**Automatic test data generation for path testing**

**using adjustment**

**1. Tóm tắt nội dung bài báo**

**2. Tổng quan**

+ Kiểm thử phần mềm và sinh dữ liệu kiểm thử phần mềm

Kiểm thử phần mềm là quá trình thực hiện một chương trình hoặc ứng dụng nào đó với mục đích là tìm kiếm các lỗi phần mềm. Nó cũng có thể được coi như là một quá trình xác nhận và chứng minh rằng một chương trình phần mềm, ứng dụng hoặc sản phẩm đáp ứng đầy đủ các yêu cầu nghiệp vụ và kỹ thuật của thiết kế và phát triển. Kiểm thử phần mềm là một phần rất quan trọng nhưng tốn nhiều công sức trong tiến trình phát triển phần mềm [1–3]. Do đó đã có nhiều nghiên cứu về kiểm thử tự động, để giảm chi phí và đạt được sự tin tưởng hơn trong kết quả thu được [4].

Công việc quan trọng trong kiểm thử phần mềm là tạo ra input dữ liệu kiểm thử [3]. Search-based test data generation mục đích là để tự động hóa công việc này, bằng cách tìm kiếm các test case (là các input data, hoặc là cả input-output) mà thỏa mãn một tiêu chuẩn kiểm thử đã được lựa chọn.

+ Phương pháp kiểm thử phần mềm sử dụng các kỹ thuật metaheuristic (Search Based Software Testing)

+ Phương pháp sinh dữ liệu kiểm thử phần mềm sử dụng metaheuristic

+ Khái niệm CFG

+ Khái niệm xác suất phủ của đường dẫn trong chương trình

**3. Bài toán sinh dữ liệu kiểm thử đường dẫn**

+ Bài toán sinh dữ liệu kiểm thử đường dẫn

Kiểm thử đường dẫn là tìm kiếm trong toàn bộ chương trình các test case phù hợp để có thể phủ được toàn bộ các đường dẫn khả thi trong chương trình được thực hiện kiểm thử (software under test (SUT)). Tuy nhiên nhìn chung là không thể thực hiện được mục tiêu này vì một số lý do. Đầu tiên là chương trình có thể có một số lượng vô hạn các đường dẫn nếu trong chương trình có các vòng lặp. Thứ hai, số lượng đường dẫn trong một chương trình là hàm mũ của số lượng các câu lệnh rẽ nhánh và nhiều đường dẫn trong số đó là không thể thực thi được. Thứ ba, số lượng test case sẽ là rất lớn vì mỗi đường dẫn có thể được phủ được bởi nhiều test case. Vì những lý do này, để có thể tạo dữ liệu kiểm thử mà phủ được toàn bộ các đường dẫn của chương trình thì bài toán sinh dữ liệu kiểm thử đường dẫn trở thành bài toán NP-đầy đủ.

Vì không thể phủ toàn bộ các đường dẫn trong một chương trình, thế nên bài toán sinh dữ liệu kiểm thử đường dẫn sẽ chọn ra một tập các đường dẫn để thực thi và tìm các dữ liệu kiểm thử để phủ được chúng. Có nhiều phương pháp sinh dữ liệu kiểm thử đã được đề xuất cho đến thời điểm hiện tại, và có thể được phân chia như là: sinh dữ liệu kiểm thử ngẫu nhiên, symbolic execution hay là các phương pháp tối thiểu hóa hàm. Sinh data ngẫu nhiên là lựa chọn các dữ liệu kiểm thử ngẫu nhiên từ tập các biến input [Cần dẫn nguồn]. Symbolic execution là phương pháp phân tích chương trình tĩnh, từ mỗi đường dẫn của chương trình sẽ có được một tập các biểu thức logic biểu diễn cho các điều kiện vị từ của đường dẫn đó, rồi sau đó dùng các constraint solver để tìm được nghiệm cho các biểu thức logic, cũng chính là dữ liệu kiểm thử để phủ đường dẫn này [cần dẫn chứng paper về Java PathFinder]. Các phương pháp tối thiểu hóa hàm, hãy cách gọi khác khác phân tích động, vì chúng dựa trên sự thực thi của chương trình. Chúng sẽ thực hiện tìm kiếm thăm dò mà trong đó các biến input được chọn sẽ được thay đổi giá trị một chút rồi chuyển lại cho chương trình cần kiểm thử [cần dẫn chứng paper của Korel].

+ Giả thuyết về hạn chế của phương pháp sinh dữ liệu kiểm thử đường dẫn sử dụng giải thuật di truyền. Giả thuyết này sẽ được kiểm chứng bằng thực nghiệm.

Trong bài báo này, giải thuật di truyền (genetic algorithm - GA) được sử dụng như là một bộ sinh dữ liệu kiểm thử. Mỗi nhiễm sắc thể biểu diễn một tập của dữ liệu kiểm thử (là một tậpcác giá trị input mà đại diện cho một test case đơn lẻ). Do đó một quần thể là một tập của các test case. Mỗi test case sẽ xác định được một đường dẫn thực thi nó, và phần lớn thì một đườngdẫn có thể được phủ bởi nhiều test case. Điều này dẫn đến là một tập các test case có thể dẫn đến toàn bộ các đường dẫn sẽ được phủ.

Các bước cơ bản của GA là (1) Initialization, (2) Evaluation, and (3) Do the following until any stopping criteria is met: (3.a) Selection,(3.b) Perturbation, and (3.c) Go back to Step (2). Ở bước (1) khi sinh quần thể đầu tiên, thì có thể khởi tạo ngẫu nhiên hoặc là sử dụng một số tri thức về quần thể đó. Bước (2) sẽ đánh giá toàn bộ cá thể của quần thể sử dụng một hàm mục tiêu cho trước. Trong (3.a) thì có một số cá thể trong quần thể được lựa chọn để đột biến bằng các xử lý tiến hóa. Bước (3.b) sẽ áp dụng các xử lý: trao đổi chéo để trộn lẫn các gen di truyền và đột biến để giới thiệu các đặc điểm di truyền mới. Bộ sinh test data giữ một danh sách của các đường dẫn mục tiêu mà chưa được phủ. Tại thời điểm bắt đầu sự tiến hóa, mọi đường dẫn mục tiêu đều nằm trong danh sách này. Ở mỗi thế hệ, mỗi dữ liệu kiểm thử trong quần thể được đánh giá (hàm mục tiêu của nó được gọi) để hướng đến các đường dẫn chưa được phủ. Khi một dữ liệu kiểm thử phủ được một đường dẫn mục tiêu, nó được ghi lại và đường dẫn mục tiêu này được loại ra khỏi danh sách.Quá trình tìm kiếm sẽ kết thúc nếu danh sách các đường dẫn mục tiêu là rỗng, hoặc là gặp phải tiêu chuẩn dừng tìm kiếm.

Tuy nhiên trong thực tế giải thuật di truyền khi áp dụng vào bài toán sinh dữ liệu kiểm thử đường dẫn là rất khó để sinh ra dữ liệu kiểm thử của các đường dẫn có xác suất thấp được phủ thấp trong chương trình. Bây giờ chúng ta xét đến bài toán phân loại tam giác Tritype như sau:

int Tritype(double a, double b, double c)

{

inttrityp = 0;

if ((a + b > c) && (b + c > a) && (c + a > b)) // branch #1

{

if ((a != b) && (b != c) && (c != a)) // branch #2

trityp = 1; // Scalene

else

if (((a == b) && (b != c)) || ((b == c) && (c != a)) ||

((c == a) && (a != b))) // branch #3

trityp = 2; // Isosceles

else

trityp = 3; // Equilateral

}

else

trityp = -1; // Not a triangle

return trityp;

}

CFG được sinh ra từ chương trình phân loại tam giác này như sau:



Tập target paths của được sinh ra từ CFG của chương trình phân loại tam giác sẽ có 4 path như sau:

{[B1,F]} // Path1: Not Triangle

{[B1,T],[B2,T]} // Path 2: Scalence

{[B1,F], [B2,F], [B3,T]} // Path3: Isosceles

{[B1,T], [B2,F], [B3,F]} // Path4: Equilateral

Trong 4 path ở trên thì xác suất có phủ được theo input data sẽ theo chiều giảm dần path 1 > path 2 > path 3 > path4. Thực tế khi sử dụng giải thuật tiến hóa để sinh test case cho program này thì sẽ sinh ra được test data phủ được path 1 và path 2, còn path 3 và path 4 thì sẽ không thể sinh ra test case để phủ được.

Run No. 1 : Wait Please .........

Path 1: a = 14.744746849514955 b = 4.817243648636693 c = 12.301081756217084

Objective call: 1

Path 2: a = 7.768767568484256 b = 1.3747887684823534 c = 2.6602222750788584

Objective call: 3

Run No. 2 : Wait Please .........

Run No. 3 : Wait Please .........

Run No. 4 : Wait Please .........

Run No. 5 : Wait Please .........

Run No. 6 : Wait Please .........

Run No. 7 : Wait Please .........

Run No. 8 : Wait Please .........

Run No. 9 : Wait Please .........

Run No. 10 : Wait Please .........

+ Giả thuyết về hiệu quả của việc kết hợp giải thuật di truyền (GA) với phân tích tĩnh. Giả thuyết này cũng sẽ được kiểm chứng bằng thực nghiệm.

Nguyên nhân của việc khó tạo ra một test case có thể phủ được các test path 3 và 4 là bởi vì GA sẽ rất khó sinh được một cá thể mà có 2 nhiễm sắc thể có giá trị giống nhau, vì xác suất tạo ra được một cặp như vậy là rất thấp, thế cho nên chúng tôi đề xuất một phương pháp kết hợp với phân tích tĩnh để có thể sinh ra được các test data phủ được các test path khó được phủ trong tập đường dẫn mục tiêu.

**4. Thiết kế các giải thuật**

+ GA và GA + Static Analysis cho bài toán sinh dữ liệu kiểm thử đường dẫn.

Step 1: Phân tích điều kiện so sánh trong các câu lệnh rẽ nhánh của chương trình

Step 2: Với các câu điều kiện rẽ nhánh này, điều chỉnh ở thủ tục sinh ra các thể mới của giải thuật di truyền.

Quay trở lại chương trình phân loại tam giác Tritype ở trên, các câu lệnh điều kiện mà cần điều chỉnh trong giải thuật di truyền là branch #2 và branch #3. Điều kiện so sánh mà cần điều chỉnh ở trong giải thuật di truyền là (a==b) và (a==b && b==c). Với các điều chỉnh này, thu được kết quả khi chạy như sau:

Run No. 1 : Wait Please .........

Path 1: a = 14.744746849514955 b = 4.817243648636693 c = 12.301081756217084

Objective call: 1

Path 2: a = 7.768767568484256 b = 1.3747887684823534 c = 2.6602222750788584

Objective call: 3

Path 3: a = 10.153558006964193 b = 10.6389422858413 c = 10.6389422858413

Objective call: 462

Run No. 2 : Wait Please .........

Path 4: a = 0.18596209465759556 b = 0.18596209465759556 c = 0.18596209465759556

Objective call: 57595

Run No. 3 : Wait Please .........

Run No. 4 : Wait Please .........

Run No. 5 : Wait Please .........

Run No. 6 : Wait Please .........

Run No. 7 : Wait Please .........

Run No. 8 : Wait Please .........

Run No. 9 : Wait Please .........

Run No. 10 : Wait Please .........

Run No. 10 : Wait Please .........

**5. Thí nghiệm và phân tích kết quả**

+ Các test programs + target paths

Bài báo này đã thực hiện thí nghiệm với các program trong bảng sau. Với các program mà có chứa vòng lặp, không giảm tính tổng quát, chúng tôi thực hiện với số lần lặp là 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | Program | Loop | Target paths | Reference |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

+ Các thông số cần phân tích: Em cần chạy mỗi test program với 2 giải thuật: GA (Không điều chỉnh), GA+ static Analysis (có điều chỉnh). Chạy 30 lần cho mỗi cặp giải thuật + test program. Ghi lại số thế hệ và lần gọi objective function của một đường dẫn mục tiêu khi nó được thực thi.

+ Phân tích kết quả: Phân tích chỉ ra GA không có khả năng phủ được các đường dẫn khó (có điều kiện bằng), trong khi GA + Static Analysis có thể. So sánh 2 giải thuật về số lượng đường dẫn trung bình phủ được trong 30 lần với mỗi test program. Phân tích chi tiết về các đường dẫn phủ được và không phủ được của cả 2 phương pháp, và thời điểm mà chúng được phủ...

**6. Kết luận và hướng phát triển**

**7. Tài liệu tham khảo**

[1] J.A. Whittaker, What is software testing? and why is it so hard?, IEEE Software 17 (1), 02/2000, 70–79.

[2] N. Mansour, M. Salame, Data generation for path testing, Software Quality Control 12(2), 06/2004, 121–136.

[3] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification & Reliability14 (2), 06/2004, 105–156.

[4] P.M.S. Bueno, M. Jino, Identification of potentially infeasible program paths by　monitoring the search for test data, Proceedings of the 15th IEEE　International Conference on Automated Software Engineering 2000 (ASE2000), IEEE Computer Society, Grenoble, France, 09/2000, 209–218.

[x] I. Hermadi, C. Lokan, R. Sarker, Dynamic stopping criteria for search-based test data generation for path testing, Information and Software Technology, 56 (4), 04/2014,395–407.