# Research in Experimental Computer Science

Suvarshi Bhadra, Alexander Conrad, Adam Smith, and Gregory M. Kapfhammer

Department of Computer Science
Allegheny College, Pennsylvania, USA
http://www.cs.allegheny.edu/~gkapfham/

Allegheny College summer research Series (ACRoSS), July 2008
Featuring images from www.campusbicycle.com and www.pdclipart.org
What is Experimental Computer Science?

Solve Problems with Algorithms

Detailed Empirical Results

**Implement** and empirically **evaluate** the efficiency and effectiveness of **algorithms** that solve real-world **problems**
What is Experimental Computer Science?

After analyzing gigabytes of data, publish results and release software tools that are useful to academics and industrialists.

Computer software as community service.
Goal: Find a backpack that will support your commute to work or school
**Question**: Can you select items so that you maximize the benefit while ensuring that the cost does not exceed the capacity?

This problem is **NP-complete** (see Garey and Johnson) and yet it also has many practical applications in both **software** and **finance**.
**Introduction**

Resource-Constrained Search-Based Cost-Aware Conclusions

---

**0/1 Knapsack Problem**

Knapsack

Capacity = 12

- **Cost = 4**
  - Benefit = 6

- **Cost = 10**
  - Benefit = 9

- **Cost = 7**
  - Benefit = 4

- **Cost = 9**
  - Benefit = 8

**Question:** Can you select items so that you **maximize** the benefit while ensuring that the cost does not **exceed** the capacity?

- This problem is **NP-complete** (see Garey and Johnson) and yet it also has many practical applications in both **software** and **finance**.
“Software entities are more complex for their size than perhaps any other human construct”

- Frederick Brooks (Professor of Computer Science at the University of North Carolina – Chapel Hill)
It is **expensive** to run a test suite $T = \langle T_1, \ldots, T_n \rangle$. **Prioritization** searches through the $n! = n \times n - 1 \times \ldots \times 1$ orderings for those that **maximize** an objective function like **coverage** or **fault detection**.
Prioritizing When Memory is Constrained

Frequent Memory Rewrites

High Testing Costs

Frequent **reads** and **writes** to memory may **increase** execution time by as much as **600%** when a Java application executes on a virtual machine with a **small heap**.

**Solution:** maximize memory **reuse** between test cases
The Impact of Test Ordering

<table>
<thead>
<tr>
<th></th>
<th>(m_1)</th>
<th>(m_2)</th>
<th>(m_3)</th>
<th>(m_4)</th>
<th>(m_5)</th>
<th>(m_6)</th>
<th>Test Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>(T_2)</td>
<td></td>
<td></td>
<td></td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>90</td>
</tr>
<tr>
<td>(T_3)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>(T_4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>90</td>
</tr>
<tr>
<td>(T_5)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
</tr>
</tbody>
</table>

- \(T = \langle T_1, T_2, T_3, T_4, T_5 \rangle\) transfers 750 units to and from memory
- \(T' = \langle T_2, T_4, T_1, T_3, T_5 \rangle\) only loads and unloads 180 units
Challenges of Test Prioritization

Real-World Suites Have Hundreds of Tests

- \( n! \) possible solutions
- For 100 test cases, \( 9.33262154 \times 10^{157} \) possible solutions
- It takes 2.22 seconds to evaluate a solution permutation
- Finding the answer would take \( 6.52846694 \times 10^{149} \) years

Problem Formulation

- Intelligently comb the search space for an effective ordering
- Can efficient techniques identify good prioritizations?
- Will the prioritizers work properly in real-world software development environments?
Challenges of Test Prioritization

Real-World Suites Have Hundreds of Tests

- $n!$ possible solutions
- For 100 test cases, $9.33262154 \times 10^{157}$ possible solutions
- It takes 2.22 seconds to evaluate a solution permutation
- Finding the answer would take $6.52846694 \times 10^{149}$ years

Problem Formulation

- **Intelligently** comb the search space for an **effective** ordering
- Can **efficient** techniques identify **good** prioritizations?
- Will the prioritizers work properly in **real-world** software development environments?
Cheapest Way to Travel the World?

Cheapest way to see Seattle, Rio, Shanghai, Mumbai, and Sydney.
Cheapest Way to Travel the World?

$5! = 120$ possible solutions
**Goal:** Find the least weight Hamiltonian path through a complete graph.
Is it possible to \textit{efficiently} create test prioritizations by \textit{solving} the “find the cheapest way to travel the world” problem?

If yes, then are the orderings \textit{effective}?
How Good are TSP Solvers?

- Implemented by researchers at the University of Vienna
- Six key algorithmic techniques
- All algorithms produce 80+ percentile rankings for the path cost metric
- Most expensive algorithm takes 200s for $n = 1000$
**Experiment Design**

- **Project Metrics**:
  - Lines of Code: 5,274
  - Data Files: 15,180
  - Data Files Size: 4.5 GB

- **Key Implementations**:
  - Synthetic Test Suite Generator
  - Cost Matrix Constructor
  - Test Suite Executor
Empirical Results

Time v No of Test Cases

Score v No of Test Cases – NN

Efficient Prioritizers

High Percentile Rankings
What is a Genetic Algorithm?

A search-based approach to test suite prioritization

Parts of a Genetic Algorithm:

- **Data Structures:**
  - Chromosome
  - Individual
  - Population

- **Functions:**
  - Selection Operator
  - Crossover Operator
  - Mutation Operator
  - Termination Condition
  - Fitness Function
**What is Coverage Effectiveness?**

- Prioritize to **increase** the CE of a test suite $CE = \frac{Actual}{Ideal} \in [0, 1]$

**Unlabeled Formulae:**

- Cover $R(T_1)$
- Cover $\bigcup_{i=1}^{n-1} R(T_i)$
- Cover $R(T)$
- $T_n$ Done
- $T_{n-1}$ Done
- Area $\int_0^{t(n)} C(T, t)$
- Testing Time $(t)$

**Definitions:**

- $C(T, t)$: Covered Test Reqs
- $T_1$ Done
- $T_{n-1}$ Done

**Research in Experimental Computer Science**
Genetic Test Suite Prioritizer

**Motivation:** There are known instances where greedy techniques always yield sub-optimal orderings. Few experiments have studied the efficiency and effectiveness of search-based techniques.

**Goals:** Identify configurations of the genetic algorithm that produce desirable results. Outperform random search.

**Project Statistics:**
- 6,369 lines of code
- 6 mutation operators
- 7 crossover operators
- 3 selection operators
- 3 fitness transformation operators
**Experiment Design**

**Metrics:**
- Runtime of prioritization technique
- Coverage effectiveness of final test ordering

**Data Sets:**
- 9 real-world
- 54 synthetic

**Configurations:**
- 10,206 configurations
- 91,854 real-world experiments
- 551,124 synthetic experiments
Preliminary investigation indicates that the genetic algorithm can produce **better** results in **less time** than random search.
Finding the Overlap in Coverage

- $R_j \rightarrow T_i$ means that requirement $R_j$ is covered by test $T_i$
- $T = \langle T_2, T_3, T_6, T_9 \rangle$ covers all of the test requirements
- Test suite reduction discards the test cases that redundantly cover the test requirements
Incorporating the Costs of a Test Case

Hypothesis: Using the execution time of a test case can improve the reduced and prioritized test suites

Compare (i) greedy choices (cost, coverage, and ratio) and (ii) algorithms
Greedy Choices Impact Effectiveness

<table>
<thead>
<tr>
<th>Greedy-by</th>
<th>( T_r )</th>
<th>( time(T_r) )</th>
<th>( T_p )</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>( {T_1, T_4} )</td>
<td>5</td>
<td>( {T_1, T_4, T_2, T_3} )</td>
<td>0.400</td>
</tr>
<tr>
<td>time</td>
<td>( {T_2, T_3, T_4} )</td>
<td>3</td>
<td>( {T_2, T_3, T_4, T_1} )</td>
<td>0.714</td>
</tr>
<tr>
<td>ratio</td>
<td>( {T_2, T_4, T_3} )</td>
<td>3</td>
<td>( {T_2, T_4, T_3, T_1} )</td>
<td>0.743</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
<th>( R_4 )</th>
<th>( R_5 )</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>4</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Using **ratio** and **time** improves the CE of the prioritized test suite
Empirical Results: Efficiency

Prioritizers are **useful** in practice because they incur **low time** overheads
**Concluding Remarks**

- **Experimental** computer science involves the implementation and evaluation of algorithms that handle real-world problems.
- Solving the software crisis with freely available data sets and free/open source computational artifacts.

[Link to research: http://www.cs.allegheny.edu/~gkapfham/research/]