Introducción

En la constante búsqueda de aprovechar al máximo los recursos disponibles, ya sea en términos de materiales, tiempo o esfuerzo, se ha vuelto esencial idear estrategias para evitar desaprovechar valiosos recursos. La complejidad de las tareas y problemas diarios ha llevado al desarrollo de propuestas y técnicas que buscan optimizar y refinar los procesos, minimizando así la pérdida de recursos y maximizando la eficiencia.

Al abordar la tarea de optimización, surge un nuevo desafío: la necesidad de encontrar soluciones rápidas y efectivas sin comprometer la exhaustividad del análisis. Aquí es donde entran en juego las técnicas de búsqueda de mejores opciones, como el backtracking, que permiten explorar de manera sistemática el espacio de soluciones en busca de la mejor alternativa. Sin embargo, al aplicar estas técnicas, surge otro dilema: el consumo considerable de tiempo y memoria de cálculo. Para superar esta limitación, se introduce el algoritmo Branch and Bound, una estrategia diseñada para optimizar la búsqueda de soluciones mediante la aplicación de criterios inteligentes y acotamientos que permiten descartar ramas no prometedoras.

En este informe, nos centramos, dentro de ese contexto, la aplicación del algoritmo Branch and Bound al clásico problema de la mochila. A través de la exploración de nodos, criterios de selección y acotamientos, demostraremos cómo esta técnica, aun pudiendo mejorar significativamente la eficiencia en la búsqueda, puede entregar resultados dispares al aplicar distintos enfoques y criterios.

Descripción del Modelo Escogido

En el contexto de la programación entera, el problema de la mochila se presenta como una tarea clásica de optimización combinatoria. Se origina en la década de 1890 y su formulación matemática típicamente implica la maximización de la función objetivo, que representa la suma de los valores de los elementos seleccionados, sujeta a una restricción de capacidad basada en los pesos de dichos elementos. Esta problemática, que encuentra aplicaciones en la gestión de recursos, logística y planificación, se clasifica dentro de los problemas NP-hard debido a su complejidad computacional. La clasificación NP-Hard implica que no se conoce un algoritmo eficiente (polinómico) para encontrar la solución óptima de manera generalizada para todos los tamaños de problemas. Según lo que menciona Pisinger, D. (1995), no conocemos más técnicas de solución exacta que una enumeración (posiblemente completa) del espacio de soluciones.

De todas formas, se puede ahorrar algo de esfuerzo usando alguna de las siguientes técnicas:

- Programación dinámica
- Relajación del espacio de estados
- Preprocesamiento
- Branch-and-bound

En el presente documento nos enfocaremos en la última de estas técnicas. El algoritmo exacto Branch and Bound (B&B) realiza una exploración exhaustiva del espacio de soluciones, utilizando criterios de selección inteligentes y acotamientos para optimizar la búsqueda y evitar la exploración de ramas no prometedoras. En cuanto a la formulación matemática, podemos considerar a la forma más simple el problema de la mochila binario:

$$\text{Maximizar: } \sum_{j=1}^{n} p_j x_j$$

$$\text{Sujeto a: } \sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0,1\},\ j = 1,\ldots,n,$$

En todas las variantes, se establecen algunos parámetros que mantienen su significado, por ejemplo, un beneficio ‘p’ y un peso ‘w’, asociado a cada elemento ‘j’ y una capacidad ‘c’. Para este caso particular asumimos como valores enteros positivos o de otra forma estaríamos hablando de otras variantes, además queda notar que para cada elemento en alguna posición ‘j’, como mucho, solo es posible seleccionarlo una vez, de allí el término binario. A partir de aquí podemos hablar de la forma general del problema de la mochila multidimensional o con múltiples restricciones:

$$\text{Maximizar: } \sum_{j=1}^{n} p_j x_j$$

$$\text{Sujeto a: } \sum_{j=1}^{n} w_{ij} x_j \leq c_i,\ i = 1,\ldots,m,$$

$$x_j \geq 0\, entero,\ j = 1,\ldots,n,$$

En este caso, además de introducir distintas restricciones ‘c’, hablamos de la posibilidad de escoger un elemento ‘x’ más de una vez para una posición ‘j’. Basándonos en estas dos formulaciones, podemos introducir el modelo que se va a desarrollar:

$$\text{Maximizar: } \sum_{j=1}^{n} p_j x_j$$

$$\text{Sujeto a: } \sum_{j=1}^{n} w_{ij} x_j \leq c_i,\ i = 1, \ldots, m,$$

$$x_j \in \{0, 1\},\ j = 1, \ldots, n,$$

Hablamos de una elección binaria para cada elemento ‘x’ en alguna posición ‘j’, pero a su vez, sujeto a más de una restricción ‘c’. Desde aquí podemos imaginar aplicaciones simples y prácticas, como la necesidad de transportar productos, maximizando el valor final, limitados por la capacidad y el peso de carga del vehículo.

Desarrollo

a) Investigación sobre el algoritmo exacto Branch and Bound

La metodología del Branch and Bound se aplica a problemas donde se busca maximizar o minimizar una función objetivo, sujeta a ciertas restricciones. A medida que explora el espacio de soluciones, el algoritmo utiliza criterios de corte y acotamiento para evitar la exploración de áreas que no conducen a soluciones óptimas. Su enfoque se basa en la división del espacio de soluciones en subconjuntos más pequeños y manejables, lo que permite explorar de manera eficiente las diferentes posibilidades y encontrar la mejor solución.

El proceso comienza con la formulación de un nodo raíz que representa el espacio de soluciones completo. Luego, el algoritmo utiliza estrategias para dividir este espacio en subconjuntos más pequeños llamados nodos hijos. Estos nodos hijos son evaluados, y aquellos que no tienen el potencial de superar la mejor solución conocida actualmente son descartados (criterio de corte).

Branch and Bound utiliza una estructura de datos como una cola o pila para gestionar los nodos y determinar el orden en el que se exploran. Dependiendo de la aplicación, se pueden emplear diferentes criterios de selección de nodos, como LIFO (Last In, First Out) o FIFO (First In, First Out).

b) Investigación del problema seleccionado

Para abordar el problema de la mochila multidimensional mediante el algoritmo Branch and Bound, se deben tener en cuenta diferentes aspectos para lograr una implementación efectiva. Se destacan algunos de los puntos clave:

- Elección de una estructura de datos para contener nodos: Una estructura completa facilita la gestión y exploración de nodos, permitiendo una mejor utilización de los recursos.
- Ordenamiento inicial de los elementos: El orden en que se exploran los elementos puede afectar significativamente el rendimiento del algoritmo.
- Implementación de un algoritmo de acotamientos y poda: Es esencial para reducir la búsqueda en el espacio de soluciones. Esto evita la exploración innecesaria de ramas que no conducen a soluciones óptimas
- Exploración de Nodos: La estrategia de exploración de nodos determina cómo se recorre el espacio de búsqueda, por lo que resulta fundamental establecer cuidadosamente su elección.

c) Implementar el modelo y resolverlo directamente a través del solver seleccionado

La implementación se ha realizado en Python, y para contrastar y validar los resultados obtenidos, se ha utilizado la interfaz lpSolveAPI de R. Esta elección se basa en la capacidad de 'lp_solve' para resolver modelos de programación entera. Esto proporciona un punto fiable de comparación para evaluar la eficacia de nuestro modelo. Típicamente, la resolución de un problema de programación lineal orientado al problema de la mochila, se vería así:

```
library(lpSolveAPI)
N_CONSTRAINTS = 2
N_DVARS = 9

# Create model
lprec <- make.lp(0, N_DVARS)

# Set objective function
set.objfn(lprec, c(40, 42, 25, 12, 75, 60, 50, 35, 20))

# Set objective (min or max)
lp.control(lprec, sense="max")

# Set constraints
add.constraint(lprec, c(4, 7, 5, 3, 2, 5, 1, 3, 6), "<=", 15)
add.constraint(lprec, c(3, 5, 3, 2, 1, 3, 5, 5, 3), "<=", 14)

# Set only binary solutions
set.type(lprec, c(1,2,3,4,5,6,7,8,9), "binary")

# Solver
solve(lprec)

# Get solution
get.objective(lprec)
# Get decision variables
get.variables(lprec)
```

Se genera el modelo, se establece la función objetivo, lo que se busca en ella, las restricciones y el tipo de solución esperado. En cuanto a la implementación en Python, es posible listar las acciones del núcleo del algoritmo:

- Creación del Nodo Raíz:

Se crea el nodo raíz con índice 0, límite superior (upper bound), valor acumulado 0, restricciones acumuladas vacías y sin elementos seleccionados.

- Bucle Principal:

Mientras existan nodos en la estructura Tree, se ejecuta el bucle principal.

- Obtención del Nodo Actual:

Se retira un nodo de la estructura Tree para procesarlo. Este nodo es el candidato actual para la exploración.

- Comprobación del Mejor Resultado:

Se verifica si el valor acumulado del nodo actual es mayor que el mejor resultado conocido. Si es así, se actualiza el mejor resultado y se guarda el nodo actual como el mejor nodo.

- Obtención del Índice del Artículo Actual:

Se obtiene el índice del artículo actual que se está considerando en el nodo actual.

- Criterios de Acotamiento:

Se implementan criterios de acotamiento específicos del algoritmo B&B para decidir si el nodo actual puede proporcionar soluciones mejores. Si no es el caso, se imprime un mensaje y se omite la exploración de este nodo.

- Determinación de los Límites para el Próximo Artículo:

Se calcula la relación valor/peso del próximo artículo para determinar los límites para la siguiente iteración.

- Exploración sin Elegir el Artículo (X_i = 0):

Se crea un nuevo nodo considerando la opción de no elegir el artículo actual (X_i = 0). Se calcula el nuevo valor acumulado, las nuevas restricciones y el nuevo límite superior. Si las restricciones son aceptables, se agrega este nuevo nodo a la estructura Tree.

- Exploración Eligiendo el Artículo (X_i = 1):

Se crea otro nuevo nodo considerando la opción de elegir el artículo actual (X_i = 1). Se calcula el nuevo valor acumulado, las nuevas restricciones y el nuevo límite superior. Si las restricciones son aceptables, se agrega este nuevo nodo a la estructura Tree. Si no, se imprime un mensaje indicando que no se agrega el nodo debido a restricciones incumplidas.

Este proceso continúa hasta que no haya más nodos en la estructura Tree, y el bucle principal se haya completado. En cada iteración, se exploran diferentes opciones y se actualizan los mejores resultados según los criterios establecidos. La estructura Tree actúa como una cola o pila, determinando si se exploran nodos de manera prioritaria en profundidad (LIFO) o amplitud (FIFO).

d) Proponer e implementar una estructura de datos para guardar los nodos explorados en el B&B

A nuestro criterio, la estructura creada debe contener lo siguiente para lograr un proceso de exploración integral a través de los nodos:

```
class Node:
    def __init__(self, index, bound, value, a_c, items):
        self.item_index = index             # Item index (or current level)
        self.upper_bound = bound            # The current upper bound
        self.accumulated_value = value      # Accumulated value
        self.accumulated_constraints = a_c  # Accumulated constraints
```

```
        self.items = items                    # {0,1} if node is taken or not
```

- *item_index*: Indica el índice o nivel actual en el árbol de decisiones, representando el elemento actual que se está considerando.
- *upper_bound*: Representa el límite superior del nodo, calculado mediante la función de cota superior. Este umbral es esencial para evaluar la viabilidad de un nodo y decidir si explorar sus subnodos respectivos.
- *accumulated_value*: Almacena la suma acumulativa de los valores de los elementos seleccionados hasta el nodo actual.
- *accumulated_constraints*: Guarda la acumulación de las restricciones hasta el nodo actual.
- *items*: Una lista binaria que indica qué elementos han sido seleccionados (1) o no (0) hasta el nodo actual.

e) Proponer e implementar dos criterios para seleccionar las variables a ramificar

```
def sort_by_value(item):
    return item.value

def sort_by_ratio(item):
    return item.value / sum(item.constraints)
```

Los dos criterios definidos, *sort_by_value* y *sort_by_ratio*, se utilizan para ordenar los elementos antes de comenzar la exploración de nodos.

- Criterio por Valor (*sort_by_value*): Este criterio ordena los elementos en función de su valor, de mayor a menor, en principio se escoge este criterio porque finalmente es lo que

se busca: maximizar el beneficio total. Al priorizar los elementos con valores más altos, se busca alcanzar la solución que aporte la mayor ganancia posible.

- Criterio por Ratio (*sort_by_ratio*): Este criterio ordena los elementos en función de la proporción entre su valor y la suma de sus restricciones. Este enfoque busca dar preferencia a elementos que tienen una alta relación valor-restricción. La teoría es que tiene la capacidad de llenar la mochila de manera eficiente y maximizar el valor total.

f) Implementar criterios de acotamiento propios del algoritmo B&B

El "*upper_bound*" es una estimación superior del valor máximo que puede alcanzar la solución óptima de un nodo en el espacio de búsqueda. En el contexto del problema de la mochila, representa el beneficio acumulado máximo que podría obtenerse al explorar ese nodo específico. Para entenderlo mejor, imaginemos que cada nodo en el árbol de búsqueda representa una posible solución parcial, donde se han seleccionado algunos elementos de la mochila. El "*upper_bound*" es como una cota superior que nos indica el máximo beneficio que podríamos obtener si continuamos explorando desde ese nodo. En cuanto a la implementación:

```
# Check if the current node CAN'T provide better solutions
if len(Tree.nodes) > 0:
    if all(current_node.upper_bound < existing_node.upper_bound for existing_node in Tree.nodes):
        continue
```

En cada iteración, obtenemos a partir de la lista de nodos pendientes por explorar el actual nodo, definido en el código como *current_node*. Si el "*upper_bound*" del nodo actual es inferior al "*upper_bound*" de todos los nodos existentes (obtenidos a partir de la clase *Tree* y su atributo *nodes*), quiere decir que el nodo actual no tiene el potencial de proporcionar una solución mejor que la que ya hemos encontrado, y, por lo tanto, se omite su exploración.

g) Seleccionar e implementar una metodología de exploración de nodos (LIFO, FIFO, etc.)

En la implementación, se han seleccionado dos metodologías de exploración de nodos: LIFO (Last In, First Out) y FIFO (First In, First Out). La estructura LIFO, o pila, sigue el principio de que el último elemento agregado es el primero en ser eliminado. Esto significa que se explorarán primero los nodos más recientes.

```
# Implementar LIFO (Pila)
# Breadth-first: Prioriza amplitud
class LIFO():
    def __init__(self):
        self.nodes = []

    def add(self, node):
        self.nodes.append(node)

    def remove(self):
        if self.nodes:
            return self.nodes.pop()
        else:
            print("La estructura está vacía.")
```

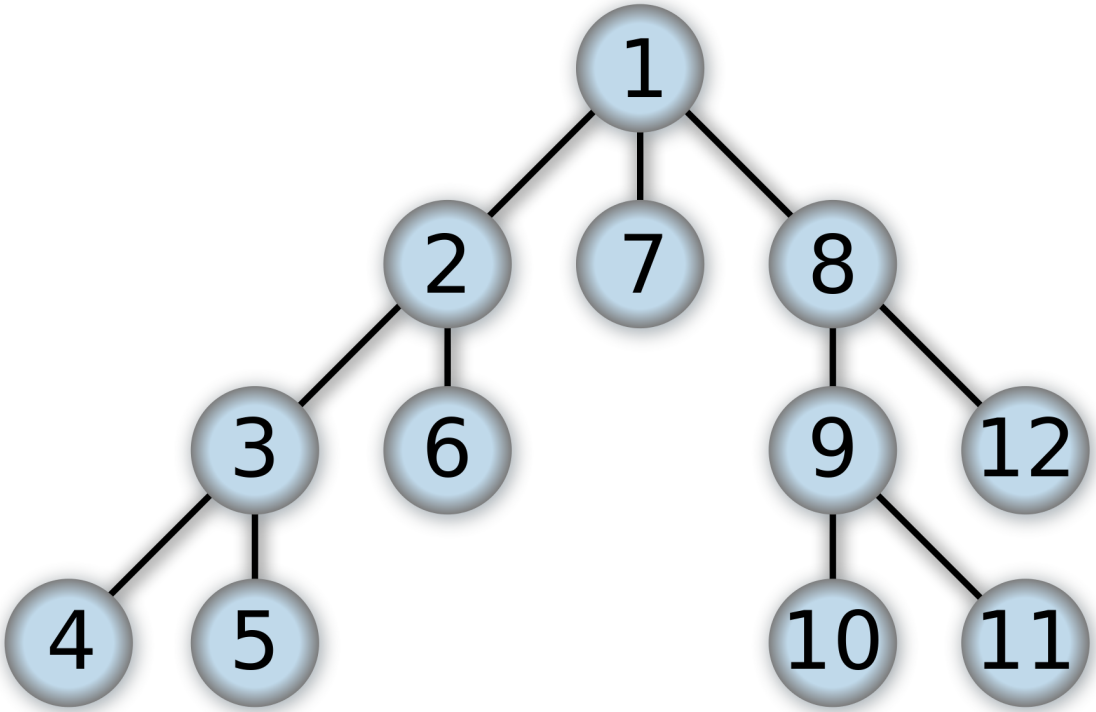
Por otro lado, la estructura FIFO, o cola, sigue el principio de que el primer elemento agregado es el primero en ser eliminado. En este caso, se exploran primero los nodos que se han añadido hace más tiempo.

```python
# Implementar FIFO (Cola)
# Depth-first: Prioriza profundidad
class FIFO():
    def __init__(self):
        self.nodes = []

    def add(self, node):
        self.nodes.append(node)

    def remove(self):
        if self.nodes:
            return self.nodes.pop(0)
        else:
            print("La estructura está vacía.")
```

A raíz de estas estructuras, cabe destacar las metodologías aplicadas: Depth-first y Breadth-first, dos estrategias fundamentales utilizadas en algoritmos de búsqueda, especialmente en árboles y grafos.

Depth-first:

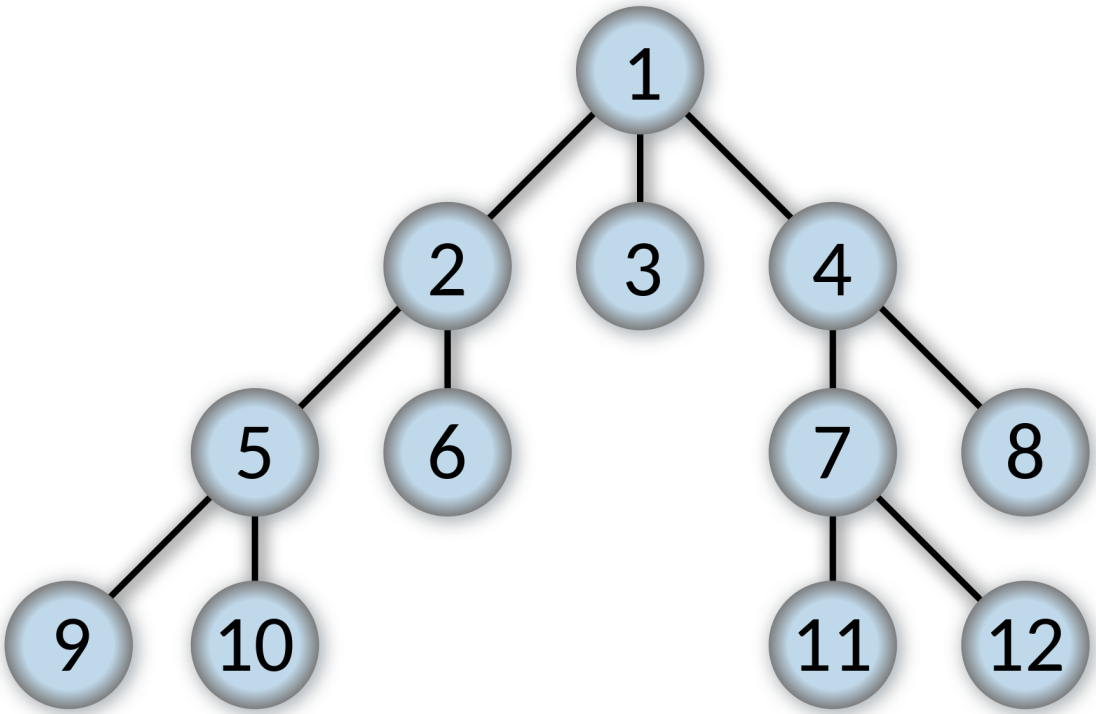
- Principio básico: Explorar lo más profundo posible antes de retroceder.
- Funcionamiento: El algoritmo avanza hacia abajo en una rama hasta que llega al final antes de retroceder y explorar otra rama.
- Ventajas: Es eficiente en términos de memoria, ya que solo necesita almacenar los nodos de la rama actual.
- Desventajas: Puede quedar atrapado en ramas profundas antes de explorar otras áreas.



Alexander Drichel (2007, 10 de enero). Depth-first tree.svg. Wikimedia Commons, la enciclopedia libre. https://commons.wikimedia.org/wiki/File:Depth-first-tree.svg

Breadth-first:

- Principio básico: Explorar niveles gradualmente desde el nodo raíz.
- Funcionamiento: El algoritmo explora todos los nodos en el mismo nivel antes de pasar al siguiente nivel.
- Ventajas: Garantiza encontrar la solución más cercana al nodo raíz y evita quedarse atrapado en ramas profundas.
- Desventajas: Puede requerir más memoria, ya que debe almacenar nodos de diferentes niveles simultáneamente.



Alexander Drichel (2005, 4 de octubre). Breadth-first tree.svg. Wikimedia Commons, la enciclopedia libre. https://commons.wikimedia.org/wiki/File:Breadth-first-tree.svg

Aplicación del modelo a un Caso de Estudio, utilizando datos simulados

Para el caso de estudio, consideremos una situación práctica (como la mencionada en un comienzo) que involucra la optimización de recursos mediante el problema de la mochila multidimensional. Se trata de una empresa de logística que debe planificar el transporte de mercancías desde un almacén central hacia varias sucursales, dando prioridad a los que provean un mayor beneficio. Dentro de este escenario, cada artículo tiene diferentes pesos y volúmenes. Es el vehículo quien determina las restricciones, como la capacidad máxima de carga y el espacio disponible. Los datos simulados involucran 20 artículos con sus respectivos beneficios, pesos y volúmenes:

Producto	Beneficio	Peso (kg)	Volumen (m^3)
1	150	4	3
2	60	1	1
3	200	6	5
4	90	2	2
5	120	3	3
6	180	5	4
7	80	2	1
8	160	4	4
9	110	3	2
10	70	2	1
11	130	3	3
12	190	5	4
13	100	2	2
14	140	3	3
15	50	1	1
16	120	3	2
17	90	2	2
18	170	4	4
19	200	5	4
20	60	2	1

Limitados a **16 kg** y **14 m³** en total. La solución entregada por medio de la biblioteca lpSolveAPI de R, entrega un resultado de $680 como beneficio, desglosados en la siguiente tabla:

R, lpSolveAPI			
Producto	**Beneficio**	**Peso (kg)**	**Volumen (m^3)**
2	60	1	1
4	90	2	2

7	80	2	1
13	100	2	2
14	140	3	3
15	50	1	1
16	120	3	2
17	90	2	2
Total	**730**	**16**	**14**

En cuanto a la implementación realizada utilizando la estructura FIFO:

Python, FIFO + sort_by_ratio (170 nodos visitados)			
Producto	**Beneficio**	**Peso (kg)**	**Volumen (m^3)**
2	60	1	1
4	90	2	2
7	80	2	1
13	100	2	2
14	140	3	3
15	50	1	1
16	120	3	2
17	90	2	2
Total	**730**	**16**	**14**

Python, FIFO + sort_by_value(170 nodos visitados)			
Producto	**Beneficio**	**Peso (kg)**	**Volumen (m^3)**
2	60	1	1
12	190	5	4
13	100	2	2
14	140	3	3
19	200	5	4
Total	**690**	**16**	**14**

Por último, los resultados correspondientes a la implementación usando la estructura LIFO en sus dos variantes:

Python, LIFO + sort_by_ratio (958 nodos visitados)			
Producto	**Beneficio**	**Peso (kg)**	**Volumen (m^3)**
2	60	1	1
4	90	2	2
7	80	2	1
13	100	2	2
14	140	3	3
15	50	1	1
16	120	3	2
17	90	2	2
Total	**730**	**16**	**14**

Python, LIFO + sort_by_value (25464 nodos visitados)			
Producto	**Beneficio**	**Peso (kg)**	**Volumen (m^3)**
2	60	1	1
7	80	2	1
13	100	2	2
14	140	3	3
15	50	1	1
16	120	3	2
18	170	4	4
Total	**720**	**16**	**14**

Análisis de resultados

Resulta fácil notar a raíz de los resultados y con una simple inspección visual que el algoritmo FIFO otorga mucho mejores resultados, al menos en cuanto a la eficiencia. Es posible observar que la estrategia FIFO, al priorizar la amplitud en la exploración del espacio de búsqueda, logra obtener soluciones óptimas con un menor número de nodos explorados, ya que se enfoca en examinar diversas opciones desde el principio. Por otro lado, la estrategia LIFO, que prioriza la profundidad en la exploración, puede estar atrapado en ramas no prometedoras.

	FIFO		LIFO	
	sort_by_ratio	**sort_by_value**	**sort_by_ratio**	**sort_by_value**
Nodos explorados	102	170	958	25464
Diferencia en resultado	0	40	10	40

Aunque tampoco podemos hablar de una regla general, al variar los datos de manera aleatoria, se pueden apreciar algunas diferencias:

	FIFO		LIFO	
	sort_by_ratio	sort_by_value	sort_by_ratio	sort_by_value
Nodos explorados	5472	2020	437	11262
Diferencia en resultado	0	10	0	0

Existen algunas variaciones en los nodos explorados, pero continúa una tendencia, donde podríamos reafirmar que la elección de un ordenamiento inicial impacta de manera crítica en la eficiencia del algoritmo.

Finalmente, no todo es eficiencia cuando los resultados no son óptimos, en este caso, se han obtenido respuestas variadas, destacando el algoritmo FIFO usando un ordenamiento inicial según el ratio entre el beneficio y la suma de restricciones, al seleccionar nodos de manera eficiente y enfocarse en áreas prometedoras del espacio de búsqueda, logrando un equilibrio efectivo entre la calidad de la solución y la eficiencia en la exploración.

Conclusiones

El método de branch and bound es un algoritmo potente para la búsqueda de soluciones óptimas enteras, en el contexto de problema de la mochila dimensional. Sin embargo, esta eficiencia puede ser afectada por su estrategia de exploración, los criterios de selección. Así también podemos destacar que para este caso, la elección de un ordenamiento FIFO, reduce la cantidad de nodos explorados, dado que prioriza la amplitud en la exploración del espacio de búsqueda.

De la misma manera, esto implica el añadir un nuevo enfoque al buscar una solución óptima. Por tanto, la utilización de un ordenamiento inicial FIFO o LIFO, afecta a la eficiencia en tiempo y en la calidad de la solución.

Bibliografía

[1] Bettinelli, A., Cacchiani, V., & Malaguti, E. (2017). A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, *29*(3), 457-473.

[2] Boyer, V., El Baz, D., & Elkihel, M. (2010). Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound. *European Journal of Industrial Engineering*, *4*(4), 434-449.

[3] Pisinger, D. (1995). Algorithms for knapsack problems. *Ph. D thesis, DIKU, University of Copenhagen, Repore 95/1*.

[4] Puchinger, J., Raidl, G. R., & Pferschy, U. (2010). The multidimensional knapsack problem: Structure and algorithms. *INFORMS Journal on Computing*, *22*(2), 250-265.