Proc. of the Intl. Conf. on Advances In Computing, Communication and Information Technology - CCIT 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-010-1 doi: 10.15224/ 978-1-63248-010-1-21

Test Cases Reduction through Prioritization Technique

Avinash Gupta, Nayaneesh Mishra, Dushyant Kumar Singh[1] and Dharmender Singh Kushwaha[1]
1 MNNIT Allahabad, India

Abstract. Regression testing is a costly and time taking affair. Because of time and resource constraints it is not possible to run all the test cases of the regression test suite. Prioritization of test cases provides a way to run highest priority test cases in the first phase. It helps to improve the percentage of fault detected in given time and is found to work better with feedback mechanism. History Based approach is one of the methods to prioritize the test cases based on the feedback of each test case. The feedback for each of the test cases is obtained from the history of each of the test cases in terms fault detection, number of executions and other such factors. This paper proposes the prioritization of test cases for the modified lines of code. The results establish that number of sessions required for running the test cases reduces by almost 33%.

Keywords: Feedback in prioritization, Prioritization, History Based, percentage of Fault detection, Test Suite

1 Introduction & Related Work

Estimates indicate that software maintenance activities account for as much as two-third of the cost of software production [16]. Regression testing forms an expensive but necessary task in the maintenance phase. Three different techniques have, therefore, been proposed for test suite reduction as - prioritization, selection and minimization of test suite.

A test suite *minimization* technique lowers the cost by reducing a test suite to a minimal subset that maintains equivalent coverage of original set with respect to particular test adequacy criterion [12]. Test suite minimization techniques [3], however, can have some drawbacks. Although one clan of researchers think that, in certain cases there is little or no loss in the ability of a minimized test suite to reveal faults [2] in comparison to its un-minimized original, the other clan thinks otherwise [12, 14]. The fault detection ability of test suites can be severely compromised by minimization.

Regression test selection technique [17] attempts to reduce the time required to retest a modified program by selecting some subset of the existing test suite, as in case of minimization. However, there is some difference. As mentioned by S. Yoo, M. Harman [14], both problems are about choosing a subset of test cases from the test suite but the key difference between these two approaches in the literature is whether the focus is upon the changes in the subject under test. Test suite minimization is often based on metrics such as coverage measured from a single version of the program under test. Some selection techniques select test cases based on program specification while most techniques select test cases based on the information about the code of the program and the modified version [18, 19, 20, 21 and 22]. However, selected test cases can be fault revealing only if a test case t is modification-revealing. But finding modification revealing test cases is an un-decidable problem [17].

Test case *prioritization* is the process of scheduling test cases in an order to meet some performance goal [13]. Prioritization gives priority to test cases based on criteria [15, 23]. The criteria may be code coverage etc. However, one of the limitations in this process is that the fault detection efficiency of the test suite may be compromised. The test suite may contain test cases on higher priority which may not be able to detect the errors [5]. Hence, several techniques have been proposed for prioritizing the existing test cases to accelerate the rate of fault detection in regression testing. Some of these approaches are Coverage-based Prioritization [13], Interaction Testing, Distribution-based Approach [7], Requirement-based Approach, and the Probabilistic Approach [6]. All these approaches apart from probabilistic approach referred above consider prioritization as an unordered, independent and one-time model. They do not take into account the performance of test cases in the previous regression test sessions, such as the number of times a test case revealed faults [15]. History Based Approach (HBA) has been applied to increase the fault detection ability of the test suite. Kim and Porter [6] considered the problem of prioritization of test cases as a probabilistic approach and defined the history-based test case prioritization. Alireza et al. [5] proposed an extension of history-based prioritization proposed in [6], and modifies the equation given by Kim and Porter [6], to have dynamic coefficients. The priority is calculated using the mathematical equation by computing the coefficients of the equation from the historical performance data.

In this paper, we propose a new approach which is an extension of the history based approach in [6]. Unlike in [6], where the prioritization equation has been applied on each test case, we apply the approach on each modified line of the code. The rest of the paper is organized as follows. In Section 2, we present the proposed approach and implementation. Section 3 describes performance analysis and comparison results. We conclude the paper and discuss future work in Section 4.

From the discussion above, it can therefore be easily deciphered that those approaches which consider history to prioritize the test cases would certainly be better in terms of fault detection effectiveness than those that did not. HBA approach takes into account the history of test cases while prioritizing them in the present session to increase the fault detection effectiveness of the test suite.

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances In Computing, Communication and Information Technology - CCIT 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-010-1 doi: 10.15224/ 978-1-63248-010-1-21

2 Proposed Approach

In our proposed approach, we have extended the previous approach [5] by prioritizing the modified lines. The history is kept for each of the modified lines which act as feedback for the next session. In our proposed approach, the test cases are selected for each modified line such that the test case is having the maximum coverage among all the test cases which contain the modified line. By the phrase 'containing a modified line' we mean that the test case executes the modified line in its line of execution.

Step1: Extract History from Database
In this phase all the parameter values of the previous session are extracted from the database by the program into arrays for use in the present session.

Step2: Input Modified Lines The modified lines are taken as input from the user through a well defined interface of the program. Any new test cases are also entered through this interface.

Step3: Find max coverage and ax coverage test case
If the modified line say, m_i is present in a test case T_i , then max coverage of line m_i = max (no. of lines in T_i /total number of lines) * 100 for all T_i in which m_i is found. T_i is the max coverage test case for m_i, if T_i has max code coverage.

Step4: Calculate Priority Value for Modified Value
For each modified line, the priority value is calculated using Eq. (1)[11].

$$PR_k = (\alpha.h_k + \beta.PR_{k-1})/k \qquad (1)$$
$$\alpha = (1 - ((fc_k + 1)/(ec_k + 1))^2)^{h_k} \qquad (2)$$
$$\beta = ((fc_k + 1)/(ec_k + 1))^x \qquad (3)$$

In Eq (1), $0 \leq \alpha, \beta < 1, \; k \geq 1$.
In Eq. (3), x=1 if the test case has revealed some fault in the previous session and x=2 if the test case has not revealed any fault in the previous session. In Eq. (1), h_k is the test cases execution history. ec_k denotes the total number of executions done by a test case till the session k. fc_k denotes the total number of faults detected by a test case till the session k. In Eq. (1), PR_0 is defined for each test case as the percentage of code coverage of the test case. The presence of PR_0 will be helpful in refining the ordering of the test cases in the first session.

Step 5: Prioritize Modified Lines
Modified lines are prioritized by their corresponding PR_k, in descending order. If the modified lines m_1, m_2 *and* m_3 have PR_k values as - PR_k [m_1] = 10.56
PR_k [m_2] = 54.56
PR_k [m_3] = 9.64. Hence ordering would be - m_2, m_1 , m_3.

Step 6: Prioritize Test cases in order of modified lines
Max coverage test case for, say $m_1 = T_2$, $m_2 = T_1$, $m_3 = T_3$
Hence ordering of test cases in order of the respective modified line m_2, m_1 *and* m_3 would be - T_1, T_2, T_3.
Step 7: Output prioritized the test cases
The final output for session k is T_1, T_2, T_3. After prioritizing, the test cases are executed. Let us assume that only 40% of all the test cases prioritized are able to get executed. Out of all the test cases executed, there are certain test cases which detect fault, and after debugging a fault is detected. Then, the parameters are updated in the following manner-

- For each executed line i, in the present session k, increment the value of parameter ec_k by 1 and set the value of h_k to 0 for the line i.
- For the rest of the lines which did not execute in the present session k, increment the value of h_k by 1.
- For each faulty line i, detected in session k, increment the value of parameter fc_k by 1 for the line i.

The database is updated with all these modifications.

Implementation

The proposed approach has been implemented using 'C' program. The database to keep the history and all the test cases has been kept in two text files: 'textcases.txt' and 'Parameters.txt'. The file 'testcases.txt' contains the test cases in the form of traces of each of the test case. This means that 'testcases.txt' contains the lines which will be covered by each test cases once they execute. There is another file called 'Parameters.txt' which keeps the history of all the parameter values which will be used to calculate the priority value PR_k for each modified line. The C program, itself has namely 2 sections:

- **An interface:**

This is the interface provided to insert all the test cases. This interface is also used to insert the line numbers of modified, deleted and added lines. It is also used to insert the line number of faulty line of the previous session.

- **Prioritizing section:**

This section calculates the priority of each modified line and outputs them in order of priority. This section also outputs all the test cases in order of priority.

Case Study I:

The proposed approach is demonstrated with an example here. We are considering a 'C' program that implements Heap Sort. The program has 60 lines of code.
The program is modified at 6 different lines numbers 12, 20, 24, 30, 38, 46. The changes in each of the modified lines are shown in Table 1. These modifications are small changes which change one operator with the other one, like changing the > with <, or changing == (equal to operator) with !=.(not equal to) or change a value of a constant or other such small changes.

Table 1: Changes in the Sample program

Line no.	Original line	Modified line
12	i>=0	i<=0
20	(left <= n && a[left] > a[i])	(left <= n || a[left] > a[i])
24	largest=i	largest=right
30	largest!=i	largest==i
38	a[i]=a[j]	a[j]=a[i]
46	i>0	i<0

The Control Flow Graph (CFG) for the program of Heap Sort and the modified program of Heap Sort is

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances In Computing, Communication and Information Technology - CCIT 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-010-1 doi: 10.15224/ 978-1-63248-010-1-21

same as given in Figure 2. The CFG has been developed using SourceCode Visualizer Tool [2] which is a plug-in of Eclipse. Based on path coverage, we have the following 9 test cases:

T1 : 11, 12, 13, 15
T2 : 11, 12, 15
T3 : 37, 38, 39, 40
T4 : 54, 55, 56, 57, 59
T5 : 54, 55, 56, 59
T6 : 43, 44, 46, 51
T7 : 43, 44, 46, 47, 48, 49, 51
T8 : 18, 19, 20, 21, 27, 28, 30, 31, 32, 34
T9 : 18, 19, 20, 24, 27, 30, 34

These test cases are kept in the 'testcases.txt' file.

It has been decided that only 40% of all the test cases can be executed in each test session.

Number of lines of code in the program of Heap Sort = 60. Number of test cases for the program of Heap Sort = 9. Number of test cases which can be executed in a test session = 40% of the total test cases = 40% of 9 = 4(approx.)

Now, for session k=1:
Step 1: Extract history from database: The test cases are extracted from the 'testcases.txt' file into the program which is developed to implement the proposed algorithm. Before the first session, for each modified line, the value of parameters, namely hk, eck, fck, prk and prk 1, is set to 0. For each modified line the value of parameter x is set to 2.

Step 2: Input line numbers of modified lines As is mentioned in Figure 3, the input interface of the program takes as input the line number of modified lines i.e. 12, 20, 24, 30, 38, 46.

Step 3: Find max coverage and max coverage test case Code coverage of a test case T= (no. of lines in the test case / total number of lines in the program)* 100. Let's consider Line 12, which is found in test cases: T1, T2.

So, Code coverage of test case T1 = (4 / 100) * 100 = 4%
Code coverage of test case T2 = (3 / 100) * 100 = 3%
Hence, max code coverage for line 12 = 4%

Since, Test case T4 is having the max code coverage value of 4% among all the test cases containing the line 12. So, max code coverage test case for line 12 = T1
As we calculated the value of max code coverage and found out the max code coverage test case for line number 12, we can similarly compute the value of max code coverage and find out the max code coverage test case for line number 20, 24, 30, 38 and 46. The results are:
Max code coverage test case (MCCTC) for line 20 = T8, and Max Code Coverage (MCC) = (10 / 100) * 100 = 10%

MCCTC for line 24, 30, 38 and 46 are T9, T8, T3 and T7 respectively. The MCC for line 24, 30, 38 and 46 are 7%, 10%, 4% and 7% respectively.

Step 4: Calculate priority values for modified lines
For session *k*=1, the status of all the parameters is as shown in the Table 2.

Table 2: Status of parameters before session k=1

Line No.	h_k	ec_k	fc_k	x	PR_k	PR_{k-1}
12	0	0	0	2	0	4
20	0	0	0	2	0	10
24	0	0	0	2	0	7
30	0	0	0	2	0	10
38	0	0	0	2	0	4
46	0	0	0	2	0	7

Now, substituting the values of h_k, ec_k, fc_k, x and PR_{k-1} in Eq. (3), Eq. (2) and Eq. (1) from Table 2, to calculate the corresponding value of PR_k for lines 12, 20, 24, 30, 38 and 46. We get: PR_k [12] = 2.0, PR_k [20] = 5.0, PR_k [24] = 3.5, PR_k [30] = 5.0, PR_k [38] = 2.0 and PR_k [46] = 3.5.

Step 5: Prioritize the modified lines in order of priority value Based on the priority values calculated in step 4, modified lines in order of priority as per PR_k values are: 20, 30, 24, 46, 12 and 38.

Table 3: Lines in order of priority and max code coverage test case for each line after session k=1

Lines in order of priority based on the value of PR_k	20	30	24	46	12	38
Max Coverage Test case for each modified line	T8	T8	T9	T7	T1	T3

Step 6: Prioritize the test cases in order of modified lines
As can be seen from Table 3, Max code coverage test case for Line 20 is T8, Line 30 is T8, Line 24 is T9, Line 46 is T7, Line 12 is T1 and line 38 is T3. Max coverage test cases in order of their respective modified lines after ignoring the duplicates are: T8, T8, T9, T7, T1 and T3. Removing the repeated test case T8 from the second place, we get T8, T9, T7, T1 and T3.

Step 7: Output prioritized the test cases
The final output for session k=1 is T8, T9, T7, T1 and T3.
Since, 40% of the total test cases can be executed i.e. 4 test cases. Test cases T8, T9, T7 and T1 are executed. T3 is left out from the execution in session 1. During execution the test cases T8, T9, T7 and T1 fail. Debugging reveals fault at lines 20, 30, 24, 46 and 12. These errors are subsequently fixed.

Session 2: (For session k=2) Lines to be prioritized in session *k*=2 are 20, 30, 24, 46, 12 and 38. The status of parameters shown in Table 3 will be used in session 2. All the seven steps of the proposed approach are followed for k=2

Table 4: Status of parameters before session k=2

Line No.	h_k	ec_k	fc_k	x	PR_k	PR_{k-1}
12	0	1	1	1	0	4
20	0	1	1	1	0	10
24	0	1	1	1	0	7

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances In Computing, Communication and Information Technology - CCIT 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-010-1 doi: 10.15224/ 978-1-63248-010-1-21

30	0	1	1	1	0	10
38	1	0	0	2	0	4
46	0	1	1	1	0	7

Table 5: Lines in order of priority and max code coverage test case for each line after session k=2

Lines in order of priority based on the value of PR_k	20	30	24	46	12	38
Max Coverage Test case for each modified line	T8	T8	T9	T7	T1	T3

The final output for session *k*=2 is as given in Table 5. Max coverage test cases in order of their respective modified lines after ignoring the duplicates are: T8, T9, T7, T1 and T3 i.e. test cases appear in the same sequence of priority as in session 1. Since only 40% of the total test cases i.e. 4 test cases can execute, the test cases T8, T9, T7 and T1 are again executed in the given order as in session 1. No new fault is found in session 2 since the lines 20, 30, 24, 46 and 12 have already been fixed in session *k* = 1.

Session 3: (For session k=3) the status of parameters shown in Table 4 will be used in session 3.

Table 6: Status of parameters before session k=3

Line No.	h_k	ec_k	fc_k	x	PR_k	PR_{k-1}
12	0	2	1	2	0	2
20	0	2	1	2	0	5
24	0	2	1	2	0	3.5
30	0	2	1	2	0	5.0
38	2	0	0	2	0	2.0
46	0	2	1	2	0	3.5

Table 7: Lines in order of priority and max code coverage test case for each line after session k=3

Lines in order of priority based on the value of PR_k	20	30	38	24	46	12
Max Coverage Test case for each modified line	8	8	3	9	7	1

Max coverage test cases in order of their respective modified lines after ignoring the duplicates in session *k* = 3 are: T8, T3, T9, T7 and T1. The four test cases of higher priority i.e. T8, T3, T9 and T7 are executed. Test case T3 fails. The program is debugged and error is revealed at line number 38. The error at line number 38 is then corrected.

Finally, all the errors were revealed in 3 sessions. Hence, we can conclude that the proposed approach has a better rate of fault detection than the Alireza approach.

Summary for case study I:
The number of sessions required to reveal all the errors = 3.
5 out of 6 revealed in first session.
Percentage of error revealed in first session = (5/6) * 100 = 83.3%
Total number of test cases executed in all the three sessions = no. of test cases executed in session 1 + no. of test cases executed in session 2 + no. of test cases executed in first session 3
= 4 + 4 + 4
=12

Case Study II

We apply the proposed algorithm on program containing even more number of lines of code than the one considered in case study II. We consider a program of linked list.

Number of lines of code in the program of Linked List = 382
Number of test cases for the program of Linked List = 51
Number of test cases which can be executed in a test session = 40% of the total test cases
= 40% of 51 = 20(approx.)

Summary of case study II:
Total number of test cases = 51
Total number of test cases executed to reveal all the errors = 14 (in one session)
This concludes that all the errors are revealed by executing just 27.45% of the total test cases and that too in just one session.

<u>Summary result of case study I and case study II:</u>

In case of programs with less number of lines of code as in case study II, most of the errors are revealed in the first few sessions after executing a very small percentage of test cases of the total number of test cases in all the sessions together. As can be seen in case study II, that 5 out 6 (83% approx.) errors get revealed in session 1. It is only to reveal the remaining one error that the number of session extends to 3.

3 Performance Analysis:

Proposed approach is illustrated by ten programs of Java, C and C++ based platforms. After that, we compared our proposed approach with Alireza et al. approach. For comparison, we applied our proposed approach as well as Alireza approach on eight example programs and five faults were seeded in each of the programs with multiple sessions of regression test. Results in Table 8 shows the faults detected in each session by our proposed approach and Alireza approach.

4 Conclusion & Future Work

Proposed approach is helpful in early detection of fault. The proposed approach brings out those test cases to the front which are relevant to the modifications made. This is because; the proposed approach is able to prioritize test cases in less number of iterations. The results establish that number of sessions required for running the test cases reduces by almost 33%. For smaller programs, reduction in number of test cases that are required to be executed from the total available set is not substantial but for larger programs, the reduction

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances In Computing, Communication and Information Technology - CCIT 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-010-1 doi: 10.15224/ 978-1-63248-010-1-21

seen is to the tune of about 70 %. The proposed approach is able to locate a high percentage of the faults in the first few test sessions. Thus the proposed approach is able to significantly reduce the regression test effort required either during bug fixing phase or during maintenance.

References

[1] H. Papadimitriou and K. Steiglitz. Combinatorial optimization: algorithms and complexity. Courier Dover Publications, 1998.

[2] T. Y. Chen and M. F. Lau.: Dividing strategies for the optimization of a test suite. Information Processing Letters, Vol. 60(3). (1996) 135–141.

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

[4] J. R. Horgan and S. London.: A data flow coverage testing tool for C. In IEEE Proceedings of the Second Symposium on Assessment of Quality Software Development Tools. IEEE (1992) 2–10.

[5] Alireza Khalilian, M. Abdollahi Azgomi, and Y. Fazlalizadeh.: An improved method for test case prioritization by incorporating historical test case data. Science of Computer Programming. Vol. 78(1). (2012) 93–116.

[6] J.-M. Kim and A. Porter.: A history-based test prioritization technique for regression testing in resource constrained environments. In Proceedings of the 24rd IEEE International Conference on Software Engineering. ICSE (2002) 119–129.

[7] D. Leon and A. Podgurski.: A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In 14th IEEE International Symposium on Software Reliability Engineering. ISSRE (2003) 442–453.

[8] M. Marr´e and A. Bertolino.: Using spanning sets for coverage testing. IEEE Transactions on Software Engineering. Vol. 9(11). IEEE (2003) 974–984.

[9] J. Offutt, J. Pan, and J. M. Voas.: Procedures for reducing the size of coverage based test sets. In Proceedings of the 12th International Conference Testing Computer Software. Citeseer (1995) 111-123.

[10] K. Aggrawal, Y. Singh, and A. Kaur.: Code coverage based technique for prioritizing test cases for regression testing. ACM SIGSOFT Software Engineering Notes, Vol. 29(5). ACM (2004) 1–4.

[11] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In Proceedings of the IEEE International Conference on Software Maintenance. IEEE (1998) 34–43.

[12] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong.: Empirical studies of test-suite reduction. Software Testing Verification and Reliability. Vol 12(4). (2002) 219–249.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold.: Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering. Vol. 27(10). IEEE (2001) 929–948.

[14] W. E.Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini.: Test set size minimization and fault detection effectiveness: A case study in a space application. Journal of Systems and Software. Vol. 48(2). (1999) 79–89.

[15] S. Yoo and M. Harman.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability. Vol. 22(2). (2012) 67–120.

[16] S.Schach. Software Engineering. Aksen Associates, Boston, MA, 1992.

[17] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. Software Engineering, IEEE Transactions on, 22(8):529–551, 1996

[18] Lee, John AN, and Xudong He. "A methodology for test selection." Journal of Systems and Software 13.3 (1990): 177-185.

[19] Agrawal, Hiralal, Joseph Robert Horgan, Edward W. Krauser, and Saul London. "Incremental Regression Testing." In ICSM, vol. 93, pp. 348-357. 1993..

[20] Binkley, David. "Semantics guided regression test cost reduction." Software Engineering, IEEE Transactions on 23.8 (1997): 498-516.

[21] Gupta, Rajiv, Mary Jean Harrold, and Mary Lou Soffa. "An approach to regression testing using slicing." Software Maintenance, 1992. Proceerdings., Conference on. IEEE, 1992.

[22] Rothermel, Gregg, and Mary Jean Harrold. "A safe, efficient regression test selection technique." ACM Transactions on Software Engineering and Methodology (TOSEM) 6.2 (1997): 173-210.

[23] Jones, James A., and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage." Software Engineering, IEEE Transactions on 29.3 (2003): 195-209.

Table 8: Proposed Approach Results

Program	Line No. modified	Session No.	Faulty Line detected by Proposed approach	Faulty Line detected by Alireza approach	No. Of Sessions in Proposed approach	No. Of Sessions in Alireza approach
Branch Coverage Sample Program	26, 30, 35, 40, 8	S1 S2	26, 30, 35 40, 8	26, 30 35, 40	2	2
Bank Account	6, 17, 22, 24, 27	S1 S2 S3	17, 22, 24, 27 - 6	17, 22, 24, 27 - 6	3	3
Library Managment	199, 172, 223, 143, 126	S1 S2 S3	199, 172,223 172, 126	199, 172 223, 172, 126	2	3
Kruskal Algorithm	97, 93, 106, 47, 110	S1 S2	97, 106, 110 47, 93	97, 106, 110 47, 93	3	3
Payroll Management System	70, 118, 124, 207, 231	S1 S2 S3	124, 118, 231, 207 70 -	124, 118, 231 70 207	2	3
Airline Reservation System	27,43, 64,97,126	S1 S2 S3	27,43, 64 97,126 -	27,43, 64 - 97,126	2	3
Tic-tac toe	159, 164, 191, 195, 261	S1 S2 S3	159, 164, 191, 195 261 -	159, 164 191, 195 261	2	3
Student Records Management System	24, 27, 66, 69, 80	S1 S2 S3	24, 27, 66 69 80	24, 27 66, 69 80	3	3

SEEK
DIGITAL LIBRARY