Eötvös Loránd University
Faculty of Informatics
Dept. of Computer Algebra

General Predicate Testing

Kovács Attila
Professor, Ph.D.

Andi Péter
Computer Science MSc

Budapest, 2023

Contents

1 Introduction

TODO: Review this before finalizing everything, all of this should be present in the thesis

In this thesis I'll introduce my automatic test case generation algorithm, the implementation of General Predicate Testing proposed by Kovacs Attila and Forgacs Istvan [1]. I'll outline the current state-of-the-art black box testing strategies, what their advantages and disadvantages are. I'll explain what GPT is, how it works in theory, and how I put it into practice. I'll show my proposed Domain Specific Language for formalizing the predicates of requirements. I'll show how this automatic test generation scales much better than the current state-of-the-art manual test generation techniques.

1.1 Common testing practices

Software testing is an essential part of the software development life cycle. Testing software allows us to be confident, that the program adheres to the requirements and works as expected. There can be functional and non-functional requirements, in this thesis I'll focus on functional requirements.

There are multiple approaches to testing software, one is Black Box testing [2], where we don't create tests from the already written source code, but we treat it as a black box and create the tests from the requirements. This approach allows us to be unbiased and focus on the requirement set.

1.2 Test case generation

2 Other test case generation methods

2.1 Equivalence Partitioning

2.2 Boundary Value Analysis

2.3 Random testing

3 Intervals

Intervals are the backbone of GPT. Instead of assigning single point to Equivalence Partitions, we use intervals. This way we can represent every possible value that we want to test our predicates with.

There weren't any off-the-shelf libraries that implemented interval handling exactly in the way I needed, so I had to create my own interval library. It consists of two main parts: Simple intervals and what I call Multiintervals. Multiintervals are made up of multiple simple intervals, but behave as an interval.

I will use the word interval both for simple intervals, or multiintervals. If a distinction needs to be made, I'll clarify. But in later chapters all that'll matter is that we are working with an interval.

This interval implementation has three important functions: intersection, union, and complement. With these I could implement all of GPT's functionality.

TODO: There was one Rust lib that had pretty good single intervals

3.1 Simple Intervals

Simple intervals are intervals in a mathematical sense. It has two endpoints, a lo (low) and a hi (high). It has two boundaries: a lo boundary and a hi boundary. A boundary is either closed [ ] and open ( ).

A special case is when an interval is unbounded. I'll use the notation of ∞ at and endpoint to mean that that side of the interval is unbounded, like: $[10, \infty)$.

An interval can contain only a single point: $[10, 10]$.

Empty intervals can exist when no points can be in an interval, like $(10, 10)$ or $(10, 10]$

An interval can only be constructed if $\text{lo} \leq \text{hi}$.

For the implementation, I'm only sotring the lo_boundary, lo, hi, hi_boundary variables. All the calculations are made with these values.

TODO: A design alternative could be to represent all the possible states interval could be in in an Algebraic Data Type, so we cannot create inconsistent state, like $[-\infty, \infty]$ or $(10, 10]$

3.1.1 Intersection

Can intersect?

Pseudocode for detecting whether the intersection of two intervals is possible:

```
case1 = self.lo > other.hi || other.lo > self.hi
case2 = self.lo == other.hi && (self.lo_boundary == Open ||
other.hi_boundary == Open)
case3 = other.lo == self.hi && (other.lo_boundary == Open ||
self.hi_boundary == Open)

return !(case1 || case2 || case3)
```

Here, we are looking at if the intervals don't intersect, because those are the easier cases to handle. We can negate this result to get when intervals intersect.

`case1` is when the intervals are above or below each other, like $[0, 10]$ and $[20, 30]$ or $[20, 30]$ and $[0, 10]$. Because $\text{lo} \leq \text{hi}$ we only have to check the lo and hi of the two intervals.

`case2` is when the intervals would have the same endpoint, but either one of the boundaries is Open. Because an open boundary doesn't contain that point, intersection can't be made as well. For example $[0, 10]$ and $(10, 20)$.

`case3` is the same as `case2`, but checking the intervals from the other way.

In all other cases the intervals would intersect.

Calculating the intersection

Pseudocode for calculating the intersection of two intervals `self` and `other`:

TODO: Here we need a lo_cmp and a hi_cmp instead of the > or <, because if they have the same value the boundary makes the difference. Also, when comparing intervals (for intersection, union, or complement) it would make the cases more explicit.

```
if self doesn't intersects_with other:
   return No Intersection

interval_with_lower_hi = if self.hi > other.hi then self else other
interval_with_higher_lo = if self.lo < other.lo then self else other

return Interval {
   lo_boundary: interval_with_higher_lo.lo_boundary
   lo: interval_with_higher_lo.lo
   hi: interval_with_lower_hi.hi
   hi_boundary: interval_with_lower_hi.hi_boundary
}
```

First we check if the intervals can even be intersected. If they don't, we return that there is no intersection. In Rust this is an `Option::None` type, in other languages it might be a `null`.

Then, if the intervals can be intersected, we look for the hi endpoint which is lower and use that as the hi point and hi boundary. We do the same and look for higher lo point and use that as a lo point.

3.1.2 Union

The union of two simple intervals can either be a simple interval (if they intersect) or a multiinterval if they don't. Because of this, we always assume that the union of two simple intervals will be a multiinterval. There is no need for the code to produce a union of simple intervals that is a simple interval, but it could be implemented.

Pseudocode:

```
return Multiinterval::from_intervals([self, other])
```

We can just create a multiinterval from the two intervals. With this constructor Multiinterval will call a `clean()` on itself, which I'll explain later. In short, in this case `clean()` would merge the two intervals if they intersect.

3.1.3 Complement

TODO: There is no inverse defined for Simple intervals lol. (I only needed complements for Multiintervals and that doesn't need the complements of simple intervals to work)

3.2 Multiintervals

Multiintervals are intervals composed of multiple simple intervals. They are required for GPT, because for example if we have a predicate that states that $x > 0 \land x \leq 10 \land x \notin \{5, 7\}$ we could represent the interval of values x could take as the the multiinterval $(0, 5)$ $(5, 7)$ $(7, 10]$.

An empty multiinterval is one which has no intervals. We don't store empty intervals in mutliintervals.

An invariant of the Multiinterval is that its intervals are sorted in increasing order.

3.2.1 Cleaning

There could be multiintervals, which are not 'clean', or not in a sematically correct form. Take for example $(10, \infty)$ $(-5, 5)$ $[0, 20]$.

Here are the steps to clean a multiinterval:

1. **Removing empty intervals.** Empty intervals hold no values, so they are unnecessary to have in a multiinterval. From the example we'd remove $(-5, -5)$
2. **Sorting the intervals.** Intervals should be in an increasing order inside a multiinterval. When comparing intervals we compair their los. The example would change from $(10, \infty)$ $[0, 20]$ to $[0, 20]$ $(10, \infty)$.
3. **Merging overlapping intervals.** If intervals would would intersect, we can merge them together. The example would become from $[0, 20]$ $(10, \infty)$ to $[0, \infty)$.

We can define a constructor `Multiinterval::from_intervals` that will always create a clean multiinterval from a list of intervals:

```
def Multiinterval::from_intervals(intervals):
   multiinterval = new Multiinterval(intervals)

   multiinterval.clean()

   return multiinterval
```

3.2.2 Intersection

Can intersect? To check that two mutliintervals intersect, we can check if any of their intervals intersect. Pseudocode:

```
for x in self.intervals:
   for y in other.intervals:
      if x.intersects_with(y):
         return True

return false
```

TODO: Because the intervals are in increasing order, we could do an $O(n)$ algorithm instead of an $O(n^2)$

Calculating the intersection

Pseudocode:

```
intersected_intervals = []

for x in self.intervals:
   for y in other.intervals:
      if x.intersects_with(y):
         intersected_intervals += x.intersect(y)

return Multiinterval::from_intervals(intersected_intervals)
```

We try to intersect all the intervals. The `Multiinterval::from_intervals` constructor will call a `clean()`, so the resulting intersected Multiinterval will be in the correct form.

3.2.3 Union

We take both Multiintervals' intervals, concatenate them and create a Multiintevral from them. This constructor call `clean()`, so it'll take care of sorting and overlapping intervals.

```
return Multiinterval::from_intervals(self.intervals ++ other.intervals)
```

3.2.4 Complement

The complement of an interval contains all the elements which are not in the interval. In simple terms, we just return all the 'space' between our intervals.

For example:

- Multiinterval: $[-42, 3)$ $(3, 67)$ $(100, 101)$ $[205, 607]$ $(700, \infty)$
- Complement: $(-\infty, -42)$ $[3, 3]$ $[67, 100]$ $[101, 205)$ $(607, 700]$.

Pseudocode:

```
if self.is_empty()
   return (-infinity, infinity)

complement_intervals = []

if intervals.lowest_lo() != -infinity:
    complement_intervals += Interval(Open, -infinity, lowest_lo, lowest_boundary)

for [a, b] in self.intervals.window(2):
   complement_intervals += Interval(a.hi_boundary.complement(), a.hi, b.lo, b.lo_boundary.complement())

if intervals.highest_hi() != -infinity:
    complement_intervals += Interval(highest_boundary, highest_hi, infinity, Open)

return new Multiinterval(complement_intervals)
```

We go through the intervals in pairs. The `.window(2)` function returns all the neighbouring pairs in a list. It is a sliding window. We always create an interval between the hi of the first and the lo of the second interval, as this is the space not covered by our multiinterval.

As in the sliding window we only look at the hi of the left side and the lo of the right side, we have to handle the case at the edges of the multiinterval. If our multiinterval is not unbounded, the complement has to be unbounded.

We don't have to clean the Multiinterval, because of its invariants it won't have empty intervals, it will be sorted, and it won't have overlapping intervals.

4 GPT Algorithm

TODO: Explain precision **TODO:** In this doc when we refer to intervals, we could mean multiintervals, for the sake of simplicity.

4.1 BVA in GPT (in, inin, on, off out)

4.1.1 Converting conditions to intervals

In GPT we create intervals from conditions, similar to Equivalence Partitioning.

Example: For the condition $x < 10$, we create the interval $(-\infty, 10)$

TODO: Explain the condition -> to interval process

An interesting case happens, when we take the not equal to condition. For $x \neq 10$ we have to generate a multiinterval, because the values x could take is $(-\infty, 10)$ $(10, \infty)$.

4.1.2 Extending BVA

Now that we have intervals to work with, we should generate the possible test values. In normal BVA we would pick single points from the intervals. GPT extends this in two ways:

1. We don't pick single values, but the the largest possible intervals. This will be helpful, when in the end we try to reduce the number of test cases, we can see the overlap between different test cases.
2. We not only look at the minimum and maximum possible acceptable value, but more. We look at not acceptable values and test that our implementation properly fails for those. **TODO:** This might be covered in BVA, further investigation needed.

TODO: Actually, GPT is an implementation of BVA with some extra functionalities, right?

4.1.3 Equivalence Partitioning in GPT

In BVA the equivalence partitions for $[1, 10)$ would be $(-\infty, 1)$ $[1, 10)$ $[10, \infty)$

In GPT we have more equivalence partitions. **TODO:** Explain why

- **In:** The interval of the acceptable values. In case of Open ends we step one with the precision to make it closed. Unbounded parts remain unbounded.

Example: $[1, 10)$ will have the In of $[1, 9.99]$
Example: $(-\infty, 10]$ will have the In of $(-\infty, 10]$

- **InIn:** One step of precision inside In.

Example: $[1, 10)$ will have the InIn of $[1.01, 9.98]$
Example: $(-\infty, 10]$ will have the InIn of $(-\infty, 9.99]$

- **On:** First possible acceptable values from the edges. These are single points, the endpoints of In. Unbounded edges have no On points. If the interval is a single point there will only be one On point.

Example: $[1, 10)$ will have the On points of $[1, 1]$ $[9.99, 9.99]$
Example: $(-\infty, 10]$ will have the On point of $[9.99, 9.99]$

- **Off:** First not acceptable values from the edges. One step outside the edges of In. Unbounded edges have no Off points.

Example: $[1, 10)$ will have the Off points of $[0.99, 0.99]$ $[10, 10]$
Example: $(-\infty, 10]$ will have the Off point of $[10.01, 10.01]$

- **Out:** Not acceptable values, except for the Off points. The complement of the In interval, stepped one, to exclude the Off points.

Example: $[1, 10)$ will have the Out interval of $(-\infty, 0.98]$ $[10.01, \infty)$
Example: $(-\infty, 10]$ will have the Off interval of $[10.01, \infty)$

TODO: Here the In interval and On points are overlapping, why the "duplication"? Because in GPT when we create the Off and Out intervals we only do it for one variable in a test case. That way we only test that that variable is handled correctly in the logic. All the other variables will have the In intervals, so they are accepted.

For multiintervals we use the same equivalence partitioning technique. We calculate the partition for all the intervals inside the multiinterval and create a multiinterval out of the partitions.

4.2 Test case generation in GPT

We can generate multiple intervals with GPT that we can select test values from. But this is just for one predicate. To test all the predicates in a condition, we use the following technique:

- Creating an NTuple from the condition.
- Generating the IN, ININ, and ON values for each variable in the NTuple.
- Generating the OFF and OUT values for each variable in the NTuple.

Let's look at each of those steps in detail:

4.2.1 Creating an NTuple from the condition

An NTuple is a tuple with N elements. In GPT I use the term NTuple for a map of the variable names to the EPs. This is because of historical reasons, originally these were literal tuples, but working with an explicit variable to EP mapping is easier to handle during graph reduction.

In GPT we can inly create NTuples from a condition that only contains conjunctions. I'll explain how to convert a condition with disjunctions to conjunctive forms in a later chapter. **TODO:** ref that chapter.

Example: The condition $x < 10$ && y in $[0, 20]$ && $z ==$ true would become the following NTuple:

```
{
   x: (-Inf, 10)
   y: [0, 20]
   z: true
}
```

4.2.2 Generating the IN, ININ, and ON values for each variable in the NTuple

Now we have an NTuple with variables that have EPs associated with them. We take the NTuple and for each variable we generate the In, InIn, and On values for each. Because the values are acceptable values, we can group them together and check all of the variables' values in one NTuple. This won't be the case for Off and Out.

One complication is, that On (and later Off) can generate multiple values. When we have could have multiple values for variables, we take the Cartesian product of the possibilities. This is further complicated by the fact that we could have N variables with M possible values and we'd have to take the Cartesian product of all of those.

Example from the previous NTuple:

```
In:   { x: (-Inf, 9.99], y: [0, 20], z: true }
InIn: { x: (-Inf, 9.98], y: [0.01, 19.99], z: true }
On:   { x: 9.99, y: 0 and 20, z: true }
The Cartesian product of this is { x: 9.99, y: 0, z: true } and { x:
9.99, y: 20, z: true }
```

Example 2: Let's look at the On and Cartesian product for the following NTuple:
`{ a: [0, 10], b: [20, 30] }`
The On values would be: `{ a: 0 and 10, b: 20 and 30 }` The Cartesian products of this is:

```
{ a: 0, b: 20 }
{ a: 0, b: 30 }
{ a: 10, b: 20 }
{ a: 10, b: 30 }
```

4.3 Hierarchical GPT

5 GPT Lang

So far, we've looked at GPT as a test generation technique. The other power of my GPT implementation is that I've developed a Domain Specific Lanuage (DSL) for defining requirements for GPT. From this DSL GPT can generate test cases. This makes the test generation process much faster. It is called GPT Lang.

GPT Lang has a C inspired syntax, to make it easier for developers to learn. Because for GPT we are only concerned with predicates, we can only write conditions in this DSL.

Let's look at an example:

```
var vip: bool
var price: num(0.01)
var second_hand_price: int

if(VIP == true && price < 50) {
   if(second_hand_price == 2)
   if(second_hand_price in [6,8])
}

if(vip == false && price >= 50)
else if(vip == true || !(price >= 50 && second_hand_price >= 20)) {
   if(30 < price && 60 < second_hand_price)
}
else
```

As you can see, GPT Lang looks similar to how we will actually implement our programs. This can be useful in more things. First, after we implement the requirements in GPT Lang and generate our test cases for Test Driven Development (TDD) we have a starting point for coding in other languages, as we have basically defined the control flow in GPT Lang. Also, if we have an existing codebase, we can easily test it with GPT, because we can convert the existing control flow to GPT Lang easily.

You can currently do the following things in GPT Lang:

- Declare variables with boolean or number types (optionally with precision)
- Declare `if`, `else if`, and `else` statements, with an optional body that can have other `if` statements. They can have any number of conditions.
- Declare conditions, which can be
  - Boolean true or false
  - Number $>, \geq, <, \leq, ==, \neq$ constant
  - Number `in` or `not in` interval
- Conditions can be negated with `!`, grouped with parenthasies `( )`, conjuncted with && or disjuncted with `||`.

5.1 Syntax

Numbers

Numbers can be either integers, floating point numbers, or `Inf` (for infinity). They can be negative.

Example.: `146`, `3.14`, `-6390`, `-Inf`, `0.01`

Type

Types can be:
- `bool`: Boolean.
- `int`: Integer.
- `num`: Number with default precision of 0.01.
- `num(<precision>)`: Number with the given precision. `<precision>` must be a positive number and not `Inf`.

Example: `bool`, `num`, `num(0.001)`, `num(1)`

Variable declaration

```
var <var_name>: <type>
```

Examples:
- `var price: num(0.01)`
- `var isVIP: bool`
- `var year: int`

Interval

```
<lo_boundary> <lo>, <hi> <hi_boundary>
```

Where
- `<lo_boundary>` is `(` or `[`
- `<lo>` and `<hi>` are numbers
- `<hi_boundary>` is `)` or `]`

Example: `(-Inf, 0]`, `[2.3, 6.75)`, `[1,1]`

Condition

Conditions represent **TODO:** something

Boolean condition:
`<var_name> <op> <true|false>` or `<true|false> <op> <var_name>`
Where `<op>` is `==` or `!=`

Binary number condition:
`<var_name> <op> <constant>` or `<constant> <op> <var_name>`

Interval condition:
`<var_name> <in|not in> <interval>`

TODO: Give examples here for conditions

TODO: Give example about && || () and !()

5.2 Parsing to the AST

5.3 Converting the AST to IR

5.4 Flattening the IR

6 Graph Reduction

6.1 Why even graph reduce

6.2 MONKE

6.3 Least Losing Nodes

6.4 Least Losing Edges

7 Code architecture

7.1 Rust

7.2 Frontend app, webassembly

8 Future improvement ideas

8.1 Coincidental Correctness

Hierons, R. M. has shown in [3], that because BVA only generates single points on boundaries, geometric shifts in boundaries can lead to accidental correctness. In other words, there are some incorrect implementations, which cannot be caught with BVA.

Consider the following example: Write a program that accepts a point as an (x, y) coordinate, x and y are integers. Return true, if the point is above the $f(x) = 0$ function's slope, otherwise return false.

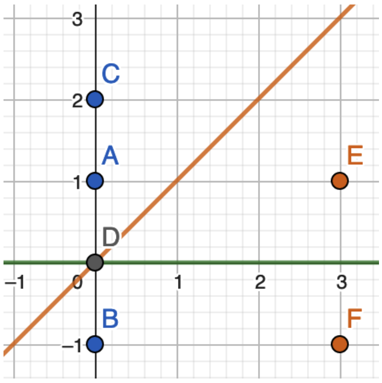
From this, we'd write it in GPT Lang as:

```
var x: int
var y: int
if(y > 0) else
```

You can notice, that we don't reference x here, so the generated test cases won't care for x either. If we default x to 0 in all test cases, we'd have the following reduced generated test cases: `A = { x: 0, y: 1 }, B = { x: 0, y: (-Inf, -1] }, C = { x: 0, y : [2, Inf) }, D = { x: 0, y: 0 }`

The problem with this, is that if the implementation checked against the $f(x) = x$ function's slope, all of our test cases would still pass. This is, because in the $x = 0$ point both $f(x) = 0$ and $f(x) = x$ behave in the same way. But we know that checking against $f(x) = x$ would be an incorrect implementation.

The following figure shows the two functions, $f(x) = 0$ in green and $f(x) = x$ in orange. In this scenario the test points A, B, C, and D see that both functions behave in the same way. But point E and F could show the bug, because they are both under the slope of $f(x) = x$.



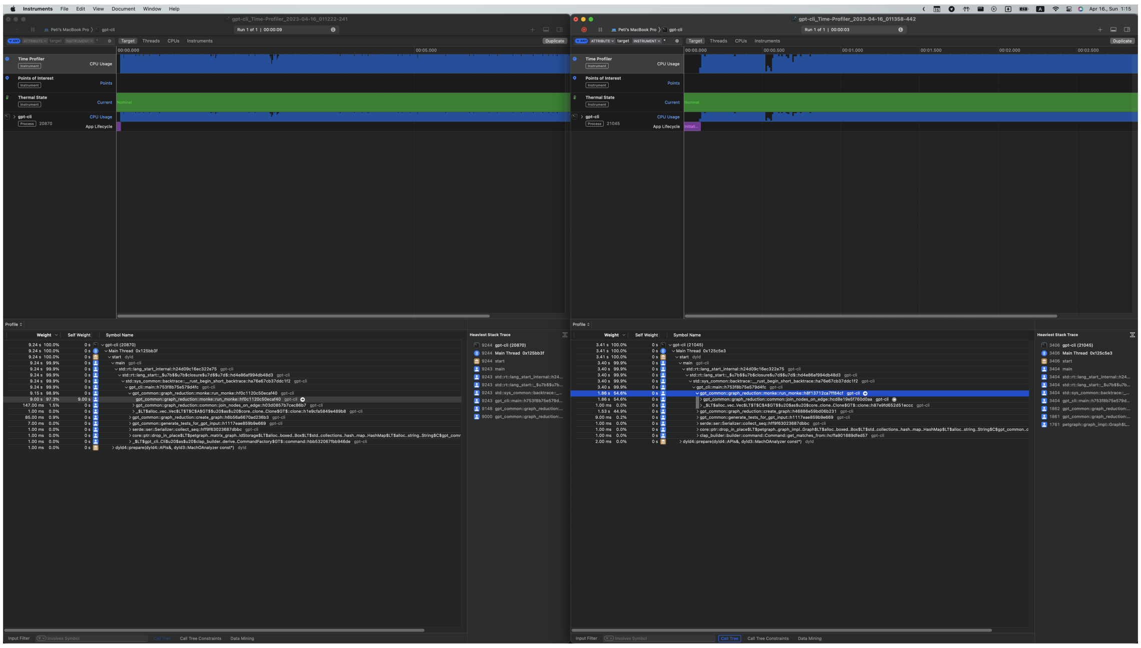
8.2 Linear Predicates

GPT can currently only handle predicates with one variable. Linear predicates are not supported.

9 TODOs

- Quote The Book
- The Book p23-24, replicate this test
- Give example about the usage of GPT, what bugs it can catch
    - Heck, dedicate a whole chapter to it, it is important
- The Book p69 is about GPT [1, p. 69]

Matrix graph was tried out, but it had 4 times worse performance. It is probably because we have to find edges in the graph for MONKE and the sparcer the matrix gets the harder that is.



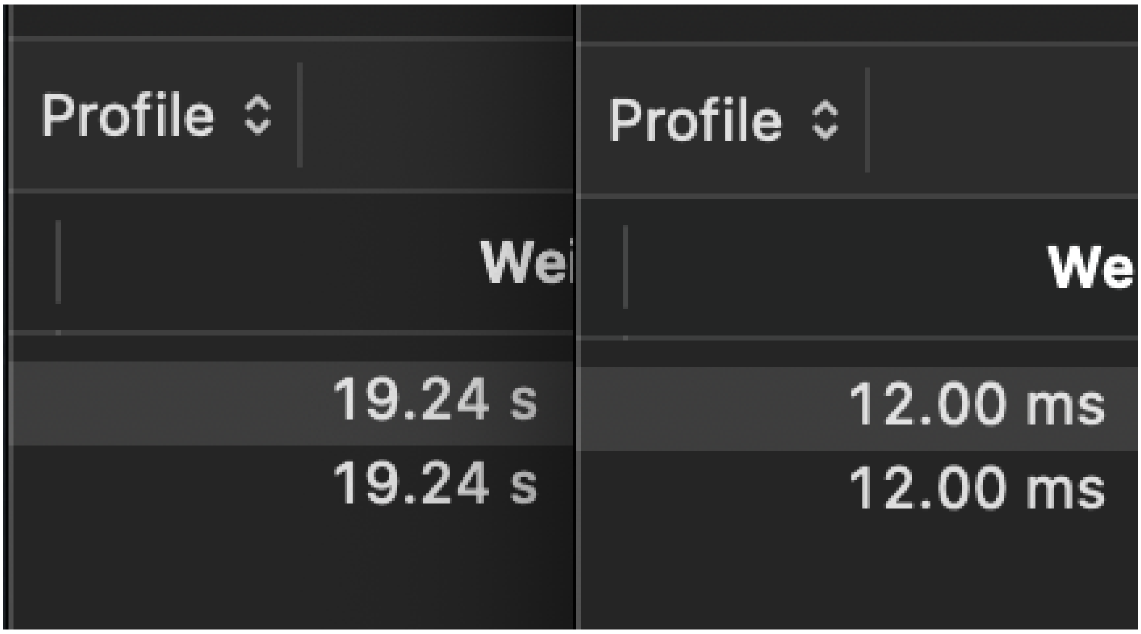
```
Weight          Self Weight   Symbol Name
9.24 s  100.0%        0 s     ⌄ gpt-cli (20870)
9.24 s  100.0%        0 s      ⌄ Main Thread  0x125bb3f
9.24 s  100.0%        0 s       ⌄ start  dyld
9.24 s   99.9%        0 s        ⌄ main  gpt-cli
9.24 s   99.9%        0 s         ⌄ std::rt::lang_start_internal::h24d09c16ec322e75  gpt-cli
9.24 s   99.9%        0 s          ⌄ std::rt::lang_start::_$u7b$$u7b$closure$u7d$$u7d$::hd4e86af994db48d3  gpt-cli
9.24 s   99.9%        0 s           ⌄ std::sys_common::backtrace::__rust_begin_short_backtrace::ha76e67cb37ddc1f2  gpt-cli
9.24 s   99.9%        0 s            ⌄ gpt_cli::main::h753f8b75e579d4fc  gpt-cli
9.15 s   98.9%        0 s             ⌄ gpt_common::graph_reduction::monke::run_monke::hf0c1120c50ecaf40  gpt-cli
9.00 s   97.3%     9.00 s                gpt_common::graph_reduction::monke::run_monke::hf0c1120c50ecaf40  gpt-cli
147.00 ms  1.5%       0 s              > gpt_common::graph_reduction::common::join_nodes_on_edge::h03d0857b7cec86b7  gpt-cli
1.00 ms   0.0%        0 s              > _$LT$alloc..vec..Vec$LT$T$C$A$GT$$u20$as$u20$core..clone..Clone$GT$::clone::h1e9cfa5849e489b8  gpt-cli
85.00 ms  0.9%        0 s             > gpt_common::graph_reduction::create_graph::h6b56a6670ad236b3  gpt-cli
7.00 ms   0.0%        0 s             > gpt_common::generate_tests_for_gpt_input::h1117eae859b9e669  gpt-cli
1.00 ms   0.0%        0 s             > serde::ser::Serializer::collect_seq::hff9f63023687dbbc  gpt-cli
1.00 ms   0.0%        0 s             > core::ptr::drop_in_place$LT$petgraph..matrix_graph..IdStorage$LT$alloc..boxed..Box$LT$std..collections..hash..map..HashMap$LT$alloc..string..String$C$gpt_comr
1.00 ms   0.0%        0 s             > _$LT$gpt_cli..Cli$u20$as$u20$clap_builder..derive..CommandFactory$GT$::command::hbb532067fbb946de  gpt-cli
1.00 ms   0.0%        0 s        > dyld4::prepare(dyld4::APIs&, dyld3::MachOAnalyzer const*)  dyld
```

```
Heaviest Stack Trace
9244  gpt-cli (20870)
9244  Main Thread  0x125bb3f
9244  start
9243  main
9243  std::rt::lang_start_internal::h24...
9243  std::rt::lang_start::_$u7b$$u7b...
9243  std::sys_common::backtrace::_...
9243  gpt_cli::main::h753f8b75e579d...
9148  gpt_common::graph_reduction:...
9000  gpt_common::graph_reduction:...
```

```
Weight          Self Weight   Symbol Name
3.41 s  100.0%        0 s     ⌄ gpt-cli (21045)
3.41 s  100.0%        0 s      ⌄ Main Thread  0x125c5e3
3.41 s  100.0%        0 s       ⌄ start  dyld
3.40 s   99.9%        0 s        ⌄ main  gpt-cli
3.40 s   99.9%        0 s         ⌄ std::rt::lang_start_internal::h24d09c16ec322e75  gpt-cli
3.40 s   99.9%        0 s          ⌄ std::rt::lang_start::_$u7b$$u7b$closure$u7d$$u7d$::hd4e86af994db48d3  gpt-cli
3.40 s   99.9%        0 s           ⌄ std::sys_common::backtrace::__rust_begin_short_backtrace::ha76e67cb37ddc1f2  gpt-cli
3.40 s   99.9%        0 s            ⌄ gpt_cli::main::h753f8b75e579d4fc  gpt-cli
1.86 s   54.6%        0 s             ⌄ gpt_common::graph_reduction::monke::run_monke::h8f13712ca7ff84cf  gpt-cli
1.86 s   54.6%        0 s              > gpt_common::graph_reduction::common::join_nodes_on_edge::hcd8e19e5f760d0aa  gpt-cli
1.00 ms   0.0%        0 s              > _$LT$alloc..vec..Vec$LT$T$C$A$GT$$u20$as$u20$core..clone..Clone$GT$::clone::h87e9fd652d51eccc  gpt-cli
1.53 s   44.9%        0 s             > gpt_common::graph_reduction::create_graph::h46886e59bd06b231  gpt-cli
9.00 ms   0.2%        0 s             > gpt_common::generate_tests_for_gpt_input::h1117eae859b9e669  gpt-cli
1.00 ms   0.0%        0 s             > serde::ser::Serializer::collect_seq::hff9f63023687dbbc  gpt-cli
1.00 ms   0.0%        0 s             > core::ptr::drop_in_place$LT$petgraph..graph_impl..Graph$LT$alloc..boxed..Box$LT$std..collections..hash..map..HashMap$LT$alloc..string..String$C$gpt_common..d
1.00 ms   0.0%        0 s             > clap_builder::builder::command::Command::get_matches_from::hcffa901889dfed57  gpt-cli
2.00 ms   0.0%        0 s        > dyld4::prepare(dyld4::APIs&, dyld3::MachOAnalyzer const*)  dyld
```

```
Heaviest Stack Trace
3406  gpt-cli (21045)
3406  Main Thread  0x125c5e3
3406  start
3404  main
3404  std::rt::lang_start_internal::h24...
3404  std::rt::lang_start::_$u7b$$u7b...
3404  std::sys_common::backtrace::_...
3404  gpt_cli::main::h753f8b75e579d...
1862  gpt_common::graph_reduction:...
1861  gpt_common::graph_reduction:...
1761  petgraph::graph_impl::Graph$L...
```

Smaller sample:



19.24 s

19.24 s

12.00 ms

12.00 ms

Number of test cases: 250 Number of edges in initial graph: 2768

Least Losing Edges Reachable: 50 Runtime: 19.24s

MONKE: 49 Runtime: 12ms

Bibliography

[1] I. Forgacs, and A. Kovacs, *Paradigm Shift in Software Testing: Practical Guide for Developers and Testers*, Independently Published, 2022.

[2] A. P. Mathur, *Foundations of Software Testing*, Delhi, India: Prentice Hall India (a Pearson Education Asia Company), 2008.

[3] R. M. Hierons, "Avoiding coincidental correctness in boundary value analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, p. 227, Jul. 2006, doi: 10.1145/1151695.1151696. [Online]. Available: https://doi.org/10.1145/1151695.1151696