SELECCIÓN DINÁMICA DE LA REGIÓN DE BÚSQUEDA A TRAVÉS DE TÉCNICAS HILL-CLIMBING PARA PROBLEMAS DE OPTIMIZACIÓN CONTINUOS.

ALEJANDRO MOROSO

Tesis para optar al título de Ingeniero Civil en Informática y Telecomunicaciones

Profesor guía: Víctor Reyes

FACULTAD DE INGENIERÍA Y CIENCIAS
ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Santiago, Chile
2023

SELECCIÓN DINÁMICA DE LA REGIÓN DE BÚSQUEDA A TRAVÉS DE TÉCNICAS HILL-CLIMBING PARA PROBLEMAS DE OPTIMIZACIÓN CONTINUOS.

ALEJANDRO MOROSO

Tesis para optar al título de Ingeniero Civil en Informática y Telecomunicaciones

Profesor guía: Víctor Reyes
Comité: Martín Gutiérrez
Comité: Nicolás Hidalgo

FACULTAD DE INGENIERÍA Y CIENCIAS
ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Santiago, Chile
2023

Alejandro Moroso
alejandro.morosoxd@gmail.com

Agradecimientos

Agradecimiento a mi profesor guía Víctor Reyes y al PROYECTO FONDECYT INICIACIÓN N° 11230225

Resumen

Este proyecto está enfocado en el uso de la técnica *Branch and Bound* para resolver problemas de optimización no lineales y continuos (NCOP), donde se utiliza la técnica de *upper bounding* con el método Abstaylor para encontrar nuevas soluciones mediante la construcción de un politopo convexo interior en el punto medio del dominio en el que se esté trabajando. Los algoritmos metaheurísticos buscan encontrar un equilibrio entre el tiempo de CPU y la calidad de la solución para la resolución de problemas de optimización. El objetivo propuesto es diseñar e integrar a la técnica de Abstaylor, un algoritmo metaheurístico de trayectoria, especificamente *Hill-Climbing*, para la selección dinámica del punto de expansión del politopo, para que este se construya en áreas prometedoras de los distintos dominios. Además, se realizarán pruebas de rendimiento con benchmarks de la comunidad de optimización, para asi conocer el impacto de la implementación del *Hill-Climbing* y analizar los resultados obtenidos.

Abstract

This project is focused on the use of the Branch and Bound technique to solve non-linear and continuous optimization problems (NCOP). The Upper bounding technique is used with the Abstaylor method to find new solutions by constructing a convex polytope interior at the midpoint of the domain being worked on. Additionally, metaheuristic algorithms are employed to strike a balance between CPU time and solution quality in optimization problem-solving. The proposed objective is to design and integrate a specific trajectory metaheuristic algorithm, namely Hill-Climbing, into the Abstaylor technique for dynamic selection of the polytope expansion point. This aims to construct the polytope in promising areas of the different domains. Furthermore, performance tests will be conducted using optimization community benchmarks to assess the impact of Hill-Climbing implementation and analyze the obtained results.

Contenido

Lista de tablas

Lista de figuras

Capítulo 1

Introducción

En este informe se abordarán los problemas de optimización no lineal y continuos (NCOP), principalmente la resolución de estos problemas con el algoritmo de *Branch and Bound* basado en intervalos. Se presentará una solución para mejorar la eficiencia de esta técnica basada en el algoritmo de *Hill-Climbing*, la que contempla distintos enfoques que serán implementados en C++. Los problemas NCOP son problemas computacionales complejos que requieren soluciones eficientes o aproximadas para ser resueltos en tiempos aceptables. El algoritmo *Branch and Bound* es una técnica o estrategia que encuentra el óptimo global a través de un árbol de búsqueda que reduce el espacio de búsqueda del problema a medida que crece este. El algoritmo *Hill-Climbing* es una metaheurística de trayectoria que se basa en mejorar iterativamente una solución hasta encontrar una de calidad.

Este documento está estructurado de la siguiente manera: En el Capítulo 1 se presenta el contexto, motivación y objetivos de esta investigación. En el Capítulo 2 se revisan los conceptos básicos sobre *Branch and Bound* basado en intervalos, algoritmos de búsqueda local, metaheurísticas, la librería Ibexopt y el estado del arte. En el Capítulo 3 se presentará la metodología del trabajo, donde se explica como se desarrolla esta investigación. En el Capítulo 4 se describe la principal contribución de este trabajo, en él presentándose el diseño de la solución junto a las distintas variaciones de esta, con el fin de tener una gran cantidad de datos para una mejor comparativa. En el Capítulo 5 se presentará cómo se realizarán las pruebas de rendimiento,

benchmarks utilizados y los resultados de cada algoritmo, además de un análisis de estos. En el Capítulo 6 se concluye y proponen líneas de investigación y desarrollo futuro.

1.1. Contextualización y motivación

Los problemas de optimización NCOP se presentan en numerosos campos de estudio, incluyendo robótica [5], la economía [2] y la ciencia de datos [8]. Estos problemas implican encontrar la mejor solución para una función objetivo, sujeta a restricciones, donde la función objetivo y las restricciones pueden ser no lineales y continuas. La mejor solución depende de diversas variables que influyen en sí, un resultado es el óptimo o no. A medida que el número de variables va aumentando, el problema resulta ser intratable mientras se utilizan algoritmos exactos. Por eso, el enfoque en la investigación de este campo ha sido encontrar algoritmos y técnicas que proporcionen el óptimo global o el óptimo deseado en un tiempo razonable. Uno de estos algoritmos es *Branch and Bound* basado en intervalos, que es una estrategia utilizada principalmente para resolver sistemas de restricciones continuas y para manejar problemas de optimización global con restricciones, con un enfoque que se basa en dividir el espacio de búsqueda en subproblemas más pequeños y explorar selectivamente solo aquellos subproblemas que tienen el potencial de contener la solución óptima. Esta estrategia es matemáticamente rigurosa al utilizar intervalos, lo que proporciona dos grandes ventajas sobre otras técnicas:

- Manejan de manera más efectiva que otros métodos numéricos cuando se trata de la precisión de los parámetros de punto flotante. Debido a que los intervalos pueden representar números de punto flotante con incertidumbres, se pueden manejar más fácilmente los errores de redondeo de la máquina.
- Son capaces de manejar el espacio de búsqueda completo, garantizando encontrar el óptimo global, siendo este el punto donde la función alcanza su valor mínimo o máximo de todo su rango de valores posibles.

La idea fundamental del *Branch and Bound* es construir un árbol de búsqueda en el que cada nodo representa un subproblema. En cada etapa del algoritmo, se selecciona un nodo activo para ser explorado. Luego, se generan y agregan nodos hijos al árbol mediante una estrategia de ramificación

que divide el subproblema original en subproblemas más pequeños. Esta ramificación se realiza de manera selectiva, descartando subproblemas que se pueden determinar como no prometedores.

También existen las metaheurísticas, que emplean dos conceptos claves para la resolución de problemas: explotación y exploración. El primero intensifica la búsqueda en un vecindario para una región determinada y el segundo busca zonas prometedoras del espacio de búsqueda. De esta manera, las metaheurísticas tienen el objetivo de encontrar un *"trade off"* entre lo que es el tiempo de CPU (sigla en inglés de Central Processing Unit) y la calidad de soluciones que ellas obtienen.

Adicionalmente, los algoritmos metaheurísticos se pueden clasificar en diversas categorías, según la manera que exploran y explotan el espacio de búsqueda. Algunas a mencionar son las de población y las de trayectoria. Estos últimos algoritmos se basan en la búsqueda local y la exploración del espacio de búsqueda mediante movimientos y cambios en una solución inicial.

Un ejemplo de estos es el algoritmo *Hill-Climbing*, que se inspira en el proceso de subir una colina para encontrar la cima más alta. Este comienza con una solución inicial y explora las soluciones vecinas, realizando pequeñas modificaciones. Si se encuentra una solución vecina que es mejor que la solución actual, esta toma como la nueva solución actual y se repite el proceso hasta alcanzar un óptimo local, es decir, una solución en la que no se pueden realizar cambios para mejorarla.

Otro modo de tratar los problemas NCOP, son los métodos de aproximación, estas técnicas son utilizadas en matemáticas y ciencias de la computación para obtener soluciones cercanas o aproximadas a problemas que no puede resolverse exactamente de manera analítica o en tiempo razonable.

Las aproximaciones se pueden realizar con distintos criterios, como minimizar el error, satisfacer ciertas restricciones establecidas o simplificar funciones complejas de manera eficiente. Esto último es especialmente útil cuando se desea obtener una aproximación que se ajuste a la función original dentro de un intervalo pequeño o dominio de interés.

Actualmente, se han obtenido buenos resultados utilizando métodos de aproximación basados en la aproximación de la función mediante la forma de Taylor para resolver los subproblemas que genera el árbol del algoritmo de *Branch and Bound*. Algunos de estos son XTaylor y AbsTaylor. Estos métodos se centran en aprovechar las propiedades de los polinomios de Taylor para obtener una aproximación local precisa de una función alrededor de

un punto específico para obtener resultados satisfactorios. A pesar de que el método AbsTaylor a diferencia de XTaylor, tiene libertad para la selección de este punto, este está definido en el punto medio del espacio de búsqueda, perdiendo grandes oportunidades de mejora.
Sin embargo, existen algoritmos metaheurísticos que, aunque no garantizan encontrar la solución óptima global, son capaces de proporcionar soluciones de alta calidad en espacios de búsqueda amplios en tiempos razonables. Por lo tanto, se pueden implementar algoritmos metaheurísticos para mejorar los métodos de aproximación que ocurren dentro del algoritmo de *Branch and Bound*. Específicamente, una selección dinámica del punto de expansión mediante el algoritmo *Hill-Climbing* para el politopo de aproximación del método Abstaylor que ocurren dentro de los nodos del árbol de búsqueda del algoritmo de *Branch and Bound* realizando esta implementación en la librería Ibexopt escrita en C++.

1.2. Objetivos

Implementación del algoritmo metaheurístico de trayectoria *Hill-Climbing*, para la selección dinámica del punto de expansión del politopo de Abstaylor dentro del algoritmo de *Branch and Bound* basado en intervalos para la resolución de problemas NCOP.

1.3. Objetivos específicos

- Adaptar y configurar el algoritmo para trabajar en dominios continuos e intervalos.
- Realizar la implementación para la obtención de datos dentro de la librería Ibex.
- Comparar el rendimiento entre las implementaciones realizadas y el algoritmo original.

1.4. Solución preliminar

Se busca una implementación de la metaheurística *Hill-Climbing* con la hipótesis de que esta le otorgue buenos puntos de expansión al método de

Abstaylor para los distintos dominios que se encuentre en el árbol, teniendo como resultados la obtención de soluciones óptimas de calidad en fases tempranas del árbol, provocando reducciones del espacio de búsqueda del problema y siendo así una menor cantidad de nodos creados por el árbol de búsqueda o una disminución en el tiempo de procesamiento del problema.

Para la estructura del *Hill-Climbing* se planea distintas versiones de esta, es decir, se diseñó distintos algoritmos de *Hill-Climbing* donde tendrán pequeñas diferencias en su estructura que les otorga una mayor rigurosidad al momento de trabajar un punto de expansión de calidad. Pero a la vez consumirán una mayor cantidad de recursos de procesamiento, ya que tendrán que realizar un mayor número de iteraciones o realizar un mayor número de cálculos, por lo que el objetivo de todo esto es tener una mayor cantidad de datos y conocer qué tantos recursos se pueden utilizar en la búsqueda de un punto de expansión sin que afecte de manera negativa el resultado final.

La implementación se realizará en la librería Ibex, que se utiliza para resolver problemas de optimización global de funciones sujetas a restricciones no lineales, además de contener ya las implementaciones de *Branch and Bound* basado en intervalos y el método de Abstaylor con la selección del punto de expansión en el medio del espacio de búsqueda y de manera aleatoria, que serán la base comparativa de esta investigación.

1.5. Resultados esperados

- Diseño e implementación del algoritmo *Hill-Climbing* dentro de la libreria ibex en C++ para una selección dinámica del punto de expansión del método Abstaylor.
- Establecer una comparativa entre el algoritmo de *Branch and Bound* con la implementación de la solución propuesta y el algoritmo original, con el fin de dar un primer acercamiento a esta investigación y conocer los resultados de eficiencia al tener una selección dinámica del punto de expansión.

Capítulo 2

Estado del arte y marco teórico

En este capítulo, se describirán las principales contribuciones y soluciones que actualmente están funcionando. Además de presentar los conceptos básicos del estado del arte y marco teórico.

2.1. Intervalos

En matemáticas, un intervalo es un conjunto de números reales que están comprendidos entre dos valores extremos. Estos pueden ser incluidos o excluidos del intervalo, lo que determina si son intervalos cerrados o abiertos. Los intervalos se representan mediante una notación de la forma $[a, b]$ o (a, b), donde a y b son los valores extremos.
La aritmética de intervalos define la extensión de los operadores elementales clásicos (por ejemplo, +, -, *, /, potencia, exp, log, sin, cos, ...). Por ejemplo, [a, b] + [c, d] = [a + b, c + d], log([a, b]) = [log(a), log(b)], etc. Es importante tener en cuenta que al realizar operaciones con intervalos, el resultado es un rango de posibles valores. No se obtiene un único valor específico. Esto se debe a que los intervalos contienen una gama de valores y las operaciones se realizan considerando todas las combinaciones posibles dentro de ese rango. También existe la operación del producto cartesiano de intervalos que genera un conjunto de pares ordenados que representa todas las combinaciones posibles de elementos de los intervalos de la operación, en cuál a este resultado se le conoce como caja o *box*.

Una función, $f : R^n \rightarrow R$ es factorizable, si se puede calcular en un número finito de pasos simples utilizando operaciones matemáticas elementales. Al extender una función factorizable a intervalos, estamos habilitando la evaluación de la función sobre intervalos en lugar de valores puntuales. Esto significa que cada componente de la función original se evalúa utilizando operaciones de intervalo, lo que da como resultado un intervalo que representa todas las posibles combinaciones de valores para las variables involucradas.

La extensión a intervalos es útil en el análisis de intervalos y métodos numéricos, especialmente cuando se abordan problemas no lineales. Al trabajar con intervalos, en lugar de valores precisos, podemos manejar la incertidumbre y propagación de errores asociados con los cálculos numéricos. La imagen óptima de la función es el intervalo más ajustado que contiene todos los posibles valores que la función puede tomar sobre el intervalo dado.

La extensión natural de una función a intervalos, denotada como $f_N(x)$, se obtiene aplicando operadores de intervalo correspondientes a las operaciones clásicas elementales a cada componente de la función original. De esta manera, obtenemos un intervalo que representa todas las posibles combinaciones de valores para las variables involucradas en la función. Por ejemplo, dada la siguiente función f_N definida de la siguiente manera:

$$f_N(x) = x_1 + (x_2 \cdot x_3)^2 + 4 \tag{2.1}$$

Si se consideran los siguientes dominios de variables, $x_1 = [1, 2]$, $x_2 = [2, 4]$ y $x_3 = [-1, 1]$, se puede calcular la imagen a través de la extensión natural; es decir, $[1,2]+([2,4]*[-1,1])^2+4 = [5,22]$. Notar que existen problemas cuando existen múltiples apariciones de las variables, debido a que los operadores de intervalo consideran cada aparición de una variable como una aparición independiente. Por ejemplo, la extensión a intervalos de la expresión $x_1 - x_1$ genera una sobre-estimación, ya que $[1,2]-[1,2] = [1,2]+[-2,-1] = [-1,1]$, por lo que, $x_1 - x_1 \neq [0,0]$. La razón es que los operadores de intervalo consideran cada aparición de una variable como una aparición independiente. Por lo tanto, $x_1 - x_1$ es equivalente a $x_1 - x_2$ cuando los intervalos relacionados con x_1 y x_2 son equivalentes, por lo cual se han desarrollado extensiones basadas en monotonicidad y de Taylor para reducir este problema.

2.2. Problemas NCOP

Los problemas NCOP (Nonlinear Continuous Optimization Problems) son problemas de optimización en los cuales se busca minimizar o maximizar una función objetivo no lineal, sujeta a un conjunto de restricciones también no lineales. Estos problemas se caracterizan por diversas dificultades, como una gran cantidad de óptimos locales, la falta de convergencia hacia un óptimo global y un espacio de búsqueda amplio y complejo debido a su naturaleza no lineal. Como resultado, se requiere una gran cantidad de recursos computacionales para alcanzar resultados óptimos. Estos están definidos de la siguiente manera:

$$\min_{x \in X} f(x) \quad \text{s.t.} \quad g(x) \leq 0, \tag{2.2}$$

En donde x corresponde a un conjunto de variables, f una función objetivo que se quiera minimizar, g un conjunto de restricciones y X la caja o *box*.
Una técnica prometedora para resolver problemas NCOP es la optimización basada en intervalos. En esta técnica, la función objetivo y las restricciones se definen en términos de intervalos, en lugar de valores numéricos precisos. El uso de intervalos permite el cálculo de encierros o cotas que brindan información valiosa para el proceso de búsqueda, lo que se aprovecha en algoritmos como el *Branch & Bound* basado en intervalos.

2.3. Branch & bound

Branch and Bound es una técnica que generalmente se usa para resolver problemas de optimización. Esta técnica se basa en la subdivisión de un espacio de búsqueda en subespacios más pequeños y manejables, utilizando una estrategia de árbol de búsqueda para explorar el espacio de solución de manera sistemática y eficiente.
En el proceso de búsqueda de soluciones mediante la técnica de *Branch and Bound* basada en intervalos, en resumen, el algoritmo parte de un nodo raíz y selecciona nodos activos de manera heurística. Luego, al nodo seleccionado se le divide el dominio seleccionando una variable y aplicando un método de bisección para crear dos nodos hijos y tener más nodos activos. Se trabaja cada nodo realizando filtrado y *upper bounding* para obtener soluciones factibles y límites superiores. El proceso se repite hasta encontrar el óptimo global. [1]

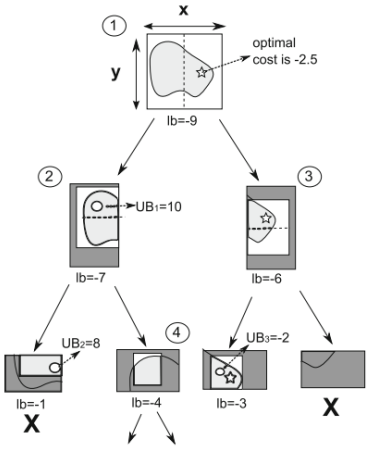

Figura 2.1. Ejemplo de árbol de búsqueda para problema NCOP de 2 variables

Observando la Figura 2.1, este algoritmo comienza con la creación de un nodo raíz, que representa el dominio inicial. A continuación, se selecciona un nodo activo de la lista de nodos pendientes, la selección de este puede ser mediante una heurística que suelen basarse en reglas generales, intuición o información del problema. Una vez seleccionado el nodo que al principio solo se puede escoger el nodo raíz. El siguiente paso es la selección de una variable para dividir su dominio. Esto puede hacerse de varias maneras, como seleccionar la variable con el mayor ancho de intervalo o la que tiene el mayor impacto en la función objetivo. Una vez que se ha seleccionado la variable, se aplica un proceso de bisección para dividir su dominio y poder crear dos nodos hijos. Estos nodos se agregan a la lista de nodos pendientes, una vez aplicado la bisección se empieza a trabajar el nodo, primero se realiza el filtrado con el objetivo de eliminar las soluciones infactibles, es decir soluciones que no satisfacen las restricciones del problema, perteneciente al dominio del nodo y finalmente, se realiza el *upper bounding*. Este proceso utiliza distintos métodos para obtener soluciones factibles del nodo y la obtención de un límite superior (UB) para el valor óptimo. Cabe mencionar que cada vez que se crea un nodo se le calcula un lower bound (lb), que es el valor mínimo que se puede obtener de la función objetivo dentro del dominio del nodo, con el objetivo de poder descartar ese nodo si su *lb* es mayor que el mejor *UB* encontrado, este proceso se repite hasta que se encuentre el óptimo global, explorando y dividiendo el dominio de manera selectiva, realizando filtrado, actualizando los límites superiores y descartando nodos del árbol. El objetivo final es encontrar la mejor solución posible dentro del

espacio de búsqueda dado.

2.3.1. Bisección

La bisección es una técnica que permite dividir el espacio de búsqueda en subregiones más pequeñas (ramas) que se exploran de forma recursiva hasta encontrar el óptimo global o determinar que no existe.
En el caso de la bisección, se divide la región de búsqueda en dos subregiones iguales, seleccionando una de las variables y dividiendo su dominio por la mitad. Para ello se utilizan heurísticas. Si la solución óptima se encuentra en una de las subregiones, se descarta la otra subregión y se sigue explorando la subregión con la solución óptima. Si no se puede determinar en qué subregión se encuentra la solución óptima, ambas subregiones se exploran de forma recursiva utilizando la misma técnica de bisección.
Existen otras formas de dividir la región de búsqueda, como la de búsqueda dorada [12], que se basa en la idea de que la función objetivo es unimodal, es decir, que tiene un único máximo o mínimo. En lugar de dividir el intervalo en dos partes iguales, se divide en una proporción fija (generalmente 0.618) que maximiza la eficiencia del método.

2.3.2. Filtrado

El procedimiento de filtrado consiste en eliminar valores inconsistentes de los dominios de las variables, es decir, valores que no satisfacen una o más restricciones. Para lograr esta tarea, se utilizan uno o varios contractores. En particular, los contractores basados en relajación lineal suelen trabajar en una relajación lineal de todo el sistema o de una parte del mismo. Generan regiones convexas que se utilizan para mejorar los límites de los dominios de las variables. Solo usando planos, se puede generar una región poliédrica convexa exterior, que puede ser representada por un sistema lineal $Ax = b$. Una vez que se genera la relajación, el objetivo es contraer los dominios de las variables del problema original. Los límites inferior y superior de los dominios de las variables se pueden encontrar minimizando y maximizando cada variable del sistema lineal.

2.3.3. Upper bounding

El *upper bounding* es una técnica utilizada en algoritmos de optimización que consiste en estimar una cota superior para la función objetivo del pro-

blema. Esta técnica es importante, ya que cada vez que se encuentra una solución factible x', se genera la restricción $f(x) < \text{UB}$, donde $\text{UB} = f(x')$, esta restricción implica una reducción del espacio factible al descartar nodos con lb mayores que el mejor UB encontrado hasta el momento, siendo lb el valor mínimo que puede alcanzar la función objetivo en el dominio contenido dentro de ese nodo, por lo que encontrar buenas soluciones factibles en fases tempranas del árbol de búsqueda provocará una reducción del espacio de búsqueda, mejorando la eficiencia en cantidad de cajas generadas y tiempo del algoritmo.
Existen diversas estrategias para encontrar soluciones factibles dentro de los dominios de los nodos del árbol de búsqueda, dependiendo del tipo de problema y las características de la función objetivo y las restricciones. Algunas de las más comunes son:

- Métodos de aproximación: en muchos casos, la función objetivo o restricciones son demasiado complejas para ser evaluadas directamente. En estos casos, se utilizan métodos de aproximación que permiten obtener una función más simple que pueda ser evaluada más fácilmente. Por ejemplo, si se tiene una función no lineal, se puede aproximar por una función lineal en una región pequeña alrededor de un punto, y utilizar esa aproximación como cota superior en esa región.

- Algoritmos metaheurísticos: en algunos casos, el cálculo exacto de la cota superior puede ser demasiado complejo o costoso computacionalmente. En estos casos, se pueden utilizar métodos metaheurísticos para obtener una aproximación de la cota superior de manera más eficiente. Por ejemplo, se puede utilizar el algoritmo *Particle Swarm* que se inspira en el comportamiento de los enjambres de animales para encontrar soluciones factibles en el espacio de búsqueda dado y evaluar la función objetivo en estas soluciones para obtener una aproximación de la cota superior.

- Búsqueda local: la búsqueda local es una técnica que consiste en explorar el vecindario de una solución inicial y seleccionar la mejor solución localmente. Se puede utilizar en el proceso de *upper bounding* al considerar que la evaluación de la función objetivo es costosa. Por ejemplo, se puede utilizar un algoritmo de búsqueda local para encontrar una solución cercana a la solución óptima, y luego evaluar la función objetivo en esa solución para obtener una cota superior más precisa. También

se pueden combinar varios algoritmos de búsqueda local para explorar diferentes vecindarios y mejorar la calidad de la solución encontrada.

2.4. Métodos de aproximación

Los métodos de aproximación son técnicas que se utilizan para obtener una aproximación de una función, cuando no es posible evaluarla de manera exacta. Estos métodos se basan en la idea de que una función complicada puede ser aproximada por otra función más simple, pero que se ajusta lo suficientemente bien a la original en un rango de valores. De esta forma, se puede utilizar la función aproximada en lugar de la original para realizar cálculos o tomar decisiones, además, dado que es una aproximación interior, la solución encontrada también es solución del problema NCOP.
Existen varios métodos de aproximación, entre ellos el método de aproximación de Taylor. Este se basa en la expansión de la función alrededor de un punto de interés utilizando su serie de Taylor. La aproximación resultante es una función polinómica que se ajusta a la función original en el punto de expansión y sus alrededores. El grado del polinomio determina la precisión de la aproximación.
Existen dos variantes principales del método de aproximación de Taylor: Abstaylor y Xtaylor. Ambos se basan en la expansión de la función en serie de Taylor, pero difieren en como el punto de expansión es seleccionado [10]. El método XTaylor extrae regiones internas utilizando una forma de Taylor de intervalo de primer orden del vector de funciones g. La forma de Taylor de intervalo garantiza que para todo $x \in \mathbf{x}$ y $j \in \{1, ..., m\}$:

$$g_j(x) \in g_j(x') + \sum_{i}^{n} J_{ij} \cdot (x_i - x'_i) \tag{2.3}$$

Aquí, J representa la matriz Jacobiana de intervalos g en la caja $\mathbf{x}$, y cada elemento J_{ij} en la matriz corresponde a una sobreestimación de la imagen de la derivada parcial $\dfrac{\partial g_j}{\partial x_i}(x)$ sobre $\mathbf{x}$.

Debido a que J_{ij} es un intervalo, la aproximación no es convexa, a menos que el punto de expansión sea una esquina de la caja $\mathbf{x}$. El método XTaylor propone usar una esquina de la caja $x^c \in \mathbf{x}$ como el punto de expansión, y la aproximación lineal se convierte en:

$$g_j(x) \leq h_j(x) = g_j(x^c) + \sum_{i}^{n} J'_{ij} \cdot (x_i - x_i^c)$$

Donde, J'_{ij} es igual al límite superior de J_{ij}, si x_i^c es igual al límite inferior de x, de lo contrario J'_{ij} es igual al límite inferior de J_{ij}.

Sin embargo, para abordar la limitación de XTaylor en la selección del punto de expansión, se presenta el método Abstaylor. Abstaylor permite que cualquier punto dentro del intervalo sea el punto de expansión. El método descompone las derivadas parciales del intervalo en dos valores: $J_{ij} = c_{ij} + d_{ij}$ donde el valor de cij representa el punto medio de la derivada del intervalo y $d_{ij} = J_{ij} - c_{ij}$

Luego, considerando la relación con la ecuación 2.3, se propone la siguiente linealización interna basada en Taylor con valores absolutos:

$$g_j(x) \leq h_j(x) = g_j(x') + \sum_{i}^{n} (c_{ij} \cdot (x_i - x'_i) + |d_{ij}| \cdot |x_i - x'_i|) \tag{2.5}$$

En este enfoque, $x' \in R^n$ puede ser cualquier punto dentro de la caja $\mathbf{x}$, como por ejemplo, el punto medio.

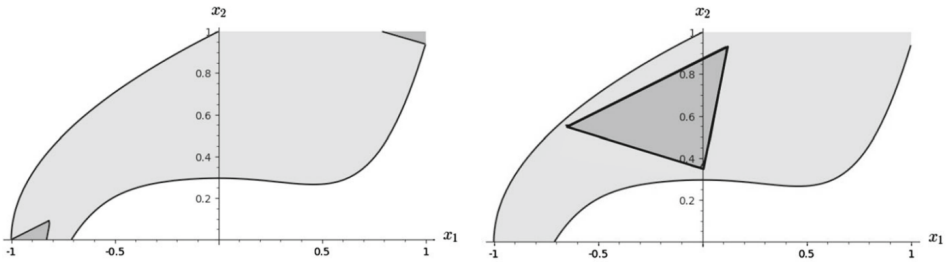

Figura 2.2. Aproximación de la función mediante el uso de la forma de Taylor.(izquierda) polígonos interiores generados por XTaylor(gris oscuro) dentro de la región factible.(derecha) polígono interior generado por Abstaylor usando el punto medio dentro de la región factible

La Figura 2.2 en el lado izquierdo se muestra en gris claro la región factible relacionada con la restricción original $g(x)$ <0 y en gris oscuro las regiones internas convexas generadas por XTaylor. Mientras que la imagen de la derecha es el politopo obtenido a través de la aproximación de AbsTaylor usando el punto medio, es importante señalar que, en este caso, el politopo obtenido es mucho más grande que el generado por XTaylor.

2.5. Algoritmos de búsqueda local

Los algoritmos de búsqueda local son técnicas heurísticas que se utilizan para encontrar soluciones aproximadas a problemas de optimización. Estos algoritmos son iterativos y trabajan a partir de una solución inicial, la cual se va mejorando en cada iteración mediante la exploración de vecindades de la solución actual. La vecindad de una solución es el conjunto de soluciones que se obtienen modificando ligeramente la solución actual. La exploración de la vecindad se realiza mediante una función de evaluación que permite determinar la calidad de las soluciones vecinas.
Existen diversos algoritmos de búsqueda local, entre los que destacan el algoritmo de búsqueda local iterada, *Simulated Annealing* y la búsqueda tabú. El algoritmo de búsqueda local iterada utiliza un conjunto de soluciones iniciales y las mejora mediante la exploración de sus vecindades. El *Simulated Annealing* utiliza una función de temperatura para aceptar soluciones peores que la actual con una probabilidad decreciente a medida que se avanza en la búsqueda. La búsqueda tabú utiliza una lista de movimientos prohibidos para evitar ciclos y explorar diferentes áreas de la solución. [6]
Los algoritmos de búsqueda local son útiles en problemas de optimización donde la función objetivo no es convexa o donde se dispone de información incompleta sobre el problema. Sin embargo, pueden quedar atrapados en óptimos locales si la función objetivo presenta múltiples óptimos locales. Por esta razón, se suelen utilizar en combinación con otras técnicas de búsqueda, como los algoritmos genéticos o los algoritmos de optimización global.

2.6. Algoritmos Metaheurísticos

Los algoritmos metaheurísticos son técnicas de búsqueda y optimización que se utilizan para resolver problemas que no tienen una solución analítica o exacta. Estos se basan en estrategias inspiradas en la naturaleza, la física, la estadística, la matemática y otros campos, y utilizan un proceso iterativo para encontrar soluciones cada vez mejores.
A diferencia de los métodos exactos, estos algoritmos son muy útiles para resolver problemas complejos en los que los algoritmos deterministas no son efectivos. Estos algoritmos han demostrado ser muy eficaces en la optimización de funciones matemáticas, la planificación de rutas, la optimización de la producción, la asignación de recursos y muchos otros campos. Sin embargo,

también tienen algunas limitaciones, como la necesidad de ajustar muchos parámetros y la falta de garantía de que la solución encontrada sea la mejor [3]. Estos algoritmos se dividen en dos grandes grupos, los de trayectoria y los de población, los primeros se basan en la idea de mejorar iterativamente una solución inicial a través de movimientos o cambios en la misma, con el objetivo de encontrar una solución óptima o de alta calidad. A medida que avanzan en la búsqueda, pueden moverse en diferentes direcciones y explorar diferentes regiones del espacio de búsqueda, un ejemplo son los algoritmos de búsqueda local mencionados anteriormente.

Por otro lado, los de población, se inician con una población inicial de soluciones aleatorias o generadas de alguna manera. Luego, se aplican operadores de selección, reproducción y variación para generar nuevas soluciones en cada iteración. Estos operadores imitan los procesos biológicos como la selección natural, la reproducción sexual y la mutación genética. Algunos ejemplos de algoritmos metaheurísticos de población son el *Particle Swarm*, Algoritmo de colonia de hormigas y el algoritmo genético. Cada uno de estos algoritmos se basa en un principio diferente, pero todos tienen en común que utilizan técnicas de búsqueda no deterministas y probabilísticas.

Las principales diferencias entre estos tipos de algoritmos son el enfoque y la eficiencia. Los algoritmos de población tienen una mayor capacidad para explorar el espacio de búsqueda debido a la diversidad de soluciones presentes en la población. Esto les permite abarcar una amplia gama de soluciones potenciales, dando un rendimiento robusto y encontrando soluciones de alta calidad en problemas complejos, pero tienden a ser más lentos en converger hacia soluciones óptimas debido a la necesidad de explorar.

Por otro lado, los algoritmos de trayectoria se centran más en la explotación de la solución actual, utilizando técnicas como el descenso de gradiente o la búsqueda local para mejorar gradualmente la solución actual, otorgándoles una convergencia rápida hacia soluciones localmente óptimas, pero pueden quedarse atrapados en mínimos locales y tener dificultades para encontrar soluciones globales en problemas más complejos.

Es importante destacar que estas diferencias no son estrictas y existen algoritmos que combinan elementos de ambos enfoques. Además, la elección entre algoritmos de población y de trayectoria depende del tipo de problema a resolver y de las características específicas del problema, como el tamaño del espacio de búsqueda, la presencia de múltiples óptimos y la disponibilidad de recursos computacionales.

2.7. Hill Climbing

El *Hill Climbing*, también conocido como Escalada de Colina, es un algoritmo metaheurístico de trayectoria utilizado en problemas de optimización. Se basa en la idea de mejorar iterativamente una solución inicial mediante movimientos hacia soluciones vecinas que mejoran la función objetivo. Este algoritmo tiene la siguiente estructura:

- Estado inicial: selecciona un estado inicial como punto de partida para la búsqueda. Puede ser generado de manera aleatoria o utilizando algún conocimiento previo.
- Función de evaluación: define una función objetivo que permita evaluar la calidad de una solución en términos de su valor. El objetivo es maximizar o minimizar esta función según el problema.
- Operador de movimiento: define cómo se generan los vecinos de un estado dado. Los vecinos son soluciones adyacentes que se obtienen mediante modificaciones simples en el estado actual.
- Operador de selección: determina cómo se selecciona el siguiente estado a explorar entre los vecinos generados. Normalmente, se usa una estrategia de mejoría (selecciona el mejor vecino).
- Condición de parada: define el criterio para finalizar la búsqueda. Puede ser una cantidad máxima de iteraciones, un límite de tiempo, o cuando no se encuentran mejores soluciones en un número determinado de iteraciones.

El algoritmo comienza con una solución inicial y evalúa su calidad según la función objetivo. Luego, busca vecinos cercanos realizando pequeños cambios o modificaciones en la solución actual. Si encuentra un vecino que mejora la función objetivo en comparación con la solución actual, se mueve hacia ese vecino y repite el proceso. Este proceso continúa hasta que ocurra la condición de parada, momento en el que se considera que se ha alcanzado un óptimo local.
Dada esta estructura se pueden obtener distintas variantes de *Hill Climbing*, ya que al cambiar cómo se realizan las acciones del algoritmo se pueden llegar a distintas soluciones. Por ejemplo, existe el *Hill Climbing Estocástico*, que su operador de selección le permite escoger soluciones de menor calidad

con el objetivo de evitar quedar atrapado en óptimos locales. Otro es el *Hill Climbing con Reinicio Aleatorio*, que una vez terminada una iteración parte desde un estado inicial completamente nuevo, ya que se quiere explorar diferentes regiones del espacio de búsqueda.
A pesar de que este algoritmo es relativamente simple, fácil de implementar y flexible para distintos problemas, tiene algunas limitaciones. Debido a su enfoque local, puede quedar atrapado en óptimos locales y al necesitar una solución inicial, esta puede afectar la calidad de la solución final encontrada.

2.8. Algoritmo genético

Los algoritmos genéticos son una clase de algoritmos metaheurísticos de tipo población inspirados en los principios de la evolución biológica. Se utilizan para resolver problemas de optimización y búsqueda en diferentes campos, como la inteligencia artificial, la computación evolutiva y la ingeniería.
El funcionamiento de este se basa en la simulación de procesos evolutivos. Cada solución del problema se representa como un cromosoma, que consta de genes que codifican características o parámetros de la solución. Los cromosomas forman una población inicial de individuos, donde cada individuo representa una solución potencial. El proceso de búsqueda y optimización se realiza mediante la aplicación de operadores genéticos, que simulan los procesos de reproducción, recombinación y mutación presentes en la evolución biológica. Estos operadores actúan sobre los individuos de la población, generando nuevas soluciones y promoviendo la mejora gradual de la calidad de las soluciones a lo largo de las generaciones. Lo que conlleva a tener la siguiente estructura:

- Codificación: es la forma en la que se representa las distintas soluciones del problema en una forma que pueda ser manipulada por el algoritmo. Por ejemplo, pueden ser cadenas binarias, arreglos de números o según el tipo de problema.

- Población inicial: se genera una población inicial de individuos de manera aleatoria o utilizando un conocimiento previo del problema. Cada individuo representa una solución.

- Evaluación de aptitud: la función de aptitud determina qué tan adecuada es una solución en términos del objetivo del problema.

- Selección: se seleccionan individuos de la población actual para reproducirse y crear una descendencia. Los individuos con una mayor aptitud tienen más probabilidad de ser seleccionados, pero también se pueden utilizar métodos de selección que promuevan la diversidad genética.

- Operadores genéticos: durante el proceso de reproducción, se aplican operadores genéticos para combinar y modificar los cromosomas de los individuos seleccionados. Los operadores principales son la recombinación (cruce) y la mutación. La recombinación mezcla información genética de dos o más individuos, mientras que la mutación introduce cambios aleatorios en los cromosomas para mantener la diversidad, creando así nuevos individuos.

- Reemplazo: la descendencia generada reemplaza a los individuos menos aptos de la población anterior, lo que implica una competencia entre los individuos de la población actual y los descendientes.

- Convergencia: el proceso de selección, reproducción y reemplazo se repite durante varias generaciones hasta que se alcance un criterio de término, como un número máximo de generaciones o una aptitud objetivo.

Una de las ventajas de estos algoritmos es su capacidad amplia de exploración del espacio de búsqueda al mantener una población diversa de soluciones, pero también la eficiencia computacional puede ser una preocupación debido al alto costo de las evaluaciones de la función objetivo, especialmente en problemas complejos. Además, la selección de parámetros adecuados, como el tamaño de la población, las tasas de recombinación y mutación, puede requerir experiencia y conocimiento para obtener un rendimiento óptimo. [7]

2.9. IbexOpt

Ibex es una biblioteca en C++ que se utiliza para procesar restricciones en números reales. Su objetivo principal es manejar restricciones no lineales teniendo en cuenta los errores de redondeo. Se basa en conceptos como aritmética de intervalos y aritmética afín [9].

Una de las características distintivas de Ibex es que incluye todos los métodos del estado del arte relacionados con *Branch and bound* basado en intervalos. Además, Ibex puede funcionar como un solucionador de caja negra, lo que significa que puede utilizarse sin necesidad de conocer los detalles internos de su implementación.

Entre los problemas que se pueden abordar con Ibex se encuentran la resolución de sistemas de ecuaciones no lineales y la optimización global de funciones sujetas a restricciones no lineales. En ambos casos, Ibex ofrece garantías sobre los resultados obtenidos, proporcionando recintos garantizados para las soluciones.

Ibex es un proyecto académico de código abierto que ha estado en desarrollo desde 2007. Su objetivo principal es proporcionar herramientas confiables y eficientes para el procesamiento de restricciones en el ámbito de los números reales.

Capítulo 3

Definición del problema y propuesta de solución

Los problemas NCOP están cada vez más presentes en diversos ámbitos, como el cálculo de trayectorias en robótica y la optimización de portafolios. Estos problemas requieren de gran potencia computacional para su resolución, lo que ha llevado a investigar y desarrollar diferentes enfoques para facilitar su solución. El algoritmo *Branch and Bound*, con su enfoque basado en intervalos, ha logrado obtener buenos resultados al buscar el óptimo global de estos problemas. La inclusión del método AbsTaylor, que utiliza la expansión de la serie de Taylor, ha mejorado aún más estos resultados con la aproximación lineal del problema mediante la construcción de un polítopo en cualquier área del espacio de búsqueda para encontrar soluciones factibles. Sin embargo, a pesar de estos avances, no se está aprovechando todo el potencial del método. Actualmente, el punto de expansión del polítopo en AbsTaylor se fija al inicio del árbol de búsqueda en el algoritmo *Branch and Bound*, siempre seleccionando el punto medio del dominio del nodo en el que se está trabajando. Por lo tanto, se propone una selección dinámica de este punto utilizando algoritmos metaheurísticos y experimentar el impacto de invertir recursos en la búsqueda de un punto de expansión óptimo para el politopo en el método AbsTaylor. Esta mejora contribuiría a:

- Un primer acercamiento a la investigación e implementación en la selección del punto de expansión de Abstaylor dentro del algoritmo *Branch*

and Bound utilizando algoritmos metaheurísticos.

- Comprobar si la selección de este punto influye en la eficiencia del algoritmo de manera positiva o negativa y conocer el impacto que se tiene en comparación a su versión anterior.

La selección dinámica del punto requería cumplir dos requisitos: 1) adaptabilidad del algoritmo en un entorno de dominio continuo e intervalos, lo cual implicaba la capacidad de trabajar con entornos y parámetros variables para explorar el área de búsqueda, y 2) eficiencia, donde se buscaba un bajo consumo de recursos y la capacidad de ofrecer soluciones aceptables en tiempos razonables. Estos requisitos eran cumplidos por los algoritmos metaheurísticos, especialmente el algoritmo de trayectoria Hill-Climbing. Su estructura permite adaptarse fácilmente a diferentes parámetros, realizar modificaciones para aumentar la rigurosidad para trabajar el punto o controlar el número de iteraciones, y su enfoque de trayectoria mejora gradualmente el punto inicial hasta alcanzar un criterio de parada.

3.0.1. Metodología

Para lograr los objetivos que dirigen este trabajo se definieron los siguientes puntos:

- Estudiar los distintos algoritmos metaheurísticos y sus capacidades para implementarlos dentro del algoritmo *Branch and Bound* basado en intervalos.
- Encontrar librería de código abierto que contenga las funciones necesarias de *Branch and bound* para la implementación del algoritmo.
- Buscar problemas NCOP para la realización de *benchmarks* del algoritmo actual y el propuesto.
- Diseñar el algoritmo seleccionado.

En cuanto al primer punto, existían diversas opciones para abordar este problema. Al estudiar los algoritmos metaheurísticos, se encontró que los algoritmos de población, aunque efectivos en problemas con espacios de búsqueda extensos, podían presentar dificultades en la adaptación a dominios o variables continuas, ya que requerían una representación y operadores adecuados para las soluciones. Dado que el algoritmo estaría en funcionamiento

en cada nodo del árbol de búsqueda, era crucial buscar eficiencia en términos de recursos y tiempo.

Por estas razones, se optó por utilizar un algoritmo metaheurístico de trayectoria, como *Hill-Climbing*, que resulta adecuado para dominios continuos e intervalos. Su estructura simple le permite trabajar con variables y parámetros variables. Además, al trabajar con una única solución en cada iteración, tiende a ser más eficiente en términos de consumo de recursos computacionales. El objetivo principal era encontrar buenos puntos de expansión, y la mejora gradual y la reducción del espacio de búsqueda que ocurren a través de la bisección del algoritmo *Branch and Bound* resultan suficientes para alcanzar soluciones aceptables.

Para abordar el segundo punto, se decidió utilizar la librería Ibex (ver Sección 2.9). La elección de esta librería se basó en su capacidad para manejar restricciones no lineales y realizar operaciones en el ámbito de los números reales. Además, incluye el algoritmo *Branch and Bound* junto con las funciones necesarias, así como la implementación del método Abstaylor dentro del upper bounding, utilizando la selección del punto de expansión en el punto medio. Sin embargo, la librería no cuenta con la implementación de ningún algoritmo metaheurístico.

Por lo tanto, será necesario realizar un estudio exhaustivo del código para comprender el funcionamiento de la librería y diseñar e implementar correctamente la solución propuesta. Es importante destacar que esta librería ha sido utilizada en diversas investigaciones dentro del mismo campo [11], lo que la convierte en una opción confiable y sólida.

Para la selección de problemas de *benchmarks* para evaluar los algoritmos, se buscó obtener una cantidad de 30 de problemas NCOP con diversidad en términos de tamaño y dificultad. Para lograrlo, se recopilaron archivos que contenían problemas NCOP de la comunidad de optimización. La selección de los problemas se realizó ejecutando el algoritmo de *Branch and Bound* con Abstaylor utilizando el punto medio proporcionado por la librería Ibex. El tiempo de ejecución necesario para obtener la solución óptima global fue la métrica utilizada para considerar los problemas en los *benchmarks* finales. El objetivo fue obtener una colección de 30 problemas con tiempos de ejecución que variaran entre 4 segundos y 40 minutos, logrando así una distribución equilibrada en términos de tiempo de resolución.

Finalmente, para el diseño del algoritmo *Hill-Climbing*, debido a la estructura que posee, es posible adaptarlo de manera correcta a los dominios

continuos e intervalos, por lo que para obtener una mayor cantidad de datos y comprender de mejor manera cómo afectan los cambios realizados en los resultados finales. Se quiere realizar distintos algoritmos de *Hill-Climbing* específicamente cuatro, con esto se refiere los procesos de Operador de selección y función de evaluación tendrán dos formas distintas, con el objetivo de crear distintas versiones de este algoritmo que vayan aumentando la rigurosidad de este y conocer el impacto que tienen estos pequeños cambios dentro de los resultados.

3.0.1.1. Desarrollo

El desarrollo de la investigación contempla dos etapas, las que buscan responder a la necesidad de dar solución a estos problemas y se definen por:

- Diseño: sobre la base de lo estudiado en el capítulo anterior, esta etapa del desarrollo consta en la elaboración del diseño de la solución. Se identifican y definen los componentes que deben ser adaptados para dar un funcionamiento a la solución en el entorno del problema, como, por ejemplo: parámetros, casos de borde, entre otros. De esta forma, el diseño promete analizar las implementaciones mediante diferentes características.

- Implementación: cuando el diseño sea el adecuado, se procede a concretizar el diseño, con el fin de obtener la solución. En esta etapa, se itera la construcción del diseño con el objetivo de que la implementación se realice de manera correcta para la resolución de los problemas a medir.

Capítulo 4

Diseño de la Solución

Este capítulo detallará todo lo que respecta a la solución, tanto su etapa de diseño (conceptual) como su posterior implementación. Se describirán los pasos y métodos utilizados para desarrollar la solución, así como las herramientas y tecnologías utilizadas en su implementación.

4.1. Diseño Hill-Climbing

En esta primera etapa, se busca establecer la secuencia de operaciones computacionales de un algoritmo *Hill-Climbing*, es decir, los procesos que son realizados por este algoritmo (ver Sección 2.7). Como se mencionó en la metodología, se pretende tener diferentes versiones de este algoritmo para obtener una mayor cantidad de datos y realizar pruebas de distintos niveles de rigurosidad al momento de trabajar el punto de expansión. Por lo tanto, algunos procesos estarán identificados con un número al final para indicar que existen en diferentes versiones del algoritmo, mientras que los procesos sin número funcionarán de la misma manera en todas las versiones. Por ejemplo, si se tiene una "función de evaluación 1" para el algoritmo de *Hill-Climbing* versión 1 va a utilizar esta función de evaluación en cada iteración, mientras que otra versión del algoritmo utilizará la "función de evaluación 2".

- Estado inicial: se espera una cantidad de intervalos variables, representando cada uno de estos el dominio de cada una de las variables del problema, por lo que se considerará como una solución inicial un valor

que se ubique entre medio de los límites del intervalo para cada intervalo recibido, por lo que para la obtención de esta solución inicial se realizará de manera aleatoria, consiguiendo un número perteneciente al intervalo.

- Operador de movimiento: con el objetivo de crear un vecindario estable de la solución y que estos no se salgan de los límites definidos por los intervalos, se optó por perturbar de manera dinámica las distintas variables de la solución actual, así considerando un vecino como la solución actual con una de sus variables perturbada por un pequeño ruido. Para el cálculo del ruido sé utilizará un número aleatorio entre cero y uno, para luego ser multiplicado por la amplitud del intervalo y una magnitud de valor 0.1, con el objetivo de que el ruido se ajuste dinámicamente al intervalo y este no tenga la posibilidad de salirse de los límites de este, también se tendrá una cantidad de vecinos igual a la cantidad de variables tenga el problema. El objetivo de esto es para que los puntos generados por el operador tengan una cierta distancia del punto inicial y no se realiza una sobre-exploración, siendo esto un vecindario estable y diferenciándolo de escoger los vecinos de manera aleatoria.

- Condición de parada: ya que se busca un consumo bajo de recursos y a la vez que el código funcionara en cada nodo, por lo que el espacio de búsqueda se repite, pero de manera reducida, la condición de parada del algoritmo es una cantidad máxima de 50 iteraciones, también si no se mejora la solución en una iteración se detendrá.

- Función de evaluación 1: para medir si un punto del espacio de búsqueda es de buena calidad para ser usado de punto de expansión, será necesario evaluarlo en las restricciones, siendo de mayor calidad mientras más se acerque al 0, esto se debe a que mientras el punto se encuentre más cercano a la restricción, podrá construirse un mejor politopo, dando una mejor aproximación del problema. La primera función de evaluación será tomando de manera aleatoria una restricción del problema y evaluando el punto, es importante mencionar que una restricción distinta para cada nodo que se trabaje para obtener una mayor diversidad de puntos.

- Función de evaluación 2: para la segunda versión de este proceso, el

punto será evaluado en todas las restricciones del problema, siendo de mayor calidad el punto que se acerque de mejor manera al cero, después de sumar todas las evaluaciones de las restricciones en valor absoluto. Este enfoque es más completo al evaluar el punto en todas las restricciones del problema, siendo también un mayor gasto de recursos de procesamiento.

- Operador de selección 1: el primer operador de selección que se tendrá será primera-mejora, es decir, cuando se genere un vecino que sea de mayor calidad que la solución actual que se está trabajando, dejará de generar vecinos el vector de movimiento y la solución encontrada pasará a ser la solución actual en la siguiente iteración, este método utiliza pocos recursos, ya que es más fácil llegar a la condición de parada al no explorar todas las posibilidades del vecindario.

- Operador de selección 2: el segundo operador será mejor-mejora, consiste en generar todo el vecindario de posibilidades antes de seleccionar la solución para la siguiente iteración, con el objetivo de escoger al mejor vecino de mayor calidad encontrado. Este proceso es más completo que el anterior al generar todo el vecindario posible antes de pasar a la siguiente iteración, por lo que también consume una mayor cantidad de recursos.

Cada algoritmo se diferencia en la combinación única de operador de selección y función de evaluación utilizados, al combinar los dos operadores de selección y las dos funciones de evaluación, se obtiene cuatro posibles combinaciones, lo que resulta en cuatro algoritmos de *Hill-Climbing* distintos. Cada uno de estos algoritmos tendrá distintos niveles de exploración y explotación del espacio de búsqueda, como también utilizarán más recursos que otros. Las diferencias de las versiones del algoritmo se muestran en la Tabla 4.1.

Versión	Operador de selección	Función de evaluación
1	Primera-mejora	Restricción aleatoria
2	Mejor-mejora	Restricción aleatoria
3	Primera-mejora	Todas las restricciones
4	Mejor-mejora	Todas las restricciones

Tabla 4.1. Diferencias entre las versiones de Hill-Climbing

4.2. Implementación en C++

Antes de la implementación del algoritmo en C++, fue necesario realizar un estudio exhaustivo de la librería Ibex y crear el ambiente de trabajo donde se ejecutaría el algoritmo para la obtención de resultados.

4.2.1. Docker

Para el ambiente de trabajo se utilizó Docker para crear el contenedor necesario para la implementación y ejecución de la solución propuesta y de la librería Ibex. Se usó una imagen de Ubuntu con la versión 22.04.2 y se instalaron los siguientes programas necesarios para el funcionamiento de Ibex:

- g++
- gcc
- flex
- bison
- python3
- make
- build-essential

Esta opción también otorga realizar cambios de sistema de manera rápida y eficiente en caso de cualquier problema, además de poder replicarlos en cualquier otro computador.

4.2.2. Automatización benchmarks

Para la obtención de resultados se utilizará un *framework* para experimentación escrito en python y construido por el usuario de github rilianx, este proporciona los siguientes beneficios:

- Permite comparar varios algoritmos con relación a un algoritmo de referencia.
- Algoritmos más prometedores son ejecutados en primer lugar, dejando a los menos prometedores para el final.

- Resumen de resultados parciales (ganancia relativa, instancias con mejores/peores resultados).
- Número de ejecuciones por algoritmo+instancia se establece dinámicamente de acuerdo a variación de resultados.
- Para cada algoritmo se pueden ver los tiempos promedios por instancia y un error asociado a distintas ejecuciones.

Este *framework* es compatible con la ejecución en múltiples hilos, también conocidos como "threads". Los hilos son unidades de ejecución independientes dentro de un proceso. Un proceso puede tener varios hilos que comparten el mismo espacio de memoria y los recursos del sistema. Cada hilo dentro de un proceso puede realizar tareas de forma concurrente y paralela, lo que permite aprovechar la capacidad de procesamiento de los sistemas modernos con múltiples núcleos de CPU y ejecutar todos los algoritmos propuestos en condiciones equitativas.

En este caso, cada instancia se ejecutará utilizando hilos. Dado que se realizarán cinco ejecuciones con diferentes semillas para cada archivo, se utilizarán cinco hilos para cada archivo. Estos hilos funcionarán de manera independiente y concurrente hasta que sea necesario escribir los resultados en el archivo de resultados correspondiente. Para configurar el uso de este *framework*, se proporciona una lista de instancias a utilizar, así como un archivo de configuración que define la ubicación del algoritmo, la cantidad de semillas a utilizar, los parámetros de entrada y el rango de tiempo permitido para cada ejecución.

4.2.3. Libreria Ibex

Durante el estudio se encontró que esta librería contiene una amplia variedad de clases y funciones que no eran relevantes para esta implementación, por lo que fue necesario identificar y utilizar únicamente las clases necesarias para nuestra solución.

En primer lugar, se utilizarán las clases *Interval* e *IntervalVector*. La clase *Interval* representa un rango de valores con dos números, que se utiliza para representar los dominios de las variables. Si ambos números son iguales, se trata de un valor concreto, ya que todas las operaciones aritméticas se deben realizan sobre intervalos. Por otro lado, la clase *IntervalVector* es un arreglo de objetos *Interval* y se utiliza para representar el dominio del problema,

un nodo en particular o un punto del espacio de búsqueda. Esta clase es crucial, ya que es el punto de entrada y salida de nuestra solución, y toda la estructura del *Hill-Climbing* trabaja con ella.

También se ocupará la clase *System*, que representa el problema NCOP con el que se está trabajando. A través de esta clase, se pueden obtener las variables, restricciones y la función objetivo del problema. Esto permite obtener la función de evaluación definida en el diseño de la solución, que es específicamente las restricciones del problema.

Finalmente, la clase Random, que a pesar de la existencia de muchas librerías que otorgan la generación de números aleatorios, utilizarlas provocaría distintos problemas, siendo principalmente el no poder igualar la semilla del generador de números que utiliza la librería que se le entrega por parámetros de entrada. La consistencia en la generación de números aleatorios era fundamental para garantizar que todos los algoritmos se ejecutaran en condiciones equitativas y comparables.

Una vez estudiado la librería, en el Pseudocódigo 1 se muestra la implementación en C++ para la selección dinámica del punto de expansión.

Algorithm 1 Pseudocódigo implementación Hill-Climbing en C++

```
Require: box, Sys
  Inicializar solución inicial rand(box.lb, box.up)
  best = f_eval(start)                          ▷ Funcion de evaluación
  best_point = start
  while stop condition do
    for i in start do
      new = new_solution()                      ▷ Vector de movimiento
      if f_eval(new) < best then
        best = f_eval(new)
        best_point = new
      end if
    end for
    start = best_point
  end while
  return best_point
```

Capítulo 5

Resultados

En esta sección se presentará los *benchmarks* usados para comprobar la eficiencia de los distintos algoritmos usados, el procedimiento y ambiente que se utilizó para cada ejecución, resultados finales de cada uno de los algoritmos propuestos, mostrando tiempo de procesamiento y cantidad de nodos generados en promedio, además de un análisis final de los resultados.

5.1. Benchmarks

Para los *benchmarks*, se utilizarán archivos proporcionados por la comunidad de optimización global. Se han seleccionado un total de 30 instancias que tienen diferentes tiempos de procesamiento aproximados. El objetivo es probar los algoritmos en problemas de diferentes tamaños. En la Tabla 5.1 se muestran los archivos utilizados.

Archivo	Cantidad de variables	Cantidad de restricciones
ex2_1_8	24	10
ex2_1_9	10	1
ex6_1_3bis	7	3
ex6_2_12	5	2
launch	38	28
ex8_5_1bis	7	6
odfits	10	6
ex6_2_8	3	1
hs113	10	8
ex2_1_7	20	10
himmel16	15	21
ex6_2_9	4	2
hs088	2	1
ex6_2_6	3	1
ex8_4_4bis	5	5
ex14_2_7	6	9
ex6_1_3	12	9
ex2_1_9bis	9	1
dualc5	8	1
hs108	9	13
ex6_2_10	6	3
dualc8	8	15
hs093	6	2
hs103	7	6
chembis	11	4
mistake	9	13
chem-1	11	4
chem	11	4
sambal	15	8
hs102	7	6

Tabla 5.1. Tabla de instancias utilizadas

Los archivos contienen las distintas variables del problema con sus respectivos intervalos de posibles valores que pueden tomar, la función objetivo a minimizar y después la serie de restricciones a las que está sujeto el problema.

Ambiente y procedimientos

En el ambiente de trabajo no se puso ningún límite de recursos, por lo que el contenedor usaba todos los recursos disponibles del computador, indicados en la Tabla 5.2.

Componente	**Especificaciones**
Procesador	Intel Core i9-9900 2.6 GHz 8-Core 16-Thread
Memoria RAM	16 GB

Tabla 5.2. Especificaciones de la Computadora

Para garantizar resultados más consistentes y reducir el impacto de la aleatoriedad en el algoritmo, se ejecutó cada archivo utilizando 5 semillas distintas. De esta manera, los resultados presentados reflejarán el promedio del tiempo y la cantidad de cajas generadas obtenidos a partir de estas 5 ejecuciones, además de que se considerará como término de la ejecución cuando el algoritmo obtenga un óptimo de 10^{-4}. Es importante mencionar que los resultados base corresponden al algoritmo sin modificaciones, es decir, el método *Abstaylor* con el punto medio como enfoque utilizado. Este enfoque se utilizará como punto de referencia para comparar los resultados de los distintos algoritmos propuestos y usando las métricas de tiempo de procesamiento y cantidad de cajas generadas para comparar que algoritmo tuvo un mejor desempeño.

5.2. Resultados algoritmos

A continuación, se presentarán los resultados obtenidos para cada algoritmo, junto con un análisis general de su desempeño en comparación con el algoritmo base. Posteriormente, se realizarán análisis finales que permitirán comparar tanto el algoritmo base como los distintos algoritmos propuestos, con el objetivo de identificar los impactos de los cambios realizados en cada uno de ellos.

Es importante destacar que los análisis se basarán en los resultados obtenidos sin hacer referencia específica a archivos individuales. El enfoque será

proporcionar una visión general de cómo cada algoritmo se comportó en relación con el algoritmo base y los cambios que se introdujeron en cada uno de ellos.

En los resultados obtenidos se marcaron los mejores resultados para cada archivo, tanto en términos de tiempo como de cantidad de cajas. Se identificaron 17 instancias en las que una de las implementaciones con *Hill-Climbing* obtuvo mejores resultados en cuanto a la cantidad de cajas, y 9 instancias en las que logró el mejor tiempo de procesamiento.

En cuanto a las mejoras superiores al 5 %, se encontraron 4 casos. Dos de estos casos presentaron mejoras únicamente en la cantidad de cajas, mientras que los otros dos mostraron diferencias tanto en el tiempo de procesamiento como en la cantidad de cajas.

5.2.1. Resultados Abstaylor punto medio

A continuación, se presentan los resultados obtenidos utilizando el algoritmo actual, el cual sirve como punto de referencia para comparar los datos y análisis de los algoritmos propuestos. Los resultados de la Tabla 5.3 representan el rendimiento actual del algoritmo y serán utilizados como base comparativa para evaluar el desempeño de los algoritmos propuestos.

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	14.4	1660
ex2_1_9	10.7	5158
ex6_1_3bis	**14.9**	11170
ex6_2_12	**8**	**7728**
launch	11.6	830
ex8_5_1bis	**8.6**	**3795**
odfits	**39.9**	**17759**
ex6_2_8	31.3	31831
hs113	**12.5**	**3172**
ex2_1_7	15.5	2449
himmel16	29.6	4608
ex6_2_9	**25.5**	21544
hs088	83.1	1326
ex6_2_6	63.9	60305
ex8_4_4bis	89.1	77113
ex14_2_7	55.7	13020
ex6_1_3	**94.9**	**16208**
ex2_1_9bis	122	75318
dualc5	94.9	54014
hs108	**115**	**37310**
ex6_2_10	**554**	**330089**
dualc8	**389**	189155
hs093	530	421494
hs103	725	258233
chembis	**1077**	480896
mistake	**1113**	388082
chem-1	**2048**	905104
chem	**2037**	905104
sambal	2534	515910
hs102	**2140**	899999

Tabla 5.3. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

5.3. Resultados Abstaylor punto aleatorio

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	14.1	1648
ex2_1_9	10.8	**5115**
ex6_1_3bis	15.3	11224
ex6_2_12	8.2	7830
launch	**6.1**	427
ex8_5_1bis	9.5	4117
odfits	40.7	18394
ex6_2_8	31.6	31805
hs113	15.4	4666
ex2_1_7	**15.5**	2469
himmel16	25.1	4030
ex6_2_9	26.4	21531
hs088	**81.3**	1317
ex6_2_6	64.3	**59838**
ex8_4_4bis	91.6	**76792**
ex14_2_7	**48.4**	**11288**
ex6_1_3	97.6	16960
ex2_1_9bis	125	75130
dualc5	98.1	54196
hs108	132	40861
ex6_2_10	561	330862
dualc8	400	190056
hs093	538	438224
hs103	**724**	254268
chembis	1084	483935
mistake	1198	429354
chem-1	2066	905385
chem	2060	905385
sambal	**2481**	**500617**
hs102	2147	897172

Tabla 5.4. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

Los datos de la Tabla 5.4 obtenidos del método de Abstaylor con la selección del punto de expansión de manera aleatoria muestran resultados similares en la mayoría de los casos analizados. En algunos casos se observa una mejora con la selección aleatoria del punto de expansión, en promedio, este enfoque tiene un desempeño ligeramente inferior al método actualmente utilizado, que consiste en seleccionar el punto en el medio del dominio.

Al calcular el promedio de los resultados obtenidos en términos de la

cantidad de cajas generadas y el tiempo de procesamiento, y al compararlos con el algoritmo actual, se observa que el método de selección aleatoria generó, en promedio, 1483 cajas más y tuvo un aumento de 4 segundos en el tiempo de procesamiento en comparación con el algoritmo actual.

5.4. Resultados Hill-Climbing versión 1

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	**13.9**	1657
ex2_1_9	10.7	5153
ex6_1_3bis	15.2	**11018**
ex6_2_12	8.2	7818
launch	28.5	2744
ex8_5_1bis	9.5	3927
odfits	41.2	18268
ex6_2_8	31.9	31823
hs113	15.6	4554
ex2_1_7	15.8	**2444**
himmel16	24.8	3878
ex6_2_9	25.9	**21432**
hs088	107	1251
ex6_2_6	64.5	60138
ex8_4_4bis	91.4	78060
ex14_2_7	58.8	13258
ex6_1_3	96.3	16743
ex2_1_9bis	123	75232
dualc5	94.7	53392
hs108	136	44554
ex6_2_10	564	330817
dualc8	411	187776
hs093	542	427201
hs103	752	257574
chembis	1092	**480678**
mistake	1189	417654
chem-1	2090	904233
chem	2073	904233
sambal	2542	502954
hs102	2250	904957

Tabla 5.5. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

Los resultados de la Tabla 5.5 obtenidos por la primera versión del algoritmo *Hill-Climbing* mostraron valores mixtos en comparación con el algoritmo

actualmente utilizado. En algunos casos, se observaron mejoras en términos de la cantidad de cajas generadas y el tiempo de procesamiento, aunque estas mejoras fueron poco significativas, mientras que en otros logra tener un peor desempeño. Sin embargo, de manera general, esta primera versión tuvo un desempeño inferior en comparación con el algoritmo actual.

Al analizar los resultados en promedio, se observa que esta primera versión generó 1167 cajas más y tuvo un aumento de 14 segundos en el tiempo de procesamiento en comparación con el algoritmo actualmente utilizado. Aunque se logró una mejora en la cantidad de cajas generadas en comparación con la versión de punto aleatorio, también se experimentó un incremento en el tiempo necesario para mejorar el punto inicial seleccionado.

Es importante destacar que esta primera versión del algoritmo fue la menos rigurosa en cuanto al consumo de recursos para mejorar la solución. No fue necesario generar todo el vecindario de soluciones, y las soluciones se evaluaron solamente en una restricción. Estos factores pueden haber influido en los resultados obtenidos.

5.5. Resultados Hill-Climbing versión 2

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	14.2	1662
ex2_1_9	11	5161
ex6_1_3bis	16.4	11197
ex6_2_12	8.9	7804
launch	9.9	750
ex8_5_1bis	9.9	4103
odfits	41.4	18268
ex6_2_8	33.3	**31752**
hs113	15.8	3942
ex2_1_7	17.5	2480
himmel16	23.9	3662
ex6_2_9	26.8	21622
hs088	115	**1232**
ex6_2_6	67.2	60166
ex8_4_4bis	93	78060.8
ex14_2_7	59.3	12547
ex6_1_3	101	17196.4
ex2_1_9bis	125	75072
dualc5	100	53564
hs108	147	42796
ex6_2_10	567	330888
dualc8	429	188848
hs093	542	**407984**
hs103	790	253816
chembis	1122	481794
mistake	1233	420313
chem-1	2108	905300
chem	2093	905300
sambal	2580	514334
hs102	2362	**885043**

Tabla 5.6. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

En cuanto a la segunda versión del algoritmo, de la Tabla 5.6 se observa un desempeño ligeramente inferior en comparación con los demás algoritmos. Durante los cálculos, se generaron 208 cajas adicionales y el tiempo de procesamiento aumentó en 25 segundos en promedio. Aunque se logró una mejora significativa en la generación de cajas en comparación con las versiones anteriores, no se alcanzaron los resultados del algoritmo actualmente utilizado (punto medio). Además, el proceso de generar el vecindario com-

pleto de soluciones y seleccionar la mejor solución resultó en un aumento en el tiempo de procesamiento.

Estos resultados indican que trabajar en la mejora del punto de expansión puede proporcionar una mejora en la generación de nodos en el árbol de búsqueda. Sin embargo, es necesario evaluar cuidadosamente el equilibrio entre la calidad de las soluciones generadas y el tiempo de ejecución del algoritmo. En el caso de esta segunda versión, aunque se logró una reducción en la cantidad de cajas generadas, el aumento en el tiempo de procesamiento podría limitar su eficacia en comparación con el algoritmo actualmente utilizado.

5.6. Resultados Hill-Climbing versión 3

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	14.8	1690
ex2_1_9	10.8	5153
ex6_1_3bis	15.3	11295
ex6_2_12	8.2	7788
launch	6.5	**218.8**
ex8_5_1bis	13.1	4248
odfits	47.7	18208
ex6_2_8	31.7	31809
hs113	17.3	4606
ex2_1_7	33.8	4602
himmel16	**19.9**	2632
ex6_2_9	26.4	21496
hs088	98	1251
ex6_2_6	65.3	60304
ex8_4_4bis	91.7	78060
ex14_2_7	71.8	12460
ex6_1_3	113	16936
ex2_1_9bis	126	75232
dualc5	98.1	**53387**
hs108	140	38535
ex6_2_10	590	330727
dualc8	424	188408
hs093	575	445561
hs103	862	247057
chembis	1265	483866
mistake	1248	393884
chem-1	2516	**903878**
chem	2490	**903878**
sambal	3000	508896
hs102	2667	929195

Tabla 5.7. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

Para los datos de la Tabla 5.7 que se obtuvieron con la tercera versión del algoritmo *Hill-Climbing*, donde se implementó la evaluación de los puntos de expansión en todas las restricciones del problema. Sin embargo, este enfoque resultó en un desempeño menos eficiente en comparación con las versiones anteriores y el algoritmo actualmente utilizado.

En promedio, esta tercera versión generó 1495 cajas más que el algoritmo actualmente utilizado. Esto indica una menor eficiencia en la generación de

soluciones válidas durante la búsqueda. Además, el tiempo de procesamiento aumentó en 86 segundos en comparación con el algoritmo actual. Este mayor tiempo de ejecución puede atribuirse a la necesidad de evaluar los puntos de expansión en todas las restricciones, lo cual representa una carga computacional adicional.

5.7. Resultados Hill-Climbing versión 4

Archivo	Promedio de tiempo en segundos	Promedio de cantidad cajas
ex2_1_8	14.9	**1613**
ex2_1_9	**10.6**	5161
ex6_1_3bis	15.3	11199
ex6_2_12	8.2	7840
launch	29.9	735
ex8_5_1bis	12.3	3808
odfits	50.1	18100
ex6_2_8	**31.1**	31782
hs113	19	4473
ex2_1_7	29	2540
himmel16	24.5	**2443**
ex6_2_9	26.4	21484
hs088	93.5	1232
ex6_2_6	**63.1**	60106
ex8_4_4bis	**89.1**	78060
ex14_2_7	89.4	12006
ex6_1_3	116	16859
ex2_1_9bis	**119**	**75072**
dualc5	**94**	53533
hs108	147	37871
ex6_2_10	561	330598
dualc8	395	**187215**
hs093	**513**	418432
hs103	1023	**245284**
chembis	1263	483676
mistake	1361	**387549**
chem-1	2517	905274
chem	2517	905274
sambal	3376	512628
hs102	3426	889192

Tabla 5.8. Promedio de tiempo en segundos y cantidad de cajas generadas por archivo

En la Tabla 5.8, donde se muestran los resultados de la última versión

del algoritmo, que se enfoca en gastar una mayor cantidad de recursos en mejorar gradualmente el punto de expansión al generar todo el vecindario y evaluar los puntos en todas las restricciones. En términos de la métrica de cajas generadas, en la mayoría de los casos, este algoritmo generó una cantidad igual o menor de cajas en comparación con el algoritmo actualmente utilizado, que es el de punto medio. Es importante destacar que se logró una reducción promedio de 978 cajas en comparación con el algoritmo de punto medio. Sin embargo, este enfoque que consume una gran cantidad de recursos también se tradujo en un tiempo de procesamiento mayor. En la mayoría de los benchmarks realizados, este algoritmo mostró un mayor tiempo de ejecución en comparación con los demás.

Estos resultados indican que, si bien la última versión del algoritmo logra generar menos cajas en promedio, el aumento significativo en el tiempo de procesamiento puede ser un factor limitante. Es importante considerar la relación entre la eficiencia en la generación de cajas y el tiempo de ejecución, ya que ambos aspectos son fundamentales para evaluar la calidad y el desempeño general del algoritmo.

5.7.1. Análisis general

En esta sección se realizará un análisis general de los resultados obtenidos, se presentarán gráficas comparativas y se explicarán los efectos de los cambios realizados en los diferentes algoritmos. Para realizar este análisis; se utilizó el promedio general de los resultados del experimento, es decir, se calculó el promedio de los resultados de los 30 archivos de *benchmark* para cada algoritmo. Cabe destacar que estos números se ven mayormente influenciados por los problemas más grandes, ya que sus resultados tienen valores numéricos más altos. Sin embargo, esto proporciona una visión general del rendimiento de los algoritmos.

A continuación, se presentarán las gráficas comparativas que muestran los resultados promedio de los algoritmos en términos de tiempo de procesamiento y cantidad de cajas generadas. Estas gráficas permitirán visualizar las diferencias y similitudes entre los algoritmos en relación con estos dos aspectos importantes.

Además, se brindará una explicación detallada de cómo los cambios realizados en cada algoritmo afectaron sus resultados. Se analizarán las mejoras o limitaciones que surgieron a partir de los cambios implementados, y se evaluará su impacto en el desempeño global de los algoritmos.

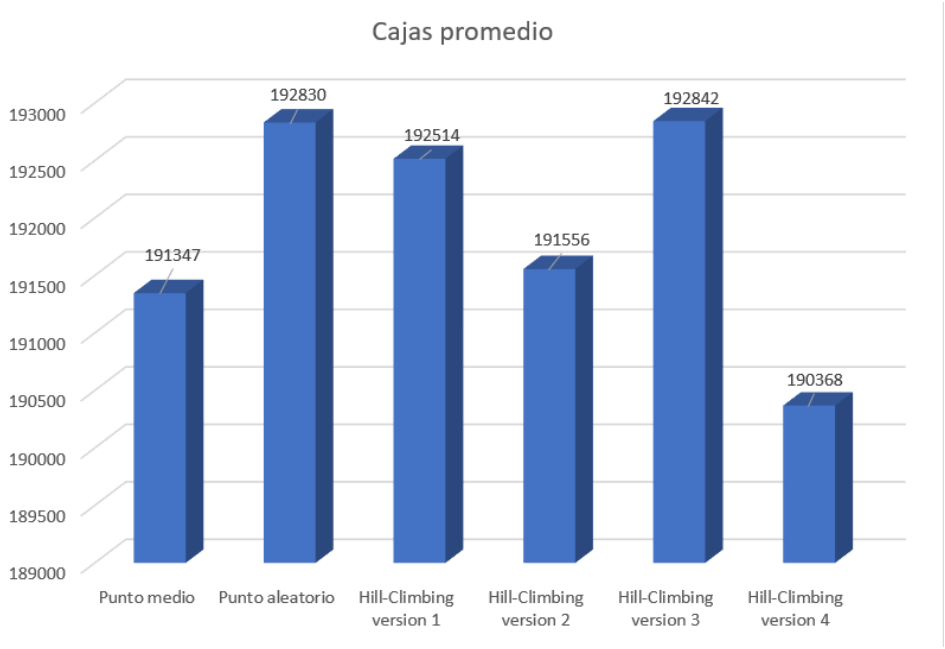

Figura 5.1. Cantidad de cajas promedio generadas de los algoritmos

La gráfica de cajas promedio en la Figura 5.1 revela que la mayoría de los algoritmos generaron más cajas que el algoritmo de punto medio. Es importante destacar que los algoritmos que utilizan el operador de selección de primera-mejora no muestran cambios significativos en la cantidad de cajas en comparación con el punto aleatorio. Esto indica que dicho operador, que selecciona el primer vecino mejor que el punto actual, no genera mejoras significativas que impacten en los resultados finales. En otras palabras, los resultados obtenidos son similares a los que se obtendrían seleccionando el punto de manera aleatoria, pero con un mayor consumo de recursos.

Por otro lado, el operador de selección mejor-mejora muestra resultados similares al punto medio cuando se evalúa en una sola restricción, y reduce la cantidad de cajas generadas cuando se evalúan los puntos en todas las restricciones. Esto demuestra que la selección del punto de expansión tiene un impacto en la cantidad de cajas generadas por el árbol de búsqueda. Además, se evidencia que las mejoras realizadas con el operador de selección primera-mejora no son suficientes para influir en el resultado final. Es decir, la evaluación del punto de expansión en todas las restricciones se presenta como un enfoque para mejorar la eficiencia del algoritmo de *Branch and Bound*.

En resumen, los resultados muestran que el operador de selección mejor-

mejora, junto con la evaluación del punto de expansión en todas las restricciones, permite una reducción en la cantidad de cajas generadas y, por lo tanto, una mejora en la eficiencia del algoritmo de *Branch and Bound* en comparación con el uso del operador de selección primera-mejora y la selección aleatoria de puntos.

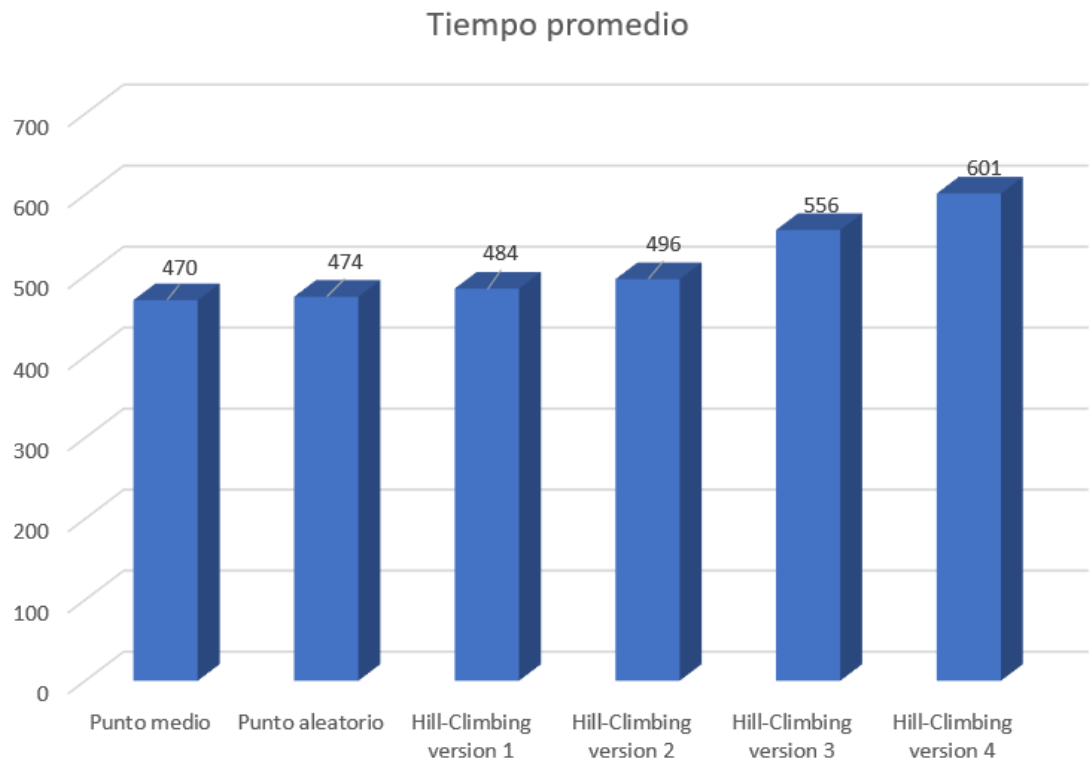

Figura 5.2. Tiempo promedio en segundos de los algoritmos

Al analizar la gráfica de tiempos de procesamiento en la Figura 5.2, se puede observar claramente que hay un aumento en el tiempo debido a la cantidad de recursos o iteraciones adicionales que se realizan para mejorar el punto de expansión en los algoritmos *Hill-Climbing*. Es importante destacar que la operación más costosa en términos de tiempo es la evaluación de los puntos en todas las restricciones, seguida de la generación del vecindario para la siguiente iteración.

A partir de estos resultados, se puede concluir que incluso un pequeño cambio realizado en cada nodo del árbol puede tener un impacto negativo en el tiempo total requerido para encontrar el óptimo. Una posible solución a este problema sería evitar la ejecución del algoritmo en cada nodo, y en

su lugar, permitir que los nodos hereden el punto de expansión obtenido previamente. Dado que el espacio de búsqueda de cada nodo es una versión reducida del nodo padre, es factible utilizar el mismo punto de calidad encontrado anteriormente, evitando así el costo computacional adicional en cada nodo.

Esta optimización puede contribuir significativamente a reducir el tiempo de ejecución y mejorar la eficiencia del algoritmo en la búsqueda del óptimo global.

5.7.2. Análisis específico

En esta sección se realizará una división de los archivos utilizados en función de su tiempo de procesamiento, con el objetivo de analizar el desempeño de los diferentes algoritmos según el tamaño del problema. Los archivos serán clasificados en categorías de acuerdo a su duración de ejecución, dividiéndolos en aquellos con tiempos inferiores a 1 minuto, inferiores a 5 minutos, inferiores a 15 minutos y superiores a 15 minutos. Esto permitirá realizar un análisis más detallado y comparar el rendimiento de los algoritmos en cada categoría.

Menos de 1 minuto

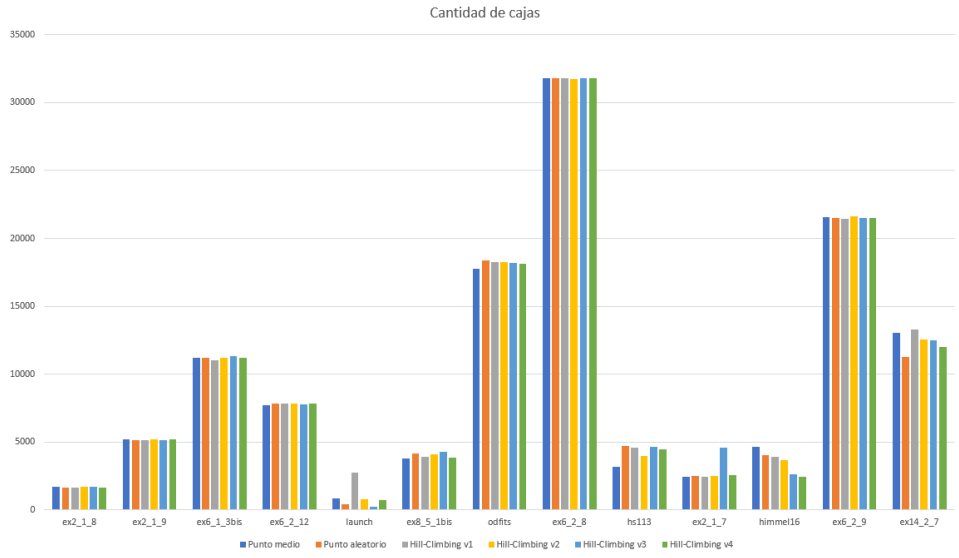

Figura 5.3. Cajas promedio de los algoritmos en cada archivo

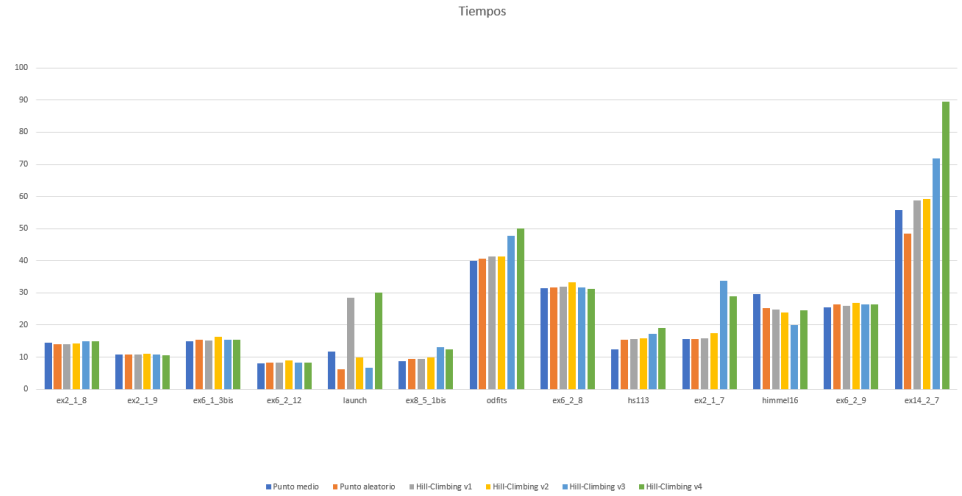

Figura 5.4. Tiempos promedio en segundos de los algoritmos en cada archivo

En relación con las Figuras 5.3 y 5.4, se puede observar que en los archivos de problemas pequeños, los algoritmos de punto aleatorio, *Hill-Climbing* v1 y *Hill-Climbing* v2 demuestran ser bastante competitivos en términos de tiempo y cantidad de cajas. Es destacable el desempeño del algoritmo de punto aleatorio, ya que en la mayoría de los casos obtiene valores iguales o inferiores tanto en tiempo como en cantidad de cajas, con la excepción del archivo "*hs*113" que muestra resultados más desfavorables en cuanto a tiempo y cantidad de cajas.

Por otro lado, tenemos el *Hill-Climbing* v3, que si bien logra reducir considerablemente la cantidad de cajas en algunos archivos como "*launch*" e "*himmel*16", en otros archivos su desempeño es igual o peor que los demás algoritmos. Además, su tiempo de ejecución aumenta significativamente para obtener resultados similares. En cuanto al *Hill-Climbing* v4, aunque logra disminuir la cantidad de cajas en la mayoría de los archivos, el beneficio no es significativo en comparación con el incremento considerable en tiempo de procesamiento, llegando a aumentar casi el doble en algunos casos.

En resumen, los primeros tres algoritmos se muestran más estables en términos de tiempo y cantidad de cajas, siendo el algoritmo de punto aleatorio el más destacado. Por otro lado, el *Hill-Climbing* v3 presenta un rendimiento inferior en archivos pequeños, y el último algoritmo, a pesar de la mejora en la cantidad de cajas, no justifica su uso debido al aumento significativo en el tiempo de ejecución.

Menos de 5 minutos

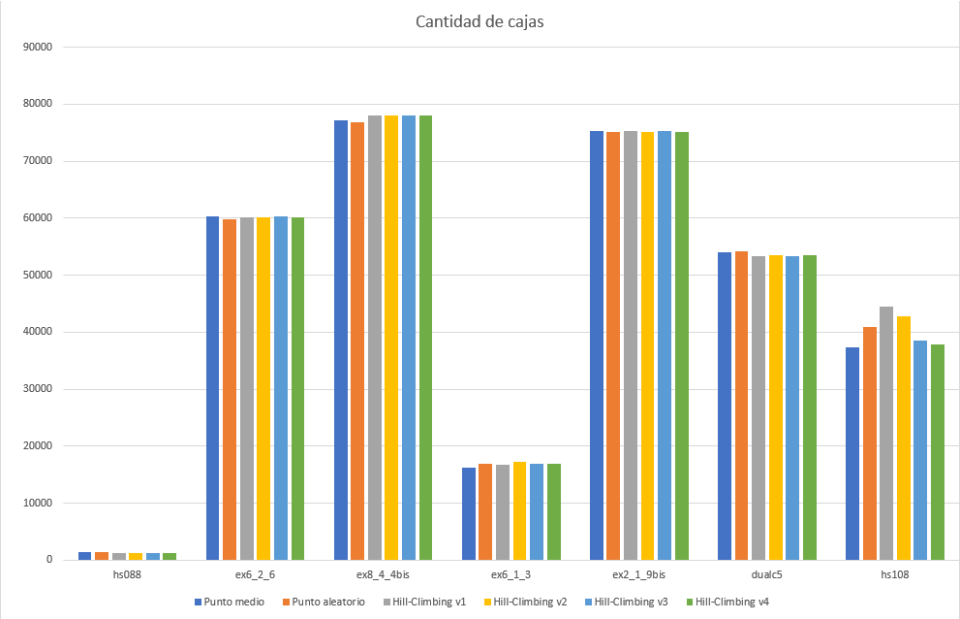

Figura 5.5. Cajas promedio de los algoritmos en cada archivo

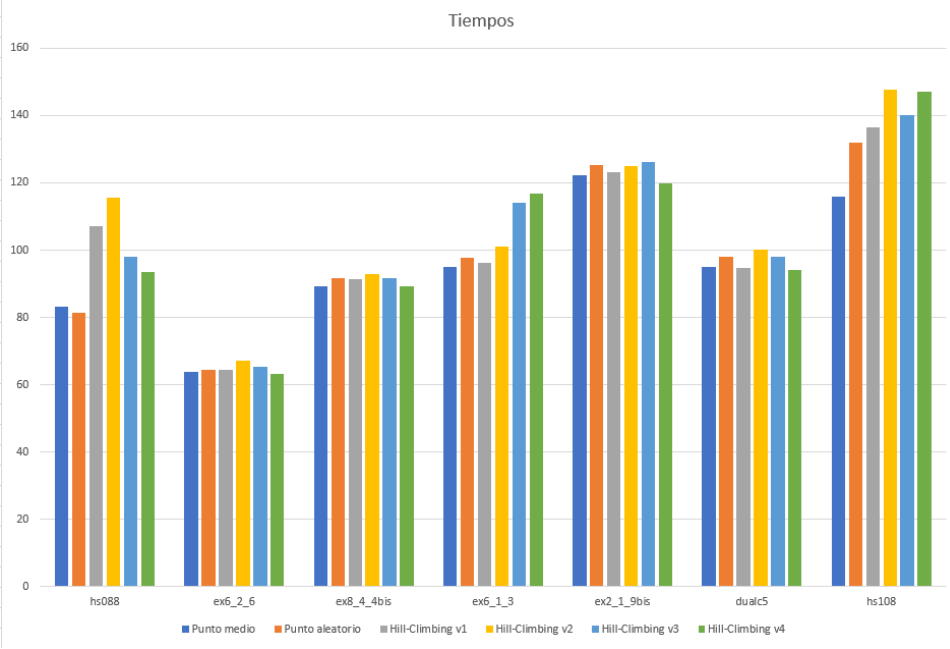

Figura 5.6. Tiempo promedio en segundos de los algoritmos en cada archivo

En relación con las Figuras 5.5 y 5.6, se puede observar que el algoritmo de punto aleatorio muestra tiempos similares al punto medio en la mayoría de los archivos de *benchmark*. Sin embargo, en algunos casos, como "*ex*8_4_4*bis*" y "*hs*108", el tiempo es ligeramente mayor. En cuanto a la generación de cajas, también se encuentra en niveles comparables al punto medio.

Por otro lado, los algoritmos *Hill-Climbing* v1 y *Hill-Climbing* v2 muestran tiempos variables en los archivos de *benchmark*, pero en general, tienden a ser mayores en comparación con el punto medio y el punto aleatorio. Además, la cantidad de cajas generadas es relativamente alta en algunos casos, como "*hs*108", lo que indica una mayor ineficiencia en términos de selección de puntos de expansión.

En cuanto a *Hill-Climbing* v3 y *Hill-Climbing* v4, demuestran un buen tiempo de procesamiento en general, excepto en "*ex*6_1_3" donde presentan un mayor tiempo en comparación con los demás algoritmos. Estos algoritmos evalúan los puntos en todas las restricciones, lo que les proporciona buenos resultados en comparación con los otros algoritmos. En términos de la cantidad de cajas generadas, se observa que ambos algoritmos producen una cantidad similar de cajas al punto medio, lo que indica un buen desempeño en este aspecto, aun dicho esto sigue sin superar en eficiencia al algoritmo

de punto medio, siendo este el más estable en ambas métricas.

Menos de 15 minutos

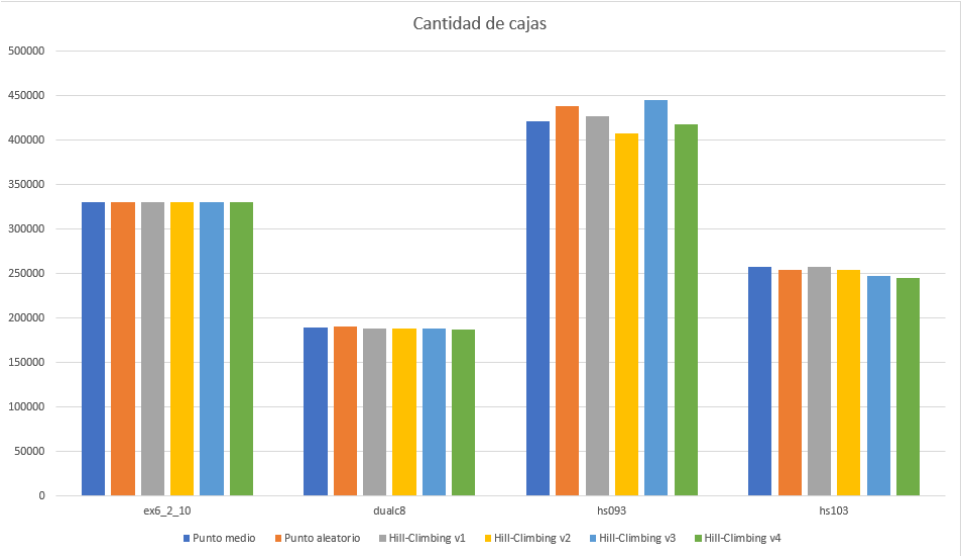

Figura 5.7. Cajas promedio de los algoritmos en cada archivo

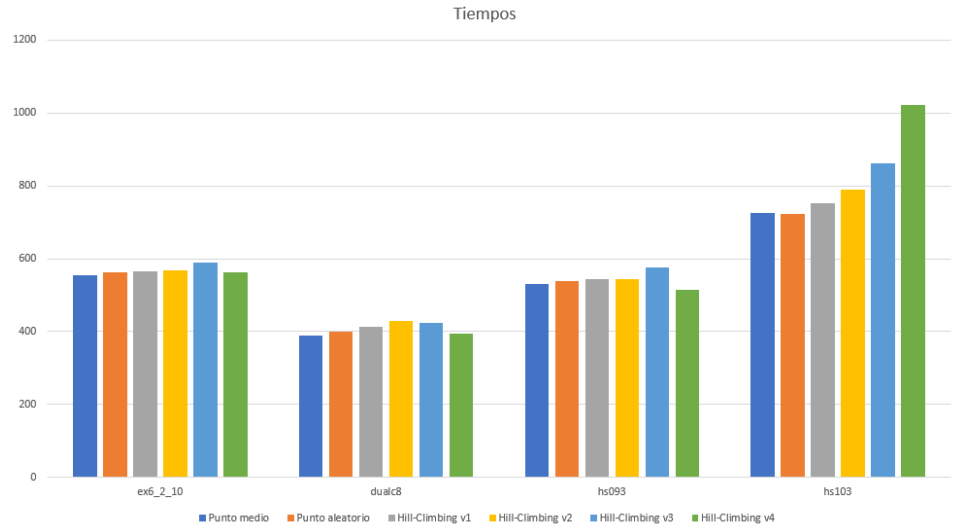

Figura 5.8. Tiempo promedio en segundos de los algoritmos en cada archivo

Al analizar las gráficas de cajas promedio y tiempo promedio para los problemas de mayor tamaño, como se muestra en las Figuras 5.7 y 5.8, se observa que la cantidad de cajas generadas comienza a ser significativamente alta, lo que reduce la influencia de la aleatoriedad en los resultados. En este escenario, el algoritmo de punto aleatorio genera la mayor cantidad de cajas, pero con un tiempo similar al del algoritmo de punto medio. Un caso particular es el problema "$hs093$", donde el algoritmo de punto aleatorio genera alrededor de 17 mil cajas adicionales, lo que indica que deja de ser eficiente la aleatoriedad en problemas de mayor tamaño.

Para los algoritmos *Hill-Climbing* v1 y *Hill-Climbing* v3, que comparten el operador de selección primera-mejora, se observa que el primero genera igual o mayor cantidad de cajas en todos los problemas, y además requiere más tiempo de procesamiento. El segundo algoritmo también genera más cajas, como en el caso del problema "$hs093$", donde produce cerca de 24 mil cajas adicionales en comparación con el algoritmo de punto medio. Esto indica que la selección primera-mejora no trabaja lo suficientemente en la mejora del punto de expansión para beneficiar al algoritmo.

Por otro lado, los algoritmos *Hill-Climbing* v2 y *Hill-Climbing* v4 presentan un buen desempeño, ya que en todos los problemas generan igual o menos cajas que el algoritmo de punto medio. Por ejemplo, en el problema "$hs093$", la versión 2 reduce cerca de 13 mil cajas en casi el mismo tiempo, mientras que la versión 4 reduce 3 mil cajas y alcanza el óptimo en menos

tiempo. Es importante destacar que aunque en la mayoría de los casos estos algoritmos tienen tiempos casi iguales, con una ligera diferencia, y una mejor cantidad de cajas generadas, en el problema "*hs*103" el *Hill-Climbing* v4 requiere considerablemente más tiempo que el resto, demorándose aproximadamente 10 minutos más que el algoritmo de punto medio y reduciendo la cantidad de cajas en 13 mil. Esto muestra que el *Hill-Climbing* v4 aún puede ser inestable en términos de tiempo de procesamiento para algunos problemas, pero el enfoque de búsqueda de punto de expansión empieza a beneficiar.

Más de 15 minutos

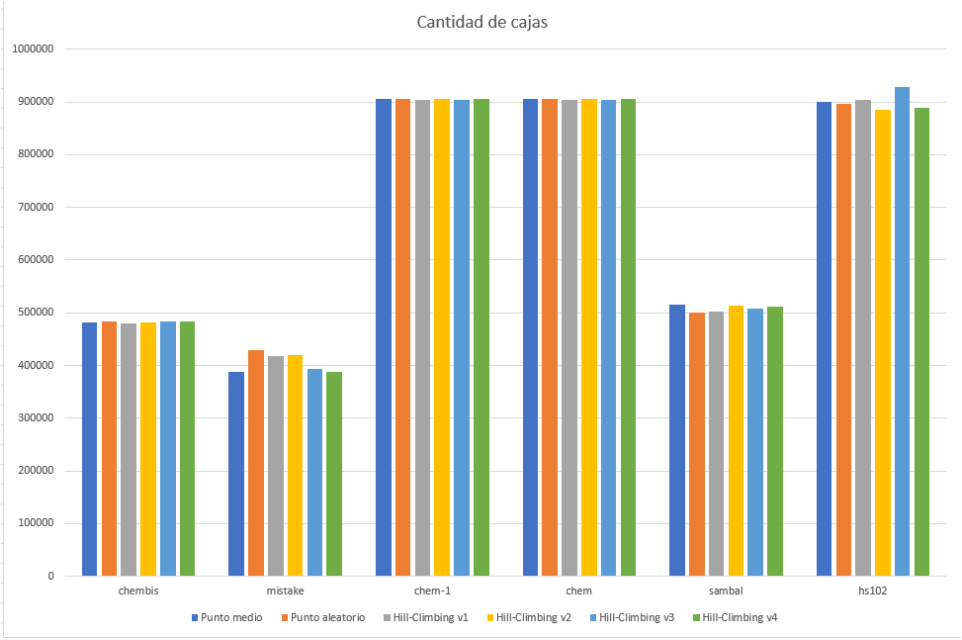

Figura 5.9. Cajas promedio de los algoritmos en cada archivo

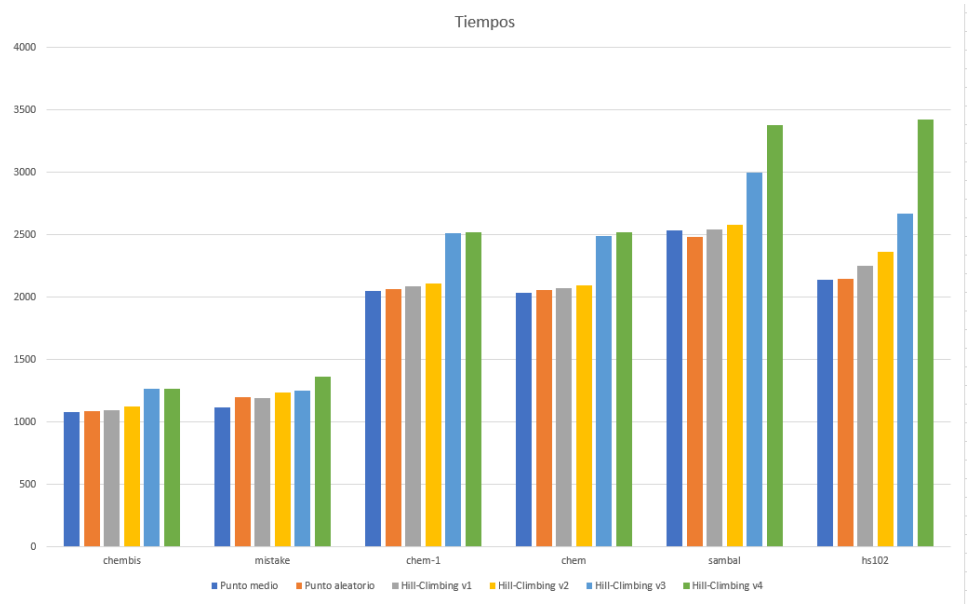

Figura 5.10. Tiempo promedio en segundos de los algoritmos en cada archivo

Al analizar los resultados de los problemas de mayor tamaño, representados en la gráfica de cajas promedio de la Figura 5.9, se observa un patrón similar al de los problemas de menos de 15 minutos. Los algoritmos *Hill-Climbing* v2 y v4 muestran una mejora en la reducción de la cantidad de cajas generadas, mientras que los demás algoritmos no logran mejorar los resultados significativamente.

Sin embargo, al examinar la gráfica de tiempo promedio en la Figura 5.10, se evidencia claramente el principal defecto de los algoritmos propuestos en la investigación. El tiempo de procesamiento presenta una forma escalonada, lo que indica que ejecutar el algoritmo *Hill-Climbing* en cada nodo del árbol de búsqueda conlleva una enorme cantidad de tiempo. Además, los pequeños cambios implementados en los distintos algoritmos *Hill-Climbing*, al trabajar con una gran cantidad de nodos, amplifican aún más el tiempo requerido para encontrar la solución óptima. Esto revela que el principal problema de los algoritmos implementados es la alta cantidad de ejecuciones realizadas sobre el problema para determinar los puntos de expansión.

Capítulo 6

Conclusiones

Para dar término al documento, se enunciarán las conclusiones del proceso de investigación, donde se abordarán las dificultades encontradas, el aporte de este proyecto, entre otros. Finalmente, en la Sección de Trabajo a futuro, se plantearán ideas de cómo se podría proseguir este proyecto, que, si bien, se tuvieron en consideración al momento iniciar este proceso, fueron descartadas para los alcances de este documento.

6.1. Conclusión

A lo largo de este documento se ha planteado la idea de implementar la metaheurística de trayectoria *Hill-Climbing* para una selección dinámica del punto de expansión utilizado por el método de aproximación Abstaylor dentro del algoritmo *Branch and bound* para la resolución de problemas NCOP. Para lo anterior, se definen los lineamientos iniciales en el Capítulo 1, los que son necesarios para no perder el sentido en la investigación, definiendo el contexto y la motivación, abarcando conceptos claves que se utilizaron a lo largo del texto. Una solución preliminar, la que permite abstraer los problemas de manera general para luego diseñarlo e implementarlo a lo que se estudia en el Capítulo 2. En este capítulo se detallan las principales contribuciones y soluciones propuestas en la literatura sobre las metaheurísticas y los problemas NCOP a la fecha de este documento, aportando el conocimiento necesario para que tanto el diseño como la implementación sea lo

correspondiente a lo estudiado. Luego, se plantea una estrategia de trabajo que se emplea durante la investigación, donde se indica como se desarrollara y que es lo que se estudiara a detalle. Con toda la teoría en mente, se procede a generar, construir, crear el diseño de la solución (Capítulo 4) propuesta para la implementación del *Hill-Climbing* dentro del algoritmo de *Branch and bound*, donde se presentan los métodos, herramientas y tecnologías utilizadas para los experimentos plasmados en el Capítulo 5 donde se presenta toda la información experimental obtenida por cada implementación según el problema, para luego, analizar los resultados obtenidos.

En general, los resultados esperados se cumplieron parcialmente, ya que se logró adaptar e implementar el algoritmo *Hill-Climbing* para una selección dinámica del punto de expansión del método Abstaylor y establecer una comparativa entre el algoritmo original y la solución propuesta, mostrando los problemas y beneficios que tiene la implementación, pero no se logra una mejorar en la eficiencia del algoritmo original siendo esta la hipótesis de este trabajo. Sin embargo, esta investigación representó un primer acercamiento para explorar diferentes enfoques y evaluar su funcionamiento. Se realizaron diferentes niveles de rigurosidad para trabajar el punto de expansión, pudiendo identificar los métodos o caminos que podrían ser útiles en el desarrollo de la idea y comprobar su funcionamiento.

Se pudo confirmar que una selección dinámica del punto de expansión sí tiene el potencial de reducir la cantidad de cajas generadas. Además, se identificó que el principal problema en las implementaciones realizadas fue la gran cantidad de ejecuciones que se realizaban dentro del árbol de búsqueda, lo cual resultaba en tiempos de procesamiento significativamente altos para alcanzar las soluciones óptimas requeridas.

El funcionamiento de la implementación en C++ fue satisfactorio en general, ya que no provoco errores en el funcionamiento general del algoritmo y se logró adaptar de manera correcta al ambiente de intervalos que se utilizaba. Sin embargo, se enfrentaron algunos problemas relacionados con el entorno de ejecución que ocasionaron errores en la aproximación de algunos números. Para abordar esta situación, se optó por utilizar Docker, lo cual permitió cambiar rápidamente el entorno y resolver esos problemas.

En cuanto a la librería Ibex, se encontró que la documentación disponible en la página web era limitada y se enfocaba principalmente en los aspectos de instalación. Esto generó la necesidad de llevar a cabo un estudio minucioso del código fuente, examinando clases, variables y funciones para obtener una

comprensión más profunda de la estructura y el funcionamiento del algoritmo *Branch and Bound*.

Es importante mencionar que el proceso de obtención de resultados fue algo extenso debido a la complejidad de los problemas abordados y a la variedad de versiones y algoritmos propuestos. No obstante, el uso de un programa de automatización contribuyó significativamente a agilizar este proceso, haciendo que fuera más manejable y eficiente.

6.2. Trabajo a futuro

Como trabajo futuro, una posible mejora para abordar el problema del alto tiempo de procesamiento de los algoritmos implementados sería optimizar la cantidad de ejecuciones realizadas dentro del algoritmo *Branch and Bound*. Una estrategia prometedora es permitir que los nodos hereden los puntos de expansión de sus nodos padres.

La idea es ejecutar inicialmente un algoritmo que entregue un punto de expansión de alta calidad para el nodo raíz. Luego, a medida que se generan los nodos hijos, dado que comparten un espacio de búsqueda reducido en una de sus variables, es posible que también puedan utilizar el mismo punto de expansión que su nodo padre. Esto reduciría significativamente la cantidad de ejecuciones del algoritmo necesario para buscar puntos de expansión de alta calidad. El algoritmo solo se ejecutaría cuando no sea posible reutilizar el punto heredado debido a la reducción del dominio del nodo.

Para implementar esta mejora, sería necesario realizar un estudio exhaustivo y realizar modificaciones en la librería Ibex, especialmente en la clase de nodos, para comprender adecuadamente el proceso de creación de nodos y determinar en qué momento se realiza este proceso. Esta optimización requeriría un enfoque cuidadoso y una comprensión profunda de la estructura y el funcionamiento de la librería Ibex.

Referencias bibliográficas

[1] Ignacio Araya and Victor Reyes. Interval branch-and-bound algorithms for optimization and constraint satisfaction: a survey and prospects. *Journal of Global Optimization*, 65:837–866, 2016.

[2] Tao Ding, Rui Bo, Fangxing Li, and Hongbin Sun. A bi-level branch and bound method for economic dispatch with disjoint prohibited zones considering network losses. *IEEE Transactions on Power Systems*, 30(6):2841–2855, 2015.

[3] Amir Hossein Gandomi, Xin-She Yang, Siamak Talatahari, and Amir Hossein Alavi. Metaheuristic algorithms in modeling and optimization. *Metaheuristic applications in structures and infrastructures*, 1, 2013.

[4] Amir Hossein Gandomi, Xin-She Yang, Siamak Talatahari, and Amir Hossein Alavi. Metaheuristic algorithms in modeling and optimization. *Metaheuristic applications in structures and infrastructures*, 1, 2013.

[5] A. Jalilvand and S. Khanmohammadi. Task scheduling in manufacturing systems based on an efficient branch and bound algorithm. In *IEEE Conference on Robotics, Automation and Mechatronics, 2004.*, volume 1, pages 271–276 vol.1, 2004.

[6] David S Johnson, Christos H Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of computer and system sciences*, 37(1):79–100, 1988.

[7] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm- a literature review. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 380–384, 2019.

[8] S.J. Liu, Y.P. Yang, and Y.S. Jiang. Terrain page scheduling in flight simulation based on branch-and-bound search. In *IET International Conference on Information Science and Control Engineering 2012 (ICISCE 2012)*, pages 1–5, 2012.

[9] Jordan Ninin. Global optimization based on contractor programming: An overview of the ibex library. In *Mathematical Aspects of Computer and Information Sciences: 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers 6*, pages 555–559. Springer, 2016.

[10] Victor Reyes and Ignacio Araya. Abstaylor: upper bounding with inner regions in nonlinear continuous global optimization problems. *Journal of Global Optimization*, 79:413–429, 2021.

[11] Gilles Trombettoni, Ignacio Araya, Bertrand Neveu, and Gilles Chabert. Inner regions and interval linearizations for global optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):99–104, Aug. 2011.

[12] William T. Vetterling Brian P. Flannery. William H. Press, Saul A. Teukolsky. *Numerical Recipes: The Art of Scientific Computing (3rd ed.)*. 2007.

UNIVERSIDAD DIEGO PORTALES

ANEXOS

Anexo A

Código Clase Hill-Climbing en C++

```
#include <vector>
#include "ibex_IntervalVector.h"
#include "ibex_System.h"
#include <iostream>
namespace ibex {
    class HillClimbing{
        public:
            HillClimbing(const IntervalVector& box, const
                System& sys);

            Vector v1(const IntervalVector& box);
            Vector v2(const IntervalVector& box);
            Vector v3(const IntervalVector& box);
            Vector v4(const IntervalVector& box);
        private:
            const IntervalVector& box;
            const System& sys;

    };
}
```

Anexo B

Código Versión 1 Hill-Climbing en C++

```
Vector HillClimbing::v1(const IntervalVector& box){
        IntervalVector inicial(box.size());
        int sizee=box.size();
        double restriccion1=RNG::rand(0,this->sys.nb_ctr
           -1); // Escoger restriccion aleatoria
        int restriccion=std::round(restriccion1);
        //------------Solucion inicial-----------------
        for(int i=0;i<box.size();i++){
            double random_num = RNG::rand(box[i].lb(),box
               [i].ub());
            inicial[i]=Interval(random_num,random_num);
        }
        //---------------------------------------------
        Interval mejor=this->sys.f_ctrs[restriccion].eval
           (inicial);  // Evaluar la restriccion en la
           solucion inicial
        bool cambio=true;
        //------------Generar vecindario----------------
        IntervalVector vecino=inicial;
        int iter=0;
        while( cambio && iter<50 ){
            cambio=false;
            for(int i=0;i<inicial.size();i++){
            //--------------Operador de movimiento
```

```
            ----------------
        double ruido= RNG::rand(0,1);
        double magnitud= (box[i].ub()-box[i].lb())
            *0.1;
        ruido=ruido*magnitud;
        vecino[i]=Interval(inicial[i].lb()+ruido,
            inicial[i].ub()+ruido);
        //--------------funcion de evaluacion
            ----------------
        if(fabs(this->sys.f_ctrs[restriccion].eval(
            vecino).mid())<fabs(mejor.mid())){
        //
            ------------------------------------------------

             mejor=this->sys.f_ctrs[restriccion].eval(
                 vecino);
             inicial=vecino;
             cambio=true;
             break;
        }else{
             vecino=inicial;
        }
    }
      iter++;
    }
    //-----------------------------------------------
    return inicial.mid();
}
```

Anexo C

Código Versión 2 Hill-Climbing en C++

```
Vector HillClimbing::v2(const IntervalVector& box){
        IntervalVector inicial(box.size());
        int sizee=box.size();
        IntervalVector mejor_vecino(box.size());
        double restriccion1=RNG::rand(0,this->sys.nb_ctr
            -1); // Escoger restriccion aleatoria
        int restriccion=std::round(restriccion1);
        //------------Solucion inicial-----------------
        for(int i=0;i<box.size();i++){
            double random_num = RNG::rand(box[i].lb(),box
                [i].ub());
            inicial[i]=Interval(random_num,random_num);
        }
        //---------------------------------------------
        Interval mejor=this->sys.f_ctrs[restriccion].eval
            (inicial);  // Evaluar la restriccion en la
            solucion inicial
        bool cambio=true;
        //------------Generar vecindario---------------
        IntervalVector vecino=inicial;
        int iter=0;
        while( cambio && iter<50 ){
            cambio=false;
            for(int i=0;i<inicial.size();i++){
```

```
            //--------------Operador de movimiento
                ----------------
            double ruido= RNG::rand(0,1);
            double magnitud= (box[i].ub()-box[i].lb())
                *0.1;
            ruido=ruido*magnitud;
            vecino[i]=Interval(inicial[i].lb()+ruido,
                inicial[i].ub()+ruido);
            //--------------funcion de evaluacion
                ----------------
            if(fabs(this->sys.f_ctrs[restriccion].eval(
                vecino).mid())<fabs(mejor.mid())){
            //
                ------------------------------------------------

                mejor=this->sys.f_ctrs[restriccion].eval(
                    vecino);
                mejor_vecino=vecino;
                vecino=inicial;
                cambio=true;
            }else{
                vecino=inicial;
            }
        }
            inicial=mejor_vecino;

            iter++;
        }
        //-----------------------------------------------
        return inicial.mid();
    }
```

Anexo D

Código Versión 3 Hill-Climbing en C++

```
Vector HillClimbing::v3(const IntervalVector& box){
        IntervalVector inicial(box.size());
        double suma=0;
        double mejor=0;
        int sizee=box.size();
        double restriccion1=RNG::rand(0,this->sys.nb_ctr
           -1); // Escoger restriccion aleatoria
        int restriccion=std::round(restriccion1);
        //------------Solucion inicial-----------------
        for(int i=0;i<box.size();i++){
            double random_num = RNG::rand(box[i].lb(),box
               [i].ub());
            inicial[i]=Interval(random_num,random_num);
        }
        //---------------------------------------------
        // Evaluar la restriccion en la solucion inicial
        for (int i=0;i<sys.nb_ctr;i++){
                mejor+=fabs(this->sys.f_ctrs[i].eval(
                   inicial).mid());
            }
        bool cambio=true;
        //------------Generar vecindario----------------
        IntervalVector vecino=inicial;
        int iter=0;
```

```
        while( cambio && iter<50 ){
            cambio=false;
            for(int i=0;i<inicial.size();i++){
            //--------------Operador de movimiento
                ----------------
            double ruido= RNG::rand(0,1);
            double magnitud= (box[i].ub()-box[i].lb())
                *0.1;
            ruido=ruido*magnitud;
            vecino[i]=Interval(inicial[i].lb()+ruido,
                inicial[i].ub()+ruido);
            //--------------funcion de evaluacion
                ----------------
            suma=0;
            for (int i=0;i<sys.nb_ctr;i++){
                suma+=fabs(this->sys.f_ctrs[i].eval(
                    vecino).mid());
            }
            if(suma<mejor){
            //
                ----------------------------------------------------

                mejor=suma;
                inicial=vecino;
                cambio=true;
                break;
            }else{
                vecino=inicial;
            }
        }
            iter++;
        }
        //-----------------------------------------------
        return inicial.mid();
    }
```

Anexo E

Código Versión 4 Hill-Climbing en C++

```
Vector HillClimbing::v4(const IntervalVector& box){
        IntervalVector inicial(box.size());
        IntervalVector mejor_vecino(box.size());
        double suma=0;
        double mejor=0;
        int sizee=box.size();
        double restriccion1=RNG::rand(0,this->sys.nb_ctr
           -1); // Escoger restriccion aleatoria
        int restriccion=std::round(restriccion1);
        //------------Solucion inicial-----------------
        for(int i=0;i<box.size();i++){
            double random_num = RNG::rand(box[i].lb(),box
               [i].ub());
            inicial[i]=Interval(random_num,random_num);
        }
        //---------------------------------------------
        // Evaluar la restriccion en la solucion inicial
        for (int i=0;i<sys.nb_ctr;i++){
                mejor+=fabs(this->sys.f_ctrs[i].eval(
                   inicial).mid());
            }
        bool cambio=true;
        //------------Generar vecindario----------------
        IntervalVector vecino=inicial;
```

```
        int iter=0;
        while( cambio && iter<50 ){
            cambio=false;
            for(int i=0;i<inicial.size();i++){
            //--------------Operador de movimiento
               -----------------
            double ruido= RNG::rand(0,1);
            double magnitud= (box[i].ub()-box[i].lb())
               *0.1;
            ruido=ruido*magnitud;
            vecino[i]=Interval(inicial[i].lb()+ruido,
               inicial[i].ub()+ruido);
            //--------------funcion de evaluacion
               -----------------
            suma=0;
            for (int i=0;i<sys.nb_ctr;i++){
                suma+=fabs(this->sys.f_ctrs[i].eval(
                   vecino).mid());
            }
            if(suma<mejor){
            //
               ----------------------------------------------------

                mejor=suma;
                mejor_vecino=vecino;
                vecino=inicial;
                cambio=true;
            }else{
                vecino=inicial;
            }
        }
            inicial=mejor_vecino;
            iter++;
        }
        //-----------------------------------------------
        return inicial.mid();
    }
```

Anexo F

Dockerfile

```
FROM ubuntu:latest


RUN apt-get update && apt-get install -y \
    bash \
    g++ \
    gcc \
    flex \
    bison \
    python2-dev \
    make \
    build-essential


CMD ["tail", "-f", "/dev/null"]
    }
```

Anexo G

Docker-Compose

```
version: '3'
services:
  myservice:
    build: .
    image: tesis
    container_name: tesis
    volumes:
      - ./ibex-lib-abst-iterative:/app
```

Anexo H

Github

https://github.com/Flyder78/MemoriaHill-Climbing