Int J Syst Assur Eng Manag (Jan-Mar 2015) 6(1):71–77
DOI 10.1007/s13198-014-0244-3

ORIGINAL ARTICLE

Understanding the effect of time-constraint bounded novel technique for regression test selection and prioritization

Bharti Suri · Shweta Singhal

Received: 9 May 2013 / Published online: 25 February 2014

Abstract It is the demand of our ever-advancing IT industry that software be updated in order to continue its use. Such a modification should not introduce any unwanted new faults in the system. For this, the existing test suite needs to be rerun, often called as regression testing. The main challenge during the regression testing process is not to exceed the desired time and budge deadlines. As a consequence various techniques such as test case selection, minimization and prioritization are used. This paper proposes and analyzes the effect of time constraint on an ant colony optimization based technique for Regression test selection and prioritization. It has been found that with an increase in the applied time constraint, there are more chances to get an optimum selected and prioritized test suite. Also it was found that the complexity of our algorithm depends on the size of the test suite and the applied time constraint and is independent of the number of faults being mutated or any other input variable.

Keywords Regression test selection · Test case prioritization · Ant colony optimization (ACO) · Time-constraint · Analysis

1 Introduction

The ever-demanding IT industry necessitates the updating of software in order to continue its use. Such a modification must not induce any unwanted changes in the system. Regression testing ensures that modifications made to the software are correct and have not adversely affected the unmodified parts of the software. The test suites build already for the earlier versions of software become extremely costly to be run entirely. For reducing the cost of regression testing, various techniques have been proposed by researchers. These include regression test selection (Graves et al. 2001; Rothermel et al. 2000), test case prioritization (Elbaum et al. 2004; Krishnamoorthi et al. 2009; Rothermel et al. 1999, 2001; Walcott et al. 2006), and combined or hybrid approach (Singh et al. 2006). Selective regression testing technique allows only a subset of the original test suite to be selected and re run. Prioritization provides an ordering of the test cases such that the test case with higher priority is executed before than the test cases with lower priority. This priority for each test case is set according to a pre-chosen criterion that increases the rate of fault detection or achieves greatest code coverage or acquires some other important feature(s) earlier. Within the given short time-budgets, we require a time-constraint based prioritization technique. The test case prioritization problem combined with a time-constraint can be reduced to the NP-complete problem of 0/1 knapsack (Ayari et al. 2007). Various combinatorial optimization problems are NP-complete. Ant Colony Optimization (ACO) is a general purpose meta-heuristic intelligent approach widely used in solving combinatorial optimization problems including travelling salesman problem, telecommunication networks, vehicle routing, and data mining (Ayari et al. 2007; Di Caro and Dorigo 1998a, b, 1996; Gomez and Baren 2005; Li and Peng Lam 2005; Li et al. 2008; Parpinelli et al. 2002; Zhao et al. 2006) etc. Thus an attempt had been made to apply ACO to the time-constraint based regression

B. Suri (✉) · S. Singhal
USICT, G.G.S.I.P.University, Dwarka, Delhi, India
e-mail: bhartisuri@gmail.com

S. Singhal
e-mail: miss.shweta.singhal@gmail.com

test prioritization problem (Singh et al. 2010). This paper formally reproposes the technique and analyses the effect of time constraint on an ACO based regression test selection and prioritization technique. The technique gave motivating results with respect to the reduction in the test suite size and the execution time. Also, the experiment on 8 test programs resulted that an increase in the applied time constraint increases the chances to get an optimum selected and prioritized test suite. The computations yielded that the complexity of our algorithm depends on the size of the test suite and the applied time constraint and is independent of the number of faults being mutated or any other input variable.

2 Related work

ACO approach was initially introduced in (Dorigo et al. 1996) and has been applied successfully to various NP complete optimization problems. Such problems include vehicle routing, scheduling, quadratic assignment, routing in internet-like networks, sequential ordering and more (Di Caro and Dorigo 1998a, b; den Besten et al. 2000; Gambardella and Dorigo 2000; Gambardella et al. 1999; Merkle et al. 2000; Stützle and Dorigo 1999). Recently, a literature review of ACO as applied to software testing was presented in (Suri and Singhal 2012). It showed the major usage of ACO in test case generation techniques and a few techniques that use ACO in the other software testing fields.

The prioritization related issues have been addressed by (Rothermel et al. 2001). They described prioritization for software development environments that were large. Kim proposed the prioritization of test cases with respect to the historical execution of existing test data (Kim and Porter 2002). An empirical study was also performed based on various greedy algorithms by (Li et al. 2007). Time-based regression test prioritization using genetic algorithms has been proposed in (Walcott et al. 2006), within which a time period is fixed for the entire regression testing process to execute.

3 Time constraint based test case selection and priortization using ACO

This technique requires an original regression test suite, along with the fault matrix and execution time of every individual test case. ACO has been mapped to generate artificial ants corresponding to the test cases existing in the original test suite. These ants then find their paths with a goal of complete fault coverage in minimum possible aggregate execution time of the test cases covered in their paths. Out of all the ants, the path with minimum execution time gets some artificial pheromone to be deposited on it. The ants are resent to explore new paths, but now there choices are affected by the closeness and amount of pheromone deposited on the path. This process gets repeated till a pre-defined time-constraint is met. This time-constraint can be different for different runs of the algorithm. The authors have already proposed such a technique in a magazine article in 2010 (Singh et al. 2010). The same technique has been formally reproposed in this section.

3.1 Pre-requisites of the algorithm

- S[N] is a list of N test cases S_1, S_2…, S_n, representing the given test suite.
- F[K] is a list of faults F_1, F_2,…, F_k injected in the code to be tested.
- SF[N][K] is the fault matrix containing information of the faults covered by each test case individually.
- P[N][N] is the path matix of size N × N, where P[i][1]….P[i][N] store the path of the ‘ith’ ant, i.e. the list of test cases visited by the at on its complete path.
- N test cases correspond to N artificial ants or N nodes in the graph.
- W[N][N] is another N × N matrix containing the weights on each of the edges in the graph. This corresponds to the amount of pheromone deposited on the path.
- AF[K] is a list storing the faults killed (detected) by each ant during their traversal. Whenever an ant visits a test case, the faults detected by the test case are considered killed by that ant.
- ET[N] keeps a record of the execution time of each ant during its path.
- TC is the applied time constraint acting as he stopping criteria for the following algorithm.
- i, j, t, it_time, and nxtnode are some temporary variables used in the algorithm.

3.2 Time-constraint based algorithm using ACO

Step 1. Initialization Process:

```
W[i][j] ← 0 for all 1≤ i, j ≤ N
it_time ← 0
TC ← value from user
ET[i] ← 0 for all 1 ≤ i ≤ N
```

Step 2. For all N ants, do

```
    P[i][1] ← i
    j ← i
    ET[i] ← ET[i] + execution time of the i^th
          test case.
    nxtnode ← i
    F[i] ← F[i] + Faults covered by the i^th test
          case.
    LOOP, repeat the following until ( F[i] !=
          All faults )  OR  ( j ≤ N )
          t ← call sel_testcase( i, nxtnode)
          j ← j + 1
          P[i][j] ← t
          ET[i] ← ET[i] + Execution time of 't'
          F[i] ← F[i] + Faults covered by 't'
    End LOOP
End For

Min_t ← min { ET[i] } for all 1 ≤ i ≤ N
Max_t ← max { ET[i] } for all 1 ≤ i ≤ N
It_time ← it_time + max_t
ET[i] ← 0 for ll 1 ≤ i ≤ N

// Pheromone updation //
W[i][j] ← W[i][j] + 1 for all the edges [i,j] on
the best path with ET[i]=min_t

// Pheromone Evaporation //
W[i][j] ← W[i][j] + 0.1 x W[i][j] for all i,j ≤ N

P[i][j] ← 0 for 1 ≤ i ≤ N
```

Repeat Step 2 till (it_time ≤ TC)

Step 3.

```
  Sel_testcase( a, nxtnode )
  {
   If( max_weight edge out of all the edges
from nxtnode to a vertex 'k' ( W[nxtnode][k] )
not already covered in P[i][1 .... N] is only one)

Then
    Return 'k'

Else
 Select a random edge [i,h], from nxtnode to
node 'h' not already covered in P[i][1 ... N]
having max (W[nxtnode][h])

 Return 'h'
}
```

Table 1 Details of the selected eight test programs

P. no.	Program name	Language	Size LOC	No. of faults	Test suite size	Test suite execution time (sec)
P1	CollAdmission	C++	281	5	9	105.32
P2	HotelMgmnt	C++	666	5	5	49.84
P3	triangle	C++	37	6	19	382
P4	quadratic	C++	38	8	19	441
P5	cost_of_pub	C++	31	8	19	382
P6	calculator	C++	101	9	25	82.5
P7	prev_day	C++	87	7	19	468
P8	railway_book	Java	129	10	26	177

3.3 Complexity of the algorithms

The complete effort of building a new algorithm is justified only if the algorithm itself takes much less time to execute than rerunning the entire regression test suite. In an effort to justify the efficiency of ACO applied to test case selection and prioritization under the restricted execution time, the complexity of the algorithm has been computed.

The algorithm proposed in the previous section has Step 1 as initialization. This step has an execution time bounded by the total number of test cases or ants, i.e., 'N'. Assuming 'T' as the original regression test suite having… $|T| = N$, as number of total test cases in it. 'TC' is the time constraint put by the user on the running time of the algorithm.

Generation of 'N' nodes or artificial ants in the Step 1 is an $O(|T|) = O(N)$ operation (Cormen et al. 2009)

The second Step has 2 nested loops running at most the number of total ants, i.e. N. Inside the 2 loops, all the statements are executed are non-repetitive and take some constant time to execute. The innermost loop calls a function sel_testcase (i, nxtnode) till all the faults are covered. This can loop for maximum N times (the total number of test cases in the test suite). Therefore, in the best case a single call will be made to the sel_testcase(), while on the other hand, the worst case requires N calls to be made for sel_testcase(). The statements inside the function sel_testcase() take constant execution time independent of the size of the input variables like test suite and the fault matrix.

The outer (for) loop repeats itself for N iterations, which is clear from the definition of the 'for' loop. Thus, these loops combined are upper-bounded by $O(N \times N) = O(N^2)$. However, Step 2 itself is repeated till the user-entered time-constraint (TC) has been reached. Thus, the final running time of our algorithm computes to be $O(TC.N^2)$. This is very motivational in comparison with other NP-complete problems (Kim and Porter 2002).

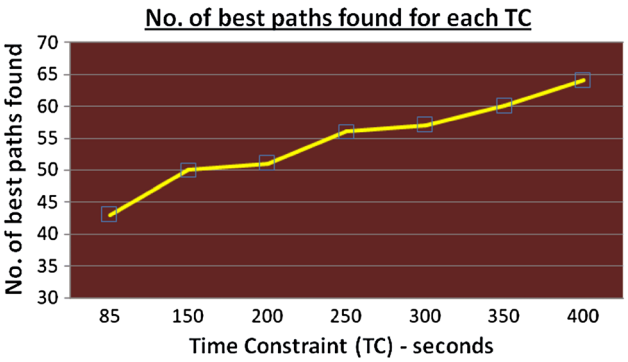

Fig. 1 Number of best paths found out of 10 versus TC

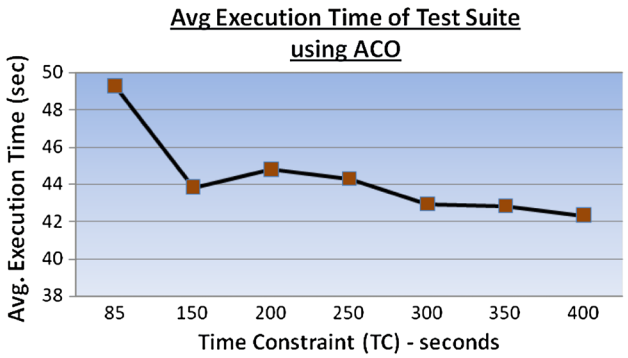

Fig. 2 Average execution time of ACO paths versus TC

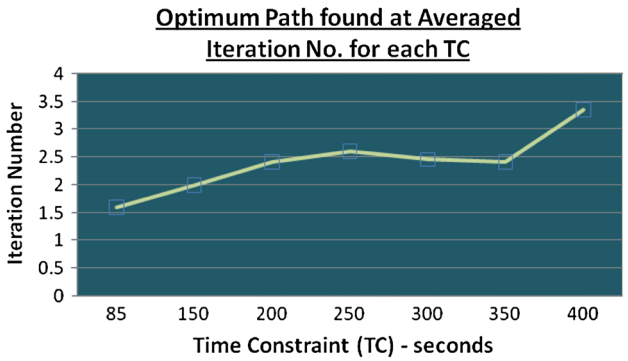

Fig. 3 Average no. of iteration at which optimum path is found versus TC

It can be concluded from the above discussion that the proposed algorithm has a running time which is independent of the number of faults seeded in the test program. The running time of the algorithm is directly proportional to the applied time-constraint and exponentially proportional to the size of the regression test suite.

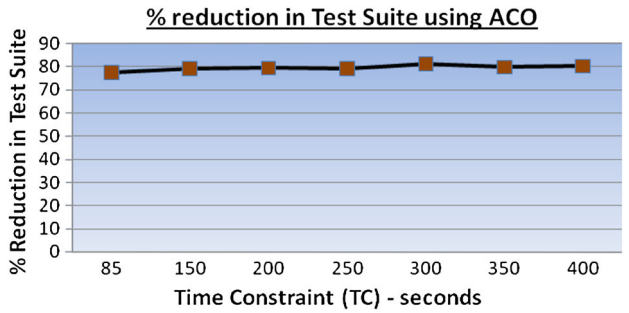

Fig. 4 Percentage reduction in test suite using ACO for the test programs

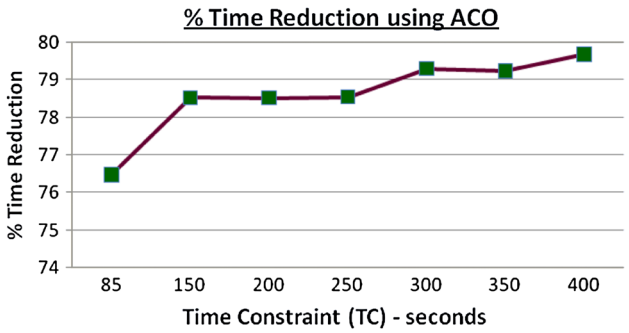

Fig. 5 Percentage reduction in execution time for ACO selected test cases versus TC

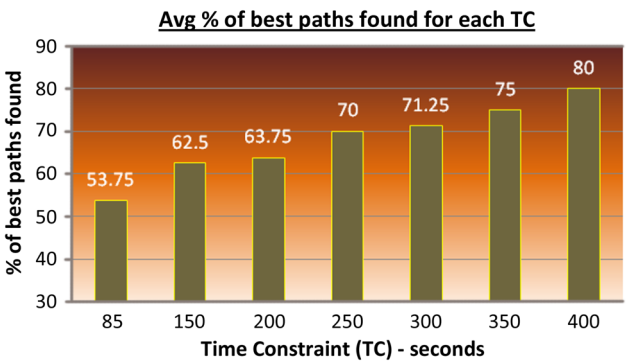

Fig. 6 Average percentage correctness achieved using ACO versus TC

4 Experimental setup

4.1 Programs

Eight programs were selected as the experimental data for our analysis. We used seven C++ and one java program. Using mutation testing technique, five to ten faults were seeded and black box test cases for the programs were generated. Brief description about the programs, their sizes, versions and test suites are given in Table 1.

4.2 Variables

The independent variables (input from user) manipulated by the experiment are:

(1) Subject programs with five to ten faults each.
(2) Various Time Constraints (TC).

4.3 For each of the programs, on each run we measured

(1) The ratio of the total execution time of the test suite to the reduced ACO execution time.
(2) Whether the optimal path has been found or not.

Due to the random nature of ACO, each execution of a program led to different paths using which three dependent variables were computed:

(1) Percentage reduction in test suite size,
(2) Percentage reduction in execution time of the test suite, and
(3) Percentage of the number of times the best path is found.

4.4 Design

To build the confidence in the ACO technique, ten runs for each of the programs for the same test suites and seven different TC values were executed. Each run yielded the path found whether or not optimal. The time constraint (TC) acting as the stopping criteria of the algorithm was varied between 85 and 400 for 70 runs of each program. Reduction in execution time, optimality of the technique and the correctness of the technique while matching with optimum path were obtained from the output data.

5 Data analysis

For our experiment, out of the complete test pool, we randomly chose a test suite of five to twenty-six test cases for P1–P8 respectively. Execution time for each of the test cases was then calculated and used as an input to the ACO algorithm. The algorithm was executed 70 times each for the test programs with varying time constraint TC = 85, 150, 200, 250, 300, 350 and 400. The results obtained are summarized in Fig. 1, 2, 3, 4, 5, 6.

5.1 Time analysis

Previously, ACO_TCSP (Suri and Singhal 2011) was tested only for a constant value of the TC on various programs. This paper, analyses the effect of seven different values of TC on all eight. Ten runs for each case were recorded, making a total of 560 ACO runs (10 (runs) * 7 (values of TC) * 8 (test programs)) on the test data. For 381 out of 560 runs, the optimum path was found using ACO_TCSP. This indicates a very high probability of getting optimal results using the ACO_TCSP tool.

The average numbers of best paths found by each test program for varying values of TC over ten runs each were recorded. The same is represented graphically in Fig. 1. It can be observed clearly that the number of best paths found increases with the increase in value of TC. Though different numbers of best paths are found by each of the eight programs, but a general increasing trend with rising TC value can be inferred. It also shows the effectiveness of ACO_TCSP in finding the optimum paths according to total fault coverage and minimum execution time criteria.

Average execution times of the final paths found by ACO_TCSP for all the eight test programs with seven different TC values averaged over ten runs each are depicted graphically in Fig. 2. It can be inferred from the figure that even though execution time of final paths for eight programs are different, but the general trend is a decrease in the execution time with an increase in TC value. Thus, as the TC is increased, paths with less amount of execution time are found by the ACO_TCSP. It can be thus derived from here that increased value of TC provides better results. The exceptions for TC value 150 can be explained usin the random nature of ACO. The execution of ACO_TCSP gave very good results in our experiment for the TC value = 150 s.

The average number of iterations of ACO_TCSP execution out of ten runs, at which the optimum path is found, were also noted. Its variation with different TC values is depicted in the graph and shown in Fig. 3. A general increase in the number of iteration finding optimum path with the increasing TC can be noted from the figure. Thus increasing the value of TC increases the chances of finding the optimum path by the proposed technique.

5.2 Cost benefit analysis

The results obtained in the previous sections were further summarized to find the cost benefit analysis of the technique. The cost of running the ACO_TCSP in addition to the cost of running the selected and prioritized test suite was compared with the cost of executing the complete regression test suite.

The percentage reduction in the size of selected test suite using ACO_TCSP averaged over the 70 runs for all eight programs was accrued and is shown pictorially in Fig. 4. The value for percentage selection of test suite has been calculated using the formula:

$$\% \text{ of test cases selected} = \frac{(\text{No. of selected tests} * 100)/\text{total}}{\text{No. of test cases}}$$

If 'n' is the size of a test suite and 's' be the size of selected test suite using ACO, then the percentage reduction in test suite size is computed by the formula:

$$\% \text{ reduction in test suite} = (n - s)/n * 100\,\%$$

ACO provides almost 80 % reductions in the size of the test suite for all the test programs without any exception. This is very high (near optimum) reduction achieved by ACO_TCSP algorithm in the test suite itself.

Similarly, we the percentage reduction in execution time of the selected test suite using ACO was computed in comparison with the complete regression test suite. Let 'T' be the total execution time of the original test suite, 'Ta' be the running time of ACO_TCSP, and 'Ts' be the execution time of selected test suite. The, percentage reduction in execution time using ACO is formulated as:

$$\% \text{ execution time reduction} = (T - (Ta + Ts))/T * 100\,\%$$

The same formula is used for the computation of the average execution time reductions over 70 runs for each test program. As has already been analysed in the previous section, the technique provides near optimum reduction in execution time of the final selected and prioritized test suite. The same has been plotted versus TC in Fig. 5. The chosen TC values range from 85 to 400 s. Again, we can clearly observe that higher percentage time reduction is achieved for higher TC and that this technique proves to be very effective in terms of time reduction as well.

5.3 Correctness of the technique

Another important aspect for analysis is the correctness of the technique, i.e. how many times the technique is able to give the optimum result (the best possible ordering). We computed the percentage of the average number of times over 70 runs for all eight programs, when the result was an optimum ordering. The values for the percentage correctness of ACO_TCSP for various TC values are presented graphically in the Fig. 6.

On examining Fig. 6, it can be observed that there exists a continuous increase in the percentage of best paths found corresponding to the increase in the value of time constraint (TC). This suggests that there might exist a TC value after which ACO_TCSP would always lead to the optimum ordering.

6 Conclusion

In this paper, we validated the technique proposed in (Singh et al. 2010) and implemented in (Suri and Singhal 2011). The results achieved are encouraging for the fact that

(i) the test suite selection and prioritization technique reduces the size of test suite,
(ii) the correctness achieved is very high (53–80 %), and
(iii) the ordered test suite potentially enables us to discover the faults earlier.

The time analysis for the varying values of TC led to four major observations:

(1) More numbers of optimum (or best) paths are found at higher values of TC.
(2) Low execution time for the selected path is achieved for higher TC values.
(3) Larger time reduction for the ACO selected test suite is achieved at higher values of TC.
(4) There are more chances of ACO_TCSP resulting in optimum paths at higher values of TC.

All these observations imply that the ACO technique in prioritization and selection gives better results at higher TC values with minimal effect of extra time taken by the algorithm. Our studies validate that the selection and prioritization technique based on fault coverage and execution time using ACO in provides near optimum results. The issue of future research may include further testing of the technique to prove that beyond a given TC, the correct ordering can always be obtained.

Software practitioners might use the ACO_TCSP tool developed to reduce the time and effort required for prioritization of test cases. This technique leads to greater time & cost savings when applied to larger and complex test suites, as compared to the smaller ones. Using the ACO approach, software practitioners can effectively select & prioritize test cases from a test suite, with the minimum execution time. Hence, the proposed algorithm may prove to be useful in the real-life situations.

References

Ayari K, Bouktif S, Antoniol G (2007) Automatic mutation test input data generation via ant colony. p 1074

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, PHI Publications

den Besten ML, Stützle T, Dorigo M (2000) Ant colony optimization for the total weighted tardiness problem. In: Schoenauer M, Deb K, Rudolph G, Yao X, Lutton E, Merelo JJ, Schwefel H-P (eds) Proceedings of PPSN-VI, sixth international conference on parallel problem solving from nature, vol 1917., Lecture notes in computer scienceSpringer, Berlin, pp 611–620

Di Caro G, Dorigo M (1998a) AntNet: distributed stigmergetic control for communications networks. J Artif Intell Res 9:317–365

Di Caro G, Dorigo M (1998b) Antnet: distributed stigmergetic control for communications networks. J Artif Intell Res 9:317–367

Dorigo M, Maniezzo V, Colorni A (1996) The ant system: optimization by a colony of cooperating agents. IEEE Trans Syst Man Cybern B Cybern 26(1):29–41

Elbaum S, Rothermel G, Kanduri S, Malishevsky AG (2004) Selecting a cost-effective test case prioritization technique. Softw Qual J 12(3):185–210

Rothermel G, Untch RH, Chu C, Harrold MJ (1999) Test case prioritization: an empirical study. In: Proceedings of the international conference on software maintenance. p 179–188

Gambardella LM, Dorigo M (2000) Ant colony system hybridized with a new local search for the sequential ordering problem. INFORMS J Comput 12(3):237–255

Gambardella LM, Taillard ÈD, Agazzi G (1999) MACS-VRPTW: a multiple ant colony system for vehicle routing problems with time windows. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimization. McGraw Hill, London, pp 63–76

Gomez O, Baren B (2005) Omicron ACO. A new ant colony optimization algorithm. clei electronic journal 8(1):paper 5

Graves TL, Harrold MJ, Kim MJ, Porter A, Rothermel G (2001) An empirical study of regression test selection techniques. ACM Trans Softw Eng Meth 10(2):149–183

Walcott KR, Soffa ML, Kapfhammer GM, Roos RS (2006) Time aware test suite prioritization. In: Proceedings of ISSTA. p 1–11

Kim JM, Porter A (2002) A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th international conference on software engineering. p 119–129

Krishnamoorthi R, Sahaaya SA, Mary A (2009) Regression test suite prioritization using genetic algorithms. Int J Hybrid Inf Technol 2(3):35

Li H, Peng Lam C (2005) Software test data generation using ant colony optimization. p 1

Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33:4

Li L, Ju S, Zhang Y (2008) Improved ant colony optimization for the travelling salesman problem. International conference on intelligent computation technology and automation. p 76

Merkle M, Middendorf, Schmeck H (2000) Ant colony optimization for resource-constrained project scheduling. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2000), Morgan Kaufmann Publishers, San Francisco. p 893–900

Parpinelli RS, Lopes HS, Freitas AA (2002) Data mining with an ant colony optimization algorithm. IEEE Trans Evol Comput 6:321–332

Rothermel G, Harrold MJ, Dedhia J (2000) Regression test selection for C++ programs. Softw Test Verification Reliab 10(2):77–109

Rothermel G, Untch RH, Chu C, Harold MJ (2001) Test case prioritization. IEEE Trans Softw Eng 27(10):928–948

Singh Y, Kaur A, Suri B (2006) A new technique for version—specific test case selection and prioritization for regression testing. J Comput Soc India 36(4):23–32

Singh Y, Kaur A, Suri B (2010) Test case prioritization using ant colony optimization. ACM SIGSOFT Softw Eng Notes 35(4):1–7

Stützle T, Dorigo M (1999) ACO algorithms for the quadratic assignment problem. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimization. McGraw Hill, London, pp 33–50

Suri B, Singhal S (2011) Analyzing test case selection & prioritization using ACO. ACM SIGSOFT Softw Eng Notes 36(6):1–5. doi:10.1145/2047414.2047431

Suri B, Singhal S (2012) Literature survey of ant colony optimization in software testing, (CONSEG). In: The Proceedings of the CSI sixth international conference on software engineering, Indore. doi: 10.1109/CONSEG.2012.6349501

Zhao P, Zhao P, Zhang X (2006) New ant colony optimization for the knapsack problem