

Automated Test Generation By Avoiding Redundant Tests

Tao Xie

Joint work with Darko Marinov (UIUC) and David Notkin

Dept. of Computer Science & Engineering
University of Washington

5 Oct. 2004, Microsoft Research

Motivation

- Tool generated test cases
 - Many test cases
 - Important to reduce by eliminating “redundant” test cases
 - Need automation
- Common approach
 - Identify “similar” test cases and eliminate
 - With minimal reduction of “quality” of test suite
- Object-oriented programs
 - Test case is a sequence of method calls on an object
 - Note: Unit tests only

Overview

- Motivation
- Rostra framework for detecting redundant tests
[ASE 04]
- Test generation by avoiding redundant tests
- Conclusions

Example Code

[Henkel&Diwan 03]

```
public class IntStack {
    private int[] store;
    private int size;
    public IntStack() { ... }
    public void push(int value) { ... }
    public int pop() { ... }
    public boolean isEmpty() { ... }
    public boolean equals(Object o) { ... }
}
```

Example Generated Tests

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```

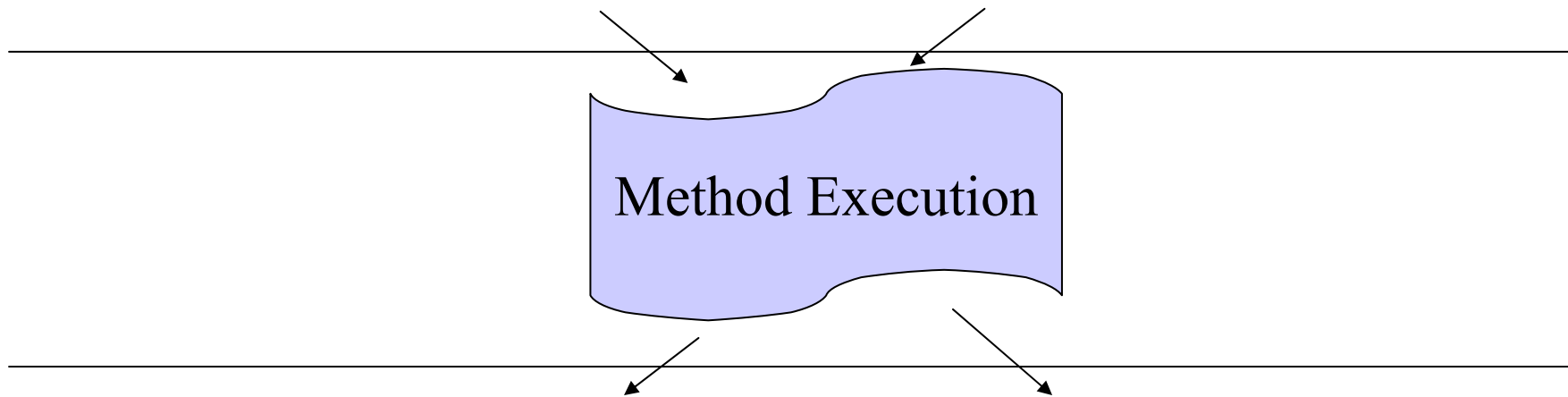
Test 3 (T3):

```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```

Same inputs \Rightarrow Same behavior

Assumption: deterministic method

Input = object state @entry + Method arguments



Output = object state @exit + Method return

Testing a method with the same inputs is unnecessary

How to represent object states?

Redundant Test Cases Defined

- Equivalent method executions
 - the same method names, signatures, and input (equivalent object states @entry and arguments)
- Redundant test case:
 - A test case is redundant for a test suite if the test suite has exercised method executions equivalent to all method executions exercised by the test case

Comparison with SpecExplorer

- Code vs. models
 - SpecExplorer generates tests from models, so one may want a different definition of redundant tests
 - One test is redundant w.r.t. another in the model but might not be redundant for the code
- Binary predicates vs. abstraction functions
 - $s1 == s2 \Leftrightarrow m(s1, s2)$ [boolean-returning method]
 - $s1 == s2 \Leftrightarrow a(s1) == a(s2)$ [abstraction function]

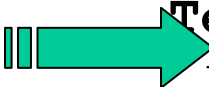
Five State-Representation Techniques

- Method-sequence representations
 - WholeSeq
 - The entire sequence
 - ModifyingSeq
 - Ignore methods that don't modify the state
- Concrete-state representations
 - WholeState
 - The full concrete state
 - MonitorEquals
 - Relevant parts of the concrete state
 - PairwiseEquals
 - **equals ()** method used to compare pairs of states

WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]



Test 1 (T1):
IntStack s1 =
 new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);

Test 3 (T3):
IntStack s3 =
 new IntStack();
s3.push(3);
s3.push(2);
s3.pop();

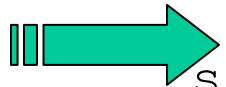
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 3 (T3):

```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```

<init>().state

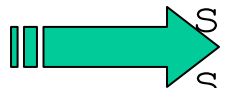
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 3 (T3):

```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```

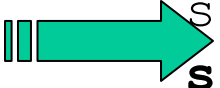
isEmpty(<init>().state).state

WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

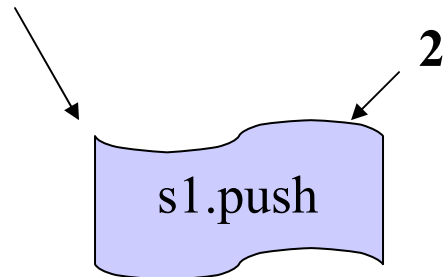
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```

push(isEmpty(<init>().state).state, 3).state



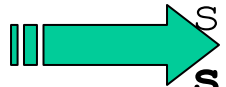
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

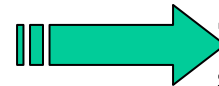
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

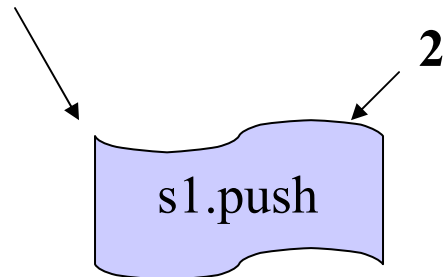


Test 3 (T3):

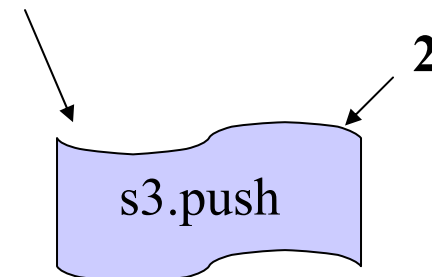
```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```



push(isEmpty(<init>().state).state, 3).state



push(<init>().state, 3).state

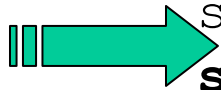


ModifyingSeq Representation

State-modifying method sequences that create objects

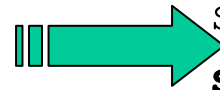
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

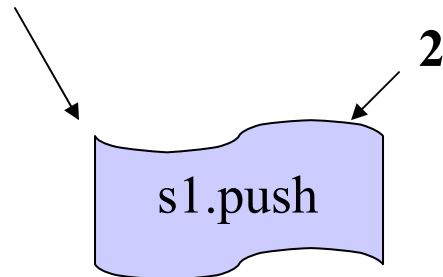


Test 3 (T3):

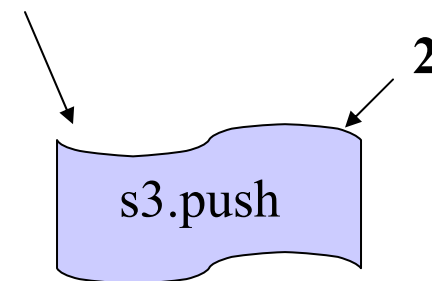
```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```



~~push(~~isEmpty~~(~~<init>~~().state, ~~state~~, 3).state~~



push(~~<init>~~().state, 3).state



WholeState Representation

The entire concrete state reachable from the object

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

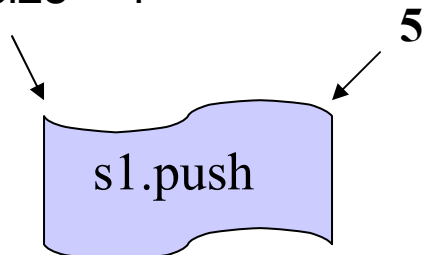


Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```

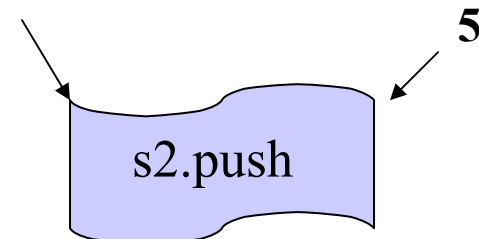


store.length = 3
store[0] = 3
store[1] = 2
store[2] = 0
size = 1



**Comparison by
isomorphism**

store.length = 3
store[0] = 3
store[1] = 0
store[2] = 0
size = 1



MonitorEquals Representation

The relevant part of the concrete state defined by *equals* (invoking *obj.equals(obj)* and monitor field accesses)

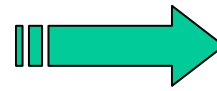
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```



store.length = 3

store[0] = 3

~~store[1] = 2~~

~~store[2] = 0~~

size = 1

5

s1.push

Comparison by
isomorphism

store.length = 3

store[0] = 3

~~store[1] = 0~~

~~store[2] = 0~~

size = 1

5

s2.push

PairwiseEquals Representation

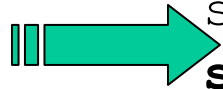
The results of *equals* invoked to compare pairs of states

Test 1 (T1):

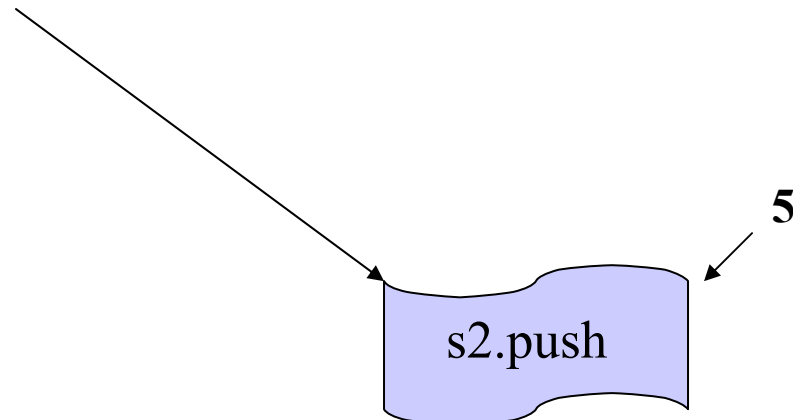
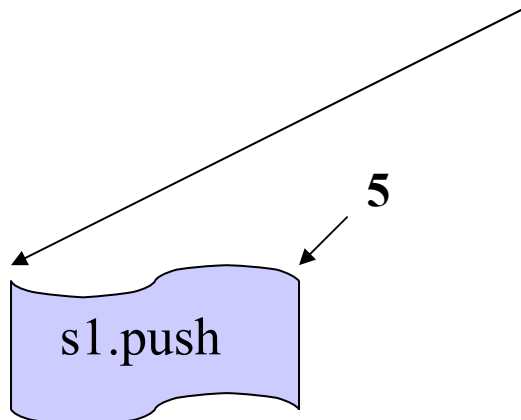
```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```



`s1.equals(s2) == true`



MonitorEquals vs. PairwiseEquals

- MonitorEquals monitors field accesses during execution of the *equals()* method and compares the monitored parts
- PairwiseEquals relies only on the output of the *equals()* method
- Example of sets

Detected Redundant Tests

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```

Test 3 (T3):

```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```

<i>technique</i>	<i>detected redundant tests w.r.t. T1</i>
WholeSeq	
ModifyingSeq	T3
WholeState	T3
MonitorEquals	T3, T2
PairwiseEquals	T3, T2

Experiment:

Evaluated Test Generation Tools

- ParaSoft Jtest 4.5
 - A commercial Java testing tool
 - Generates tests with method-call lengths up to three
- JCrasher 0.2.7
 - An academic robustness testing tool
 - Generates tests with method-call lengths of one

Questions to Be Answered

- How much do we benefit after applying Rostra on tests generated by Jtest and JCrasher?
- Does redundant-test removal decrease test suite quality?

Experimental Subjects

<i>class</i>	<i>methods</i>	<i>public methods</i>	<i>ncnb loc</i>	<i>Jtest tests</i>	<i>JCrasher tests</i>
IntStack	5	5	44	94	6
UBStack	11	11	106	1423	14
ShoppingCart	9	8	70	470	31
BankAccount	7	7	34	519	135
BinSearchTree	13	8	246	277	56
BinomialHeap	22	17	535	6205	438
DisjSet	10	7	166	779	64
FibonacciHeap	24	14	468	3743	150
HeapMap	27	19	597	5186	47
LinkedList	38	32	398	3028	86
TreeMap	61	25	949	931	1000

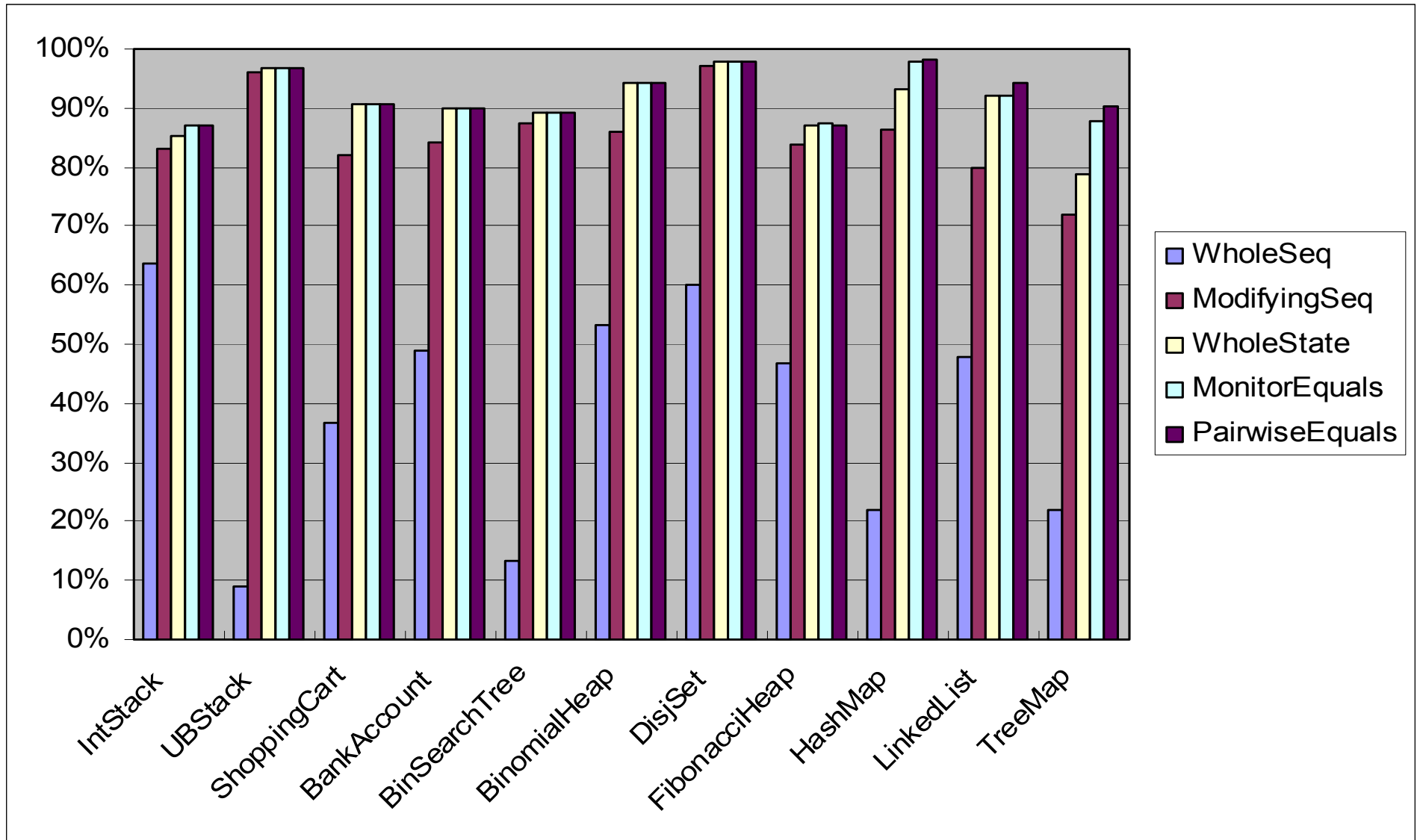
Assumptions About Subjects

- Method-sequence representations assume that each method does not modify argument state
- MonitorEquals and PairwiseEquals representations assume a user-defined `equals()`

Quality of Original Test Suites

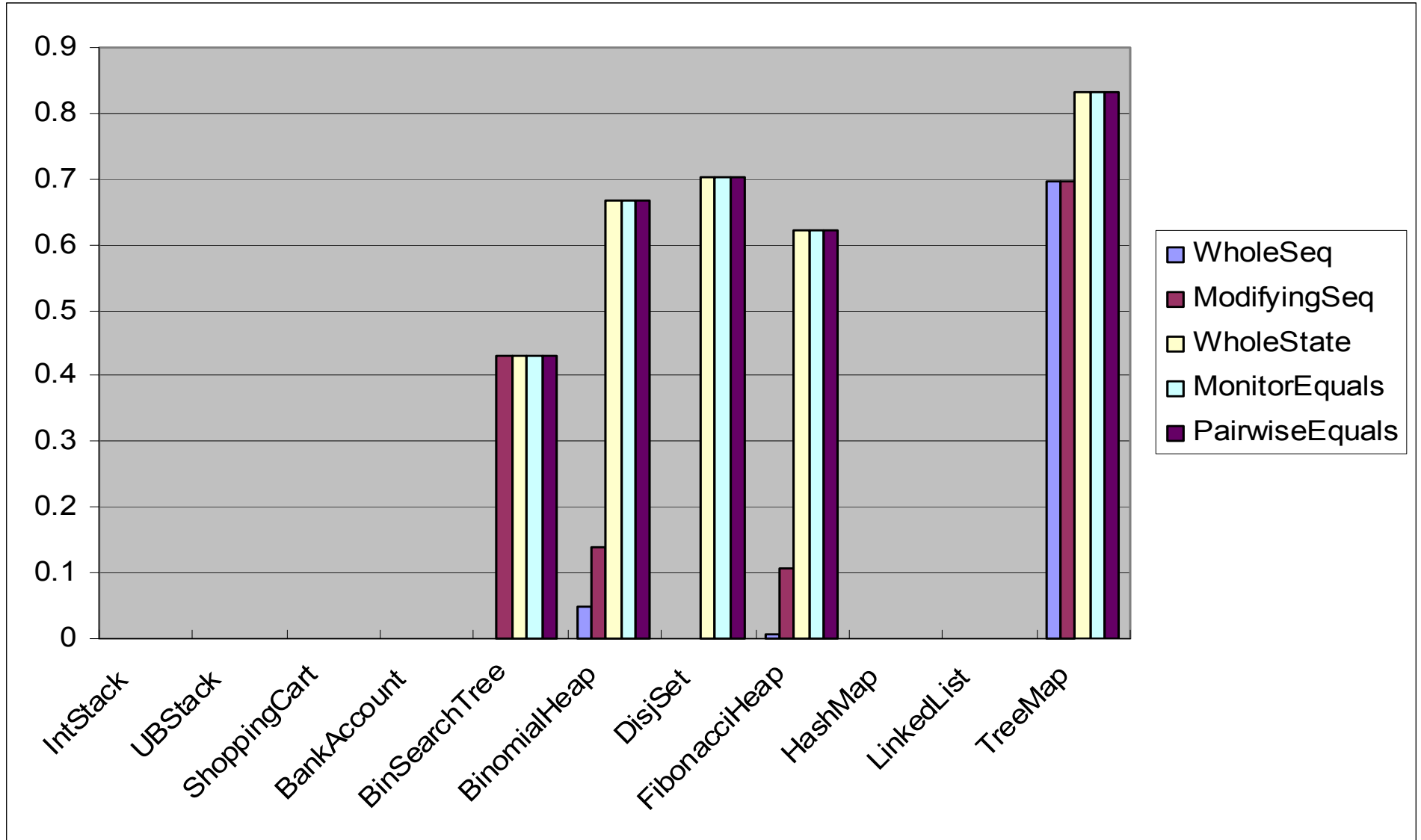
	Jtest-generated tests	JCrasher-generated tests
Avg num uncaught exceptions	4	2
Avg Branch cov	77%	52%
Avg mutant killing ratio (600 mutants)	53%	30%

Redundancy among Jtest-generated Tests



- The last three techniques detect around 90% redundant tests
- Detected redundancy in increasing order for five techniques

Redundancy among JCrasher-generated Tests



- The last three techniques detect over 50% on half subjects
- JCrasher generates fewer tests and shorter tests

Quality of Minimized Test Suites

- All five techniques on JCrasher preserve all measurements
- The first three techniques on Jtest preserve all measurements.
- Two *equals* techniques on Jtest decrease (with only small loss in 2 programs)
 - in branch cov %
 - in mutant killing %

Comparison of Five Techniques

- Time and space taken to find redundant tests
 - from a couple of seconds to several minutes across subjects
 - in roughly increasing order except for pairwiseEquals (being the least expensive)
- The number of redundant tests found
 - in increasing order
(WholeSeq, ModifyingSeq, WholeState, MonitorEquals, PairwiseEquals)

Redundancy Prediction

- ModifySeq tends to be better than WholeSeq
 - when more state-preserving methods
- WholeState tends to be better than ModifySeq
 - when more nonprogressive state-modifying methods
- MonitorEquals or PairwiseEquals tend to be better than WholeState
 - when more irrelevant object fields and these fields are assigned different values during method sequence calls

Evolution of ParaSoft Jtest

- Version 4.5 (released in March 2002) allows method-call lengths (1 — 3) [studied in this work, first published in a tech report in Jan 2004]
- Version 5.0 (released in Feb 2004) allows method-call length of only 1
- Recently ParaSoft notified us that Version 6.0 (internal version, not yet released) has addressed the test redundancy issue identified by us and added back the option to generate long call sequence

Overview

- Motivation
- Rostra framework for detecting redundant tests
[ASE 04]
- Test generation by avoiding redundant tests
- Conclusions

Test Generation Algorithm

Breadth-first exploration of (object) state space

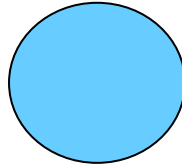
Inputs: some argument lists A for methods under test

1. iterationNum = 1
2. put initial object states into a frontier set S
3. create an empty new frontier set S'
4. for each state s in S {
 for each argument list a in A {
 invoke s on a (a new test)
 if method-exit state s' is a new state
 put s' into S'
 }
}
5. iterationNum++
6. replace S with S'
7. if (S is not empty && iterationNum \leq MAXITER) goto 3

Example

Inputs: `push(3)` , `push(2)` , `push(5)` , `pop()` , `isEmpty()`

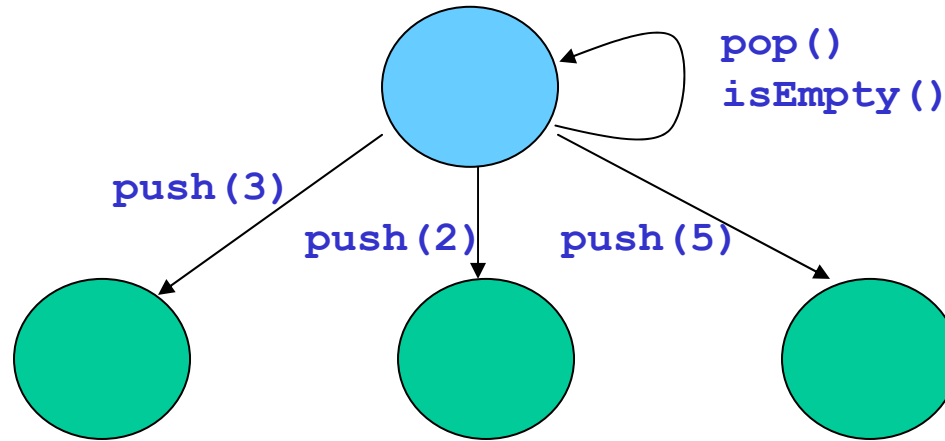
after `new IntStack()`



Example

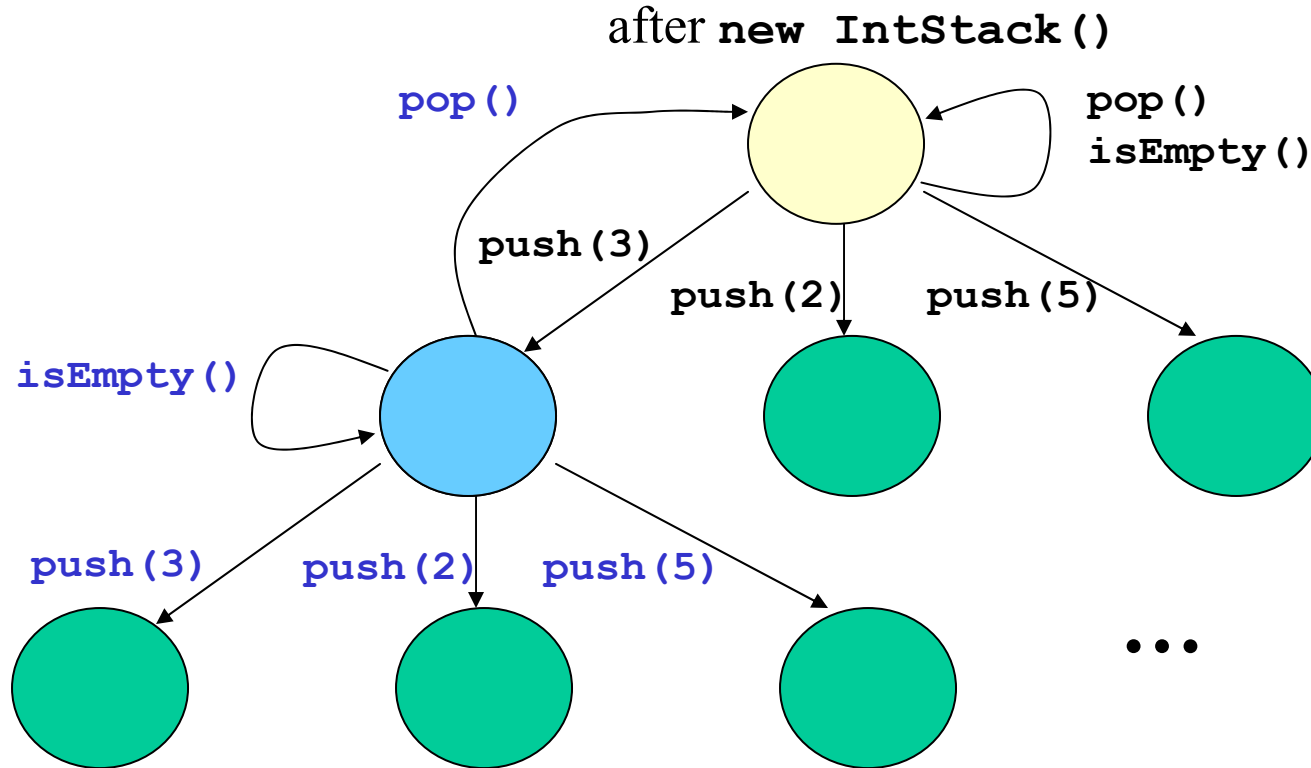
Inputs: `push(3)` , `push(2)` , `push(5)` , `pop()` , `isEmpty()`

after `new IntStack()`



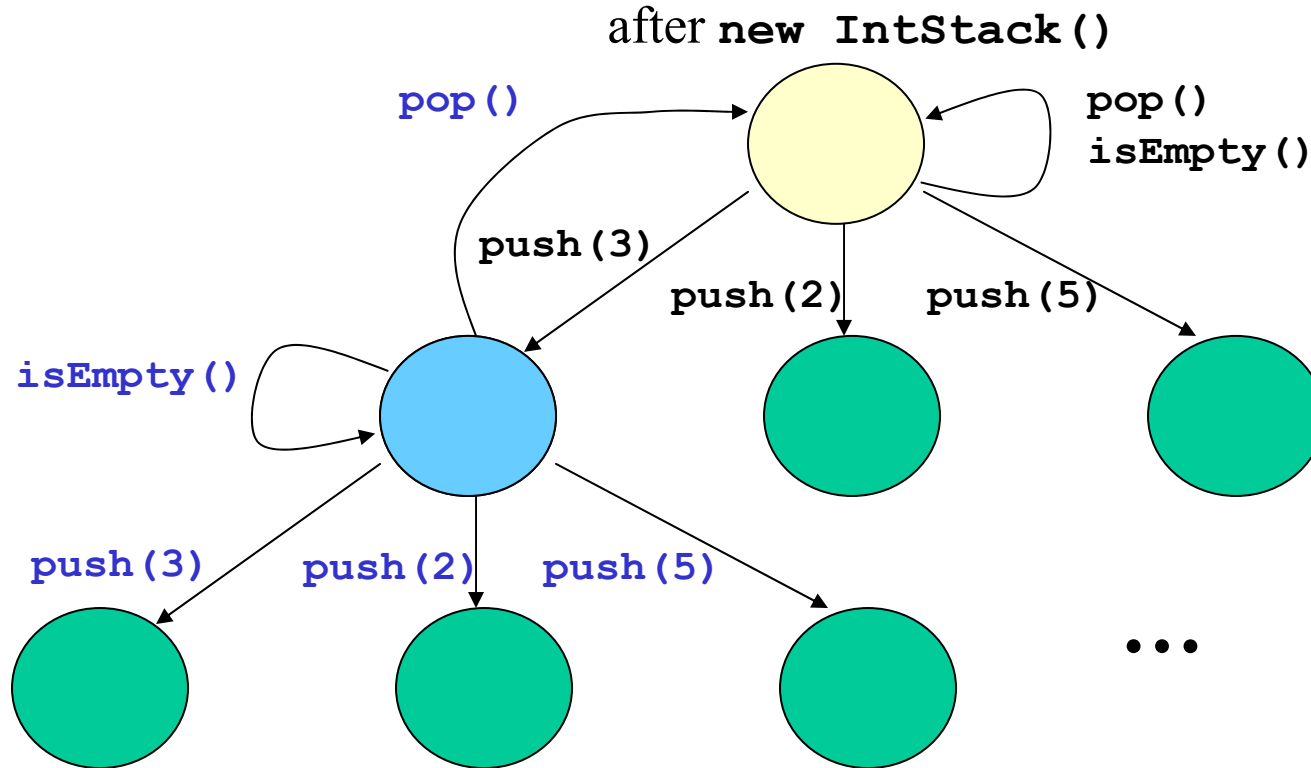
Example

Inputs: `push(3)`, `push(2)`, `push(5)`, `pop()`, `isEmpty()`



Example

Inputs: `push(3)`, `push(2)`, `push(5)`, `pop()`, `isEmpty()`



After n iterations, state space $O(3^n)$

If the stack is a unique int stack, we need 10 different arguments to reach size of 10, state space 9864100!

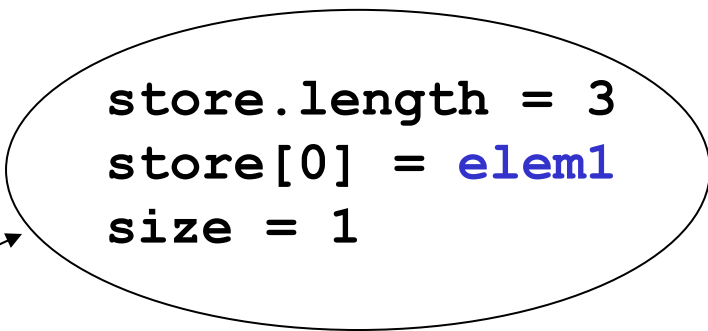
Symbolic Execution to Rescue

- Execute a method on symbolic argument values
 - Inputs: `push(SymbolicInt svalue)`, `pop()`, `isEmpty()`
- Use code instrumentation for symbolic execution [Khurshid et al. TACAS 03]
- Explore paths within the method execution
 - Use reexecution to implement backtracking [not store states inside a method execution]
 - Build a path condition for each path
 - Check satisfiability of path condition [CVC Lite, Omega]

Symbolic State Representation

- Symbolic state: $\langle \text{state}, \text{path condition} \rangle$
(For now, assume the stack is a unique int stack)

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);
```



store.length = 3
store[0] = elem1
size = 1

State 1

Symbolic State Representation

- Symbolic state: $\langle \text{state}, \text{path condition} \rangle$
(For now, assume the stack is a unique int stack)

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);  
s1.push(elem2);
```

store.length = 3
store[0] = elem1
size = 1

State 1

elem1 != elem2

store.length = 3
store[0] = elem1
store[1] = elem2
size = 2

State 2

elem1 == elem2

store.length = 3
store[0] = elem1
size = 1

State 3

Equivalent Symbolic States?

- Symbolic state: $\langle \text{state}, \text{path condition} \rangle$
- Equivalence: equiv state part + equiv pc

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);  
s1.push(elem2);
```

store.length = 3
store[0] = elem1
size = 1

State 1

elem1 != elem2

store.length = 3
store[0] = elem1
store[1] = elem2
size = 2

State 2

elem1 == elem2

store.length = 3
store[0] = elem1
size = 1

State 3

Removing Irrelevant Constraints

- Symbolic state: $\langle \text{state}, \text{path condition} \rangle$
- Equivalence: equiv state part + equiv pc

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);  
s1.push(elem2);
```

store.length = 3
store[0] = elem1
size = 1

State 1

elem1 != elem2

store.length = 3
store[0] = elem1
store[1] = elem2
size = 2

State 2

~~elem1 == elem2~~

store.length = 3
store[0] = elem1
size = 1

State 3

Comparison of Path Conditions

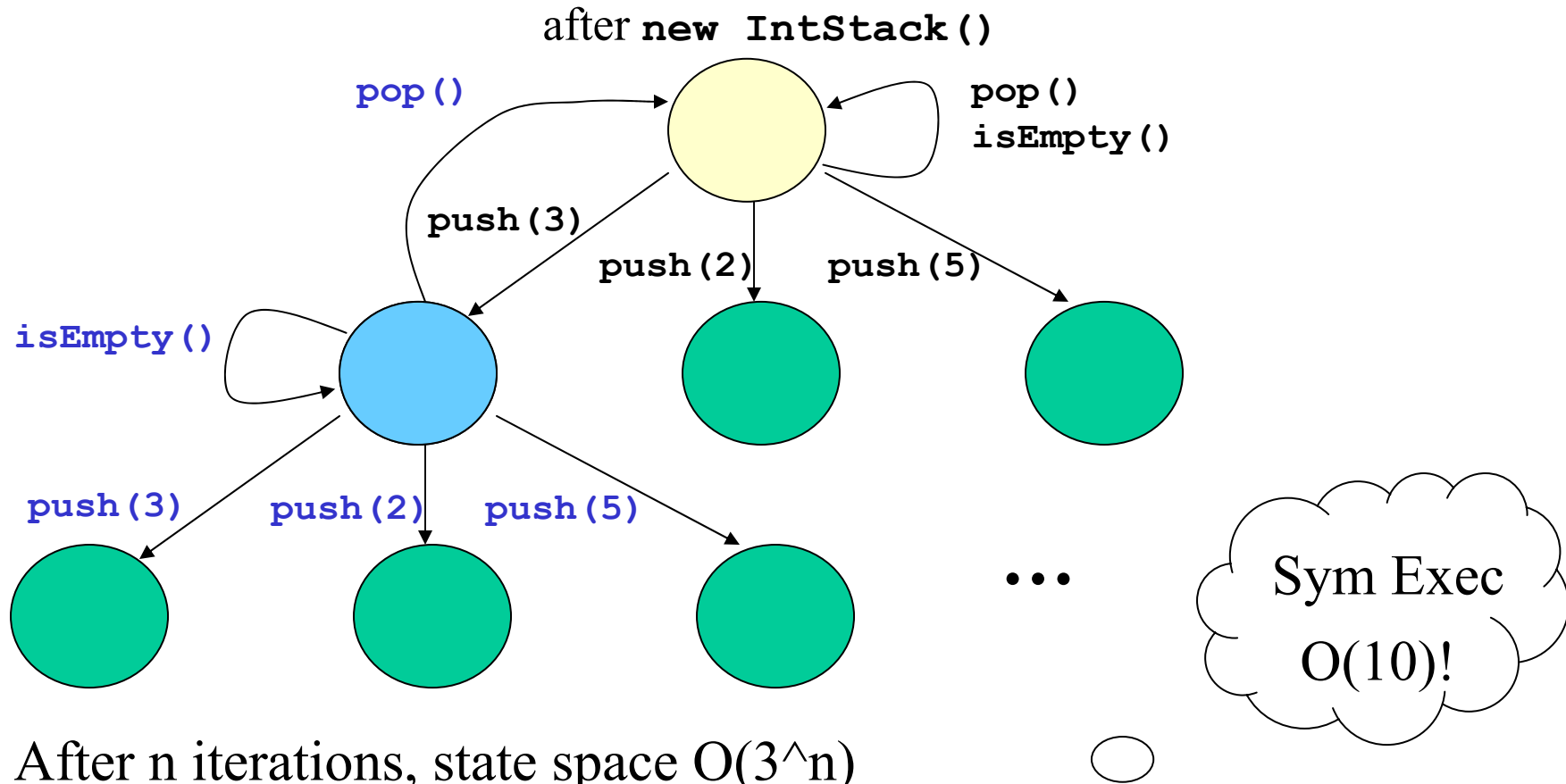
- Currently based on implications
 - $S1: \langle s1, pc1 \rangle$ an old state
 - $S2: \langle s2, pc2 \rangle$ a state under consideration
 - $s1$ is equiv to $s2$
 - if $pc2 \Rightarrow pc1$, $S2$ isn't considered a new state
 - e.g. $(x > \text{deleted} \ \&\& \ \text{deleted} > y) \Rightarrow (x > y)$
 - otherwise, $S2$ is a new state
- A pure method modifies path conditions but doesn't produce new states
- Merge symbolic states with equiv state parts
 - OR to connect their path conditions

Heuristics

- To make the generation faster, but have less guarantees than testing all sequences/paths
 - symbolically explore some sequences
 - e.g. ignoring PC's in state comparison
 - e.g. randomly choose some methods/args to invoke
 - symbolically explore some paths
 - e.g. favor paths that potentially lead to uncovered branches
 - put some parameters concrete
 - e.g. `TreeMap.put(SymbolicInt key, Object value)`

Example – Concrete Inputs

Inputs: `push(3)`, `push(2)`, `push(5)`, `pop()`, `isEmpty()`

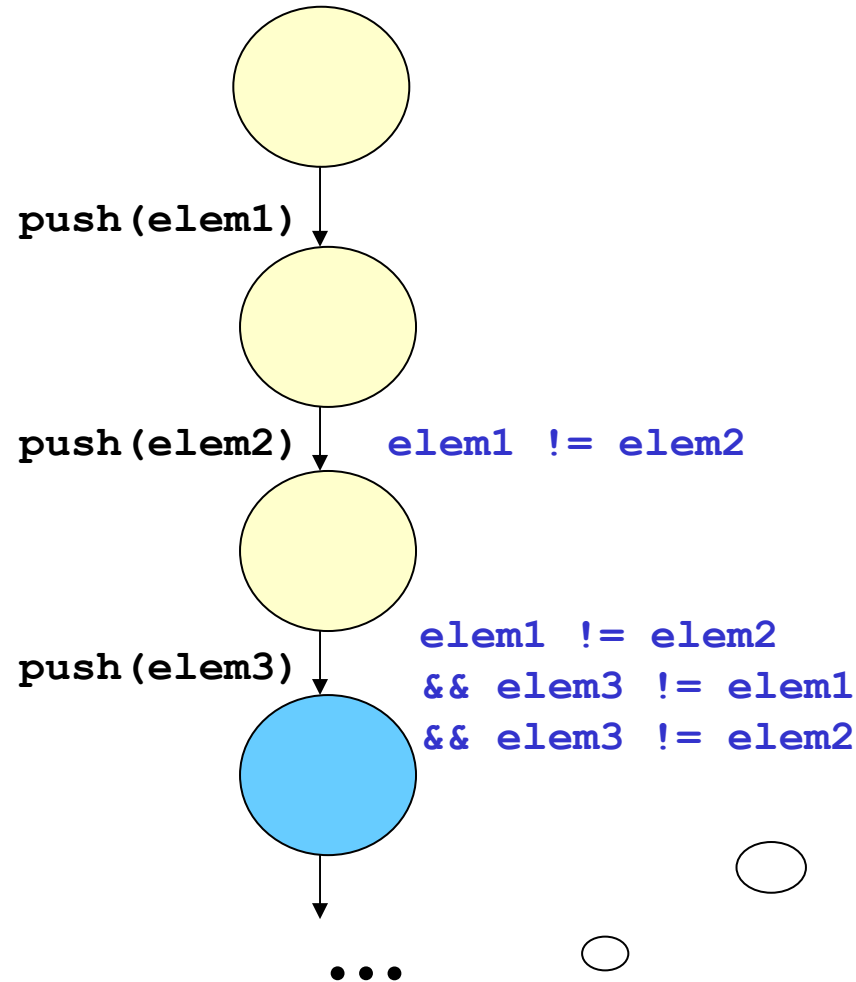


After n iterations, state space $O(3^n)$

If the stack is a unique int stack, we need 10 different arguments to reach size of 10, state space $9864100!$

Example – Symbolic Inputs

Inputs: `push(SymbolicInt svalue)`
after `new UIntStack()`



Sym Exec
O(10)!

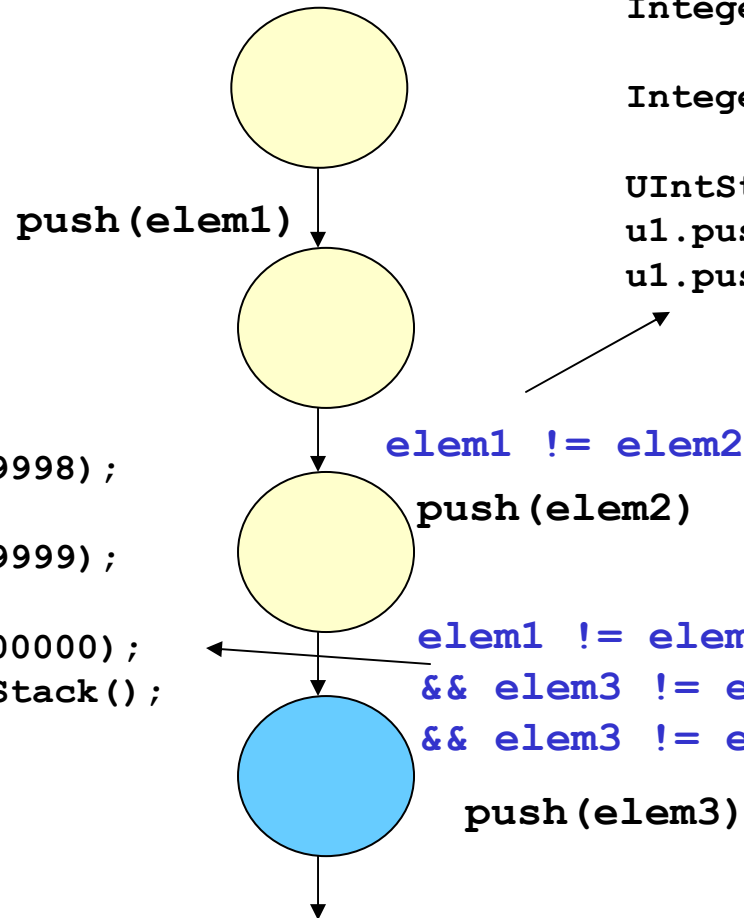
If the stack is a unique int stack, we need 10 different arguments to reach size of 10, state space 9864100!

From Symbolic to Concrete Inputs

Inputs: `push(SymbolicInt svalue)`

after `new UIntStack()`

```
Integer push_0_1 =  
    new Integer(-1000000);  
Integer push_0_2 =  
    new Integer(-999999);  
UIntStack u1 = new UIntStack();  
u1.push(push_0_1);  
u1.push(push_0_2);
```



```
Integer push_0_1 =  
    new Integer(-999998);  
Integer push_0_3 =  
    new Integer(-999999);  
Integer push_0_2 =  
    new Integer(-1000000);  
UIntStack u1 = new UIntStack();  
u1.push(push_0_1);  
u1.push(push_0_2);  
u1.push(push_0_3);
```

- Derive concrete arguments using a constraint solver [POOC]
- Export concrete tests to executable JUnit classes

Preliminary Experimental Results

Pentium IV 2.8 GHz Sun JDK 1.4.2 with 512MB allocated mem

<i>Class</i>	<i>After iteration #</i>	<i>State space (#nonequiv states)</i>		<i>Time (in seconds)</i>	
		<i>concrete</i>	<i>symexec</i>	<i>concrete</i>	<i>symexec</i>
UIntStack (push)	6	1957	7	24.32	0.69
	7	---after 6	8	---	0.91
	8	---	9	---	1.24
	9	---	10	---	1.71
	10	---	11	---	2.41
TreeMap (put, remove)	4	25	15	1.54	1.10
	5	72	41	4.66	3.40
	6	185	139	18.16	11.08
	7	537	531	66.58	50.62
	8	---after 6	2251	---	271.15 ₄₈

Discussion

- Automate code instrumentation for symbolic execution or symbolic interpretation
- Symbolic exec for non-primitive variables
 - SHE (Symbolic Heap Execution)
 - Build path conditions involving references
- Test prioritization: explore how different strategies for “on-the-fly” test generation lead to detecting faults?

Related Work

- State equivalence using observational equivalence [Bernot et al. 91, Doong&Frankl 94, Henkel&Diwan 03]
 - Expensive because of number of sequences
- State equivalence based on user-defined abstraction functions [Grieskamp et al. 02]
- Generalized symbolic execution [Khurshid et al. 03] and Java Pathfinder test generation [Visser et al. 04]
 - Symbolic execution requiring both `repOk()/precondition` and “conservative” `repOk()/precondition`
 - Concrete method sequence generation (slower than symbolic Rostra)

Conclusions

- Redundant tests add cost without any benefit
- Existing test generation tools can be potentially improved (by incorporating Rostra framework)
- Symbolic execution can further reduce the cost
- The experimental results have shown
 - High redundancy among their generated tests
 - Removing them does not decrease test suite quality
 - Symbolic execution enables generation of longer method sequences

Questions?

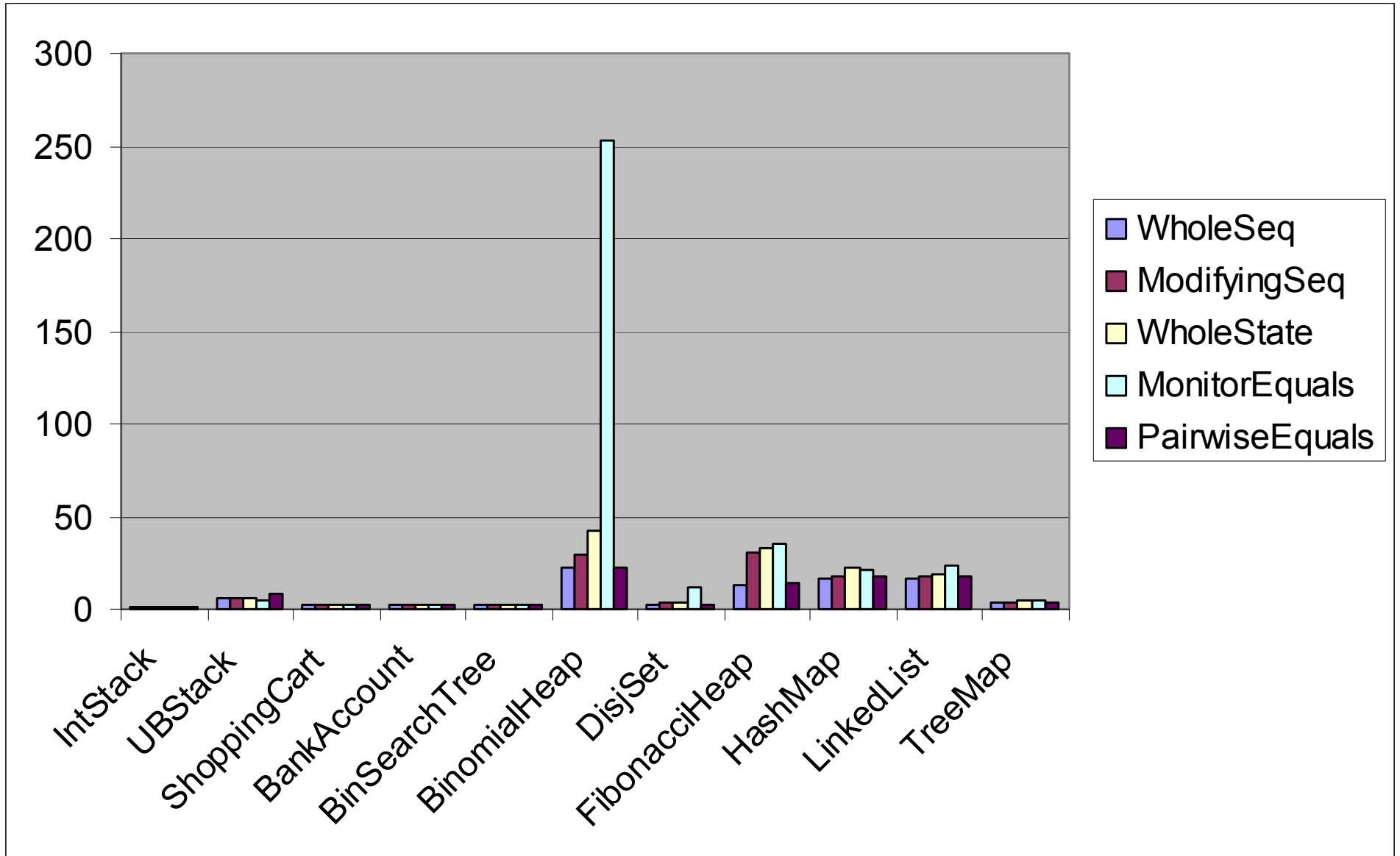
Applications

- **Assessment:** compare the quality of different test suites.
- **Selection:** select a subset of automatically generated tests to augment an existing test suite.
- **Minimization:** minimize an automatically generated test suite for correctness inspection and regression executions.
- **Generation:** avoid generating and executing redundant tests

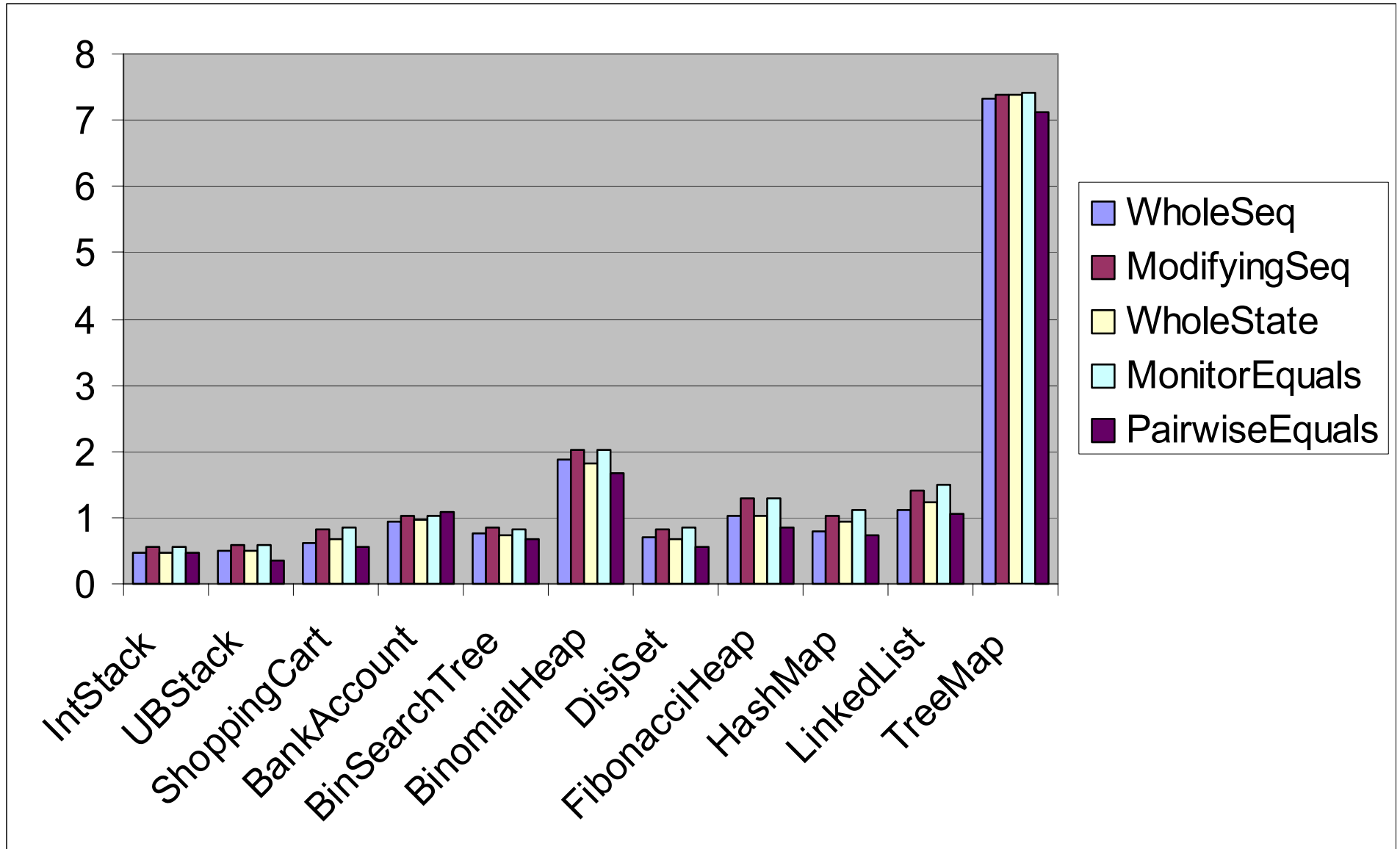
Syntactic Comparison

- Syntactic comparison (based on of PC string)
 - Remove irrelevant constraints (e.g. $\text{elem1} == \text{elem2}$)
 - Optimization during PC buildup
 - $c1$: a constraint in PC; $c2$: a constraint to be considered
 - if $c1 \Rightarrow c2$, we don't add $c2$ to PC (e.g. $x > 1 \Rightarrow x \geq 1$)
 - if $c2 \Rightarrow c1$, we remove $c1$ after we add $c2$ to PC

Elapsed Real Time in Minimizing Jtest-Generated Tests (in secs)



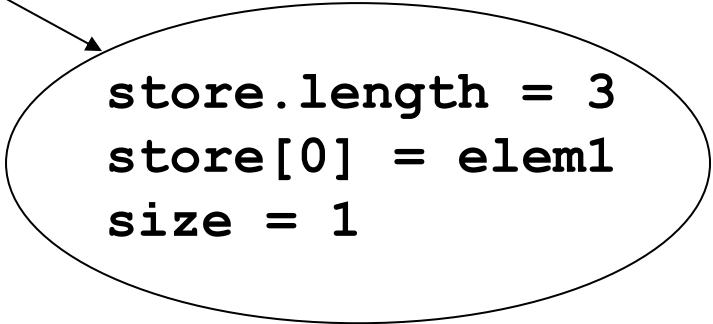
Elapsed Real Time in Minimizing JCrasher-Generated Tests (in secs)



Optimization Heuristics

- Without considering path conditions

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);
```



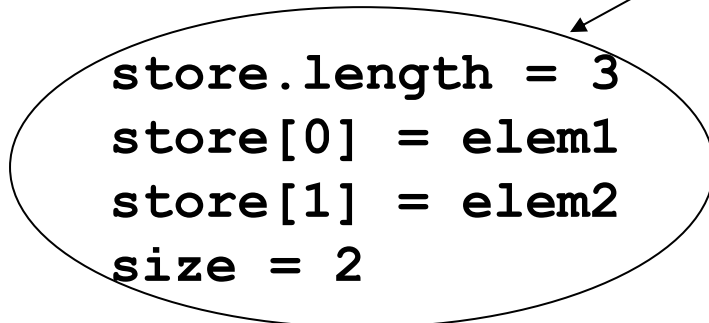
store.length = 3
store[0] = elem1
size = 1

Optimization Heuristics

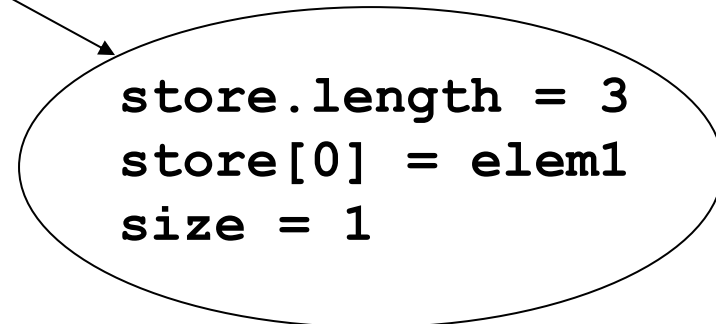
- Without considering path conditions

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);  
s1.push(elem2);
```

elem1 != elem2



elem1 == elem2



Method-Sequence Representation with Symbolic Execution

- Considering path conditions

```
SymbolicInt elem1 =  
    new SymbolicInt();  
SymbolicInt elem2 =  
    new SymbolicInt();  
IntStack s1 =  
    new IntStack();  
s1.push(elem1);  
s1.push(elem2);
```

push(<init>().state,
elem1).state

State 1

elem1 != elem2

push(
push(<init>().state,
elem1).state,
elem2).state

State 2

elem1 == elem2

push(
push(<init>().state,
elem1).state,
elem2).state

State 3