương trình C++ để hỗ trợ cho việc phân tích và xác định các lỗi, trong chương trình C++ cũng hỗ trợ việc phân tích liên thủ tục nếu có thiết lập. Sau khi hoàn tất, công cụ opt của LLVM sẽ giúp xây dựng đồ thị lời gọi và truyền vào chương trình C++ để phân tích với cú pháp như sau:

```
opt -passes=print-callgraph <testfilename>.bc 2>&1
/dev/null | ./<analyzer> <arguments>
```

Cú pháp này sử dụng cơ chế đường ống - *pipeline* của Bash Shell để thực hiện và hỗ trợ điều hướng lỗi với `/dev/null`.

5.2. Kết quả thống kê

Với bộ dữ liệu thu thập, nhóm tác giả đã thực hiện so sánh kiểm chứng giữa quy trình thực hiện và kỹ thuật phân tích tĩnh mẫu ở mức phân tích liên thủ tục một, $T_{support} = 3$ (ngưỡng *support* – số lần xuất hiện trong chương trình phải thỏa mãn hơn 3 lần) và $T_{confidence} = 75$ (ngưỡng *confidence* – xác suất hàm y xuất hiện khi hàm x xuất hiện là mốc 75%). Kết quả của kỹ thuật phân tích mẫu thủ công dường như rất khó để phát hiện các lỗi liên quan đến bất biến cặp hàm do chi phí tính toán và bổ sung các lỗi khác, điều này dẫn đến việc kỹ thuật phân tích tĩnh mẫu thủ công do nhóm tác giả thực hiện chỉ ghi nhận khoảng từ 60-65% lỗi so với quy trình thực hiện kiểm tra và rất dễ sai sót. Đối với quy trình thực hiện kiểm tra, kết quả cho ra ở với bộ tham số mẫu đối chiếu được ghi nhận như sau:



Hình 9: Kết quả thống kê lỗi với mức tham số mẫu đưa ra.

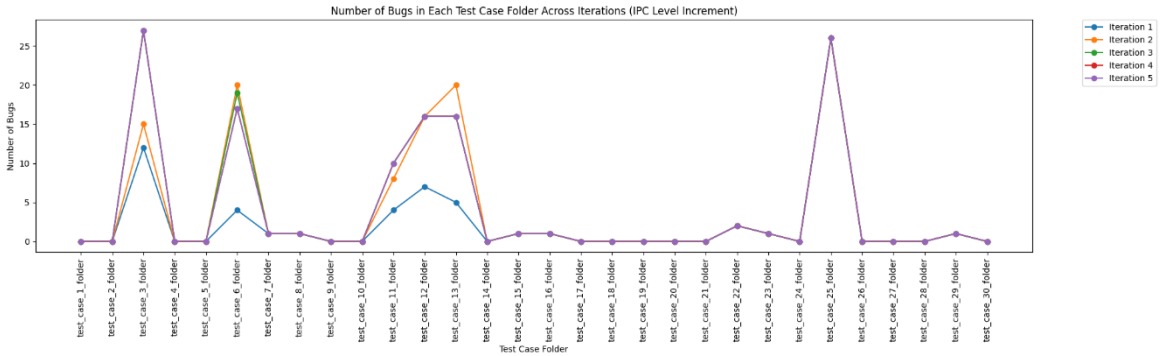
Kết quả này phản ánh khả năng phát hiện lỗi tốt hơn của quy trình với việc kết hợp các công cụ vào quá trình hỗ trợ phân tích lỗi.

5.3. So sánh khả năng phát hiện lỗi khi tham số thay đổi tuyến tính

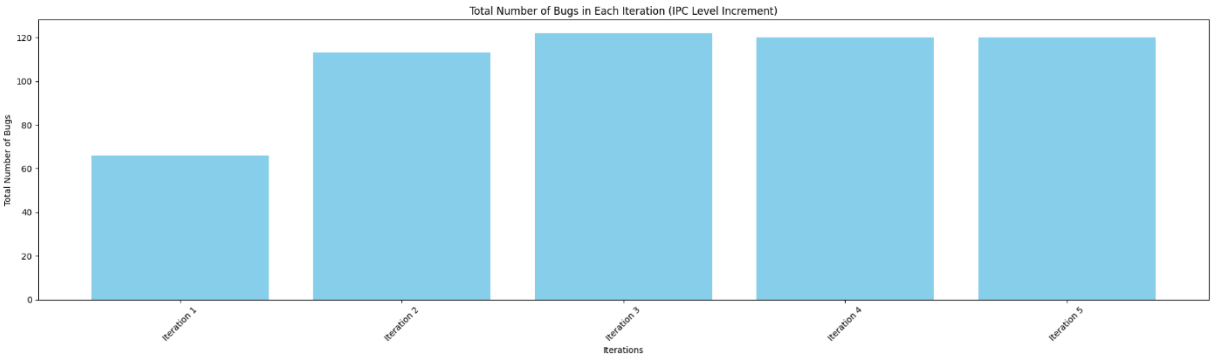
Để tìm kiếm mẫu tham số phù hợp hơn cho quá trình tìm kiếm lỗi, nhóm tác giả thực hiện việc thay đổi 1 trong 3 các tham số và giữ nguyên các tham số còn lại.

Khi thực hiện tăng tuyến tính mức độ phân tích liên thủ tục

Đối với việc tăng tuyến tính mức độ phân tích liên thủ tục, nhóm tác giả chỉ dừng lại ở mức từ 1 đến 5. Bởi vì trên thực tế làm việc trung bình trong một dự án, một chuỗi lời gọi – *call chain* thường sẽ không vượt quá độ dài là 10 hàm. Do đó, với tình trạng thiết bị và thời gian thực hiện, việc xem xét phân tích liên thủ tục ở một chuỗi với khoảng mức từ 1 (không thực hiện phân tích) đến 5 (phân tích ở chuỗi 5 lời gọi) là phù hợp để khảo sát. Kết quả được nhóm tác giả đưa ra ở dạng biểu đồ đường và biểu đồ cột như sau:



Hình 10: Kết quả thống kê số lượng lỗi với mức độ phân tích liên thủ tục từ 1 đến 5.



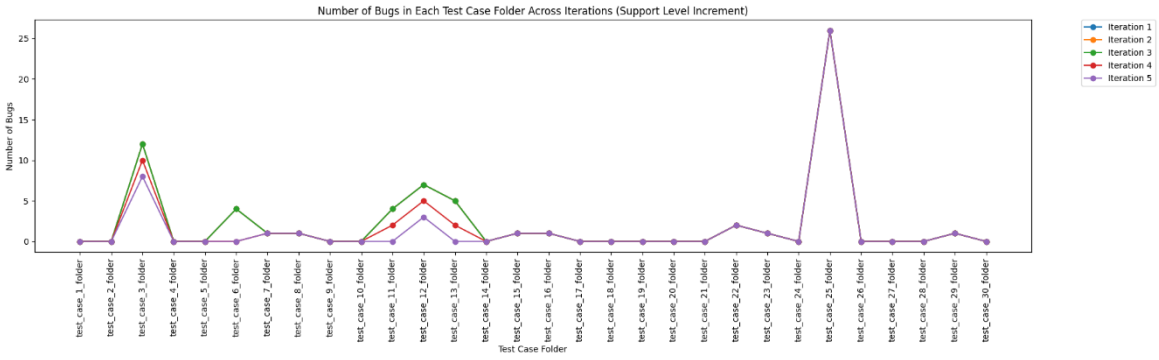
Hình 11: Kết quả thống kê tổng số lượng lỗi ở mỗi mức phân tích.

Dựa vào các kết quả thu được, số lượng lỗi có sự tăng trưởng với mức tăng dần từ 1 đến 3 nhưng chững lại khi mức phân tích cao hơn. Nhóm tác giả nhận định việc nâng cao

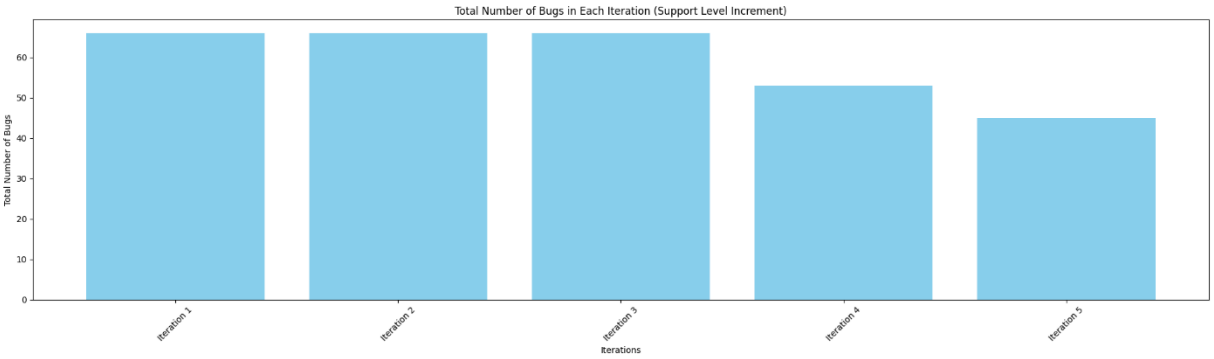
mức phân tích có thể giúp phát hiện các lỗi tiềm năng nhiều hơn nhưng việc nâng mức độ phân tích ở mức quá cao sẽ không mang lại quá nhiều tác động do điều này có thể tùy thuộc vào bộ dữ liệu thí nghiệm và các bộ thư viện đi kèm, nếu với bộ dữ liệu đơn giản và bộ thư viện được xây dựng sẵn như stdio.h, iostream, ... thì độ phức tạp giữa các hàm được gọi trên bề mặt chương trình và các lời gọi hàm ẩn bên trong thư viện không quá cao. Do đó, việc lựa chọn phân tích ở mức 2 và 3 là hoàn toàn đủ trong các trường hợp sử dụng bình thường.

Khi thực hiện tăng tuyến tính tham số support

Đối với tham số *support* sử dụng trong thử nghiệm, cũng cùng tiêu chuẩn sử dụng tương tự với mức phân tích liên thủ tục. Kết quả chạy thử nghiệm được ghi nhận như sau:



Hình 12: Kết quả thống kê với tham số ngưỡng support từ 1 đến 5.

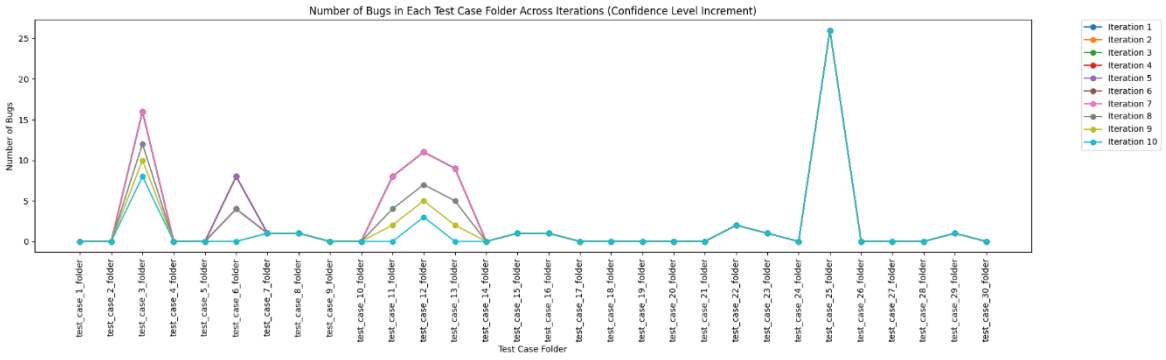


Hình 13: Kết quả thống kê tổng số lượng lỗi ở mỗi mức tham số ngưỡng support.

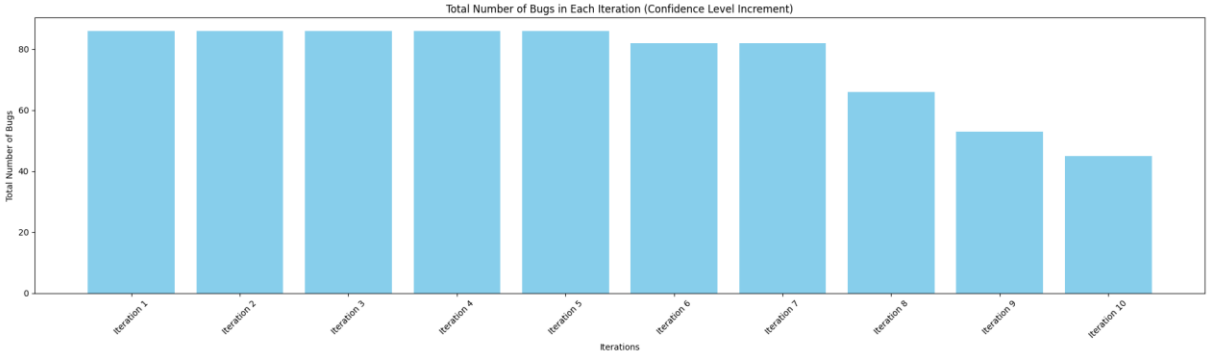
Kết quả thu thập phản ánh sự sụt giảm lỗi phát hiện, điều này có thể giải thích cho mức kỳ vọng cao về số lần xuất hiện của hàm hoặc cặp hàm trong chương trình. Do đó, nhóm tác giả đề xuất khoảng tham số lựa chọn phù hợp là từ 2 đến 5, đối mức 1 không được lựa chọn vì mức này phản ánh sự không chọn lọc với hàm và cặp hàm trong việc xác định và cục bộ hóa lỗi.

Khi thực hiện tăng tuyến tính tham số confidence

Với tham số *confidence*, kết quả thống kê thu nhận được cũng mang xu hướng giảm tương tự với tham số *support* nhưng lại có sự khác biệt ở mức 50%. Từ mốc 0% đến 50%, việc thay đổi tham số không phản ánh bất kỳ thay đổi ảnh hưởng nào lên việc phát hiện lỗi. Từ mốc 50% trở đi, việc phát hiện lỗi có sự phân hóa và lựa chọn kỹ càng hơn. Điều này để thể hiện rõ bên dưới:



Hình 14: Kết quả thống kê với tham số ngưỡng confidence từ 0 đến 90.



Hình 15: Kết quả thống kê tổng số lượng lỗi ở mỗi mức tham số ngưỡng confidence.

Chương 6. KẾT LUẬN VÀ ĐỊNH HƯỚNG PHÁT TRIỂN

6.1. Kết luận

Qua quá trình tìm hiểu và phân tích, báo cáo đã làm rõ vai trò của đồ thị lời gọi trong việc mô hình hóa cấu trúc và hành vi thực thi của chương trình, từ đó trở thành nền tảng hữu ích cho các kỹ thuật phát hiện lỗi phần mềm. Đặc biệt, khi kết hợp với phương pháp suy luận bất biến và các công cụ hỗ trợ khác, đồ thị lời gọi có thể giúp phát hiện hiệu quả nhiều loại lỗi hành vi rất khó phát hiện bằng các phương pháp truyền thống.

Báo cáo đã phân tích ba kỹ thuật giảm đồ thị lời gọi phổ biến: Total Reduction, Total Reduction with Edge Weights, và Zero-One-Many Reduction. Mỗi kỹ thuật đều có ưu và nhược điểm riêng, ảnh hưởng trực tiếp đến khả năng giữ lại thông tin thực thi, mức độ trừu tượng hóa hành vi và hiệu quả phát hiện lỗi.

Thông qua thử nghiệm, phương pháp kết hợp của báo cáo cũng cho thấy hiệu quả rõ rệt khi vừa hỗ trợ tiền phân tích và phát hiện các lỗi, vừa áp dụng phân tích và giảm kích thước đồ thị, vừa giữ được thông tin cấu trúc và tần suất, từ đó hỗ trợ tốt hơn cho việc phát hiện bất thường khác trong chương trình.

6.1.1. Thách thức và cơ hội trong việc áp dụng thực tế

Thách thức

Trong các dự án thực tế, quy mô của đồ thị lời gọi có thể nảy sinh đến hàng chục ngàn đỉnh hàm và hàng triệu cạnh lời gọi. Do đó việc xây dựng, lưu trữ và phân tích đồ thị lời gọi đòi hỏi tài nguyên tính toán lớn và kỹ thuật tối ưu dữ liệu. Bên cạnh đó, việc phát triển liên tục không ngừng nghỉ của dự án cũng phát sinh ra nhiều lời gọi thư viện hoặc các lời gọi dư thừa làm nhiễu loạn quá trình suy luận bất biến và phát hiện lỗi. Đồng thời, các hành vi được xem là "bình thường" cũng đa dạng tùy thuộc vào yêu cầu thiết kế, điều này làm cho việc suy luận có thể dẫn đến các trường hợp báo sai lỗi - *false-positive* và bỏ sót lỗi - *false-negative*.

Cơ hội

Mặc cho các thách thức kể trên, phương pháp vẫn có không ít ưu điểm đáng để áp dụng. Nó có thể tự tích hợp như một tính năng tự động hóa kiểm lỗi trong quy trình phát triển phát hành liên tục - *Continuous Integration/Continuous Deployment* - CI/CD, giúp đưa ra các cảnh báo lỗi sớm mà không cần viết thêm test case. Ngoài ra, phương pháp cũng góp phần giúp hỗ trợ bảo trì và phân tích tác động trong quá trình nâng cấp, làm sạch mã nguồn.

6.2. Định hướng phát triển

Suy luận bất biến với học máy và thống kê nâng cao

Thay thế việc sử dụng dựa trên nền tảng các điều kiện - *rule-based* hoặc thu thập thống kê đơn giản, việc suy luận bất biến có thể sử dụng tích hợp học máy để học các mẫu hành vi bình thường trên đồ thị lời gọi, hay sử dụng mạng neural đồ thị - *graph neural network* - GNN để khai thác cấu trúc đồ thị tốt hơn hoặc thay thế / kết hợp suy luận bất biến với suy luận Bayes - *Bayesian inference* để mô hình hóa trình tự lời gọi phức tạp.

Ngữ cảnh hóa bất biến - *Context-sensitive invariants*

Trong bối cảnh các phương pháp sử dụng bất biến hiện tại, chưa có các nghiên cứu để bổ sung ngữ cảnh cho các bất biến. Việc nghiên cứu hướng bất biến có ngữ cảnh cũng là một hướng đi sẽ giúp phát hiện các lỗi xảy ra chỉ khi một hàm được gọi từ một ngữ cảnh cụ thể.

TÀI LIỆU THAM KHẢO

[1] Frank Eichinger, Klemens Bohm, Matthias Huber, "Improved Software Fault Detection with Graph Mining," Helsinki, 2008.

[2] Michael D. Ernst, Jake Cockrell, William G. Griswold, David Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, tập 27, số 2, pp. 99-123, 2001.

[3] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, Philip S. Yu, "Mining Behavior Graphs for Backtrace of Noncrashing Bugs," trong *SIAM International Conference on Data Mining*, California, 2005.

[4] Frank Eichinger, Klemens Bohm, "Software-Bug Localization with Graph Mining," trong *Managing and Mining Graph Data*, Karlsruhe, Springer, 2010, pp. 515-546.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, John Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, tập 25, số 5, pp. 22-29, 2008.

[6] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian, "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach," *Proceedings of the ACM on Programming Languages*, tập 8, số OOPSLA1, pp. 474-499, 2024.

[7] Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, Anders Møller, "Indirection-Bounded Call Graph Analysis," trong *European Conference on Object-Oriented Programming (ECOOP)*, Saarland, 2024.

[8] Michael D. Bond, Graham Z. Baker, Samuel Z. Guyer, "Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses," trong *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[9] Yannis Smaragdakis, Christoph Csallner, "Combining Static and Dynamic Reasoning for Bug Detection," trong *First International Conference, Tests and Proofs - TAP*, Zurich, Springer, 2007, pp. 1-16.

[10] I. K. Isaev, D. V. Sidorov, "The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs," *Programming and Computer Software*, tập 36, số 4, pp. 225-236, 2010.

[11] Frank Eichinger, Klemens Bohm, Matthias Huber, "Mining Edge-Weighted Call Graphs to Localise Software Bugs," *Machine Learning and Knowledge Discovery in Databases*, pp. 333-348, 2008.

[12] Giuseppe Di Fatta, Stefan Leue, Evghenia Stegantova, "Discriminative Pattern Mining in Software Fault Detection," *Proceedings of the 3rd international workshop on Software quality assurance*, pp. 62-69, 2006.

[13] Swarup Kumar Sahoo, John Criswell, Chase Geigle, Vikram Adve, "Using likely invariants for automated software fault localization," trong *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2013.

[14] L. Tan, "(ECE453/CS447/ECE653/CS647/SE465) Software Testing, Quality Assurance, and Maintenance Project, Version 1," University of Waterloo, Waterloo, 2016.

[15] Edmund Wong, Irene Huang, Taiyue Liu, Lin Tan, "Tutorial 2: LLVM," Lin Tan, Waterloo.

[16] Rui Abreu, Alberto González, Peter Zoeteweij, Arjan J. C. van Gemund, "Automatic Software Fault Localization using Generic Program Invariants," trong *Proceedings of the 2008 ACM symposium on Applied computing*, Ceará, 2008.

[17] Rong Wang, Zuohua Ding, Ning Gui, Yang Liu, "Detecting Bugs of Concurrent Programs With Program Invariants," *IEEE Transactions on Reliability*, tập 66, số 2, pp. 425-439, 2017.

[18] Sudheendra Hangal, Monica S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," trong *Proceedings of the 24th International Conference on Software Engineering*, Orlando, 2002.

[19] Daniel Grahn, Junjie Zhang, "An Analysis of C/C++ Datasets for Machine Learning-Assisted Software Vulnerability Detection," trong *Proceedings of the Conference on Applied Machine Learning for Information Security*, Arlington, 2021.

[20] S. Delphine Immaculate, M. Farida Begam, M. Floramary, "Software Bug Prediction Using Supervised Machine Learning Algorithms," trong *2019 International Conference on Data Science and Communication (IconDSC)*, Bangalore, 2019.

[21] Michael Pradel, Koushik Sen, "DeepBugs: ALearning Approach to Name-Based Bug Detection," *Proceedings of the ACM on Programming Languages*, tập 2, số OOPSLA, pp. 1-25, 2018.

[22] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, Foutse Khomh, "Automatic Fault Detection for Deep Learning Programs Using Graph Transformation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, tập 31, số 1, pp. 1-27, 2021.

[23] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Chunming Hu, Yang Liu, "Detecting Condition-Related Bugs with Control Flow Graph Neural Network," trong *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Seattle, 2023.

[24] B. Ryder, "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering*, Các tập %1 cuûa %2SE-5, số 3, pp. 216-226, 1979.

[25] "Sonar Source," software bugs: developer's guide, [Trực tuyến]. Available: https://www.sonarsource.com/learn/software-bugs/. [Đã truy cập 19 May 2025].

[26] Prabhdeep Singh, Kiran Deep Singh, "An Improved Technique for Storage and Reduction of Call Graph in Computer Memory," trong *International Conference on Advancements in Engineering and Technology*, Sangrur, 2012.