**Parallel Particle Swarm Optimization-based**

**Test Data Generation for Path Coverage Testing**

**Abstract**

# 1. Introduction

Software has widely affected our work and lives, and brought us great convenience. However, its failure may lead to significant economic loss or threat to life safety. For example, the explosion incident of Ariane-V rocket [1] and the BP deepwater horizon disaster [2] are the typical instances of such failures in the past years. As a consequence, software quality has become an important concern for people in the current information society.

Testing has been proved as one of the effective approaches to ensure and improve software quality over the past three decades. Tuy nhiên, phần lớn các công việc của software testing đều được thực hiện bằng manually, gây tốn kém rất nhiều effort và cost trong quá trình phát triển phần mềm.

Trong những năm gần đây, meta-heuristic search (MHS) techniques được áp dụng rộng rãi trong lĩnh vực software testing, hình thành nên một xu hướng nghiên cứu được gọi là search-based software testing (SBST) [7], đặc biệt là cho bài toán automatic test data generation. The general idea behind search-based test data generation is to select a set of test cases from program input space to meet the testing requirement. The basic procedure is to generate a test suite that covers a specific test adequacy criterion, which is usually expressed as a fitness function. When a coverage criterion is selected, the search activity should attempt to produce a test suite which can cover all construct elements mentioned in the criterion. Meanwhile, testing is a time-consuming and labor-intensive activity. Hence, the size of test suite should be as small as possible to reduce the testing effort and cost.

Among the existing meta-heuristic search techniques, such as simulated annealing (SA) and generic algorithm (GA), are the most popular algorithms, and have been widely adopted for generating test data. Although they can produce test data with appropriate fault-prone ability [7,8], they fail to produce them quickly due to their slow evolutionary speed. Recently, as a swarm intelligence technique, particle swarm optimization (PSO) [9,10] has become a hot research topic in the area of intelligent computing. Its significant feature is the simplicity and fast convergence speed.

Xem xét một program under test như sau:

int getDayNum(int year, int month)

{

int maxDay=0;

if(month≥1 && month≤12) //bch1: branch 1

{

if(month=2) //bch2: branch 2

{

if(year%400=0||(year%4=0&&year%100=0))

//bch3: branch 3

maxDay=29;

else //bch4: branch 4

maxDay=28;

}

else if(month=4||month=6||month=9||month=11)

//bch5: branch 5

maxDay=30;

else //bch6: branch 6

maxDay=31;

}

else //bch7: branch 7

maxDay=-1;

return maxDay;

}

Trong paper này, chúng tôi vẫn sử dụng PSO để sinh test data cho một program under test. Tuy nhiên khác với Mao [1], cách tiếp cận của chúng tôi là, với mỗi branch predicate, thì tạo ra một fitness function. Sau đó chúng tôi sẽ sử dụng parallel particle swarm optimization để tìm nghiệm thỏa mãn cho mỗi fitness function này, có nghĩa là sinh được test data cho branch predicate này.

# 2. Background

## 2.1. Fitness function

Table 1. The branch functions for several kinds of branch predicates

|  |  |  |
| --- | --- | --- |
| No | Predicate | Branch distance function *f*(bch*i*) |
| 1 | Boolean | If *true* then 0 else *k* |
| 2 | ¬a | Negation is propagated over *a* |
| 3 | a = b | If abs*(a* − *b)* = 0 then 0 else abs*(a* − *b)* + *k* |
| 4 | a ≠ b | If abs*(a* − *b)* ≠ 0 then 0 else *k* |
| 5 | a < b | If *a* − *b <* 0 then 0 else abs*(a* − *b)* + *k* |
| 6 | a ≤ b | If *a* − *b* ≤ 0 then 0 else abs*(a* − *b)* + *k* |
| 7 | a > b | If *b* − *a >* 0 then 0 else abs*(b* − *a)* + *k* |
| 8 | a ≥ b | If *b* − *a* ≥ 0 then 0 else abs*(b* − *a)* + *k* |
| 9 | a and b | *f (a)* + *f (b)* |
| 10 | a or b | min*( f (a), f (b))* |

Dựa vào công thức tính này chúng tôi sẽ phát triển thành function tính giá trị tại các decision node được trình bày ở phần sau.

## 2.2. Particle Swarm Optimization

Particle Swarm Optimization (PSO) được giới thiệu lần đầu tiên vào năm 1995 bởi Kennedy and Eberhart, và hiện nay được ứng dụng rộng rãi trong các bài toán tối ưu hóa. PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called gbest.

After finding the two best values, the particle updates its velocity and positions with following equation (a) and (b).  
v[] = v[] + c1 \* rand() \* (pbest[] - present[]) + c2 \* rand() \* (gbest[] - present[]) (a)

present[] = persent[] + v[] (b)

v[] is the particle velocity, persent[] is the current particle (solution). pbest[] and gbest[] are defined as stated before. rand () is a random number between (0,1). c1, c2 are learning factors, usually c1 = c2 = 2.

PSO algorithm được mô tả như sau:

|  |
| --- |
| **Algorithm 1**: Particle Swarm Optimization (PSO) |
| **Input:** Fitness function |
| **Output:** The best solution |
| 1: **for each** particle |
| 2:    initialize particle |
| 3: **end for** |
| 4: **do** |
| 5: **for each** particle |
| 6: calculate fitness value |
| 7: **if** the fitness value is better than the best fitness value (pBest) in history then |
| 8:   set current value as the new pBest |
| 9: **end if** |
| 10: **end for** |
| 11: choose the particle with the best fitness value of all the particles as the gBest |
| 12: **for each** particle |
| 13: calculate particle velocity according equation (a) |
| 14:   update particle position according equation (b) |
| 15: **end for** |
| 16: **while** maximum iterations or minimum error criteria is not attained |

Particles' velocities on each dimension are clamped to a maximum velocity Vmax. If the sum of accelerations would cause the velocity on that dimension to exceed Vmax, which is a parameter specified by the user. Then the velocity on that dimension is limited to Vmax.

# 3. Related work

Trong thập niên 1990, genetic algorithm là phương pháp được chọn lựa để sinh test data. Jones et al[2] và Pargas et al [3] đã nghiên cứu để sử dụng GA trong việc sinh ra test case để conver được các nhánh trong một program. Các thực nghiệm cũng đã cho thấy rằng với các small program thì GA có hiệu quả hơn hẳn so với các thuật toán tìm kiếm ngẫu nhiên. Trong những năm gần đây, McMinn [3] and Harman [4] đã thực hiện các nghiên cứu thực nghiệm dựa trên GA để sinh test data cho các large-scale programs, và cho thấy hiệu quả của nó hơn hẳn các thuật toán tìm kiếm meta-heuristic search khác.

Mặc dù đã đạt được các tiến bộ đáng kể trong việc sinh ra test data, nhưng GA có một nhược điểm là tốc độ sinh test data để phủ được toàn bộ các branch của một program under test còn hạn chế. Do đó, phương pháp PSO, được đề xuất bởi Kennedy and Eberhart [5], có thể giải quyết được nhược điểm này.

Windisch et al. [17] là nhóm tác giả đầu tiên áp dụng PSO vào việc automatic test data generation. Họ đã cải tiến PSO thành comprehensive learning particle swarm optimization (CL-PSO) để sinh generate structural test data, but some experiments have confirmed that the convergence speed of CL-PSO is perhaps worse than the basic PSO.

Mao [1] không cải tiến gì cho PSO, mà tạo ra hàm fitness function bằng cách sử dụng branch functions for branch predicates kết hợp với branch weight của program under test, rồi sau đó dùng PSO để tìm nghiệm cho fitness function này. Kết quả thực nghiệm với 1 benchmark gồm có 8 program under test, phương pháp này đã chứng minh được rằng giải thuật PSO có hiệu quả hơn giải thuật GA trong việc sinh test data. Tuy nhiên vẫn có nhược điểm là việc tính toán các branch weight cho một program under test vẫn hoàn toàn là công việc manual. Trong paper này, đề xuất của chúng tôi sẽ khắc phục được hạn chế này, trong khi vẫn đảm bảo hiệu quả của phương pháp automatic test data generation bằng PSO.

# 4. Proprosed approach

Cách tiếp cận của chúng tôi được thể hiện qua sơ đồ sau:



## 4.1. Perform statistical analysis to find out all test paths

Đầu tiên, chúng tôi thực hiện statistical analysis để tìm toàn bộ các test paths của program under test. Việc này có thể được thực hiện bằng 2 bước nhỏ sau đây:

*1) Control flow graph generation:* Test case generation from source code directly is more complicated and difficult than from CFG. CFG is a directed graph visualizing logic structures of program simplify [8] and defined as follow:

**Definition 1 (CFG).** Given a function, a corresponding CFG is defined as a pair G = (V, E), where V = {v0, v1…, vn} is a set of vertices representing statements, E = {(vi, vj)|vi, vj V} V x V is a set of edges. Each edge (vi, vj) implies the statement corresponding to vj is executed after vi.

|  |
| --- |
| **Algorithm 2**: Generate CFG |
| **Input** : f : source code |
| **Output**: graph: CFG |
| 1: B = a set of blocks by dividing f |
| 2: G = a graph by linking all blocks in B to each other |
| 3: update graph by replacing f with G |
| 4: **if** G contains *return/break/continue* statements **then** |
| 5: update the destination of *return/break/continue* pointers in the graph |
| 6: **end if** |
| 7: **for** each block M in B do |
| 8: **if** block M can be divided into smaller blocks **then** |
| 9: Generate CFG(M) |
| 10: **end if** |
| 11: **end for** |

Với program under test getDayNum() bên trên, khi áp dụng giải thuật Generate CFG thì sẽ thu được một CFG như hình bên dưới.



*2) Test paths generation:*

**Definition 2 (Path).** Given a CFG G = (V, E), a path is a sequence of vertices {v0, v1,..., vk | (vi, vi+1) E, 0 < k < n} , where n is the number of vertices.

**Definition 3 (Test path).** Given a CFG G = (V, E), a test path is a path {v0, v1,..., vk | (vi, vi+1) E}, where v0 and vi+1 are corresponding to the start vertex and end vertex of the CFG.

|  |
| --- |
| **Algorithm 3**: Traverse CFG |
| **Input** : v: the initial vertex of the CFG  depth: the maximum number of iterations for a loop  path: a global variable used to store a discovered test path |
| **Output**: P: a set of feasible test paths |
| 1: **if** v = NULL or v is the end vertex then |
| 2: add path to P |
| 3: **else if** the number occurrences of v in path ≤depth **then** |
| 4: add v to the end of path |
| 5: **if** (v is not a decision) **or** (v is decision and path is feasible) **then** |
| 6: **for each** adjacent vertex u to v **do** |
| 7: TraverseCFG(u, depth, path) |
| 8: **end for** |
| 9: **end if** |
| 10: remove the latest vertex added in path from it |
| 11: **end if** |

Sử dụng thuật toán này cho program under test ở trên, sẽ thu được 5 test paths được biểu diễn bởi các decision như sau

|  |  |  |
| --- | --- | --- |
| No | PathID | Path’s decision nodes |
| 1 | path1 | [(month≥1 && month≤12), T], [(month=2), T], [(year%400=0||(year%4=0&&year%100=0)), T] |
| 2 | path2 | [(month≥1 && month≤12), T], [(month=2), T], [(year%400=0||(year%4=0&&year%100=0)), F] |
| 3 | path3 | [(month≥1 && month≤12), T], [(month=2), F], [(month=4||month=6||month=9||month=11), T] |
| 4 | path4 | [(month≥1 && month≤12), T], [(month=2), F], [(month=4||month=6||month=9||month=11), F] |
| 5 | path5 | [(month≥1 && month≤12), F] |

## 4.2. Thiết lập fitness function cho mỗi test path

Từ công thức tính branch distance, chúng tôi phát triển thành function bên dưới để tính giá trị tại các decision node.

|  |
| --- |
| **Algorithm 4**: Branch distance function (fBchDist) |
| **Input:** double a, condition type, double b |
| **Output:** Branch distance value |
| 1: **switch** (condition type)  2: **case** “=”: |
| 3: if abs(*a* − *b*) = 0 then retrun 0 else return (abs(*a* − *b*) + *k*) |
| 4: **case** “≠”: |
| 5: if abs*(a* − *b)* ≠ 0 then return 0 else return *k* |
| 6: **case** “<”: |
| 7: if *a* − *b <* 0 then return 0 else return (abs*(a* − *b)* + *k*) |
| 8: **case** “≤”: |
| 9: if *a* − *b* ≤ 0 then return 0 else return (abs*(a* − *b)* + *k*) |
| 10: **case** “>”: |
| 11: if *b* − *a >* 0 then return 0 else return (abs*(b* − *a)* + *k*) |
| 12: **case** “≥”: |
| 13 if *b* − *a* ≥ 0 then return 0 else return (abs*(b* − *a)* + *k*) |
| 14: **end switch** |

Do mỗi test path được biểu diễn bằng các decision, thế nên để thiết lập fitness function cho test path, chúng tôi thiết lập các fitness function cho mỗi decision node của test path đó. Mỗi decision node sẽ có 2 khả năng là TRUE(T) và FALSE(F), thế cho nên sẽ có 2 fitness function cho mỗi decision node tương ứng với 2 khả năng này. Công thức tính fitness function cho mỗi decision node thì ta áp dụng giải thuật tính branch distance ở trên.

|  |  |  |  |
| --- | --- | --- | --- |
| No | Decision node | Fitness function | ID |
| 1 | [(month ≥ 1 && month ≤ 12), T] | fBchDist(month, ≥, 1) + fBchDist(month, ≤, 12) | F1T |
| 2 | [(month ≥ 1 && month ≥ 12), F] | min(fBchDist(month, <, 1), fBchDist(month, >, 12)) | F1F |
| 3 | [(month = 2), T] | fBchDist(month, =, 2) | F2T |
| 4 | [(month = 2), F] | fBchDist(month, ≠, 2) | F2F |
| 5 | [(year%400=0||  (year%4=0&&year%100=0)), T] | min(fBchDist(year%400, =, 0),  (fBchDist(year%4, =, 0) + fBchDist(year%100, =, 0))) | F3T |
| 6 | [(year%400=0||  (year%4=0&&year%100=0)), F] | fBchDist(year%400, ≠, 0) + min(fBchDist(year%4, ≠, 0), fBchDist(year%100, ≠, 0)) | F3F |
| 7 | [(month=4||month=6||  month=9||month=11), T] | min(fBchDist(month, =, 4), fBchDist(month, =, 6), fBchDist(month, =, 9), fBchDist(month, =, 11)) | F4T |
| 8 | [(month=4||month=6||  month=9||month=11), F] | fBchDist(month, ≠, 4) + fBchDist(month, ≠, 6) +  fBchDist(month, ≠, 9) + fBchDist(month, ≠, 11) | F4F |

Từ công thức tính fitness cho mỗi decision node, ta hàm fitness cho mỗi test path như sau:

|  |  |  |
| --- | --- | --- |
| No | PathID | Test path fitness functions |
| 1 | path1 | f1 = F1T + F2T + F3T |
| 2 | path2 | f2 = F1T + F2T + F3F |
| 3 | path3 | f3 = F1T + F2F + F4T |
| 4 | path4 | f4 = F1T + F2F + F4F |
| 5 | path5 | f5 = F1F |

## 4.3. Parallel Particle Swarm Optimization

Với mỗi fitness function của mỗi test path, chúng tôi sử dụng một PSO để tìm nghiệm cho nó. Để có thể đồng thời tìm được nghiệm cho tất cả các fitness function, chúng tôi thực hiện song song hóa PSO. Để PSO có thể chạy song song được, thì ta cần định nghĩa nó như là 1 class extend class Thread của Java như sau:

public class PSOProcess extends Thread {}

Việc song song hóa PSO có thể tiến hành bởi giải thuật như sau

|  |
| --- |
| **Algorithm 5**: Parallel Particle Swarm Optimization |
| **Input:** list of fitness function |
| **Output:** test data for each fitness function |
| 1: **for each** fitness function  2: Khởi tạo một object pso của class PSOProcess |
| 3: Gán fitness function cho object pso |
| 4: Thực thi object pso: pso.start(); |
| 5: **end for** |

# 5. Experimental result

So sánh với kết quả của Mao[1] theo các benchmark và tiêu chí mà Mao [1] đã sử dụng. Hai tiêu chí được mang ra so sánh là:

* Success rate (SR) is the probability of all branches which can be covered by the generated test data.
* Average coverage (AC) is the average of the branch coverage achieved by all test inputs in 1,000 runs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program under test | Success rate (%) | | Average coverage (%) | |
| Mao[10]’s PSO | MOPSO | Mao[10]’s PSO | MOPSO |
| triangleType | 99.80 | 100.0 | 99.94 | 100.0 |
| calDay | 100.0 | 100.0 | 100.0 | 100.0 |
| cal | 100.0 | 100.0 | 100.0 | 100.0 |
| remainder | 100.0 | 100.0 | 100.0 | 100.0 |
| computeTax | 99.80 | 100.0 | 99.98 | 100.0 |
| bessj | 100.0 | 100.0 | 100.0 | 100.0 |
| printCalendar | 99.10 | 100.0 | 99.72 | 100.0 |
| line | 99.20 | 100.0 | 99.86 | 100.0 |

# 6. Conclusion

**References**

1. Cargill, T.: Exception handling: a false sense of security. C++ Rep. **6**(9), 423–431 (1994). http://www.awprofessional.com/content/images/020163371x/supplements/ExceptionHandlingArticle.html

2. Shafer, D.; Laplante, P.A.: The bp oil spill: could software be a culprit? IT Prof. **12**(5), 6–9 (2010)

1. C. Mao: Generating Test Data for Software Structural Testing Based on Particle Swarm Optimization. Arabian Journal for Science and Engineering, vol 39, issue 6, pp 4593–4607 (June 2014).

2. Jones, B.F.; Sthamer, H.H.; Eyres, D.E.: Automated structural testing using genetic algorithms. Softw. Eng. J. **11**(5), 299–306 (1996)

3. Pargas, R.P.; Harrold, M.J.; Peck, R.: Automated structural testing using genetic algorithms. Softw. Test. Verif. Reliab. **9**(4), 263–282 (1999)

4. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14, 105–156 (2004)

5. Harman, M.; McMinn, P.: A theoretical and empirical study of search-based testing: local, global, and hybrid search. IEEE Trans. Softw. Eng. 36(2), 226–247 (2010)

6. Kennedy, J.; Eberhart, R.C.: Particle swam optimization. In: Proceedings of IEEE International Conference on Neural Networks (ICNN’95), pp. 1942–1948 (1995)

7. McMinn, P.: Search-based software testing: past, present and future. In: Proceedings of ICSE Workshop on the Search-Based Software Testing (SBST’11), pp. 153–163 (2011)

17. Windisch, A.; Wappler, S.; Wegener, J.: Applying particle swarm optimization to software testing. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO’07), pp. 1121–1128 (2007)