**Parallel Particle Swarm Optimization-based**

**Test Data Generation for Path Coverage Testing**

**Abstract**

Automatic test data generation is still a problem attracting much interest in software testing. The Particle Swarm Optimization (PSO) approach is a swarm intelligence technique which can be used to generate test data automatically for path coverage testing. This paper proposes the approach of the Parallel Particle Swarm Optimization (PPSO) so that test data can be generated simultaneously for each test path of the given program under test (PUT). The proposed approach is also applied to some PUTs of the given benchmark. Experimental results demonstrate that PPSO which can generate suitable test data has higher path coverage than the previous one.

# 1. Introduction

Phần mềm đang ngày càng được ứng dụng rộng rãi và xuất hiện trong mọi ngõ ngách công việc và cuộc sống của chúng ta, mang lại những tiện ích vô cùng to lớn. Tuy nhiên, trong những năm qua, những failure của nó có thể dẫn đến những hệ lụy khôn lường cả về kinh tế lẫn tính mạng con người, điển hình như là the explosion incident of Ariane-V rocket [1] and the BP deepwater horizon disaster [2]. Chính vì vậy, như một lẽ tự nhiên, chất lượng của phần mềm đã và đang trở thành một mối quan tâm hàng đầu cho con người trong xã hội thông tin hiện nay.

Software testing đã và đang chứng minh được là một trong các phương thức hiệu quả để đảm bảo và nâng cao chất lượng phần mềm trong vài thập niên trở lại đây. 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. Do đó làm sao để tự động hóa được quá trình testing vẫn là một bài toán mở hiện nay.

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) [3], 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 which is usually expressed as a fitness function. When a coverage criterion is selected as the testing requirement, the search activity should attempt to produce a test suite which can cover all construct elements mentioned in the criterion.

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 generate test data with appropriate fault-prone ability [4, 5], they fail to produce them quickly due to their slow evolutionary speed. Recently, as a swarm intelligence technique, particle swarm optimization (PSO) [6, 7, 8] has become a hot research topic in the area of intelligent computing. Its significant feature is the simplicity and fast convergence speed.

Mặc dù vậy, các nghiên cứu hiện nay về sử dụng PSO trong test data generation vẫn còn có những hạn chế nhất định. Ví dụ, hãy xem xét một program under test đã được sử dụng trong paper của Mao [9] 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;

}

Đối với program under test này, Mao đã sử dụng PSO để sinh test data bằng cách xây dựng một fitness function duy nhất là sự kết hợp của công thức Korel [10] với lại các branch weight. Đề xuất này có 2 nhược điểm, đó là việc xây dựng các hàm branch weight được thực hiện hoàn toàn bằng manual, và có một số PUT thì ko thể sinh được test data phủ hết tất cả các test path được. Để khắc phục nhược điểm này, trong paper này chúng tôi vẫn sử dụng PSO để sinh test data cho một program under test. Tuy nhiên khác với Mao [9], cách tiếp cận của chúng tôi là, với mỗi test path thì gán cho nó 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 test path này.

# 2. Background

## 2.1. Fitness function

When using PSO, a test path coverage test data generation is transformed into an optimization problem. To cover a test path during execution, we must find appropriate values for the input variables which satisfy related branch predicates. The usual way is to use Korel’s branch distance function [10]. As a result, generating test data for a desired branch is transformed into searching input values which minimizes the return value of its Korel function. Table 1 gives some common formulas which are used in branch distance functions. To generate test data for a desired path P, we define a fitness function F(P) as the sum of all related branch distance functions. For these reasons, generating path coverage test data can be converted into searching input values which can minimize the return value of function F(P).

**Table 1.** Korel’s 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*)) |

In Korel function, k is the smallest step between 2 operands in the condition. Giống như Mao, chúng tôi cũng thiết lập giá trị cho k = 0.1. 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 [11], và hiện nay được ứng dụng rộng rãi trong các bài toán tối ưu hóa. So với các giải thuật tìm kiếm tối ưu khác như là GA hay SA, thì PSO có ưu điểm là tốc độ hội tụ nhanh hơn và được cài đặt dễ dàng hơn. 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).  
(a)

(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). *c*1, *c*2 are learning factors, usually *c*1 = *c*2 = 2.

PSO algorithm được mô tả như sau:

|  |
| --- |
| **Algorithm 1**: Particle Swarm Optimization (PSO) |
| **Input:** *F*: Fitness function |
| **Output:** *gBest*: 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 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

Giải thuật PSO được đề xuất bởi Kennedy and Eberhart [11], ban đầu chỉ là một thuật toán sử dụng cho các bài toán tối ưu hóa. Tuy nhiên với ưu điểm vượt trội về tốc độ hội tụ và cài đặt dễ dàng so với các thuật toán tối ưu khác, nên đã nhanh chóng được áp dụng như là một thuật toán tìm kiếm meta-heuristic search vào các bài toán automatic test data generation.

Windisch et al. [6] 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.

Jia et al. [7] create an automatic test data generating tool named particle swarm optimization data generation tool (PSODGT). The PSODGT is characterized by the following two features. First, the PSODGT adopts the condition-decision coverage (C/DC) as the criterion of software testing, aiming to build an efficient test data set that covers all conditions. Second, the PSODGT uses a particle swarm optimization (PSO) approach to generate test data set. In addition, a new position initialization technique is developed for PSO. Instead of initializing the test data randomly, the proposed technique uses the previously-found test data that can reach the target condition as the initial positions so that the search speed of PSODGT can be further accelerated. The PSODGT is tested on four practical programs.

Mao [9] và Zhang et al. [8] đều có cách tiếp cận giống nhau, đó là không cải tiến gì cho PSO, mà chỉ xây dựng một 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, đề xuất 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, chính điều này làm giảm tính automatic của đề xuất. 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:



**Fig. 1.** The basic steps for PPSO-based test data generation

## 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 [12] and defined as follow:

**Definition 1 (CFG).** *Given a function, a corresponding CFG is defined as a pair G =* (*V*, *E*), *where V* ={*v*0*, v1*,…*vn*} *is a set of vertices representing statements, E =* {(*vi, vj*)*|vi, vj V*} *V V is a set of edges. Each edge* (*vi*, *vj*) *implies the statement corresponding to vj is executed after vi.*

|  |
| --- |
| **Algorithm 2**: GenerateCFG |
| **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: GenerateCFG(*M*) |
| 10: **end if** |
| 11: **end for** |

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



**Fig. 2.** CFG of PUT getDayNum

*2) Test paths generation:*

In order to generate test data, a set of feasible test paths is discovered by traversing the given CFG. Path and test path are defined as follows:

**Definition 2 (Path).** *Given a CFG G =* (*V, E*)*, a path is a sequence of vertices* {*v0, v*1*,..., 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* {*v*0*, v*1*,..., vk |* (*vi, vi*+1) *E*}*, where v*0 *and vi+*1 *are corresponding to the start vertex and end vertex of the CFG.*

|  |
| --- |
| **Algorithm 3**: TraverseCFG |
| **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

**Table 2.** All test paths of PUT getDayNum

|  |  |  |
| --- | --- | --- |
| 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 xây dựng 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.

**Table 3.** Fitness functions for each decision node of PUT getDayNum

|  |  |  |  |
| --- | --- | --- | --- |
| 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 có fitness function cho mỗi test path như sau:

**Table 4.** Fitness functions for test path of PUT getDayNum

|  |  |  |
| --- | --- | --- |
| 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(PPSO) |
| **Input:** list of fitness function |
| **Output:** test data for each fitness function |
| 1: **for each** fitness function fi  2: Khởi tạo một object psoi của class PSOProcess |
| 3: Gán fitness function cho object psoi |
| 4: Execute object pso: pso.start(); |
| 5: **end for** |

# 5. Experimental analysis

Chúng tôi so sánh với kết quả thực hiện chúng tôi với đề xuất của Mao [9] theo 2 tiêu chí là khả năng automatic test data generation và hiệu quả coverage của mỗi đề xuất với mỗi program under test.

## 5.1. Khả năng automatic

Khi đề cập đến một phương pháp sinh test data tự động thì khả năng “tự động” được bao nhiêu phần là một trong các tiêu chí then chốt quyết định hiệu quả của một đề xuất. Mao [9] chỉ sử dụng duy nhất 1 hàm fitness để sinh test data cho toàn bộ các test path của một PUT, do đó phải kết hợp thêm branch weight cho từng test path vào trong fitness function. Việc xây dựng hàm branch weight là một công việc hoàn toàn mang tính manual, và đôi khi với các PUT dài và phức tạp thì còn khó hơn cả việc sinh test data cho các test path, do đó điều này đã ảnh hưởng đến hiệu quả của thuật toán.

Theo chiều ngược lại, lợi dụng tính hội tụ nhanh của PSO algorithm, chúng tôi đề xuất giải pháp sử dụng mỗi fitness function cho mỗi test path, rồi thực hiện song song PSO để tìm nghiệm cho các fitness function này. Đề xuất này có những lợi ích rõ ràng như sau:

1, Không cần phải xây dựng hàm branch weight, thế cho nên là tính tự động hóa của đề xuất sẽ được nâng cao hơn

2, Các hàm fitness function được xây dựng tự động dựa vào các decision node của mỗi test path, và các decision node này có thể được hoàn toàn tạo ra tự động từ một PUT với các algorithm 2 và 3 đã trình bày ở phần trên. Điều này rõ ràng đã nâng cao hơn khả năng automatic trong của đề xuất của chúng tôi.

## 5.2. Khả năng coverage

Hai tiêu chí được mang ra so sánh với kết quả của Mao [9] là:

* Success rate (SR) is the probability of all branches which can be covered by the generated test data. Để thực hiện kiểm nghiệm kết quả thực tế theo tiêu chí này, chúng tôi đã thực hiện PPSO 1000 lần, và tính số lần sinh được test data phủ được hết các test path. Công thức của SR được tính như sau:
* Average coverage (AC) is the average of the branch coverage achieved by all test inputs in 1,000 runs. Cũng tương tự như ở trên, để thực hiện kiểm nghiệm kết quả thực tế theo tiêu chí này, chúng tôi đã thực hiện PPSO 1000 lần, và tính coverage trung bình cho mỗi lần chạy. Công thức của AC được tính cho mỗi PUT như sau:

Kết quả so sánh chi tiết cả 2 tiêu chí này với các PUT của benchmark mà Mao [9] đã sử dụng được thể hiện chi tiết qua bảng sau:

**Table 5.** Comparison between Mao's approach and PPSO

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program under test | Success rate (%) | | Average coverage (%) | |
| Mao[10]’s PSO | PPSO | Mao[10]’s PSO | PPSO |
| 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 |

Do mỗi test path được gán với mỗi PSO, cho nên điều này đảm bảo mỗi lần chạy PPSO thì lúc nào mỗi PSO cũng sinh được test data mà phủ được test path mà nó đang gắn vào.

# 6. Conclusion

This paper has introduced and evaluated a parallel PSO approach for the branch coverage test adequacy criterion of software testing. Chúng tôi đưa ra một đề xuất là sẽ gán một hàm fitness của PSO cho mỗi test path của một PUT, sau đó thực hiện song song hóa các PSO này để sinh ra được các test data phủ cho các test path của một PUT. Các phân tích kết quả thực nghiệm cho thấy đề xuất của chúng tôi có hiệu quả hơn các phương pháp sinh test data sử dụng PSO đang có hiện tại, xét cả trên khía cạnh tự động hóa lẫn khả năng coverage cho một PUT.

In future work, some issues should be incorporated into deep investigation. The search capability of PSO algorithm could be enhanced through absorbing some other strategies in intelligent computing. To exploit more reasonable form of fitness function is also a valuable research topic. At present, we only display the results of some benchmark programs from academe. So the experiments on some industrial programs are worthy of being deeply studied.

**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)
3. 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)
4. M. A. Ahmed and I. Hermadi: GA-based Multiple Paths Test Data Generator. Computers & Operations Research, vol. 35, pp 3107--3124 (2008).
5. J. Malburg and G. Fraser: Search-based testing using constraint-based mutation. Journal Software Testing, Verification & Reliability, vol. 24(6), 472--495 (2014).
6. 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)
7. Yanli Zhang, Aiguo Li, "Automatic Generating All-Path Test Data of a Program Based on PSO", vol. 04, pp. 189-193, 2009, doi:10.1109/WCSE.2009.98
8. Ya-Hui Jia, Wei-Neng Chen, Jun Zhang, Jing-Jing Li, Generating Software Test Data by Particle Swarm Optimization, in the Proceedings of 10th International Conference, SEAL 2014, Dunedin, New Zealand, December 15-18, 2014
9. 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).
10. B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, vol. 16, 870-879 (1990).
11. Kennedy, J.; Eberhart, R.C.: Particle swam optimization. In: Proceedings of IEEE International Conference on Neural Networks (ICNN’95), pp. 1942–1948 (1995)
12. Robert Gold, Control flow graph and code coverage, in: Int. J. Appl. Math. Comput. Sci., Vol. 20, No. 4, 2010, pp. 739-749