Automated Test Case Generation

Search-Based Software Testing, Genetic Algorithms, Coverage Criteria

Annibale Panichella
a.panichella@tudelft.nl
@AnnibalePanic

TUDelft

Outline

- Rethinking Testing
- Search Based Software Testing
- Genetic Algorithms
- Automated test case generation
- Future directions

TUDelft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
           } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
                else
  9.              type = “SCALENE”;
           }
      }
}
```

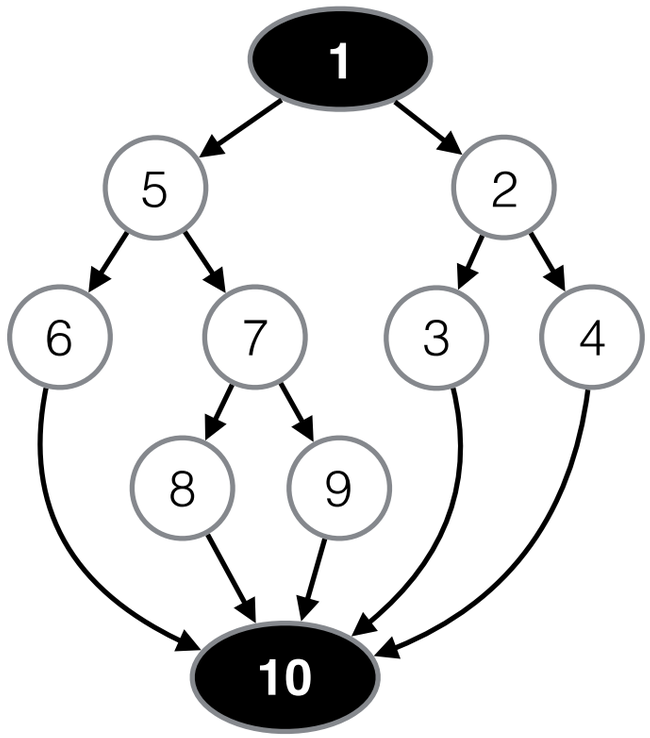
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
                else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph



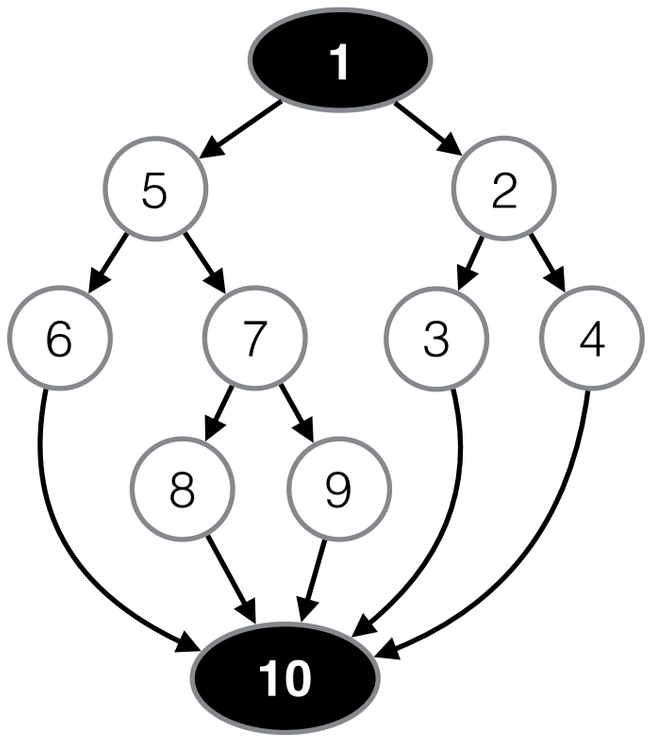
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
              else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph




```
@Test
public void test(){
   // Constructor (init)
   // Method Calls
   // Assertions  (check)
}
```

Test Case

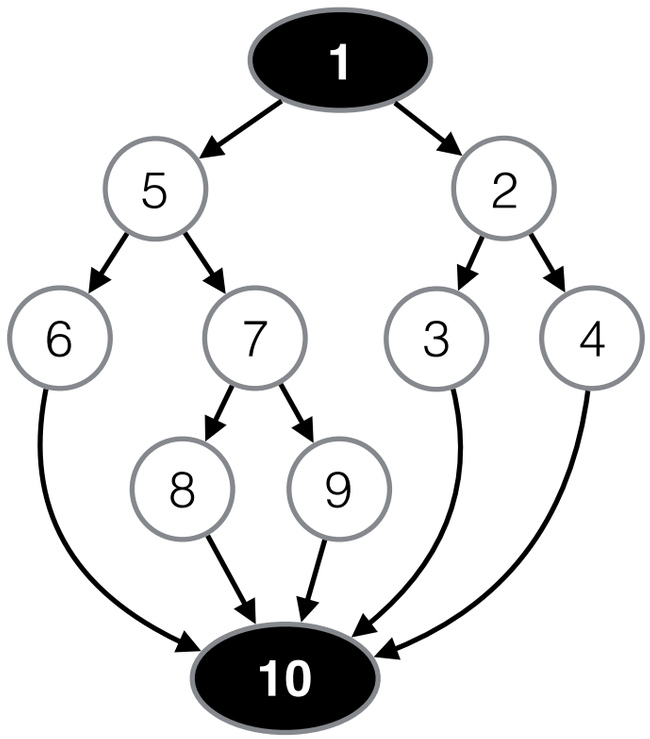
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
              else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph




```
@Test
public void test(){
 Triangle t = new Triangle (1,2,3);
  // Method Calls
  // Assertions  (check)
}
```

Test Case

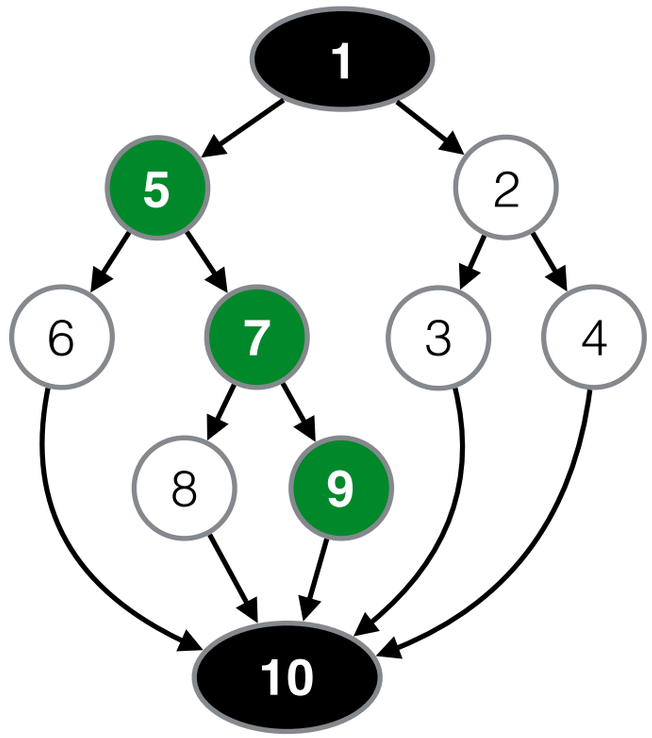
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
              else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph




```
@Test
public void test(){
 Triangle t = new Triangle (1,2,3);
 t.computeTriangleType();
  // Assertions  (check)
}
```

Test Case

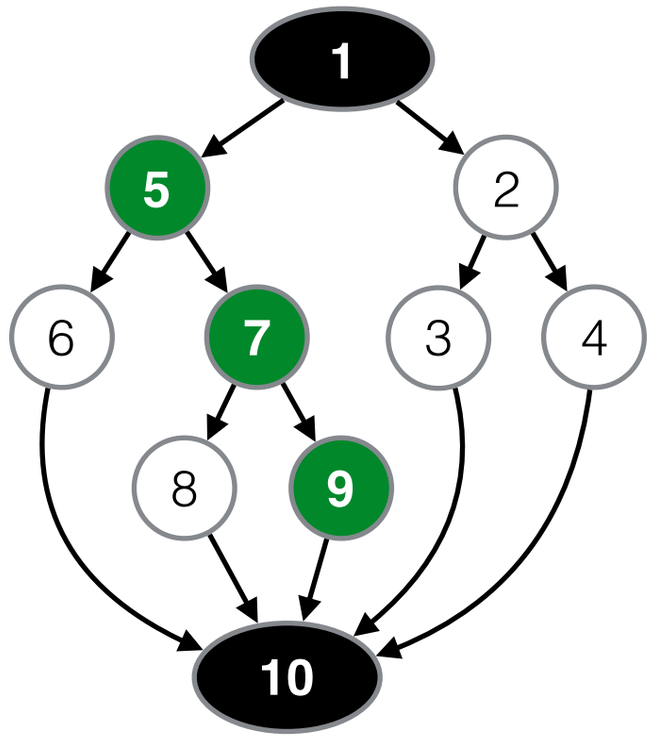
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
              else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph




```
@Test
public void test(){
 Triangle t = new Triangle (1,2,3);
 t.computeTriangleType();
 String type = t.getType();
 assertTrue(type.equals(“SCALENE”));
}
```

Test Case

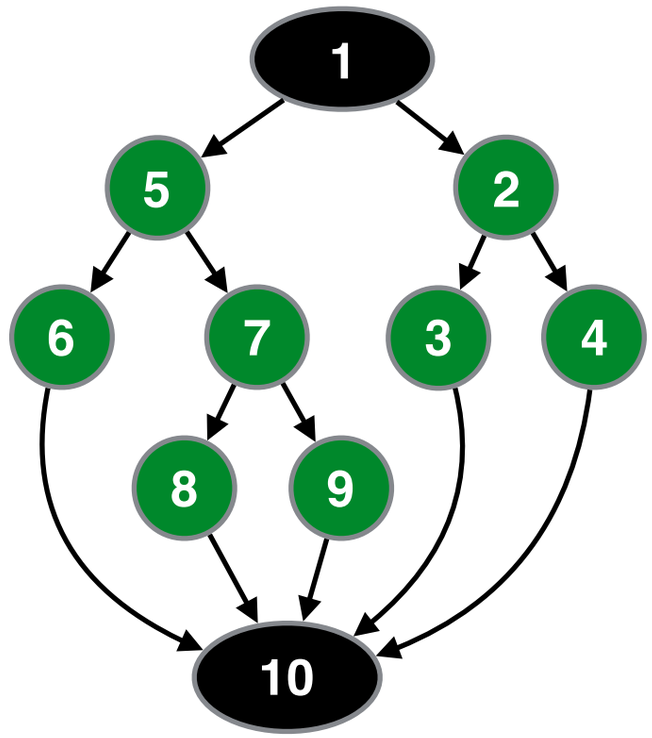
TU Delft

Unit Testing

Class Under Test (CUT)

```
class Triangle {
 int a, b, c; //sides
 String type = "NOT_TRIANGLE";

 Triangle (int a, int b, int c){…}

 void computeTriangleType() {
  1.  if (a == b) {
  2.      if (b == c)
  3.          type = "EQUILATERAL";
          else
  4.          type = "ISOSCELES";
      } else {
  5.      if (a == c) {
  6.          type = "ISOSCELES";
          } else {
  7.          if (b == c)
  8.              type = “ISOSCELES”;
              else
  9.              type = “SCALENE”;
          }
      }
}
```

Control Flow Graph



Goal: Testing the CUT as much as possible

TU Delft

How Much to Test?

Statement coverage

Targets = {1, 2, 3,4 ,5 ,6, 7, 8, 9,10}

Branch coverage

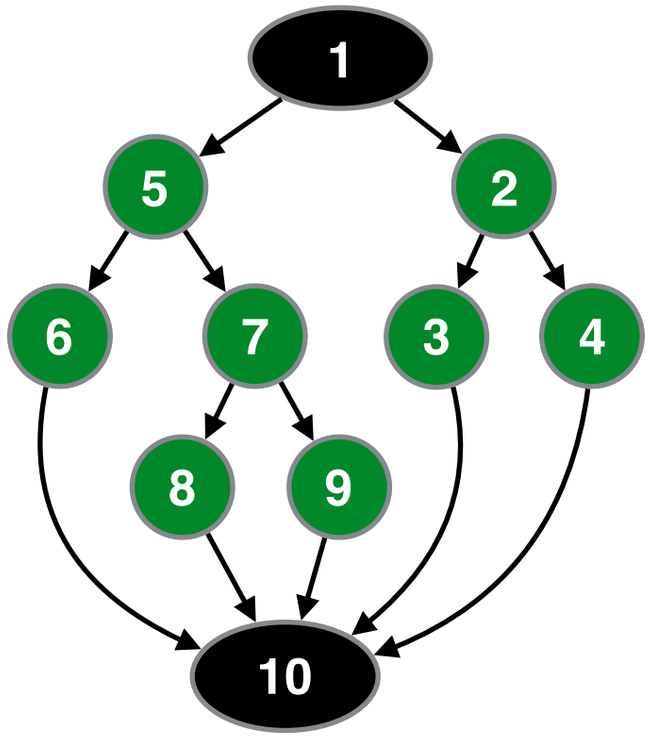
Targets = {<1,5>, <1,2>,<5,6>, <5,7>, <2,3>, <2,4>, <6,10>, <7,8>, <7,9>, <3,10>, <4,10>, <8,10>, <9,10>}

Path coverage

Targets = {<1,5,6,10>, <1,5,7,8,10>, <1,5,7,9,10>, <1,2,3,10>, <1,2,4,10>}

Mutation Coverage?

Control Flow Graph



TUDelft

Manual Testing

Manual design of test cases

1. Select one target
   - for example, branch <7,8>

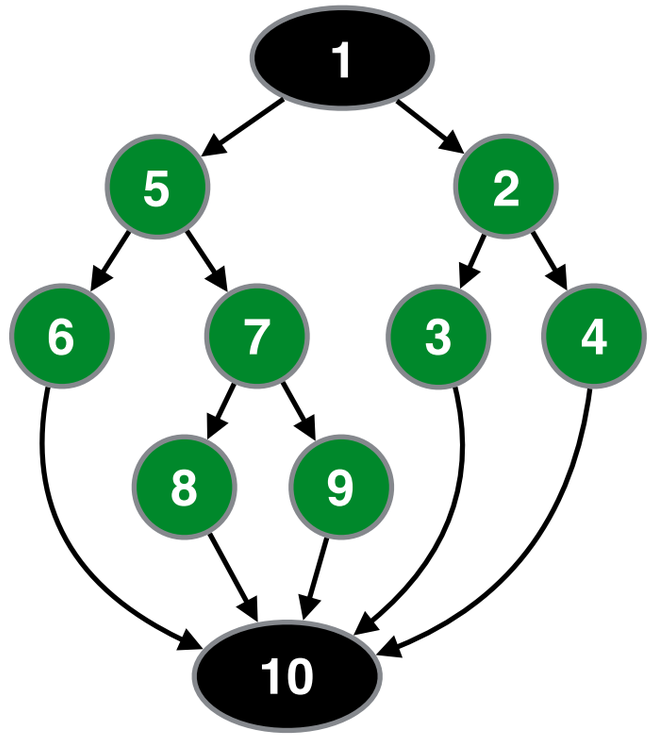
2. Determine method calls and input parameters that allow to cover the selected target

3. Write the test case

4. Execute the test to verify whether it reaches or not the target (if no the test has to be modified)

5. Repeat steps 1-4 until all targets are covered

Control Flow Graph



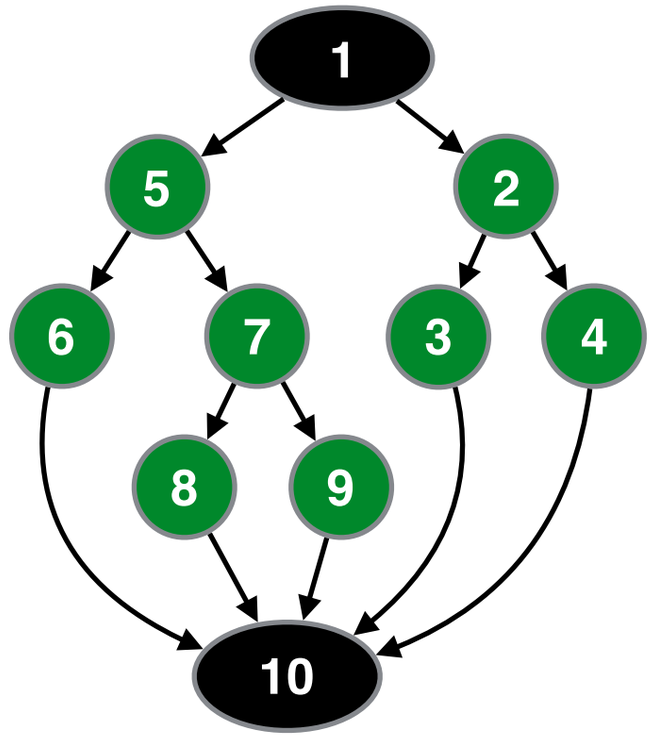
TUDelft

Manual Testing

Challenges:

- Laborious and time consuming?
- Tedious?
- Difficult?
- Error-prone?
- Effective? What about faults?
- Execution time?

Control Flow Graph



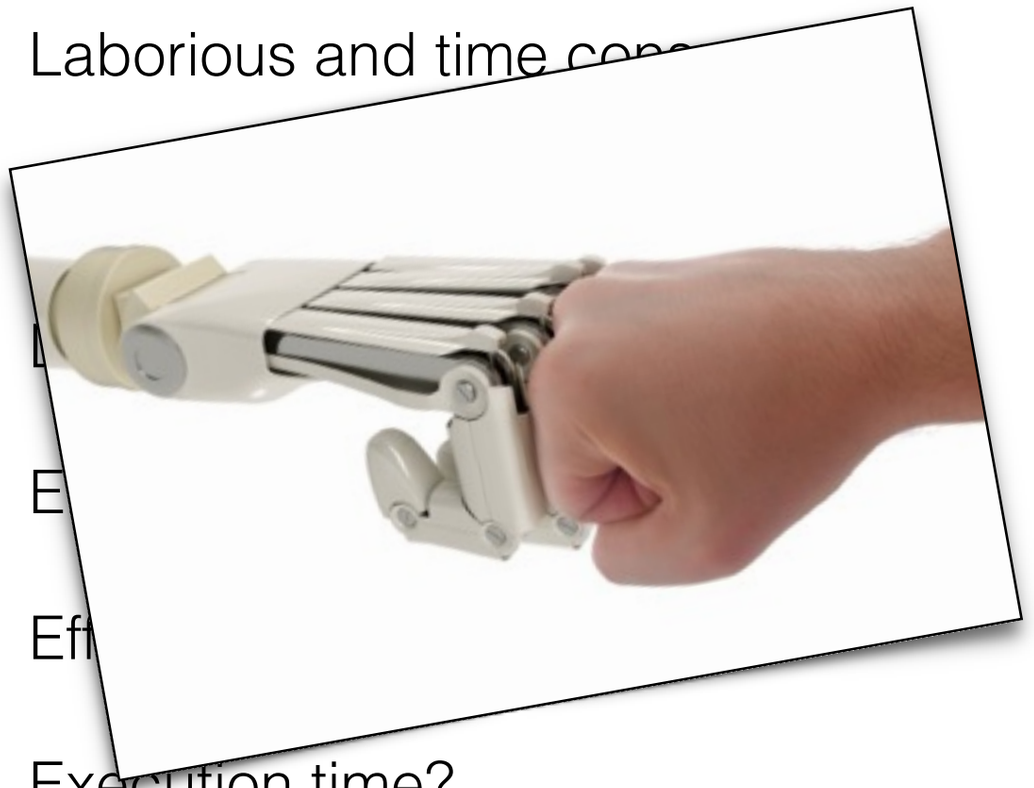
TU Delft

Automated

~~Manual~~ Testing

Challenges:

- Laborious and time cons



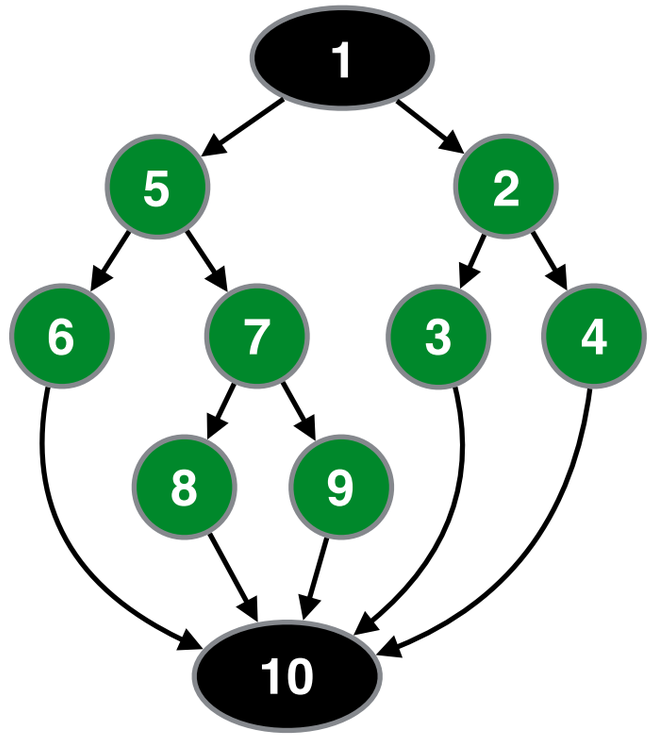
-

- E

- Eff

- Execution time?

Control Flow Graph



TUDelft

Automated

~~Manual~~ Testing

Generation

~~Manual design~~ of test cases

1. Select one target
   - for example, branch <7,8>

Search for

2. ~~Determine~~ method calls and input parameters that allow to cover the selected target

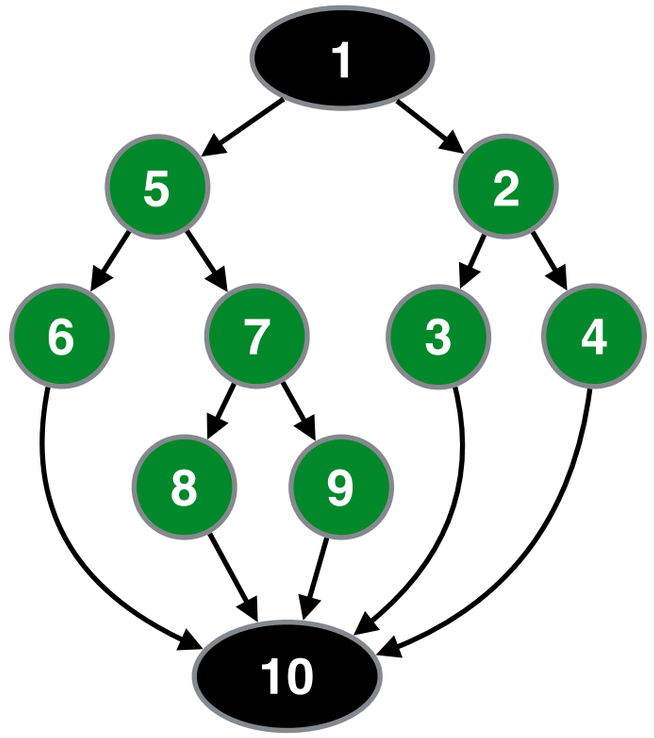
Store

3. ~~Write~~ the test case

4. Execute the test to verify ~~whether~~ if reaches or not ~~the~~ target (if no the test ~~has~~ to be modified)

5. Repeat steps 1-4 until all targets are covered (maximise)

Control Flow Graph



Observation: This looks like a search/ optimization problem

TU Delft

Search-Based Software Testing

«The application of meta-heuristic search-based optimization techniques to find near-optimal solutions in software testing problems.»

TUDelft

Search-Based Software Testing

«The application of meta-heuristic search-based optimization techniques to find near-optimal solutions in software testing problems.»

Problem Reformulation: reformulating typical testing problems as optimization problems

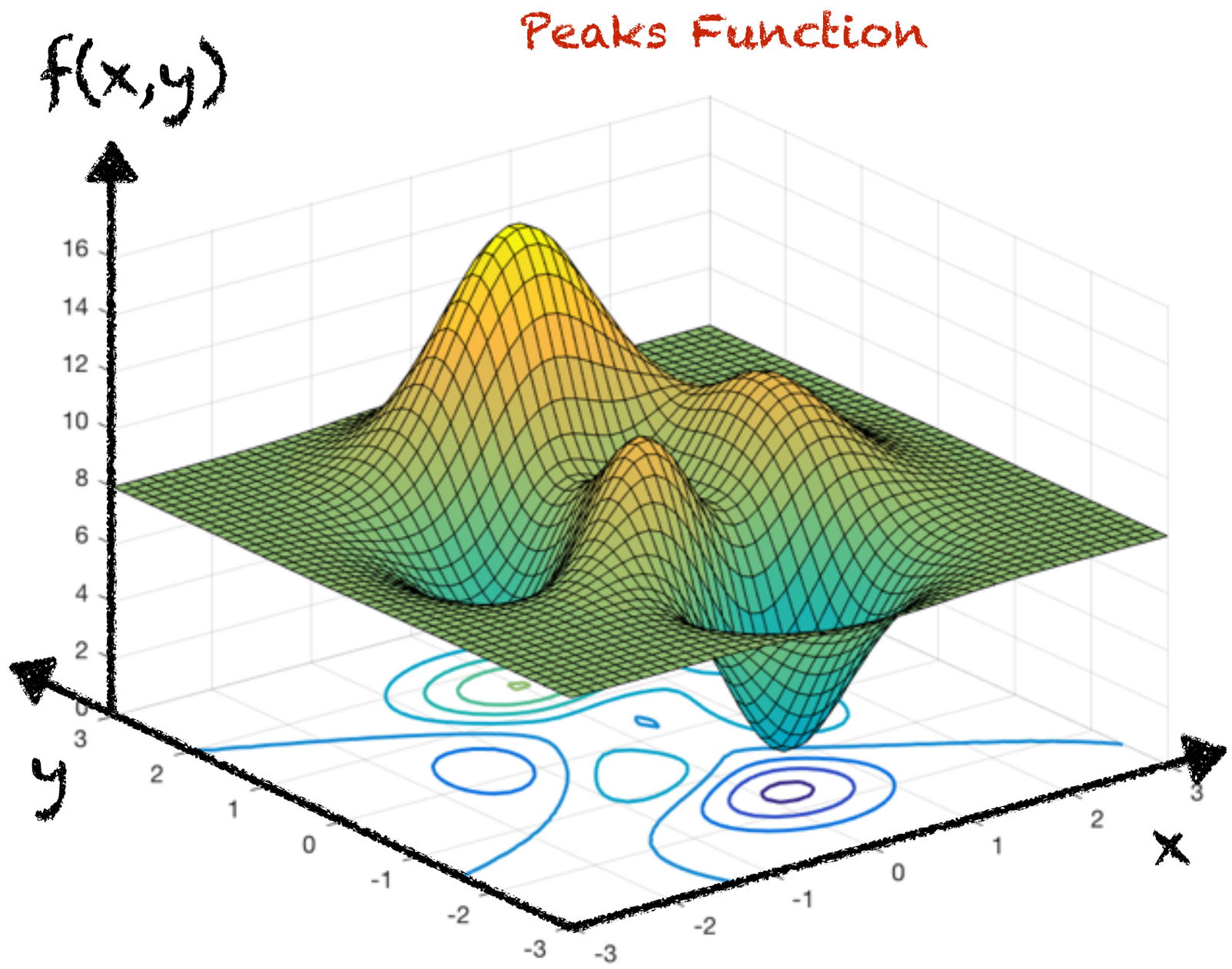
Fitness Function: definition of functions to optimise

Optimization Algorithms: applying search algorithm to solve such functions

- Random Search
- Hill climbing
- Genetic Algorithms
- Simulated Annealing
- Tabu Search
- Particle Swarm Optimization
- …

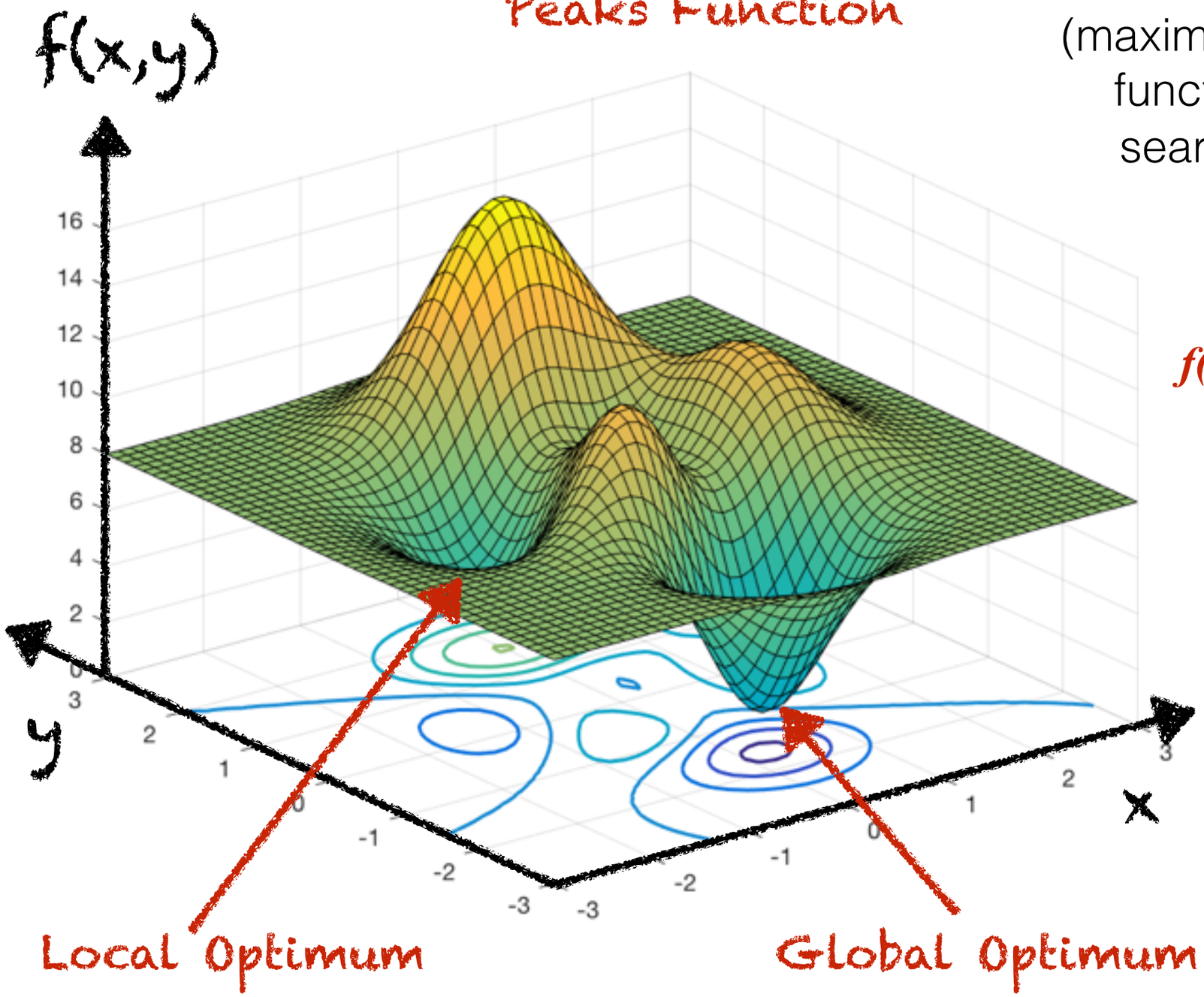
TUDelft

Search Problem



TU Delft

Search Problem



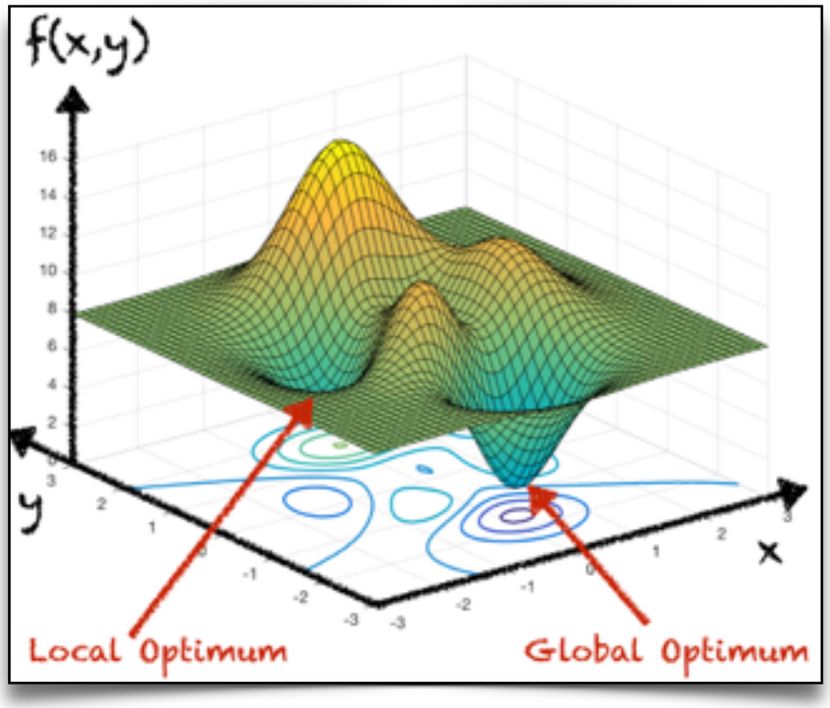
Find a vector $[x^*,y^*]$ which minimises (maximises) the objective function $f(x,y)$ over a search space $[X,Y]$:

$$\forall\, x \in X$$
$$\forall\, y \in Y$$
$$f(x^*,y^*) \leqslant f(x,y)$$

TU Delft

Search Algorithms

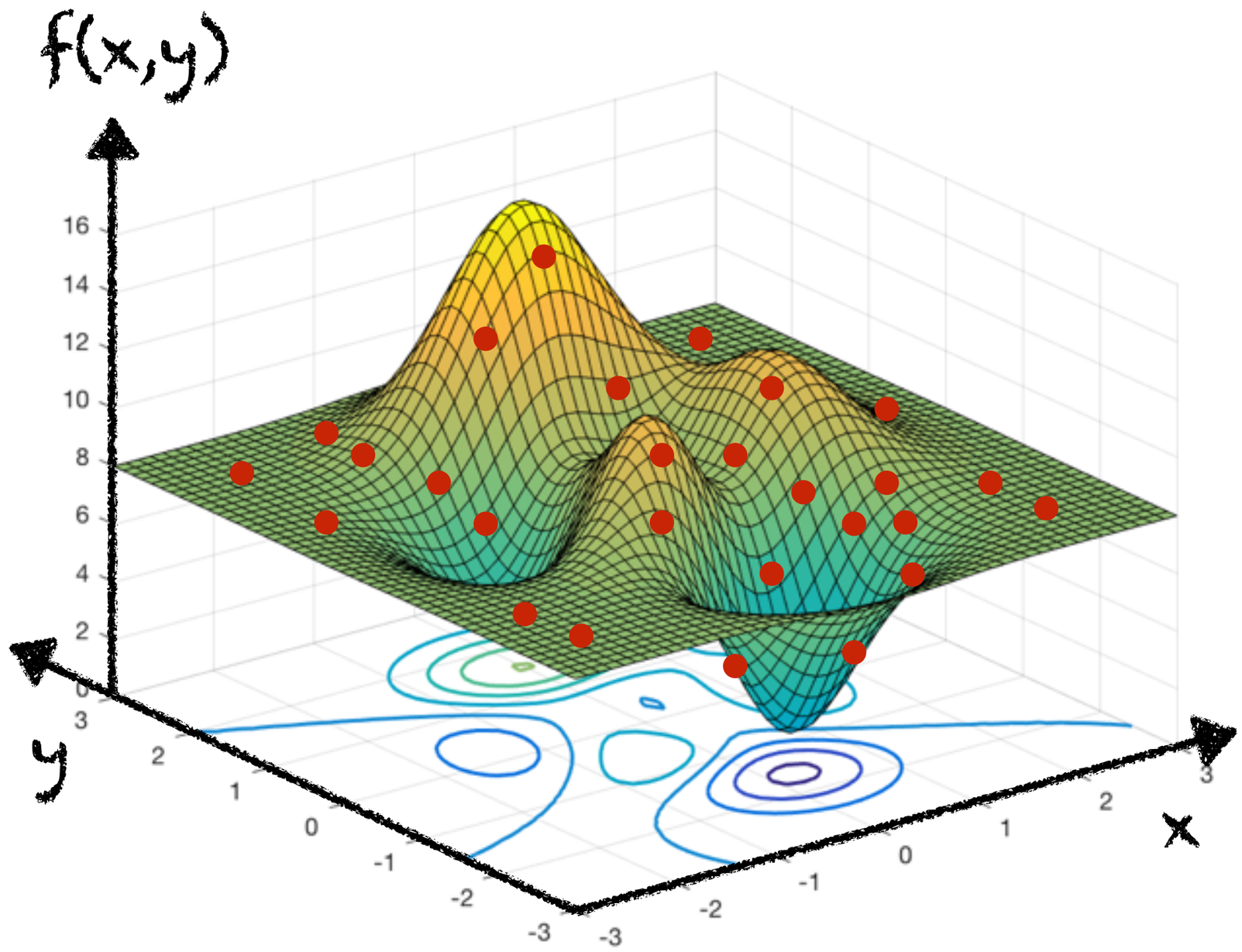
Search Algorithms:
- Exhaustive search
- Random search
- Meta-heuristic search



Meta-heuristic algorithm: iterative optimization of candidate solutions for arbitrary problem instances. It is not ensured to find a global optimum, it finds a so called near optimal solutions.

Random Algorithms

Random Algorithm: “try random values and take the best trial”

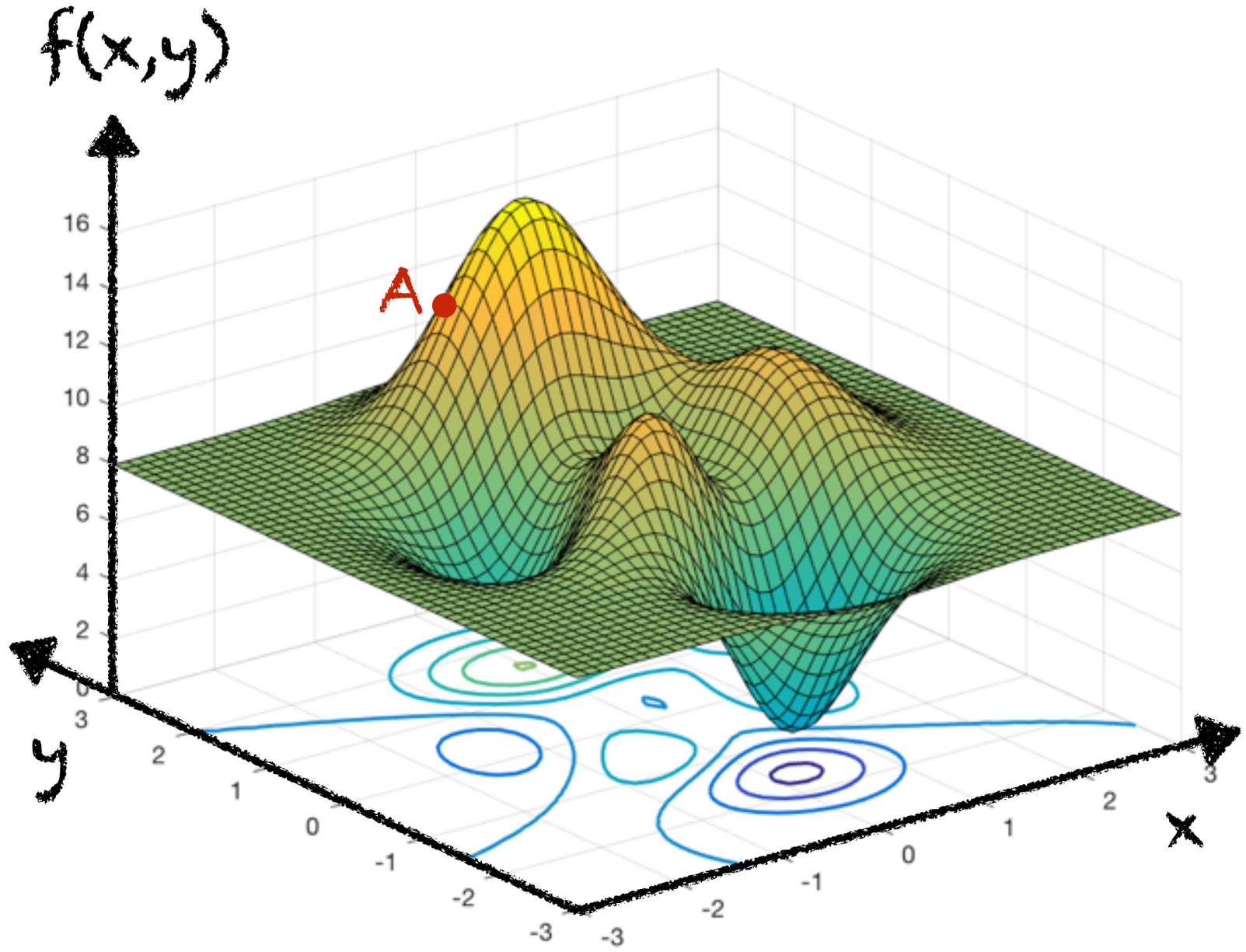


TUDelft

Hill Climbing

Pseudo-code:

1) Start with a random solution *A*

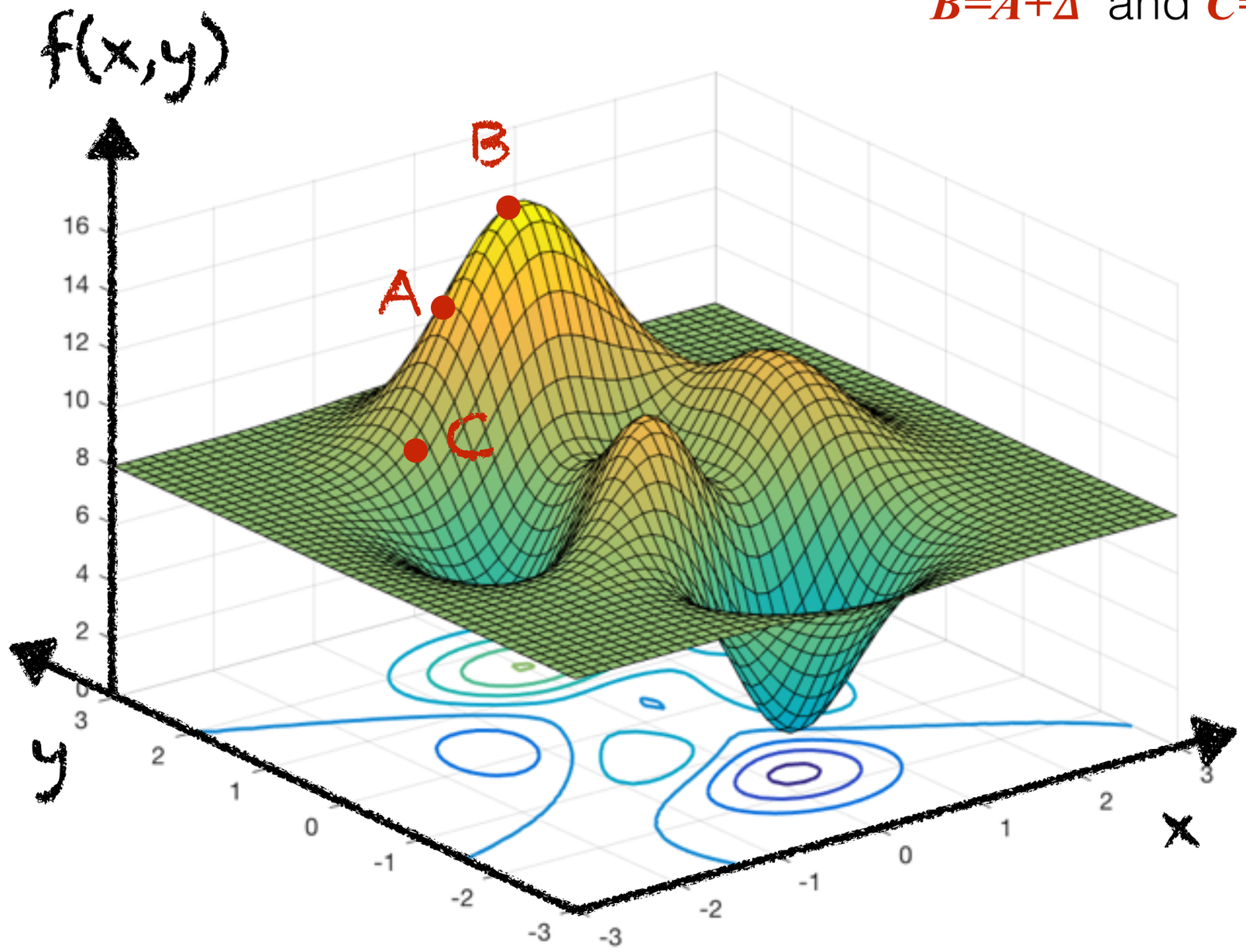


TU Delft

Hill Climbing

Pseudo-code:

1) Start with a random solution ***A***
2) Try some "***nearby***" solutions ***B=A+Δ*** and ***C=A-Δ***

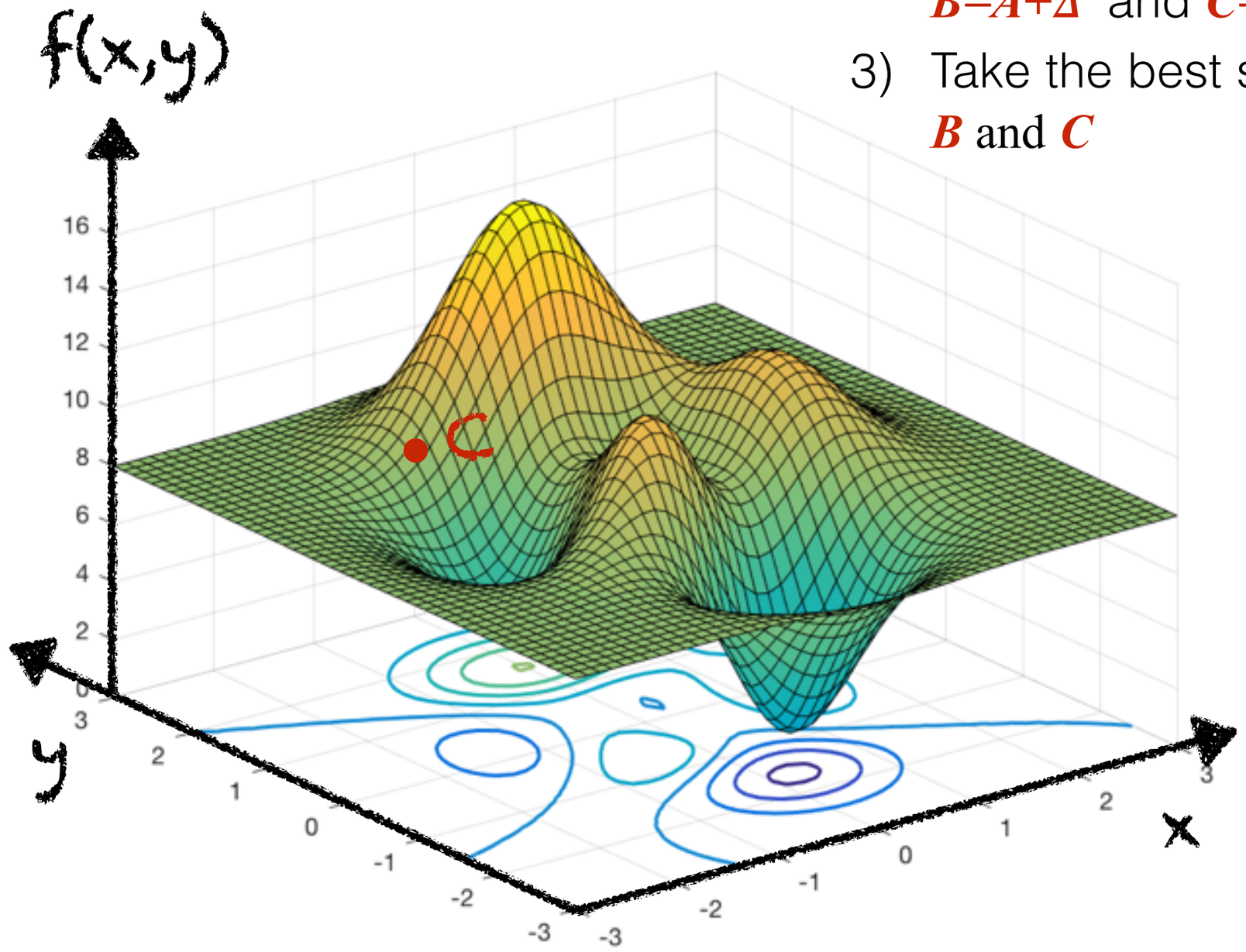


TUDelft

Hill Climbing

Pseudo-code:

1) Start with a random solution ***A***
2) Try some "***nearby***" solutions ***B=A+Δ*** and ***C=A-Δ***
3) Take the best solution among ***A***, ***B*** and ***C***

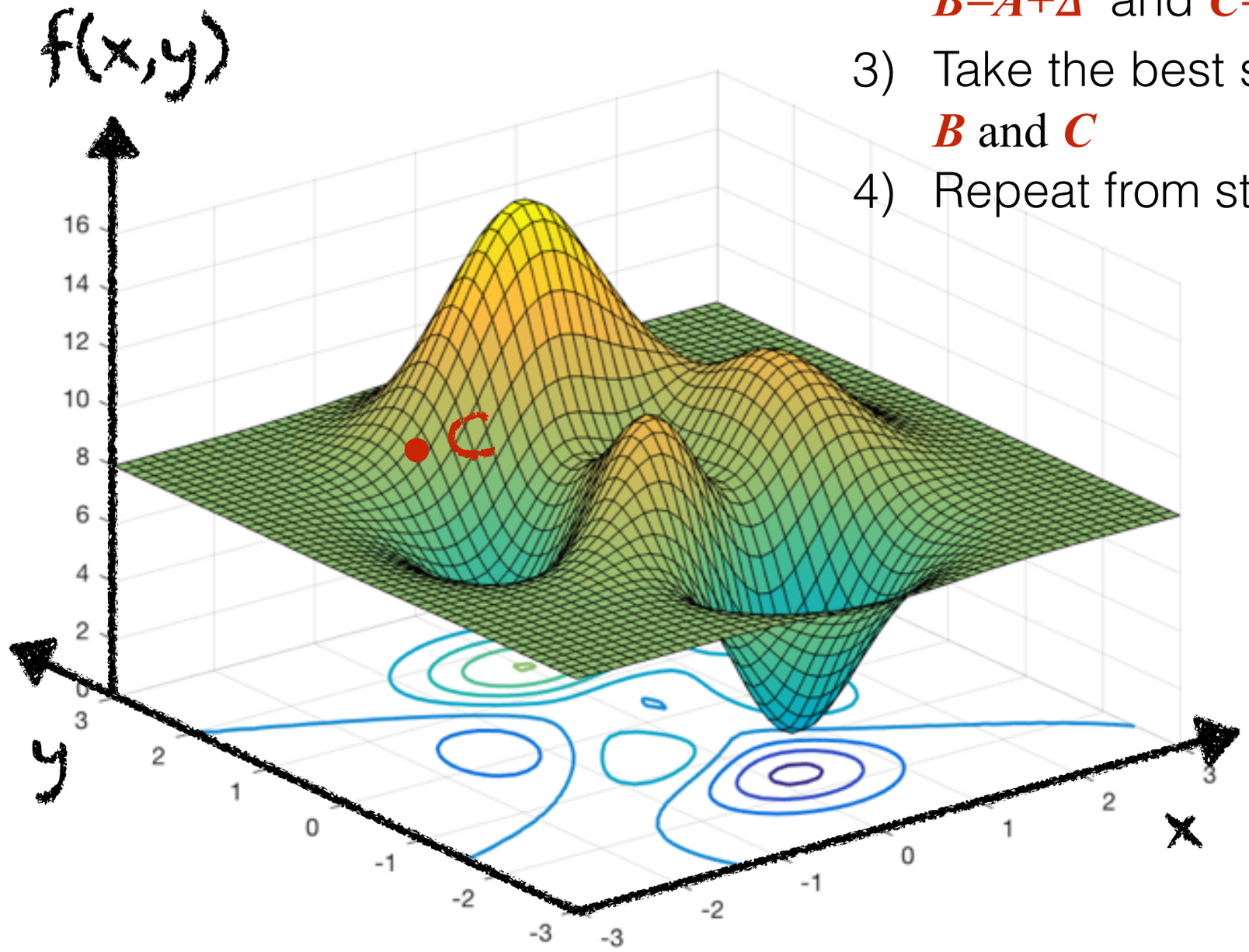


TU Delft

Hill Climbing

Pseudo-code:

1) Start with a random solution ***A***
2) Try some "***nearby***" solutions ***B=A+Δ*** and ***C=A-Δ***
3) Take the best solution among ***A***, ***B*** and ***C***
4) Repeat from step (1)



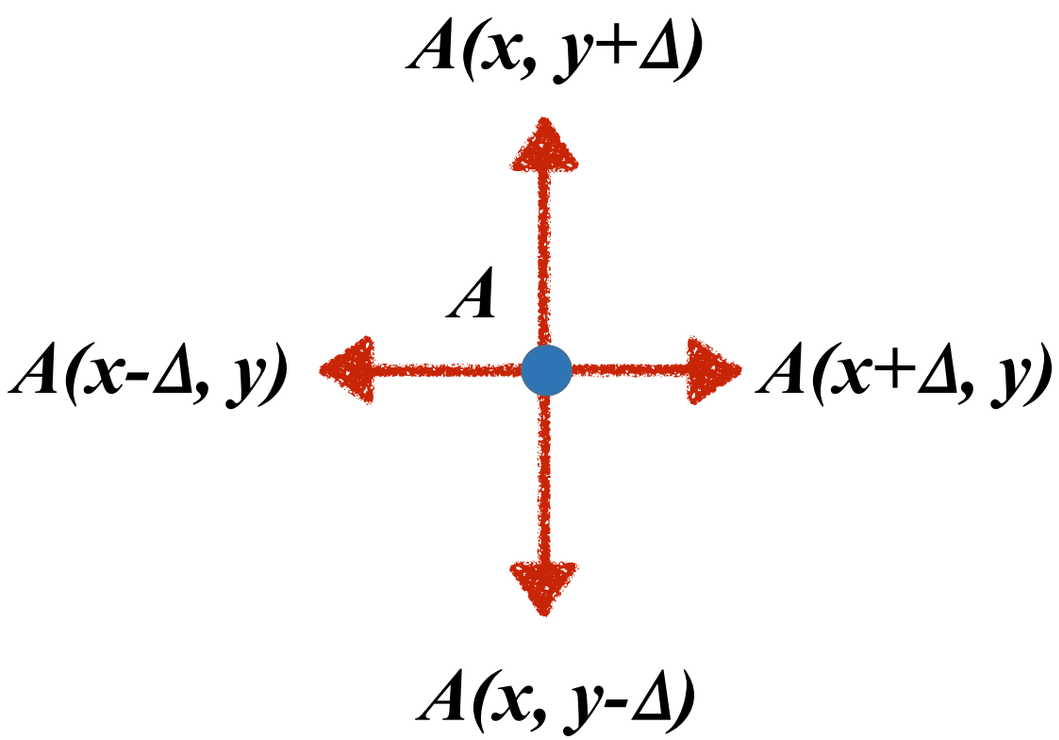
TUDelft

Hill Climbing

Pseudo-code:

1) Start with a random solution A
2) Try some "***nearby***" solutions $B=A+\Delta$ and $C=A-\Delta$
3) Take the best solution among A, B and C
4) Repeat from step (1)

How many nearby solutions?



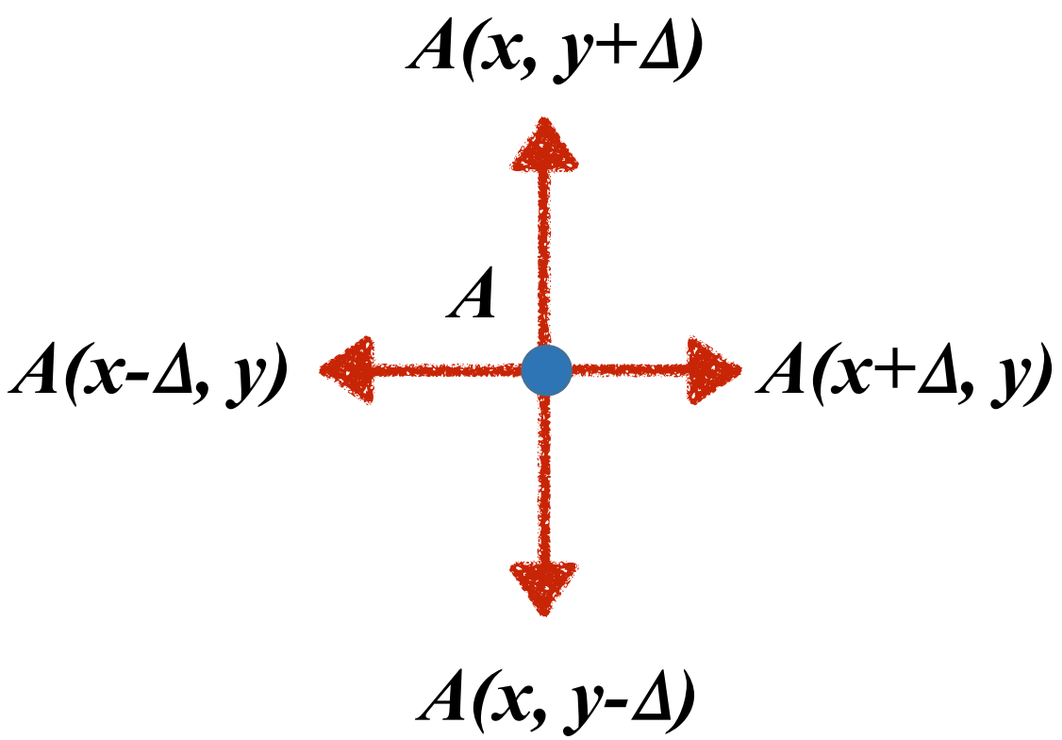
TU Delft

Hill Climbing

Pseudo-code:

1) Start with a random solution A
2) Try some "***nearby***" solutions $B=A+\Delta$ and $C=A-\Delta$
3) Take the best solution among A, B and C
4) Repeat from step (1)

How many nearby solutions?



Hill Climbing is a local search algorithm, thus, it can converge toward local (sub-optimal) solutions

TU Delft

Hill Climbing

Hill Climbing: it is equivalent to **greedy** or steepest descent algorithm

Hill Climbing Variants:

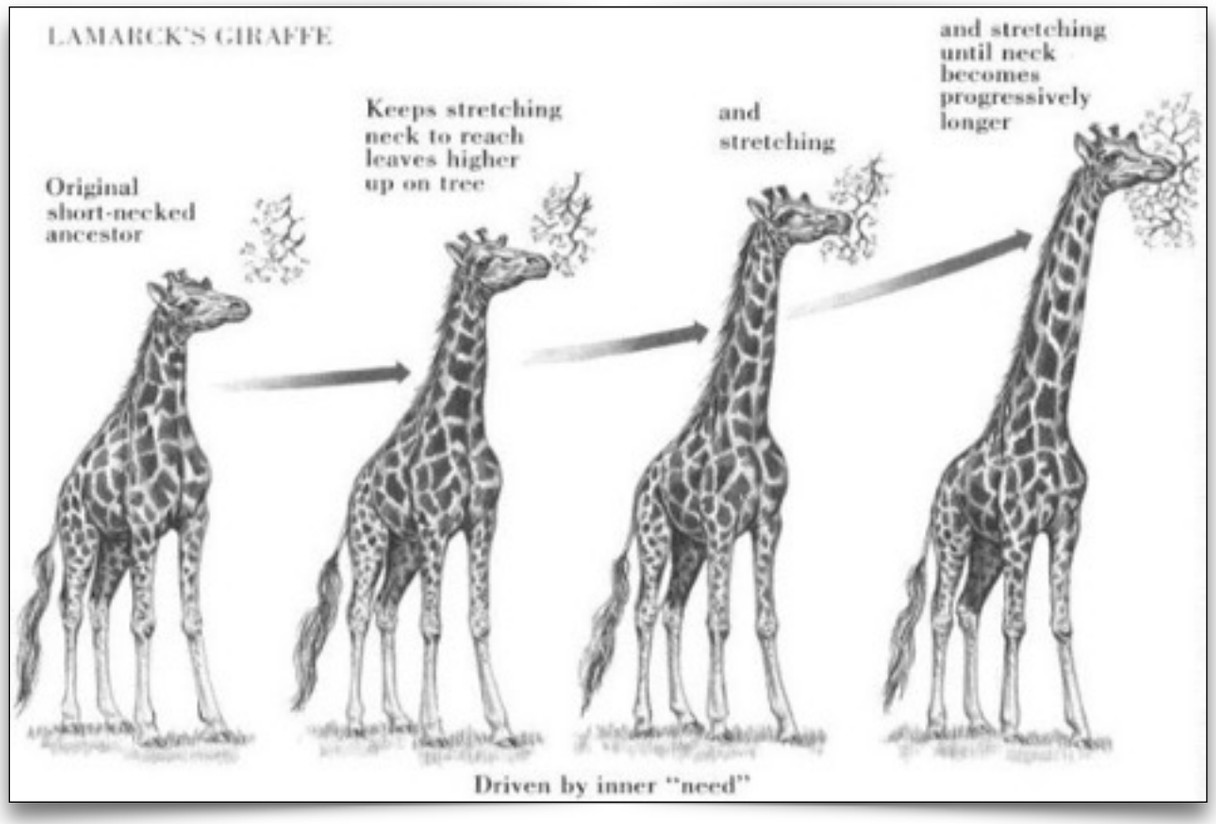
- **Stochastic hill climbing**: does not examine all neighbours before deciding how to move. Rather, it selects a neighbour at random, and decides (based on the amount of improvement in that neighbour) whether to move to that neighbour or to examine another.

- **Random-restart hill climbing**: random restart when no improving solution is found (or improvement is small)

TUDelft

Genetic Algorithms

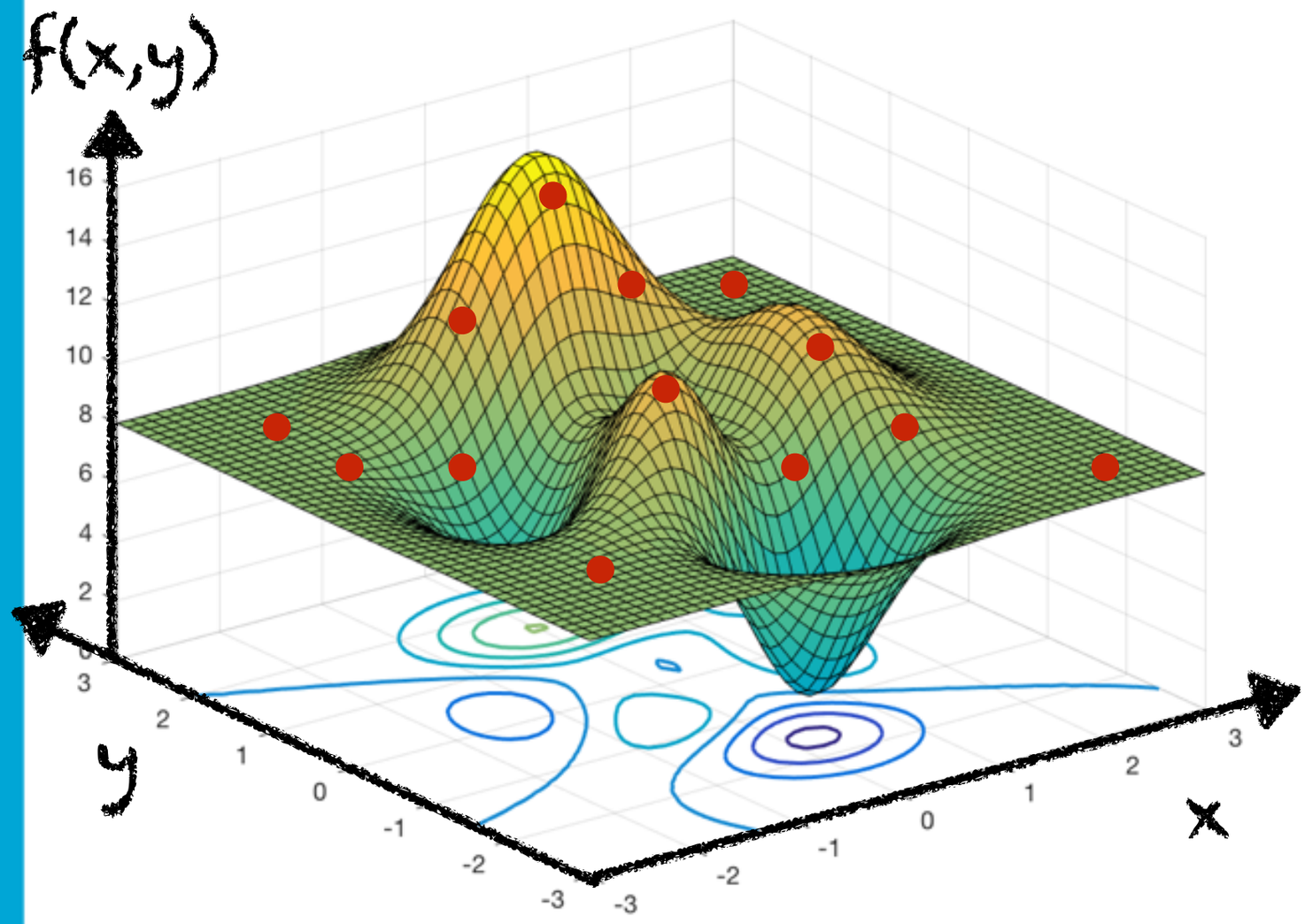
Genetic Algorithm: search algorithm inspired be evolution theory



1. Natural selection: Individuals that best fit the natural environment survive (worst die)
2. Reproduction: survived individuals generate offsprings (next generation)
3. Mutation: offsprings inherits properties of their parents (with some mutations)
4. Iteration: generation by generation the new offspring fit better the environment than their parents

TU Delft

Genetic Algorithms

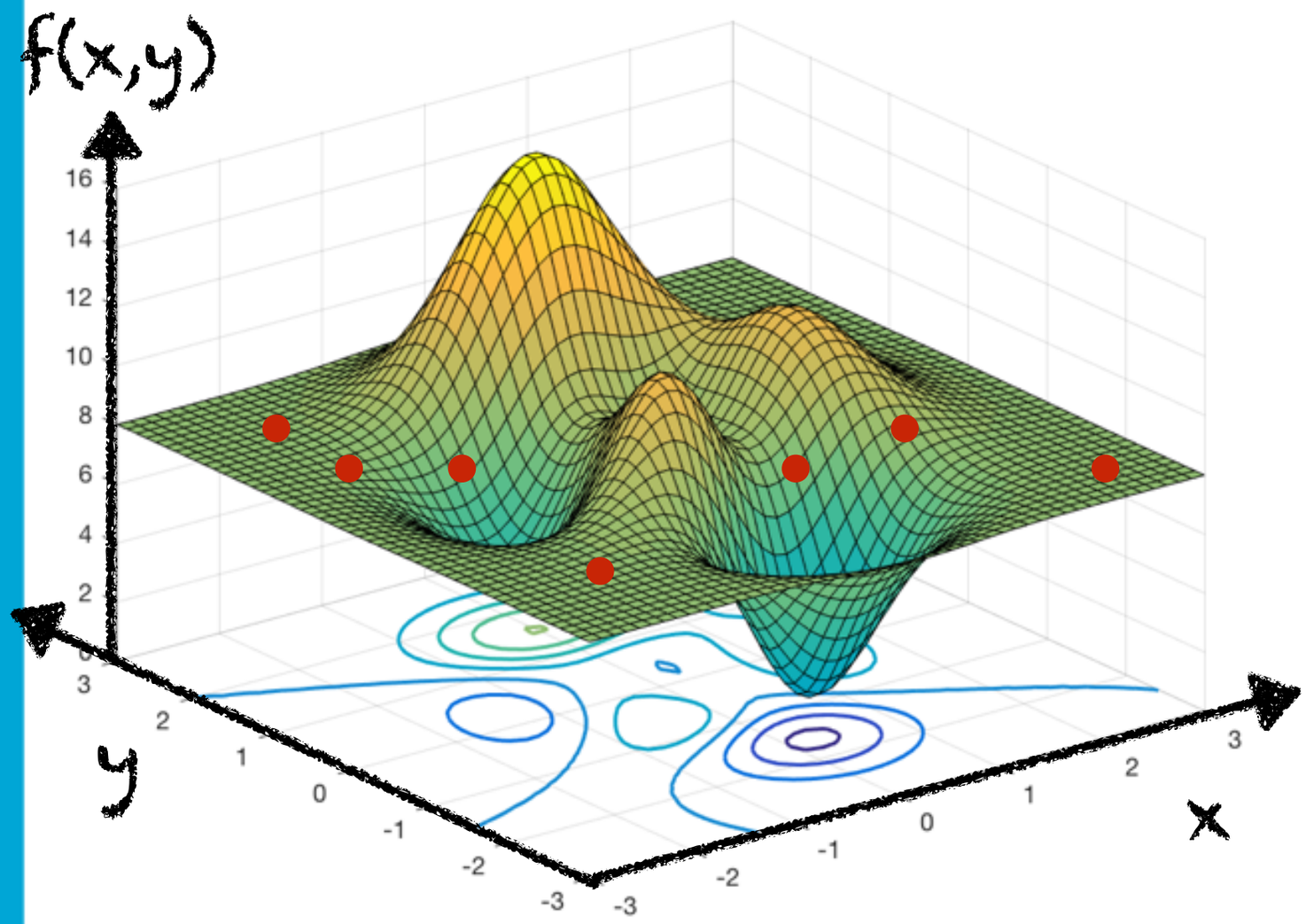


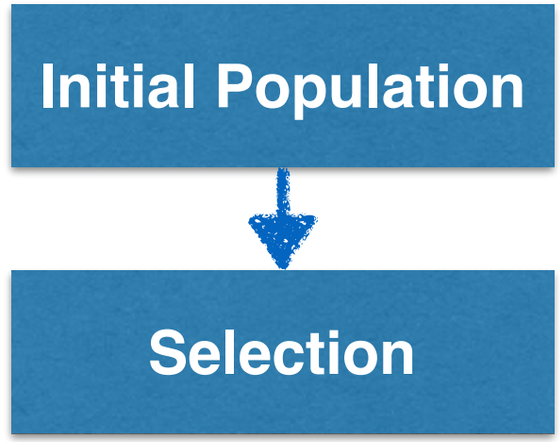
Initial Population

Set of randomly generated solutions

TUDelft

Genetic Algorithms



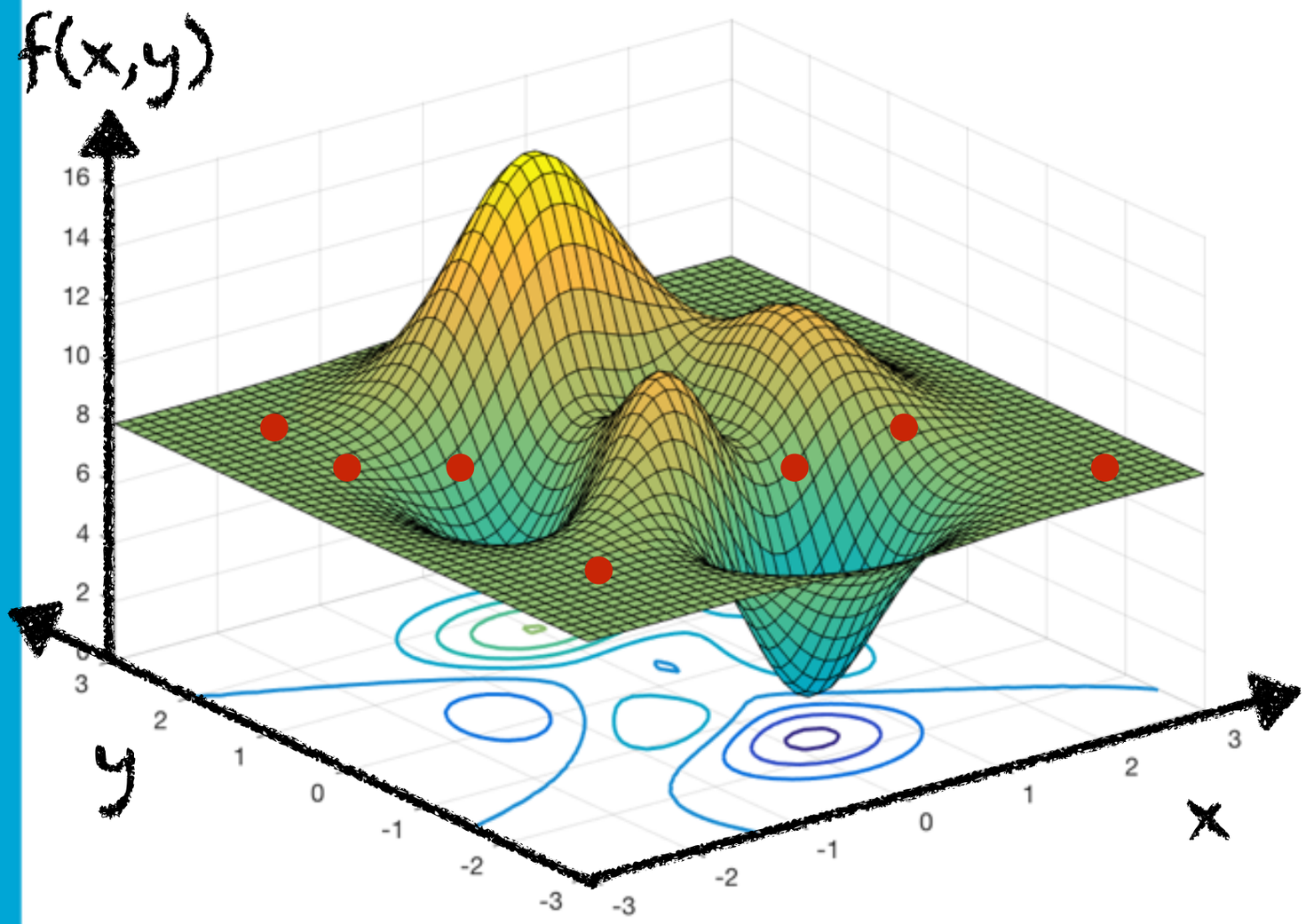


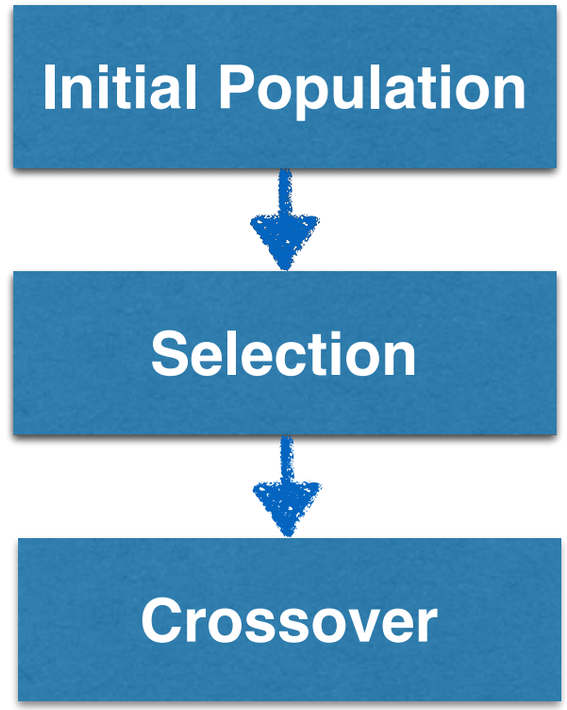
Best individuals are those with a lower (fittest) function value ***f*(.)**

The function to optimise is often called "**fitness function**"

TU Delft

Genetic Algorithms

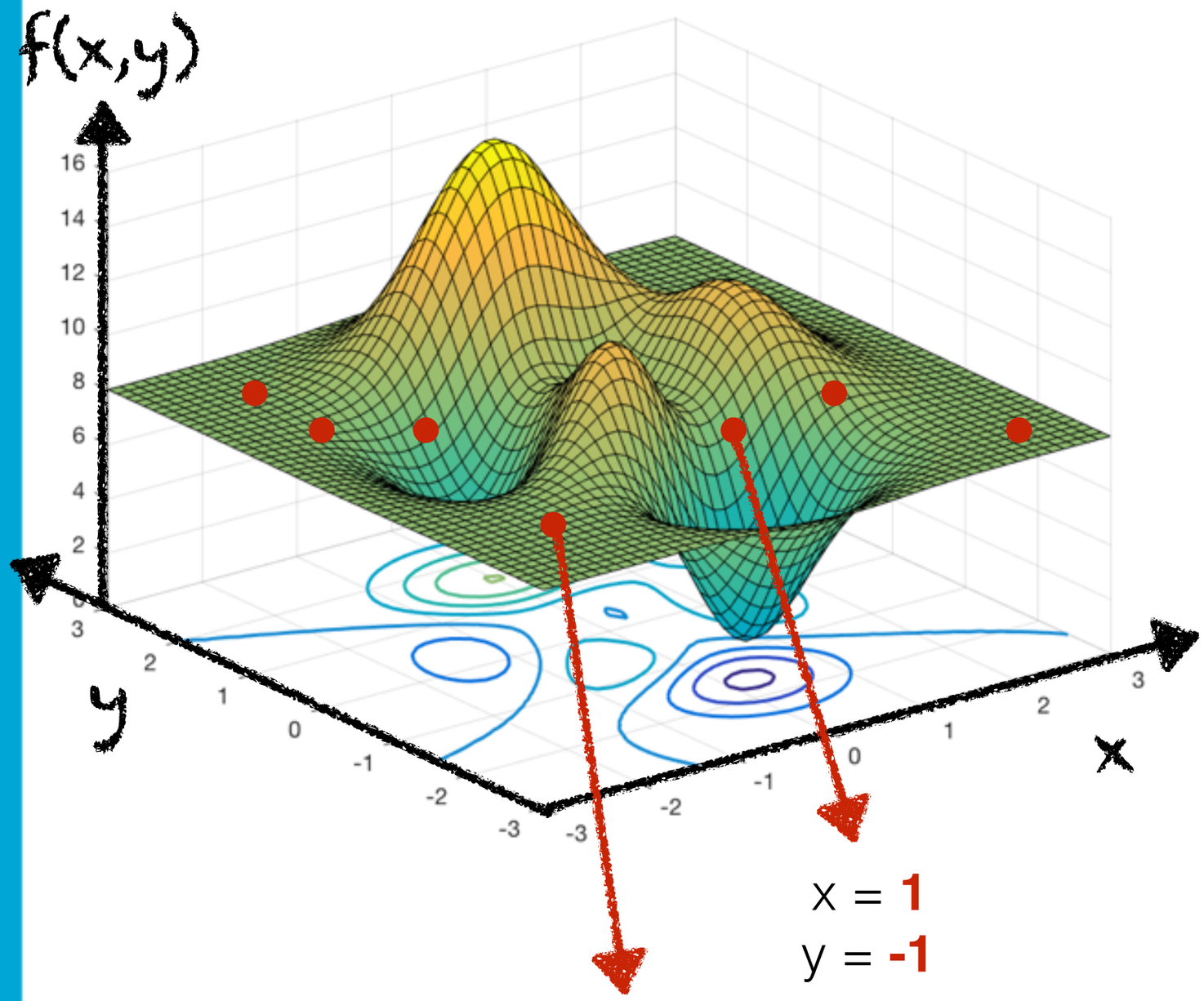




Genetic Recombination

TU Delft

Genetic Algorithms



x = **1**
y = **-1**

x = **-2**
y = **-2**

Initial Population

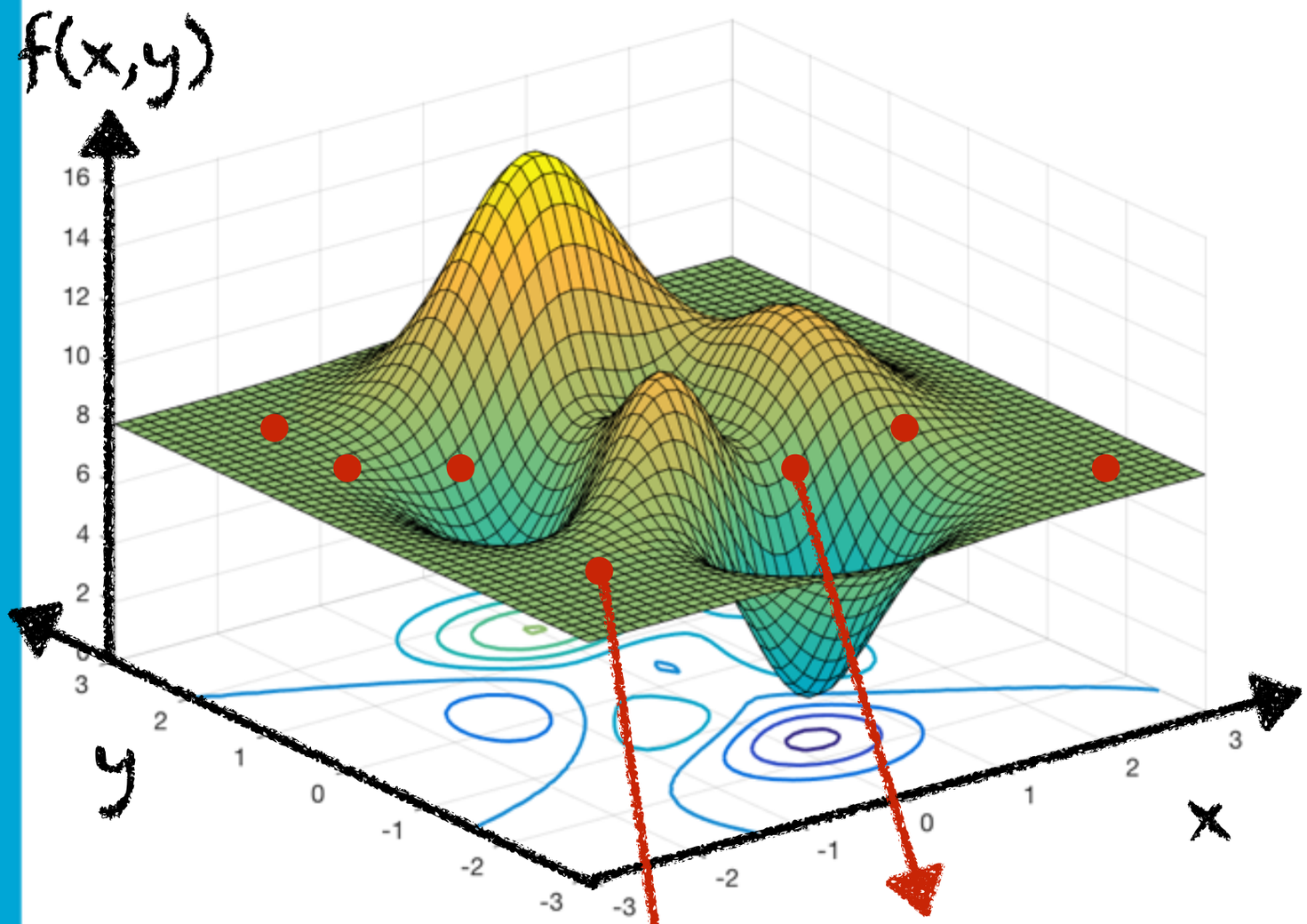
Selection

Crossover

1. Select pairs of solutions (parents)

TU Delft

Genetic Algorithms



Single-point crossover

$\frac{x = 1}{y = -1}$ $\frac{x = -2}{y = -2}$

$\frac{x = -2}{y = -1}$ $\frac{x = 1}{y = -2}$

Initial Population

Selection

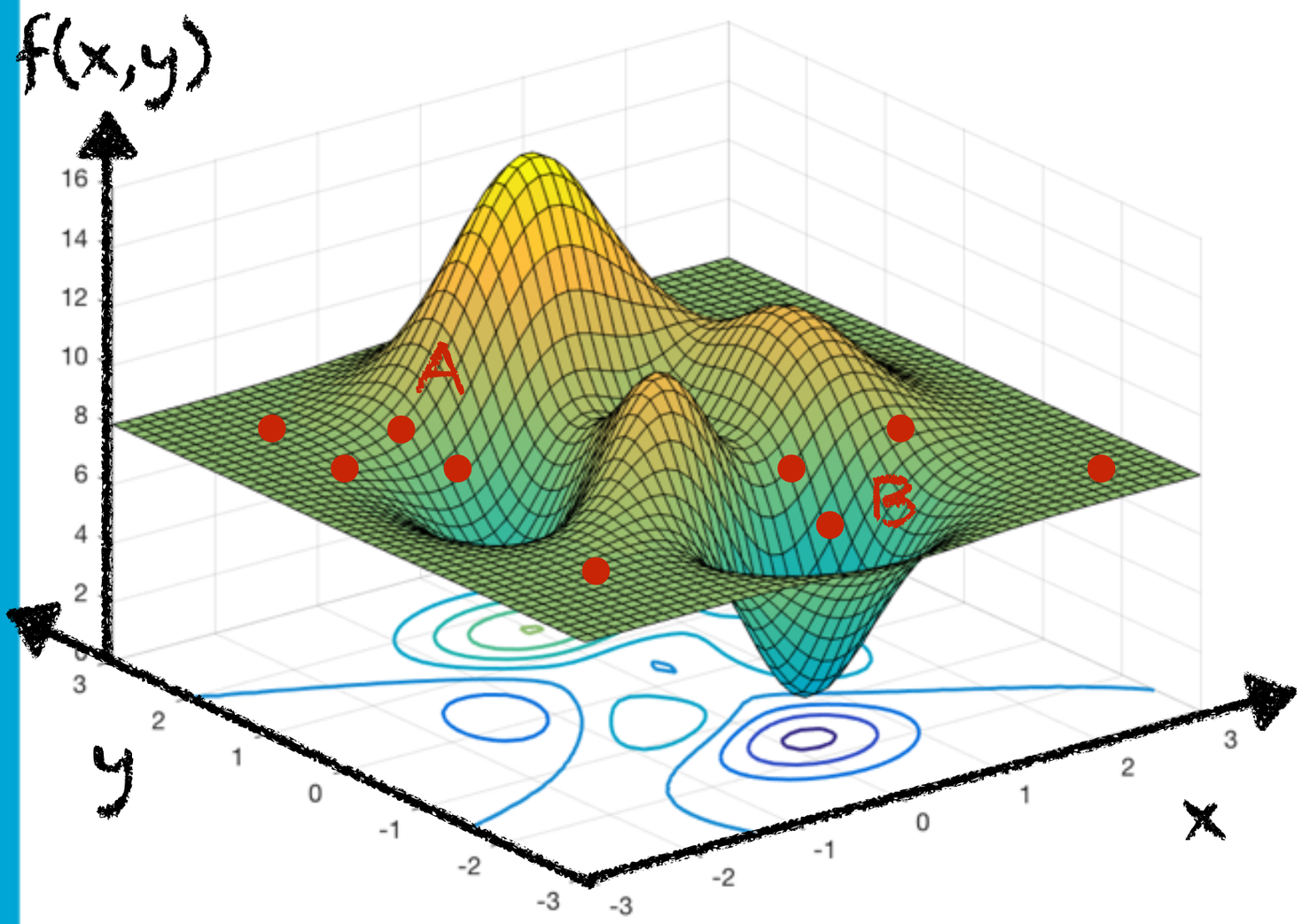
Crossover

1. Select pairs of solutions (parents)

2. Generate new solutions (offsprings)

TUDelft

Genetic Algorithms



Single-point crossover

A
x = **-2**
y = **-1**

B
x = **1**
y = **-2**

Initial Population

Selection

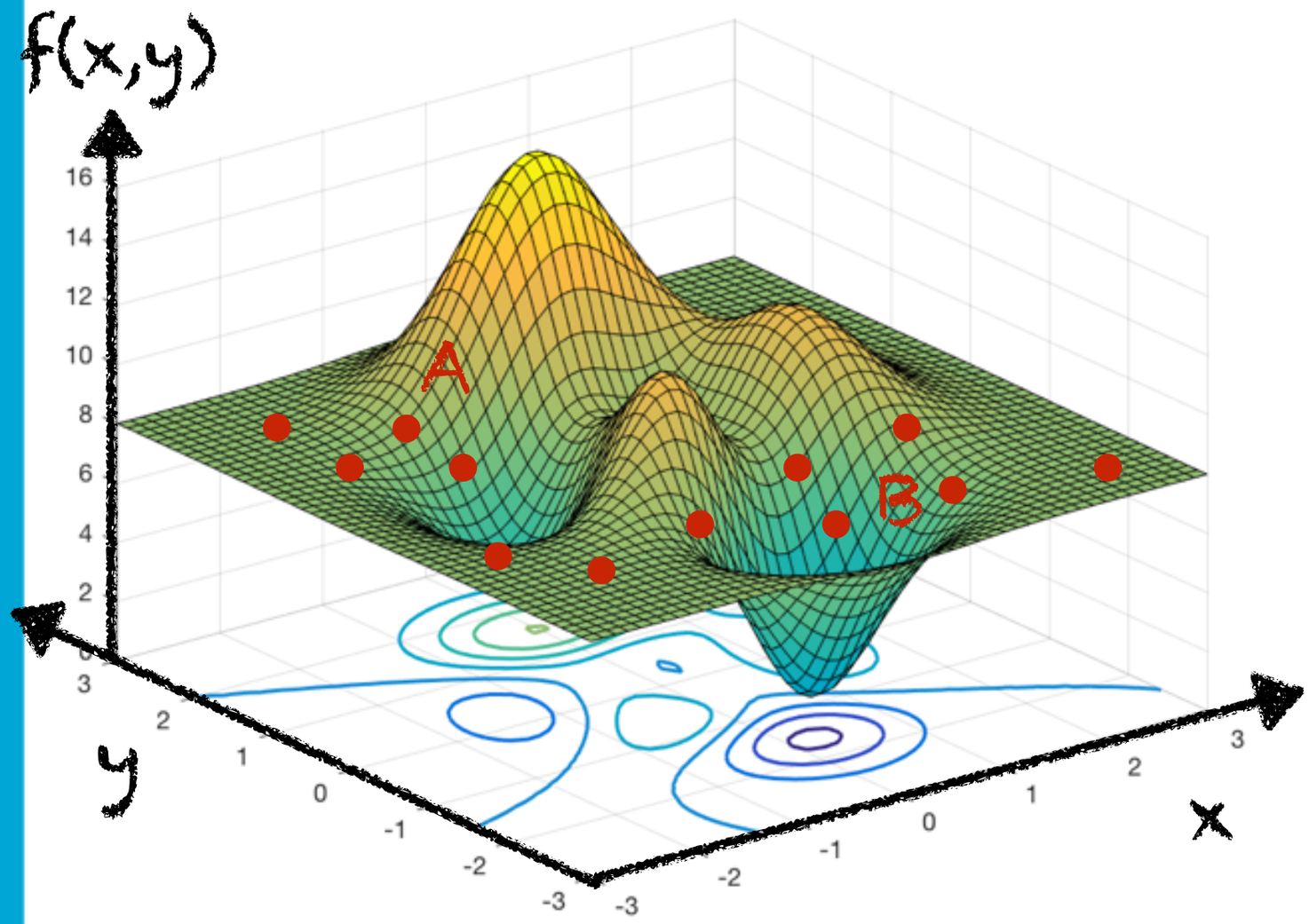
Crossover

1. Select pairs of solutions (parents)

2. Generate new solutions (offsprings)

TUDelft

Genetic Algorithms



Single-point crossover

A
x = **-2**
y = **-1**

B
x = **1**
y = **-2**

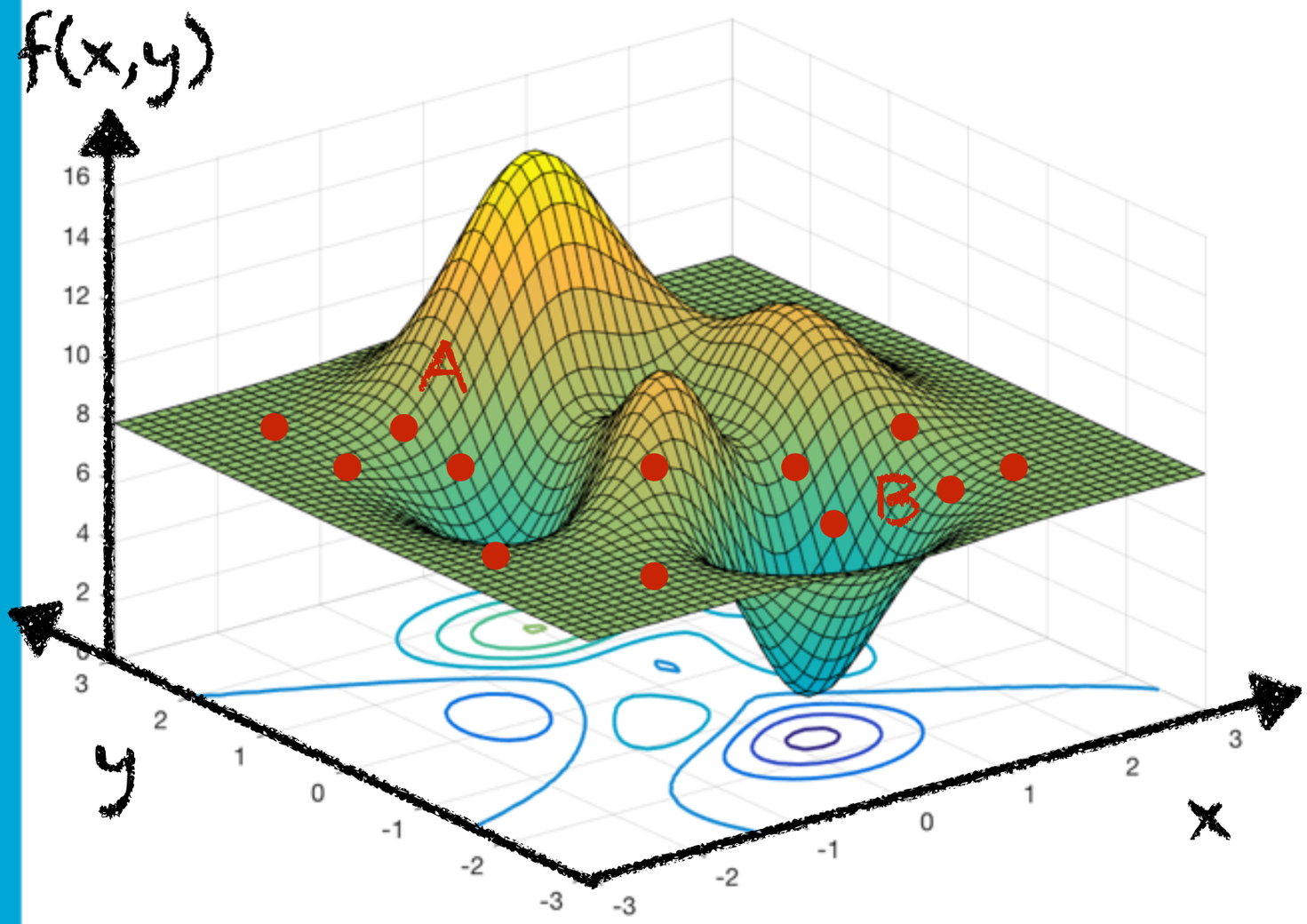
Initial Population

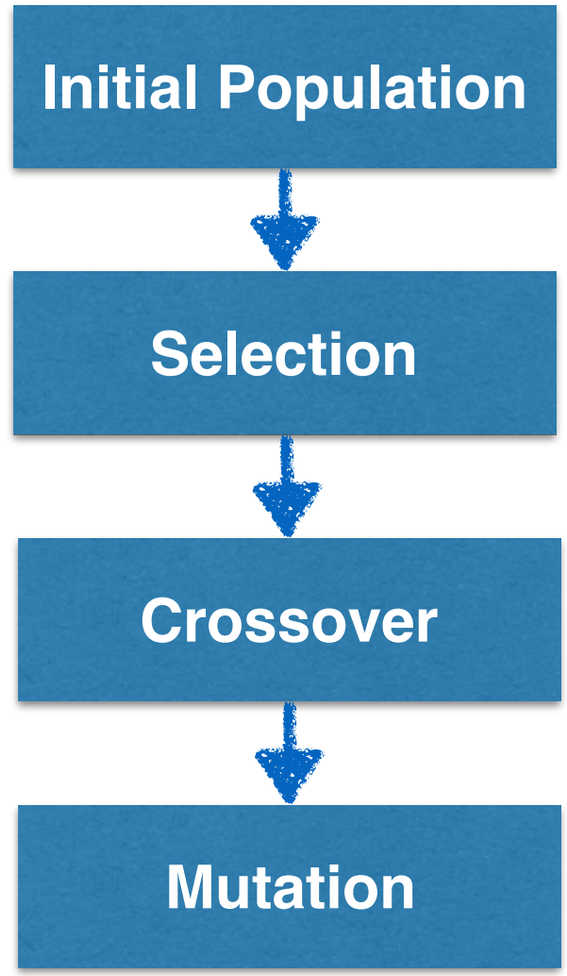
Selection

Crossover

TU Delft

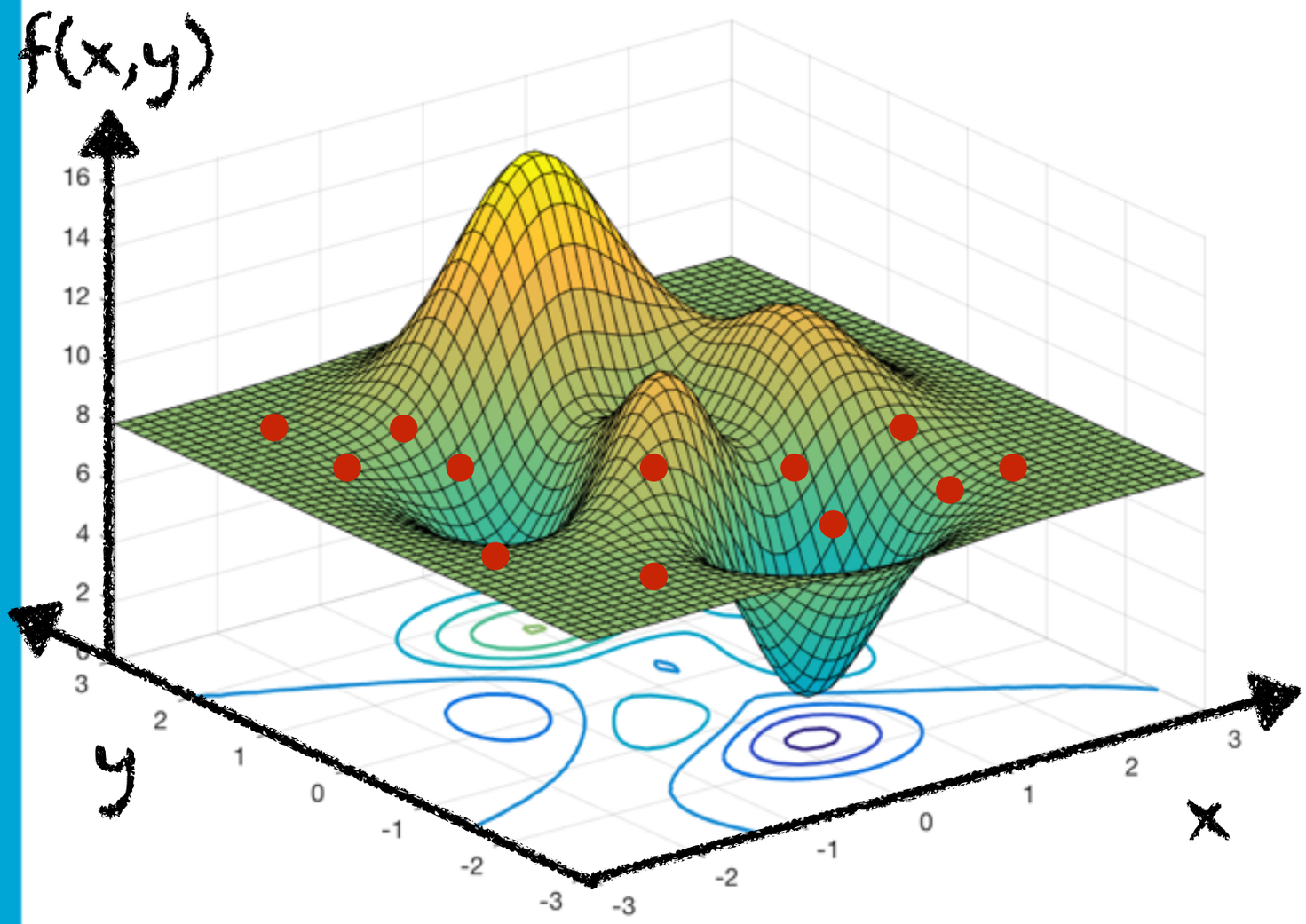
Genetic Algorithms

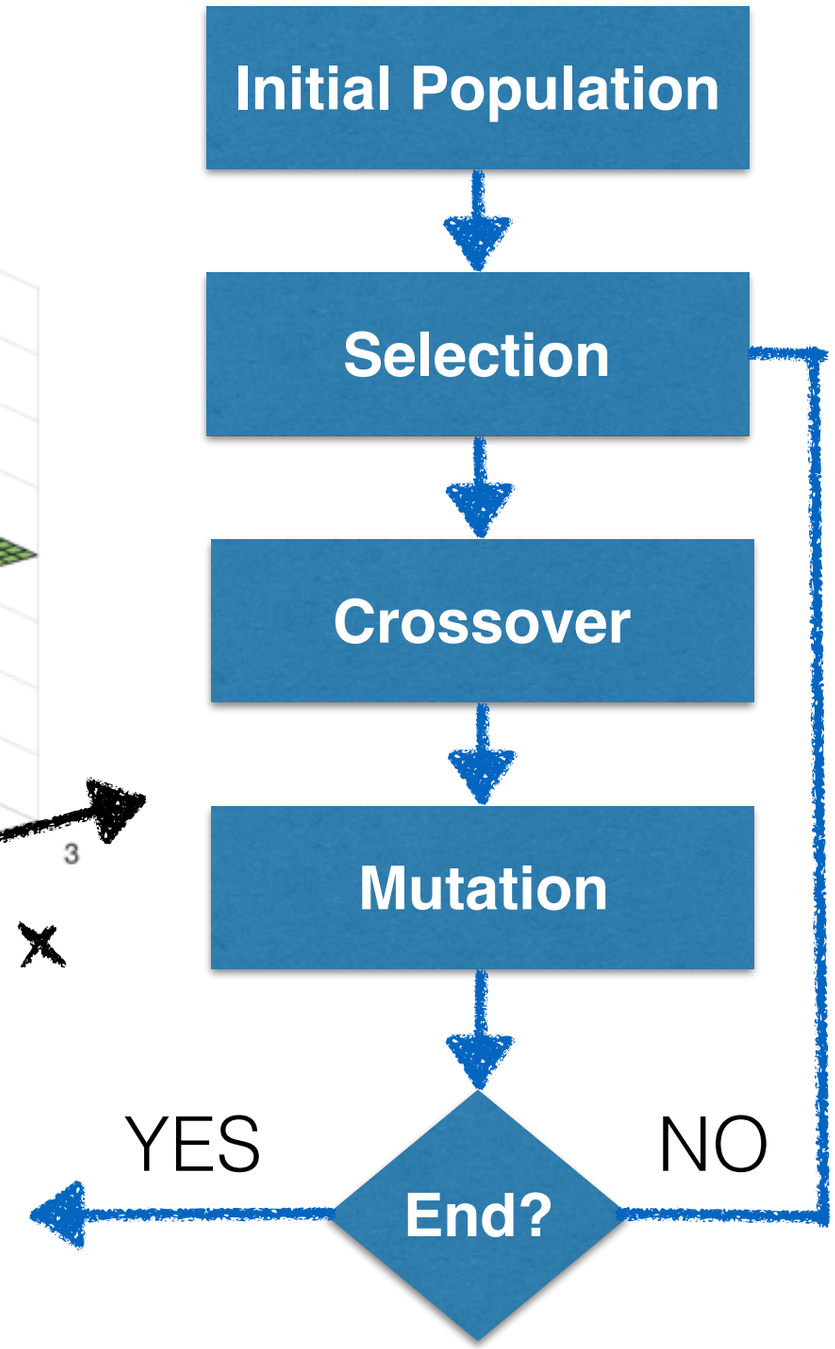




TU Delft

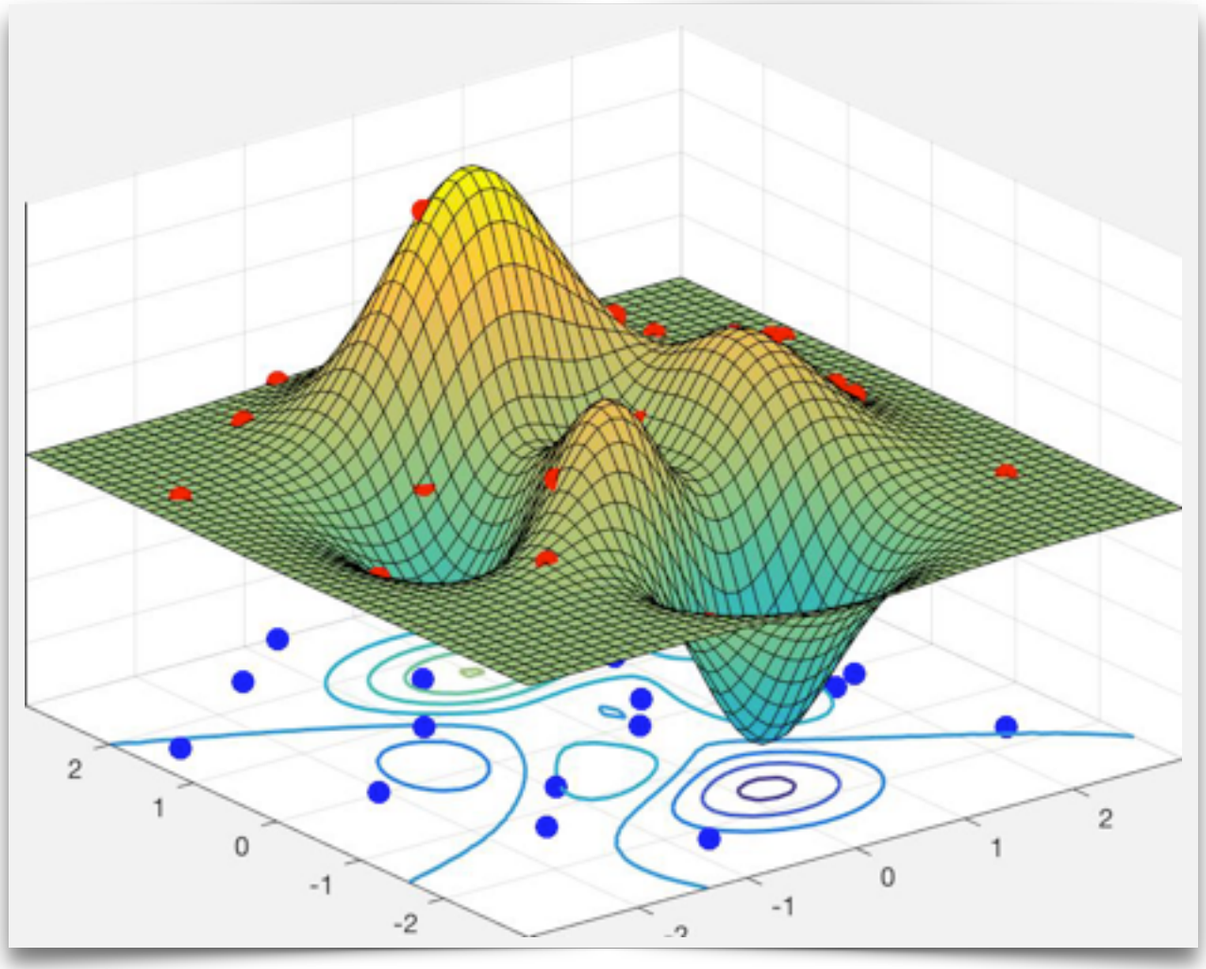
Genetic Algorithms

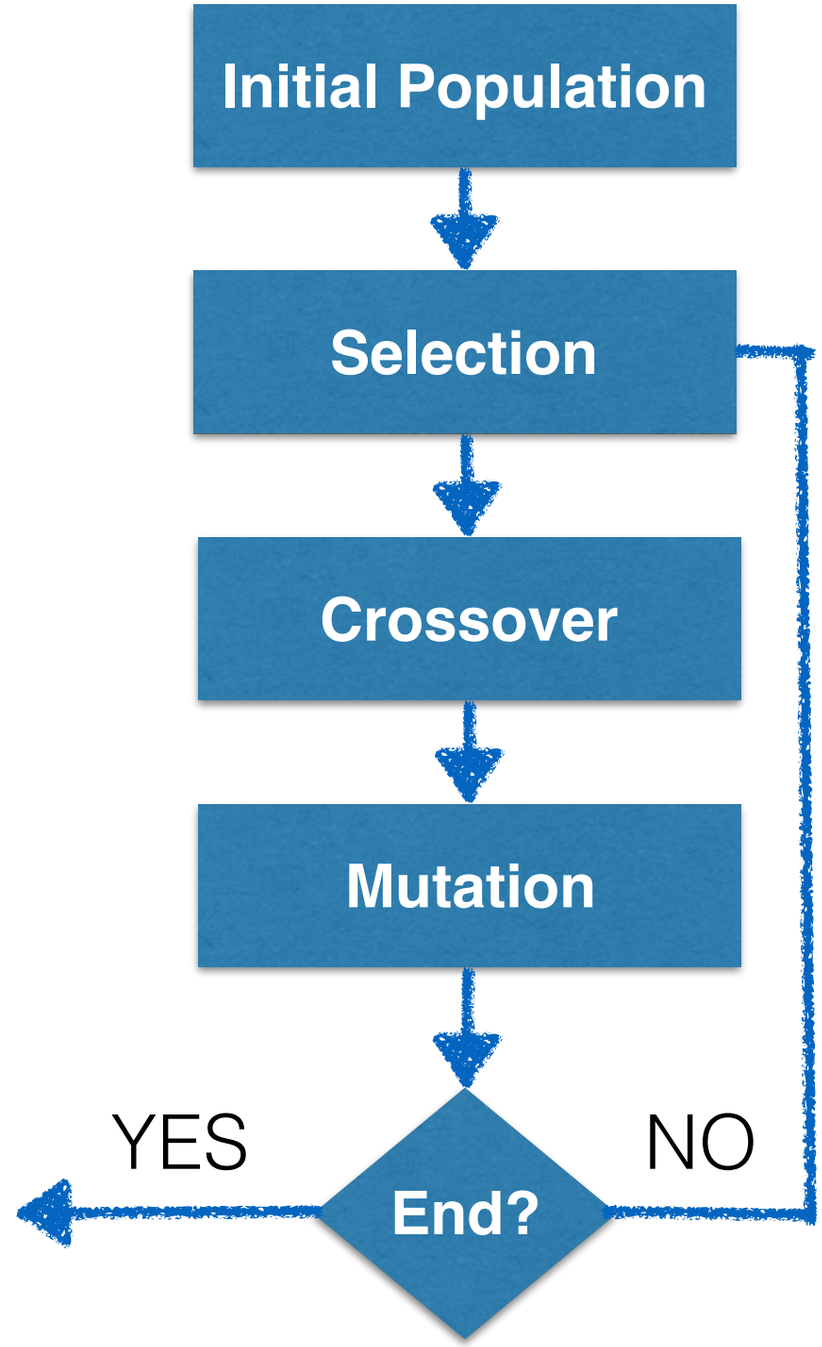




TU Delft

Genetic Algorithms





TU Delft

No Free Lunch Theorem

Theorem: given a search budget ***m*** (e.g., time) the probability of reaching the near optimal value ***f**** using algorithm **a1** is the same as the probability of obtaining the same near-optimal value using another, arbitrarily algorithm **a2**.

$$\sum_f P(d_m^y | f, m, a_1) = \sum_f P(d_m^y | f, m, a_2),$$

If a meta-heuristic **a1** performs better than random search on a problem ***f1***, there will be another problem ***f2*** on which random algorithm performs better than **a2**.

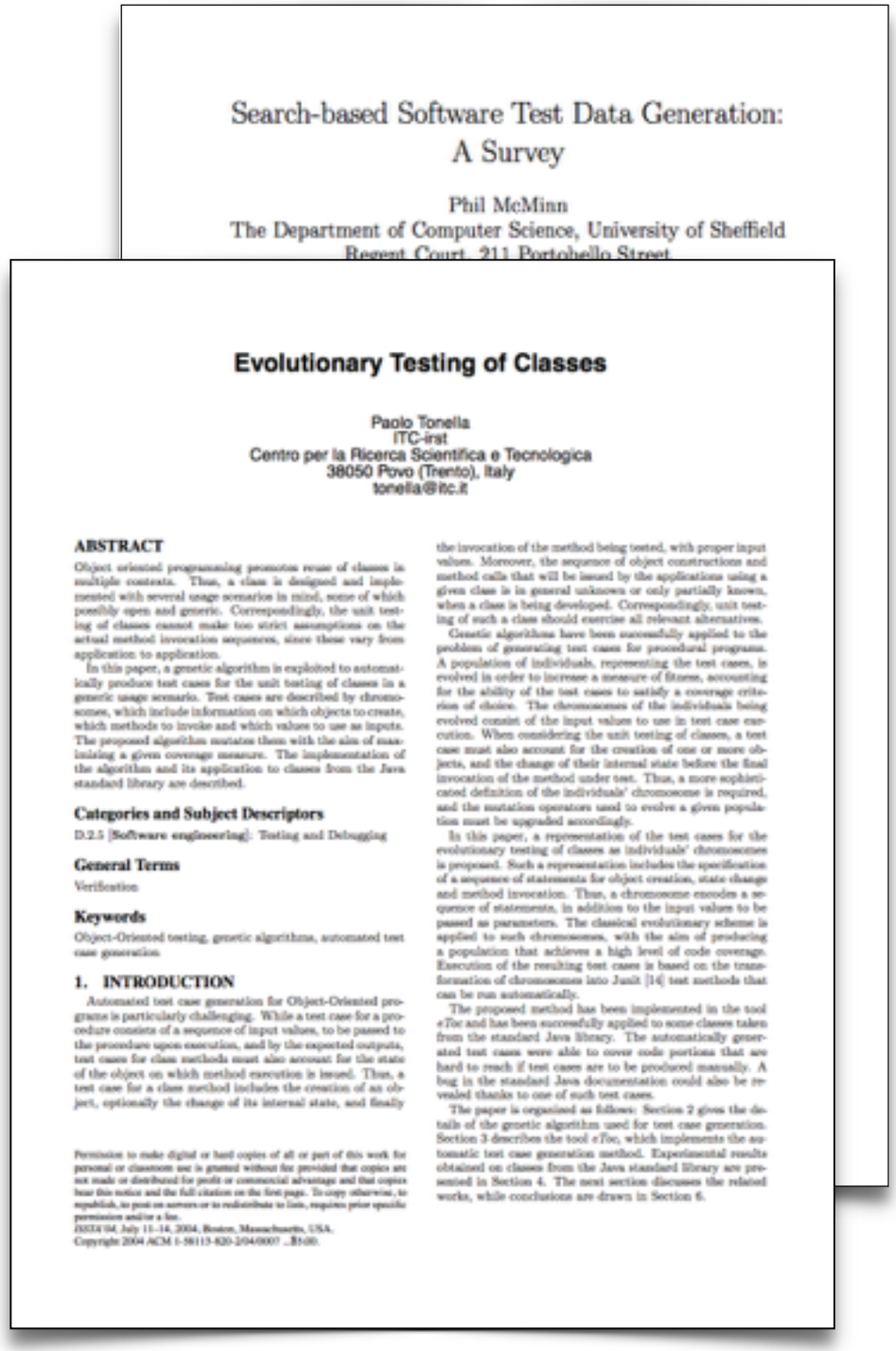
D.H. Wolpert, W.G. W.G. Macready,
No Free Lunch Theorems for Optimization,
IEEE Transactions on Evolutionary Computation, 1997

TUDelft

Automated Test Case Generation

Evolutionary Testing

Phil McMinn, "*Search-based Software Test Data Generation*"
Software Testing, Verification and Reliability


Search-based Software Test Data Generation:
A Survey

Phil McMinn
The Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street

Evolutionary Testing of Classes

Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
tonella@itc.it

ABSTRACT

Object oriented programming promotes reuse of classes in multiple contexts. Thus, a class is designed and implemented with several usage scenarios in mind, some of which possibly open and generic. Correspondingly, the unit testing of classes cannot make too strict assumptions on the actual method invocation sequences, since these vary from application to application.

In this paper, a genetic algorithm is exploited to automatically produce test cases for the unit testing of classes in a generic usage scenario. Test cases are described by chromosomes, which include information on which objects to create, which methods to invoke and which values to use as inputs. The proposed algorithm mutates them with the aim of maximizing a given coverage measure. The implementation of the algorithm and its application to classes from the Java standard library are described.

Categories and Subject Descriptors

D.2.5 [Software engineering]: Testing and Debugging

General Terms

Verification

Keywords

Object-Oriented testing, genetic algorithms, automated test case generation.

1. INTRODUCTION

Automated test case generation for Object-Oriented programs is particularly challenging. While a test case for a procedure consists of a sequence of input values, to be passed to the procedure upon execution, and by the expected outputs, test cases for class methods must also account for the state of the object on which method execution is issued. Thus, a test case for a class method includes the creation of an object, optionally the change of its internal state, and finally the invocation of the method being tested, with proper input values. Moreover, the sequence of object constructions and method calls that will be issued by the applications using a given class is in general unknown or only partially known, when a class is being developed. Correspondingly, unit testing of such a class should exercise all relevant alternatives.

Genetic algorithms have been successfully applied to the problem of generating test cases for procedural programs. A population of individuals, representing the test cases, is evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice. The chromosomes of the individuals being evolved consist of the input values to use in test case execution. When considering the unit testing of classes, a test case must also account for the creation of one or more objects, and the change of their internal state before the final invocation of the method under test. Thus, a more sophisticated definition of the individuals' chromosome is required, and the mutation operators used to evolve a given population must be upgraded accordingly.

In this paper, a representation of the test cases for the evolutionary testing of classes as individuals' chromosomes is proposed. Such a representation includes the specification of a sequence of statements for object creation, state change and method invocation. Thus, a chromosome encodes a sequence of statements, in addition to the input values to be passed as parameters. The classical evolutionary scheme is applied to such chromosomes, with the aim of producing a population that achieves a high level of code coverage. Execution of the resulting test cases is based on the transformation of chromosomes into Junit [14] test methods that can be run automatically.

The proposed method has been implemented in the tool *eToc* and has been successfully applied to some classes taken from the standard Java library. The automatically generated test cases were able to cover code portions that are hard to reach if test cases are to be produced manually. A bug in the standard Java documentation could also be revealed thanks to one of such test cases.

The paper is organized as follows: Section 2 gives the details of the genetic algorithm used for test case generation. Section 3 describes the tool *eToc*, which implements the automatic test case generation method. Experimental results obtained on classes from the Java standard library are presented in Section 4. The next section discusses the related works, while conclusions are drawn in Section 6.

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

Paolo Tonella, "*Evolutionary testing of classes* "
International Symposium on Software Testing and Analysis, ISSTA 2004

TUDelft

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;

   public Triangle (double a, double b, double c){…}
   private void checkRightAngle() {…}
   public void computeTriangleType() {…}
   private boolean isTriangle() {…}
}
```

TUDelft

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;

   public Triangle (double a, double b, double c){
      side1 = a;
      side2 = b;
      side3 = c;
   }

   private void checkRightAngle() {…}
   public void computeTriangleType() {…}
   private boolean isTriangle() {…}
}
```

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;

   public Triangle (double a, double b, double c){…}

   private void checkRightAngle() {
     if (side1*side1 + side2*side2 == side3*side3)
        type = “RIGHT_ANGLE”;
     else if (side1*side1 + side3*side3 == side2*side2)
        type = “RIGHT_ANGLE”;
     else if (side3*side3 + side2*side2 == side1*side1)
        type = “RIGHT_ANGLE”;
     else
        type = “SCALENE”;
   }

   public void computeTriangleType() {…}
   private boolean isTriangle() {…}
}
```

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;
   public Triangle (double a, double b, double c){…}
   private void checkRightAngle() {…}

   public void computeTriangleType(){…}

   private boolean isTriangle() {
     if (side1<=0)
        return false;
     if (side2<=0)
        return false;
     if (side3<=0)
        return false;

     return true;
   }
}
```

TU Delft

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;
   public Triangle (double a, double b, double c){…}
   private void checkRightAngle() {…}

   public void computeTriangleType(){
     if (isTriangle()){
          if (side1 == side2) {
              if (side2 == side3)
                  type = "EQUILATERAL";
              else
                  type = "ISOSCELES";
          } else {
              if (side1 == side3) {
                  type = "ISOSCELES";
              } else {
                  if (side2 == side3)
                       type = “ISOSCELES”;
                  else
                       checkRightAngle();
              }
          }
     }// if isTriangle()
   }
   private boolean isTriangle() {…}
}
```

TU Delft

Running Example

```
class Triangle {
   private double side1, side2, side3;
   private String type = “NOT_A_TRIANGLE”;

   public Triangle (double a, double b, double c){…}
   private void checkRightAngle() {…}
   public void computeTriangleType() {…}
   private boolean isTriangle() {…}
}
```

Goal: Automatic generation of test cases using GAs in order to achieve the maximum statement coverage

Automated Testing

Generation of test cases

1. Select one statement (target)

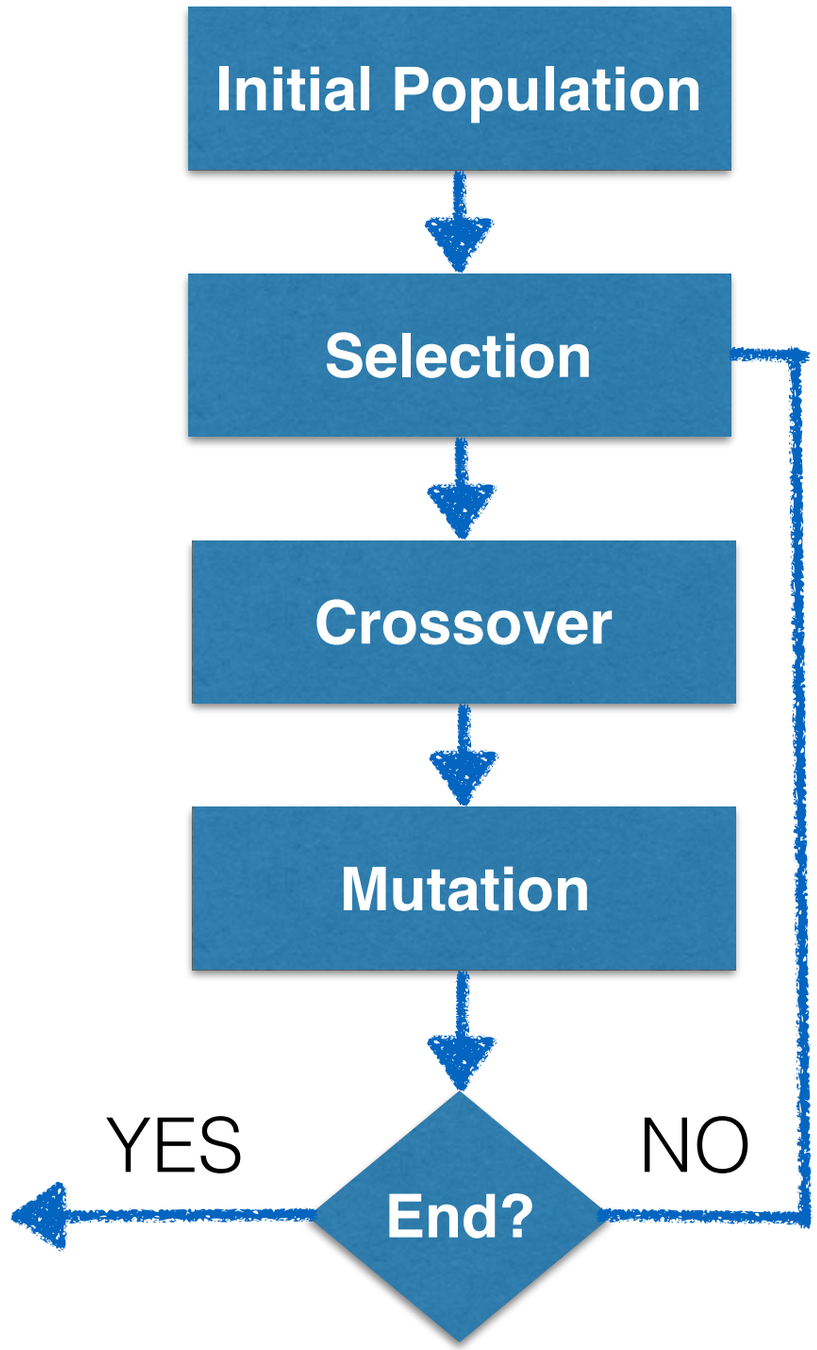
2. Using GAs to search for method calls and input parameters that allow to cover the selected target

3. Store the test case

4. Repeat steps 1-4 until all statements are covered

Goal-oriented
or
Single-target

Genetic Algorithms



TU Delft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Under Test API

```
public ClassUT(…)
…
public ClassUT(…)

public method1(…)
…
public methodK(…)
```

TU Delft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Under Test API

```
public ClassUT(…)
…
public ClassUT(…)

public method1(…)
…
public methodK(…)
```

Constructors

Public Methods

Random Test

```
@Test
public void test(){
  // constructor
  // method calls
  //assertions
}
```

TU Delft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Under Test API

```
public ClassUT(…)
…
public ClassUT(…)

public method1(…)
…
public methodK(…)
```

Pick one of the available constructors

With random parameters input

Random Test

```
@Test
public void test(){
  ClassUT c = new ClassUT();
  // method calls

  //assertions
}
```

TU Delft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Under Test API

```
public ClassUT(…)
…
public ClassUT(…)

public method1(…)
…
public methodK(…)
```

Pick one or more public methods

With random parameters input

Random Test

```
@Test
public void test(){
  ClassUT c = new ClassUT();
  c.method1(…);
  c.method3(…);

  //assertions
}
```

TUDelft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Under Test API

```
public ClassUT(…)
…
public ClassUT(…)

public method1(…)
…
public methodK(…)
```

Use get methods to check the state of the objects after execution

Random Test

```
@Test
public void test(){
  ClassUT c = new ClassUT();
  c.method1(…);
  c.method3(…);

  assertTrue(m.get(),…);
}
```

TUDelft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

How to generate random tests:

1) Pick one of the available constructors (with random input)

2) Pick one or more public methods (with random input)

3) Generate the assertions by checking the final state of the object using get methods

TUDelft

Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

Class Triangle API

```
Triangle (double a,
          double b,
          double c)

computeTriangleType()
```

Random Test

```
@Test
public void test(){
  Triangle c = new Triangle(1.0, 2.1, 4.2);

   c.computeTriangleType();

  assertTrue(c.getTriangleType(), “SCALENE”);
}
```

TU Delft

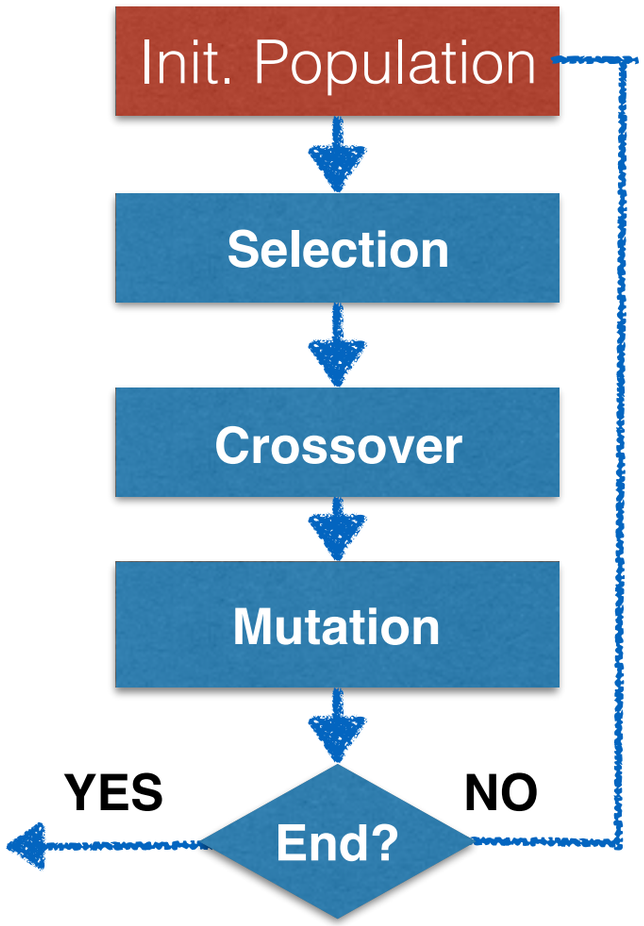
Initial Population

The initial population is a set of randomly generated test cases. Therefore, we need an algorithm for generating the initial set of tests

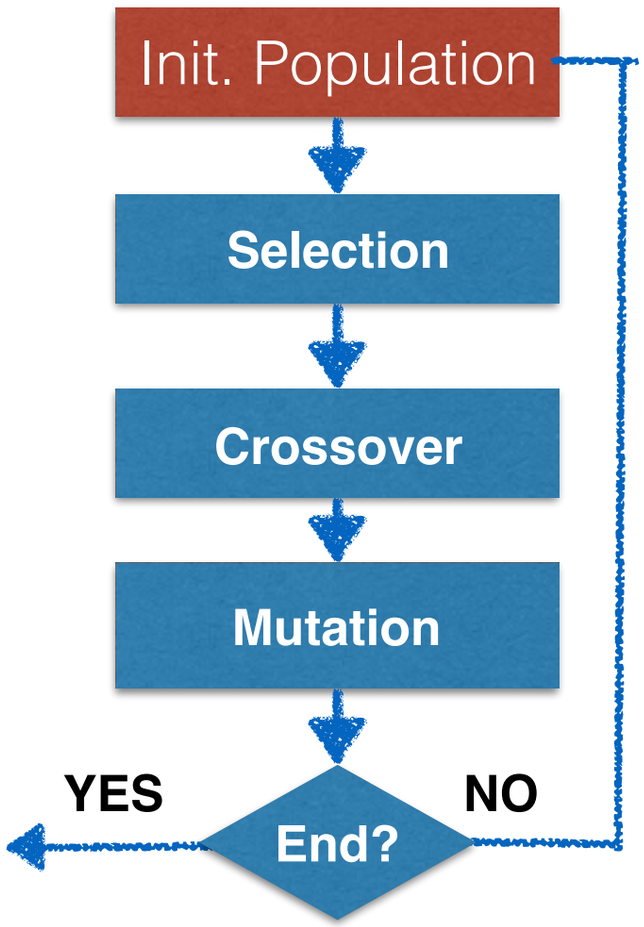
What if one or more input parameters are not primitive types but objects of the same/another class?

TUDelft

Starting GAs



Starting GAs



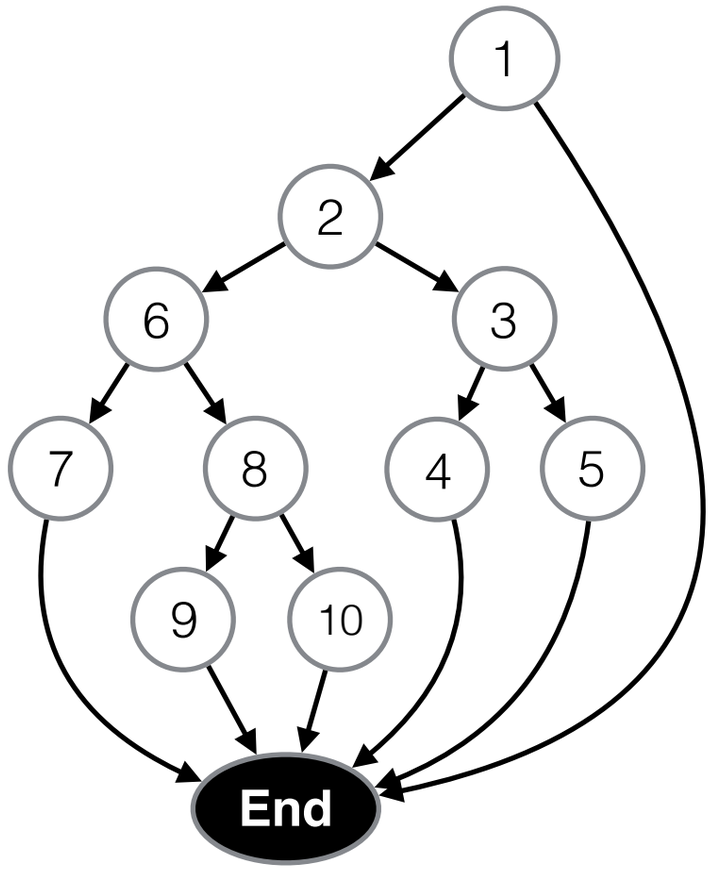
Example of randomly generated initial population:

Pop8 = { x1 = (2,2,3),
x2 = (2,3,5),
x3 = (-2,3,6),
x4 = (2,3,7),
x5 = (2,2,3),
x6 = (3,4,5),
x7 = (3,5,7),
x8 = (6,8,4) }

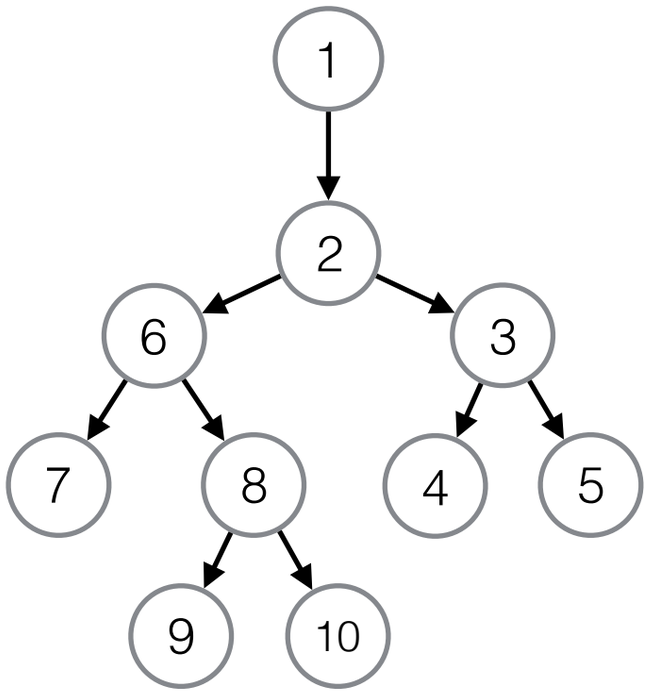
N.B.: In our running example we have only one method with input parameter (the constructor). In the following, we will use only the input vector to denote the test case

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.                type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```



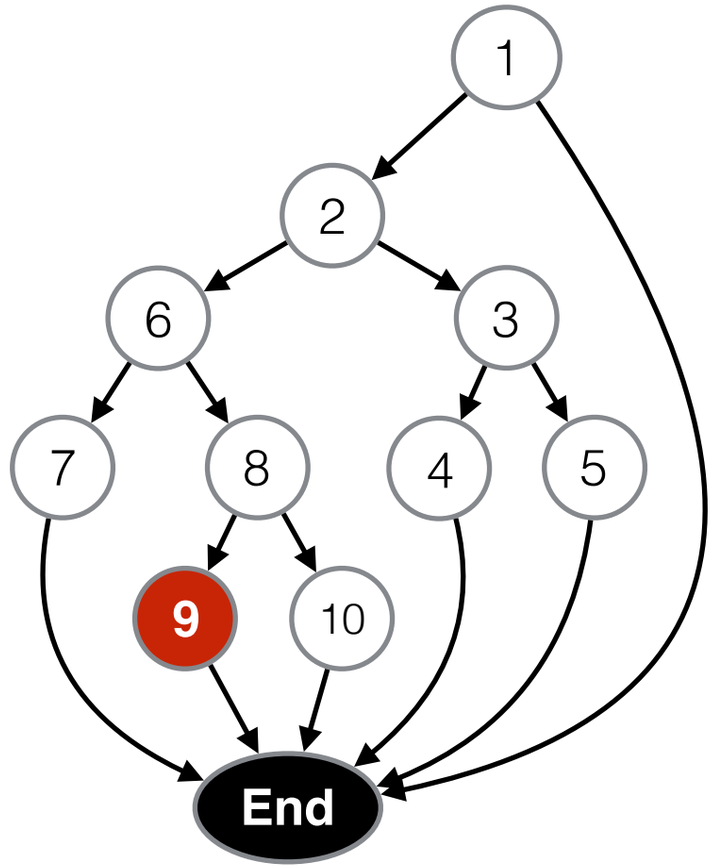
Control flow graph



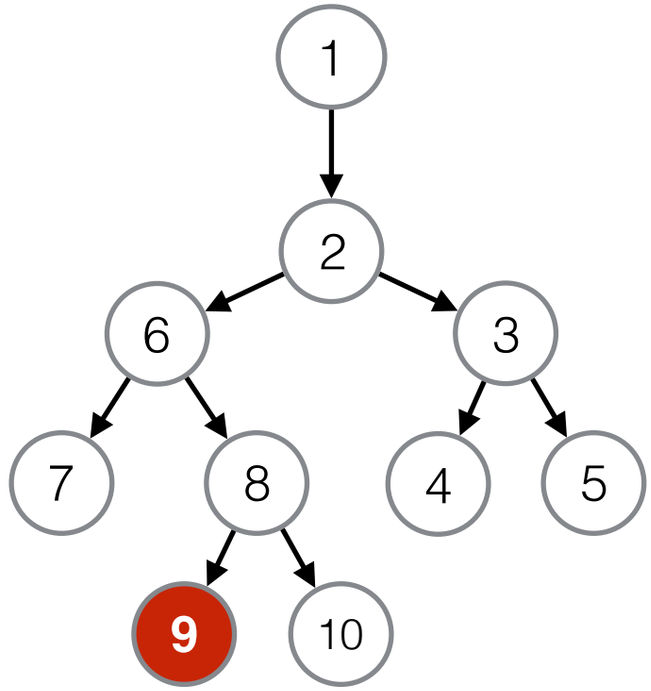
Dependency graph

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.  target ⟶  type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```



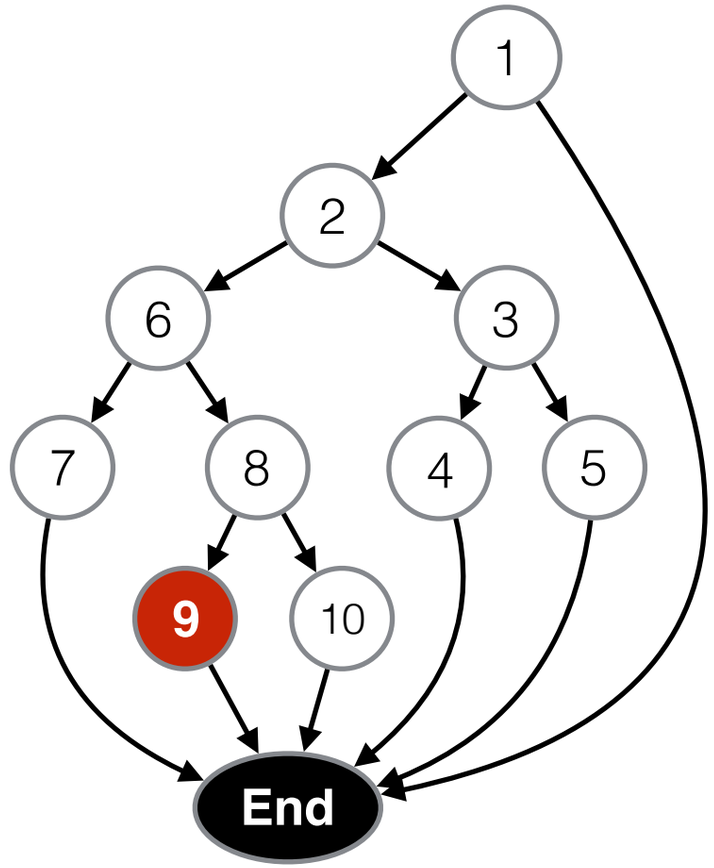
Control flow graph



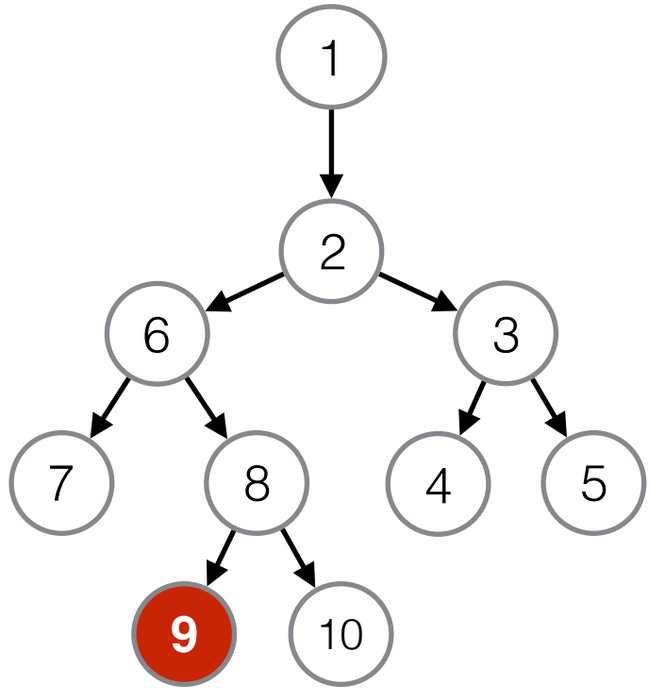
Dependency graph

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.         if (side2 == side3)
4.             type = "EQUILATERAL";
           else
5.             type = "ISOSCELES";
      } else {
6.         if (side1 == side3) {
7.             type = "ISOSCELES";
           } else {
8.             if (side2 == side3)
9.  target ->      type = “ISOSCELES”;
               else
10.               checkRightAngle();
               }
           }
      }// if isTriangle()
  }}
```



Control flow graph



Dependency graph

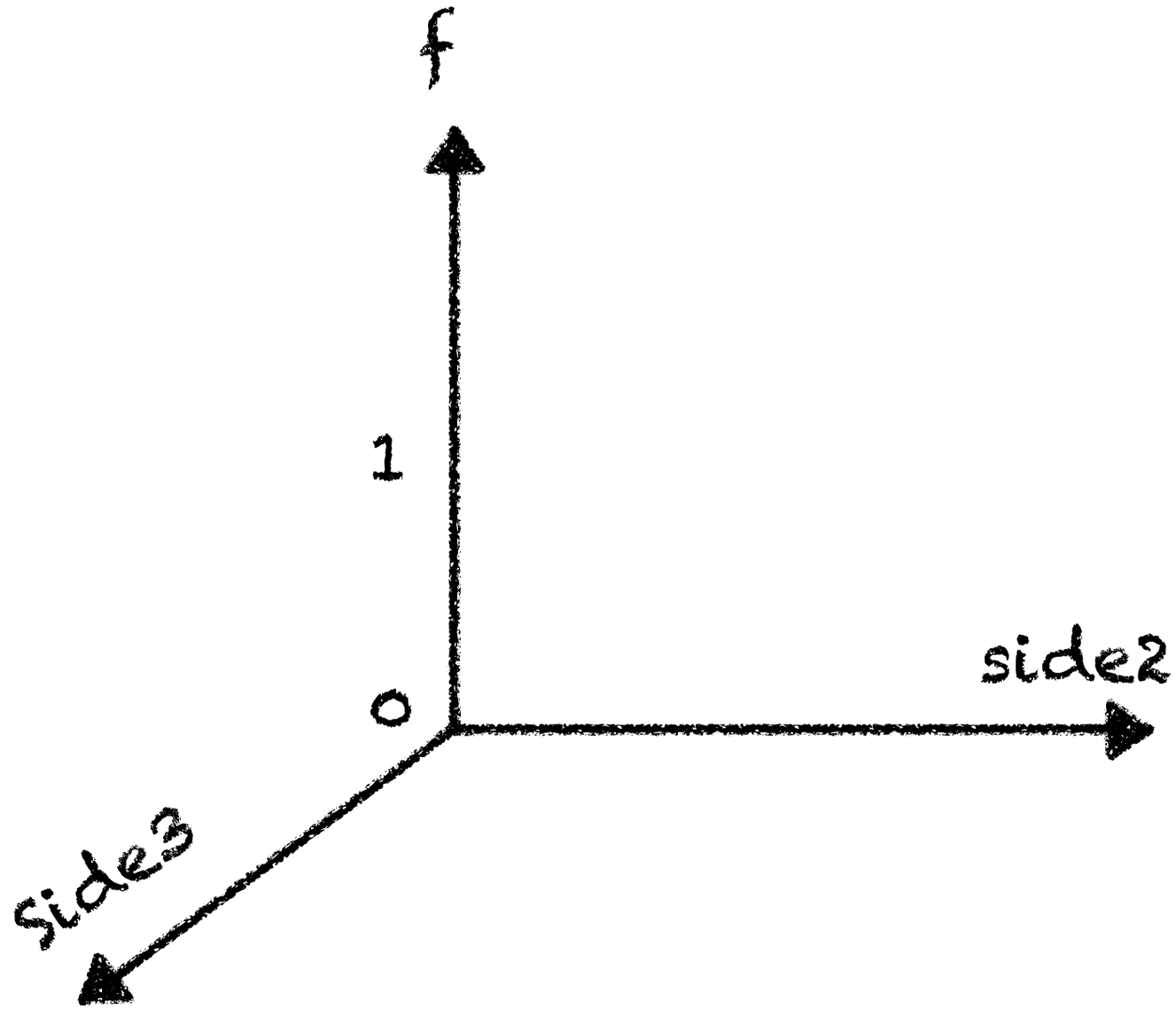
A simple Fitness Function could be:

$$f(side2; side3) = \begin{cases} 0 \text{ if side2 == side3)} \\ 1 \text{ if side2 != side3)} \end{cases}$$

TUDelft

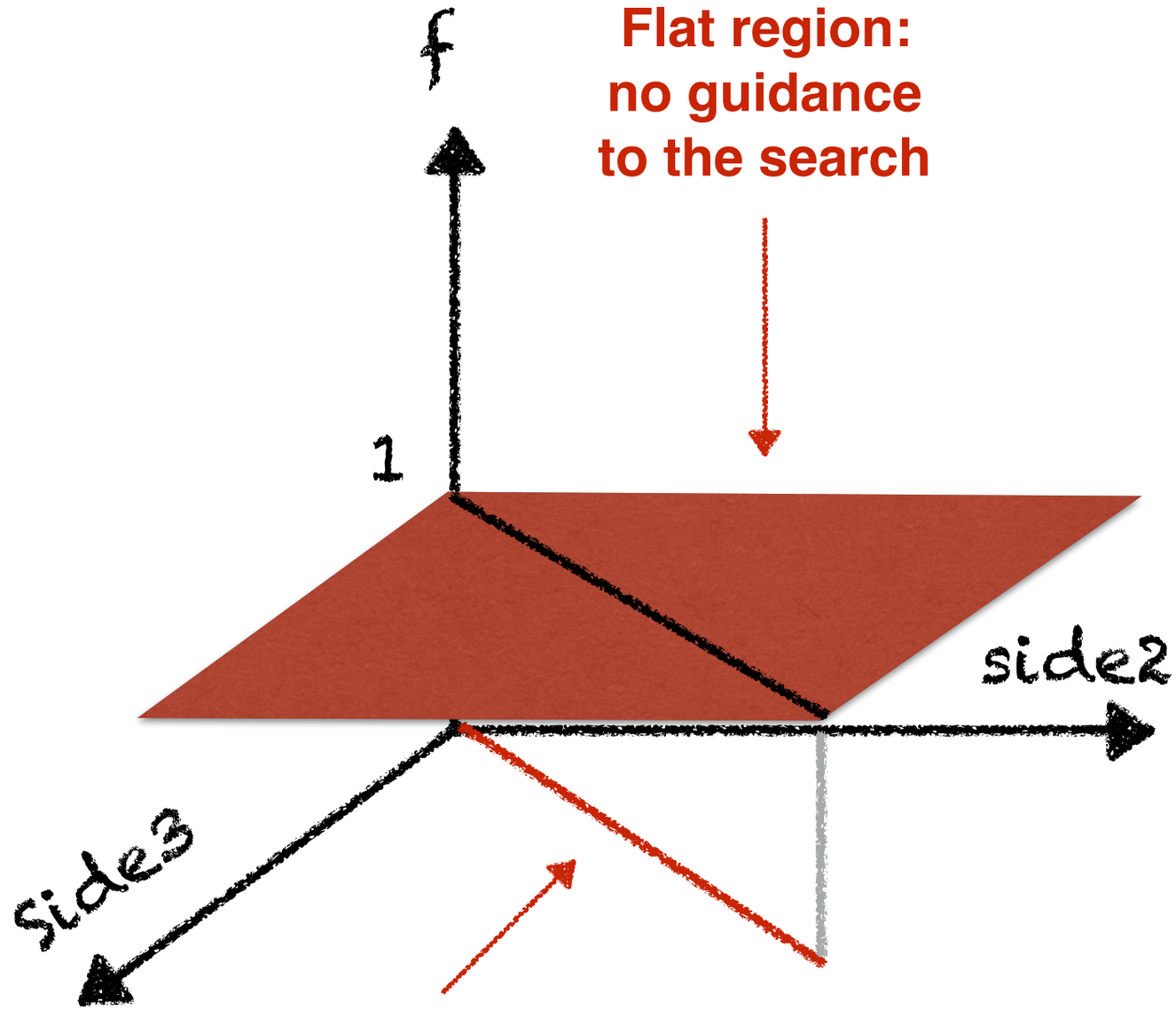
Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.         if (side2 == side3)
4.             type = "EQUILATERAL";
           else
5.             type = "ISOSCELES";
      } else {
6.         if (side1 == side3) {
7.             type = "ISOSCELES";
           } else {
8.             if (side2 == side3)
9.  target ->      type = “ISOSCELES”;
               else
10.                checkRightAngle();
               }
           }
      }// if isTriangle()
  }}
```



Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.         if (side2 == side3)
4.             type = "EQUILATERAL";
           else
5.             type = "ISOSCELES";
      } else {
6.         if (side1 == side3) {
7.             type = "ISOSCELES";
           } else {
8.             if (side2 == side3)
9.  target ->      type = “ISOSCELES”;
               else
10.               checkRightAngle();
               }
           }
      }// if isTriangle()
  }}
```



Flat region: no guidance to the search

Points with side2 == side3

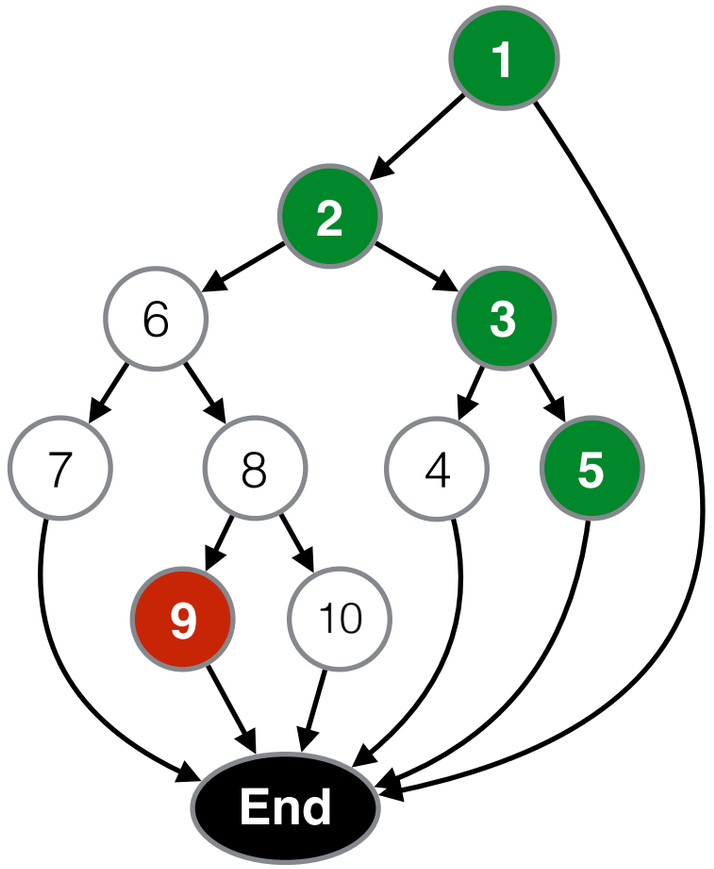
Flag problem: flat fitness function to not guide the search

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



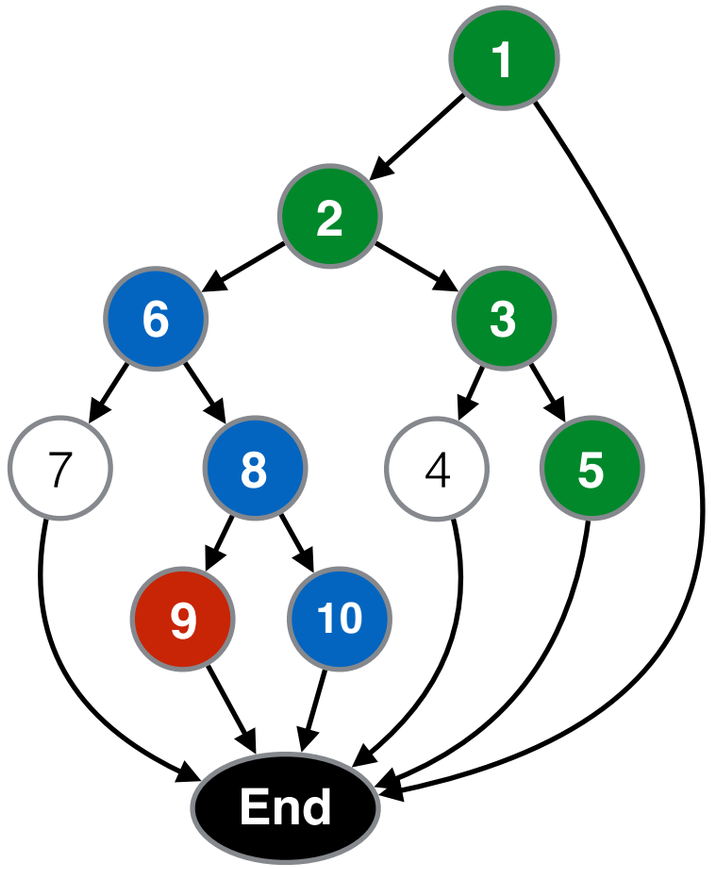
x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>**

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>**

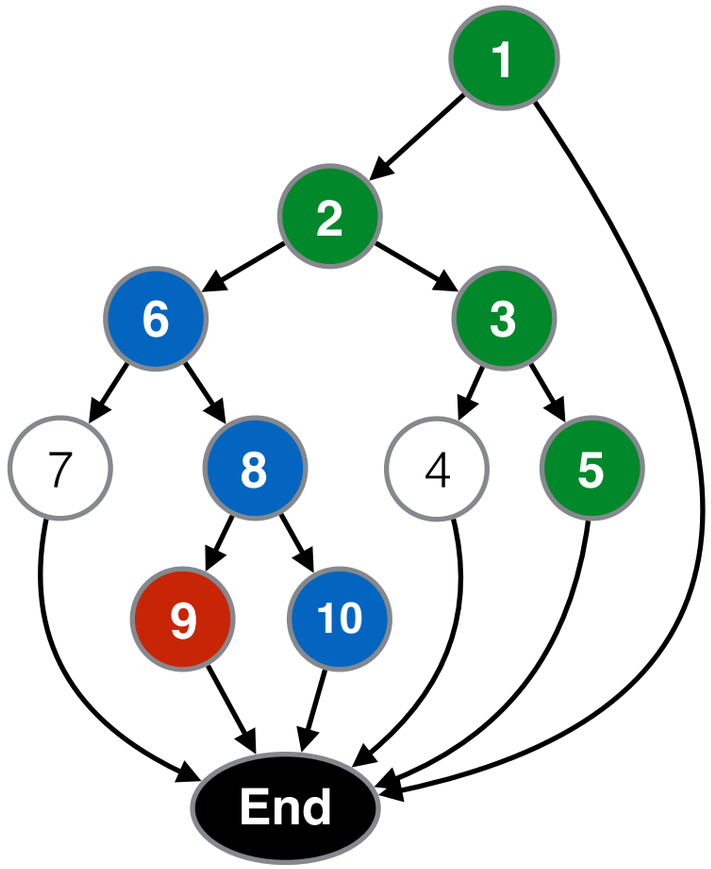
x2 = (2, 3, 5) **Path(x2) = <1, 2, 6, 8, 10>**

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>**

x2 = (2, 3, 5) **Path(x2) = <1, 2, 6, 8, 10>**

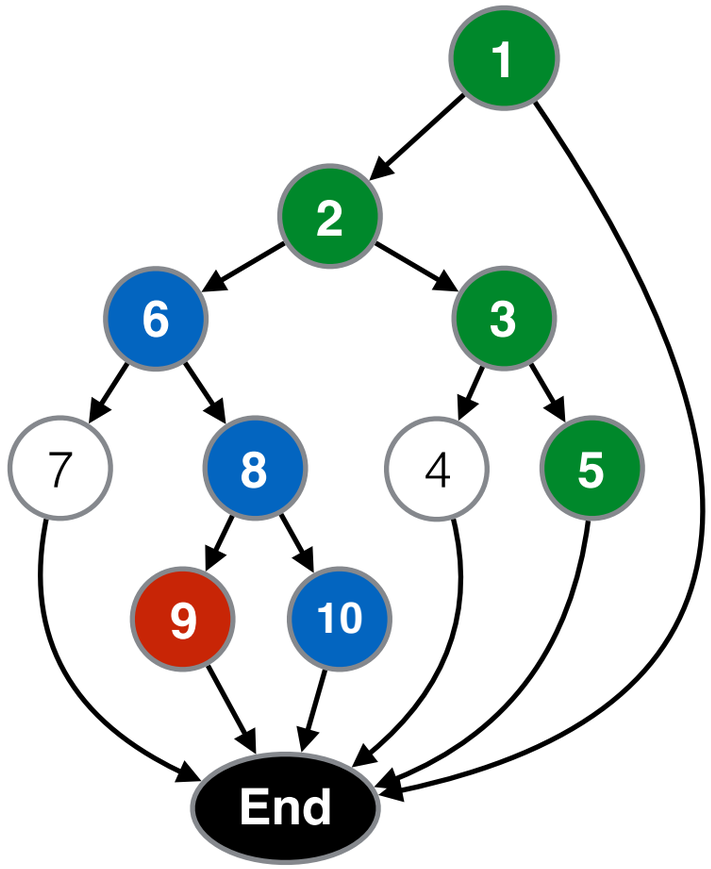
x3 = (-2,10,8) **Path(x3) = ?**

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>

x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>

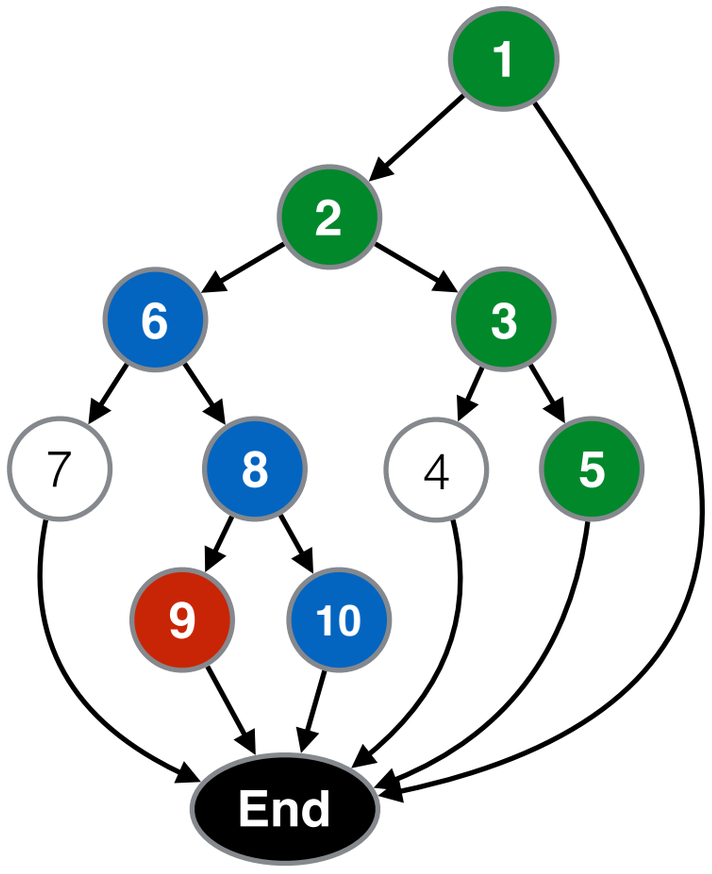
x3 = (-2,10,8) Path(x3) = ?

What is the closest TC to cover the statement 9?

Approach Level

Approach_level(P(x), t)

Given the execution trace obtained by running program **P** with test case **x**, the approach level is the minimum number of control nodes between an executed statement and the coverage target **t**.



x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>**

x2 = (2, 3, 5) **Path(x2) = <1, 2, 6, 8, 10>**

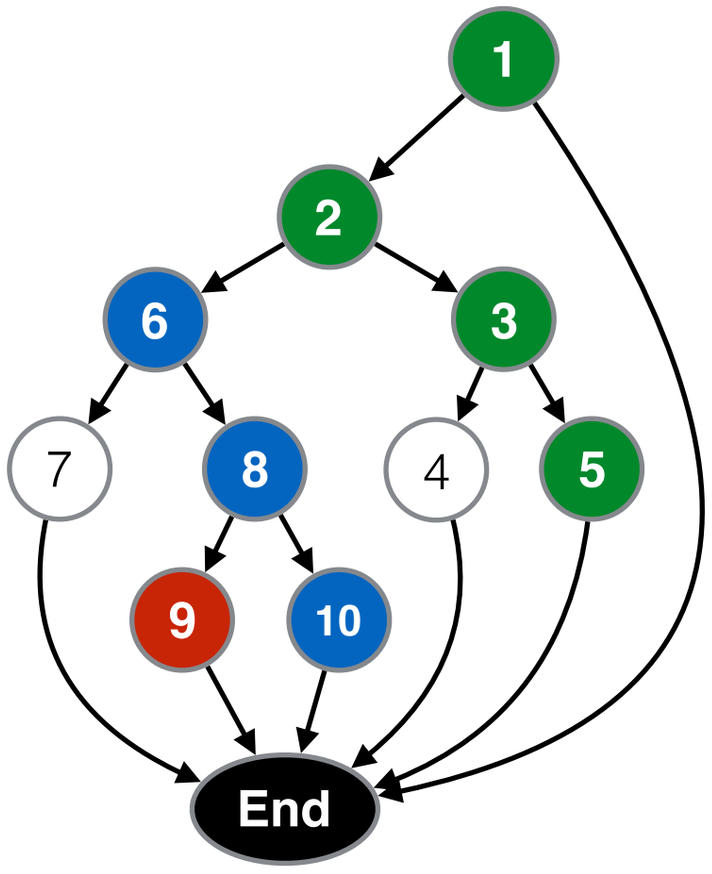
Approach_Level(x1) = ?

Approach_Level(x2) = ?

Approach Level

Approach_level(P(x), t)

Given the execution trace obtained by running program **P** with test case **x**, the approach level is the minimum number of control nodes between an executed statement and the coverage target **t**.



x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>**

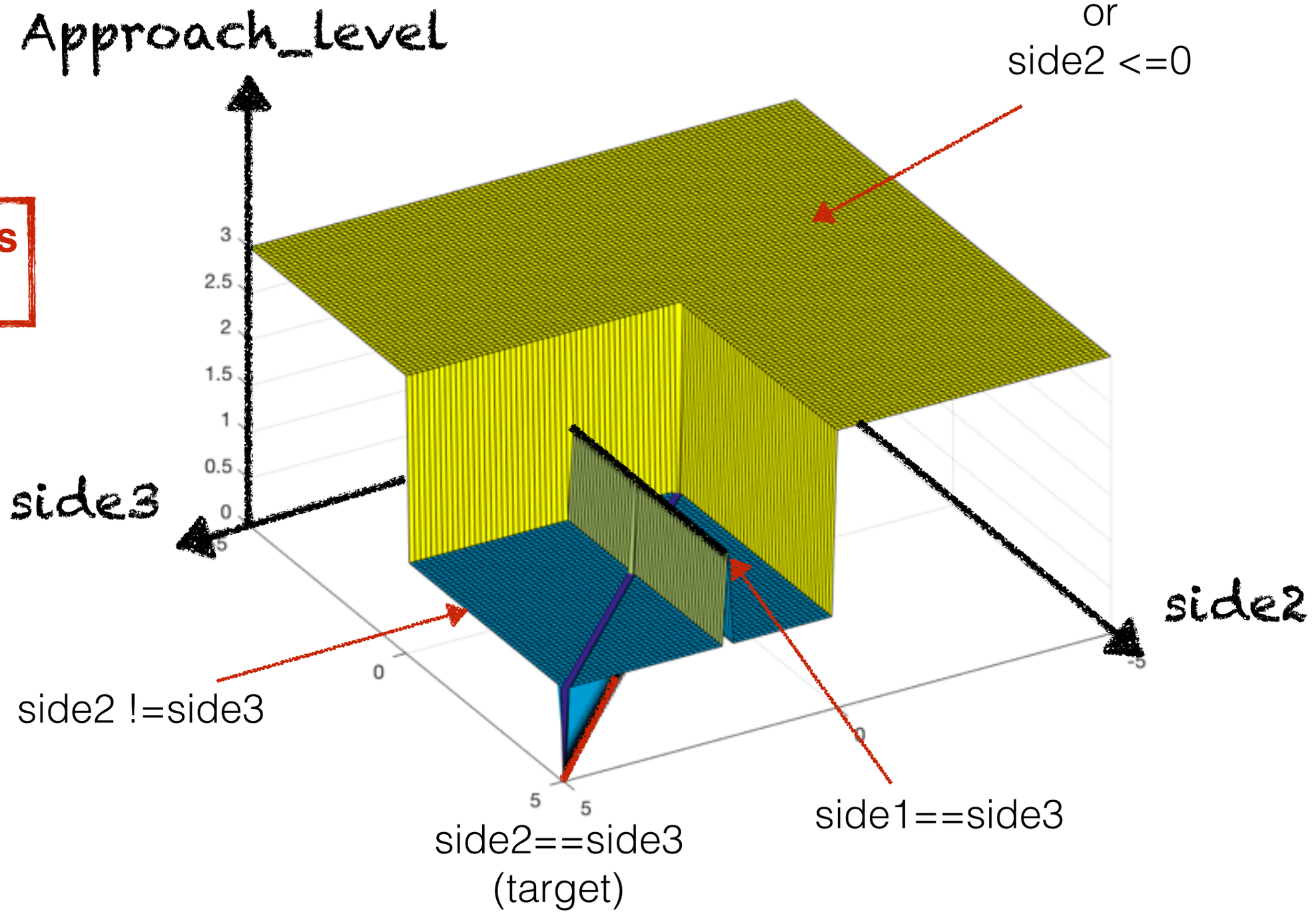
x2 = (2, 3, 5) **Path(x2) = <1, 2, 6, 8, 10>**

Approach_Level(x1) = 2

Approach_Level(x2) = 0

Approach Level

The function is still too flat

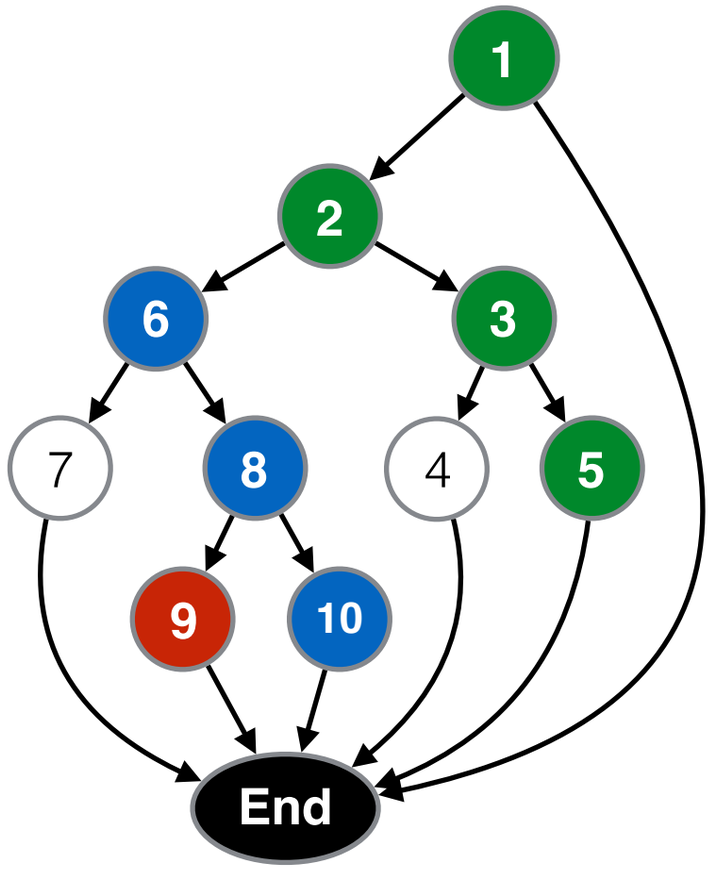


Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



x1 = (2, 2, 3) **Path(x1) = <1, 2, 3, 5>** **AL=2**

x2 = (2, 3, 5) **Path(x2) = <1, 2, 6, 8, 10>** **AL=0**

x3 = (2, 3, 10) **Path(x3) = <1, 2, 6, 8, 10>** **AL=0**

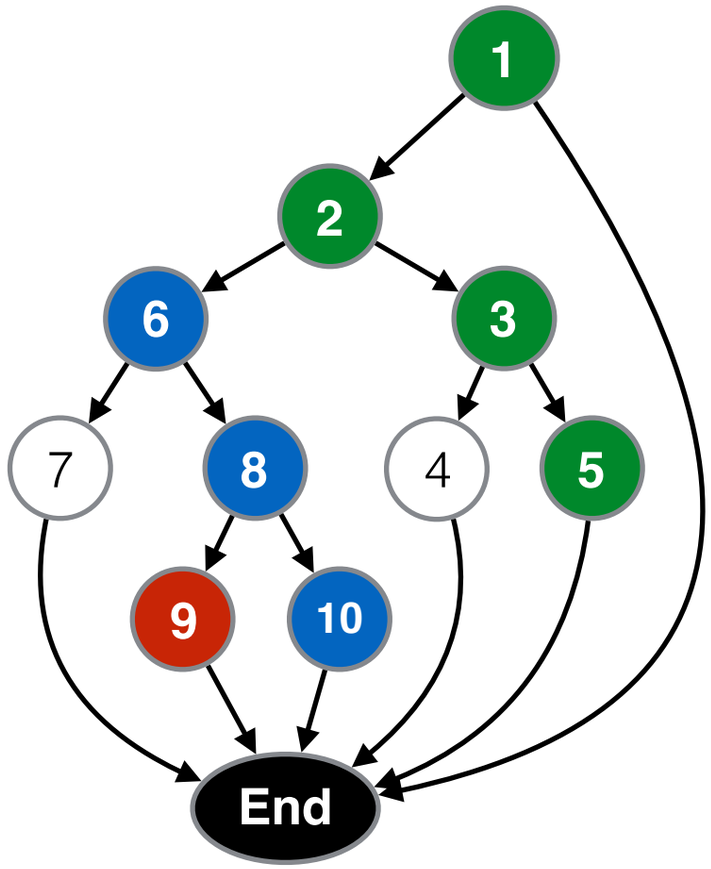
What is the closest TC to cover the statement 9?

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



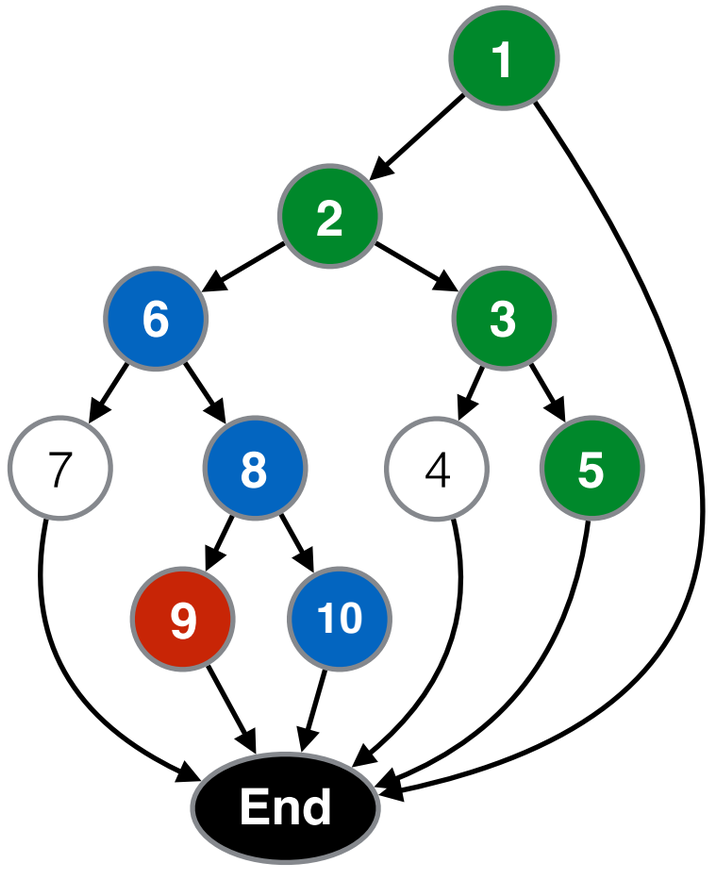
x1 = (2, 2, 3)	Path(x1) = <1, 2, 3, 5>	-
x2 = (2, 3, 5)	Path(x2) = <1, 2, 6, 8, 10>	if (3==5)
x3 = (2, 3, 10)	Path(x3) = <1, 2, 6, 8, 10>	if (2==10)

Fitness Function?

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target

Control flow graph



x1 = (2, 2, 3)	Path(x1) = <1, 2, 3, 5>	-
x2 = (2, 3, 5)	Path(x2) = <1, 2, 6, 8, 10>	abs(3-5) =2
x3 = (2, 3, 10)	Path(x3) = <1, 2, 6, 8, 10>	abs(2-10)=8

Branch Distance

Branch_distance(P(x), t)
Given the first control node where the execution diverges from the target **t**, the predicate at such node is converted to a distance (from taking the desired branch), normalised between 0 and 1.

Such a distance measure how fare the test case is from taking the desired branch. For boolean and numerical variables a, b:

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a	d = {0 if a == true; K otherwise}
!a	d = {K if a == true; 0 otherwise}
a == b	d = {0 if a == b; abs(a - b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a - b + K otherwise}
a <= b	d = {0 if a <= b; a - b + K otherwise}
a > b	d = {0 if a > b; b - a + K otherwise}
a >= b	d = {0 if a >= b; b - a + K otherwise}

Branch Distance

Branch_distance(P(x), t)

For string variables a and b, the branch distance is computed using the following rules:

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a == b	d = {0 if a == b; edit_dist(a, b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a[j] - b[j] + K otherwise}
a <= b	d = {0 if a <= b; a[j] - b[j] + K otherwise}
a > b	d = {0 if a > b; b[j] - a[j] + K otherwise}
a >= b	d = {0 if a >= b; b[j] - a[j] + K otherwise}

where j is the position of the first different character such that **a[j]!=b[j]**, while **a[i] == b[i]** for i<j (a[j]-b[j]) is set to zero if a==b).

Example of edit distance: **edit_dist("abcd", "abbb")=2**

Branch Distance

Branch distance rules for composite predicate

Condition c = composite predicate	Distance BD(c) = d / (d + 1)
!p	Negation is propagated inside p
p & q	d = d(p) + d(q)
p | q	d = min(d(p), d(q))
p XOR q = p & !q | !p & q	d = min(d(p)+d(!q), d(!p)+d(q))

How to normalise the branch distance?

branch_distance(t) = **d/(d+1)**

where **d** is computed according to the distance rules reported in the previous tables

Fitness Function

For statement (or branch) coverage, given a specific coverage target **t**, a widely used fitness function (to be minimised) is:

f(x) = approach_level(P(x), t) + branch_distance(P(x),t)

Approach_level(P(x), t)

Given the execution trace obtained by running program **P** with test case **x**, the approach level is the minimum number of control nodes between an executed statement and the coverage target **t**.

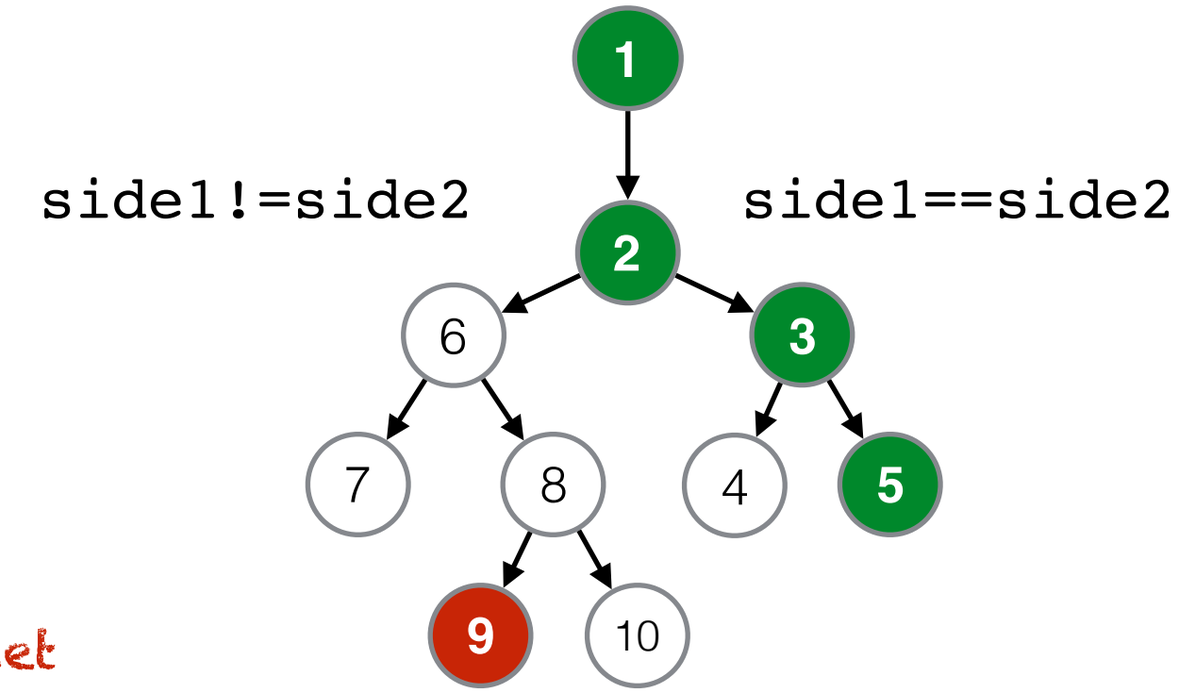
Branch_distance(P(x), t)

Given the first control node where the execution diverges from the target **t**, the predicate at such node is converted to a distance (from taking the desired branch), normalised between 0 and 1.

Fitness Function

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.                type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```

target



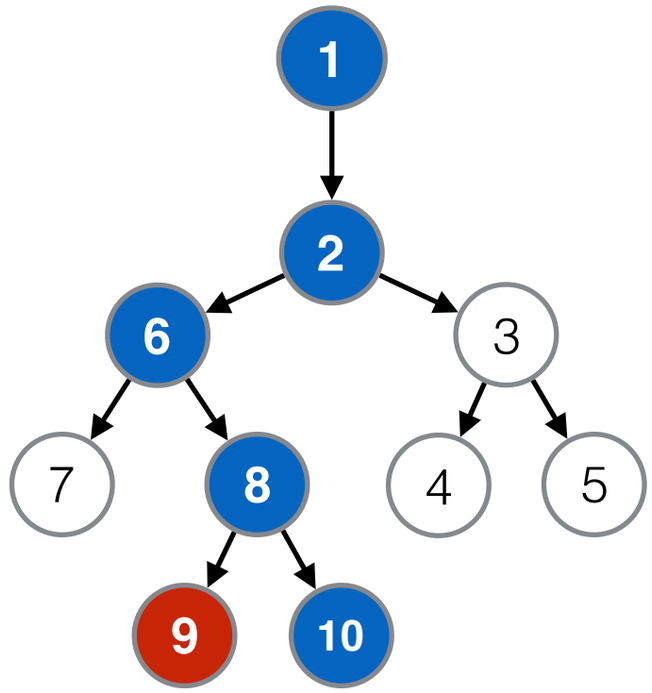
d(2 != 2) = 1
BD (2 != 2) = 1/ (1+1) = 0.5
f(x1) = 2 + 0.5 = 2.5

x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5> AL=2 f = 2.5
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10> AL=0
x3 = (1, 2, 10) Path(x3) = <1, 2, 6, 8, 10> AL=0

Fitness Function

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.                type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```

target



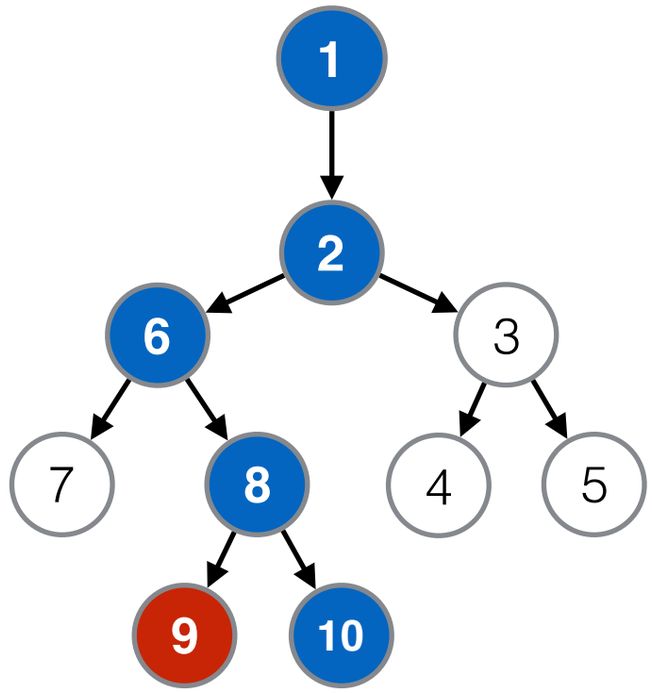
d(3 == 5) = abs(3-5) = 2
BD (3 == 5) = 2 / (2+1) = 0.66
f(Ch1) = 0 + 0.66 = 0.66

x1 = (2, 2, 3)	Path(x1) = <1, 2, 3, 5>	AL=2	f = 2.5
x2 = (2, 3, 5)	Path(x2) = <1, 2, 6, 8, 10>	AL=0	f = 0.66
x3 = (1, 2, 10)	Path(x3) = <1, 2, 6, 8, 10>	AL=0	

Fitness Function

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.                type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```

target



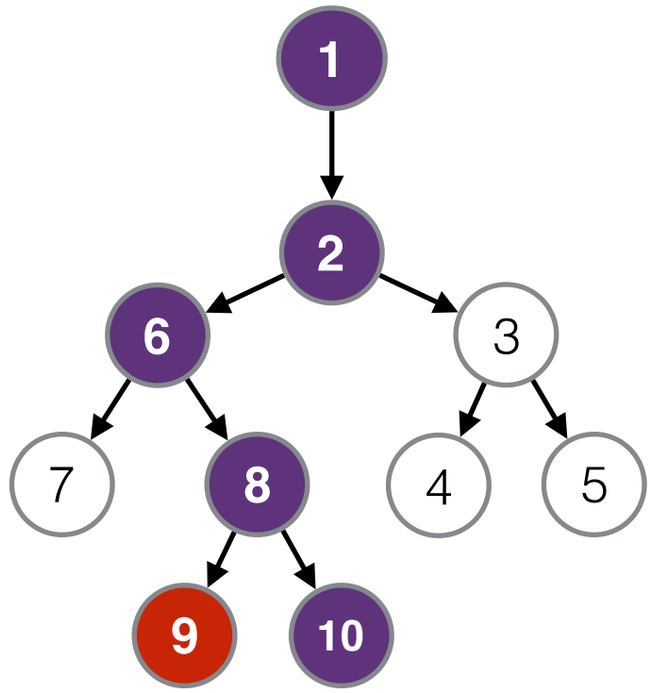
d(3 == 5) = abs(3-5) = 2
BD (3 == 5) = 2 / (2+1) = 0.66
f(Ch1) = 0 + 0.66 = 0.66

x1 = (2, 2, 3)	Path(x1) = <1, 2, 3, 5>	AL=2	f = 2.5
x2 = (2, 3, 5)	Path(x2) = <1, 2, 6, 8, 10>	AL=0	f = 0.66
x3 = (1, 2, 10)	Path(x3) = <1, 2, 6, 8, 10>	AL=0	f = ?

Fitness Function

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.         if (side2 == side3)
4.             type = "EQUILATERAL";
           else
5.             type = "ISOSCELES";
      } else {
6.         if (side1 == side3) {
7.             type = "ISOSCELES";
           } else {
8.             if (side2 == side3)
9.                 type = “ISOSCELES”;
               else
10.                checkRightAngle();
               }
           }
      }// if isTriangle()
  }}
```

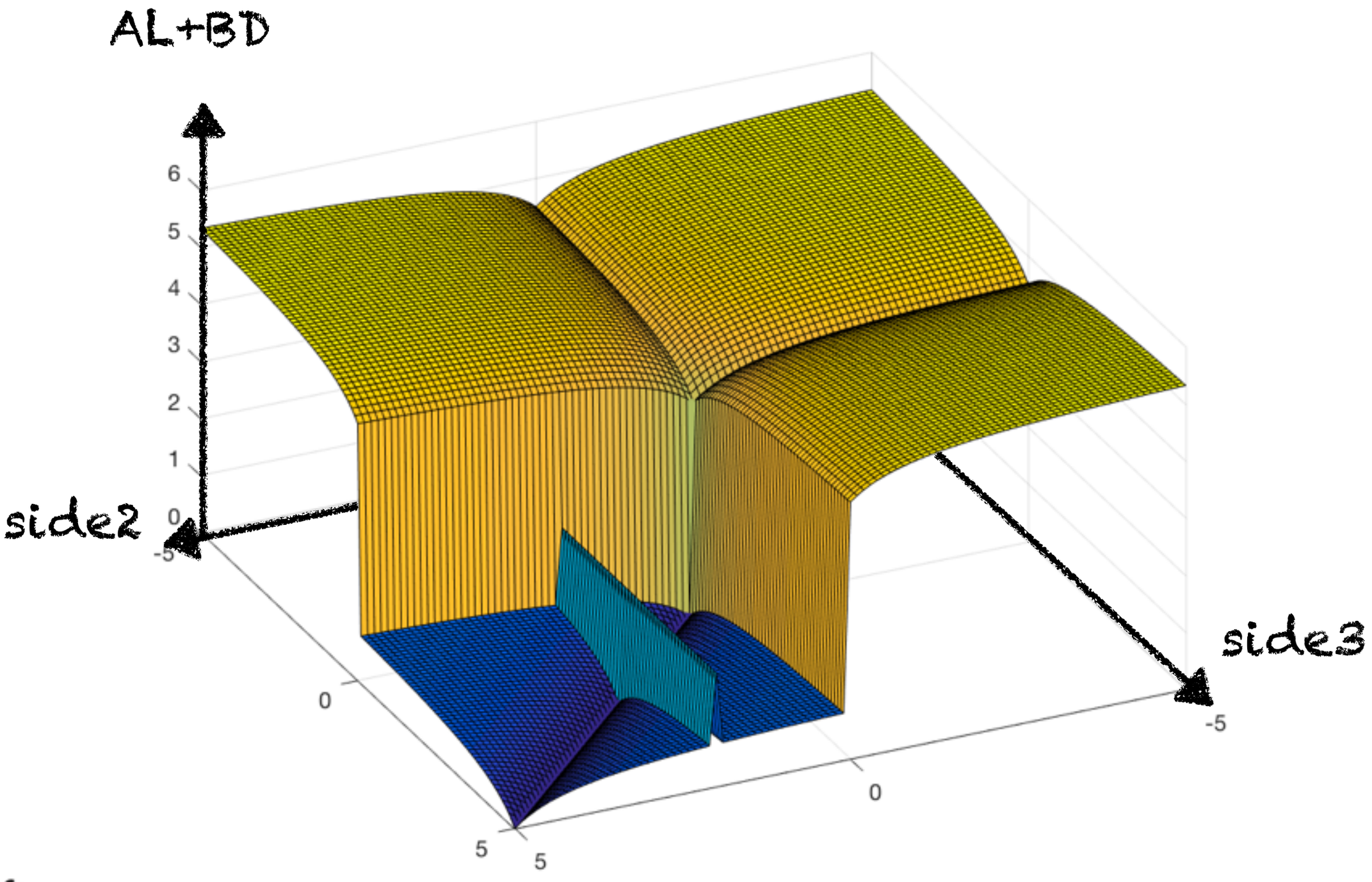
target



d(2 == 10) = abs(2-10) = 8
BD (2 == 10) = 8 / (8+1) = 0.89
f(x3) = 0 + 0.89 = 0.89

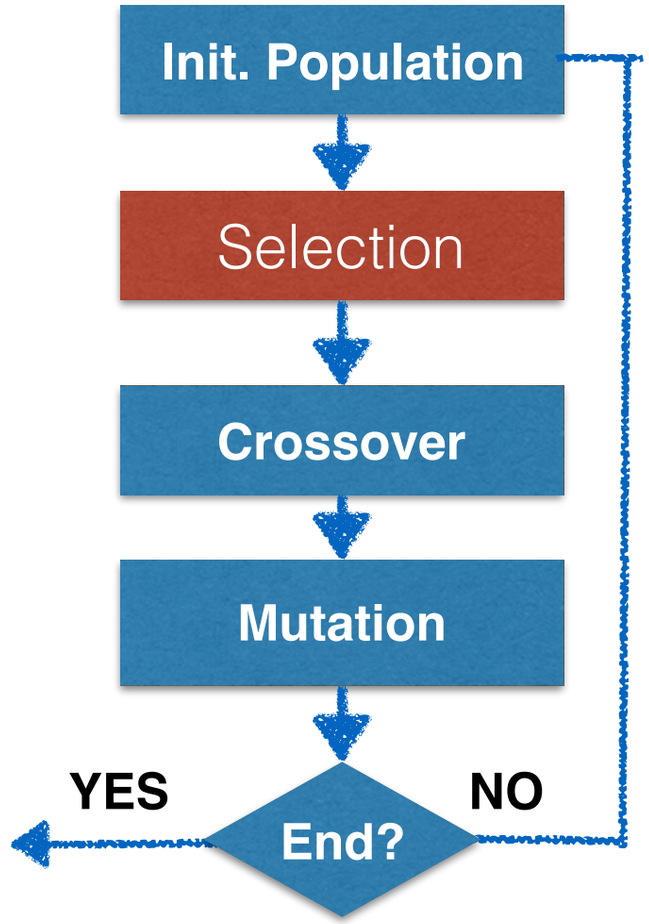
x1 = (2, 2, 3)	Path(x1) = <1, 2, 3, 5>	AL=2	f = 2.5
x2 = (2, 3, 5)	Path(x2) = <1, 2, 6, 8, 10>	AL=0	f = 0.66
x3 = (1, 2, 10)	Path(x3) = <1, 2, 6, 8, 10>	AL=0	f = 0.89

Fitness Function



Selection

x1 = (2,2,3) f = 2.50
x2 = (2,3,5) f = 0.66
x3 = (-2,3,6) f = 3.66
x4 = (2,3,7) f = 0.80
x5 = (2,2,3) f = 2.50
x6 = (3,4,5) f = 0.50
x7 = (3,5,7) f = 0.66
x8 = (6,8,4) f = 0.80



Roulette Wheel Selection

x1 = (2,2,3)	f = 2.50	P ≈ 1/f = 0.40
x2 = (2,3,5)	f = 0.66	P ≈ 1/f = 1.52
x3 = (-2,3,6)	f = 3.66	P ≈ 1/f = 0.27
x4 = (2,3,7)	f = 0.80	P ≈ 1/f = 1.25
x5 = (2,2,3)	f = 2.50	P ≈ 1/f = 0.40
x6 = (3,4,5)	f = 0.50	P ≈ 1/f = 2.00
x7 = (3,5,7)	f = 0.66	P ≈ 1/f = 1.51
x8 = (6,8,4)	f = 0.80	P ≈ 1/f = 1.25

Tot. = 8.60

Roulette wheel selection

1) Assign to each test case a probability equal to 1/f (inverse of the fitness score)

Roulette Wheel Selection

x1 = (2,2,3)	f = 2.50	P ≈ 1/f = **5%**
x2 = (2,3,5)	f = 0.66	P ≈ 1/f = **18%**
x3 = (-2,3,6)	f = 3.66	P ≈ 1/f = **3%**
x4 = (2,3,7)	f = 0.80	P ≈ 1/f = 15%
x5 = (2,2,3)	f = 2.50	P ≈ 1/f = 5%
x6 = (3,4,5)	f = 0.50	P ≈ 1/f = 23%
x7 = (3,5,7)	f = 0.66	P ≈ 1/f = 18%
x8 = (6,8,4)	f = 0.80	P ≈ 1/f = 15%

Tot. = 100%

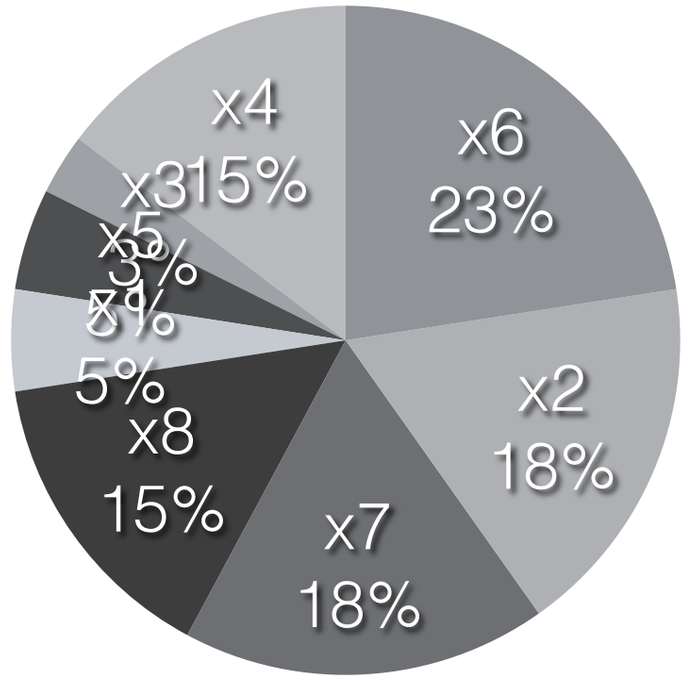
Roulette wheel selection

1) Assign to each test case a probability equal to 1/f (inverse of the fitness score)
2) Normalise the obtained probability

Roulette Wheel Selection

x1 = (2,2,3)	f = 2.50	P ≈ 1/f = **5%**
x2 = (2,3,5)	f = 0.66	P ≈ 1/f = **18%**
x3 = (-2,3,6)	f = 3.66	P ≈ 1/f = **3%**
x4 = (2,3,7)	f = 0.80	P ≈ 1/f = 15%
x5 = (2,2,3)	f = 2.50	P ≈ 1/f = 5%
x6 = (3,4,5)	f = 0.50	P ≈ 1/f = 23%
x7 = (3,5,7)	f = 0.66	P ≈ 1/f = 18%
x8 = (6,8,4)	f = 0.80	P ≈ 1/f = 15%

Tot. = 100%



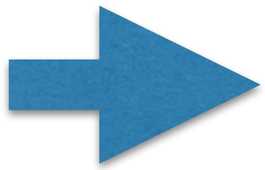
Roulette wheel

Roulette wheel selection

1) Assign to each test case a probability equal to 1/f (inverse of the fitness score)
2) Normalise the obtained probability
3) Each test case has a probability to be selected that is proportional to its slice in the roulette wheel

Tournament Selection

		Tournaments	Winners
x1 = (2,2,3)	f = 2.50	x2, x7	
x2 = (2,3,5)	f = 0.66	x1, x5	
x3 = (-2,3,6)	f = 3.66	x3, x8	
x4 = (2,3,7)	f = 0.80	x3, x2	
x5 = (2,2,3)	f = 2.50	x6, x5	
x6 = (3,4,5)	f = 0.50	x6, x4	
x7 = (3,5,7)	f = 0.66	x8, x3	
x8 = (6,8,4)	f = 0.80	x8, x1	

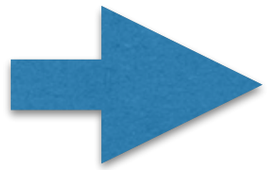


Binary tournament selection

1) Randomly choose pairs of test cases (solutions)
2) Select the fittest (better) individuals from each pair

Tournament Selection

		Tournaments	Winners
x1 = (2,2,3)	f = 2.50	**x2**, x7	x2 = (2,3,5)
x2 = (2,3,5)	f = 0.66	**x1**, x5	x5 = (2,2,3)
x3 = (-2,3,6)	f = 3.66	x3, **x8**	x8 = (6,8,4)
x4 = (2,3,7)	f = 0.80	x3, **x2**	x2 = (2,3,5)
x5 = (2,2,3)	f = 2.50	**x6**, x5	x6 = (3,4,5)
x6 = (3,4,5)	f = 0.50	**x6**, x4	x6 = (3,4,5)
x7 = (3,5,7)	f = 0.66	**x8**, x3	x8 = (6,8,4)
x8 = (6,8,4)	f = 0.80	**x8**, x1	x8 = (6,8,4)



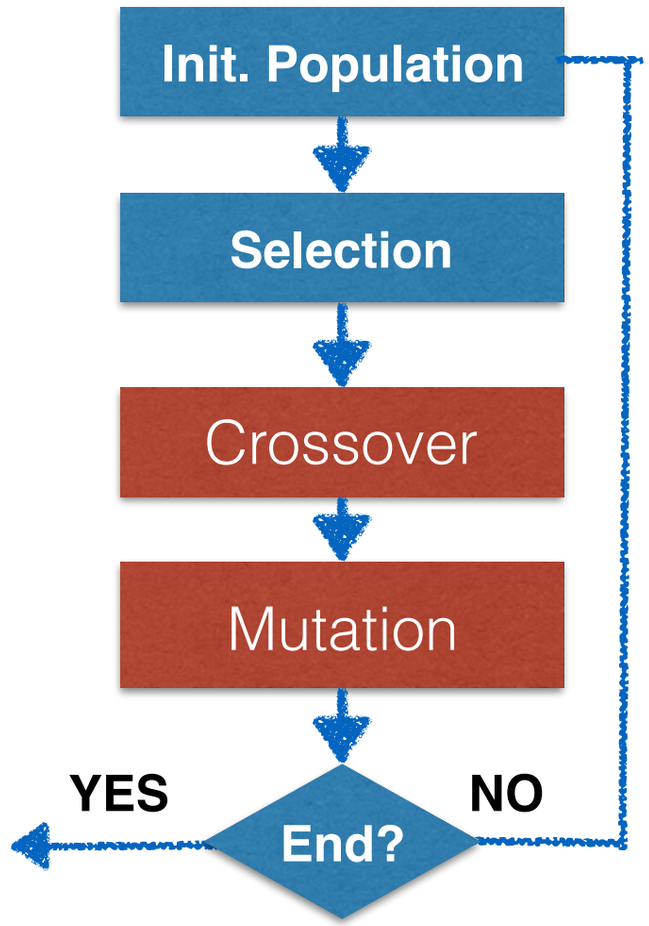
Binary tournament selection

1) Randomly choose pairs of test cases (solutions)
2) Select the fittest (better) individuals from each pair

Reproduction (Crossover)

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target



Reproduction (Crossover)

Winners	Parents	Cut-point	Offsprings
x1 = (2,3,5)	x1	-	o1 = (2,3,5)
x2 = (**2,2,3**)	x2, x5	1	o2 = (**2,4,5**)
x3 = (**6,8,4**)	x3, x8	1	o3 = (**3,2,3**)
x4 = (**2,3,5**)	x4, x6	2	o4 = (**6,8,4**)
x5 = (**3,4,5**)	x7	-	o5 = (**6,8,4**)
x6 = (3,4,5)			o6 = (**2,4,4**)
x7 = (6,8,4)			o7 = (**6,8,5**)
x8 = (**6,8,4**)			o8 = (6,8,4)

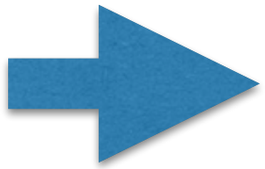
One-point crossover (probability = 0.8)

It takes two parents and cuts their chromosome strings at some randomly chosen position and the produced substrings are then swapped to produce two new full-length chromosomes.

Reproduction (Mutation)

Offsprings

o1 = (2,3,5)
o2 = (2,4,5)
o3 = (3,2,3)
o4 = (6,8,4)
o5 = (6,8,4)
o6 = (2,4,4)
o7 = (6,8,5)
o8 = (6,8,4)



Mutated Offsprings

o1 = (2,3,5)
o2 = (2,4,**2**)
o3 = (3,2,3)
o4 = (**1**,8,4)
o5 = (6,8,4)
o6 = (2,4,4)
o7 = (6,**5**,5)
o8 = (6,8,4)

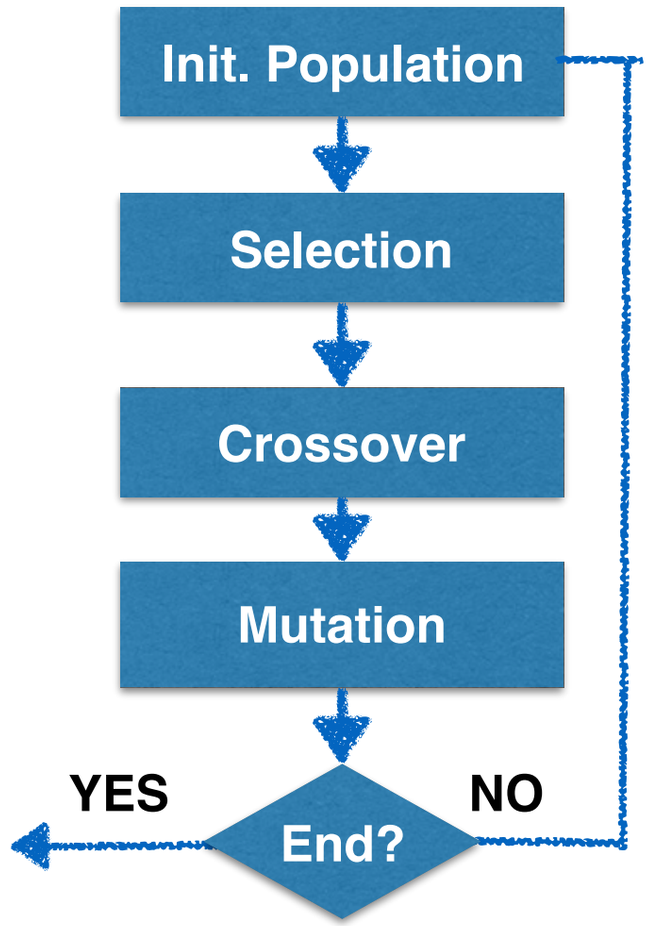
Mutation: randomly change some genes (elements within each chromosome)

Mutation probability: 1/n where n=chromosome length

Iterating (generations)

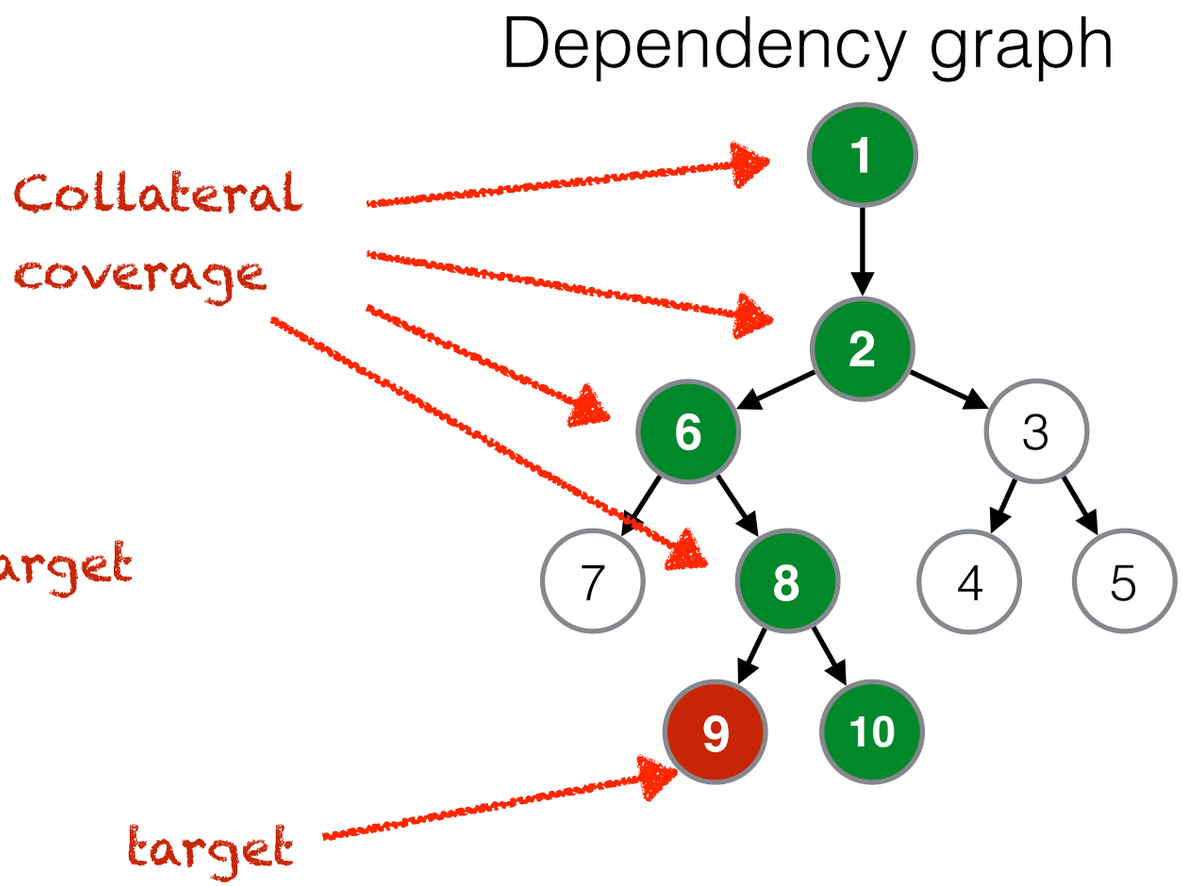
```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
         } else {
8.           if (side2 == side3)
9.               type = “ISOSCELES”;
             else
10.              checkRightAngle();
             }
         }
     }// if isTriangle()
  }}
```

target



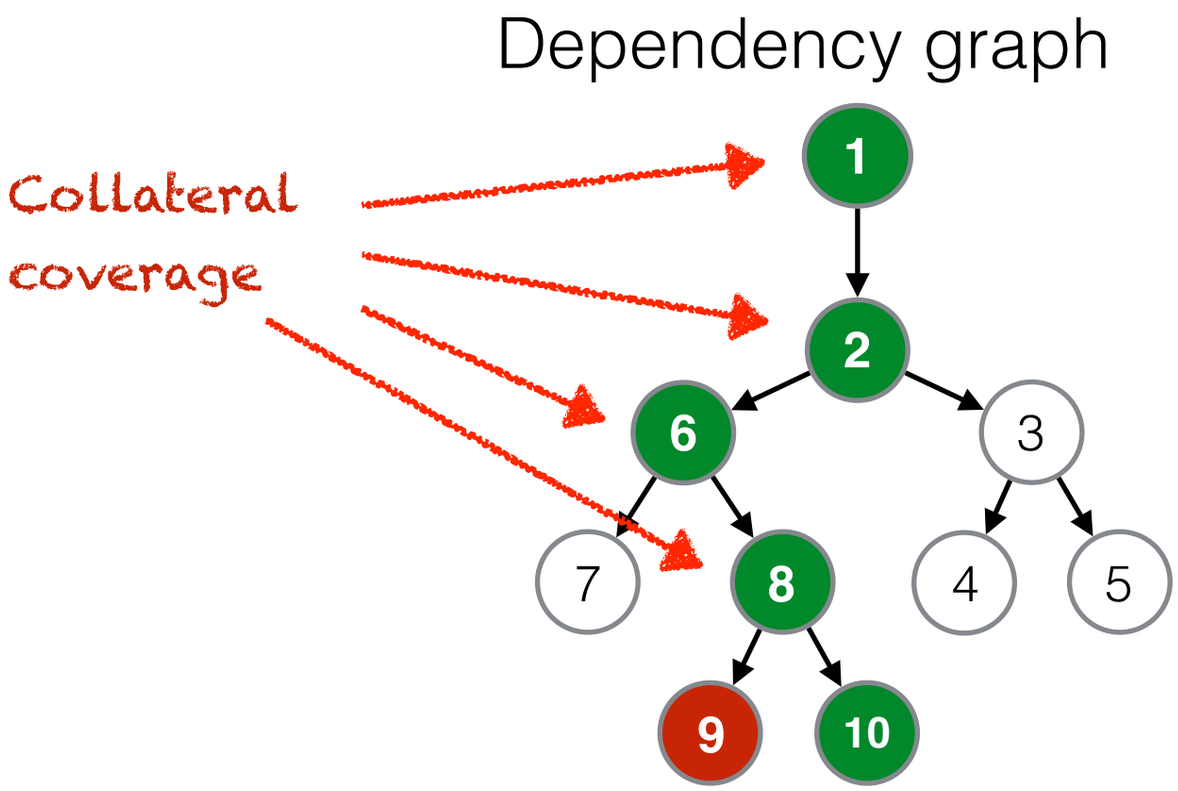
Collateral coverage

```
class Triangle {

 void computeTriangleType() {
1. if (isTriangle()){
2.   if (side1 == side2) {
3.        if (side2 == side3)
4.            type = "EQUILATERAL";
          else
5.            type = "ISOSCELES";
      } else {
6.        if (side1 == side3) {
7.            type = "ISOSCELES";
          } else {
8.            if (side2 == side3)
9.                type = “ISOSCELES”;
              else
10.               checkRightAngle();
              }
          }
      }// if isTriangle()
  }}
```



Collateral coverage

When a target is covered accidentally in test case generation cycle, it is removed from the list of yet uncovered targets and the corresponding (covering) test case is stored to form the final test suite.



Collateral coverage: if a test case involved in collateral coverage are not detected and stored, it can be shown that random testing performs asymptotically better than search-based testing

Arcuri et al. **Formal analysis of the effectiveness and predictability of random testing. ISSTA**, *pp. 219-230, 2010.*

Running Example

```
class Triangle {
    private double side1, side2, side3;
    private String type = “NOT_A_TRIANGLE”;

    public Triangle (double a, double b, double c){…}
    private void checkRightAngle() {…}
    public void computeTriangleType() {…}
    private boolean isTriangle() {…}
}
```

TU Delft

Running example

```
class Triangle {
   int a, b, c; //sides
   int type = NOT_A_TRIANGLE;

   Triangle (int a, int b, int c){…}
   void checkRightAngle() {…}
   void computeTriangleType() {…}
   boolean isTriangle() {…}
   public static void main (String args[]) {…}
}
```

Class under test

Initial Test Cases

```
@Test
public void test(){
   Triangle c = new Triangle(8.0, 2.1
   c.computeTriangleType();
   assertTrue
}
```

```
@Test
public void test(){
   Triangle c = new Triangle(1
   c.computeTriangleType();
   assertTrue(c.getTriangleTyp
}
```

```
@Test
public void test(){
   Triangle c = new Triangle(1.0, 2.1, 4.2);
   c.computeTriangleType();
   assertTrue(c.getTriangleType, …);
}
```

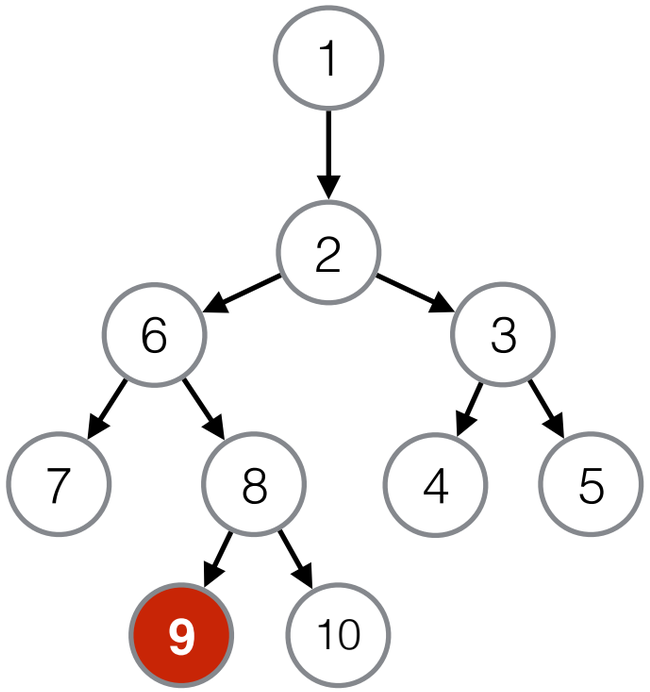
```
@Test
public void test(){
   Triangle c = new Triangle(-2.0, 12.0, 0);
   c.computeTriangleType();
   assertTrue(c.getTriangleType, …);
}
```

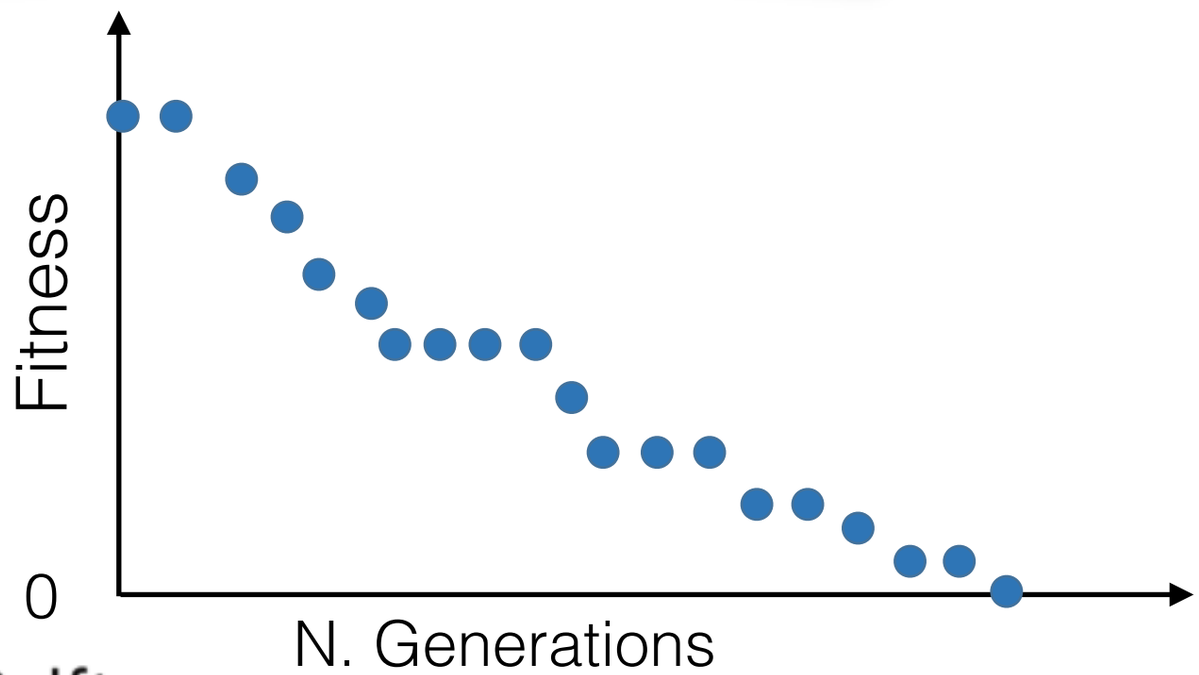
TUDelft

Running example

```
class Triangle {

 void computeTriangleType() {
   if (isTriangle()){
     if (side1 == side2) {
          if (side2 == side3)
              type = "EQUILATERAL";
          else
              type = "ISOSCELES";
      } else {
          if (side1 == side3) {
              type = "ISOSCELES";
           } else {
             if (side2 == side3)
                 type = “ISOSCELES”;
             else
                checkRightAngle();
                }
          }
      }// if isTriangle()
 }}
```

target



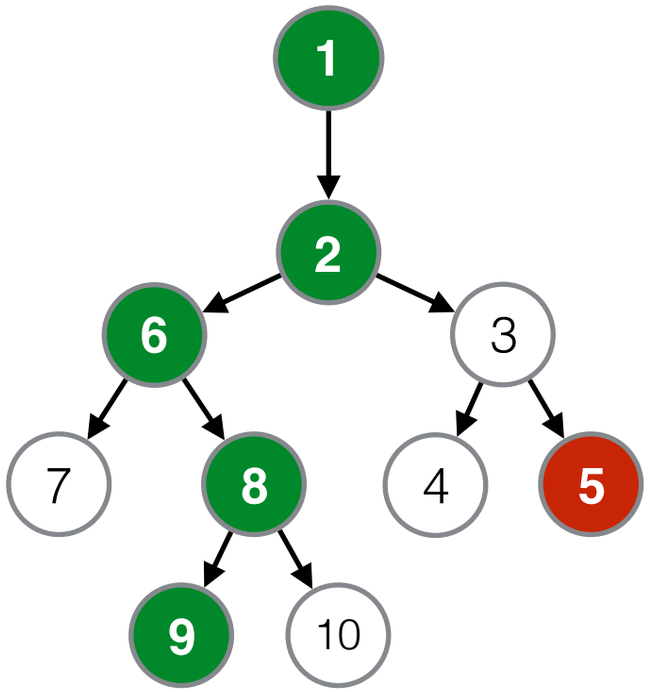


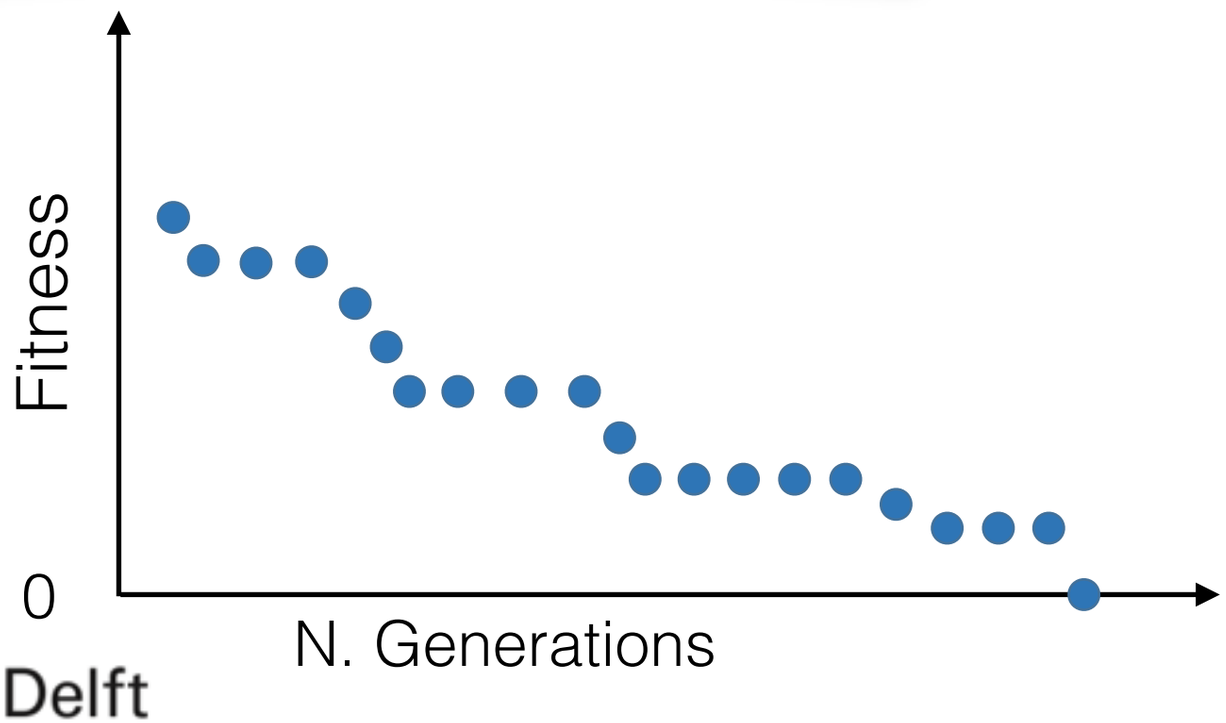
TUDelft

Running example

```
class Triangle {

 void computeTriangleType() {
   if (isTriangle()){
     if (side1 == side2) {
          if (side2 == side3)
              type = "EQUILATERAL";
          else
              type = "ISOSCELES";
      } else {
          if (side1 == side3) {
              type = "ISOSCELES";
           } else {
             if (side2 == side3)
                 type = “ISOSCELES”;
             else
                checkRightAngle();
                }
          }
      }// if isTriangle()
}}
```

target



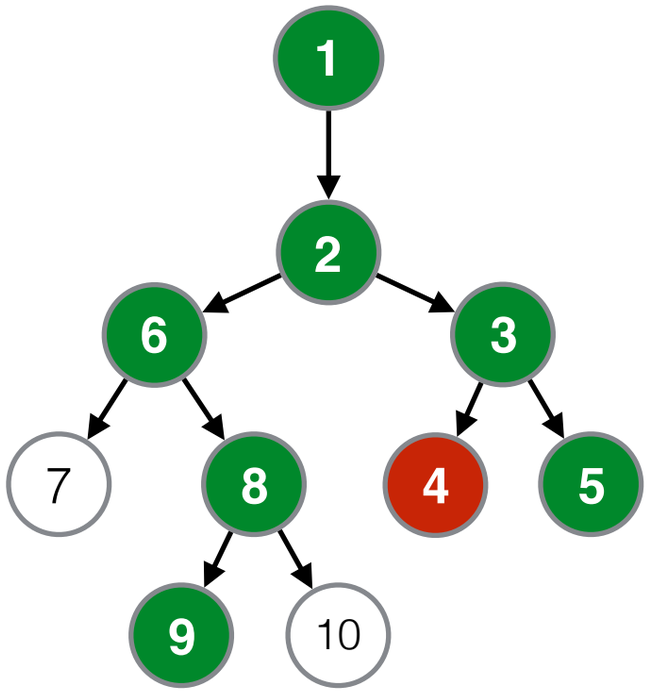


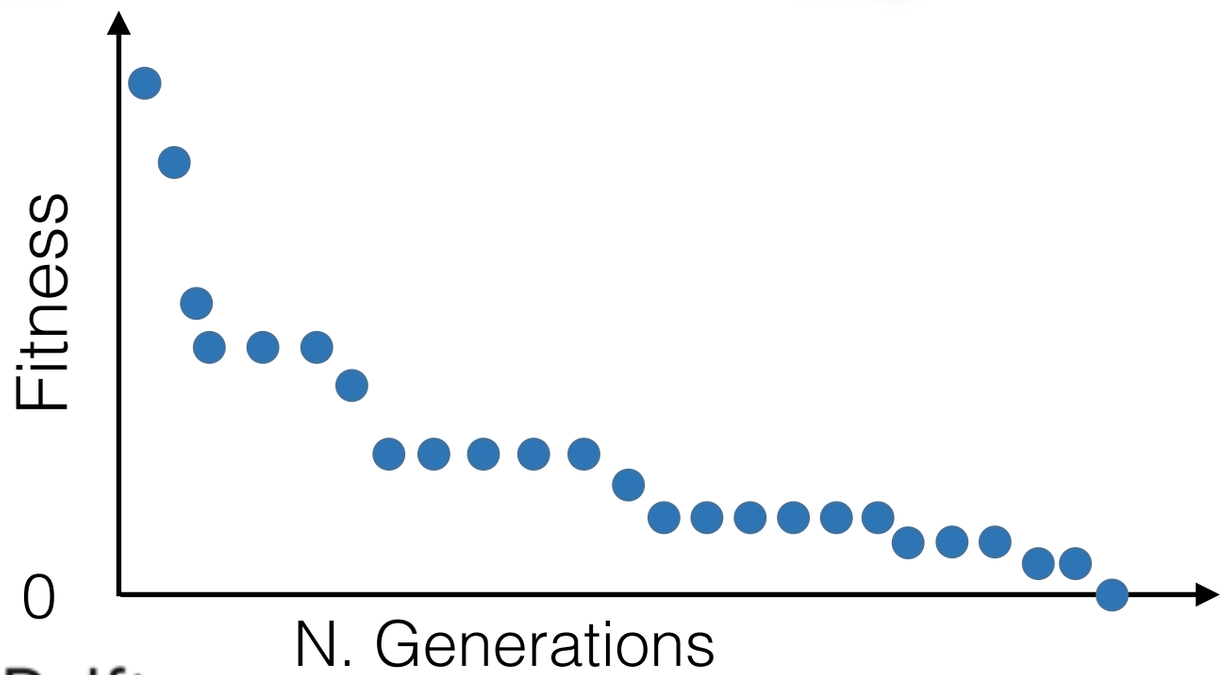
TC1 = (4,3,3)

Running example

```
class Triangle {

 void computeTriangleType() {
   if (isTriangle()){
     if (side1 == side2) {
          if (side2 == side3)
              type = "EQUILATERAL";
          else
              type = "ISOSCELES";
      } else {
          if (side1 == side3) {
              type = "ISOSCELES";
           } else {
             if (side2 == side3)
                 type = “ISOSCELES”;
             else
                checkRightAngle();
                }
          }
      }// if isTriangle()
}}
```

target



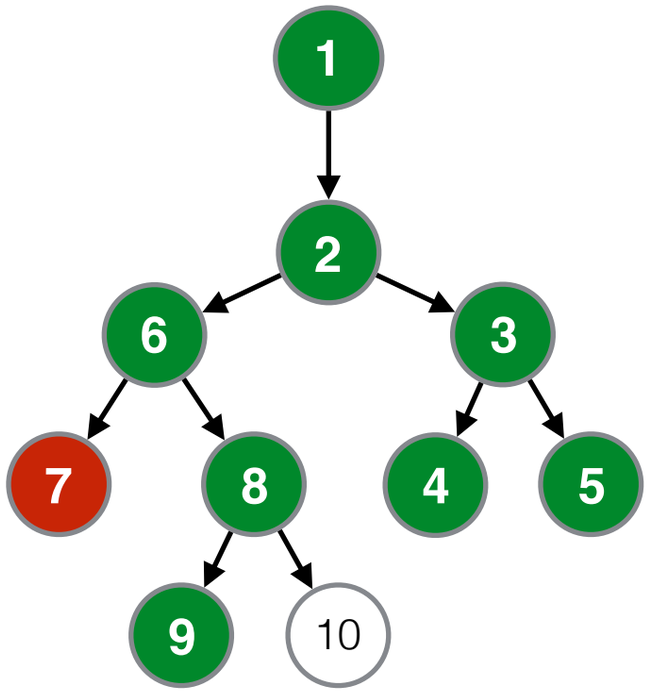


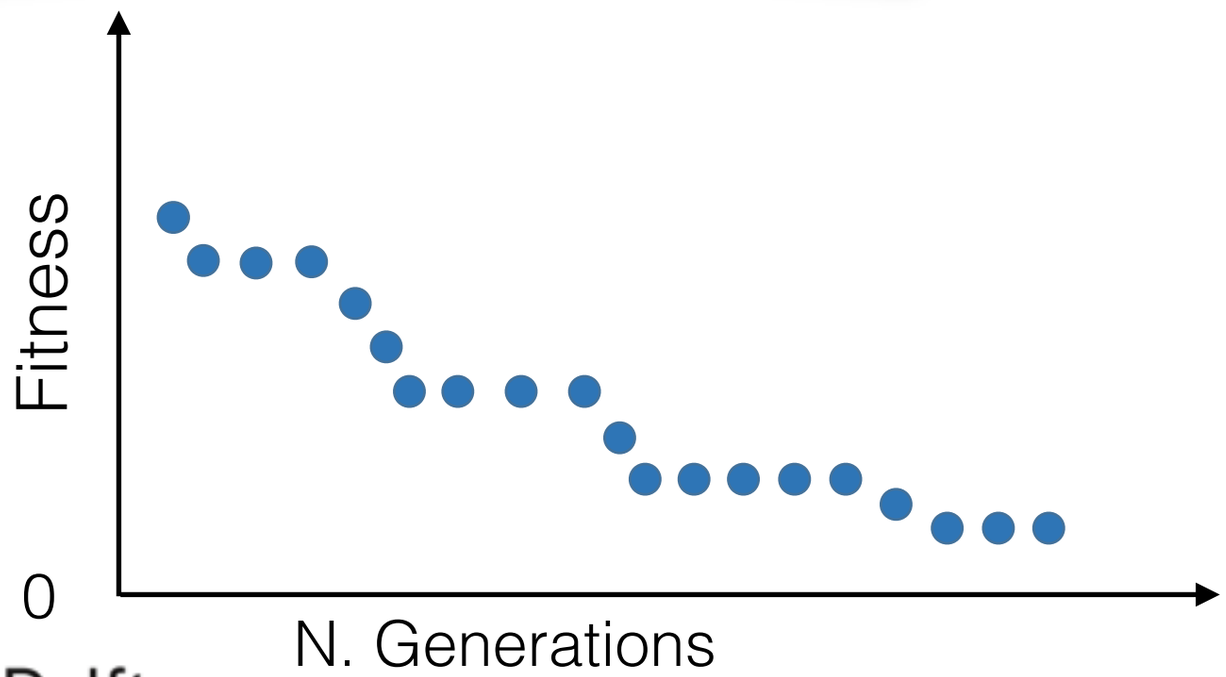
TC1 = (4,3,3)
TC2 = (2,2,4)

Running example

```
class Triangle {

 void computeTriangleType() {
   if (isTriangle()){
     if (side1 == side2) {
          if (side2 == side3)
              type = "EQUILATERAL";
          else
              type = "ISOSCELES";
      } else {
          if (side1 == side3) {
              type = "ISOSCELES";
           } else {
             if (side2 == side3)
                 type = “ISOSCELES”;
             else
                checkRightAngle();
                }
          }
      }// if isTriangle()
}}
```

target

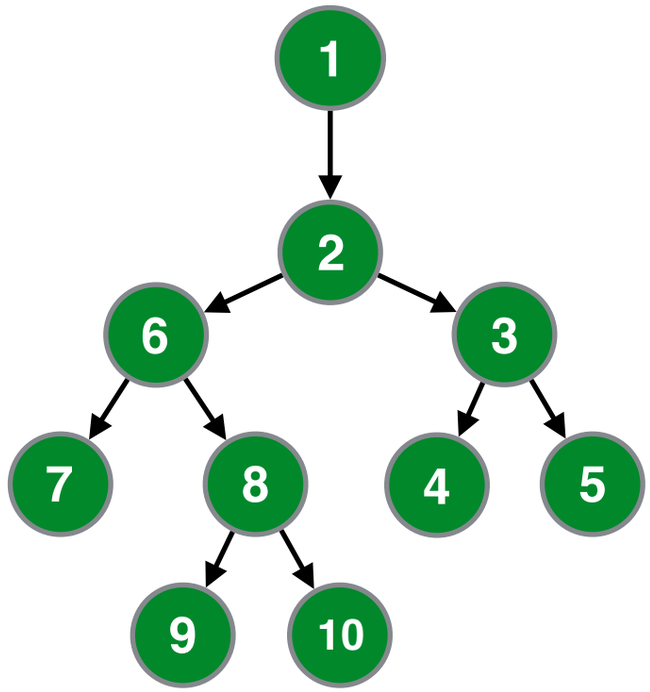





```
TC1 = (4,3,3)
TC2 = (2,2,4)
TC2 = (5,5,5)
```

Running example

```
class Triangle {

 void computeTriangleType() {
   if (isTriangle()){
     if (side1 == side2) {
          if (side2 == side3)
              type = "EQUILATERAL";
          else
              type = "ISOSCELES";
      } else {
          if (side1 == side3) {
              type = "ISOSCELES";
           } else {
             if (side2 == side3)
                 type = “ISOSCELES”;
             else
                checkRightAngle();
                }
         }
     }// if isTriangle()
}}
```



TC1 = (4,3,3)
TC2 = (2,2,4)
TC3 = (5,5,5)
TC4 = (4,3,4)
TC5 = (1,2,3)

The final test suite consists of all chromosome that have been found to cover (even accidentally) one or more yet to cover statements.

Assertions

Assertions are **inserted automatically** as follows:

- At the end of execution get methods can be used to test the state of the object

- The actual state is then used to fill the assertions

- The number of assertions can be too large. There is a need for filtering (minimise) the assertions (e.g., using mutation analysis)

TUDelft

Assertions

```
class TestLinkedList extends TestCase {
    public void testCase1() {
        LinkedList x0 = new LinkedList();
        …
        Integer x1 = null;
        x0.addFirst(x1);
        Integer x2 = null;
        assertTrue(x0.remove(x2));
        …
    }
}
```

Simple assertion

```
class TestLinkedList extends TestCase {
    public void testCase3() {
        LinkedList x0 = new LinkedList();
        …
        Integer x1 = new Integer(2);
        x0.addLast(x1);
        assertTrue(x0.toString().equals("[2]"));
    }
}
```

Assertion with object

```
class TestLinkedList extends TestCase {
    public void testCase2() {
        LinkedList x0 = new LinkedList();
        …
        try {
            x0.listIterator(78);
            fail();
        } catch(IndexOutOfBoundException e) {}
}
```

Assertions with exception

TU Delft

Limitations

1) Some coverage targets may be infeasible

2) Some coverage targets may be very difficult to achieve

3) Since a limited search budget is available for test case generation:

- Infeasible targets may use the search budget without reaching any target
- Difficult target may use most of search budget, leaving lots of easier coverage target uncovered
- The order in which targets are considered affects the final results

How to solve multiple targets at once? How?

Further Techniques

G. Fraser, A.Arcuri
"*Whole Test Suite Generation*"
IEEE Transaction on Software Engineering, 2013

Whole Test Suite Generation

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

Abstract—Not all bugs lead to program crashes, and not always is there a formal specification to check the correctness of a software test's outcome. A common scenario in software testing is therefore that test data is generated, and a tester manually adds test oracles. As this is a difficult task, it is important to produce small yet representative test sets, and this representativeness is typically measured using code coverage. There is, however, a fundamental problem with the common approach of targeting one coverage goal at a time: Coverage goals are not independent, not equally difficult, and sometimes infeasible – the result of test generation is therefore dependent on the order of coverage goals and how many of them are feasible. To overcome this problem, we propose a novel paradigm in which whole test suites are evolved with the aim of covering all coverage goals at the same time, while keeping the total size as small as possible. This approach has several advantages, as for example its effectiveness is not affected by the number of infeasible targets in the code. We have implemented this novel approach in the EvoSuite tool, and compared it to the common approach of addressing one goal at a time. Evaluated on open source libraries and an industrial case study for a total of 1,741 classes, we show that EvoSuite achieved up to 188 times the branch coverage of a traditional approach targeting single branches, with up to 62% smaller test suites.

Index Terms—Search based software engineering, length, branch coverage, genetic algorithm, infeasible goal, collateral coverage

1 Introduction

It is widely recognized that software testing is an essential component of any successful software development process. A software test consists of an input that executes the program and a definition of the expected outcome. Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. Yet, the problem of the expected outcome persists, and has become known as the *oracle problem*. Sometimes, essential properties of programs are formally specified, or have to hold universally such that no explicit oracles need to be defined (e.g., programs should normally not crash). However, in the general case one cannot assume the availability of an automated oracle. This means that, if we produce test inputs, then a human tester needs to specify the oracle in terms of the expected outcome. To make this feasible, test generation needs to aim not only at high code coverage, but also at small test suites that make oracle generation as easy as possible.

■ *Gordon Fraser is with Saarland University – Computer Science, Saarbrücken, Germany, email: fraser@cs.uni-saarland.de. Andrea Arcuri is with the Certus Software V&V Center at Simula Research Laboratory, P.O. Box 134, Lysaker, Norway, email: arcuri@simula.no*

```
1  public class Stack {
2    int[] values = new int[3];
3    int size = 0;
4
5    void push(int x) {
6      if(size >= values.length)      <- Requires a full stack
7        resize();
8      if(size < values.length)       <- Else branch is infeasible
9        values[size++] = x;
10   }
11
12   int pop() {
13     if(size > 0)<- May imply coverage in push and resize
14       return values[size--];
15     else
16       throw new EmptyStackException();
17   }
18
19   private void resize(){
20     int[] tmp = new int[values.length + 2];
21     for(int i = 0; i < values.length; i++)
22       tmp[i] = values[i];
23     values = tmp;
24   }
25 }
```

Fig. 1. Example stack implementation: Some branches are more difficult to cover than others, some lead to coverage of further branches, and some can be infeasible.

A common approach in the literature is to generate a test case for each coverage goal (e.g., branches in branch coverage), and then to combine them in a single test suite (e.g., see [43]). However, the size of a resulting test suite is difficult to predict, as a test case generated for one goal may implicitly also cover any number of further coverage goals. This is usually called collateral or serendipitous coverage (e.g., [25]). For example, consider the stack implementation in Figure 1: Covering the true branch in Line 11 is necessarily preceded by the true branch in Line 7, and may or may not also be preceded by the true branch in Line 5. In fact, the order in which each goal is select can thus play a major role, as there can be dependencies among goals. Although there have been attempts to exploit collateral coverage to optimize test generation (e.g., [25]), to the best of our knowledge there is no conclusive evaluation in the literature of their effectiveness.

A.Panichella, F. M. Kifetew, P. Tonella
"*Reformulating Branch Coverage as Many-Objective Optimization Problems*"
IIEEE International Conference on Software Testing, Verification, and Validation (ICST), 2015

Reformulating Branch Coverage as a Many-Objective Optimization Problem

Annibale Panichella*, Fitsum Meshesha Kifetew[†‡], Paolo Tonella[‡]
*Delft University of Technology, The Netherlands
[‡]University of Trento, Trento, Italy
[†]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 many-objective branch coverage problem.

Results achieved on 64 Java classes indicate that the proposed many-objective algorithm is significantly more effective and more efficient than the whole suite approach. In particular, effectiveness (coverage) was significantly improved in 66% of the subjects and efficiency (search budget consumed) was improved in 62% of the subjects on which effectiveness remains the same.

Keywords: Evolutionary testing, many-objective optimization, branch coverage.

I. Introduction

Test automation, and input data generation in particular, has received substantial attention from researchers. When instantiated for branch coverage, the problem can be formulated as: find a set of test cases which maximize the number of covered branches in the software under test (SUT). Solutions based on search meta-heuristics (aka, search-based testing) deal with this problem by targeting one branch at a time and typically using genetic algorithms (GA) to generate test cases that get closer and closer to the target branch [1], [2], under the guidance of a fitness function, measuring the distance between each test case trace and the branch. With this approach, a single-objective GA is performed multiple times, changing the target branch each time, until all branches are covered or the total search budget is consumed. In the end, the final test suite is obtained by combining all generated test cases in a single test suite, including those responsible for accidental coverage.

Such a single-target strategy presents several issues [3]. For example, some targets may be more difficult to cover than others, thus, an improper allocation of the testing budget might affect the effectiveness of the search process (e.g., 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 meta-heuristics 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 it is not efficient for some kinds of problems (e.g., problems with non-convex region in the search space) [4]. Further studies have also demonstrated that many-objective approaches can be more efficient than single-objective approaches when solving the same complex problem [5], [6], i.e., a many-objective reformulation of complex problems reduces the probability of being trapped in local optima, also leading to a better convergence rate [5]. This remains true even when the multiple objectives are eventually aggregated into a single objective for the purpose of selecting a specific solution at the end of the (many-objective) search [5].

Stemming from the consideration that many-objective approaches can be more efficient when solving complex problems [5], [6], in this paper we propose to explicitly reformulate the branch coverage criterion as a many-objective optimization problem, where different branches are considered as different objectives to be optimized. In this new formulation, a candidate solution is a test case, while its *fitness* is measured according to all (yet uncovered) branches at the same time, adopting the multi-objective notion of optimality. As noted by Arcuri and Fraser [7], the branch coverage criterion lends itself to a many-objective problem, but it poses scalability problems to traditional many-objective algorithms since a typical class can have hundreds if not thousands of objectives (e.g., branches). However, we observe that branch coverage presents some peculiarities with respect to traditional (numerical) problems and we exploit them to overcome the scalability issues associated with traditional algorithms.

In this paper, we introduce a novel highly-scalable many-objective GA, named MOSA (Many-Objective Sorting Algorithm), that modifies the selection scheme used by existing

TUDelft