Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Template tesi ISISLab

Relatore
Giuseppe Scanniello
Correlatore
Dott. Nome Cognome

Candidato
Michelangelo Esposito

Academic Year 2021-2022

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

List of Figures

Contents

Chapter 1

Introduction

Things to add:

- where different types of coverage are useful
- Add formulas for coverage criteria

Chapter 2

Problem formulation

2.1 Coverage testing

Coverage is one of the metrics employed during testing to assess what portion of the source code is "covered" by the test suite i.e., what portion of the code is executed when the tests run; it is essential to extract information about the general quality of a test suite and helps determine how comprehensively the software is being verified. As a result, coverage can be classified as a white-box testing technique.
Source code coverage can be expressed according to different criteria:

- Statement coverage aims executing every single statement in the code.
- Branch coverage, also known as decision coverage, measures how many decision structure have been fully explored by the test cases. Out of all the branches in a program, some may be unreachable/infeasible [**DBLP:conf/issta/YatesM89**], i.e. there is no input configuration that will result in their execution, due to a developer error; as a result, when performing coverage testing, such branches may pose a threat since, if undetected, may drain test budget.
- Mutation coverage, also known as fault-based testing, aims at purposefully introducing faults in the program in order to check whether the test suite is able to identify them. If the fault is correctly detected, the mutant is "killed". One issue of mutation is scalability, since generating and compiling the mutants, before running the test cases, can be time-consuming end quickly exhaust testing resources. Additionally, the introduced mutations can be classified as weak or strong: with strong

mutation, the artificial fault is propagated to an observable behavior, while weak mutants are confined to more specific environments.

- Function coverage measures how many functions have been called by the test cases.

- Condition coverage determines the number of boolean conditions/expressions executed in the conditional statements.

To reach statement coverage, it is sufficient to execute a branch in which the statement is control dependent.

A high coverage can sometimes be deceiving, however: in the case of Machine Learning Systems (MLS), for example, where typically the source code is made up of a sequence of library functions and API invocations [**DBLP:journals/ese/RiccioJSHWT20**], thus resulting in very high statement and branch coverage with relatively modest test suites. Additionally, the effectiveness of such systems is highly determined by the dataset employed for model training and validation, which cannot be covered by tradition test cases.

Coverage can also be measured at any testing levels; while at the unit test-level we focus mostly on the coverage of statements and branches, at the system-testing level, the coverage targets shift towards more complex elements, such as menu items, business transactions or other operations that require multiple components of the system to work properly.

A coverage goal is a particular target that we want to cover in respect to the chosen criterion, i.e. a particular branch of an if statement.

2.2 An overview on search problems

As humans, everyday we perform search: from looking for our car keys in the house to searching for a new book to purchase, this action is engraved in our daily life. In the same way, search is one of the most fundamental operations on which computer science has focused since its early days: performing efficient search is the core operation of many classes of algorithms, such as the ones responsible for route planning, computer vision, robotics, automated software testing, puzzle solving, and many others.

Any search problem is typically defined by:

- A search space: the set of elements in which we search for our solution. Examples include paths in a graph, the numbers to insert in a sorted list, or the web pages accesses by a web index.

- A condition that defines the characteristics of candidate solutions.

formal problem definition with graphs, distance functions, ...

A search algorithm will therefore examine the search space according to some criteria and attempt to find a suitable solution. Finding any candidate solution is just one of the objective of search problems however, often times we are interested in finding the best possible solution, the optimum; such problems are referred to as optimization problems.

The exploration of the search space can happen in many different ways. The most basic approach consists of exploring the elements one by one, till a solution/the optimum is found, in a brute-force manner. Applying this simple solution for problems whose search space is somewhat limited can be done without too many repercussions on execution time, however, given the exponential size of the search spaces of most practical problems, a brute-force approach is infeasible.

For problems in which we have no choice but to employ brute-force search, there may be some room for improvement by using heuristics. Heuristics are estimates of the distance function to reach a goal state from the current node [**HeuristicSearch**].

Therefore, what it is preferred to obtain a solution that may not be the optimum, but rather it is "good enough" for our objective. This approach is called local search.

Hill climbing, simulated annealing, GAs... Genetic Algorithms (GAs) are an example of an evolutionary search approach for test case generation; starting from an initial, often randomly generated, population of test cases, the algorithm keeps evolving the individuals according to simulated natural evolution theory principles. In this context, a typical fitness function of a GA would measure the distance between the execution trace of the generated test cases and the coverage targets.

2.3 Search-based test case generation

Search-based approaches for test case generation use optimization algorithms to attempt to find the best candidate test case with the objective to maximize fault detection.

2.4 Iterative single-target search

The simplest way of approaching the evolutionary search problem for test case generation is by iteratively determining a coverage goal in the source

code (i.e. a particular branch), and executing the GA to find a test case that achieves this coverage. A strategy for iterative search typically includes:

- Enumerating the targets to cover, i.e. the individual branches.
- Performing the single-objective search for each target, until all the targets are covered, or the budget has expired.
- Building the final test suite by combining all the generated test cases.

Formally, the search problem for branch coverage can be formulated as follows:

Problem 1. Let $B = \{b_1, ..., b_k\}$ be the set of branches in a program; find a test suite $T = \{t_1, ..., t_n\}$ of n test cases that covers all feasible branches, minimizing the fitness function:

$$minf_B(T) = |M| - |M_T| + \sum_{b \in B} d(b, T) \tag{2.1}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods by all test cases in T, and d(b, T) indicates the minimal normalized branch distance for branch $b \in B$

Add infeasible branches

Focusing on one coverage goal at a time can be a poor strategy, however. Foremost, a search algorithms could get "trapped" while attempting to cover an expensive branch and waste a large portion of the testing budget [**DBLP:journals/tse/FraserA13**]. Secondly, the order by which coverage targets are selected may end up largely impacting the final performance. Additionally, this approach assumes that all coverage goal are equally important and independent of each other; this is often not the case as, for example, covering the true branch of an if statement may be easier than covering the true branch of another if statement that requires a complex chain of operations to be properly satisfied. Finally, covering a particular branch may have collateral coverage over other branches in the test case's path.

2.5 Whole test suite approach

The issues with iterative search suggest that multi-target evolutionary approaches may reveal more effective and reliable. These are known as whole test suites approaches (WS) [**QSIC11**] and their goal is to evolve the entire

test suite simultaneously, rather than iteratively covering the single branches/statements; this eliminates the two issues of the iterative approach: the algorithms can't get stuck on an expensive branch, since the all coverage targets are being searched simultaneously and, for the same reason, the order of test case generation becomes irrelevant, preventing adversary influence in the final test suite.

In the context of the WS approach, the fitness function used is still one for the entire test suite and is computed as an aggregated value from the fitness values measured for the single test cases, in order to take into consideration all coverage targets simultaneously for the chosen criterion.

In fact, each test case in a test suite is associated with the target closest to its execution trace. The sum over all test cases of such minimum distances from the targets provides the overall, test-suite-level fitness. The additive combination of multiple targets into a single, scalar objective function is known as sum scalarisation in the theory of optimization [**SearchMethodologies**].

Branch coverage is typically the most used coverage criterion for test case generation. In this context, a fitness function is based on two parameters:

- Approach level: represents how far the execution path of a given test case is from covering the target branch.
- Branch distance: represents how far the input data is from changing the conditional value of the branch.

Given that the branch distance can be arbitrarily greater than the approach level, usually this value is normalized [**DBLP:conf/icst/Arcuri10**].

While being more effective than the iterative approach, algorithms based on WS principles suffer from the problems of sum-scalarization in many-objective optimization, among which the inefficient convergence occurring in the non-convex regions of the search space

2.6 Many-objective search

The final approach consists in performing a many-objective search; here different coverage targets are considered different objectives to be optimized. Studies in literature [**DBLP:conf/ppsn/HandlLK08**] have demonstrated that when applying many-objective search to a complex problem previously

approached with single-objective search may bring improvements, since the probability of being trapped in a local maxima is reduced, also leading to a faster convergence rate [**DBLP:conf/icst/PanichellaKT15**].

The new problem can then be formulated as follows [**DBLP:journals/tse/PanichellaKT18**]:

As a result of the nature of this problem, the fitness function doesn't stem from the aggregated fitness scores of the individual test cases anymore; rather, it is measured according to the multi-objective nature of optimality, with the goals of minimizing each of the fitness functions of the generated test case population.

Chapter 3

Literature

3.1 EvoSuite

EvoSuite is an example of an evolutionary algorithm that optimizes the whole test suite towards just one coverage criterion, rather than generating test cases directed towards multiple coverage criteria. With EvoSuite, any collateral coverage isn't a concern since all coverage is intentional, given that the ultimate goal is to generate the whole test suite. The algorithm starts with a randomly generated population of test suites. The fitness function rewards better coverage of the source code; if two suites have the same coverage, the one with fewer statements is chosen. For each test suite, its fitness is measured by executing all of its test cases and keeping track of the executed methods and of the minimal branch distance for each branch.

Expand on bloat in EvoSuite

3.2 NSGA-II

A popular algorithm for many-objective search problems is the Non-dominated Sorting Genetic Algorithm II (NSGA-II). This algorithm is based on three principles:

- It uses elitism when evolving the population: the most fit individuals are carried over along the offsprings.
- It uses an explicit diversity-preserving mechanism, the Crowding distance.
- It emphasizes the non-dominated solutions, as its name suggests.

In the context of test cases, domination can be expressed by the following relation:

Definition 1. A test case x dominates another test case y, also written $x \prec y$ if and only if ...

Algorithm 1: NSGA-II

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
output : A test suite T

```
1  begin
2      t ← 0
3      P_t ← RANDOM-POPULATION(M)
4      while not(search_budget_consumed) do
5          Q_t ← GENERATE-OFFSPRING(P_t)
6          R_t ← P_t ∪ Q_t
7          F ← FAST-NONDOMINATED-SORT(R_t)
8          P_{t+1} ← 0
9          d ← 1
10         while (|P_{t+1}| + |F_d| ⩽ M) do
11             CROWDING-DISTANCE-ASSIGNMENT(F_d)
               P_{t+1} ← P_{t+1} ∪ F_d
12             d ← d + 1
13         Sort(F_d) // according to the crowding distance
14         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
15         t ← t + 1
16     S ← t + 1
```

The NSGA-II algorithm works as follows:

- Starting from an initial population of individuals Pt, generate an offspring population Qt of equal size and merge the two together, obtaining the population Rt.
- Perform non-dominated sorting of the individuals in Rt based on target indicators and classify them by fronts, i.e. they are sorted according to an ascending level of non-domination. This ensures that the top Pareto-optimal individuals will survive to the next generation.
- If one of the fronts in the sorted sequence doesn't fit in terms of population size, crowding distance sorting is performed.

- Create the new population based on crowded tournament selection, then perform crossover and mutation.

Figure 2.2 summarizes the main loop of the algorithm:

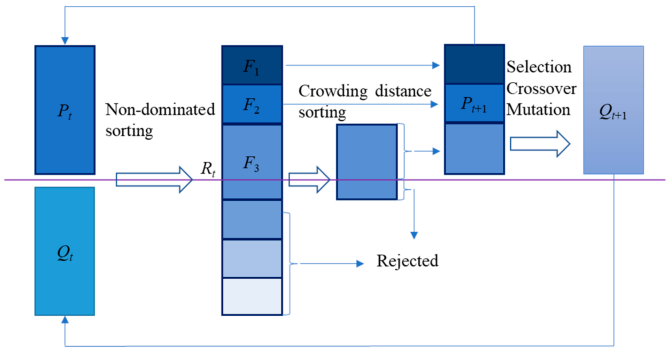

Figure 3.1: NSGA-II algorithm main loop

In the context of software engineering, NSGA-II has been applied to problems such as software refactoring and test case prioritization, with two or three objectives. If the number of objectives begins to grow, however, the performance of the algorithm doesn't scale up well [**DBLP:journals/csur/LiLTY15**]. To overcome these limitations, there have been various adjustments and optimization of this algorithm...

3.3 SPEA2

Strength Pareto Evolutionary Algorithm (SPEA2) Similarly to NSGA-II, SPEA2 isn't suitable for problems with more than three objectives. []

3.4 MOSA

MOSA [**DBLP:conf/icst/PanichellaKT15**] is a GA proposed as an improvement over NSGA-II and SPEA2 for many-objective test case generation. With this algorithm, coverage of the different branches makes up the set of objectives to be optimized. The main characteristic about MOSA is that instead of ranking candidates for selection based on their Pareto optimality, it uses a preference criterion; this criterion selects the test case with the lowest objective score for each uncovered target; these selected individuals

are given a higher chance of survival, while other test cases are ranked with the traditional NSGA-II approach.

As the first step, MOSA starts from a randomly generated initial population of test cases. To create the next generation, MOSA generates offsprings by implementing the classic operations of selection, crossover and mutation. Before selection occurs however, for each uncovered branch, the test case with the lowest objective score(branch distance + approach level) is determined; these test cases will have assigned the rank 0 and will make up the first front. The remainder of the test cases will be sorted according to the traditional NSGA-II approach.

After the rank assignment step is complete, the crowding distance is calculated to determine which individuals to select: the individuals with higher distance from the rest are given a higher chance of being selected. The algorithm attempts to select as many test cases as possible, starting from front 0, until the population size is reached.

Algorithm 2: MOSA

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
output : A test suite T

```
1  begin
2      t ← 0
3      P_t ← RANDOM-POPULATION(M)
4      archive ← UPDATE-ARCHIVE(P_t, ∅)
5      while not(search_budget_consumed) do
6          Q_t ← GENERATE-OFFSPRING(P_t)
7          archive ← UPDATE-ARCHIVE(Q_t, archive)
8          R_t ← P_t ∪ Q_t
9          F ← PREFERENCE-SORTING(R_t)
10         P_{t+1} ← 0
11         d ← 0
12         while (|P_{t+1}| + |F_d| ⩽ M) do
13             CROWDING-DISTANCE-ASSIGNMENT(F_d)
               P_{t+1} ← P_{t+1} ∪ F_d
14             d ← d + 1
15         Sort(F_d) // according to the crowding distance
16         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
17         t ← t + 1
18     T ← archive
```

Algorithm 3: PREFERENCE-SORTING

input : $T = \{t_1, ..., t_n\}$, a set of candidate test cases
Population size M
output : A non-dominated ranking assignment F

```
1  begin
2      F_0 // first front
3      for u_i ∈ U and u_i is uncovered do
4          // for each uncovered test target, the best test case according
             to the preference criterion is selected
5          t_best ← test case in T with minimum objective score for u_i
6          F_0 ← F_0 ∪ {t_best}
7      T ← T − F_0
8      if |F_0| > M then
9          F_1 ← T
10     else
11         U* ← {u ∈ U : u is uncovered}
12         E ← FAST-NONDOMINATED-SORT(T , {u ∈ U*})
13         d ← 0
14         for All non-dominated fronts in E do
15             F_{d+1} ← E_d
```

Algorithm 4: DOMINANCE-COMPARATOR

input : Two test cases to compare, $t1$ and $t2$
$U = \{u_1, ..., u_m\}$ the set of coverage targets of a program

```
1  begin
2      dominates1 ← false
3      dominates2 ← false
4      for u_i ∈ U and u_i is uncovered do
5          f_i^1 ← values of u_i for t_1
6          f_i^2 ← values of u_i for t_2
7          if f_i^1 < f_i^2 then
8              dominates1 ← true
9          if f_i^2 < f_i^1 then
10             dominates2 ← true
11         if dominates1 == true and dominates2 == true then
12             break
13     if dominates1 == dominates2 then
14         // t1 and t2 don't dominate each other
15     else
16         if dominates1 == true then
17             // t1 dominates t2
18         else
19             // t2 dominates t1
```

Finally, MOSA uses an archivee population that keeps track of the best performing test cases, in order to form the final test suite. The archive is continuously updated as follows:

Algorithm 5: UPDATE-ARCHIVE

input : A set of candidate test cases T
An archive A
output : An updated archive A

```
1  begin
2      for u_i ∈ U do
3          t_best ← ∅
4          best_length ← ∞
5          if u_i already covered then
6              t_best ← test case in A covering u_i
7              best_length ← number of statements in t_best
8          for t_j ∈ T do
9              score ← objective score of t_j for target u_i
10             length ← number of statements in t_j
11             if score == 0 and length ≤ best_length then
12                 replace t_best with t_j in A
13                 t_best ← t_j
14                 best_length ← length
15     return A
```

3.5 DynaMOSA

One of the limitations of MOSA is its inability to consider the implicit dependencies between targets, which simply appear as independent objectives to optimize. Such dependencies can be quite common in practice: most commonly, there exist branches which may only be satisfied if and only if other branches in the outer scope have been already covered.

DynaMOSA, Dynamic Many-Objective Sorting Algorithm [**DBLP:journals/tse/PanichellaKT18**] is an algorithm that focuses on these dynamic dependencies, and has been proposed as an evolution of MOSA. Before introducing the algorithm, a few definitions are needed [**DBLP:journals/tse/PanichellaKT18**]:

Definition 2. (Dominator): A statement $s1$ dominates another statement $s2$ if every execution path to $s2$ passes through $s1$.

Definition 3. (Post-dominator): A statement $s1$ post-dominates another statement $s2$ if every execution path from $s2$ to the exit point passes through $s1$.

Definition 4. (Control dependency): There is a control dependency between program statement $s1$ and program statement $s2$ iff: (1)$s2$ is not a postdominator of $s1$, and (2) there exist a path in the control flow graph between $s1$ and $s2$ whose nodes are postdominated by $s2$.

Definition 5. (Control dependency graph): The graph $G = \langle N, E, s \rangle$, consisting of nodes $n \in N$ that represent program statements, and edges $e \in E \subseteq NXN$ that represent control dependencies between program statements, is called control dependency graph. Node $s \in N$ represents the entry node, which is connected to all nodes that are not under the control dependency of another node.

This definition can be extended to other coverage criteria. For example, given two branches $b1$ and $b2$, we say that $b1$ holds a control dependency on $b2$ if $b1$ is postdominated by a statement $s1$ which holds a control dependency on a node $s2$ that postdominates $b2$.

DynaMOSA uses the control dependency graph to identify which targets are independent from each other and which ones can be covered only after satisfying previous targets in the graph. Algorithm 6 highlights the test case evolution in DynaMOSA.

Algorithm 6: DynaMOSA

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
$G = \langle N, E, s \rangle$: control dependency graph of the program
$\phi : E \rightarrow U$: partial mapping between edges and targets
output : A test suite T

```
1  begin
2      U* ← targets in U with not control dependencies
3      t ← 0
4      P_t ← RANDOM-POPULATION(M)
5      archive ← UPDATE-ARCHIVE(P_t, ∅)
6      U* ← UPDATE-TARGETS(U*, G, φ)
7      while not(search_budget_consumed) do
8          Q_t ← GENERATE-OFFSPRING(P_t)
9          archive ← UPDATE-ARCHIVE(Q_t, archive)
10         U* ← UPDATE-TARGETS(U*, G, φ)
11         R_t ← P_t ∪ Q_t
12         F ← PREFERENCE-SORTING(R_t, U*)
13         P_{t+1} ← 0
14         d ← 0
15         while (|P_{t+1}| + |F_d| ⩽ M) do
16             CROWDING-DISTANCE-ASSIGNMENT(F_d, U*)
               P_{t+1} ← P_{t+1} ∪ F_d
17             d ← d + 1
18         Sort(F_d) // according to the crowding distance
19         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
20         t ← t + 1
21     T ← archive
```

The main difference with MOSA is how the initial target population is selected; with DynaMOSA, only the targets that are free from dependencies are part of this set. Then, in each iteration, the set of non-dependent targets is updated by using the following procedure:

Algorithm 7: UPDATE-TARGETS

input : $G = \langle N, E, s \rangle$: control dependency graph of the program
$U^* \subseteq U$: current set of targets
$\phi : E \to U$: partial mapping between edges and targets
output : U^*: updated set of targets to optimize

```
1  begin
2      for u ∈ U do
3          if u is covered then
4              U* ← U* − {u}
5              e_u ← edge in G for the target u
6              visit(e_u)
7  function visit(e_u ∈ E):
8      for each unvisited e_n ∈ E control depenmdent on e_u do
9          if φ(e_n) is not covered then
10             U* ← U* ∪ {φ(e_n)}
11             set e_n as visited
12         else
13             visit(e_m)
```

This routine update the sets of selected targets U^* in order to include any uncovered targets that are control dependent on the newly covered target. In the case of newly covered targets, the procedure iterates over the control graph to find all control dependent targets.

3.6 OCELOT

Optimal Coverage sEarch-based tooL for sOftware Testing, OCELOT [**DBLP:conf/ssbse/ScalabrinoC**
is a test case generation tool for C programs that implements both a multi-target approach based on MOSA, and new iterative single-target appproach named LIPS, Linear Independent Path-based Search.

Tested with MOSA but not with DynaMOSA.

Chapter 4

Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Chapter 5

Math reference

1. $$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \lim_{n\to\infty}\frac{n}{\sqrt[n]{n!}}$$

2. $$e = \sum_{n=0}^{\infty}\frac{1}{n!}$$

3. $$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{2}{3 + \cfrac{3}{4 + \ddots}}}}$$

4. $$\int_a^b f(x)dx$$

5. $$x_1, x_2, ..., x_n$$