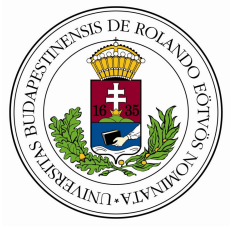


Eötvös Loránd University
Faculty of Informatics
Dept. of Computer Algebra

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]. **TODO:** This feels like they proposed the implementation. 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, where we create tests sets from the requirement specifications. In practice, this means that we're not looking at how to code is written when writing tests. This way we can systematically test the correctness of outputs for given inputs [2].

The state-of-the-art black box testing methods are: [3] [4] [1]

1. *Equivalence Partitioning*: The input and output domains can be partitioned in a way, that values in each partition belong to the same Equivalence Class. This way, test cases are only required to have one value from each partition.
2. *Boundary Value Analysis*: Test cases are created from the boundaries of Equivalence Classes. These can be the values just below, on, or just above the boundaries. This can catch usual off-by-one errors.
3. *Fuzzing*: **TODO:** Fuzz testing is used for finding implementation bugs, using malformed/semi-malformed data injection in an automated or semi-automated session.
4. *Cause-Effect Graph*: **TODO:** It is a testing technique, in which testing begins by creating a graph and establishing the relation between the effect and its causes. Identity, negation, logic OR and logic AND are the four basic symbols which expresses the interdependency between cause and effect.
5. *Orthogonal Array Testing*: **TODO:** OAT can be applied to problems in which the input domain is relatively small, but too large to accommodate exhaustive testing.

6. *All Pair Testing*: **TODO:** In all pair testing technique, test cases are designs to execute all possible discrete combinations of each pair of input parameters. Its main objective is to have a set of test cases that covers all the pairs.
7. *State Transition Testing*: **TODO:** This type of testing is useful for testing state machine and also for navigation of graphical user interface.

Next I'll explain Equivalence Class Partitioning and Boundary Value Analysis in more detail, as GPT is based on those.

1.1.1 Equivalence Class Partitioning

1.1.2 Boundary Value Analysis

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 lo $\leq$ 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 lo $\leq$ 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

Kovacs Attila and Forgacs Istvan proposed General Predicate Testing [1, p. 69]. It is a method of black box test case generation, based on and extending BVA.

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 What is Graph Reduction?

TODO: Replace vertex and vertices with nodes

When GPT generates test cases, it generates an interval (or boolean value) for variables. This means, that the discrete test points should come from those intervals. But there could be a case when multiple test cases would have intersecting intervals. For example:

- T1: `{x: [0, 10]}`
- T2: `{x: (-10, 5)}`
- T3: `{x: [0, 200]}`

For example in this case if we select $x = 2$ as out test point, it would cover all the three test cases.

This way, we can reduce the number of our total test cases, by trying to find test cases (NTuples) which intersect and calculating their intersection.

In this example, the intersection of T1 and T2 is `{x: [0, 5)}` and the intersection of that and T3 is `{x: [0, 5)}`.

6.1.1 NTuple intersection

First, I'll have to clarify, that booleans only intersect when their values are the same. So `true` intersects with `true`, and `false` with `false`. This can be derived from intervals, for example if `true` is $[0, 0]$ and `false` is $[1, 1]$. This method can also be applied to Enums (which is a future improvement idea, as detailed in **TODO:** link future improvement chapter about enums).

Two NTuples intersect, when all of their variables intersect. If a varible is not present in an NTuple, we treat it as if it could take all the possible values. In practise this means, that we just use the value of that variable from the other NTuple.

Example: The intersection of `{x: [0, 100], y: false, z: [5,5]}` and `{z: [10, 20], y: false}` is `{x: [0, 100], y: false, z: [5,5]}`.

because NTuples have intersections, in the following sections I'll give examples only with simple intervals, to keep them concise. But those examples can be used for NTuples as well.

6.1.2 Graph representation

So our goal is to intersect NTuples in a way, that in the end we have the least number of NTuples possible. We can imagine this problem as a Graph, where the vertices are the NTuples, and we have an edge between two verities, if those NTuples have an intersection.

Example:

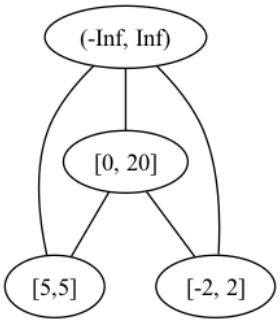


Figure 1:

What happens when we intersect two vertices?

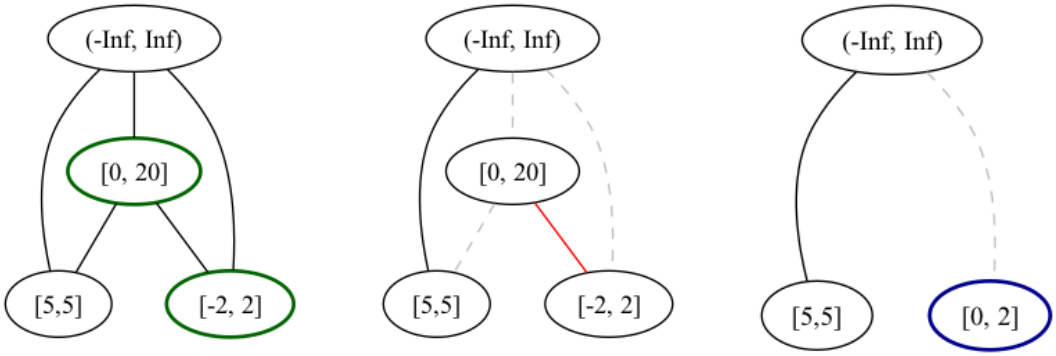


Figure 2: One intersection in Graph Reduction

In Figure 2 we can see, that we select $[0, 20]$ and $[-2, 2]$ for intersection. The detailed process for intersection:

1. We identify the edge connecting them. *(red)*
2. We identify the edges and vertices they are connected to *(gray dashed)*
3. We remove the edges connecting them to other edges *(gray dashed)* and remove the edge connecting them *(red)*.
4. We intresect the vertices, and replace the two original vertices with the new intersected vertex. *(blue)*
5. We look at the vertices they were originally connected to *(gray dashed)* and see if the new intersected vertex intersects them. If yes, we restore those edges.

In step 5 it is enough to look at the edges which were originally connected to the vertices instead of the whole graph. This is, because with intersections our intervals can only get smaller, so we'd never add new edges to the graph that weren't there before.

6.1.3 Strategies for optimising Graph Reduction

This Graph Reduction is an NP-complete problem [1, p. 116]. Because of this, we can only create algorithms that approximate the most optimally reduced graph.

As you can see, every step of our Graph Reduction basically creates a new graph. We remove and replace vertices, we remove edges. This poses a problem, because the order of reduction matters. Consider the following example: **TODO:** O-O-O-O ahol ha O-O O-O bol 2 lesz, de O O-O O bol 3 lesz

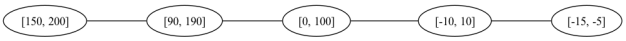


Figure 3: hmm

Here, if we'd reduce $[0, 100]$ with $[0, 10]$ we'd get $[0, 10]$, which doesn't intersect will any of the other vertices. If **TODO:** finish

If this problem would be one dimentional, as in, we only had simple intervals, we could use the $O(n \log n)$ solution proposed in [5]. But, that algorithm assumes that the intervals can be sorted in increasing order by their hi endpoints. Because we are dealing with NTuples and we could have n intervals for each interval, we can't use this algorithm, because we can't define a partial ordering on it that'd work in this case.

6.1.4 Abstracting Graph Reduction

Graph reduction can be posed in a more abstract and generic way. This allows us to concentrate more on the reduction part, without getting lost in the details.

After we construct our graph (with the intersection of NTuples), we don't actually need to know the contents of the vertices. In the previous section the definition for joining edges doesn't actually need to know whether the vertices intersect or not, they only need to know how the edges are connected. (That was derived from intersections, but we can focus on generic edges and vertices from now on.)

We can rephrase the problem the following way: Given a graph, join nodes on edges until no edges remain in the graph. When joining two nodes, the joint node will have edges to nodes to which both the original nodes had edges to. If there were nodes only one of the original nodes had edges to, the joint node won't have that edge.

Terminology:

- Losing an edge: When joining nodes, the joint node won't have an edge to a node to which the original nodes had edges to.
- Retaining nodes: When joining nodes, the joint node will have an edge to a node to which the original nodes had edges to.

With this, we can define some properties.

1. When joining nodes, the joint node will retain edges to nodes to which both the joint nodes have edges to.
2. When joining nodes, edges will be lost to nodes to which only one joining node had edges to.

3. If we have N nodes which all have edges between them, so it is a complete graph, they can be reduced down to one node. This comes from 1. and 2., because we only lose edges where not all nodes have edges to that node, but because we have a complete graph we don't lose edges.
4. We can freely reduce nodes where after joining we retain all the edges.

Hypothesis: There is an optimal way to reduce an asyclic component of nodes. The optimal way is to start joining nodes "from the edges", meaning, from nodes which only have one edge. We can trace back all acyclic components to the example with 4 nodes joined in a line:

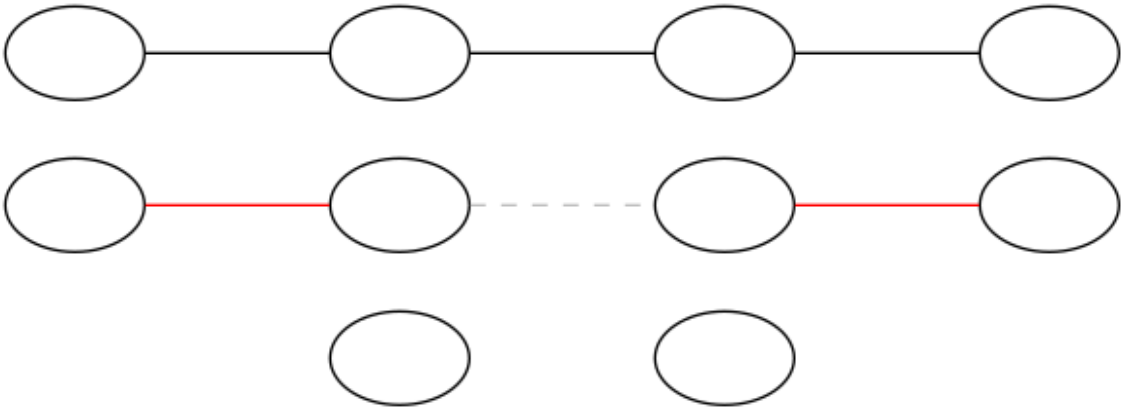


Figure 4: Optimal join

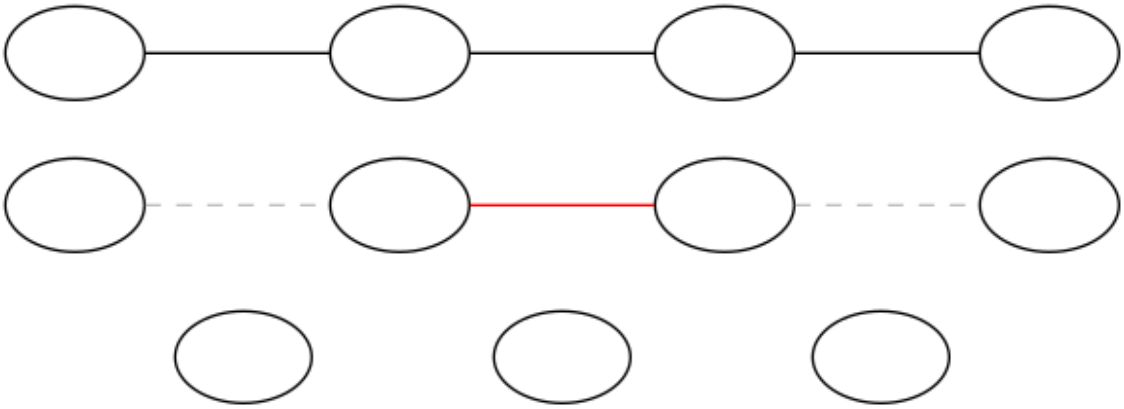


Figure 5: Not optimal join

In the first example we're first joining edges from the ends. This means, that we only lose one edge when joining. In the second example, when we are joining in the middle, we lose two edges. This is why my hypothesis is, that joining from the edges is a good strategy.

As we can see, we can reduce the same starting point to either 2 or 3 three nodes in the end. This means, that there will always be an optimal solution to this problem, and suboptimal ones.

Components with more than 4 nodes exhibit the same behavior. There are also cases, where a component can split into two 4 node chains (like in the example) and then the difference between the optimal and not optimal solutions is 2 (one for each

4 node segment). This can be achieved with a 10 node chain and joining the two nodes in the middle.

6.2 MONKE

MONKE stands for Minimal Overhead Node Kollision Elliminator. This means, that this algorithm has little to no heuristics (no overhead) and it elliminates kolliding (intersecting) nodes, A.K.A. joining them.

MONKE works in a very simple way: We take the list of edges in the graph, always take one and join the nodes on the edge. Repeat, until there are no edges in the graph.

Suprisingly, this algorithm works quite well, as we'll see in Section 6.5.

Hypothesis: If there was a seriously wrong way to join nodes, as in, lose a lot of nodes that other joins could retain, MONKE wouldn't work that well. Because MONKE is essentially a random algorithm, it wouldn't give optimal results. Therefore, my hypothesis is, that there are no completely wrong joins that can be made. Yes, as we've seen in the previous section, there are ways to have an optimal reduction, but suboptimal reductions are rare.

6.3 Least Losing Nodes

6.4 Least Losing Edges

6.5 Comparing the Graph Reduction Algorithms

7 Examples in GPT

7.1 Paid Vacation Days

I'll show an example requirement set and test generation with the Paid Vacation Days example [1, p. 100]. With this we can see how my implementation differs from the algorithm proposed by Kovacs Attila and Forgacs Istvan. I'll refer to it as "the book" from now on.

Paid vacation days

- R1: The number of paid vacation days depends on age and years of service. Every employee receives at least 22 days per year.

Additional days are provided according to the following criteria:

- R2-1: Employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days.
- R2-2: Employees of age 60 or more with at least 30 years of service receive 3 extra days, on top of possible additional days already given based on R2-1.
- R2-3: If an employee has at least 15 years of service, 2 extra days are given. These two days are also provided for employees of age 45 or more. These extra days cannot be combined with the other extra days.

Display the vacation days. The ages are integers and calculated with respect to the last day of December of the current year.

Let's look at the if statements one by one and compare the generated test cases. Note: this is without graph reduction. The test cases from the book will have an ID starting with B, my GPT implementation will have an M prefix.

The full GPT Lang definition can be found in Appendix A.1.

R1-1: IF age < 18 AND service < 30 THEN...

In the book, the test cases are:

No.	Test Case
B1	ON: age = 17, service = 29
B2	OFF1: age = 18, service < 30
B3	IN: age << 18, service << 30
B4	OUT1 age > 18, service < 30
B5	OFF2: age < 18, service = 30
B6	OUT2: age < 18, service > 30

With my GPT:

No.	Test Case	Book No.
M1	age: $(-\infty, 17]$, service: $(-\infty, 29]$	-
M2	age: $[17, 17]$, service: $[29, 29]$	B1
M3	age: $(-\infty, 16]$, service: $(-\infty, 28]$	B3
M4	age: $(-\infty, 17]$, service: $[30, 30]$	B5
M5	age: $(-\infty, 17]$, service: $[31, \infty)$	B6
M6	age: $[18, 18]$, service: $(-\infty, 29]$	B2
M7	age: $[19, \infty)$, service: $(-\infty, 29]$	B4

As we can see, my GPT covers all the test cases in the book, but has an additional test case: M1. This is, because in the book B3 is said to be an IN, but it is actually an ININ. M1 in this case is the IN. **TODO:** hmm

R1-1: IF age ≥ 60 AND service < 30 THEN...

No.	Test Case
B7	ON: age = 60, service = 29
B8	OFF1: age = 59, service < 30
B9	OFF2: age ≥ 60, service = 30
B10	IN: age > 60, service << 30
B11	OUT1 age << 60, service < 30
B12	OUT2: age ≥ 60, service > 30

With my GPT:

No.	Test Case	Book No.
M8	age: $[60, \infty)$, service: $(-\infty, 29]$	-
M9	age: $[60, 60]$, service: $[29, 29]$	B7
M10	age: $[61, \infty)$, service: $(-\infty, 28]$	B10
M11	age: $[59, 59]$, service: $(-\infty, 29]$	B8
M12	age: $(-\infty, 58]$, service: $(-\infty, 29]$	B11
M13	age: $[60, \infty)$, service: $[30, 30]$	B9
M14	age: $[60, \infty)$, service: $[31, \infty)$	B12

As we can see here as well, my GPT covered all the test cases from the book. The additional test case M8 is the IN, for the same reason as previously.

R1-1: IF service ≥ 30 AND age < 60 AND age ≥ 18 THEN...

No.	Test Case
B13	OUT1: age << 18, service ≥ 30
B14	OFF1: age = 17, service ≥ 30
B15	ON1/IN2: age = 18, service > 30
B16	ON2: age = 59, service = 30
B17	OFF2: age < 60 && age ≥ 18, service = 29
B18	OUT2: age < 60 && age ≥ 18, service << 30
B19	OFF3: age = 60, service ≥ 30
B20	OUT3: age > 60, service ≥ 30

With my GPT:

No.	Test Case	Book No.
M15	age: $[18, 59]$, service: $[30, \infty)$	-
M16	age: $[18, 18]$, service: $[30, 30]$	-
M17	age: $[59, 59]$, service: $[30, 30]$	B16
M18	age: $[19, 58]$, service: $[31, \infty)$	-
M19	age: $[18, 59]$, service: $[29, 29]$	B17
M20	age: $[18, 59]$, service: $(-\infty, 28]$	B18
M21	age: $[17, 17]$, service: $[30, \infty)$	B14
M22	age: $[60, 60]$, service: $[30, \infty)$	B19
M23	age: $(-\infty, 16]$, service: $[30, \infty)$	B13
M24	age: $[61, \infty)$, service: $[30, \infty)$	B20

Here we can see a difference between my GPT and the book. In my implementation, the two different predicates for age (age < 60 AND age ≥ 18) get merged to one Interval: $[18, 60)$.

For B15 I don't have an exact test case. This is because B15 combined the ON1 and the IN2. I have a separate test case for ON1 with M16 and a separate one for IN2 with M18.

M15 is the IN, as discussed previously.

All in all, with analyzing B15 we can say that my test cases cover the ones in the book.

R1-2: IF service ≥ 30 AND age ≥ 60 THEN...

No.	Test Case
B21	OUT1: age << 60, service ≥ 30
B22	OFF1: age = 59, service ≥ 30
B23	ON: age = 60, service = 30
B24	OFF2: age ≥ 60, service = 29
B25	IN: age > 60, service > 30
B26	OUT2: age ≥ 60, service << 30

With my GPT:

No.	Test Case	Book No.
M25	age: $[60, \infty)$, service: $[30, \infty)$	-
M26	age: $[60, 60]$, service: $[30, 30]$	B23
M27	age: $[61, \infty)$, service: $[31, \infty)$	B25
M28	age: $[60, \infty)$, service: $[29, 29]$	B24
M29	age: $[60, \infty)$, service: $(-\infty, 28]$	B26
M30	age: $[59, 59]$, service: $[30, \infty)$	B22
M31	age: $(-\infty, 58]$, service: $[30, \infty)$	B21

All the test cases from the book are covered. M25 is the IN, as mentioned previously.

R1-3: IF service ≥ 15 AND age < 45 AND age ≥ 18 AND service < 30 THEN...

No.	Test Case
B27	OUT1: age << 18, service ≥ 15 && service < 30
B28	OFF1: age = 17, service ≥ 15 && service < 30
B29	OFF2: age < 45 && age ≥ 18, service = 14
B30	ON1: age = 18, service = 15
B31	OFF3: age < 45 && age ≥ 18, service = 30
B32	OUT2: age < 45 && age ≥ 18, service > 30
B33	OUT3: age < 45 && age >= 18, service << 15
B34	ON2: age = 44, service = 29
B35	OFF4: age = 45, service ≥ 15 && service < 30
B36	OUT4: age > 45, service ≥ 15 and service < 30

With my GPT:

No.	Test Case	Book No.
M32	age: $[18, 44]$, service: $[15, 29]$	-
M33	age: $[18, 18]$, service: $[15, 15]$	B30
M34	age: $[44, 44]$, service: $[15, 15]$	-
M35	age: $[18, 18]$, service: $[29, 29]$	-
M36	age: $[44, 44]$, service: $[29, 29]$	B34
M37	age: $[19, 43]$, service: $[16, 28]$	-
M38	age: $[18, 44]$, service: $[14, 14]$	B29
M39	age: $[18, 44]$, service: $[30, 30]$	B31
M40	age: $[18, 44]$, service: $(-\infty, 13]$	B33
M41	age: $[18, 44]$, service: $[31, \infty)$	B32
M42	age: $[17, 17]$, service: $[15, 29]$	B28
M43	age: $[45, 45]$, service: $[15, 29]$	B35
M44	age: $(-\infty, 16]$, service: $[15, 29]$	B27
M45	age: $[46, \infty)$, service: $[15, 29]$	B36

All the test cases from the book are covered by my GPT.

M32 is the IN as explained previously.

M33, M35, and M37 are ON points. My GPT merges treats service as $[15, 30)$ and age as $[18, 45)$. When generating ON points age will have 18 and 44, service will have 15 and 29. My GPT takes the Cartesian product of these, that's why M34, M35, and M37 appeared, in addition to M36 which is in the book as B34.

R1-3: IF age ≥ 45 AND service < 30 AND age < 60 THEN...

No.	Test Case
B37	OUT1: age << 45, service < 30
B38	OFF2: age = 44, service < 30
B39	ON1: age = 45, service = 29
B40	OFF2: age ≥ 45 && age < 60, service = 30
B41	OUT2: age ≥ 45 && age < 60, service > 30
B42	ON2: age = 59, service << 30
B43	OFF3: age = 60, service < 30
B44	OUT: age > 60, service < 30

With my GPT:

No.	Test Case	Book No.
M46	age: $[45, 59]$, service: $(-\infty, 29]$	-
M47	age: $[45, 45]$, service: $[29, 29]$	B39
M48	age: $[59, 59]$, service: $[29, 29]$	-
M49	age: $[46, 58]$, service: $(-\infty, 28]$	-
M50	age: $[44, 44]$, service: $(-\infty, 29]$	B38
M51	age: $[60, 60]$, service: $(-\infty, 29]$	B43
M52	age: $(-\infty, 43]$, service: $(-\infty, 29]$	B37
M53	age: $[61, \infty)$, service: $(-\infty, 29]$	B44
M54	age: $[45, 59]$, service: $[30, 30]$	B40
M55	age: $[45, 59]$, service: $[31, \infty)$	B41

All the test cases in the book are covered, except for B42. B42 is the combination of an ON and an ININ. The ININ for service (and age) is M49. The ON for age is M48.

M46 is the IN, same as previously.

Summary

In total, the book has 44 test cases, my GPT generated 55 test cases.

The 11 test cases not in the book are:
- M1, M8, M15, M25, M32, M46 are INs (+6 test cases)
- B42 -> M47 + M48 (+1 test case)
- B15 -> M16 + M18 (+1 test case)
- M34, M35, M37 are additional ONs (+3 test cases)

After graph reduction

With my GPT and MONKE:

No.	Test Case
1	age: $[17, 17]$, service: $[29, 29]$
2	age: $[59, 59]$, service: $[30, 30]$
3	age: $(-\infty, 16]$, service: $[15, 28]$
4	age: $[18, 18]$, service: $(-\infty, 13]$
5	age: $[45, 59]$, service: $[31, \infty)$
6	age: $(-\infty, 16]$, service: $[30, 30]$
7	age: $[17, 17]$, service: $[31, \infty)$
8	age: $[61, \infty)$, service: $[15, 28]$
9	age: $[60, 60]$, service: $[29, 29]$
10	age: $[18, 18]$, service: $[15, 15]$
11	age: $[60, 60]$, service: $[30, 30]$
12	age: $[60, 60]$, service: $[31, \infty)$
13	age: $[18, 18]$, service: $[29, 29]$
14	age: $[46, 58]$, service: $(-\infty, 28]$
15	age: $[19, 44]$, service: $[31, \infty)$
16	age: $[18, 18]$, service: $[30, 30]$
17	age: $[44, 44]$, service: $[15, 15]$
18	age: $[18, 44]$, service: $[14, 14]$
19	age: $[44, 44]$, service: $[29, 29]$
20	age: $[45, 45]$, service: $[29, 29]$
21	age: $[19, 43]$, service: $[16, 28]$
22	age: $[61, \infty)$, service: $[31, \infty)$
23	age: $[59, 59]$, service: $[29, 29]$

In the book, the number of reduced test cases is 18. This is 40.9% of the original 44 test set.

My GPT with MONKE reduced the number of test cases to 23. This is 41.8% of the original test set.

Conclusion

In conclusion, my GPT implementation generated all the test cases that were mentioned in the book. It also generated a few more test cases **TODO:** reason why this is good . The graph reduction reduced the number of test cases by a similar percentage than the reduction in the book.

Let's make use of || in GPT Lang

"R2-1 Employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days."

The wording or R2-1 uses 'or's. In the book this was written only with conjunctions. In GPT Lang it looked like this:

```
if(age < 18 && service < 30)
if(age >= 60 && service < 30)
if(service >= 30 && age < 60 && age >= 18)
```

But we can write it in GPT Lang with ||s:

```
if(age < 18 || age >= 60 || service >= 30)
```

As we can see, it is significantly easier to write, and we don't have to think about how to manually create conjunctions and cover all the cases, as GPT does that for us.

TODO: Explore the non variable order dependent version

8 Code architecture

8.1 Rust

8.2 Frontend app, webassembly

9 Future improvement ideas

9.1 Coincidental Correctness

Hierons, R. M. has shown in [6], 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.

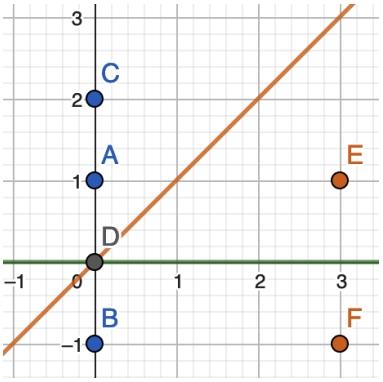
From this, we'd write it in GPT Lang as:

```
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 $f(x) = x$.



9.2 Linear Predicates

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

9.3 Different equivalence partitions for implementations

When implementing a program, if we differ by some, we might unkowingly create an implementation that has different equivalence partitions.

Example:

Paid Vacation Days reference implementation:

```
function paidVacationDays(age, service) {
   let days = 22

   if (age < 18 || age >= 60 || service >= 30) {
      days += 5
   }
   if(age >= 45 && age < 60 && service < 30) {
      days += 2
   }
   if(age >= 18 && age < 45 && service >= 15 && service < 30) {
      days += 2
   }
   if(service >= 30 && age >= 60) {
      days += 3
   }

   return days
}
```

Paid vacation days different implementation:

```
function paidVacationDays(age, service) {
   let days = 22

   if (age < 18 || age >= 60 || service >= 30) {
      days += 5

      if (age >= 60 && service >= 30) {
         days += 3
      }
   } else if (service >= 15 || age >= 45 /* <- This */) {
      days += 2
   }

   return days
}
```

In the second implementation, if we replace that `age >= 45` with any number in $[46, 59]$ the original tests in **TODO:** link chapter of paid vacations still all pass. But we know for sure that we've made an error, because we've replaced that number.

An example test case for this error would be `{age: [46, 59], service: (-Inf, 14]}`.

This is, because this implementation has different equivalence partitions. In the original tests we only had to test M49 to test that service goes up to 28, not 14.

Because of graph reduction and how we select test points, it's possible that we select a number from $[15, 28]$. In this implementation with the `||` this test case will 'short circuit', because it sees a test case with `service >= 15` and it won't test the `age` part.

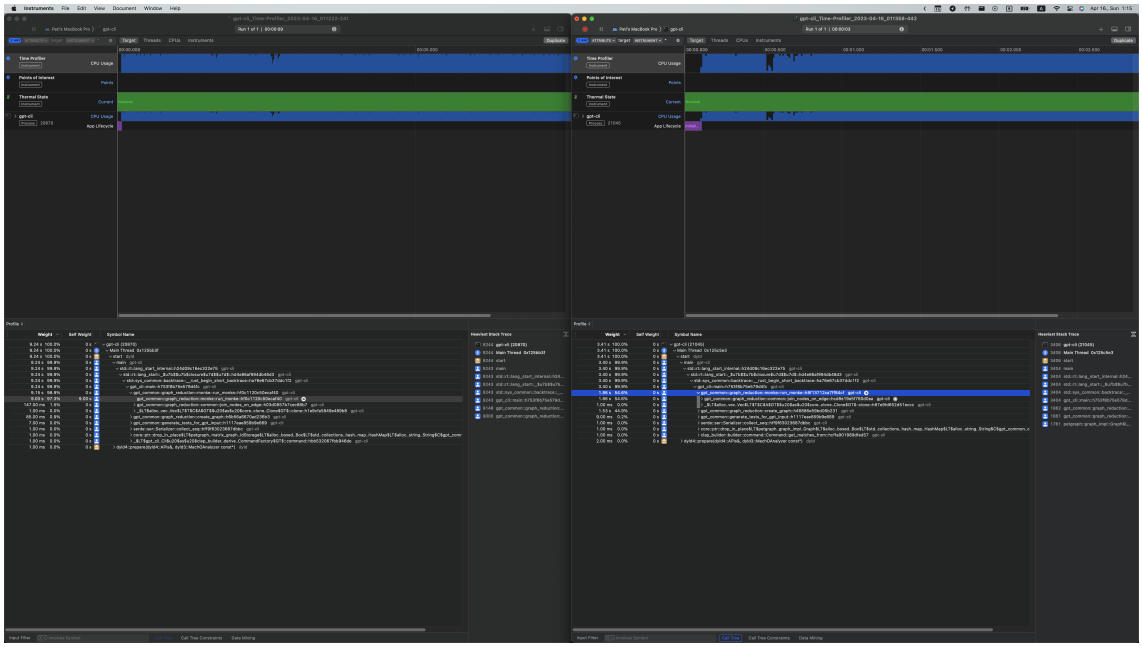
Because we've differed from the equivalence partitions that we've defined our GPT Lang definition with, the way we can solve this is to create another GPT Lang definition for this implementation. This way, we'll have a test case that covers the `service >= 15 || age >= 45` condition, which didn't exist in the original one.

This is related to the Competent Programmer hypothesis **TODO:** link , because we don't want to test implementations that are vastly different or more overcomplicated than a simple solution. In this case, the else if, the nested if and additional `||` made this implementation more complex.

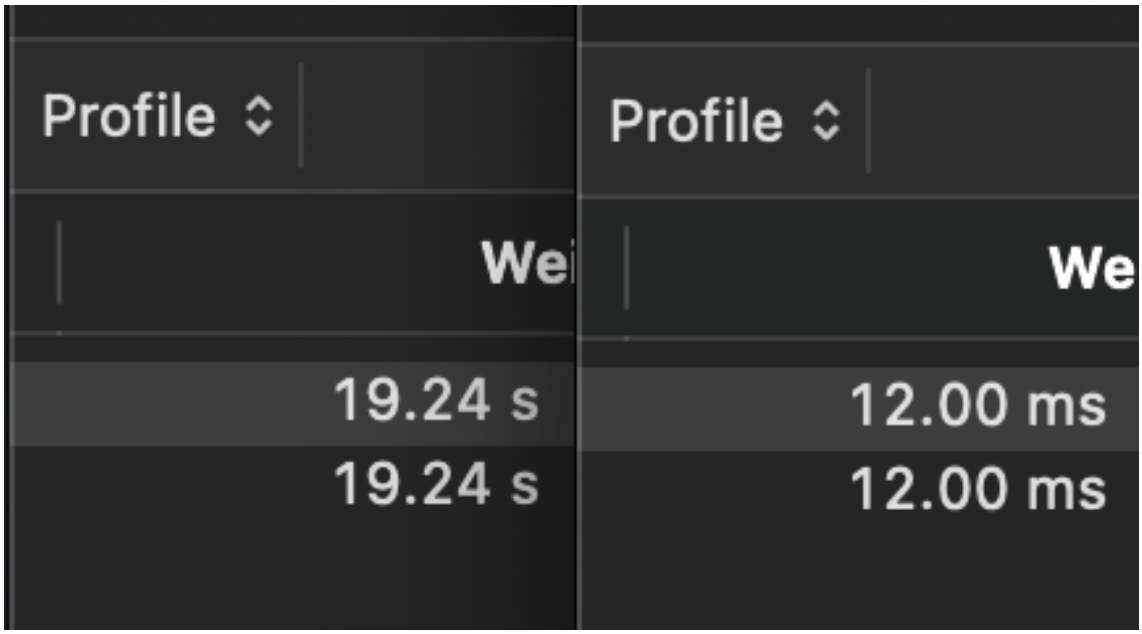
10 TODOs

- 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]
- Recalculate the ONs on IN-ININs in the paid vacations example. Show some examples that the book can't cover, but my GPT covers it.

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.



Smaller sample:



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

Least Losing Edges Reachable: 50 Runtime: 19.24s

MONKE: 49 Runtime: 12ms

- Murane argues that “One disadvantage with boundary value analysis is that it is not as systematic as other prescriptive testing techniques” [2]. GPT is systematic and test cases can be generated automatically
- Contrary to what Analyzing boundaries can’t only be done on number types, but it can be extrapolated for other types as well. As Rabino has shown, it is possible to analyze boundary conditions of physical properties like wind speed or Reynolds Number for Turbulence models [7].

A. Appendix

A.1 Planned Vacation Days

TODO: This might need to be updated

```
/*
Paid vacation days

R1 The number of paid vacation days depends on age and years of
service. Every employee receives at least 22 days per year.

Additional days are provided according to the following criteria:
R2-1 Employees younger than 18 or at least 60 years, or employees with
at least 30 years of service will receive 5 extra days.
R2-2 Employees of age 60 or more with at least 30 years of service
receive 3 extra days, on top of possible additional days already given
based on R2-1.
R2-3 If an employee has at least 15 years of service, 2 extra days
are given. These two days are also provided for employees of age 45
or more. These extra days cannot be combined with the other extra
days.

Display the vacation days. The ages are integers and calculated with
respect to the last day of December of the current year.
*/

var age: int
var service: int

// R1
if(age < 18 && service < 30)
if(age >= 60 && service < 30)
if(service >= 30 && age < 60 && age >= 18)

// R1-2
if(service >= 30 && age >= 60)

// R1-3
if(service >= 15 && age < 45 && age >= 18 && service < 30)
if(age >= 45 && service < 30 && age < 60)
```

Bibliography

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

[2] T. Murnane, and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in *Proc. 2001 Australian Softw. Eng. Conf.*, 2001, pp. 12–20.

[3] S. Nidhra, and J. Dondeti, "Black box and white box testing techniques-a literature review," *Int. J. Embedded Syst. Appl. (Ijesa)*, vol. 2, no. 2, pp. 29–50, 2012.

[4] M. E. Khan, and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, 2012.

[5] dkaeae (https://cs.stackexchange.com/users/70382/dkaeae), "Given a set of intervals on the real line, find a minimum set of points that 'cover' all the intervals." [Online]. Available: https://cs.stackexchange.com/q/101911 (Computer Science Stack Exchange)

[6] 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

[7] N. Rabino, "Analysis and qualitative effects of large breasts on aerodynamic performance and wake of a "miss kobayashi's dragon maid" character," 2018.