International Journal of Research and Development in Applied Science and Engineering (IJRDASE)
ISSN: 2454-6844

A Review on Software Testing Suite Optimization by PSO

Azeem Ahmed
Department of Computer Science Engg.
L.I.T., Lucknow (U.P.), India
mail2ahmadazeem@gmail.com

Vipin Jaiswal
Department of Computer Science Engg.
L.I.T., Lucknow (U.P.), India
dir@litlucknow.ac.in

Abstract: Software testing is an expensive, but important, process. Unfortunately, there may be insufficient resources to allow for the re–execution of all test cases during regression testing. In this situation, test case prioritization techniques aim to improve the effectiveness of testing, by ordering the test cases so that the most beneficial are executed first. Previous work on test case prioritisation has focused on Greedy Algorithms. However, it is known that these algorithms may produce sub–optimal results, because they may construct results that denote only local minima within the search space. By contrast, meta–heuristic and evolutionary search algorithms aim to avoid such problems.

Keywords: Greedy Algorithm, Test case prioritization, PSO, Software testing

1. Introduction:

Software testing is a frequently applied but expensive maintenance process that aims to (re-)verify modified software. Many approaches for improving the testing processes have been investigated. Test case prioritisation [18] is one of these approaches, which orders test cases so that the test cases with highest priority, according to some criterion (a 'fitness metric'), are executed first. Rothermel et al. [18] define the test case prioritisation problem and describe several issues relevant to its solution. In the Microsoft Developer Network (MSDN) library, the achievement of adequate coverage without wasting time is a primary consideration when conducting regression tests [13]. Furthermore, several testing standards require branch adequate coverage, making the speedy achievement of coverage an important aspect of the regression testing process. In previous work, many techniques for software test case prioritization have been described. Most of the proposed techniques were code–based, relying on information relating test cases to coverage of code elements. In [6, 17, 18], Rothermel et al. investigated several prioritizing techniques such as total statement (or branch) coverage prioritization and additional statement (or branch) coverage prioritization, that can improve the rate of fault detection. In [22], Wong et al. prioritized test cases according to the criterion of 'increasing cost per additional coverage'. In [20], Srivastava and Thiagarajan studied a prioritisation technique that was based on the changes that have been made to the program and focused on the objective function of "impacted block coverage". Non–coverage based techniques in the literature include fault–exposing–potential (FEP) prioritization [18], history–based test prioritization [11], and the incorporation of varying test costs and fault severities into test case prioritization [5, 6]. Greedy Algorithms have been widely employed for test case prioritization. Greedy Algorithms incrementally add test cases to an initially empty sequence. The choice of which test case to add next is simple: it is that which achieves the maximum value for the desired metric (e.g., some measure of coverage). However, as Rothermel et al. [18] point out, this greedy prioritization algorithm may not always choose the optimal test case ordering. Greedy Algorithms also require that it is possible to define the improvement in fitness obtained by the addition of a single element to a partially constructed sequence. As this paper will show, in the case of regression test case prioritization, this is not always possible. A simple example, based on statement coverage, is presented in Table 1.1. If the aim is to achieve full statement coverage as soon as possible, a Greedy Algorithm may select A,B,C. However, the optimal test case orderings for this example are B,C,A and C,B,A

2. Related Work:

This work aims to find optimization techniques for test suite optimization (TSO) in Regression testing(R/T). **[1]. Zheng Li**, have studied various optimization techniques and algorithms in order to find the technique that should be used for test suite optimization (TSO) in regression testing. The techniques like hill climbing, greedy algorithm, additional greedy algorithm,2-optimal greedy algorithm, genetic algorithm, particle swarm optimization(PSO) are studied and compared using different parameters. These techniques are compared on the basis of their code coverage area, computational effort and execution time.

This work delivers review of techniques used for test suit optimization. It also gives detail about the comparison of different optimization techniques. The review can help the researchers in comparing the optimization techniques. It provides future work idea that can be used in improving regression testing. In my opinion we can use particle swarm optimization technique for test suite optimization purpose by improving the code coverage area of test suite. This survey also shows that there is scope of future work in the field of regression testing. We can use this review for obtaining a better technique for test suit optimization in regression testing.

Where the creation, understanding, and assessment of software testing and regression testing techniques are concerned, controlled experimentation is an indispensable research methodology. Obtaining the infrastructure necessary to support such experimentation, however, is difficult and expensive. As a result, progress in experimentation with testing techniques has been slow, and empirical data on the costs and effectiveness of techniques remains relatively scarce. To help address this problem, **[2] V. Basili,** have been designing and constructing infrastructure to support controlled experimentation with testing and regression testing techniques. This work reports on the challenges faced by researchers experimenting with testing techniques, including those that inform the design of our infrastructure. The work then describes the infrastructure that we are creating in response to these challenges, and that we are now making available to other researchers, and discusses the impact that this infrastructure has and can be expected to have.
V. Basili have presented our infrastructure for supporting controlled experimentation with testing techniques, and we have described several of the ways in which it can potentially help address many of the challenges faced by researchers wishing to conduct controlled experiments on testing. We close this article by providing additional discussion of the impact, both demonstrated and potential, of this infrastructure. First, we remark on the impact of our infrastructure to date. Many of the infrastructure objects described in the previous section are only now being made available to other researchers. The Siemens and space programs, however, in the format extended and organized by ourselves, have been available to other researchers since 1999, and have seen widespread use. In addition to our own works describing experimentation using these artifacts (over twenty such works have appeared, we have identified seven other works not involving creators of this initial infrastructure that describe controlled experiments 9 involving testing techniques using the Siemens and/or space programs. The artifacts have also been used in for a study of dynamic invariant detection (attesting to the feasibility of using the infrastructure in areas beyond those limited to testing). In our review of the literature, we have found no similar usage of other artifacts *for controlled experimentation in software testing*; the willingness of other researchers to use the Siemens and space artifacts thus attests to the potential for infrastructure, once made available, to have an impact on research. This same willingness, however, also illustrates the need for improvementsto infrastructure, given that the Siemens and space artifacts present only a small sample of the population of programs, versions, tests, and faults. It seems reasonable, then, to expect our extended infrastructure to be used for experimentation by others, and to help extend the validity of experimental results through widened scope. Indeed, we ourselves have been able to use several of the newer infrastructure objects that are about to be released in controlled experiments described in recent publications as well as in three publications currently under review.
In terms of impact, it is also worthwhile to discuss the costs involved in preparing infrastructure; it is these costs that we *save* when we re-use infrastructure. For example, the emp-server and bash objects required between 80 and 300 person-hours per version to prepare; two faculty and five graduate research assistants have been involved in this preparation. The flex, grep, make, sed and gzip programs involved two faculty, three graduate students, and five undergraduate students; these students worked 10-20 hours per week on these programs for between 20 and 30 weeks. These costs are not costs typically affordable by researchers; it is only by amortizing the costs over the potential controlled experiments that can follow that we render the costs acceptable. Finally, there are several additional potential benefits to be realized through sharing of infrastructure in terms of challenges addressed; these translate into a reduction of threats to validity that would exist were the infrastructure not shared. By sharing our infrastructure with others, we can expect to receive feedback that will improve it. User feedback will allow us to improve the robustness of our tools and the clarity and completeness of our documentation, enhancing the opportunities for replication of experiments, aggregation of findings, and manipulation of individual factors. We are in the process of setting up mechanisms for encouraging researchers who use our infrastructure to contribute additions to it in the form of new fault data, new test suites, and variants of programs and versions that function on other operational platforms. Ultimately, we expect the community of researchers to assemble additional artifacts using the formats and tools prescribed, and contribute them to the infrastructure, which will increase the range and representativeness of artifacts available to support experimentation. Through this effort we hope to aid the entire testing research community in pursuing controlled experimentation with testing techniques, increasing our understanding of these techniques and the factors that affect them in ways that can be achieved only through such experimentation.

This survey has presented by **[3]. Agrawal G**. on empirical work involving program slicing of relatively large scale programs (ranging from a few thousand lines of code to tens and even hundreds of thousands of lines of code). The results concern questions about the efficacy of slicing, its applications, the nature of the slices themselves, and the interplay between slicing and other source code analyses. Results from over thirty works, that contain empirical results are included. Some general trends in the results reported are evident and also some directions for future work emerge:
(1) *Slice size.* In two separate studies with two different static slicing tools and two different ways of determining slicing criteria, it was reported that a typical static slice was

approximately one third of the size of the program from which it was constructed. There has only been a single study of dynamic slice size and the results indicate that the typical slice size is approximately one fifth of the size of the program from which it is constructed. There also appears to be a small (perhaps not significant) difference in forward and backward slice size. Finally, there is disagreement in the empirical literature as to whether calling context makes a difference in slice size with some studies finding little impactand others significant impact. More work is needed in this area to clear up this important issue.
(2) *The impact of supporting analyses.* It has been shown that a variety of supporting analyses and algorithms used for tuning slice construction have an impact upon slice size. However, one of the striking aspects of the empirical results on supporting analyses is the way in which a dramatic increase in precision of some supporting analyses do not lead to a commensurate decrease in slice size. Pointer analysis is a prime example. This phenomenon is worthy of further investigation.
(3) *Slicing tools and techniques.* This chapter has reported several areas of work which show how the standard approaches to slicing have been improved upon. The primary observation which emerges is that this continues to be a worthwhile area of research and where there are, as yet undiscovered, techniques for improving slicing. Therefore, the results for slice sizes should be regarded as upper bounds, rather than limits. Finally, more work is required to better understand the pros and cons of the essentially demand driven, data flow slicing techniques as compared to the caching graph reachability based slicing techniques.
(4) *Applications of slices.* The growing body of empirical evidence for applications
of slices and in particular what might be termed the 'non-traditional' applications is encouraging. Though slicing was original deemed to be an end in itself, more recent work has used slicing as a part of an overall approach or as a way of solving problems for which it was not originally intended (such as clone detection). These results are encouraging, because they indicate that slicing has many applications beyond those originally envisaged.
(5) *Human comprehension studies.* The application of program slicing to aid human comprehension has shown that slicing does, indeed, assist programmers when performing comprehension-intensive activities such as debugging, fault localization, and impact analysis. Furthermore, there is evidence to suggest that slicing-aware programmers work differently than non-slicing aware programmers.
In many ways the surface has only been scratched: the studies concern primarily static slicing. There are some indications that more recent forms of slicing, such as amorphous and conditioned slicing may offer even greater benefits in programmer comprehension activities.
In conclusion, there is now a large body of evidence to suggest that slicing is practical and effective. New application areas and improvements to slice computation techniques are being regularly introduced, and results regarding the existing techniques and applications are encouraging.

[4]. T. Abdel-Hamid presented a search{based approach for planning resource allocation in large software projects, which aims to find an optimal or near optimal order in which to allocate work packages to programming teams, in order to minimize the project duration. The approach is validated by an empirical study of a large, commercial Y2K massive maintenance project, comparing random scheduling, hill climbing, simulating annealing and genetic algorithms, applied to two different problem encodings. Results show that a genome encoding the work package ordering, and alertness function obtained by queuing simulation constitute the best choice, both in terms of quality of results and number of fitness evaluations required to achieve them.
This work has demonstrated that search{based techniques can be applied to optimise resource allocation in a software engineering project. Three search based techniques were evaluated. Each was applied to two very different encoding strategies. Each encoding represents the way in which the work packages of the overall project are to be allocated to teams of programmers. The ordering encoding, which combines the search{based approach with a queuing simulation model, was found to outperform the other approaches. For the less optimal encoding the GA performed significantly better than the other approaches. For the optimal encoding, though GA starts better simulated annealing and hill climbing approaches soon catch up, so that the overall difference between the three approaches appears to be small, compared to the problem of establishing an effective encoding. Finally, the work reports the results of experiments that alter the size of the project teams. While for a small overall stang level, double-sized teams do not improve performance, increasing the overall sting level is sufficiently high, it proved effective to have double.

Regression testing assures changed programs against unintended amendments. Rearranging the execution order of test cases is a key idea to improve their effectiveness. Paradoxically, many test case prioritization techniques resolve tie cases using the random selection approach, and yet random ordering of test cases has been considered as ineffective. Existing unit testing research unveils that adaptive random testing (ART) is a promising candidate that may replace random testing (RT). In this work, **[5]. K. P. Chan**, not only propose a new family of coverage-based ART techniques, but also show empirically that they are statistically superior to the RT-based technique in detecting faults. Furthermore, one of the ART prioritization techniques is consistently comparable to some of the best coverage-based prioritization techniques (namely, the “additional” techniques) and yet involves much less time cost. Test case prioritization is a means to achieve target objectives in software testing by reordering the

execution sequences of test suites. Many existing test case prioritization techniques use random selection to resolve tie cases. Paradoxically, the random approach for test case prioritization has a long tradition to be deemed as ineffective. Adaptive random testing (ART), which aims to spread test cases as evenly and early as possible over the input domain, has been proposed for test case generation. Empirical results have shown that ART can be 40 to 50% more effective than random testing in revealing the first failure of a program, which is close to the theoretical limit. In regression testing, however, further refinements of ART may be feasible in the presence of coverage information.
This work proposed the first family of adaptive random test case prioritization techniques, and conducts an experiment to evaluate its performance. It explores the ART prioritization techniques with different test set distance definitions at different code coverage levels rather than spreading test cases as evenly and early as possible over the input domain. The empirical results show that our techniques are significantly more effective than random ordering. Moreover, the ART-br-maxmin prioritization technique is a good candidate for practical use because it can be as efficient and statistically as effective as traditional coverage-based prioritization techniques in revealing failures. In the future, we will investigate other test case measures and study beyond code coverage. Furthermore, they also want to extend our ART prioritization techniques to testing concurrent programs. Finally, as stated in the introduction, the use of random ordering to resolve tie-cases with existing greedy algorithms is deemed as ineffective. We would like to apply ART to resolve tie-cases in order to combine the merits of our techniques with other approaches.

Test case prioritization is an approach aiming at increasing the rate of faults detection during the testing phase, by reordering test case execution. Many techniques for regression test case prioritization have been proposed. In this work, **[6] . G. Antoniol** performed a simulation experiment to study five search algorithms for test case prioritization and compare the performance of these algorithms. The target of the study is to have an in-depth investigation and improve the generality of the comparison results. The simulation study provides two useful guidelines:
(1)Two search algorithms, Additional Greedy Algorithm and 2- Optimal Greedy Algorithm, outperform the other three search algorithms in most cases.
(2) The performance of the five search algorithms will be affected by the overlap of test cases with regard to test requirements This work discusses the performance of five typical techniques, Greedy Algorithm, Additional Greedy Algorithm, 2-Optimal Greedy Algorithm, Hill Climbing Algorithm and Genetic Algorithm, in various cases for test case prioritization. It also reports empirical results of a comparison of these five techniques. Since we assume that each test case takes the same time to be accomplished, the only criterion to compare the performance of these five techniques is the APRCI. The main findings of our experiment can be summarized as follows:
1. The performance of each algorithm rises with the increase of σ and the decrease of μ.
2. When *roverlap*=1, these five algorithms have nearly the same performance. When *roverlap* is extremely large, the effect of each algorithm is really limited and few APRC scores can be improved.
3. When *roverlap* is moderate, all these algorithms can significantly improve the APRC scores. In most cases, Additional Greedy Algorithm and 2-Optimal Greedy
Algorithm perform better than the other three algorithms and they have the best performance when the value of *roverlap* is around 3. There is no significant difference between the performance of these two algorithms. Based on the findings of our experiment, we have the following recommendation for different ranges of *roverlap*. When *roverlap* is close to 1, any of these five algorithms can be chosen.
When *roverlap* is extremely large (*roverlap*>50), there is no need to apply any of these algorithms since the algorithms themselves are time-consuming. When *roverlap* is moderate, Additional Greedy Algorithm and 2-Optimal Algorithm are the preferable choices.
In this work, the execution time of each test case is assumed the same while in practice, the execution time of each test case differs a lot and there is usually a time limit for test case execution. So applying these techniques to time-aware test case prioritization [13][15] is a topic for our future work. And another work in the future is to do a quantitative statistical analysis on how various factors influence the performance of each search algorithm**.**

[7]. David Wendell Binkley worked argues that regression test optimization problems such as selection and prioritization require multi objective optimization in order to adequately cater for real world regression testing scenarios. The work presents several examples of costs and values that could be incorporated into such a Multi Objective Regression Test Optimization (MORTO) approach Testing is complex. There is no direct measure of fault revelation likelihood and there are many different types of cost involved. This complex interplay between cost and value is further compounded by the many additional validity constraints, making test case selection and prioritization problems that naturally involve multi objective optimization. This work argues that such a Multi Objective Regression Test Optimization (MORTO) approach is long overdue.

In this work **[8] . A. Arcuri and L. Briand** have introduced and empirically evaluated two multi objective test suite prioritisation techniques, based on the multi objective evolutionary algorithms NSGA-II and TAEA. They evaluate them against the state-of-the-art on a set of five utility programs from the Software Infrastructure Repository (SIR), together with a larger program, mysql, from which we extracted fault data for 20 of its faults (all of which have status “closed”). Even when the tester has only a single objective in mind, it is useful to use a multi objective formulation to improve early fault revelation. Since fault-revealing tests are unidentifiable at prioritisation time, the tester is forced to use a surrogate. They introduce a three-objective formulation of the problem, in which all three objectives are coverage-related surrogates. Our results demonstrate that this approach is highly effective; we find faults significantly faster than the state-of-the-art and with large effect size in 19 out of the 22 cases studied. We also introduce coverage compaction algorithm which dramatically reduces coverage data size, and thereby algorithm execution time. On the larger program, mysql, the additional greedy algorithm takes 1.4 hours to prioritise without compaction, but only 7 seconds after compaction. The performance improvement is even more dramatic for the multi objective algorithms. Their performance is improved from over eight days to a little over one minute. Since compaction can be applied to any and all regression testing approaches, we believe that these performance improvements may make an important contribution to the practical application of repression test optimisation in future work.

The aim of test case prioritisation is to determine an ordering of test cases that maximises the likelihood of early fault revelation. Previous prioritisation techniques have tended to be single objective, for which the additional greedy algorithm is the current state-of-the-art. Unlike test suite minimisation, multi objective test case prioritisation has not been thoroughly evaluated. This work presents an extensive empirical study of the effectiveness of multi objective test case prioritisation, evaluating it on multiple versions of five widely-used benchmark programs and a much larger real world system of over 1 million lines of code. The work also presents a lossless coverage compaction algorithm that dramatically scales the performance of all algorithms studied by between 2 and 4 orders of magnitude, making prioritization practical for even very demanding problems.

[9] A. Arcuri and L. Briand worked empirically evaluates seven different test case prioritisation algorithms: three instantiations of the additional greedy algorithm with different fault detection surrogates, two multi objective formulations usingMOEAs (NSGA- II and TAEA), and two hybrid algorithms that augment the MOEAs with the additional greedy seeding. These algorithms are evaluated on a set of five utility programs from the Software Infrastructure Repository (SIR), together with a larger program, mysql, from which 20 real faults with “closed” status have been extracted. The results show that MOEAs and hybrid algorithms can produce solutions whose prioritisation effectiveness, measured by the widely studied APFDc metric, is either equal or superior to those of solutions produced by the additional greedy algorithms. The work also introduces a coverage compaction algorithm that dramatically, yet losslessly, reduces coverage data size, and thereby algorithm execution time. On the largest program, mysql, the additional greedy algorithms can take more than two hours to prioritise without compaction, but only 12 seconds after compaction. The performance improvement is even more dramatic for the MOEAs. Their performance is improved from over eight days to a little over one minute. Since compaction can be applied to any and all regression testing approaches, these performance improvements may make an important contribution to the practical application of regression test optimisation in future work.

Regression testing is an expensive, but important, process. Unfortunately, there may be insufficient resources to allow for the re–execution of all test cases during regression testing. In this situation, test case prioritisation techniques aim to improve the effectiveness of regression testing, by ordering the test cases so that the most beneficial are executed first. Previous work on regression test case prioritisation has focused on Greedy Algorithms. However, it is known that these algorithms may produce sub–optimal results, because they may construct results that denote only local minima within the search space. By contrast, metaheuristic and evolutionary search algorithms aim to avoid such problems.

[10]. G. Antoniol, presented results from an empirical study of the application of several greedy, meta–heuristic and evolutionary search algorithms to six programs, ranging from 374 to 11,148 lines of code for 3 choices of fitness metric. The work addresses the problems of choice of fitness metric, characterisation of landscape and determination of the most suitable search technique to apply. The empirical results replicate previous results concerning Greedy Algorithms. They shed light on the nature of the regression testing search space, indicating that it is multi–modal. The results also show that Genetic Algorithms perform well, although Greedy approaches are surprisingly effective, given the multi–modal nature of the landscape.

This work described five algorithms for the sequencing problem in test case prioritisation for regression testing. It presented the results of an empirical study that investigated their relative effectiveness. The data and analysis indicate that the Greedy Algorithm performs much worse than Additional Greedy, 2–Optimal and Genetic Algorithms overall. Also, the 2–Optimal Algorithm overcomes the weakness of the Greedy Algorithm and Additional Greedy Algorithm (see Table 1) referred to by previous authors. However, the experiments indicate that, in terms of effectiveness, there is no significant difference between the performance of the 2–Optimal and Additional Greedy Algorithms. This suggests that, where applicable the cheaper–to–implement–and–execute Additional Greedy Algorithm should be used. The choice of coverage criterion does not affect the efficiency of algorithms for the test case prioritization problem. The size of the test suite determines the size of the search space, therefore affecting the 30 complexity of test case prioritisation problem. The size of the program does not have a direct effect, but increases the difficulty of computing fitness values. Studies regarding the performance of meta–heuristic algorithms led to several conclusions that have practical ramifications. The results produced by Hill Climbing show that the nature of the fitness landscape is multi–modal. The results produced by the Genetic Algorithm indicate that it is not the best of the five considered in all cases, but that in most cases the differences between the performance and that of the Greedy approach is not significant. However, an analysis of the fitness function shows that there are situations in which it is important to consider the entire ordering and, for such cases, Greedy Algorithms are unlikely to be appropriate (see Figure 4). Given their generality, the fact that Genetic Algorithms perform so well is cause for encouragement. The criteria studied were based on code coverage, which is different from criteria based on fault detection. The application of meta–heuristic algorithms to fault detection based prioritisation problems could possibly yield different results, but this is a topic for future work.

3. Conclusion:

This suggests that, where applicable the cheaper–to–implement–and–execute Additional Greedy Algorithm should be used. The choice of coverage criterion does not affect the efficiency of algorithms for the test case prioritization problem. The size of the program does not have a direct effect, but increases the difficulty of computing fitness values.

References:

[1]. Zheng Li, Mark Harman and Robert M. Hierons, "Search Algorithms for Regression Test Case Prioritization", IEEE Transaction Paper on Software Engineering ,Vol. 33 Issue 4, pp.225-237, 2007
[2]. V. Basili, R. Selby, E. Heinz, and D. Hutchens. Experimentation in software engineering. IEEE Trans. Softw. Eng., 12(7):733–743, July 1986.
[3]. Agrawal G., Guo L., "Evaluating explicitly context-sensitive program slicing", in: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah, 2001, pp. 6-12.
[4]. T. Abdel-Hamid. The dynamics of software project sta_ng: a system dynamics
based simulation approach. IEEE Transactions on Software Engineering, 15(2):109{119, 1989.
[5]. K. P. Chan, T. Y. Chen, and D. P. Towey. Restricted random testing. In Proceedings of the 7th European Conference on Software Quality (ECSQ 2002), volume 2349 of Lecture Notes in Computer Science, pages 321–330. Springer, Berlin, Germany, 2002.
[6]. G. Antoniol, M.D. Penta, and M. Harman, Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05), pp. 240-249, 2005.
[7]. David Wendell Binkley. The application of program slicing to regression testing. Information and Software Technology Special Issue on Program Slicing, 40(11 and 12):583–594, 1998.
[8]. A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.
[9] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.
[10]. G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In 21st IEEE International Conference on Software Maintenance (ICSM 2005), pages 240–249, Budapest, Hungary, 2005. IEEE Computer Society Press, Los Alamitos, California, USA.
[11] Zadeh, L.A., (1965), "Fuzzy sets, Information and Control", 8, pp 338–353.
[12] Eberhart, R. C. and Shi, Y., A Modified Particle Swarm Optimization, Proceedings of IEEE International Conference on Evolutionary Computation, 1998, pp.69-73.
[13] Eberhart, R. C. and Shi, Y., Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization, Proceedings of the 2000 Congress on Evolutionary Computation, Vol. 1, 2000, pp.84-88.
[14] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.
[15] C. R. Reeves, editor. Modern heuristic techniques for combinatorial problems. JohnWiley & Sons, Inc., New York, NY, USA, 1993.
[16] G. Reinelt. TSPLIB— A traveling salesman problem library. ORSA Journal on Computing,
3(4):376–384, 1991.
[17] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In Proceedings ICSM 1999, pages 179–188, Sept. 1999.
[18] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10):929–948, Oct. 2001.
[19] S. S. Skiena. The algorithm design manual. Springer-Verlag, New York, NY, USA, 1998.
[20] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 97–106, New York, NY, USA, 2002. ACM Press.
[21] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352–357, 1984.

[22] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In Proceedings of the Eighth International Symposium on Software Reliability Engineering, pages 230–238. IEEE Computer Society, Nov. 1997.