Search based Testing

• The use of **metaheuristics** in test generation

• **Main idea**: Transform the test generation problem into an optimization problem, which can be further solved by applying metaheuristics.

• **Search-based testing** (SBT) is an **automated search** in a **large space** guided by a problem-specific **fitness (objective) function**

• **Local search**: evolves **one** candidate solution, e.g. hill climbing, simulated annealing

• **Global search**: evolves **a population** of candidate solutions, e.g. genetic algorithms, particle swarm optimization

Motivation - SBST

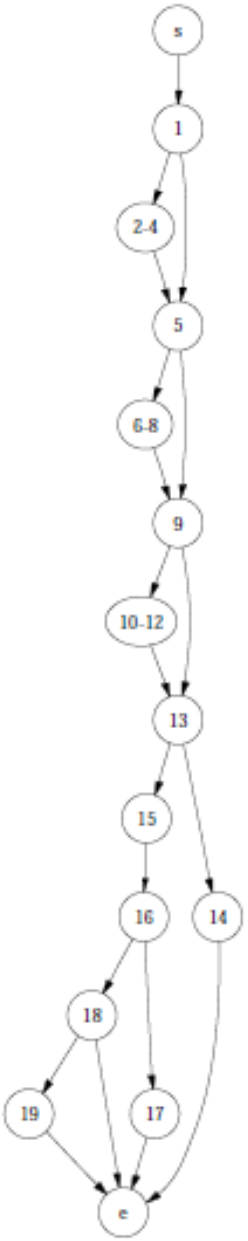
- Consider the selection of test value to trigger a path in a program

- Possible approach: symbolic execution

- Symbolic Execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure

Symbolic Execution - forward traversal

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`  int type;`
1	`  if (a > b)`
2-4	`  {  int t = a; a = b; b = t;  }`
5	`  if (a > c)`
6-8	`  {  int t = a; a = c; c = t;  }`
9	`  if (b > c)`
10-12	`  {  int t = b; b = c; c = t;  }`
13	`  if (a + b <= c)`
	`  {`
14	`  type = NOT_A_TRIANGLE;`
	`  }`
	`  else`
	`  {`
15	`  type = SCALENE;`
16	`  if (a == b && b == c)`
	`  {`
17	`  type = EQUILATERAL;`
	`  }`
18	`  else if (a == b || b == c)`
	`  {`
19	`  type = ISOSCELES;`
	`  }`
	`  }`
e	`  return type;`
	`}`



Path : < s; 1; 5; 9; 10; 11; 12; 13; 14; e >

Initially: a = i, b = j, c = k

(1) i <= j
(2) i <= k
(3) j > k

t = j
b = k
c = t

(4) i + k <= j

Symbolic Execution

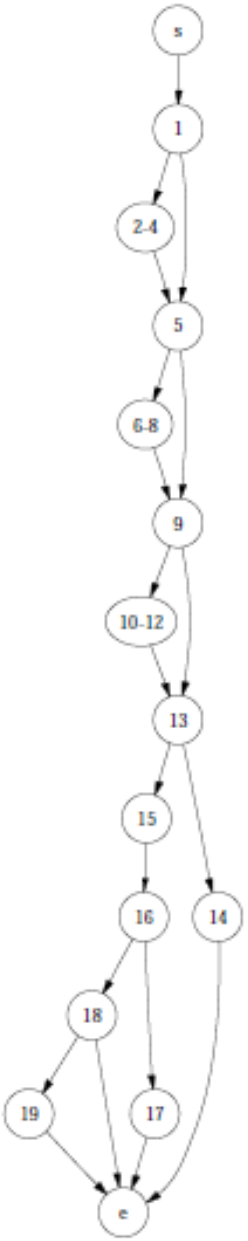
• Backward traversal :starts with the final node and follows the path in a reverse manner to the start node.

•The resulting constraint system is the same as for forward traversal, but no storage is required for the intermediate symbolic expressions of variables.

•Forward traversal, however, allows for early detection of infeasible paths whereas backward traversal does not

Symbolic Execution – fwd vs bwd traversal

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`  int type;`
1	`  if (a > b)`
2-4	`  {  int t = a; a = b; b = t;  }`
5	`  if (a > c)`
6-8	`  {  int t = a; a = c; c = t;  }`
9	`  if (b > c)`
10-12	`  {  int t = b; b = c; c = t;  }`
13	`  if (a + b <= c)`
	`  {`
14	`  type = NOT_A_TRIANGLE;`
	`  }`
	`  else`
	`  {`
15	`  type = SCALENE;`
16	`  if (a == b && b == c)`
	`  {`
17	`  type = EQUILATERAL;`
	`  }`
18	`  else if (a == b || b == c)`
	`  {`
19	`  type = ISOSCELES;`
	`  }`
	`  }`
e	`  return type;`
	`}`



Path : < s; 1; 2; 3; 4; 5; 6; 7; 8; 9; 13; ... ; e >

Constraints derived (up to node 9):

(1) i > j
(2) j > k
(3) i <= j

Backward traversal:
backwards from e through to 13 first, and to
node 1 before it would be possible to determine this fact.

Symbolic Execution

• If the constraints are linear, linear programming techniques can be applied.

• However, constraint satisfaction problems are in general NP-complete.

• Heuristic methods are recomended for solving NP-complete problems with large domains.

• Symbolic execution has several other problems, for example resolving computed storage locations such as array subscripts.

```
a[i] = 0;
a[j] = 1;
if (a[i] > 0)
{
    // perform some action
}
```

Hill climbing

Select a starting solution $s \in S$

Repeat

    Select $s' \in N(s)$ such that $obj(s') > obj(s)$ according to ascent strategy

    $s \leftarrow s'$

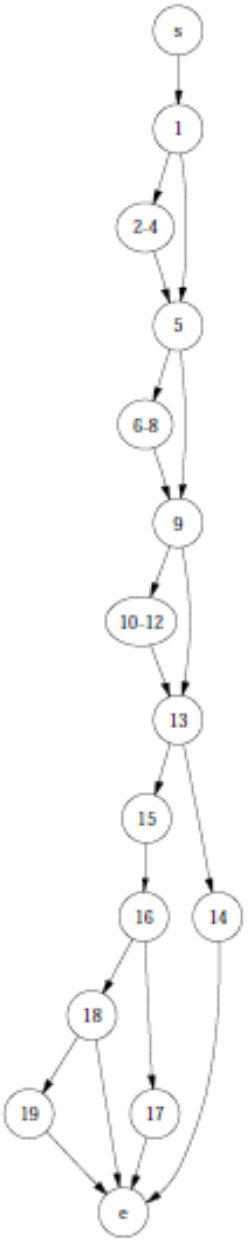
Until $obj(s) \geq obj(s'), \forall s' \in N(s)$

Local search for path testing

- The tester selects a path through the program
- A straight-line version of the path is then produced
- Branching statements are then replaced with path constraints, of the form $ci = 0$, $ci > 0$ or $ci \geq 0$, where ci is an estimate of how close the constraint is to being satised.
- Then the conditions ci need to be minimised
- For example, a branch predicate of the form a == b might be rearranged into the path constraint abs(a - b) = 0.

Local search for path testing

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`int type;`
1	`if (a > b)`
2-4	`{ int t = a; a = b; b = t; }`
5	`if (a > c)`
6-8	`{ int t = a; a = c; c = t; }`
9	`if (b > c)`
10-12	`{ int t = b; b = c; c = t; }`
13	`if (a + b <= c)`
	`{`
14	`type = NOT_A_TRIANGLE;`
	`}`
	`else`
	`{`
15	`type = SCALENE;`
16	`if (a == b && b == c)`
	`{`
17	`type = EQUILATERAL;`
	`}`
18	`else if (a == b || b == c)`
	`{`
19	`type = ISOSCELES;`
	`}`
	`}`
e	`return type;`
	`}`



Path : < s; 1; 5; 9; 10; 11; 12; 13; 14; e >

c1 = (b-a) >= 0

c2 = (c-a) >= 0

c3 = (b-c) > 0

int t = b; b = c; c = t;

c4 = (c - (a+b)) >= 0

type = NOT_A_TRIANGLE;

Local search for path testing

c1 = (b-a) >= 0

c2 = (c-a) >= 0

c3 = (b-c) > 0

int t = b; b = c; c = t;

c4 = (c - (a+b)) >= 0

type = NOT_A_TRIANGLE;

- A fitness function si then defined on the basis of the constraints c1, …, c4
- Problem: if all constraints are considered together (in one fitness function), then contradictory constraints may be problematic
- Solution: alternating variable method

Alternating variable method

c1 = (b-a) >= 0

c2 = (c-a) >= 0

c3 = (b-c) > 0

int t = b; b = c; c = t;

c4 = (c - (a+b)) >= 0

type = NOT_A_TRIANGLE;

Path : < s; 1; 5; 9; 10; 11; 12; 13; 14; e >

Consider (a = 10, b = 20, c = 30)

c1 and c2 are satisfied, c3 is not (the execution will diverge)

At this point a local search is invoked using a fitness function based on c3

Korel's objective functions

Relational predicate	f	rel
$a > b$	$b - a$	$<$
$a \geq b$	$b - a$	$\leq$
$a < b$	$a - b$	$<$
$a \leq b$	$a - b$	$\leq$
$a = b$	$abs(a - b)$	$=$
$a \neq b$	$-abs(a - b)$	$<$

Alternating variable method

c1 = (b-a) >= 0

c2 = (c-a) >= 0

c3 = (b-c) > 0

int t = b; b = c; c = t;

c4 = (c - (a+b)) >= 0

type = NOT_A_TRIANGLE;

Path : < s; 1; 5; 9; 10; 11; 12; 13; 14; e >

The objective function f for c3 is c – b > 0

The objective function describes how close the predicate is to being true

Each variable is taken in turn and its value is adjusted (while the others are kept constant)

a does not influence f

b is increased $\Rightarrow$ improved f

(a = 10, b = 31, c = 30)

Alternating variable method

c1 = (b-a) >= 0

(a = 10, b = 31, c = 30)

c2 = (c-a) >= 0

The objective function for c4 is (a + b) - c <= 0

c3 = (b-c) > 0

b cannot be decreased (it leads to the violation of c3)

int t = b; b = c; c = t;

c4 = (c - (a+b)) >= 0

b is increased

type = NOT_A_TRIANGLE;

b is increased $\Rightarrow$ improved fitness (since the values of b and c have been swapped)

Path : < s; 1; 5; 9; 10; 11; 12; 13; 14; e >

(a = 10, b = 40, c = 30)

Alternating variable method

•As with all local searches, the final result is dependent on the starting solution.

```
void nested_example(int a, int b, int c)
{
    if (a == b)
        if (b == c)
            if (c < 0)
                // target
}
```

• Consider (a=10, b=10, c=10)

• Control flow proceeds directly down to the final branching node.

•However the variable c can not be changed to a value less than 0, because the already successful sub-path up to the final branching node will be violated. In this case, the search will fail.

Alternating variable method

•In order to make the search more efficient, an “influences” which detects which input variables are able to influence the outcome at the current branching node, can be constructed

•A risk analysis of input variables is also undertaken in order to decide if they could potentially violate the already successful sub-path.

•For example at node 5, it is more attractive to manipulate c rather than a or b, since changing a or b may change the current successful sub-path through node 1.

Goal oriented approach

•In the previous approach: for fulfilling a structural coverage criterion like statement coverage, a path has to be selected for each individual uncovered statement.

•The Goal-Oriented Approach removes this requirement.

•This is achieved through the classification of branches in the control flow graph of the program with respect to a target node as either *critical*, *semi-critical* or *non-essential*.

Goal oriented approach

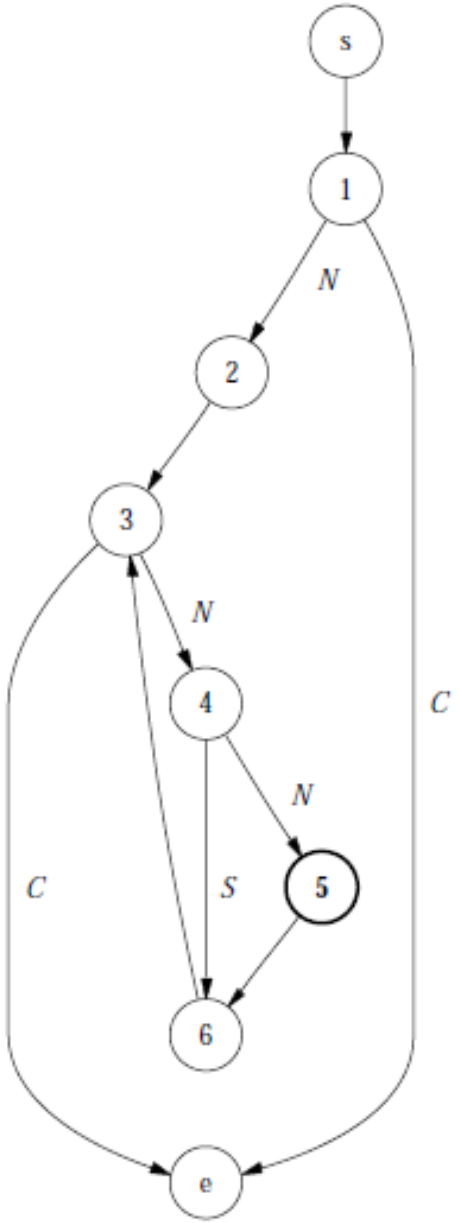
•A **critical branch** is the edge which leads the execution path away from the target node.

•A **semi-critical** branch is one which leads to the target node, but only via the backward edge of a loop.

•Finally, a **non-essential** branch is neither critical or semi-critical.

Goal oriented approach

CFG Node	
s	`void goal_oriented_example(int a)`
	`{`
1	`    if (a > 0)`
	`    {`
2	`        int b = 10;`
3	`        while (b > 0)`
	`        {`
4	`            if (b == 1)`
	`            {`
5	`                // target`
	`            }`
6	`            b --;`
	`        }`
	`    }`



Goal oriented approach – critical branch

•If control flow is driven down a critical branch, there is no prospect of the target being reached.

•Therefore, an objective function is associated with the branch predicate of the alternative branch.

•The alternating variable search method is then employed to seek inputs so the alternative branch is taken instead.

•If the required inputs cannot be found, the overall process terminates, with the target remaining unexecuted.

Goal oriented approach – critical branch

CFG Node	
s	`void goal_oriented_example(int a)`
	`{`
1	`    if (a > 0)`
	`    {`
2	`        int b = 10;`
3	`        while (b > 0)`
	`        {`
4	`            if (b == 1)`
	`            {`
5	`                // target`
	`            }`
6	`            b --;`
	`        }`
	`    }`

If the input vector is (a=0) the false branch from condition 1 is taken

This branch is critical.

The search procedure is invoked to change the value of a.

Goal oriented approach – semi-critical branch

•A semi-critical branch is one which leads to the target node, but only via the backward edge of a loop. The alternative branch from the same branching node leads directly to the target node.

•In the case where the execution is driven down a semi-critical branch, the alternating variable method is again invoked to seek inputs for the execution of the alternative branch.

•If suitable input values cannot be found, however, the process does not terminate. Execution is allowed to go down the semi-critical branch, in the hope of taking the alternative branch in the next iteration of the loop.

Goal oriented approach – semi-critical branch

CFG Node	
s	`void goal_oriented_example(int a)`
	`{`
1	`if (a > 0)`
	`{`
2	`int b = 10;`
3	`while (b > 0)`
	`{`
4	`if (b == 1)`
	`{`
5	`// target`
	`}`
6	`b --;`
	`}`
	`}`

The false branch from condition 4 is semi-critical

The search can not change the outcome at this branch

So the flow of control is allowed to continue around the loop a further nine times.

Finally, the true branch from node 4 is taken, and the target is reached.

Goal oriented approach – non-essential branch

- A non-essential branch is neither critical or semi-critical.

- Nonessential branches do not determine whether the target will be reached, regardless of their position in the control flow graph.

- Therefore, execution is allowed to proceed unhindered through these branches.

Goal oriented approach

•As the Goal-Oriented method also employs the alternating variable local search, it suffers from similar problems to those of Korel's original approach.

•The removal of the requirement to select a path, although relieving some effort on behalf of the tester, introduces new ways in which the test data search can fail.

Goal oriented approach

CFG Node	
s	`void chaining_approach_example(int a)`
	`{`
1	`    int b = 0;`
2	`    if (a > 0)`
	`    {`
3	`        b = a;`
	`    }`
4	`    if (b >= 10)`
	`    {`
5	`        // target`
	`    }`
	`    // ...`
	`}`

Consider what happens when the initial input vector is selected so that a < 0 (approximately half of the input domain).

The critical false branch from node 4 is taken.

The search will fail, since small exploratory moves of a will have no e effect on the objective function associated with this condition, which is concerned only with the value of b.

Goal oriented approach

•In this example, one could attribute the failure to the use of a local search technique.

•A global search technique such as a Genetic Algorithm is likely to sample the input domain more thoroughly and find the required value of a.

•This situation could be avoided if the data dependencies of the test goal were also taken into account by search (the Chaining Approach).

Avoiding local optima with local search

- Simulated annealing
- Tracey’s approach - avoids faulty paths

Simulated annealing

Select a starting solution $s \in S$
Select an initial temperature $t > 0$
Repeat
    $it \leftarrow 0$
    Repeat
        Select $s' \in N(s)$ at random
        $\Delta e \leftarrow obj(s') - obj(s)$
        If $\Delta e < 0$
            $s \leftarrow s'$
        Else
            Generate random number r, $0 \leq r < 1$
            If $r < e^{-\frac{\delta}{t}}$ Then $s \leftarrow s'$
        End If
        $it \leftarrow it + 1$
    Until $it = num\_solns$
    Decrease t according to cooling schedule
Until Stopping Condition Reached

Tracey's function

Relational predicate	Objective function obj
Boolean	if $TRUE$ then 0 else K
$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a-b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a-b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b-a) + K$
$a \geq b$	if $b - a \geq 0$ then 0 else $(b-a) + K$
$\neg a$	Negation is moved inwards and propagated over a

Connective	Objective function obj
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$

Tracey's function

•The objective functions used are in principle identical to those employed by Korel, except the use of a nonzero positive failure constant K.

•It removes the need to use a relation rel within the function.

•In this way, the objective function always returns a value above zero if the predicate is false, and zero when it is true.

• Formulae for logical operators are added.

Trayce's approach

•In order to reduce the chances of the search becoming stuck in local optima, Tracey drops the constraint employed by Korel that the newly generated solution must conform to an already successful sub-path.

•However, the means of doing this results in the search losing some information about its progress.

•This is because solutions which diverge away from the target down earlier critical branches are assigned similar objective values to those diverging away at a later stage.

Trayce's approach

CFG Node	
s	`void landscape_example(int i, int j)`
	`{`
1	`    if (i >= 10 && i <= 20)`
	`    {`
2	`        if (j >= 0 && j <= 10)`
	`        {`
3	`            // target statement`
	`            // ...`
	`        }`
	`    }`
	`}`

- For the target statement at node 3, the false branches from nodes 1 and 2 are critical.

- Under Korel's scheme, if the current solution is (i=10, j=-1), diverging down the critical branch from node 2, the vector (i=9, j=-1) would not be given consideration, because the already successful sub-path up to node 2 is violated.

- In Tracey's method, a move can take place between solutions, and furthermore, the solutions are rewarded identical objective values – since the distance values taken at the different branching nodes are the same.