Smarter Load Test Suite Generation with Input Selection Support

Pingyu Zhang, Sebastian Elbaum, Matthew Dwyer
Collaborators: Corina Pasareanu, Indradeep Ghosh
MVD 10’, The University of Iowa, Iowa City, IA
September, 18, 2010
Load Testing

Will the application meet its performance requirements?
At what point will it not?
Route Generation App
Load Testing of Route Generation App

• Assume: response time requirement
• Check by generating tests with larger input rate or input size
  – More requests per second
  – Larger routes
• Limitation of generated tests
  – Do not know worst case
  – Often traverse single execution paths – less chances of findings faults
  – May not be able to scale beyond certain size because of technology limitations
Size is not all that matters

- Response time to find a route for 100 cities ranges from 30 seconds to 28 minutes (56X)

- Depends on the location of the cities provides – the particular inputs values can matter as much as the size
How to Improve: Need Smarter Test Generation that Considers Input Values

For that we need to look into the underlying code
How to Improve: Need Smarter Test Generation that Considers Input Values

```
static private void tspsearch (int nodes,
    int[] edges, int[] weight, int[] dist
    int[] row[], int[] column[], int[]
cursol[], int[] front[], int[] back[])
{
    if (edges == (nodes - 2)) {
        // complete route found
        ...
    } else {
        // identify candidate edge to add
        for (i=1; i<elms; i++)
            for (j=1; j<elms; j++)
                for (k=1; k<elms; k++)
                    ...
        ...
        tspsearch(nodes, edges+1, weight, ...);
    }
    ...
}
```
How to Improve: Need Smarter Test Generation that Considers Input Values

Loops & Recursions -> long paths -> heavy load

Challenge: How to craft an input that leads to such paths?

For that we need to look into the underlying code

```java
static private void tspsearch(int nodes, int edges, int weight, int dist[][], int row[], int column[], int curpos[], int front[], int back[]){

    if (edges == (nodes - 2)){
        // complete route found
    } else{
        // identify candidate edge to add
        for (i=1; i<=elms; i++)
            for (j=1; j<=elms; j++)
                for (k=1; k<=elms; k++)

        tspsearch(nodes, edges+1, weight, ...);

    if (thresh < dist[0][0]) {
        // Edge didn't help - try again
        tspsearch(nodes, edges, weight, ...);
    }

...}
```
Symbolic Execution

• Goal: A test input for every program path

```java
foo(int x, int y) {
    z = 2*x;
    if (z == x)
        if (x > y+8)
            print("Hi");
}
```

- Use symbolic test generation to explore program paths
  - Widely used in automated software testing: DART, CUTE, EXE, JPF, ...
Findings Long Paths with Symbolic Execution

- Naïve algorithm
  - Step 1: Generate every path on N inputs
  - Step 2: Return input for the longest path
- Cannot scale
  - For the TSP example, on an input of 5 cities, a full symbolic execution reveals 142352 possible paths, and takes 171 min
  - On input of 6 cities, full SE fails to finish in 4 hours
  - On larger inputs, problem remains unsolved...
Can we guide symbolic execution to focus on longest paths?

Directed Symbolic Execution

Histogram of paths in terms of bytecode count (5 cities), on a total of 142352 paths
Iterative-Deepening Beam Symbolic Execution

- Directed symbolic execution
  - All paths are allowed to deepen certain steps, then form a frontier
  - A path is more promising if it traverse through loops or recursions
  - Select a percentage of promising states from the frontier and resumes search towards the next frontier
  - Iterate on these steps until the ending criteria is met
Iterative-Deepening Beam Symbolic Execution

• Back to the TSP example
  – With Iterative-deepening search, we find 10 tests in 11 min (6% of full search)
  – All of them falls into the most expensive bar
Outline

• Introduction & Background

• Symbolic Generation of Load Tests
  – Parameterized Beam Search
  – Selecting Promising States
  – Dealing with Solver Limitations

• Implementation
  – Record and Replay of Paths

• Evaluation
  – RQ1: Effectiveness and Cost
  – RQ2: Scalability
Formalizing SymbolicLoadGeneration (SLG)

**Algorithm 1 SymbolicLoadGeneration** *(init, rate, levelPCSize, maxPCSize)*

```plaintext
Algorithm 1 SymbolicLoadGeneration (init, rate, levelPCSize, maxPCSize)

currentPCSize ← 0
promising ← init
search ← true
while search do
    currentPCSize ← currentPCSize + levelPCSize
    if currentPCSize > maxPCSize then
        currentPCSize ← maxPCSize
        search ← false
    end if
    frontier ← boundedSE(promising, currentPCSize)
    promising ← selectStates(frontier, rate)
    if largestPCSize(promising) < currentPCSize then
        search ← false
    end if
end while
return promising
```

Init state

-
Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration (init, rate, levelPCSsize, maxPCSsize)

\[
\begin{align*}
\text{currentPCSsize} & \leftarrow 0 \\
\text{promising} & \leftarrow \text{init} \\
\text{search} & \leftarrow \text{true} \\
\text{while} \text{ search} \text{ true} \text{ do} & \\
\text{currentPCSsize} & \leftarrow \text{currentPCSsize} + \text{levelPCSsize} \\
\text{if} \text{ currentPCSsize} > \text{maxPCSsize} \text{ then} & \\
\text{currentPCSsize} & \leftarrow \text{maxPCSsize} \\
\text{search} & \leftarrow \text{false} \\
\text{end if} & \\
\text{frontier} & \leftarrow \text{boundedSE( promising, currentPCSsize)} \\
\text{promising} & \leftarrow \text{selectStates(frontier, rate)} \\
\text{if largestPCSsize(promising) < currentPCSsize then} & \\
\text{search} & \leftarrow \text{false} \\
\text{end if} & \\
\text{end while} & \\
\text{return promising} &
\end{align*}
\]
Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration (init, rate, levelPCSize, maxPCSize)

```
currentPCSize ← 0
promising ← init
search ← true
while search do
  currentPCSize ← currentPCSize + levelPCSize
  if currentPCSize > maxPCSize then
    currentPCSize ← maxPCSize
    search ← false
  end if
  frontier ← boundedSE(promising, currentPCSize)
  promising ← selectStates(frontier, rate)
  if largestPCSize(promising) < currentPCSize then
    search ← false
  end if
end while
return promising
```
**Formalizing SymbolicLoadGeneration (SLG)**

**Algorithm 1** SymbolicLoadGeneration \((\text{init}, \text{rate}, \text{levelPCSize}, \text{maxPCSize})\)

```
currentPCSize ← 0
promising ← init
search ← true
while search do
    currentPCSize ← currentPCSize + levelPCSize
    if currentPCSize > maxPCSize then
        currentPCSize ← maxPCSize
        search ← false
    end if
    frontier ← boundedSE( promising, currentPCSize )
    promising ← selectStates(frontier, rate)
    if largestPCSize( promising ) < currentPCSize then
        search ← false
    end if
end while
return promising
```
Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration \((\text{init}, \text{rate}, \text{levelPCSize}, \text{maxPCSize})\)

\[
\begin{align*}
\text{currentPCSize} &\leftarrow 0 \\
\text{promising} &\leftarrow \text{init} \\
\text{search} &\leftarrow \text{true} \\
\text{while} \ \text{search} \ \text{do} &\ \\
\ &\ \text{currentPCSize} \leftarrow \text{currentPCSize} + \text{levelPCSize} \\
\ &\ \text{if} \ \text{currentPCSize} > \text{maxPCSize} \ \text{then} \\
\ &\ \ \ \text{currentPCSize} \leftarrow \text{maxPCSize} \\
\ &\ \ \ \text{search} \leftarrow \text{false} \\
\ &\ \ \ \text{end if} \\
\ &\ \ \ \text{frontier} \leftarrow \text{boundedSE( promising, currentPCSize)} \\
\ &\ \ \ \text{promising} \leftarrow \text{selectStates(frontier, rate)} \\
\ &\ \ \ \text{if} \ \text{largestPCSize( promising)} < \text{currentPCSize} \ \text{then} \\
\ &\ \ \ \ \ \ \ \text{search} \leftarrow \text{false} \\
\ &\ \ \ \\text{end if} \\
\ &\ \ \ \text{end while} \\
\text{return} \ \text{promising}
\end{align*}
\]
Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration \((init, \text{rate}, \text{levelPCSize}, \text{maxPCSize})\)

\begin{algorithm}
\begin{algorithmic}
\STATE currentPCSize \leftarrow 0
\STATE promising \leftarrow init
\STATE search \leftarrow true
\WHILE{search}
\STATE currentPCSize \leftarrow currentPCSize + \text{levelPCSize}
\IF{currentPCSize > maxPCSize}
\STATE currentPCSize \leftarrow maxPCSize
\STATE search \leftarrow false
\ENDIF
\STATE frontier \leftarrow boundedSE(\text{promising}, \text{currentPCSize})
\STATE promising \leftarrow selectStates(frontier, \text{rate})
\IF{largestPCSize(\text{promising}) < currentPCSize}
\STATE search \leftarrow false
\ENDIF
\ENDWHILE
\RETURN promising
\end{algorithmic}
\end{algorithm}

Init state

Frontier 1

Frontier 2
Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration (init, rate, levelPCSize, maxPCSize)

\[
\text{currentPCSize} \leftarrow 0 \\
\text{promising} \leftarrow \text{init} \\
\text{search} \leftarrow \text{true} \\
\text{while } \text{search} \text{ true do} \\
\quad \text{currentPCSize} \leftarrow \text{currentPCSize} + \text{levelPCSize} \\
\quad \text{if } \text{currentPCSize} > \text{maxPCSize} \text{ then} \\
\quad\quad \text{currentPCSize} \leftarrow \text{maxPCSize} \\
\quad\quad \text{search} \leftarrow \text{false} \\
\quad \text{end if} \\
\quad \text{frontier} \leftarrow \text{boundedSE} (\text{promising}, \text{currentPCSize}) \\
\quad \text{promising} \leftarrow \text{selectStates} (\text{frontier}, \text{rate}) \\
\quad \text{if } \text{largestPCSize} (\text{promising}) < \text{currentPCSize} \text{ then} \\
\quad\quad \text{search} \leftarrow \text{false} \\
\quad \text{end if} \\
\text{end while} \\
\text{return } \text{promising}
\]
How to Choose the Parameters

• Input parameters
  – *init*: test engineer should define input size
  – *rate*: iterative adjustment
  – *levelPCSize*: iterative adjustment, with simple heuristics (#branches)
  – *maxPCSize*: imposed by solver capability

• Tradeoff
  – *rate* & *levelPCSize* are essential for balancing between test quality and generation cost

Algorithm 1 SymbolicLoadGeneration (*init*, *rate*, *levelPCSize*, *maxPCSize*)

```plaintext
currentPCSize ← 0
promising ← *init*
search ← true
while search do
  currentPCSize ← currentPCSize + *levelPCSize*
  if currentPCSize > *maxPCSize* then
    currentPCSize ← *maxPCSize*
    search ← false
  end if
  frontier ← boundedSE(*promising*, currentPCSize)
  promising ← selectStates(frontier, *rate*)
  if largestPCSize(*promising*) < currentPCSize then
    search ← false
  end if
end while
return *promising*
```
Selecting Promising States

- Possible candidate strategies
  - Random
  - bytecode count / weighted bytecode count
  - Weight branches to bias towards loops and recursions
Selecting Promising States

- Possible candidate strategies
  - Random
  - bytecode count / weighted bytecode count
  - Weight branches to bias towards loops and recursions

- We implemented *Iteration Sensitive Branch Count* (ISBC)
  - for branch $b$, $ISBC(b) = loopNesting(b) + recursionNesting(b)$
  - For path $p$, where $B$ is the set of branches taken,
    $$ISBC(p) = \sum_{b \in B} (1 + w \times ISBC(b))$$

```plaintext
Algorithm 1 SymbolicLoadGeneration (init, rate, levelPCSize, maxPCSize)

| currentPCSize ← 0 |
| promising ← init |
| search ← true |

while search do
  currentPCSize ← currentPCSize + levelPCSize |
  if currentPCSize > maxPCSize then
    currentPCSize ← maxPCSize |
    search ← false |
  end if

frontier ← boundedSE(promising, currentPCSize) |
promising ← selectStates(frontier, rate) |
if largestPCSize(promising) < currentPCSize then
  search ← false |
end if
end while

return promising
```
Selecting Promising States

- Possible candidate strategies
  - Random
  - bytecode count / weighted bytecode count
  - Weight branches to bias towards loops and recursions

- Promote diversity \textit{Path Condition Difference Estimation} (PCDE)
  - First, compute the number of PCs in which each clause participates
  - Diversity of PC is \( \forall c \in ( \bigcup_{s \in \text{frontier}} PC(s)) : \text{count}(c) = |\{s \mid s \in \text{frontier} \land c \in PC(s)\}| \)

\[
\forall s \in \text{frontier} : PCDE(s) = \sum_{c \in PC(s)} \text{count}(c)
\]

\[
\begin{array}{c|c|c}
\text{c1} & 3 & \text{c1} & 3 & \text{c1} & 3 \\
\text{c2} & 2 & \text{c2} & 2 & \text{c4} & 2 \\
\text{c3} & 1 & \text{c4} & 2 & \text{c5} & 1 \\
\end{array}
\]

PC1 7 6
PC2 6
PC3 6

\begin{algorithm}
\caption{SymbolicLoadGeneration (\textit{init}, rate, levelPCSize, maxPCSize)}
\begin{algorithmic}
\State \text{currentPCSize} \leftarrow 0
\State \text{promising} \leftarrow \text{init}
\State \text{search} \leftarrow \text{true}
\While {\text{search}}
\State \text{currentPCSize} \leftarrow \text{currentPCSize} + \text{levelPCSize}
\If {\text{currentPCSize} > \text{maxPCSize}}
\State \text{currentPCSize} \leftarrow \text{maxPCSize}
\State \text{search} \leftarrow \text{false}
\EndIf
\State \text{frontier} \leftarrow \text{boundedSE(\textit{promising}, currentPCSize)}
\State \text{promising} \leftarrow \text{selectStates(frontier, rate)}
\If {\text{largestPCSize(\textit{promising})} < \text{currentPCSize}}
\State \text{search} \leftarrow \text{false}
\EndIf
\EndWhile
\State \text{return} \text{promising}
\end{algorithmic}
\end{algorithm}
Dealing with Solver Limitations

- **Challenge**
  - Every solver has an upper limit
  - It severely limits the input size that we can handle

- **Solution:** *ConstraintLimitedLoadGeneration* (CLLG-k)
  - Build a wrapper algorithm that uses SLG as a routine
  - Chains partial solutions together
  - Achieves scalability but sacrifices test quality
  - Introduce a new parameter: *maxSolverConstraints*
Outline

• Introduction & Background
• Symbolic Generation of Load Tests
  – Parameterized Beam Search
  – Selecting Promising States
  – Dealing with Solver Limitations
• Implementation
  – Record and Replay of Paths
• Evaluation
  – RQ1: Effectiveness and Cost
  – RQ2: Scalability
Implementation

• Implemented as an extension to jpf-symbc (SPF now)
• Record & replay of paths
  – At the frontier, for each selected promising state, externalize the branch choices made along the path leading to the state
  – Restarts JPF using recorded choices to guide execution up the frontier, then resumes search

• Advantage
  – No need to call solver during replay
  – Easy to parallelize
Implementation

• Test Instantiation
  – Tried three solvers with JPF: choco, cvc3, yices
  – Implemented new Yices Java API to work with JPF
  – Yices appears to be the most efficient solver

• JPF “Verify” Eclipse plug-in support
  – Config parameters with “.jpf” script and runs with Eclipse JPF plug-in

• Code base on NASA Babelfish repository
  – Listed as “symbc-load” extension
Outline

• Introduction & Background
• Symbolic Generation of Load Tests
  – Parameterized Beam Search
  – Selecting Promising States
  – Dealing with Solver Limitations
• Implementation
  – Record and Replay of Paths
• Evaluation
  – RQ1: Effectiveness and Cost
  – RQ2: Scalability
Evaluating SLG & CLLG-k

• RQ1
  – How cost-effective is SLG?
  – Study the rate parameter, on various small programs with fixed input size.

• RQ2
  – How scalable is CLLG-k (k stands for maxSolverConstraints)?
  – Study the maxSolverConstraints parameter for it’s effect on scalability
  – Use JZlib compression package as artifact
Study Design for RQ1

• Initialize
  – Each artifact takes an input of 1000 symbols
  – complete graph with symbolic weights / vectors with symbolic integers and booleans

• Parameters
  – levelPCSsize (see table), maxPCSsize=3000, rate=10%, 1%, 0.1%

• Control treatment
  – Random: generate & run for each test, always keeps the best 10
  – SLG and Random run for the same amount of time

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Source</th>
<th>LOC</th>
<th>Complexity</th>
<th>levelPCSsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford</td>
<td>JGraphT</td>
<td>835</td>
<td>$O(V \times E)$</td>
<td>12</td>
</tr>
<tr>
<td>Edmonds Karp</td>
<td>JGraphT</td>
<td>360</td>
<td>$O(V \times E^2)$</td>
<td>22</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>JGraphT</td>
<td>130</td>
<td>$O(n!)$</td>
<td>21</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>JGraphT</td>
<td>140</td>
<td>$O(\log(V) \times E)$</td>
<td>21</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>NASA</td>
<td>231</td>
<td>–</td>
<td>35</td>
</tr>
<tr>
<td>JZlib Compression</td>
<td>JCraft</td>
<td>4439</td>
<td>–</td>
<td>40</td>
</tr>
</tbody>
</table>
Results for RQ1: Effectiveness and Cost

- Average Execution Time of 10 tests

<table>
<thead>
<tr>
<th></th>
<th>SLG 10% rate</th>
<th>Random</th>
<th>SLG 1% rate</th>
<th>Random</th>
<th>SLG 0.1% rate</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>18.2</td>
<td>13.5</td>
<td>18.1</td>
<td>13.0</td>
<td>18.1</td>
<td>12.8</td>
</tr>
<tr>
<td>Edmonds Karp Max. Flow</td>
<td>19.1</td>
<td>14.5</td>
<td>18.6</td>
<td>12.7</td>
<td>17.0</td>
<td>12.1</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>205.3</td>
<td>77.2</td>
<td>204.2</td>
<td>61.3</td>
<td>198.5</td>
<td>58.4</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>2.2</td>
<td>1.8</td>
<td>2.1</td>
<td>1.8</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>0.003</td>
<td>0.002</td>
<td>0.003</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>
Results for RQ1: Effectiveness and Cost

- **Average Execution Time of 10 tests**

<table>
<thead>
<tr>
<th></th>
<th>SLG 10% rate</th>
<th>Random</th>
<th>SLG 1% rate</th>
<th>Random</th>
<th>SLG 0.1% rate</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>18.2</td>
<td>13.5</td>
<td>18.1</td>
<td>13.0</td>
<td>18.1</td>
<td>12.8</td>
</tr>
<tr>
<td>Edmonds Karp Max. Flow</td>
<td>19.1</td>
<td>14.5</td>
<td>18.6</td>
<td>12.7</td>
<td>17.0</td>
<td>12.1</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>205.3</td>
<td>77.2</td>
<td>204.2</td>
<td>61.3</td>
<td>198.5</td>
<td>58.4</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>2.2</td>
<td>1.8</td>
<td>2.1</td>
<td>1.8</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>0.003</td>
<td>0.002</td>
<td>0.003</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

- SLG tests beats Random across artifacts, across rates
Results for RQ1: Effectiveness and Cost

- Average Execution Time of 10 tests

<table>
<thead>
<tr>
<th>Test Generation Strategy</th>
<th>SLG 10% rate</th>
<th>Random</th>
<th>SLG 1% rate</th>
<th>Random</th>
<th>SLG 0.1% rate</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>18.2</td>
<td>13.5</td>
<td>18.1</td>
<td>13.0</td>
<td>18.1</td>
<td>12.8</td>
</tr>
<tr>
<td>Edmonds Karp Max. Flow</td>
<td>19.1</td>
<td>14.5</td>
<td>18.6</td>
<td>12.7</td>
<td>17.0</td>
<td>12.1</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>205.3</td>
<td>77.2</td>
<td>204.2</td>
<td>61.3</td>
<td>198.5</td>
<td>58.4</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>2.2</td>
<td>1.8</td>
<td>2.1</td>
<td>1.8</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>0.003</td>
<td>0.002</td>
<td>0.003</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

- SLG tests beats Random across artifacts, across rates
- Some have larger differences, some have smaller ones
Results for RQ1: Effectiveness and Cost

- **Average Execution Time of 10 tests**

<table>
<thead>
<tr>
<th></th>
<th>Test Generation Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SLG 10% rate</td>
</tr>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>18.2</td>
</tr>
<tr>
<td>Edmonds Karp Max. Flow</td>
<td>19.1</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>205.3</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>2.2</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>0.003</td>
</tr>
</tbody>
</table>

- SLG tests beats Random across artifacts, across rates
- Some have larger differences, some have smaller ones
- Reduction in *rate* did not degrade execution time very much
Results for RQ1: Effectiveness and Cost

• Average Execution Time of 10 tests

<table>
<thead>
<tr>
<th>Test Generation Strategy</th>
<th>SLG 10% rate</th>
<th>Random</th>
<th>SLG 1% rate</th>
<th>Random</th>
<th>SLG 0.1% rate</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>18.2</td>
<td>13.5</td>
<td>18.1</td>
<td>13.0</td>
<td>18.1</td>
<td>12.8</td>
</tr>
<tr>
<td>Edmonds Karp Max. Flow</td>
<td>19.1</td>
<td>14.5</td>
<td>18.6</td>
<td>12.7</td>
<td>17.0</td>
<td>12.1</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>205.3</td>
<td>77.2</td>
<td>204.2</td>
<td>61.3</td>
<td>198.5</td>
<td>58.4</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>2.2</td>
<td>1.8</td>
<td>2.1</td>
<td>1.8</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>0.003</td>
<td>0.002</td>
<td>0.003</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

  - SLG tests beats Random across artifacts, across rates
  - Some have larger differences, some have smaller ones
  - Reduction in rate did not degrade execution time very much

• Generation Costs

<table>
<thead>
<tr>
<th>SLG rate</th>
<th>10%</th>
<th>1%</th>
<th>0.1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford Shortest Path</td>
<td>178</td>
<td>77</td>
<td>24</td>
</tr>
<tr>
<td>Edmonds Karp Maximum Flow</td>
<td>212</td>
<td>105</td>
<td>35</td>
</tr>
<tr>
<td>Traveling Salesman Problem</td>
<td>231</td>
<td>101</td>
<td>35</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>96</td>
<td>37</td>
<td>14</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>112</td>
<td>43</td>
<td>15</td>
</tr>
</tbody>
</table>

Higher complexity

Large search space
Study Design for RQ2

• Initialize
  – Input size range from 1KB to 100MB

• Parameters
  – levelPCSize (see table), maxSloverConstraints=250, 500, 1000, 2000, rate=0.1%

• Control treatment
  – Random: generate & run for each test, always keeps the best 10

• 3-hour cap enforced across runs

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Source</th>
<th>LOC</th>
<th>Complexity</th>
<th>levelPCSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman Ford</td>
<td>JGraphT</td>
<td>835</td>
<td>$O(V \times E)$</td>
<td>12</td>
</tr>
<tr>
<td>Edmonds Karp</td>
<td>JGraphT</td>
<td>360</td>
<td>$O(V \times E^2)$</td>
<td>22</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>JGraphT</td>
<td>130</td>
<td>$O(n!)$</td>
<td>21</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>JGraphT</td>
<td>140</td>
<td>$O(\log(V) \times E)$</td>
<td>21</td>
</tr>
<tr>
<td>Wheel Brake System</td>
<td>NASA</td>
<td>231</td>
<td>—</td>
<td>35</td>
</tr>
<tr>
<td>JZlib Compression</td>
<td>JCraft</td>
<td>4439</td>
<td>—</td>
<td>40</td>
</tr>
</tbody>
</table>
Results for RQ2: Scalability

- Average execution time of 10 tests
  - Labels are generation costs in minutes

(a) Input sizes from 1MB - 100MB

(b) Input sizes from 1MB - 100MB
Results for RQ2: Scalability

• Average execution time of 10 tests
  – Labels are generation costs in minutes
  – No-limit scales to 100K only
Results for RQ2: Scalability

• Average execution time of 10 tests
  – Labels are generation costs in minutes
  – No-limit scales to 100K only
  – Reduction in limit helps with scalability, but degrades test quality
Results for RQ2: Scalability

- Average execution time of 10 tests
  - Labels are generation costs in minutes
  - No-limit scales to 100K only
  - Reduction in limit helps with scalability, but degrades test quality
  - 250-limit has the best scalability, still generates tests 8X better than Random
Diversity among Load Tests

- How to evaluate diversity among tests?
  - JZlib tests themselves are incomprehensible byte sets
  - Could analyze program behavior in terms of branch traces

For 1000-limit, 1MB input

<table>
<thead>
<tr>
<th>Test</th>
<th>Branch Count</th>
<th>Loop Count</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>216975</td>
<td>26327</td>
<td>((L1 L2)* L3 L4* ((L5* L6)* L7))** L10</td>
</tr>
<tr>
<td>2</td>
<td>221734</td>
<td>25437</td>
<td>((L1 L2)* (L3 L4)* ((L5* L6)* L7))** L10</td>
</tr>
<tr>
<td>3</td>
<td>234385</td>
<td>26654</td>
<td>((L1 L2)* L3 L4* ((L5 L6)* L8)* L9)** L10</td>
</tr>
<tr>
<td>4</td>
<td>232119</td>
<td>25587</td>
<td>same as test 2</td>
</tr>
<tr>
<td>5</td>
<td>211995</td>
<td>26544</td>
<td>((L1 L2)* L4* ((L5* L6)* L8))** L10</td>
</tr>
<tr>
<td>6</td>
<td>216981</td>
<td>26439</td>
<td>same as test 2</td>
</tr>
<tr>
<td>7</td>
<td>219588</td>
<td>26325</td>
<td>((L1 L2)* L4* ((L5* L6)* (L7* L8))<em>)</em> L10</td>
</tr>
<tr>
<td>8</td>
<td>224438</td>
<td>26435</td>
<td>((L1 L2)* L3 L4* ((L5 L6)* L8)* L10</td>
</tr>
<tr>
<td>9</td>
<td>215437</td>
<td>27751</td>
<td>((L1 L2)* L3 L4* ((L5* L6)* L7))** L10</td>
</tr>
<tr>
<td>10</td>
<td>231375</td>
<td>26415</td>
<td>same as test 7</td>
</tr>
</tbody>
</table>

- No identical tests, as shown by #branch and #loop
- 7 unique patterns, as shown by RegEx on loop execution sequences
Summary

• Identified
  – the need for value selection support in load testing

• Proposed
  – an approach that use symbolic execution with iterative-deepening beam search to generate load tests

• Implemented
  – SLG and CLLG-k in JPF

• Evaluated
  – with real world applications
Future Work

- Compositional load generation
  - inspired by unix pipes

```
tar cvf - . | gzip > myfile.tar.gz
```

- Collect PCs on each program and combine them together by relaxing and resolving contradictions
- May scale beyond pipes
Discussion