Search-Based Testing

Phil McMinn
University of Sheffield, UK
Overview

How and why Search-Based Testing Works
Examples: Temporal, Functional, Structural
Search-Based Test Data Generation
Empirical and Theoretical Studies
Testability Transformation
Input Domain Reduction
Acknowledgements

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

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

- Manual design of test cases / scenarios
- Laborious, time-consuming
- Tedious

(difficult!
(Where are the faults ???)

The hard part...)
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.
Random Test Data Generation
Fitness-guided search

Fitness

Input
Fitness-guided search

Fitness

Input
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 Functions

Often easy

We often define metrics

Need not be complex
Conventional testing

manual design of test cases / scenarios

laborious, time-consuming

tedious
difficult!
(where are the faults ???)

Search-Based Testing:

automatically - 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
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
White box testing

fitness function analyses the outcome of decision statements and the values of variables in predicates
White-box + testing for assertion violations

Safety condition (desired state):

\[
\text{speed} < 150 \text{mph}
\]

Fitness function:

\[
f = 150 - \text{speed}
\]

Fitness minimised

If \( f \) is zero or less a fault is found
Search Techniques
Hill Climbing
Hill Climbing

Input

Fitness

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

Fitness

Input
Simulated Annealing

Worse solutions temporarily accepted
Evolutionary Algorithm

Fitness

Input
Evolutionary Algorithm
Evolutionary Testing

- Mutation
- Recombination
- Selection
- Insertion

Fitness Evaluation

Test cases
Monitoring
Execution

End?
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...

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

<table>
<thead>
<tr>
<th>Representation</th>
<th>A method of encoding all possible inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbourhood</td>
<td>Usually straightforward</td>
</tr>
<tr>
<td>Fitness Function</td>
<td>Inputs are already in data structures</td>
</tr>
</tbody>
</table>
Ingredients for Search-Based Testing

Representation

Part of our understanding of the problem

Neighbourhood

We need to know our near neighbours

Fitness Function
<table>
<thead>
<tr>
<th>Ingredients for Search-Based Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Representation</strong></td>
</tr>
<tr>
<td><strong>Neighbourhood</strong></td>
</tr>
<tr>
<td><strong>Fitness Function</strong></td>
</tr>
</tbody>
</table>
More search algorithms

Tabu Search

Particle Swarm Optimisation

Ant Colony Optimisation

Genetic Programming

Estimation of Distribution Algorithms
Important Publications


Important Publications

Surveys


Important Publications


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

The test data executes the 'wrong' path
The outcomes at key decision statements matter.

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

TARGET = 2
TARGET = 1
TARGET = 0

minimisation
Analysing predicates

Approach level alone gives us coarse values

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

$\begin{align*}
    a &= 50, b = 0 \\
    a &= 45, b = 5 \\
    a &= 40, b = 10 \\
    a &= 35, b = 15 \\
    a &= 30, b = 20 \\
    a &= 25, b = 25
\end{align*}$

going ‘closer’ to being true
Branch distance

Associate a distance formula with different relational predicates

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

- `a = 50, b = 0` branch distance = 50
- `a = 45, b = 5` branch distance = 40
- `a = 40, b = 10` branch distance = 30
- `a = 35, b = 15` branch distance = 20
- `a = 30, b = 20` branch distance = 10
- `a = 25, b = 25` branch distance = 0

Getting ‘closer’ to being true
### Branch distances for relational predicates

<table>
<thead>
<tr>
<th>Relational predicate</th>
<th>Objective function (obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>if (TRUE) then 0 else (K)</td>
</tr>
<tr>
<td>(a = b)</td>
<td>if (abs(a - b) = 0) then 0 else (abs(a - b) + K)</td>
</tr>
<tr>
<td>(a \neq b)</td>
<td>if (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
            }
        }
    }
    ...
```

*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

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

- Increase
- Decrease

Start with an increase, then decrease, then increase again, and so on.
Alternating Variable Method

Accelerated hill climb

Fitness

Input variable value
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

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

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

    ...
}
```
A search-based test data generator tool
IGUANA (Java)

search algorithm

fitness computation

Java Native Interface

Test object (C code compiled to a DLL)

inputs

information from test object instrumentation
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;
}
```
1. Parse the code and extract control dependency graph

```
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;
}
```

for monitoring control flow and variable values in predicates
Test Object Preparation

3. Map inputs to a vector

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

Straightforward in many cases

Inputs composed of dynamic data structures are harder to compose

Kiran Lakhotia, Mark Harman and Phil McMinn.
Handling Dynamic Data Structures in Search-Based Testing.
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(3, equals(0, a, 0))) {
        flag = 1;
    }
    if (node(4, is_true(flag) && equals(1, b, 0))) {
        return 2;
    }
    return -1;
}
```
Each branching condition is replaced by a call to the function `node(...)`

```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;
}
```

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
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
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
- Evolutionary Testing

8

20

9
Wins for the AVM

Success Rate (%)

Evolutionary Testing

Hill Climbing
Wins for the AVM

Average number of fitness evaluations

- Evolutionary Testing
- Hill Climbing
When does the AVM win?
Wins for Evolutionary Testing
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 via **crossover**.

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

1010100011110000111010
1111101010000000101011
0001001010000111101011
The schema theory predicts that schema of above average fitness will proliferate in subsequent generations of the evolutionary search.
Schemata
The Genetic Algorithm
Royal Road

S1: 11111111111111111111111111111111
S2: ****1111111111111111111111111111
S3: ********111111111111111111111111
S4: ************1111111111111111111111
S5: ****************1111111111111111111111
S6: ****************1111111111111111111111
S7: ********1111111111111111111111111111
S8: 11111111111111111111111111111111111111
S9: 11111111111111111111111111111111111111
S10: 11111111111111111111111111111111111111
S11: 11111111111111111111111111111111111111
S12: 11111111111111111111111111111111111111
S13: 11111111111111111111111111111111111111
S14: 11111111111111111111111111111111111111
S15: 11111111111111111111111111111111111111
When Crossover Helps

Executes the target

R1

Q1

P1

P2

Q2

P3

P4

R2

Q1

P1

P2

Q2

P3

P4
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) \mid a = b\}

\{(50, 50, 25) \}

\{(100, 100, 10) \}

...  

\{(a, b, c) \mid 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

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

\[(a, b, c) \mid a = b \land b \geq 100 \land c \leq 10\] covering schema

```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
    ...
```
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...) | 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

\[\{(a, b, c, d \ldots) | a = b \land b = c \land c = d \ldots\}\]

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

Progressive Landscape
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


Testability Transformation
The ‘Flag’ Problem

```c
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) {
        ...
    }
}
```

```c
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

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

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

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

```c
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
        }
    }
}
```

```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 (%)
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’
Nested branches

Change in success rate after applying transformation (%)

Nested branches

Dependent predicates
Nested branches

Independent and some dependent predicates

Dependent predicates

Change in success rate after applying transformation (%)
When not preserving program equivalence can go wrong

```
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

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
    }
}
```
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
    }
}
```

-100,000 ... 100,000
-100,000 ... 100,000
-100,000 ... 100,000
200,001
Variable Dependency Analysis

```c
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

```java
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:

100 x 100 x 1

100 x 100 x 100
Results with Random Testing
Results with AVM
Effect on AVM

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

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

Increases:
- irrelevant1
- irrelevant2
- irrelevant3
- required1

Decreases:
- 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
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.
Other applications of Search-Based Testing

Mutation testing

State machine testing
Other applications of Search-Based Testing

**Test suite reduction**


**(Time-aware) Test suite prioritisation**

Other applications of Search-Based Testing

Combinatorial Interaction Testing

GUI Testing