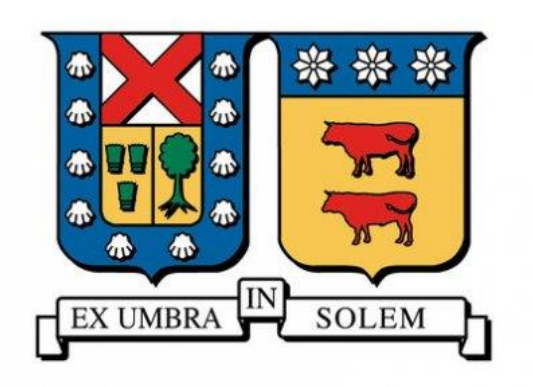

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INDUSTRIAS

DESARROLLO DE ALGORITMO BRANCH & BOUND EN PARALELO PARA LA RESOLUCIÓN DE PROBLEMAS DE OPTIMIZACIÓN COMBINATORIAL

MEMORIA PARA OPTAR AL TÍTULO DE

INGENIERO CIVIL INDUSTRIAL

AUTOR

PABLO IGNACIO LARREA EGAÑA

PROFESOR GUÍA: **RODRIGO MENA**

PROFESOR CORREFERENTE: **PABLO VIVEROS**

SANTIAGO, 28 DE AGOSTO, 2020.

AGRADECIMIENTOS

Quiero agradecer a mi familia, por todas las oportunidades que me han dado desde siempre, por el cariño, el consejo y el apoyo incondicional que me han permitido andar con confianza y seguridad en las decisiones que he tomado a lo largo de mi vida.

Muchas gracias a perroperro, por acompañarme por tantos años, esperando que ahora siga conmigo esté donde esté.

Gracias a la gente de biblioteca, las tías, la jefa, por brindarme un lugar tan cálido para estar en la universidad. Por apoyarme y siempre tener disposición para escuchar.

Gracias a los amigops por el apoyo desde tiempos inmemoriables, por la confianza y los años de compañía. Son amistades que tengo fe en que durarán por siempre, aunque cada uno tome su propio camino. Igualmente, gracias a todas esas amistades formadas desde la época del colegio, con quienes agradezco de corazón el aún mantener una relación cercana, soportando el paso del tiempo.

Gracias a mis amigos y amigas de la universidad, son en gran parte la fuente de la confianza que tengo en mi mismo hoy en día. Me gustaría agradecerles por separado, pero saben que probablemente terminaría escribiendo un libro aparte! Agradezco el eterno cariño y la disposición.

Por último, agradecer a mi profesor guía, especialmente por la paciencia, porque el proceso fue lento y complejo, especialmente en un inicio, y no siempre me comuniqué de la mejor manera. Gracias por las oportunidades, la simpatía y el interés.

RESUMEN EJECUTIVO

En el presente trabajo de memoria se presenta el estudio, desarrollo y evaluación de la paralelización de procesos en la ejecución de un algoritmo Branch and Bound para la resolución eficiente de problemas combinatoriales.

Con lo anterior, el objetivo es estudiar el comportamiento que adquiere el algoritmo de resolución de problemas de optimización de variable entera y/o binaria al adoptar esta metodología. ¿Mejoran los tiempos de ejecución del algoritmo Branch and Bound paralelizado frente al serial? Y, de ser así, ¿con qué técnicas de exploración se aprovechan de mejor manera los beneficios obtenidos por esto? ¿Vale la pena paralelizar?

El algoritmo Branch and Bound paralelo fue programado utilizado Python 3.5.6, con su respectivo modo *multiprocessing*, apoyado con el lenguaje de modelamiento Pyomo 5.6.9. Pyomo entrega las herramientas para definir los componentes de un problema de optimización, siendo luego manipulados para armar el árbol de subproblemas y entregar las soluciones correspondientes. La resolución de los nodos del árbol se realizó mediante el uso del solver Gurobi 8.1.1, con apoyo de una licencia académica, haciendo uso de un equipo con dos procesadores Intel Xeon Silver 4114 CPU @2.20 Ghz y 2.19 Ghz, con un sistema operativo Windows 10 Pro for Workstations y 64 GB RAM.

Se determinó un beneficio importante en los tiempos de resolución de los problemas, con aceleraciones siempre positivas que dan a conocer la eficiencia de la paralelización. El cambio más notorio se da al pasar desde una resolución serial a una paralelizada con dos procesos. Desde dicho punto, la eficiencia de la paralelización decrece potencialmente, aunque siempre se mantiene positiva. La estrategia centrada en exploración en profundidad (DFS) es quién recibe más cambios en sus velocidades de

resolución por parte de esto. Ahora bien, también se determinó que la estrategia centrada en el mejor valor (BFS) es aquella que provee los mejores resultados en cuanto a tiempos de ejecución. Aunque no se descarta que pudiese ser opacada por la DFS en ambientes de gran cantidad de procesos paralelos, aún mayores a los utilizados en el presente estudio. Se plantea además la existencia de un máximo de aceleración que se puede alcanzar al hacer uso de la paralelización, tras lo cual, al aumentar la cantidad de procesos utilizados, se consiguen peores tiempos de ejecución.

Por último, se sugiere realizar a futuro un estudio en profundidad de técnicas de almacenamiento de información para los nodos. Debido a la cantidad de data que se debe almacenar en la memoria física (RAM) del equipo computacional, instancias de gran tamaño tienden a desestabilizar el funcionamiento correcto del computador, por lo que su manejo correcto es clave.

Palabras clave: Optimización, Python, Paralelización, Branch and Bound, Multiprocesamiento.

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. PROBLEMA DE INVESTIGACIÓN

Dentro de los métodos clásicos para la resolución de problemas de optimización entera, el algoritmo Branch and Bound destaca por su relativa eficiencia en encontrar una solución óptima, permitiendo trabajar el problema como si fuese lineal y continuo mediante la relajación de sus variables (removiendo las restricciones de integralidad) (Lee & Mitchell, 2009). Este nuevo problema constituye la raíz de un "árbol de búsqueda", el cual crece creando nuevos subproblemas, donde las variables de decisión de naturaleza originalmente entera –ahora continua– son acotadas en pos de alcanzar valores enteros mediante un proceso de ramificación y acotamiento tanto superior como inferior (de ahí el nombre branch and bound). El cumplimiento de la integralidad de las variables mediante las nuevas cotas da origen a valores óptimos locales de los subproblemas (denominadas como soluciones factibles), estableciendo el "valor incumbente" como la mejor de éstas. Un beneficio del algoritmo Branch and Bound se basa en el hecho de que tanto subproblemas infactibles como subproblemas con soluciones peores a la de la incumbente pueden ser descartados, "podando" el árbol de exploración y acelerando la búsqueda del óptimo global (Kianfar, 2011).

Considerando lo anterior, es posible notar que los factores principales que definen la eficiencia del algoritmo se basan en el orden en que se exploran los subproblemas (Archibald, Maier, McCreesh, Stewart, & Trinder, 2018; Paulavičius, Žilinskas, & Grothey, 2010), pudiendo diferenciar múltiples estrategias para la ramificación y selección de nodos (González, 2010).

La dependencia de la eficiencia de resolución respecto al orden en que se abordan (generan y resuelven) los subproblemas supone un inconveniente importante: su naturaleza compleja y no-determinística a nivel computacional acarrea como

consecuencia que los tiempos de resolución de problemas puedan crecer rápidamente a medida de que aumentan los requerimientos, lo cual aumenta los tiempos de resolución en forma exponencial (He, Tian, & Guo, 2018).

Bajo esta complicación surge la idea de la implementación de Branch and Bound "en paralelo", haciendo referencia a la posibilidad de dividir la carga computacional entre varios procesadores, efectivamente reduciendo los tiempos de resolución de modelos al asignar distintos subproblemas a cada uno (Herrera, Salmerón, Hendrix, Asenjo, & Casado, 2017), resolviéndolos literalmente "en paralelo" pero compartiendo el valor de la incumbente (Smirnov & Voloshinov, 2017). Esta paralelización puede darse tanto a nivel de subproblemas (o nodos), al ser resueltos por múltiples procesos al mismo tiempo (generalmente para subproblemas de gran tamaño y complejidad), o a nivel de árbol, con un enfoque en la exploración acelerada de éste (Crainic, Le Cun, & Roucairol, 2006).

Esta aceleración no necesariamente plantea una relación lineal entre la cantidad de procesadores utilizados y la velocidad de resolución de los problemas de optimización, pudiendo encontrarse anomalías particulares de cada problema que se pueden traducir en aceleraciones inferiores o superiores a la linealidad (algunos casos particulares pueden incluso presentar desaceleraciones) (Gendron & Crainic, 1994).

Crainic, Le Cun, & Roucairol (2006) plantean un ejemplo referido a la exploración de un problema combinatorial altamente complejo con un árbol de 11.892.208.412 nodos, para lo cual se utilizaron en promedio 700 máquinas conectadas en paralelo para su resolución:

> Estas máquinas se distribuyeron en dos laboratorios nacionales (Argonne, NCSA), cinco universidades Americanas (Wisconsin, Georgia tech, New

> Mexico, Colombia, Northwestern), y se conectó a la red italiana INFN. ¡El tiempo empleado fue de ~1 semana! Pero, el tiempo secuencial equivalente en un HPC3000, por ejemplo, ¡era de un estimado de 218.823.577s, o 7 años! (p. 2)

Como se puede notar, el uso de computación paralela resulta indispensable para resolver problemas de optimización en instancias donde la resolución secuencial no resulta rentable o no puede ser alcanzable en un periodo acotado.

A nivel más cotidiano, un aumento en las velocidades de resolución de problemas de optimización podría encontrar su utilidad en contextos reales como la asignación efectiva de trabajos en un horario establecido, la optimización de rutas de transporte o el manejo eficiente de inventarios. Aquí puede resultar indispensable la resolución diaria de problemas de optimización, por lo que la paralelización y su consiguiente aceleración toman un papel crucial.

Haciendo uso de la herramienta Pyomo desarrollada para Python, se plantea el desarrollo e implementación de un algoritmo de Branch and Bound paralelo utilizando el solver Gurobi para la resolución de los subproblemas de optimización, debido a su desempeño frente a competidores (Mittelmann, Benchmarking Optimization Software - a (Hi)Story, 2020), especialmente en cuanto a la resolución de problemas lineales mediante el método Simplex (Mittelmann, Latest Benchmarks of Optimization Software, 2017), además de contar con facilidades para trabajar con multiprocesamiento paralelo (Anand, Aggarwal, & Chahar, 2017). Mediante el algoritmo se pretende analizar los ahorros en los tiempos computacionales en la resolución de problemas combinatoriales, proponiendo además estrategias de exploración adecuadas para lograr esto (tanto de selección y generación de subproblemas como de elección de variables de decisión).

Bajo el contexto de la optimización de problemas combinatoriales usando Branch and Bound, se plantean las siguientes interrogantes:

- ¿Qué tan eficiente es la resolución en paralelo por sobre la resolución convencional no paralelizada?
- ¿Qué impacto podría generar el introducir distintas estrategias de exploración sobre el algoritmo?

2. OBJETIVOS

2.1 Objetivo General

Desarrollar una herramienta computacional de paralelización de algoritmos Branch and Bound mediante el lenguaje de programación Python y el lenguaje de modelamiento Pyomo para analizar el comportamiento de los problemas combinatoriales al ser resueltos en paralelo.

2.2 Objetivos Específicos

- Desarrollar un algoritmo computacional centrado en paralelizar Branch and Bound para problemas de optimización.
- Definir y programar estrategias de exploración de nodos y selección de variables para el algoritmo paralelizado.
- Definir problemas combinatoriales a nivel de benchmarking para estudiar la eficiencia de su resolución mediante el algoritmo.
- Medir eficiencia de la paralelización a través de la comparación del algoritmo serial con su contraparte paralela en distintos calibres para evaluar la existencia de mejoras en las velocidades de búsqueda y determinación de óptimos.
- Analizar impacto de las estrategias de exploración y selección de variable en un algoritmo paralelo sobre sus tiempos de resolución.

3. MARCO TEÓRICO

3.1. Problemas de optimización combinatorial

Un problema de optimización entera es aquel en que algunas o todas las variables de decisión que lo componen deben tomar valores discretos. Cuando éstas además permiten determinar conjuntos óptimos de objetos y costos para resolver un problema mediante el uso de valores binarios, se dice que se está trabajando en un problema de optimización combinatorial (COP) (Hoffman & Ralphs, 2013).

3.1.1 Los problemas más frecuentes de optimización combinatorial

La literatura se basa constantemente en una serie de problemas tipo para la composición de diversos modelos de optimización. Estos problemas han sido estudiados continuamente en el ámbito de la optimización combinatorial y han permitido la creación y el desarrollo de algoritmos de aproximación y/o de resolución especializados.

Entre aquellos que destacan debido a su aparición y revisión constante en los estudios, se encuentran:

- **Problema de la mochila (Knapsack Problem)**: Denominado también en su forma combinatoria como el "Problema de la mochila 0-1", es reconocido como uno de los problemas clásicos de la optimización combinatorial, consistente en optimizar el valor de los objetos en una mochila bajo una restricción de peso máximo (Kellerer, Knapsack, 2016).

El problema puede ser formulado formalmente como:

$$\begin{aligned} max \quad & \sum_{i=1}^{n} x_i p_i \\ s.a \quad & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,1\}, \qquad i = 1, \dots, n \end{aligned} \tag{1}$$

Siendo p_i el valor de cada objeto i y w_i su peso. W es el peso máximo soportado por la mochila y x_i es la variable de decisión. Si $x_i = 1$, entonces el objeto i va en la mochila.

Se distingue además como un problema diverso en aplicaciones, tanto a nivel teórico como práctico:

> El interés teórico aparece principalmente por su estructura simple la cual, por una mano, permite la explotación de un número de propiedades combinatoriales y, por la otra, permite resolver problemas de optimización más complejos a través de subproblemas del tipo mochila. Desde un punto de vista práctico, esos problemas pueden modelar muchas situaciones industriales: presupuestos de capital, sistemas de carga y descarga y corte de stock, por mencionar las aplicaciones más clásicas (Martello & Toth, 1990).

Esto ha llevado al desarrollo de variaciones del problema original, destacando:

- **Problema de la mochila fraccionada (Fractional Knapsack Problem)**: Similar al anterior, con la diferencia de que la variable de

decisión puede tomar valores reales entre 0 y 1, es decir, se define que $x_i \in [0,1]$, pudiendo llevar fracciones de objetos en la mochila. (Chandra Jaiswal, Singh, Maurya, & Kumar, 2011). Cabe decir que, al no verse restringido por restricciones de integralidad, el problema pasaría a ser continuo.

- **Problema de la mochila unificada (Unified Knapsack Problem)**: Consistente en la unificación de los problemas de la mochila 0-1 y mochila fraccionada, bajo el concepto de "Divisibilidad" (Divisibility) define que algunos objetos pueden ser fraccionados, mientras que otros no (Chandra Jaiswal, Singh, Maurya, & Kumar, 2011). Si todos los objetos son divisibles, el problema pasa a ser fraccional. Por el contrario, si ningún objeto posee esta propiedad, el problema pasa a ser 0-1.

- **Problema multidimensional de la mochila (Multidimensional or Multiple Knapsack Problem)**: Similar al problema 0-1, con la diferencia de que ahora se cuenta con m mochilas para cargar.

  Formalmente:

$$\begin{aligned} max \quad & \sum_{i=1}^{n} x_i p_i \\ s.a \quad & \sum_{i=1}^{n} x_i w_{ij} \leq W_j \quad j = 1, \dots, m \\ & x_i \in \{0,1\}, \qquad i = 1, \dots, n \end{aligned} \tag{2}$$

  Es importante notar que, para el caso del problema multidimensional, cada mochila posee restricciones propias: soportan

una carga máxima distinta y el peso de cada objeto varía dependiendo de la mochila. Debido a que ahora existen m restricciones referentes a las mochilas, el problema también es conocido como *m-dimensional* (Beasley, 2008).

- **Problema del vendedor viajero (Traveling Salesman Problem o TSP)**: Definido en la *Encylocpedia of GIS* como la búsqueda de la ruta más corta que un vendedor puede utilizar para pasar una única vez por cada "ciudad" que quiere visitar, regresando luego a su lugar de origen (Wilfahrt & Kim, 2008).

  Formalmente, de acuerdo con la formulación descrita por Marinakis (2009):

$$
\begin{aligned}
min \quad & \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} & & \\
s.a \quad & \sum_{j \in V} x_{ij} = 1 & & i \in V \\
& \sum_{i \in V} x_{ij} = 1 & & j \in V \\
& \sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 & & \forall S \subset V, \quad S \neq \emptyset \\
& x_i \in \{0,1\}, & & i, j \in V
\end{aligned}
\tag{3}
$$

  Siendo c_{ij} la distancia o "costo" entre una ciudad i y una ciudad j del conjunto de todas las ciudades, V. Por otro lado, x_{ij} corresponde a la variable de decisión, la cual indica si se utiliza o no la ruta de i a j. S es un subconjunto de ciudades, utilizado para corroborar que todas las ciudades de la solución se encuentren unidas en un único viaje (eliminando los *subtours*) (Marinakis, 2009). El TSP, como se le denomina comúnmente, suele representarse mediante

un grafo que muestra las ciudades que se encuentran conectadas por un posible camino, junto con los valores correspondientes de c_{ij}. El problema se denomina como *simétrico* si $c_{ij} = c_{ji}$. De lo contrario, pasa a ser *asimétrico* (Punnen, 2017).

- **Problema del ruteo de vehículos (Vehicle Routing Problem o VRP)**: Originalmente planteado como una generalización del TSP, el VRP busca optimizar un conjunto de rutas de entrega en pos de un costo mínimo.

Formalmente, Laporte (1992) define su formulación clásica como:

$$
\begin{aligned}
min \quad & \sum_{i \neq j} c_{ij} x_{ij} & & \\
s.a \quad & \sum_{j=1}^{n} x_{ij} = 1 & & (i = 1, \dots, n) \\
& \sum_{i=1}^{n} x_{ij} = 1 & & (j = 1, \dots, n) \\
& \sum_{i,j \in S} x_{ij} \leq |S| - \upsilon(S) & & (S \subset V\{1\}: |S| \geq 2) \\
& x_{ij} \in \{0,1\}, & & (i, j = 1, \dots, n; i \neq j)
\end{aligned}
\tag{4}
$$

Para el conjunto $V = \{1, \dots, n\}$ de clientes, c_{ij} es la distancia (o costo de viaje) entre cada par de éstos, x_{ij} la variable de decisión que determina si se utiliza o no el camino (o arco) de i a j. S es un subconjunto de clientes, utilizado para asegurar que no existan *subtours* en la solución. De forma similar al TSP, si $c_{ij} = c_{ji}$, el problema es *simétrico* mientras que, de lo contrario, es *asimétrico*.

El problema clásico de VRP supone un conocimiento previo de toda la información referida a los clientes (o puntos de entrega), bajo lo cual el

problema se denomina como estático. Éste pasa a ser dinámico si existe información que solo se revela a medida que pasa el tiempo (por ejemplo, si los pedidos de los clientes ocurren mientras se lleva a cabo una ruta, como ocurriría con los servicios de emergencia) (Potvin, 2008).

Debido a su utilidad en la planificación de sistemas de distribución y de uso de vehículos, el problema se ha diversificado enormemente, incluyendo nuevas restricciones en búsqueda de poder adaptar de mejor manera el problema a un contexto real, destacando las variaciones de VRP con Ventanas de Tiempo, VRP con *Pick-Up* y *Delivery* simultáneo y VRP Capacitado como aquellas más importantes en la literatura (Liong, Wan, & Omar, 2008).

- **Problema de la asignación de trabajo (Job-shop Scheduling Problem o JSP)**: Definido como uno de los problemas de asignación de recursos más estudiado debido a las posibles aplicaciones de su algoritmo en problemas del mundo real, el JSP busca asignar trabajos entre distintas máquinas, buscando minimizar el tiempo requerido para completar el trabajo de mayor duración. Cada trabajo se compone de diversas operaciones que deben ser asignadas en máquinas específicas, por lo que la calidad de la solución del algoritmo se basa en la asignación eficiente de horarios de uso para éstas (Wang, Tsai, & Chiang, 2018).

Šeda (2007) define matemáticamente el problema base como:

$$\begin{array}{lll} min & \max\limits_{i \in O} (t_i + p_i) & \\ s.a & t_i \geq 0 & \forall i \in O \\ & t_i + p_i \leq t_j & \forall i, j \in O,\ i \rightarrow j \\ & (t_i + p_i \leq t_i) \vee (t_i + p_i \leq t_i) & \forall i, j \in O, i \neq j, M_i = M_j \end{array} \quad (5)$$

Siendo t_i el tiempo de inicio de la operación i, y p_i su tiempo de procesamiento. M es el conjunto de máquinas a utilizar, J es el conjunto de trabajos (no escrito de forma explícita en la formulación matemática) y O es el conjunto de operaciones, donde se presentan relaciones de precedencia entre operaciones del mismo trabajo ($i \rightarrow j \Rightarrow J_i = J_j$). Es interesante notar que el problema se basa, entonces, en optimizar dichas relaciones de precedencia al establecer un orden de procesamiento para cada máquina.

3.1.2 Problemas diseñados versus problemas del mundo real

Problemas de optimización como el TSP o el problema de la mochila poseen una formulación precisa. Por lo mismo, Przybylek, Wierzbicki, & Michalewicz (2018) los definen como "problemas diseñados", ajenos a las condiciones complejas de la vida real. Los autores plantean que, por otro lado, los problemas de optimización del mundo real "no han sido diseñados por nadie, pero ocurren en procesos reales de negocio. Suelen tener formulaciones complejas. Para resolver estos problemas, primero tenemos que construir sus modelos, y la calidad de la solución obtenida dependerá de la calidad del modelo." (p. 1)

El Problema del Ladrón Viajero (Traveling Thief Problem o TTP) (Bonyadi, Michalewicz, & Barone, 2013) es un ejemplo del traspaso de problemas diseñados al mundo real, interconectando los problemas de la mochila y el TSP, al inspirarse en la optimización de un tren de carga minera, el cual debe pasar por múltiples minas, cargando y descargando de forma eficiente en cada punto. La mayor complejidad radica en la interconexión entre ambos problemas, pues la optimización de la carga puede requerir de la sub-optimización de la ruta y viceversa. La resolución de cada componente por separado no necesariamente lleva al óptimo del problema general

(Przybylek, Wierzbicki, & Michalewicz, 2018). El problema se puede complicar aún más al considerar horarios de trabajo y períodos de mantención de trenes. Por esta razón se plantea que la complejidad de un problema del mundo real no depende únicamente de su tamaño (en cuanto a variables de decisión y restricciones asociadas), sino que ahora entran en consideración todos los sub-problemas y sus interacciones (tanto de combinación como de interdependencia) (Bonyadi, Michalewicz, & Barone, 2013).

Bonyadi & Michalewicz (2016) mencionan otro ejemplo de problema de optimización del mundo real centrado en cadenas de suministro: el transporte de tanques de agua (Solk, Mann, Mohais, & Michalewicz, 2013), donde se consideran la cantidad de clientes, sus ubicaciones, características del pedido (en cuanto a tamaños) y tiempos de entrega. Adicionalmente, se tiene en consideración las estaciones de carga y descarga de los contenedores de agua y la cantidad y tipo de vehículos disponibles para cada entrega. Como es posible notar, este problema cuenta con múltiples sub-problemas (ruteo de entregas, selección de estaciones, agrupamiento de contenedores, etc), que deben ser resueltos teniendo en consideración las interacciones entre sí, abarcando variaciones de VRP y problemas de la mochila, entre otros.

3.1.3 Problemas combinatoriales desde la complejidad computacional

En el mundo de la ciencia computacional, algunos de los problemas combinatoriales se relacionan con la clase de complejidad NP-Hard (Woeginger, 2003). La clase de complejidad NP representa al conjunto de los problemas de decisión para los cuales las soluciones de cada problema pueden ser comprobadas en un tiempo polinómico[1], pero que no necesariamente poseen un algoritmo que permita resolverlos en un tiempo polinómico (Korte & Vygen, NP-Completeness, 2008). Los problemas pertenecientes a

[1] Es decir, los tiempos de resolución aumentan de manera polinómica, y no exponencial, al aumentar el tamaño de la entrada.

la clase NP-Hard son entonces, a lo menos tan difíciles de resolver como los problemas de la clase NP.

Los problemas de la mochila y el vendedor viajero, por ejemplo, se presentan como NP-Hard (Korte & Vygen, The Knapsack Problem, 2008; Wilfahrt & Kim, 2008), por lo que frecuentemente se opta por buscar soluciones aproximadas y no exactas al resolverlos, pues el crecimiento impredecible de los tiempos de resolución de los algoritmos puede enlentecer enormemente el encontrar soluciones, especialmente ante instancias con gran cantidad de variables (Korte & Vygen, NP-Completeness, 2008).

Para calcular la complejidad de un algoritmo, se suele utilizar la notación asintótica $\mathcal{O}(g(n))$, la cual acota de forma superior al tiempo de resolución de éste (Tian, Guo, & He, 2018). Formalmente:

$$\mathcal{O}\big(g(n)\big) = \{f(n)\colon\ \exists c, n_0\ /\ 0 \leq f(n) \leq cg(n), n \geq n_0\} \qquad (6)$$

Siendo n la cantidad de variables, c y n_0 constantes. El problema del vendedor viajero, por ejemplo, posee una complejidad computacional de $\mathcal{O}(n^2 2^n)$ (Tian, Guo, & He, 2018).

Algunos problemas, sin embargo, poseen un algoritmo alternativo, de tiempo pseudopolinomial, lo que permite establecer cotas polinomiales en los tiempos de resolución. El problema de la mochila 0-1, por ejemplo, puede ser replanteado como un algoritmo recursivo de complejidad computacional $\mathcal{O}(nc)$, creado a partir de sub-instancias del problema original, siendo n la cantidad de objetos posibles para empacar y c la capacidad máxima de la mochila (Martello & Toth, 1990).

Aunque la misma definición de los problemas NP indica que no existe un algoritmo "eficiente" para su resolución, aún se puede aprovechar la implementación de recursos que permitan acelerar estos métodos. La paralelización de procesos computacionales destaca como uno de éstos recursos, y parece vislumbrarse como una de las mejores

alternativas actuales para mejorar los tiempos de resolución de problemas de gran complejidad computacional. Robson (1992) postula que "[...] donde algoritmos sofisticados pueden ser paralelizados, el resultado bien puede ser que los problemas muy grandes que parecían insolubles pueden ser resueltos fácilmente con las maquinas que existen hoy [...]" (p.379).

3.2. Antecedentes del algoritmo Branch and Bound

Propuestos originalmente por los trabajos de Markowitz-Manne (1957), Eastman (1958) y Land-Doig (1960) como un "algoritmo automático para problemas de programación discreta", el algoritmo de optimización Branch and Bound (BnB) comprende una serie de pasos matemáticos de fácil entendimiento basados en el "dividir y conquistar" buscando soluciones factibles para problemas de variable entera mediante la separación del modelo en nodos o subproblemas más pequeños, comenzando con la solución del modelo relajado, creando posteriormente ramas con restricciones de integralidad y distinto acotamiento de las variables de decisión asociadas al modelo original (Lee & Mitchell, 2009).

$$P = Maximize\ 5x_1 + 4x_2$$
$$s.a. \quad x_1 + x_2 \leq 5$$
$$10\,x_1 + 6x_2 \leq 45$$
$$x_1 \geq 0; x_2 \geq 0;\ x_1, x_2 \in \mathbb{Z}$$

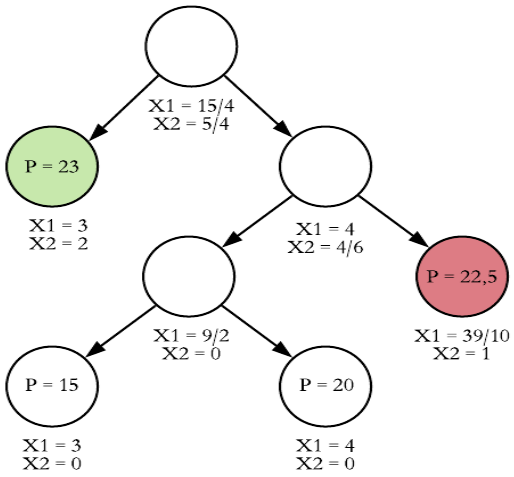

Figura 1. Representación gráfica de Branch and Bound para un problema de variable entera.

(Fuente: Elaboración propia).

El algoritmo sigue creando ramas y nodos hasta llegar a resultados inviables o hasta encontrar soluciones factibles que cumplan con las condiciones de integralidad, las cuales son luego comparadas en busca de un óptimo global. La mejor solución entera encontrada hasta el momento se denomina “incumbente” y sirve de punto de referencia para las siguientes exploraciones del árbol, pudiendo “podar” o cortar la indagación en aquellos nodos con soluciones peores a ésta, efectivamente agilizando el proceso de búsqueda, como se aprecia en el nodo rojo de la Figura 1.

```
Variables iniciales:
› listaNodos         #Lista con los nodos activos
› soluciónIncumbente #La mejor solucion encontrada
› nodoIncumbente     #El nodo con mejor solucion
› nodoInicial        #Problema totalmente relajado

### INICIO ###

Paso 1:

› Agregar nodoInicial en listaNodos

Paso 2:

› Elegir nodo de listaNodos
› Sacar nodo de listaNodos
› Generar nodosHijos acotando el nodo seleccionado

› Para cada nodoHijo generado:

        Resolver nodoHijo

        Si la soluciónHijo < soluciónIncumbente:
                Eliminar nodoHijo

        Si la soluciónHijo > soluciónIncumbente
        y variablesCumplenIntegralidad:

                problema EsFactible
                nodoIncumbente = nodoHijo
                soluciónIncumbente = soluciónHijo

        DeLoContrario:
                Agregar nodoIncumbente a la listaNodos

› Si listaNodos no está vacía:
        Volver al Paso 2

Paso 3:
› Si problema EsFactible:
        nodoÓptimo = nodoIncumbente
        soluciónÓptima = soluciónIncumbente

› Si no:
        problema EsInfactible

### FIN ###
```

Figura 2. Pseudocódigo de un algoritmo estándar de Branch and Bound. (Fuente: Elaboración propia).

El algoritmo se detiene cuando no quedan nodos activos, es decir, todas las ramas han sido exploradas o podadas. La mejor solución encontrada al llegar a ese punto corresponde al valor óptimo del problema.

3.2.1 Avances en la exploración del algoritmo

Desde su planteamiento original, se han propuesto y desarrollado múltiples formas de analizar el árbol de búsqueda, determinando estrategias para la selección de variables en la ramificación y para la selección de nodos a explorar. Dentro de estas últimas los trabajos de Allouche, Givry, Karsirelos, Schiex & Zytnicki (2015), González (2010) y Paulavičius, Žilinskas, & Grothey (2010) destacan:

- **DFS**: *Depth-First Search*, consistente en bajar por los nodos de la forma más rápida posible, buscando cualquier solución factible antes de seguir el procedimiento en la siguiente rama. Pretende encontrar una primera solución factible de forma rápida, aunque ésta no necesariamente sea una solución de "buena calidad" o de valor cercano al óptimo global, con el fin de tener una incumbente sobre la cual podar ramas del árbol lo más tempranamente posible. Utilizando como ejemplo el pseudocódigo descrito en la Figura 2, el nodo_A escogido correspondería al último de los *nodos_activos*, es decir, a aquel agregado más recientemente a la lista (similar a una estrategia LIFO).
- **BFS**: *Best-First Search*, basada en utilizar sólo los nodos con mayor cota superior. Al contrario de la estrategia anterior, la búsqueda de la mejor cota entre los nodos activos suele ralentizar la búsqueda de soluciones factibles. Sin embargo, la primera solución encontrada suele ser mejor que la obtenida mediante DFS, lo cual puede implicar una poda de ramas más eficiente. Utilizando nuevamente a la Figura 2 como ejemplo, el nodo_A seleccionado de

la lista de *nodos_activos* sería aquel almacenado con la solución de mejor valor (el mayor en un caso de maximización y el menor en caso de minimización).

- **BrFS**: *Breadth-First Search*, centrada en calcular los nodos más antiguos primero, consiguiendo que el árbol de búsqueda se explore "por nivel". Es una técnica de búsqueda más exhaustiva, pues no se centra en encontrar una solución de forma rápida, si no que primero concentra los recursos en la construcción de un árbol de búsqueda extendido. Debido a que esta técnica no suele presentar beneficios frente a las anteriores, no suele mencionarse mucho en la literatura. De acuerdo a la Figura 2, el nodo_A escogido en cada iteración correspondería al primero de la lista de *nodos_activos* (es decir, de forma similar a una estrategia FIFO).
- **HDFS** y **HBFS**: *Hybrid-Depth-First Search* y *Hybrid-Best-First Search*, respectivamente. Ambas estrategias son mezclas entre DFS y BFS, donde alguna de las dos prima sobre la otra, aunque se busca obtener los beneficios de ambos métodos. Un ejemplo de esta técnica se puede apreciar gráficamente en la Figura 3.

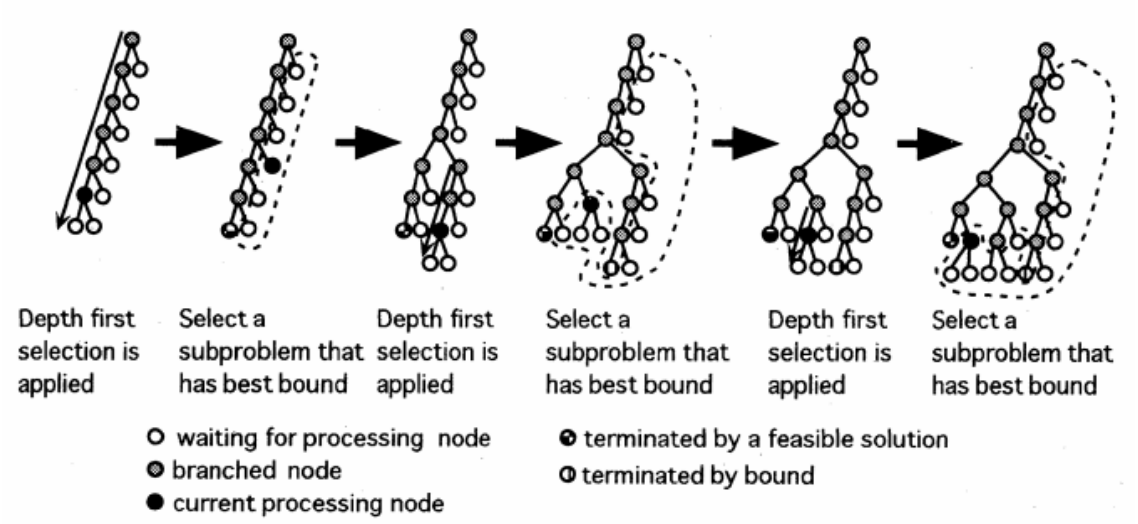

Figura 3. Estrategia de selección híbrida. (Fuente: Shinano, Higaki, & Hirabayashi (1995)).

Paulavičius, Žilinskas, & Grothey (2010) proponen tiempos más cortos de resolución al utilizar estrategias de DFS y BrFS debido al tiempo extra consumido por la estrategia BFS al analizar prioridades de nodos en cada iteración. Sin embargo, proponen que esta relación cambia ante procesamiento paralelo, tornándose la estrategia DFS en la de peor rendimiento, mientras que BFS y BrFS presentan los mejores niveles de eficiencia.

Para elegir la variable a ser acotada durante cada iteración del algoritmo de exploración, diversos métodos de selección han sido propuestos y estudiados. Es necesario recordar que las candidatas corresponden a aquellas que deben alcanzar un valor discreto. Ente los criterios mencionados por Lee & Mitchell (2009) destacan:

- **Variable entera menos factible**: seleccionando aquella con un valor actual más lejano al entero.
- **Variable entera más factible**: contrario al anterior, seleccionando la variable con un valor actual más cercano al entero.
- **Ramificación fuerte (Strong Branching)**: Se selecciona una lista de N variables binarias con el valor actual más fraccionario. Luego, sobre cada

una de éstas, se realiza una iteración de simplex ajustando su valor a 0 y luego a 1. Aquella variable que posea la mejor suma de estos valores es entonces seleccionada para la siguiente iteración del algoritmo Branch and Bound.

- **Selección de prioridades**: Variables seleccionadas de acuerdo a prioridades asignadas por el usuario o por los coeficientes de la función objetivo.

Haciendo uso del algoritmo Branch and Bound, se ha dado origen a métodos de resolución más especializados, tales como Branch and Cut (Akrotirianakis, Rustem, & Maros, 2000), enfocado en unir Branch and Bound con Corte de Planos para problemas de programación entera mixta no lineal (MINLP) (Mitchell, 2008), y Branch and Price, enfocado en problemas de optimización entera que cuentan con una cantidad enorme de variables (Savelsbergh, 2008).

A nivel de ciencias computacionales, los algoritmos Branch and Bound resaltan en su uso para la exploración de soluciones óptimas en problemas de búsqueda y de clase de complejidad NP-difícil (Costanzo, Baduel, Caromel, & Matsuoka, 2007).

3.2.2 El "gap" como medida de eficacia

Considerando la magnitud de algunos problemas de optimización, es necesario establecer formas de medir el avance de las soluciones encontradas. Algunas de las medidas más utilizadas corresponden a las de gap absoluto y gap relativo.

González (2010) define al gap absoluto como "la diferencia [absoluta] entre el valor objetivo de la mejor solución factible (cota primaria) y la cota dual global (mejor relajación lineal)" y al gap relativo como la "diferencia relativa al valor de la cota dual", es decir:

$$gap_{absoluto} = |c_p - c_d| \tag{7}$$

$$gap_{relativo} = \left|\frac{c_p - c_d}{c_d}\right| \tag{8}$$

Siendo c_p la cota primaria y c_d la cota dual global.

Cabe destacar que ambas diferencias se plantean como absolutos debido a que, para maximización, $c_p < c_d$, mientras que para minimización $c_p > c_d$. Considerando que el gap finalmente calcula la "distancia" entre estos valores, los resultados obtenidos siempre son mayores o iguales a 0. Mientras menor sea este valor, mejor es la calidad de la solución, pudiendo llegar a cero al encontrar un óptimo global.

En un problema de maximización, la cota primaria estaría dada por el valor objetivo más alto que cumple con las restricciones de integralidad del problema, mientras que la cota dual global correspondería a la solución más alta encontrada por un nodo activo del árbol de búsqueda, aún si no cumple con las restricciones de integralidad. Al comenzar el algoritmo, por tanto, esta última cota vendría determinada por la solución del problema completamente relajado, disminuyendo en magnitud a medida de que se explora el árbol.

Al trabajar con problemas altamente complejos, puede ser poco viable o necesario el encontrar el óptimo global, por lo que el gap puede utilizarse para determinar un final prematuro del algoritmo tras alcanzarse una solución lo suficientemente cercana al óptimo. Tanto los solvers comerciales Gurobi (Gurobi Optimization, 2019) como CPLEX (IBM Corporation, 2019) plantean, como valor estándar, un gap relativo de 10^{-4} para modelos de programación entera.

3.2.3 Branch and Bound en paralelo

Laursen (1993) postula que un algoritmo Branch and Bound puede mejorar de manera sustancial, o mejorando las funciones de acotamiento de cada problema específico, o realizando una paralelización efectiva de éste.

Con el termino de *paralelización* se propone el dividir la carga de procesamiento del algoritmo en múltiples núcleos computacionales, con el fin de poder agilizar la resolución de los problemas de optimización que pudiesen enfrentarse a tiempos exponenciales o polinómicos (Crainic, Le Cun, & Roucairol, 2006).

Gendron & Crainic (1994) proponen la existencia de tres tipos de paralelización de algoritmos Branch and Bound:

- Paralelismo de tipo 1: Paralelización al realizar operaciones sobre nodos del árbol (El autor propone como ejemplo el realizar acotamiento paralelo de subproblemas).
- Paralelismo de tipo 2: Construcción paralela de ramas de un mismo árbol, al evaluar subproblemas de forma paralela.
- Paralelismo de tipo 3: Construcción paralela de múltiples árboles de Branch and Bound utilizando operaciones variadas, como estrategias de ramificación y selección de nodo distintas entre sí.

Corrêa & Ferreira (1995) discuten dos sistemas de paralelización para problemas discretos, denominados como sincrónicos (synchronous) y asincrónicos (asynchronous), donde los primeros se basan en obtener de manera simultánea la información compartida entre procesadores, manteniendo mayor coordinación, mientras que los últimos permiten que cada procesador actúe otorgando y recibiendo información de la estructura global de manera independiente, acelerando la exploración al no tener que esperar a que todos los procesadores cumplan con la iteración antes de pasar a la

siguiente. Es importante notar que “implementaciones sincronizadas sólo son útiles bajo casos especiales o con pocos procesadores y generalmente conducen a cuellos de botella comunicacionales” (Ferreira & Corrêa, 2005), esto debido a que un procesador pudiese demorarse más que el resto en resolver un subproblema, retrasando al resto de los procesadores en el caso sincrónico. Una implementación asincrónica efectivamente elimina esta complicación. Es importante notar que estos factores afectan especialmente al paralelismo de tipo 2 definido previamente, pues la velocidad de construcción de las ramas depende de si existe o no un sistema sincrónico.

La literatura plantea la existencia de posibles *anomalías de aceleración* (speedup anomalies), las cuales pueden causar que la relación entre la cantidad de procesadores y la velocidad de resolución del algoritmo difiera mucho de la linealidad, presentando comportamientos anti-intuitivos (Corrêa & Ferreira, Parallel best-first branch- and-bound in discrete optimization: A framework, 2005), esto es, aumentando los tiempos de resolución aún por sobre los que tendría un procesamiento secuencial y no paralelo, o aumentando la velocidad en un ratio incluso mayor a la cantidad de procesadores utilizados (Lai & Sahni, 2002).

3.2.4 La pereza y la impaciencia en la estrategia de exploración

Clausen & Perregaard (1999) postulan que la estrategia de DFS suele ser más eficiente que BFS, debido a un concepto definido como *laziness* (pereza), propuesto como la postergación del cálculo de cotas durante el mayor tiempo posible, calculándolas solo al seleccionarse el nodo para su resolución, contrastado con *eagerness* (impaciencia), correspondiente al método tradicional de calcular las cotas de los nodos activos tras cada iteración del algoritmo. El estudio propuesto por los autores concluye que:

> [...] En general BFS es inferior a DFS, tanto en el caso secuencial como paralelo, y tanto en tiempos de ejecución como en nodos acotados [...]. En cuanto a DFS, no parece haber un ganador claro entre la estrategia perezosa e impaciente, ni en una configuración paralela o serial. (pp. 14-15)

3.2.5 Medidas de eficiencia de un algoritmo de Branch and Bound paralelo

Gendron & Crainic (1994) proponen dos formas de medición para la eficiencia de algoritmos Branch and Bound paralelos:

- Cantidad de subproblemas generados, ya sea mediante el número de subproblemas totales creados por el algoritmo o por la cantidad de supbroblemas generados antes de encontrar la solución óptima.
- Aceleración y eficiencia ante el uso de múltiples procesadores, calculando la variación entre el tiempo de resolución del algoritmo serial y el algoritmo paralelizado. Considerando el tiempo de resolución $T(p)$ al utilizar p procesadores, los autores definen:
    - Aceleración (Speedup): $S(p) = T(1)/T(p)$
    - Eficiencia: $E(p) = S(p)/p$

En el artículo *Parallel Branch-And-Bound Algorithms: Survey and Synthesis* (1994), Gendron & Crainic mencionan que el determinar la forma de calcular el tiempo $T(p)$ puede resultar complejo, pues existen múltiples formas de aplicar el algoritmo, con técnicas de exploración de diferente eficiencia. Considerando esto, proponen utilizar el algoritmo paralelo para ambos casos, a pesar de que se utilice solo un procesador, manteniendo además las mismas técnicas de búsqueda para los casos de $p = 1$ y $p > 1$. El indicador de aceleración obtenido mediante el uso del mismo algoritmo para la

versión lineal y paralela es denominado por Crainic, Le Cun, & Roucairol (2006) como aceleración relativa, al contrario de la versión absoluta, que utiliza el tiempo del mejor algoritmo serial como comparativo. Por temas de simplicidad, desde ahora el término de aceleración hará referencia al tipo relativo.

El indicador de aceleración, como su nombre lo indica, representa los cambios en la velocidad de resolución del algoritmo ante el uso de múltiples procesadores en paralelo. Un valor de $S(p) = 2$, por ejemplo, sería signo de que el algoritmo paralelizado con p procesadores encuentra la solución óptima dos veces más rápido que el mismo algoritmo con sólo un procesador. El indicador de eficiencia, por otro lado, indica cuán eficiente es, en promedio, el uso de cada procesador.

Cabe mencionar que ambas medidas presentan un peso mínimo en costo computacional, debido a que la primera se presenta simplemente como un contador, mientras que la segunda se puede realizar de forma manual tras haber finalizado el algoritmo, habiendo insertado previamente un temporizador en éste que indique el tiempo de resolución, como puede realizarse mediante el módulo *time* (Python Software Foundation, 2020).

Considerando este último punto, es necesario determinar el concepto de tiempo de manera más específica para el trabajo computacional. Por lo general se distinguen dos tipos de temporalidad: tiempo de CPU (o CPU time) y tiempo real (o wallclock time). El primero considera el tiempo durante el cual un proceso específico ha utilizado una CPU, mientras que el segundo contabiliza el tiempo de forma real, como un "reloj de pared", lo cual incluiría por tanto períodos de espera de los procesos para utilizar el procesador (The Free Software Foundation, 2008). Barr & Hickman (1993) plantean que, en sistemas de un solo procesador, el tiempo de CPU es la elección más común, mientras que para el caso de uso paralelo de procesadores se sugiere el uso del tiempo

real. Esto debido a que lo que se busca alcanzar con la paralelización es, por lo general, la reducción de los tiempos reales de ejecución.

3.3. Computación paralela en Python

Con el propósito conseguir el trabajo coordinado entre diversos procesadores a la hora de resolver un problema de optimización, es necesario establecer previamente comunicación entre éstos. Si bien en Python Wiki (2019) se menciona una gran cantidad de librerías para procesamiento paralelo y multiprocesamiento en Python, Marowka (2018) centra principalmente la discusión de la paralelización en cuatro soluciones, de las cuales destacan el uso de los módulos *Threading* (Python Software Foundation, 2020) y *Multiprocessing* (Python Software Foundation, 2020), además del paquete *mpi4py* (Dalcin L. , 2019).

Para entender las diferencias fundamentales entre los módulos *Threading* y *Multiprocessing*, es necesario plantear previamente la existencia del GIL o *Global Interpreter Lock*, definido como un bloqueo total dentro del intérprete de CPython subyacente, creado para evitar posibles *deadlocks* entre tareas múltiples. Está diseñado para proteger el acceso a objetos de Python al prevenir que múltiples hilos se ejecuten simultáneamente (Hunt, Threading, 2019). Esto debido a que, al contrario que como ocurre con los procesos, los hilos utilizan un espacio de memoria compartida, y el funcionamiento paralelo de hilos puede corromper el intérprete al sobrescribir información de manera simultánea.

El módulo *Threading*, entonces, no funciona realmente en paralelo, sólo lo aparenta al trabajar con hilos, de bajo impacto en la memoria del sistema. Si bien pudiese ser útil ante algunas necesidades, la mayor desventaja se encuentra en que la comunicación entre hilos es compleja debido al GIL (Hunt, Threading, 2019).

Por otro lado, sobresale el módulo *Multiprocessing*, basado en el uso de procesadores enteros por sobre hilos individuales. Si bien esto se traduce en un mayor peso en la memoria, se presenta la posibilidad de permitir la comunicación paralela entre procesos al evitar el GIL (Python Software Foundation, 2020), pudiendo además compartir información entre éstos (Hunt, Multiprocessing, 2019). Además, el uso de espacios de memoria separados aporta mayor seguridad en el funcionamiento del intérprete.

El paquete *mpi4py* (MPI for Python) funciona en base al sistema de traspaso de mensajes *Message Passing Interface* (MPI), el cual se establece como un estándar para la comunicación paralela entre computadores y procesos (Dalcin, Paz, Kler, & Cosimo, 2011). Traduciendo sintaxis de C/C++ y Fortran, *mpi4py* permite adaptar este estándar para su uso en Python, permitiendo la comunicación paralela eficiente y flexible utilizando el módulo *cPickle* para empacar y desempacar información (Dalcín, Paz, & Storti, MPI for Python, 2005). Si bien MPI se establece como un estándar de alta eficiencia en la comunidad científica, su manejo es más complejo que el requerido por las alternativas anteriores, pues necesita de conocimientos profundos sobre el estándar MPI y el funcionamiento de C/C++.

3.4 El lenguaje de modelamiento Pyomo

Pyomo es un software de libre acceso introducido el año 2008 como parte del proyecto COIN-OR, creado con el propósito de servir de base para modelar y resolver problemas de optimización de diversos tipos en Python (Hart et al. 2017). Los mayores beneficios que éste presenta se apoyan en la amplia librería que Python posee, además de que el lenguaje facilita la integración de computación distribuida, permitiendo la

ejecución de solvers de manera sincronizada en distintos ambientes (Hart, Watson, & Woodruff, 2011).

El software permite programar en base a modelos abstractos, es decir, separando la declaración del modelo de su data, de manera similar a AMPL. Pyomo permite incluso adoptar el formato de data de AMPL para su uso propio (Hart et al. 2017).

Tras la definición de un modelo abstracto, se crean instancias concretas con la data otorgada, siendo éstas posteriormente resueltas con los solver designados.

4. METODOLOGÍA

4.1. Elaboración de un algoritmo de Branch and Bound

Haciendo uso de Python 3.5.6 se programa un algoritmo de Branch and Bound, encargado principalmente de establecer el "esqueleto" del árbol de búsqueda, ramificando y seleccionando nodos de acuerdo con estrategias establecidas dentro del código. En un principio, el código fue escrito en Python 2.7.15 pero, debido a restricciones de dicha versión en algunos de los módulos utilizados, se opta por realizar el traspaso a Python 3. Adicionalmente, se crea la posibilidad de paralelizar la construcción del árbol haciendo uso del módulo *Multiprocessing*, creando procesos paralelos comunicados entre sí.

El código está dividido en 2 librerías, utilizando un archivo adicional para modelar el problema de ejemplo. Ésta división separa el algoritmo de armado de árbol y paralelización de las estrategias de búsqueda.

El código cuenta con las estrategias de exploración impaciente BrFS, BFS y DFS.

A su vez, en cuanto a la selección de variables, destacan los posibles criterios de más/menos factible, así como el método por ramificación fuerte.

El criterio para determinar una variable entera es de 10^{-4} (absoluto), esto es, la diferencia entre el valor actual y el del entero más próximo no puede ser mayor a ésta cifra.

El valor relativo utilizado para el gap óptimo se ajusta a 10^{-4}, por defecto.

El modelo a ser resuelto es escrito utilizando la formulación abstracta propuesta por el lenguaje de modelamiento Pyomo 5.6.9. Adicionalmente, la resolución de los nodos se externaliza al algoritmo de construcción, utilizando el solver Gurobi 8.1.1 con apoyo

de una licencia académica para cumplir esta función, aunque se da la opción de poder cambiar de solver si se desea.

La ejecución del algoritmo se realiza utilizando un ambiente creado en Conda 4.7.10. Originalmente se pensó en utilizar el código en un IDE como PyScripter o Spyder, pero ambos presentaron dificultad para trabajar con el módulo de multiprocesamiento de forma correcta.

Para invocar el algoritmo en el archivo del modelo abstracto, se importa el módulo **parallel_bnb**, utilizando su función *parallelize* para la resolución del problema utilizando el algoritmo programado.

Los argumentos de dicha función distinguen:

- *model*: El modelo abstracto a ser cargado en el programa.
- *data*: La data correspondiente al modelo.
- *optsolver*: Solver a utilizar.
- *processes*: Cantidad de procesos y, por tanto, núcleos computacionales en paralelo que se desea utilizar. Por defecto, se utilizarán todos los núcleos disponibles en el equipo.
- *strat*: Estrategia de exploración a utilizar. Acepta tanto valores enteros como strings. (1 o "brfs": Estrategia BrFS; 2, "bfs" o "befs": Estrategia BFS; 3 o "dfs": Estrategia DFS. Valor por defecto: 2).
- *initial_incumbent*: Valor inicial de la incumbente. Puede ser introducido por el usuario. Por defecto su valor es *None* (no se tiene valor inicial).

Cabe mencionar que resulta imprescindible que la función se invoque dentro de un bloque **if __name__ == "__main__":** para su correcto funcionamiento.

El algoritmo termina cuando todos los procesos paralelos de Branch and Bound finalizan su ejecución, lo que ocurre de forma independiente para cada uno cuando se cumple la primera de las siguientes condiciones:

1. **No quedan nodos por explorar**, esto es, cuando todas las ramas han sido podadas y los nodos se encuentran resueltos. Ésta es la condición clásica de término para un algoritmo Branch and Bound.
2. **Se cumple la condición de gap**, alcanzando una diferencia relativa de 10^{-4} entre la mejor solución de los nodos activos y el mejor valor que cumple con las condiciones de integralidad. Esta condición trabaja de forma independiente en cuanto a los nodos activos y la cota dual de cada proceso. Sin embargo, el valor óptimo integral correspondiente a la cota primaria, se basa en el incumbente global, compartido entre todos.
3. **El valor incumbente global es superior a la cota dual local**, indicando que la rama sobre la que se está trabajando fue podada debido a que su mejor valor posible (aún sin cumplir con las condiciones de integralidad) es peor al de la solución incumbente actual.

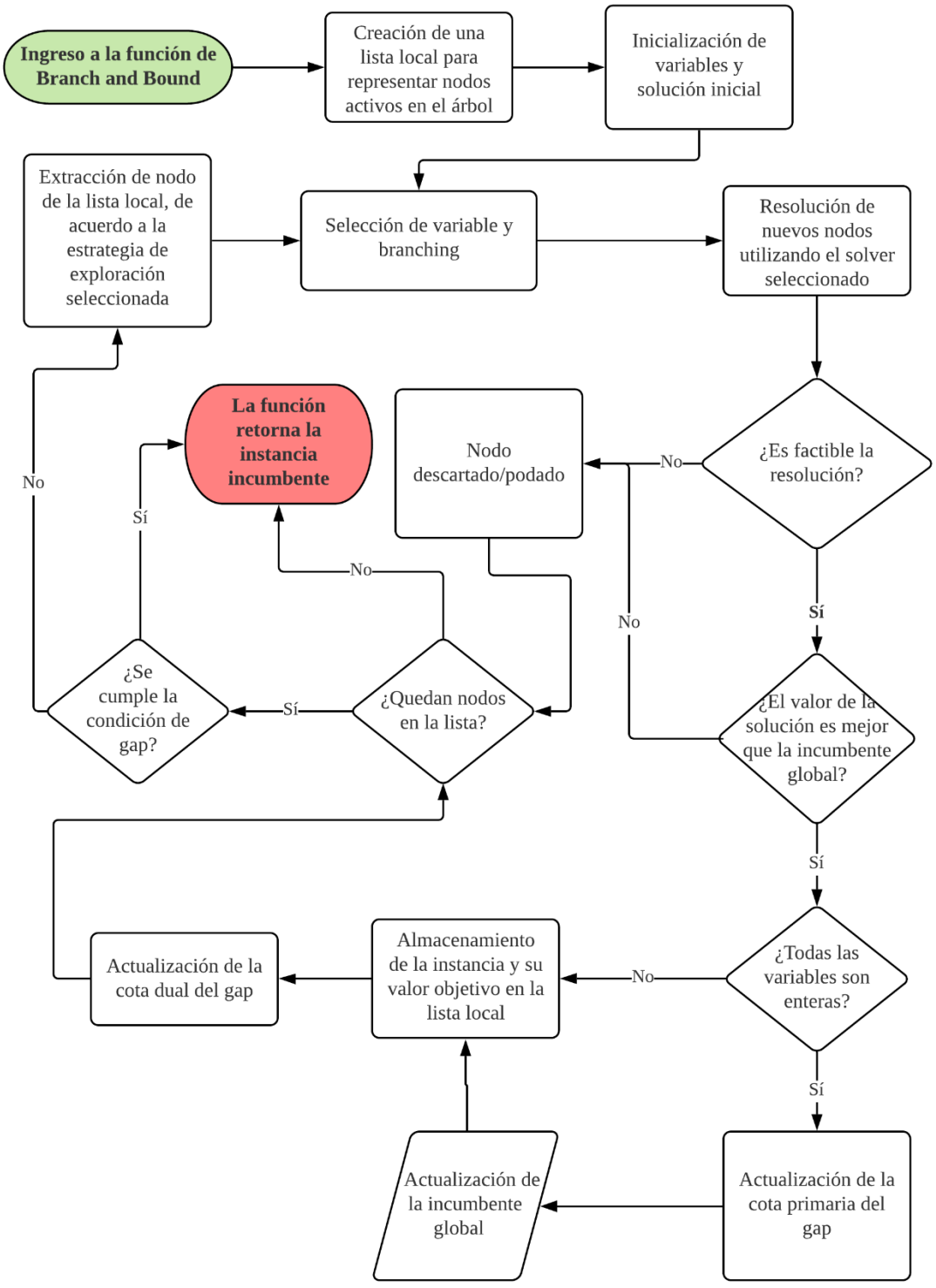

Figura 4. Diagrama simplificado de la función bnbparallelbranching. *Ésta se ejecuta en múltiples procesadores de forma paralela mediante la función* parallelize, *compartiendo la incumbente global. (Fuente: Elaboración propia)*

4.1.1 Descripción de las funciones principales desarrolladas en el algoritmo

- ***relative_gap***: cálculo del gap relativo, utilizando la cota dual secundaria global y la cota primaria local al proceso.
- ***bnb_branching***: construcción de un árbol de búsqueda serial. Utilizado cuando se desea trabajar con sólo un procesador. Incluye las condiciones de término del branching.
- ***bnbparallelbranching***: construcción de un árbol de búsqueda en paralelo. Utilizado cuando se trabaja con más de un procesador, ya que considera la comunicación entre distintos procesos. De manera similar al formato serial, incluye las condiciones de término del branching. Una forma gráfica simplificada de esta función se puede apreciar en la Figura 4.
- ***init_bnbparallelbranching***: construcción de los primeros nodos para trabajar con la función *bnbparallelbranching*. Utiliza el modelo original y obtiene 2^n nodos iniciales "de misma profundidad", donde n es la profundidad de la exploración inicial. La cantidad de nodos equivale a la cantidad de procesadores con los que se desea trabajar[2].
- ***parallelize***: inicialización del algoritmo y distribución de los procesos. Además recibe y coordina los valores de la incumbente global entre procesadores, comunicándolos. Al finalizar el algoritmo, muestra al usuario el valor óptimo obtenido, así como sus variables, en el formato usual entregado por Pyomo al imprimir un modelo concreto.
- ***tree_for_threads***: utilizada en la primera iteración de Branch and Bound paralelo, devuelve una cantidad de nodos para explorar similar a la cantidad de procesos paralelos deseados para trabajar, de modo de que posteriormente

[2] En caso de querer trabajar con un número procesadores que no es potencia de 2, algunos de los nodos se explorarán inicialmente con una profundidad menor al resto, efectivamente reduciendo la cantidad de nodos iniciales.

a cada proceso se le pueda ser asignado un nodo para comenzar la exploración.

- ***i_var_relaxation***: relajación de variables originalmente definidas como enteras o binarias.
- ***var_relaxation***: aplicación de la función *i_var_relaxation* sobre cada variable entera o binaria del modelo original.
- ***is_integer***: comprueba la integralidad de las variables obtenidas tras resolver las instancias en el árbol de exploración, sujeto a una tolerancia establecida por 10^{-4}, correspondiente a la diferencia absoluta entre el valor de la variable y su entero más cercano.
- ***condition_integer***: aplica el módulo *is_integer* sobre todas las variables originalmente enteras o binarias. Devuelve un valor booleano *True* si todas cumplen con la condición, indicando que la instancia resuelta cumple con las restricciones de integralidad del modelo original.
- ***most_fractional_variable***: estrategia de branching (elección de la variable a acotar), seleccionando a aquella cuyo valor actual diste más de un entero.
- ***less_fractional_variable***: estrategia de branching (elección de la variable a acotar), seleccionando a aquella cuyo valor actual se acerque más a un entero, sin cumplir con la tolerancia de integralidad requerida por la función *is_integer*.
- ***strong_branching***: estrategia de branching que selecciona aquella variable con más influencia agregada sobre la función objetivo (cambio más significativo en el valor objetivo tras realizar una iteración).
- ***branching_strat***: devuelve la variable seleccionada con la estrategia de branching elegida.

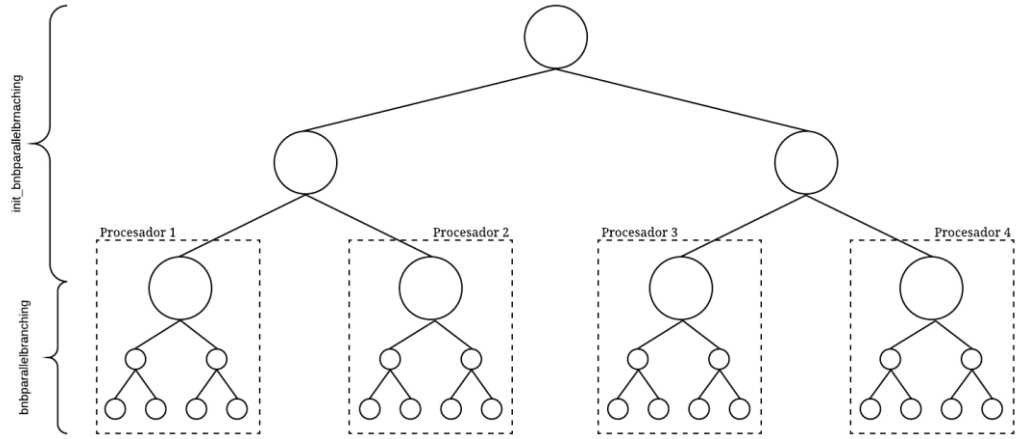

Figura 5. Representación gráfica de la distribución del algoritmo en paralelo para 4 procesadores.

(Fuente: Elaboración propia)

4.2. Paralelización

La paralelización programada en el algoritmo se adapta al *Tipo 2* propuesto por Gendron & Crainic (1994), al crear ramas paralelas de un mismo árbol, atribuyendo las mismas estrategias de exploración y selección de variables a cada una de éstas. Cada rama es asignada a un procesador distinto, compartiendo entre sí el valor de la incumbente en tiempo real, con el fin de poder llevar a cabo la poda de ramas de forma eficiente.

De forma más específica al código, la distribución paralela se lleva a cabo mediante el módulo *Multiprocessing* debido a la facilidad de uso y beneficios frente a alternativas como *Threading* y *mpi4py*. Si bien no posee las ventajas de trabajar sobre un lenguaje altamente eficiente como C, el hecho de poder manejar paralelización de forma relativamente fácil, segura y pudiendo evitar el GIL de Python (logrando así intercomunicar los procesos), ya es suficiente para preferir esta alternativa.

El mayor inconveniente puede presentarse frente a la rapidez de inicio del algoritmo, pues la inicialización de procesos resulta más lenta que la inicialización de hilos. Sin embargo, el módulo permite trabajar en paralelo explotando todos los

procesadores al mismo tiempo, factor que no puede ser aprovechado al utilizar *Threading*. Este problema asociado al traspaso directo de grandes cantidades de información al crear procesos paralelos mediante *multiprocessing* puede significar pérdidas importantes de eficiencia, creando tiempos de inicialización que pueden incluso superar a los tiempos de resolución del modelo. Para reducir esto, el algoritmo serializa y almacena de forma externa la información del modelo como una secuencia de bytes en un archivo de texto, siendo este luego importado dentro de cada proceso paralelo. Gracias a esto, ya no es necesario traspasar la data directamente a cada proceso, debido a que basta con entregarle la ubicación del archivo para que éste pueda realizar el importe y decodificación[3].

4.3. Evaluación de eficiencia

Para utilizar las medidas estudiadas para medir la eficiencia y calidad del algoritmo y las soluciones, se implementa en el código el valor del *gap relativo*, así como de la cota primaria y dual, para su visualización y entendimiento en la CUI o GUI. Adicionalmente, mediante el módulo *time*, se calcula el tiempo de ejecución del algoritmo, pudiendo evaluar el estado de éste tras determinados periodos de tiempo, así como también la duración total del algoritmo previo a la determinación del óptimo global, gracias a lo cual será posible determinar tanto la aceleración como la eficiencia de la paralelización, de acuerdo a los indicadores propuestos por Gendron & Crainic (1994).

Por último, debido al almacenamiento de los nodos activos en una lista local, es posible saber fácilmente la cantidad de éstos en todo momento, por lo que también será

[3] Este proceso de codificación o serialización y de-serialización para traspaso ágil de información se realiza mediante los procesos de *Pickling* y *Unpickling*, implementados por el módulo *pickle* de Python (https://docs.python.org/3.5/library/pickle.html).

implementada la cifra de nodos activos versus tiempo como medida de eficiencia del algoritmo.

4.4. Resolución de problemas

Los problemas tipo para la evaluación del algoritmo fueron extraídos de la compilación *Benchmark instances for the Multidimensional Knapsack Problem* (Drake, 2015), centrados en el problema multidimensional de la mochila. Si bien el compilado contiene datos para más de 300 problemas con múltiples instancias, para el testeo se utilizaron los archivos correspondientes a los problemas *pet7, pb6, sento2* y *weing8*, todos de variable binaria, con tamaños y complejidad diferente, tabulados en la Tabla 1. Problemas combinatoriales de Benchmarking utilizados para la evaluación de la paralelización mediante el algoritmo de Branch and Bound programado.. Las restricciones indicadas no incluyen a aquellas relacionadas con la naturaleza de las variables, como no negatividad o integralidad. La prueba se realizó sobre un total de 1, 2, 4, 8, 16 y 32 núcleo(s) computacionales, utilizando un equipo con dos procesadores Intel Xeon Silver 4114 CPU @2.20 Ghz y 2.19 Ghz, con un sistema operativo Windows 10 Pro for Workstations y 64 GB RAM.

Tabla 1. Problemas combinatoriales de Benchmarking utilizados para la evaluación de la paralelización mediante el algoritmo de Branch and Bound programado.

Problema	Variables de decisión	Restricciones	Óptimo
pet7	50	5	16537
pb6	40	30	776
sento2	60	30	8722
weing8	105	2	624319

5. RESULTADOS

5.1 Análisis y discusión

Tanto los tiempos de resolución como los indicadores de aceleración y eficiencia, tras resolver los problemas tipo con el algoritmo de Branch and Bound paralelo, fueron obtenidos utilizando dos estrategias de exploración: DFS (Depth-First Search) y BFS (Best-First Search), ambas *impacientes*, según la definición de Clausen & Perregaard (1999). Además, se acotó la variable menos factible (más fraccionaria) durante cada iteración. Estos resultados fueron posteriormente graficados con el propósito de facilitar la visualización del comportamiento del ratio de aceleración como del de eficiencia al aumentar la cantidad de núcleos computacionales trabajando en paralelo para la resolución del algoritmo.

Tabla 2. Tiempos wallclock de resolución de diversos problemas de la mochila múltiple, medidos en segundos, utilizando la estrategia DFS.

Resolución [s]	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	2.493,91	1.362,47	942,96	532,69	194,53	124,44
pb6	859,46	525,30	422,41	281,91	205,50	195,48
sento2	3.136,43	1.299,35	1.049,55	621,66	427,79	406,64
weing8	1.078,92	1.068,05	828,82	348,35	306,60	282,94

Tabla 3. Tiempos wallclock de resolución de diversos problemas de la mochila múltiple, medidos en segundos, utilizando la estrategia BFS.

Resolución [s]	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	56,28	46,12	48,05	46,37	53,35	53,42
pb6	198,09	134,71	103,57	93,70	82,45	90,77
sento2	654,07	404,21	340,33	183,27	122,22	127,53
weing8	528,29	464,03	324,04	173,20	148,61	159,42

Sin importar la estrategia de búsqueda seleccionada, el algoritmo sigue un camino específico para llegar a cada solución factible. No obstante, este recorrido no siempre es fructífero, pudiendo obtener soluciones peores a la incumbente o llegando a infactibilidades al final de cada rama de exploración. La paralelización ayuda a reducir el efecto que puede suponer el que un proceso "se vaya por la rama equivocada" al indagar en el árbol de búsqueda pues, al mismo tiempo, los otros procesos paralelos se encuentran cubriendo más posibilidades de ruta, situación que no ocurre al utilizarse un algoritmo serial, donde el algoritmo completo debe llegar a una solución, sea factible o no, antes de comenzar con la siguiente iteración.

Aquí se halla el principal beneficio aportado por el método paralelo, pues esta exploración simultánea aumenta las posibilidades de seguir el camino correcto en cualquier momento dado, efectivamente reduciendo los tiempos de resolución del algoritmo Branch and Bound, como queda demostrado en los resultados mostrados en las Tabla 2. Tiempos wallclock de resolución de diversos problemas de la mochila múltiple, medidos en segundos, utilizando la estrategia DFS. y Tabla 3. Tiempos wallclock de resolución de diversos problemas de la mochila múltiple, medidos en segundos, utilizando la estrategia BFS.. Se debe considerar adicionalmente que, al compartir información de manera continua, las ramas de cada proceso pueden ser descartadas sin haber sido exploradas en su totalidad, en base a la comparación entre la mejor solución local obtenida hasta el momento y la incumbente global, lo cual aporta aún más a la velocidad de resolución del algoritmo.

5.1.1 Análisis de las métricas de paralelización

El punto central de la discusión se basa en la utilidad que la paralelización de procesos posee en el contexto de la resolución de problemas de optimización combinatorial. ¿Es ésta una técnica viable y eficiente para este propósito?

Tabla 4. Aceleración (Speedup) de diversos problemas de la mochila múltiple, utilizando una estrategia de exploración DFS.

Aceleración	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	1,00x	1,83x	2,64x	4,68x	12,82x	20,04x
pb6	1,00x	1,64x	2,03x	3,05x	4,18x	4,40x
sento2	1,00x	2,41x	2,99x	5,05x	7,33x	7,71x
weing8	1,00x	1,01x	1,30x	3,10x	3,52x	3,81x

Tabla 5. Aceleración (Speedup) de diversos problemas de la mochila múltiple, utilizando una estrategia de exploración BFS.

Aceleración	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	1,00x	1,22x	1,17x	1,21x	1,05x	1,05x
pb6	1,00x	1,47x	1,91x	2,11x	2,40x	2,18x
sento2	1,00x	1,62x	1,92x	3,57x	5,35x	5,13x
weing8	1,00x	1,14x	1,63x	3,05x	3,55x	3,31x

Las Tabla 4 y 5 muestran ratios de aceleración positivos para todos los casos. Esto implica una superioridad del algoritmo paralelizado por sobre el serial en cuanto a mejora en los tiempos de resolución, al menos para los problemas de benchmarking planteados.

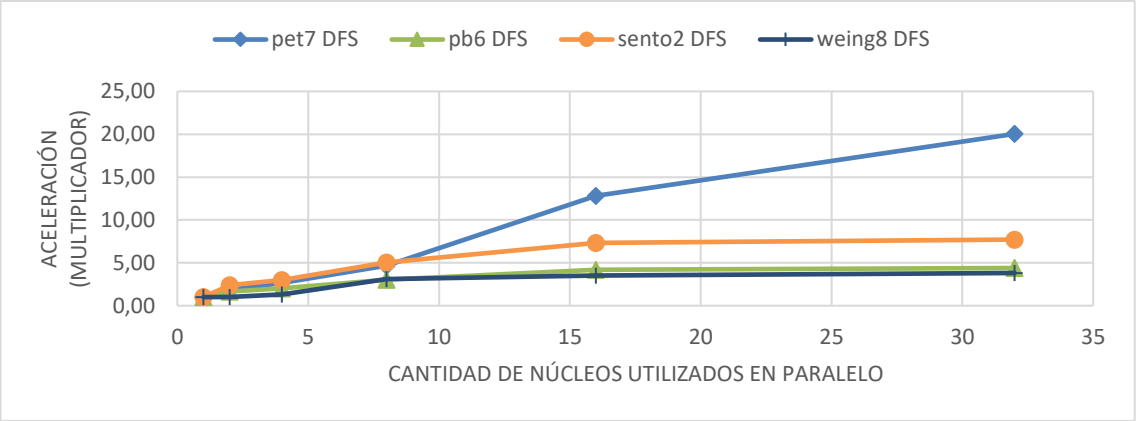

Figura 6. Representación gráfica del ratio de aceleración ante la paralelización del algoritmo utilizando una estrategia DFS.

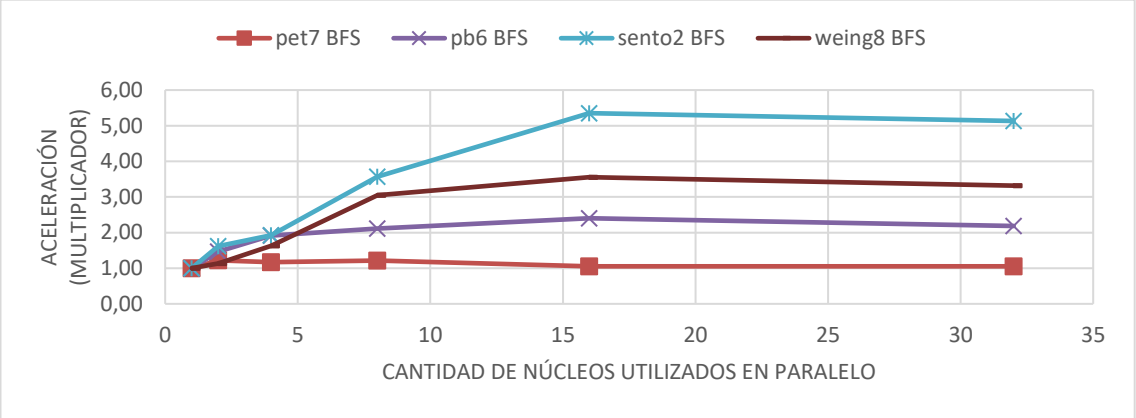

Figura 7. Representación gráfica del ratio de aceleración ante la paralelización del algoritmo utilizando una estrategia BFS.

Resulta interesante notar que, la relación entre la aceleración y el número de núcleos utilizados suele distar de la linealidad. Como se puede apreciar en las Tabla 6 y

Tabla 7, la tendencia suele demostrar un comportamiento que se ajusta más a un crecimiento logarítmico o potencial fraccionario (tipo raíz), con la excepción del caso anómalo presentado por la resolución del problema *pet7* al utilizar una estrategia BFS.

Tabla 6. Coeficientes de determinación del ajuste de la aceleración ante la paralelización para los problemas resueltos mediante la estrategia DFS.

R^2	Lineal	Potencial	Logarítmica

pet7	0,979	0,982	0,833
pb6	0,787	0,964	0,967
sento2	0,790	0,932	0,972
weing8	0,729	0,889	0,892

Tabla 7. Coeficientes de determinación del ajuste de la aceleración ante la paralelización para los problemas resueltos mediante la estrategia BFS.

R^2	Lineal	Potencial	Logarítmica
pet7	0,137	0,008	0,011
pb6	0,437	0,800	0,836
sento2	0,738	0,947	0,921
weing8	0,609	0,897	0,880

El hecho de que, para el caso general, el ratio de aceleración registrado se incline hacia un comportamiento potencial fraccionado indica un aumento decreciente en las velocidades de resolución ante el aumento de procesos en paralelo, tanto para la estrategia DFS como para la BFS. Para ambos casos, el cambio más marcado ocurre al pasar de 1 a 2 procesos paralelos.

Tabla 8. Medición de la eficiencia de paralelización de diversos problemas de la mochila múltiple resueltos utilizando una estrategia de exploración DFS.

Eficiencia	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	100,0%	91,5%	66,1%	58,5%	80,1%	62,6%
pb6	100,0%	81,8%	50,9%	38,1%	26,1%	13,7%
sento2	100,0%	120,7%	74,7%	63,1%	45,8%	24,1%
weing8	100,0%	50,5%	32,5%	38,7%	22,0%	11,9%

Tabla 9. Medición de la eficiencia de paralelización de diversos problemas de la mochila múltiple resueltos utilizando una estrategia de exploración BFS.

Eficiencia	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	100,0%	61,0%	29,3%	15,2%	6,6%	3,3%
pb6	100,0%	73,5%	47,8%	26,4%	15,0%	6,8%
sento2	100,0%	80,9%	48,0%	44,6%	33,4%	16,0%
weing8	100,0%	56,9%	40,8%	38,1%	22,2%	10,4%

Como se puede notar en la métrica de eficiencia, reportada en las Tabla 8 y Tabla 9, las mediciones de eficiencia de los núcleos, utilizando sólo dos en paralelo, se ubican entre el 50% y el 120%, mientras que al pasar a cuatro, éste intervalo decrece fuertemente, ubicándose entre 29% y 75%. Se debe recordar que ante un escenario ideal, de linealidad entre la cantidad de procesos paralelos y la aceleración obtenida, la eficiencia debiese ser constante y de 100%. El decrecimiento visto en los resultados reales indica que, si bien la paralelización mejora los tiempos de resolución, el agregar más procesos paralelos no siempre producirá cambios significativos. El problema *pb6* resuelto con una estrategia DFS, por ejemplo, muestra un cambio relativo en la aceleración de 37,0% al pasar de 8 a 16 núcleos paralelos. Sin embargo, pasando de 16 a 32, el cambio en la aceleración es de sólo 5,26%. Esto se ve apoyado también en la tendencia de los gráficos de eficiencia (Figura 8. Representación gráfica del ratio de eficiencia ante la paralelización del algoritmo utilizando una estrategia DFS. y Figura 9. Representación gráfica del ratio de eficiencia ante la paralelización del algoritmo utilizando una estrategia BFS.), los cuales muestran una convergencia teórica al 0% ante el uso de infinitos procesadores, asumiendo una tendencia potencial decreciente en esta medida.

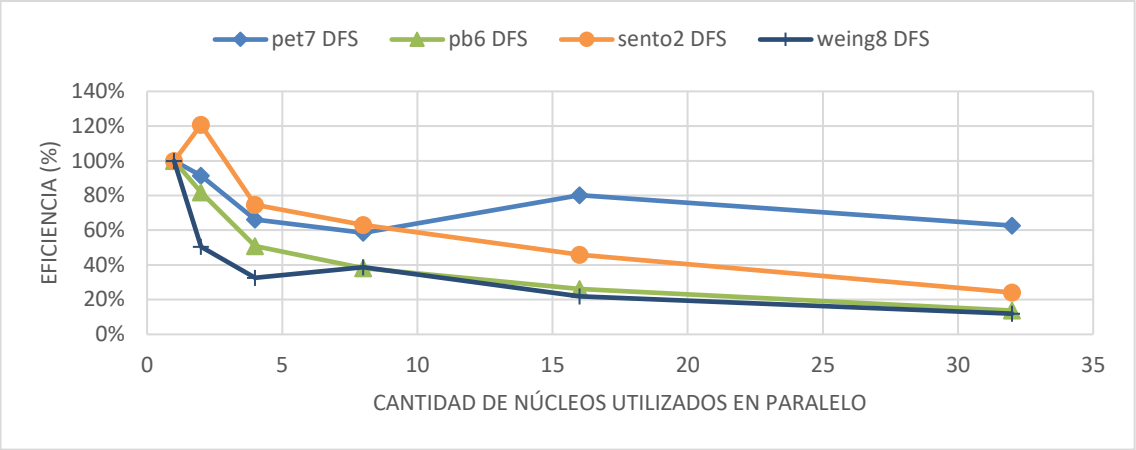

Figura 8. Representación gráfica del ratio de eficiencia ante la paralelización del algoritmo utilizando una estrategia DFS.

Tabla 10. Coeficientes de determinación del ajuste del ratio de eficiencia ante la paralelización, bajo un tendencia potencial decreciente, para los problemas resueltos mediante la estrategia DFS.

R^2	Potencial
pet7	0,494
pb6	0,978
sento2	0,878
weing8	0,913

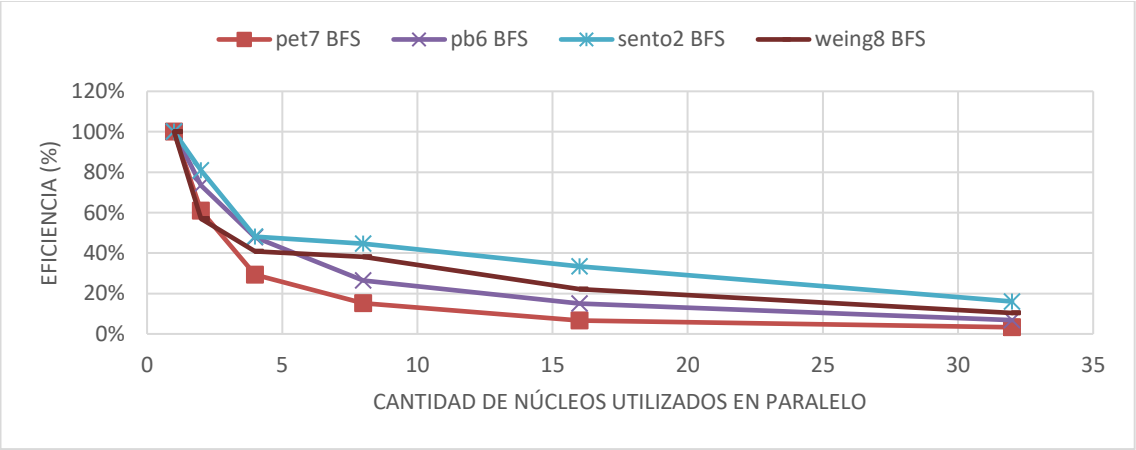

Figura 9. Representación gráfica del ratio de eficiencia ante la paralelización del algoritmo utilizando una estrategia BFS.

Tabla 11. Coeficientes de determinación del ajuste del ratio de eficiencia ante la paralelización, bajo un tendencia potencial decreciente, para los problemas resueltos mediante la estrategia BFS.

R^2	Potencial
pet7	0,996
pb6	0,979
sento2	0,943
weing8	0,946

Esta última observación también lleva a afirmar que la aceleración teórica obtenida con el algoritmo paralelizado converge en un valor máximo, dependiente del problema de optimización a resolver.

Ahora bien, a nivel práctico, se debe tener en cuenta que esto se encuentra también sujeto a los tiempos de recopilación de información de parte del algoritmo; problemas con grandes cantidades de datos implican mayores tiempos de inicialización de procesos, los cuales podrían traducirse en pérdidas de eficiencia que, en casos extremos, podrían reportar valores inferiores al 0%. Por lo tanto, en términos reales, la aceleración ya no converge; ahora posee un punto máximo que, tras ser alcanzado, comienza a presentar desaceleración, pudiendo enlentecer la ejecución el algoritmo de tal manera que incluso la resolución secuencial pudiese arrojar mejores velocidades de resolución respecto a la paralelización.

5.1.2 El caso anómalo

Tras la discusión planteada anteriormente, aún queda la cuestión referida al comportamiento anómalo del problema *pet7* en el caso BFS. Considerando la estrategia de exploración elegida, se plantea que en este problema la ruta designada por el nodo de mejor valor objetivo lleva directamente a la solución óptima, por lo que la adición de procesos para explorar paralelamente el resto de las ramas del árbol no aporta realmente a la velocidad de resolución, pudiendo incluso retrasar el tiempo total de ejecución del

algoritmo al añadir tiempos de procesamiento adicionales destinados al inicio de los procesos y a la recopilación de la información obtenida por éstos.

Analizando el conteo de nodos resueltos por el algoritmo al utilizar sólo un núcleo para llevar a cabo la resolución, se puede determinar que la ruta óptima se encontró en 50 iteraciones. Al considerar que dicho modelo cuenta con 50 variables y que sólo una se acota durante cada iteración, la hipótesis planteada anteriormente se ve fundamentada.

Se debe mencionar que, si bien este caso es excepcional, los contras obtenidos por la paralelización quedan opacados por los beneficios *posibles* en los tiempos de resolución, comenzando por el hecho de que las aceleraciones siguen siendo positivas, aunque estas sean al 1,00x. Si bien se puede pensar en el anómalo escenario de resolución de este problema como *el peor caso* para la paralelización debido a que la aceleración disminuye al aumentar la cantidad de procesos paralelos, se debe tener también en cuenta que, por lo general, no se posee conocimiento previo sobre la ruta óptima al momento de resolver un problema y que la posibilidad de encontrarse con una anomalía similar no es muy alta. Al final, se debe evaluar si se desea tomar el riesgo de aumentar en unos pocos segundos el tiempo total de resolución bajo la posibilidad de acrecentar la aceleración significativamente.

5.1.3 Las estrategias de exploración

Ambas estrategias de exploración utilizadas para la resolución de problemas haciendo uso del algoritmo presentaron tendencias similares en sus métricas. Sin embargo, tanto sus tiempos de resolución, como la cantidad de nodos resueltos y las tasas de aceleración presentan notorias diferencias. Contrario a lo propuesto por Clausen & Perregaard (1999), los resultados obtenidos plantean que la estrategia BFS posee

mayor rapidez en las resoluciones, explorando además pocos nodos, con respecto a la DFS.

Tabla 12. Cantidad de nodos resueltos durante la ejecución del algoritmo de Branch and Bound con una estrategia DFS.

Nodos	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	9.052	10.590	9.684	8.574	6.066	4.138
pb6	2.139	2.594	3.494	3.871	4.003	4.888
sento2	9.229	7.148	9.433	11.810	15.392	21.191
weing8	3.743	4.962	7.459	5.111	7.205	8.819

Tabla 13. Cantidad de nodos resueltos durante la ejecución del algoritmo de Branch and Bound con una estrategia BFS.

Nodos	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos	16 Núcleos	32 Núcleos
pet7	50	72	184	158	191	489
pb6	286	377	482	632	948	1.059
sento2	943	902	940	990	1.329	1.785
weing8	868	1.698	2.213	2.055	2.306	3.623

Como se menciona en su definición, la estrategia BFS se enfoca en “calidad por sobre cantidad” y esto se aprecia en los resultados conseguidos. En una primera instancia, podría plantearse que, al buscar el mejor valor durante cada iteración, problemas de gran cantidad de variables requieren de muchas iteraciones para poder recién encontrar una primera solución entera factible al hacer uso de una estrategia BFS, siendo ésta una posible fuente de enlentecimiento del algoritmo.

La estrategia DFS, por el otro lado, abarca grandes cantidades de nodos que pueden encontrar una solución factible rápidamente, pudiendo ésta ser o no de buena calidad. Existe una mayor tasa de incertidumbre sobre la utilidad que pudiese tener llegar a tener la solución encontrada, por lo que es de esperar que el encontrar múltiples soluciones paralelamente pudiese aumentar las posibilidades de conseguir una buena incumbente.

Éste es justamente el beneficio que se da al paralelizar la estrategia DFS que no se denota tan fuertemente en la otra estrategia analizada.

Destaca de sobremanera la diferencia entre la cantidad de nodos resueltos por ambas estrategias. Buscando soluciones en base al mejor nodo por sobre una exploración enfocada en la profundidad, la cantidad de iteraciones decrece enormemente. En el caso de menor contraste, con el problema *weing8* paralelizado con 32 núcleos, la estrategia BFS resuelve 58,92% menos nodos que su contraparte DFS. Para el caso más extremo, en la resolución serial del problema *pet7*, este porcentaje alcanza un impactante 99,45%. Al utilizar una estrategia centrada en priorizar una búsqueda en profundidad, el seleccionar una "rama equivocada" para la exploración del árbol puede traer consigo consecuencias severas en la eficiencia del algoritmo, tanto a nivel de velocidad de resolución como en uso de la memoria física del equipo computacional. Por lo mismo, el intentar abordar un problema de gran tamaño con una estrategia DFS puede terminar produciendo inestabilidad en el equipo, lo cual puede incluso llegar a imposibilitar su resolución. Lamentablemente, debido a que generalmente se desconoce la cantidad de nodos que serán explorado, no es posible establecer *a priori* si el problema producirá o no saturación en la RAM del computador donde se utilizará el algoritmo de Branch and Bound, sea o no paralelizado.

En cuanto a los cambios en la velocidad vistos dentro de cada estrategia, como se puede notar al comparar los resultados, la aceleración provista por la paralelización demuestra propensiones a un crecimiento más rápido para la exploración en profundidad que para la exploración por mejor valor. Si bien los tiempos de la BFS siguen siendo mejores, los cambios aportados por la paralelización se notan mucho más en la DFS; el ritmo al que decrecen los tiempos es mayor, lo cual sugiere que ante instancias donde se trabaja con una gran cantidad de núcleos computacionales, la

paralelización puede traer como resultado mejores tiempos en la DFS que en la BFS, tanto en los ratios de aceleración como en el aumento de la velocidad de resolución. Esto se da, además, en relación al tamaño de los problemas a ser resueltos, tanto en variables de decisión como en restricciones.

Llama la atención la discrepancia con las conclusiones obtenidas en el trabajo de Paulavičius, Žilinskas, & Grothey (2010), quienes declaran que la estrategia DFS posee mejor rendimiento que la BFS en el caso serial, debido a que esta última utiliza mucho tiempo extra en la comparación y priorización de nodos para explorar. Esto, sin embargo, no se ve en los resultados del trabajo actual; el tiempo utilizado en la predisposición de nodos es ínfimo y no afecta de manera significativa a los tiempos de resolución. Los beneficios aportados por la solución de buena calidad, sin embargo, sí presentan mejoras sustanciales en la velocidad de resolución. La causa principal de esta diferencia se puede hallar en el tamaño de los problemas resueltos y en las técnicas de acotación utilizadas. A medida de que el problema aumenta de tamaño, los tiempos de priorización comienzan a tomar importancia y la estrategia BFS puede perder ventaja frente a la DFS. Debido a que en el caso paralelo se aligera la evaluación de nodos debido a su distribución en distintos núcleos computacionales, la estrategia BFS mantiene su eficiencia de mejor manera, por lo que en el caso paralelo los resultados no difieren de aquellos obtenidos por los autores.

6. CONCLUSIONES

6.1 ¿Vale la pena paralelizar?

El uso de técnicas que permitan aprovechar al máximo la eficiencia computacional resulta un punto muy importante en el ámbito de la optimización, aún más si su implementación es relativamente simple y directa. Tras crear un algoritmo de resolución de problemas de optimización, como es el caso del Branch and Bound desarrollado, la paralelización puede realizarse directamente y sin mayores complejidades, lo cual indica que, de por sí, la paralelización de procesos demuestra mucho potencial para las ciencias decisionales. Sin embargo, para aprovechar el máximo de su potencial, es indispensable contar con un algoritmo optimizado. Para el caso del algoritmo de Branch and Bound desarrollado para este estudio, si bien fue posible realizar el estudio de diversos problemas combinatoriales, hubo algunos casos que no pudieron ser resueltos debido a dificultades computacionales, referidas especialmente al manejo de la memoria física del equipo. Implementaciones más eficientes, por tanto, deberán tener en consideración una forma de almacenamiento de información que permita soportar más nodos sin llegar a colapsar el funcionamiento del computador donde el algoritmo se está ejecutando. Debido además a los problemas presentados por el GIL de Python, surge además la recomendación de realizar el algoritmo en un lenguaje más ágil y con menos restricciones, como C++.

En cuanto a los tiempos de resolución de problemas de optimización, el procesamiento paralelo de procesos es – en la mayor parte de los casos – una fuente de mejora notoria, aún si sólo se utilizan dos núcleos computacionales paralelizados. La exploración paralela de un árbol de búsqueda, de por sí, ya se plantea como una alternativa intuitivamente útil. Si adicionalmente se da la posibilidad de interconectar

estos procesos mediante la sincronización del valor incumbente, agilizando así la poda de ramas, la alternativa se torna aún más ventajosa frente a la contraparte serial, y difícilmente se vislumbran razones para no implementarla.

Si bien existe un máximo en la cantidad de núcleos que pueden utilizarse a la vez para aprovechar al máximo la eficiencia real de la paralelización, mientras las métricas indiquen una eficiencia no-negativa, el uso de un algoritmo paralelo sobre uno secuencial es recomendado, aún si no puede conocerse de antemano la cantidad óptima de procesos paralelos que se deban utilizar. A nivel de un sólo equipo al año 2020, el máximo de núcleos encontrados en un procesador computacional es de 64, por lo que resulta complejo que la paralelización añada tiempos fuertemente influyentes a la resolución debido al inicio de procesos al utilizar el máximo de la capacidad de un equipo.

En cuanto a las estrategias de exploración, la búsqueda de la incumbente utilizando una estrategia BFS *impaciente* se presenta como la mejor alternativa, especialmente ante el uso de un algoritmo serial, aunque en el caso paralelo tampoco se queda atrás. La estrategia DFS tiende a verse más afectada por cambios en la paralelización, por lo que se plantea que, ante un uso masivo de procesos paralelos, este modo de búsqueda podría llegar a presentar resultados mejores que en la BFS. A su vez, el tamaño de los problemas puede llegar a ser influyente en esta decisión, pues estrategias como BFS pueden perder eficiencia si no consiguen encontrar soluciones en un periodo extenso de tiempo.

6.2 Limitaciones de la investigación y posibles estudios futuros

Si bien el estudio se enfocó en el desarrollo y posterior análisis del algoritmo paralelizado, se plantea la posibilidad de realizar un análisis más exhaustivo utilizando

una mayor cantidad de procesos paralelos, especialmente para estudiar más a fondo los planteamientos del valor máximo de aceleración y las diferencias que pudiesen existir en éste al realizar la resolución de problemas con distintas estrategias de exploración. Bajo lo mismo, también es importante considerar el uso de múltiples equipos computacionales para expandir la cantidad de procesos que se pueden crear paralelamente. De la misma forma, se propone como buena posibilidad el habilitar un servidor externo para adjudicarle esta función.

Además, se plantea la posibilidad de ejecutar el algoritmo con problemas de mayor tamaño, para evaluar las aceleraciones experimentadas a una mayor escala. Esto considerando la disponibilidad de un equipo de múltiples núcleos computacionales con la capacidad necesaria para resolver un problema de optimización durante el curso de varios días.

Previo a la ejecución, sin embargo, debe sortearse el mayor problema que el algoritmo tiene para enfrentar problemas de gran tamaño: la cantidad masiva de memoria RAM que puede llegar a requerir debido al almacenamiento temporal de la información de los nodos. Este inconveniente dificulta la resolución de problemas de benchmarking de mayor exigencia, debido a que la RAM tiende a saturarse completamente en medio de la ejecución del código, lo cual imposibilita la resolución completa del algoritmo. Se plantea la posibilidad de buscar métodos de compresión de la data o formas de almacenar la información de manera temporal en la memoria virtual, sin incurrir de forma tan fuerte en el uso de la memoria física.

A nivel algorítmico, la velocidad de resolución puede mejorarse en el caso de incluir técnicas que permitan acotar el problema durante una fase de *presolving*, pudiendo además contar con extensiones del algoritmo que permitan encontrar posibles soluciones iniciales haciendo uso de meta-heurísticas. Podría resultar de gran utilidad el

analizar el impacto que ambos factores pudiesen tener sobre las métricas calculadas en el presente estudio.

En cuanto a las estrategias de exploración, tras haber analizado los comportamientos de tanto la estrategia BFS como la DFS, podría resultar útil programar el funcionamiento de híbridos entre ambas, buscando aprovechar los beneficios que éstas presentan al funcionar por separado, basándose en el trabajo de Shinano, Higaki, & Hirabayashi (1995).

La resolución de problemas de benchmarking también puede expandirse, especialmente a distintos tipos de problemas de optimización. Durante el estudio de los problemas tipo, no fue posible encontrar problemas de variable entera que pudiesen ser procesados correctamente. Aquellos ofrecidos por MIPLIB 2017 – The Mixed Integer Programming Library (https://miplib.zib.de/) vienen presentados en el formato .mps, el cual no puede ser leído actualmente por la versión actual de Pyomo. La extracción de la data a un formato más amigable para el lenguaje de modelamiento tampoco fue posible debido a su poca legibilidad.

Por último, la implementación de otros solver como apoyo en la resolución aportaría a la gama de problemas que el algoritmo de Branch and Bound paralelo puede abarcar. Actualmente Gurobi no cuenta con las herramientas necesarias para resolver problemas no lineales, por lo que la ejecución utilizando solvers como BARON (https://minlp.com/baron) o IPOPT (https://github.com/coin-or/Ipopt) puede resultar de gran utilidad.

7. REFERENCIAS

Akrotirianakis, I., Rustem, B., & Maros, I. (2000). An Outer Approximation based Branch and Cut Algorithm for convex 0-1 MINLP problems. *Optimization Methods and Software, 16*(1), 21-47. doi:10.1080/10556780108805827

Allouche, D., Givry, S., Katsirelos, G., Schiex, T., & Zytnicki, M. (2015). Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In G. Pesant (Ed.), *International Conference on Principles and Practice of Constraint Programming* (pp. 12-29). Cham: Springer. doi:10.1007/978-3-319-23219-5_2

Anand, R., Aggarwal, D., & Chahar, V. (2017). A Comparative Analysis of Optimization Solvers. doi:10.1080/09720510.2017.1395182

Archibald, B., Maier, P., McCreesh, C., Stewart, R., & Trinder, P. (2018). Replicable parallel branch and bound search. *Journal of Parallel and Distributed Computing, 113*, 92-114. doi:10.1016/j.jpdc.2017.10.010.

Barr, R. S., & Hickman, B. L. (1993). Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions. *INFORMS J. Comput., 5*, 2-18. doi:10.1287/ijoc.5.1.2

Beasley, J. E. (2008). Multidimensional Knapsack Problems. (C. A. Floudas, & P. M. Pardalos, Eds.) *Encyclopedia of Optimization*, 2402-2406. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-74759-0_412

Bonyadi, M. R., & Michalewicz, Z. (2016). Evolutionary Computation for Real-World Problems. (S. Matwin, & J. Mielniczuk, Eds.) *Challenges in Computational Statistics and Data Mining. Studies in Computational Intelligence, 605*. doi:https://doi.org/10.1007/978-3-319-18781-5_1

Bonyadi, M. R., Michalewicz, Z., & Barone, L. (2013). The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, 1037-1044. doi:10.1109/CEC.2013.6557681

Chandra Jaiswal, U., Singh, A., Maurya, O., & Kumar, A. (2011). Unified Knapsack Problem. (V. Das, J. Stephen, & Y. Chaba, Eds.) *Computer Networks and Information Technologies. CNC 2011. Communications in Computer and Information Science, 142*, 325-330. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-3-642-19542-6_60

Clausen, J., & Perregaard, M. (1999, Enero). On the best search strategy in parallel branch-and-bound:Best-First Search versus Lazy Depth-First Search [En la mejor estrategia de búsqueda en branch-and-bound paralelo: Búsqueda Best-First versus búsqueda Depth-First perezosa]. *Annals of Operations Research, 90*, 1-17. doi:https://doi-org.usm.idm.oclc.org/10.1023/A:1018952429396

Corrêa, R., & Ferreira, A. (1995). A Distributed Implementation of Asynchronous Parallel Branch and Bound. (A. Ferreira, & J. Rolim, Eds.) *Parallel Algorithms for Irregular Problems: State of the Art*, 157-176. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-1-4757-6130-6_8

Corrêa, R., & Ferreira, A. (2005). Parallel best-first branch- and-bound in discrete optimization: A framework. (A. Ferreira, & P. Pardalos, Eds.) *Solving Combinatorial Optimization Problems in Parallel*, 171-200. doi:https://doi-org.usm.idm.oclc.org/10.1007/BFb0027122

Costanzo, A., Baduel, L., Caromel, D., & Matsuoka, S. (2007). Grid'BnB: A Parallel Branch & Bound Framework for Grids. In P. M. Aluru S. (Ed.), *High Performance Computing – HiPC 2007* (pp. 566-579). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-77220-0_51

Crainic, T. G., Le Cun, B., & Roucairol, C. (2006). Parallel Branch-and-Bound Algorithms [Algoritmos de Branch-and-Bound Paralelo]. (E.-G. Talbi, Ed.) *Parallel Combinatorial Optimization*, 1-28. doi:10.1002/0470053925

Dalcín, L. (2013). *MPI for Python*. Retrieved from https://mpi4py.readthedocs.io/en/stable/

Dalcin, L. (2019). *mpi4py*, 3.0.3. Retrieved Abril 18, 2020, from https://bitbucket.org/mpi4py/mpi4py/src

Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources, 34*(9), 1124-1139. doi:https://doi.org/10.1016/j.advwatres.2011.04.013.

Dalcín, L., Paz, R., & Storti, M. (2005). MPI for Python. *Journal of Parallel and Distributed Computing, 65*(9), 1108-1115. doi:10.1016/j.jpdc.2005.03.010

Dalcín, L., Paz, R., & Storti, M. (2005). MPI for Python. *Journal of Parallel and Distributed Computing, 65*(9), 1108-1115. doi:https://doi.org/10.1016/j.jpdc.2005.03.010

Dalcín, L., Paz, R., Storti, M., & D'Elía, J. (2008). MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing, 68*(5), 655-662. doi:10.1016/j.jpdc.2007.09.005

Doig, A., & Land, A. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica, 28*(3), 497-520. doi:10.2307/1910129

Drake, J. H. (2015). Benchmark instances for the Multidimensional Knapsack Problem. doi:10.13140/2.1.3578.9122

Dubois, P. F. (1999). *Pyfort -- The Python-Fortran connection tool*. Retrieved from http://pyfortran.sourceforge.net/

Gendron, B., & Crainic, T. G. (1994). Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research, 42*(6), 1042-1066. Retrieved from www.jstor.org/stable/171985

González, G. (2010). SISTEMA DE VISUALIZACIÓN Y EVALUACIÓN DEL DESEMPEÑO DE ALGORITMOS BRANCH AND BOUND PARA PROBLEMAS MIP. Valparaíso: Universidad Técnica Federico Santa María. Retrieved from http://hdl.handle.net/11673/1817

Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI* (Tercera ed.). Cambridge, Massachusetts: The MIT Press. Retrieved from http://www.hds.bme.hu/~fhegedus/00%20-%20Numerics/B2015%20Using%20MPI%20-%20Portable%20Parallel%20Programming%20with%20the%20Message-Passing%20Interface.pdf

Gurobi Optimization. (2019). *MIPGap*. Retrieved Abril 20, 2020, from Documentation: https://www.gurobi.com/documentation/9.0/refman/mipgap2.html

Hackebeil, G. (2019, Marzo 12). *Welcome to PyBnB*. Retrieved from PyBnB: https://pybnb.readthedocs.io/en/stable/index.html

Hart, W. E., Laird, C. D., Watson, J. P., Woorduff, D. L., Hackebeil, G. A., Nicholson, B. L., & Siirola, J. D. (2017). *Pyomo - Optimization Modeling in Python* (Segunda ed., Vol. 67). Springer.

Hart, W. E., Watson, J. P., & Woodruff, D. L. (2011). Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation, 3*, 219-260. doi:10.1007/s12532-011-0026-8

He, M., Tian, W., & Guo, W. (2018). On the Classification of NP Complete Problems and Their Duality Feature. *International Journal of Computer Science & Information Technology, 10*(1). doi:10.5121/ijcsit.2018.10106

Herrera, J. F., Salmerón, J. M., Hendrix, E. M., Asenjo, R., & Casado, L. G. (2017). On parallel Branch and Bound frameworks for Global Optimization. *Journal of Global Optimization, 69*(3), 547-560. doi:10.1007/s10898-017-0508-y

Hoffman, K., & Ralphs, T. (2013). Integer and Combinatorial Optimization. In S. Gass, & M. Fu (Eds.), *Encyclopedia of Operations Research and Management Science.* Boston, MA: Springer.

Hunt, J. (2019). Multiprocessing. In J. Hunt, *Advanced Guide to Python 3 Programming* (pp. 363-376). Springer, Cham. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-3-030-25943-3_31

Hunt, J. (2019). Threading. In J. Hunt, *Advanced Guide to Python 3 Programming [Guía avanzada de programación en Python 3]* (pp. 347-361). Springer, Cham. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-3-030-25943-3_30

IBM Corporation. (2019). *IBM ILOG CPLEX Optimization Studio V12.9.0 documentation*. Retrieved Abril 20, 2020, from IBM Knowledge Center: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/EpGap.html

Karp, R. M. (1971). Reducibility Among Combinatorial Problems. *Proceedings of a symposium on the Complexity of Computer Computations.* Nueva York. doi:10.1007/978-3-540-68279-0_8

Kellerer, H. (2008). Knapsack. (M.-Y. Kao, Ed.) *Encyclopedia of Algorithms*, 419-421. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-30162-4_192

Kellerer, H. (2016). Knapsack. (M.-Y. Kao, Ed.) *Encyclopedia of Algorithms*, 1048-1051. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-1-4939-2864-4_192

Kianfar, K. (2011). Branch-and-Bound Algorithms. *Wiley Encyclopedia of Operations Research and Management Science*. doi:10.1002/9780470400531.eorms0116

Korte, B., & Vygen, J. (2008). NP-Completeness. In B. Korte, & J. Vygen, *Combinatorial Optimization. Algorithms and Combinatorics* (Cuarta ed., Vol. 21, pp. 359-392). Berlín, Heidelberg: Springer. doi:10.1007/978-3-540-71844-4

Korte, B., & Vygen, J. (2008). The Knapsack Problem. In B. Korte, & J. Vygen, *Combinatorial Optimization* (Cuarta ed., Vol. 21, pp. 439-448). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-71844-4

Lai, T.-H., & Sahni, S. (2002). Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM, 27*(6), 594-602. doi:10.1145/358080.358103

Laporte, G. (1992). The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research, 59*(3), 345-358. doi:https://doi.org/10.1016/0377-2217(92)90192-C

Laursen, P. S. (1992). Simple approaches to parallel Branch and Bound. *Parallel Computing, 19*(2), 143-152. doi:10.1016/0167-8191(93)90044-L

Lee, E. K., & Mitchell, J. E. (2009). Integer Programming: Branch and Bound Methods. (C. A. Floudas, & P. M. Pardalos, Eds.) *Encyclopedia of Optimization*, 1634-1643. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-74759-0_286

Liong, C. -Y., Wan, I., & Omar, K. (2008). Vehicle routing problem: Models and solutions. *Journal of Quality Measurement and Analysis*(4), 205-218. Retrieved from https://www.researchgate.net/publication/313005083_Vehicle_routing_problem_Models_and_solutions

Manne, A. S., & Markowitz, H. M. (1957). On the Solution of Discrete Programming Problems. *Econometrica, 25*(1), 84-110. doi:10.2307/1907744

Marinakis, Y. (2009). Heuristic and Metaheuristic Algorithms for the Traveling Salesman Problem. (C. A. Floudas, & P. M. Pardalos, Eds.) *Encyclopedia of Optimization*, 1499-1506. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-74759-0_262

Marowka, A. (2018). On parallel software engineering education using python. *Education and Information Technologies, 23*, 357-372. doi:https://doi-org.usm.idm.oclc.org/10.1007/s10639-017-9607-0

Martello, S., & Toth, P. (1990). *Knapsack Problems.* New York: Wiley. Retrieved from www.or.deis.unibo.it/knapsack.html

Meng, F., Chu, D., Li, K., & Zhou, X. (2019). Multiple-class multidimensional knapsack optimisation problem and its solution approaches. *Knowledge-Based Systems, 166*, 1-17. doi:10.1016/j.knosys.2018.11.006

Miller, P. (2004). *pyMPI: Putting the Py in MPI*. Retrieved from http://pympi.sourceforge.net/

Mitchell, J. E. (2008). Integer Programming: Branch and Cut Algorithms. In C. A. Floudas, & P. M. Pardalos (Eds.), *Encyclopedia of Optimization* (Vol. I, pp. 1643-1649). Boston, MA: Springer. doi:10.1007/978-0-387-74759-0_287

Mittelmann, H. D. (2017, Octubre 23). Latest Benchmarks of Optimization Software. *INFORMS Annual Meeting 2017.* Houston, TX. Retrieved from http://plato.asu.edu/talks/informs2017.pdf

Mittelmann, H. D. (2020). Benchmarking Optimization Software - a (Hi)Story. *SN Operations Research Forum*(1). doi:https://doi-org.usm.idm.oclc.org/10.1007/s43069-020-0002-0

Nielsen, O. (2013). *PyPar*. Retrieved from https://github.com/daleroberts/pypar

Paulavičius, R., Žilinskas, J., & Grothey, A. (2010). Investigation of selection strategies in branch and bound algorithm with simplicial partitions and combination of Lipschitz bounds. *Optimization Letters*, 173-183. doi:https://doi-org.usm.idm.oclc.org/10.1007/s11590-009-0156-3

Potvin, J. (2008). Vehicle Routing. (C. Floudas, & P. Pardalos, Eds.) *Encyclopedia of Optimization*, 4019-4022. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-74759-0

Przybylek, M. R., Wierzbicki, A., & Michalewicz, Z. (2018, Septiembre). Decomposition Algorithms for a Multi-hard Problem. *Evolutionary Computation, 26*(3), 507-533. doi:https://doi.org/10.1162/evco_a_00211

Punnen, A. P. (2017). The Traveling Salesman Problem: Applications, Formulations and Variations. (G. Gutin, & A. P. Punnen, Eds.) *Combinatorial Optimization, 12*, 1-4. doi:https://doi-org.usm.idm.oclc.org/10.1007/0-306-48213-4_1

Python Software Foundation. (2020, Mayo 18). *multiprocessing — Process-based parallelism*. Retrieved Mayo 18, 2020, from Documentation: https://docs.python.org/3/library/multiprocessing.html

Python Software Foundation. (2020, Mayo 18). *threading — Thread-based parallelism*. Retrieved Mayo 18, 2020, from Documentation: https://docs.python.org/3/library/threading.html

Python Software Foundation. (2020, Mayo 18). *time — Time access and conversions*. Retrieved from https://docs.python.org/3/library/time.html

Python Wiki. (2019, Septiembre 9). *ParallelProcessing*. Retrieved Abril 18, 2020, from https://wiki.python.org/moin/ParallelProcessing

Robson, J. M. (1992). Parallel Algorithms for NP-Complete Problems. (R. Baeza-Yates, & U. Manber, Eds.) *Computer Science*, 379-382. doi:https://doi.org/10.1007/978-1-4615-3422-8_31

Savelsbergh, M. W. (2008). Branch and Price: Integer Programming with Column Generation. In C. A. Floudas, & P. M. Pardalos (Eds.), *Encyclopedia of Optimization* (Vol. I, pp. 328-331). Boston, MA: Springer. doi:10.1007/978-0-387-74759-0_58

Šeda, M. (2007). Mathematical Models of Flow Shop and Job Shop Scheduling Problems. *World Academy of Science, Engineering and Technology, International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering, 1*, 307-312. Retrieved from https://www.semanticscholar.org/paper/Mathematical-Models-of-Flow-Shop-and-Job-Shop-Seda/6cbda0bdddb573f58fde2fdd78558c275419782c

Shinano, Y., Higaki, M., & HIrabayashi, R. (1995, Diciembre). A Selection Rule for Parallel Branch and Bound Algorithms. *Transactions of the Society of Instrument and Control Engineers, 32*, 1379-1387. doi:10.9746/sicetr1965.32.1379

Smirnov, S., & Voloshinov, V. (2017). Implementation of Concurrent Parallelization of Branch-and-bound algorithm in Everest Distributed Environment. *Procedia Computer Science, 119*, 83-89. doi:10.1016/j.procs.2017.11.163.

Solk, J., Mann, I., Mohais, A., & Michalewicz, Z. (2013). Combining vehicle routing and packing for optimal delivery schedules of water tanks. *OR Insight*(26), 167-190. doi:https://doi.org/10.1057/ori.2013.1

Taslaman, N. S. (2013). *Exponential-time algorithms and complexity of NP-hard graph problems.* Tesis doctoral, IT-Universitetet i København, Section of Theoretical Computer Science, Copenhagen. Retrieved from https://pure.itu.dk/portal/files/39516792/nina_thesis.pdf

The Free Software Foundation. (2008, Noviembre 3). *21.4 Processor And CPU Time*. Retrieved from The GNU C Library: https://www.gnu.org/software/libc/manual/html_node/Processor-And-CPU-Time.html#Processor-And-CPU-Time

Tian, W., Guo, W., & He, M. (2018). On the Classification of NP Complete Problems and Their Duality Feature. *International Journal of Computer Science and Information Technology*(10), 67-78. doi:10.5121/ijcsit.2018.10106

Wang, S.-J., Tsai, C.-W., & Chiang, M.-C. (2018). A High Performance Search Algorithm for Job-Shop Scheduling Problem. *Procedia Computer Science, 141*, 119-126. doi:https://doi.org/10.1016/j.procs.2018.10.157.

Wilfahrt, R., & Kim, S. (2008). Traveling Salesman Problem (TSP). (S. Shekhar, & H. Xiong, Eds.) *Encyclopedia of GIS*, 1173-1176. doi:https://doi-org.usm.idm.oclc.org/10.1007/978-0-387-35973-1_1406

Woeginger, G. J. (2003). Exact algorithms for NP-hard Problems: A Survey. In J. M., R. G., & R. G. (Eds.), *Combinatorial Optimization — Eureka, You Shrink!* (Vol. 2570, pp. 185-207). Berlín, Heidelberg: Springer. doi:10.1007/3-540-36478-1_17