

BugLoc: Bug Localization in Multi Threaded Application via Graph Mining Approach

A. Adhiselvam¹, E. Kirubakaran², R. Sukumar³

¹Assistant Professor, Department of MCA, S.T.E.T Women's College, Mannargudi

²Additional General Manager, STTP (System), Bharat Heavy Electrical Limited, Trichy

³Professor, Department of CSE, Sethu Institute of Technology, Kariapatti

Email: adhiselvam@yahoo.com, ekiru@bheltry.co.in, rsukumar@sethu.ac.in

Abstract - Detection of software bugs and its occurrences, repudiation and its root cause is a very difficult process in large multi threaded applications. It is a must for a software developer or software organization to identify bugs in their applications and to remove or overcome them. The application should be protected from malfunctioning. Many of the compilers and Integrated Development Environments are effectively identifying errors and bugs in applications while running or compiling, but they fail in detecting actual cause for the bugs in the running applications. The developer has to reframe or recreate the package with the new one without bugs. It is time consuming and effort is wasted in Software Development Life Cycle. There is a possibility to use graph mining techniques in detecting software bugs. But there are many problems in using graph mining techniques. Managing large graph data, processing nodes with links and processing subgraphs are the problems to be faced in graph mining approach. This paper presents a novel algorithm named BugLoc which is capable of detecting bugs from the multi threaded software application. The BugLoc uses object template to store graph data which reduces graph management complexities. It also uses substring analysis method in detecting frequent subgraphs. The BugLoc then analyses frequent subgraphs to detect exact location of the software bugs. The experimental results show that the algorithm is very efficient, accurate and scalable for large graph dataset.

Keywords - Graph Mining, Pattern Mining, Heuristic Approach, Frequent Subgraph, Bug Localization

I. INTRODUCTION

Graph mining is widely used mining technique to mine large records of graph dataset that can be used to analyze Protein-Protein Interactions (PPI) [4], where the nodes represent the proteins. In the case of networking it is used to analyze the relationship about the devices, clients and nodes. Using this we can analyze the social relationships between the clients on the network. Our idea is to use graph mining technique [3] to detect bugs present in the software applications. In this paper, sample graph dataset is used, which represents thread, sub thread, total process and positive-negative process values. We use these data to form graph where the nodes in the graph represent the thread and its sub nodes represent sub thread, values in between the nodes mentions the threshold value. Graph mining is a better method in mining large logical data in detecting systematic relations than the data mining techniques. Graph mining is not a simple task to implement, forming a graph with large dataset and managing it seems to be a

complicated task. Many graph mining algorithms in the past presents a technique to mine relations about the graph data but none has attempted to mine the bug analysis and have not succeeded in detection of software bugs.

In the existing system, graph data is handled using grow and store method, whereas the graph mining algorithm like Apriori algorithm follows the heuristic approach [14]-[16]. The graph data is dynamically adopted while executing the algorithm. Grow and store method fails for large number of data. The GRAMI algorithm shows better solution in managing graph data. It holds the template of the data but not the actual values of the dataset. It also provides better solution in managing large data but it fails in analyzing Constraint Satisfaction Problem (CSP)[13]. However the CSP is not related to our model of work but its drawback should be related with our threshold calculation between thread nodes. gSpan algorithm solves the CSP in many graph dataset and stands as the standard graph mining algorithm for long time. In this paper, we adopt some methodologies in gSpan algorithm and develop the heuristic approach in dataset of thread processes to localize bugs in software application. In this paper, the following problems have been addressed to succeed in our approach of bug localization. They are:

- i. The large number of graph dataset should be formed as graph nodes and properly handled to perform further processes.
- ii. The threshold value should be calculated properly for every node.
- iii. The frequent threshold value should be calculated for every node in the graph.
- iv. The frequent subgraph should be identified and mined properly.
- v. The thread with the bug should be identified with the frequent threshold calculation and negative threshold value.

This paper discusses how to solve these problems through graph mining approach and detect the bugs in multi threaded application using our proposed algorithm named BugLoc. The organization of the paper is as follows. Section 2 provides related work of our research problem. Design of Bug localizer algorithm is provided in Section 3. Section 4 describes experiment and results of the algorithm. Final section contributes conclusion and future work.

II. RELATED WORK

This section discusses related work in many different directions. In Transactional mining [8], this setting is

concerned with mining frequent subgraphs on a dataset of many, usually small graphs. FSQ [2] constructs new candidate patterns by joining smaller frequent ones. The drawback of this approach is the costly join-operation and the pruning of false positives. gSpan[6] proposed a variation of the pattern growth approach. It uses an extension mechanism, where subgraphs grow directly from a single subgraph instead of joining two previous subgraphs. Other methods focused particularly on subsets of frequent subgraphs. MARGIN returns maximal subgraphs only, whereas CloseGraph generates subgraphs that have strictly smaller support than any of their parts. LEAP and GRAPH SIG, on the other hand, discovered important subgraphs that are not necessarily frequent. Although GRAMI [1] focused on the single large graph setting, it may also be easily specialized to support graph transactions. In single graph mining, the equally important single graph is stored as unique pattern with less work. The major difference is definition of an appropriate anti-monotone support metric. SIGRAM used the Maximum Independent Set (MIS) metric and proposed an algorithm that finds frequent connected subgraphs in a single, labeled, sparse and undirected graph [7],[8]. SIGRAM followed a grow-and-store approach, that is, it needs to store intermediate results in order to evaluate frequencies. Overall, SIGRAM needs to enumerate all isomorphism and relies on the expensive computation of MIS which is NP complete [11]. Hence the method is very expensive in practice.

Since the number of intermediate embeddings increases exponentially with the graph size, such approaches do not scale for large graphs. In contrast, GRAMI does not need to construct all the isomorphism. Hence, it can be scaled much larger graphs. More importantly, GRAMI supports frequent subgraph and pattern mining [28]. Thus, it allows for exact isomorphism matching and the more general distance-constrained pattern matching. Additionally, GRAMI supports constraint-based mining and works on directed, undirected, single and multi-labeled graphs. In Approximate mining [9], there is a work on approximate techniques for solving the frequent subgraph mining problem as well. In GREW, the authors proposed a heuristic approach that prunes large parts of the search space, but discovers only a small subset of the answers. GAPPROX employs an approximate version of the MIS metric. It mainly relies on enumerating all intermediate isomorphism but allows approximate matches [5]. SEUS is another approximate method that constructs a compact summary of the input graph.

This facilitates pruning many infrequent candidates, however, it is only useful when the input graph contains few and very frequent subgraphs [17]-[19],[27]. SUBDUE is a branch-and-bound technique that mines subgraphs that can be used to compress the original graph. Khan et al. proposed proximity patterns, which relax the connectivity constraint of subgraphs and identify frequent patterns that cannot be found by other approaches[9]. In contrast to the existing works, AGRAMI, an approximate version of GRAMI, may miss some frequent subgraphs, but the returned results do not have false positives. In Subgraph isomorphism[20]-[22], the frequent subgraph mining problem relies on the computation of subgraph isomorphism. This problem is NP-complete and the first practical algorithm that addressed that problem which follows

a backtracking technique. Since then, several performance enhancements were proposed, ranging from CSP based techniques, search order optimization, indexing to parallelization.

Although the state-of-the-art subgraph isomorphism techniques lead to significant improvements, they are not as effective in the frequent subgraph mining problem for two reasons: First, subgraph isomorphism techniques are effective in finding all appearances of a subgraph, while for the frequent subgraph mining task, it is sufficient to find the minimum appearances that satisfy the support threshold. This difference affects the way in which graph nodes are traversed, minimizing the number of node visited during search. Additionally, modern techniques employed global pruning and indexing techniques. GRAMI is based on a novel CSP method that overcomes the previous shortcomings and outperforms state-of-the-art subgraph isomorphism techniques. There are many works on pattern matching [22],[23] over graphs as well. R-JOIN supports reachability queries in a directed graph [10]-[12]. If two nodes v and v' are reachable in the query, then their corresponding mappings u and u' in the graph must also be reachable. DISTANCE-JOIN extends the idea to undirected graphs and accommodates constraints on the distance in the path. GRAMI presents an extension to support frequent pattern mining [24]-[26], the extended version adopts the pattern [29]. The conventional approaches have the drawbacks that every method occurs once within the graph. This leads to small graphs, which allows for graph-mining-based bug localization even with large software application. On the other side, much information about the program execution is lost, e.g., frequencies of the execution of methods and information on different structural patterns within the graph. The entropy approach omits substructures which are called more than twice in a row. Thus, it keeps more information than the other one, with the risk of generating very large graphs. The paper [30], the author compared two algorithms named gSpan and CloseGraph. Authors pointed out that these two algorithms showing a better performing in managing graph data. They also concluded that there is a possibility of detecting software bugs by improving those algorithms.

III. BUG LOCALIZER

From studying the above literature we assume that there will be a better way in detecting the bugs in software threads. For this, we have to adopt a new technique that should be differentiating from the previous mining techniques. The only problem that we face in localizing bugs is the way in which we can localize the bugs. In existing methodologies call graph uses the threshold value to analyze the weightage between the two nodes. The weightage threshold mentions the time difference between query submission and response time [9].

So, in our bug localization technique we must consider parameters such as positive process (Pprocess) and negative process (Nprocess) to detect weightage between two nodes. We calculate the threshold value by calculating difference between number of Pprocess and Nprocess occurred in between two threads. The Pprocess is the total count of processes executed before the error occurred. The Nprocess is

the number of errors which causes occurrence of the bug in the thread. The sample dataset is given in the Figure 1.

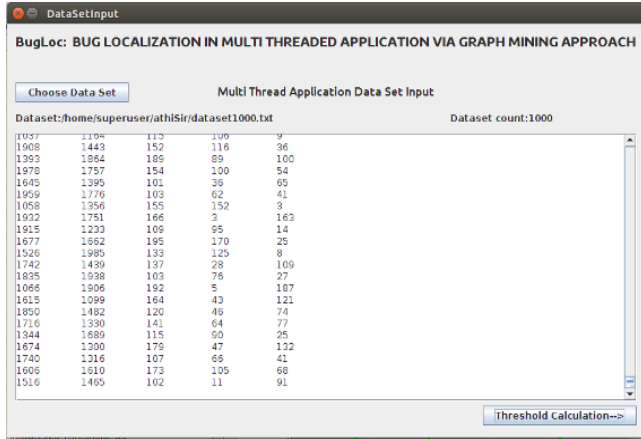


Figure 1 shows a screenshot of the 'DataSetInput' window. It displays a table with 5 columns: Thread ID, Subthread, Tprocess, Pprocess, and Nprocess. The data is organized into rows, with the first row being 1037, 1194, 112, 199, 9. The table is titled 'Multi Thread Application Data Set Input' and 'Dataset count:1000'.

Figure 1: Sample graph dataset

The format of dataset is given Table 1. The first parameter (XXXX) mentions the thread id in the program, which is also called as the base thread. The second parameter (YYYY) mentions the sub thread created by the base thread. The third parameter is Tprocess, which denotes the total number of processes that has to be executed between the thread and subthread. The fourth parameter is the Pprocess, which denotes the number of positive process executed in between base thread and sub thread. The fifth parameter is the Nprocess which denotes the negative process occurred due to the bug.

Table 1: Dataset format

Thread	Subthread	Tprocess	Pprocess	Nprocess
XXXX	YYYY	M	N	O

Using this sample graph dataset we form a graph and mine it using frequent subgraph mining technique with our proposed algorithm BugLoc. BugLoc includes the following five steps. Each step is presented in the following sub sections.

- Step 1: Threshold Calculation
- Step 2: Frequency Threshold Calculation
- Step 3: Thread Flow Identification
- Step 4: Frequent Subgraph Identification
- Step 5: Bug Localization

A. Threshold Calculation

Initially, the algorithm reads the input data from the file which holds the graph dataset. Graph dataset is read and divided into parameter values. The parameter values are stored as an object template which is used to reduce the memory usage while processing the large graph dataset. And then the threshold value is calculated from the input parameters such as Pprocess and Nprocess. The threshold value is the difference between the Pprocess and the Nprocess. The calculated threshold value is also stored into the object template. The procedure for threshold calculation is stated as follows. The output of this procedure is shown in Figure 2.

procedure thresholdCalculation ():
 read dataset

for data in dataset
 split data
 get thread
 get sub_thread
 get pprocess
 get nprocess
 threshold=difference(pprocess, nprocess)
end for
write threshold data
end procedure

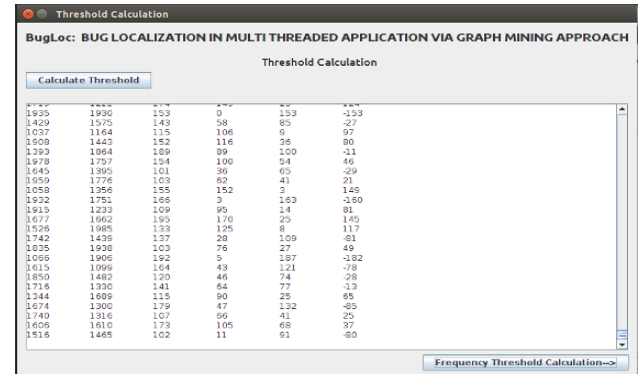


Figure 2 shows a screenshot of the 'Threshold Calculation' window. It displays a table with 5 columns: Thread ID, Subthread, Tprocess, Pprocess, and Nprocess. The data is organized into rows, with the first row being 1935, 1930, 153, 0, 153. The table is titled 'BugLoc: BUG LOCALIZATION IN MULTI THREADED APPLICATION VIA GRAPH MINING APPROACH'.

Figure 2: Result of Threshold Calculation

B. Frequency Threshold Calculation

After calculating the threshold values, frequency threshold values have to be identified for every thread. To calculate the frequency threshold value, the repeated threshold value is identified from the calculated threshold values in the object template. Now the frequency threshold value is also stored into the object template. In the similar way frequency threshold value is calculated for every threshold in the object template. The procedure to calculate frequency threshold is given below. The output of this procedure is shown in Figure 3.

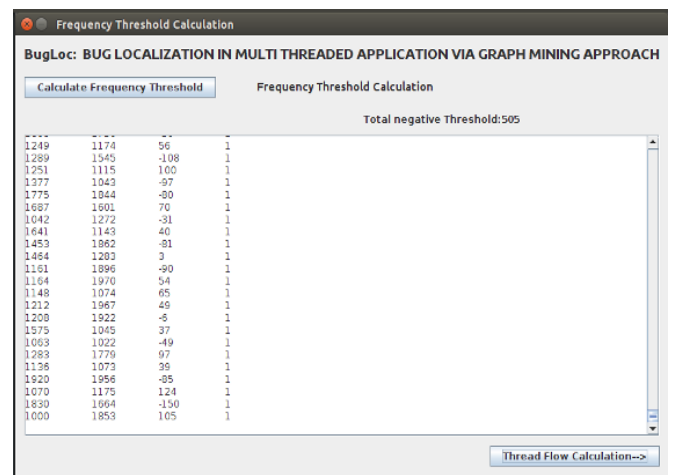


Figure 3 shows a screenshot of the 'Frequency Threshold Calculation' window. It displays a table with 5 columns: Thread ID, Subthread, Tprocess, Pprocess, and Nprocess. The data is organized into rows, with the first row being 1249, 1174, 56, 1, 1. The table is titled 'BugLoc: BUG LOCALIZATION IN MULTI THREADED APPLICATION VIA GRAPH MINING APPROACH'.

Figure 3: Result of Frequency Threshold Calculation

procedure frequencyThresholdCalculation ():
 get all_flows
 for flow in flow
 for nodes in flow
 set threshold ← threshold (node, node+1)
 end for

```

end for
for flow in flows
  for node in flow
    for threshold in node
      get repeated threshold
    end for
  end for
end for
store frequencyThreshold
end procedure

```

C. Thread Flow Identification

The next step is to identify the thread flow from the object template. This step of our algorithm performs thread flow calculation by ordering the thread and its sub thread until the last sub thread found from the object template. This is continued for every base thread. A separate array list is maintained to hold the set of thread flows. Each thread flow is represented as a complete string. The procedure for thread flow identification is given below. The output of thread flow identification is shown in Figure 4.

```

procedure threadFlowIdentification( ):
  read dataset from threshold_file
  for data in dataset
    listofthread ← data[0]
    listofthread ← data[1]
    listofthread ← data[2]
  end for
  sub_thread_present ← true
  while (sub_thread_present):
    for thread in list
      get sub_thread for thread
      add flow ← sub_thread
      thread ← sub_thread
      if !sub_thread_present
        sub_thread_present ← false
      end if
    end for
  end while
  write flow to file
end procedure

```

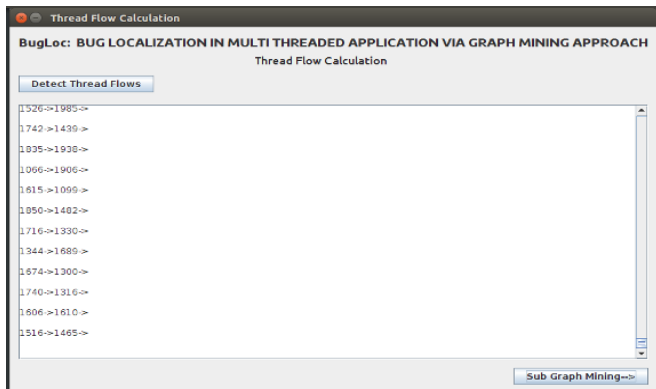


Figure 4: Result of Thread Flow Identification

algorithm. Unlike the existing algorithms, our algorithm takes each thread flow as a complete string, and holds the entire strings into single object. We do not need to use individual object for every node. By holding the thread flow as a single object, it can be easily managed and the frequent subgraph can be identified by processing thread flows using substring analysis method. The repeated substring in the thread flow mentions the frequent subgraph identified from every thread flow. The procedure for frequent subgraph identification is given below. The output of this step is shown in Figure 5.

```

procedure frequentSubgraphIdentification ( ):

```

```

  get all_flows
  for node in flows
    join (nodes)
    store flow
  end for
  for flow1 in flows
    for flow2 in flows
      if flow2 contains flow1
        increment(flow1_count)
      end if
    end for
  end for
  sub_graph ← flow1
end procedure

```

E. Bug Localization

The next and final step of the algorithm is to identify bug threads from the object template. Threshold values, frequency threshold values and frequent subgraphs are obtained from the previous steps. The bug thread can be identified by tracing threads in every frequent subgraph. If a thread in the frequent subgraph has negative threshold value with positive frequency threshold value, then it is added to the bug thread list. To filter out the bug threads exactly, the repeated bug threads have to be identified from the bug thread list. The algorithm does it by detecting the repeated bug threads with same threshold value in the bug thread list. Finally all the detected bug threads are saved as the separate object template which represents the bug thread and its threshold value. Bug localization procedure is given below. The output of bug localization is shown in Figure 6.

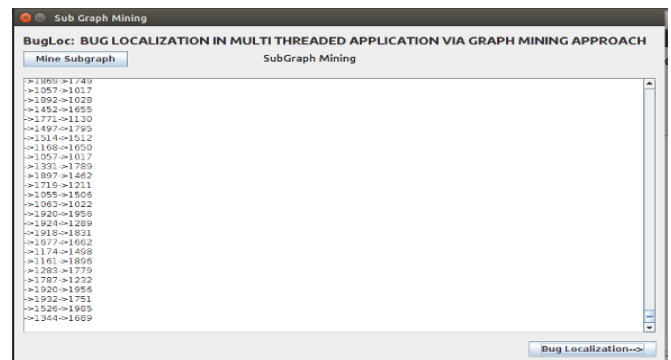


Figure 5: Result of Frequent Subgraph identification

D. Frequent Subgraph Identification

Frequent Subgraph identification is the main process which adopts the techniques of graph mining related to gSpan

```

procedure bugLocalization( ):

```

```

  get all_flows
  for flow in flows

```

```

for node in flow
  if node with frequency_threshold
    end_node_of_flow ← node
    if node_frequency < frequency_threshold
      bug_node ← node
    end if
  end if
end for
for threshold in thresholds
  for node in flow
    if threshold < 0
      if node_frequency = frequency_thread
        bug_node.Add( node)
      end if
    end if
  end for
end for
end procedure

```

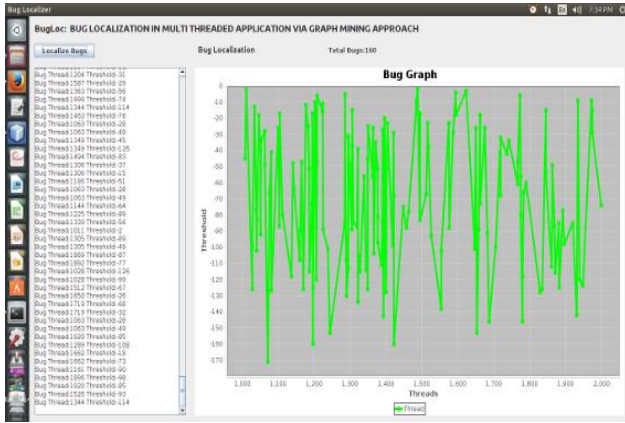


Figure 6: Result of Bug Localization

IV. EXPERIMENTAL EVALUATION

The proposed BugLoc algorithm is implemented with Java JDK 1.7 and tested with the sample graph datasets. The algorithm has been executed under the system with the configuration of Pentium D processor with 4GB RAM memory. The sample datasets of various counts 500, 1000, 2000 and 3000 are given as input to our BugLoc algorithm. For 500 count of sample dataset, it identifies 135 threads and 127 thread flows. From this, 12 bug threads are detected with the threshold value -93 in 15 seconds. When 1000 count of sample dataset is given as input, 446 threads and 338 thread flows are identified with 75 bug threads of threshold value -133 in 27 seconds. For count of 2000 dataset, the algorithm detects 92 bug threads in 41 seconds with threshold value of -85 from 865 threads and 533 thread flows. The algorithm is also tested with 3000 count of dataset.

Table 2 shows the input and output of the algorithm. Figure 7 shows relationship between dataset count and execution time. Figure 8 shows comparison between Number of threads and Number of thread flows. Figure 9 shows comparison between number of thread flows and number of bugs.

Table 2: Summary of input and output of the proposed algorithm

Dataset count	No. of threads	No. of thread flows	Threshold values	Bug threads	Time in seconds
500	135	127	-93	12	15
1000	446	338	-133	75	27
2000	865	533	-85	92	41
3000	1057	692	-146	83	74

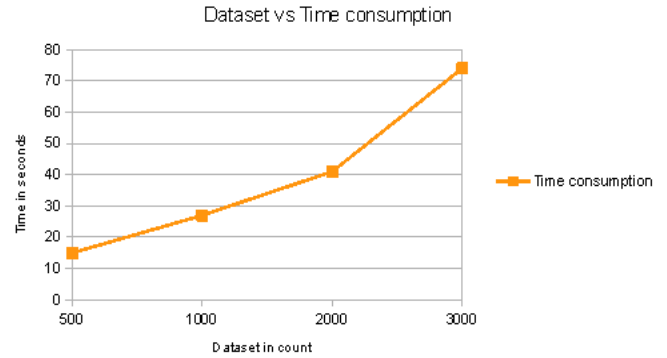


Figure 7: Relationship between dataset and time consumption

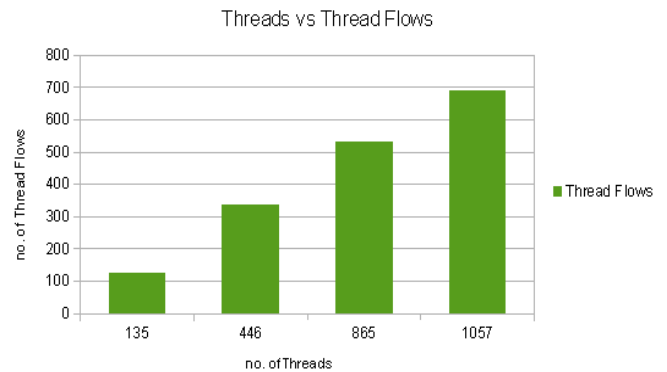


Figure 8: Comparison between number of threads and number of thread flows

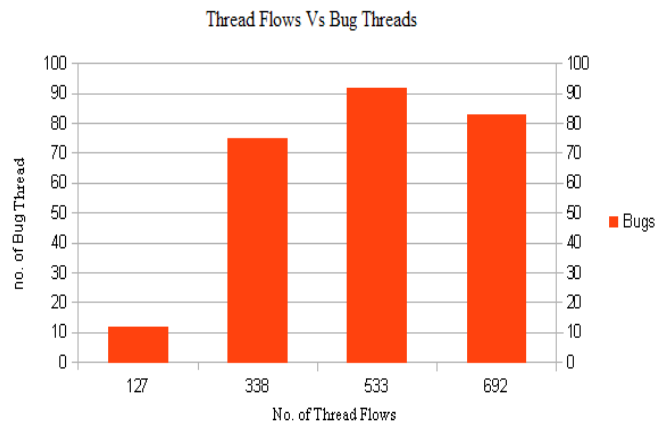


Figure 9: Comparison between number of thread flows and number of bug Threads

V. CONCLUSION

The proposed algorithm BugLoc has the tendency to localize bugs in multithreaded applications. From the experimental results it is verified that the new approach in detecting bugs in multithreaded application is working properly. This algorithm is based on graph mining techniques. With the detection of bug thread from the frequent subgraphs, we can clearly identify thread flow which causes the error. So, our proposed algorithm is used not only to detect bug thread but also used to identify the bug thread's route cause. Presently the algorithm computes the bug threads from given graph dataset. In future, this algorithm can be implemented in real time applications. The algorithm could be optimized to adopt grow and store method to hold thread as the node. This algorithm can also be extended to implement into a software application to generate the bug reports and send them automatically to the software developers.

References

- [1] Elseidy, Mohammed, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. "Grami: Frequent subgraph and pattern mining in a single large graph." *Proceedings of the VLDB Endowment* 7, no. 7 (2014): 517-528.
- [2] Huan, Jun, Wei Wang, and Jan Prins. "Efficient mining of frequent subgraphs in the presence of isomorphism." In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pp. 549-552. IEEE, 2003.
- [3] Schaeffer, Satu Elisa. "Graph clustering." *Computer Science Review* 1, no. 1 (2007): 27-64.
- [4] Lin, Chuan, Young-rae Cho, Woo-chang Hwang, Pengjun Pei, and Aidong Zhang. "Clustering methods in protein-protein interaction network." *Knowledge Discovery in Bioinformatics: techniques, methods and application* (2007): 1-35.
- [5] Kim, Jinha, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. "Taming subgraph isomorphism for RDF query processing." *Proceedings of the VLDB Endowment* 8, no. 11 (2015): 1238-1249.
- [6] Bringmann, Björn, and Siegfried Nijssen. "What is frequent in a single graph?." In *Advances in Knowledge Discovery and Data Mining*, pp. 858-863. Springer Berlin Heidelberg, 2008.
- [7] Lakshmi, K. "Frequent Subgraph Mining Algorithms--A Survey And Framework For Classification." (2012).
- [8] Cleve, Jürgen, Uwe Lämmel, and Stefan Wissuwa. "Data Mining on Transaction Data." *Global Markets in Dynamic Environments*, Lisboa (2005).
- [9] Silvestri, Claudio, and Salvatore Orlando. "Approximate mining of frequent patterns on streams." *Intelligent Data Analysis* 11, no. 1 (2007): 49-73.
- [10] Cheng, Jiefeng, Jeffrey Xu Yu, and Bolin Ding. "Multi Reachability Query Processing." (2008).
- [11] Woeginger, Gerhard J. "Exact algorithms for NP-hard problems: A survey." In *Combinatorial Optimization—Eureka, You Shrink!*, pp. 185-207. Springer Berlin Heidelberg, 2003.
- [12] Dor, Dorit, and Michael Tarsi. "A simple algorithm to construct a consistent extension of a partially oriented graph." *Technical Report R-185, Cognitive Systems Laboratory, UCLA* (1992).
- [13] Liu, Zhe. "Algorithms for Constraint Satisfaction Problems (CSPs)." PhD diss., University of Waterloo, 1998.
- [14] Wang, Qingguo, Mian Pan, Yi Shang, and Dmitry Korkin. "A Fast Heuristic Search Algorithm for Finding the Longest Common Subsequence of Multiple Strings." In *AAAI*. 2010.
- [15] Qian, Qi, Rong Jin, Jinfeng Yi, Lijun Zhang, and Shenghuo Zhu. "Efficient distance metric learning by adaptive sampling and mini-batch stochastic gradient descent (SGD)." *Machine Learning* 99, no. 3 (2015): 353-372.
- [16] Thomas, David B., Wayne Luk, Philip HW Leong, and John D. Villaseñor. "Gaussian random number generators." *ACM Computing Surveys (CSUR)* 39, no. 4 (2007): 11.
- [17] Fiedler, Mathias, and Christian Borgelt. "Support Computation for Mining Frequent Subgraphs in a Single Graph." In *MLG*. 2007.
- [18] Ugander, Johan, Lars Backstrom, and Jon Kleinberg. "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections." In *Proceedings of the 22nd international conference on World Wide Web*, pp. 1307-1318. International World Wide Web Conferences Steering Committee, 2013.
- [19] Chekuri, Chandra, and Nitish Korula. "Pruning 2-connected graphs." *Algorithmica* 62, no. 1-2 (2012): 436-463.
- [20] Cook, Diane J., and Lawrence B. Holder. "Substructure discovery using minimum description length and background knowledge." *Journal of Artificial Intelligence Research* (1994): 231-255.
- [21] He, Huahai, and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 405-418. ACM, 2008.
- [22] Lee, Jinsoo, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. "An in-depth comparison of subgraph isomorphism algorithms in graph databases." In *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 133-144. VLDB Endowment, 2012.
- [23] Cortadella, Jordi, and Gabriel Valiente. "A Relational View of Subgraph Isomorphism." In *RelMiCS*, pp. 45-54. 2000.
- [24] Robardet, Céline. "Constraint-based pattern mining in dynamic graphs." In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pp. 950-955. IEEE, 2009.
- [25] Yan, Xifeng, Hong Cheng, Jiawei Han, and Philip S. Yu. "Mining significant graph patterns by leap search." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 433-444. ACM, 2008.

- [26] Chen, Chen, Xifeng Yan, Feida Zhu, and Jiawei Han. "gapprox: Mining frequent approximate patterns from a massive network." In Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on, pp. 445-450. IEEE, 2007.
- [27] Lakshmi, K., and T. Meyyappan. "A comparative study of frequent subgraph mining algorithms." International Journal of Information Technology Convergence and Services 2, no. 2 (2012): 23.
- [28] Zou, Zhaonian, Jianzhong Li, Hong Gao, and Shuo Zhang. "Frequent subgraph pattern mining on uncertain graph data." In Proceedings of the 18th ACM conference on Information and knowledge management, pp. 583-592. ACM, 2009.
- [29] Adhiselvam, A., E. Kirubakaran, and R. Sukumar. "An Enhanced Approach for Software Bug Localization using Map Reduce Technique based Apriori (MRTBA) Algorithm." Indian Journal of Science and Technology 8, no. 35, 2015.
- [30] Adhiselvam, A., E. Kirubakaran, and R. Sukumar. "A Study On Frequent Subgraph Mining Algorithm and Techniques for Software Bug Localization." International Quarterly Journal in Scientific Transactions in Environment and Technovation 8, no. 4, 2015.