Automated Test Case Generation:

Metaheuristic Search

CSCE 747 - Lecture 21 - 03/29/2016

Testing as a Search Problem

- Do you have a **goal** in mind when testing?
- Can that goal be **measured**?
- Then you are **searching** for a test suite that achieves that goal.
  - Out of the near-infinite set of inputs, I would like a set of inputs that…
    - obey those properties.
    - cover all branches.
    - try all 2-way pairs of representative values.
    - (etc)

Testing as a Search Problem

- "I want to find all faults" cannot be checked.
- However, almost all testing goals can.
  - Boolean: Property Satisfied/Not Satisfied
  - Numeric: % Coverage Obtained
- If we can take a candidate solution and check whether it meets our goal, then computers can search for a solution.
- Many search techniques for automated test case generation.

Search Process

- Choose a solution. If it does not accomplish the goal, try another.
- Keep trying new solutions until goal is achieved or all solutions are tried.
- The order that solutions are tried is 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.

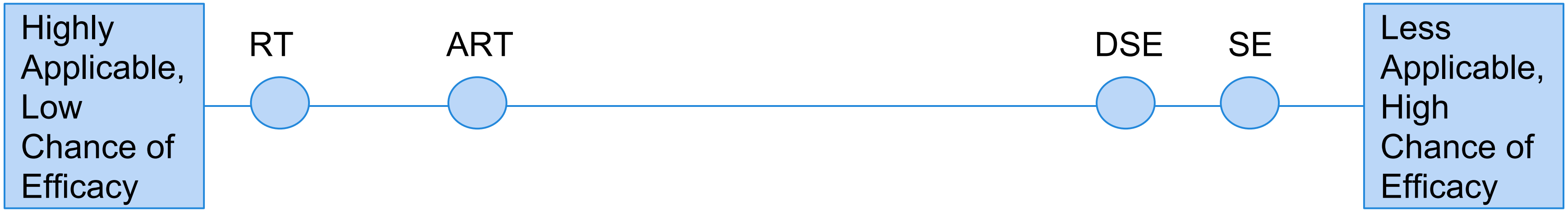
Search Budget

- Exhaustive Search - try all solutions.
- Most software has near-infinite number of inputs. We generally cannot try all solutions without constraining the problem.
- Search can be bound by a **search budget**.
  - Number of attempts made.
  - Time allotted to the search.
- **Optimization** problem:
  - Search for the best solution possible given the search budget.

Search Heuristics

- Simple strategy: randomly generate input.
    - Fast, easy to understand, very bad at finding faults.
- Adaptive random testing applies strategies to control the distribution of random test generation.
    - Retains benefits of RT, more likely to find faults.
- Dynamic symbolic execution extracts logical expressions describing program paths, and generates input from those expressions.

Search Heuristics


Highly Applicable, Low Chance of Efficacy
RT
ART
DSE
SE
Less Applicable, High Chance of Efficacy

- Random Testing:
  - Very fast, easy to implement, requires no information about the system.
  - Unlikely to satisfy goals.
- Adaptive Random Testing:
  - Almost as efficient as RT, same other benefits, far more likely to satisfy goals. Still based on random chance.
- (Dynamic) Symbolic Execution:
  - Will find an exact solution if possible.
  - Many restrictions on complexity and data structures of the programs supported.

Optimization Problem

- Often too many restrictions to make exhaustive search feasible.
- No way to try all inputs or abstract complex systems. Instead, need a strategy to sample from the input space.
  - But not in a purely random manner.
- How can we find the best solution possible given a limited search budget?
  - Can apply optimization algorithms.
  - Called **metaheuristic search techniques**.

Metaheuristic Search

Optimization Problem

- If we can calculate a score related to attainment of a testing goal, then we have an **optimization target**.
- Test generation as an optimization problem:
  - Generate a test (or set of tests).
  - Score each of them using a **fitness function**.
  - Manipulate the solution according to a search strategy (the "**metaheuristic**").

Metaheuristic Search

- Choose a smart strategy to sample from the search space.
  - Not purely random - fitness function guides the search towards better solutions.
  - The metaheuristic changes its approach based on past attempts.
- No guarantee of an optimal solution…
  - … but if we're smart, we'll hit something close enough.
- Computationally feasible, and often more effective than random search.

Local Search

- Generate a potential solution.
- Score it using your fitness function.
- Attempt to improve it by looking at its **local neighborhood**.
  - Test cases minorly different from the current choice.
  - Keep making small, incremental improvements.
- Very fast and efficient if you make a good initial guess.
- Can get stuck in local maxima if not.
  - Reset strategies help.

Generating Neighbors

- “Neighbors” are tests created by making a small change to the current test.
- Single method call:
  - Switch value of boolean, other values from an enumerated set, bounded range of numeric choices.
- Multiple method calls:
  - Insert a new method call.
  - Delete or replace an existing call.
    - Can replace by changing the method called or the parameters.
- Important to control size of a neighborhood.

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.
- Strategies:
  - Steepest Ascent - examine all neighbors, take the one with the highest improvement.
  - Random Ascent - examine neighbors at random, and choose the first to show any improvement.

Simulated Annealing

- Choose a neighboring test case.
  - If it is a better solution, select it.
  - If not, select it at probability:
    prob(score, newScore, time, temp) = $e^{((score - newScore) * (time / temp))}$
  - Governed by temperature function:
    temp(time, maxTime) = (maxTime - time) / maxTime
- Repeat until search budget expires and return best solution.
- Initially, large random jumps around the search space. Over time, search stabilizes.

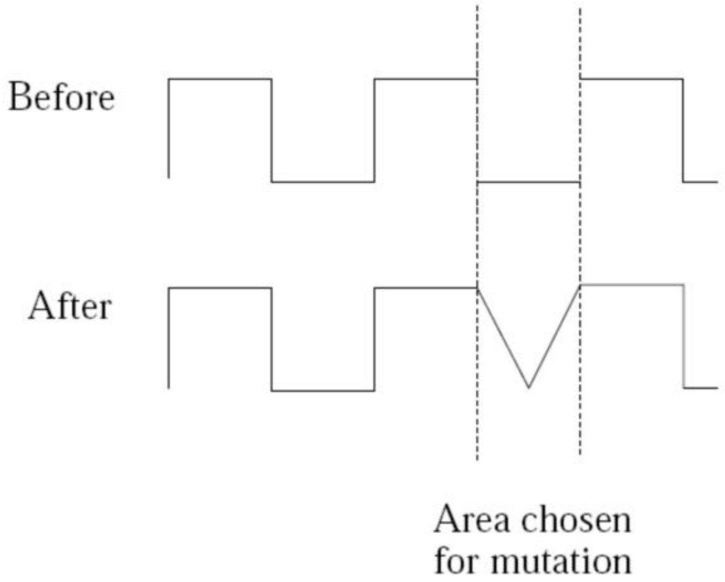
Global Search

- Generate a set of solutions.
- Score them.
- At a certain probability, sample from other regions of the space.
- Strategies typically based on natural processes - swarm attack patterns, ant colony behavior, species evolution.
  - Models of how populations interact and change.

Genetic Algorithms

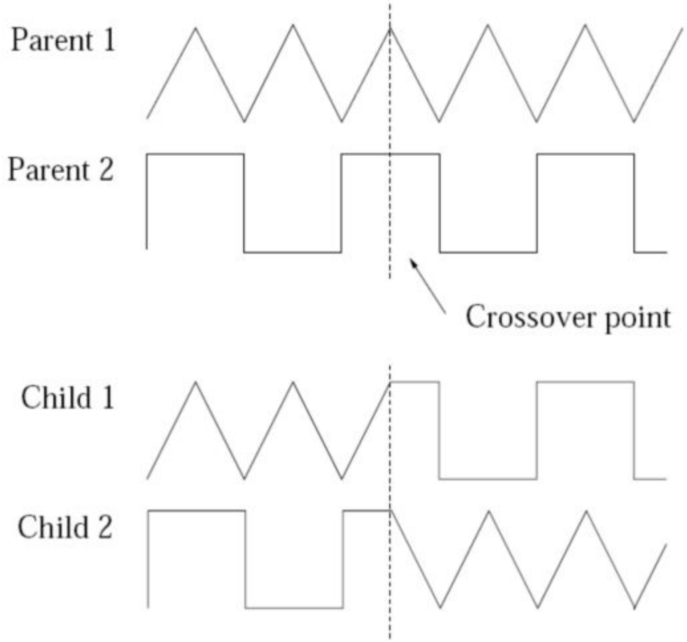
- Over multiple generations, evolve a population - favoring good solutions and filtering out bad solutions.
- Diversity is introduced to the population each generation by:
    - Keeping some of the best solutions.
    - Randomly generating some population members.
    - Creating “offspring” through mutation and gene crossover.

Genetic Algorithms - Mutation



- Create a copy of a high-scoring test.
- Impose a small change to that test.
  - Follow the rules for determining the neighbors of a test.
  - Choose a mutation from that set.
- A good test could be improved by checking one of its neighbors.

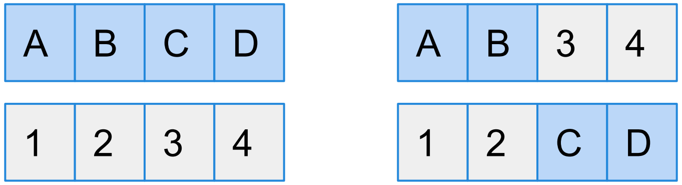
Genetic Algorithms - Crossover



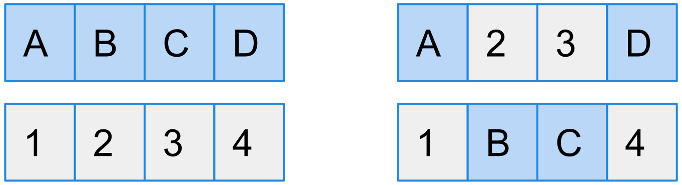
- Take two high-scoring tests, and attempt to combine aspects of them into one.
- Choose one element, sample from a probability distribution to decide which parent to inherit from.
- By combining features from two good tests, we may produce a better test.

Genetic Algorithms - Crossover

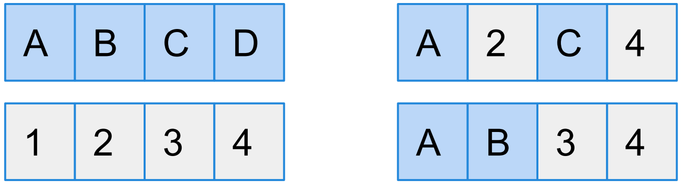
- One Point Crossover
  - Splice at a randomly chosen crossover point.



- Uniform Crossover
  - Each point is a potential crossover point.



- Discrete Recombination
  - Instead of sampling once per index for both children, it is done for for every child.



Particle Swarm Optimization

- A swarm of agents each attempt to search for good test cases.
- When another agent finds a better solution than the best known "worldwide", they tell everybody.
- Each agent mutates their solution based on their knowledge of the best local solution and the best global solution.
- Over time, the agents converge on the best solutions.

Particle Swarm Optimization

- Each agent *i* has velocity v_i and position p_i.
    - Position: Their current solution.
    - Velocity: The amount of change to be made to the solution.
        - Bound by a maximum velocity.
    - Vectors along all dimensions in the solution.
        - (i.e., method parameters)
- Each round, velocity and position are updated based on current local and global knowledge.

Particle Swarm Optimization

- Update Rules:
  - $v_i^d = \omega v_i^d + \alpha\beta(best^g - p_i^d) + \gamma\delta(best^l - p_i^d)$
    - ω is an inertial weight.
      - $\omega = \omega_{max} - (\omega_{max} - \omega_{min})$ time / maxTime
      - Decreases linearly over time
    - α and γ are user-set acceleration coefficients.
    - β and δ are random numbers
    - $best^g$ is the global best score. $best^l$ is the local best score.
      - Guide the velocity and position of the agent.
  - $p_i^d = p_i^d + v_i^d$

Fitness Functions

- Solutions are judged by a “fitness function” that takes in the solution and calculates a score.
  - Distance from the current solution to the “ideal” solution.
    - How close are you to covering a testing goal?
  - Smaller scores are typically better.
  - Must offer information to guide the search.
  - Must be cheap to calculate - performed 100s-1000s of times per generation.

Structural Coverage

- Normally measured as proportion of test obligations covered to total obligations.
- This serves as a score - how good are current testing efforts.
- However, this is not an ideal fitness function.
  - Does not inform the search process.
  - Instead - can we score a test such that we can learn from the attempt?
    - Not just “is this good”, but “how close is this to ideal?”

Branch Coverage Fitness Function

- Instead of raw coverage, use the branch distance and approach level:

  fitness(s,b) = AL(s,b) + normalize(BD(s,b))

- Approach level - count of the branch's control-dependent nodes not yet executed.
- Branch distance - if the other branch is taken, measure how close the target branch was from being taken.

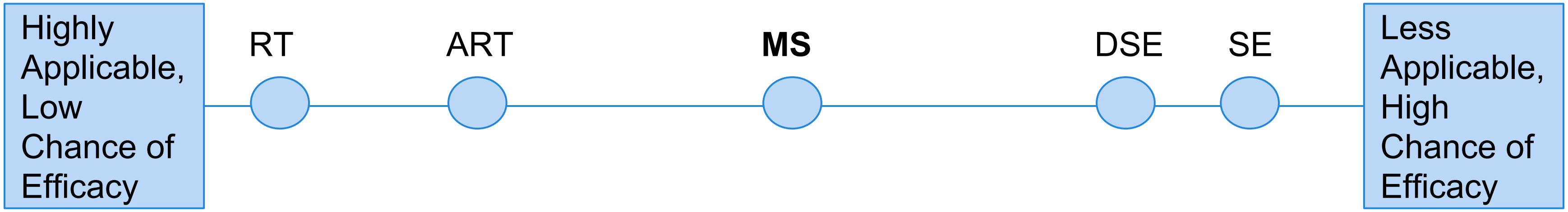
Branch Coverage Fitness Function

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

- Goal, true branch of Node 2.
- If x==10 evaluates to false, branch distance = (abs(x-10)+k).
- Closer x is to 10, closer the branch distance.

evosuite demonstration

Comparing Approaches



Highly Applicable, Low Chance of Efficacy

RT

ART

MS

DSE

SE

Less Applicable, High Chance of Efficacy

- Less efficient than ART.
  - But more likely to achieve the testing goal.
- Fewer restrictions than DSE.
  - But no guarantee of optimality.
- Choice of fitness function is important.
  - Must be fast to calculate.
  - Must quickly converge on optimal solutions.

Combining Approaches

```
class Foo {
    int x = 0;
    void inc(){
        x++;
    }
    int getX(){
        return x;
    }
}
```

```
class Bar{
    String x;
    Bar(String x){
        this.x = x;
    }
    void coverMe(Foo f){
      String y = x + f.getX();
      if(y.equals(“baz5”))
        // target
    }
}
```

- MS can achieve high coverage, but will not guess “baz”.
- DSE can identify “baz”, but will not call Foo.inc() five times.
- By combining the two, the target can be covered.

Not Just Test Generation...

Metaheuristic search can be applied to any problem with:

- A large search space.
- Fitness function and solution generation methods with low computational complexity.
- Approximate continuity in the fitness function.
- No known optimal solution.

Automated Program Repair

- Popular projects may have hundreds of bugs reported *per day*.
- Repair techniques, like GenProg, automatically produce patches that can repair common bug types.
- Many bugs can be fixed with just a few changes to the source code - inserting new code, deleting or moving existing code.
- GenProg uses the same ideas to *search* for repairs automatically.

GenProg

- **Genetic programming** - solutions represent sequences of edits to the source code.
- Each candidate patch is applied to the program to produce a new program.
- See if the patched program passes all tests.
  - Fitness function: how many tests pass?
- Use crossover and mutation to evolve better patches.

GenProg Results

- GenProg repaired 55 out of 105 bugs at an average cost of $8 per bug.
    - Large projects - over 5 million lines of code, 10000 test cases.
- Able to patch infinite loops, segmentation faults, buffer overflows, denial of service vulnerabilities, “wrong output” faults, and more.

Automated Code Transplantation

- Not just patches…
- Many coding tasks involve “reinventing the wheel” - redesigning and writing code to perform a function that already exists in some other project.
- What if we could slice out that code (“organ”) from a “donor” program and transplant it to the right “vein” in the target software?

muScalpel

- Uses a form of genetic programming.
- Initial population of 1 statement patches.
  - Organs need very few statements from the donor.
  - Starting with one line at a time allows muScalpel to find efficient solutions quickly.
- Search evolves both organs and veins.
  - Optimize the set of code transplanted from the donor, and the optimal location to place that code in the target software.
- Apply tests to ensure correctness of both original code and new features.

muScalpel Results

- Transplantation of H.264 video codec from x264 system to VLC media player.
  - Took VLC developers 20 days to write the code manually.
  - Took muScalpal 26 hours to transplant automatically.
- In 12 of 15 experiments, successful transplants that passed all tests.

We Have Learned

- ART is often ineffective, and DSE is limited in the scope of the programs it can be applied to.
- Metaheuristic search strikes middle ground:
  - Less efficient than ART, but often more effective.
  - Able to generate tests for programs that DSE cannot address.
- Smart strategies for sampling from a search space. Designed to produce near-optimal solutions within a limited search budget.

We Have Learned

- Local methods attempt to make small changes to a current solution.
    - Hill climbers, simulated annealing.
- Global methods try solutions from all around the search space.
    - Genetic algorithms, particle swarm optimization.
- Can also be used to automate patching and feature transplantation.

Next Time

- Software Inspections
  - Ch. 18

- Homework:
  - Assignment 4 - due April 5!