Search-based Testing
Software Engineering
Gordon Fraser • Saarland University

So many different test objectives...

But how to create the tests?

Test Data Generation
Given a function and a location we want to reach, how do we derive inputs to the function that lead the control flow to the desired statement?
Search-based Software Engineering

- Cast problems of software engineering as optimization problems
- Search problems in software engineering are **BIG**
- Apply meta-heuristic search algorithms to solve these problems

Search Algorithms

Random Testing

- (Totally) uninformed search
- If a systematic approach is not better than random testing, it is not useful
- Cheap & easy to implement
- Works pretty well in many cases
Heuristics

- Might not always find the best solution
- But finds a good solution in reasonable time
- Sacrifices completeness but increases efficiency
- Useful in solving tough problems
  Which could not be solved any other
  Which would take very long time to compute
- Use problem specific metrics

Meta-Heuristics

- Problem-independent algorithms to solve optimization problems
- Problem-specific implementation of certain parts (as heuristic)

Search Space

Search space every possible position (x,y)

Current solution: (3,3)
Search Space
Possible move:
(3,3), (3,4)

Neighborhood
Any possible move

Search Space
The heuristic estimates how close a candidate solution is to the optimum - the heuristic defines a search landscape.
Hill Climbing

1. Select Random Value

2. Explore Neighborhood

3. Choose best neighbor
Hill Climbing

4. Repeat until optimum is found

Local Optima

How do we know we're here?

Local Optima

How do we get out of here?
Plateaux

No guidance - search is random

Hill Climbing

- Simple hill climbing
  First better neighbor is chosen
- Steepest ascent
  Best neighbor is chosen
- Random-restart
  Randomly reset and start with a different random value
- Resets can lead out of local optima

Tabu Search

The overall approach is to avoid entrainment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited.
(Fred Glover 1987)
Tabu Search

- Keep track of last k moves
- "Tabu list"
- When choosing next move, take no solution that is on tabu list
- Avoids cycles of length k
Simulated Annealing

- Accept also worse solutions with $p = e^{\frac{\delta}{t}}$
- $t = $ temperature
- $\delta = $ difference in objective value

Evolutionary Algorithms

- Gene
  - Unit of information • passed from one generation to another
- Natural Selection
- Survival of The Fittest
- Origin of New Species

Biological Evolution
Genetic Algorithms

- Initialize Population
- Evaluate Population
- While not done
- Return best solution
- Select parents
- Recombine parents

- Fitness function?
- Stopping criterion?
- Initialize Population
- Evaluate Population
- While not done
- Return best solution
- Select parents
- Recombine parents
- Selection function?
- Recombination operators?
- Recombination frequency?

Encoding

- Binary encoding
- Number encoding
- Any encoding really...
- Encoding is problem specific
Encoding

• Search space: (x,y)

\[ \begin{bmatrix} 2 & 4 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \]

Initial Population

• Randomly chosen
• Existing solutions
• Coverage test suite for weaker criterion
• Manually generated test cases

Define:

- Fitness function?
- Stopping criterion?
- Encoding? Population size?
- Selection function?
- Recombination operators?

Initialize Population

Evaluate Population

While not done

Return best solution

Select parents

Recombine parents
Fitness

- How good is a candidate solution?
- Is solution A better than solution B?
- Fitness quantifies the “goodness” of a solution
- Fitness is problem specific
- Fitness function determines the search landscape

Kin Compensation

- Diversity in the population is key to successful search
- Kin compensation:
  - Fitness penalty if identical individual already exists
Fitness Proportionate Selection

- A.k.a. Roulette wheel selection
- Probability of choosing an individual is proportional to its fitness

<table>
<thead>
<tr>
<th>Individual</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
</tr>
<tr>
<td>3</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>1.4</td>
</tr>
<tr>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>0.8</td>
</tr>
<tr>
<td>8</td>
<td>0.6</td>
</tr>
<tr>
<td>9</td>
<td>0.4</td>
</tr>
<tr>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>11</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Chosen value: 0.81
Chosen value: 0.32
Chosen value: 0.01
• Individual 1 will dominate selection
• Don’t use roulette wheel selection!

Problems with Selection

• Selective pressure:
  The higher, the more likely the fittest are chosen

• Stagnation:
  Selective pressure too small

• Premature convergence:
  Selective pressure too high

Rank Selection

• Rank individuals according to their fitness

• No difference whether fittest candidate is
ten times fitter than the next fittest or
0.001% fitter

• Rank selection is preferable in practice

  Fitness(Pos) = 2 - Bias + 2*(Bias - 1)^* \( \frac{\text{Pos} - 1}{\text{Num} - 1} \)
Tournament Selection

- \( N = \) Tournament size
- Select \( N \) individuals randomly
- Best of the \( N \) individuals is selected
- Tournament size defines selective pressure
- A worse individual can win with a given probability

<table>
<thead>
<tr>
<th>Individual</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>0.58</td>
</tr>
<tr>
<td>3</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>0.21</td>
</tr>
<tr>
<td>5</td>
<td>0.12</td>
</tr>
<tr>
<td>6</td>
<td>0.11</td>
</tr>
<tr>
<td>7</td>
<td>0.1</td>
</tr>
<tr>
<td>8</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>0.05</td>
</tr>
<tr>
<td>10</td>
<td>0.01</td>
</tr>
<tr>
<td>11</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Tournament size: 4

\[
\begin{array}{c}
4 \\
6 \\
2 \\
8 \\
\hline
2 (1, 8)
\end{array}
\]
Elitism

- Don’t throw away best individuals
- Retain an “elite” that moves from one generation to the next unchanged

Crossover

```
1 0 0 1 0 1
0 0 1 0 1 1
```
Crossover

- Single point crossover
  Choose one point in parents and split/merge at that point
- Two point crossover
  Choose two points in parents and exchange middle parts
- Fixed vs. variable length
  Same crossover point in both parents - constant length
- Uniform crossover
  Genes are randomly chosen from either parent chromosome

Mutation

- Change of genes
- Mutation depends on genetic operators
- Binary representation - bit flip
- For binary encoding - use Gray code
  Guarantees Hamming distance of 1 for bit flips

Gray Code

- Binary code: 0111
- Right shift inverted code: 0011
- 0111 xor 0011
- 0100

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>000</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Determining Parameters

- Which crossover function?
- Which mutation operators?
- Which selection function?
- Selection bias?
- Tournament size?
- Encoding?
- How often does crossover occur?
- How often does mutation occur?
- How many generations?
- Population size?
- Elite size?

No Free Lunch Theorem

- In the entire domain of search problem, all search algorithms perform equally on average
- Each algorithm makes assumptions on the underlying problem
- Choose / adapt algorithms to specific domain

Test Data Generation
Objectives

- Structural testing
- Given: Entry function / function to test
- Test function has parameters
- Determine values for parameters to satisfy a chosen test requirement
- Search space: Values of input parameters

Figure 12.2: The control flow graph of function cgi_decode from Figure 12.1

While the program is executed, one statement (or basic block) after the other is covered – i.e., executed at least once – but not all of them. Here, the input is “test”; checkmarks indicate executed blocks.
public int gcd(int x, int y) {
    int tmp;
    while (y != 0) {
        tmp = x % y;
        x = y;
        y = tmp;
    }
    return x;
}

Dominators

- Node A dominates B if every path to B goes through A
- Immediate dominator: Closest dominator on any path from the root
- Root node has no immediate dominator
public int gcd

public int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
        tmp = x % y;
        x = y;
        y = tmp;
    }
    return x; }

Post Dominators

- Dominators viewed in reverse (paths from exit node)
- Node A post-dominates B if all paths from B to exit must go through A
- Immediate post dominator: Closest dominator on any path to the exit node
Calculating Post Dominance

\[ \text{PDom}(n_e) = \{n_e\} \]
\[ \text{PDom}(n) = \bigcap_{s \in \text{succ}(n)} \text{PDom}(s) \cup \{n\} \]

- Initialize PDom(n) with set of all nodes for all n except exit node ne
- while some PDom(n) changed
  - recalculate PDom(n) for all n with above formula

Control Dependence

- A is control dependent on B if:
  - B has at least two successors in the CFG
  - B dominates A
  - B is not post-dominated by A
  - There is a successor of B that is post-dominated by A

Nodes C,D,E, are post-dominated by E. Node B has successors both within and without the gray area, so it controls whether E is executed; thus E is control-dependent on B.
In the control dependency graph a node is control dependent on its parent node.
How close were we to executing E?

Distance = 1

Approach Level

- Number of control dependent edges between goal and chosen path
- Approach = Number of dependent nodes - number of executed nodes
void landscape_example(int i, int j) {
    if (i >= 10 && i <= 20) {
        if (j >= 0 && j <= 10) {
            // target statement
            // ...
        }
    }
}
Branch Distance

- Critical branch = branch where control flow diverged from reaching target
- Distance to branch = distance to predicate being true / false
- Distance metric for logical formulas
- E.g. distance from true - false = 1

Branch distance

<table>
<thead>
<tr>
<th>Condition</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>( \text{abs}(a-b) = 0 ? 0 : \text{abs}(a-b) + K )</td>
</tr>
<tr>
<td>a != b</td>
<td>( \text{abs}(a-b) \neq 0 ? 0 \text{ else } K )</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>( a - b &lt; 0 ? 0 : (a - b) + K )</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>( a - b \leq 0 ? 0 : (a - b) + K )</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>( b - a &lt; 0 ? 0 : (b - a) + K )</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>( b - a \leq 0 ? 0 : (b - a) + K )</td>
</tr>
</tbody>
</table>

Branch distance

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>( \text{true} ? 0 : K )</td>
</tr>
<tr>
<td>A &amp;&amp; B</td>
<td>distance(A) + distance(B)</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>!a</td>
<td>Move inward and propagate</td>
</tr>
</tbody>
</table>
Example

\[ j = 5 \]

- \( j \geq 0 \)
- \( j \geq 10 \)
- \( j \geq 10 \) \& \( j \leq 20 \)

Objective Function Value

Figure 14: Objective function landscape of Tracy [47] for example of Figure 11

Objective Function Value

Figure 15: Objective function landscape of Wegener et al. [48] for example of Figure 11

Control+Branch Distance

- Control distance results in plateaux
- Branch distance results in local optima
- Combination!
- (dependent - executed) + branch_distance

Approach level

Branch distance at node of diversion
void landscape_example(int i, int j) {
    if (i >= 10 && i <= 20) {
        if (j >= 0 && j <= 10) {
            // target statement
            // ...
        }
    }
}

• $i = 0, j = 0$
• $i = 10, j = 25$
• Approach level: 2
• Approach level: 1
• Branch distance: 10
• Branch distance: 15

Which one is closer?
Normalization Functions

- Approach level should be dominant in fitness
- The branch distance has to be normalized

\[ 1 - \alpha^x \]

\[ \frac{x}{x + 1} \]

Use this!

Measuring Fitness

To measure the fitness of an individual, we need to

- Execute the test case
- Record the control flow path taken
- Record the branch distance at every branch
- Instrumentation is necessary for this

```c
void landscape_example(int i, int j) {
    if (i >= 10 && i <= 20) {
        if (j >= 0 && j <= 10) {
            // target statement
            // ...
        }
    }
}
```
void landscape_example(int i, int j) {
    Trace(Entry);
    if (i >= 10 && i <= 20) {
        Trace(Branch1);
        if (j >= 0 && j <= 10) {
            Trace(Branch2);
            // target statement
            // ...
        }
    }
}

What's missing in this example is that the distance to the false outcome of a branch predicate also needs to be traced.
Example

Classify triangle by the length of the sides

Equilateral  Isosceles  Scalene

int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (!(a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}
Let's simulate hill climbing on the example. The first candidate solution is chosen randomly.
Branch Distance

- Input (21, 53, 38)
- Distance to (a <= 0 || b <= 0 || c <= 0)?
- Minimum of distance to a<=0, b<=0, c<=0
- a <= 0
- a - b <= 0 ? 0 : (a - b) + K (K = 1)
- Minimum of (21, 53, 38) = 22

Fitness

- Approach level: 1
- Branch distance: 22
- Fitness = 1 + 22/(22+1) = 1.96

The best neighbor of the first solution (x decreased by 1) is the next candidate in the search.
Search

- Next best solution:
  - (20, 53, 38) - Approach 1, Branch 21
  - (19, 53, 38) - Approach 1, Branch 20
  - (18, 53, 38) - Approach 1, Branch 19
  - (17, 53, 38) - Approach 1, Branch 18
  - (16, 53, 38) - Approach 1, Branch 17
  - (15, 53, 38)

Search

- (15, 53, 38) - Approach 1, Branch 16
- (14, 53, 38) - Approach 1, Branch 15
- ...
- (0, 53, 38) - Approach 0, Branch 0
int triangle(int x, int y, int z) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}

Again, the first candidate solution is selected randomly.
Branch Distance

- Input (5, 0, 9)
- Distance to !(a <= 0 || b <= 0 || c <= 0)?
- Sum of distance to a>0, b>0, c>0
- b - a < 0 ? 0 : (b - a) + K (K = 1)
- 0 + 1 + 0

Fitness

- Input (5, 0, 9)
- Approach level: 3
- Branch distance: 1
- Fitness = 3 + 1/(1+1) = 3.5

The best neighbour is chosen as next candidate.
Branch Distance

- Input (5, 1, 9)
- Distance to !(a == b && b == c)?
- Sum of distance to a==b + b==c
- abs(a-b) = 0 ? 0 : abs(a-b) + K (K = 1)
- 5 + 9

Fitness

- Input (5, 1, 9)
- Approach level: 2
- Branch distance: 14
- Fitness = 2 + 14/(14+1) = 2.9

Search

- (5, 2, 9) - Approach 2, Branch 12
- (5, 3, 9) - Approach 2, Branch 10
- (5, 4, 9) - Approach 2, Branch 8
- (5, 5, 9)
int triangle(int x, int y, int z) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4;
    }
    if (! (a + b > c && a + c > b &&
          b + c > a)) {
        return 3;
    }
    if (a == b && b == c) {
        return 4;
    }
    if (a == b || b == c || a == c) {
        return 2;
    }
    return 1;
}

Branch Distance

- Input (5, 5, 9)
- Distance to !(a == b && b == c)?
- Sum of distance to a==b + b==c
- abs(a-b) = 0 ? 0 : abs(a-b) + K (K = 1)
- 0 + 5

Fitness

- Input (5, 5, 9)
- Approach level: 1
- Branch distance: 5
- Fitness = 2 + 5/(5+1) = 1.8
Search

- (5, 5, 8) - Approach 1, Branch 4
- (5, 5, 7) - Approach 1, Branch 3
- (5, 5, 6) - Approach 1, Branch 2
- (5, 5, 5) - Approach 0, Branch 0

```c
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}
```
```c
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}
```

The third branch we calculate with a genetic algorithm. The initial population is chosen randomly.

Each individual needs to be executed so that we can derive the approach level and branch distance.
int triangle(int x, int y, int z) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4;
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4;
    }
    if (a == b && b == c) {
        return 3;
    }
    if (a == b || b == c || a == c) {
        return 2;
    }
    return 1;
}

// Examples
(2, 7, 1)  // Approach: 3  Branch: 5
(7, 5, 1)  // Approach: 3  Branch: 2
(0, 3, 6)  // Approach: 4  Branch: 1
After fitness evaluation, recombination can start.

Let's assume individuals 1 and 3 are chosen by our selection algorithm.

Crossover is applied.
And mutation as well.

Two more individuals are selected (1,2)

This time, neither crossover nor mutation are applied.
The offspring now forms our new population.

After fitness evaluation, recombination starts.

Selected individuals: 1, 4
We apply crossover and mutation.
And repeat the recombination until our new population is big enough.

The new population contains an individual that is a solution.
Testability Transformation

If a program is difficult for test generation... ...transform it to a version more amenable to testing

Testability Transformation

• Generate tests
• Throw away transformed program
Testability Transformation

- Transforming source code might require co-transformation of coverage criteria
- Transformed program is only used to generate test data, and then discarded
- Transformation does not need to preserve meaning

The Flag Problem

```java
if(x == 10) {
    // ..
}

if(flag) {
    // ..
}
```

Flag Level 0

```java
if(x == 10) {
    // ..
}

if(flag) {
    // ..
}
```
Flag Level 1

```java
boolean flag = (x == 10);
// ..
if(flag) {
    // ..
}
// ..
```

Neither flag nor x are redefined before predicate, so we can simply replace flag with (x == 10) in the if-expression.

Flag Level 2

```java
boolean flag = (x >= 10) && (y < z);
// ..
x = 5;
y = x;
// ..
if(flag) {
    // ..
}
// ..
```

Flag Level 2

```java
int temp_x = x; int temp_y = y;
boolean flag = (temp_x >= 10) &&
(temp_y < z);
// ..
x = 5;
y = x;
// ..
if(flag) {
    // ..
}
// ..
```

Now the flag problem is reduced to level 1.
Flag Level 3

```c
x = y + 1;
y = x * 2;
flag = x > y;
y = y + flag;
flag = flag || y == 0;
x = y * x;
if(flag) ...
```

Flag Problem

- Level 4: Sequence of flag contains conditionals
- Level 5: Definition of flag in different loop-body than flag use

Nested Predicates

At this level, we can use amorphous slicing to derive a predicate for the flag (no, you don’t need to do this in the exam).

```c
void original(double a, double b) {
    if(a == b) {
        double c = b + 1;
        if(c == 0) {
            // target
        }
    }
}
```
Nested Predicates

```c
void transformed(double a, double b) {
    double _dist = 0;
    _dist += branch_distance(a == b);
    double c = b + 1;
    _dist += branch_distance(c == 0);
    if(_dist == 0.0) {
        // target
    }
}
```

Test Data Generation

Given a function and a location we want to reach, how do we derive inputs to the function that lead the control flow to the desired statement?

**Biological Evolution**
- Gene
- Units of information passed from one generation to another
- Natural Selection
- Survival of the Fittest
- Origin of New Species