CHALMERS
UNIVERSITY OF TECHNOLOGY

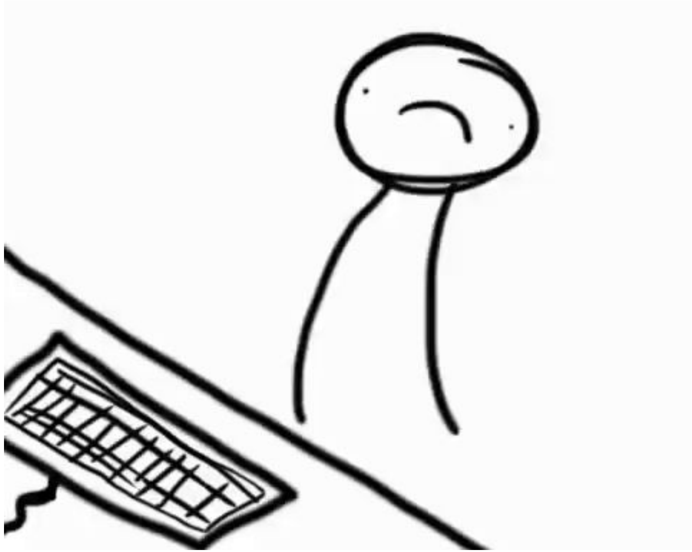
UNIVERSITY OF GOTHENBURG

Lecture 14: Automated Test Case Generation

Gregory Gay
DIT636/DAT560 - March 4, 2026

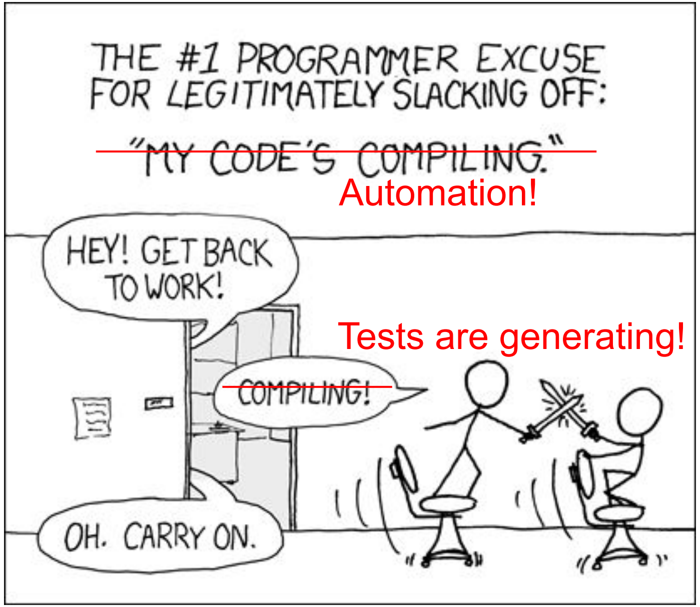
Automating Test Creation

- Testing is invaluable…
- … but **expensive**.
  - We test for ***many*** purposes.
  - Near-infinite number of possible tests we could try.
  - Hard to achieve volume.



Automating Test Creation

- Relieve cost by automating test creation.
  - **Traditional Focus: Generate test input**.
    - Just need to add assertions.
    - (Or measure crashes, performance, etc.)
  - New approaches have *some* ability to **generate test oracles**.



Techniques for Generating Tests

Rationalists (Static)

Generate tests based on **analysis of the source code and other text**.

Empiricists (Dynamic)

Generate tests based on **feedback from executing the system**.

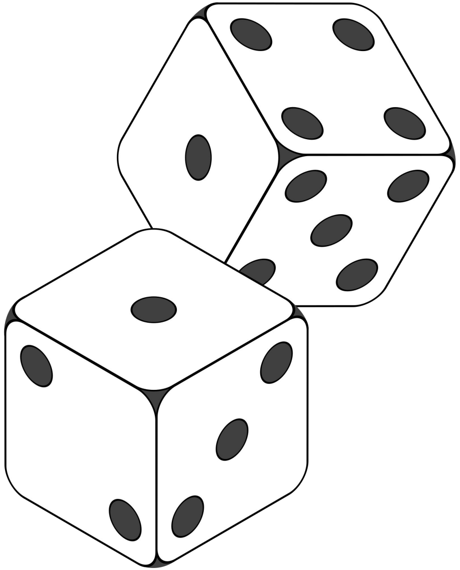
Today's Goals

- Search-Based Test Generation
  - Test creation as an optimization problem, based on feedback from executing the code.
  - Generate -> Execute -> Evolve
- LLM-Based Test Generation
  - Test creation based on textual analysis.

Search-Based Test Generation

Random Generation

- Randomly formulate test cases.
  - Unit testing: choose a class in the system, choose random methods, call with random parameter values.
  - System-level testing: choose an interface, choose random functions from interface, call with random values.
- Keep trying until goal attained or you run out of time.



Example - BMI Calculation

$$BMI = \frac{weight}{(height)^2}$$

	Age						
Classification	$[2,4]$	$(4,7]$	$(7,10]$	$(10,13]$	$(13,16]$	$(16,19]$	> 19
Underweight	≤ 14	≤ 13.5	≤ 14	≤ 15	≤ 16.5	≤ 17.5	< 18.5
Normal weight	≤ 17.5	≤ 14	≤ 20	≤ 22	≤ 24.5	≤ 26.5	< 25
Overweight	≤ 18.5	≤ 20	≤ 22	≤ 26.5	≤ 29	≤ 31	< 30
Obese	> 18.5	> 20	> 22	> 26.5	> 29	> 31	< 40
Severely obese	—	—	—	—	—	—	≥ 40

BMICalc

height
weight
age

bmi_value()
classify_bmi_adults()
classify_bmi_teens_and_children()

Example - BMI Calculation

```
def test_bmi_value_valid():
    bmi_calc = BMICalc(150, 41, 18)
    bmi_value = bmi_calc.bmi_value()
    assert  bmi_value == 18.2

def test_bmi_adult():
    bmi_calc = BMICalc(160, 65, 21)
    bmi_class = bmi_calc.classify_bmi_adults()
    assert bmi_class == "Overweight"

def test_bmi_children_4y():
    bmi_calc = BMICalc(100, 13, 4)
    bmi_class = bmi_calc.classify_bmi_teens_and_children()
    assert bmi_class == "Underweight"
```

BMICalc
height weight age
bmi_value() classify_bmi_adults() classify_bmi_teens_and_children()

Random Generation - BMI Example

- Create an empty test case:

```
def test_1():
```

- Instantiate the class-under-test with random values:

```
def test_1():
    cut = BMICalc(180, 50, 40)
```

- Insert 1+ method calls or assignments to class variables.
  - Number of calls is random
  - Which method/variable is random
  - Method parameters are random values

BMICalc
height weight age
bmi_value() classify_bmi_adults() classify_bmi_teens_and_children()

```
def test_1():
  cut = BMICalc(180, 50, 40)
  output = cut.bmi_value()
  cut.height = 15681
  output2 = cut.classify_bmi_adults()
```

Random Search

- Sometime viable:
  - Extremely fast.
  - Easy to implement, easy to understand.
  - All inputs considered equal, so no designer bias.
- However…

Test Creation as a Search Problem

- Do you have a **goal** in mind when testing?
  - *Make the program crash, achieve code coverage, find performance bottlenecks, …*
- **Searching** for a test suite that achieves that goal.
  - Based on **guess-and-check** process.

Test Creation as a Search Problem

- Many testing goals can be measured:
  - How many exceptions were thrown?
  - How fast was the code?
  - What percentage of lines of code were covered?
  - How diverse is our input?
- If goal can be measured, search can be automated.

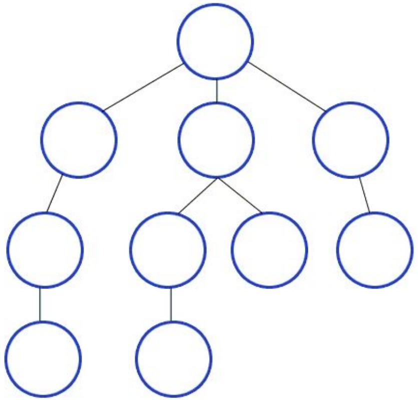
Search-Based Test Generation

- **Make one or more guesses.**
  - Generate one or more individual test cases or full test suites.
- **Check whether goal is met.**
  - Score each guess.
- **Try until time runs out.**
  - Alter the solution based on feedback and try again!

Search Strategy

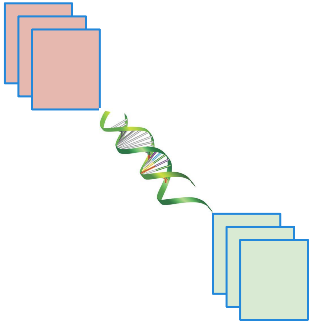
- The order that solutions are tried is the key to efficiently finding a solution.
- A search follows some defined strategy.
  - Called a "**metaheuristic**".
- Metaheuristics are used to choose solutions and to ignore solutions known to be unviable.
  - Smarter than pure random guessing!

Heuristics - Graph Search

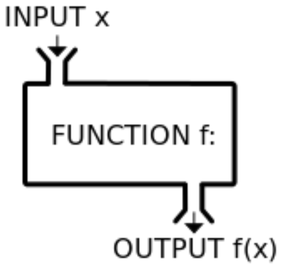


- Arrange nodes into a hierarchy.
  - Breadth-first search looks at all nodes on the same level.
  - Depth-first search drops down hierarchy until backtracking must occur.
- Attempt to estimate shortest path.
  - A* search examines distance traveled and estimates optimal next step.
  - Requires domain-specific scoring function.

Search-Based Test Generation



+



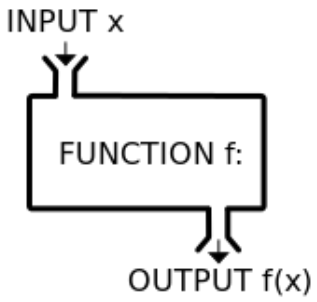
=

The Metaheuristic (Sampling Strategy)	**The Fitness Functions (Feedback Strategies)**	**(Goals)**
Genetic Algorithm Simulated Annealing Hill Climber (...)	Distance to Coverage Goals Count of Executions Thrown Input or Output Diversity (...)	Cause Crashes Cover Code Structure, Generate Covering Array, (...)

Solution Representation

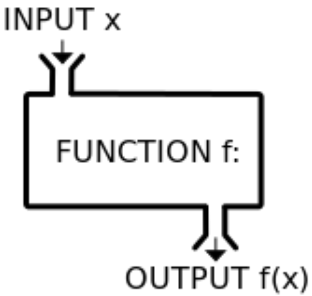
- Must decide what a solution “looks like”.
- For unit testing:
  - A solution is a test suite.
  - A test suite contains 1+ test cases.
  - Each test case interacts with a class-under-test.
  - Each test case initialized the class-under-test.
  - Each test case contains one or more actions.
    - An action is a method call or variable assignment.
    - Each action has parameters (method parameters or values to assign to variables).

Fitness Functions



- Domain-based scoring functions that determine how good a potential solution is.
  - Should represent goals of tester.
  - Must return a numeric score.
    - % of a checklist
    - raw number
    - NOT Boolean (no feedback)
  - Can be maximized or minimized.

Fitness Functions



- **Should offer feedback:**
  - Small change in solution should not lead to large change in score.
  - Best functions calculate *distance* to optimality.
- **Can optimize more than one at once.**
  - Independently optimize functions
  - Combine into single score.

Example - Code Coverage

- **Goal:** Attain Branch Coverage over the code.
  - Tests must reach all branching points (i.e., if-statement) and execute all possible outcomes.

```
if(x < 10){
    // Do something.
}else if (x == 10){
    // Do something else.
}
```

In this code:

- Two Branches
- Each must evaluate to `true` and `false`.

Example - Code Coverage

- **Goal:** Attain Branch Coverage over the code.
- **Fitness function (Basic):**
  - Measure coverage and try to maximize % covered.
  - **Good:** Measurable indicator of progress. Can use standard tools (pytest-cov, Cobertura).
  - **Bad:** No information on how to improve coverage.

Example - Code Coverage

- Advanced: Distance-Based Function
- **fitness = branch distance + approach level**
  - **Approach level**
    - Number of branching points we need to execute to get to the target branching point.
  - **Branch distance**
    - If other outcome is taken, how "close" was the target outcome?
    - How much do we need to change program values to get the outcome we wanted?

Example - Branch Coverage

```
if(x < 10){ // Branch 1
    // Do something.
}else if (x == 10){ // Branch 2
    // Do something else.
}
```

Goal: Branch 2, True Outcome

Approach Level

- If Branch 1 is true, approach level = 1
- If Branch 1 is false, approach level = 0

Branch Distance

- If x==10 evaluates to false, branch distance = (abs(x-10)+k).
- Closer x is to 10, closer the branch distance.

Other Common Fitness Functions

- Number of methods called by test suite
- Number of crashes or exceptions thrown
- Diversity of input or output
- Detection of planted faults
- Amount of energy consumed
- Amount of data downloaded/uploaded
- **… (anything that reflects what a *good* test is)**

Bloat Penalty

- Small penalty subtracted from fitness.
- Limits number of tests and number of actions.

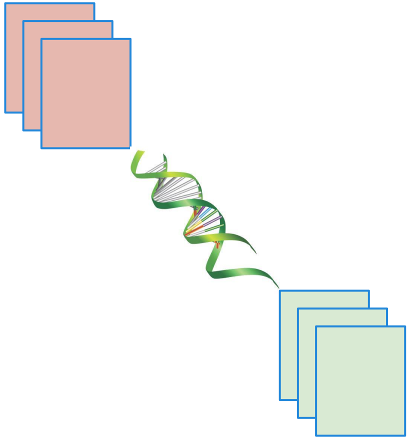
ex. 10

$$
\begin{aligned}
bloat\_penalty(solution) = {} & (num\_test\_cases/num\_tests\_penalty) \\
& + (average\_test\_length/length\_test\_penalty)
\end{aligned}
$$

ex. 30

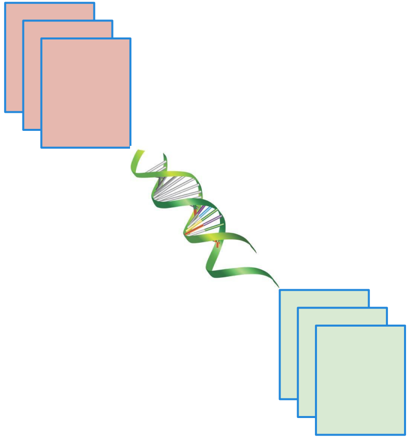
- Important not to penalize too heavily.

The Metaheuristic



- Decides how to select and revise solutions.
  - Changes approach based on past guesses.
  - Fitness functions give feedback.
  - Population mechanisms choose new solutions and determine how solutions evolve.

The Metaheuristic



- Decides how to select and revise solutions.
  - Small changes to single solution **(local search).**
  - Large changes to many solutions **(global search)**.
  - Often based on natural phenomena.
    - (swarm behavior, evolution)
  - Trade-off between speed, complexity, and understandability.

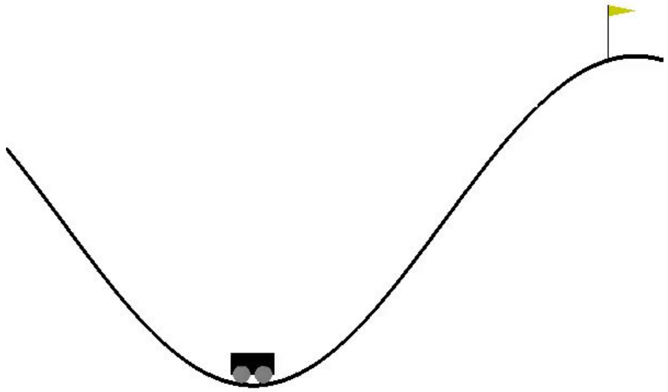
How Long Do We Spend Searching?

- Exhaustive search not viable.
- Search can be bound by a **search budget**.
  - Number of guesses.
  - Time allotted to the search (number of minutes/seconds).
- **Optimization problem**:
  - *Best solution possible before running out of budget.*

Local Search

- Generate and score a single potential solution.
- Attempt to improve by looking at its **neighborhood**.
  - Make small, incremental improvements.
- Very fast, efficient if good initial guess.
  - Get “stuck” if bad guess.
  - Often include reset strategies.

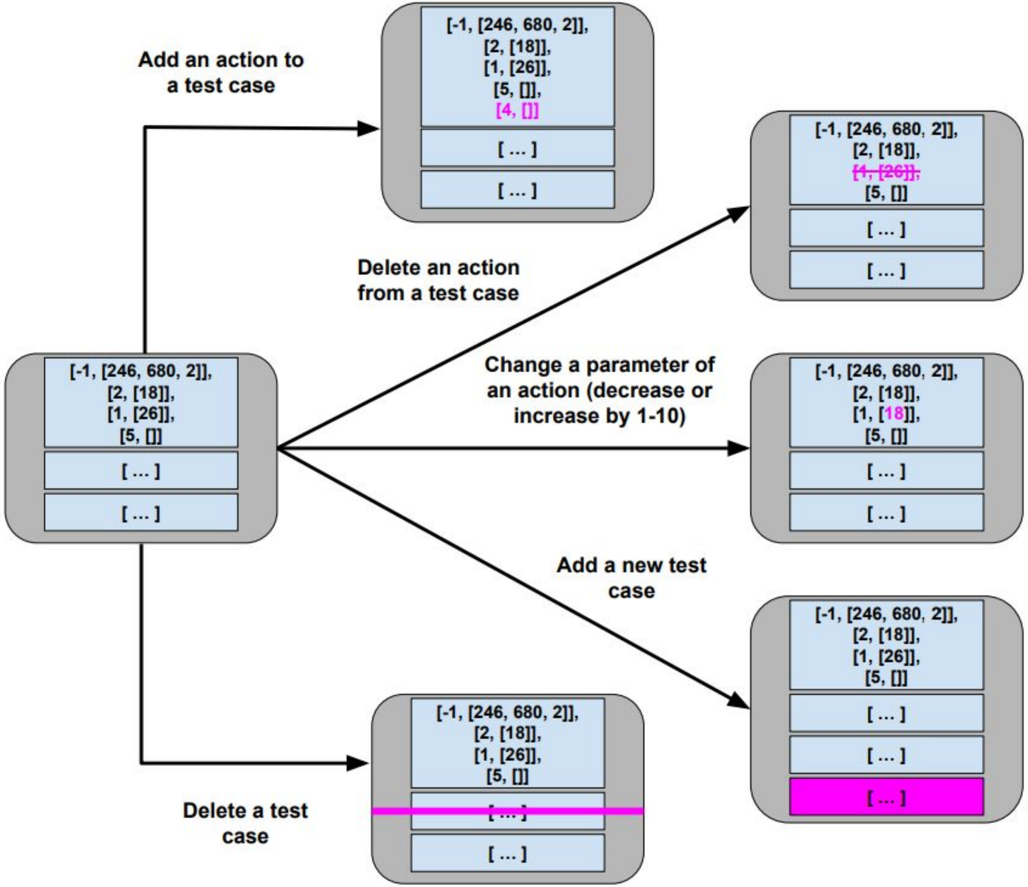
Hill Climbing



- Generate a random initial solution.
- Each generation (while budget remains):
  - Attempt up to `max_tries` *mutations* to the solution.
    - If a mutation results in a better solution, set this as the new solution.
    - Keep track of the best mutation seen to date.
  - If we run out of tries, reset to a new random initial solution.

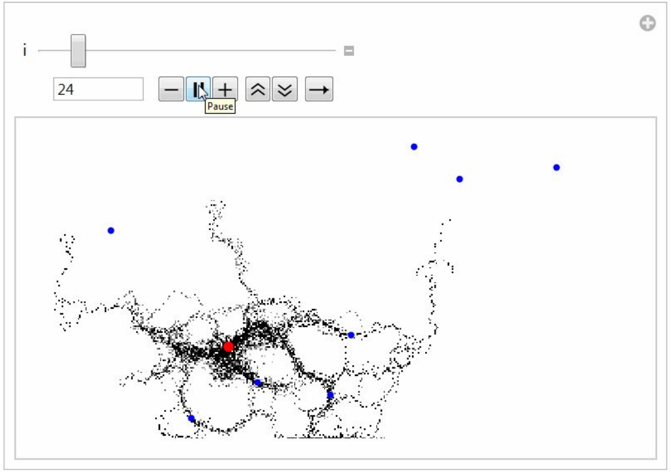
Mutation

- Small change to current solution.
- Impose one of these changes at a time:



Let’s take a break.

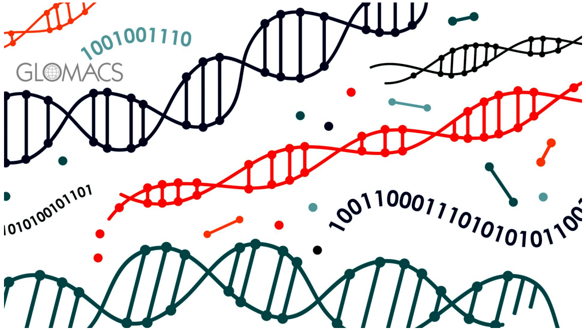
Global Search



- Generate multiple solutions.
- Evolve by examining whole search space.
- Typically based on natural processes.
    - Swarm patterns, foraging behavior, evolution.
    - Models of how populations interact and change.

Genetic Algorithm

- Over multiple generations, evolve a population.
  - Good solutions persist and reproduce.
  - Bad solutions are filtered out.
- Diversity is introduced by:
  - **Selecting** the best solutions.
  - Creating “offspring” through **mutation** and **crossover**.

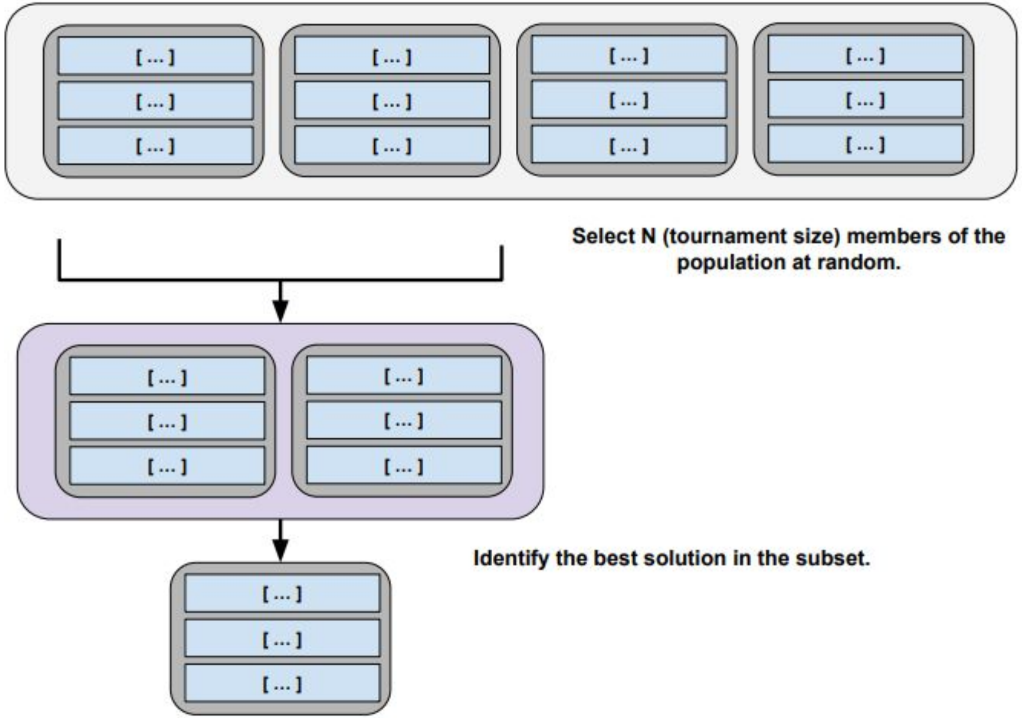


Genetic Algorithm

- Create a random initial population.
- Start a new generation (while budget remains):
  - Create new empty population.
  - While space remains:
    - **Select** two "good" members of current population.
    - At a small probability, replace these members with "children" combining genes of members (**crossover**).
    - At a small probability, **mutate** each member.
    - Add members to **new population**.
  - If no better solution is found for N generations, terminate early (**stagnation**).

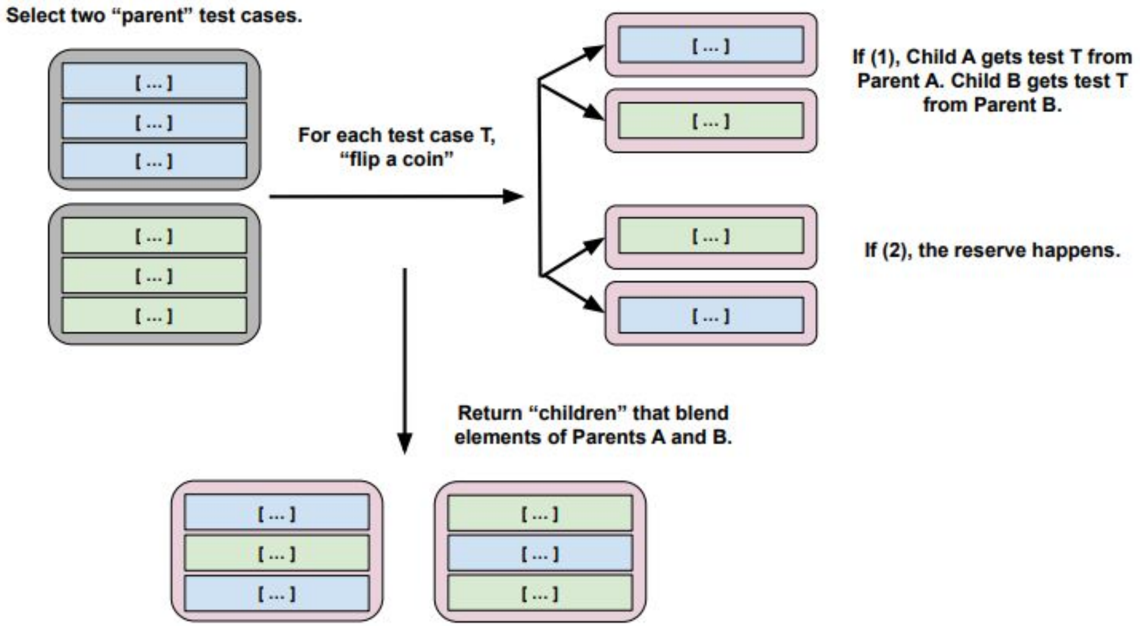
Selection

- Rather than searching for "best" population member:
  - Select a random subset.
  - Calculate fitness for each.
  - Return best.



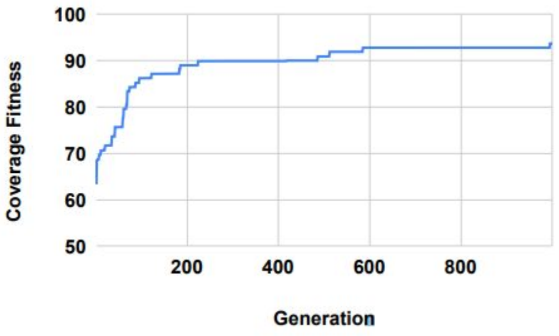
Crossover

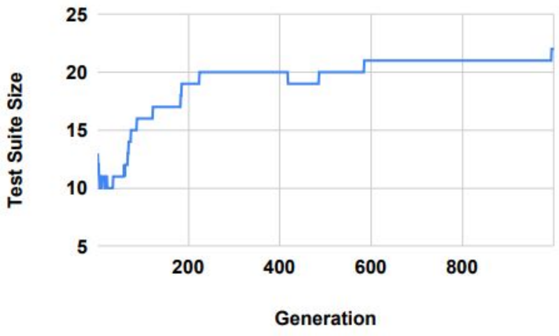
- Creates “child” solutions by combining tests from each parent.

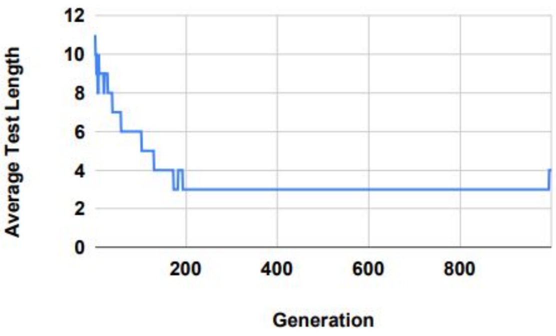


1000 Generations of Evolution

- Genetic Algorithm run for 1000 generations for BMICalc.
- Stagnation turned off.
- Highly variable until ~ 200 generations, then small changes afterwards.







Examples of Generated Test Cases

```
def test_0():
    cut = bmi_calculator.BMICalc(120,860,13)
        cut.classify_bmi_teens_and_children()

def test_2():
        cut = bmi_calculator.BMICalc(43,243,59)
        cut.classify_bmi_adults()
        cut.height = 526
        cut.classify_bmi_adults()
        cut.classify_bmi_adults()

def test_5():
        cut = bmi_calculator.BMICalc(374,343,17)
        cut.age = 123
        cut.classify_bmi_adults()
        cut.age = 18
        cut.classify_bmi_teens_and_children()
        cut.weight = 396
        cut.classify_bmi_teens_and_children()
```

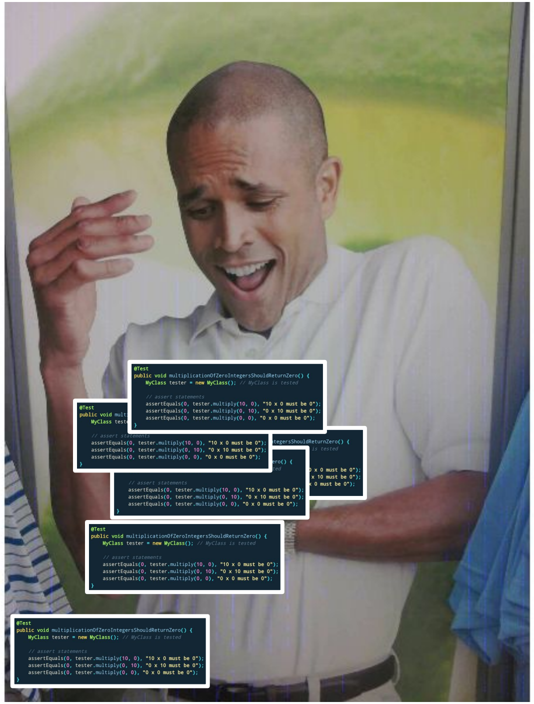
```
def test_7():
        cut = bmi_calculator.BMICalc(609,-1,94)

def test_11():
        cut = bmi_calculator.BMICalc(491,712,20)
        cut.classify_bmi_adults()

def test_17():
        cut = bmi_calculator.BMICalc(608,717,6)
        cut.classify_bmi_teens_and_children()
        cut.age = 91
        cut.classify_bmi_teens_and_children()
        cut.classify_bmi_teens_and_children()
```

What Do I Do With These Inputs?

- If looking for crashes, just run generated input.
- If you need to judge correctness, add assertions.
    - Suggested: general properties, rather than specific expected output.
        - **No**: `assertEquals(output, 2)`
        - **Yes:** `assertTrue(output % 2 == 0)`

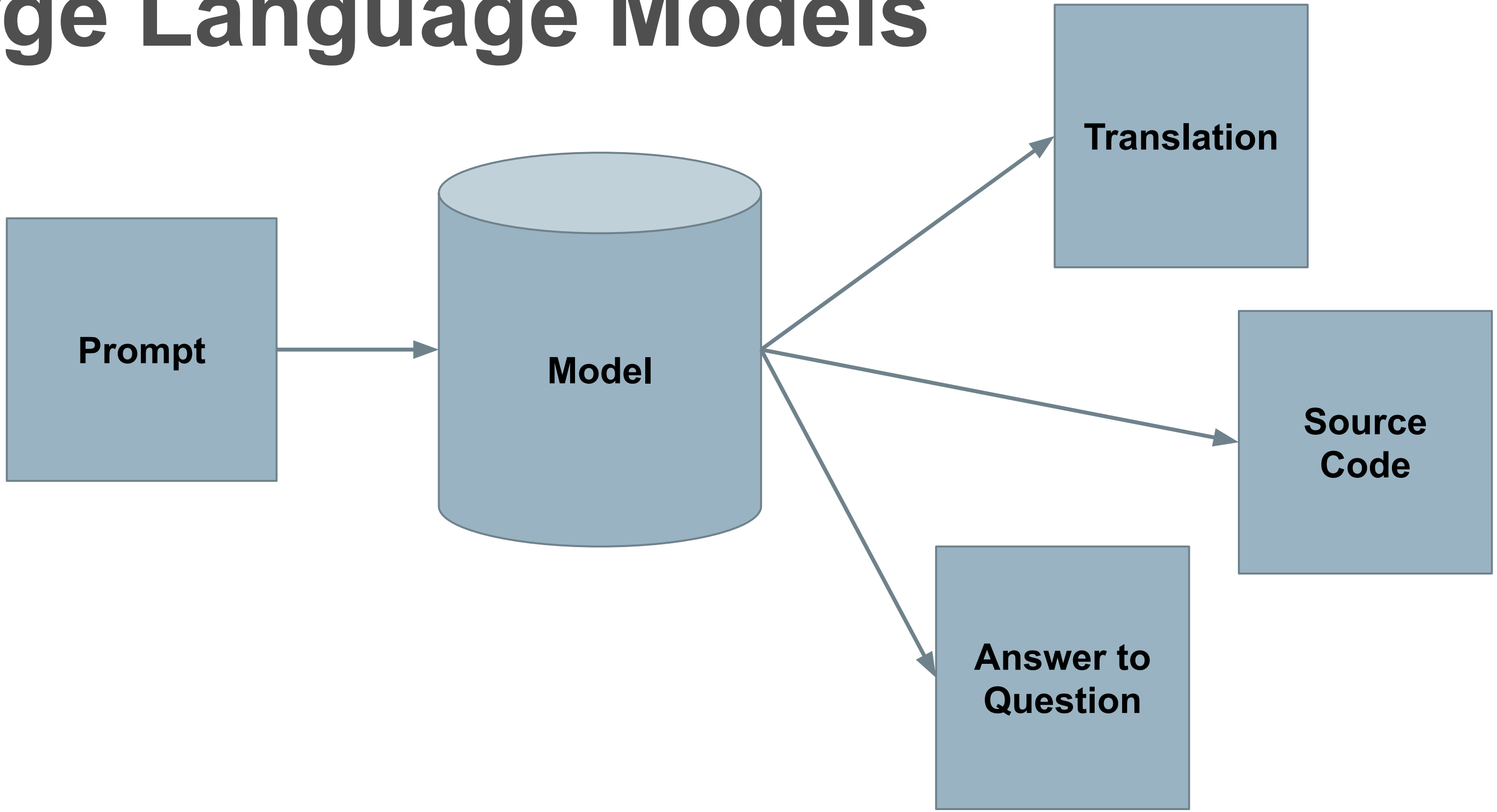


I Want to Try This Out!

- Python:
  - https://greg4cr.github.io/pdf/21ai4se.pdf
  - https://github.com/Greg4cr/PythonUnitTestGeneration
- Java: http://www.evosuite.org/
- C/C++: https://aflplus.plus/

Large Language Models

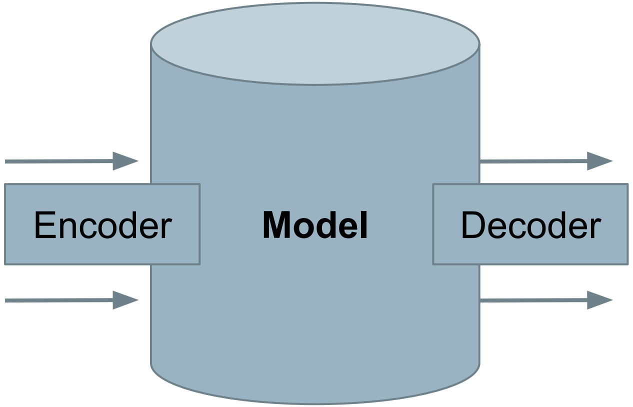
Large Language Models



Large Language Models

"The **server** brought me the **bill**"

"We sent an **HTTP request** to the **server**"



for my **three-course dinner**.

to update the **database record**.

Important Considerations

- **Prompt Design**
  - The structure and information provided in the prompt.
- **Model Selection**
  - Type of model.
  - Closed vs open source, Local vs remote execution.
- **Agentic Structure**
  - Tool use, memory, workflow.

Prompt Engineering

- General principle: **Clear context, better results**.
  - Information about the code-under-test.
  - Expectations on the results.
- Basic Components:
  - {Role} - Persona the LLM should adopt.
  - {Context} - Details about the code-under-test.
  - {Instructions} - Instructions on test generation.
  - {Examples} - Examples of existing test cases.

Basic Structure

You are a software test engineer, developing unit test cases for a Python class.

{Context}

Create a unit test suite of Pytest-formatted test cases for this class.

{Examples}

Varying Context

- High-level description:
  - The purpose of this Python class is to calculate the BMI value and classification of adults, as well as teens and children.
- Description, method signatures:
  - The purpose of this Python class is to calculate the BMI value and classification of adults, as well as teens and children. This class has three variables: height, weight, and age. It offers setter methods height(self, height), age(self, age), and weight(self, weight). These methods check for negative values. The class also offers the following methods: bmi_value(self), classify_bmi_teens_and_children(self), and classify_bmi_adults(self)
- Description, code:
  - The purpose of this Python class is to calculate the BMI value and classification of adults, as well as teens and children. The code of the class is: {code}

Examples

- Can include examples of human-written tests:
  - **Zero-Shot:** No examples provided.
  - **One-Shot:** One example test provided.
  - **Few-Shot:** Multiple examples provided.

Here is an example of an existing test case for the class:

```
def test_bmi_adult():
    adult_age = 21
    bmi_calc = bmi_calculator.BMICalc(160, 65, 21)
    bmi_class = bmi_calc.classify_bmi_adults()
    assert bmi_class == "Overweight"
```

This test checks the BMI classification of an adult who is 160 cm tall, weights 65 kilograms, and is 21 years old.

Demonstration

Choosing an LLM

- Type of model:
  - **Instruction:** Tuned for following directions and returning results in a specified format.
  - **Chat:** Tuned for conversations with a user (e.g., Q&A).
- Size (number of parameters)
  - More generally yields better results, but much higher computational cost.

Choosing an LLM

- **Open Source:** Creators disclose contents of the training data and how the model was tuned.
  - MapNEO, OLMo
- **Open Weight:** Creators disclose how model was tuned, but not training data.
  - DeepSeek, Llama, Mistral
- **Closed Source:** Neither data or weights disclosed.
  - OpenAI models

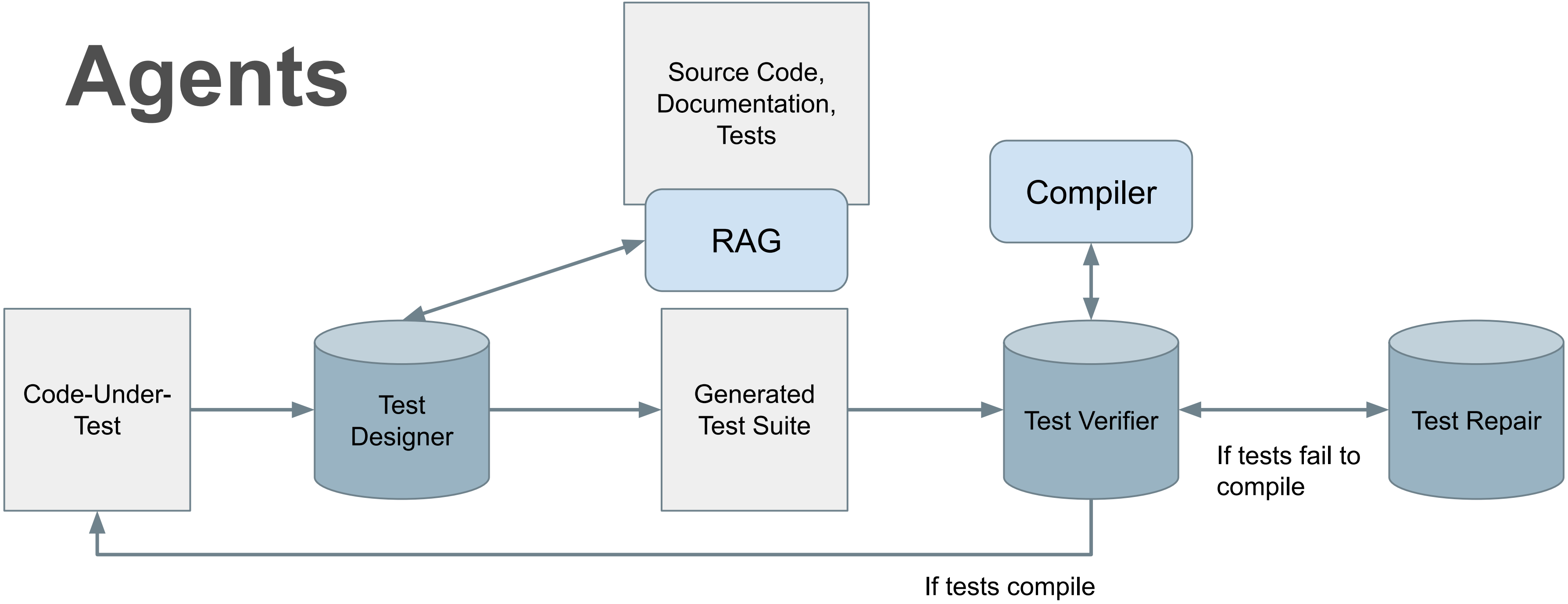
Choosing an LLM

- **Local execution:** Model deployed locally.
- **Remote execution:** Model executed via API on servers owned by model creator.
- Consider costs of both options.
  - License/access vs hardware requirements
- Data privacy concerns with remote execution.
  - OpenAI stores and uses your input data unless you pay for a corporate license.

Agents

- An **agent** pairs an LLM/prompt with:
  - **Tool access** - LLM can access other programs, invoke scripts, access data store.
  - **Memory** - LLM stores intermediate reasoning for later.
    - Debugging for developer.
    - Improve future results by using earlier starting point.
- Common to split a task into sub-tasks completed by **cooperating agents**.

Agents



- RAG gives way to look up relevant items in project documentation.
  - Reduces hallucinations
- Compiler can verify that tests are not broken.
  - Can repair broken/hallucinated test code.

Comparing Approaches

Search-Based Test Generation

- Advantages:
  - Does not require knowledge of the code.
    - Do not need similar training data.
  - Can be implemented for any system, language, platform.
  - Can be parallelized and is computationally efficient.

Search-Based Test Generation

- Disadvantages:
  - Lacks knowledge of the code.
    - Random selection of input - “blind guessing”
    - Improving coverage requires being guided to the right input.
    - Tests are hard to understand.
      - Input and method sequences that a human may not pick.
      - Limited “rationale” for test case purpose.

LLM-Based Test Generation

- Advantages:
  - Can infer how the code works.
    - (as long as there is similar training data)
    - Can be more coverage of program outcomes/behaviors.
  - Tests closer to what a human would produce.
    - Each test has a single purpose.
    - Understandable input and method sequences.
  - Can generate documentation and assertions.
    - More complete test cases.

LLM-Based Test Generation

- Disadvantages:
  - Inferences from code may be incorrect.
    - Code may not compile.
    - Code may contain hallucinated functionality/methods.
    - Tests may not correspond to actual implementation, just similar training examples.
    - Tests may assume faulty code is correct.
  - Tests may achieve limited coverage.
  - Limited ability to generate tests that expose performance/quality issues.

Summary

- Search-Based Test Generation
  - Test creation as an optimization problem, based on feedback from executing the code.
  - Generate -> Execute -> Evolve
- LLM-Based Test Generation
  - Test creation based on textual analysis.
  - Growing in prevalence, capability.

Next Time

- Testing in Industry
  - Guest lectures from Spotfire, TestScouts
  - Attend! (Some students got internships last two years)

- Assignment 4 - Due March 8
- Assignment 5 - Due March 13

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY