Non-conventional Testing Techniques

A mini-lecture series

CSE498 Collaborative Design (W) - Secure and Efficient C++ Software Development

02/09/2026

Kira Chan

https://cse.msu.edu/~chanken1/

First a note on last class's question

- A student asked if there are some properties that should be true by default, and lead to failure if the program breaks

- "A good test case is one that fails"
  - This really means one that detects INCORRECT BEHAVIOURS
  - Checking if 0 != 1 is not a good test case
- **Purpose of (unit) testing is to identify inconsistencies or incorrect behaviour in code, *with respect to* functional requirements (specifications, documentations, etc.)**
  - It is not to pass the test cases (side effect)
  - Identify the symptoms of the bugs or incorrect behavior

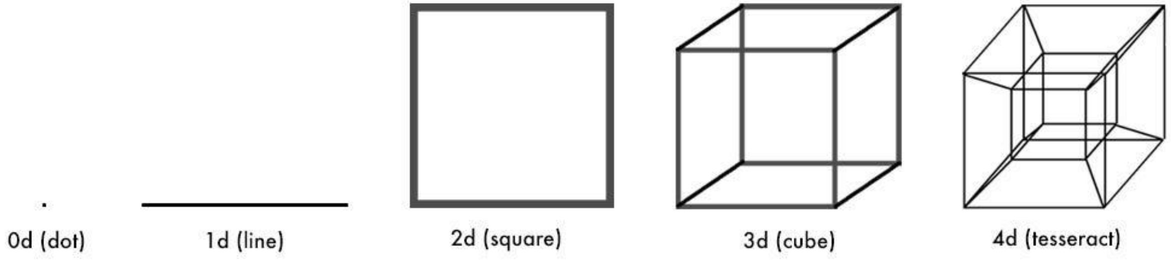
Functional vs Non-functional

- The purpose is to identify if you have build the right software
  - Focus on the **functional requirements** (what the system should do)
  - Testable with respect to some “ground truth”
    - Ideal program vs your current version
    - Test cases identify the difference between the two

- What to not test here
  - **Non-functional requirements** (how the system will do it)
    - Software should complete within 3 minutes
    - Is the system out of memory (and thus we cannot write)
    - The system should be easy to use
    - Can be often subjective (measurable, but the line can be drawn anywhere)

Summary from last class

- We discussed how we should think and develop unit test cases
- Walked through some examples and identified common ways to "miss" test cases
- We have discussed that it is impossible to test every possible input
- Testing is easier with low number of parameters
  - 1 input parameter, can sequentially test
  - 2 parameters? R^2
  - 3 parameters? R^3
  - …



What are some potential issues?

- Testing bias
  - You have some domain knowledge, and you are using them to test your program
  - You also have an expectation of what kinds of inputs are legal (vs what the program will actually accept)
- Time consumption
- Sometimes can hide complex bugs (looks right most of the time)
- Unexpected or Invalid [out of range] inputs (how did we get there?)
  - Save a number
  - File gets corrupted
  - Load the number (now out of your expected input range)
- …

Is there anyway to check for test case completeness?

- Coverage-based testing
  - How much of your code have passed through the test cases
- Input-space coverage
  - How much of the possible input space have we covered
- Output space coverage
  - How much of your possible output space have you explored

A software tester walks into a bar

- Runs into a bar.
- Crawls into a bar.
- Dances into a bar.
- Flies into a bar.
- Jumps into a bar.
- And orders:
- a beer.
- 2 beers.
- 0 beers.
- 99999999 beers.
- a lizard in a beer glass.
- -1 beer.
- "qwertyuiop" beers.
- **Testing complete** ☑

A real customer walks into the bar

- Asks where the bathroom is.
- The bar goes up in flames.

Testing is hard

- Humans test with bias
- When you test your code, you test it *gently*

- *Can we automatically generate some test cases?*

Towards Automated Testing

- Fuzz testing (fuzzing)
- Evolutionary Search-based testing
- Fault injection
- Mutation-based testing
- Random testing
- Symbolic and Concolic testing
- Metamorphic testing
  - Leverage relation between input and outputs

Fuzzing / Fuzz testing

Black-box model of a function

Input arguments:
int x, int y

Function foo(...)

Output:
int z

Some magic occurs here
(we will build the magic)

Functions are instead input transformers

Fault discovery

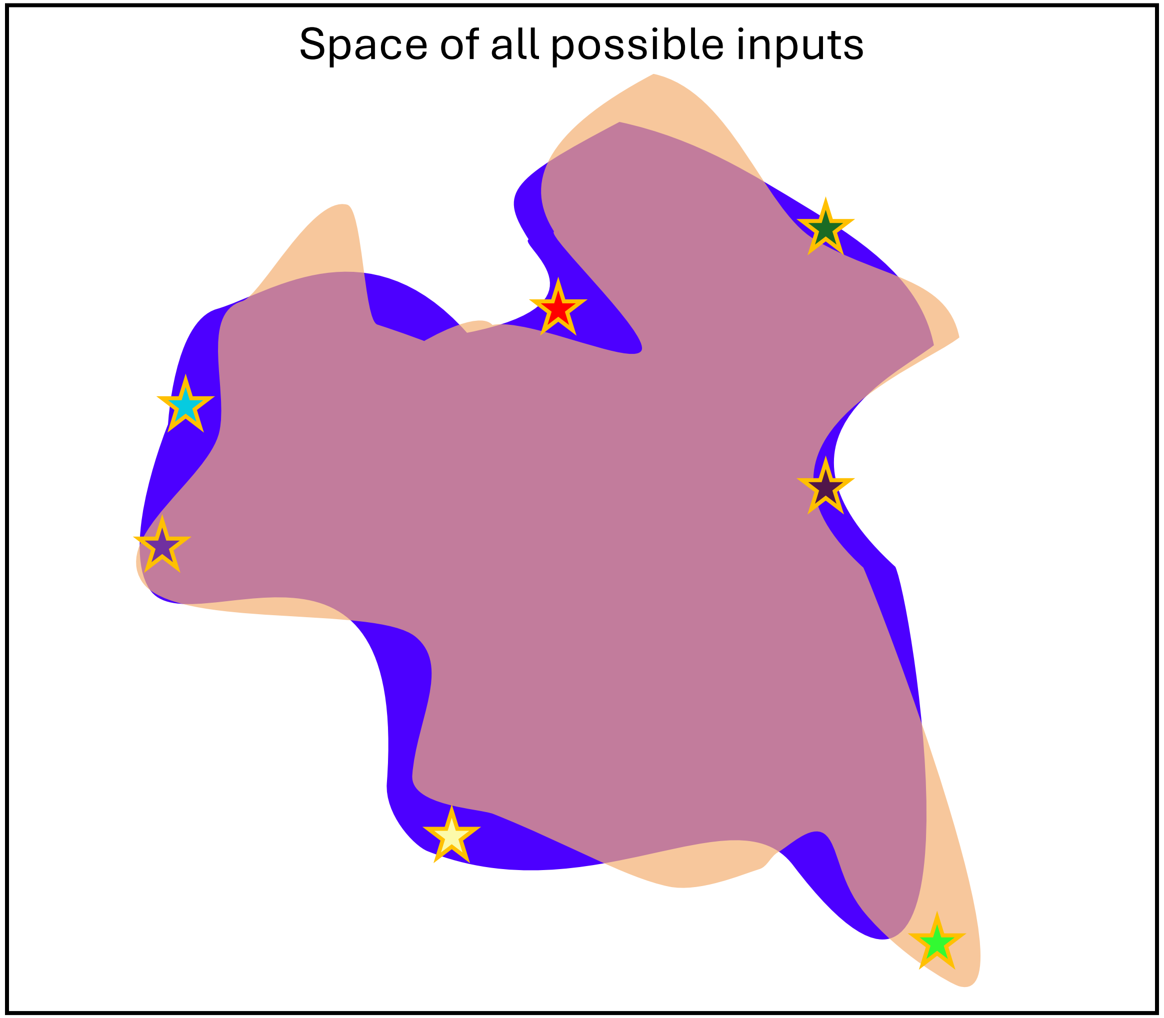
Ground truth (correct program)

Current version

Correctly Input/Output

Incorrect Input/Output

Fault and test case



Automatic ways to discover these test cases

Fuzzing / Fuzz testing

- Developed by **Barton Miller** (Prof. @ University of Wisconsin – Madison)
  - 1988 class project

- Originally meant to test the reliability of UNIX command programs
- Identified that 25-33% of utilities they tested fails
  - Exhibits some incorrect behavior (wrong output, crashes, infinite loops, etc.)

Generate many semi-valid inputs that are not rejected by parsers,
but lead to unexpected behaviors to expose edge cases

Fuzzing / Fuzz testing

1. Generate random input or command-line argument
   - Fits the expected shape (number of parameters, input type, etc.)
   - Values are random
2. Execute the utility script
   - If the script successfully outputs, then it PASSES
     - Does not check the correctness of the program
     - If you have some way to check, you can (e.g., you are optimising a correct function)
   - If the script crashes, hangs, or exhibits memory leak, then it FAILS
3. Repeat Step 1 and 2 over and over many times
   - Millions and billions of different configurations

- If you find a failure, you can add it to the test case repository

Types of fuzzing

- Coverage-guided fuzzing
  - Tracks “code coverage”
  - Trigger all the code logic to make sure output is right
  - Does not guarantee the program is correct
- Mutation-based fuzzing
  - Modifies existing data and input to find crashes
  - Advantages: you know what input crashed it, you can trace it

Use cases

- Incorrect behaviours
- Input errors and crashes
- Can catch memory issues
  - Memory leaks
  - Buffer overflows
  - Use after frees
  - Stack overflows
  - Crashes, etc.
- Security issues
- Can find some very odd programming errors
  - Parallel threading issues (because of how much repeats are happening)
  - Race conditions!!!

Fault Injection

Fault injection

- Stress testing program
- Try explicitly incorrect inputs that lead to errors
  - High voltage
  - Extreme temperature
  - Force incorrect instructions or corrupting critical data
- Particular interest in creating errors to see how the program will handing these exceptions
  - Otherwise, might not be tested often

Example usage: input testing

- Your program loads in a user configuration file
- You want to see what happens if the file gets corrupted!

- Inject a corrupted configuration file to the system to see what happens
  - *Fault tolerance*

Example usage: exception handling

- Exceptions happen… well during exceptional cases
- Might not be able to trigger it with normal test cases
  - API calls that have unexpected values
- You can test the error handling by crafting and injecting the incorrect API return structure

Evolutionary-based Testing

(A bit of my research space)

Evolutionary Inspired Search Testing

- Leverage things we see in nature
- How genes work (genetic algorithm)
- Particle Swarm Optimisation
- Ant Colony Optimisation

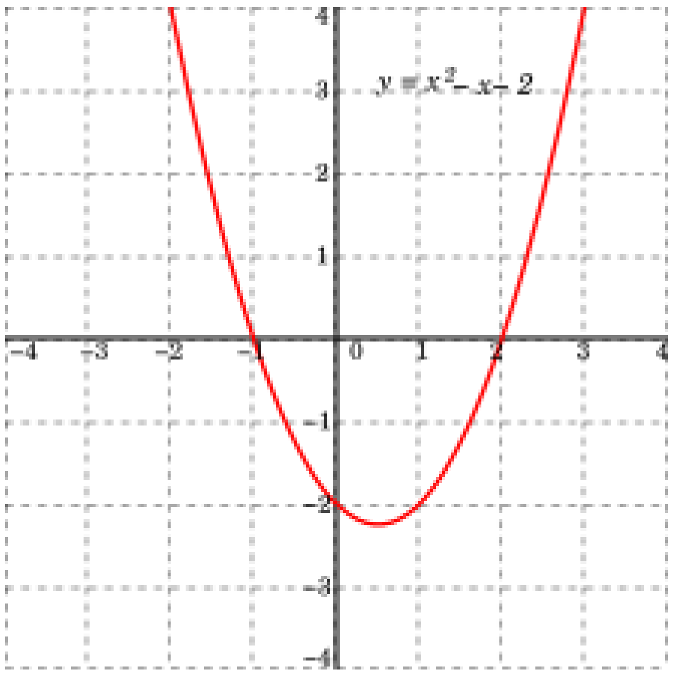
Skipping a lot of details

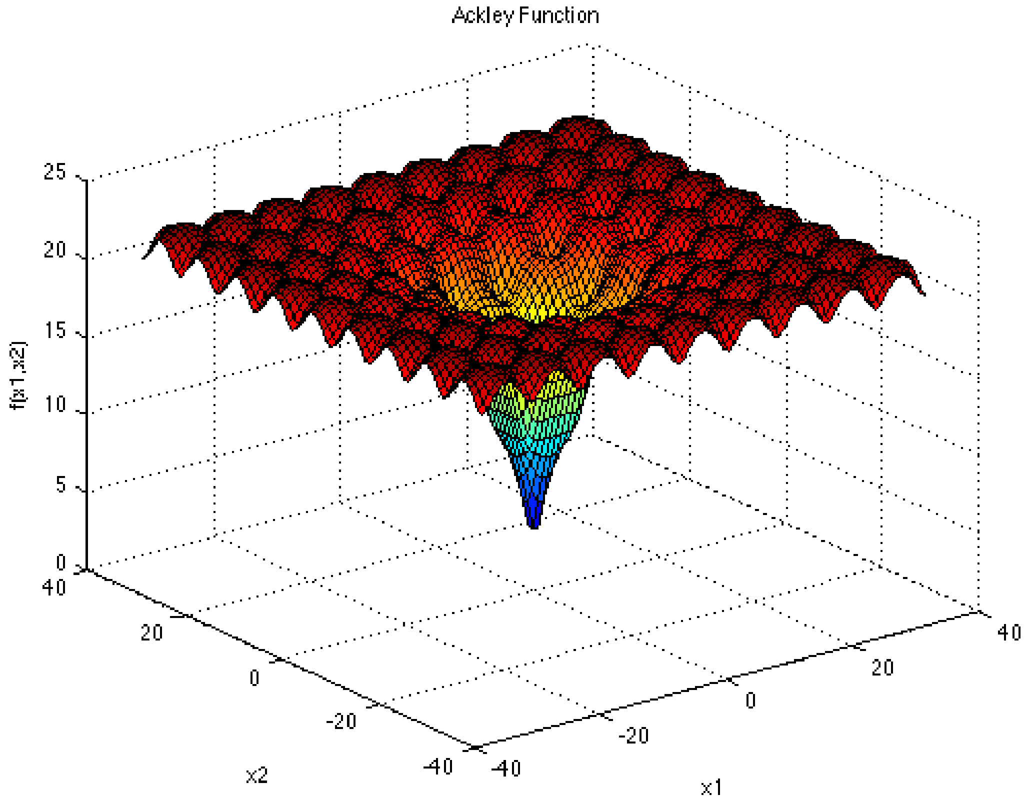
- I am leaving out a lot of details here, otherwise we will spend a semester on this
- Focusing on high-level how it happens

Evolutionary Search-based Testing

- Inspired by Darwinian Evolution
- Map solutions to a ``genome''
- Use nature inspired techniques to evolve solutions (a population)
- Mutation, crossover, and selection mechanism
- Evaluate an individual based on how well they perform (i.e., how close to error)
- Individuals who have high scores get to reproduce
- Slowly get to the solution (incorrect inputs)

Solving for optimum

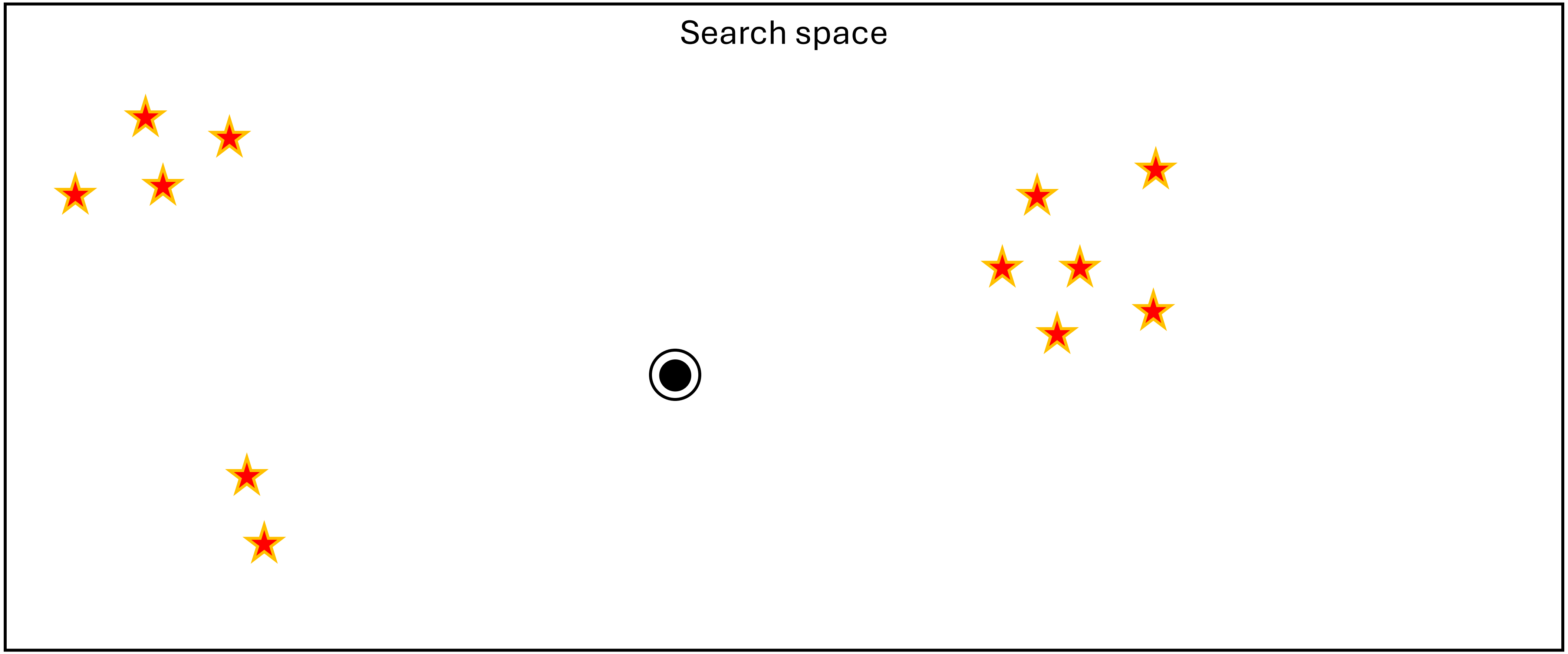




Notes on EC testing

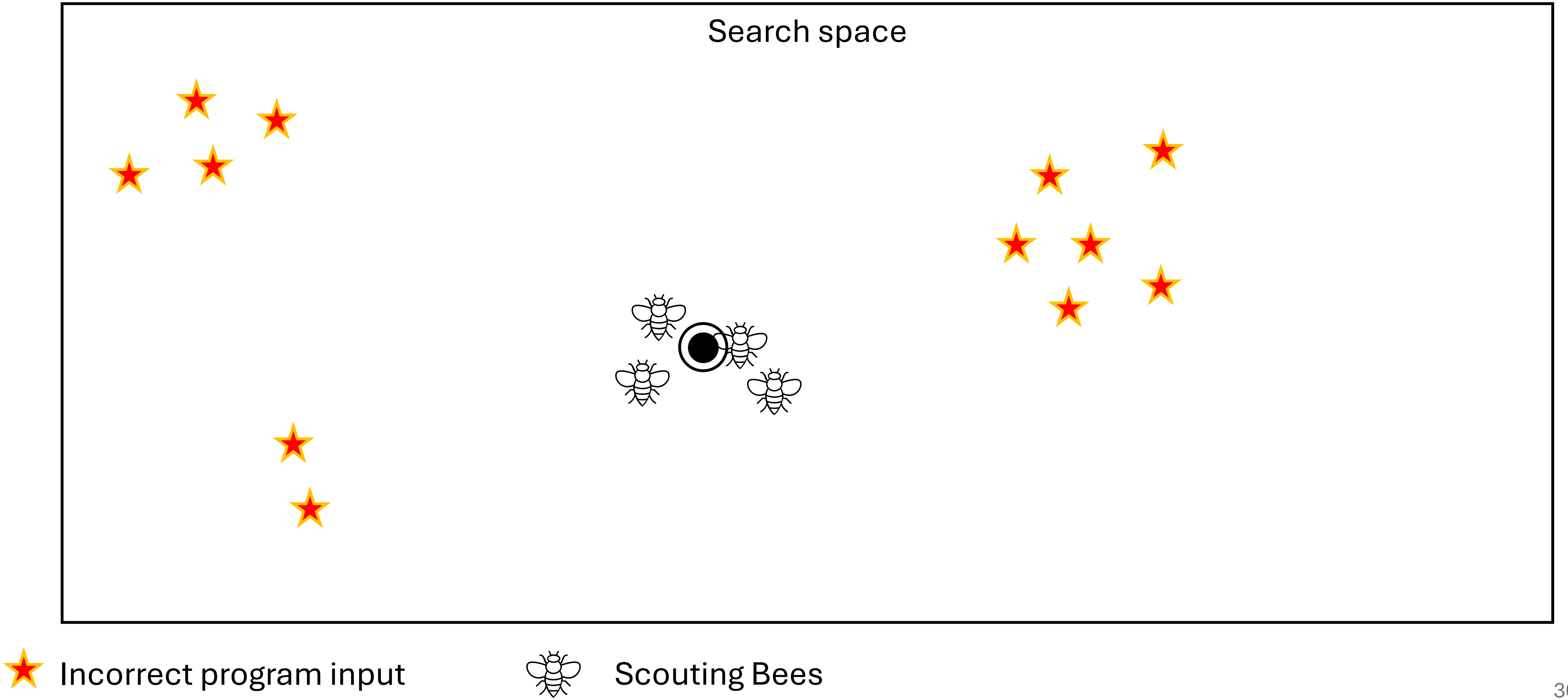
- Computationally expensive
- Discovers non-intuitive test cases
- Very useful for edge case discovery

Swarming algorithms (ant / bees / others)

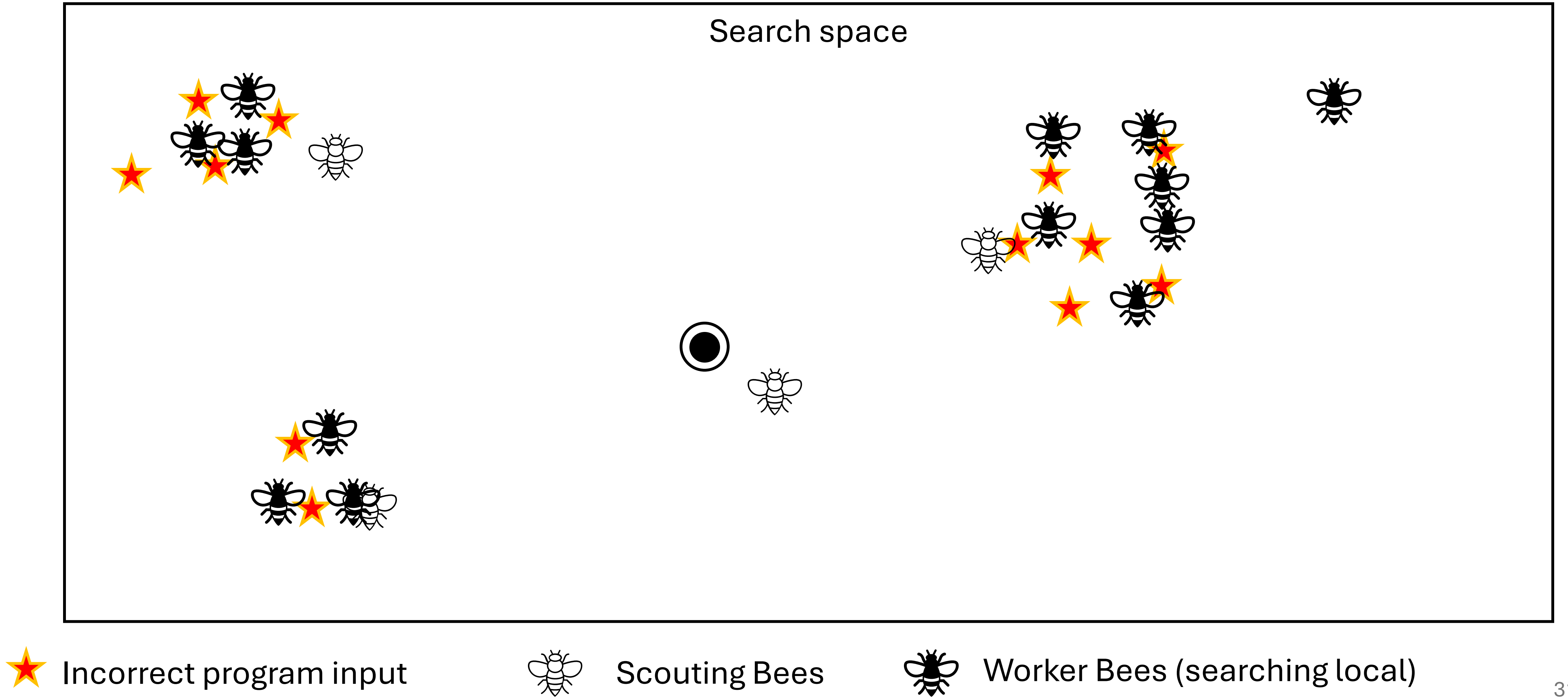


Incorrect program input

Swarming algorithms (ant / bees / others)



Swarming algorithms (ant / bees / others)



Completeness

- When do you stop testing?

- These techniques are not meant to replace traditional testing
- They are complementary, often meant to help find bugs you otherwise would not due to developer bias

- You can write the code and manually test during the day
- Automated testing approaches are done overnight when developers are home

Person of the Day
John Henry Holland

- Pioneer of Evolutionary Computing field
- Introduced genetic algorithms in 1960s
- Received his M.S. and Ph.D. at University of Michigan and have been a professor there