



CHALMERS
UNIVERSITY OF TECHNOLOGY



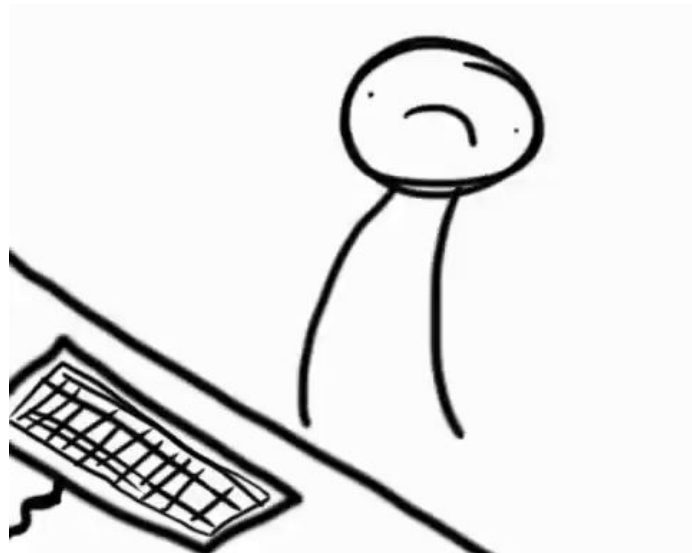
UNIVERSITY OF GOTHENBURG

Lecture 12: Automated Test Case Generation

Gregory Gay
TDA 594/DIT 593 - December 9, 2021

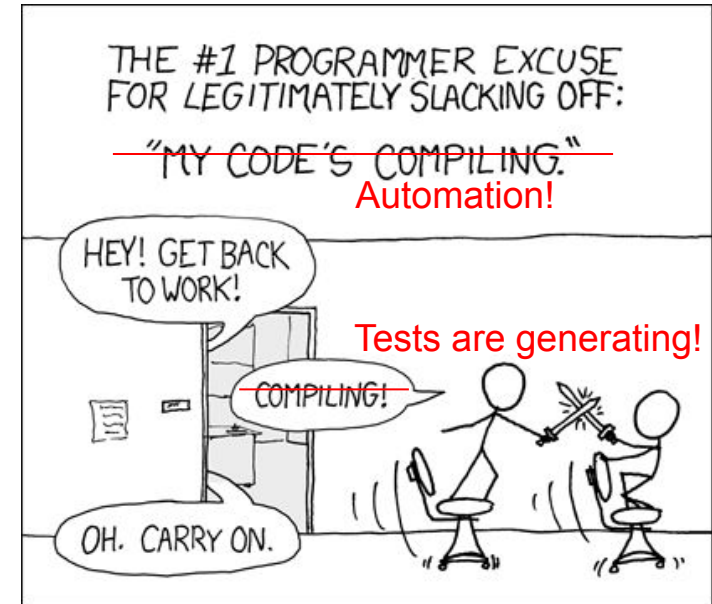
Automating Test Creation

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



Automation of Test Creation

- Relieve cost by automating test creation.
 - Repetitive tasks that do not **need** human attention.
 - **Generate test input.**
 - Need to add assertions.
 - Or just look for crashes.



Today's Goals

- Introduce Search-Based Test Generation
 - (AKA: Fuzzing)
 - Test Creation as a Search Problem
 - Metaheuristic Search
 - Fitness Functions
- Example - Generating Covering Arrays for Combinatorial Interaction Testing

Test Creation as a Search Problem

- Do you have a **goal** in mind when testing?
 - *Make the program crash, achieve code coverage, cover all 2-way interactions, ...*
- You are **searching** for a test suite that achieves that goal.
 - Algorithm samples possible test input to find those tests.

Test Creation as a Search Problem

- “I want to find all faults” cannot be measured.
- *However, a lot of testing goals can be.*
 - Check whether properties satisfied (boolean)
 - Measure code coverage (%)
 - Count the number of crashes or exceptions thrown (#)
- 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 suites.
- **Check whether goal is met.**
 - Score each guess.
- **Try until time runs out.**
 - Alter the population based on strategy and try again!

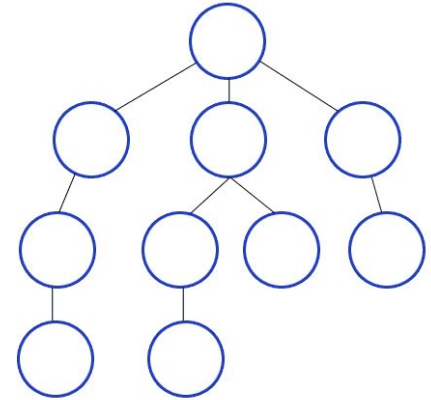


Search Strategy

- The order that solutions are tried is the key to efficiently finding a solution.
- A search follows some defined strategy.
 - Called a “**heuristic**”.
- Heuristics 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.

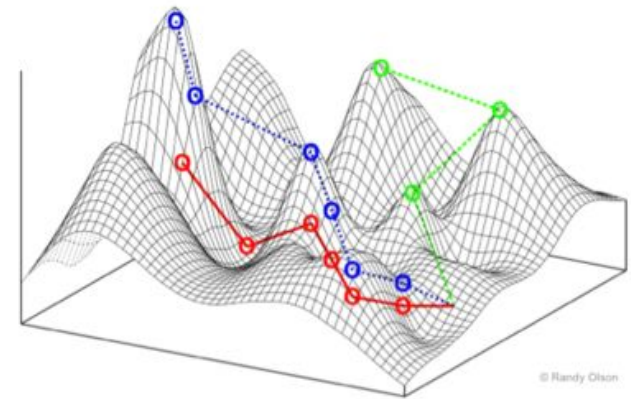


How Long Do We Spend Searching?

- 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.*

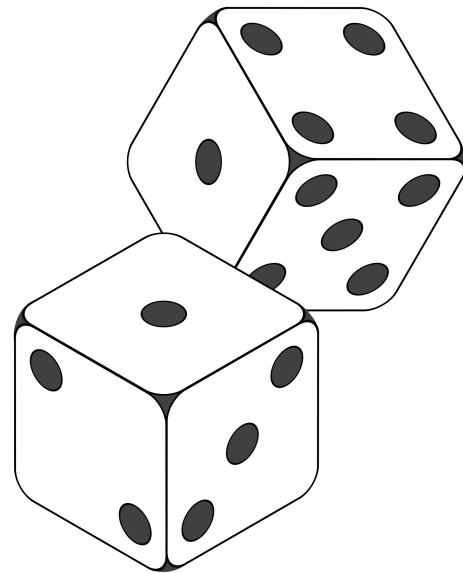
Generation as Optimization Problem

- Search heuristic becomes important.
 - If time bound: time to create, execute, and evaluate.
 - If attempt bound: strategy used to choose next solution.
 - Ignoring bad solutions, learning what a solution good.
- In practice, **efficiency in both categories is desired.**



Random Search

- Randomly formulate a solution.
 - 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 budget expires.



Random Search

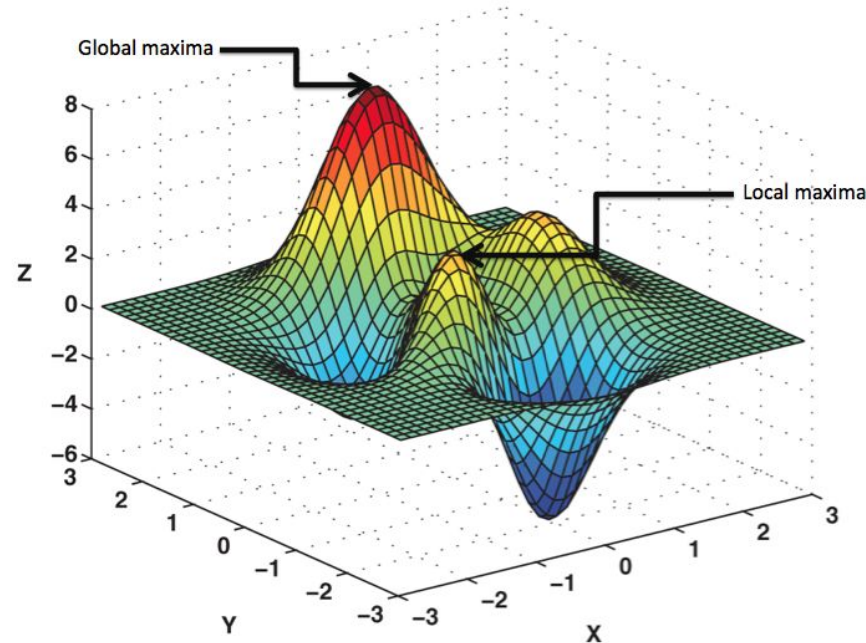
- Sometime viable:
 - Extremely fast.
 - Easy to implement, easy to understand.
 - All inputs considered equal, so no designer bias.

- However...



Metaheuristic Search

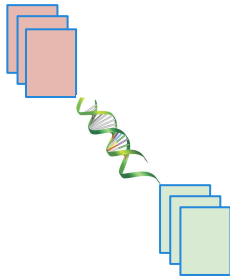
- Random search is naive.
 - Only possible to cover a small % of full input space.
- Metaheuristic search adds intelligence to random.
 - Feedback and sampling strategies.
 - Still fast, able to learn from bad guesses.



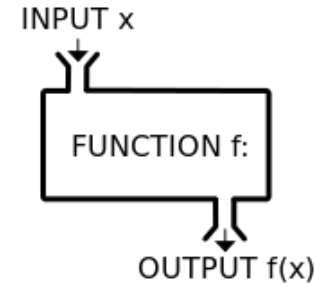
Metaheuristic Search

Mechanics of Optimization

AKA: How can I get a computer to search?

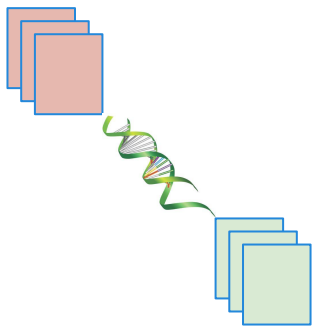


Metaheuristic



Fitness Function(s)

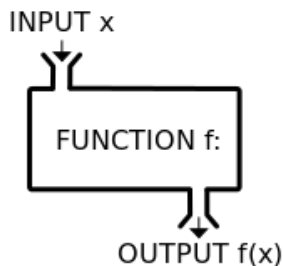
Search-Based Test Generation



The Metaheuristic (Sampling Strategy)

Genetic Algorithm
Simulated Annealing
Hill Climber
(...)

+



The Fitness Functions (Feedback Strategies)

Distance to Coverage Goals
Count of Executions Thrown
Input or Output Diversity
(...)

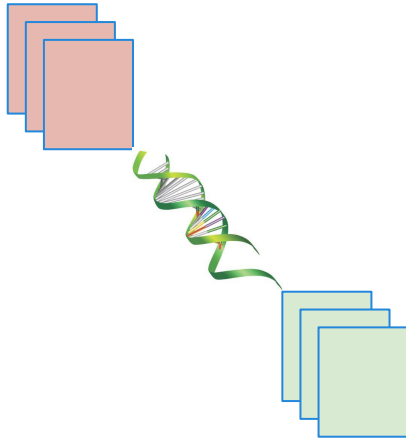
=



(Goals)

Cause Crashes
Cover Code Structure,
Generate Covering Array,
(...)

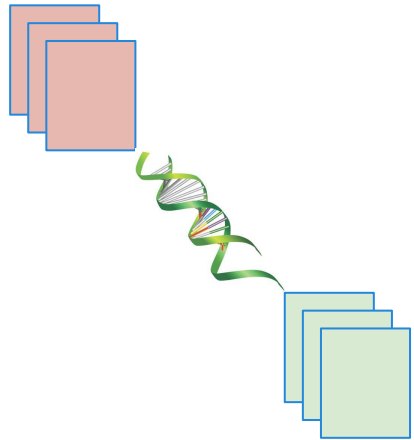
The Metaheuristic



- 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 adjustments (**local search**) or sampling from the whole space (**global search**).
 - One solution at a time or entire populations.
 - Often based on natural phenomena (swarm behavior, evolution).
 - Trade-off between speed, complexity, and understandability.



“Solutions”

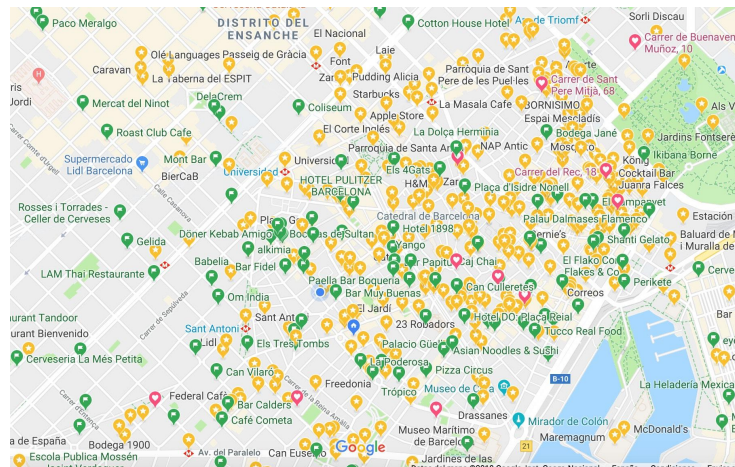
- What is a solution?
 - **Test Case:** Evolved in isolation from other test cases.
 - **Test Suite:** A set of test cases, evolved together.
- Depends on how goal attainment measured.
 - Code Coverage
 - Test Case: Target one code section at a time.
 - Test Suite: Target coverage of entire class/system.

Local Search

- Generate and score a 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.

Exploring the Neighborhood

- Small changes to solution.
 - For each call:
 - Switch value of boolean, other values from an enumerated set, bounded range of numeric choices.
 - Full test case:
 - Insert a new call.
 - Delete or replace an existing call.
 - Can replace by changing the function called or its parameters.

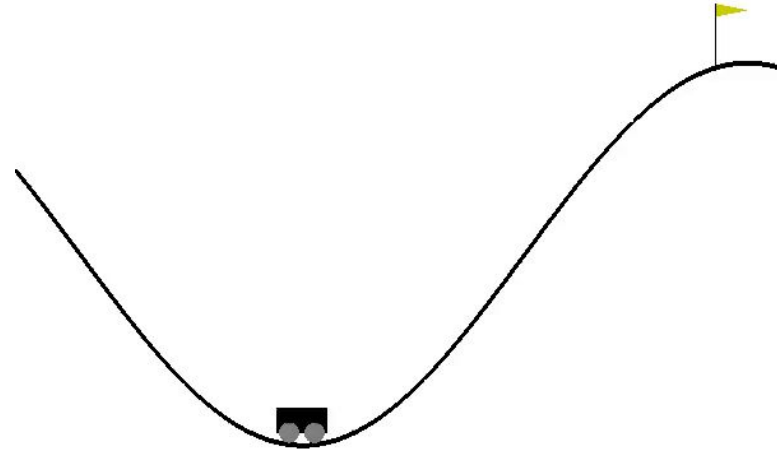


Hill Climbing

- Pick a initial solution at random.
- Examine the local neighborhood.
- Choose the best neighbor and “move” to it.
- Repeat until no better solution can be found.
 - Climbs mountains in fitness function landscape.
 - Restart when no improvement can be found.

Hill Climbing Strategies

- **Steepest Ascent**
 - Examine all neighbors
 - Pick one with highest improvement.
- **Random Ascent**
 - Examine random neighbors.
 - Choose first to show *any* improvement.



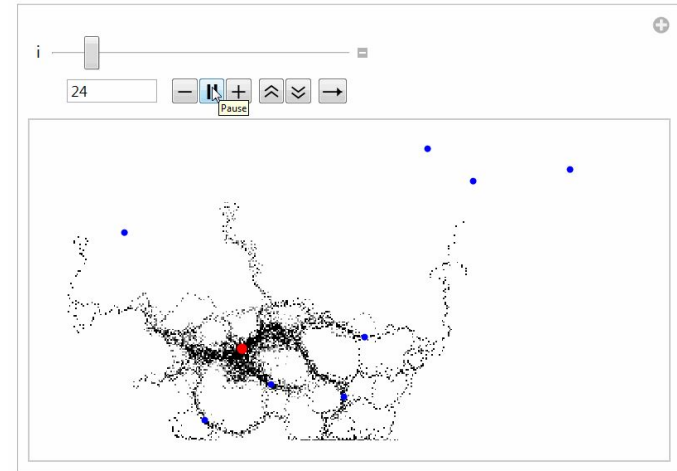
Simulated Annealing

- Choose a neighboring test case.
 - If better, select it. If not, select it at probability:
$$\text{prob}(\text{score}, \text{newScore}, \text{time}, \text{temp}) = e^{((\text{score} - \text{newScore}) * (\text{time} / \text{temp}))}$$
 - Governed by temperature function:
$$\text{temp}(\text{time}, \text{maxTime}) = (\text{maxTime} - \text{time}) / \text{maxTime}$$
- Initially, large jumps around search space.
 - Stabilizes over time.



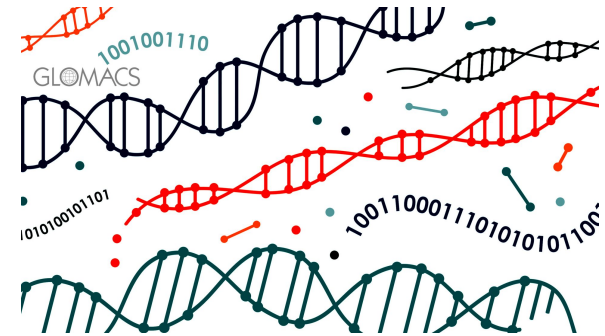
Global Search

- 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.

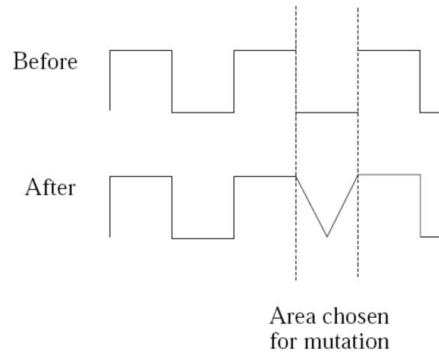


Genetic Algorithms

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

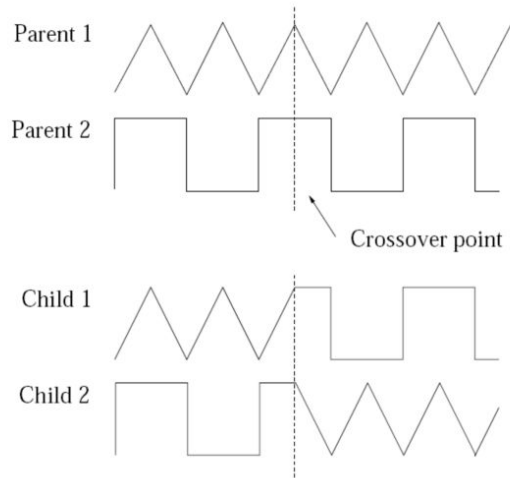


Genetic Algorithms - Mutation



- Copy a high-scoring solution.
- Impose a small change.
 - (add/delete/modify a function call, change an input value)
 - Follow the rules for determining the neighbors of a test.
 - Choose a neighbor from that set.

Genetic Algorithms - Crossover



- By “breeding” two good tests, we may produce better tests.
- Form two new solutions.
 - Sample from probability distribution to decide which parent to inherit from.

Genetic Algorithms - Crossover

- One Point Crossover
 - Splice at crossover point.
- Uniform Crossover
 - Flip coin at each line, second child gets other option.
- Discrete Recombination
 - Flip coin at each line for both children.

A	B	C	D
1	2	3	4

A	B	3	4
1	2	C	D

A	B	C	D
1	2	3	4

A	2	3	D
1	B	C	4

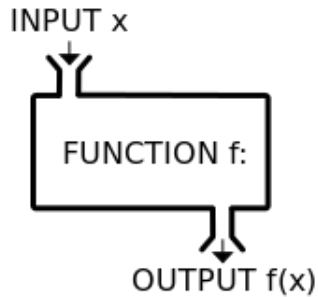
A	B	C	D
1	2	3	4

A	2	C	4
A	B	3	4

Let's take a break.

Fitness Functions

- Domain-based scoring functions that determine how good a potential solution is.
 - Should offer feedback:
 - Percentage of goal attained.
 - *Better - information on how to improve solution.*
 - **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 reach branching point (i.e., if-statement) and execute all possible outcomes.
- **Fitness function (Attempt 1):**
 - Measure coverage and try to maximize % covered.
 - **Good:** Measurable indicator of progress.
 - **Bad:** No information on how to improve coverage.

Example - Code Coverage

- Attempt 2: 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 $x \neq 10$ evaluates to false, branch distance = $(\text{abs}(x-10)+k)$.
- 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**)

What are your testing goals?

(and would they make good fitness functions?)

What Do I Do With These Inputs?

- If looking for crashes, just run generated input.
- If you need to judge correctness, add assertions.
 - General properties, not specific output.
 - **No:** `assertEquals(output, 2)`
 - **Yes:** `assertTrue(output % 2 == 0)`



Generating Covering Arrays for Combinatorial Interaction Testing

CIT

Allow Content to Load	Notify About Pop-Ups	Allow Cookies	Warn About Add-Ons	Warn About Attack Sites	Warn About Forgeries
Allow	Yes	Allow	Yes	Yes	Yes
Restrict	No	Restrict	No	No	No
Block		Block			

- Instead of testing all combinations, test all 2-way interactions.
- **Covering Array:** A set of test specifications that covers all pairs of values.
 - From 144 specifications to 9
- Generating **smallest** covering array NP-hard.
- Metaheuristic search can easily generate near-smallest covering array.

Allow Content	Allow Cookies	Pop-Ups	Add-Ons	Attacks	Forgeries
Allow	Allow	Yes	Yes	Yes	Yes
Allow	Restrict	No	No	-	No
Allow	Block	No	No	No	Yes
Restrict	Allow	Yes	No	No	No
Restrict	Restrict	Yes	-	-	Yes
Restrict	Block	No	Yes	Yes	No
Block	Allow	No	-	-	Yes
Block	Restrict	-	Yes	No	-
Block	Block	Yes	No	Yes	No

Generating Covering Arrays

1. Generating Random Solutions
2. Calculating Solution Fitness
3. Evolving Solutions
 - a. Mutation (Genetic Algorithm) / Neighboring Solution (Local Search)
 - b. Crossover (Genetic Algorithm)

Generating Random Solution

1. Calculate list of pairs to cover.
2. Until list is empty:
 - a. Generate random test specification.
 - b. Remove covered pairs from list.
 - c. Add specification to covering array.
3. Return covering array.

(Content = Allow, Pop-Ups = Yes)
 (Content = Allow, Pop-Ups = No)
 (Content = Restrict, Pop-Ups = Yes)
 ...

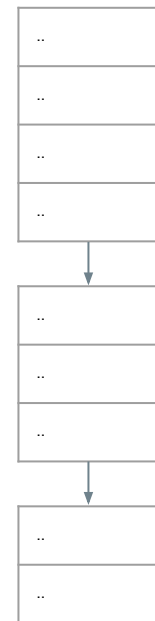
Allow	No	Block	No	Yes	Yes
-------	----	-------	----	-----	-----

(Content = Allow, Pop-Ups = Yes)
~~(Content = Allow, Pop-Ups = No)~~
 (Content = Restrict, Pop-Ups = Yes)
 ...

..
Allow	No	Block	No	Yes	Yes

Fitness Function

- Size of the covering array.
 - `coveringArray.length()`;
 - Want to minimize the score (smaller arrays are better)
- Can be measured, fast calculation.
- Tells us which solutions are better.
- Does not offer detailed feedback, but still works.

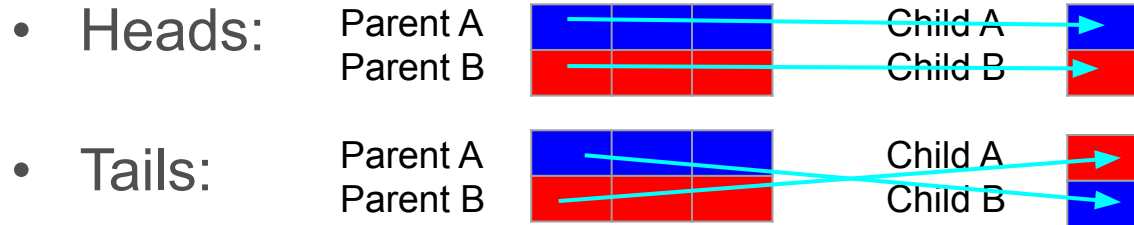


Mutation

- Change a value in a test specification.
 - Set a limit on number of changes made at one time.
 - Maybe we can make it smaller with a few changes?
- If all pairs covered in fewer tests, discard remainder.
- If no longer a covering array:
 - Throw out solution OR
 - Revert change and try again OR
 - Revert change and mutate different solution.

Crossover

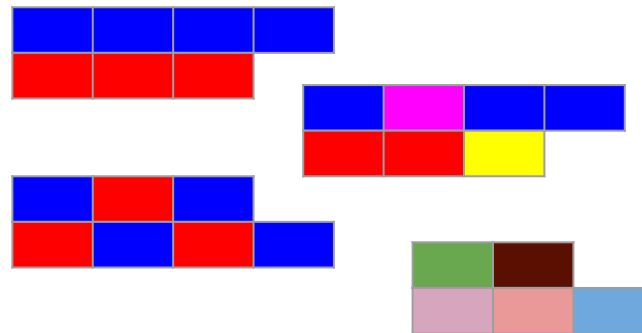
- Take two covering arrays, create two “child” arrays.
- For each specification, flip a coin:



- Check each child:
 - If not a covering array, discard.
 - If still a covering array, remove redundant specifications.

Search Process

- Generate population at random.
- Score each by size.
- Create new population.
 - Retain best arrays (10%)
 - Create mutations (30%)
 - Create children (30%)
 - Generate random (30%)
- Repeat until budget expires.



Not Just Test Generation...

Can be applied to any problem with:

- Large search space.
- Fitness function and solution generation with low computational complexity.
- Approximate continuity in fitness function scoring.
- No known optimal solution.

Automated Program Repair

- Produce patches for common bug types.
- Many bugs can be fixed with just a few changes to the source code - inserting new code, and deleting or moving existing code.
 - Add null values check.
 - Change conditional expression.
 - Move a line within a try-catch block.

Generate and Validate

- **Genetic programming** - solutions represent sequences of edits to the source code.
- **Generate and validate approach:**
 - Fitness function: how many tests pass?
 - Patches that pass more tests create new population:
 - Mutation: Change one edit into another.
 - Crossover: Merge edits from two parent patches.

GenProg Results

- Repaired 55/105 bugs at average \$8 per bug.
 - Projects with over 5 million lines of code
 - Supported by 10000 test cases.
- Patch infinite loops, segmentation faults, buffer overflows, denial of service vulnerabilities, “wrong output” faults, and more.

Risks of Automation

- Structural coverage is important.
 - Unless we execute a statement, we're unlikely to detect a fault in that statement.
- More important: how we execute the code.
 - Humans incorporate context from a project.
 - "Context" is difficult for automation to derive.
 - One-size-fits-all approaches.

Limitations of Automation

- Automation produces different tests than humans.
 - “shortest-path” approach to attaining coverage.
 - Apply input different from what humans would try.
 - Execute sequences of calls that a human might not try.
- Automation **can be** very effective, but more work is needed to improve it.

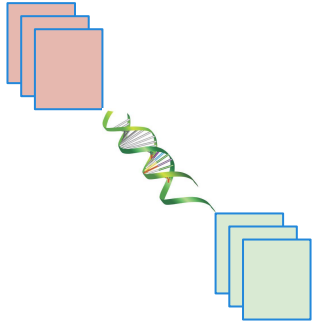
I Want to Try This Out!

- Fuzzing often based on metaheuristic search.
 - AFL (American Fuzzy Lop), Google OSS-Fuzz use genetic algorithms, fitness = code coverage.
 - <http://lcamtuf.coredump.cx/afl/>
 - <https://google.github.io/oss-fuzz>
 - system-level tests
 - The Fuzzing Book has tutorials and code for many specialized approaches:
 - <https://www.fuzzingbook.org/>

I Want to Try This Out!

- Python:
 - Tutorial for beginners:
<https://greg4cr.github.io/pdf/21ai4se.pdf>
 - <https://github.com/Greg4cr/PythonUnitTestGeneration>
- EvoSuite for Java: <http://www.evosuite.org/>
- Sapienz (Facebook) tests Android/iOS apps
 - Will be open-source in ~~end of 2020~~ 2022?.
 - Older version available
 - <https://github.com/Rhapsod/sapienz/>

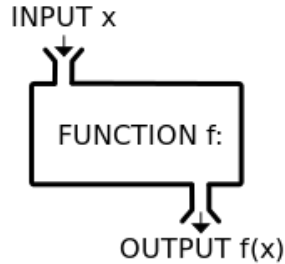
Summary



The Metaheuristic (Algorithm)

Genetic Algorithm
Simulated Annealing
Hill Climber
(...)

+



=



The Fitness Functions (Feedback Strategies)

Distance to Coverage Goals
Count of Executions Thrown
Input or Output Diversity
(...)

(Goals)

Cause Crashes
Cover Code Structure,
Maximize Battery Use,
(...)

Next Time

- User Interface design for variable systems
- Assignment 4 - Due Sunday
- Assignment 5 - December 19
 - Final assignment - on testing
 - Questions?



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY