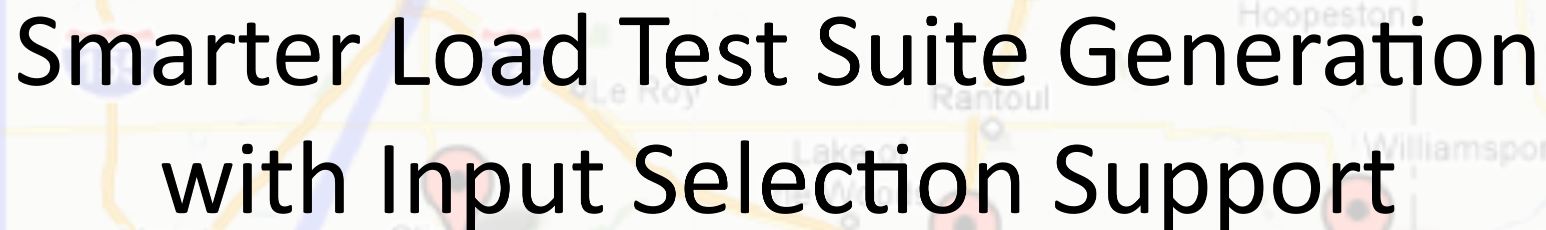


Smarter Load Test Suite Generation with Input Selection Support

Pingyu Zhang, Sebastian Elbaum, Matthew Dwyer

Collaborators: Corina Pasareanu, Indradeep Ghosh

MVD 10', The University of Iowa, Iowa City, IA

September, 18, 2010

University of Nebraska Lincoln

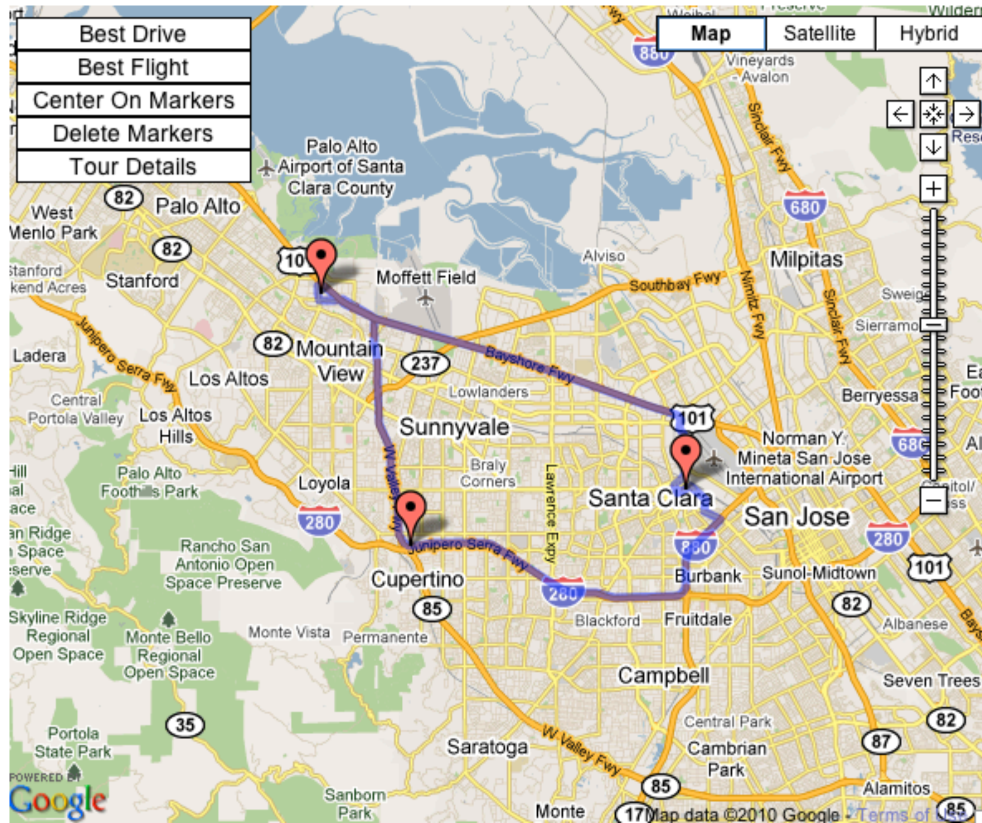
FUJITSU

Load Testing

Will the application meet its performance requirements?

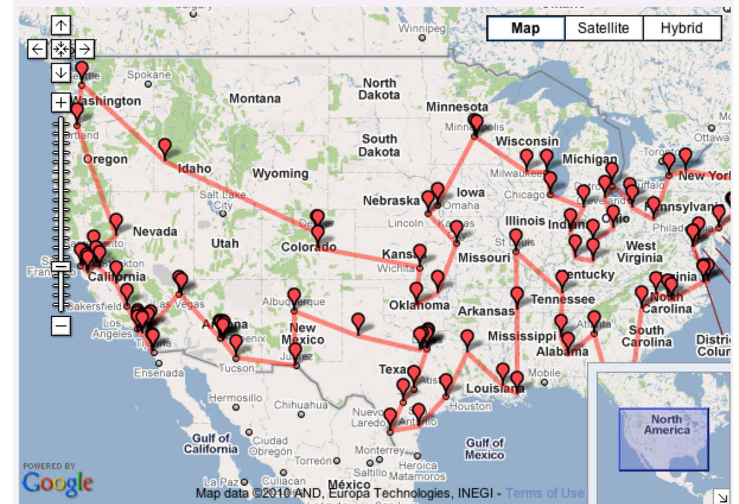
At what point will it not?

Route Generation App



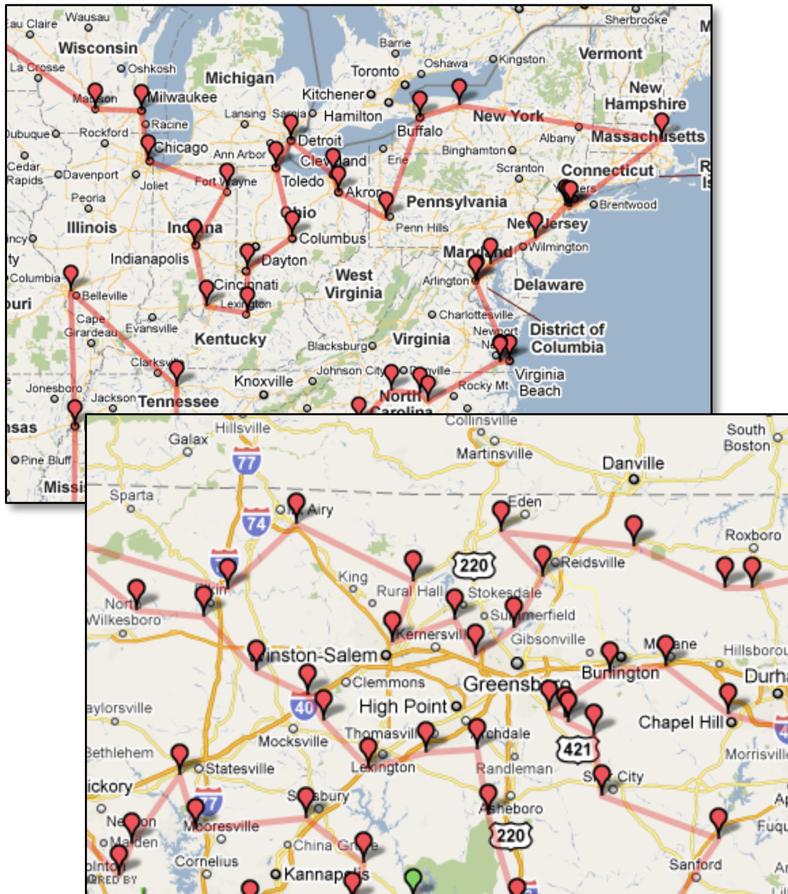
Load Testing of Route Generation App

- 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



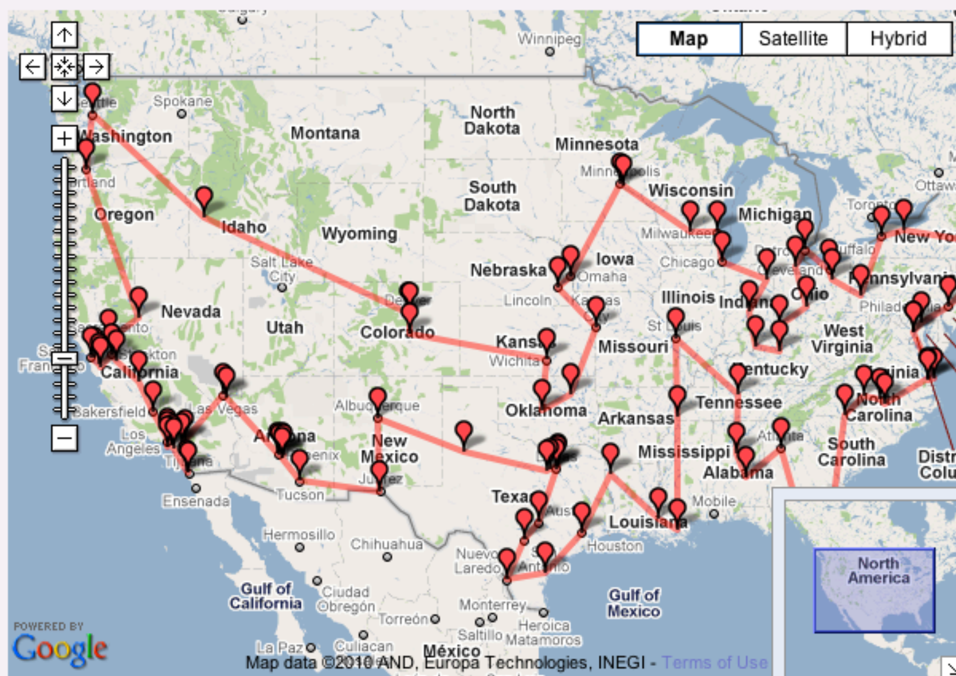
Routes with 100 cities

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

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

For that we need to look into the underlying code

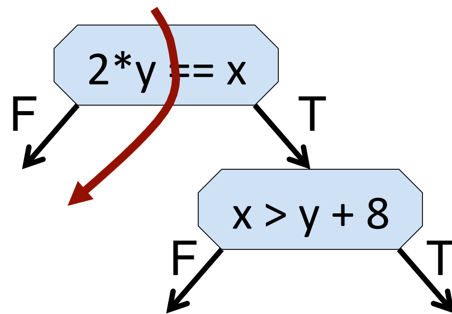
Loops & Recursions -> long paths -> heavy load

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

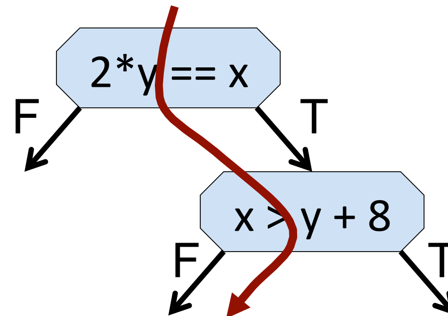
Symbolic Execution

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

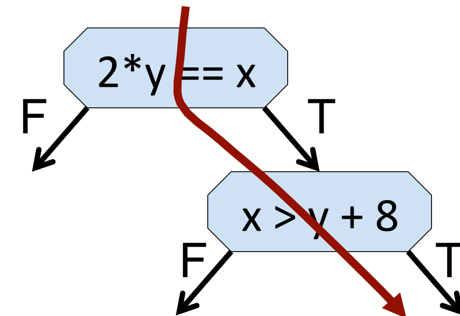
- Goal: A test input for every program path



PC: $2y \neq x$
Input: x=0, y=1



PC: $2y = x \wedge x \leq y+8$
Input: x=1, y=2



PC: $2y = x \wedge x > y+8$
Input: x=-10, y=-20

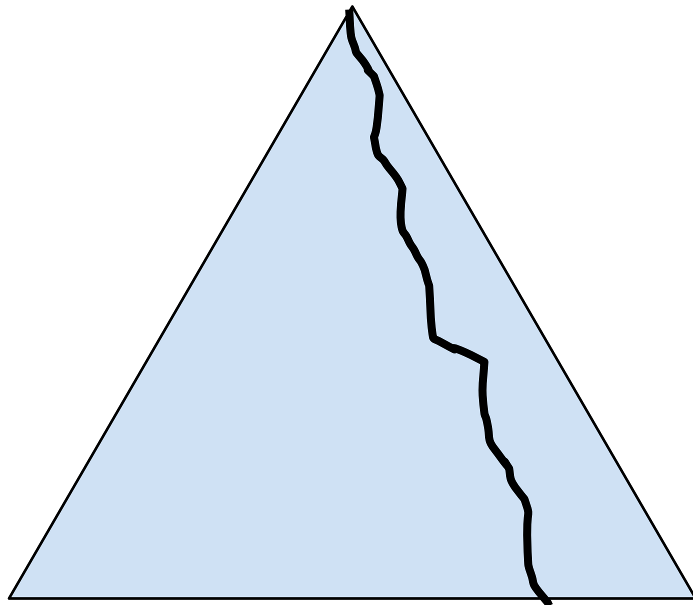
- Use symbolic test generation to explore program paths
  - Widely used in automated software testing: DART, CUTE, EXE, JPF, …

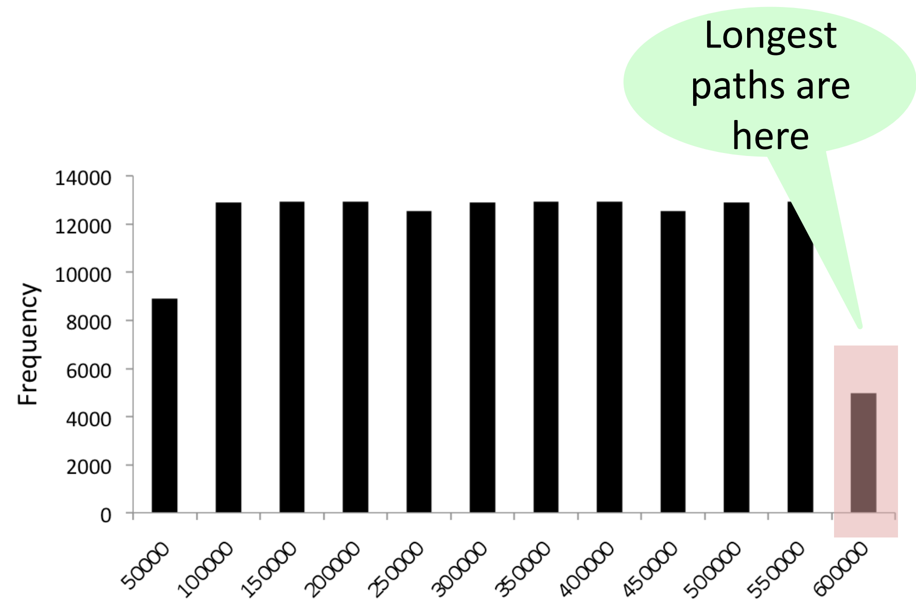
Findings Long Paths with Symbolic Execution

- 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…

Directed Symbolic Execution

- Can we guide symbolic execution to focus on longest paths?

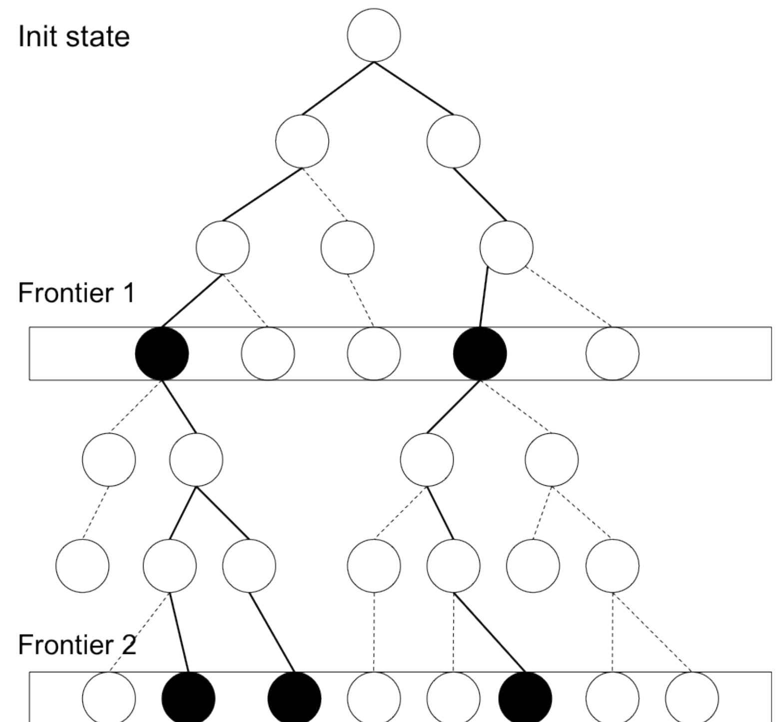




Histogram of paths in terms of bytecode count (5 cities), on a total of 142352 paths

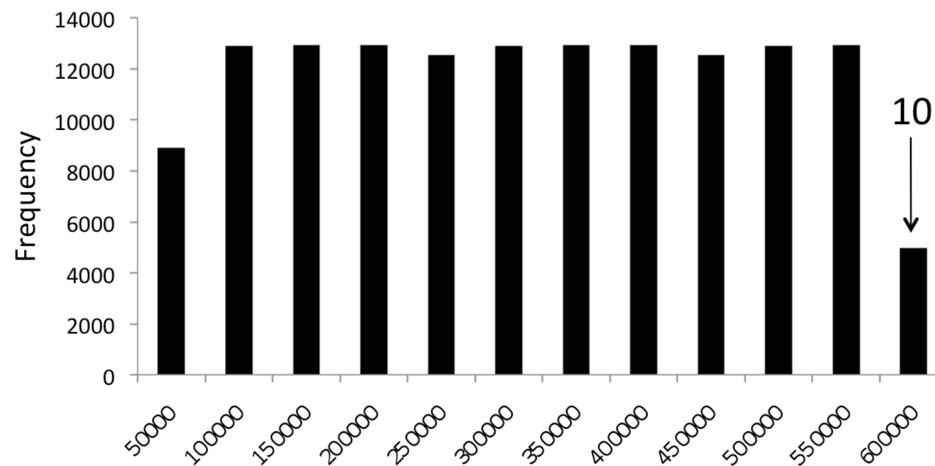
Iterative-Deepening Beam Symbolic Execution

- 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

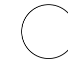
- Introduction & Background
- Symbolic Generation of Load Tests
  - Parameterized Beam Search
  - Selecting Promising States
  - Dealing with Solver Limitations
- Implementation
  - Record and Replay of Paths
- Evaluation
  - RQ1: Effectiveness and Cost
  - RQ2: Scalability

Formalizing SymbolicLoadGeneration (SLG)

Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, 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

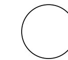


Formalizing SymbolicLoadGeneration (SLG)

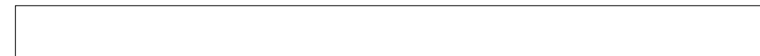
Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, rate, levelPCSize, maxPCSize)

```
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



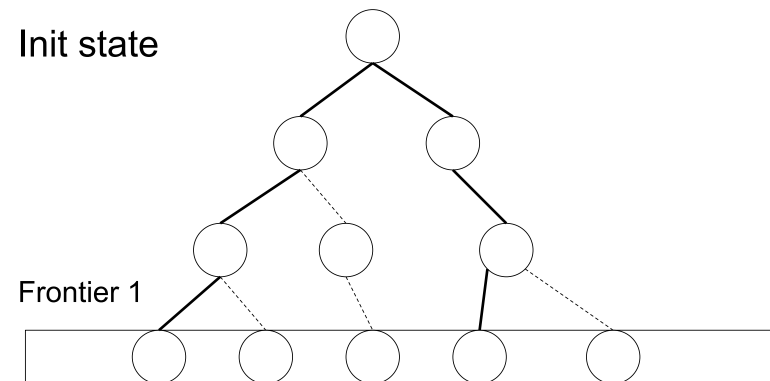
Frontier 1



Formalizing SymbolicLoadGeneration (SLG)

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

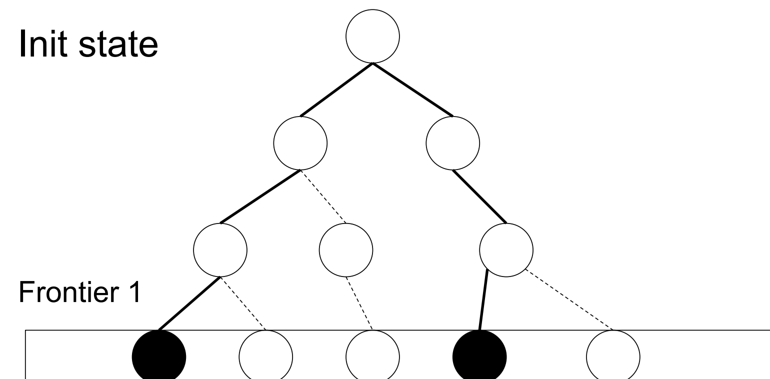
currentPCSize ← 0
$\overline{promising}$ ← $\overline{init}$
search ← true
while search **do**
  currentPCSize ← currentPCSize + levelPCSize
  if currentPCSize > maxPCSize **then**
    currentPCSize ← maxPCSize
    search ← false
  end if
  $\overline{frontier}$ ← boundedSE($\overline{promising}$, currentPCSize)
  $\overline{promising}$ ← selectStates($\overline{frontier}$, rate)
  if largestPCSize($\overline{promising}$) < currentPCSize **then**
    search ← false
  end if
end while
return $\overline{promising}$



Formalizing SymbolicLoadGeneration (SLG)

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

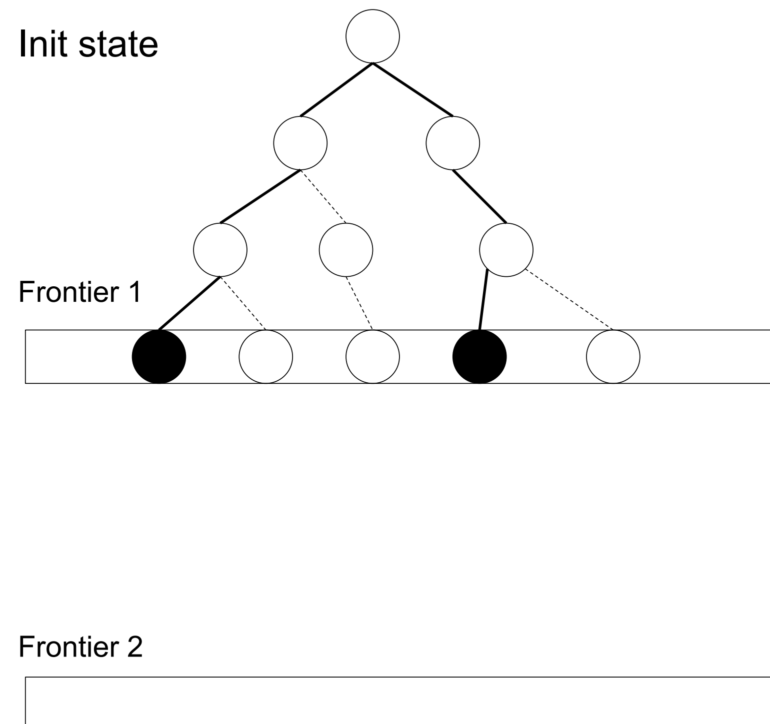
currentPCSize ← 0
$\overline{promising}$ ← $\overline{init}$
search ← true
while search **do**
  currentPCSize ← currentPCSize + levelPCSize
  if currentPCSize > maxPCSize **then**
    currentPCSize ← maxPCSize
    search ← false
  end if
  $\overline{frontier}$ ← boundedSE($\overline{promising}$, currentPCSize)
  $\overline{promising}$ ← selectStates($\overline{frontier}$, rate)
  if largestPCSize($\overline{promising}$) < currentPCSize **then**
    search ← false
  end if
end while
return $\overline{promising}$



Formalizing SymbolicLoadGeneration (SLG)

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

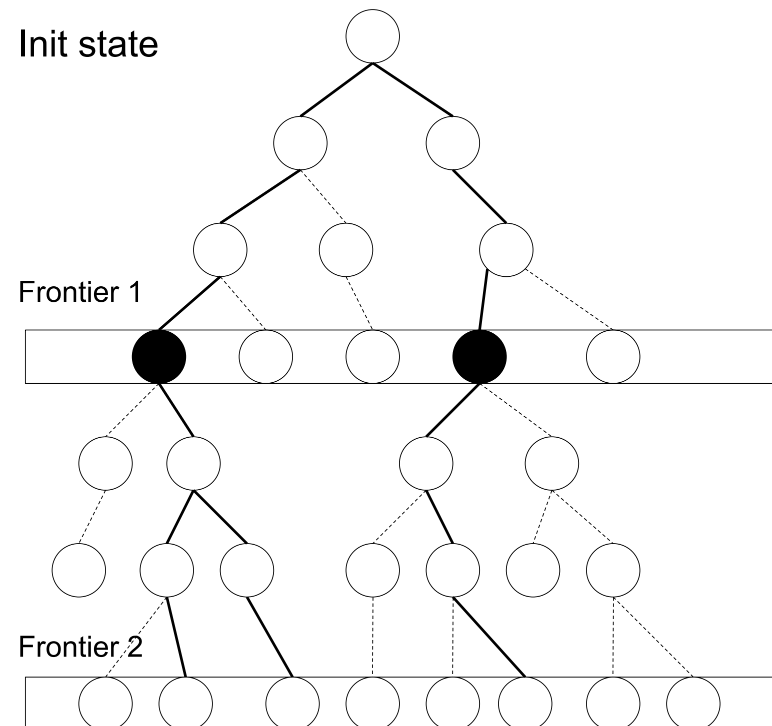
```
currentPCSize ← 0
promising ← init
search ← true
while search do
    currentPCSize ← currentPCSize + levelPCSize
    if currentPCSize > maxPCSize then
        currentPCSize ← maxPCSize
        search ← false
    end if
    frontier ← boundedSE( promising, currentPCSize)
    promising ← selectStates(frontier,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



Formalizing SymbolicLoadGeneration (SLG)

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

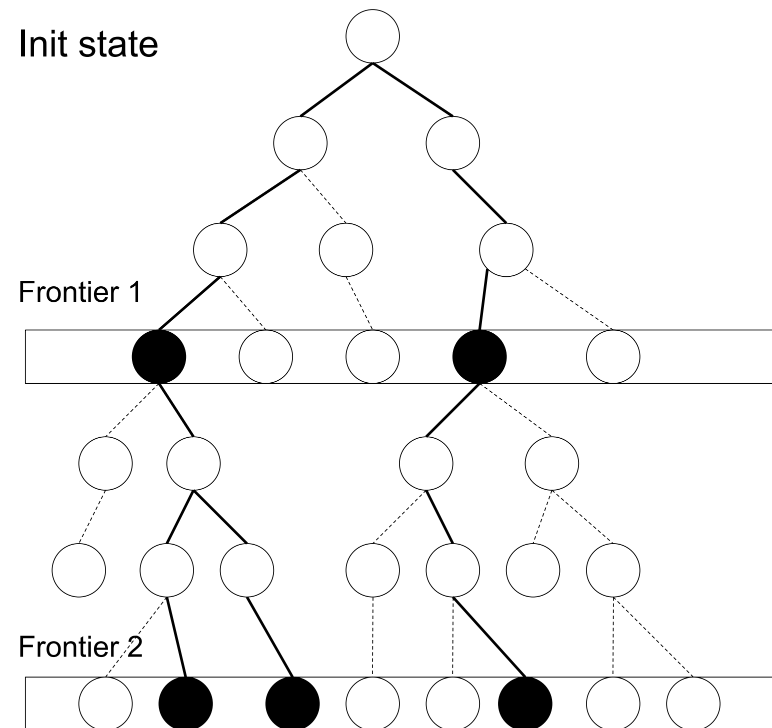
```
currentPCSize ← 0
promising ← init
search ← true
while search do
   currentPCSize ← currentPCSize + levelPCSize
   if currentPCSize > maxPCSize then
      currentPCSize ← maxPCSize
      search ← false
   end if
   frontier ← boundedSE( promising, currentPCSize)
   promising ← selectStates(frontier,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



Formalizing SymbolicLoadGeneration (SLG)

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

currentPCSize $\leftarrow$ 0
$\overline{promising} \leftarrow \overline{init}$
search $\leftarrow$ true
while search **do**
  currentPCSize $\leftarrow$ currentPCSize + levelPCSize
  if currentPCSize > maxPCSize **then**
    currentPCSize $\leftarrow$ maxPCSize
    search $\leftarrow$ false
  end if
  $\overline{frontier} \leftarrow$ boundedSE($\overline{promising}$, currentPCSize)
  $\overline{promising} \leftarrow$ selectStates($\overline{frontier}$, rate)
  if largestPCSize($\overline{promising}$) < currentPCSize **then**
    search $\leftarrow$ false
  end if
end while
return $\overline{promising}$



How to Choose the Parameters

Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, 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
```

- 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

Selecting Promising States

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

Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, 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
- 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,

$$ISBC(p) = \sum_{b \in B} (1 + w * ISBC(b))$$

Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, 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
```

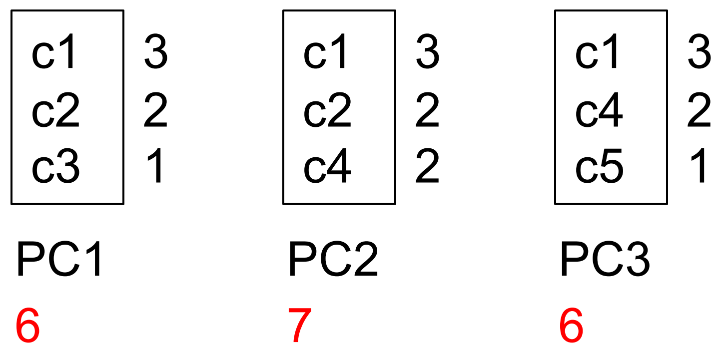
Selecting Promising States

- Possible candidate strategies
  - Random
  - bytecode count / weighted bytecode count
  - Weight branches to bias towards loops and recursions
- Promote diversity *Path Condition Difference Estimation* (PCDE)
  - First, compute the number of PCs in which each clause participates
  - Diversity of PC is $\forall c \in (\bigcup_{s \in frontier} PC(s)) : count(c) = |\{s \mid s \in frontier \wedge c \in PC(s)\}|$

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

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

currentPCSize $\leftarrow$ 0
$\overline{promising} \leftarrow \overline{init}$
search $\leftarrow$ true
while search **do**
  currentPCSize $\leftarrow$ currentPCSize + levelPCSize
  if currentPCSize $>$ maxPCSize **then**
    currentPCSize $\leftarrow$ maxPCSize
    search $\leftarrow$ false
  end if
  $\overline{frontier} \leftarrow$ boundedSE($\overline{promising}$, currentPCSize)
  $\overline{promising} \leftarrow$ selectStates($\overline{frontier}$, rate)
  if largestPCSize($\overline{promising}$) $<$ currentPCSize **then**
    search $\leftarrow$ false
  end if
end while
return $\overline{promising}$



Dealing with Solver Limitations

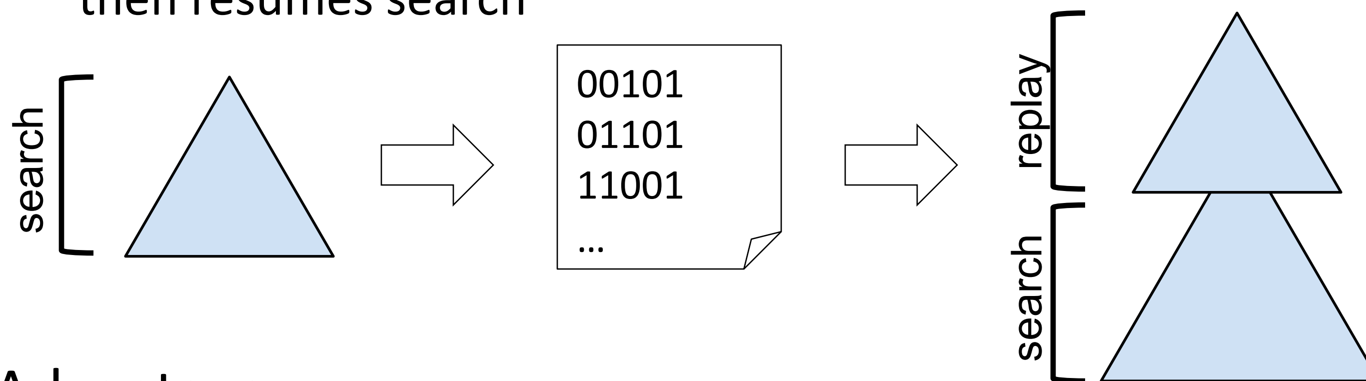
- Challenge
  - Every solver has an upper limit
  - It severely limits the input size that we can handle
- Solution: *ConstraintLimitedLoadGeneration* (CLLG-k)
  - Build a wrapper algorithm that uses SLG as a routine
  - Chains partial solutions together
  - Achieves scalability but sacrifices test quality
  - 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

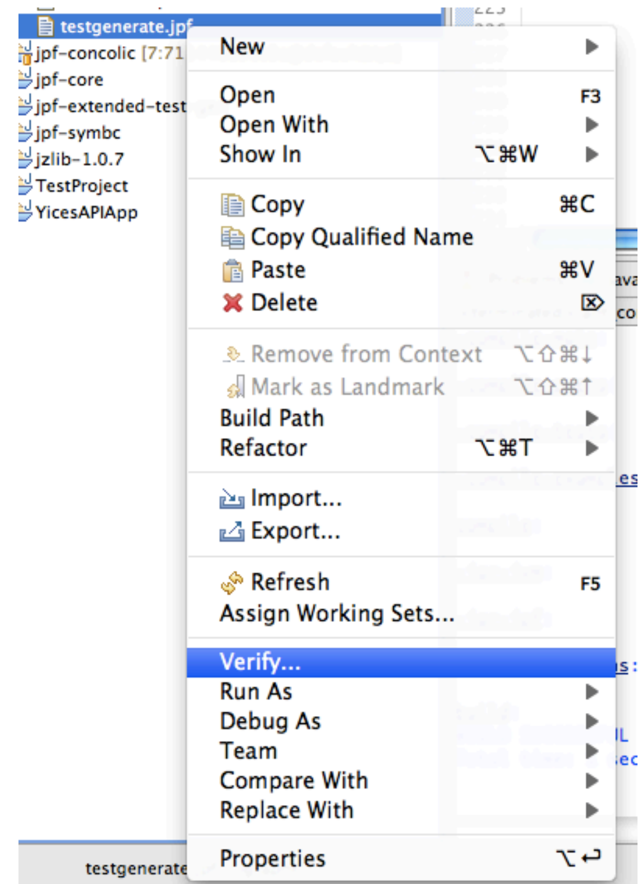
- Implemented as an extension to jpf-symbc (SPF now)
- Record & replay of paths
  - At the frontier, for each selected promising state, externalize the branch choices made along the path leading to the state
  - Restarts JPF using recorded choices to guide execution up the frontier, then resumes search



- 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
  - *levelPCSize* (see table), *maxPCSize*=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

Artifact	**Source**	**LOC**	**Complexity**	**levelPCSize**
Bellman Ford	JGraphT	835	$O(V \times E)$	12
Edmonds Karp	JGraphT	360	$O(V \times E^2)$	22
Traveling Salesman	JGraphT	130	$O(n!)$	21
Transitive Closure	JGraphT	140	$O(log(V) \times E)$	21
Wheel Brake System	NASA	231	–	35
JZlib Compression	JCraft	4439	–	40

Results for RQ1: Effectiveness and Cost

- Average Execution Time of 10 tests

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

Results for RQ1: Effectiveness and Cost

- Average Execution Time of 10 tests

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

  – SLG tests beats Random across artifacts, across rates

Results for RQ1: Effectiveness and Cost

- Average Execution Time of 10 tests

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

  - 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

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

  - 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

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

  - 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

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

Higher complexity

Large search space

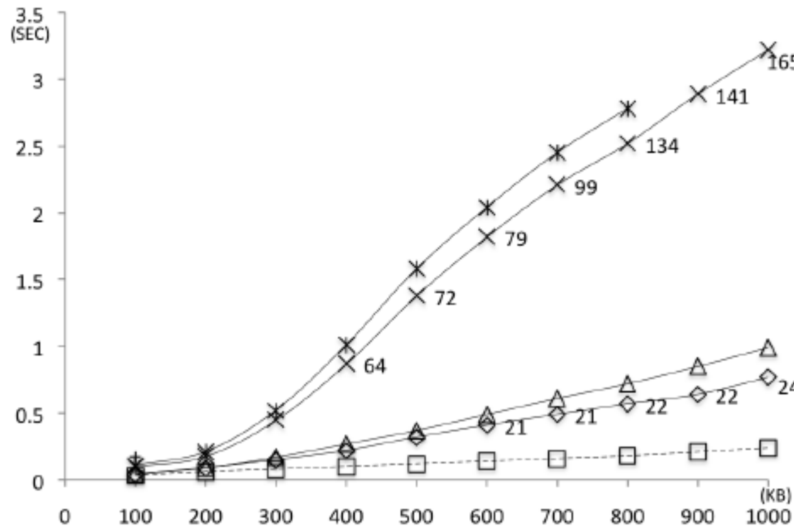
Study Design for RQ2

- 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

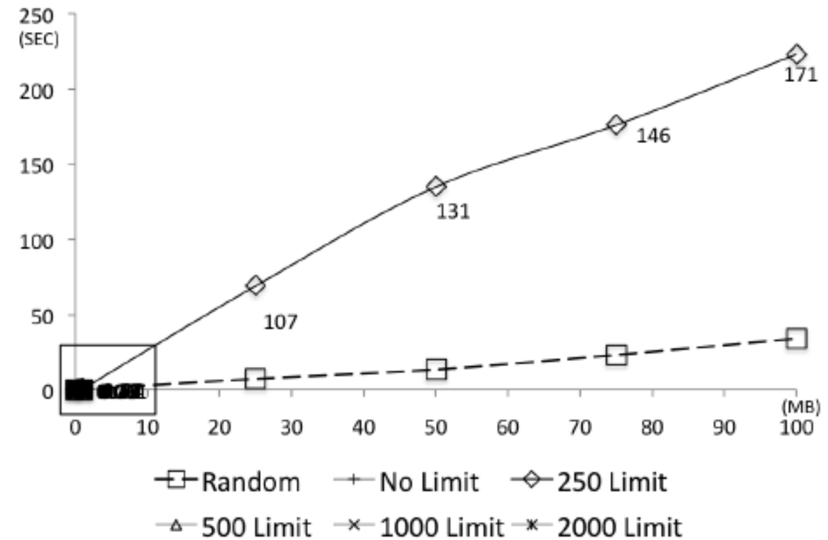
Artifact	**Source**	**LOC**	**Complexity**	**levelPCSize**
Bellman Ford	JGraphT	835	$O(V \times E)$	12
Edmonds Karp	JGraphT	360	$O(V \times E^2)$	22
Traveling Salesman	JGraphT	130	$O(n!)$	21
Transitive Closure	JGraphT	140	$O(log(V) \times E)$	21
Wheel Brake System	NASA	231	–	35
JZlib Compression	JCraft	4439	–	40

Results for RQ2: Scalability

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



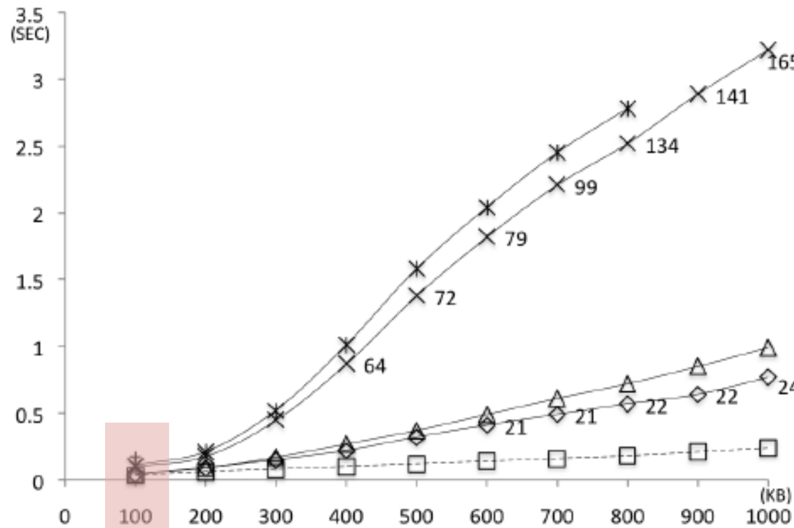
(a) Input sizes from 1MB - 100MB



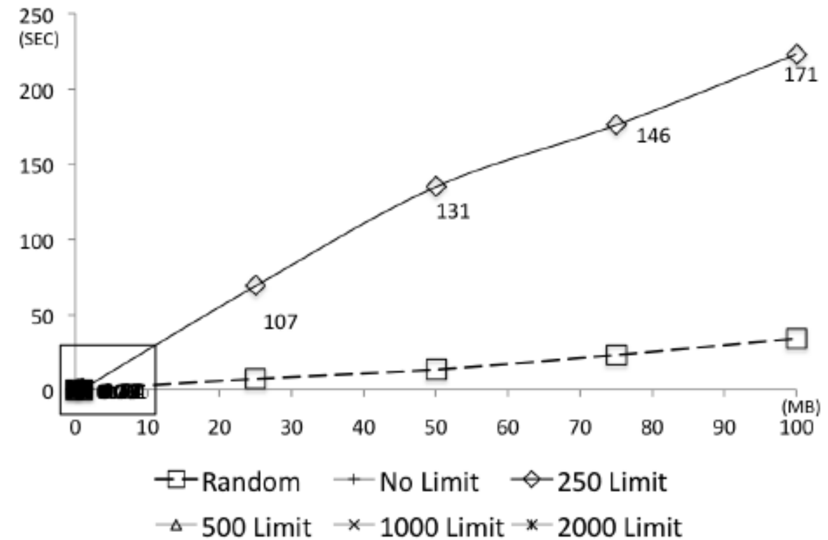
(b) Input sizes from 1MB - 100MB

Results for RQ2: Scalability

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



(a) Input sizes from 1MB - 100MB

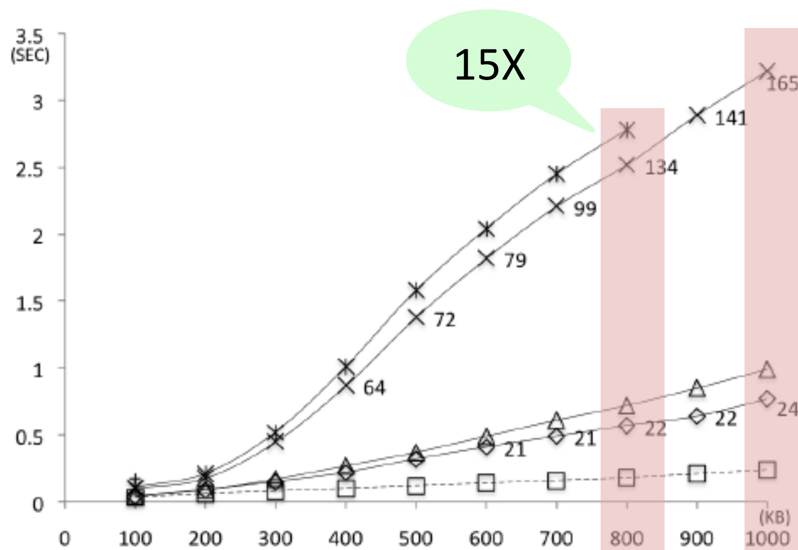


(b) Input sizes from 1MB - 100MB

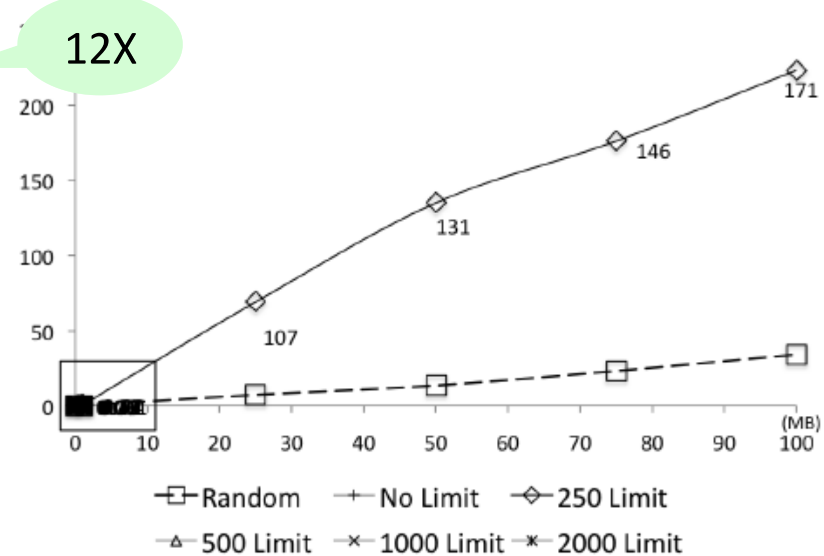
  - No-limit scales to 100K only

Results for RQ2: Scalability

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



(a) Input sizes from 1MB - 100MB

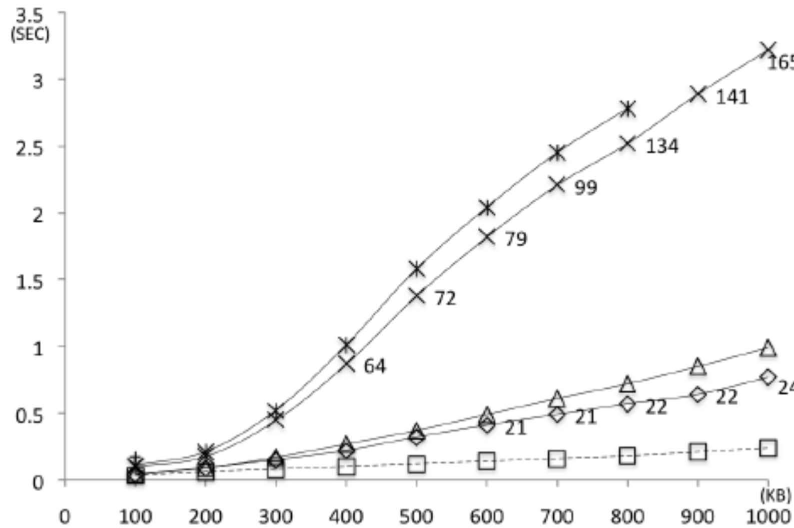


(b) Input sizes from 1MB - 100MB

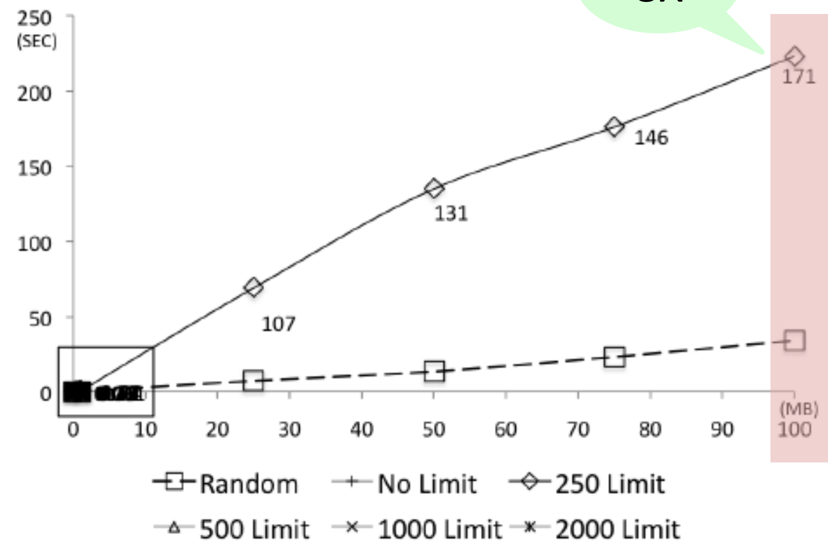
  - No-limit scales to 100K only
  - Reduction in limit helps with scalability, but degrades test quality

Results for RQ2: Scalability

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



(a) Input sizes from 1MB - 100MB



(b) Input sizes from 1MB - 100MB

  - 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

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

  - 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