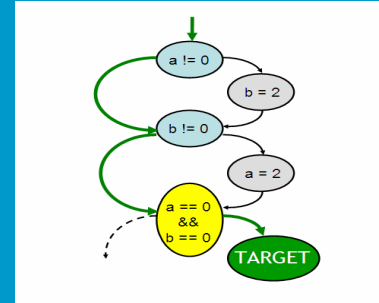
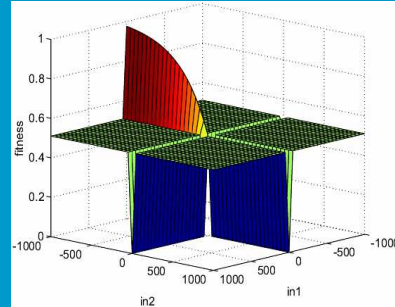
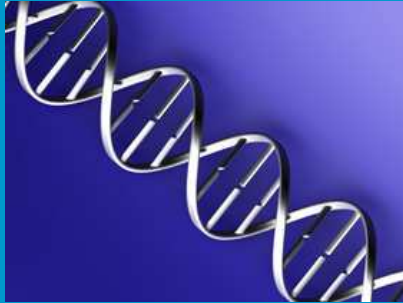




The
University
Of
Sheffield.



Search-Based Software Testing

An Overview

Phil McMinn
University of Sheffield

Sheffield



Sheffield



Overview of the Overview

- The problems of testing
- Why search-based approaches are useful
- Paradigms
 - Structural
 - Search-space reduction
 - Landscapes
 - Functional
 - 'Grey Box'

Testing, testing, testing...

- Practical problems
- ... but there are also fundamental problems

Exhaustive Testing

```
double cliparc (double cx,  
                double cy,  
                double rad,  
                double start,  
                double end,  
                int iclipx,  
                int iclipy,  
                int icliprad,  
                int flag)  
{  
    ...  
}
```

Exhaustive Testing

```
double cliparc (double cx,  
                double cy,  
                double rad,  
                double start,  
                double end,  
                int iclipx,  
                int iclipy,  
                int icliprad,  
                int flag)
```

```
{
```

```
...
```

```
}
```

5 double
variables

-100 to 100,
accuracy of 0.1

4 integer
variables

-100 to 100

= an input domain size
of 5×10^{25}

Exhaustive Testing

```
double cliparc (double cx,  
                double cy,  
                double rad,
```

5 double
variables

-100 to 100,
step of 0.1

Even if we tried each possible input
vector at a rate of
100 per second
it would take

16 quadrillion years...

er
s
100

main size

OF 5×10^{20}

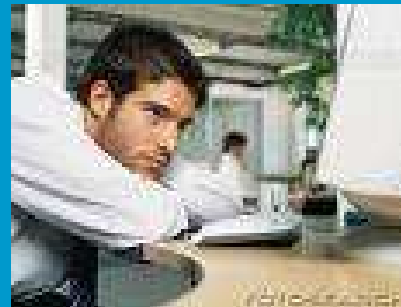
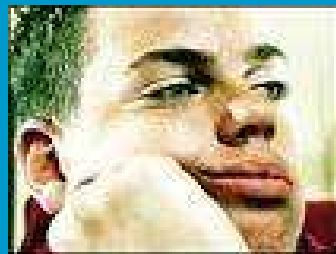
```
{  
...  
}
```

The Need for Test Methods

- Structural
- Functional
- ... and so on

The Need for Automation

Laborious



Costly



Search-Based Software Testing

... is an approach to automating the activities of testing
using metaheuristic search techniques



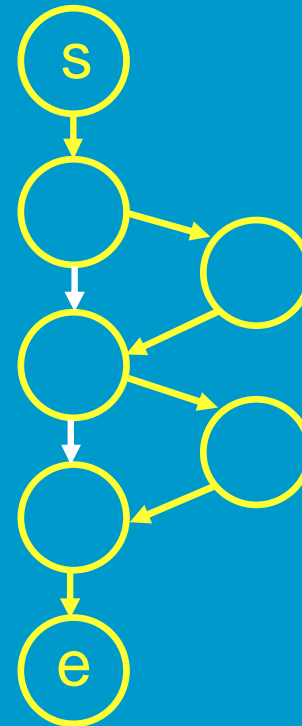
The
University
Of
Sheffield.

Structural (White-Box) Testing

Path Coverage



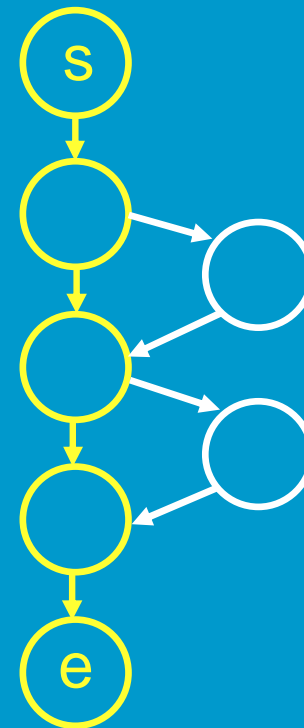
```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+" "+b+" "+c);  
}
```



Path Coverage



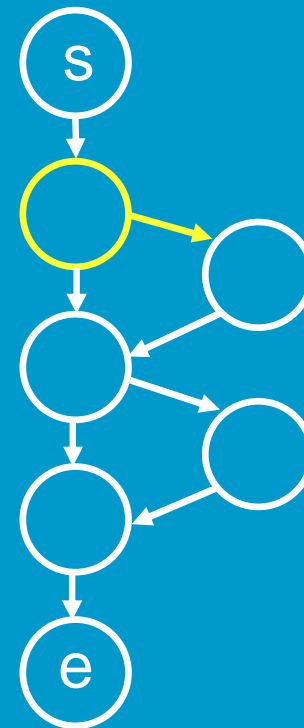
```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+" "+b+" "+c);  
}
```





Branch Coverage

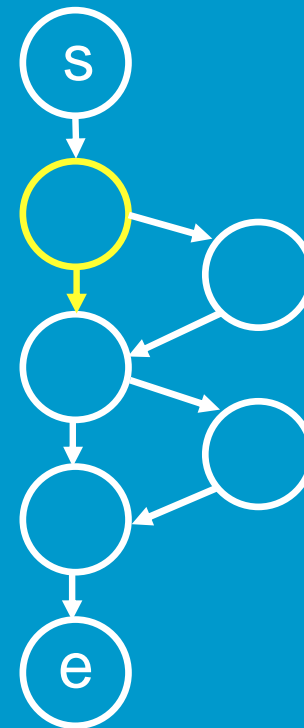
```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+" "+b+" "+c);  
}
```





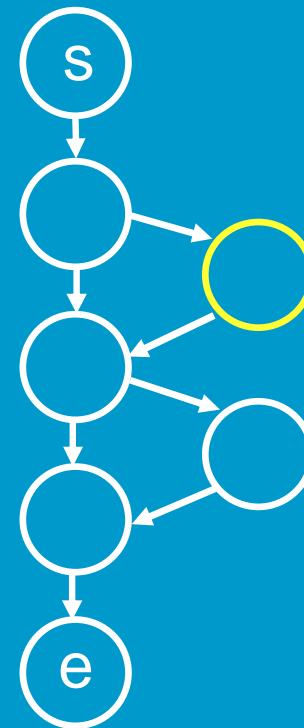
Branch Coverage

```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+" "+b+" "+c);  
}
```



Statement Coverage

```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+" "+b+" "+c);  
}
```



This is why we care

```
double
cliparc(double cx, double cy, double rad, double start,
        double end, int iclipx, int iclipy, int icliprad, int flag)
{
    double clipx, clipy, cliprad;
    double x, y, tx, ty, dist;
    double alpha, theta, phi, a1, a2, d, l;
    double sclip = 0, eclip = 0;
    bool in;

    clipx = (double) iclipx;
    clipy = (double) iclipy;
    cliprad = (double) icliprad;
    x = cx - clipx;
    y = cy - clipy;
    dist = sqrt((double) (x * x + y * y));

    if (!rad || !cliprad)
        return(-1);
    if (dist + rad < cliprad) {
        /* The arc is entirely in the boundary. */
        //DrawArc((int)cx, (int)cy, (int)rad, start, end);
        return(flag?start:end);
    } else if ((dist - rad >= cliprad) || (rad - dist >= cliprad)) {
        /* The arc is outside of the boundary. */
        return(-1);
    }
    /* Now let's figure out the angles at which the arc crosses the
     * circle. We know dist != 0.
     */
    if (x)
        phi = atan2((double) y, (double) x);
    else if (y > 0)
        phi = M_PI * 1.5;
    else
        phi = M_PI / 2;
    if (cx > clipx)
        theta = M_PI + phi;
    else
        theta = phi;

    alpha = (double) (dist * dist + rad * rad - cliprad * cliprad) /
        (2 * dist * rad);
```

```
/* Sanity check */
if (alpha > 1.0)
    alpha = 0.0;
else if (alpha < -1.0)
    alpha = M_PI;
else
    alpha = acos(alpha);

a1 = theta + alpha;
a2 = theta - alpha;
while (a1 < 0)
    a1 += M_PI * 2;
while (a2 < 0)
    a2 += M_PI * 2;
while (a1 >= M_PI * 2)
    a1 -= M_PI * 2;
while (a2 >= M_PI * 2)
    a2 -= M_PI * 2;

tx = cos(start) * rad + x;
ty = sin(start) * rad + y;
d = sqrt((double) tx * tx + ty * ty);
in = (d > cliprad) ? false : true;

/* Now begin with start. If the point is in, draw to either end, a1,
 * or a2, whichever comes first.
 */
d = M_PI * 3;
if ((end < d) && (end > start))
    d = end;
if ((a1 < d) && (a1 > start))
    d = a1;
if ((a2 < d) && (a2 > start))
    d = a2;

// ....
```

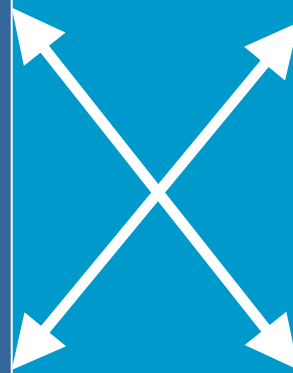
Widely studied

Search Method

- Hill climbing
- Simulated annealing
- Genetic algorithms
- Memetic algorithms
- Scatter search
- Particle Swarm Optimization
- ...

Fitness Function

- Target-based
 - Path
 - Branch
 - Statement
 - Def-use pair
 - ... and more
- Coverage-based



Random Search

```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
  
    if (a == b) {  
        // target 2  
    }  
  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
  
}
```

Random Search

```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
  
    if (a == b) {  
        // target 2  
    }  
  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
  
}
```

Range of a: -500 ... 500

Range of b: -500 ... 500

Random Search

Range of a: -500 ... 500

Range of b: -500 ... 500

```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
  
    if (a == b) {  
        // target 2  
    }  
  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
  
}
```

Approx 1:2

Random Search

Range of a: -500 ... 500

Range of b: -500 ... 500

```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
    if (a == b) {  
        // target 2  
    }  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
}
```

Approx 1 : 1000

Random Search

Range of a: -500 ... 500

Range of b: -500 ... 500

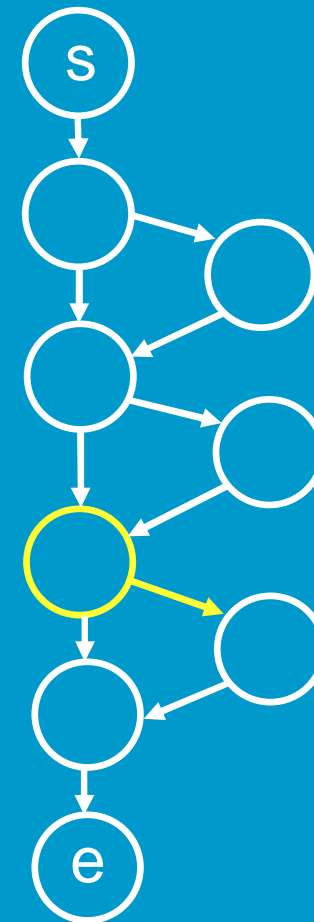
```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
  
    if (a == b) {  
        // target 2  
    }  
  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
}
```

Approx $1 : 10^6$

Target-Based Approaches

- Each uncovered path, branch, statement etc. becomes the individual 'target' of the search

```
Void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
    if (a == b) {  
        // target 2  
    }  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
}
```



Predicate 'Distance'

```
void example(int a, int b)
{
    c = a * a;
    if (b == c)
    {
        // TARGET
    }
}
```

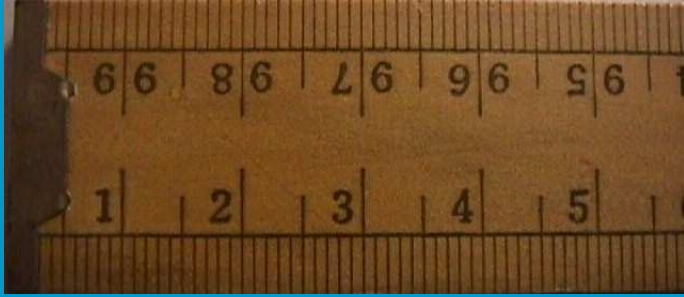
Program executed in order to find branch distance values

'if (b == c)'

- Distance formula = $\text{abs}(b-c)$
- $a = 2, b = 1 \rightarrow c = 4 \rightarrow \text{abs}(b-c) = 3$
- $a = 2, b = 2 \rightarrow c = 4 \rightarrow \text{abs}(b-c) = 2$
- $a = 2, b = 4 \rightarrow c = 4 \rightarrow \text{abs}(b-c) = 0$

distance minimised to zero

True branch executed



Distance Formulas

$a == b$	$\text{abs}(a-b)$
$a <= b$	$a - b$
$a >= b$	$b - a$
...	

... and so on

Miller and Spooner Approach (1976)

```
void example(int a,  
            int b,  
            int c) {  
    if (a > b) {  
        int t = a; a = b; b = t;  
    }  
    if (b > c) {  
        int t = b; b = c; c = t;  
    }  
    print(a+ " "+b+ " "+c);  
}
```

```
void example(int a,  
            int b,  
            int c) {  
    minimise(b - a < 0)  
    int t = a; a = b; b = t;  
    minimise(c - b < 0)  
    int t = b; b = c; c = t;  
    print(a+ " "+b+ " "+c);  
}
```

Meet the first* Search-Based Software Engineers



Webb Miller



David Spooner

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

* Unless anyone knows different



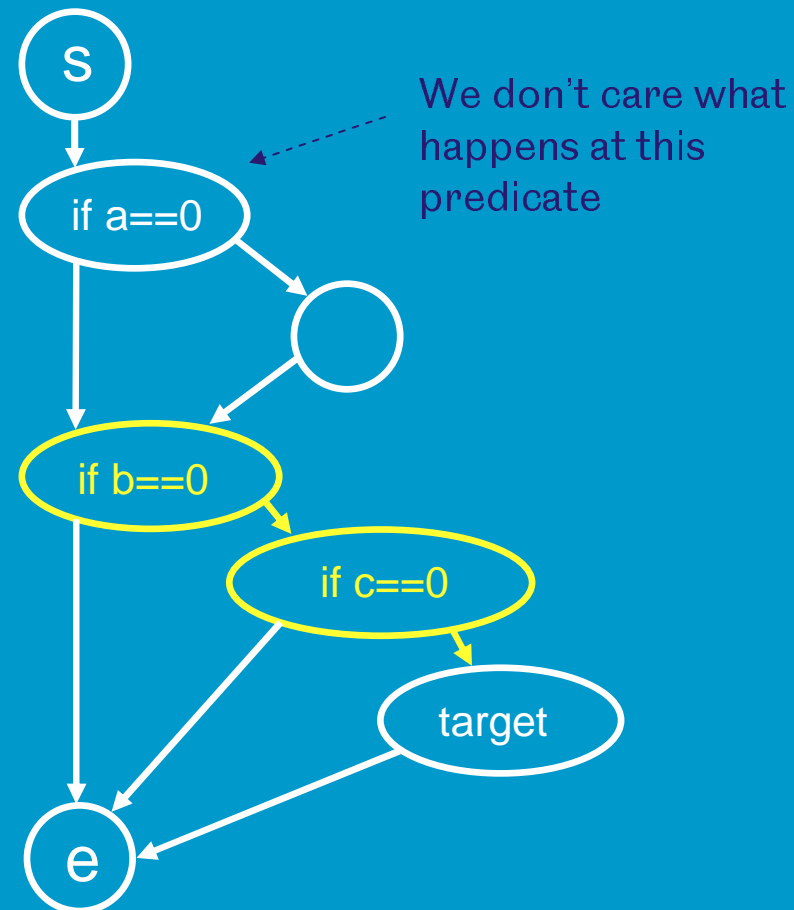
Goal-Oriented Approach (Korel 1990/2)

- Miller and Spooner approach for paths
- To execute a target such as a branch, have to specify a path
- Korel removed this requirement:
 - Classification of edges in CFG

Goal-Oriented Approach: Branch Classification

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

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



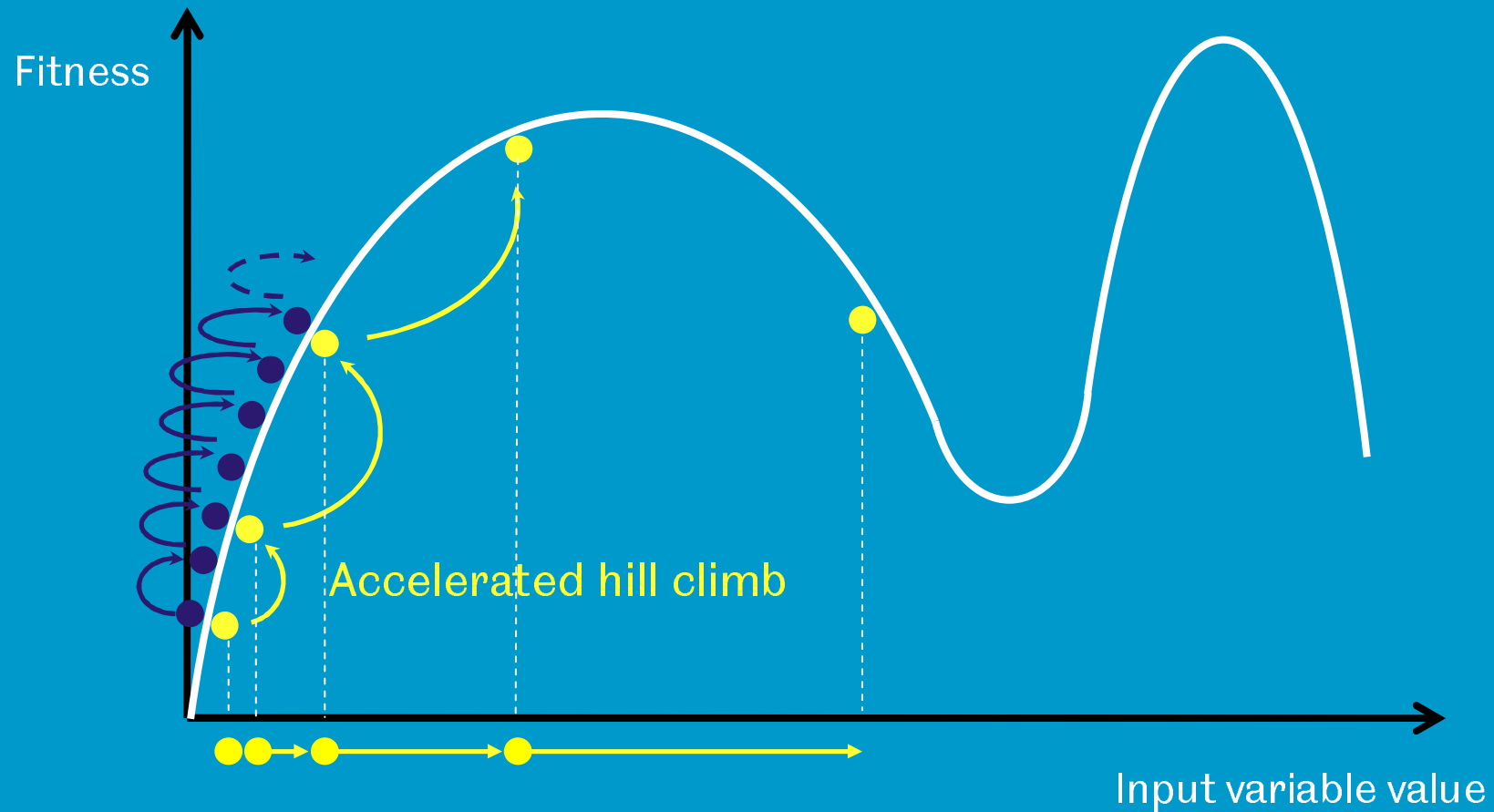
Critical branches must be executed in order for the target to be reached

Goal-Oriented Approach:

Alternating Variable Method

- The **alternating variable method** is similar to **hill climbing**
- Restricted neighbourhood
 - **One input variable at a time**
 - **Probe Moves**
- Accelerated **pattern** moves

Goal-Oriented Approach: Alternating Variable Method



Goal-Oriented Approach:

Alternating Variable Method

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

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

- Randomly generate start point $\langle a=10, b=20, c=30 \rangle$
- Consider 'a'
 - Probe moves on a have no effect
- Consider 'b'
 - Decrease probe move has effect
 - Accelerate until $b=0$
- Consider 'c' ...

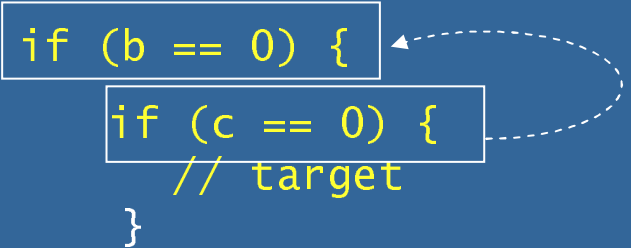
Goal-Oriented Approach:

Alternating Variable Method

- Fitness is only 'local'

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

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

A diagram illustrating the 'Alternating Variable Method'. It shows a code snippet with three nested if statements: 'if (a == 0) { ... }', 'if (b == 0) {', and 'if (c == 0) { // target }'. The 'if (b == 0) {' and 'if (c == 0) {' lines are highlighted with white boxes. A dashed white arrow points from the 'if (c == 0) {' box back to the 'if (b == 0) {' box, indicating a return path or a change in focus.

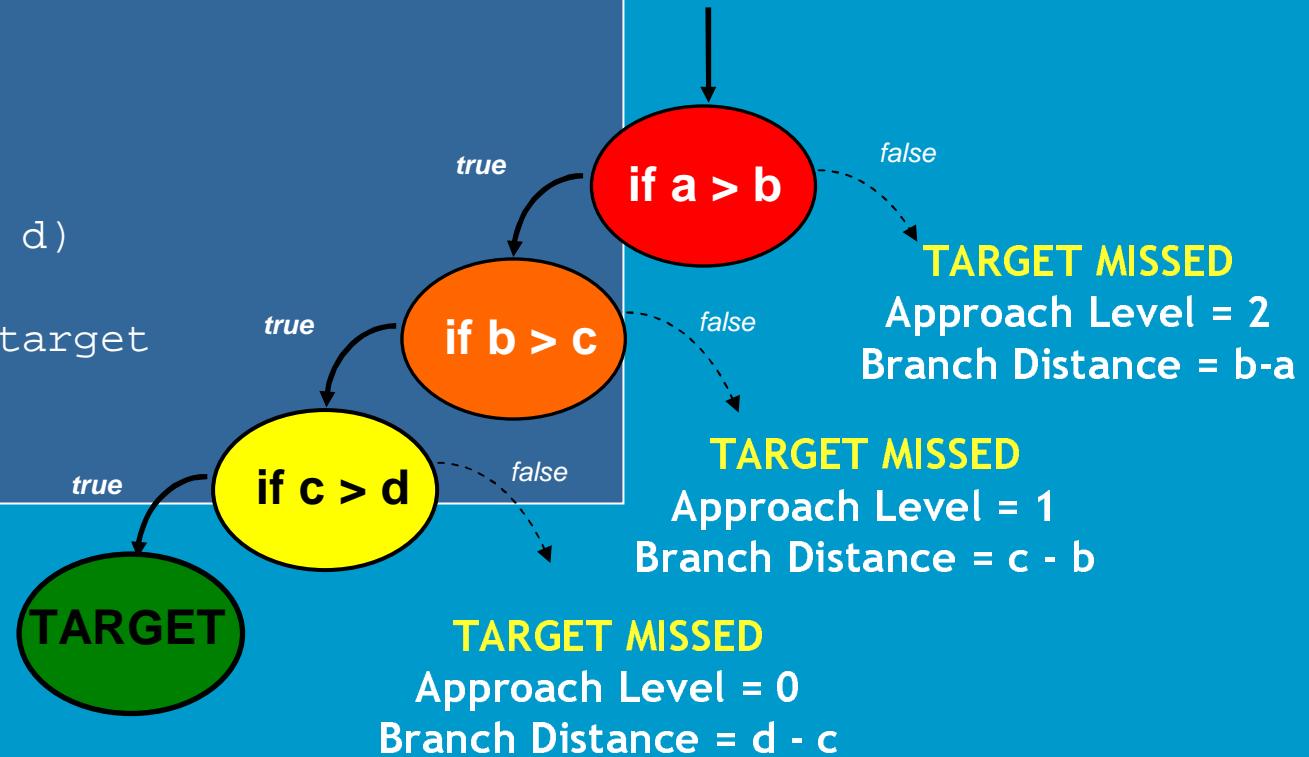


Evolutionary Approach

(Wegener et al. 2001)

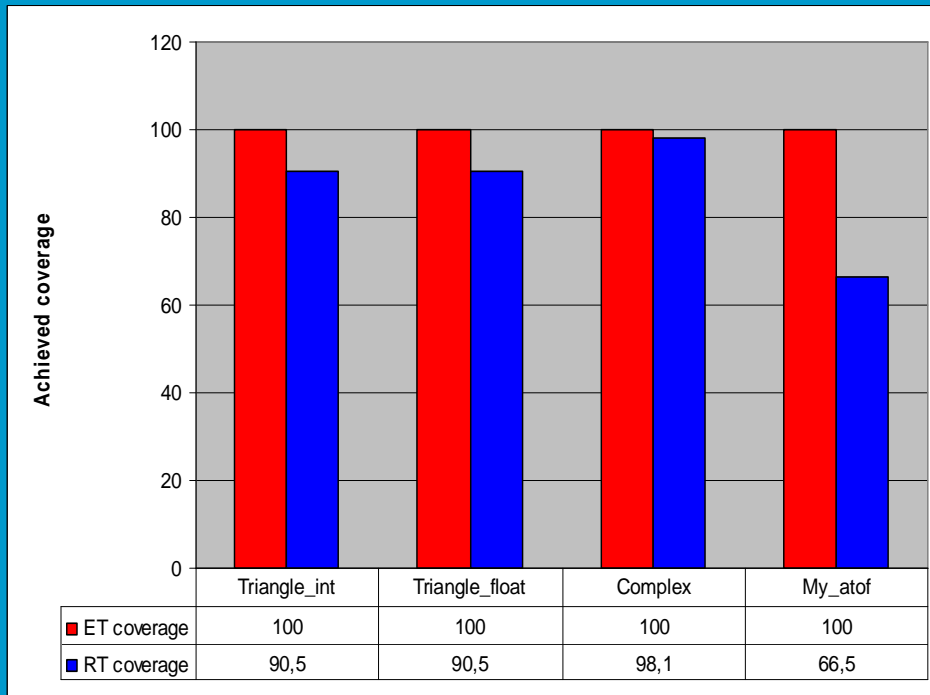
Fitness = Approach Level + normalized branch distance

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

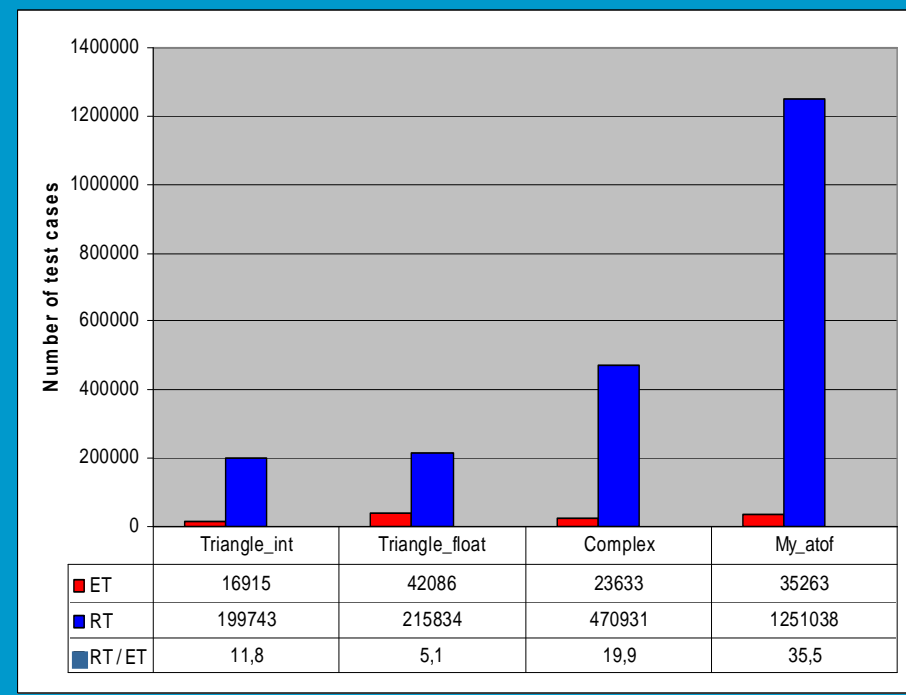


Evolutionary Approach

(Wegener et al. 2001)



Effectiveness



'Effort'

Coverage-Based Approaches

- Reward individuals on the basis of the number of structures covered (Roper, 1995)
- Or, punish individuals on the basis of the number of structures covered that have already been covered (Watkins, 1994)
- **Advantages**
 - simple to implement
- **Disadvantages**
 - lack of guidance to hard-to-execute structures

Coverage-Based Approaches

```
void example(int a, int b) {  
    if (a > b) {  
        // target 1  
    }  
  
    if (a == 0) {  
        // target 2  
    }  
  
    if (a == 0 && b == 0) {  
        // target 3  
    }  
}
```

→ Approx 1 : 10⁶

Timeline of Milestones

1976: Miller and Spooner

- dynamic approach
- predicate distance calculations
- simple numerical minimisation search
- paths

1990: Korel

- alternating variable method
- paths

1992: Korel – Goal-Oriented Approach

- branch classification method
- branches

1992: Xanthakis et al.

- first genetic algorithm approach
- paths

1995/6: Watkins et al., Roper et al

- coverage-oriented approaches
- genetic algorithms

1999: Pargas et al.

- genetic algorithm method for branches
- fitness function based on control dependencies

2001: Wegener et al.

- genetic algorithms with real-valued encodings
- fitness function combines critical branches (approach level) and branch distance
- defined new fitness functions for branches, paths, d-u pairs and more

Post-2001: 'Post-modern era'

- source code analysis to aid the search (testability transformation etc.)
- search landscape analysis

... we'll come back to this



The
University
Of
Sheffield.

Input Domain Reduction

Input Domain/Search Space Reduction

(Harman, Hassoun, Lakhotia, McMinn, Wegener 2007)

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

Effect of Reduction

```
void super-unsized-me(int irrelevant1, - - - -> -100,000 ... 100,000
                    int irrelevant2, - - - -> -100,000 ... 100,000
                    int im-ur-man) - - - -> -100,000 ... 100,000
```

= approx. 10^{16}

Effect of Reduction

```
void super-unsized-me(int irrelevant1,  
                     int irrelevant2,  
                     int im-ur-man)
```

~~-----▶ -100,000 ... 100,000~~
x
~~-----▶ -100,000 ... 100,000~~
x
-----▶ -100,000 ... 100,000

= 200,001

Variable Dependency Analysis

```
void example(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
 - hill climbing
 - genetic algorithm

- On

- 2 embedded controllers provided by DaimlerChrysler
- Functions from gimp, spice, tiff image library

360 branches tested with and without input domain reduction

(repeated 60 times)

Effect on Random Search

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

~~int b, int c~~ → -49 ... 50
→ -49 ... 50
→ -49 ... 50

x
x } 10⁶

Any value of a, any value of b, c == 0

Probability of executing target:
 $\frac{100 \times 100 \times 1}{100^3} = 1/100$

Effect on Random Search

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

    if (b == 0) {
        ...
    }

    if (c == 0) {
        // target
    }
}
```

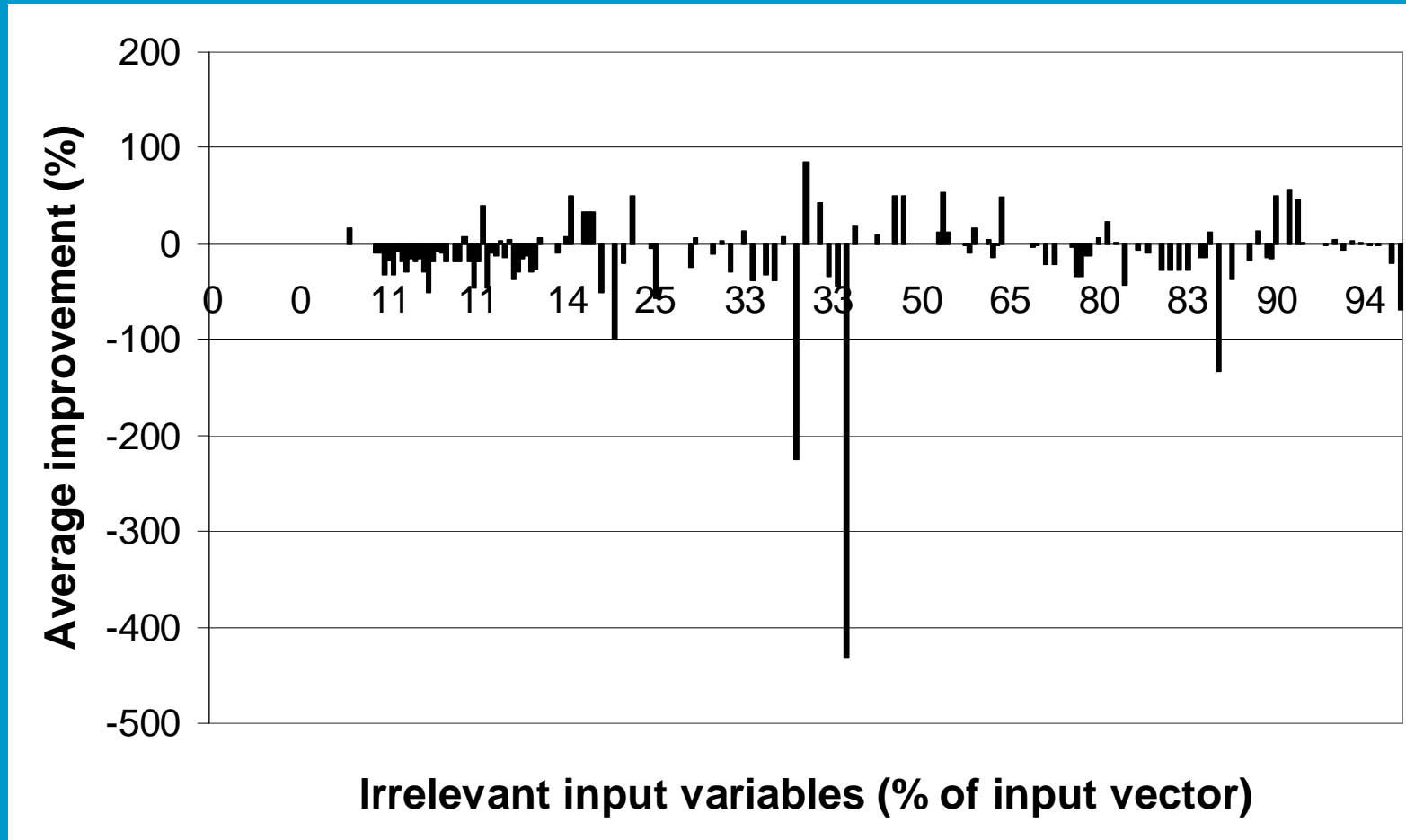


$c == 0$

Probability of executing target:

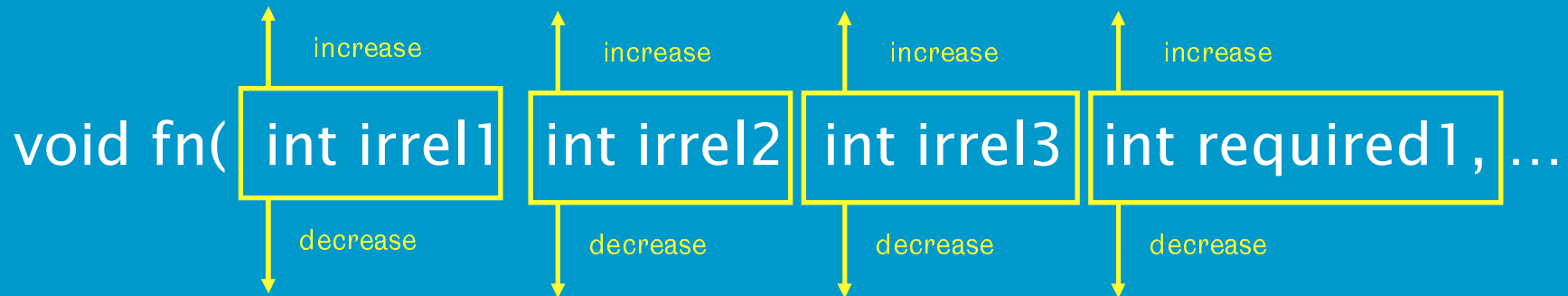
$$\frac{1}{100} = 1/100$$

Results with Random Testing



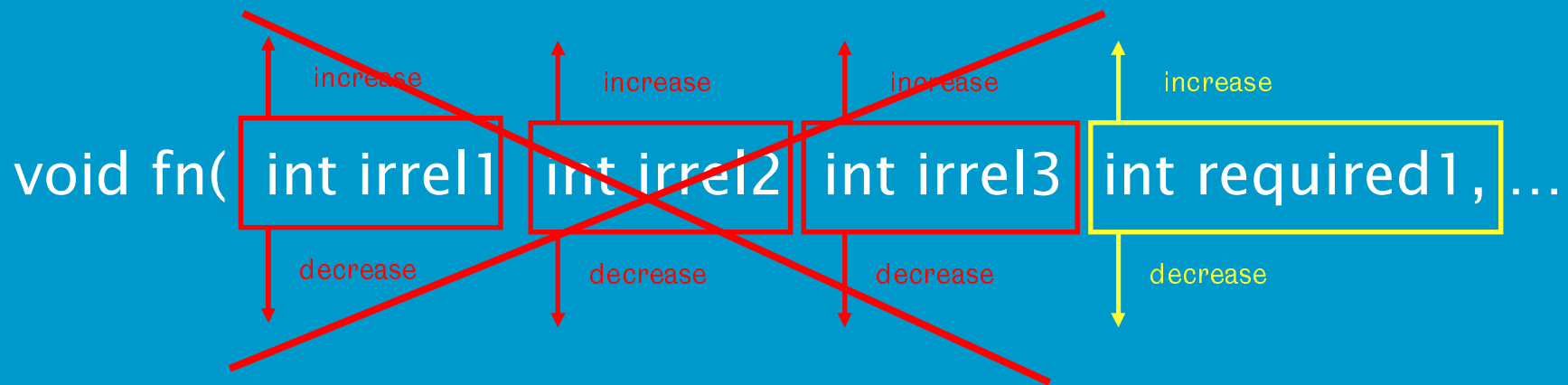
Effect on Hill Climbing

- Saves probe moves (i.e. fitness evaluations) around irrelevant variables

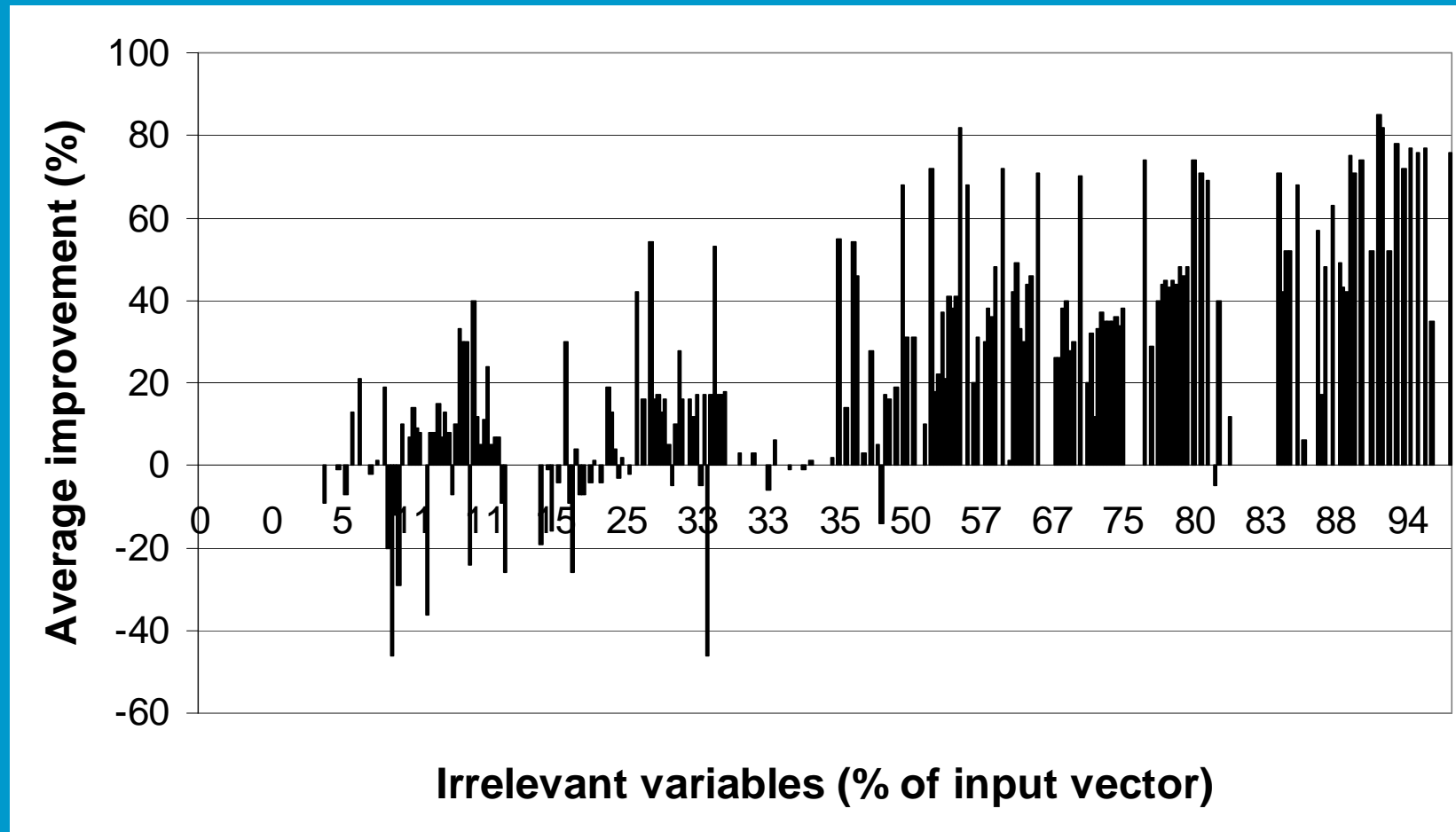


Effect on Hill Climbing

- Saves probe moves (i.e. fitness evaluations) around irrelevant variables



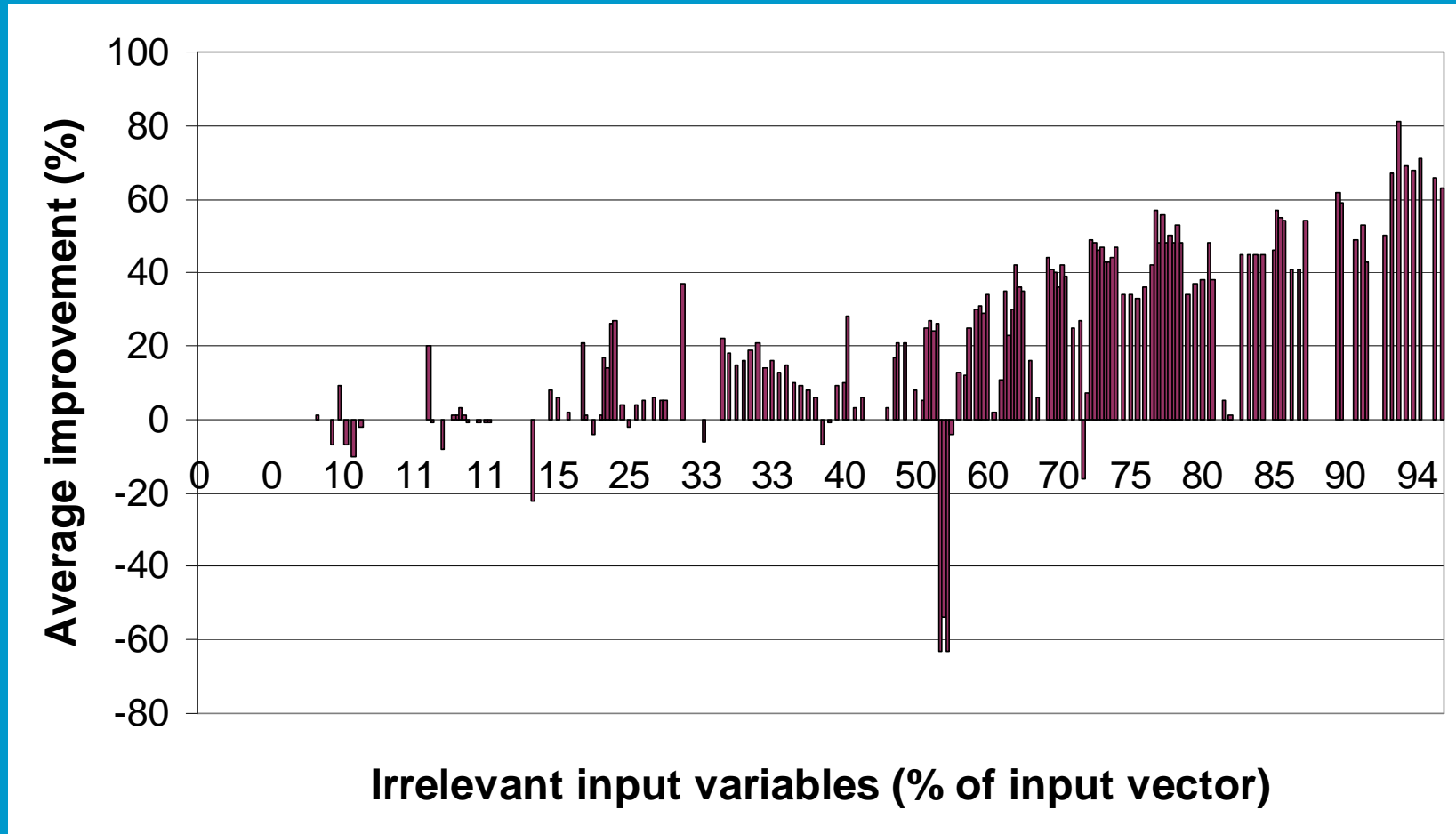
Results with Hill Climbing



Effect on Genetic Algorithm

- Saves mutations (i.e. fitness evaluations) on irrelevant variables
- Concentrate mutations on variables that matter
 - Likely to speed up search

Results with Genetic Algorithm



Statistical Analysis

- Applying paired t-tests with a significance level of 1%
- Significant reduction in effort for
 - Hill climbing: 82 branches
 - Genetic Algorithm: 86 branches

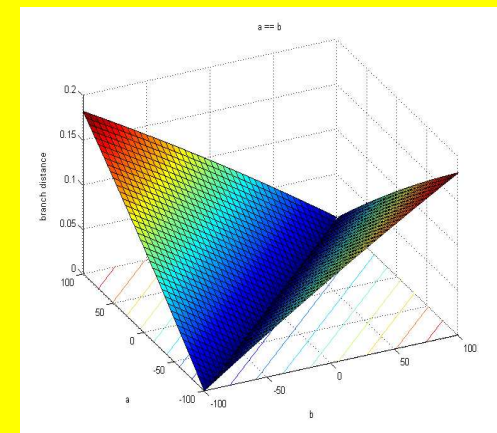
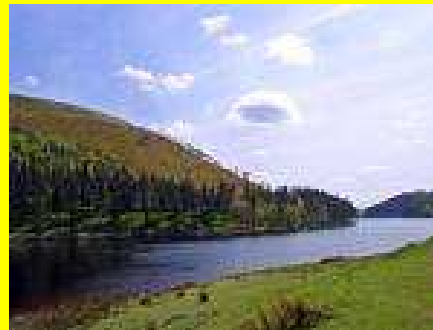
Summary

- Variable dependency analysis can be used to reduce input domains
- This can **reduce search effort** for
 - Hill climbing
 - Genetic Algorithms
- Perhaps surprisingly, there is **little change for random search**



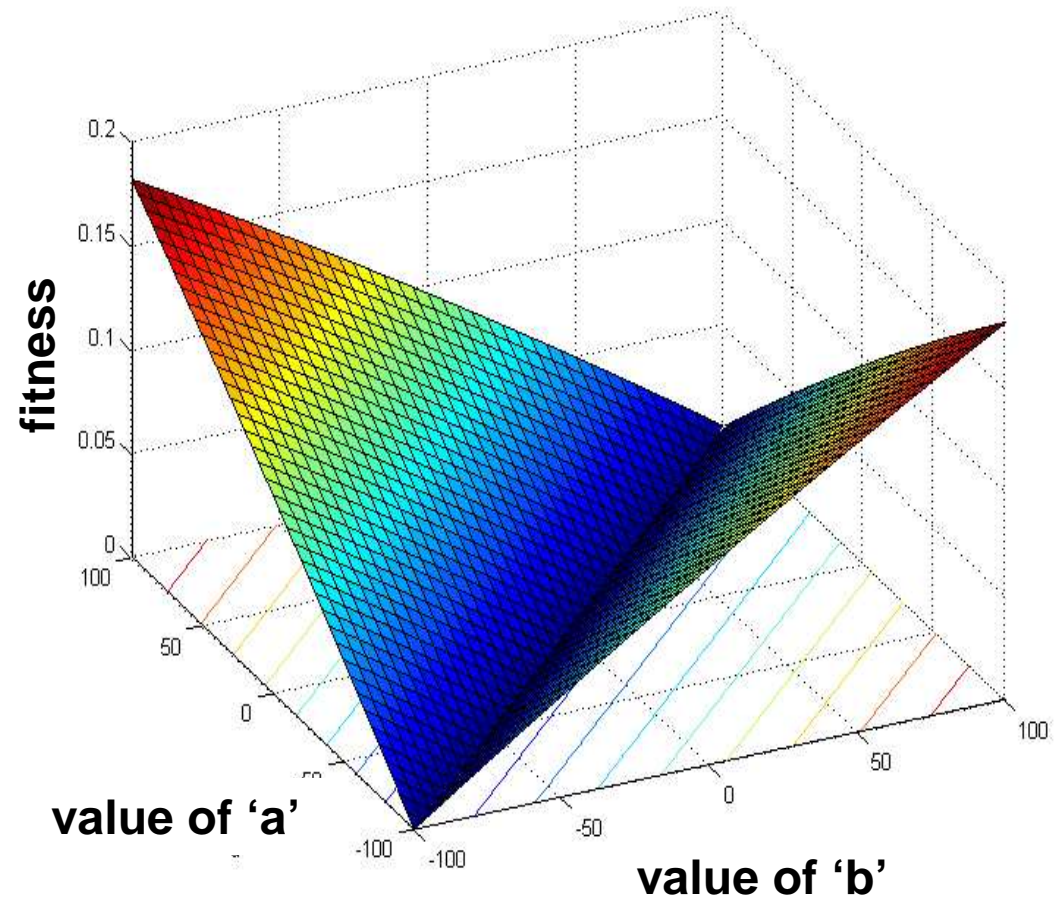
The
University
Of
Sheffield.

Landscapes and Search-Based Testing

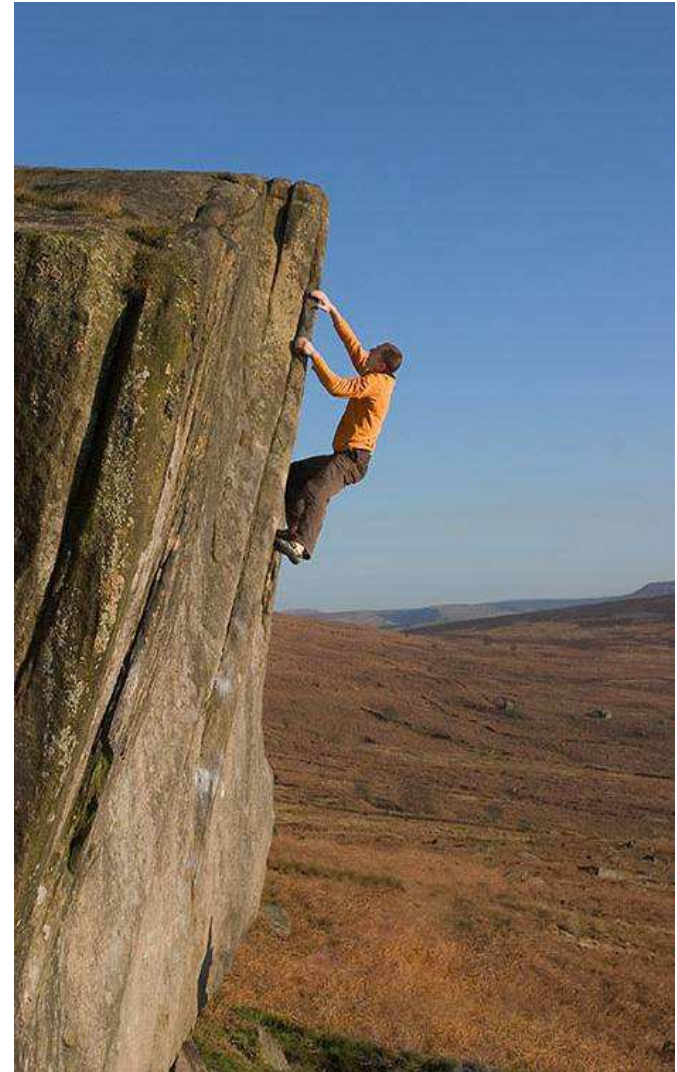
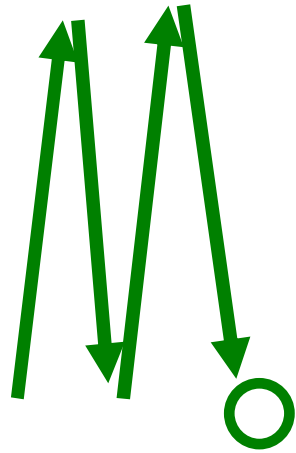


An Ideal Fitness Landscape

```
void very_simple_example  
  (double a, double b)  
{  
  if (a == b)  
  {  
    // TARGET  
  }  
}
```



Nesting



Testability Transformation

Harman et al (2004)

- Create a new version of the program from the original
 - So that the **search is improved**
- The two programs do not necessarily have to be functionally equivalent
 - The goal is simply that test data for the new transformed program **is good for the real version**
- Transformed program is a 'means to an end' – we can discard once we're done

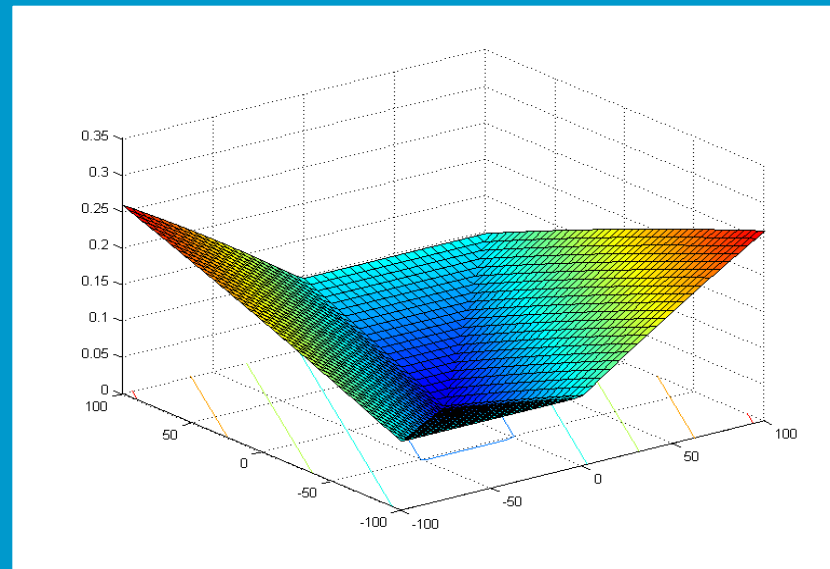
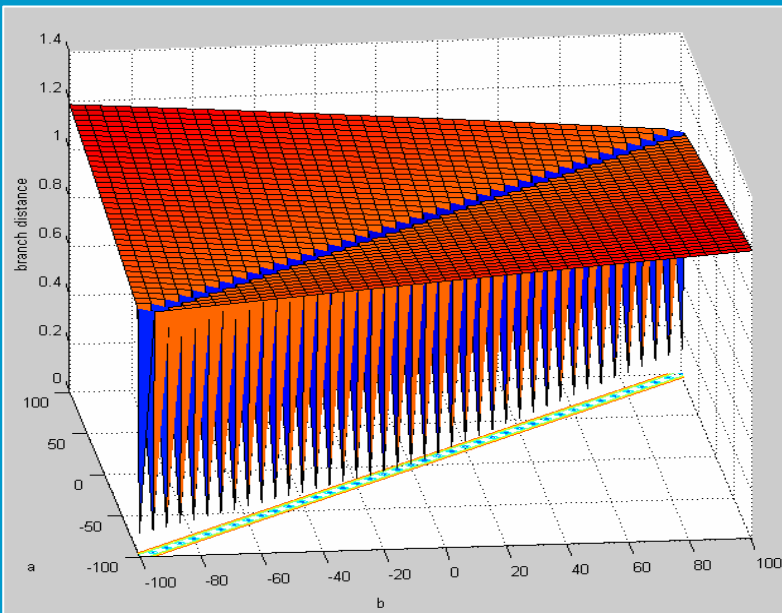
Nesting Testability Transformation

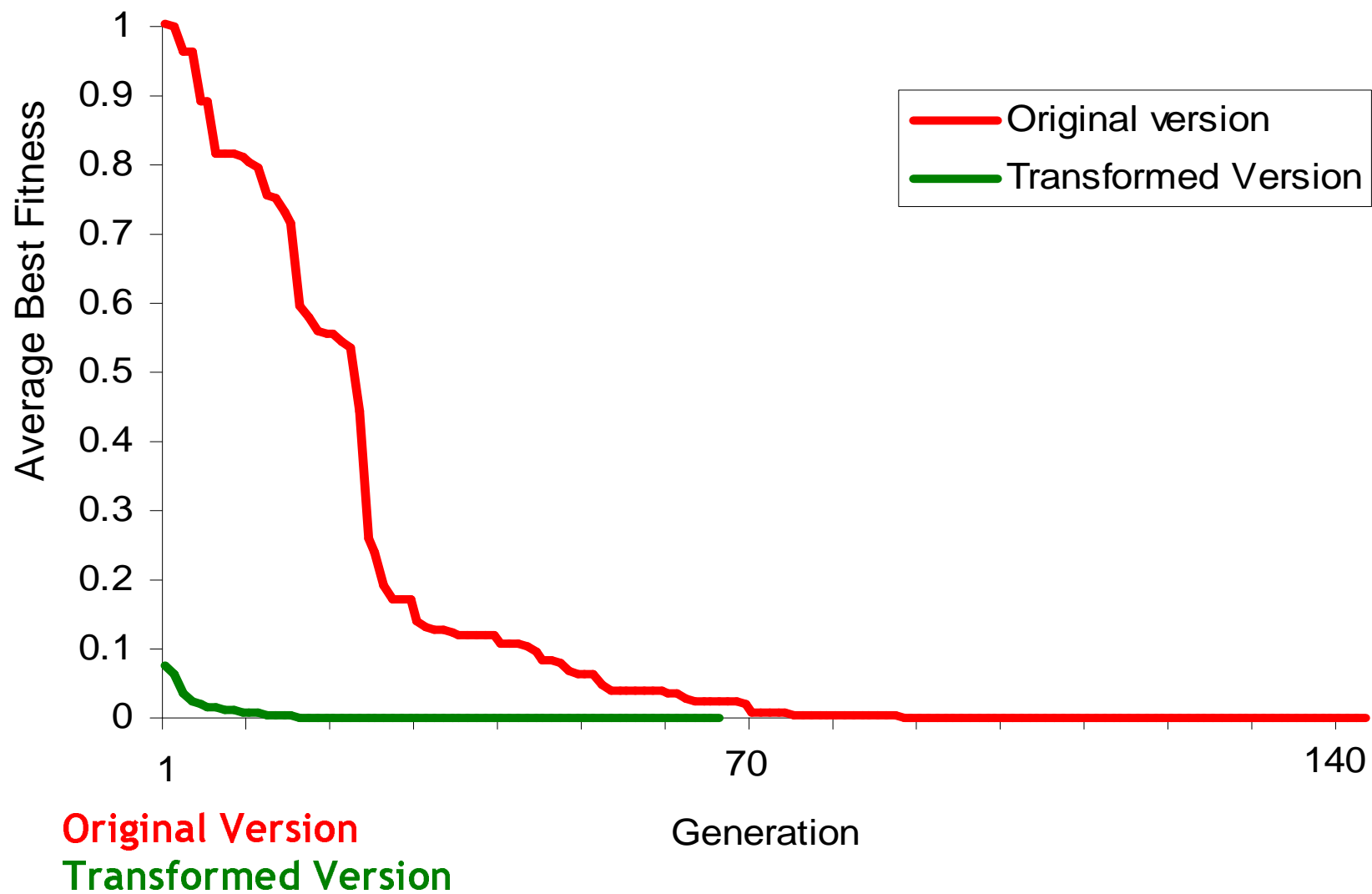
Original \longrightarrow Transformed

```
void study1(double a, double b)
{
  if (a == b)
  {
    if (b == 0)
    {
      // TARGET
    }
  }
  ...
}
```

```
void study1_transformed(double a, double b)
{
  double _dist = 0;
  _dist += distance(a == b);
  _dist += distance(b == 0);

  if (_dist == 0)
  {
    // TARGET
  }
  ...
}
```



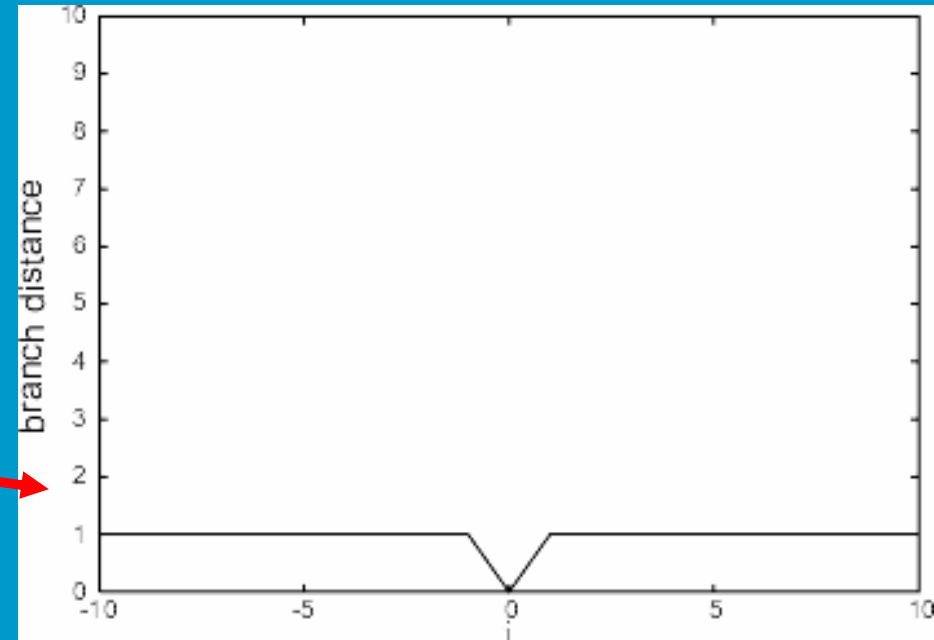




Flag variables

Node

```
(s) void flag_example(int i) {  
(1)     int flag = 0;  
(2)     if (i == 0)  
(3)         flag = 1;  
(4)     if (flag)  
(5)         // target node  
(e) }
```





Chaining Approach

(Ferguson & Korel 1996)

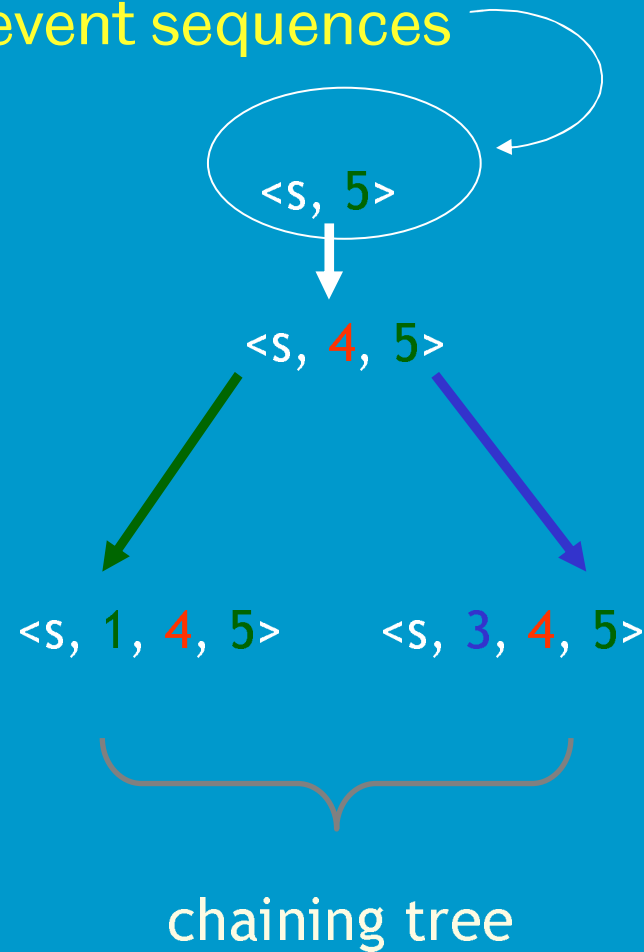
- The chaining approach uses **data flow analysis** with the intent of providing the search with **more information**

Node	
(s)	void flag_example(int i) {
(1)	int flag = 0;
(2)	if (i == 0)
(3)	flag = 1;
(4)	if (flag)
(5)	// target node
(e)	}

Chaining Approach (Ferguson & Korel 1996)

Find **last definitions** and construct **event sequences**

Node	
(s)	void flag_example(int i) {
(1)	int flag = 0;
(2)	if (i == 0)
(3)	flag = 1;
(4)	if (flag)
(5)	// target node
(e)	}



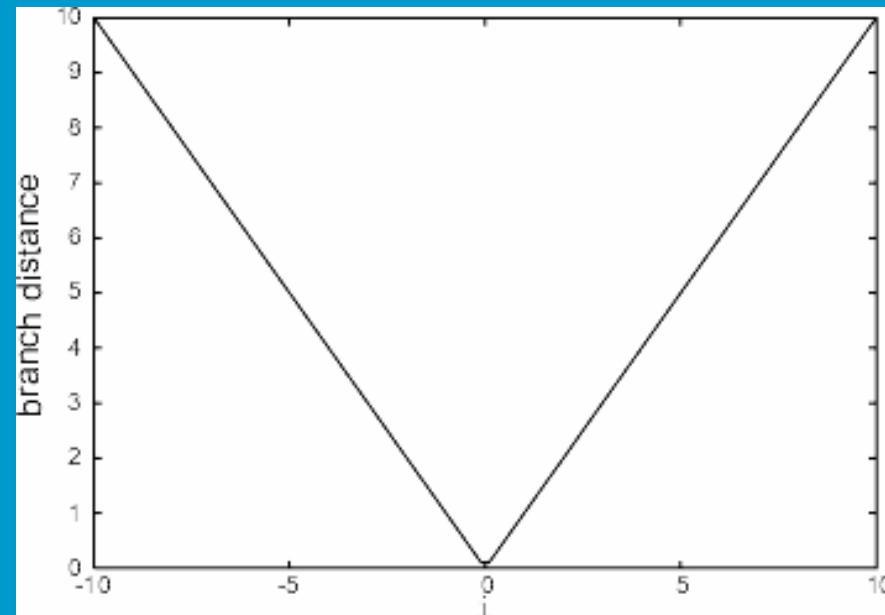
Chaining Approach (Ferguson & Korel 1996)

Search again, this time trying to find test data for an event sequence rather than just the target

<s, 3, 4, 5>

Need to make this predicate true

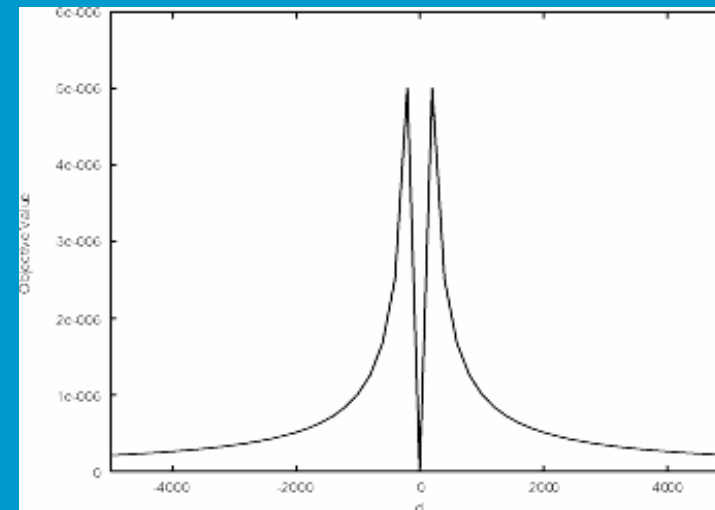
Node	
(s)	void flag_example(int i) {
(1)	int flag = 0;
(2)	if (i == 0)
(3)	flag = 1;
(4)	if (flag)
(5)	// target node
(e)	}



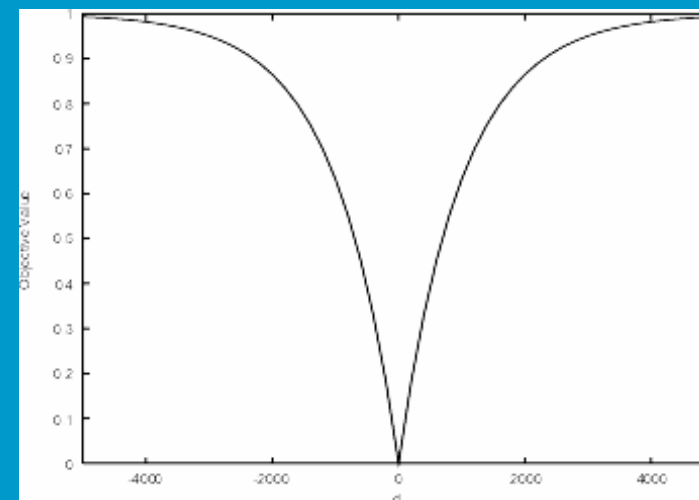
Deceptive Program

<s, 5, 6>

Node	
(s)	void deceptive(double d) {
(1)	double r;
(2)	if (r == 0.0)
(3)	r = 0.0;
	else
(4)	r = 1.0 / d;
(5)	if (r == 0.0)
(6)	// target node
(e)	}



<s, 3, 5, 6>





The
University
Of
Sheffield.

Functional (Black-Box) Testing

Verifying the Functional Behaviour of Modules (Tracey 2000)

Pre condition:

$n \geq 0$ and $n \leq 10$



Wrapping Counter



Post conditions:

$n < 10 \rightarrow r = n + 1$

$n = 10 \rightarrow r = 0$

Verifying the Functional Behaviour of Modules (Tracey 2000)

- Combine pre-condition with **negated post condition**



Describes a **failure**

$n \geq 0$ and $n \leq 10$ and $n < 10 \rightarrow r \neq n + 1$

$n \geq 0$ and $n \leq 10$ and $n = 10 \rightarrow r \neq 0$



Search for
test data

Verifying the Functional Behaviour of Modules (Tracey 2000)

Pre condition:

$n \geq 0$ and $n \leq 10$

```
int wrapping_counter(int n)
{
    int r;
    if (n > 10) {
        r = 0;
    } else {
        r = n + 1;
    }
    return r;
}
```

Post condition:

$n < 10 \rightarrow r = n + 1$

$n = 10 \rightarrow r = 0$

Verifying Modules (Tracey 2000)

Should
be
 $n \geq 10$

```
int wrapping_counter(int n)
{
    int r;
    if (n > 10) {
        r = 0;
    } else {
        r = n + 1;
    }

    return r;
}
```

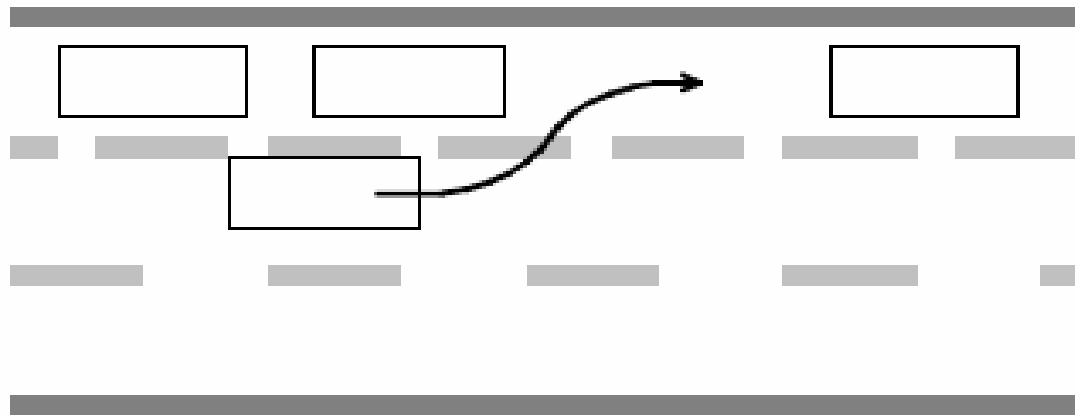
distance($n \geq 0$) and distance($n \leq 10$) and
distance($n = 10$) and distance($r \neq 0$)

Verifying Modules (Tracey 2000)

- Applied to two subsystems of a safety-critical nuclear primary protection system
- Combined 90 pages of **formal VDM specification**
- Pre- and post conditions derived manually
- 170 non-equivalent mutants generated
- **Random 90%, GA 100%**

Testing a Parking System

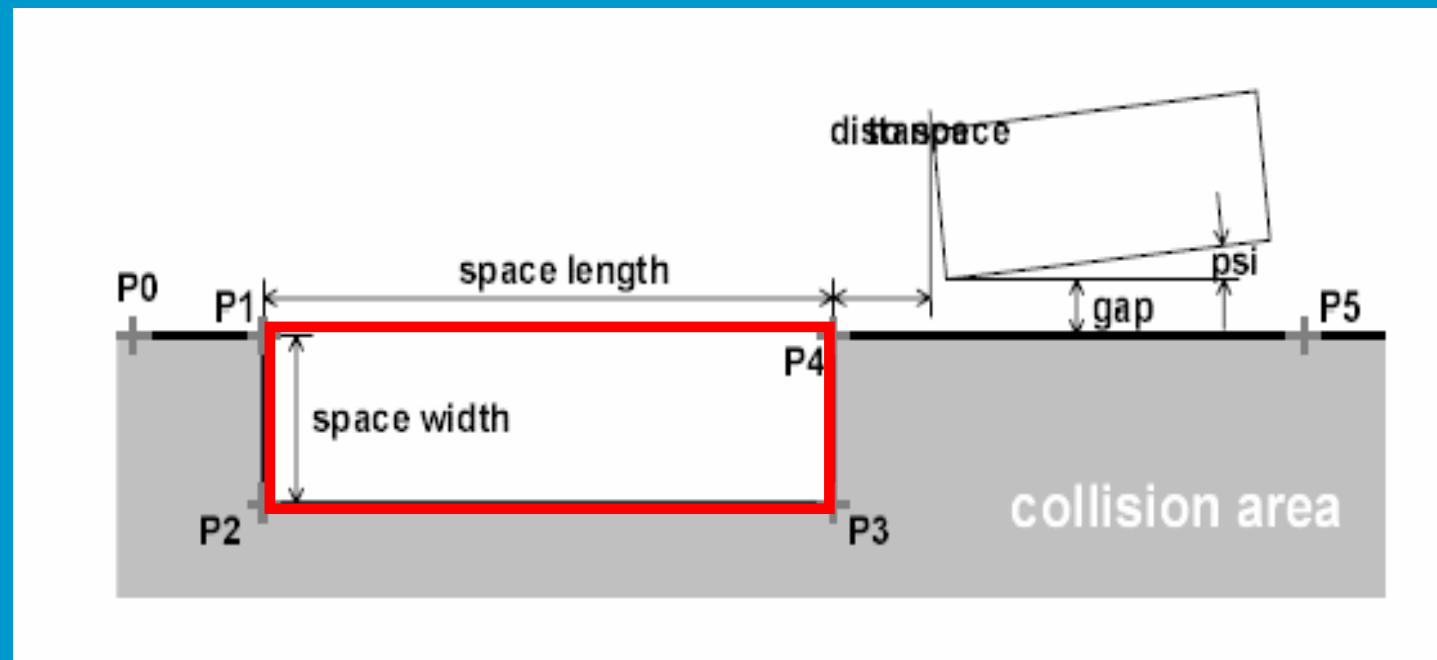
(Buehler and Wegener 2004)



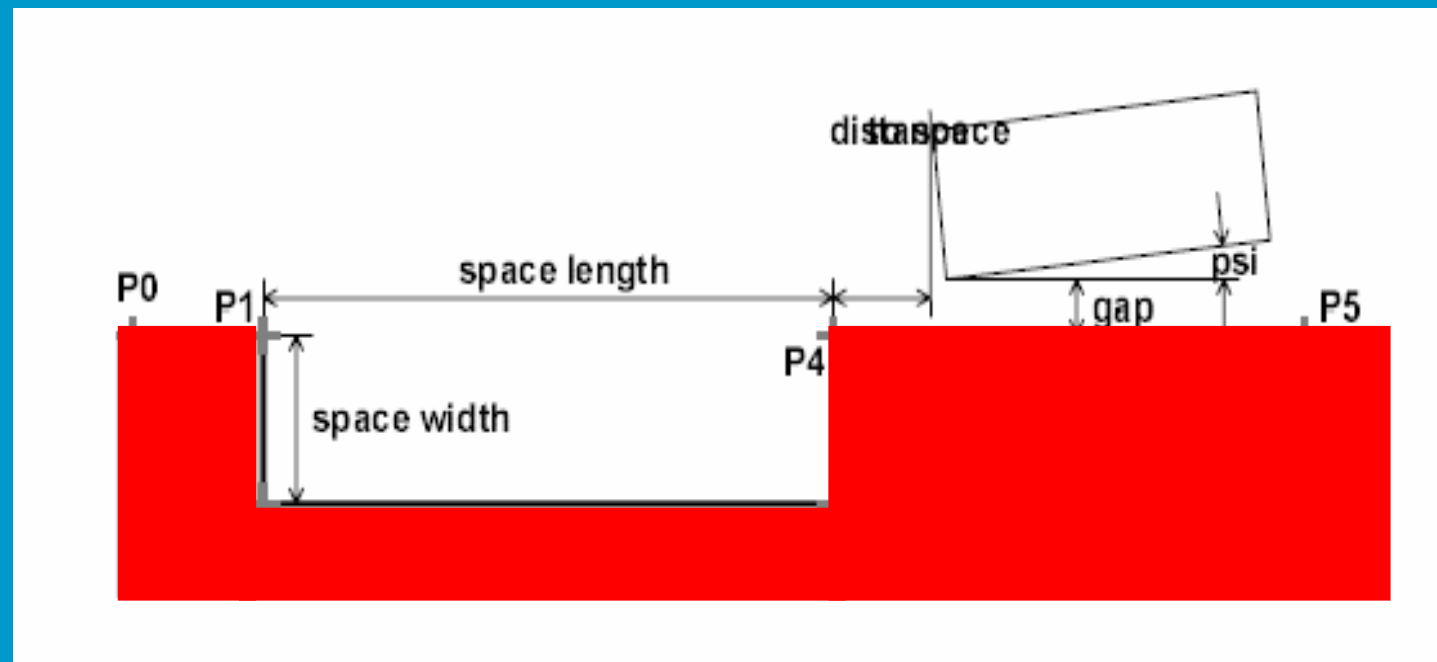
Functionality of an Autonomous Parking System

A simulated collision with the system represents a **successful test**

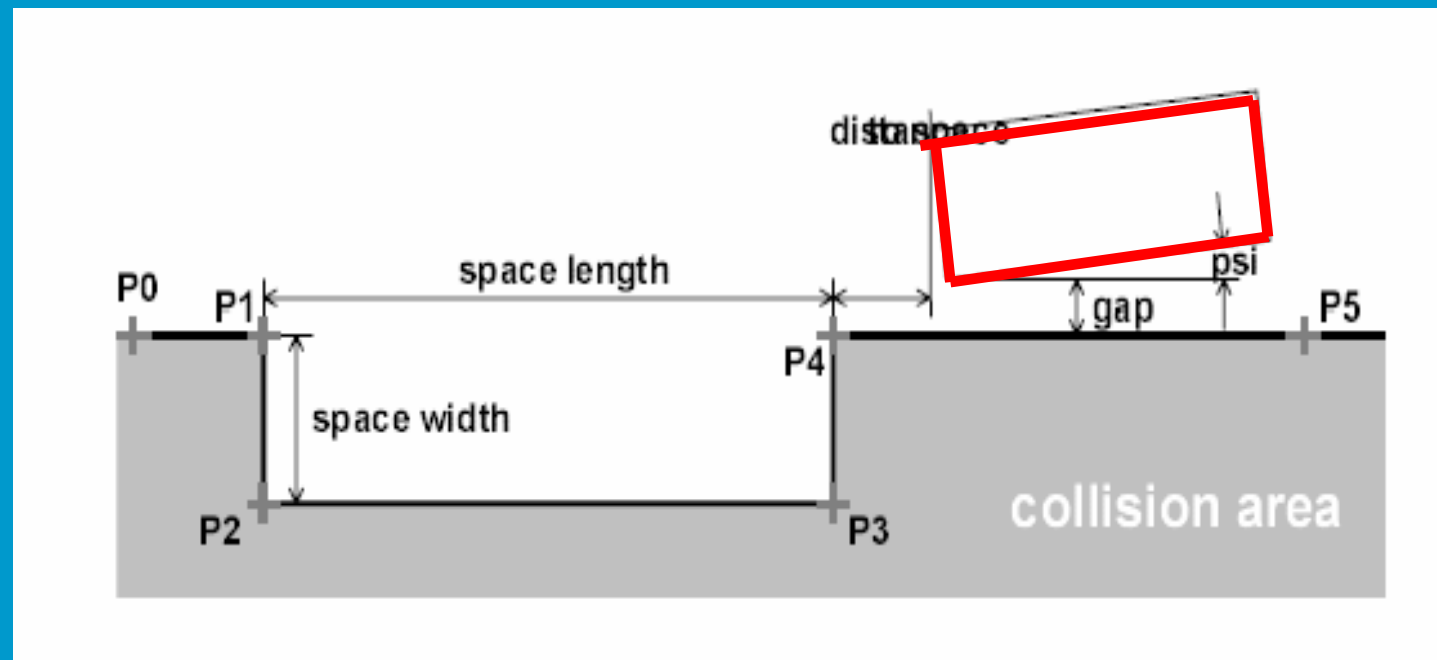
- Individuals of the search are encoded **parking scenarios**
 - parking space dimensions



- Individuals of the search are encoded **parking scenarios**
 - parking space dimensions
 - collision areas



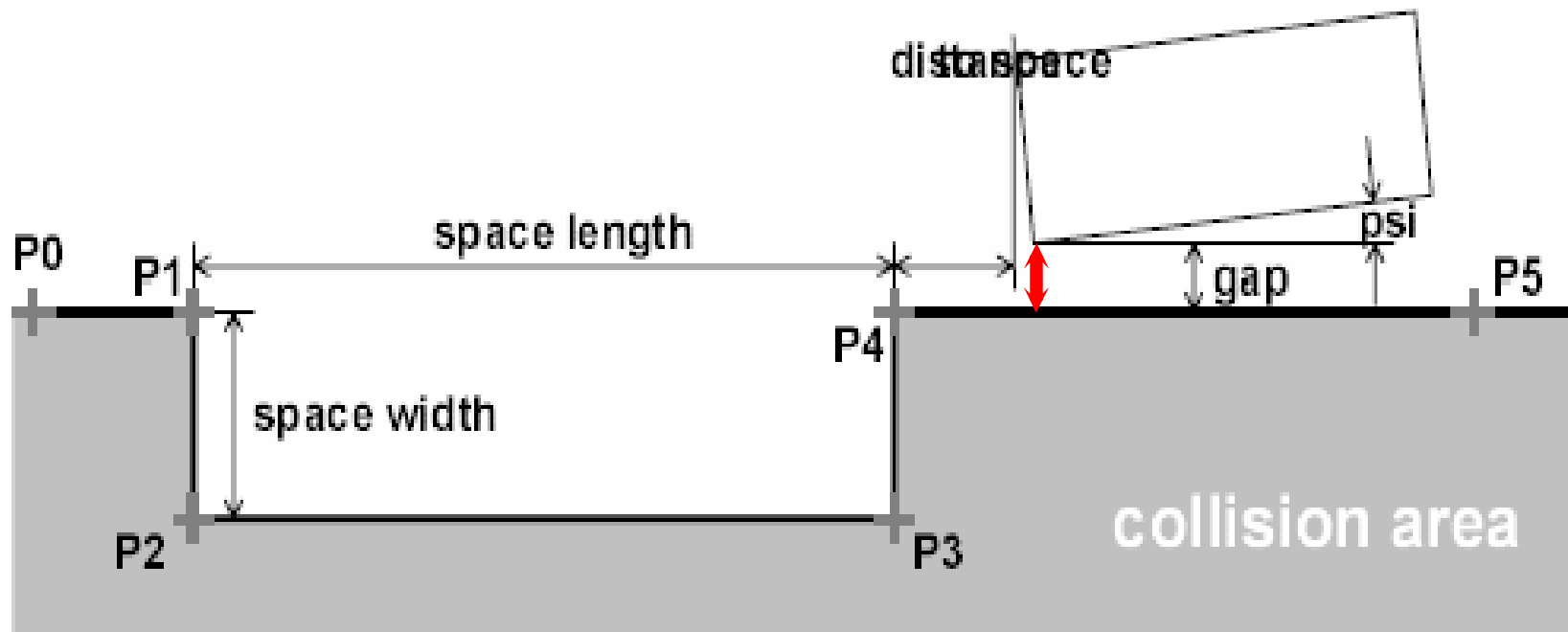
- Individuals of the search are encoded **parking scenarios**
 - parking space dimensions
 - collision areas
 - starting position of the car



Testing a Parking System

(Buehler and Wegener 2004)

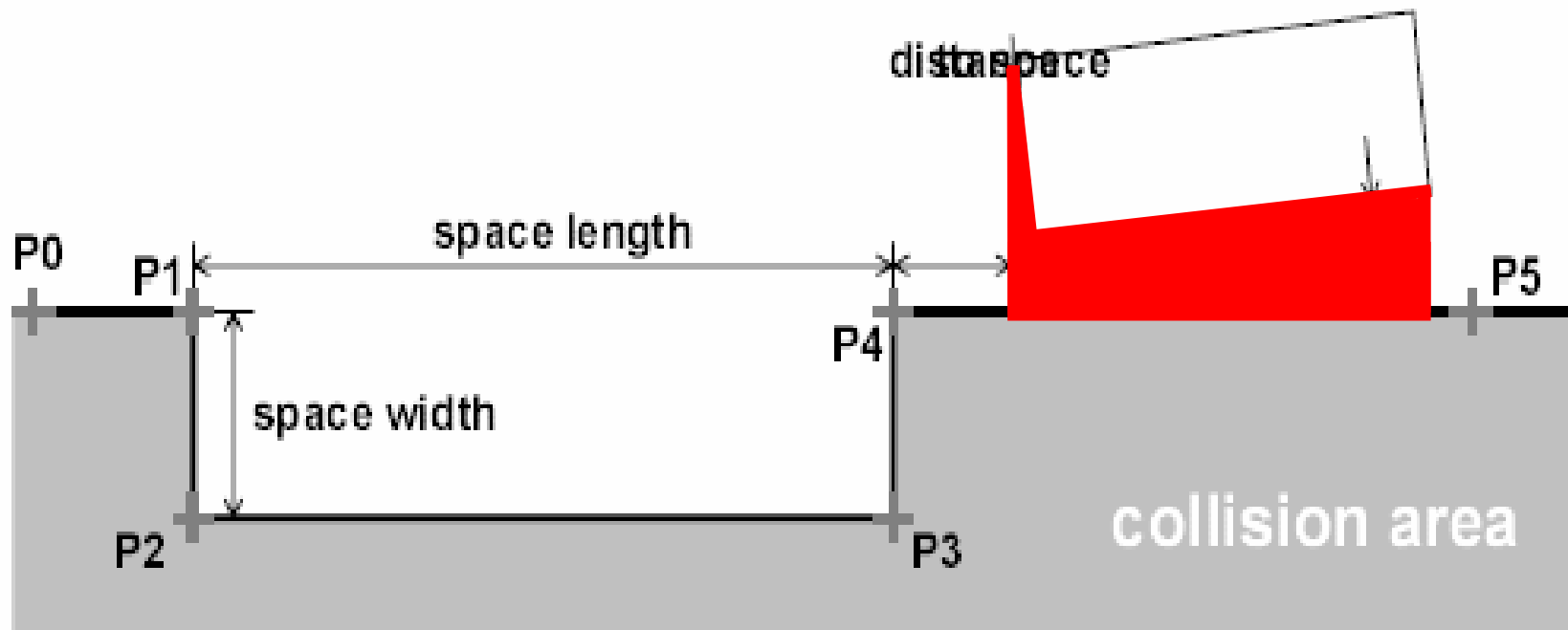
Fitness function



Testing a Parking System

(Buehler and Wegener 2004)

Fitness function



Testing a Parking System

(Buehler and Wegener 2004)

Errors discovered

- Parking space far away, close to collision area on one side
- Problems with the simulation environment



The
University
Of
Sheffield.

'Grey-Box' Testing

Assertion Testing

(Korel and Al-Yami 1996)

- Assertions are **constraints** that can be **embedded** in a program, e.g.

```
assert (d != 0);
```

```
r = n / d;
```

- When an assertion is found to be **false**, an **error has occurred**.
- When can use search-based approaches to try to **automatically falsify assertions**

Assertion Testing

(Korel and Al-Yami 1996)

- Assertions are usually formulas
- Korel's tool allowed assertions to take the form of program code
- E.g. code to check an array is sorted:

```
(*@ assertion:  
while (i < max) {  
    report_violation(a[i+1] > a[i]);  
}  
@*)
```

```
(*@ assertion:  
while (i < max) {  
    report_violation(a[i+1] > a[i]);  
}  
@*)
```

We can just adapt
structural testing
techniques

Equivalent to



```
while (i < max) {  
    if(a[i+1] > a[i]) {  
        // target  
    }  
}
```

Assertion Testing (Korel and Al-Yami 1996)

- 9 Pascal programs, with assertions embedded
- 25 mutant versions produced
- **92% of faults** were uncovered using the **assertion approach**
- **Only 8% of faults** found using **branch coverage**
- Assertions enhance fault detection for certain types of faults

Exception Testing

- Tracey (2000) extended Korel's work to automatically try and induce
 - Array out of bounds errors
 - Arithmetic exceptions (e.g. division by zero)
 - Programmer exceptions (e.g. in Java)

The End

- P. McMinn
Search-Based Software Test Data
Generation: A Survey
Software Testing Verification & Reliability,
June 2004

But there is more...

- **Testing Object Oriented Programs**
(Tonella 2004, Wappler et al 2005...)
- **Execution Time Testing** (Muehler and Wegener 1998)
- **Stress-based Testing** (Labiche et al 2005)
- **Test-case prioritisation**
(Walcott et al, 2006, Yoo et al 2007...)
- **Generating Unique Input/Output Sequences for FSM testing** (Derderian et al, 2006)

Search-Based Software Testing @ ICST 2008 - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://www.dcs.shef.ac.uk/~phil/sbtest/

Google PageRank

1st International Workshop on Search-Based Software Testing

in conjunction with ICST 2008
IEEE International Conference on Software Testing, Verification and Validation
April 9-11th 2008 ~ Lillehammer, Norway

[ICST 2008](#) | [Call for papers](#) | [Important Dates](#) | [Paper Submission](#) | [Program Committee](#) | [Keynote](#) | [Program](#)

Call for Papers

Search-based software testing is the use of random or directed search techniques (hill climbing, genetic algorithms etc.) to address problems in the software testing domain.

There has been an explosion of activity in the search-based software testing field of late, particularly in the test data generation field. Recent work has also focused on other aspects such as model-based testing, real-time testing, interaction testing, testing of service-oriented architectures and test case prioritization.

The 1st International Workshop on Search-Based Software Testing, to be held in conjunction with the 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008), aims to provide:

- a platform for interaction between search-based researchers and researchers from other areas of the software testing community with the aim of strengthening and developing search-based testing research
- an open forum for the discussion of new ideas and future directions
- a dedicated workshop for the search-based software testing community

Researchers and practitioners are invited to submit short (5 pages) or full papers (10 pages) to the workshop, describing **original research, experience or tools** relating to Search-Based Software Testing.

Papers should address a problem in the software testing domain and should approach the solution to the problem using a search strategy. Search-based techniques are taken to include (but are not limited to) random search, local search (i.e. hill climbing, simulated annealing etc.), evolutionary algorithms (i.e. genetic algorithms, evolution strategies, genetic programming), ant colony optimization and particle swarm optimization.

While experimental results are important for research papers, contributions that do not contain results, but rather present new approaches, concepts and/or theory will also be considered.

Papers will appear online in the IEEE digital library, along with the rest of the proceedings for ICST 2008.

Extended version of papers from the workshop will be eligible for the special issue of IEEE Transactions on Software Engineering on Search-Based Software Engineering (to appear in 2009).

If you have any queries, please contact the program chair: **Phil McMinn** email: p.mcminn@dcs.shef.ac.uk

Done



Search-Based Software Testing @ ICST 2008 - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://www.dcs.shef.ac.uk/~phil/sbtest/

1st International Workshop on
Search-Based Software Testing
in conjunction with ICST 2008
IEEE International Conference on Software Testing, Verification and Validation
April 9-11th 2008 ~ Lillehammer, Norway

ICST 2008 | Call for papers | Important Dates | Paper Submission | Program Committee | Keynote | Program

Call for Papers

Search-based software testing is the use of random search algorithms etc.) to address problems in the software testing domain.

There has been an explosion of activity in the search-based software testing field of late, particularly in the test data generation field. Recent work has also focused on other aspects such as model-based testing, real-time testing, interaction testing, testing of service-oriented architectures and test case prioritization.

The 1st International Workshop on Search-Based Software Testing is part of the 17th International Conference on Software Testing, Verification and Validation (ICST 2008) to be held in Lillehammer, Norway, April 9-11, 2008.

- a platform for interaction between researchers and practitioners in the software testing community
- an open forum for the presentation of research results
- a dedicated workshop on search-based software testing

Researchers and practitioners are invited to submit papers on their research, experience or theoretical work in the area of search-based software testing.

Papers should address a problem in the software testing domain and should approach the solution to the problem using a search strategy. Search-based techniques are taken to include (but are not limited to) random search, local search (i.e. hill climbing, simulated annealing etc.), evolutionary algorithms (i.e. genetic algorithms, evolution strategies, genetic programming), ant colony optimization and particle swarm optimization.

While experimental results are important for research papers, contributions that do not contain results, but rather present new approaches, concepts and/or theory will also be considered.

Papers will appear online in the IEEE digital library, along with the rest of the proceedings for ICST 2008.

Extended version of papers from the workshop will be eligible for the special issue of IEEE Transactions on Software Engineering on Search-Based Software Engineering (to appear in 2009).

If you have any queries, please contact the program chair: **Phil McMinn** email: p.mcminn@dcs.shef.ac.uk

Done



<http://www.dcs.shef.ac.uk/~phil/sbtest>
Paper submission: Jan 2008



ICST 2008
First International Conference on Software Testing, Verification, and Validation
Lillehammer, Norway April 9-11, 2008