AUTOMATICALLY FINDING PATCHES USING GENETIC PROGRAMMING

Paper by Westley Weimer, ThanhVu Nguyen,
Claire Le Goues, Stephanie Forrest

Presentation by Ben Mishkanian
3/5/2015

Genetic Programming (GP)

1. Start with an input program
2. Generate new program through code "mutation" and "crossover"
3. Evaluate new program, and discard it if it is highly unsatisfactory.
4. Programs that are somewhat satisfactory are taken as input for step #1.

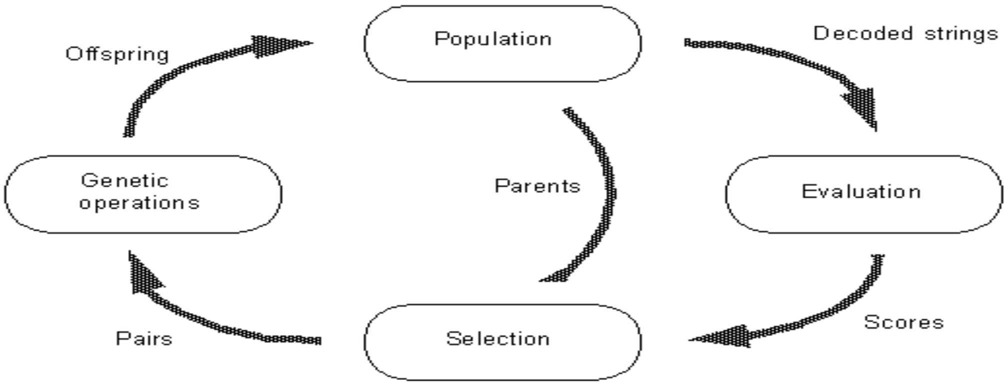
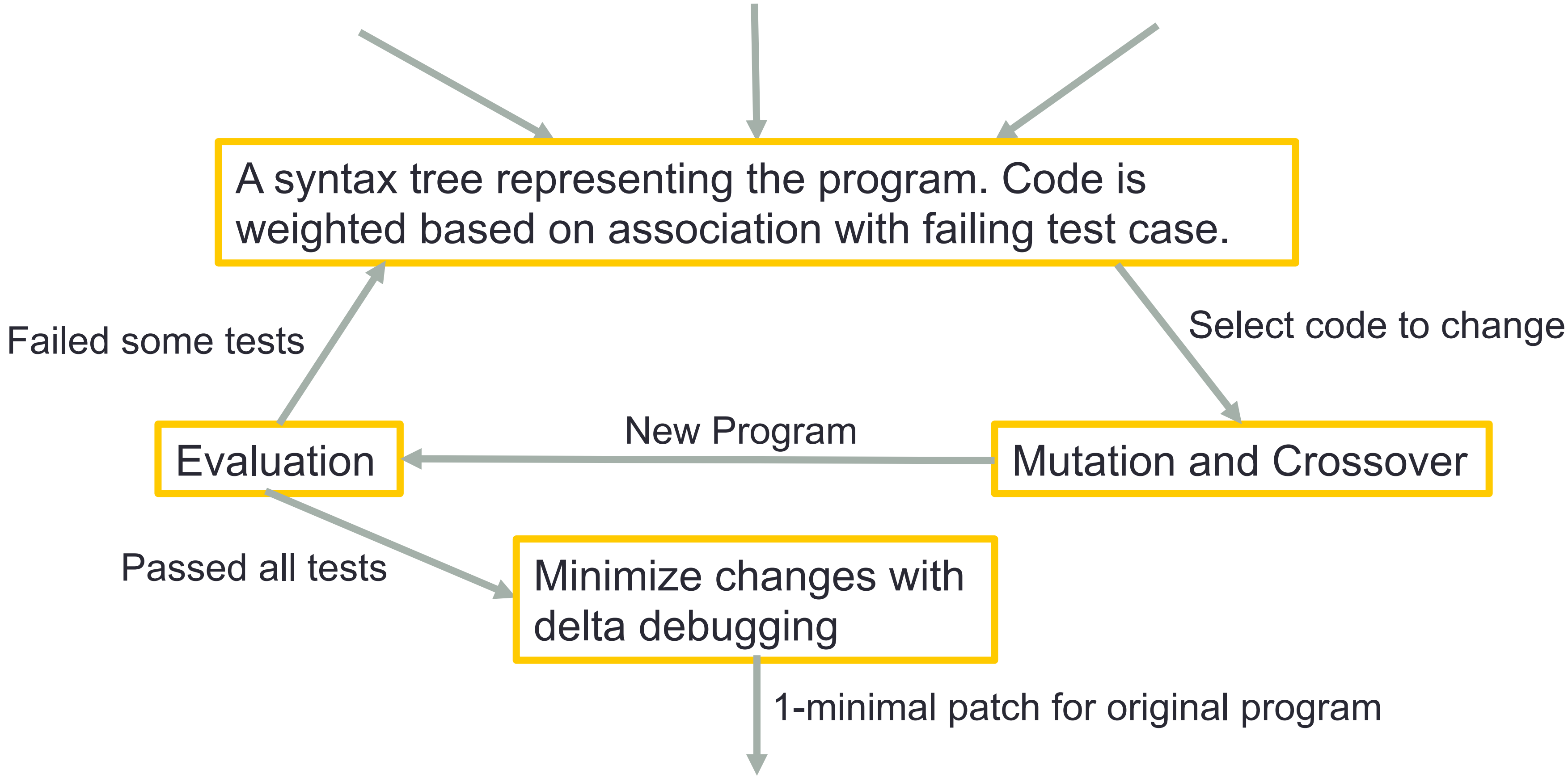


Figure 5.2: The "reproduction" cycle.

GP Algorithm for Automatically Repairing Bugs

Inputs: { Buggy program Passing test cases Failing test case }



Output: { program that passes all given test cases }

Example – Inputs

```
void gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
  }
  while (b != 0)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  printf("%d", a);
  exit(0);
}
```

gcd(1071,1029) is a passing test case

gcd(0,55) is a failing test case

Example – Possible Mutation

```
void gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
    exit(0);
    a = a - b;
  }
  while (b != 0)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  printf("%d", a);
  exit(0);
}
```

- This location was mutated because it is associated with failure.
- This code was chosen because it was seen in another section.

Example – Minimal Patch

```
void gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
    exit(0);
  }
  while (b != 0)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  printf("%d", a);
  exit(0);
}
```

Overview of the strategy

1. Create simplified representation of program
2. Select code to change
3. Select a change
4. Evaluate new program; Go to step 5 if all tests pass, else go to step 2
5. Reduce the changes to a minimal patch

Note: Interchangeable Terminology

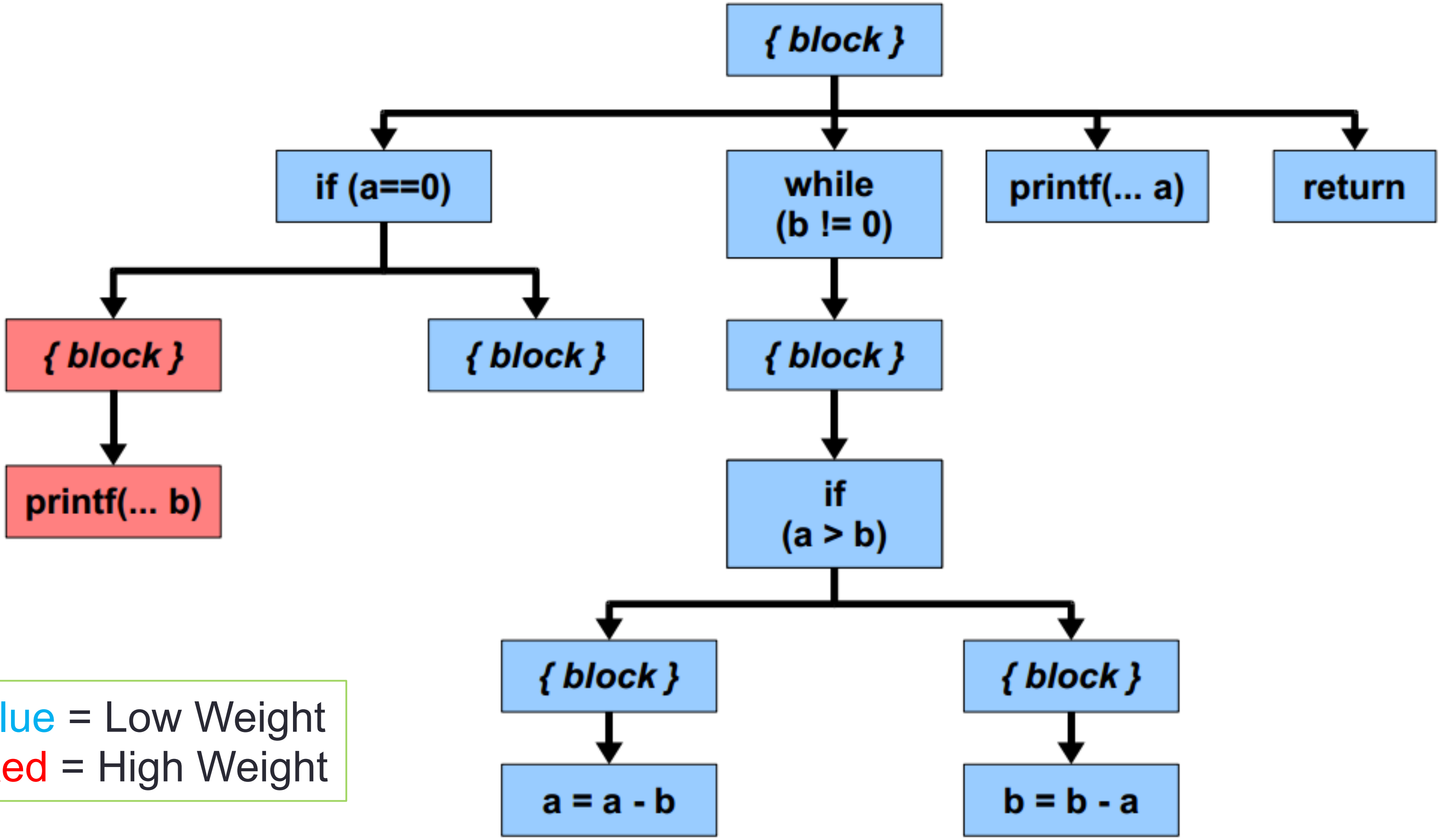
These all refer to the same thing:

- Variant
- Individual
- Chromosome
- Child
- Candidate
- Program

Step 1: Program Representation

- Abstract Syntax Tree (AST)
  - Represents statements and structure of program
- Weighted Path
  - Weights indicate association with passing/failing test cases
  - Weight is 1 for statements only reached in the failing case
  - Weight is 0.01 for statements ever reached in a passing case

Example AST with weighed paths

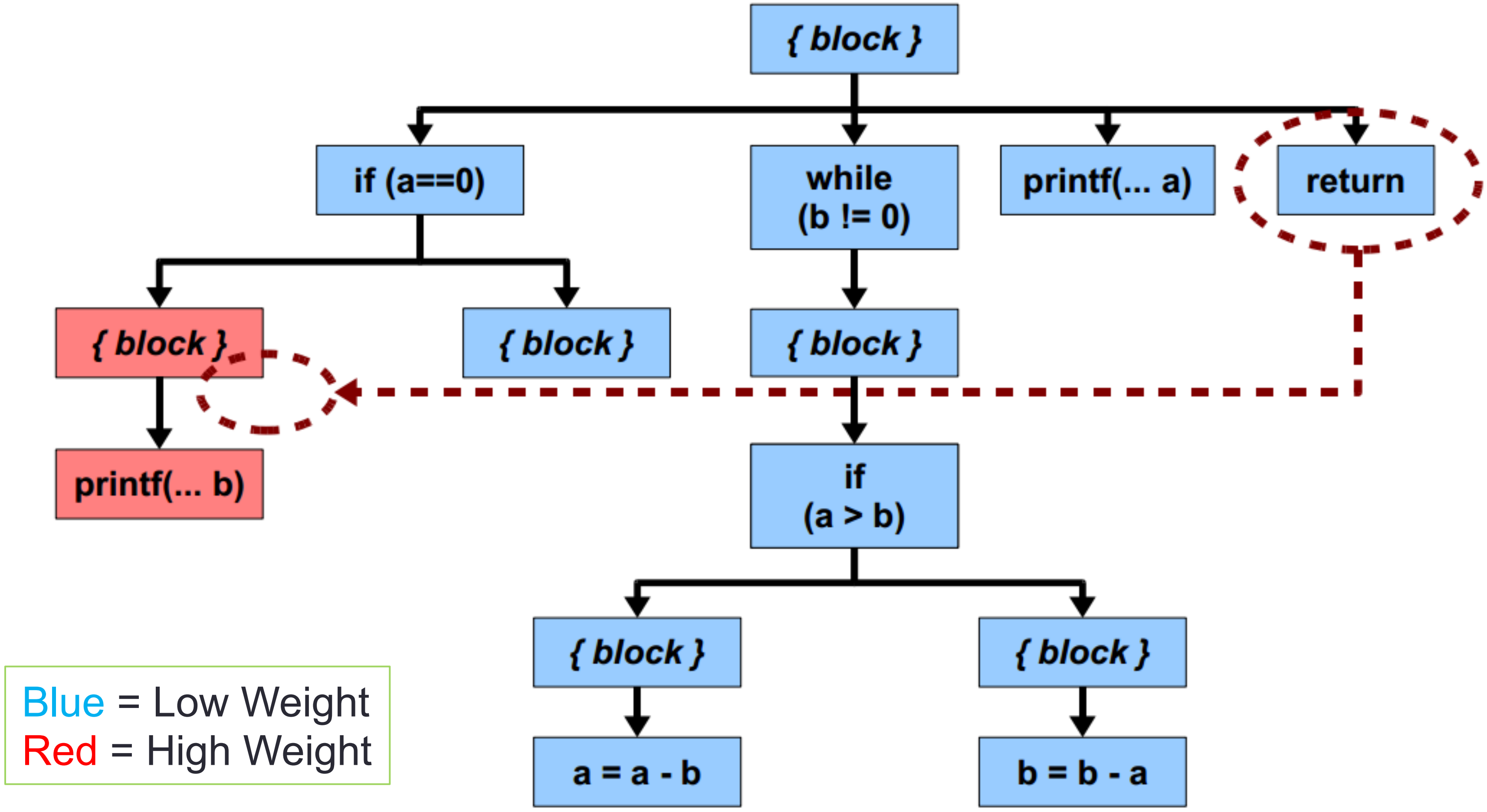


Blue = Low Weight
Red = High Weight

Step 2-3: Mutation & Crossover

- Mutation: Changes a statement on the weighted path
  - Biased to favor changing statements with high weight
  - Any new code inserted/swapped is copied from another part of the program
- Crossover: Combines two parts from different individuals
  - One individual is always the original program
  - Highly weighted statements are more likely to be crossed over

Example Mutation



Step 4: Fitness Evaluation

- Individuals are evaluated based on how many test cases they pass
- If no individual passes all test cases, the best one is chosen for further mutation/crossover
- If an individual passes all test cases, it becomes the patch candidate and is passed to the minimizer

Step 5: Patch Minimization

- The patch candidate may include code that has no bearing on satisfaction of the test cases
- We remove this extraneous code through delta debugging
- $O(n^2)$ time

Experiments/Evaluation

- The experiments evaluate:
  - Performance and scalability
  - Runtime
  - Success rate of search
  - Effect of testcases on repair quality
- Tested 10 open source programs
  - One negative test case per program
  - Set pop_size = 40
  - Set max generations to 10
  - 100 random trials per program

Experimental Results

- Avg. successful trial takes 184.7 seconds after 36.9 fitness evaluations
  - 54% of time spent executing test cases
  - 30% of time spent compiling individuals
- Over half the trials produced a repair (success)
  - Standard deviation of success rates is very large
- Avg. patch candidate produced after 3.5 crossovers, 1.8 mutations, over 6 generations.
- Final minimized patch is 4 lines on average

Patch Quality

- In experiments, all patches repair the negative test case while not affecting the positive test cases
  - Note: This does not prove that nothing was broken
- Anything not tested by the positive test cases is susceptible to change
  - The repair is very quick, but may sacrifice functionality
  - Useful as a quick patch while developers diagnose the problem
- When the negative test case contributes a lot of additional code coverage, success rate becomes very low
- Adding positive test cases increases success rate, at the cost of runtime.

Limitations

- Can generate wrong patch for nondeterministic programs
- Functionality not tested by positive cases may be broken
- Using too many test cases will cause slowdown
- Relies on the assumption that the broken code resides in an area not reached by the positive test cases
- Relies on the assumption that the correct code to use already exists in some part of the program