UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA AUTOMATIZADA PARA GENERAR CÓDIGO USANDO PROGRAMACIÓN GENÉTICA EN UN LENGUAJE DE PROGRAMACIÓN ORIENTADO A OBJETOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

VICENTE FERNANDO ILLANES VENEGAS

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
CESÁR GUERRERO SALDIVIA
JÉRÉMY BARBAY

SANTIAGO DE CHILE
2021

Resumen

Hoy en día, las técnicas de aprendizaje de máquinas para generar código fuente son un área de investigación importante y prometedor. Se ha demostrado que la programación genética, o *genetic programming* (GP), contribuye eficientemente en la reparación de software y optimización de programas. Sin embargo, la mayoría de los avances aplicando GP para generar código fuente se limita a programas escritos en el lenguaje de programación C. Esto restringe a que muchas de las decisiones de diseño que se toman en cuenta en las soluciones implementadas son consecuencia de que este lenguaje es procedural y con tipos estáticos.
Esta memoria explora el uso de la programación genética para generar código fuente en un lenguaje orientado a objetos con tipado dinámico. Esto implica que la técnica desarrollada debe tomar en cuenta el contexto clásico de estos lenguajes, que incluye: clases, métodos y objetos. Para este fin, se complementa a GP de un sistema de pesos que toma en cuenta la jerarquía y herencia de clases, la modularidad, variables de instancia y la localización de métodos. En cuanto a la calidad del código generado por GP, se utiliza un conjunto de pruebas unitarias (*tests*) que encapsulan los requerimientos funcionales que el código generado debe cumplir.
Los experimentos se diseñan sobre el lenguaje de programación Pharo. En particular, se ejecuta el algoritmo en la librería Roassal, una herramienta de visualización escrita en este lenguaje, con el objetivo de generar el cuerpo de los métodos de una sola línea que pertenecen a Roassal. Los resultados obtenidos reflejan que GP dotado de un sistema de pesos permite producir código de una sola línea con una precisión del 51 %, a diferencia del 44 % utilizando un algoritmo GP puro. Estos resultados contribuyen a demostrar la factibilidad de que GP efectivamente puede construir código fuente en aplicaciones con programación orientada a objetos y dinámicamente tipado, y que, en un futuro, contribuya a construir parches automatizados que logren resolver errores de software en programas con las características mencionadas.

Esta dedicatoria se divide en 9 partes:
Para Marcia,
Para Fernando,
Para Ignacio,
Para Joaquín,
Para Cecilia,
Para Graciela,
Para Carlos,
Para Mireya,
y para todos aquellos que luchan por sus sueños.

Agradecimientos

Agradecer al profesor Alexandre Bergel por su dedicación y entrega durante todo el desarrollo del proyecto. Desde la invitación a formar parte de este, y cómo durante todo el proceso estuvo atento ante cualquier dificultad que apareciese, otorgando consejos, retroalimentación y herramientas para enfrentar los problemas. También agradecer al equipo de ISCLAB por apoyarme sobre lo que concierne a Pharo y sus herramientas.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La generación de código fuente utilizando inteligencia artificial es un campo de investigación desafiante y en constante desarrollo. En particular, el aprendizaje de máquinas tiene el potencial de reducir significativamente la carga del programador en corregir manualmente errores de software, siendo este uno de los problemas más relevantes para los desarrolladores [1, 2]. Esta problemática nace del hecho de que un programa puede tener múltiples errores (o *bugs*). Además, a pesar de que hoy en día existen técnicas que de forma iterativa y sistemática aportan en la corrección del código [3], este sigue siento un proceso tedioso y monótono que implica un gasto importante de tiempo y recursos. Esto último se refleja en que la mantención y depuración son tareas cruciales en el continuo proceso de extensión y mejora en algún proyecto de software. En efecto, estas disyuntivas corresponden a un gran porcentaje del trabajo necesario en su desarrollo. De hecho, existen fuentes que señalan que este valor corresponde al 90 % del costo total de un proyecto de software común. [4]

La mayoría de los avances en esta área se han limitado a la generación de código fuente en el lenguaje C [1, 2]. Por lo que las soluciones implementadas se basan en el hecho de que este lenguaje es procedural y estáticamente tipado. La relevancia entonces de esta memoria, es investigar que tan factible es generar código en lenguajes con otras características. En particular, explorar la programación orientada a objetos, o *object oriented programming* (OOP) y tipos definidos de manera dinámica. Por lo tanto, este trabajo de título considera dos preguntas fundamentales para la definición de la solución a construir:

1. ¿Cuales son los requerimientos funcionales que debe cumplir el programa para considerarse correcto?

2. ¿Cuáles son los componentes que deben formar parte del código a generar?

Entonces, a partir de estas incógnitas se lleva a cabo esta investigación, donde se realiza un estudio sobre la generación de código utilizando programación genética. Específicamente se busca emplear la técnica desarrollada para encontrar el cuerpo de métodos en contextos arbitrarios. En particular, se consideran aquellos métodos que se definen exactamente con solo una línea de código.

1.1. Objetivos

En la programación, los requerimientos funcionales definen explícitamente lo que un extracto de código debe hacer. A su vez, sirve para evaluar que tan aceptable es el código para cumplir su propósito. Lo anterior también aplica en el contexto de la construcción automatizada de código en esta investigación. En particular, la definición concreta de estos requisitos esta sujeto a un conjunto de prueba unitarias, o *tests*, que pertenecen a la librería del método buscado. Este conjunto define explícitamente el funcionamiento deseado de la aplicación en su totalidad.

Entonces, como objetivo principal se propone desarrollar una técnica automatizada que utiliza programación genética para generar código fuente en un contexto particular. Concretamente, se busca cubrir la necesidad de construir código en un lenguaje de programación orientado a objetos y tipos dinámicos. El trabajo realizado se lleva a cabo en un lenguaje que cumple con estas características, llamado Pharo Smalltalk [5][1]. Además, otro objetivo es aportar antecedentes para demostrar que en un futuro se logre construir parches correctivos de software en un lenguaje de programación con las particularidades mencionadas.

1.2. Pregunta de investigación y desafíos

Como antecedentes a esta memoria, existen diversos trabajados realizados sobre el lenguaje de programación C [1, 2], donde se logran generar código fuente que sirven de parches correctivos. En consecuencia, es interesante tomar en cuenta la siguiente pregunta de investigación:

¿Qué tan adecuada es la programación genética para generar código fuente para un lenguaje de programación orientado a objetos, basado en clases y de tipo dinámico?

Para poder responder esta pregunta, la idea es realizar el trabajo hecho en C de manera similar en Pharo. En una primera impresión, esto parece fácil de implementar. Sin embargo, existen diferencias cruciales entre los dos lenguajes que hacen desafiante construir la heurística en este lenguaje. En primer lugar, Pharo es completamente orientado a objetos, cualquier estructura dentro de este es un objeto. A diferencia del lenguaje C, que es procedural. Otra diferencia importante, son los tipos dinámicos de un lenguaje OOP, perdiendo la posibilidad de descartar programas de manera estática como en C. Si además se considera la homogeneidad y simplicidad de Pharo para construir programas[2], es razonable asumir que la construcción de código se resuelve enviando un mensaje a un objeto en particular. Por lo tanto, el programa genético necesita generar código simple donde los objetos involucrados envían mensajes que entienden a otros objetos. El algoritmo recibe como conjunto de entrada el nombre formal del método (y por consiguiente la clase y el paquete a los cuales pertenece) y un conjunto de *tests* pertenecientes a la misma aplicación. El objetivo es encontrar el código que corresponda al cuerpo de un método buscado arbitrario, aprobando exitosamente todos los casos de prueba.

[1] `http://pharo.org`

[2] El cómputo completo de un programa se expresa utilizando una única construcción sintáctica: el envío de mensajes entre objetos

El algoritmo utiliza los *tests* para calcular que tan cercana es una solución candidata al código que realmente se necesita generar según los requisitos funcionales. Por lo tanto, otra importante disyuntiva de investigación es confirmar que la calidad[3] del conjunto de casos de prueba influye directamente en el rendimiento de la técnica. Esto debido a que la técnica se centrará solamente en aprobar los casos de prueba existentes, sin considerar funcionalidades que no son evaluadas por los *tests*. Concretamente, la finalidad del algoritmo es minimizar la diferencia entre la cantidad total de *tests* y el total de casos aprobados por el código candidato, y se detendrá cuando algún individuo cumpla que la diferencia entre ambos conjuntos sea cero. Además, el conocer de antemano el entorno en que se utiliza el método (por ejemplo, la clase a la cual pertenece), permite definir un contexto que guíe la búsqueda del código de mejor manera. Este contexto se traduce explícitamente en un sistema de pesos que asocia probabilidades o valores que se utilizan al generar un árbol de sintaxis abstracta, o en las operaciones genéticas. En particular, los pesos reflejan una heurística que considera el estilo de programación, la jerarquía y herencia de clases, las referencias externas de clases, las variables de instancia, la locación de los métodos y los *tests* utilizados. Este conjunto de pesos se calibra a partir de una gran base de código. Con estos detalles en mente, se busca demostrar la creación automática de métodos de solo una línea pertenecientes a la librería Roassal [6], una herramienta de visualización ingenieril construida en Pharo.

1.3. Resultados principales

Con respecto a la validación de la la solución implementada, se realizaron dos experimentos principales: (1) aplicar la técnica sobre una cantidad importante de métodos, y (2) utilizar distintos sistemas de pesos para encontrar el código referente a un método complejo. En (1) los principales resultados obtenidos son: que un sistema uniforme de pesos permite obtener una precisión del 44 %, mientras que si a la técnica se le dota de un sistema más sofisticado se obtiene un 51 % de precisión. Por otro lado, en (2) se concluye que una combinación ponderada de los sistemas contribuye a una búsqueda más efectiva, tanto en tiempo de búsqueda como en la solución encontrada.

El trabajo de título presentado en este informe, ha sido aceptado como artículo de investigación titulado *Generating Object-Oriented Source Code Using Genetic Programming* en el *Workshop Genetic Improvement*[4], el cual es parte de la conferencia internacional ICSE 2021[5]. Esto demuestra la relevancia de la investigación expuesta en esta memoria, ya que es de interés para la comunidad científica especializada en este campo.

1.4. Estructura de la memoria

Esta memoria se divide de la siguiente manera: el capítulo 2 explica el estado del arte detrás de esta investigación. El capítulo 3 describe detalladamente el problema y su relevancia. A continuación, el capítulo 4 explica la solución construida, la metodología utilizada y cómo se utiliza la programación genética para representar código escrito en Pharo. Después, en

[3]Cantidad, si son suficientes para cubrir correctamente la funcionalidad esperada.

[4]`http://geneticimprovementofsoftware.com/events/icse2021.html`

[5]*43th International Conference on Software Engineering*, `https://conf.researchr.org/home/icse-2021`

el capitulo 5 se especifica la validación que demuestra cómo la solución propuesta resuelve en parte el problema planteado. Finalmente, en el capítulo 6 se concluye y se realiza una reflexión de la relevancia del trabajo realizado en la memoria.

Capítulo 2

Estado del Arte

En esta sección se exponen los principales conceptos involucrados en la investigación. En primer lugar, la sección 2.1 explica en que consiste la *programación genética*, la cual es una metodología existente, que se utiliza para resolver problemas de búsqueda, y cómo esta se utiliza de base para la solución desarrollada durante la memoria. También, se exhibe una investigación realizada en el lenguaje C para dar a entender por qué esta investigación puede aportar en el campo de la reparación de software (sección 2.2).

2.1. Programación Genética

En el área de la inteligencia artificial, la programación genética, o *genetic programming* (GP), es una técnica computacional inspirada en la teoría de la evolución Darwiniana, que permite encontrar un programa computacional que resuelve una determinada tarea, de forma automatizada e iterativa [7, 8]. A través de iteraciones o generaciones, la ejecución de GP mantiene un conjunto de programas candidatos a solucionar el problema particular. Este conjunto se conoce como la *población de individuos*, donde cada uno de sus integrantes es un programa computacional representado por su árbol de sintaxis abstracta, o *abstract syntax tree* (AST). Como se explica en la figura 2.1, durante generación en generación, GP utiliza un algoritmo que transforma la población actual en otro grupo de individuos, con la idea de obtener mejores representantes para aproximarse a la solución deseada.

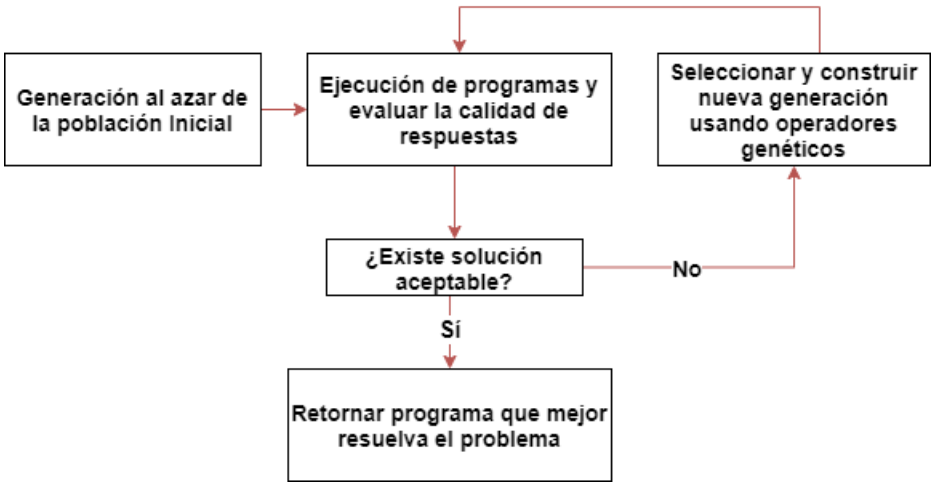

Figura 2.1: Esquema principal de GP.

2.1.1. Representación de Programas

Como se mencionó anteriormente, los individuos de GP son programas representados a través de su AST. En la figura 2.2, se puede observar el árbol correspondiente para el programa `if` $x > y$ `then` $x + 1$ `else` y. Un detalle importante es que las constantes y variables son las hojas del árbol. En GP estos elementos se conocen como los valores *terminales*.
Por otro lado, los *operadores o funciones* son los nodos internos. La combinación de estos dos conjuntos dan origen al *universo de búsqueda* de GP. La población inicial del algoritmo, son programas construidos al azar considerando estos dos conjuntos. Por una parte, se define un número constante de individuos que definen el tamaño de la población y, por otro lado, la altura máxima que podría llegar a tener el AST de un individuo, dando la libertad que estos tengan menor altura utilizando probabilidades para detener el crecimiento del árbol.

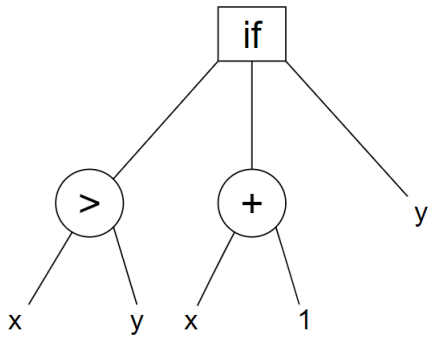

Figura 2.2: Representación AST del programa (`if` $x > y$ `then` $x + 1$ `else` y).

2.1.2. Calidad de un individuo

Para saber que tan efectivos son los programas generados, se necesita algún mecanismo que evalúe cada individuo verificando cuantitativamente si algunos de ellos resuelve el problema o se aproxima a la solución. Una vez que se encuentra aquel individuo que cumple con los requerimiento necesarios, significa que se ha encontrado el individuo adecuado a retornar por

el algoritmo de GP. Lo anterior es tarea esencial de la función de ajuste o aptitud, o de *fitness*, que recibe como argumento un AST y retorna un número interpretable que, según el contexto del problema, se busca maximizar o minimizar. Esta función encapsula los requerimientos que debe cumplir un individuo para los objetivos del proceso de búsqueda en cuestión.

2.1.3. Heurísticas de selección

Es natural querer obligar al algoritmo a buscar individuos con mejores resultados de *fitness*, por lo que la técnica al construir una nueva generación, necesita una estrategia de clasificación de la población; de manera de favorecer la elección de individuos con buena evaluación de *fitness*, otorgando una menor oportunidad de ser escogidos a individuos mal evaluados. Notar que no se prohíbe que el algoritmo seleccione un individuo con resultados deficientes, esto se permite para evitar un posible sesgo de parte del algoritmo y otorgar mayor variabilidad entre los integrantes de la población. Es importante mencionar que el proceso de selección debe mantener constante el tamaño de la población. A continuación, se presentan dos formas comunes de seleccionar a los individuos que dejarán *descendientes* para la próxima generación de individuos:

- **Torneo**: Se seleccionan N (comúnmente se eligen 5) individuos al azar. Luego se escoge al mejor de ellos según valores de *fitness*.
- **Ruleta**: Se considera la suma total de *fitness* de la población F como una *ruleta* o área circular. Luego, se asigna a cada individuo una parte proporcional según su propio valor de *fitness* y F. Es decir, los mejores programas recibirán una mayor fracción asignada que los peores. Finalmente, se escoge al azar un valor entre $[0, 1]$ y se retorna aquel individuo en ese valor de la ruleta.

2.1.4. Elitismo

Antes de usar una heurística de selección, es posible pre seleccionar de forma discriminada a los mejores individuos de la población (1-5 individuos), permitiendo que estos continúen evolucionando en las próximas generaciones. Esta estrategia permite que la calidad de la solución ofrecida por GP no empeore de una generación a otra. [9]

2.1.5. Operadores evolutivos

Las transformaciones para mejorar los individuos son realizadas por operadores genéticos especializados para trabajar sobre árboles:

- **Cruzamiento (o *crossover*):** a partir de dos individuos P y Q seleccionados, se eligen nodos al azar de cada uno y se combinan para generar uno o dos nuevos programas. Un ejemplo donde solo se genera un nuevo individuo se puede observar en la figura 2.3[1]. Primero, se selecciona un nodo al azar de cada árbol padre: p y q, que servirán como puntos de corte. Luego, se realiza una copia del árbol P y reemplazamos el subárbol con raíz p por el subárbol con raíz q. En el caso de querer generar un segundo individuo, se considera el proceso análogo usando una copia de Q.

[1]Fuente de imagen: [8], página 12.

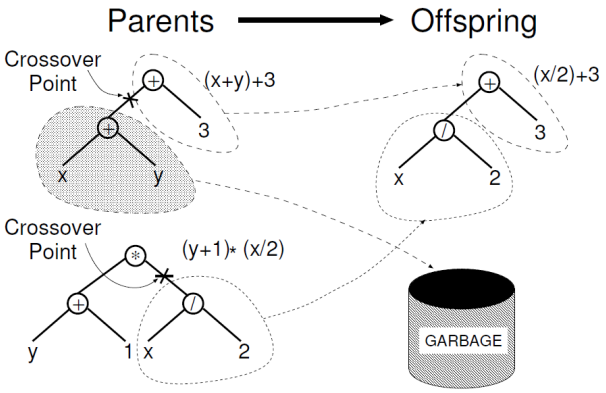

Figura 2.3: Ejemplo *crossover* entre dos árboles.

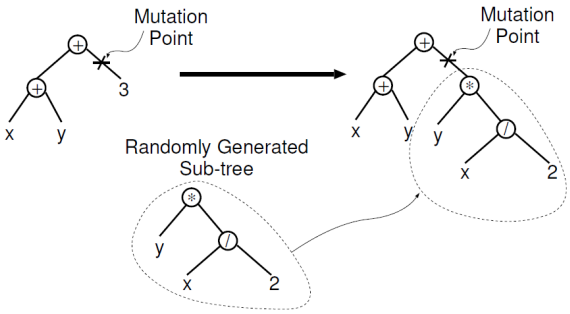

Figura 2.4: Ejemplo mutación de un individuo.

- **Mutación:** se genera un nuevo individuo a partir de un programa al cual se le altera una parte de manera al azar, tal como se observa en la figura 2.4[2]. Esta operación, se puede definir en términos del cruzamiento entre un individuo de la población, y un programa completamente nuevo generado por GP.

2.1.6. Valores de entrada de GP

Para llevar acabo la ejecución del algoritmo, es importante tener en cuenta valores que permitan controlar la evolución de los programas, estos son los hiper-parámetros. A continuación se presentan los más relevantes:

- Tamaño de la población o número de individuos.
- Número máximo de generaciones, esto es para evitar que el algoritmo funcione indefinidamente, ya que existe la posibilidad de que no se encuentre la solución. Por otro lado, existen problemas en que no es tan claro para el programador cuál es el objetivo final del algoritmo. En este caso se retorna el mejor individuo hasta el momento.
- Tasa de mutación, porcentaje que indica que tan seguido se aplicará este operador genético. Mientras menor sea el valor, mayor será la probabilidad de no aplicar una mutación.
- En caso de recurrir al elitismo, definir la cantidad de individuos a preseleccionar.

[2]Fuente de imagen: [8], página 13.

2.2. Reparación de Software

A pesar de que la programación genética se formuló a principios de la década de los cincuenta, sus mayores aportes al mundo han sido recientes, proporcionando novedosas soluciones en varias áreas, como videojuegos, problemas numéricos de ordenamiento y búsqueda, entre otros; pero hasta ahora ha sido un gran desafío desarrollar soluciones que permitan un reparo automático de software. Los desarrolladores humanos aún deben desarrollar, mantener y reparar código fuente en gran medida manualmente.

La mayoría de los avances desarrollados en esta área se limitan al lenguaje de programación C. Un ejemplo de lo anterior, es el trabajo realizado por Forrest et al [1], donde se generan resultados exitosos usando GP. Se logran encontrar parches correctivos para una cantidad específica de diez programas sumando un total de 63 mil líneas de código. Por ejemplo, uno de estos programas es encontrar de manera correcta el máximo común divisor entre dos números enteros a y b, el cual se puede observar en la figura 2.5 [3].

```
1  /* requires: a >= 0, b >= 0 */
2  void gcd(int a, int b){
3      if(a == 0) {
4          printf("%d", b);
5      }
6      while (b != 0)
7          if (a > b)
8              a = a - b;
9          else
10             b = b - a;
11
12     printf("%d", a);
13     return;
14 }
```

Figura 2.5: Programa en C defectuoso

Este programa tiene el grave error de que si $a = 0$ y $b > 0$, itera indefinidamente en las líneas 6, 9 y 10, a pesar de que en la línea 4 entrega la respuesta correcta. Para resolver esto, la técnica desarrollada recibe como entrada un programa defectuoso P y dos conjuntos de *tests* que sirven para medir cuantitativamente la funcionalidad deseada. El primer grupo tiene N pruebas unitarias positivas, que implícitamente definen los requerimientos funcionales que cubre P, incluso antes de ser corregido. Un caso de prueba positivo podría ser `gcd(12, 4) = 4`, ya que a pesar del error que contiene el programa anterior, este calcula correctamente el máximo común divisor entre dos enteros positivos. El segundo conjunto corresponde a M pruebas unitarias negativas, que evidencian el defecto o el error a resolver. En este caso, serviría `gcd(0, 9) = 9`.

Una vez definido el conjunto de entrada, se inicia el proceso evolutivo de GP tal como se describió en la sección 2.1. A medida que el algoritmo avanza, se generan variantes del programa original, realizando modificaciones al azar para tratar de arreglar el error. Esto implica que muchas veces no se logra los efectos deseados, generando programas que no compilan o que no pasan exitosamente ningún caso de prueba. Estas variantes del programa original son los individuos de la población que mantiene GP, representados por su AST y un

[3] Ejemplo proveniente de investigación publicada por Forrest y cia [1]

weighted path, que es una lista de pares (s, w). Cada par representa una sección del código s de la variante y un peso w. Esta lista cumple la función de guiar la búsqueda de la variante que se necesita. De esta forma se puede tener una idea de donde insertar el código. En el caso del programa de la figura 2.5, esta lista tendría tamaño 11.

Este proceso se repite hasta cumplir una cierta condición de término que resuelve el problema inicial, para este problema se necesita una variante del programa que apruebe exitosamente todos los casos de prueba. En la figura 2.6[4]. , se puede observar una propuesta de variante al problema del programa original, donde se resuelve la iteración infinita agregando un código de retorno en la línea 4. A pesar de que podemos retornar esta respuesta, se puede observar que en la línea 5 hay código muerto, es decir, que nunca se ejecuta. Por lo tanto, en un paso final, GP necesita aplicar un algoritmo para minimizar el parche, obteniendo el conjunto mínimo de líneas que deben ser agregadas para permitir que el programa inicial apruebe todos los casos de prueba.

```
1  /* requires: a >= 0, b >= 0 */
2  void gcd_2(int a, int b){
3      if(a == 0) {
4          printf("%d", b);
5          exit(0);            //inserted
6          a = a - b;          //inserted
7      }
8      while (b != 0)
9          if (a > b)
10             a = a - b;
11         else
12             b = b - a;
13
14     printf("%d", a);
15     return;
16 }
```

Figura 2.6: Propuesta de corrección al programa en C.

[4]Ejemplo proveniente de investigación publicada por Forrest y cia [1]

Capítulo 3

Problema

El objetivo principal a lo largo de esta memoria es generar código fuente en el lenguaje Pharo para métodos definidos en solo una línea o sentencia. La propiedad fundamental que debe cumplir el resultado entregado por la técnica desarrollada es, cumplir con los requerimientos funcionales que se necesitan, los cuales son definidos a través de un conjunto de *tests*.
Considerando la manera en que Pharo permite a los usuarios generar programas útiles y funcionales, la solución generada por GP debe determinar cuales objetos y métodos deben ser parte de la solución. Sin embargo, debido a las propiedades de polimorfismo y delegación en un lenguaje OOP, no es sencillo elegir el objeto ni el método indicado para resolver el problema. Por ejemplo, en el caso de que la solución a considerar es una secuencia de llamadas de métodos, los argumentos y objetos respectivos deben cumplir con tipos no explícitos para evitar generar errores no deseados. Para ejemplificar este problema, considere el método presentado en la figura 3.1. Este método, cuyo nombre formal es "`addShape:`", pertenece a la clase `RSAbstractChartElement` y recibe como argumento formal `aShape`[1].

```
1 RSAbstractChartElement >> addShape: aShape
2     "add a shape to the canvas"
3     chart canvas add: aShape.
```

Figura 3.1: Método "`addShape:`" escrito en Pharo

El ejemplo presenta en la línea 3 el cuerpo del método que agrega una nueva figura `aShape` a una colección de elementos. Este método utiliza la variable de instancia `chart`, para que envíe el mensaje `chart`. La expresión `chart canvas` retorna una colección de elementos. Finalmente, esta colección es capaz de enviar el mensaje `add:` que agrega el argumento a la colección, cumpliendo con el propósito de la definición del método. Cabe destacar, que no es sencillo establecer el tipo de `chart` ya que el método `canvas` es implementado por una

[1]Con respecto al envío de mensajes, en Pharo existen tres formas de hacer esto, de forma unaria, el cuál solamente necesita un objeto que envía un método (`anObject aMessage`), binaria (`anObject aMessage: otherObject`) y mensajes con una cantidad arbitraria de argumentos (`anObject akey: anotherObject akey2: anotherObject2`). En cualquier caso el método en cuestión debe ser implementado por la clase a la que pertenece `anObject`

decena de clases en el ambiente de Pharo. Los métodos a utilizar pueden provenir de otras clases distinta a la del método buscado. Esto puede aumentar considerablemente el tamaño del universo de búsqueda del cual el GP genera los AST candidatos. Entonces, es necesario establecer un mecanismo de prioridad entre las clases disponibles para guiar y controlar la búsqueda del código que se busca generar. Además, el GP debe extraer la mayor cantidad de información posible sobre los tipos de las variables de instancia, ya que necesita saber que objetos son capaces de mandar un mensaje. En el ejemplo, la variable `chart` entiende y envía el mensaje `canvas`.

La importancia de contar con una solución aporta en dar un primer acercamiento para demostrar la factibilidad de explorar la reparación de software en lenguajes de programación orientado a objetos con tipos dinámicos. A pesar que la técnica se implementa en Pharo, se espera que la solución debiese generalizar, de modo que sea sencillo de aplicar en otros lenguajes de similares características. Aunque, existe la dependencia de sintaxis y semántica de un programa que puede variar entre lenguajes.

3.1. Limitaciones de la solución

Al momento de definir los requisitos indispensables que debe cumplir la técnica, se busca también mostrar que esta funciona bajo ciertos escenarios específicos. Principalmente, podemos separar la solución ofrecida en tres grandes fases:

1. Calibrar parámetros del algoritmo según contexto ofrecido por el conjunto de entrada.
2. Construir una posible solución controlado por un número máximo de tiempo (iteraciones).
3. Analizar la efectividad de la respuesta entregada por el algoritmo.

La programación genética (GP) permite llevar a cabo todo este proceso a cabalidad. En particular, se cuenta con que GP es flexible, pues es posible agregar condicionantes a la ejecución del algoritmo que se adapten al contexto de crear código fuente en un lenguaje con programación orientada a objetos con tipos dinámicos. Concretamente, a través de los individuos se representan programas que tienen sentido para el lenguaje Pharo para que estos puedan ser correctamente compilados y ejecutados.

El análisis de la solución construida se lleva acabo fielmente utilizando la función de *fitness* que permite discriminar entre un individuo y otro. La comparación se realiza cuantitativamente a través del conjunto de *tests* disponibles, ejecutando y contando la cantidad de pruebas exitosas del código construido una vez compilado. Sin embargo, este conjunto a su vez limita que la mejor solución sea aquella que simplemente cumple con todos los *tests*, y que no necesariamente cumpla con buenas prácticas de programación, siendo "sintácticamente más limpia", o que pase a llevar alguna propiedad fundamental en OOP, como lo son la delegación y la modularidad. Por lo tanto, la solución ofrecida se restringe a encontrar una sentencia que sea semánticamente equivalente al código real buscado. De esta forma, se presenta una limitación de la solución sujeta a la calidad del conjunto de *tests*. Un caso sencillo es considerar dos objetos `x` e `y` de distinta clase `X` e `Y` respectivamente. Ambas clases implementan el método `max` que retorna un valor numérico. Suponiendo que la técnica se encuentra buscando la expresión `x max`, y que tanto la clase `X` como `Y` son parte del universo

de búsqueda, se puede dar el caso que el conjunto de *tests* no sea lo suficientemente contundente, y no valide el caso particular de que no necesariamente `x max = y max`. Por lo que, el algoritmo se comporta de tal manera de encontrar uno de los códigos de la igualdad.

Una falla de software puede deberse a una sola línea de código. Tales fallas se denominan *errores de una sola declaración* y sus construcciones sintácticas siguen patrones relativamente simples [10]. Por lo tanto, las técnicas de reparación de software consideran estos errores como un objetivo relevante. Entonces, para evaluar la solución que se lleva a cabo en esta memoria, se emplea GP para construir código de métodos arbitrarios que consten de solo una sentencia en su cuerpo para centrarse únicamente en la generación de código y evitar la localización de errores.

En OOP, es importante la relación entre objetos y métodos de una misma clase. Dado que se busca trabajar en un lenguaje en concreto, podemos asegurar que los individuos generados cumplan con la sintaxis de Pharo. Sin embargo, se limita al hecho de que la solución será el retorno de un valor o una llamada a un método. La técnica solo puede construir código en base a objetos y métodos, descartando de esta forma constantes numéricas, booleanas y estructuras como arreglos. También se descartan los bloques de Pharo, que corresponden a la sintaxis de los *loops* en el lenguaje.

3.2. Criterios de aceptación

La validación de la técnica es una etapa importante en el proceso de investigación. Lo anterior implica que la solución desarrollada debe ponerse a prueba en distintos escenarios. Concretamente, se busca utilizar aplicativos en Pharo que posean métodos que cumplan con las condiciones mencionadas para que la técnica pueda usarlas como objetivo de búsqueda. Cabe señalar la importancia del conjunto de *tests* en este punto, ya que se toman en cuenta aquellos métodos que al modificarse se ve reflejado en el rendimiento de la cantidad de *tests* aprobados. En caso de lo contrario, se podría dar el caso que sea trivial encontrar el código de un método que desde un comienzo no afecta ningún caso de prueba.

Capítulo 4

Solución

Una vez definida la problemática que aborda la memoria, es pertinente describir la solución implementada que resuelve el problema en cuestión. En primer lugar, la sección 4.1 describe la metodología detalladamente, que consiste en los pasos de ejecución que debe seguir la técnica para encontrar una solución aceptable. Por otro lado, la sección 4.2 se refiere a los cambios que se deben hacer para que el programa genético pueda abordar el paradigma de OOP y utilizar el contexto de programas en Pharo.

4.1. Metodología

Para llevar a cabo la solución ofrecida, se presenta la metodología de trabajo a utilizar durante la memoria en la figura 4.1. El proceso se divide en 7 pasos diferentes, donde cada uno describe una fase particular del diseño estructural que permite el funcionamiento de la técnica.

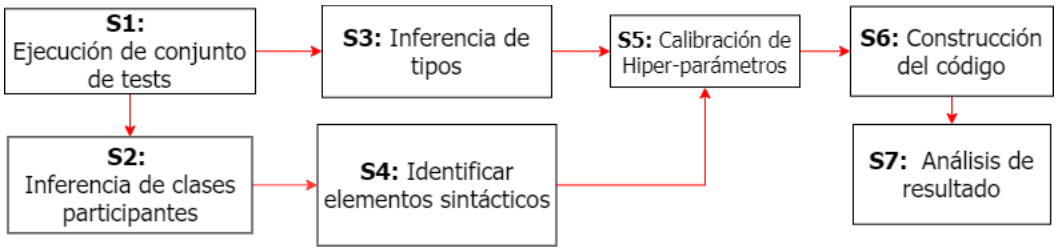

Figura 4.1: Proceso desarrollo de la técnica.

```
1 RSAbstractPlot >> maxValueX
2     "Return the maximum X value of the plot, excluding NaN and infinite"
3     ^ self definedValuesX max
```

Figura 4.2: Método de ejemplo, el objetivo del algoritmo es encontrar el cuerpo compuesto de una única sentencia en la línea número 3.

Para ilustrar el funcionamiento de la técnica, se considera encontrar el cuerpo del método definido en la figura 4.2 llamado `maxValueX`, el cual pertenece a la clase `RSAbtractPlot` y este

a su vez pertenece al paquete `Roassal3-Chart` en la herramienta de visualización Roassal implementada en Pharo [6]. El cuerpo del método corresponde a la línea número 3, la cual retorna el máximo valor definido para valores de eje X. Este ocupa la palabra reservada `self`[1], que envía el mensaje `definedValuesX`. La expresión `self definedValuesX` retorna una colección de valores. Finalmente, el mensaje `max` calcula el máximo valor dentro de esta colección. Dado que los programas se expresan exclusivamente mediante llamadas a métodos en Pharo, la generación de código fuente debe determinar qué otros métodos (definidos en la clase o no) deben utilizarse. Como tal, la secuencia de llamadas debe cumplir con requisitos de tipos no explícitos como en cualquier programa escrito usando lenguaje de programación de tipo dinámico. Esta sección detalla la metodología de trabajo e ilustra cómo se genera un cuerpo de método con solo una declaración(^ `selfdefinedValuesX max` en el ejemplo). La metodología asume la presencia de pruebas unitarias que cubren (directa o indirectamente) el método cuyo cuerpo se debe generar.

S1 - Ejecución de conjunto de *tests*. El primer paso que debe considerar el algoritmo es ejecutar las pruebas unitarias pertenecientes al aplicativo del método a generar. Las pruebas se ejecutan en un entorno controlado en el que se supervisa cada llamada al método. Se utiliza el perfilamiento generado por la herramienta Spy [11] para (1) monitorear todos los valores utilizados como argumentos y valores de retorno, y (2) para extraer las llamadas a métodos del paquete durante el flujo de ejecución.

El modelo obtenido a partir del análisis de las pruebas unitarias es clave para inferir las clases que pueden estar involucradas en el cuerpo del método (S2) e identificar los posibles tipos de las variables de instancia (S3). Además, el número total de *tests* debe estar claramente establecido para definir adecuadamente la función de *fitness*, pues la ejecución de las pruebas unitarias se repite cada vez que se compila el AST de un individuo como el cuerpo del método buscado. Suponiendo que el tamaño del conjunto de pruebas unitarias es igual a N, entonces la función de *fitness* retorna valores de acuerdo a la diferencia numérica entre N y el número de *tests* exitosos una vez que el código candidato a sido compilado. GP entonces tiene como objetivo minimizar esta diferencia a cero.

Existen casos en que el individuo debe ser totalmente descartado pues no compila o genera errores en tiempo de ejecución. En estos casos, se asigna un valor de *fitness* mucho mayor a N. Como ejemplo considere la propuesta de solución de código de la técnica en la figura 4.3, al compilar esta solución y ejecutar el conjunto de casos de prueba, se obtiene un error de bucle infinito, ya que este retorna el objeto actual que llama al mismo método. Es normal que existan otros métodos con el mismo nombre formal, pues pueden existir métodos que se implementan de diferente manera a lo largo de las de clases participantes. De hecho este problema se presenta con cualquier cadena de llamadas donde se tiene un método que retorne el objeto actual[2] (*self*) y que luego se llame a un método con la misma firma formal del método que estamos buscando. Es por lo anterior, que se implementa un mecanismo que captura este tipo de situaciones y castiga a todos los individuos que no terminen la ejecución de los *tests* o presenten un error en tiempo de compilación y ejecución durante la función de *fitness*.

[1] palabra clave que se refiere al objeto actual, equivalente a `this` en Java y C++.

[2] Generalmente, los métodos que realizan mutaciones de variables retornan el objeto actual implícitamente.

```
1 RSAbstractPlot >> maxValueX
2       "Return the maximum X value of the plot, excluding NaN and infinite"
3     ^ self maxValueX
```

Figura 4.3: Ejemplo solución ofrecida por la técnica que produce error de loop infinito

S2 - Inferencia de clases involucradas. La segunda parte de la metodología es obtener información con respecto al contexto en el cual el método es buscado. En particular, es necesario inferir qué clases dentro del ambiente podrían estar involucradas en la generación del código buscado. Estas clases son útiles para definir el universo de búsqueda, el cual es integrado por todos los métodos y variables de instancia (objetos) posibles que podrían ser invocados en el cuerpo del método buscado.

La definición del conjunto de las clases participantes se deduce del modelo obtenido en el paso anterior S1. La cobertura del conjunto de pruebas unitarias define qué parte del código base se debe ejecutar durante la ejecución de los casos de prueba. Las clases que son utilizadas (o cubiertas) total o parcialmente por las pruebas unitarias se consideran elementos que potencialmente pueden estar involucradas en el cuerpo del método que se busca construir. Lógicamente, se incluyen tanto la clase a la cual pertenece el método objetivo, como aquellas que pertenecen al mismo paquete. En el ejemplo de la figura 4.2, se sabe que la clase `RSAbstractPlot` pertenece al paquete `Roassal3-Charts`, dando un total de 41 clases participantes en la ejecución de los *tests*.

En esencia, considerar todos los métodos posibles dentro del ambiente de Pharo es inviable, pues el universo de búsqueda aumentaría demasiado su tamaño, logrando ser lo bastante problemático para que prácticamente sea imposible para GP encontrar un resultado aceptable. Es por lo anterior, que una primera decisión de diseño relevante es que se restringe al algoritmo a producir una solución basada en códigos de otros lugares cercanos, como en la clase o el paquete. Esto produce que los métodos más cercanos al método buscado incidan con mayor probabilidad en el código generado. Esto considera implícitamente que existen métodos cuyas funcionalidades solo inciden de manera interna en la clase [3].

S3 - Inferencia de Tipos. Los lenguajes de programación con tipos dinámicos, como Pharo, no indican explícitamente el tipo de los elementos que forman parte del lenguaje. De esta forma, los programadores confían en el conocimiento y la documentación para deducir los tipos de valores, argumentos, valores de retorno, etc.

En el caso de la técnica desarrollada, es importante deducir la firma de los métodos participantes para acelerar el proceso de búsqueda. La firma de un método consiste en su nombre formal, el número y el tipo de los argumentos, y el tipo de retorno. A partir del resultado del paso S1, esta información se obtiene en función de los valores durante la ejecución de las prue-

[3]Métodos privados.

bas unitarias. Por lo tanto, se construye un modelo de tipos simple que consiste en identificar el tipo de una variable a partir de las clases del objeto contenidas en esa variable durante la ejecución. Las firmas de método se emplearán en los pasos siguientes para calibrar los pesos y generar una secuencia lo más segura posible de tipos de llamadas al método. Nuevamente, la cobertura del conjunto de pruebas es determinante, pues es posible que existan variables que no se utilizaron durante la ejecución, por lo cual no fue posible extraer información alguna sobre ellas.

S4- Identificar elementos sintácticos. Un paso relevante para la programación genética es determinar qué elementos sintácticos son utilizados para representar los nodos de la representación abstracta de los programas candidatos de la solución.

En primer lugar, se extraen las variables de instancia pertenecientes a la clase y sus superclases del método en cuestión. Estas variables son los objetos candidatos que podrán enviar mensajes dentro del código generado. En segundo lugar, es razonable construir y agregar la variable *self* como parte de este conjunto, pues la clase del método es conocida.

Con respecto a los métodos candidatos, estos son extraídos desde las clases participantes obtenidas en paso S2. Sumando además la inferencia de tipos, basta con que una variable entienda cómo enviar el mensaje para ser considerado. A partir del paso S3, se conoce la posibilidad de que el método necesite de argumentos, considerando los tipos y nombres formales correspondientes. Estos elementos también son agregados al universo de búsqueda de GP, ya que sería extraño (o muy poco probable) que un método no ocupe sus argumentos. Por lo tanto, una decisión de diseño importante es asegurar que con altas posibilidades, los argumentos sean ocupados dentro del cuerpo del método producido.

S5 - Calibración de hiper-parámetros. Una vez que recolecta la información sobre el contexto en que el método es usado, la búsqueda del código fuente puede comenzar considerando los hiper-parámetros clásicos asociados a la programación genética (por ejemplo, el tamaño de la población, probabilidad de realizar un operador genético). Sin embargo, existe la posibilidad de que el número de métodos y variables sea muy alto por lo que algunos de ellos nunca sean elegidos para formar parte de un AST. Como consecuencia de lo anterior, el algoritmo podría nunca detectar la solución al problema. Para considerar lo anterior, la técnica asocia pesos a cada elemento sintáctico del universo de búsqueda para que el algoritmo durante la aplicación de los operadores genéticos y en la creación de la población inicial. Se consideran estos pesos como parte de los hiper-parámetros ya que se utilizan para controlar la generación de individuos del proceso evolutivo. Mientras más alto es el valor asignado a un elemento, mayor es la probabilidad de que el algoritmo lo escoja para formar parte de un AST.

Los pesos se determinan mediante la extracción empírica de una gran base de código de Pharo. En particular, se descubre que los métodos definidos en una clase tienen más probabilidades de acceder a las variables definidas en esa clase que las variables definidas en su cadena de superclase. De manera similar, es más probable que un método invoque un método dentro de la misma clase en lugar de invocar un método localizado en la cadena de superclase. La intuición de esta idea es similar a la utilizada en el paso S1 con respecto a la distribución espacial de las clases, métodos y variables.

Empíricamente, sobre la cantidad real de llamadas que realizan los métodos se obtiene que un método que tiene solo una declaración realiza entre 0 y 2 llamadas con una alta probabilidad. Aproximadamente el 73 % de los métodos de declaración única no realizan ninguna llamada (por ejemplo, métodos de acceso y mutación de variables). Por el contrario, solo el

2 % de los métodos de una sola línea de código realiza 6 o más llamadas. Entonces, para construir el cuerpo del método se permite la cadena de invocación de otros métodos, pero limitando a que el largo de la cadena no exceda la longitud máxima de llamadas encontrada empíricamente. Por otro lado, durante las ejecuciones de las pruebas, se determina el número de veces que se invoca cada método. Estos valores se utilizan para generar nodos en el AST. Lo racional es que un método que se ejecuta con frecuencia es más probable que aparezca en el método que se va a generar.

Los experimentos se llevan a cabo sobre métodos cuya funcionalidad es definida en solo una línea de código, entonces un individuo consiste solo en una sentencia que utiliza variables y/o métodos en Pharo. Esta sentencia puede tomar 3 formas dentro de la gramática del lenguaje: código de asignación, llamada a otro método o de retorno. Las probabilidades con que un AST es construido bajo una de estas funcionalidades, corresponden a 15 %, 25 % y 60 % respectivamente. Las asignaciones o mutaciones sobre valores dentro del código tienen prohibido reasignar la variable *self*, pues este debe siempre referir al objeto actual.

S6 - Generación de código El proceso de búsqueda de código es impulsado por el algoritmo de programación genética. La función de ajuste que el algoritmo intenta optimizar es el número de pruebas unitarias aprobadas exitosamente. Como se mencionó en la etapa S1, puede suceder que algunos individuos tarden mucho en ejecutarse. La ejecución ocurre dentro de un hilo dedicado especialmente en controlar los tiempos de ejecución. En caso de que el hilo tome una cantidad de tiempo excesiva, es detenido y penalizado por el controlador, y otorga un valor de *fitness* comparativamente malo al individuo en ejecución.

S7 - Análisis de resultados. Se necesita un mecanismo final que verifica que tan aceptable es la solución ofrecida por GP en el paso anterior. Esta solución maximiza la cantidad de *tests* aprobados. Existen varios resultados posibles:

- El cuerpo del método generado es idéntico al original. En este caso, el algoritmo produjo el mismo código que el programador estableció desde un principio en la aplicación.
- El cuerpo del método generado es sintácticamente diferente pero semánticamente equivalente según el conjunto de pruebas unitario. Por ejemplo, considere la siguiente posible solución de la figura 4.4, considerando el código original al principio de esta sección. El método `definedValuesX` retorna la variable de instancia `xValues` que contiene la colección de valores. Entonces, se ha obtenido una respuesta que no es exactamente la esperada. Sin embargo, es un resultado satisfactorio pues la semántica del código generado al compilarse permite al método pasar todos los *tests*.

```
1 RSAbstractPlot >> maxValueX
2       "Return the maximum X value of the plot, excluding NaN and infinite"
3     ^ xValues max
```

Figura 4.4: Posible solución ofrecida por la técnica

- El cuerpo del método generado es sintáctica y semánticamente diferente al cuerpo del método original:

- Si el valor de *fitness* del mejor individuo no alcanzó a valer 0, significa que algunas pruebas están fallando, entonces el método es demasiado complejo para el algoritmo. En este caso, aumentar la población o ajustar algunos parámetros puede resolver en este caso.
- Si la función de *fitness* indica que todas las pruebas unitarias pasan exitosamente, esto es, que el método no es ejecutado durante las pruebas, lo que significa que el método no es un buen candidato para ser ejecutado por la solución descrita ya que no está cubierto por las pruebas. Para este caso, se deben agregar nuevas pruebas unitarias.

En resumen, la metodología que presenta esta sección está diseñada para usar programación genética para generar cuerpos de método para un programa que (i) está escrito en un lenguaje de programación de tipo dinámico y (ii) está acompañado de un conjunto de pruebas unitarias que definen los requerimientos funcionales. Se aplica la metodología al lenguaje de programación Pharo; sin embargo, no está ligado a él y se cree que se puede aplicar en un lenguaje diferente con características similares. La sección 4.2 analiza los detalles sobre la implementación del algoritmo de programación genética.

4.2. Programación Genética

4.2.1. Representación de programas

La técnica representa cada programa candidato como un árbol de sintaxis abstracta (AST), que incluye una instrucción en una sola línea de código que se compila como el cuerpo del método buscado. Las diferentes maneras en que este código puede estar compuesto o constituido, influye directamente en cómo el programa genético funciona. La figura 4.5 muestra la gramática aceptada por los AST que conforman los individuos de la técnica.

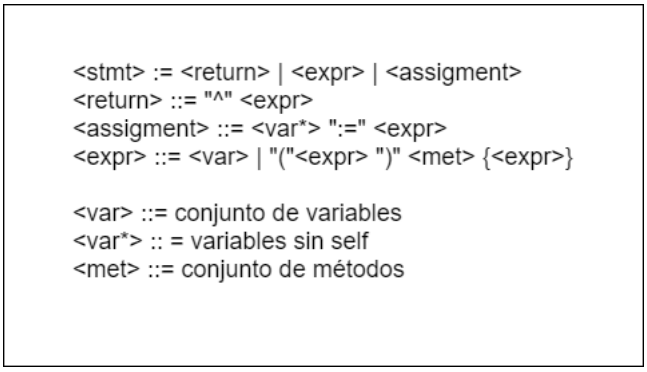

Figura 4.5: Gramática del lenguaje aceptado por los programas representados por GP

Cada vez que GP construye un nuevo individuo, debe seleccionar el tipo de sentencia de manera al azar, donde cada tipo tiene una probabilidad asignada de ser escogido. La implementación considera 3 variaciones de sentencias:

- Expresión de retorno: el programa retorna un valor. Es posible agregar llamadas a métodos para obtener resultados intermedios, como es el caso de `maxValueX`. En el caso de Pharo, estas sentencias comienzan con el símbolo ^.

- Expresión de Asignación o Mutación: programas que realizan una mutación de alguna variable de instancia. En este caso, hay que tener cuidado de no generar una asignación que modifique el estado de alguna palabra reservada en el sistema como *self*.
- Sentencia Indirecta: una llamada indirecta a otro método que realiza otra acción. Retornan el objeto actual indirectamente (*self*).

4.2.2. Selección y operadores genéticos

El clásico algoritmo de la ruleta es usado para la selección de individuos para la siguiente generación. Donde la probabilidad de que un individuo sea escogido es directamente proporcional al *fitness* obtenido con respecto al total de la población. El proceso de selección se refleja en el seudo código de la figura 4.6. Si el tamaño de la población es P, entonces el algoritmo de selección es aplicado $P/2$ veces, como se señala en la línea 4. En cada ejecución se seleccionan dos individuos, a los cuales se les aplica los operadores genéticos sobre sus AST, generando de esta forma dos nuevos individuos.

Algorithm 1: Crear nueva generación

```
1  P = Tamaño población;
2  NP = Nueva población;
3  NP = ∅;
4  for i=1:P/2 do
5  |  (ind1, ind2) = crear 2 nuevos individuos;
6  |  while verificar(ind1, ind2) do
7  |  |  (ind1, ind2) = repetir creación de individuos;
8  |  end while
9  |  NP = NP ⋃{ind1, ind2}
10 end for
11 return NP;
```

Figura 4.6: Algoritmo de creación de nueva población de individuos

Como se ha mencionado en la sección 4.1, durante todo el proceso de búsqueda el conjunto de individuos será consistente tanto sintáctica como semánticamente dentro del lenguaje Pharo. Por lo tanto, los operadores no deben generar individuos irrelevantes, como lo sería una asignación a *self*, o un cuerpo con demasiadas llamadas a otros métodos. Entonces, entre las líneas 6 y 8 de la figura 4.6, cada vez que se intenta agregar un nuevo individuo a la población de la próxima generación, se deben cumplir satisfactoriamente ciertos predicados que verifican que el individuo cumple los requisitos para ser aceptado; y en el caso de que no se cumpla alguna regla, el individuo en cuestión se descarta y se vuelven a aplicar los operadores hasta generar un individuo coherente. Esto, agrega cierto grado de no determinismo pues no se puede predecir el número de veces que se aplicarán los operadores genéticos.

Para la creación de dos nuevos individuos, la técnica considera los dos operadores genéticos tradicionales: cruzamiento entre individuos y mutación. Como se observa en la figura 4.7, el primer operador genera dos nuevos individuos a partir de dos individuos padres. Por otro lado, la mutación se encarga de modificar un subárbol del AST evolucionado. Con respecto a la cantidad de veces que se ejecutan los operadores, el *crossover* se aplica siempre durante el proceso de creación de nuevos individuos a partir de dos individuos de la generación anterior, mientras que la mutación tendrá una probabilidad asociada que indica si se aplica o no.

Algorithm 2: Generación de dos nuevos individuos

```
1  AP = Población actual;
2  TM = tasa de mutación;
3  ind_1 = Ruleta(AP);
4  ind_2 = Ruleta(AP);
5  (c_1, c_2) = crossover(ind_2, ind_2);
6  if (rand(0,1) ≥ TM) then
7  |  c_1 = mutación(c_1);
8  end if
9  if (rand(0,1) ≥ TM) then
10 |  c_2 = mutación(c_2);
11 end if
12 return (c_1, c_2)
```

Figura 4.7: Algoritmo para crear dos nuevos individuos

El cruzamiento es calculado usando un punto de corte entre el AST de uno de los padres, mientras que en el otro padre se selecciona un subárbol. Luego, se intercambian los subárboles con respecto a este punto seleccionado. Esto genera dos nuevos nuevos individuos. En cambio, la mutación se define en términos del cruzamiento entre un individuo de la población actual y otro completamente nuevo generado por el algoritmo.

4.2.3. Función de ajuste, aptitud o *fitness*

Dado un individuo, la función de *fitness* calcula cuantitativamente que tan aceptable y efectiva es la solución ofrecida por éste. Estos valores son importantes para la continuación del algoritmo en futuras generaciones. Como en el algoritmo de selección, que requiere el valor de *fitness* de cada programa para poder discernir cuales tienen mayor aptitud para ser escogidos. El valor de la función de *fitness* F, según un individuo I, es calculado como la diferencia entre el total de *tests* T y el número de *tests* exitosos E aprobados por I:

$$F(I) = T - E$$

Cabe destacar que el objetivo de GP es minimizar este valor a cero. Aquellos métodos que no compilan, generan errores o que realizan demasiadas llamadas, son penalizadas con un alto valor de *fitness*, para que el algoritmo con una alta probabilidad descarte estos programas. Además, es importante notar que el rendimiento de la función de *fitness* depende exclusivamente del conjunto de *tests* unitarios. Si desde un comienzo gran parte de los *tests* son aprobados, será mucho más difícil que la técnica genere buenos resultados. En adición a lo anterior, es muy importante que los *tests* efectivamente cubran las funcionalidades de los métodos buscados.

Capítulo 5

Validación

Uno de los pasos más importantes dentro del desarrollo de software, es demostrar que efectivamente la solución presentada resuelve el problema. Esta validación se realiza ejecutando la técnica en un contexto real de aplicación y reportando los resultados obtenidos. En la sección 5.1 se describe la muestra de métodos utilizados que la solución construida tendrá como objetivo, y también se menciona la calibración de los hiper parámetros que utiliza el algoritmo genético. En segundo lugar, la sección 5.2 expone los distintos sistemas de pesos que complementan a GP en la búsqueda del código, donde cada uno de estos considera particularidades del paradigma OOP. Finalmente, la sección 5.3 y 5.4, se definen casos de estudio relevantes con sus respectivos resultados.

5.1. Experimentos

Para poner a prueba la solución ofrecida en el capítulo anterior, se realizan experimentos sobre la librería de visualización Roassal [6]. Específicamente en el paquete `Roassal3-Chart`. Esta paquete tiene más de 180 métodos, de los cuales el 53 %s tiene un cuerpo de solo una línea de código y son cubiertos por el conjunto de pruebas unitarias. Los métodos abstractos no se consideran, ya que es trivial que las pruebas pasen exitosamente cuando estos se redefinen en una clase menor, y también aquellos métodos que usan bloques (en Pharo, estas son declaraciones de bucles o *loops*). Entonces, la técnica tiene como objetivo construir iterativamente el cuerpo de cada uno de estos métodos por separado. En otras palabras, se asume la definición original de cada método, a excepción del cuerpo que pertenece al método que se quiere producir. Como se mencionó anteriormente, la declaración producida por la técnica solo tiene tres formas posibles: retorno, asignación o una sentencia indirecta.

5.1.1. Muestra y casos de prueba

La librería cuenta con un paquete de testeo llamado `Roassal3-Chart-Tests` con 68 *tests* unitarios. En primer lugar, se extraen los métodos de Roassal de una sola línea que participan de algún modo en los casos de prueba para formar la muestra de los métodos que la técnica deberá buscar. Se cuenta con un total de 34 métodos que participan estáticamente en los *tests*, es decir, se utilizan directamente en el código de estos. Por otro lado, estos métodos

pueden realizar llamadas a otros métodos durante el flujo de ejecución que no están necesariamente presentes de forma estática. Por lo que se cuenta con 65 métodos diferentes que se utilizan indirectamente. Se realiza esta distinción, pues una de las premisas de investigación es que la solución implementada necesita un conjunto de pruebas que cubra eficazmente las funcionalidades que se busca que la aplicación cumpla. Lo anterior, debido a que el cálculo de *fitness* es usado para determinar cuales individuos resuelven mejor el problema. Por ejemplo, si antes de iniciar la búsqueda, un 80 % de los *tests* aprueba exitosamente con un cuerpo de método cualquiera, entonces será más difícil para la técnica encontrar el código correcto, ya que los margenes de mejora se vuelven más finos, es decir, las posibles transformaciones que sufran los individuos según los operadores genéticos deben ser más precisas, cambiando detalles en el código.

Subdivisión de la muestra

. Se cuenta con un total de 99 métodos en la muestra. Este conjunto a su vez puede ser subdividido, dependiendo de la funcionalidad que cumple cada método dentro de la aplicación:

- Los métodos que retornan una variable de instancia son llamados accesores (o *accessors*). Por ejemplo, GP es capaz de encontrar el método `shape` de la clase `RSLinePlot` presente en la figura 5.1. Este método simplemente retorna la variable de instancia *shape*, que pertenece a la misma clase. Hay un total de 33 métodos accesores.

```
1 RSLinePlot >> shape
2     "Return instance variable shape"
3     ^ shape
```

Figura 5.1: *Accessor* perteneciente a los métodos que son buscados por la técnica desarrollada.

- Los *setters* son aquellos métodos que mutan valores de una variable de instancia en una clase. Estos métodos necesitan un argumento que será el nuevo valor asignado. En total, se cuenta con 23 de estos métodos en la muestra. En la figura 5.2 se aprecia un ejemplo de método de esta categoría.

```
1 RSLinePlot >> shape: aShape
2     shape := aShape
```

Figura 5.2: Ejemplo de *setter* que es encontrado por el algoritmo.

- Una tercera categoría de métodos *complejos* es considerada. Este conjunto contiene a aquellos métodos que realizan llamados a otros métodos. En el capítulo anterior, el método `maxValueX` pertenece a este grupo. El 43 % de la muestra son métodos de este tipo.

5.1.2. Hiper-Parámetros

La técnica necesita parámetros que controlen el flujo de ejecución. En primer lugar, se consideran los hiper-parámetros típicos de la programación genética. El tamaño de la pobla-

ción es de 30 individuos. Mientras que la tasa de mutación es del 20 %. El número máximo de generaciones es 30. Por otro lado, algunos parámetros necesitaron una previa investigación empírica. Por ejemplo, los individuos que se construyen deben considerar el contexto de OOP para construir programas consistentes. A priori, una línea de código puede generar M llamadas arbitrarias a otros métodos. En la realidad, el número de llamadas es bastante limitado.

En el gráfico de la figura 5.3, los métodos se distribuyen según el número de llamadas reales que realizan. Estos porcentajes se utilizan en GP, donde si p % de los métodos hace n llamadas, entonces con probabilidad p, GP crea una expresión que hace n llamadas en el cuerpo del método.

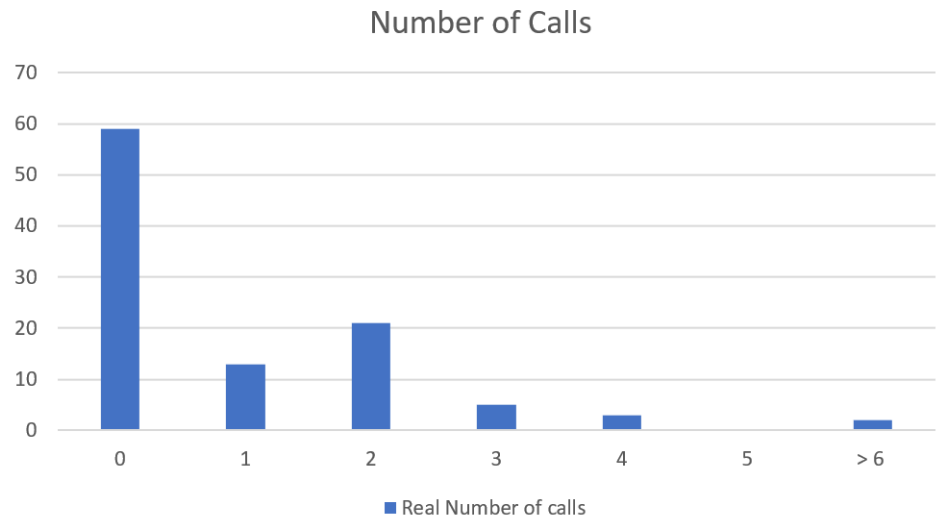

Figura 5.3: Distribución de cantidad de llamadas reales de métodos en Roassal.

Además, al aplicar operaciones genéticas es posible que aumente el número de llamadas de un individuo, por lo que revisa si el programa cumple o no con un cierto número de llamadas (este número se decide de manera al azar). Si el individuo no cumple, se descarta y las operaciones genéticas se aplican nuevamente. Dentro de los métodos de Roassal, el 57 % no llama a otros métodos, sino que devuelve un valor. El 33 % realiza entre 1 y 2 llamadas. Por lo tanto, es sensato que un individuo tenga muy pocas posibilidades de poseer un AST que realice más de 2 llamadas.

5.1.3. Optimizaciones

Cuando el programa genético calcula el *fitness*, se guarda este valor en memoria para futuras consultas. Entonces aquellos individuos que son copiados a la próxima generación, no recalculan su valor asociado. Lógicamente, si un individuo cambia su AST, el *fitness* debe ser calculado nuevamente.

5.2. Sistema de pesos

La necesidad de tomar en cuenta el contexto en que vive el método buscado surge del hecho de que GP necesita crear individuos de acuerdo al universo de variables y métodos disponibles, pero este conjunto crece enormemente si se consideran todos los elementos accesibles por las clases participantes. En consecuencia, GP podría explorar solo una fracción restrictiva del universo dependiendo del tamaño de la población, o en peor caso, jamás utilizar componentes relevantes que generan el resultado esperado. Por lo tanto es necesario construir una estructura que genere un *ranking* de prioridades entre los métodos y variables que influya en el algoritmo a la hora de escoger componentes para el código generado. En el fondo se busca concentrar la búsqueda en un grupo de métodos y variables que cumplan cierto predicado. Este último debe particionar el total de elementos del universo de búsqueda otorgando mayor probabilidad de elección a aquellos que cumplen con las reglas que define el predicado. Por ejemplo, si se considera un predicado que siempre es verdadero o constante entre los elementos, significa que todos los elementos tienen la misma relación con el predicado. Por lo que se obtiene un sistema de pesos donde todos los elementos tienen igual probabilidad de ser escogidos. Concretamente, el sistema asigna valores (pesos) a cada entidad, lo cual el algoritmo considera como probabilidades al momento de seleccionar y construir un nuevo nodo de AST.

La técnica se ejecuta en distintos escenarios con respecto al sistema de pesos que se utiliza durante la selección de métodos y variables. Cabe destacar que independientemente del sistema utilizado, la técnica cuenta con la información de tipos y la limitación de números de llamadas para construir programas que sí tengan sentido.

- **Naive**: Sin sistema de pesos, equivalente a un algoritmo clásico de programación genética.
- **Uniforme**: Todos los métodos tienen la misma probabilidad de ser escogidos. Por otro lado, las variables dentro de una misma clase tienen probabilidad uniforme. Sin embargo, a medida que se avanza en las superclases la probabilidad va disminuyendo. Por lo tanto, las variables pertenecientes a la misma clase que el método buscado, tienen mayores chances de ser escogidas para la solución construida por el algoritmo.

  La asignación de pesos procede de la siguiente manera:

  1. Se inicia el proceso desde la clase a la cual pertenece el método buscado, considerando un valor máximo inicial W_o. La única excepción es que a la variable `self` se le asigna el doble de este valor.
  2. Luego, cada variable dentro de una misma clase tienen el mismo peso w.
  3. A medida que se avanza en la jerarquía de clases, se considera lo siguiente: si A es la clase padre de B y el peso de las variables de B es w, entonces cada variable de la clase A tienen peso igual a $w/2$.
  4. Finalmente, a cada elemento fuera de la jerarquía de clases, se le asigna un valor mínimo de peso.

  La tabla 5.1 describe un ejemplo de asignación uniforme de pesos para las variables según jerarquía de clases, donde el método buscado es `maxValueX` con $W_o = 500$. La jerarquía de clases se compone de `RSAbstractPlot` (la clase del método buscado) y su clase padre `RSAbstractChartElement`.

Clase	Variable	Peso Asignado
RSAbstractPlot	self	1000
	xValues	500
	yValues	500
	shape	500
RSAbstractChartElement	chart	250
	xScale	250
	yScale	250

Tabla 5.1: Ejemplo de asignación de pesos según jerarquía de clases, con $W_o = 500$ y método buscado es `maxValueX`

- **Uniforme jerárquico**: tanto las variables como los métodos siguen el proceso de asignación descrito anteriormente. En el caso de los métodos fuera de la jerarquía de clases, se les asigna un valor de peso mínimo igual a 1.
- **Similitud**: Este sistema de pesos considera otro aspecto particular de la programación orientada a objetos. En particular, asigna valores según la *similitud textual* entre los distintos elementos sintácticos y el método buscado. Para esto se utiliza la distancia de Levenshtein [12] calculada entre nombre formales de cada elemento con el nombre del método buscado. La intuición que busca aplicar este sistema es que los elementos estructurales, como las clases y métodos, que poseen un nombre similar, probablemente poseen una definición/tarea similar. En general, los nombres dados a los elementos estructurales pueden expresar aspectos de la colaboración, funcionalidad y responsabilidad que el elemento debe tener y, por lo tanto, tener alguna similitud en su definición formal. Por lo anterior, es razonable considerar la similitud textual entre el método buscado y los miembros del universo de búsqueda.

  El algoritmo de Levenshtein calcula la similitud entre dos cadenas de texto w_1 y w_2, contando el número de inserciones, intercambios o eliminaciones de caracteres que se deben hacer para transformar w_1 en w_2. Por lo tanto, la mejor similitud posible es 0, es decir, no es necesario hacer ninguna modificación. El peso asignado a un elemento estructural e se calcula como:

  $$\mathit{Weight}(e) = C - \alpha * \mathit{Similarity}(\mathrm{e})$$

  Donde para los experimentos se utiliza $C = 1000$ y $\alpha = 10$. Mientras más cercano a cero la similitud, mayor será el valor asignado a un método o variable de ser escogido. Siguiendo el ejemplo de la tabla 5.1, el valor de $\mathit{Weight}(\texttt{xValues}) = 1000 - 10 * 3 = 970$, ya que se requieren 4 modificaciones para transformar `xValues` en `maxValueX`.
- **Ponderación**: Este sistema combina tanto la jerarquía de clases, como la similitud textual. El peso final se calcula como una suma ponderada entre estos dos sistemas:

  $$\mathit{Weight}(I) = a * \mathit{Similarity}(I) + b * \mathit{Hierarchy}(I), \quad \text{con a + b = 1}$$

  Como ejemplo, si se utiliza $a = b = 0{,}5$, el peso de la variable `xValues` se calcula como: $\mathit{Weight}(\texttt{xValues}) = 970 * 0{,}5 + 500 * 0{,}5 = 735$.

5.3. Experimentos sobre la muestra de métodos

En esta sección se detalla un experimento realizado sobre el total de la muestra explicada en la sección 5.1. En particular se emplea la técnica en dos de los escenarios explicados en la sección anterior: (1) utilizando el sistema de pesos uniforme, y (2) considerando la similitud textual. Se reportan los resultados obtenidos por cada sistema aplicando la solución implementada para cada método de la muestra. Este experimento busca analizar la capacidad de la técnica en encontrar métodos según su funcionalidad (*accessors*, *setters* o *complejo*). Por otro lado, se clasifican los métodos considerando el hilo de ejecución de los *tests*, es decir, se separan aquellos métodos que son llamados sintácticamente en el código de las pruebas (*llamada directa*), de los métodos que son referenciados en la ejecución por el llamado de otro método (*llamada indirecta*).

5.3.1. Resultados de los experimentos utilizando técnica con sistema uniforme de pesos

La tabla 5.2 resume los resultados experimentales donde se emplea un sistema de pesos uniforme sobre los métodos que conforman la muestra explicada en la sección 5.1.1. Se mencionó anteriormente que se distinguen a los métodos entre dos categorías que toman en cuenta el flujo de ejecución de los casos de prueba:

- 34 métodos llamados directamente por los *tests* (es decir, el código contiene una referencia al método), y
- 65 métodos que son llamados indirectamente por los *tests* (la referencia se genera por el llamado de otro método).

La intuición que se busca explorar es observar si mientras mayor es la distancia, en términos de llamadas intermedias, entre el método a generar y la prueba unitaria; más difícil es la generación del código para la técnica. Los métodos generados correctamente son aquellos que lograron tener una aptitud 0. La columna Tiempo es la estimación según la suma de todos los procesos de búsqueda.

Métodos	Sistema	Soluciones Correctas	Tiempo (DD:HH:MM:SS)
Llamados directamente (34)	Uniforme	16	01:10:17:32
Llamados indirectamente (65)	Uniforme	28	02:17:13:17

Tabla 5.2: Resultados de experimentos utilizando un sistema de pesos uniforme sobre métodos.

Caracterización de métodos encontrados

Entonces, si la técnica utiliza pesos uniformes para todos los métodos, mientras que los valores para las variables se definen según la jerarquía de clases, GP genera correctamente 44 métodos de la muestra utilizada (= 44 %). La Tabla 5.3 detalla el número de métodos divididos por *accesors*, *setters* y complejos.

A partir de esta información se observa que generar *accesors* no representa un gran desafío para GP incluso con una selección uniforme entre métodos. Solo es necesario seleccionar la

Métodos	Llamados Directamente	Llamados indirectamente	%
Accessors	11	19	91 %
Setters	0	9	39 %
Complejos	5	0	12 %

Tabla 5.3: Caracterización métodos encontrados por el sistema uniforme de pesos.

variable correspondiente (que forma parte del universo de búsqueda) y formar un AST del tipo retorno, lo que tiene una alta probabilidad de que ocurra, ya que las variables representan una porción mucho menor dentro del universo. Por otro lado, existe la posibilidad de encontrar *setters* con éxito pero con menor precisión en el caso de los que son llamados directamente. Esto debería deberse al hecho de que existen más restricciones al construir una asignación, ya que la variable de instancia debe coincidir con el valor en cuestión (nuevamente, existe una alta probabilidad de usar el argumento). Por último, es notable mencionar que solo fue posible construir código que cumpla con la semántica necesaria para cumplir con las pruebas, sin lograr la sintaxis esperada. Esto implica romper una importante regla de programación orientada a objetos sobre la delegación de responsabilidades de un elemento dentro de la aplicación. Por ejemplo, la solución ofrecida para `maxValueX` ofrecida en la figura 4.4, no cumple con la delegación que correspondería otorgar al llamado de `definedValuesX`. Otro aspecto relevante, y que confirma una pregunta de investigación, es que la solución obtenida por la técnica es sensible a la calidad del conjunto de pruebas unitarias. Por ejemplo, el método `isVerticalBarPlot` de la clase `RSAbstractPlot` retorna siempre el valor booleano falso. Entonces, GP buscará una línea de código que satisfaga esto, la solución que ofrece GP es llamar a otro método `isLinePlot` de la misma clase y que también devuelve falso; en términos de semántica, esto no supone ningún problema. Pero con respecto a la programación orientada a objetos, pueden causar problemas en las clases hijas de `RSAbstractPlot`, pues si se redefine el método `isLinePlot`, inevitablemente causaría fallas en los *tests*. Lo anterior, se resuelve agregando una prueba que asegure que el resultado de ambos métodos no tiene que coincidir necesariamente de acuerdo con un contexto de uso de la aplicación.

5.3.2. Resultados de los experimentos utilizando técnica con sistema de pesos sobre similitud textual

La tabla 5.4 resume los resultados obtenidos utilizando similitud textual como sistema de pesos para métodos y variables. Comparado con el sistema uniforme, la técnica es capaz de encontrar más soluciones en menos tiempo.

Métodos	Sistema	Soluciones Correctas	Tiempo (DD:HH:MM:SS)
Llamados directamente (34)	Similitud	17	01:08:18:07
Llamados indirectamente (65)	Similitud	34	02:05:16:39

Tabla 5.4: Resultados experimentales con similitud textual entre el nombre formal del método buscado y el universo de búsqueda.

Caracterización de métodos encontrados

Si la información entre las tablas 5.2 y 5.4 es comparada, se observa que ambos sistemas son efectivos en generar métodos llamados directamente en el código de los *tests*. Sin

embargo, como ya se ha presentado anteriormente, emplear la similitud textual como sistema de pesos aumenta considerablemente la eficacia de la búsqueda en algunos casos como se exponen en la tabla 5.5. En particular, aumenta el número de *setters* encontrados, esto ocurre probablemente porque la similitud es máxima entre el nombre formal del método y la variable a ser mutada. Por otra parte, se obtienen resultados cuantitativamente equivalentes con el sistema uniforme en cuanto a métodos complejos (5), pero en el caso de la similitud existen casos donde se mantiene la lógica de la delegación de tareas dentro de la aplicación, lo que no ocurre con el sistema uniforme.

Métodos	Directamente	Indirectamente	%
Accessors	11	21	97 %
Setters	3	11	61 %
Complejos	3	2	12 %

Tabla 5.5: Caracterización de los métodos encontrados por similitud textual.

5.4. Experimentos sobre distintos sistemas de pesos

En esta sección, se presentan experimentos que se aplican sobre un método complejo aplicando distintos sistemas de pesos. Uno de los objetivos del sistema de pesos es colaborar con el algoritmo de programación genética para optimizar los tiempos de búsqueda. Si se considera un método complejo, es decir, su cuerpo contiene al menos una llamada a otros mensajes. Este es el caso de `maxValueX` presente en el capítulo 4. Al recordar la definición del cuerpo de este método: ^ `self definedValuesX max`, donde cabe destacar que se realizan 2 llamadas adicionales y que `self` es el elemento que cumple la función de emisor del primer mensaje. Por lo tanto, es interesante utilizar los distintos sistemas de pesos disponibles, para comparar soluciones encontradas y tiempos de búsqueda. Los sistemas de pesos utilizados para experimentar son los mencionados en la sección 5.2. Cabe destacar que, independiente del sistema de pesos, el algoritmo en todo momento toma en cuenta la restricción de cantidad de llamadas de métodos y la inferencia de tipos extraída en la calibración previa de la técnica.

5.4.1. Caracterización de los resultados

La exposición de los resultados, se divide entre los distintos sistemas de pesos utilizados, ya que es la única diferencia entre cada escenario. Las columnas utilizadas en las tablas son las siguientes:

- Solución: código final generado por la técnica.
- *Fitness* de la solución.
- Promedio mejor *fitness* (PMF) a lo largo de las generaciones.
- Promedio peor *fitness* (PPF) a lo largo de las generaciones.
- Última generación (UG) activa del algoritmo.
- Última generación en que cambió el mejor *fitness* de la población actual (FG). Por ejemplo, si la solución es encontrada en la generación 6, necesariamente fue la última vez que el valor de *fitness* mejoró, entonces esta columna también vale 6. Esto permite detectar falsos positivos apresurados del algoritmo.

- Tiempo creación de población (TCP).
- Tiempo total generaciones. (TTG).
- Tiempo total. (TCP+TTG)

5.4.2. Experimento N°1: Naive

En las tabla 5.6 se reflejan los resultados para la técnica sin utilizar un sistema de pesos. Cabe destacar que la técnica es capaz de encontrar una solución aceptable el 40 % de las veces. Incluso en el primer experimento, la técnica construye eficazmente en la primera generación. Además, las soluciones encontradas son bastante aceptables, donde una variable simple realiza un llamado a otro método. Lo anterior se atribuye a lo útil que es calibrar previamente al algoritmo dado el contexto donde reside el método. También, se utiliza la limitación de llamadas e inferencia de tipos sobre el lenguaje OOP, que implícitamente utiliza el contexto de objetos y métodos. Sin embargo, como se observa en la tabla 5.7, esta versión aun sigue siendo demasiado inestable en cuanto a tiempos de búsqueda e impreciso a la hora de encontrar la solución, pues los tiempos entre los experimento positivos (1 y 5) difieren bastante.

Experimento	Solución	Fitness Solución	PMF	PPF
1	^ (xValues max)	0	0,00	55,00
2	^ (yValues max)	8	13,60	6055,60
3	^ (yValues min)	14	37,23	4055,40
4	^ (yValues min)	14	32,07	2055,20
5	^ (xValues max)	0	23,31	3133,08

Tabla 5.6: Resultados experimentales sin sistema de pesos.

Experimento	UG	FG	TCP	TTG	TCP + TTG
1	1	1	0:00:05:01.973776	0:00:05:03.790203	0:00:10:05.763979
2	30	28	0:00:05:52.684635	0:03:11:11.550000	0:03:17:04.234635
3	30	18	0:00:06:19.513805	0:03:18:34.577440	0:03:24:54.91245
4	30	17	0:00:06:48.298964	0:03:04:09.699749	0:03:10:57.998713
5	18	18	0:00:06:33.518487	0:01:54:35.212371	0:02:01:08.730858
TOTAL			0:00:29:35.989667	0:11:33:34.829763	0:12:03:10.819430

Tabla 5.7: Tiempos experimentales sin sistema de pesos.

5.4.3. Experimento N°2: Uniforme

En el caso del sistema uniforme de pesos, la técnica es capaz de encontrar la solución un 40 % de las veces pero en menor tiempo con respecto a la técnica sin un sistema de pesos. Además, se aprecia una mayor variabilidad en las soluciones construidas, utilizando variables y métodos que hasta al momento no se habían ofrecido como solución.

Experimento	Solución	Fitness Solución	PMF	PPF
1	^xValues	55	55,00	4422,07
2	^ (xValues max)	0	36,7	55
3	^ (xValues max)	0	15,8	2050,4
4	^ (self maxValueY)	8	11,1	720,1
5	^ (xValues average)	12	42,1	2455,2

Tabla 5.8: Resultados experimentales con sistema de pesos uniforme.

Experimento	UG	FG	TCP	TTG	TCP + TTG
1	30	1	0:00:07:21.049312	0:03:07:12.352736	0:03:14:33.402048
2	3	3	0:00:06:31.482238	0:00:19:34.446714	0:00:26:05.928652
3	5	5	0:00:08:20.257608	0:00:39:39.288044	0:00:47:59.545652
4	30	3	0:00:07:29.484332	0:03:44:35.529969	0:03:52:05.014301
5	30	22	0:00:05:42.376706	0:03:04:09.301189	0:03:09:51.677895
TOTAL			0:00:35:24.650196	0:10:55:10.918652	0:11:30:35.568548

Tabla 5.9: Tiempos experimentales con sistema uniforme de pesos.

5.4.4. Experimento N°3: Uniforme jerárquico

En el caso del sistema que considera completamente la jerarquía, ya que se incluyen los métodos en el proceso, se observan los peores resultados obtenidos pues el algoritmo no fue capaz de encontrar una solución en ninguna iteración. Además de lograr los peores tiempos de búsqueda.

Experimento	Solución	Fitness Solución	PMF	PPF
1	^ (self minValueY)	14	46,9	2055,2
2	^ (self index)	16	16,0	7453,7
3	^ (self maxValueY)	8	25,6	4055,4
4	^ (self maxValueY)	8	10,6	10383,4
5	^ (self minValueY)	14	27,7	14056,4

Tabla 5.10: Resultados experimentales con sistema de pesos de jerarquía de clases.

Experimento	UG	FG	TCP	TTG	TCP + TTG
1	30	26	0:00:06:29.587788	0:03:11:30.884596	0:03:18:00.472384
2	30	26	0:00:06:30.669488	0:03:27:51.075294	0:03:34:22.744782
3	30	22	0:00:06:20.897742	0:02:58:57.864692	0:03:05:18.762434
4	30	6	0:00:05:47.623330	0:03:07:34.663185	0:03:13:22.286515
5	30	11	0:00:06:57.617122	0:03:05:07.471531	0:03:12:05.886530
TOTAL			0:00:32:07.395470	0:15:51:01.959298	0:16:23:09.354768

Tabla 5.11: Tiempos experimentales con sistema jerárquico de pesos.

5.4.5. Experimento N°4: Similitud textual

La similitud textual toma en cuenta aquellos elementos sintácticos que son semejantes al nombre formal del método buscado. Por lo que es natural que, variables como `xValues` y `yValues` aparezcan en las soluciones ofrecidas por este sistema. Como se observa en la tabla 5.12, el algoritmo fue capaz de construir una solución el 100 % de las veces. También, destacar que los tiempos obtenidos por cada experimento varían bastante, observando que en algunos fueron necesarios menos de 1/6 de las generaciones, mientras que en otros fue necesario más del 60 % de las iteraciones permitidas.

Experimento	Solución	Fitness Solución	PMF	PPF
1	^ (xValues max)	0	41,7	2664,0
2	^ (xValues max)	0	31,7	55,0
3	^ (xValues max)	0	15,8	12056,2
4	^ (xValues max)	0	12,3	6916,6
5	^ (xValues max)	0	27,5	55,0

Tabla 5.12: Resultados experimentales con sistema de pesos considerando similitud entre elementos sintácticos y el nombre formal del método buscado.

Experimento	UG	FG	TCP	TTG	TCP + TTG
1	23	23	0:00:06:01.927870	0:02:11:31.923144	0:02:17:33.015931
2	20	20	0:00:05:30.343105	0:01:52:17.432754	0:01:57:47.775859
3	5	5	0:00:05:35.616709	0:00:29:46.183402	0:00:35:21.800111
4	22	22	0:00:05:43.633278	0:02:08:14.66207	0:02:13:57.699485
5	2	2	0:00:05:34.352414	0:00:11:21.350042	0:00:16:55:702456
TOTAL			0:00:28:23.382930	0:06:53:10.955549	0:07:21:33.993842

Tabla 5.13: Tiempos experimentales con sistema de similitud textual de pesos.

5.4.6. Experimento N°5: Ponderado

Con respecto a un último escenario, donde se complementan la similitud textual con la jerarquía de clases, se aprecia que la técnica encuentra un 80 % de las veces una solución como se aprecia en la tabla 5.14, lo que vendría siendo peor que considerar solamente la similitud textual como en el experimento n°4. Sin embargo, la tabla 5.15 expone que los tiempos de búsqueda mejoran considerablemente, más aún, esta diferencia se hace más notoria al quitar los mejores/peores iteraciones de cada sistema. Esto da a entender que una buena combinación de estos sistemas puede mejorar la eficacia temporal de la técnica. Ahora corresponde evaluar si los ponderadores (a y b) influyen en el trabajo de la técnica con otros valores.

Experimento	Solución	Fitness Solución	PMF	PPF
1	^ (yValues min)	14	35,9	4055,4
2	^ (xValues max)	0	0	55,0
3	^ (xValues max)	0	49,24	3584,76
4	^ (xValues max)	0	47,14	8627,29
5	^ (xValues max)	0	4,0	55,0

Tabla 5.14: Resultados experimentales con sistema ponderado de pesos ($a = b = 0{,}5$).

Experimento	UG	FG	TCP	TTG	TCP + TTG
1	30	17	0:00:06:13.674096	0:03:10:08.802578	0:03:16:22.476674
2	1	1	0:00:06:11.157683	0:00:06:20.542482	0:00:12:31:700165
3	17	17	0:00:06:12.839423	0:01:47:39.767459	0:01:53:52.606882
4	7	7	0:00:06:06.657339	0:00:43:20.114508	0:00:49:32.771847
5	2	2	0:00:06:05.168694	0:00:12:29.926079	0:00:18:35.094773
TOTAL			0:00:30:55.497235	0:05:59:59.153106	0:06:30:54.650341

Tabla 5.15: Tiempos experimentales con sistema ponderado de pesos ($a = b = 0,5$).

5.5. Sumario de experimentos

En resumen, los experimentos en la sección 5.3 expone la capacidad de la técnica para encontrar 99 métodos distintos dentro de una aplicación. En particular, hay métodos que son simples de encontrar independientemente del sistema de pesos utilizado (*accessors* y *setters*). Además, no se aprecia una relación importante entre los resultados y los métodos que son utilizados directa o indirectamente en los *tests*. Los sistemas considerados utilizan distintos componentes de la programación orientada a objetos. Por un lado, el sistema uniforme considera la jerarquía de clases de las variables de instancia. Mientras que la similitud textual considera las relaciones de colaboración funcionales que pueden tener tanto métodos como clases. Este último sistema permite encontrar mayor cantidad de métodos, en menor tiempo y con la particularidad de que con los métodos más complejos encontrados, se mantienen las relaciones de delegación características de la OOP.

Los experimentos realizados en la sección 5.4 permiten identificar que sistemas de pesos son más efectivos frente a la búsqueda del cuerpo de un método complejo. En primer lugar,

la solución encontrada por el algoritmo siempre fue la misma expresión independiente del sistema utilizado: ^ `xValues max`. Esto ocurre debido a que la variable de instancia empleada se le asignan altos valores de peso en las distintas heurísticas. Además, las soluciones ofrecidas solo realizan una llamada adicional, tal como se define empíricamente en la figura 5.3 alrededor del 70 % de los individuos generados realizan menos de 2 llamadas. Por lo que si se da mayor flexibilidad a los valores asociados a los porcentajes, se obtendrían soluciones con mayor cantidad de llamadas. Con respecto a la efectividad de cada sistema, es interesante apreciar que el experimento n°3 es bastante precario por sí solo. Sin embargo, al complementarlo con el sistema que considera la similitud textual, los tiempos de búsqueda mejoran considerablemente. Se concluye entonces que la combinación de distintas heurísticas que consideran distintos elementos del contexto de la programación orientada a objetos es la más efectiva.

Capítulo 6

Conclusión

En un principio, esta investigación propuso demostrar la factibilidad de la generación de código fuente en lenguajes con programación orientada a objetos y tipos dinámicos. De acuerdo a los resultados obtenidos en los experimentos realizados, es posible afirmar que, efectivamente, es posible la construcción de código en un lenguaje de estas características utilizando una novedosa técnica que construye una solución que se adapta al contexto OOP. En una segunda pregunta de investigación, la semántica de la solución que construye la técnica depende de la calidad del conjunto de pruebas unitarias que se recibe como entrada del algoritmo. Este conjunto es parte fundamental de la búsqueda pues define los requerimientos funcionales que explícitamente debe cumplir la solución provista por la técnica. La técnica desarrollada es capaz de encontrar un total de 51 métodos de una sola línea de forma satisfactoria de un total de 99. Además, es notable que cada sistema de pesos experimentado posee sus propias ventajas; esto motiva a buscar una armonía entre los sistemas y considerar otros aspectos que aún no se consideran, de tal manera de incrementar la precisión para encontrar toda clase de métodos.

Para resumir, desde el contexto de programación en que el método buscado es utilizado, es posible extraer información importante para encontrar el código requerido. El algoritmo usa programación genética combinado con un sistema de pesos que toma en cuenta el contexto del problema. Esto permite escalar a una solución que encuentra cuerpos de código no triviales (métodos que realizan llamadas a otros métodos). Concretamente, los resultados obtenidos señalan que al usar como sistema de pesos la combinación de distintos aspectos de la programación orientada a objetos, como la jerarquía de clases o la modularidad para designar responsabilidades (similitud textual), permite obtener una técnica más efectiva, tanto en tiempos de búsqueda como en construcción de una solución pertinente.

Sin embargo, dado que la técnica funciona bajo un escenario en que los métodos que se buscan son de una sola sentencia de código, impide a que esta investigación sea la versión final para ofrecer un mecanismo que construya parches correctivos de programas escritos en lenguajes OOP con tipos dinámicos. Por lo tanto, una mejora importante para esta memoria sería considerar el caso en que los métodos tengan mayor cantidad de líneas de código. Además, los experimentos son validados dentro de un mismo entorno de aplicación (Roassal), por lo que también sería interesante probar la técnica con otras librerías de Pharo. A pesar

de estas dificultades mencionadas, este trabajo es un aporte al área de reparación de software y que puede dar hincapié a que en un futuro parches correctivos puedan ser generados para programa escritos en Pharo o similares.

Bibliografía

[1] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. *A genetic programming approach to automated software repair*. Proceedings of the 11th Annual conference on Genetic and evolutionary computation , GECCO '09, July, 2009, 2009.

[2] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 3–13. IEEE Press, 2012.

[3] L. Lamport. *The 'Hoare logic' of concurrent programs.* Acta Informatica 14, 21–37, 1980.

[4] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[5] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[6] Alexandre Bergel. *Agile Visualization*. LULU Press, 2016.

[7] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[8] John R. Koza I and Riccardo Poli. *A Genetic Programming Tutorial. Introductory Tutorials in Optimization, Search and Decision Support*. Chapter 8, 2003.

[9] Shumeet Baluja and Rich Caruana. *Removing the Genetics from the Standard Genetic Algorithm*. ICML, 1995.

[10] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 573–577, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Computer Languages, Systems & Structures* (Volume 38), April, 2012.

[12] Rishin Haldar and Debajyoti Mukhopadhyay. *Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach.* 2011.