Search-Based Testing

Phil McMinn

University of Sheffield, UK
Overview

How and why Search-Based Testing Works

Examples

Temporal, Functional, Structural

Applications in Mutation Testing

Test suite prioritisation

Search-Based Test Data Generation

Testability Transformation

Input Domain Reduction

Empirical and Theoretical Studies

Future Directions
Acknowledgements

The material in some of these slides has kindly been provided by:

Mark Harman (KCL/UCL)
Joachim Wegener (Berner & Matner)
Conventional Testing

manual design of test cases / scenarios
Conventional Testing

manual design of test cases / scenarios

laborious,

time-consuming
Conventional Testing

manual design of test cases / scenarios

laborious,
tedious
time-consuming
Conventional Testing

- Manual design of test cases / scenarios
  - Tedious
  - Laborious, time-consuming
  - Difficult!
    (Where are the faults ???)
Conventional Testing

- Manual design of test cases / scenarios
  - Laborious, time-consuming
  - Tedious
  - Difficult!
    (where are the faults ???)
    The hard part...
Random Test Data Generation

Input
Random Test Data Generation
Random Test Data Generation
Search-Based Testing is an automated search of a potentially large input space. The search is guided by a problem-specific ‘fitness function’.
Search-Based Testing is an automated search of a potentially large input space.

The search is guided by a problem-specific ‘fitness function’.

The fitness function guides the search to the test goal.
Fitness Function

The fitness function scores different inputs to the system according to the test goal.
Fitness Function

The fitness function scores different inputs to the system according to the test goal which ones are ‘good’ (that we should develop/evolve further)
Fitness Function

The fitness function scores different inputs to the system according to the test goal.

- which ones are ‘good’
  (that we should develop/evolve further)

- which ones are useless
  (that we can forget about)
Fitness-guided search

Fitness

Input
Fitness-guided search

Fitness

Input
Fitness-guided search

Fitness

Input
First publication on SBST

Automatic Generation of Floating-Point Test Data
IEEE Transactions on Software Engineering, 1976

Webb Miller       David Spooner

Winner of the 2009 Accomplishment by a Senior Scientist Award
of the International Society for Computational Biology

2009 Time100
Scientists and Thinkers
Publications since 1976

source: SEBASE publications repository [http://www.sebase.org](http://www.sebase.org)
International Search-Based Testing Workshop

collocated with ICST

4th event at ICST 2011
next year

check websites for submission deadlines etc.:

http://sites.google.com/site/icst2011/

http://sites.google.com/site/icst2011workshops/
International Symposium on Search-Based Software Engineering

www.ssbse.org

Benevento, Italy.
7th - 9th September 2010
International Symposium on Search-Based Software Engineering

www.ssbse.org

2011: Co-location with FSE.
Szeged, Hungary

Phil McMinn
General Chair

Myra Cohen and Mel Ó Cinnéide
Program Chairs
Fitness Functions

Often easy

We often define metrics

Need not be complex
Daimler Temporal Testing

Fitness = duration

optimal point in time for triggering the airbag igniter

deceleration

$t_{\text{min}}$ $t_{\text{max}}$

$t$
Conventional testing

manual design of test cases / scenarios

laborious, tedious
time-consuming
difficult!
(where are the faults ???)
Conventional testing

manual design of test cases / scenarios

laborious, tedious, time-consuming
difficult! (where are the faults ???)

Search-Based Testing:

automatic - may sometimes be time consuming, but it is not a human’s time being consumed
Conventional testing

- manual design of test cases / scenarios
- laborious, tedious
- time-consuming
difficult!

Search-Based Testing:
- automatic - may sometimes be time consuming, but it is not a human’s time being consumed

Search-Based Testing:
- a good fitness function will lead the search to the faults

(Where are the faults???)
Generating vs Checking

Conventional Software Testing Research
Write a method to construct test cases

Search-Based Testing
Write a method to determine how good a test case is
Generating vs Checking

Conventional Software Testing Research

Write a method to construct test cases

Search-Based Testing

Write a fitness function to determine how good a test case is
Daimler Autonomous Parking System

Input

psi

gap

dist2space

space length

space width

dist2space
Daimler Autonomous Parking System

Fitness
Generation 0
Generation 0

Generation 10

critical

collision

Generation 20
Usual approach to testing

Manually generated input situations

Simulation environment

Software under test

Output of test
Test setup

Search-Based Testing approach

Automatically generated inputs

Search Algorithm → Simulation environment → Software under test

Feedback loop
outputs (converted to fitness values)
O. Bühler and J. Wegener: Evolutionary functional testing, Computers & Operations Research, 2008 - Elsevier

Structural testing
Structural testing

fitness function analyses the outcome of decision statements and the values of variables in predicates
Structural testing

fitness function analyses the outcome of decision statements and the values of variables in predicates

More later ...
Assertion testing
Assertion testing

assertion condition

\textit{speed} < 150\text{mph}
Assertion testing

assertion condition
speed < 150mph

fitness function:
\[ f = 150 - \text{speed} \]
Assertion testing

assertion condition
speed < 150mph

fitness function:
f = 150 - speed

fitness minimised
If f is zero or less a fault is found
Search Techniques
Hill Climbing

Fitness

Input
Hill Climbing

- No better solution in neighbourhood
- Stuck at a local optima
Hill Climbing - Restarts
Hill Climbing - Restarts

Fitness

Input
Hill Climbing - Restarts
Hill Climbing - Restarts

Fitness

Input
Hill Climbing - Restarts

Fitness

Input
Hill Climbing - Restarts

Fitness vs Input graph showing a peak and a restart point.
Hill Climbing - Restarts
Hill Climbing - Restarts
Simulated Annealing

Fitness

Input
Simulated Annealing
Simulated Annealing

Input

Fitness
Simulated Annealing

Fitness vs Input
Simulated Annealing

Fitness

Input
Simulated Annealing

Fitness

Input
Simulated Annealing

Fitness vs. Input

Worse solutions temporarily accepted
Evolutionary Algorithm

Fitness

Input
Evolutionary Algorithm

Fitness

Input
Evolutionary Algorithm
Evolutionary Algorithms

inspired by Darwinian Evolution and concept of survival of the fittest

Crossover

Mutation
Crossover

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>-5</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>
Crossover

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
Crossover

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
Crossover

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
Crossover

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>
Mutation

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
Mutation

```c
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
        }
    }
    ...
}
```
void test_me(int a, int b, int c, int d) {
    if (a == b) {
        if (c == d) {
            // branch we want to execute
            ...
        }
    }
    ...
}
Evolutionary Testing
Evolutionary Testing

- Mutation
- Crossover
- Selection
- Insertion
- Fitness Evaluation
- End?
Which search method?

Depends on characteristics of the search landscape
Which search method?

Depends on characteristics of the search landscape
Which search method?

Depends on characteristics of the search landscape
Which search method?

Depends on characteristics of the search landscape
Which search method?

Depends on characteristics of the search landscape
Which search method?

Some landscapes are hard for some searches but easy for others...

...and vice versa...
Which search method?

Some landscapes are hard for some searches but easy for others...

...and vice versa...
Which search method?

Some landscapes are hard for some searches but easy for others...

...and vice versa...
Which search method?

Some landscapes are hard for some searches but easy for others... and vice versa...

more on this later...
Ingredients for an optimising search algorithm

Representation
Neighbourhood
Fitness Function
Ingredients for Search-Based Testing

- Representation
- Neighbourhood
- Fitness Function
Ingredients for Search-Based Testing

**Representation**  A method of encoding all possible inputs

**Neighbourhood**  Usually straightforward

**Fitness Function**  Inputs are already in data structures
Ingredients for Search-Based Testing

**Representation**
Part of our understanding of the problem

**Neighbourhood**
We need to know our near neighbours

**Fitness Function**
Ingredients for Search-Based Testing

**Representation**  Transformation of the test goal to a numerical function

**Neighbourhood**  Numerical values indicate how ‘good’ an input is

**Fitness Function**
More search algorithms
More search algorithms

Tabu Search

Particle Swarm Optimisation

Ant Colony Optimisation

Genetic Programming

Estimation of Distribution Algorithms
SBST Surveys & Reviews


Getting started in SBSE


More applications of Search-Based Testing
Mutation Testing

```java
... if (a > 0 && b > 0) {
    return 1;
}
...
```

original
Mutation Testing

```
if (a > 0 && b > 0) {
    return 1;
}

if (a < 0 && b > 0) {
    return 1;
}
```
Mutation Testing

```
... if (a > 0 || b > 0) {
    return 1;
}
...
```

mutant

```
... if (a < 0 && b > 0) {
    return 1;
}
...
```

mutant
Mutation Testing

mutant

if (a > 0 || b > 0) {
    return 1;
}
...

higher order mutant

if (a < 0 || b > 0) {
    return 1;
}
...

mutant

if (a < 0 && b > 0) {
    return 1;
}
...
Mutation Testing

Fewer Equivalent Mutants

mutant

higher order mutant

mutant

Finding Good HOMs

Due to the large number of potential HOMs finding the ones that are most valuable is hard.

We want:

- Subtle mutants
- Reduced Test Effort
Finding Good HOMs

Due to the large number of potential HOMs finding the ones that are most valuable is hard.

We want:

- Subtle mutants
- Reduced Test Effort

HOMs that are hard to kill, corner cases where undiscovered faults reside.
Finding Good HOMs

Due to the large number of potential HOMs finding the ones that are most valuable is hard

We want:

- Subtle mutants
- Reduced Test Effort

HOMs that are hard to kill, corner cases where undiscovered faults reside

HOMs that subsume as many first-order mutants as possible
Subtle Mutants - Fitness Function

\[ \text{killability} = \frac{\# \text{ of test cases that kill the mutant}}{\text{total } \# \text{ of test cases}} \]

\[ f(M_1...n) = \frac{\text{killability}(M_1...n)}{\bigcup_i \text{killability}(M_i)} \]

- \( > 1 \) HOM is weaker than FOMs
- \( < 1 \) HOM is stronger than FOMs
- \( = 0 \), potential equivalent mutant

## Time aware test suite prioritisation

<table>
<thead>
<tr>
<th></th>
<th>$\phi_1$</th>
<th>$\phi_2$</th>
<th>$\phi_3$</th>
<th>$\phi_4$</th>
<th>$\phi_5$</th>
<th>$\phi_6$</th>
<th>$\phi_7$</th>
<th>$\phi_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$T_2$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>$T_3$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>$T_4$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>$T_5$</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>$T_6$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># Faults</th>
<th>Time Cost(Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>7</td>
<td>9</td>
<td>0.778</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>$T_3$</td>
<td>2</td>
<td>3</td>
<td>0.667</td>
</tr>
<tr>
<td>$T_4$</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_5$</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_6$</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Time aware test suite prioritisation

Suppose a 12 minute time budget

<table>
<thead>
<tr>
<th>Fault</th>
<th>Time Cost (Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>7</td>
<td>0.778</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>$T_3$</td>
<td>2</td>
<td>0.667</td>
</tr>
<tr>
<td>$T_4$</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_5$</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_6$</td>
<td>3</td>
<td>0.75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time Limit: 12 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault $(\sigma_1)$</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_3$</td>
</tr>
<tr>
<td>$T_4$</td>
</tr>
</tbody>
</table>

| Tot. Faults | 7 | 8 | 7 | 8 |
| Tot. Time | 9 | 12 | 10 | 11 |
Time aware test suite prioritisation

Order by considering the number of faults that can be detected

<table>
<thead>
<tr>
<th># Faults</th>
<th>Time Cost (Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>0.778</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.667</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Time Limit: 12 minutes

<table>
<thead>
<tr>
<th>Fault ($\sigma_1$)</th>
<th>Time ($\sigma_2$)</th>
<th>APFD ($\sigma_3$)</th>
<th>Intelligent ($\sigma_4$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_2$</td>
<td>$T_5$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$T_1$</td>
<td>$T_4$</td>
<td>$T_3$</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$T_5$</td>
<td>$T_3$</td>
<td></td>
</tr>
</tbody>
</table>

Tot. Faults: 7
Tot. Time: 9
Time aware test suite prioritisation

Order by considering the time only

<table>
<thead>
<tr>
<th></th>
<th>$\phi_1$</th>
<th>$\phi_2$</th>
<th>$\phi_3$</th>
<th>$\phi_4$</th>
<th>$\phi_5$</th>
<th>$\phi_6$</th>
<th>$\phi_7$</th>
<th>$\phi_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$T_2$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_4$</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>$T_5$</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$T_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Faults</th>
<th>Time Cost (Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>7</td>
<td>0.778</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>$T_3$</td>
<td>2</td>
<td>0.667</td>
</tr>
<tr>
<td>$T_4$</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_5$</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>$T_6$</td>
<td>3</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Time Limit: 12 minutes

<table>
<thead>
<tr>
<th>Fault $(\sigma_1)$</th>
<th>Time $(\sigma_2)$</th>
<th>APFD $(\sigma_3)$</th>
<th>Intelligent $(\sigma_4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_2$</td>
<td>$T_5$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$T_1$</td>
<td>$T_1$</td>
<td>$T_4$</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$T_3$</td>
<td></td>
<td>$T_3$</td>
</tr>
<tr>
<td>$T_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Tot. Faults | 7 | 8 | 7 | 8 |
| Tot. Time   | 9 | 12 | 10 | 11 |
Time aware test suite prioritisation

<table>
<thead>
<tr>
<th>$\phi_1$</th>
<th>$\phi_2$</th>
<th>$\phi_3$</th>
<th>$\phi_4$</th>
<th>$\phi_5$</th>
<th>$\phi_6$</th>
<th>$\phi_7$</th>
<th>$\phi_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T_2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>T_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>T_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Faults</th>
<th>Time Cost (Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>7</td>
<td>0.778</td>
</tr>
<tr>
<td>T_2</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>T_3</td>
<td>2</td>
<td>0.667</td>
</tr>
<tr>
<td>T_4</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>T_5</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>T_6</td>
<td>3</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Average % of faults detected

<table>
<thead>
<tr>
<th>Time Limit: 12 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault ($\sigma_1$)</td>
</tr>
<tr>
<td>T_1</td>
</tr>
<tr>
<td>T_3</td>
</tr>
<tr>
<td>Tot. Faults</td>
</tr>
<tr>
<td>Tot. Time</td>
</tr>
</tbody>
</table>
Time aware test suite prioritisation

<table>
<thead>
<tr>
<th></th>
<th>φ₁</th>
<th>φ₂</th>
<th>φ₃</th>
<th>φ₄</th>
<th>φ₅</th>
<th>φ₆</th>
<th>φ₇</th>
<th>φ₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T₂</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₄</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>T₅</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>T₆</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Faults</th>
<th>Time Cost (Mins)</th>
<th>Avg. Faults per Min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>7</td>
<td>0.778</td>
</tr>
<tr>
<td>T₂</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>T₃</td>
<td>2</td>
<td>0.667</td>
</tr>
<tr>
<td>T₄</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>T₅</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>T₆</td>
<td>3</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Intelligent heuristic search

<table>
<thead>
<tr>
<th>Time Limit: 12 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault (σ₁)</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>T₁</td>
</tr>
<tr>
<td>T₃</td>
</tr>
<tr>
<td>T₄</td>
</tr>
<tr>
<td>T₅</td>
</tr>
</tbody>
</table>

| Tot. Faults | 7 | 8 | 7 | 8 |
| Tot. Time   | 9 | 12| 10| 11|
Time aware test suite prioritisation

Intelligent heuristic search

Same no. of faults found, less time
Fitness Function

The tester is unlikely to know the location of faults
Need to estimate how likely a test is to find defects

% of code coverage is used to estimate a suite’s potential

\[ \times \]

the time taken to execute the test suite
Results

Mutations used to seed faults
Multi-objective Search

Instead of combining the objectives into one fitness function, handle them as distinctive goals.

| Time | Faults |
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B
Multi-objective Search

Fitness Function A

Fitness Function B

The Pareto Front
Some Results

Three objectives
Other applications of Search-Based Testing

Combinatorial Interaction Testing

GUI Testing
Other applications of Search-Based Testing

Stress testing


State machine testing

Search-Based Structural Test Data Generation
Covering a structure
Fitness evaluation
Fitness evaluation

The test data executes the ‘wrong’ path
Analysing control flow
Analysing control flow

The outcomes at key decision statements matter.

These are the decisions on which the target is control dependent.
Approach Level

TARGET = 2
Approach Level

TARGET = 2

TARGET = 1
Approach Level

TARGET = 2

TARGET = 1

TARGET = 0
Approach Level

TARGET

minimisation

= 2
= 1
= 0
Approach Level


Analysing predicates

Approach level alone gives us coarse values

```java
if (a == b) {
    // ....
}
```

- a = 50, b = 0
- a = 45, b = 5
- a = 40, b = 10
- a = 35, b = 15
- a = 30, b = 20
- a = 25, b = 25
Analysing predicates

Approach level alone gives us coarse values

\[
\text{if } (a == b) \{ \\
\quad \text{// ....} \\
\}
\]

- \(a = 50, b = 0\)
- \(a = 45, b = 5\)
- \(a = 40, b = 10\)
- \(a = 35, b = 15\)
- \(a = 30, b = 20\)
- \(a = 25, b = 25\)

going ‘closer’
to being true
Branch distance

Associate a distance formula with different relational predicates

```java
if (a == b) {
    // ....
}
```

getting ‘closer’ to being true

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>branch distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>45</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>35</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
</tbody>
</table>
Branch distances for relational predicates

<table>
<thead>
<tr>
<th>Relational predicate</th>
<th>Objective function $\textit{obj}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>if $\text{TRUE}$ then 0 else $K$</td>
</tr>
<tr>
<td>$a = b$</td>
<td>if $\text{abs}(a - b) = 0$ then 0 else $\text{abs}(a - b) + K$</td>
</tr>
<tr>
<td>$a \neq b$</td>
<td>if $\text{abs}(a - b) \neq 0$ then 0 else $K$</td>
</tr>
<tr>
<td>$a &lt; b$</td>
<td>if $a - b &lt; 0$ then 0 else $(a - b) + K$</td>
</tr>
<tr>
<td>$a \leq b$</td>
<td>if $a - b \leq 0$ then 0 else $(a - b) + K$</td>
</tr>
<tr>
<td>$a &gt; b$</td>
<td>if $b - a &lt; 0$ then 0 else $(b - a) + K$</td>
</tr>
<tr>
<td>$a \geq b$</td>
<td>if $b - a \leq 0$ then 0 else $(b - a) + K$</td>
</tr>
<tr>
<td>$\neg a$</td>
<td>Negation is moved inwards and propagated over $a$</td>
</tr>
</tbody>
</table>


Putting it all together

Fitness = approach Level + normalised branch distance

```c
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
            }
        }
    }
    ...
```
Putting it all together

Fitness = approach Level + normalised branch distance

```c
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
            }
        }
    }
    ...
```

TARGET MISSED
Approach Level = 2
Branch Distance = b - a

normalised branch distance between 0 and 1
indicates how close approach level is to being penetrated
Putting it all together

Fitness = approach Level + normalised branch distance

```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
            }
        }
    }
    ...
```

normalised branch distance between 0 and 1
indicates how close approach level is to being penetrated

TARGET MISSED
Approach Level = 2
Branch Distance = b - a

TARGET MISSED
Approach Level = 1
Branch Distance = c - b
Putting it all together

Fitness = approach Level + *normalised* branch distance

```c
void f1(int a, int b, int c, int d)
{
    if (a >= b)
    {
        if (b >= c)
        {
            if (c >= d)
            {
                // target
            }
        }
    }
    ...
}
```

normalised branch distance between 0 and 1
indicates how close approach level is to being penetrated
Normalisation Functions

Since the ‘maximum’ branch distance is generally unknown we need a non-standard normalisation function

\[ 1 - \alpha^{-x} \]

Baresel (2000), alpha = 1.001
Normalisation Functions

Since the ‘maximum’ branch distance is generally unknown we need a non-standard normalisation function

\[ \frac{x}{x + \beta} \]

Arcuri (2010), beta = 1
Alternating Variable Method

‘Probe’ moves

void fn(input1, input2, input3, ....)
Alternating Variable Method

‘Probe’ moves

```c
void fn(input1, input2, input3, ...)
```
Alternating Variable Method

‘Probe’ moves

```
void fn(input1, input2, input3, ....)
```

Arrows: decrease, increase, decrease, increase
Alternating Variable Method

‘Probe’ moves

```c
void fn(input1, input2, input3, ...)
```
Alternating Variable Method

Accelerated hill climb

Fitness

Input variable value
Alternating Variable Method

Accelerated hill climb

Fitness

Input variable value
Alternating Variable Method

Accelerated hill climb
Alternating Variable Method

Accelerated hill climb

Fitness vs. Input variable value
Alternating Variable Method

Accelerated hill climb
Alternating Variable Method

Accelerated hill climb

Fitness

Input variable value
Alternating Variable Method

Accelerated hill climb

Fitness vs. Input variable value
Alternating Variable Method

1. Randomly generate start point
   
   $a=10$, $b=20$, $c=30$

```java
void example(int a, int b, ...) {
    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```
Alternating Variable Method

1. Randomly generate start point
   \[ a=10, b=20, c=30 \]

2. ‘Probe’ moves on \( a \)
   \[ a=9, b=20, c=30 \]
   \[ a=11, b=20, c=30 \]
   no effect

```java
void example(int a, int b, ...) {
    if (a == 0) {
        ...
    }
    if (b == 0) {
        // target
    }
    ...
}
```
Alternating Variable Method

1. Randomly generate start point
   \[ a=10, \ b=20, \ c=30 \]

2. ‘Probe’ moves on \( a \)
   \[ a=9, \ b=20, \ c=30 \]
   no effect
   \[ a=11, \ b=20, \ c=30 \]

3. ‘Probe’ moves on \( b \)
   \[ a=10, \ b=19, \ c=30 \]
   improved branch distance

```cpp
void example(int a, int b, ...) {
    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }
    ...
}
```
Alternating Variable Method

1. Randomly generate start point
   \[ a=10, b=20, c=30 \]

2. ‘Probe’ moves on \( a \)
   \[ a=9, b=20, c=30 \] no effect
   \[ a=11, b=20, c=30 \]

3. ‘Probe’ moves on \( b \)
   \[ a=10, b=19, c=30 \] improved branch distance

4. Accelerated moves in direction of improvement

\[ \text{void example(int a, int b, ...)} \{
  \text{if (a == 0)} \{
    \ldots
  \}
  \}
  \text{if (b == 0)} \{
    \text{// target}
  \}
  \ldots
\} \]
Key Publications

Alternating variable method


Evolutionary structural test data generation

A search-based test data generator tool
Input Generation Using Automated Novel Algorithms
IGUANA (Java)

Test object
(C code compiled to a DLL)
IGUANA

(Java)

Test object

(C code compiled to a DLL)

inputs
IGUANA
(Java)

Test object
(C code compiled to a DLL)

inputs

fitness computation

information from test object instrumentation
IGUANA
(Java)

search algorithm

fitness computation

Test object
(C code compiled to a DLL)

inputs

information from test object instrumentation
IGUANA (Java)

search algorithm

fitness computation

Java Native Interface

inputs

information from test object instrumentation

Test object
(C code compiled to a DLL)
A function for testing

```c
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```
Test Object Preparation

1. Parse the code and extract control dependency graph

```c
int test_me(int a, int b, int c) {
    int flag = 0;
    if (a == b) {
        if (b == c) {
            return 1;
        }
    }
    if (a == 0) {
        flag = 1;
    }
    if (flag && b == 0) {
        return 2;
    }
    return -1;
}
```

“which decisions are key for the execution of individual structural targets”? 
Test Object Preparation

2. Instrument the code

```c
int test_me(int a, int b, int c) {
    int flag = 0;
    if (a == b) {
        if (b == c) {
            return 1;
        }
    }
    if (a == 0) {
        flag = 1;
    }
    if (flag && b == 0) {
        return 2;
    }
    return -1;
}
```
Test Object Preparation

3. Map inputs to a vector

```c
int test_me(int a,
            int b,
            int c)
{
    ...
}
```
Test Object Preparation

3. Map inputs to a vector

```c
int test_me(int a,
            int b,
            int c)
{
    ...
}
```

Straightforward in many cases
Test Object Preparation

3. Map inputs to a vector

```plaintext
int test_me(int a,
            int b,
            int c)
{
    ...
}
```

Straightforward in many cases

Inputs composed of dynamic data structures are harder to compose

Instrumentation

```c
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```
Instrumentation

```c
int test_me(int a, int b, int c) {
    int flag = 0;
    if (a == b) {
        if (b == c) {
            return 1;
        }
    }
    if (a == 0) {
        flag = 1;
    }
    if (flag && b == 0) {
        return 2;
    }
    return -1;
}
```

```c
int test_me(int a, int b, int c) {
    int flag = 0;
    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }
    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }
    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }
    return -1;
}
```
Each branching condition is replaced by a call to the function `node(...)`. The instrumentation should only observe the program and not alter its behaviour.
The first parameter is the control flow graph node ID of the decision statement.

The second parameter is a boolean condition that replicates the structure in the original program (i.e. including short-circuiting).
Relational predicates are replaced with functions that compute branch distance.
The instrumentation tells us:

Which decision nodes were executed

and their outcome (branch distances)

Therefore we can find which decision control flow diverged from a target for an input....

... and compute the approach level from the control dependence graph

... and lookup the branch distance

fitness value for an input
Input: <20, 20, 30>

```c
int test_me(int a, int b, int c)
{
    int flag = 0;
    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }
    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }
    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }
    return -1;
}
```

Diverged at node 2
approach level: 0
branch distance: 10
fitness = 0.009945219
Testability Transformation
The ‘Flag’ Problem

```c
void testme(int a, int b, ....)
{
    ...
    flag = (a == 0 && b == 0);
    if (flag) {
        ....
    }
}
```
void testme(int a, int b, ...) 
{
    ...
    flag = (a == 0 && b == 0);
    if (flag) {
        ...
    }
}
Program Transformation

```c
void testme(int a, int b, ....) {
    
    flag = (a == 0 && b == 0);

    if (flag) {
        
    }
}

void testme(int a, int b, ....) {
    
    flag = (a == 0 && b == 0);

    if (a == 0 && b == 0) {
        
    }
}
```
Programs will inevitably have features that heuristic searches handle less well.

Programs will inevitably have features that heuristic searches handle less well

Testability transformation:
change the program to improve test data generation

Programs will inevitably have features that heuristic searches handle less well.

Testability transformation:
change the program to improve test data generation
... whilst preserving test adequacy

Nesting

```c
void testme(int a, int b, ...) {
    if (a == 0) {
        if (b == 0) {
            ...
        }
    }
}
```
Testability Transformation

void testme(int a, int b, ...)
{
    if (a == 0) {
        if (b == 0) {
            // target
        }
    }
    ...
}

void tt_testme(int a, int b, ...)
{
    double _distance = 0;
    _distance += distance(a == 0); // if (a == 0)
    _distance += distance(b == 0); // if (b == 0)
    // target
    ...
}
Testability Transformation

void testme(int a, int b, ...)
{
    if (a == 0) {
        if (b == 0) {
            // target
        }
    }
    ...
}

void tt_testme(int a, int b, ...)
{
    double _distance = 0;
    _distance += distance(a == 0); // if (a == 0)
    _distance += distance(b == 0); // if (b == 0)
    // target
    ...
}

Note that the programs are no longer equivalent
Testability Transformation

void testme(int a, int b, ...)
{
    if (a == 0) {
        if (b == 0) {
            // target
        }
    }
    ...
}

void tt_testme(int a, int b, ...)
{
    double _distance = 0;
    _distance += distance(a == 0); // if (a == 0)
    _distance += distance(b == 0); // if (b == 0)
    // target
    ...
}

Note that the programs are no longer equivalent
But we don’t care - so long as we get the test data is still adequate
Nesting & Local Optima

```c
void local_optima(double x, double y)
{
    if (x == y)
    {
        if (x == 0)
        {
            // target
        }
    }
}
```
Nesting & Local Optima

```c
void local_optima(double x, double y)
{
    if (x == y)
    {
        if (x == 0)
        {
            // target
        }
    }
}
```
Nesting & Local Optima

```c
void local_optima(double x, double y)
{
    if (x == y)
    {
        if (x == 0)
        {
            // target
        }
    }
}
```

```c
void tt_local_optima(double x, double y)
{
    double _distance = 0;

    _distance += distance(x == y); // if (x == y)

    _distance += distance(x == 0); // if (x == 0)

    // target
}
```
Nesting & Local Optima

```c
void local_optima(double x, double y)
{
    if (x == y)
    {
        if (x == 0)
        {
            // target
        }
    }
}
```

```c
void tt_local_optima(double x, double y)
{
    double _distance = 0;
    _distance += distance(x == y); // if (x == y)
    _distance += distance(x == 0); // if (x == 0)
    // target
}
```
Results - Industrial & Open source code

Change in success rate after applying transformation (%) vs Nested branches
Dependent & Independent Predicates

Independent Predicates influenced by disjoint sets of input variables Can be optimised in parallel

e.g. ‘a == 0’ and ‘b == 0’
Dependent & Independent Predicates

Dependent Predicates influenced by non-disjoint sets of input variables

Interactions between predicates inhibit parallel optimisation

e.g. ‘a == b’ and ‘b == c’
Dependent predicates
Nested branches

Independent and some dependent predicates

Change in success rate after applying transformation (%)

Nested branches

Dependent predicates
if (div != 0)
{
    ans = i / div;
}

_distance += distance(div != 0);  // if (div != 0)

ans = i / div;
...

When not preserving program equivalence can go wrong

```java
if (div != 0) {
    ans = i / div;
}

_distance += distance(div != 0); // if (div != 0)
ans = i / div;
...
```
we are testing to cover structure
... but the structure is the problem
so we transform the program
... but this alters the structure
we are testing to cover structure
... but the structure is the problem
so we transform the program
... but this alters the structure
we are testing to cover structure
... but the structure is the problem
so we transform the program
... but this alters the structure

so we need to be careful: are we still testing according to the same criterion?
Input Domain Reduction
```c
void super-unsize-me(int irrelevant1,
                      int irrelevant2,
                      int im-ur-man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }

    if (im-ur-man == 0) {
        // target
    }
}
```
```c
void super-unsize-me(int irrelevant1,
                     int irrelevant2,
                     int im-ur-man)
{
    if (irrelevant1 == 0) { ... }    
    if (irrelevant2 == 0) { ... }    
    if (im-ur-man == 0) {
        // target
    }
}
```
```c
void super_unsize_me(int irrelevant1,
                     int irrelevant2,
                     int im_ur_man)
{
    if (irrelevant1 == 0) { ... }  
    if (irrelevant2 == 0) { ... }  

    if (im_ur_man == 0) {
        // target
    }
}
```
Effect of Reduction

```c
void super-unsize-me(int irrelevant1,
                     int irrelevant2,
                     int im-ur-man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }
    if (im-ur-man == 0) {
        // target
    }
}
```

-100,000 ... 100,000
-100,000 ... 100,000
-100,000 ... 100,000

approx $10^{16}$
Effect of Reduction

```c
void super-unsize-me(int irrelevant1,
                     int irrelevant2,
                     int im-ur-man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }

    if (im-ur-man == 0) {
        // target
    }
}
```
Effect of Reduction

```c
void super-unsize-me(int irrelevant1,
    int irrelevant2,
    int im-ur-man)
{
    if (irrelevant1 == 0) { ... }  
    if (irrelevant2 == 0) { ... }  
    if (im-ur-man == 0) {
        // target
    }
}
```
Effect of Reduction

```c
void super-unsize-me(int irrelevant1,
int irrelevant2,
int im-ur-man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }
    if (im-ur-man == 0) {
        // target
    }
}
```
Variable Dependency Analysis

```c
void vada(int a, int b, int c)
{
    if (a == 0) {
        b = a;
    }

    if (b == 0) {
        // target
    }
}
```
Variable Dependency Analysis

```c
void vada(int a, int b, int c)
{
    if (a == 0) {
        b = a;
    }

    if (b == 0) {
        // target
    }
}
```
Variable Dependency Analysis

```c
void vada(int a, int b, int c)
{
    if (a == 0) {
        b = a;
    }

    if (b == 0) {
        // target
    }
}
```
void vada(int a, int b, int c)
{
    if (a == 0) {
        b = a;
    }

    if (b == 0) {
        // target
    }
}
Empirical Study

Studied the effects of reduction with:

- Random Search
- Alternating Variable Method
- Evolutionary Testing

Case studies:

- Defroster
- F2
- Gimp
- Spice
- Tiff
Effect on Random Testing

```c
void super_unsize_me(int irrelevant1,
int irrelevant2,
int im_ur_man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }
    if (im_ur_man == 0) {
        // target
    }
}
```

Probability of executing target:

\[
\frac{100 \times 100 \times 1}{100 \times 100 \times 100}
\]
Effect on Random Testing

```c
void super-unsize-me(int irrelevant1,
                     int irrelevant2,
                     int im-ur-man)
{
    if (irrelevant1 == 0) { ... }
    if (irrelevant2 == 0) { ... }
    if (im-ur-man == 0) {
        // target
    }
}
```

Probability of executing target:

\[
\frac{100 \times 100 \times 100}{100 \times 100 \times 1} = 100
\]
Results with Random Testing

![Graph showing average improvement vs. irrelevant input variables (percentage of input vector)]
Results with AVM
Effect on AVM

Saves probe moves (and thus wasteful fitness evaluations) around irrelevant variables

```c
void fn( irrelevant1, irrelevant2, irrelevant3, required1 .... )
```
Effect on AVM

Saves probe moves (and thus wasteful fitness evaluations) around irrelevant variables

```c
void fn( irrelevant1, irrelevant2, irrelevant3, required1 .... )
```
Effect on ET

Saves mutations on irrelevant variables

Mutations concentrated on the variables that *matter*

Likely to speed up the search
Results with ET

![Graph showing average improvement against irrelevant input variables as a percentage of the input vector.](image-url)
Conclusions for Input Domain Reduction

Variable dependency analysis can be used to reduce input domains.

This can reduce search effort for the AVM and ET.

Perhaps surprisingly, there is no overall change for random search.
Which search algorithm?
Empirical Study

Bibclean
Defroster
F2
Eurocheck
Gimp
Space
Spice
Tiff
Totinfo

760 branches in
~5 kLOC

Mark Harman and Phil McMinn.
A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search.
Interesting branches

- Alternating Variable Method: 8
- Evolutionary Testing: 9
-交集: 20
Wins for the AVM
Wins for the AVM

Average number of fitness evaluations

- *Evolutionary Testing*
- *Hill Climbing*

- **gimp_rgb_to_hsl (4T)**
- **gimp_rgb_to_hsv (5F)**
- **gimp_rgb_to_hsv4 (11F)**
- **gradient_calc_bilinear_factor (6T)**
- **gradient_calc_conical_asym_factor (3F)**
- **gradient_calc_conical_sym_factor (3F)**
- **gradient_calc_spiral_factor (3F)**
- **cliparc (13F)**
- **cliparc (15T)**
- **cliparc (15F)**
- **TIFF_SetSample (5T)**
When does the AVM win?
Wins for Evolutionary Testing

Success Rate

Evolutionary Testing

Hill Climbing
When does ET win?

The branches in question were part of a routine for validating ISBN/ISSN strings.

When a valid character is found, a counter variable is incremented.
When does ET win?

Evolutionary algorithms incorporate a population of candidate solutions through crossover.

Crossover enables valid characters to be crossed over into different ISBN/ISSN strings.
Schemata

Subsets of useful genes

- e.g. substrings of 1's in a binary all ones problem

```
1010100011110000111010
1111101010000000101011
00010010000011101011
000100101000011101011
```
The schema theory predicts that schema of above average fitness will proliferate in subsequent generations of the evolutionary search. Subsets of useful genes, e.g., substrings of 1’s in a binary all ones problem.
crossover of fit schemata can lead to even fitter, higher order schemata
Royal Roads

landscape structures where there is a ‘red carpet’ for crossovers to follow
The Genetic Algorithm
Royal Road

S1:  1111**************************************************************************
S2:  ****1111**************************************************************************
S3:  ********1111**************************************************************************
S4:  ************1111**************************************************************************
S5:  ****************1111**************************************************************************
S6:  *****************1111**************************************************************************
S7:  ************************1111******
S8:  11111111**************************************************************************
S9:  1111111111111111**************************************************************************
S10: 1111111111111111111111111111111111111111
S11: 1111111111111111111111111111111111111111
S12: 1111111111111111111111111111111111111111
S13: 1111111111111111111111111111111111111111
S14: 1111111111111111111111111111111111111111
S15: 1111111111111111111111111111111111111111
When Crossover Helps
When Crossover Helps

Executes the target

\[ \begin{array}{c}
\text{R1} \\
\text{Q1} \\
\text{P1} \quad \text{P2} \\
\text{Q2} \\
\text{P3} \quad \text{P4} \\
\text{R2} \\
\text{Q1} \\
\text{P1} \\
\text{Q2} \\
\text{P3} \quad \text{P4} \\
\end{array} \]
When Crossover Helps
When Crossover Helps

Executes the target

Contains 4 valid characters

Contains 2 valid characters

Contains 1 valid character

Contains 4 valid characters

Contains 2 valid characters

Contains 1 valid character

Contains 2 valid characters

Contains 1 valid character

Contains 2 valid characters

Contains 1 valid character

Contains 2 valid characters

Contains 1 valid character

Contains 2 valid characters

Contains 1 valid character

Contains 2 valid characters

Contains 1 valid character
Headless Chicken Test

Investigations into Crossover

Royal Roads


HIFF


Real Royal Roads


Ignoble Trails

Evolutionary Testing
Schemata

```c
void example(int a, int b, int c) {
    int count = 0;

    if (a == b) count ++;
    if (b >= 100) count ++;
    if (c <= 10) count ++;

    if (count == 3) {
        // target branch
    }

    ...
}
```

\{ (a, b, c) | a = b \}
\{(50, 50, 25) \}
\{(100, 100, 10) \}
...

\{ (a, b, c) | a > 0 \}
\{(50, 10, 25) \}
\{(100, -50, 10) \}
...
Crossover of good schemata

\{(a, b, c) \mid a = b\} \quad \{(a, b, c) \mid b \geq 100\}

\{(a, b, c) \mid a = b \land b \geq 100\}

```c
void example(int a, int b, int c) {
    int count = 0;
    if (a == b) count ++;
    if (b >= 100) count ++;
    if (c <= 10) count ++;
    if (count == 3) {
        // target branch
    ...
```
Crossover of good schemata

\[
\{(a, b, c) \mid a = b\} \quad \{(a, b, c) \mid b \geq 100\}
\]

\[
\{(a, b, c) \mid a = b \land b \geq 100\}
\]

void example(int a, int b, int c) {
    int count = 0;
    if (a == b) count ++;
    if (b >= 100) count ++;
    if (c <= 10) count ++;
    if (count == 3) {
        // target branch
        ...
    }
}
Crossover of good schemata

\{(a, b, c) \mid a = b\} \quad \{(a, b, c) \mid b \geq 100\} \\
\{(a, b, c) \mid a = b \land b \geq 100\}

building block
subschema

building block
subschema

void example(int a, int b, int c) {
  int count = 0;
  if (a == b) count ++;
  if (b >= 100) count ++;
  if (c <= 10) count ++;
  if (count == 3) {
    // target branch
    ...
  }
}
Crossover of good schemata

\{(a, b, c) \mid a = b \land b \geq 100\} \quad \text{and} \quad \{(a, b, c) \mid c \leq 10\}

\{(a, b, c) \mid a = b \land b \geq 100 \land c \leq 10\}

```c
void example(int a, int b, int c) {
    int count = 0;
    if (a == b) count ++;
    if (b >= 100) count ++;
    if (c <= 10) count ++;
    if (count == 3) {
        // target branch
        ...
    }
}
```
Crossover of good schemata

\{(a, b, c) \mid a = b \land b \geq 100\} \quad \{(a, b, c) \mid c \leq 10\}

\{(a, b, c) \mid a = b \land b \geq 100 \land c \leq 10\}

void example(int a, int b, int c) {
    int count = 0;
    if (a == b) count ++;
    if (b >= 100) count ++;
    if (c <= 10) count ++;
    if (count == 3) {
        // target branch
        ...
    }
}

covering schema
What types of program and program structure enable Evolutionary Testing to perform well, through crossover and how?

1. Large numbers of conjuncts in the input condition

\[
\{(a, b, c...) \mid a = b \land b \geq 100 \land c \leq 10 \ldots\}
\]

each represents a ‘sub’-test data generation problem that can be solved independently and combined with other partial solutions

What types of program and program structure enable Evolutionary Testing to perform well, through crossover and how?

2. Conjuncts should reference disjoint sets of variables

\{(a, b, c, d ... | a = b \land b = c \land c = d ... \}

the solution of each conjunct independently does not necessarily result in an overall solution.
Progressive Landscape

![3D landscape diagram with axes labeled 'fitness', 'input', 'input']
Crossover - Conclusions

1. Large numbers of conjuncts in the input condition
2. Conjuncts should reference disjoint sets of variables

Crossover lends itself to programs/units that process large data structures (e.g. strings, arrays) resulting in input condition conjuncts with disjoint variables
Crossover - Conclusions

1. Large numbers of conjuncts in the input condition
2. Conjuncts should reference disjoint sets of variables

Crossover lends itself to programs/units that process large data structures (e.g. strings, arrays) resulting in input condition conjuncts with disjoint variables

... or units that require large sequences of method calls to move an object into a required state
Crossover - Conclusions

1. Large numbers of conjuncts in the input condition
2. Conjuncts should reference disjoint sets of variables

Crossover lends itself to programs/units that process large data structures (e.g. strings, arrays) resulting in input condition conjuncts with disjoint variables

... or units that require large sequences of method calls to move an object into a required state

E.g. testing for a full stack - push(...), push(...), push(...)
Other Theoretical Work


Future directions...
The Oracle Problem

Determining the correct output for a given input is called the oracle problem.

Software engineering research has devoted much attention to automated oracles.

... but many systems do not have automated oracles.
Typically the responsibility falls on the human

Reducing human effort, the **human oracle cost** remains an important problem
Test data generation and human oracle cost

Quantitative

generate test data that maximises coverage but minimises the number of test cases

reduce size of test cases
Test data generation and human oracle cost

Quantitative

generate test data that maximises coverage but minimises the number of test cases


reduce size of test cases

Test data generation and human oracle cost

Qualitative e.g. how easily can the scenario comprising a test case be understood so that the output can be evaluated?
Test data generation and human oracle cost

Qualitative

E.g. how easily can the scenario comprising a test case be understood so that the output can be evaluated?
Test data generation and human oracle cost

Qualitative

e.g. how easily can the scenario comprising generated test data be understood so that the output can be evaluated?

Phil McMinn, Mark Stevenson and Mark Harman.
Calendar program

Takes two dates
(represented by 3 integers each)

Finds the number of days between the dates
Calendar program

Some example dates generated:

-4048/-10854/-29141
10430/3140/6733
3063/31358/8201

Machine-generated test data tends to not fit the operational profile of a program particularly well
Seeding knowledge

Programmer test case used as the starting point of the search process

16/1/2010
2/1/2009
2/32/2010
In what ways can operational profile knowledge be obtained?

Programmers

The Program
In what ways can operational profile knowledge be obtained?

Programmers

Likely to have run the program with a sanity check

The Program

These can be seeded to bias the test data generation process
In what ways can operational profile knowledge be obtained?

- Programmers
- The Program
- Identifier names
- Code comments
- Sanitisation routines

```c
int days_between(int start_month, int start_day, int start_year,
                 int end_month, int end_day, int end_year)
{
    int days = 0;

    // sanitize month inputs
    if (start_month < 1) start_month = 1;
    if (end_month < 1) end_month = 1;
    if (start_month > 12) start_month = 12;
    if (end_month > 12) end_month = 12;

    // sanitize day inputs
    if (start_day < 1) start_day = 1;
    if (end_day < 1) end_day = 1;
```
Identifier names

Give clues to the sort of inputs that might be expected

dayOfTheWeek    url

country_of_origin

Can be analysed in conjunction with large-scale natural language lexicons such as WordNet
Sanitisation Routines

```c
int days_between(int start_month, int start_day, int start_year,
                 int end_month, int end_day, int end_year)
{
    int days = 0;

    // sanitize month inputs
    if (start_month < 1) start_month = 1;
    if (end_month < 1) end_month = 1;
    if (start_month > 12) start_month = 12;
    if (end_month > 12) end_month = 12;

    // sanitize day inputs
    if (start_day < 1) start_day = 1;
    if (end_day < 1) end_day = 1;
    if (start_day > month_days(start_month, start_year))
        start_day = month_days(start_month, start_year);
    if (end_day > month_days(end_month, end_year))
        end_day = month_days(end_month, end_year);

    // swap dates if start date before end date
    if (((end_year < start_year) ||
         (end_year == start_year && end_month < start_month)) ||
         (end_year == start_year && end_month == start_month &&
          end_day < start_day)) {
        int t = end_month; end_month = start_month; start_month = t;
        t = end_day; end_day = start_day; start_day = t;
        t = end_year; end_year = start_year; start_year = t;
    }
}```

Defensive programming routines might be used to ‘correct’ a program’s own test data.
Test Data Re-use

days += month_days(start_month, start_year) - start_day;
days += end_day;
if (start_year == end_year) {
    int month = start_month + 1;
    while (month < end_month) {
        days += month_days(month, start_year); month ++;
    }
} else {
    int year; int month = start_month + 1;
    while (month <= 12) {
        days += month_days(month, start_year); month ++;
    }
    month = 1;
    while (month < end_month) {
        days += month_days(month, end_year); month ++;
    }
    year = start_year + 1;
    while (year < end_year) {
        days += 365;
        if (is_leap_year(year)) days ++;
        year ++;
days += month_days(start_month, start_year) - start_day;
days += end_day;
if (start_year == end_year) {
    int month = start_month + 1;
    while (month < end_month) {
        days += month_days(month, start_year); month ++;
    }
} else {
    int year; int month = start_month + 1;
    while (month <= 12) {
        days += month_days(month, start_year); month ++;
    }
    month = 1;
    while (month < end_month) {
        days += month_days(month, end_year); month ++;
    }
    year = start_year + 1;
    while (year < end_year) {
        days += 365;
        if (is_leap_year(year)) days ++;
        year ++;
    }
Program similarity and test data re-use

- Code structure
  - clone detection
  - plagiarism detection
- Code comments
- Identifier names
Will these techniques work?

Will they compromise fault-finding capability?
web: http://www.dcs.shef.ac.uk/~phil
email: p.mcminn@sheffield.ac.uk
twitter: @philmcminn
Questions & Discussion