

# Search-based Testing of Procedural Programs: Iterative Single-Target or Multi-Target Approach?



Simone Scalabrino



Giovanni Grano



Dario Di Nucci



Rocco Oliveto



Andrea De Lucia

“

The overall cost of testing has been estimated at being at least half of the entire development cost, if not more.

*Boris Beizer*

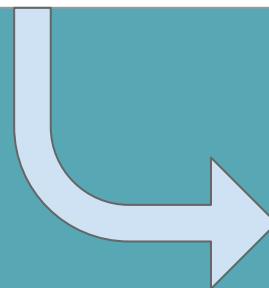
Beizer. Software testing techniques. 2003. Dreamtech Press.

“

Software developers only spend a quarter of their work time engineering tests, whereas they think they test half of their time.

*Beller et al.*

```
700 gdouble
701 gradient_calc_bilinear_factor (gdouble dist,
702                                     gdouble *vec,
703                                     gdouble offset,
704                                     gdouble x,
705                                     gdouble y)
706 {
707     if (dist == 0.0)
708     {
709         return 0.0;
710     }
711     else
712     {
713         gdouble r;
714         gdouble rat;
715
716         /* Calculate linear offset from the start line outward */
717
718         offset = offset / 100.0;
719
720         r = vec[0] * x + vec[1] * y;
721         rat = r / dist;
722
723         if (fabs (rat) < offset)
724             return 0.0;
725         else if (offset == 1.0)
726             return (rat == 1.0) ? 1.0 : 0.0;
727         else
728             return (fabs (rat) - offset) / (1.0 - offset);
729     }
730 }
```



```
1 #include <stdlib.h>
2 #include <check.h>
3 START_TEST(ocelot_testcasel)
4 {
5     double __val0 = 7883.606068766041;
6     double __val1 = 7472.893282147905;
7     double __val2 = 8109.302460798321;
8     double __val3 = -5018.78129027117;
9
10    double __array0[2] = {-6866.205482964684, -350.3045266881127};
11
12    void* __ptr0 = &__array0;
13
14    gdouble __arg0 = __val0;
15    gdouble __arg2 = __val1;
16    gdouble __arg3 = __val2;
17    gdouble __arg4 = __val3;
18    gradient_calc_bilinear_factor(__arg0,__ptr0,__arg2,__arg3,__arg4);
19
20    /* REPLACE THE ASSERTION BELOW */
21    ck_assert_str_eq("OK", "OK");
22}
23 END_TEST
24
```

## Test case generation

# Test case generation SBST Methodologies

**Automatic Generation of Floating-Point Test**

WEBB MILLER AND DAVID L. SPOONER

**Abstract** In that it is not guaranteed to produce a good path whenever such is the other hand, we know of no guaranteed scheme whose execution time does not, in general, grow exponentially with the length of the path.

**Index Terms** Automatic test data generation, branching, data constraints, execution paths, software evaluation, synthesis.

**Numerical Maximization Methods Generating Test Data**

Given the problem of generating floating-point test data, we propose a numerical maximization method that can be used to generate test data that can be successfully solved with each maximization technique.

**An Automated Framework for Structural Test-Data Generation**

Nigel Tracey, John Clark, Keith Mander, Julian McDermid

Department of Computer Science, University of York, Heslington, York, YO1 5DD, England. Tel.: +44 1904 432749 [njt, jcl, kmndr, jum]@york.ac.uk

**Abstract** This paper presents an alternative to the current state-of-the-art for generating test data for programs. The approach is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Journal of Software Testing, Verification and Reliability, 1990, to appear. © 1990 copyright Jim Wegener\*, Andre Baresel, Hartmen Stamer**

**Automated Software Test Data Generation**

HODIÁN KOREL, MEMBER, IEEE

**Abstract** Test data generation in program testing is the process of identifying a set of test data which satisfies given criteria. Most of the existing test data generators [5], [9], [10], [14], [16] are symbolic, i.e., they frequently require complex algebraic manipulations, representation of arrays. In this paper we present an alternative to the symbolic approach for generating test data. It is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Information and Software Technology** 43 (2001) 841–854  
www.elsevier.com/locate/infsof

**Test-Data Generation Using Genetic Algorithms**

Roy P. Parhami, Mary Jean Harrold, Robert R. Peck

Department of Computer Science, Clemson University, SC USA 29631-1900; Department of Computer and Information Science, The Ohio State University, Columbus, OH USA 43210-1277; C. parhami@clemson.edu, mjh@cs.clemson.edu, rrp@ccs.ohio-state.edu

**Abstract** This paper presents an alternative to the current state-of-the-art for generating test data for programs. The approach is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Journal of Software Testing, Verification and Reliability, 1990, to appear. © 1990 copyright Jim Wegener\*, Andre Baresel, Hartmen Stamer**

**It Does Matter How You Normalise the Branch Distance in Search Based Software Testing**

Andrea Arcuri

Software Research Laboratory, P.O. Box 124, Lysaker, Norway. arcuri@atmika.no

**Abstract** This paper compares two different ways of normalizing branch distances in search-based software testing. The first way is based on the number of branches in the program, while the second is based on the number of nodes. The results show that the second way is more effective than the first one. This is because the second way takes into account the fact that some branches are more important than others. The results also show that the second way is more efficient than the first one. This is because the second way is more likely to find a solution that is closer to the target than the first way.

**Evolutionary Testing of Classes**

Paolo Tonella

Centro per la Ricerca Scientifica e Tecnologica, 38050 Posta (Trento), Italy. tonella@itc.it

**Abstract** This paper presents an alternative to the current state-of-the-art for generating test data for programs. The approach is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Reformulating Branch Coverage as a Many-Objective Optimization Problem**

De Panigrahi<sup>1</sup>, Patum Molesha Kafekci<sup>1</sup>, Paolo Tonella<sup>2</sup>

<sup>1</sup>Datta University of Technology, The Netherlands. <sup>2</sup>University of Trento, Italy. aarcuri@atmika.no

**Abstract** This paper presents an alternative to the current state-of-the-art for generating test data for programs. The approach is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Whole Test Suite Generation**

Gordon Fraser, Member, IEEE, and Andrea Arcuri, Member, IEEE

**Abstract** This paper presents an alternative to the current state-of-the-art for generating test data for programs. The approach is based on test-case generation methods, and it is able to generate test data for programs that have been instrumented with symbolic annotations. Test data is derived for the program under test, and the generated test data is used to verify that the program is correct. Once a set of test cases has been generated, the user can then verify that the results of the execution of the program are as expected. The approach is general and can be applied to a wide range of programs.

**Example Stack Implementations**

public class Stack {  
 int value = new int[3];  
 int size = 0;  
 void push(int x) {  
 if (size < value.length) {  
 value[size] = x; //Requires a full stack;  
 } else {  
 value[size] = value[size] + x; //Requires an overflow;  
 }  
 size++;  
 }  
 void pop() {  
 if (size < value.length) {  
 value[size] = value[size] - 1; //Requires a full stack;  
 } else {  
 value[size] = value[size] - 1; //Requires an underflow;  
 }  
 size--;  
 }  
}

**1. INTRODUCTION**

It is widely recognized that software testing is an essential component of any successful software development process. A variety of software testing methodologies have been proposed and a definition of the expected outcome. Many techniques to automatically generate tests have been proposed over the years. One of the most common approaches is the use of random testing, which is a form of automated test data generation. However, the research challenge is to generate test cases that will effectively validate the system in response to the user's requirements. This observation has led to the development of a number of test case generation systems [1–10]. Functional [11] and Non-functional Testing [12].

The field of generating test cases with random testing is a recent survey [13]. However, despite this considerable publication rate, there is very little work on the oracle cost problem. That is, how many test cases are required to achieve a certain level of coverage? There is also a lack of work on the effectiveness of searching for good test inputs that can be used in these very memory constrained scenarios. It is also not address the quality of the generated test cases. That is, how many good test cases are produced by random testing?

**2. Introduction**

In this paper, we introduce three algorithms for solving the oracle cost problem: the test data set problem, the oracle cost problem, and the oracle coverage problem. We also introduce a new formulation of the oracle cost problem that is based on the number of test cases required to achieve a certain level of coverage, while minimizing the number of test cases, and the oracle coverage problem, while maximizing the number of test cases required to achieve a certain level of coverage.

**3. Oracle Cost Problem**

One of the main challenges in generating test cases is to find ways to make the use of the test data set problem. This problem is to find the minimum number of test cases required to achieve a certain level of coverage. This problem is a well-known problem in the field of software testing and the analysis of the System Under Test (SUT). In order to do this, we seek test inputs that cover a targeted branch in the SUT, while also

**4. Oracle Coverage Problem**

Another challenge in generating test cases is to find ways to make the use of the oracle coverage problem. This problem is to find the maximum number of test cases required to achieve a certain level of coverage. This problem is a well-known problem in the field of software testing and the analysis of the System Under Test (SUT). In order to do this, we seek test inputs that cover a targeted branch in the SUT, while also

**5. Conclusion**

This paper presents three algorithms for solving the oracle cost problem, the oracle coverage problem, and the oracle coverage problem. We also introduce a new formulation of the oracle cost problem that is based on the number of test cases required to achieve a certain level of coverage, while minimizing the number of test cases, and the oracle coverage problem, while maximizing the number of test cases required to achieve a certain level of coverage.



EVSUITE

# AUSTIN

I nput G eneration U sing A utomated N ovel A lgorithms

eToc - evolutionary Testing of classes

Test case generation  
SBST Tools





OCELOT

Optimal Coverage sEarch-based  
tooL for sOftware Testing



## Test Case Generation for C

Fully Implemented in Java and C (through JNI)

Based on JMetal Framework

Structs and Pointers Handling

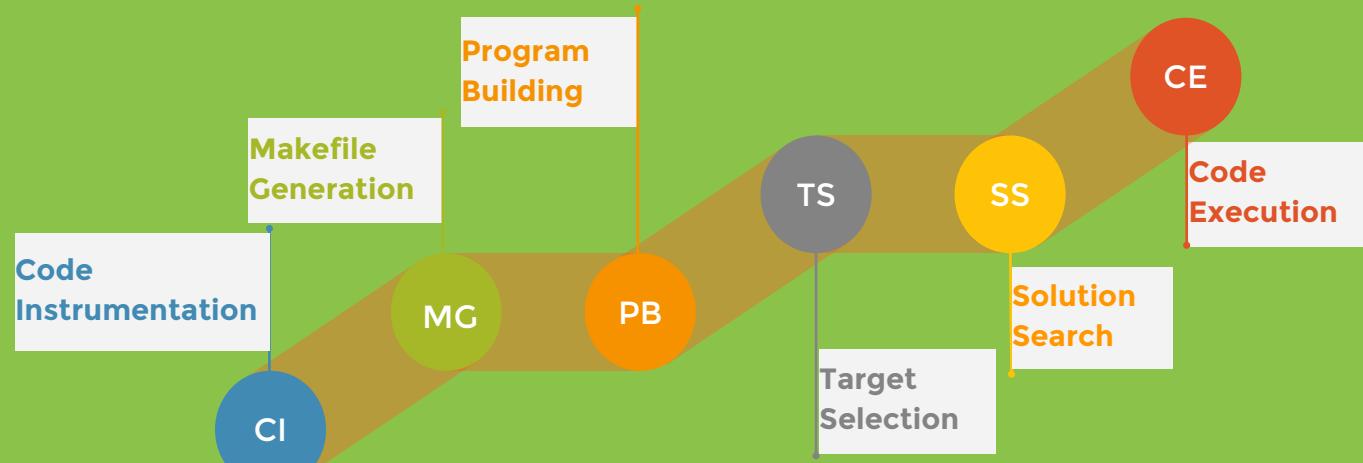
Check Unit Testing Framework

OCELOT  
Features

2015	Change	Programming Language	Ratings	Change
		Java	18.236%	-1.33%
		C	10.955%	-4.67%
		C++	6.657%	-0.13%
		C#	5.493%	+0.58%
		Python	4.302%	+0.64%
▲		JavaScript	2.929%	+0.59%
▼		PHP	2.847%	+0.32%
▲		Assembly language	2.417%	+0.61%
▼		Visual Basic .NET	2.343%	+0.28%
▼		Perl	2.333%	+0.43%
▲		Delphi/Object Pascal	2.169%	+0.42%
		Ruby	1.965%	+0.18%
▲		Swift	1.930%	+0.74%
▼		Objective-C	1.849%	+0.03%
▲		MATLAB	1.826%	+0.65%
		Groovy	1.818%	+1.31%
		Visual Basic	1.761%	+0.23%
		R	1.684%	+0.64%
		Go	1.625%	+1.37%

OCELOT  
Why C?





# OCELOT

## Process Overview

## MOSA

### Multi-Objective Sorting Algorithm\*

*Panichella et al.*

## LIPS

### Linearly Independent Path based Search

*Scalabrino et al.*

OCELOT

Target Selection  
Algorithms



# Reformulating Branch Coverage as a Many-Objective Optimization Problem

Annibale Panichella\*, Fitsum Meshesha Kifetew<sup>†‡</sup>, Paolo Tonella<sup>‡</sup>

\*Delft University of Technology, The Netherlands

<sup>†</sup>University of Trento, Trento, Italy

<sup>‡</sup>Fondazione Bruno Kessler, Trento, Italy

a.panichella@tudelft.nl, kifetew@fbk.eu, tonella@fbk.eu

**Abstract**—Test data generation has been extensively investigated as a search problem, where the search goal is to maximize the number of covered program elements (e.g., branches). Recently, the whole suite approach, which combines the fitness functions of single branches into an aggregate, test suite-level fitness, has been demonstrated to be superior to the traditional single-branch at a time approach.

In this paper, we propose to consider branch coverage directly as a many-objective optimization problem, instead of aggregating multiple objectives into a single value, as in the whole suite approach. Since programs may have hundreds of branches (objectives), traditional many-objective algorithms that are designed for numerical optimization problems with less than 15 objectives are not applicable. Hence, we introduce a novel highly scalable many-objective genetic algorithm, called MOSA (Many-Objective Sorting Algorithm), suitably defined for the

targeting infeasible goals will by definition fail and the related effort is wasted [3]). To overcome these limitations, Fraser and Arcuri [3] have recently proposed the *whole suite* (WS) approach that uses a search strategy based on (i) a different representation of candidate solutions (i.e., test suites instead of single test cases); and (ii) a new single fitness function that considers all testing goals simultaneously. From the optimization point of view, WS applies the sum scalarization approach that combines multiple target goals (i.e., multiple branch distances) into a single, scalar objective function [4], thus allowing for the application of single-objective metaheuristics such as standard GA.

Previous works on numerical optimization have shown that the sum scalarization approach to many-objective optimization has a number of drawbacks, among which the main one is that

## Fitness function reformulation

Find a set of test cases that optimizes the branch coverage of each branch

## Dominance and Pareto optimality

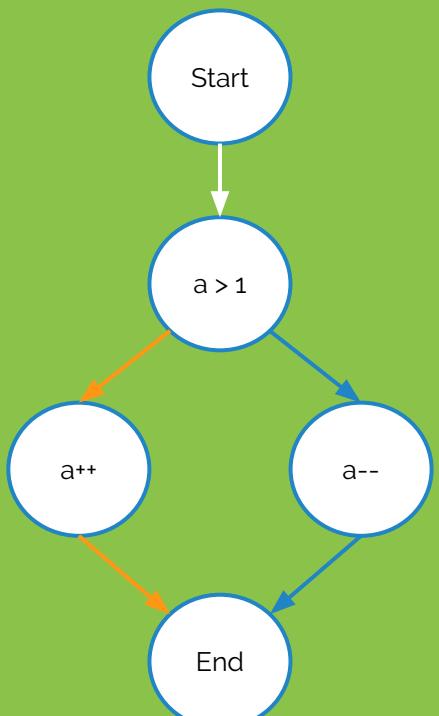
Each solution is evaluated in terms of Pareto dominance and optimality,

## Preference Sorting

A new ranking algorithm for sorting the solutions

# Many-Objective Sorting Algorithm

# Linearly Independent Path based Search



## Single target selection algorithm

Inspired by McCabe baseline method and Dynamic Symbolic Execution

- Starts with random test data ( $t_0$ )
- Selects a new target at each iteration

## Seeding

Final population of the previous iteration is reused

## Collateral coverage

Considers the targets serendipitously achieved

## Search budget optimization

Allocates  $SB_i/n_i$  evaluations for each target

## Independent of the search algorithm

Current implementation based on GA



## EMPIRICAL EVALUATION



MOSA

VS

LIPS

RQ1  
Effectiveness

Branch  
Coverage

RQ2  
Efficiency

Execution  
Time

RQ3  
Oracle Coast

Test Suite  
Size

Research Questions



## Design

Context  
Settings  
Experiment Details

35 C functions  
605 branches

Gimp: Gnu Image Manipulation Program  
GSL: Gnu Scientific Library  
SGLIB: a generic library for C  
spice: analogue circuit simulator

Population Size	100
Crossover Rate	0.90
Mutation Rate	1/#variables
Search Budget	200.000

30 runs  
Average Values  
Wilcoxon's Test (p-value 0.05)  
Vargha-Delaney Test

## Results RQ1: Effectiveness

	MOSA	LIPS
Overall Branch Coverage	84,73%	86.29%
Cases in which is better	<sup>1</sup>	<sup>10<sup>2</sup></sup>

<sup>1</sup>1 case with large effect size

<sup>2</sup>8 cases with medium/large effect size

## Results RQ2: Efficiency

	MOSA	LIPS
Average Execution Time	14.80s	5.03s
Cases in which is better	0	35 <sup>1</sup>

<sup>1</sup> with large effect size

## Results RQ2: Efficiency

	MOSA	LIPS
Average Execution Time	14.80s	5.03s
Cases in which is better	0	35 <sup>1</sup>

Too much time for ranking the Pareto Fronts!

<sup>1</sup> with large effect size

	MOSA	LIPS
Average # Test Cases	4.4	6.1
Cases in which is better	35 <sup>1</sup>	0

## Results RQ3: Oracle Cost

<sup>1</sup>33 cases with large effect size

## Results RQ3: Oracle Cost

	MOSA	LIPS
Average # Test Cases	4.4	6.1
Cases in which is better	35 <sup>1</sup>	0

LIPS does not directly  
handle the oracle  
problem!

<sup>1</sup>33 cases with large effect size

**Worse than LIPS in terms of branch coverage**  
**Worse than LIPS in terms of execution time<sup>1</sup>**

**LIPS\***  
no collateral coverage



	MOSA	LIPS*
Average # Test Cases	4.4	4.25
Cases in which is better	6	7

<sup>1</sup> Although better than MOSA

	MOSA <sup>1</sup>	LIPS
Average # Test Cases	3.61	3.66
Cases in which is better	6 <sup>2</sup>	2 <sup>3</sup>

## LIPS + minimization

<sup>1</sup> For a fair comparison the minimization was applied also to MOSA suites  
<sup>2</sup> 4 cases with medium/large effect size  
<sup>3</sup> 1 case with large effect size

## LIPS + minimization

	MOSA <sup>1</sup>	LIPS
Average # Test Cases	3.61	3.66
Cases in which is better	6 <sup>2</sup>	2 <sup>3</sup>

Minimization  
execution  
time < 1s

<sup>1</sup> For a fair comparison the minimization was applied also to MOSA suites

<sup>2</sup> 4 cases with medium/large effect size

<sup>3</sup> 1 case with large effect size

## **Conclusions**

```

100    void
101    rotGradient_calc_kilometer_factor (double* dist,
102                                     double* offset,
103                                     double* offset,
104                                     double* p1
105    {
106        if (dist == 0.0)
107        {
108            *offset = 0.0;
109            return 0.0;
110        }
111        else
112        {
113            *offset = dist;
114            /* calculate linear offset from the start time reward */
115            offset = offset / 100.0;
116            offset = offset * vvec11 * p1;
117            p1 = vvec10 + vvec11 * p1;
118            rot = p1 / dist;
119            rotare_r11 = rot;
120            rotare_r12 = rot;
121            rotare_r13 = rot;
122            rotare_r14 = rot;
123            rotare_r15 = rot;
124            rotare_r16 = rot;
125            rotare_r17 = rot;
126            rotare_r18 = rot;
127            rotare_r19 = rot;
128            rotare_r20 = rot;
129            rotare_r21 = rot;
130            rotare_r22 = rot;
131            rotare_r23 = rot;
132            rotare_r24 = rot;
133            rotare_r25 = rot;
134            rotare_r26 = rot;
135            rotare_r27 = rot;
136            rotare_r28 = rot;
137            rotare_r29 = rot;
138            rotare_r30 = rot;
139        }
140        return (dist * rot - offset) / (0.0 + offset);
141    }

```

```

1 #include <assert.h>
2 #include <check.h>
4 /* START_TEST(cacof, testcase) */
5 double _val1 = -0.00010000000000000000;
6 double _val2 = -0.00010000000000000000;
7 double _val3 = 0.00010000000000000000;
8 double _val4 = 0.00010000000000000000;
9 double _array[2] = {0.00010000000000000000, -0.00010000000000000000};
12 void* __ptrb = b_.arrayb;
13
14 double __argb = val1;
15 double __arg2 = -val1;
16 double __arg3 = val2;
17 double __arg4 = -val2;
18
19 /* REPLACE THE ASSERTION BELOW */
20 ck_assert_int_eq((int)_array[0], 0);
21 ck_assert_int_eq((int)_array[1], 0);
22
23 END_TEST
24

```

Test case generation

## Conclusions

```

100    phasor;
101    real gradient_calc_kilometer_factor(phasor &dist,
102                                         real offset,
103                                         real offset,
104                                         phasor &t);
105
106    if (offset == 0.0)
107    {
108        return 0.0;
109    }
110    else
111    {
112        phasor &vec0 = dist;
113        /* calculate linear offset from the start time offset */
114        offset = offset / 100.0;
115        offset = offset * vec0.g;
116        vec0 = vec0.D + vec0.v * vec0.t + g;
117        dist = r / dist;
118        return (vec0.r - offset) / dist;
119    }
120    return (vec0.r - offset) / dist;
121}

```

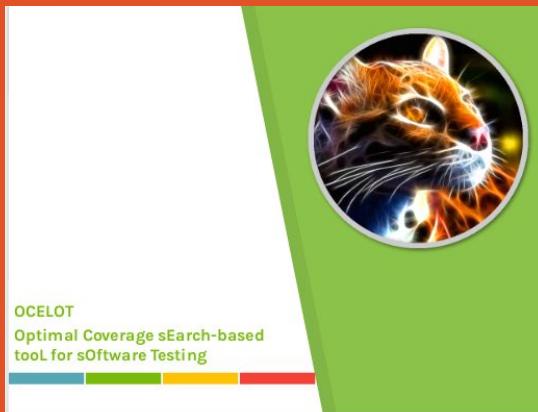
Test case generation

---

```

1 #include <assert.h>
2 #include <check.h>
3
4 /*START_TEST(ocelet, test_case1)
5 double _val1 = -0.00010000000000000000;
6 double _val2 = -0.00010000000000000000;
7 double _val3 = 0.00010000000000000000;
8 double _val4 = 0.00010000000000000000;
9 double _array[2] = {0.00010000000000000000, -0.00010000000000000000};
10 void* __ptr64 = &_array[0];
11
12 ophandle __phobj;
13 ophandle __arg0 = __val1;
14 ophandle __arg1 = __val2;
15 ophandle __arg2 = __val3;
16 ophandle __arg3 = __val4;
17 ophandle __arg4 = __val1;
18
19 /* REPLACE THE ASSERTION BELOW */
20 ck_assert_int_eq(0, 0);
21 ck_assert_str_eq("0", "0");
22
23 END_TEST
24

```



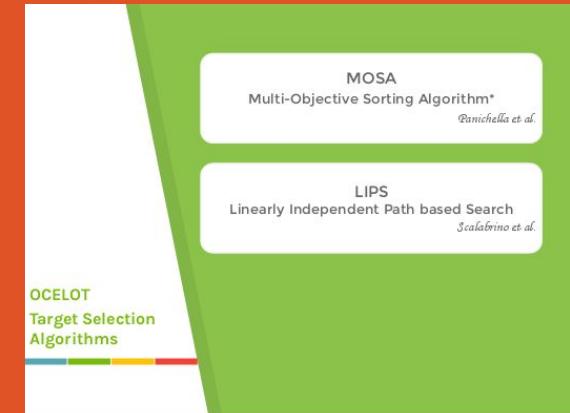
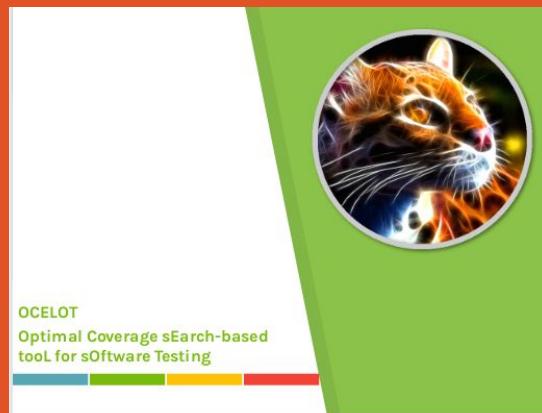
## Conclusions

```

100    private
101    void gradient_calc_bilinear_factor (private float_& dist,
102                                         private float_& offset,
103                                         private float_& p1
104                                         )
105    {
106        if (start == 0.0)
107        {
108            // return 0.0;
109        }
110        else
111        {
112            // calculate linear offset from the start time offset
113            offset = offset / 100.0;
114            offset = offset * 0.01 + p1;
115            p1 = r / dist;
116            offset = offset * p1;
117            offset = offset * offset;
118            offset = offset * 0.01 + 1.0 + p1;
119            offset = offset * offset;
120            offset = offset / (0.01 + offset);
121        }
122    }
123
124    private
125    void check (void)
126    {
127        TESTCASE("check");
128
129        double val0 = -0.00010000000000000002;
130        double val1 = 0.00010000000000000002;
131        double val2 = 0.00010000000000000002;
132        double val3 = 0.00010000000000000002;
133
134        double _array[2] = {0.00010000000000000002, -0.00010000000000000002};
135        void* __ptr = &_array[0];
136
137        _phandle_arg0 = val0;
138        _phandle_arg1 = val1;
139        _phandle_arg2 = val2;
140        _phandle_arg3 = val3;
141
142        /* REPLACE THE ASSERTION BELOW */
143        ck_assert_int_eq(0, 0);
144
145    }
146
147 END_TEST
148

```

Test case generation



## Conclusions

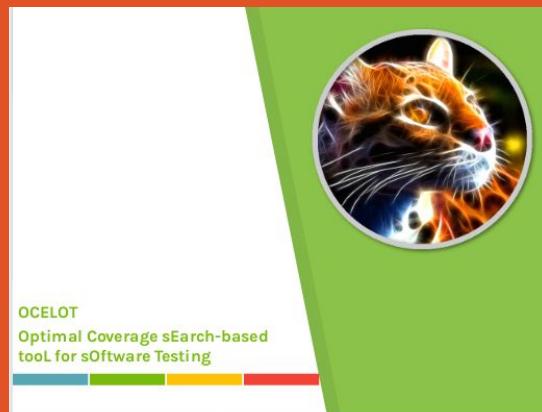
```

100 private
101 gradient_calc_kilometer_factor (private float,
102                                     private float,
103                                     private offset,
104                                     private st
105                                     );
106
107     if (start <= 0.0)
108     {
109         return 0.0;
110     }
111     else
112     {
113         private float r;
114
115         /* calculate corner offset from the start time reward */
116         offset = offset / 100.0;
117
118         r = vecDot + v * vecDot * p;
119
120         return (r - offset) / (1.0 + r);
121     }
122
123     private float max (float a, float b)
124     {
125         return (a > b) ? a : b;
126     }
127
128     private float min (float a, float b)
129     {
130         return (a < b) ? a : b;
131     }
132
133     private void calc_kilometer_factor (float offset);
134
135     private void calc_kilometer_factor (float offset)
136     {
137         private float r;
138
139         r = vecDot + v * vecDot * p;
140
141         offset = r / dist;
142
143         max (offset, -1.0);
144         min (offset, 1.0);
145
146         offset = offset / (1.0 + r);
147
148         return (max (float) - offset) / (1.0 - offset);
149     }
150 }

```

Test case generation

Research Questions

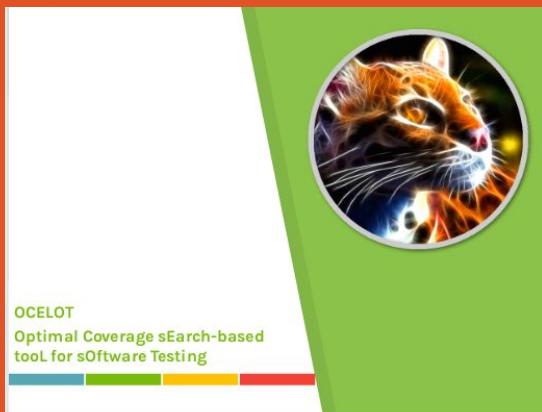


# Conclusions

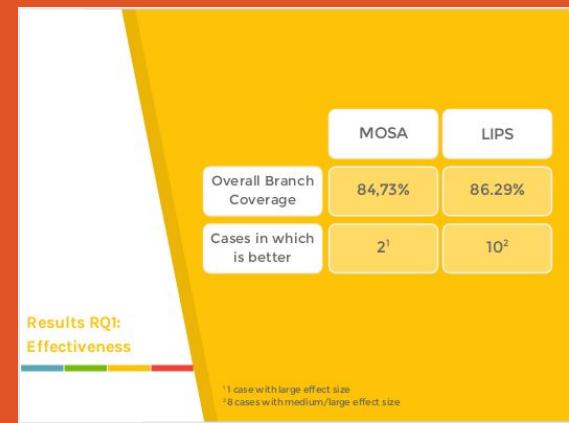
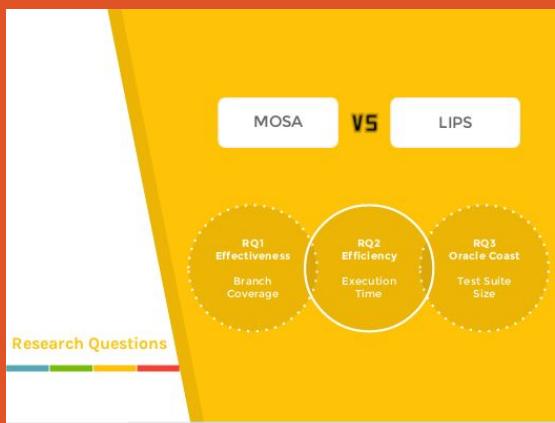
```

100 private
101 gradient_calc_kilometer_factor (private float,
102    private float,
103    private offset,
104    private st
105)
106 {
107     if (start <= 0.0)
108     {
109         return 0.0;
110     }
111     else
112     {
113         private float offset;
114         /* calculate linear offset from the start time reward */
115         offset = offset / 100.0;
116         x = val10 + x * val11 + y;
117         zet = r / dist;
118         offset = offset * zet;
119         reward = -1.0;
120         reward -= 1.0 * zet;
121         reward += offset;
122     }
123     return (float) (int) (offset / 0.01 + offset);
124 }

```



# Conclusions

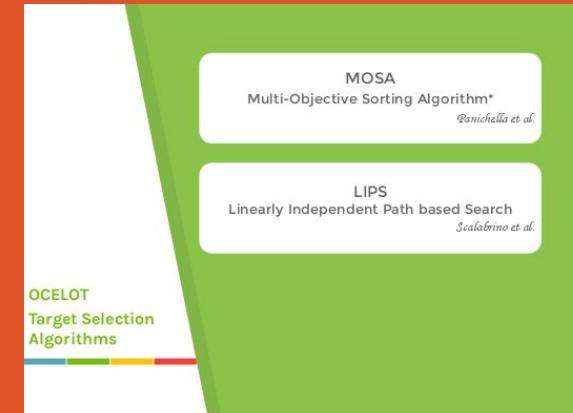
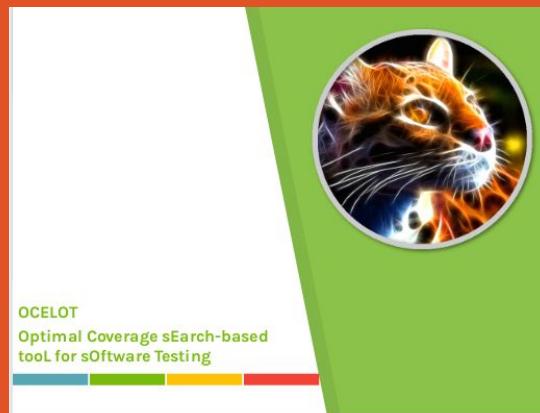


```

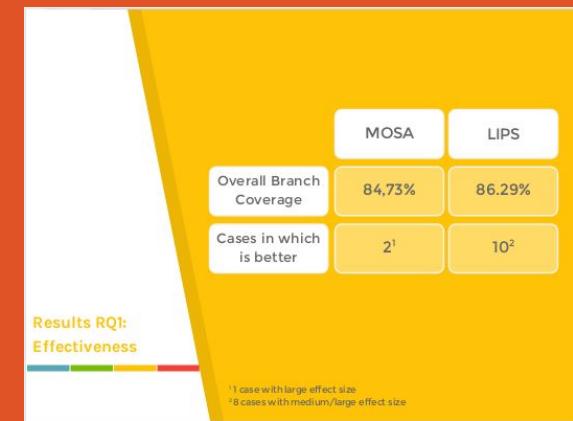
100    void calculate_offset(double dist,
101        double offset,
102        double offset_min,
103        double offset_max);
104    {
105        if (dist <= 0.0)
106        {
107            return 0.0;
108        }
109        else
110        {
111            double offset_rnd = rand();
112            /* calculate linear offset from the start time reward */
113            offset = offset / 100.0;
114            offset += offset_rnd * 0.01;
115            offset = val1 * offset + val2 * offset_min + val3 * offset_max;
116            offset = offset / dist;
117            offset = offset * 1.0;
118            offset_min = -1.0 + 1.0 * offset;
119            offset_max = 1.0 + 1.0 * offset;
120            /* return final offset / (1.0 - offset) */
121        }
122    }

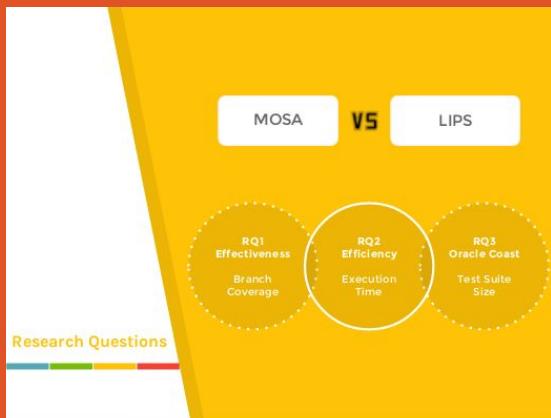
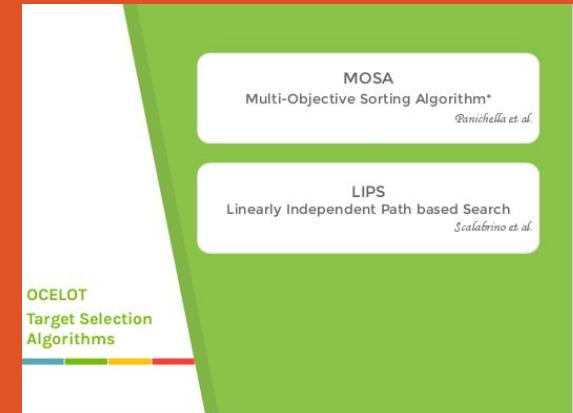
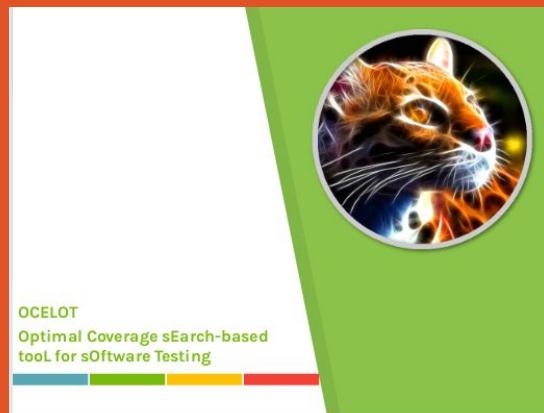
```

**Test case generation**

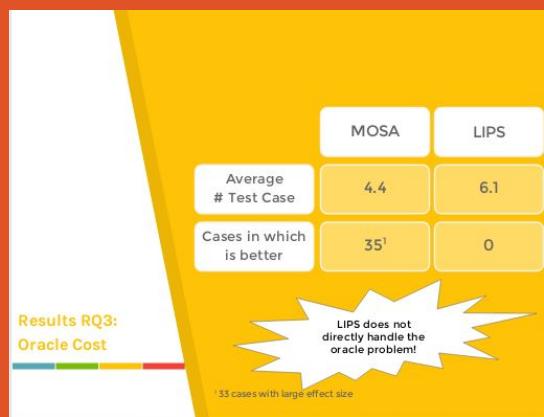
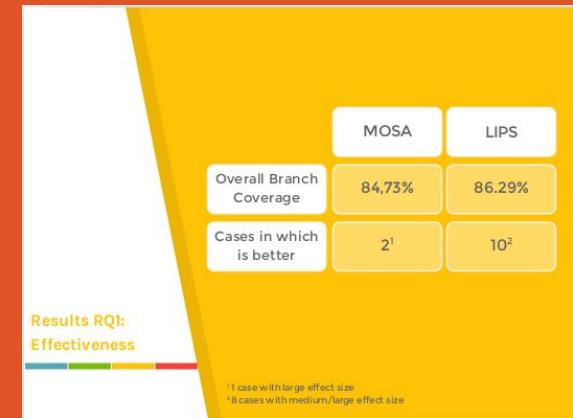


# Conclusions





# Conclusions

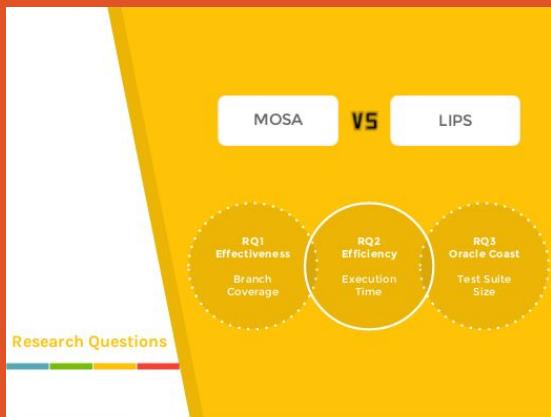
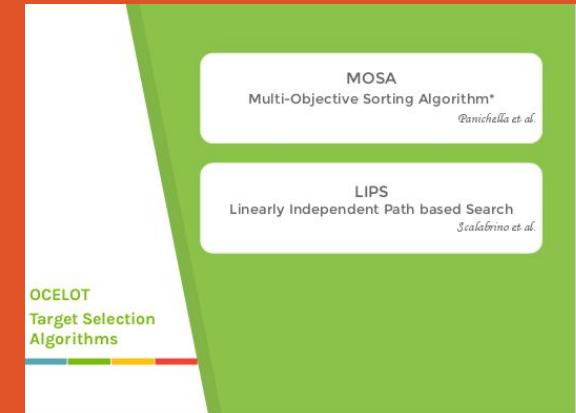
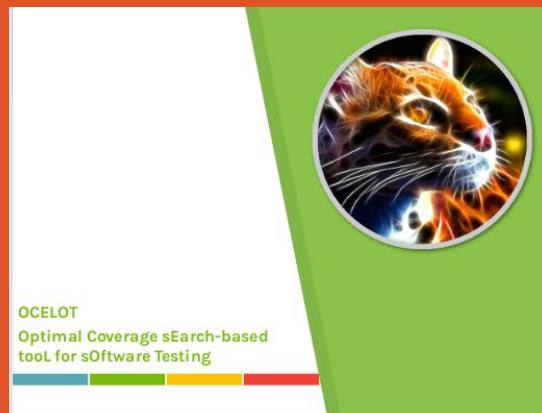


Test case generation

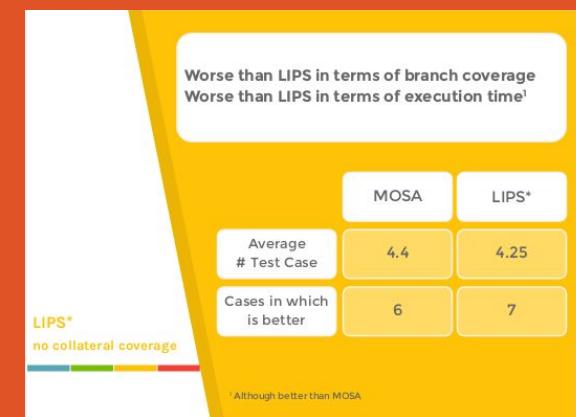
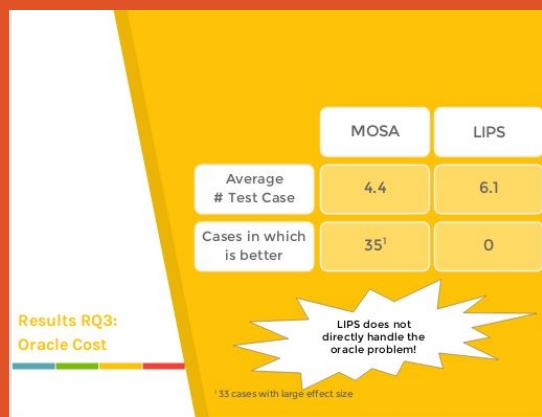
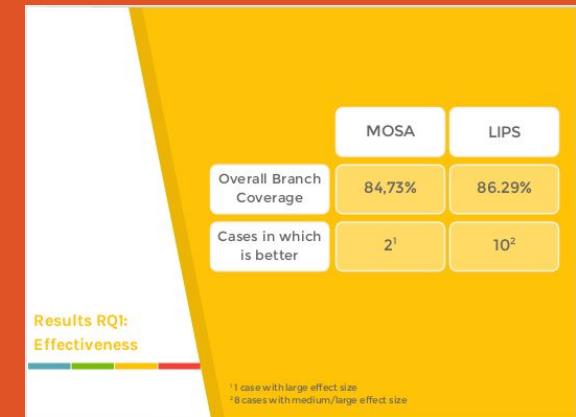
```

100 phobos
101 gradient_calc_bilinear_factor (double offset,
102   double val1,
103   double val2,
104   double val3,
105   double val4);
106 {
107     if (offset <= 0.0)
108       return 0.0;
109     else
110       phobos();
111   }
112
113   /* calculate linear offset from the start point onward */
114   offset = offset / 100.0;
115
116   /* = val1 + (val2 - val1) * offset */
117   x = val1 + offset * (val2 - val1);
118
119   /* = val3 + (val4 - val3) * offset */
120   z = val3 + offset * (val4 - val3);
121
122   /* = (x - z) * offset */
123   y = (x - z) * offset;
124
125   /* return final offset */
126   return (float) (x + y);
127 }
128
129 double _array[2] = {-0.0009,-0.004629544084,-0.35838452956801127};
130 void* __ptr64 = h_c_array;
131
132 double __arg0 = val0;
133 double __arg1 = val1;
134 double __arg2 = val2;
135 double __arg3 = val3;
136 double __arg4 = val4;
137 double __arg5 = val5;
138 double __arg6 = val6;
139
140 /* REPLACE THE ASSERTION BELOW */
141 ck_assert_int_eq(0,0);
142
143 END_TEST
144

```



# Conclusions



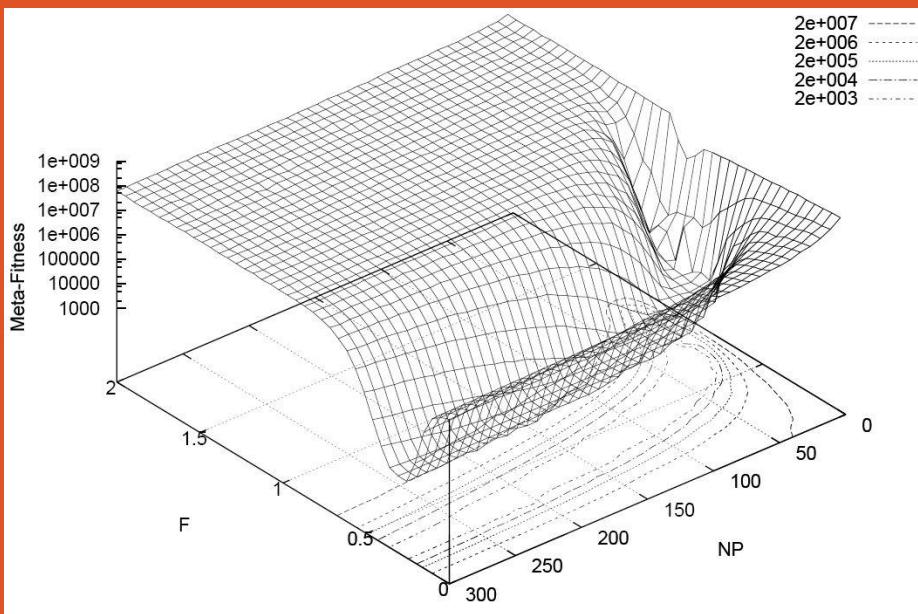
## **Future works**

## Future works

Replicate on larger dataset



# Future works



Replicate on larger dataset

Study C landscapes

## Future works

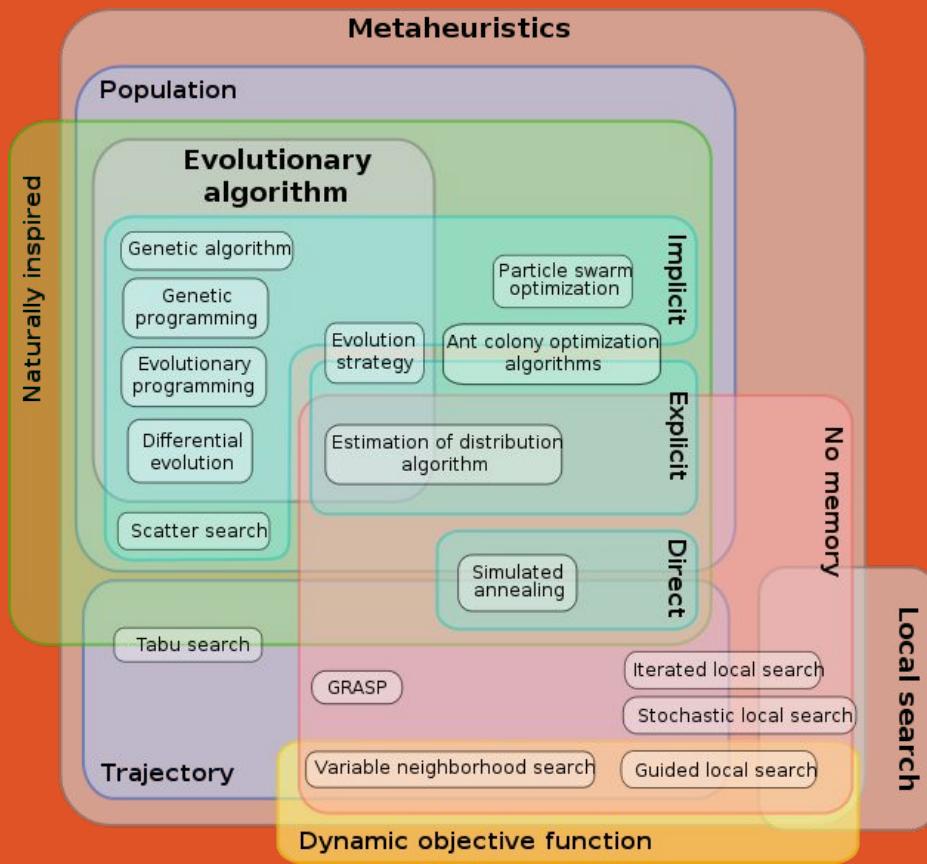
Replicate on larger dataset

Study C landscapes

Add LIPS to Evosuite



# Future works



Replicate on larger dataset

Study C landscapes

Add LIPS to Evosuite

Go beyond  
Genetic Algorithm

**Thanks for your attention!**

**Questions?**



**Dario Di Nucci**

**University of Salerno**

**[ddinucci@unisa.it](mailto:ddinucci@unisa.it)**

**<http://www.sesa.unisa.it/people/ddinucci/>**