Heuristics for Scalable Dynamic Test Generation

Jacob Burnim, Koushik Sen

OVERVIEW

- **Goal:** Automatically generate effective test cases for large software systems with concolic execution.
- **The Challenge**: Path Space Explosion.
  - Even medium-sized programs have 10^{100}+ paths.
  - Completely intractable to exhaustively test each path.
- **Revised Goals:**
  - Focus on branch coverage.
  - Attain better branch coverage with fewer tests.
- **Our Approach:** Try to explore only the "important" program paths.
- Propose three search strategies:
  - Control-flow directed search
  - Uniformly random path search
  - Random-branch search

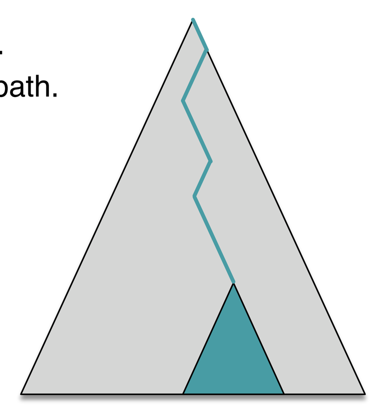


FIGURE 1 – Illustration of low coverage of traditional concolic testing on a huge path space.

COMPUTATION TREE OF A PROGRAM

- The **computation tree** is the tree of all feasible program executions.
- Each node corresponds to an execution of a conditional statement.
- Each edge a sequence of non-conditional statements executed between two successive conditional statements.
- The conditions along a path characterize the equivalence class of inputs that exercise the path.

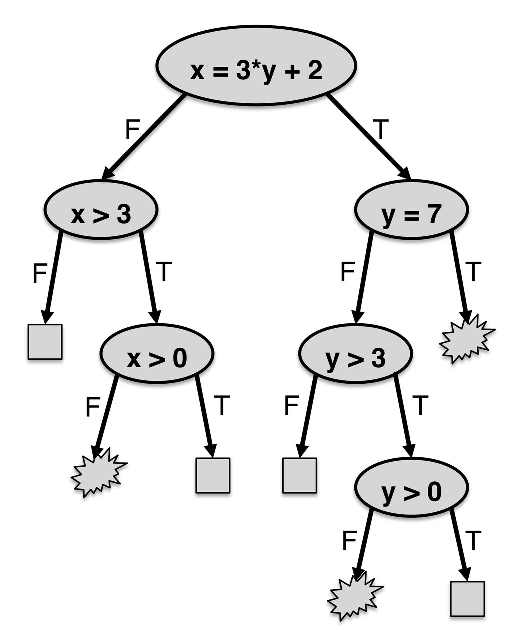
FIGURE 2 – A simple C program, which serves as our running example.

```
main(int x, int y) {
  int h = 3*y + 2;
  if (x == h)
    f(y);
  else
    g(x);
}

f(int a) {
  if (a == 7)
    ABORT;
  else
    g(a);
}

g(int z) {
  if (z > 3)
    assert(z > 0);
}
```

FIGURE 3 – Computation tree of the example program from Figure 2.



CONCOLIC TEST GENERATION

- **Concrete + symbolic execution**: technique to generate an input exercising each possible path through the program under test.
- **Basic Procedure:**
  - Given a concrete execution path, do symbolic execution along path.
    - Simplify using concrete values.
  - Yields **symbolic path constraint** -- a formula characterizing which inputs exercise the path. $x = 3y + 2 \wedge y \neq 7 \wedge y > 3$
  - Pick a conditional branch along path to negate.
    - Solve for inputs so branch is not taken. $x = 3y + 2 \wedge y \neq 7 \wedge y \leq 3$
  - Running on new inputs yields new concrete execution.
- Exhaustively explores every path in computation tree – traditionally with a depth-first search.
  - Given a path, always negate the deepest conditional not yet negated.

CONTROL-FLOW DIRECTED SEARCH

- **Key Idea:** Use static structure of test program to guide search.
  - Dynamic search with a static heuristic.
  - Search guided by knowledge of **unexplored** parts of path space.
- **Heuristic**: Explore paths statically "close" to uncovered branches.
  - Use distance in the control flow and static call graph (CFCG).
  - Negate conditionals with minimal distance to uncovered branches.
- **Search Strategy:**
  - Local search – operates on one current execution path at a time.
  - Given a current execution path:
    1. Negate conditional with minimal distance to an uncovered branch.
    2. Try to force program down all shortest static paths from negated conditional to an uncovered branch.
    3. If all paths are infeasible:
       - Increase our estimate of distance from the conditional to any uncovered branch.
       - Keep negating conditionals with minimal distances.
    4. Otherwise, we have an execution through an uncovered branch:
       - Update coverage and recompute estimated distances.
       - Restart the search on the new execution.
  - Can restart on new, random paths when progress slows.

CONTROL FLOW AND STATIC CALL GRAPH

- **Control flow and static call graph** (CFCG):
  - Control flow graph for each procedure.
  - **Call edge** from each call site to the called function.
  - No return edges back to call sites.
- **Static path:** a path through the CFCG
  - Can either skip over a function call or enter the called function and never leave.
  - Assumptions:
    - All functions return.
    - Paths in skipped functions need not be explored.
- Distance in CFCG captures difficulty of forcing execution down a static path
  - Edges leaving a conditional have weight one; others weight zero.
  - During search, negating any conditional requires one iteration.

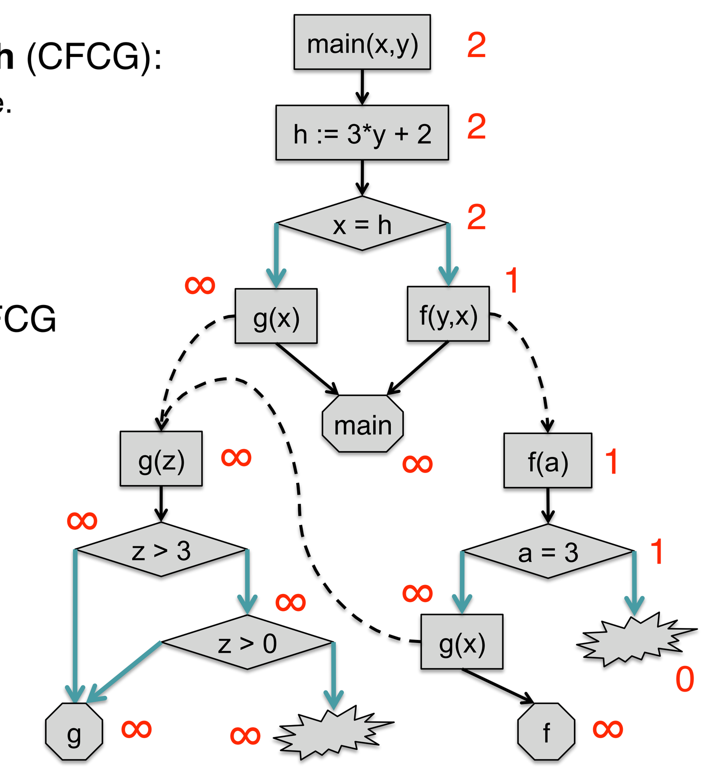


FIGURE 4 – Control flow and static call graph for the example program in Figure 2, with distances to the ABORT statement in function f.

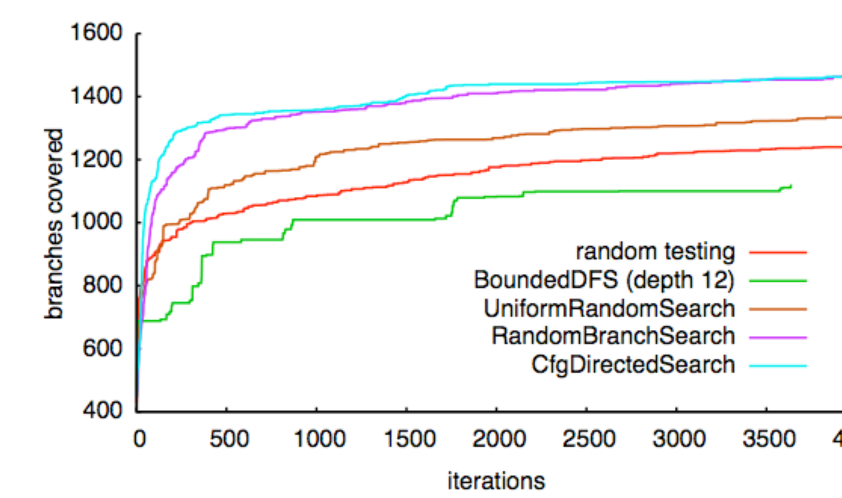
Call edges are dashed lines; conditional edges with weight one are in blue.

IMPLEMENTATION

- CREST, an open-source test generation tool for C.
  - C++ platform for experimenting with concolic search strategies.
  - Uses CIL to instrument programs and extract control flow.
  - Library for symbolic execution along a concrete execution.
  - Uses Yices SMT solver.
- Implemented our strategies in CREST.
- Available at: **http://crest.googlecode.com**

BENCHMARK: GNU grep 2.2

- Popular regular expression matching tool.
- 15K lines of C -- 4184 branches, an estimated 2854 of which are reachable..
- Input: Length-20 regexp and 40 characters of text.



- Control-flow and random-branch strategies cover 1/3 of branches (1/2 of reachable) in a few minutes.
- Better coverage than traditional depth-first search or random testing in the same number of runs of grep.

EXAMPLE OF CONTROL-FLOW DIRECTED SEARCH

Goal: Reach the ABORT statement in function f.

We initially run the program on zero inputs.

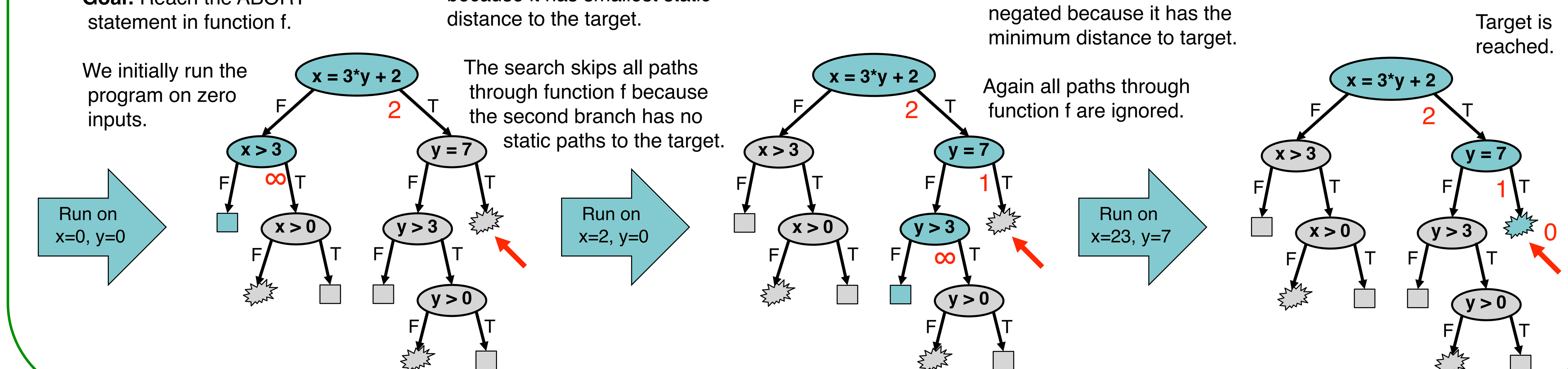
The first branch is negated because it has smallest static distance to the target.

The search skips all paths through function f because the second branch has no static paths to the target.

The second branch is negated because it has the minimum distance to target.

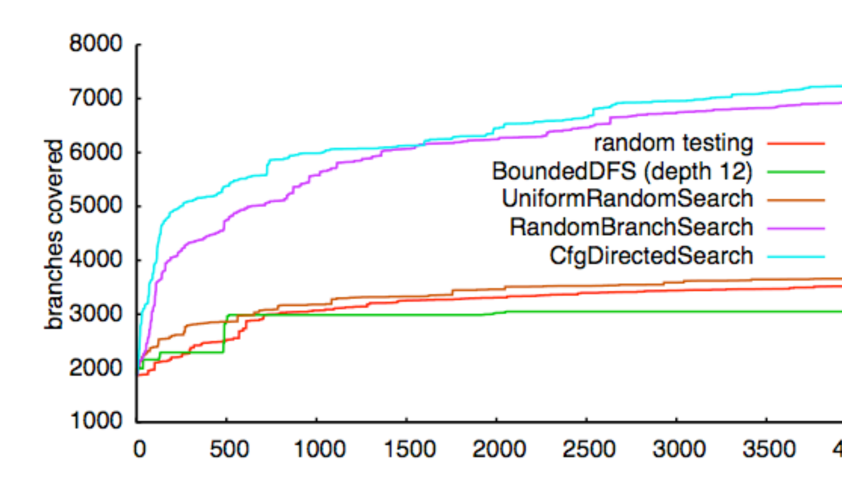
Again all paths through function f are ignored.

Target is reached.



BENCHMARK: Vim 5.7

- Popular open-source text editor.
- 150K lines of C -- 39,166 branches, an estimated 23,400 of which are reachable
- Input: 20 symbolic characters.



- Top two strategies cover nearly 1/5 of branches (1/3 of reachable branches) in 2-3 hours of testing.
- Gap between control-flow and random-branch strategies over others seems to grow with benchmark size.

UNIFORMLY-RANDOM PATH SEARCH

- **Key Idea**: Test program on **uniformly random paths** rather than on **random inputs**.
  - Much more likely to cover some paths than with random inputs – e.g. only $1/2^{64}$ inputs reach ABORT.
- Run program, then negate each branch with probability 1/2.
  - Generates each path with L feasible branches with probability 2^{-L}
  - Takes L/2 expected iterations.

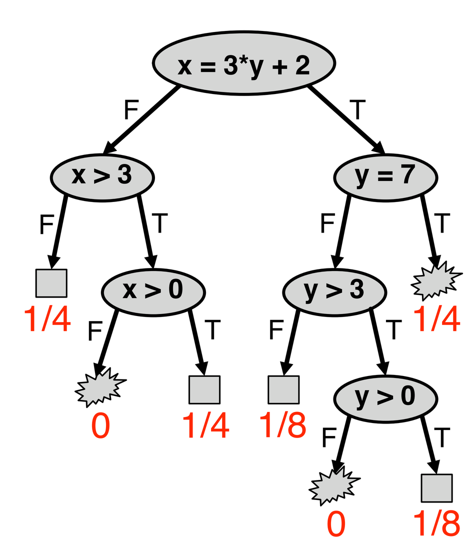


FIGURE 5 – Computation tree with probabilities for each path under uniformly-random path search.

RANDOM-BRANCH SEARCH

- **Motivation:** Generating paths uniformly at random is too expensive – L/2 expected iterations per path with L feasible branches.
- Instead, each iteration randomly pick and negate one feasible branch along the current execution path.
  - Samples some random walk through the path space.
- Found to be more effective in practice then generating uniformly random paths.

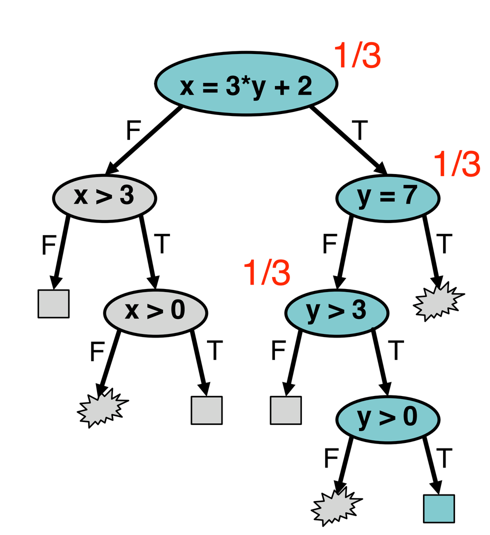


FIGURE 6 – Computation tree with probabilities for each branch to be negated given a current execution.