RAFI: Parallel Dynamic Test-suite Reduction for Software

Najneen Qureshi, Manish Kumar Mukhija, Satish Kumar

Department of Computer Science, ARYA Institute of Engineering and Technology, Jaipur, Rajasthan, India

Corresponding author: Najneen Qureshi, Email: nazneenqureshi938@gmail.com

A pattern in programming testing diminishes the size of a test suite while protecting its general quality. For programming, test cases and a bunch of requirements are given. Each test case is covering a few requirements. In this paper, we will probably discover the technique for test-suite reduction to compute the subset of test cases covering every one of the requirements. While this issue has acquired huge consideration, it is as yet hard to track down the littlest subset of test cases and generally utilized strategies to tackle this issue with inexact arrangements. In this paper, we will likely track down the technique for test-suite reduction to find the subset of test cases that are covering every one of the requirements across renditions. There are as of now existing greedy algorithms and exponential-time algorithms to discover the TRS in a rendition explicit and across adaptations. We proposed another similar greedy heuristic technique RAFI to discover negligible test sets across forms. Our methodology shows that: (i) RAFI is a lot quicker than the exponential-time method and roughly multiple times quicker than the traditional greedy technique. (ii) RAFI strategy generally accomplishes a similar decrease rate contrasted with the traditional greedy method.

Keywords: Test-Suite Reduction, Traditional Greedy, Traditional Greedy parallel, Test-Coverage graph, Dynamic Reduction, Flower, RAFI.

2021. In Rahul Srivastava & Aditya Kumar Singh Pundir (eds.), *New Frontiers in Communication and Intelligent Systems*, 165–176. Computing & Intelligent Systems, SCRS, India. https://doi.org/10.52458/978-81-95502-00-4-20

1 Introduction

We can testify that software products quality is a massive problem for the software industry, and software failure is a significant loss for a software company. Software testing is an essential technique for software quality solace, and it is a challenging but necessary process of the growth of information. If we increase software size, we increase the number of test cases, and automatically cost will be increased. If we design software test cases, then we have to fulfill the demand for software quality. So a cardinal set of the test cases is redundant. We are selecting a minimum cardinal subset for selecting a test case of requirements. Test Suite Reduction (TSR) [3-5] can solve the condition's problem and decrease the activity load, automatically reducing software testing costs. Test Suite Reduction [6-8] comes under regression testing [1-2]. This paper can satisfy how we select a minimum number of test cases and fulfill all the requirements. It is the key to the topic of test case reduction. Next, every Software has some defect, fewer features, and some BUGs, so we are releasing the new version of the software with a BUG fix or additional feature. Before releasing the feature again, we have to test our software for quality and correctness purposes. For this, the normal way is to refine the minimal test set to execute our software. Then it will become more costly to find the reduced test set again. In this paper, we propose a RAFI method to find the reduced test set across the versions by using the information of added features and the previous version's reduced test set.

Now we will understand what the test suite reduction problem is? We will define the test suite reduction problem [10-11] as:

- **a.** If T = {r1, r2, r3, r4, r5 ...rn} are the requirements of the software.
- **b.** If R = {t1, t2, t3, t4, t5...tn} are the test cases of software.
- **c.** If S = {(t, r)} S is the relationship between test cases and the requirements.

Some test cases cover each requirement, and each test case will cover some requirements. In the example (see Fig. 1), * indicates that there is a relationship between the corresponding requirement and the test case [9]. If there is a *, it means that there is a edge corresponding to that requirement and test case. The above notation can be said to be the array representation of the coverage graph. Coverage Graph is the graph that will show the relationship between the test cases and the requirements. We can also represent this relation by bipartite graphs [11-13].

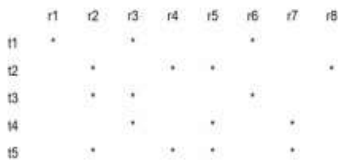

Fig. 1. Test Cases Cover the Requirements

In the above Fig. 1, there are a lot of test sets which are covering all the requirements. For example {t1, t2, t4, t5}, {t1, t2, t4}, etc. But we observe that the second subset is minimal than the first one. So we can say that subset {t1, t2, t4} is the minimal subset covering all the requirements. In this t1 is covering {r1,r3}, t2 is covering {r2, r4, r5, r8} and t3 is covering {r3, r5, r7}. We can say that {t1, t2, t4} covers all the requirements and the reduced test set. There are many approaches for TSR, some are exponential, and some are polynomial-time approaches.

2 Background and Related Work

This section discussed some existing methods of implementing Test Suite Reduction techniques, which we can use to find the reduced test set in a particular version or across the versions. Some of the techniques are the Exponential and polynomial-time algorithms.

2.1 Exponential Time Algorithms

There are several algorithms for Test Suite reduction whose time complexity is exponential, which gives the exact and the Flower method is one method.

2.1.1 FLOWER Method

The flower method [14] comes under the exact approach. We can get the optimal solution by using this method. In this method, we solve a problem as a flow network [15] and use the Ford Fulkerson algorithm [15]. The flower method represents a flow network where every edge has a capacity. Also, given two vertices, source S and sink D, in the graph, find out the maximum possible flow from S to D. Flow on edge does not exceed the given capacity of the edge. Inflow is equal to outflow for every vertex except S to D. Time complexity of this algorithm is exponential, equal to O((2^|test cases|)*edges * |requirements|), which seems an exponential time approach. The Flower method is an integer linear programming (ILP) method [16-17].

2.2 Polynomial Time Algorithms

There are several algorithms for Test Suite reduction whose time complexity is polynomial, which give approximate results. The traditional greedy method is one of the methods.

2.2.1 Traditional greedy method

Traditional greedy [18] is a heuristic approach to find the reduced test set that covers all requirements. We will not get the exact result that we are expecting, but we can get the approximate solution for this. This approach iteratively selects a test case that will cover all the maximum number of remaining requirements. We performed this iteratively until all requirements are not covered. A traditional Greedy method is a heuristic approach, so time complexity is polynomial. Its time complexity depends on the number of requirements, the number of test cases, and the relationship between the test case and requirements. The basic Greedy method's time complexity is O(r * t * minimum(r, t)), where t is the number of test cases, and r is the number of requirements.

2.2.2 HGS (Harrold Gupta Soffa)

In HGS [19], we create a group of requirements covered by one of the test cases. Then we find a group of requirements for each test case. HGS selects the test cases which satisfy the sets. The RTS contains the selected test cases.

2.2.3 GE

In GE [20], we select the test cases covering the requirements whose cardinality is one. The rest of the process is the same as the traditional greedy method for finding the reduced test set.

3 RAFI Sequential Method

In every Software, new versions are released. So before releasing the new version of the software, it must pass all the test cases for quality purposes. Due to the new version of the software, our old coverage got changed. Some edges can be added or deleted. If the edges are added, then we can say that some test cases are now covering more requirements, and if edges got deleted, then we can say that

some tests are not covering requirements that they previously covered. Now we have to find the reduced test set in the new coverage graph. This section implemented the RAFI method to find the reduced test set in the new Coverage graph. In the RAFI method, we are using some information, and that information is 1) a reduced test set of the old coverage graph 2) edges that were modified (added and deleted).

There are two types of edge modification in the old coverage graph.

1. Edge deletion
2. Edge addition

We will see how we are handling the edge modification to find the reduced test set and how we are taking advantage of the old reduced test set.

3.1 Handel edge deletion

Two types of edge deletion are possible

1. Deleted edge belongs to the old reduced test set
2. Deleted edge that does not belong to the old reduced test set

3.1.1 Deleted edge belongs to the old reduced test set

The graph in Fig. 2 shows an example of edge deletion, which belongs to the old reduced test set. When we ran the old coverage graph in a traditional greedy method, we found that the reduced test set was {t1, t2}. After deletion of the edge{t1, r1}. We found that t1 belongs to the old reduced test set, and the old reduced test set now uncovers r1, so we have to cover r1. First, we will check whether the current old reduced test set covers r1 or not. The example below r1 is not covered by any selected test case {t1, t2}, so we will add a test case that will cover the requirement r1 because t3 covers the r1 requirement the t3 is added to the old reduced test case. Now the reduced test case will become {t1, t2, and t3}. Now we try to optimize the reduced test set. We had added the t3 test case because t1 is not covering r1, so we try to cover all requirements without the t1 test case in the new reduced test set. In this case, we can cover all the requirements by {t2, t3}. The new reduced test set will be {t2 and t3}.

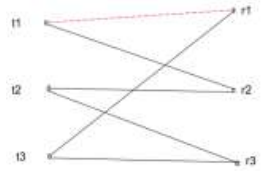

Fig. 2. New coverage graph after the deletion of the edge {t1, r1}

3.1.2 Deleted edge does not belong to the old reduced test set

The graph in Fig. 3 shows an example of edge deletion, which does not belong to the old reduced test set. When we ran the old coverage graph in a traditional greedy method, we found that the reduced test set was {t1, t2}. Now we deleted the edge {t3, r1}. We know that t1 does not belong to the old reduced test set. We found that r1 still got covered by the old reduced test set, so nothing will be affected. The new reduced test set will be {t1 and t2} as the old reduced test set.

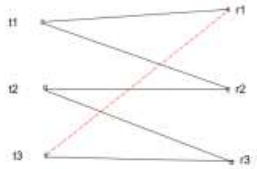

Fig. 3. New coverage graph after the deletion of the edge {t3, r1}

Algorithm

1. *Reduced test set as ReducedSet ;*
2. *Delete the edge as DeletedEdge;*
3. *delete DeletedEdge from CoverageGraph ;*
4. *if DeletedEdge belongs in TestCase(t1), which is in ReducedSet then*
5. *nothing to do for DeletedEdge ;*
6. *else if DeletedEdge belongs in Requirement(r1), which is covered by TestCase(ti) which is in ReducedSet then*
7. *nothing to do for DeletedEdge ;*
8. *else*
9. *Add one TestCases that satisfy that Requirement(r1);*
10. *end*
11. *if all requirements that covered by TestCase(t1)can be covered by ReducedSet - { TestCase(t1) } then*
12. *ReducedSet = ReducedSet - { TestCase(t1) } ;*
13. *end*
14. *end*
15. *return Reduced test set as ReducedSet ;*

Line 1 In the ReducedSet set, we will store the final reduced test set. Line 2 DeletedEdge, we store the test case and the requirement relationship. Line 4-5 If the deleted edge does not belong to the reduced test set, we will do nothing. Line 6-7 If the deleted edge belongs to the reduced test set and r1 is still covered by another test case that belongs to the reduced test set. We will do nothing. Line 9 If not, then we will add the test case which covered r1. Line 11-12 if all requirements which t1 covers get covered by the current reduced test set except t1, we can remove t1 from the reduced test set. Line 15 returns a reduced test set.

3.2 Handel edge addition

Two types of edge addition are possible

1. Added edge belongs to the old reduced test set
2. Added edge does not belong to the old reduced test set

3.2.1 Added edge belongs to the old reduced test set

The graph in Fig. 4 shows an example of edge addition which belongs to the old reduced test set. When we ran the old coverage graph in a traditional greedy method, we found that the reduced test set was {t1, t2}. After adding the edge {t2, r1}, we found that t2 belongs to the old reduced test set, and r1 is now covered by the other test case, which belongs to the old reduced test set. Therefore, we try to optimize the reduced test set. For this, we will check the cardinality of the test case, which was covering r1. If that cardinality is one, then we can remove that test case from the old reduced test set. In the

below example, the cardinality of t1 is one, so we will remove t1 from the old reduced test set. In this case, we can cover all the requirements by {t3} only. So the new reduced test set will be {t3}.

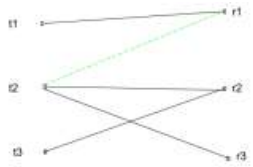

Fig. 4. New coverage graph after adding the edge {t2, r1}

3.2.2 Added edge does not belong to the old reduced test set

The graph in Fig. 5 shows an example of edge addition that does not belong to the old reduced test set. When we ran the old coverage graph in a traditional greedy method, we found that the reduced test was {t1, t2}. Now we added the edge {t3, r2}. We know that t3 does not belong to the old reduced test set. We found that r1 is not covered by another test case that belongs to the old reduced test set, so nothing will be affected. The new reduced test set will be {t2 and t3} as the old reduced test set.

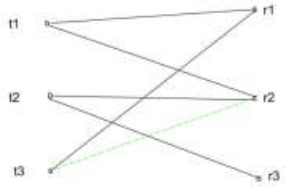

Fig. 5. New coverage graph after adding the edge {t3, r2}

Algorithm

1. Reduced testSet as ReducedSet ;
2. Add edge as AddEdge(e1) ;
3. Add AddEdge(e1) (connected by Requirement(r1) and TestCase(t1)) in CoverageGraph ;
4. if AddEdge(e1) belongs to TestCase(t1) which is not in Reduced_Set then
5. do nothing for AddEdge(e1) ;
6. else
7. if AddEdge(e1) belongs to Requirement(r1) that is covered by ReducedSet - { TestCase(ti) } and degree of TestCase(ti) is one then
8. ReducedSet = ReducedSet - { TestCase(ti) }
9. else
10. do nothing for AddEdge(e1) ;
11. end
12. end
13. return Reduced testSet as ReducedSet ;

Line1 In the ReducedSet set, we will store the final reduced test set. Line 2 AddedEdge, we store the test case and the requirement relationship. Line 4-5 If the added edge does not belong to the reduced test

set, we will do nothing. Line 7-8 If the added edge belongs to the reduced test set and the test case's cardinality, which covered the old reduced test set requirement, is one, then we will remove that test case. Line 13 returns a reduced test set.

4 Experiment Evaluation, Result and Analysis

This research work evaluated Flower Method, the traditional greedy sequential method, the traditional greedy parallel method, and the parallel RAFI method. We evaluated different factors in different benchmarks.

RQ1: Which method is time-efficient - the Traditional greedy method or the Flower method?.

RQ2: How much can we reduce the size of the test suite?

RQ3: Is the RAFI method more time-efficient to find the reduced test set compared with traditional greedy across versions?

RQ4: Is the RAFI method efficient in terms of reduction compared with traditional greedy across versions?

4.1 Experiment Subject

In this, we will evaluate our experiment in the set of standard benchmarks (SIR). In this, there are a total of eight benchmarks, namely, tcas, totinfo, schedule, printtokens, printtokens2, replace, schedule2, space. From the benchmarks in Table 1, we can say that the requirements are between 58 to 1515. Total test cases are between 1052 to 13585. The total numbers of edges are between 12825 to 5713638. We will run our Flower method, traditional greedy method, both sequential and parallel, in these standard benchmarks. We can see the properties of each benchmark from Table 1.

Table 1. Standard Benchmarks

S. No.	Benchmark	Test Cases	Requirements	Edges
1	totinfo	1052	132	88682
2	tcas	1608	58	39413
3	space	13585	1514	5713638
4	schedule2	2710	125	258929
5	schedule	2650	100	203000
6	replace	5542	229	628464
7	printtokens2	4115	200	12825
8	printtokens	4130	157	344020

4.2 Experiment Setup

Our aim is 1) Find the minimal test suite such that it can cover all the requirements. 2) Find the time taken to compute the minimal test suite for Flower, Traditional greedy sequential. 3) Speedup for RAFI

when we vary the number of threads 5) Speedup for RAFI when we vary the percentage of adding edges 4) Reduction rate for RAFI when we vary the percentage of adding edges 5) Speedup for RAFI when we vary dynamic changes in edges 6) Reduction rate for RAFI when we vary dynamic changes in edges. We will run all our implemented methods in the standard benchmarks (SIR). For evaluating the RAFI method, there are three parameters on which the performance can be varied. The parameters are

1. The number of threads on which we ran the RAFI.
2. Structural change in the old coverage graph means the percentage edges which were modified.
3. In edge modification, what is the percentage of added edges, and what is the percentage of the remaining edges deleted?

We will evaluate the RAFI method for these parameters. We will fix two parameters and will vary the remaining one parameter.

The operating system we are using on our machine is Linux version 4.4.0-159-generic, Operating System Type: 64-bit processor, RAM: 4 GB, and Processor: Intel 2.60GHz*4

4.3 Result and analysis

In this, we will explore the result of the Flower method, and the traditional greedy sequential method for finding the Reduced test set, and the RAFI parallel method for finding the reduced test set in the modified coverage graph. Here time complexity of RAFI method depends on various factors, it will depends on the previous coverage graph, and new modified coverage graph. It will also depend on the factors that how degree of test cases and requirements are changing.

4.3.1 Time Efficient

We ran Flower and traditional greedy methods to find the minimal test set to cover all the requirements. We ran our experiment on the benchmarks given in Table 1. We found the time complexity of the Flower method to be exponential. Therefore, it took exponential time and failed to find the Reduced test set in a reasonable time. On the other hand, the traditional greedy method is a polynomial-time algorithm. So we were able to find the reduced test set in a few milliseconds. Hence we can say that the traditional greedy algorithm can find the reduced test set in a reasonable time while the Flower method fails to do this. Fig. 6 shows how much the traditional greedy method is taking to find the reduced test set for different benchmarks.

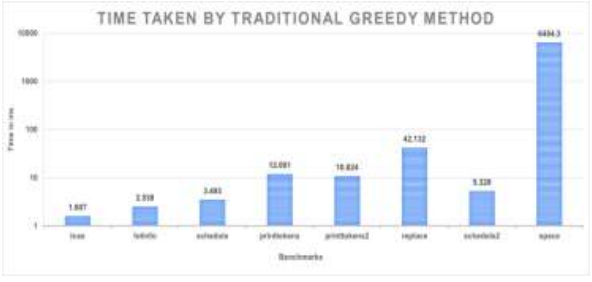

Fig. 6. Time taken by the traditional greedy method to find the test cases' reduced set size for different benchmarks

4.3.2 Reduced size of the test set

We ran our traditional greedy method for different benchmarks and found how much they reduced the test set to find the reduced test set to cover all the requirements. Fig. 7 shows the reduced set size of all benchmarks which we are using for evaluation.

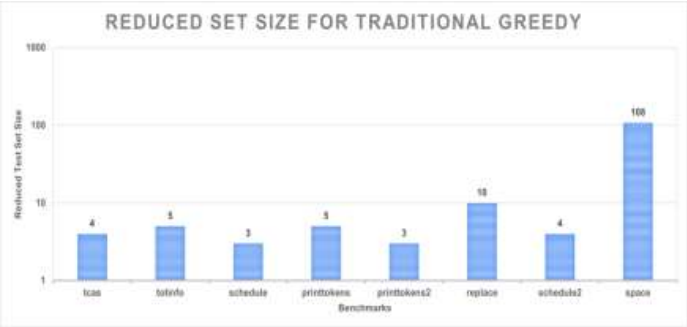

Fig. 7. Reduced set the size of the test cases for different benchmarks

4.3.3 Rate of Speed up when we vary the number of threads

We will vary the number of threads from 1 to 8. The percentage of dynamic changes in edges will be fixed which is 10%, and fixed 25% of edges added and 75% deleted. We calculated the speedup, the ratio of time taken by Greedy, and the time taken by RAFI. We obtained the maximum speedup when the number of threads is 4.

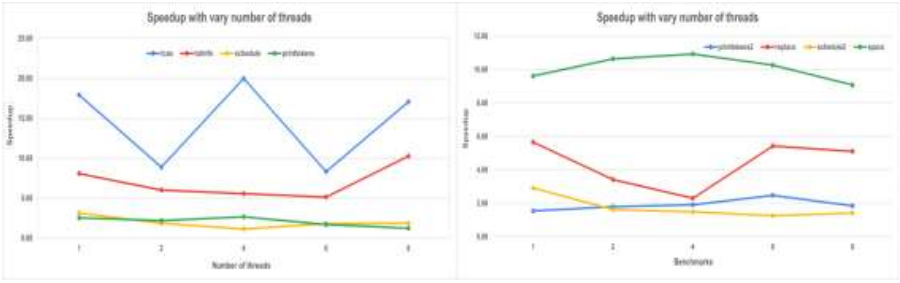

Fig. 8 Speedup of the RAFI method compares traditional greedy parallel to find the reduced set size of the test cases for different benchmarks

From these two graphs in Fig. 8, we can see that as we increase the number of threads, we are getting speedup, but after a certain time, speedup got reduced. Because when the number of threads was increased, most of the time was being wasted in scheduling the thread instead of doing the required work.

4.3.4 Rate of speed up when we vary the percentage of adding edges

We will vary the percentage of edge addition from 0% to 100%, and the rest will be edge deletion. We fixed the number of threads to 4 and fixed the percentage dynamic change to 1%. We calculated the speedup, which is the ratio of time taken by Greedy and time taken by RAFI. We found the maximum

speed of 489 for schedule2 benchmarks. The maximum speedup we got when the percentage of adding edges was 100%.

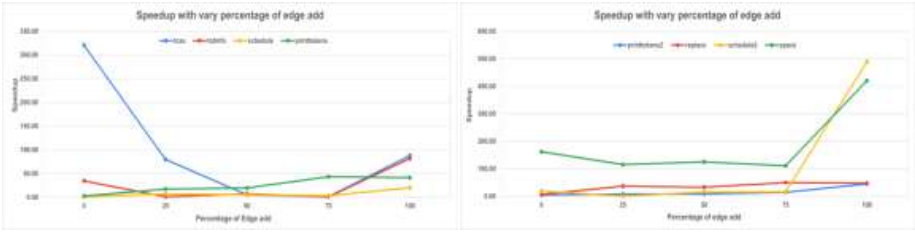

Fig. 9. Speedup of the RAFI method compares traditional greedy parallel to find the reduced set size of the test cases for different benchmarks

From these two graphs in Fig. 9, we can see that as we are increasing the percentage of edge addition, then speed was increased except in the tcas benchmark.

4.3.5 Reduction rate when we vary the percentage of adding edges

In this, we evaluated the normalized reduced set size for the above three parameters. We calculated the reduction rate, which is the ratio of reduced test set size of traditional Greedy and reduced set size of RAFI method.

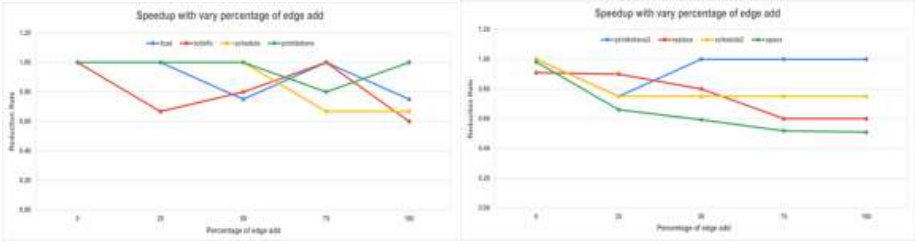

Fig. 10. Reduction rate of the RAFI method with a comparison of traditional greedy parallel for different benchmarks

From these two graphs in Fig. 10, we can see that the reduction rate varies from 0.52 to 1. Still, there is a possibility that we can get a reduction rate greater than one. The maximum speed up of reduction rate we are getting when the percentage of adding edges is 0% and deleting edges is 100%.

4.3.6 Rate of speed up when we vary dynamic changes in edges

In this, we will vary the percentage of dynamic changes in edges from 0.1 to 15%. We fixed the number of threads to 4 and fixed the 25% of edges added and 75% deleted. We calculated the speedup, which is the ratio of time taken by Greedy and time taken by RAFI. We obtained the maximum speed up when the dynamic changes were 0.1%.

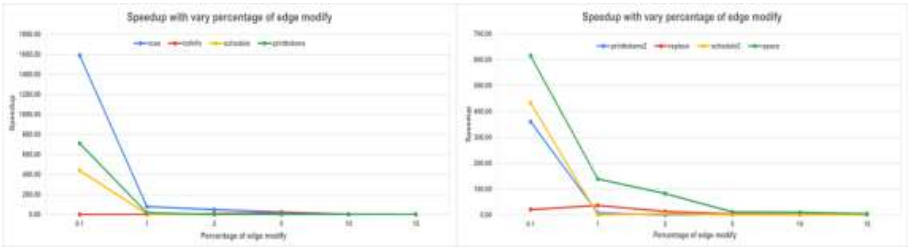

Fig. 11. Speedup of the RAFI method compares traditional greedy parallel to find the reduced set size of the test cases for different benchmarks

From these two graphs in Fig. 11, we can see the maximum speedup we get for benchmark tcas, equal to 1590, and the maximum speed up when the percentage of dynamic changes is 0.1%.

4.3.7 Reduction rate when we vary dynamic changes in edges

In this, we evaluated the normalized reduced set size for the above three parameters. We calculated the reduction rate, which is the ratio of reduced test set size of traditional Greedy and reduced set size of RAFI method.

From these two graphs in Fig. 12, we can see that the reduction rate varies from 0.34 to 1. Still, there is a possibility that we can get a reduction rate greater than one. The maximum speed up of reduction rate was obtained when the percentage of dynamic changes is 0.1%.

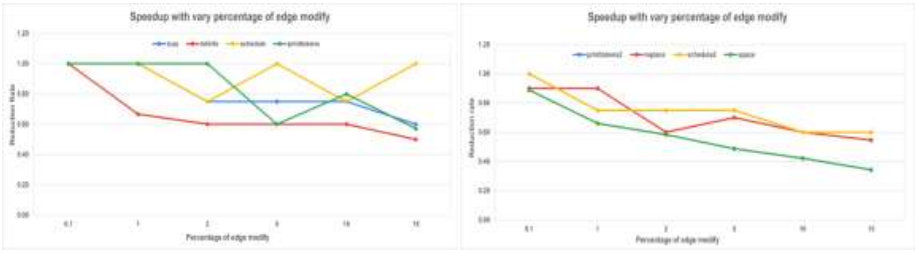

Fig. 12. Reduction rate of the RAFI method with the comparison of traditional greedy parallel for different benchmarks

5 Conclusion and Future Work

Given a set of test cases and the software requirement, we aimed to find the reduced test set, such that it covers all the requirements. We implemented some existing exponential methods such as Flower and some polynomial methods such as the Traditional greedy method. We ran these methods in some standard benchmarks (SIR) and found that the traditional greedy method can find the reduced test set reasonably, but the Flower method failed to do this in a reasonable time. Further, we found that the traditional greedy method can reduce the test set up to 0.07%. Further, we implemented the RAFI method to find the reduced test set in the modified coverage graph. We aimed to find the reduced test set using the old reduced test set and edges modified to cover all the requirements. We ran the RAFI method, and for comparison, we ran the Traditional greedy method. We found that RAFI gives a reduced test set in very little time compared to Traditional Greedy. We got speedup up to 1590X. The reduction rate varies from 0.38 to 1 for the RAFI method.

We can test different data structures like trees and adjacency lists to store the graph to reduce memory access and space in future work. Further, adding the notion of test case execution time, suppose edges are weighted. Here, weighted means that some test cases may take longer to execute. Then, in this situation, how will the reduced test set be found?. Because if test cases are weighted, then there is a possibility that the reduced test set whose cardinality is more can give better results.

References

[1] Ti, L. C. Et al. (2017). Empirically evaluating Greedy-based test suite reduction methods at different levels of test suite complexity. *Science of Computer Programming,* 150: 1-25.

[2] Kapfhammer, G. M. (2010). Regression testing. In Encyclopedia of Software Engineering Three-Volume Set (Print). *Auerbach Publications*, 893–915.

[3] Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173-210.

[4] Rothermel, G. et al. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of International Conference on Software Maintenance* (Cat. No. 98CB36272).

[5] Wong, W. E. et al. (1998). Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4): 347-369.

[6] Gligoric et al. (2015). Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[7] Marchetto, A. et al. (2015). A multi-objective technique to prioritize test cases. *IEEE Transactions on Software Engineering*, 42(10): 918-940.

[8] Zhang, L. et al. (2011). An empirical study of junit test-suite reduction. In *IEEE 22nd International Symposium on Software Reliability Engineering*.

[9] Lin, J. W. et al. (2009). Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology*, 51(4): 679-690.

[10] Jeffrey, Dennis, and Gupta, N. (2007). Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2): 108-123.

[11] Qu, X. et al (2008). Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[12] Jeffrey, Dennis, and Gupta, N. (2005). Test suite reduction with selective redundancy. In *21st IEEE International Conference on Software Maintenance*.

[13] Gotlieb, Arnaud, and Marijan, D. (2014). FLOWER: optimal test suite reduction as a network maximum flow. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[14] Cormen, T. H. et al. (2009). Introduction to algorithms. *MIT press*.

[15] Black, Jennifer, Melachrinoudis, E. and Kaeli, D. (2004). Bi-criteria models for all-uses test suite reduction. In *Proceedings of 26th International Conference on Software Engineering*.

[16] Chen, Z. et al. (2008). A Degraded ILP Approach for Test Suite Reduction. In *SEKE,* 494-499.

[17] Li, Z., Harman, M. and Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4): 225-237.

[18] Harrold, M. J., Gupta, R. And Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3): 270-285.

[19] Smith, A. M. and Kapfhammer, G. M. (2009). An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 2009 ACM symposium on Applied Computing*, 461-467.

[20] Chen, T. Y. and Lau, M. F. (2003). On the divide-and-conquer approach towards test suite reduction. *Information sciences*, 152: 89-119.