Structural Testing

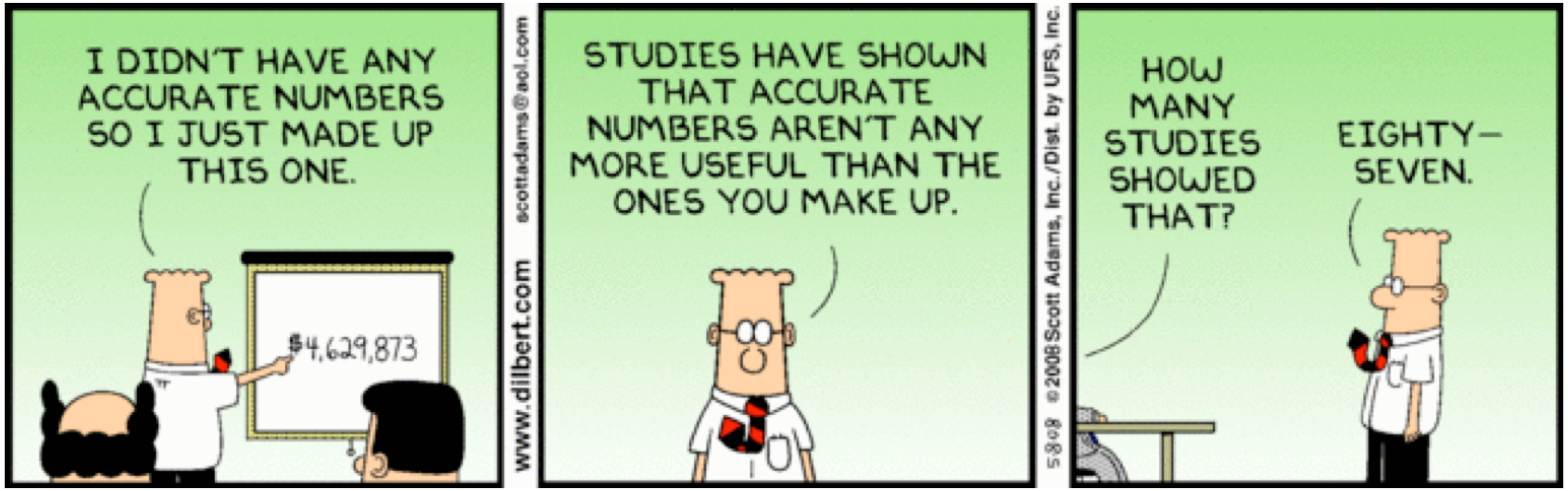
CS454 AI-based Software Engineering: Tutorial for SBST

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

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

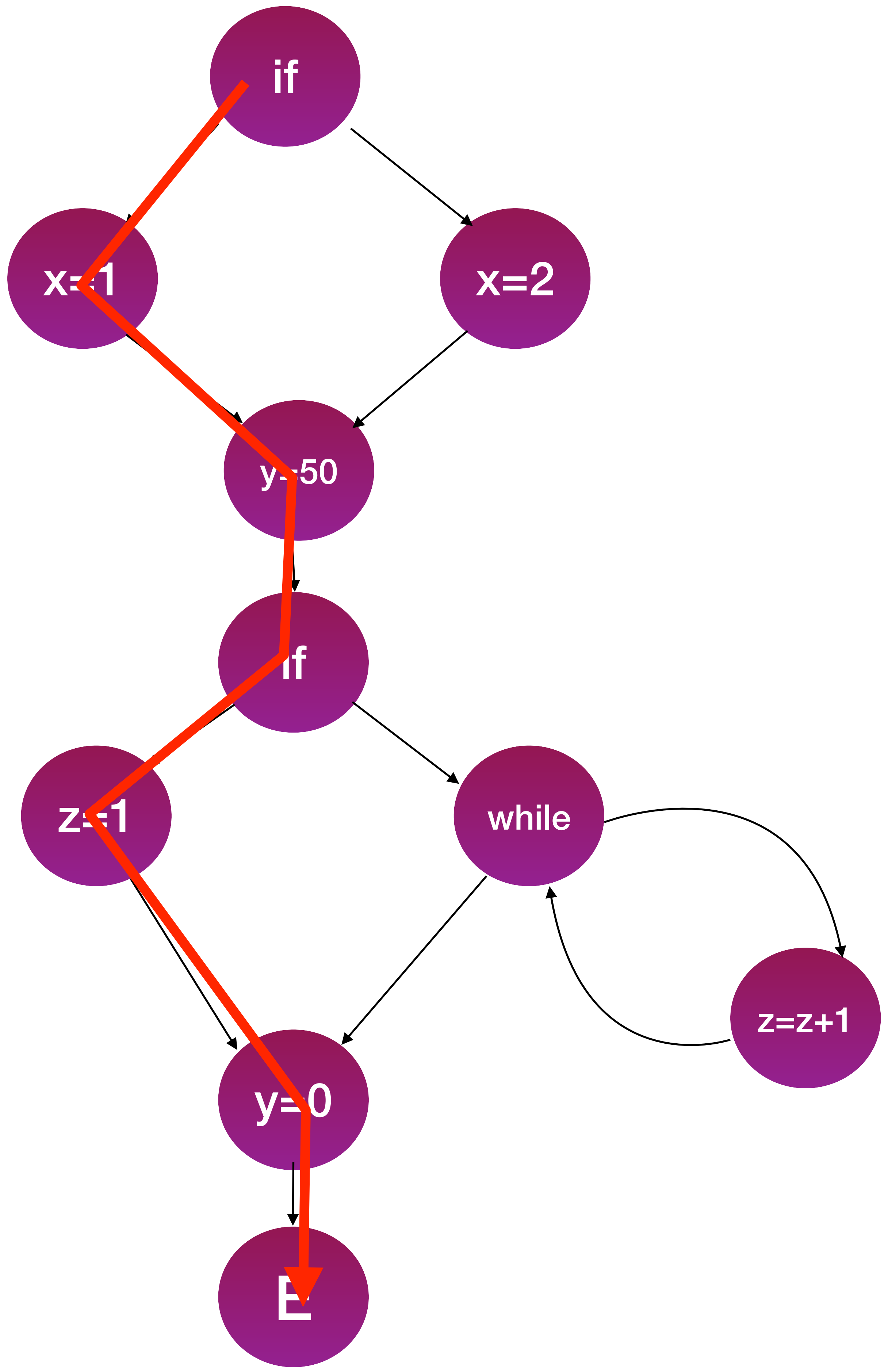
- Search-Based Testing can cover an arbitrary statement/branch you want - repeat until you reach 100%
- Dynamic Symbolic Execution (aka Concolic Testing) achieves path coverage (the former CS453 was big on this topic: there is a CS492 Special Topic in CS taught by Prof. Moonzoo Kim)
- Both techniques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

Path Condition

- 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;
```

What is the path condition?



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

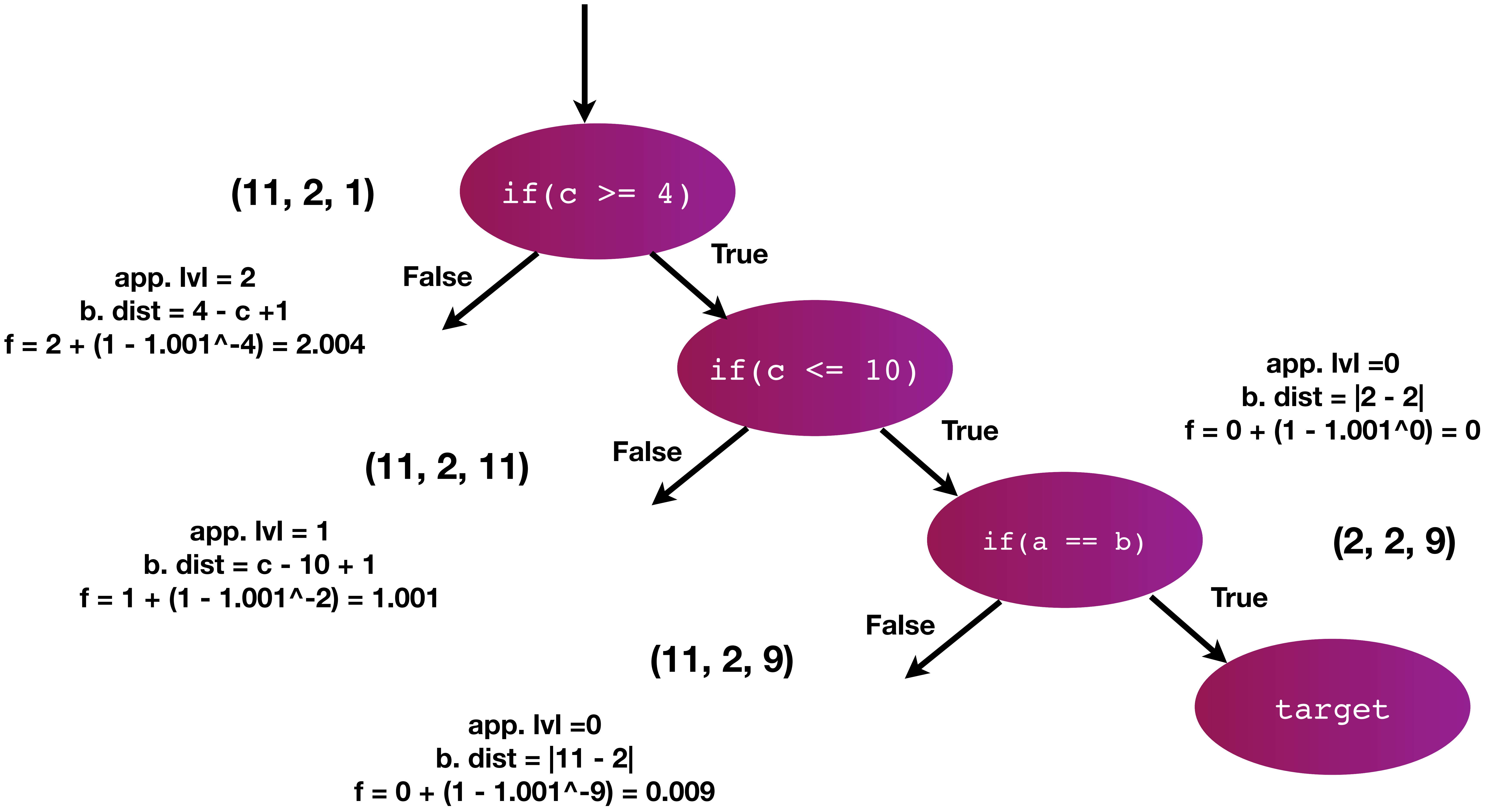
- If you want to satisfy the predicate `x == y`, you convert this to branch distance of `b = |x - y|` and seek the values of `x` and `y` that minimise `b` to `0`
    - then you will have `x` and `y` that are equal to each other
- If you want to satisfy the predicate `y >= x`, you convert this to branch 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)`

Branch Distance

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.

Fitness Function



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