Prioritizing Test Suites by Finding Hamiltonian Paths: Preliminary Studies and Initial Results

Suvarshi Bhadra and Gregory M Kapfhammer
Allegheny College
Department of Computer Science
suvarshi.bhadra@gmail.com, gkapfham@allegheny.edu

Introduction. This paper describes a technique for prioritizing a test suite by finding the least weight Hamiltonian path in a complete graph that represents relative testing costs. Our technique is especially useful when testing confronts constraints such as quotas in a Web service [3], memory overhead [4], or test execution time [5]. During the testing of modern mobile computing devices (e.g., handsets running Google Android), it is often challenging to properly handle memory constraints. Thus, even though we anticipate that our approach is valuable in a wide variety of limited resource environments, this paper focuses on prioritizing test suites for memory constrained execution. In summary, the important contributions of this paper include:

1. The formulation of a graph-theoretic approach to test prioritization that leverages traveling salesperson problem (TSP) solvers to find Hamiltonian paths.
2. A detailed empirical evaluation that uses synthetic test suites to determine the efficiency and effectiveness of the TSP solvers that re-order the test cases.

Motivating Example. In the context of resource constrained mobile devices, frequent reads and writes to memory may increase testing time by as much as 600% when a Java application executes on a virtual machine with a small heap [4]. To this end, we prioritize a test suite in order to minimize the loading and unloading of bytes associated with both the native code of methods and objects allocated to the heap (tools such as Jikes RVM, HProf, and DJProf enable the collection of this information). Furthermore, our technique ensures that the ordering of a test suite will never consume more than a specified amount of memory. For instance, Figure 1 describes a test suite $T = \langle T_1, T_2, T_3, T_4, T_5 \rangle$ that tests a program consisting of six equi-sized methods. In this example, each method consumes 30 units of memory and a dot in the matrix indicates that a test case calls a method during execution.

Assuming that the size of a test case is the sum of the sizes of the methods it calls and that the memory constraint is the size of the largest test case (i.e., T_1, T_2, or T_4), then it is clear that different orderings require a dissimilar number of loads and unloads. If the test executor always loads and unloads all of a method's code and data, then the initial ordering $T = \langle T_1, T_2, T_3, T_4, T_5 \rangle$ necessitates the transfer of 750 units to and from memory. This prioritization incurs a high cost because running T_2 after T_1 causes methods m_1,m_2, and m_3 to be loaded and then immediately unloaded in order to make room for m_4,m_5, and m_6. In contrast, the ordering $T' = \langle T_2, T_4, T_1, T_3, T_5 \rangle$ only calls for the loading and unloading of 180 units because T' maximizes method reuse between neighboring test cases. The remainder of this paper describes techniques that use test cost information to create an improved T' from test suite T.

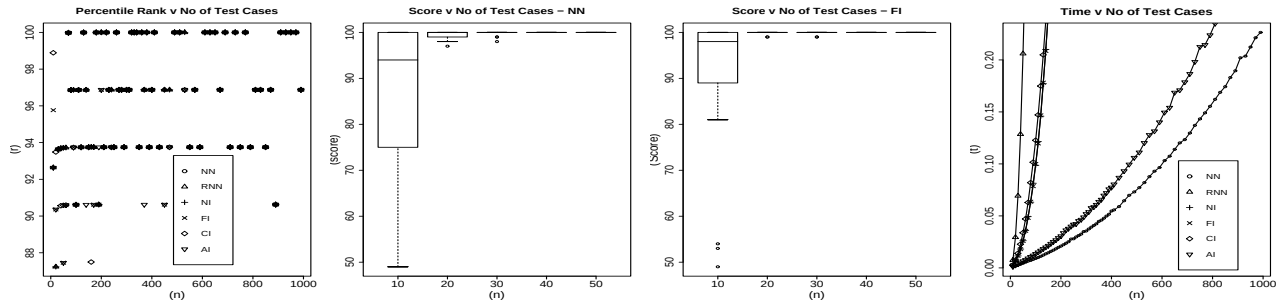
	m_1 30	m_2 30	m_3 30	m_4 30	m_5 30	m_6 30	Test Size
T_1	•	•	•				90
T_2				•	•	•	90
T_3	•	•	•				90
T_4				•	•	•	90
T_5	•	•					60

Figure 1: Representation of a Test Suite.

Technique. Given a test suite $T = \langle T_1, \ldots, T_i, T_j, \ldots, T_n \rangle$, we formulate a corresponding asymmetric and complete graph $K_n = \langle V, E \rangle$ such that $|V| = n$ and $|E| = n \times (n-1)/2$. For all of the adjacent test cases $T_i, T_j \in T$, we construct K_n so that $v_i, v_j \in V$ and $e_{ij}, e_{ji} \in E$ when e_{ij} is an edge from v_i to v_j with cost c_{ij}. A Hamiltonian path P through K_n is a path that visits each vertex $v_i \in V$ exactly once and the cost of a Hamiltonian path, denoted C_P, is the sum of the costs for all of the edges in P. Viewing P as a tuple of edges enables us to construct a prioritized test suite T' such that for each $e_{kl} \in P$ we have $\langle T_k, T_l \rangle \in T'$. If c_{ij} is the cost, in bytes loaded and unloaded, associated with executing test T_j after T_i, then finding a low cost Hamiltonian path P will yield a test prioritization T' that reduces memory transfers and subsequently decreases test execution time.

In order to ensure that T' does reduce overall testing time, it is important to devise a cost value c_{ij} that properly expresses the costs for running T_j after T_i. In this paper, we exclusively focus on the overlap in the method calls for two adjacent test cases. If $call(T_i)$ and $call(T_j)$ respectively denote the sets of methods invoked by T_i and T_j during testing, then we calculate the edge cost with $c_{ij} = 2 \times size(call(T_j) \backslash call(T_i))$. In this equation, $size$ is a function that returns the bytes consumed by all of the methods in its input set. For instance, applying the calling information in Figure 1 to the test ordering in T yields a cost value $c_{12} = 2 \times size(\{m_4, m_5, m_6\} \setminus \{m_1, m_2, m_3\}) = 180$ (the other costs would be calculated in an analogous fashion).

Our formulation of c_{ij} anticipates that the test executor will unload enough methods to accommodate the new methods that are required by T_j. If we also accept the simplifying assumption that the size of methods to be unloaded is approximately equal to the size of those that must be loaded, then c_{ij} may be calculated by doubling



(a) (b) (c) (d)

Figure 2: Experimental Results from Using TSP Solvers to Prioritize Synthetic Test Suites.

$size(call(T_j) \setminus call(T_i))$. Finally, if we suppose that the resource constrained device always contains enough memory to fully execute any single test, then our cost equation ensures that the memory limit will not be exceeded. Yet, our cost formulation is simplistic because it focuses on the overlap in method calls for tests T_i and T_j, thus ignoring all of the test cases that ran before T_i and will execute after T_j.

When T actually runs, there is the possibility that all of the methods in the set $call(T_j) \setminus call(T_i)$ do not have to be loaded because they already exist in memory from a test case that ran prior to T_i. Furthermore, since methods are not always equi-sized, we are not guaranteed that the size of the unloaded methods will exactly equal those that are loaded. Finally, it is possible that the device's memory constraint may not be reached until the test executor runs a certain number of tests. Up to the point in time when the test cases consume all of the available memory, the real cost associated with running the tests only corresponds to loading native code bodies and allocating objects to the heap. In summary, the estimated cost of running T_j after T_i may be higher than the actual cost evident during test execution.

We use TSP solvers implemented in the R programming language [2] to identify a low cost P within K_n and thus create T'. After representing K_n as an $n \times n$ matrix, we use the nearest neighbor (NN), repetitive nearest neighbor (RNN), nearest insertion (NI), farthest insertion (FI), cheapest insertion (CI), and arbitrary insertion (AI) TSP solvers to form P [2]. Since these solvers use heuristics to create P, it is possible that the resulting path may not be the optimal one for K_n. Due to the fact that our calculation of c_{ij} is only a rough estimate for the actual cost of running $\langle T_i, T_j \rangle$, it is also possible that the length of P may not directly correspond to the real cost of executing T'.

Experiment Design. Since synthetic test suites enable the study of the trade-offs in the efficiency and effectiveness of testing techniques [1], we randomly generated test suites that contain up to $n = 1000$ tests. For each value of n, the generation procedure constructs 32 distinct tables like the one in Figure 1. Yet, the generator assigns random method sizes and it constructs $call(T_i)$ sets that vary in both their size and contents. We used each of the randomly generated test suites as an input to the six TSP solvers and recorded the value of C_P and the resulting prioritization T'. We also implemented a test suite executor that can run T' in order to calculate the exact number of loads and unloads and thereby determine the actual cost of a prioritization. Since there are $n!$ orderings of T, we compare each T' to the orderings in a sampling set S such that $|S| \leq n!$. If B is the set of prioritizations in S that T' is better than in terms of actual cost, then $score(T') = |B|/|S| \times 100 \in [0, 100]$ is the quality of T' with higher values representing better test prioritizations. Since the value of $n!$ may be prohibitively large, we populate S with n^2 randomly generated orderings.

Experimental Results. In the context of the random orderings in S, Figure 2(a) gives the percentile rank of C_P for the path P created by each TSP solver. Since small values for C_P are desirable, a percentile rank corresponds to the percentage of the S prioritizations for which the solver's value of C_P is lower. While the results in Figure 2(a) indicate that there is some variability in the value of C_P for different solvers, it is evident that the techniques find low cost Hamiltonian paths in the 80th percentile rank or better. Figures 2(b) and 2(c) furnish box plots where an individual box and whisker represents the variability of $score$ across the 32 distinct test tables. These graphs demonstrate that the value of $score$ improves as the size of the test suite increases (we omit graphs for additional techniques because they show the same empirical trend). Figures 2(b) and 2(c) do not contain results beyond $n = 50$ because of the high computational cost associated with running T' in the test executor and storing the full history of memory transfers. Finally, Figure 2(d) suggests that as n ranges from 10 to 1000, prioritization time varies from less than .05 seconds to more than 200 seconds. We also observe that the most effective techniques (e.g., RNN, CI, FI, NI) require more time than the other solvers. Given these promising preliminary results, we intend to refine our prioritizers and evaluate them with both synthetic and real world test suites.

References

[1] F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1), 2007.

[2] M. Hahsler and K. Hornik. TSP - infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2), 2007.

[3] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Proc. of ICSM*, 2008.

[4] G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in resource constrained execution environments. In *Proc. of ASE*, 2005.

[5] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proc. of ISSTA*, 2006.