Structural Testing

CS453, Shin Yoo

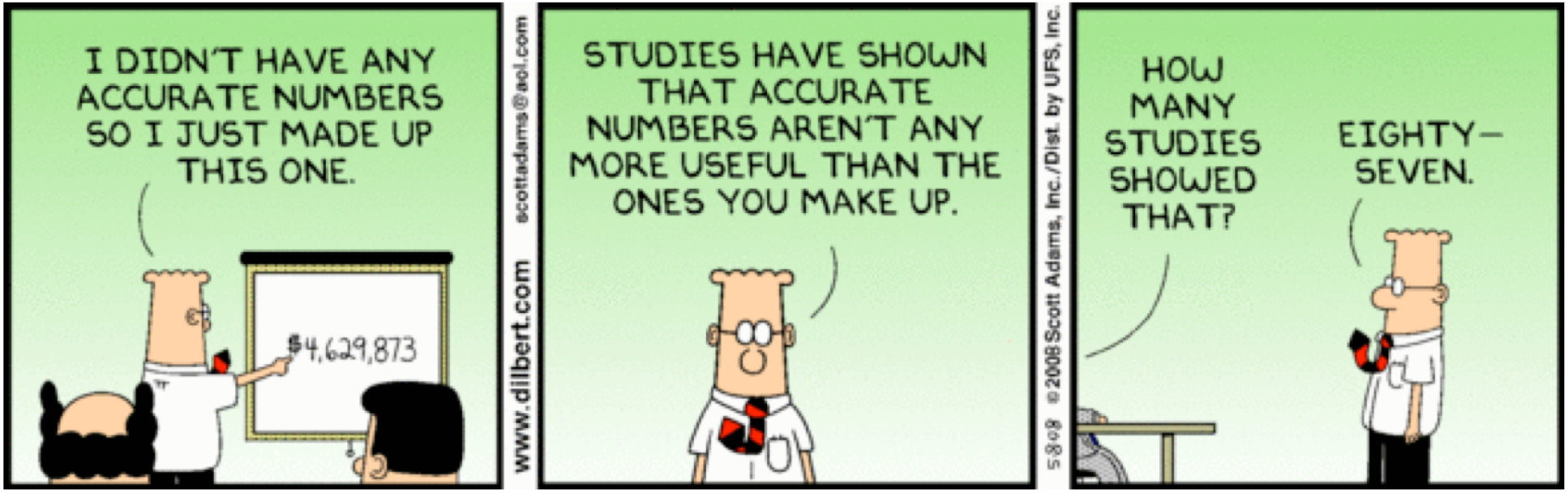
What is structural testing?

- Structural testing measures the quality of testing based on the internal structure of the code. For example,
  - we can ask “have I tested all variable declarations?”
  - but can’t ask “have I tested all functionalities in the requirement documentation?”
- It is most relevant to unit testing, where your view of the entire system is at the code level

Testing Adequacy Criteria

- Allows you to measure the adequacy of testing against the achievement of certain structural criteria
    - For example, statement coverage (100% means you have executed all statements in the program)
- All structural adequacy criteria is necessary but not sufficient to detect faults; no test adequacy criteria except exhaustive testing can guarantee detecting all faults

What does it really mean?



In industry at the time of writing

- Organisations strive to reach less than 100% coverage (if they care at all)
  - Statement and branch coverage are used widely
  - Certain industry, e.g. avionics, legally require 100% coverage
- Over 75% is practically regarded as good enough, but research claims that you need at least over 90% for satisfactory fault detection:
  - Hutchins, Foster, Goradia and Ostrand, ‘Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria’, Proc. 16th Int’l Conference on Software Engineering (ICSE-16), 1994
- Many tools exist to measure coverage of all kinds described here

Test Data Generation

- Very active research area in automatically generating test input to achieve these criteria during the last 10 years; mature enough to get a big break
- The big question: can we generate a test suite that achieves (branch/ statement/All Path) coverage automatically?
- Traditional tools:
  - You. Think and write down.
  - Random: generate random inputs until you cover everything (not very likely in some cases)
  - User session: but they weren't testing really

Cutting Edge Techniques

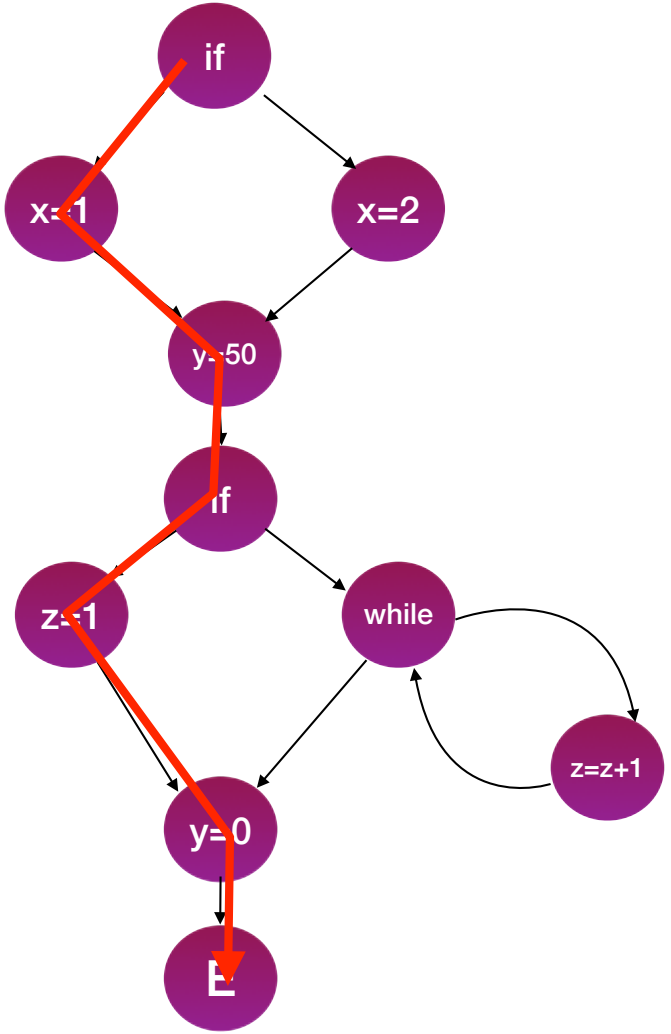
- Search-Based Testing can cover an arbitrary statement/ branch you want - repeat until you reach 100%
- Dynamic Symbolic Execution (aka Concolic Testing) achieves path coverage (the former CS453 was big on this topic: there is a CS492 Special Topic in CS taught by Prof. Moonzoo Kim)
- Both techniques are based on what we call Path Conditions

Path Condition

What is the path condition?

- A collection of predicates that leads the program execution down to a specific path

```
if(y > 13) x=1; else x=2;
y = 50;
if(w == 4) z = 1;
else{
    while(...)
    z = z + 1;
}
y = 0;
```



y > 13 && w == 4

Path Condition

- If you obtain a set of input values that satisfies a given path condition, you cover the corresponding path
- Search-Based Testing converts the path condition into a fitness function and uses meta-heuristic search to find the values
- DSE uses constraint solvers to find the values

Search-Based Testing

- General Idea
  - Convert path conditions into a mathematical fitness function
  - Use meta-heuristic search algorithms to maximise/minimise this function
    - start with one or more random input values
    - essentially, you try slightly different solutions every time and pick the one that is fitter
    - repeat with the fitter solution
  - When the goal is met, you have your test input values

Search-Based Testing

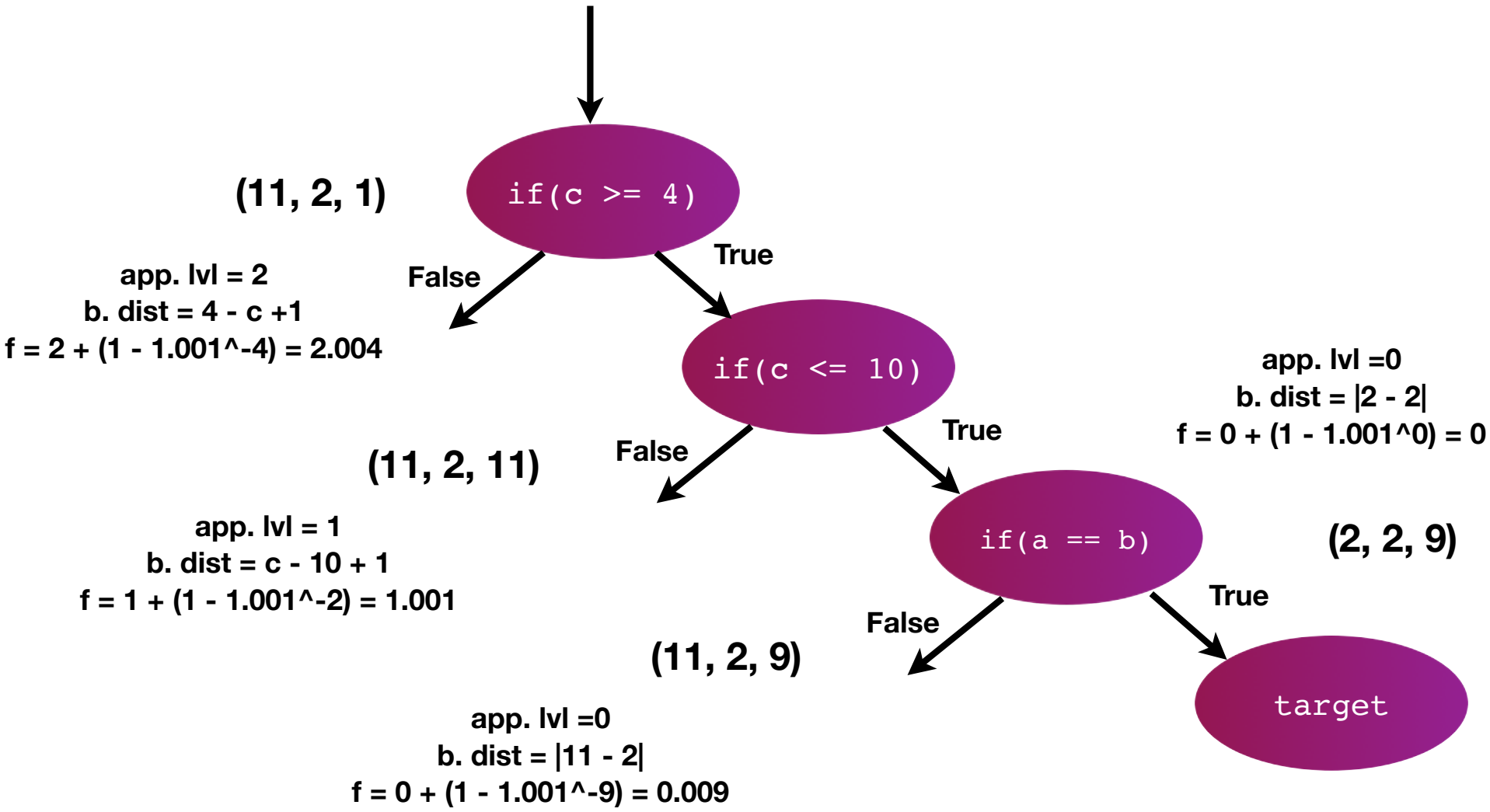
- Fitness function for branch coverage = [approach level] + normalise([branch distance])
- For a target branch and a given path that does not cover the target:
  - Approach level: number of un-penetrated nesting levels surrounding the target
  - Branch distance: how close the input came to satisfying the condition of the last predicate that went wrong

Branch Distance

- If you want to satisfy the predicate `x == y`, you convert this to branch distance of `b = |x - y|` and seek the values of `x` and `y` that minimise `b` to `0`
  - then you will have `x` and `y` that are equal to each other
- If you want to satisfy the predicate `y >= x`, you convert this to branch distance of `b = x - y + K` and seek the values of `x` and `y` that minimise `b` to `0`
  - then you will have `y` that is larger than `x` by `K`
- Normalise `b` to `1 - 1.001^(-b)`

Predicate	f	minimise until..
a > b	b - a + K	f < 0
a >= b	b - a + K	f <= 0
a < b	a - b + K	f < 0
a <= b	a - b + K	f <= 0
a == b	|a - b|	f == 0
a != b	-|a - b|	f < 0

B. Korel, “Automated software test data generation,” IEEE Trans. Softw. Eng., vol. 16, pp. 870–879, August 1990.

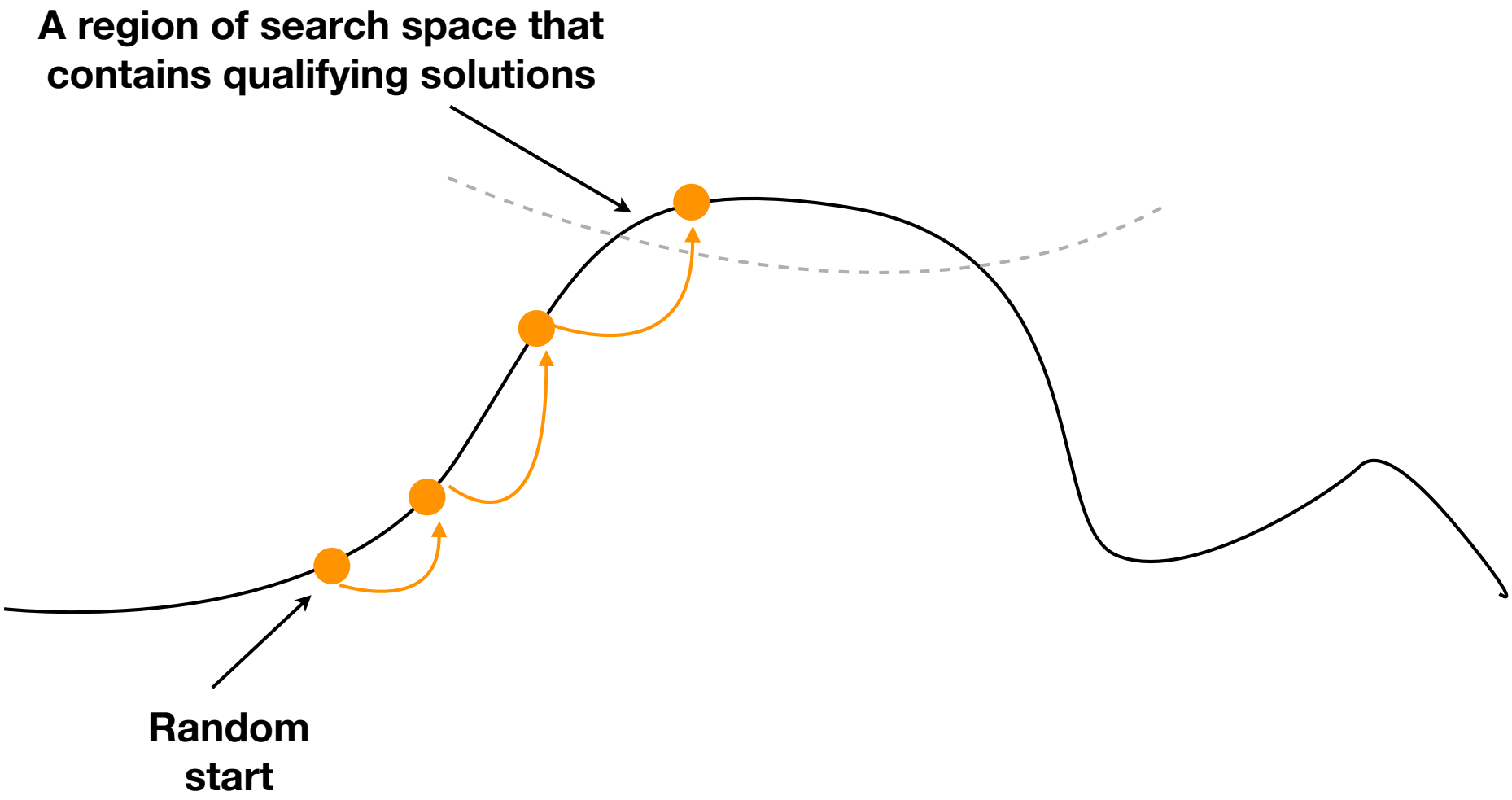


Test input (a, b, c), K = 1

Search-Based Testing

- There are many search algorithms other than hill climbing
- All operate on the same principle
  - Convert the path condition into a fitness function
  - Measure fitness by executing program with candidate input
  - Pick the solution with best fitness

Test Data Generation: Local Search



Alternating Variable Method (AVM)

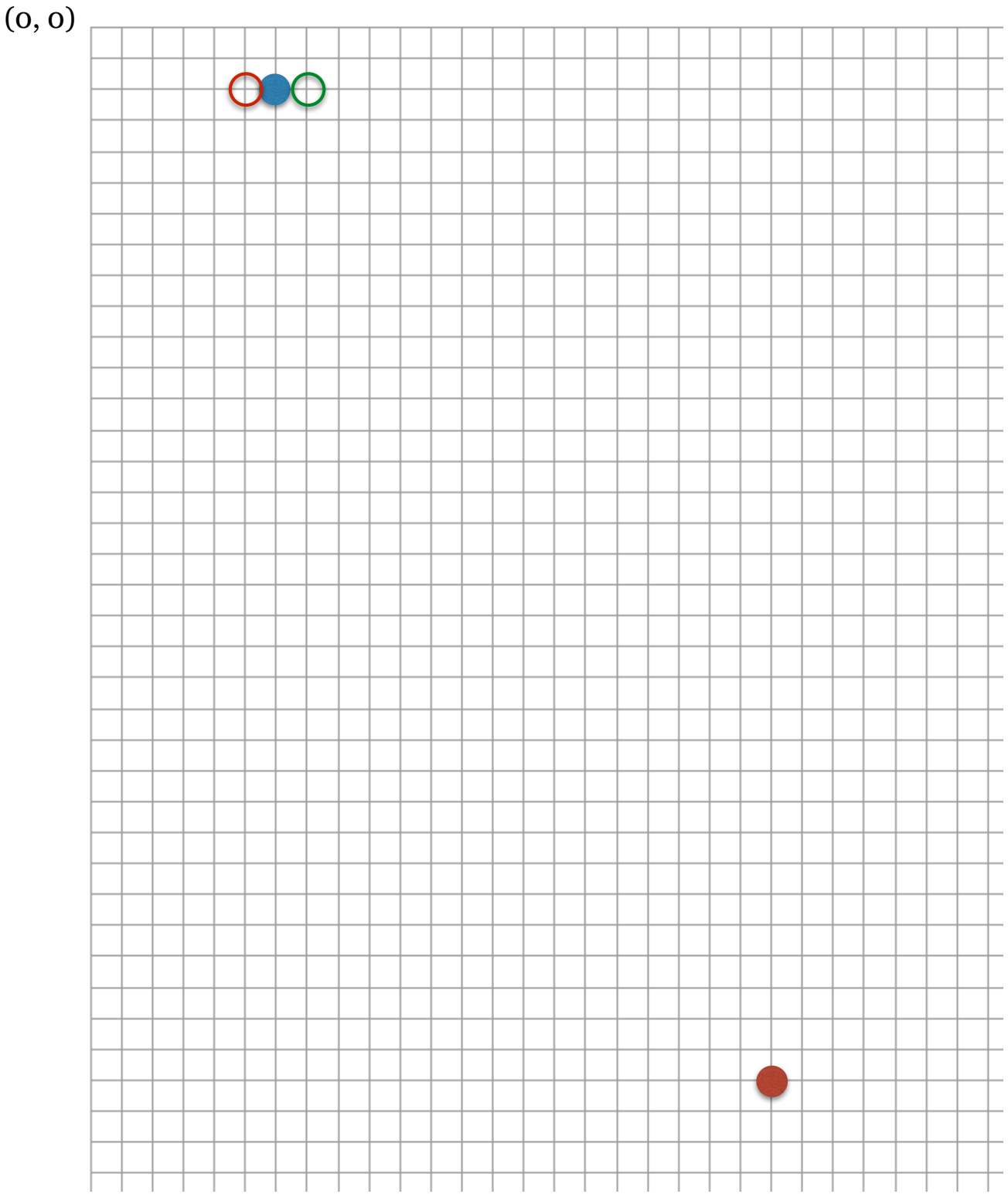
- A type of Pattern Search: searches for an input vector that can maximise/minimise a given objective function
- It has two operation modes: exploratory move, and pattern move.
  - For each variable:
    - Use exploratory move to decide which direction results in fitter solutions
    - Use pattern move to accelerate to that direction

Alternating Variable Method

- Based on the known empirical results, AVM is one of the most effective algorithm for achieving C/C++ structural coverage

  - M. Harman and P. McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007), pages pp. 73–83. ACM Press, July 2007.

  - M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE Transactions on Software Engineering, 36(2):226–247, 2010.



AVM: Exploratory Move

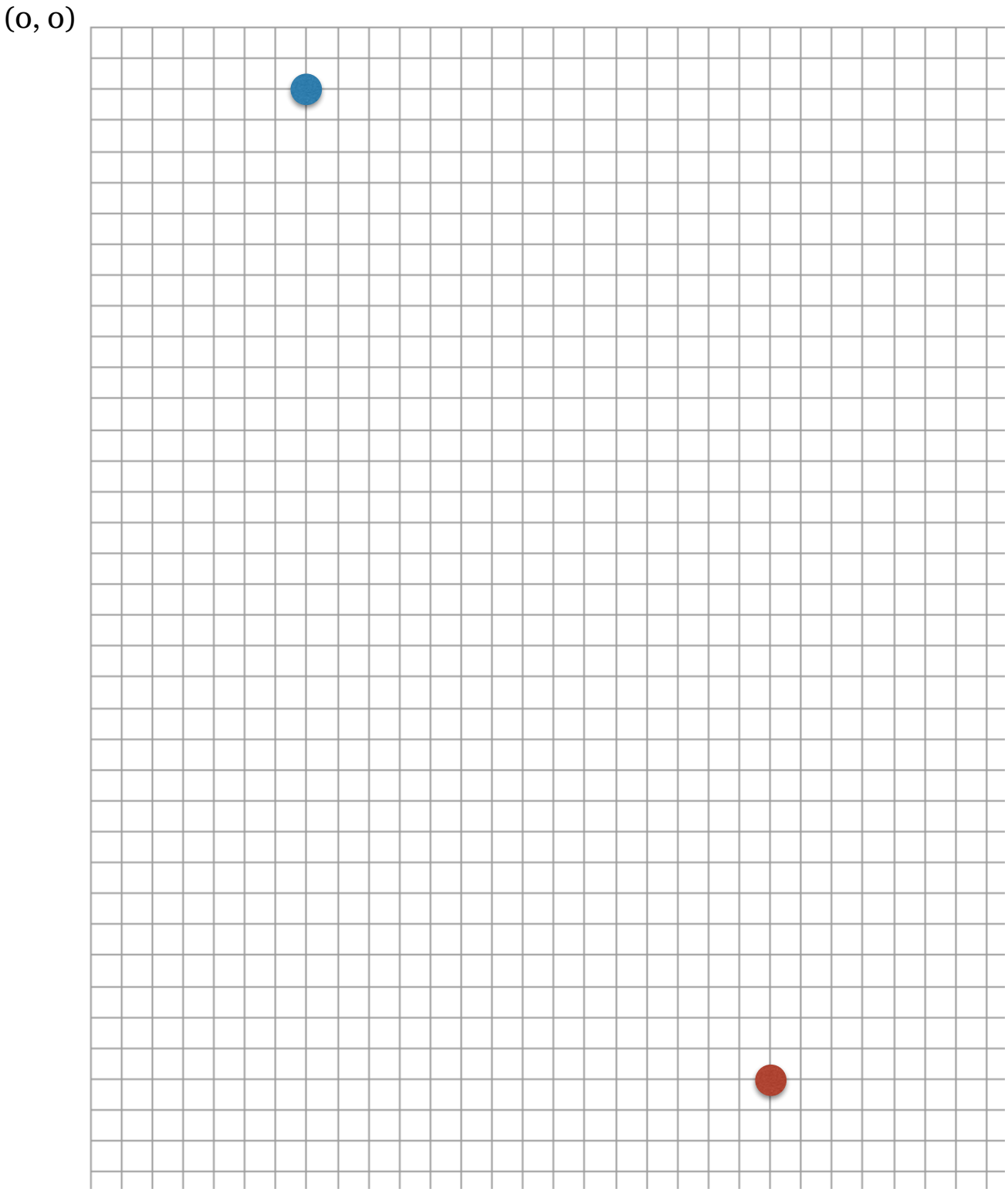
Starting from(6, 2), we want to search for the red dot at (22, 34). We can measure the distance to the goal.

First we try exploratory move for x: make the smallest change, and see which direction results in reduced distance. The initial distance is 35.77.

-1: (5, 2) **Increased** (36.23). X

+1: (7, 2) **Decreased** (35.34) O

Consequently, x needs to be increased at the moment.



AVM: Pattern Move

Now that we decided to increase x, try doubling the difference as long as the distance continues to decrease. At the beginning of the pattern move, x is equal to 7.

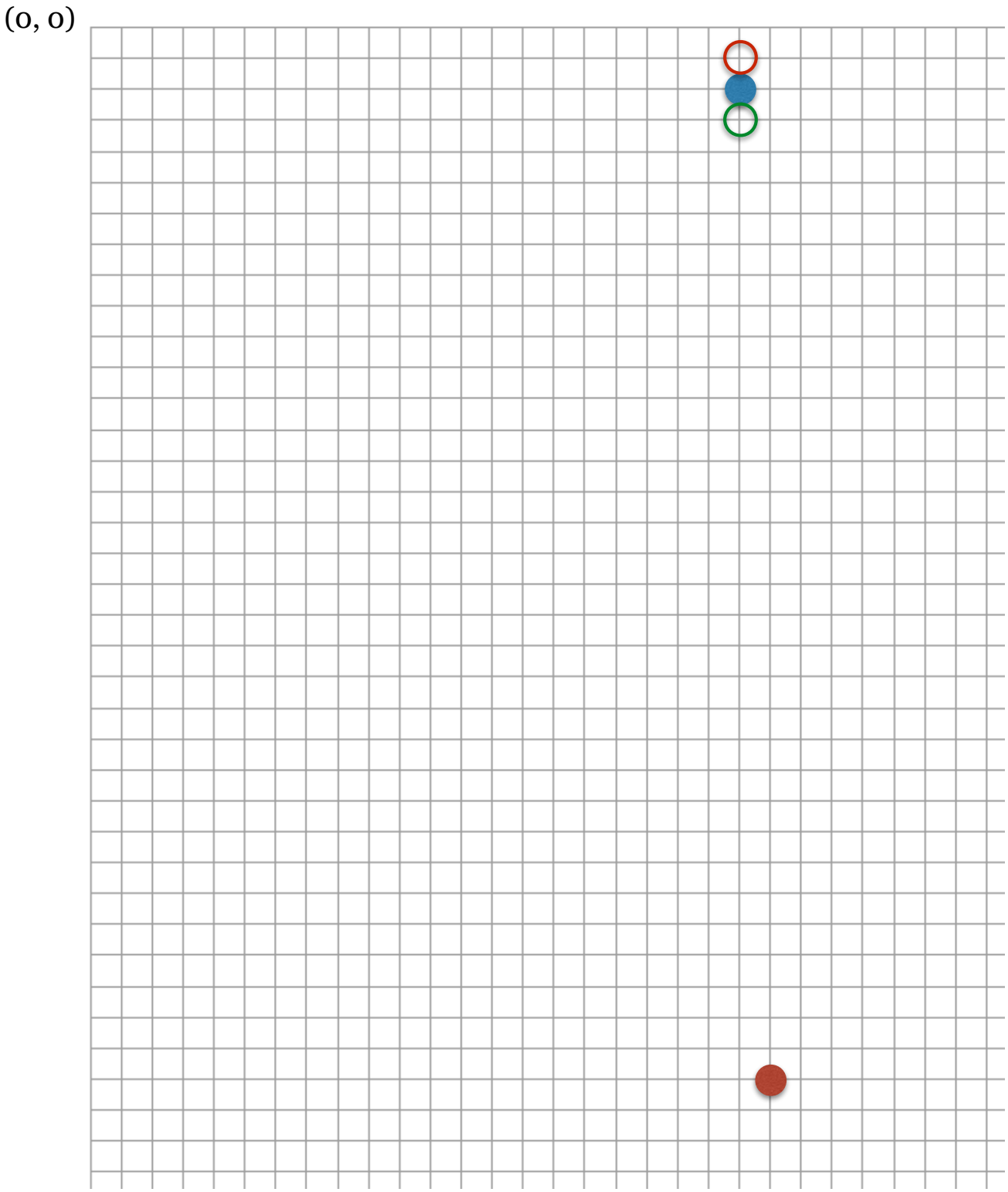
x = 9 (Δx=2): **decrease** (34.53)

x = 13 (Δx=4): **decrease** (33.24)

x = 21 (Δx=8): **decrease** (32.01)

x = 37?(Δx=16): **increase** (35.34)

With increment of 16, the distance starts to grow: this is called overshooting. In this case, we cancel the last pattern move, and start the exploratory move for the next variable, y.



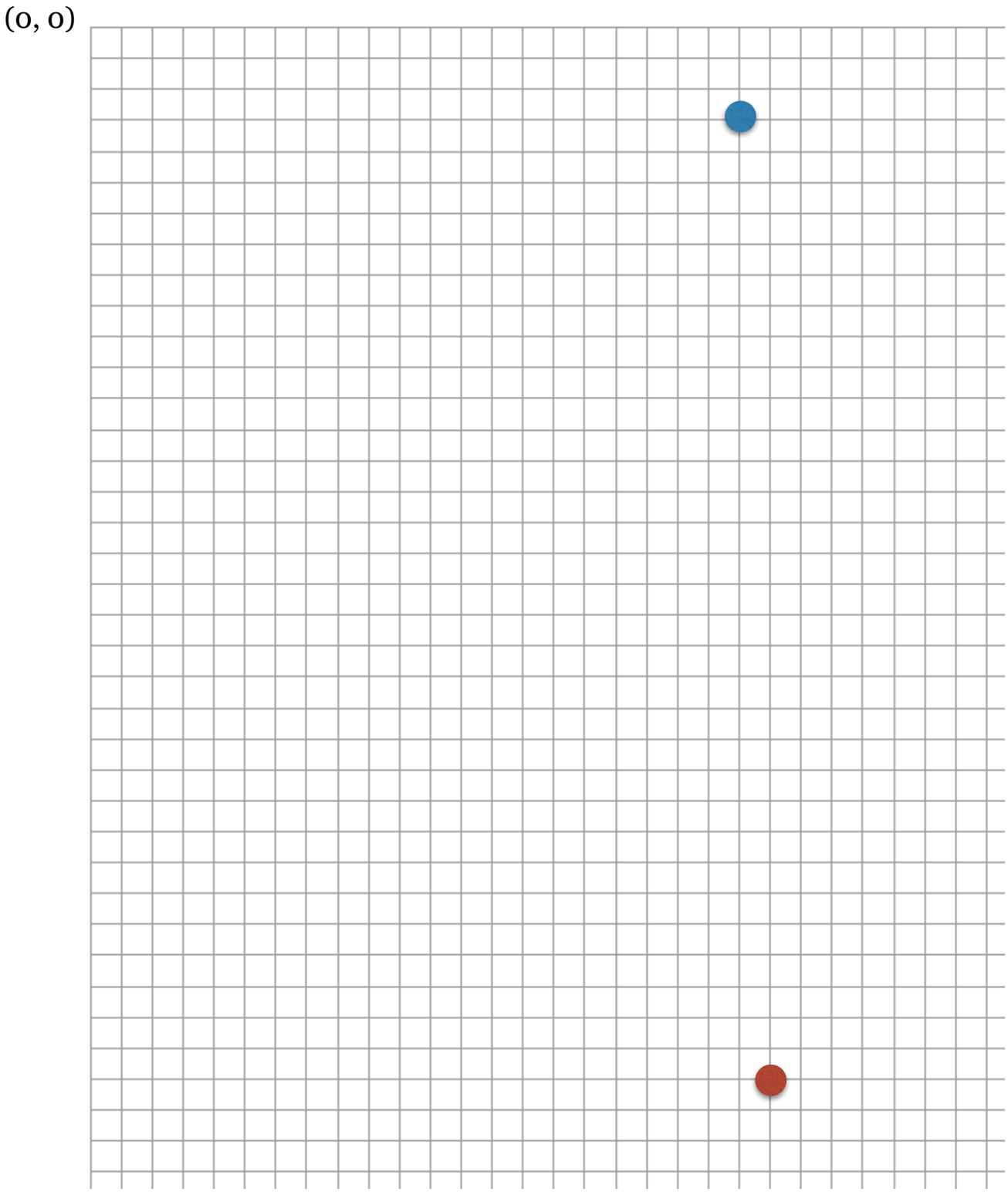
AVM: Exploratory Move

We now change y by 1 and decide the direction. The distance from the last location, (21, 2), is 32.01.

-1: (21, 1) **increase** (33.01). X

+1: (21, 3) **decrease** (31.01) O

So y needs to be increased.



AVM: Pattern Move

We increase the variable y with pattern moves now. Initially y is 3.

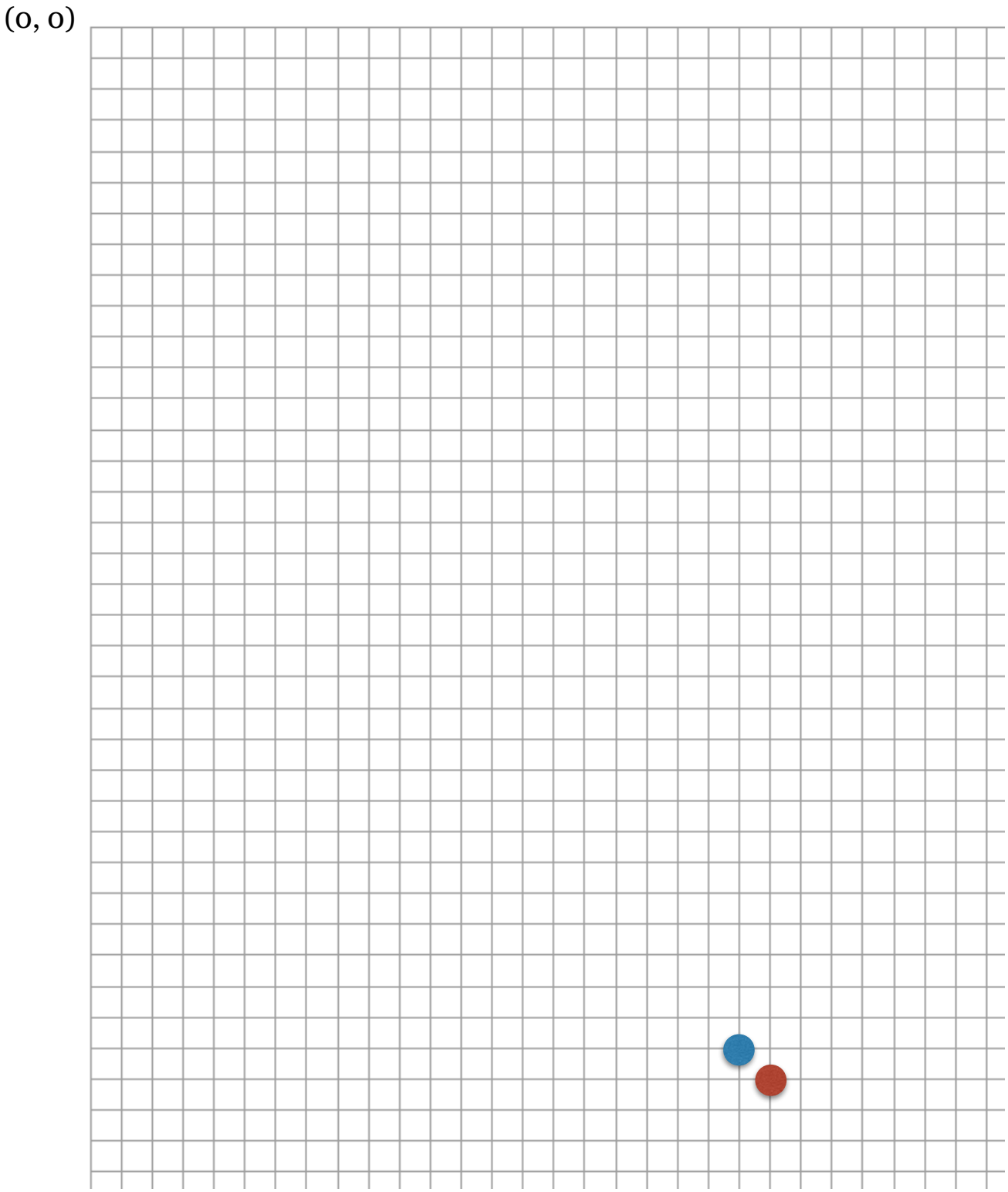
y = 5 (Δy=2): **decrease** (29.01)

y = 9 (Δy=4): **decrease** (25.01)

y = 17 (Δy=8): **decrease** (17.02)

y = 33(Δy=16): **decrease** (1.41)

y = 65(Δy=32): **Overshooting!**



AVM: Exploratory Move

After overshooting of y, we start the exploratory move for x. We decide to increase, but as soon as we try +2, it overshoots. After cancellation of this, we have the correct x.

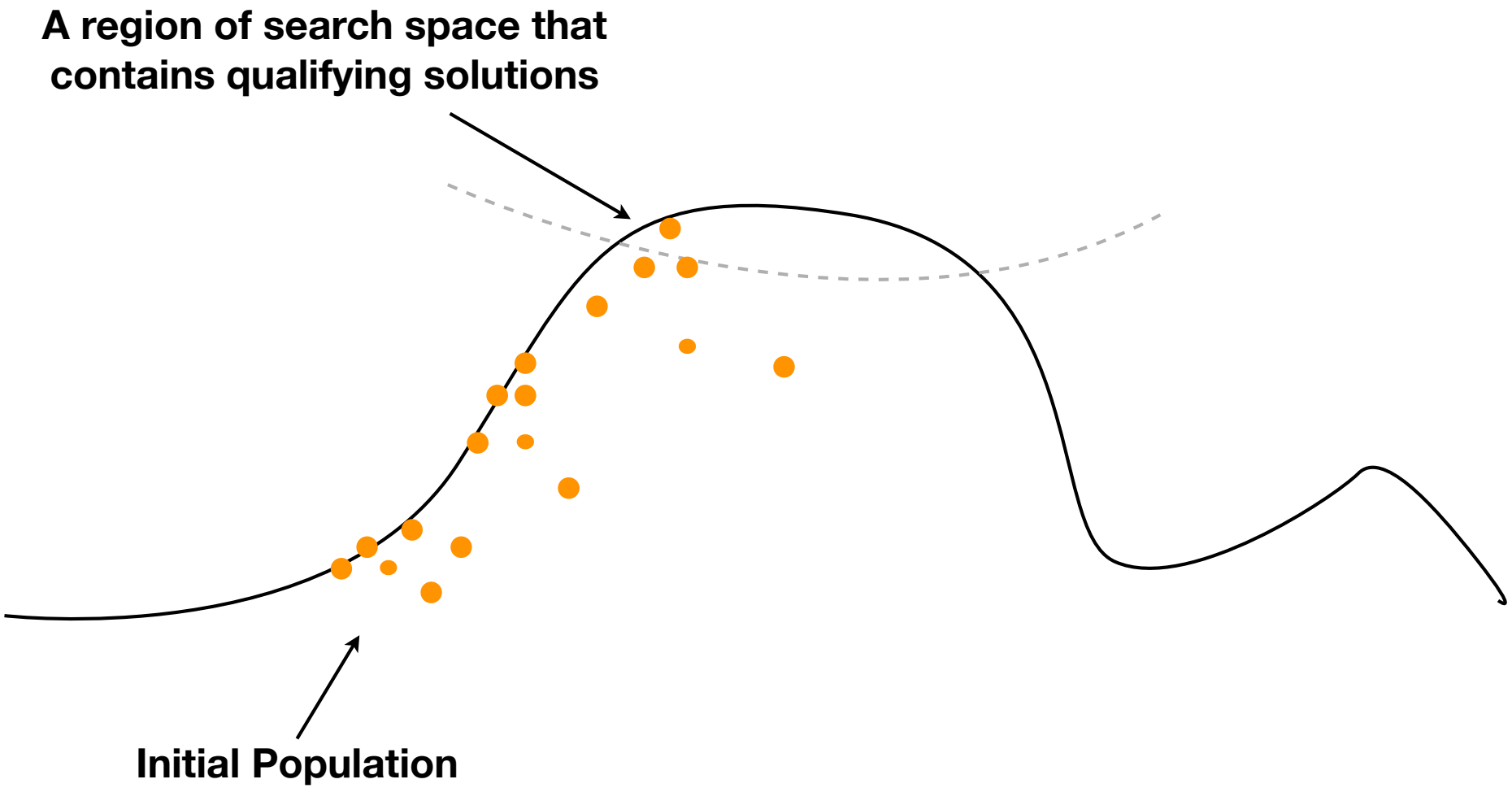
After one more exploratory move for y, we reach the goal.

Alternating Variable Method

- For a reference implementation and basic applications, see: http://avmframework.org

- P. McMinn and G. M. Kapfhammer. AVMf: An open-source framework and implementation of the alternating variable method. In International Symposium on Search-Based Software Engineering (SSBSE 2016), volume 9962 of Lecture Notes in Computer Science, pages 259–266. Springer, 2016.

Test Data Generation : Evolutionary Testing



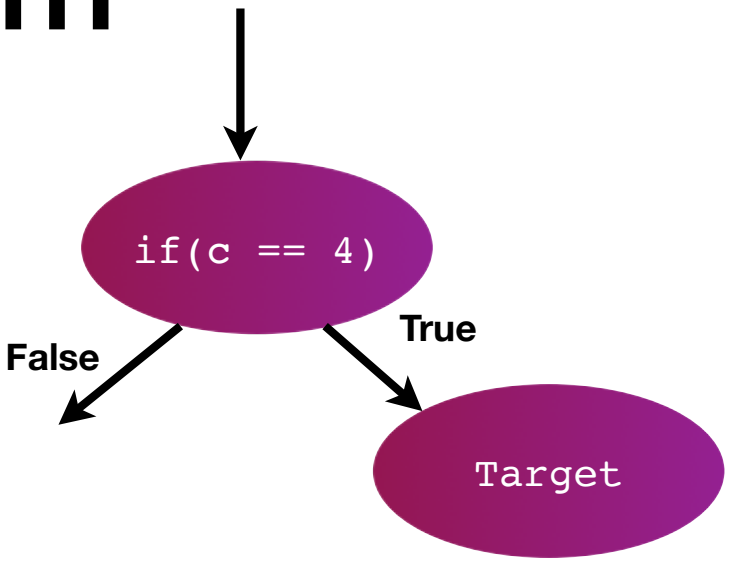
Global Search: Genetic Algorithm

- GAs borrow the theory of Darwinian evolution to search for solutions
- The underlying intuition is that, if two solutions (*parents*) have two distinct and good partial solutions (*genes*), it is possible to get an even better solutions (*offsprings*) by combining parts of them (*offsprings take genes from parents*)

```
//Outline of Genetic Algorithm
P = random population
Evaluate(P)
Repeat until termination:
    parents = select_from(P)
    offsprings = crossover(parents)
    offsprings = mutate(offsprings)
    Evaluate(newGen)
    P = select(P U newGen)
```

An Example of Search Algorithm

- Hill Climbing
  - start with random value
  - calculate fitness
  - check out neighbours
  - calculate fitness
  - if there is a fitter neighbour, move
  - repeat until succeed



c = 7: b. dist = 3, norm. = 1 - 1.001^-3 = 0.0029

neighbours of 7: 6 and 8

c = 6: b. dist = 2, norm. = 1 - 1.001^-2 = 0.0019

c = 8: b. dist = 4, norm. = 1 - 1.001^-4 = 0.0039

so we move to 6 and consider 5 and 7

...

Normalisation

- “[with obj.] Mathematics multiply (a series, function or item of data) by a factor that makes the norm or some associated quantity such as an integral equal to a desired value (usually 1).”

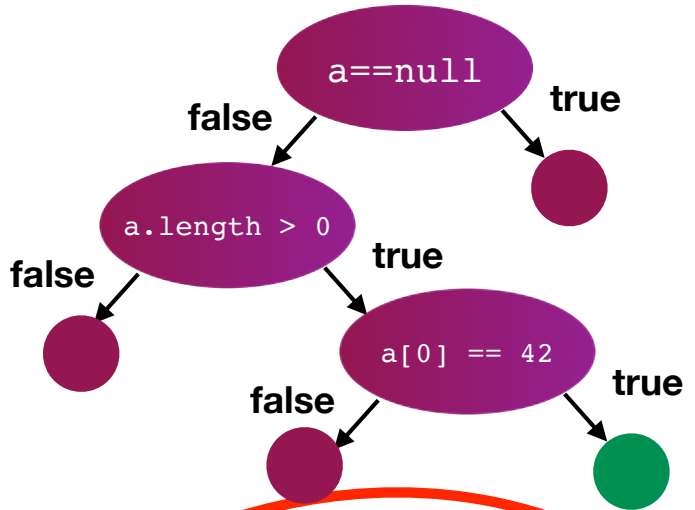
- We normalise the branch distance because…?

  - Approach level is counting, whereas branch distance can be arbitrarily large: we want to consider approach level before branch distance, so branch distance should be in [0, 1).

Dynamic Symbolic Execution

- Think of it as how you execute a code in your head, only using a program
- Start with a basic(random) input, execute program and collect path conditions from the executed path
- Negate the last clause of the path condition
- Solve the resulting constraint using a solver, execute, repeat
- Constraints Solver: given constraints, e.g. x > 5 && y != 3 && z == 18, the solver will give you (x, y, z) = (9, 0, 18)

```
void testme(int[] a)
{
  if(a == null) return;
  if(a.length > 0)
  {
    if(a[0] == 42)
      throw new Exception("bug");
  }
}
```



Solve

Execute

Constraints to Solve	Data	Observed Path Condition
	null	a==null
a!=null	{}	a!=null && !(a.length > 0)
a!=null && a.length > 0	{0}	a!=null && a.length > 0 && a[0] != 42
a!=null && a.length > 0 && a[0] == 42	{42}	No more path!

Negate last condition and choose another path

Dynamic Symbolic Execution

- Microsoft initially developed a tool for .NET framework called Pex.
  - You *could* play with it at http://pex4fun.com but the site no longer works :(
  - It has been incorporated into Visual Studio
- Other famous implementations include KLEE, CUTE/ jCUTE, CREST, and Java Path Finder, but none are one-button-away ready.

Pros and Cons

- SBST is also applicable to non-structural criteria (e.g. worst case execution time analysis), but the concept of “distance” can be difficult (same as in adaptive random testing); it can also take quite a long time (especially evolutionary algorithms)

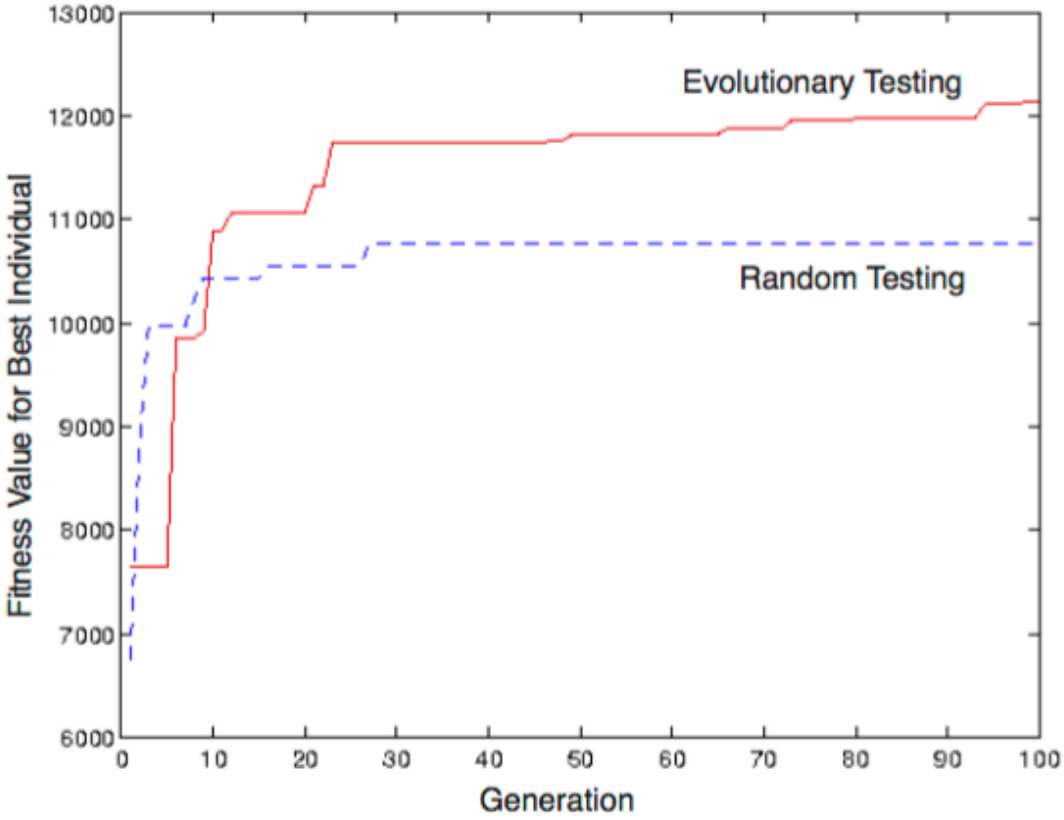
- DSE is usually very fast and effective; however, everything depends on the power of constraint solver

  - They are much stronger than they used to be

  - There are still exceptions where constraint solvers struggle, e.g. anything with floating point number

Testing Non-functional Requirements

- Daimler Chrysler: testing Worst Case Execution Time of an airbag controller
- SBST produced much better results compared to random testing or static analysis



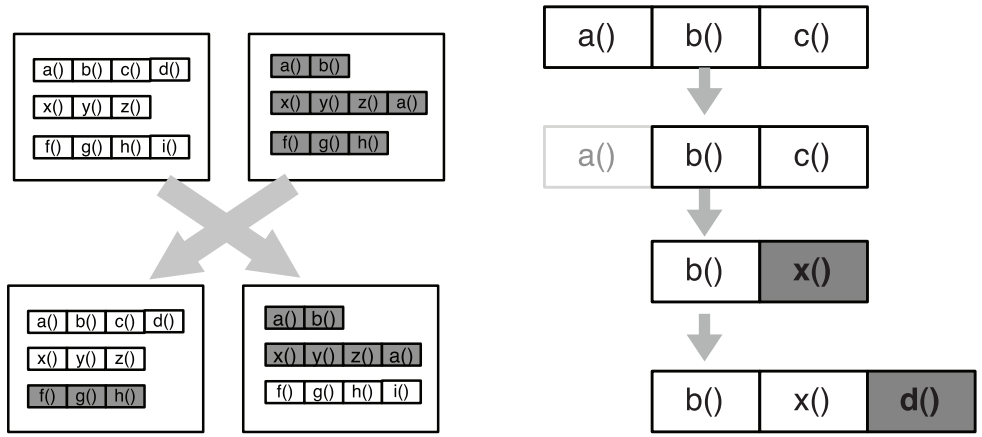
J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Systems, 15(3):275 – 298, 1998.

Testing OO Programs

- We have double layers of problems:
  - Which values (and object instances) to use for method arguments
  - Which sequence to call methods

Whole Test Suite Generation

- Use evolutionary algorithms to evolve the entire test suite, rather than single test input
- Fitness is essentially the sum of all branches: we do not care about approach levels.
  - A set of method invocation sequences that collectively cover the most branches is eventually evolved and selected.



(a) Test Suite Crossover (b) Test Case Mutation

Fig. 3. Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level; mutation is applied to test cases and test suites.

Demo.