Automated Test Case Generation

CSCE 747 - Lecture 24 - 04/06/2017

Software Testing

- Fundamentally: Try input, see what happens.
- We do not just try *any* input.
- We create test cases to:
  - Find faults
  - Obtain coverage over structural elements
  - Cover combinations of representative values
  - Execute DU pairs
  - Model important use scenarios
  - Establish reliability measurements
  - **What do all of these have in common?**

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.

Graph Search Heuristic Examples

- Arrange nodes into a hierarchy.
  - Breadth-first search explores a graph by trying all nodes one level deeper than the current node.
  - Depth-first search explores until backtracking must occur.
  - Naive, but easy to understand and implement.
- Attempt to estimate shortest path.
  - A* search finds a path through a graph by
    - calculating the distance travelled
    - examining all “next moves” and estimating which will get them closest to the goal.
  - Requires domain-specific scoring function.

Search Budget

- Default: all solutions will be attempted.
- 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.
- Search techniques try to find a solution before the budget expires

Search Budget

- T/F goals difficult to solve under a budget.
  - May return “unknown” if satisfying solution not found
  - Or, problem must be constrained to be exhaustively solvable.
- Measurable goals transform a search into an **optimization** problem.
  - Partial solutions can be returned:
    - “This test suite obtained the highest branch coverage.”
  - Search for the best solution possible given the search budget.

Optimization

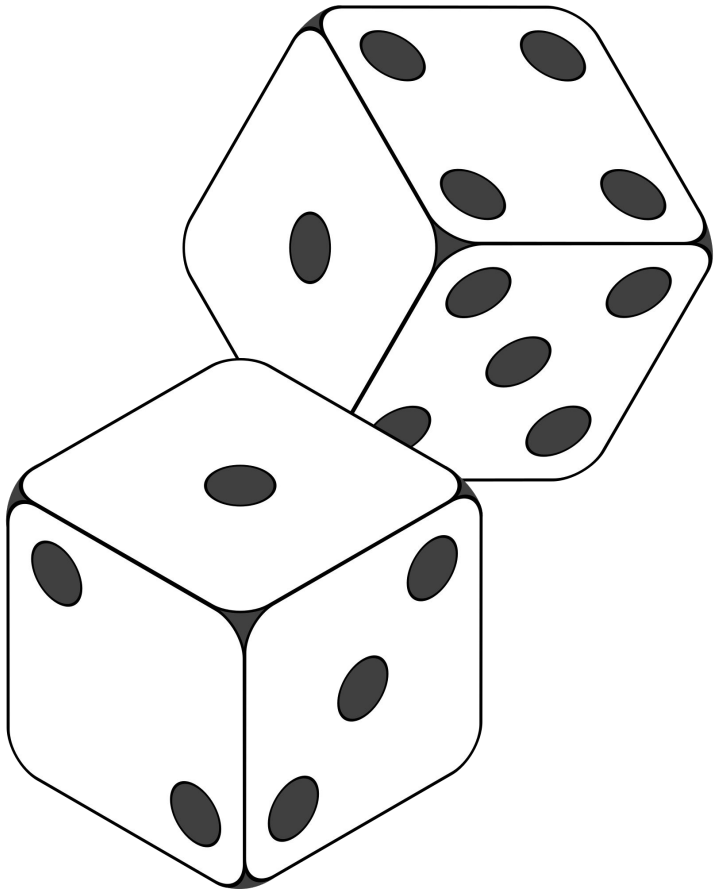
- Search for the best solution possible given the search budget.
- The search heuristic becomes important.
  - If time bound, time to create, execute, and evaluate a solution is important.
  - If attempt bound, the strategy used to choose solutions is important.
  - In practice, efficiency in both categories is desired.
- Many search strategies are possible.

Random Search

- Randomly formulate a solution.
  - Choose a class in the system.
  - Generate a series of method calls to that class.
    - Random number of calls, range 1-(max test case size)
    - Methods to call chosen randomly from list of methods.
    - Method parameters generated randomly.
      - Random integer, random string, etc.
- If it doesn't work, try another solution. Keep trying until the goal is attained or budget expires.

Random Testing

- Very popular test generation method.
  - Extremely fast.
  - Requires no planning.
  - Easy to implement.
  - Easy to understand.
  - All inputs considered equal, so no designer bias.



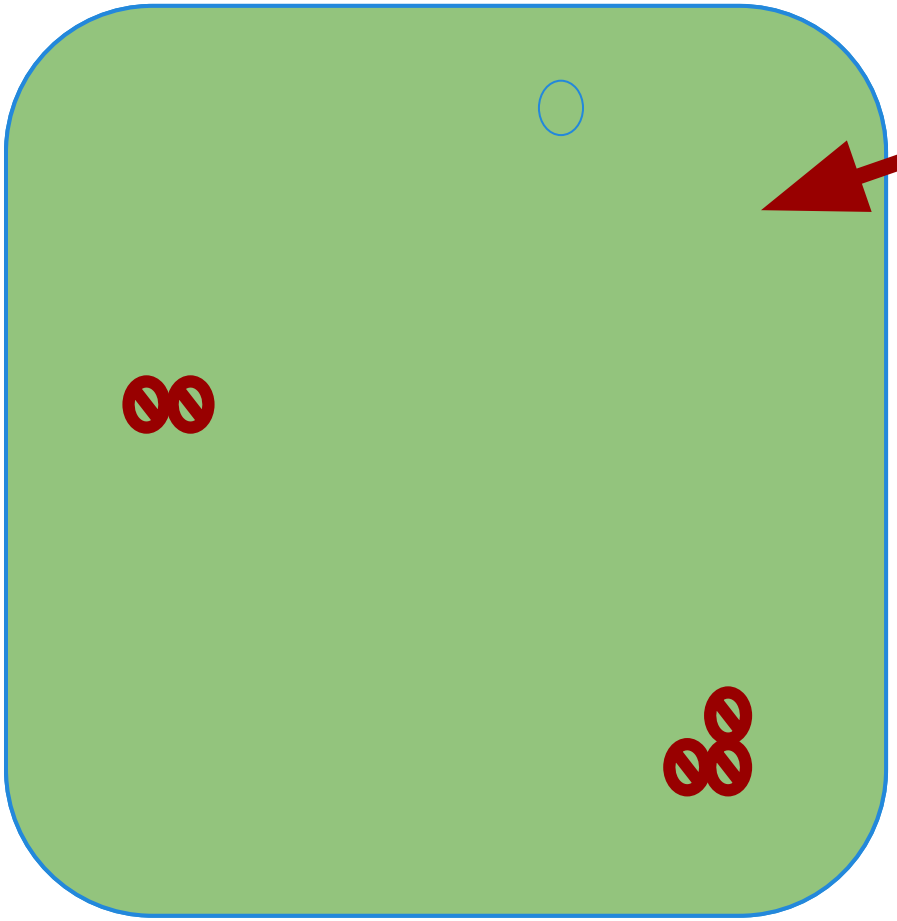
Why Not Random?

Range of Search Techniques

- Adaptive Random Testing
  - Ensuring an even distribution of randomly-chosen inputs across the search space.
- (Dynamic) Symbolic Execution
  - Combining targeted exhaustive search with a random search to generate test cases.
- Metaheuristic Search
  - Using optimization techniques to intelligently converge on effective test cases.

Adaptive Random Testing

The Input Space

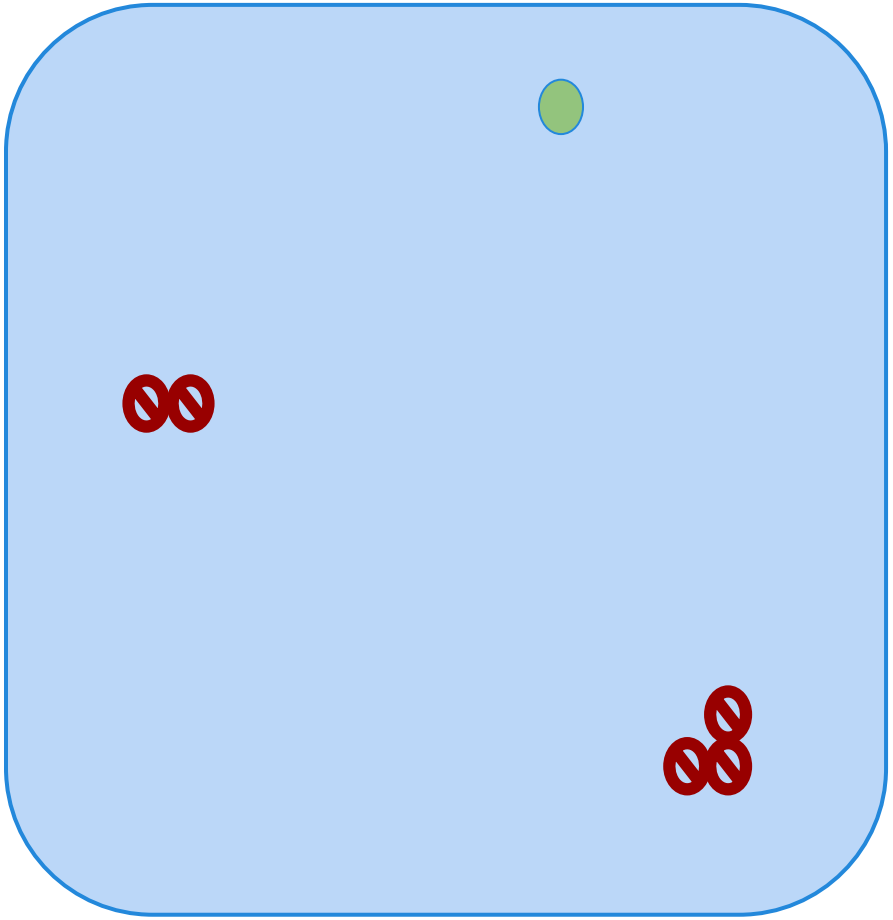


Some tests will pass when executed. Others will fail.

Faults are sparse in the space of all inputs, but dense in some parts of the space where they appear.

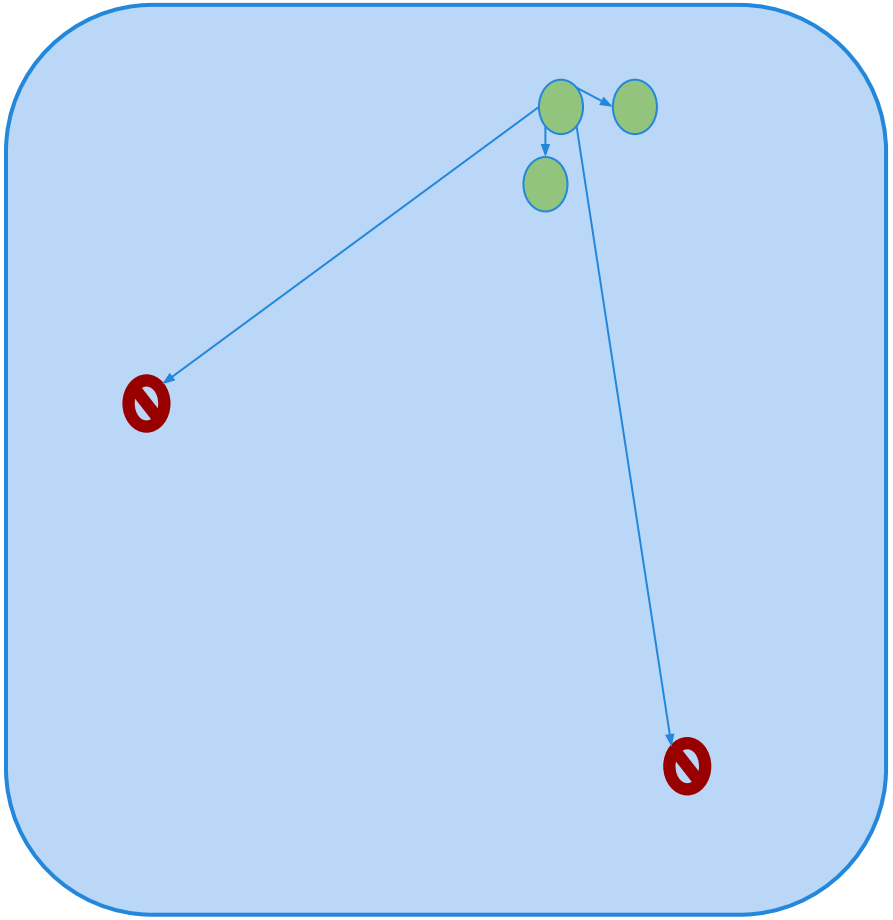
If an input causes a failure, a similar input is likely to also cause a failure.

Choosing Tests



- Failing inputs form small contiguous regions in the input space.
  - If a test fails, similar input is also likely to fail.
- Similar effect for most test goals.
  - If a test covers a branch, similar input is also likely to cover that branch.
- We want to find the region that meets our goal.

Adaptive Random Tests



- Rather than choosing tests completely at random, favor **input diversity**.
  - If a test does not meet a goal, new input should be very different from used input.
- Basis of **Adaptive Random Testing.**

FSCS-ART Algorithm

- **Fixed Size Candidate Set** algorithm.
- Makes use of two sets of test cases:
  - Executed Set - the tests already executed that did not meet our goal.
  - Candidate Set - a set of random tests that have been generated, but not yet tried.
- Initially, generate a test and execute it.
- Then, generate N candidate tests.
- Choose the test *furthest* from the executed test, and execute it. Add that to the set of executed tests.

Performance vs Random Testing

- FSCS-ART generates many more tests than are executed.
  - In RT, all tests generated are executed.
  - Execution is more time consuming than generation.
- In theory, ART should find a solution faster than pure random testing.
  - ART finds solutions in 50-60% of the time of RT.
  - More tests generated per round, the faster goals are attained. Original authors found n=10 to offer best effectiveness vs performance ratio.

Choosing a Candidate Test

- Executed set T = {t1,t2,...,tm}, Candidate set C = {c1,c2,...,cn}, $C \cap T = \varnothing$
  - **Maxi-min method:** We must choose candidate ci such that for all $j \in \{1,2,...,n\}$, $\min_{i=1}^{m} dist(ci,ti) >= \min_{i=1}^{m} dist(cj,ti)$
  - **Maxi-sum method:** We must choose candidate ci such that for all $j \in \{1,2,...,n\}$, $sum_{i=1}^{m} dist(ci,ti) >= sum_{i=1}^{m} dist(cj,ti)$
  - Inputs near boundary of domain are more likely to be chosen with maxi-min.

Distance Functions

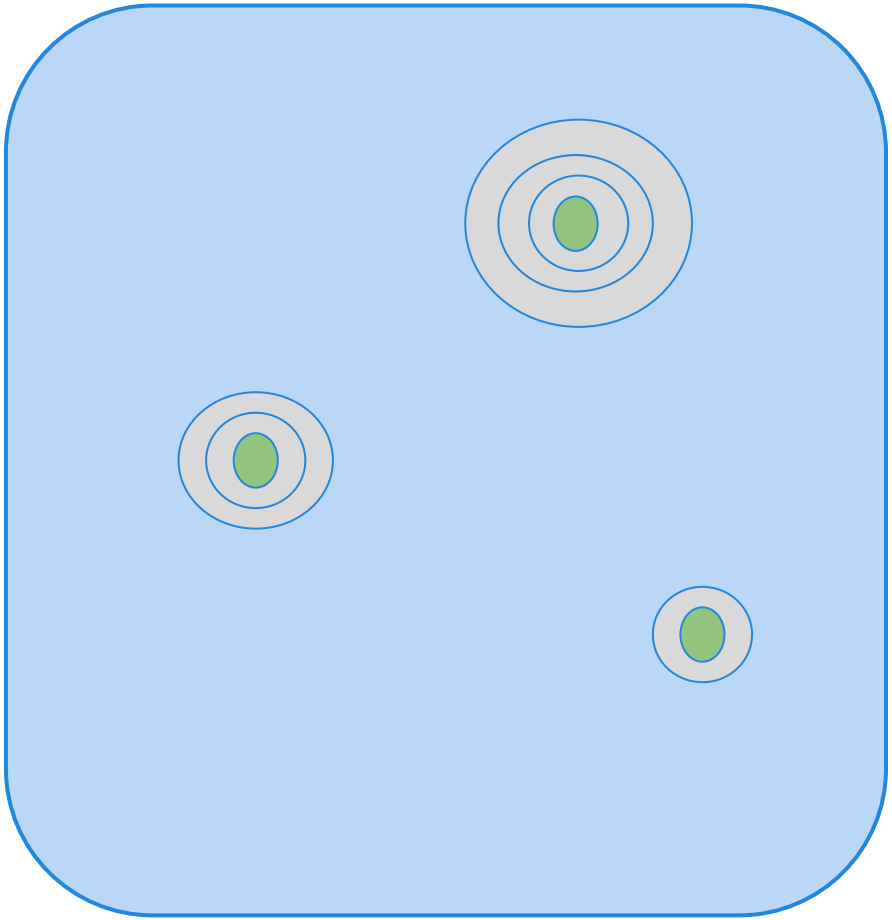
- Important to choose a reliable distance function for your input type.
- For numeric values, Euclidean Distance
  - Compare two vectors of numeric variables, a = (a1, a2, ... , am), b = (b1, b2, … , bm), dist(a,b) = root($\text{sum}_{i=1}^{m}$ $(ai - bi)^2$)
- For strings, Levenshtein Distance
  - $$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Exclusion Zones

- Introduce **exclusion zones** - “bubbles” around tests that do not meet our goals.
  - Circular zone around that input.
- No inputs may be chosen from those zones.
- Does not require input to be “furthest” away, but ensures a minimum level of diversity in random choices.

Exclusion Zones

- Initially - exclusion zone is set to a user-defined size *N*.
- All exclusion zones are of equal size.
- Zones shrink as more tests are executed.
  - If there are m zones, each will have an area of N/m, and radius root(N/(mπ))



Dynamic Partitioning

- Break input space into regions, and chooses a test from the largest unexplored region.
  - Generate a random test t.
  - Divide the input space into two regions, with one containing t.
  - Choose the largest untested region and generate a test in that region.
  - Each time a test is executed, shrink that region, then generate a test from the largest remaining region.
- Helps ensure even spread across the input space.

(Dynamic)
Symbolic Execution

Symbolic Execution

- Process of building predicates that describe which execution paths will be taken and their effect on program state.
  - Determines the conditions under which a path can be taken.
  - Identifies infeasible paths and paths that can be taken when they shouldn’t.
  - Can be used to generate tests targeted at particular paths in the system.

Symbolic Execution

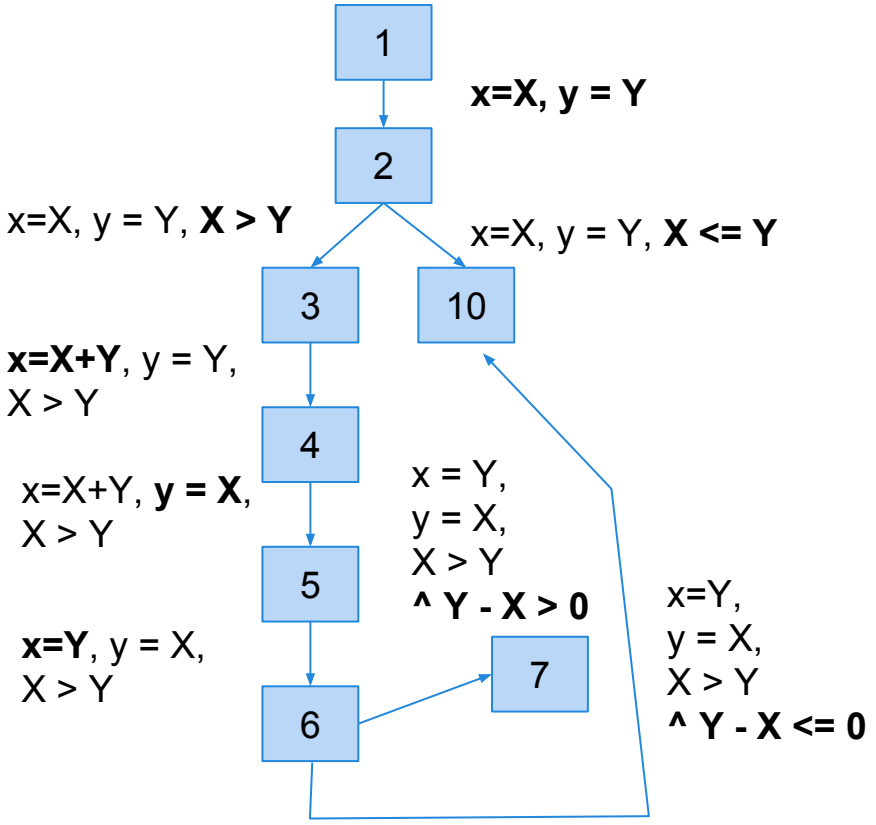
- Symbolic execution builds a set of **path constraints**.
  - Boolean formulas over the symbolic inputs.
  - Describes the constraints on the input variables that must be satisfied to arrive at a program location.
  - At each point, the path constraints are updated with any information that can inform the choice of input.
- If a path constraint is unsatisfiable, that path is infeasible.
- If it is satisfiable, any solution of that formula is a concrete test case.

Symbolic Execution Example

Three paths:

1. 1,2,10
   a. PC: **X <= Y**
2. 1,2,3,4,5,6,7
   a. PC: **X > Y ^ Y - X > 0**
3. 1,2,3,4,5,6,10
   a. PC: **X > Y ^ Y - X <= 0**

- Test case is obtained by finding a solution that satisfies the path constraint.
- Path 2: **x=2, y=1**
- Path 3: **Unsatisfiable**



Symbolic Execution

- Symbolic execution extracts a logical structure from the program.
  - Small, solvable mathematical model that can be exhaustively searched (in theory).
- The search problem:
  - Choose a path constraint.
  - Find values for the input variables that satisfy the path constraint.
- Exhaustive search, but limited in scope.

Symbolic Execution

- Exhaustive search, but limited in scope.
  - Can we find input that executes this path?
  - Boolean outcome - yes or no. No partial solutions.
    - Or, “unknown” if algorithm is unable to solve.
  - Tends to be used to achieve coverage.
  - Can be tied to particular testing goals by changing how path constraint is formulated.
    - i.e., MC/DC obligations
- Searching for a solution to a satisfiability modulo theories (SMT) problem.

Satisfiability Modulo Theories

- Searching for a solution to a satisfiability modulo theories (SMT) problem.
  - Generalization of Boolean Satisfiability (SAT)
- Express properties as conjunctive normal form expressions:
  - `f = (!x2 || x5) && (x1 || !x3 || x4) && (x4 || ! x5) && (x1|| x2)`
- SAT: variables are boolean. SMT: predicates.
  - Can include numeric expressions, as long as a model of decidability exists.

Search Based on SMT

- Express properties as conjunctive normal form expressions:
  - `f = (!x2 || x5) && (x1 || !x3 || x4) && (x4 || ! x5) && (x1|| x2)`
- Choose a variable and attempt to assign a value based on how it affects the CNF expression.
  - If we want `x2` to be false, choose a value that imposes that change.
- Continue until CNF expression is satisfied.

Branch & Bound Algorithm

- Set a variable to a particular value.
  - true, false, a numeric value.
- Apply that value to the CNF expression.
- See whether that value satisfies all of the clauses that it appears in.
  - If so, assign a value to the next variable.
  - If not, backtrack (bound) and apply the other value.
- Prune branches of the boolean decision tree as values are applies.

Branch & Bound Algorithm

```
f = (!x2 || x5) && (x1 || !x3 || x4) && (x4
|| ! x5) && (x1|| x2)
```

1. **Set x1 to false.**

   ```
   f = (!x2 || x5) && (0 || !x3 || x4) &&
   (x4 || ! x5) && (0 || x2)
   ```

2. **Set x2 to false.**

   ```
   f = (1 || x5) && (0 || !x3 || x4) && (x4
   || ! x5) && (0 || 0)
   ```

3. **Backtrack and set x2 to true.**

   ```
   f = (0 || x5) && (0 || !x3 || x4) && (x4
   || ! x5) && (0 || 1)
   ```

DPLL Algorithm

- Set a variable to a particular value
  - true, false, a numeric value
- Apply that value to the CNF expression.
- If the value satisfies a clause, that clause is removed from the formula.
- If the variable is negated, but does not satisfy a clause, then the variable is removed from that clause.
- Repeat until a solution is found.

DPLL Algorithm

```
f = (!x2 || x5) && (x1 || !x3 || x4) && (x4 || ! x5) && (x1|| x2)
```

1. **Set x2 to false.**

   ```
   f = (1 || x5) && (x1 || !x3 || x4) && (x4 || ! x5) && (x1|| 0)
   f = (x1 || !x3 || x4) && (x4 || ! x5) && (x1)
   ```

2. **Set x1 to true.**

   ```
   f = (1 || !x3 || x4) && (x4 || ! x5) && (1)
   f = (x4 || ! x5)
   ```

3. **Set x4 to false, then x5 to false.**

Limitations

- Path explosion - too many paths to build constraints for.
  - To handle infinite path situations, constraints must be discarded in favor of summaries of execution.
- Path complexity
  - Solvers are limited in the scope of the constraints they can solve.
    - Cannot solve expressions with nonlinear operations such as multiplication, division, sin(x).
    - Cannot solve for complex data structures such as trees or pointers.
      - Inputs cannot be from an infinite set.

Dynamic Symbolic Execution

- In parallel, execute symbolic and concrete executions.
  - In the concrete execution, log the results of each operation that can impact a path condition or the values of symbolic variables.
  - Choose random input for the initial concrete execution, and execute the program.
  - Symbolically re-execute the program on the path followed by the trace and generate path conditions.

Dynamic Symbolic Execution

- Negate the last constraint and solve the PC to generate new input.
  - If an individual predicate is unsolvable, substitute the concrete value from the trace.
  - If the PC is unsolvable with the negated constraint, negate the next constraint.
- Continue negating one constraint at a time to explore new paths.
  - Refine path conditions as new paths are taken.

DSE Example

```
typedef struct cell{
    int v;
    struct cell *next;
} cell;

int f(int v){
    return 2*v + 1;
}
```

```
textme(cell *p, int x){
    if(x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if(p->next == p)
                    ERROR;
    return 0;
}
```

p = P, p->v = V,
p->next = N, x = X

X > 0

X > 0 ^ P != NULL

X > 0 ^ P != NULL
^ ((2X+1) != V)

Negate the last predicate, and solve the PC:
(X > 0) ^ !(P = NULL) ^ **!((2X+1) != V)**
Generate new concrete input:
x = 1, p->v = 3, p->next = NULL

We Have Learned

- When we create test cases, we are usually searching for tests that fulfill a goal.
  - Such as code coverage.
- If we have a measurable goal, algorithms can perform a search process, automating the creation of test inputs.
- Searches can be exhaustive, or bound by a search budget.

We Have Learned

- 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.

Next Time

- Metaheuristic Search
  - Test Generation
  - Genetic Programming

- Homework:
  - Assignment 4 - due Tuesday!