A Memory Efficient Algorithm for Combinatorial-based Test Suite Prioritization

Michael Burton
Computer Science, Utah State University,
Logan, UT 84322, USA
michael.burton@aggiemail.usu.edu

Tara Noble
Cognitive, Linguistic & Psychological Sciences, Brown University,
Providence, RI 02912, USA

Chelynn Day
Computer Science, Utah State University,
Logan, UT 84322, USA

Quentin Mayo
Computer Science, University of North Texas,
Denton, TX 76203, USA

March 12, 2014

Abstract

In testing web applications, details of user visits may be recorded in a web log and converted to test cases. This is called user-session-based testing and studies have shown that such tests may be effective at revealing faults. However, for popular web applications with a larger user base, many user-sessions may build up and test suite management techniques are needed. In this paper, we focus on the problem of test suite prioritization. That is, given a large test suite, reorder the test cases according to a criterion that is hypothesized

JCMCC **90** (2014), pp. 123-137

to increase the rate of fault detection. Previous work shows that 2-way combinatorial-based prioritization is an effective prioritization criterion. We develop a greedy algorithm where we consider memory usage and the time that it takes to prioritize test suites. We represent software tests in a graph by storing unique parameters as vertices and n-way sets as edges or series of edges. Our experiments demonstrate the efficiency of this approach.

1 Introduction

Test suite prioritization is useful for regression testing when an existing piece of software or web application accumulates a large number of use cases. As changes are made to the system, new tests may be added and increase the overall size and time to execute a test suite. Running all of these tests may take weeks in some cases [4]. One approach to managing such large test suites is to prioritize the suite so that the "best" tests are run first to detect faults as quickly as possible. Previous work has shown that combinatorial-based criteria have provided a good rate of fault detection in several studies [2]. In this work, we focus on an algorithm to provide such prioritization.

The usage of t-way combinatorial testing may vary given the specifics of the software or the requirements of the test, as discussed below, but in general it describes any set of n parameters with their associated values. One example of a single parameter and value (referred to throughout this paper as a t-tuple) in a web application would be the name of a text box and the value entered; a 2-way pair might include this tuple and the name and value of a radio button checked on the next URL. Since existing research has determined that many faults arise due to interaction between parameters, [3] one way to prioritize a test suite is to first include test cases that cover a significant number of n-way sets, or eliminate those tests which do not include any additional sets.

Previous research has used greedy algorithms for combinatorial-based testing with an existing test suite [2]. In a large test suite, implementing these algorithms is non-trivial - particularly for n-way sets of size 3 or greater. Because the number of sets grows exponentially with the number of t-tuples represented in the test suite, and existing test suites may be very large, the memory footprint and runtime of any program to implement these algorithms can present a problematic bottleneck. One intuitive implementation would be to store all n-way sets as objects in hash tables, in order to associate them with the covering test cases. When testing this implementation with a 1,257 KB test suite file with 96 test cases using a

computer with a 2.5GHz dual core processor and 8GB of RAM we found it could quickly prioritize by 2-way pairs. However, this same test also resulted in the computer running out of memory when trying to prioritize by 3-way sets. While some optimizations help alleviate this problem, none provide the efficiency necessary to prioritize larger test suites without the help of large amounts of extra RAM. Instead, the information must be stored in a fundamentally different way, without redundancy. A sacrifice in runtime might be unavoidable to make such changes, but is justified by reduction in memory use. Storing all the information from the test suite in a graph may allow efficient traversal within tight space constraints. The graph represents n-way sets as connections between parameters, and associates these connections with the test cases that cover them. Although this implementation does not have the benefit of constant time access and there is a time and space demand to initially store the graph, it ultimately provides greater utility by allowing the user to run much larger test suites.

2 Previous Research

S. Manchester (2012) provides an overview of the importance and challenges of n-way combinatorial testing and regression testing with user sessions in general. For an application seeded with faults, he found that both 2-way and 3-way prioritization provide fault detection rates within 1% of each other, and reducing by 2-way pair coverage detects up to 90% of the faults present in the entire test suite [3]. Because 3-way prioritization generally yields only slightly better fault coverage, and the time and memory requirements are much higher than for 2-way, combinatorial testing software will often only be used to generate 2-way coverage. Therefore, such software might justifiably be designed primarily for 2-way coverage use. By studying known causes of medical device failures, Wallace and Kuhn[6] found that most failures were caused by 2 or fewer parameters interacting. By testing all pairs of parameters, 98% of problems could have been detected. In another study however, Kuhn et al. [3] found that for a given web application, 2-way coverage revealed only 76% of all faults, while the remaining faults could have been detected by covering larger sets of parameters. Thus, the appropriate set size to cover will depend on the software and its internal dependencies. For this reason, we wanted to maintain as much scalability as possible within our algorithms to allow for coverage of larger sets.

Manchester also provides an overview of the open-source software Combinatoria based Prioritization for User-session-based Testing (CPUT), which he uses to implement the algorithms for 2- and 3-way coverage. The CPUT software allows the user to take test cases, converted to XML format from an

Apache server log, and prioritize them using a number of different criteria. It includes functionality to prioritize by 2- and 3- way sets, length, and a number of frequency criteria. Many of the algorithms it implements take only milliseconds to run on our 1,257 KB test suite. To run the 3-way prioritization on the same suite required a computer with 16 GB RAM and 121.9 seconds.

To respond to this need for efficiency, we chose to fundamentally change the underlying structure of the test suite data. Our goal then became two fold: (1) use graph theory and any optimizations possible to improve memory use, both theoretically and experimentally, and (2) maintain enough flexibility to quickly implement variations on the existing algorithms. Ultimately we added functionality to prioritize using sequential, intra-window, and consecutive 2-way set coverage as well as sequential 3-way set coverage.

3 Graph Implementation

We store three kinds of information in our graph. Vertices represent the unique parameters in the test suite; in the case of a web application, a parameter is defined by a unique URL, name, and value. Edges represent all pairs of parameters present in the test suite. Finally, edges are associated with the specific test cases which cover them. The test cases are represented as objects in the program to allow for sorting, scoring, and display, but they are not associated with any tuples or pairs, and so increase memory use by only a small linear amount. On load, the graph is built as the XML file is read, and remains unchanged throughout use of the program.

Because none of the prioritization algorithms implemented require information about parameter names or values, t-tuples are stored only as unique integers. On load, a separate encoding class maps each string representation of a t-tuple to an integer. This offers additional memory benefits and reduces runtime during parameter comparison. This encoding class is used by 3-way algorithms to compare URLs, but otherwise all parameters can be handled as integers. This decision allowed us to build the entire graph as a single array list, where each index corresponds to a vertex. An additional optimization described below required that all parameters be encoded as integers in the order in which they appear in the test suite.

Each index in the graph holds a list of edge objects, and each edge object stores its connected vertex as an integer. That edge represents a connection between the two vertices which is present in the test suite. In addition to the connected vertex, the edge class contains three lists of test cases. The lists are used with corresponding prioritizations and are ignored if not applicable to the current prioritization criteria:

1. **Forward test cases:** all test cases which cover that pair of parameters in sequential order (the first parameter appears in the test case before the second).
2. **Backward test cases:** all test cases which contain the current pair in the opposite order. Backward test cases are added after the entire forward graph is built; every forward test case for which the integer value of the first parameter is greater than the integer value of the second parameter are also stored as a backward test case. This adds some redundancy to the system, but provides runtime benefits which will be explained later. (A test case will not be added to the backward test cases list if it is already in that edge's forward test cases list.)
3. **Consecutive test cases:** all test cases which contain the pair in sequential order and on consecutive URLs. In a test case, if a user enters a value on one page and another value on the page that immediately follows, that test case covers those two parameter values as a consecutive pair and is added to that edge's list of consecutive test cases. This list is a subset of the list of forward test cases.

Each test case is stored in a single object, and lists point directly to these objects. As mentioned above, there is some redundancy built into these test case lists to allow for flexibility in scoring, but once the graph is built it remains a constant size.

We originally treated pairs as identical, regardless of the order they occurred in. However, we quickly added in support for additional types of n-way pairs, including: sequential, consecutive, and intra-window. Sequential pairs are considered unique if they contain the same parameters in a different order. Consecutive pairs are pairs of parameters between consecutive URLs. Intra-window pairs are those between parameters on the same URL. Because the edges created from intra-window pairs have no intersection with any other edges in the graph, we store them in a separate graph with the same internal structure. This improves efficiency when we prioritize by these pairs, as we do not have to iterate through any non-intra-window edges. Using the graph structure also allows existing prioritization algorithms such as the Ant Colony Optimization described by Singh[5], that

are based on graph theory, to be easily implemented since the prioritization and data storage would require little to no change.

Algorithm 1: Create Graph

```
1  begin
2      textbf graph be a list to store all the test suite's unique parameters
3      textbf sessionList be a list of all the sessions in the test suite file
4      sessionNumber = −1
5      foreach Session session in sessionList do
6          sessionNumber + +
7          textbf currentCase be a test case object storing session's data
8          textbf urlList be a list of all the URLs in session
9          foreach URL url in urlList do
10             foreach Parameter param in url do
11                 vertex = encode(url.name, param.name, param.value)
12                 if vertex is not in graph then
13                     add vertex to graph
                   end
               end
           end
14         foreach URL url_1 in the urlList do
15             foreach Parameter par_1 in url_1 do
16                 foreach URL url_2 in urlList that is after url_1 do
17                     if url_1 and url_2 refer to the same url then
18                         skip to next url_2
                       end
19                     foreach Parameter par_2 in url_2 do
20                         if par_1 and par_2 make a new edge then
21                             create a new edge and add it to the graph
                           end
22                         if par_1 & par_2 make an existing edge then
23                             add currentCase to the edge's test case list
                           end
                       end
                   end
               end
           end
       end
   end
```

4 Prioritization Algorithm

The prioritization algorithm remains approximately the same for each n-way pair option, with slight modifications. In essence, the list of test cases of length n is sorted in place by iterating through the graph n times, each time finding the next test case with the greatest number of unique, uncovered pairs. More specifically, each test case contains a score for the given prioritization selection; these scores are initially set to zero. We iterate through each vertex in the graph, and each edge associated with that vertex. We then iterate through the test cases covering that edge. If any of those test cases has already been sorted, then the edge is considered covered and marked as such to increase speed on the next iteration. If the edge is still uncovered, then the score of each test case which covers it is incremented. During this process, we maintain a pointer to the test case with the highest score, and when the graph has been completely traversed, swap this test case with the first unsorted one in the list. The scores of the unsorted tests are all reset to zero, and the next iteration begins. During sequential and consecutive prioritization, only the corresponding lists of test cases are considered in the algorithm. During non-sequential prioritization, both the backward and forward test case lists are considered, but only on edges which go from a smaller vertex to a larger. In this way, we are able to guarantee, without the need of checking for an edge in the opposite direction, that no edges are counted twice.

Three-way sets pose an additional problem, as they are represented by two edges as opposed to one. The three-way algorithm takes an extra step by following each edge to its connected vertex, iterating through the edges from this vertex, and taking the intersection of each edges test list with the original. This generates a list of common test cases, whose scores we then increment. This increases the runtime by a linear factor of the number of test cases, and the only change in memory usage is storing the list of common test cases. The same technique could be used to prioritize by n-way sets of any size, with an additional increase in run time.

Algorithm 2: Pairwise Prioritization

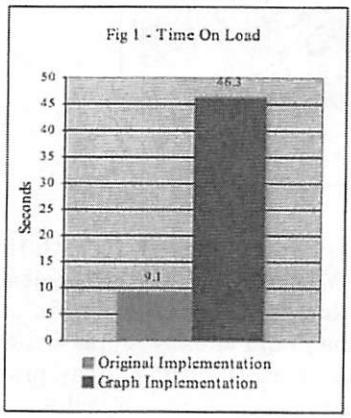
```
1  begin
2      foreach Edge edge in Graph graph do
3          edge.Covered = false
       end
4      for index = 0 to testList.Size −1 do
5          foreach unordered Test Case testCase in testList do
6              testCase.Score= 0
           end
7          highestScore = −1
8          indexOfHighest = 1
9          foreach Vertex v1 in Graph g do
10             foreach Edge e in v1 do
11                 if e.isCovered = true then
12                     skip to the next e
                   end
13                 v2 = e.getConnectedVertex()
14                 if v1 > v2 then
15                     skip to the next v1
                   end
16                 textbfallCases be a list of all the test cases in e
17                 foreach ordered Test Case tc_ordered in testList do
18                     foreach Test Case tc_allCases in e do
19                         if tc_ordered & tc_allCases are equal then
20                             e.isCovered = true
21                             skip to the next e
                           end
                       end
22                     foreach Test Case tc in allCases do
23                         if tc.currentIndex < testList.Size &
                           tc.currentIndex >= index then
24                             tc.Score++
                           end
25                         if tc.Score > highestScore then
26                             highestScore = tc.Score
27                             indexOfHighest = tc.getCurrentIndex()
                           end
                       end
                   end
               end
           end
28         swap the test cases at indexOfHighest and index
       end
   end
```

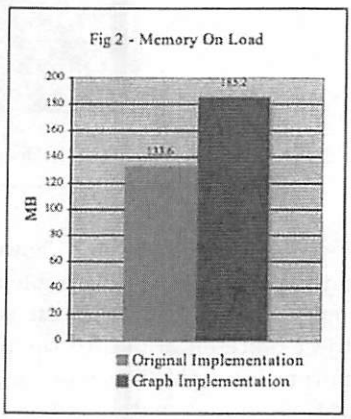
5 Results

In the worst case, every parameter is linked to every other parameter in every test. In this case, the program has a memory footprint of $O(p^2t)$, where p is the total number of parameters in the entire test suite, and t is the number of test cases. There is a relatively low constant on this equation, as very little information is stored in objects and almost none of it is stored in multiple places. This usage remains nearly constant throughout use of the program.

There is a small sacrifice in our theoretical runtime with this implementation. Because the graph is of size p^2t, and we iterate through it t times, the worst case runtime is at least p^2t^2. However, since we must also iterate through the each test case list to determine if the edge is covered by a previous test case, the worst case is bounded by an additional factor of t. With the added optimization of marking edges as covered, however, this worst case is simplified enormously. Because the graph can only grow to such a size if every test case contains every parameter, then finding any test case will cover all the edges. This will be detected on the second iteration, and decrease the runtime by a factor of t. Another potential "worst case" for run time is that in which the parameters are evenly divided between test cases (i.e., each test case contains $\frac{p}{t}$ parameters with no overlap). In this case, the runtime will be $(p^2t^2 * \frac{p}{t})$, or p^3t for 2-way prioritization.

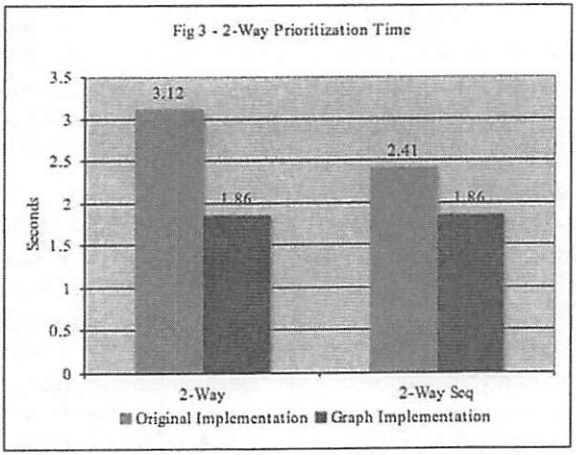
Because the runtime and memory analysis of the graph implementation depend on multiple factors and the constant is not immediately known, it is worthwhile to analyze the program experimentally, particularly in comparison to a previous implementation.

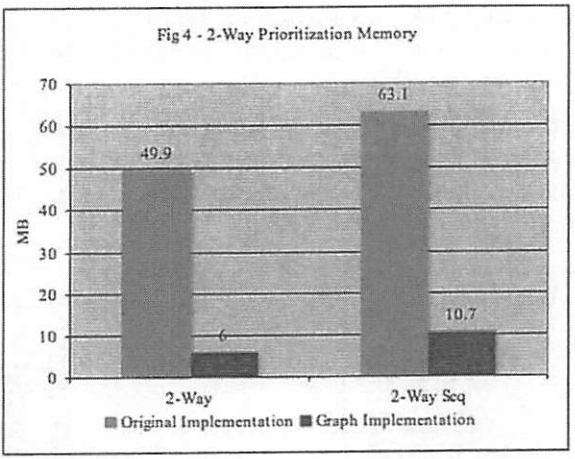




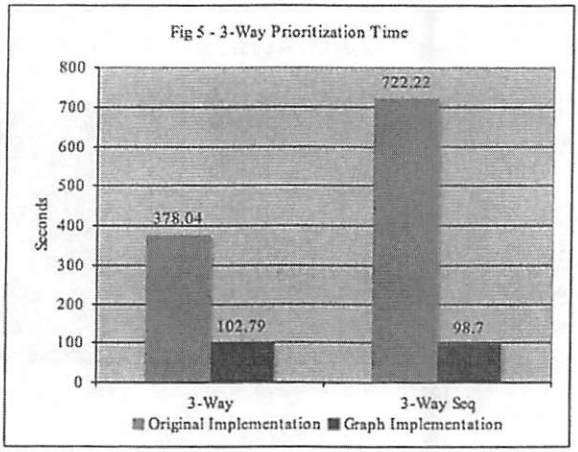
Figures 1 and 2 display the time and memory usage of the program to

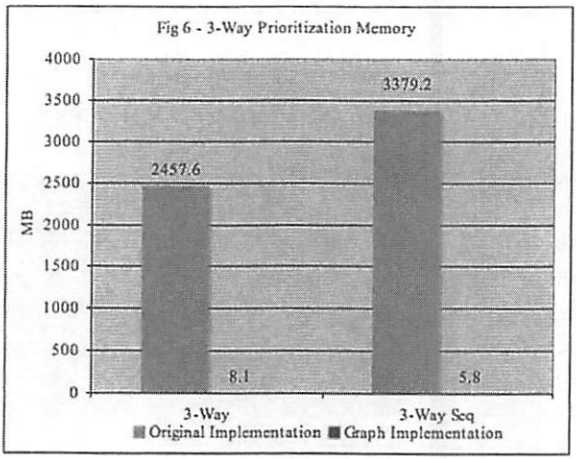
load in our sample test suite of 96 tests. This is where we see the biggest drawback of our method - load time increases by roughly 5 times, and extra memory is used.





In figures 3 through 6, however, we easily see the benefit in both time and memory use to this implementation. Where the original implementation must create and store an exponentially increasing number of additional object representations for the n-way sets, the graph already represents this information. This saves massive amounts of memory during 3-way prioritization and ultimately reduces runtime, as we see in figures 5 and 6.





The scalability of an algorithm is also an important factor in combinatorial testing since test suites are continually growing. To determine the algorithm's scalability we did tests based on input file size growth as well as internal data growth. Figures 7 and 8 show the memory usage and runtime of both the old and new implementations as the file size of the test suite increases. File size is not an ideal predictor of complexity, as it tells us very little about the distribution of parameters within the test suite. Nonetheless, with a larger file, there will generally be more sets, which increases memory use. The graphs show that due to the higher memory complexity of the previous implementation, the memory requirements increase very quickly, especially when compared to the new implementation.

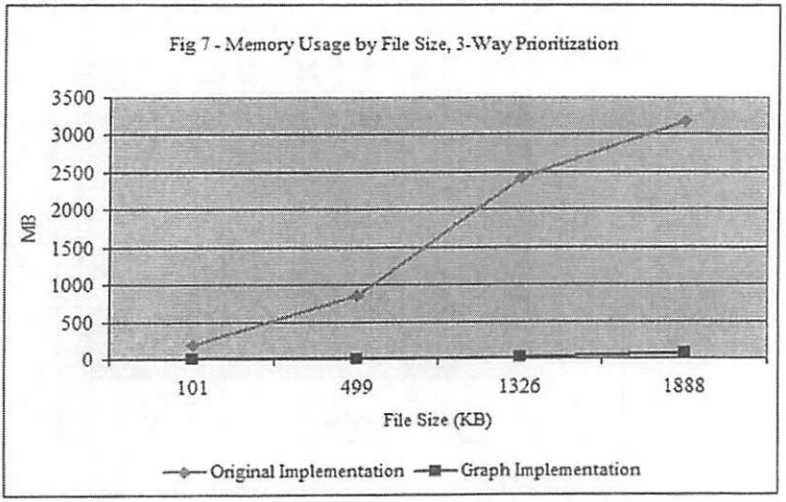


Fig 7 - Memory Usage by File Size, 3-Way Prioritization
3500
3000
2500
2000
1500
1000
500
0
MB
101
499
1326
1888
File Size (KB)
Original Implementation
Graph Implementation

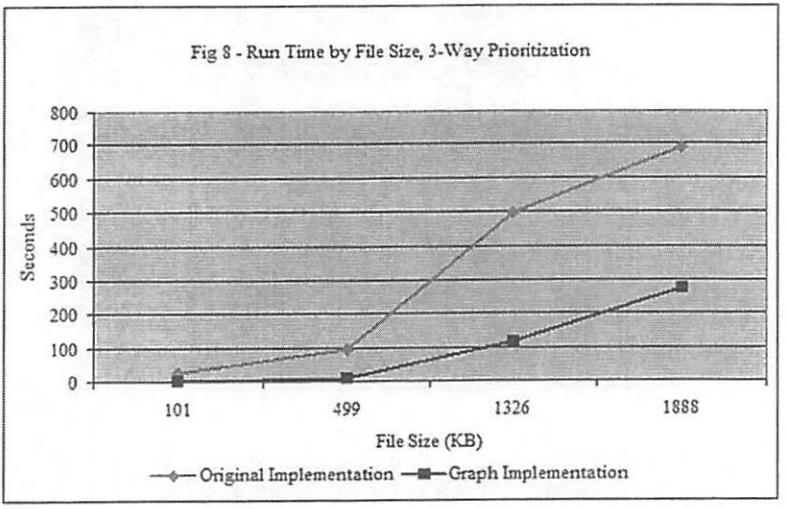
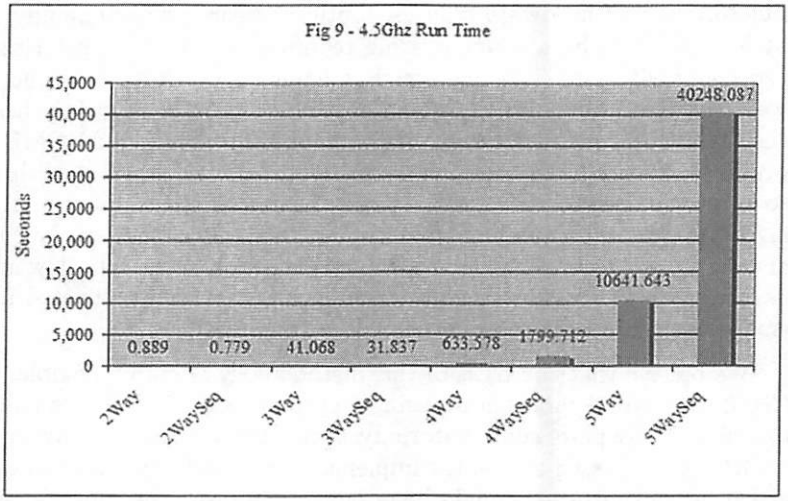
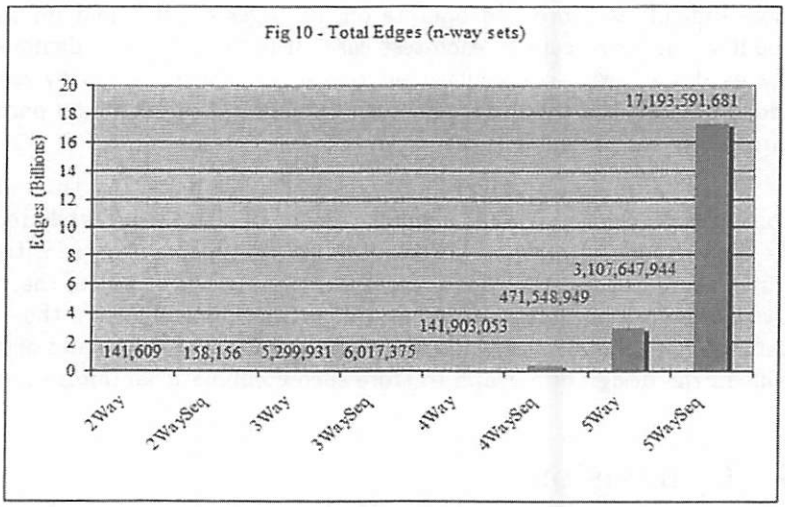


Fig 8 - Run Time by File Size, 3-Way Prioritization
800
700
600
500
400
300
200
100
0
Seconds
101
499
1326
1888
File Size (KB)
Original Implementation
Graph Implementation





For testing the scalability with respect to internal data growth we implemented the standard and sequential prioritization methods for 4-way and 5-way sets. The 4-way and 5-way set methods were only implemented in the graph-based algorithm due to the fact that the previous algorithm's inability to handle them. These tests were run on a system with a 4.5Ghz 8-core processor and 16GB RAM. Figures 9 and 10 show the run time growth and internal data growth. While the run time increases drastically with the 5-way sets, it does so proportionally to the internal data and with-

out software or hardware failures. Future research will hopefully help to at least reduce the amount of time required to run these prioritizations, especially with CPUT being aimed at having a quick turn-around to give web developers the time to correct any detected faults. During these tests the algorithm's memory usage had a maximum increase of 130MB, which occurred when running the 5-way methods. This shows that with respect to memory, despite the rapid growth of internal data and run time, the algorithm has the capability of handling test suites that are much larger and likely with even higher n-way set prioritizations. The low memory usage may also provide insights for applications within restricted memory environments such as those discussed by Bhadra[1].

We believe that the basis of this methodology is highly flexible. As the CPUT software demonstrates, using a graph to store information allows us to define 2-way pairs combinatorially, sequentially, consecutively, or within a URL. As discussed above, the implementation best supports 2-way coverage; other optimizations might be possible to improve the runtime on 3-way sets or larger. When n is greater than 2, n-way consecutive sets become more difficult to store and operate on, as edges do not hold information about when they occur in each test case. If further research demonstrates the need for larger consecutive set coverage, this functionality could be added without the need of restructuring the graph by keeping a parameter chronology within the test cases.

We use a separate graph to store intra-window pairs, but this is an optional design choice. It adds a small amount of complexity while reducing the size of each graph and therefore traversal time during prioritization. Further research must be done into the relative effectiveness of the various n-way set types in prioritization, as well as their distribution in the average test suite; this knowledge would allow greater improvements and optimizations in the design of a graph to store such combinatorial information.

6 Conclusion

This work presents an algorithm for test suite prioritization by combinatorial-based coverage. A major challenge in test suite prioritization by combinatorial-based coverage occurs when test suites grow large and require significant processing in terms of memory and time. Our graph-based approach provides a means to prioritize such test suites. Let us first acknowledge that storing combinatorial testing information in a graph requires some sacrifices. There are additional time and memory demands made to build such a graph as well as. These sacrifices were made in order to avoid two major

memory pitfalls associated with time efficiency, redundancy and exponential growth. We removed the redundancy of associating test cases with pairs, and chose to store all information in lists rather than hash tables. In both cases, this meant sacrificing constant time access, but we found that the graph had very little use for these particular accesses and provided for further memory reduction. Experimental results suggest that this change offered worthwhile benefits to memory use. On an experimental dataset of only 1,257 KB, our results show that the memory reduction will eventually offer a runtime advantage as well. Since this software is most useful when applied to large test suites, which benefit greatly from prioritization, memory efficiency is crucial.

References

[1] S. Bhadra, A. Conrad, C. Hurkes, B. Kirklin, G.M. Kapfhammer, *An experimental study of methods for executing test suites in memory constrained environments.* Automation of Software Test (2009),27-35.

[2] R.C. Bryce, S. Sampath, A.M. Memon, *Developing a Single Model and Test Prioritization Strategies for Event-Driven Software.* IEEE Transactions on Software Engineering **37**(2011), 48-64.

[3] D.R. Kuhn, D.R. Wallace, A.M. Gallo Jr., *Software fault interactions and implications for software testing.* IEEE Transactions on Software Engineering **30**(2004), 418-421

[4] G. Rothermel, R.H. Untch, Chengyun Chu, M.J. Harrold, *Prioritizing test cases for regression testing.* IEEE Transactions on Software Engineering **27**(2001), 929-948.

[5] Yogesh Singh, Arvinder Kaur, and Bharti Suri, *Test case prioritization using ant colony optimization.* SIGSOFT Softw. Eng. Notes **35**(2010), 1-7.

[6] D.R. Wallace, D.R. Kuhn, *Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data.* Int. J. Rel. Qual. Saf. Eng. **8**(2001), 351.