UNIVERSIDAD DE ANTIOQUIA

LENGUAJE ESPECÍFICO DE DOMINIO PARA LA DESCRIPCIÓN DE METAHEURÍSTICAS PARALELAS

Autor

Santiago Bedoya Betancur

Universidad de Antioquia

Facultad de Ingeniería

Departamento de Ingeniería de Sistemas

Medellín, Colombia

2021

Lenguaje específico de dominio para la descripción de metaheurísticas paralelas.

Santiago Bedoya Betancur

Trabajo de Grado como requisito para optar al título de:
Ingeniero de Sistemas

Asesor:
Danny Alexandro Múnera

Universidad de Antioquia

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas
Medellín, Colombia
2021

Índice

Índice de tablas

Índice de figuras

Índice de Códigos

1. Resumen

Las metaheurísticas son estrategias generales que guían a una serie de heurísticas a limitar el espacio de búsqueda de un problema. Los más recientes esfuerzos en el campo se han centrado en el desarrollo de metaheurísticas híbridas y paralelas las cuales poseen un diseño e implementación más complejos, necesitando así de conocimientos concretos en el campo del paradigma de programación paralela. Existe entonces una brecha entre los conocimientos de programación de los expertos diseñadores de metaheurísticas y los programadores de aplicaciones paralelas. Una posible solución es un Lenguaje Específico de Dominio (DSL por sus siglas en inglés) el cual es un lenguaje de programación que cuenta con funciones y sintaxis cercanas o propiamente extraídas del dominio en el que se encuentra enfocado. En este proyecto se analiza, diseña, implementa y valida un DSL construido en conjunto con un experto del dominio. Finalmente se encontró que el DSL propuesto permite disminuir la brecha que poseen los expertos del dominio en los conocimientos de programación paralela, permitiendo ejecutar metaheurísticas de manera paralela con solo describir la forma de la ejecución esperada, sin incurrir en la modificación directa de código.

Palabras clave: Metaheurísticas, Lenguaje Específico de Dominio, Paralelismo, Intérpretes

2. Introducción

Las metaheurísticas son a menudo el enfoque más eficiente para abordar los más difíciles problemas de optimización combinatoria. Son procedimientos de alto nivel que realizan elecciones (es decir, heurísticas) para limitar la parte del espacio de búsqueda que será visitado, haciendo que los problemas sean manejables. Las metaheurísticas se pueden clasificar principalmente en dos categorías: de solución única y poblacionales. Las metaheurísticas de solución única mantienen, modifican y mejoran paso a paso una única solución candidata, por lo anterior, también son denominadas metaheurísticas basadas en trayectoria. Por otro lado, las metaheurísticas basadas en poblaciones modifican y mejoran una población, es decir, un conjunto de individuos que corresponden a soluciones candidatas.

Generalmente, las metaheurísticas implementan dos estrategias de búsqueda: intensificación y diversificación [1]. La intensificación guía al solucionador a explorar profundamente una parte prometedora del espacio de búsqueda. En contraste, la diversificación apunta a extender la búsqueda a diferentes partes del espacio de búsqueda [2]. Para obtener el mejor rendimiento, una metaheurística debería proporcionar un buen equilibrio entre intensificación y diversificación. Por diseño, algunas heurísticas son mejores para intensificar la búsqueda, mientras que otras lo son para diversificarla.

De manera más general, cada metaheurística puede desempeñarse de manera diferente según el problema o incluso en la instancia donde se resuelve. Una metaheurística también variará según los parámetros de ajuste elegidos. La tendencia actual es diseñar metaheurísticas híbridas, combinando diferentes métodos para beneficiarse de las ventajas individuales de cada uno [3]. Un enfoque eficaz consiste en combinar un algoritmo evolutivo con un método de solución única (muy a menudo un procedimiento de búsqueda local). Estos métodos híbridos se llaman algoritmos meméticos [4]. Las metaheurísticas híbridas tienden a ser procedimientos complejos, difíciles de diseñar, implementar y ajustar, por lo tanto, la mayoría de ellos solo combinan dos métodos.

A pesar de los buenos resultados obtenidos con el uso de metaheurísticas híbridas, aún es necesario reducir los tiempos de procesamiento necesarios para casos más difíciles [5]. Una posible respuesta implica recurrir a la ejecución paralela [6]. Por ejemplo, varias instancias de una metaheurística dada pueden ejecutarse en paralelo para desarrollar exploraciones de manera concurrente del espacio de búsqueda, ya sea de forma independiente o cooperativa mediante la comunicación entre procesos concurrentes. El primero es más fácil de implementar en máquinas paralelas, ya que las metaheurísticas se ejecutan independientes entre sí y la ejecución se detiene tan pronto como cualquiera de ellos encuentra una solución [7][8].

Si bien la tendencia en el desarrollo de metaheurísticas recurre a la utili-

zación de técnicas de hibridación y paralelismo, el proceso de diseño de estos métodos es usualmente complejo. Un problema a la hora de implementar metaheurísticas paralelas e híbridas es la barrera que impone la curva de aprendizaje de las tecnologías usualmente indicadas para este fin. La implementación de estas metaheurísticas requiere de conocimientos en el paradigma de programación paralela, que normalmente vienen de la mano con competencias en lenguajes de bajo nivel como C o C++ y de librerías o API's especializadas para este propósito. Existe entonces una brecha entre los conocimientos de programación de los expertos diseñadores de metaheurísticas (expertos del dominio) y los programadores de aplicaciones paralelas.

Para el problema anterior, existe un concepto de programación que pretende cerrar la brecha entre los expertos en el dominio y los programadores de aplicaciones, los Lenguajes Específicos de Dominio (DSL, por sus siglas en inglés). Un DSL es un lenguaje de programación de computadores con expresividad limitada enfocado a un dominio específico [9]. Un claro ejemplo de DSL es SQL, el cual es un lenguaje para administrar bases de datos relacionales.

Teniendo en cuenta lo expuesto anteriormente, en este trabajo se describe la implementación de un DSL que facilita el proceso de ejecución de metaheurísticas híbridas y paralelas, permitiendo que los expertos en el dominio puedan probar varias configuraciones de los métodos de solución sin requerir un conocimiento basto en técnicas de programación paralela.

3. Objetivos

3.1. Objetivo general

Desarrollar un Lenguaje Específico de Dominio (DSL, por sus siglas en inglés) para la descripción de metaheurísticas híbridas y paralelas.

3.2. Objetivos específicos

- Identificar las principales técnicas para el diseño e implementación de un DSL.
- Diseñar un DSL para la descripción de metaheurísticas híbridas y paralelas.
- Implementar un prototipo para la generación de código a partir de una especificación del dominio.
- Validar el DSL implementado a través un caso de estudio.

4. Marco teórico

Con el fin de entender un poco más los términos usados en este informe, en esta sección se presenta una breve descripción de los principales temas a tratar.

4.1. Metaheurísticas

Según [10] una metaheurística es un marco algorítmico de alto nivel, que es independiente del problema y proporciona un conjunto de pautas para desarrollar algorítmos de optimización heurística. En otras palabras, se podría decir que las metaheurísticas son estrategias (independientes del problema), que guían una heurística en un algoritmo de optimización, permitiendo así incrementar su desempeño a la hora de encontrar una solución viable. Existen muchas metaheurísticas que se han propuesto a lo largo del tiempo por diferentes autores, pero en general hay tres clases en las que se podrían agrupar debido a la manera en que manipulan las soluciones para obtener buenos resultados [10], siendo estas: *Metaherísticas de Búsqueda local, Constructivas y Poblacionales.*

- **Metaheurísticas de Búsqueda Local:** En esta clase se llegan a encontrar buenas soluciones mediante cambios pequeños (también llamados movimientos) realizados de manera iterativa sobre una única solución que es conocida como *solución actual*. También son llamadas *Metaheurísticas de Trayectoria* debido a que cada uno de sus movimientos sucesivos genera una secuencia que se conoce como trayectoria.
- **Metaheurísticas Constructivas:** A diferencia de la anterior éstas no trabajan sobre una solución, construyen una. Inician con el conjunto vacío y se va añadiendo un elemento en cada iteración, normalmente el mejor elemento posible, dando forma a la solución final. Durante este proceso se opera con una solución parcial de la cual no se podría determinar su viabilidad.
- **Metaheurísticas Poblacionales:** En esta clase se logran obtener buenos resultados a partir del intercambio de atributos entre dos o más soluciones de un conjunto, llamado población. Los principales miembros de esta clase son los algoritmos evolutivos, lo cuales imitan los principios de la evolución natural.

4.1.1. Metaheurísticas híbridas y paralelas

Con el paso de los años se ha evidenciado que la combinación de diferentes metaheurísticas puede proporcionar un mejor desempeño, a esta clase se le conoce como *Metaheurísticas Híbridas*. Una manera de mejorar los tiempos de procesamiento es paralelizar la ejecución de las mismas, en donde, diferentes metaheurísticas que trabajan potencialmente con los mismos datos, se les asignan diferentes procesadores dentro de una misma máquina y se ejecutan al mismo tiempo, en ocasiones, permitiendo el intercambio de información [11], aunque

no siempre se utiliza esta característica debido a la alta complejidad que implica desarrollar un mecanismo que permita la comunicación entre estas.

4.1.2. Marco de trabajo para metaherurísticas híbridas y paralelas

El marco de trabajo de búsqueda cooperativa local paralela (CPLS por sus siglas en inglés) fue diseñado para facilitar la implementación de solucionadores de búsqueda local paralelos y cooperativos [12]. CPLS está disponible como una biblioteca de código abierto escrita en el lenguaje de programación concurrente IBM X10. Desde el punto de vista del usuario, solo se necesita codificar la metaheurística deseada, utilizando una interfaz de programación de aplicaciones (API) de CPLS. El API proporciona abstracciones orientadas a objetos para ocultar al usuario todos los detalles de la implementación paralela, permitiendo al usuario escribir un programa puramente secuencial, sin tener que preocuparse por el paralelismo y la comunicación.

CPLS aumenta la estrategia de paralelismo independiente con un mecanismo de comunicación ajustable, que permite la cooperación entre las múltiples instancias de las metaheurísticas. También ofrece varios parámetros que controlan este mecanismo. El usuario simplemente escribe el código con llamadas a funciones de CPLS para ejecutar periódicamente 2 acciones:

- Informar sobre su configuración actual (Acción de informe).
- Recuperar una solución y, si es lo suficientemente buena, el solucionador la adopta, es decir, abandona su solución actual y la reemplaza por la recibida (Acción de actualización).

En tiempo de ejecución, cada instancia de búsqueda local se encapsula en un nodo, también conocido como *worker*. El punto es utilizar todas las unidades de procesamiento disponibles de la máquina asignando cada nodo a un núcleo físico. CPLS organiza a los nodos en equipos (*teams*), donde un equipo consta de n nodos cuyo objetivo es intensificar la búsqueda en una región particular del espacio de búsqueda mediante la comunicación dentro del equipo, n puede oscilar entre 1 y el número máximo de núcleos que posea la máquina.

El parámetro n está directamente relacionado con la relación costo-beneficio entre intensificación y diversificación, ya que se espera que diferentes equipos exploren diferentes regiones del espacio de búsqueda. Cuando n es 1, el marco coincide con la estrategia de paralelismo independiente, se espera que cada equipo de 1 nodo esté trabajando en una región diferente del espacio de búsqueda, sin ningún esfuerzo por buscar una intensificación paralela. Cuando n es igual al número máximo de nodos (creando solo 1 equipo en la ejecución), el marco alcanza el nivel máximo de intensificación (hay que tener en cuenta que una cierta cantidad de diversificación es inherentemente proporcionada por el paralelismo, entre 2 acciones de cooperación, debido a la naturaleza estocástica de las metaheurísticas). Ajustar el valor de n permite al usuario ajustar la relación

costo-beneficio entre intensificación y diversificación.

Por diseño, un equipo busca intensificar la búsqueda en la región más prometedora encontrada por cualquiera de sus integrantes. Los parámetros que guían la intensificación son el intervalo de informe R y el intervalo de actualización U: cada R iteraciones, cada nodo envía su configuración actual y la métrica de costo asociada a su nodo principal. El nodo principal es particular dentro del equipo, puesto que periódicamente recopila y procesa esta información, conservando las mejores configuraciones en el *Elite Pool* (EP). Cada U iteraciones, los nodos recuperan aleatoriamente una configuración de *EP*, un nodo puede adoptar la configuración del *EP*, si es mejor que su propia configuración actual.

4.2. Lenguajes Específicos del Dominio

Un DSL es un lenguaje de programación de computadores con expresividad limitada enfocado a un dominio específico [9], es decir, un lenguaje de programación que cuenta tanto con funciones como con una sintaxis que son cercanas o propiamente extraídas del dominio en el cuál se encuentra enfocado.

Hay cuatro elementos claves para lo expuesto anteriormente y que [9] define como:

- **Lenguaje de programación de computadoras:** El primero y más obvio, es que sea un lenguaje de programación que pueda ser utilizado por humanos e interpretado por una computadora.
- **Naturaleza del lenguaje:** Un DSL es un lenguaje de programación y, como tal, debe tener un sentido de fluidez donde la expresividad provenga no solo de las expresiones individuales, sino también de la forma en que pueden componerse juntas.
- **Expresividad limitada:** Contrario a un lenguaje de programación multipropósito (como Java o Python), un DSL soporta una cantidad limitada de características, las cuales son las necesarias para el dominio, es decir, no es posible hacer un programa completo solo con este, pero si una pequeña parte de uno.
- **Enfoque de dominio:** Limitar el alcance del lenguaje a un dominio es lo que lo hace realemente útil.

4.2.1. Tipos

Dentro de la gran cantidad de DSL que existen, se podrían clasificar en dos tipos: *internos* y *externos*.

- **DSL Internos:** Transforman el lenguaje anfitrión en un DSL en sí, es decir, utilizan el lenguaje anfitrión de la aplicación pero solo se usan pequeños

conjuntos de las carácterísticas propias de este, dando como resultado la sensación de un lenguaje personalizado.

- **DSL Externos:** Son implementados en un lenguaje completamente diferente al lenguaje anfitrión de la aplicación. Comunmente cuentan con una sintaxis personalizada y necesitan de un compilador o intérprete para poder ser procesado.

4.3. Gramáticas

Las gramáticas son un conjunto de normas y reglas que permiten crear los cimientos del lenguaje. Las normas de la gramática se conocen como producciones.

Según [13] una gramática se define formalmente de la siguiente manera:

G $=< N, T, P, S >$, dónde:

- **N:** Es el conjunto finito de símbolos llamados *No Terminales*.
- **T:** Es el conjunto finito símbolos llamados *Terminales*.
- **P:** Conjunto de reglas de producción, de la forma $(N \cup T)^* N (N \cup T)^* \rightarrow (N \cup T)^*$, donde lo que se encuentra antes del símbolo $\rightarrow$ recibe el nombre de *lado izquierdo* de la producción y lo que se encuentra después de dicho símbolo recibe el nombre de *lado derecho* de la producción.

  El símbolo $\rightarrow$ se lee: *lado izquierdo* ***deriva en*** *lado derecho*.
- **S:** Es el símbolo inicial.

Nota: Los conjuntos $\boldsymbol{N}$ y $\boldsymbol{T}$ son disjuntos, es decir, $N \cap T = \emptyset$.

El lenguaje generado por una gramática G, denotado por $L(G)$, es el conjunto de secuenias de símbolos terminales que se pueden derivar a partir de S.

La notación utilizada consta de letras mayúsculas entre los signos de menor que ($<$) y mayor que ($>$) para denotar los símbolos no terminales y letras mayúsculas, minúsulas y demás caracteres necesarios para denotar los símbolos terminales.

5. Análisis

Inicialmente era pertinente verificar si el proyecto que se tenía planeado a desarrollar fuese, efectivamente, un DSL, como bien se especificó previamente en la sección 4.2, un DSL cuenta con cuatro carácterísticas que se verificaron de la siguiente manera:

- **Enfoque de dominio:** El dominio se puede delimitar claramente, siendo este ***la ejecución de metaheurísticas en paralelo***.

- **Lenguaje de programación de computadoras:** Al tener la necesidad de realizar diferentes configuraciones de una ejecución de manera sencilla, las cuales impactaban directamente sobre la máquina, se tenía claro que se requería de un lenguaje de programación que fuera sencillo de modificar y entender para un humano y que a su vez fuera interpretado por una computadora.

- **Naturaleza del lenguaje:** Se contaba una expresividad dada por el mismo dominio, en donde términos como *worker*, *team*, *pool*, entre otros, no solo tenían sentido individualmente, si no que a la hora de ponerlos en conjunto se podía describir fácilmente la actividad esperada.

- **Expresividad limitada:** Al ser una funcionalidad muy concreta, no se necesitaba de muchas características propias de un lenguaje de programación, solo era necesario describir una ejecución bajo los términos propios del dominio abarcado.

En el siguiente paso se definió cual de los dos tipos de DSL se ajustaba más a la propuesta inicial, es decir, si se realizaría una adaptación de un lenguaje de programación multipropósito utilizando solo un conjunto de las características del mismo (DSL Interno) o si por el contrario se crearía un lenguaje completamente aparte (DSL Externo). Para la toma de esta decisión se tuvo en cuenta la entrevista con Danny Múnera, asesor de este trabajo de grado y experto del dominio, en esta entrevista se lograron identificar las siguientes necesidades:

- **Facilidad de uso:** A la hora de utilizar el DSL no deben de ser requeridos conocimientos avanzados en programación, bien sea programación tradicional o paralela. Es decir, que realizar una ejecución fuera lo más transparente posible para un experto del dominio.

- **Expresividad:** Cuando el experto del dominio esté interactuando con el DSL, se requiere que este identifique de una manera clara la configuración actual, permitiendo así realizar cambios rápidamente sobre la actual configuración.

- **Flexibilidad:** El DSL debe de permitir modificar fácilmente los diferentes parámetros y características de la ejecución, logrando así, realizar diversas configuraciones de manera sencilla.

Para los apartados anteriores, según [9], un DSL externo es claramente la mejor opción, dado que el punto que prima es la comunicación con el experto del dominio, la cual se verá entorpecida con los DSL internos debido a que siempre están ligados a la sintaxis del lenguaje anfitrión, el cual presenta ruido y puede convertirse una barrera de comunicación. Una desventaja que presenta el DSL externo frente al interno, es que no contaría con todas las características que brindan los diferentes Entornos de Desarrollo Integrados, tales como, autocompletado, resaltado de palabras clave e identificación de errores sintácticos.

Según lo presentado anteriormente, se decide continuar con el desarrollo de un DSL externo, teniendo en cuenta la importancia que tiene la comunicación con el experto del dominio.

6. Diseño

A grandes rasgos, un DSL se compone de dos partes esenciales, una sintaxis, la cual engloba a todas las expresiones permitidas dentro del programa, y una semántica, que da el significado a lo que está escrito, es decir, dicta el comportamiento del programa al ser ejecutado. En el Código 1 se observa la versión final de la sintaxis construída en conjunto con el experto del dominio.

```
Config {
    caracteristica_1: valor_1
    caracteristica_2: valor_2
    caracteristica_3: valor_3
    ...
    caracteristica_n: valor_n
}

Execution {
  Team <multiplicidad>{
    Worker <multiplicidad> {
        caracteristica_1: valor_1
        ...
        caracteristica_n: valor_n
    }
    Pool {
        caracteristica_1: valor_1
        ...
        caracteristica_n: valor_n
    }
  }
}
```

Código 1: Sintaxis propuesta para el DSL

Para capturar todas las expresiones legales permitidas dentro del lenguaje, se construye la gramática mostrada en el Código 2, cubriendo así, la parte sintáctica del Lenguaje Específico de Dominio.

```
1. <S> -> <CONFIG><EXECUTION>
2. <CONFIG> -> Config { <ASSIGN> }
3. <EXECUTION> -> Execution { <TEAM> }
4. <TEAM> -> Team <MULTIPLICITY> { <WORKER> <POOL> }
5. <TEAM> -> <TEAM><TEAM>
6. <WORKER> -> Worker <MULTIPLICITY> { <ASSIGN> }
7. <WORKER> -> <WORKER><WORKER>
8. <POOL> -> <POOLA>
9. <POOL> -> <POOLA><POOLA>
10. <POOLA> -> Pool { <ASSIGN> }
```

```
11. <ASSIGN> -> <KEY>:<VALUE>
12. <ASSIGN> -> <ASSIGN><ASSIGN>
13. <MULTIPLICITY> -> < <NUMBER> >
14. <MULTIPLICITY> -> ε
15. <KEY> -> <STRING>
16. <VALUE> -> <STRING>
17. <VALUE> -> <NUMBER>
18. <STRING> -> [a-zA-Z0-9_-/."]+
19. <NUMBER> -> [0-9]+
```

Código 2: Gramática obtenida a partir de la sintáxis

Para la parte semántica, es necesario dejar en claro las asociaciones y relaciones que tienen entre sí los elementos del DSL, además de parámetros o funciones que los elementos poseen, para esto se hace la construcción de un modelo semántico mostrado en la Figura 1. Finalmente se describe el comportamiento que se espera tener en cada una de las palabras reservadas del lenguaje, las cuales son:

- **Config:** Es en donde se consignan diferentes carácterísticas que sirvan como parámetros de configuración general para la ejecución, de manera preliminar se logran identificar algunos parámetros, tales como, el tipo de modelo que se va a ejecutar (model), el objetivo que se espera obtener (target), un tiempo máximo de ejecución (maxTime), entre otros. Solo se permite una instancia de *Config* por archivo.

- **Execution:** Es donde se describen las diferentes configuraciones de los *Teams* para una ejecución en concreto. Solo se permite una instancia de *Execution* por archivo. Además, se espera que actúe como orquestadora, realizando la carga de los parámetros descritos en *Config* y lanzando la ejecución descrita en sí misma.

- **Team:** Lo compone por lo menos un *Worker* y a lo más dos *Pools*. Puede contar con multiplicidad, es decir, si se especifica un número `n` entre los símbolos *mayor qué* y *menor qué*, se contará con `n` *Teams* con la misma distribución de *Workers* y *Pools*.

- **Worker:** Es el que lleva a cabo la ejecución del problema. Se identificaron 3 características importantes, que son, el tipo de metaheurística que va a ejecutar (MH) y las referencias a los *pools* que va a utilizar, tanto para solicitar información (requestPool) como para insertar (updatePool). Puede contar con multiplicidad, es decir, si se especifica un número `n` entre los símbolos *mayor qué* y *menor qué*, se contará con `n` *Workers* con las mismas caraterísticas.

- **Pool:** Es dónde se almacenan los mejores resultados de los *Workers* de un *Team*, siendo accedidos por todos los *Workers* del *Team* permitiendo así la mejora de las soluciones que tienen en determinado momento. El acceso a un *Pool* se hará de manera sincronizada, es decir, en ningún momento

del tiempo dos o más *Workers* estarán a la vez obteniendo o insertando nueva información. Se identificaron 3 características importantes, que son, el Id, la política con la que cuenta y el tamaño del *Pool*.

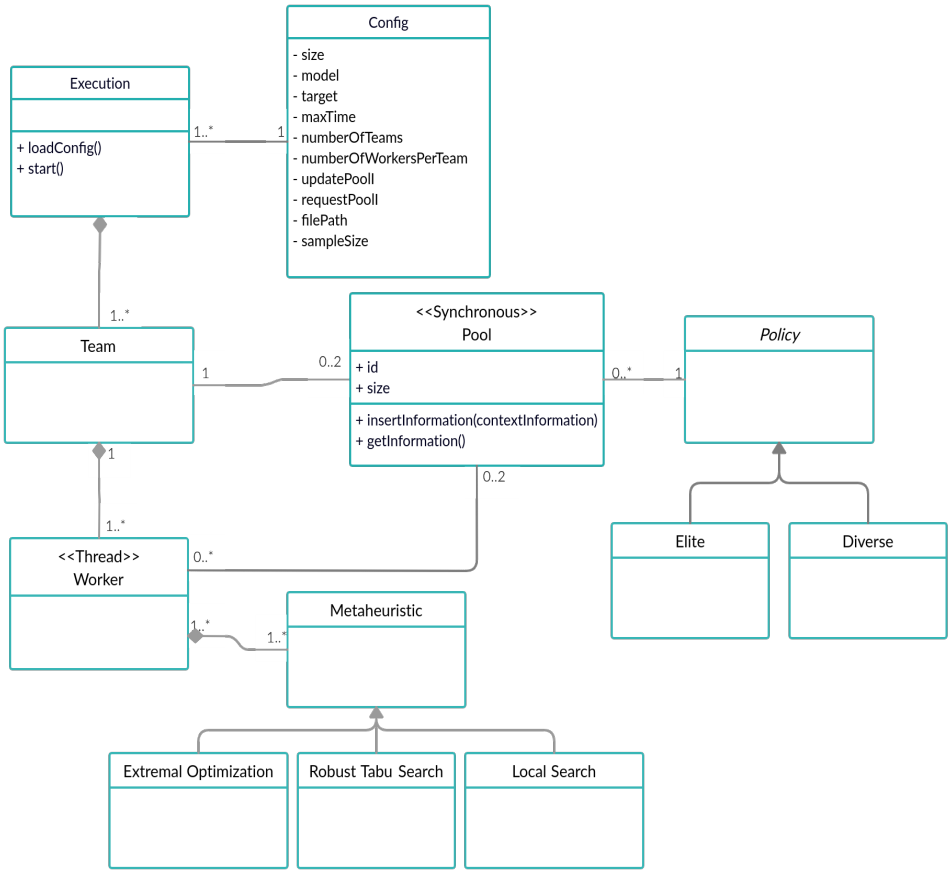

Figura 1: Modelo semántico

7. Implementación

La implementación del DSL se hizo a través de la herramienta ANTLR (ANother Tool for Language Recognition) la cual permite generar analizadores para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. Se usa ampliamente para crear lenguajes, herramientas y marcos de trabajo [14]. A partir de una gramática que se aloja en un archivo de extensión `g4`, ANTLR genera una serie de archivos que en conjunto constituyen un analizador

que puede construir y recorrer árboles de sintaxis abstracta.

El archivo con extención g4 es posible segmentarlo en 3 secciones:

- Encabezado: Es donde se declara el nombre de la gramática y además se consignan algunas de las configuraciones necesarias para la ejecución, tales como las importaciones de las clases a utilizar.
- No Terminales: Es donde se hace la especificación de la gramática de atributos, declarando el comportamiento del lenguaje.
- Terminales: Es donde se hace la declaración de los *tokens* que representan los caracteres o cadenas permitidas dentro del lenguaje.

Dadas las características de la herramienta se construirá un intérprete, el cual ejecutará directamente el código fuente del programa, lo interpretará a través de la gramática consignada en el archivo de extensión g4 y generará los datos de salida según el comportamiento que se haya especificado. A los lenguajes que hacen uso de un intérprete se conocen como lenguajes interpretados, un ejemplo de lenguaje interpretado es Python.

Para la construcción del lenguaje se contará con 3 importantes etapas que son: *Análisis léxico*, *Análisis sintáctico* y *Análisis semántico*. El insumo principal para estas etapas será la gramática construída en la sección 6.

7.1. Análisis Léxico

En esta etapa es donde se hace la declaración e identificación de los *tokens* que representan los caracteres o cadenas permitidas dentro del lenguaje, se relaciona directamente con la sección de declaración de Terminales del archivo *g4* y con los terminales mostrados en el Código 2. En este código se especifican las palabras reservadas del lenguaje, símbolos, constantes y caracteres a omitir tal como los espacios en blanco. Los tokens se representan con un par clave/valor, en donde la clave estará en mayúscula y el valor podrá ser una cadena concreta de caracteres o una expresión regular. En el Código 3 se encuentran los tokens usados en el caso particular del presente DSL.

```
// Palabras reservadas
CONFIG: 'Config';
EXECUTION: 'Execution';
TEAM: 'Team';
WORKER: 'Worker';
POOL: 'Pool';

// Simbolos
OPEN_CURLY_BRACKET: '{';
CLOSE_CURLY_BRACKET: '}';
GREATER_THAN: '>';
LESS_THAN: '<';
```

```
COLON: ':';

// Constantes
NUMBER: [0-9]+;
STRING: [a-zA-Z0-9_\-\/\.\"]+;

// Espacios en blanco
WS :[ \t\r\n]+ -> skip;
```

Código 3: Terminales de la gramática traducida al archivo g4

Es importante el orden al declarar los tokens, comenzando desde los más específicos hasta los más generales. En esta etapa del proceso de interpretación solo se identifican los tokens, más no se verifica el orden de entrada ni se validan reglas sintácticas.

7.2. Análisis Sintáctico

En esta etapa se realiza el proceso de verificación del orden de los tokens para considerar que la entrada se encuentra bien formada sintácticamente. Se hará la traducción de la gramática mostrada en el Código 2 a la sintaxis esperada por el archivo `g4`. Los *terminales* ya fueron expuestos anteriormente en la Sección 7.1, por otro lado, los *no terminales* se representan con un par clave/valor en donde la clave estará en minúscula y el valor será un conjunto de *terminales* y *no terminales*. En el archivo, es posible simplificar la declaración de los no terminales con el uso de expresiones regulares, donde se logran identificar las siguientes:

- $*$ → Indica que pueden haber cero o más repeticiones del token al cual se le aplica.
- $+$ → Indica que debe de haber cómo mínimo una o más repeticiones del token al cual se le aplica.
- ? → Indica que el elemento al que se le aplica puede estar o no, haciéndolo un elemento opcional, para hacer opcional un conjunto de terminales y no terminales es necesario agruparlos entre paréntesis.
- $|$ → Actúa como el operador lógico *OR*, donde coincide con la expresión anterior o posterior al simbolo.

En el Código 4 se encuentran los no terminales específicados bajo la sintaxsis del archivo.

```
start:
    CONFIG OPEN_CURLY_BRACKET
        assign*
    CLOSE_CURLY_BRACKET

    EXECUTION OPEN_CURLY_BRACKET
```

```
            team+
        CLOSE_CURLY_BRACKET

    team:
        TEAM (LESS_THAN NUMBER GREATER_THAN)?
        OPEN_CURLY_BRACKET
            worker+
            pool*
        CLOSE_CURLY_BRACKET

    worker:
        WORKER (LESS_THAN NUMBER GREATER_THAN)?
        OPEN_CURLY_BRACKET
            assign*
        CLOSE_CURLY_BRACKET

    pool:
        POOL OPEN_CURLY_BRACKET
            assign*
        CLOSE_CURLY_BRACKET

    assign:
        STRING COLON (STRING | NUMBER)
```

Código 4: No Terminales de la gramática traducida al archivo g4

De este proceso se obtiene como salida un *árbol de sintaxis concreta*, en la raíz se encuentra el símbolo inicial, los nodos representan a los no terminales y las hojas serían terminales. En la Figura 2 se puede observar el árbol de sintaxis abstracta que se construye a partir del Código 5. La ejecución del programa en este punto no da resultados, debido a que aún no se han asociado funcionalidades a los elementos de la sintaxis.

```
    Config {
        size: 15
        target: 4000
    }
    Execution {
        Team <2> {
            Worker {
            }
        }
    }
```

Código 5: Configuración de ejemplo

7.3. Análisis Semántico

Finalmente es necesario dotar de un comportamiento al DSL, esto es posible de lograrlo con la implementación de una gramática de atributos, la cual relaciona un conjunto de atributos con cada símbolo de una gramática libre de contexto. Cada producción tiene asociados un conjunto de reglas semánticas que

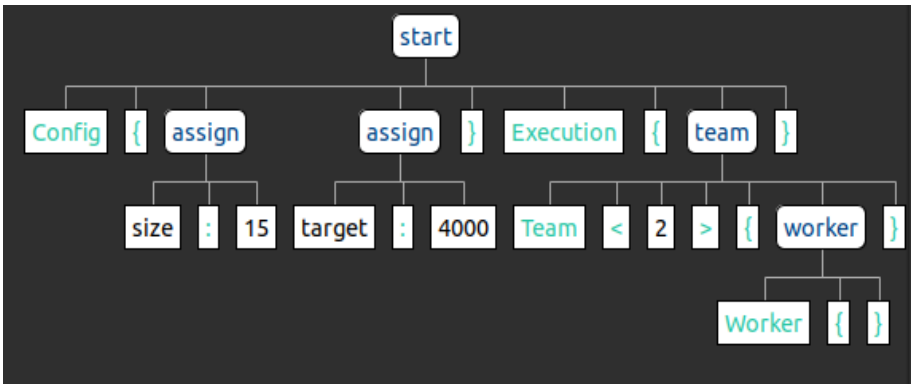

Figura 2: Árbol de Sintaxis Concreta

toman la forma de asignación a atributos [15].

En la implementación se hará usó del *Patrón Intérprete*, en el cual se construirá una clase por cada regla en el lenguaje y contendrá un método para interpretar una expresión [16].

En el archivo de extensión *g4* es posible transformar la gramática mostrada en el Código 4 a una gramática de atributos mediante la inserción de código Java.

En el Código 6 se encuentra el ejemplo con el no terminal *assign*, el cual ya se encuentra modificado con el código Java correspondiente y en el Código 7 se encuentra la clase *Constant* con su respectivo método *execute* que se encarga de ejecutar la expresión tal y como lo indica el patrón *Intérprete*.

```
      assign  returns [ASTNode node]:
        {
          String key;
          Constant value;
        }
         STRING { key = String.valueOf($STRING.text); } COLON
        (
          STRING  { value = new Constant(String.valueOf($STRING.
text)); }
          |
          NUMBER { value = new Constant(Integer.parseInt($NUMBER.
text)); }
        )
        { $node = new Assign(key, value);
      };
```

Código 6: No Terminal modificado con atributos

```
      public class Constant implements ASTNode {

        private Object value;

        public Constant(Object value) {
          super();
          this.value = value;
        }

        @Override
        public Object execute(Map<String, Object> symbolTable)
  throws Exception {
          return value;
        }

      }
```

Código 7: No Terminal modificado con atributos

8. Validación

Luego de realizar la implementación del DSL, se propuso hacer un experimento en donde se solucionarán algunas instancias del problema de asignación cuadrática (QAP por sus siglas en inglés), el cual fue propuesto por primera vez en 1957 por Koopman y Beckmann [17] como un módelo para un problema de *ubicación* de *instalaciones*. Este consiste en asignar un conjunto de n instalaciones a un conjunto de n ubicaciones, minimizando el costo asociado a los flujos de los artículos entre las instalaciones y la distancia entre ellas. Sea F la matríz de flujos, donde f_{ij} es el flujo entre las instalaciones i y j; sea D la matríz de distancias, donde d_{kl} es la *distancia* entre las *ubicaciones* k y l. El objetivo es encontrar una asignación óptima de instalaciones a ubicaciones con un costo total mínimo, definido como la suma de todos los productos entre *flujos* y *distancias* [18].

La ecuación 1 presenta una formulación del QAP donde $\phi(i)$ es la ubicación a la que se asigna instalación i y $d_{\phi(i)\phi(j)}$ es la distancia entra las ubicaciones $\phi(i)$ y $\phi(j)$.

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} * d_{\phi(i)\phi(j)} \tag{1}$$

Por otro lado, el objetivo del experimento es demostrar la flexibilidad del DSL evaluando 3 instancias tomadas de la librería *QAP Lib*, esta web creada en 1991 sirve como repositorio de las instancias del problema [19], para las soluciones óptimas encontradas se cuenta con la permutación óptima mientras que para las soluciones no óptimas se encuentran los límites inferiores. Para realizar la validación fueron seleccionadas las instancias *tai20a, tai30a* y *tai40a*, las

cuales cuentan con diferentes tamaños y mejores soluciones conocidas.

Se plantearon tres tipos diferentes de configuraciones, que serían replicadas en la ejecución de las tres instancias seleccionadas. A continuación se muestran fragmentos de los códigos que serán ejecutados, cabe resaltar que cada configuración cuenta con dos fragmentos importantes, el primero de ellos está denotado por la palabra reservada del lenguaje ***Config*** que contiene los metadatos de cada configuración. El segundo fragmento está denotado por la palabra clave ***Execution*** que describe la forma de que tendrá.

La primera instancia, correspondiente a ***tai20a***, cuenta con los siguientes metadatos:

```
Config {
  size: 20
  target: 703482
  maxTime: 300000
  updatePoolI: 600
  requestPoolI: 300
  numberOfTeams: 4
  numberOfWorkersPerTeam: 16
  filePath: /home/user/test/tai20a.qap
  sampleSize: 30
}
```

Código 8: Metadatos de la configuración 1

La segunda instancia, correspondiente a ***tai30a***, cuenta con los siguientes metadatos:

```
Config {
  size: 30
  target: 1818146
  maxTime: 300000
  updatePoolI: 6000
  requestPoolI: 3000
  numberOfTeams: 3
  filePath: /home/user/test/tai30a.qap
  sampleSize: 30
}
```

Código 9: Metadatos de la configuración 2

Finalmente, la tercera instancia, correspondiente a ***tai40a***, cuenta con los siguientes metadatos:

```
Config {
  size: 40
  target: 3139370
  maxTime: 300000
  updatePoolI: 600000
  requestPoolI: 300000
  numberOfTeams: 2
  filePath: /home/user/test/tai40a.qap
```

```
    sampleSize: 30
}
```

Código 10: Metadatos de la configuración 3

Si bien el DSL cuenta con soporte para múltiples *pools*, para el experimento se contará con uno solo, el cual será el mismo tanto para solicitar datos como para contribuir con nuevos. Esto quiere decir que cada uno de los *workers* que serán mostrados a continuación contarán con las siguientes propiedades:

```
Worker {
    updatePool: 1
    requestPool: 1
}
```

Código 11: Id del Pool

De la misma manera, cada uno de los *teams* tendrán un *pool* con las siguientes características:

```
Pool {
    ID: 1
    policy: Elite
    size: 4
}
```

Código 12: Configuración del Pool

En la primera ejecución de cada instancia, llamada ***Equipos Homogéneos***, se espera tener cuatro *teams* con dieciseis *workers*, los *workers* se dividirán de la siguiente manera: cinco con una metaheurística del tipo **Adaptive**, cinco con una del tipo **Extremal** y seis del tipo **RoT**. Lo descrito anteriormente es posible visualizarlo de manera gráfica en la Figura 3 y también su equivalente en el DSL en el Código 13.

```
Execution {
  Team <4>{
    Worker <5> {
      MH: Adaptive
    }
    Worker <5> {
      MH: Extremal
    }
    Worker <6> {
      MH: RoT
    }
  }
}
```

Código 13: Equipos Homogéneos

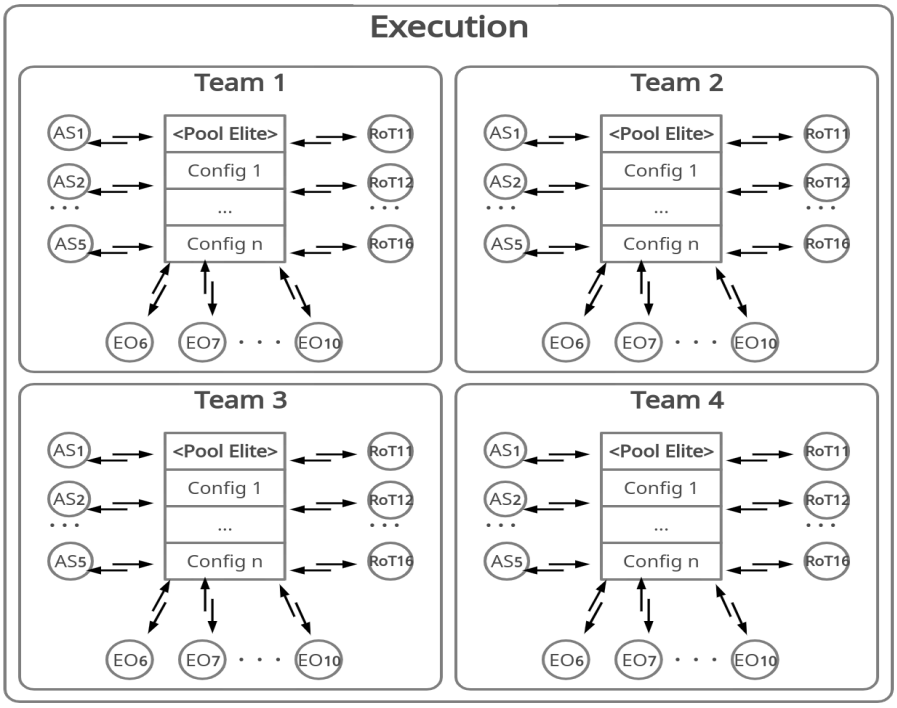

Figura 3: Representación gráfica de los Equipos Homogéneos

En la segunda ejecución, llamada ***Equipos Heterogéneos***, se espera tener tres *teams*, cada uno con veintiún *workers* del mismo tipo de metaheurística. Cada *team* se concentrará en una metaheurística diferente. Lo descrito anteriormente es posible visualizarlo de manera gráfica en la Figura 4 y también su equivalente en el DSL en el Código 14.

```
Execution {
  Team {
    Worker <21> {
      MH: Adaptive
    }
  }
  Team {
    Worker <21> {
      MH: Extremal
    }
  }
  Team {
    Worker <22> {
      MH: RoT
    }
  }
```

```
}
```

Código 14: Equipos Heterogéneos

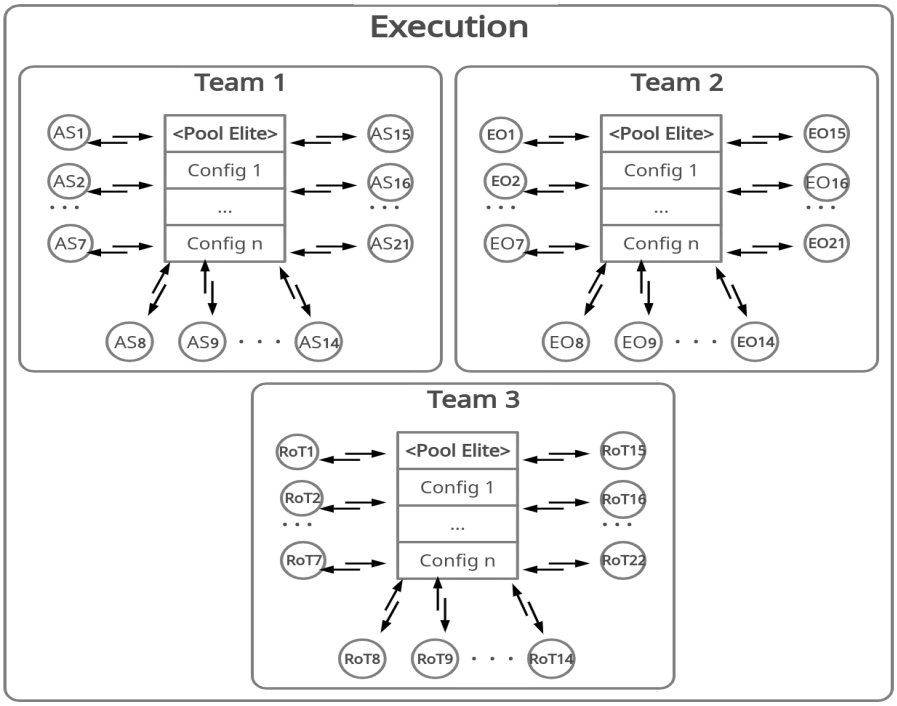

Figura 4: Representación gráfica de los Equipos Heterogéneos

Finalmente en la última ejecución, llamada ***Equipos Heterogéneos de diferente tamaño***, se espera tener tres *teams*, en el primero los *workers* se dividirán de la siguiente manera: dos con una metaheurística del tipo **Adaptive (AS)**, dos con una del tipo **Extremal (EO)** y dos del tipo **RoT**. Los otros dos *teams* serán iguales y los *workers* se dividirán de la siguiente manera: nueve con una metaheurística del tipo **Adaptive (AS)**, diez con una del tipo **Extremal (EO)** y diez del tipo **RoT**. Lo descrito anteriormente es posible visualizarlo de manera gráfica en la Figura 5 y también su equivalente en el DSL en el Código 15.

```
Execution {
  Team {
    Worker <2> {
      MH: Adaptive
    }
    Worker <2> {
```

```
        MH: Extremal
      }
      Worker <2> {
        MH: RoT
      }
    }
    Team <2> {
      Worker <9> {
        MH: Adaptive
      }
      Worker <10> {
        MH: Extremal
      }
      Worker <10> {
        MH: RoT
      }
    }
  }
```

Código 15: Equipos Heterogéneos de diferente tamaño

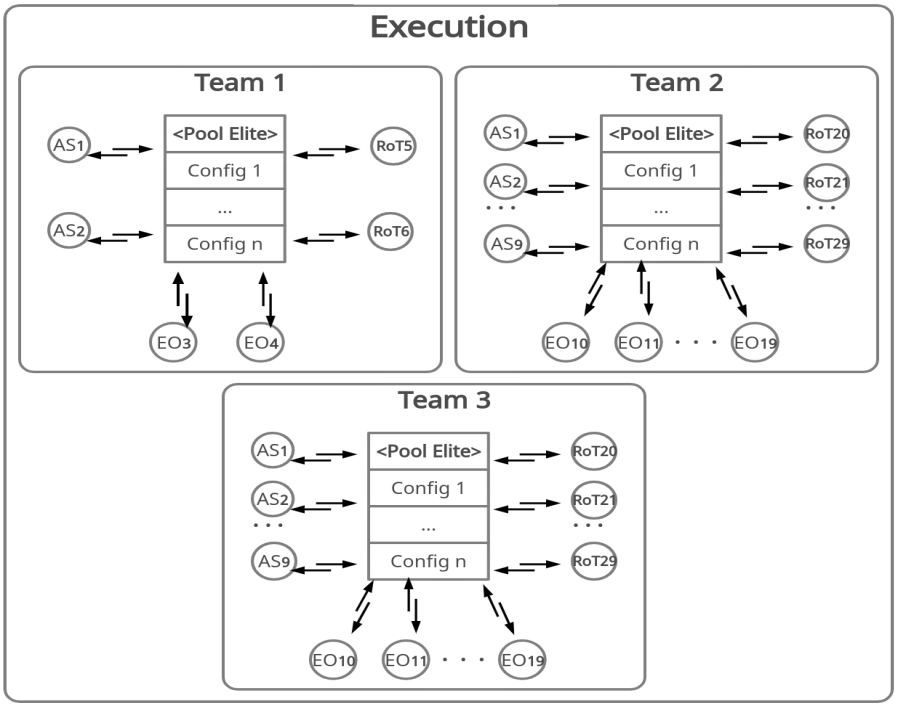

Figura 5: Representación gráfica de los Equipos Heterogéneos de diferente tamaño

En las Figuras 13, 14 y 15, las metaheurísticas **Extremal** y **Adaptive** se muestran como **EO** y **AS** respectivamente.

Las ejecuciones se realizaron en un servidor dotado con 132 GB de memoria y un procesador AMD Opteron(tm) 6380, el cual cuenta con 64 CPU's, 4 sockets y 32 núcleos con capacidad para ejecutar 2 hilos simultáneamente.

Se decidió ejecutar cada experimento trienta veces para tener un nivel de confianza alto de los resultados obtenidos según el *Teorema del Límite Central*, asumiendo que los resultados generados por los experimentos se distribuyen de manera normal. Para automatizar la ejecución de los experimentos se utilizó el script de bash visto en el Código 16 que realizó las trienta ejecuciones de cada configuración y guardó los resultados obtenidos de cada una en un archivo plano con extensión *.res*.

```
#!/bin/bash

for pmetfile in test/*.pmet; do
    java -jar dsl-pmet.jar Main "$pmetfile" >> "$(basename "
$pmetfile" .pmet).res"
done
```

Código 16: Bash Script

En la Tabla 1 se muestran los resultados obtenidos luego de la ejecución de cada configuración, en concreto el tiempo promedio y la desviación porcentual promedio *(APD)*.

	Configuración 1		**Configuración 2**		**Configuración 3**	
	Tiempo	APD	Tiempo	APD	Tiempo	APD
Tai20a	0.1254	0.0	0.0874	0.0	0.1079	0.0
Tai30a	0.3199	0.0	0.3005	0.0	0.3866	0.0
Tai40a	203.63	0.04	276.54	0.06	274.17	0.06

Tabla 1: Resultados

En todas las configuraciones de las instancias *Tai20a* y *Tai30a* se obtiene un APD de cero (0.0), lo cual significa que en cada una de las ejecuciones se alcanzó el mismo resultado, que a su vez coincidía con el resultado esperado. Si bien en las configuraciones de la instancia *Tai40a* no se obtuvo cero (0.0) en el APD, se llegó a resultados cercanos no mayores a cero punto uno (0.1), lo cual se traduce a que el margen de error entre los resultados esperados y los obtenidos es pequeño. Además, es pertinente anotar que el tiempo promedio aumenta con el tamaño de la instancia tal y como era de esperarse.

Teniendo en cuenta que la tres configuraciones fueron igual de efectivas en las instancias *Tai20a* y *Tai30a*, la Configuración #2 resultó ser la más rápida en

ambos casos, mientras que para la instancia *Tai40a* la Configuración #1 resultó ser la más rápida y a la vez más efectiva al obtener el menor APD (0.04).

9. Conclusiones

El Lenguaje Específico de Dominio propuesto logra disminuir la brecha que poseen los expertos del dominio en los conocimientos de programación paralela, permitiendo a los usuarios ejecutar metaheurísticas de manera paralela con solo describir la forma de la ejecución esperada, llevando fácilmente estas descripciones a archivos *.pmet*, los cuales no necesitan modificar o intervenir directamente con código que implemente el paradigma de programación paralela.

Referencias

[1] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys 35(3)*, pp. 268–308, 2003.

[2] H. Hoos and T. St utzle, *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.

[3] A. Misevicius, "A tabu search algorithm for the quadratic assignment problem," *Computational Optimization and Applications*, vol. 30, pp. 95–111, Jan 2005.

[4] P. Moscato, C. Cotta, and A. Mendes, "Memetic algorithms," *New Optimization Techniques in Engineering. Studies in Fuzziness and Soft Computing, vol 141*, pp. 53–85, 2004.

[5] M. Saifullah Hussin, "Stochastic local search algorithms for single and biobjective quadratic assignment problems," *UNIVERSITE LIBRE DE BRUXELLES*, 2016.

[6] T. G. Crainic and M. Toulouse, *Parallel Meta-heuristics*, pp. 497–541. 2010.

[7] C. Novoa, A. Qasem, and A. Chaparala, "A simd tabu search implementation for solving the quadratic assignment problem with gpu acceleration," *Conference on Scientific Advancements Enabled by Enhanced Cyberinfrastructure - XSEDE*, pp. 1–8, 2005.

[8] S. Tsutsui and N. Fujimoto, *An Analytical Study of Parallel GA with Independent Runs on GPUs, vol. 8*. 2013.

[9] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[10] K. Trabelsi, M. Sevaux, P. Coussy, A. Rossi, and K. Sörensen, *Metaheuristics*. 2010.

[11] C. Blum and A. Roli, "Hybrid metaheuristics: An introduction," *Hybrid Metaheuristics. Studies in Computational Intelligence, vol 114*, 2008.

[12] J. Lopez, D. Munera, D. Diaz, and S. Abreu, "On integrating population-based metaheuristics with cooperative parallelism," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 601–608, 2018.

[13] R. Gómez Marín and A. Sicard Ramírez, *Informática teórica: Elementos propedéuticos.* Fondo Editorial de la Universidad EAFIT, 2009.

[14] T. Parr, "Antlr - another tool for language recognition." Recuperado de https://www.antlr.org/.

[15] M. Arroyo, "Gramáticas de atributos, clasificación y aportes en técnicas de evaluación," *EUniversidad Nacional del Sur*, Dic 2008.

[16] V. Sarcar, *Interpreter Pattern*, pp. 437–462. Berkeley, CA: Apress, 2020.

[17] T. C. Koopmans and M. Beckmann, "Assignment Problems and the Location of Economic Activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957.

[18] L. Kaufman and F. Broeckx, "An algorithm for the quadratic assignment problem using Bender's decomposition," *European Journal of Operational Research*, vol. 2, pp. 207–211, May 1978.

[19] R. Burkard, S. Karisch, and F. Rendl, "Qaplib – a quadratic assignment problem library." Recuperado de https://coral.ise.lehigh.edu/data-sets/qaplib/qaplib-problem-instances-and-solutions/#Ta.