UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Diseño de Metaheurística para problemas combinatorios costosos

Autor
Irene Trigueros Lorca

Directores
Daniel Molina Cabrera
Francisco Herrera Triguero

Facultad de Ciencias

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Facultad de Ciencias

Granada, Mayo de 2023

UNIVERSIDAD DE GRANADA

Diseño de Metaheurística para problemas combinatorios costosos

Autor
Irene Trigueros Lorca

Directores
Daniel Molina Cabrera
Francisco Herrera Triguero

Diseño de Metaheurística para problemas combinatorios costosos

Irene Trigueros Lorca

Palabras clave: metaheurística, combinatorio, costoso, optimización, algoritmo genético, exploración, explotación, diversidad

Resumen

Existen problemas de mucho interés cuyo coste de evaluación es excesivamente elevado (a este tipo de problemas los llamaremos problemas costosos o problemas *expensive*), como podrían ser los problemas de optimización de redes neuronales, por lo que resulta interesante desarrollar algoritmos capaces de obtener soluciones competitivas en muy pocas evaluaciones. Además, en la literatura ya encontramos algoritmos específicos en problemas *expensives* de optimización continua; pero no ocurre lo mismo con el caso de problemas combinatorios, para los que no se han hallado ningún algoritmo capaz de resolverlos.

Por lo tanto, en este proyecto se propone un algoritmo metaheurístico nuevo para resolver problemas combinatorios costosos (*expensive*), siguiendo un formato semejante al de un diario de desarrollo. Se tomará como algoritmo base un algoritmo genético y, una vez analizado su rendimiento, se implementarán una serie de modificaciones que, progresivamente, compondrán la versión final del algoritmo. Estas modificaciones se justificarán observando y analizando los resultados obtenidos en los intentos anteriores con el objetivo de encontrar formas de aprovechar al máximo las pocas evaluaciones que nos podemos permitir en este tipo de problemas. Se describen detalladamente las tareas adicionales llevadas a cabo propias del desarrollo de un algoritmo de esta clase. Además, se realizarán análisis experimentales que irán demostrando que las sucesivas versiones del algoritmo van mejorando las anteriores. Las conclusiones que se alcanzan indican que el algoritmo presentado en este proyecto es altamente competitivo.

Metaheuristics design for expensive combinatorial problems

Irene Trigueros Lorca

Keywords: metaheuristic, combinatory, expensive, optimization, genetic algorithm, exploration, explotation, diversity

Abstract

There are very interesting problems which their computational cost is excessively high (these kind of problems will be refered as expensive problems), such as the neural network optimization problems, therefore it is compelling to develope algorithms that are able to obtain competitive solutions in few evaluations. Furthermore, specific algorithms for the optimization of expensive continous problems can be found in the literature; but that is not the case for the combinatory expensive problems, for those which no algorithms able to solve them have been found.

Hence, throughout this project, an original metaheuristic algorithm is going to be proposed with the purpose of solving expensive problems, following a format similar to that of a developer diary. A genetic algorithm is going to be chosen as the base algorithm and, once its performance is analyzed, a series of modifications are going to be implemented, which, at the end, will compose the final version of the algorithm. These modifications are going to be justified by observing and analyzing the results obtained through the previous attempts with the objective of finding ways of making the most of the few iterations available in these kind of problems. Additional tasks regarding the development of this type of algorithms are described in depth. Furthermore, experimental analysis is included, where both base and the following versions algorithms are compared. The final conclusions suggest that the algorithm presented within this project is highly performant.

Yo, **Irene Trigueros Lorca**, alumno de la titulación Doble Grado en Ingeniería Informática y Matemáticas de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada** y de la **Facultad de Ciencias**, con DNI 77385991F, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca de ambos centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Irene Trigueros Lorca

Granada a, 17 de junio de 2023.

D. **Daniel Molina Cabrera**, Profesor del Departamento Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

D. **Francisco Herrera Triguero**, Profesor del Departamento Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Diseño de Metaheurística para problemas combinatorios costosos***, ha sido realizado bajo su supervisión por **Irene Trigueros Lorca**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 17 de junio de 2023.

Los directores:

Daniel Molina Cabrera **Francisco Herrera Triguero**

Agradecimientos

Poner aquí agradecimientos...

Índice general

Índice de figuras

Capítulo 1: Introducción

Cuando se debe afrontar un problema, la variedad de aproximaciones a seguir resulta ser muy amplia y su elección depende en gran medida de dos factores fundamentales: qué clase de solución se desea extraer del problema, y de qué recursos se dispone para ello. Por ello, la rama más clásica de la computación siempre ha tratado de resolver los problemas presentados de forma exacta. Es decir, ha tratado cada problema como si solo existiera una sola solución al mismo, la óptima. Esta forma de pensamiento se basa en la confianza que se tiene en que los problemas que tradicionalmente eran irresolubles para los humanos, serían más accesibles para los computadores, gracias a su capacidad superior de cómputo pesado.

Esto último resulta cierto en muchos escenarios, fundamentalmente en lo referente a los problemas más puramente matemáticos: operaciones que un humano podría tardar años en resolver a mano, un ordenador podría resolverlas en cuestión de minutos. Sin embargo, la mayoría de problemas que nos encontramos en el mundo real son complejos y difíciles de resolver, lo que implica que no se pueda dar con la solución óptima en un tiempo razonable.

Uno de los problemas más conocidos capaz de ilustrar este hecho es el **Problema del Viajante de Comercio**. Este problema consiste en que, dado un conjunto de ciudades por las que el comercial debe pasar, se debe encontrar el orden en que visita las ciudades de forma que el comercial recorra la menor distancia posible. Aunque sea un problema sencillo de formular, la cantidad de posibles caminos incrementa en gran medida a la vez que el número de ciudades a recorrer aumenta. Por tanto, al aumentar el tamaño del problema, toda técnica conocida para extraer la solución exacta requeriría de un tiempo de ejecución que deja de ser asequible incluso para los ordenadores. Es decir, tenemos métodos que nos llevan a la solución, pero no existen formas de ejecutarlos.

Por ello, tenemos que encontrar alternativas más viables para la resolución de los problemas, es decir, tenemos que usar algoritmos aproximados, que proporcionan buenas soluciones (no necesariamente la óptima) en un tiempo razonable. Esto es, haremos uso de estrategias de diseño generales para procedimientos heurísticos de resolución de problemas: las metaheurísticas.

Los problemas de optimización costosa (EOP) o **problemas *expensives*** se refieren a los problemas que requieren costes elevados, o incluso inasequibles, con el fin de evaluar los candidatos a soluciones. Este tipo de problemas existen en una gran cantidad, y cada vez con más frecuencia, de aplicaciones significativas del mundo real. Un tipo de problemas *expensive* podrían ser los problemas de optimización de redes neuronales profundas utilizando metaheurísticas. Un ejemplo esto se podría encontrar en el artículo [2]; en ese estudio se proponen y se comparan tres métodos híbridos en combinación con el popular clasificador con redes neuronales para el modelado de incendios forestales.

También, cabe destacar que "coste elevado" es más un concepto relativo que uno absoluto en la mayoría de problemas del mundo real. Por ejemplo, en situaciones dinámicas, como podría ser recalcular una ruta porque se haya cortado una calle o porque se ha producido un atasco, o incluso

situaciones de emergencia como epidemias o desastres naturales, transporte y envío de materiales para operaciones diarias importantes para salvar vidas, etc., el coste de optimización en situaciones normales se convierte en un coste demasiado elevado. Para el caso de los problemas de parámetros reales nos encontramos con que se están planteando cada vez más algoritmos especialmente diseñados para problemas *expensive* (por ejemplo, que la función de evaluación dependa de una simulación).

Ahora bien, cada vez es más común usar problemas combinatorios en problemas complejos, lo que implica un mayor coste de evaluación. Un ejemplo de esto lo podemos encontrar en el artículo [5], donde en su introducción se detalla el por qué el *Well Placement Optimization Problem* (WPOP) en el desarrollo y gestión de campos petroleros se considera un problema *expensive* (se debe al cálculo tan complejo de ecuaciones diferenciales para predecir la influencia de la estrategia de producción en las propiedades geológicas y petrofísicas de la reserva). Si bien hemos comentado que en el caso de parámetros reales (caso continuo) se han propuesto algoritmos específicos, esto no ha sido el caso para el ámbito de los problemas combinatorios *expensive*, para los que no se han encontrado ninguna referencia. Por lo que es de gran interés crear un algoritmo que resulte útil para este tipo de situaciones, con el fin de reducir los costes todo lo posible.

En este Trabajo de Fin de Grado vamos a diseñar, implementar y proponer un algoritmo especialmente diseñado para problemas combinatorios *expensive*. El objetivo principal de este algoritmo será encontrar buenas soluciones en una cantidad de tiempo bastante reducida, lo que vamos a traducir en un menor número de iteraciones. Para ello, procederemos a tomar un problema sobre el que trabajar y extraer conclusiones y un algoritmo de referencia sobre el que realizaremos un proceso recursivo: introduciremos alguna modificación y compararemos los resultados obtenidos con los del algoritmo de referencia, en caso de mejorarlos, este algoritmo con modificación se convertirá en el nuevo algoritmo de referencia. De esta forma se puede garantizar que los resultados que finalmente alcanzaremos son competitivos.

Capítulo 2: Estimación y Presupuesto

En este capítulo se detalla cómo se ha organizado el trabajo y el tiempo dedicado a cada una de ellas. El orden en el que se han realizado dichas tareas, queda representado en un Diagrama de Gantt () para su mejor comprensión. Además, se realiza una estimación del presupuesto necesario para desarrollar el proyecto.

2.1. Planificación

La planificación previa de este proyecto se ha realizado siguiendo una metodología ágil. Es decir, la planificación se va adaptando dependiendo cómo hayan transcurrido las tareas anteriores. Es el modelo de planificación que más se ajusta a este tipo de trabajo, ya que, *a priori*, se desconoce la dificultad de las tareas a realizar.

Sin embargo, es cierto que antes de empezar el trabajo se estableció una planificación base bastante amplia para poder asegurar que se iba a finalizar el proyecto a tiempo.

2.1.1. Planificación Base

La planificación inicial que se estableció antes de iniciar el proyecto se puede expresar mediante la información representada en la tabla 2.1:

Tabla 2.1: Planificación Base

Resumen	**Tareas**	**Planificación**
Investigar el problema	Obtener instancias del problema Buscar algoritmos que lo resuelvan Buscar posibles implementaciones	Noviembre
Algoritmos de referencia	Elegir un algoritmo como referencia y estudiarlo Reducir el problema y meter equilibrio	Enero-Febrero
Experimentación	Formas de inicialización no aleatorias → Diseño experimental	2 semanas
Modificaciones	Propuesta de modificaciones Obtención de resultados	Marzo-Abril
Memoria	Escribir el informe	Mayo

2.1.2. Planificación Final

Tareas realizadas

Si bien es cierto que se han realizado una gran cantidad de tareas (sobre todo distintas modificaciones sobre algoritmos), con el fin de mantener la simplicidad, se han agrupado algunas tareas que tenían funciones similares. Esta agrupación también es de ayuda para simplificar la estimación del tiempo que se le ha dedicado a cada una de las tareas. Así, las tareas realizadas se resumen en la siguiente lista:

1. **Planteamiento y comprensión del problema**: Revisión del trabajo a realizar y reuniones con los tutores para proponer modificaciones y comprender mejor y aclarar todos los matices del Trabajo de Fin de Grado.
2. **Búsqueda de información y lecturas**: Búsqueda y lectura comprensiva de todos los artículos y documentos necesarios para la realización del proyecto.
3. **Planificación del proyecto**: Planificación de algunos aspectos que usar como base, así como las tareas que eran necesarias inicialmente. También hace referencia a partes de reuniones con los tutores para modificar las planificaciones (añadiendo o eliminando tareas) dependiendo del progreso alcanzado y los resultados obtenidos.
4. **Implementación de la propuesta inicial**: Implementación del código de los algoritmos base.
5. **Adaptación de los algoritmos base**: Modificación del código de los algoritmos base para adaptarlos al problema en cuestión.
6. **Modificación de los algoritmos**: Sucesivas modificaciones sobre el algoritmo base y los algoritmos que mejores resultados proporcionaban con el fin de mejorarlos aún más.
7. **Obtención de resultados**: Ejecución del código para obtener todos los resultados y cambiarlos de formato para su posterior análisis.
8. **Análisis de los resultados**: Interpretación de los resultados obtenidos.
9. **Revisión de la parte experimental**: Una vez dada por finalizada la parte experimental, se ha hecho una revisión exhaustiva de los códigos y de los resultados obtenidos.
10. **Elaboración de la memoria**: Desarrollo del informe.
11. **Revisión de la memoria**: Una vez terminado el trabajo, se ha hecho una revisión exhaustiva de la memoria.

Téngase en cuenta hay tareas que se han realizado casi simultáneamente, como serían la “Modificación de los algoritmos”, “Obtención de los resultados” y “Análisis de los resultados”. Esto se debe a la necesidad de saber cómo han influido las modificaciones para empezar a estudiar qué otra modificación podría ser beneficiosa. Por ejemplo, si se converge rápidamente a una solución, hay que estudiar por qué ha pasado y, una vez hecha la hipótesis, estudiar qué se podría modificar para que no suceda.

Una estimación del tiempo (en horas) dedicado a cada tarea se puede encontrar en la tabla 2.2.

Tiempo dedicado

Tabla 2.2: Tiempo dedicado

Actividad	**Duración (horas)**
Planteamiento y comprensión del problema	20
Búsqueda de información y lecturas	15
Planificación del proyecto	5
Implementación de la propuesta inicial	10
Adaptación de los algoritmos bases	5
Modificación de los algoritmos	200
Obtención de resultados	50
Análisis de los resultados	10
Revisión de la parte experimental	5
Elaboración de la memoria	170
Revisión de la memoria	10
Total	500

2.2. Presupuesto

Si quisiéramos valorar económicamente el proyecto, tenemos que tener en cuenta dos aspectos fundamentales: el precio de la mano de obra y el de cómputo como si tuviésemos que pagarlo. Además, también se incluirá el precio del ordenador en el que se ha realizado todo el trabajo. En la tabla 2.3 se puede ver un resumen de los cálculos realizados junto con el presupuesto final necesario.

El precio de la mano de obra son 25€ la hora.

El ordenador portátil usado para la realización de las ejecuciones ha sido un Asus Tuf Gaming A15 FA506IU-HN278 con un procesador AMD® Ryzen™ 7 4800H APU y 16GB (8GB×2) de RAM. Tras una breve búsqueda se puede comprobar que el precio actual de dicho portátil sería de 1300€.

Para calcular el coste de tiempo de cómputo es ha utilizado la calculadora de precios de Amazon Web Services (AWS Pricing Calculator). Elegiremos el servicio EC2, y dentro de eso elegimos las características más similares al ordenador utilizado; lo que nos lleva a la instancia c5d.2xlarge. Comprobamos el coste por hora del uso de dicho servicio y tenemos que este sería 0.48$/hora, es decir, aproximadamente 0.443€/hora.

Teniendo todos estos datos, podemos proceder a calcular cuál sería el presupuesto final de del proyecto:

$$450h \cdot 25\frac{€}{h} + 1300€ + 50h \cdot 0.443\frac{€}{h} = 12572.15€$$

Tabla 2.3: Cálculo del presupuesto

Concepto	**Precio base (€)**	**Tiempo (h)**	**Precio total (€)**
Mano de obra	25	450	11250
Ordenador	1300	—	1300
Tiempo de cómputo	0.443	50	22.15
Total			**12572.15**

En resumen, el **presupuesto de este proyecto queda fijado en 12572.15€**.

Capítulo 3: Repaso Bibliográfico

En esta sección se lleva a cabo una revisión bibliográfica sobre el tema en el que hemos centrado el proyecto. Buscamos con ello, llevar a cabo un pequeño recordatorio para poder entender los distintos ámbitos que más se han tratado y centrado los estudios en cuanto al diseño de metaheurísticas, las cuales se usan para resolver problemas cada vez más complejos. Posteriormente utilizaremos estos conocimientos para nuestro propio diseño de una metaheurística útil para problemas combinatorios *expensive*.

3.1. Contexto Bibliográfico

La computación evolutiva (*evolutionary computation*, EC) es un área de la ciencia de la computación que usa ideas de la evolución biológica para resolver problemas computacionales. La evolución es, en efecto, un método de búsqueda entre un número enorme de posibilidades de "soluciones" que permitan a los organismos sobrevivir y reproducirse en sus ambientes. También se puede ver la evolución como un método de adaptación a un entorno cambiante. En [1] nos podemos encontrar con un resumen del desarrollo de la computación evolutiva en el tiempo junto con aplicaciones reales de algoritmos evolutivos (tanto comerciales como científicas), como podría ser el uso de programación genética para mejorar estrategias óptimas de recolección. Si consideramos la computación evolutiva como un medio para encontrar buenas soluciones (aunque no sean las óptimas) dado un problema de optimización, es natural considerar que su hibridación con métodos de optimización existentes resultará en mejorar su rendimiento al explotar sus ventajas. Tales métodos de optimización se refieren desde algoritmos exactos estudiados en programación matemática [14] hasta algoritmos heurísticos hechos a medida para unos problemas dados. Los llamados algoritmos metaheurísticos también tienen un objetivo similar, y pueden ser combinados con EC, incluso si ambos enfoques suelen competir entre si. En el libro [22] se presentan varias posibilidades de combinar ECs con métodos de optimización, poniendo énfasis en la optimización de problemas combinatorios, tales como métodos *greedy*, construcciones heurísticas de soluciones factibles, programación dinámica, etc.

Una de las versiones de algoritmos evolutivos más utilizadas son los algoritmos genéticos (AG), que será en el que nos centremos ya que supondrá ser un algoritmo base para el desarrollo de este proyecto. Los algoritmos genéticos son algoritmos basados en poblaciones que se pueden describir como la combinación de dos procesos: la generación de elementos del espacio de búsqueda (recombinación o mutación de la población) y la actualización (selección y redimensionamiento) para producir nuevas soluciones basadas en el conjunto de puntos que se crean junto con los de la anterior población [33] [35]. En [1] se presenta una lista de componentes para la versión más simple de un AG, que se puede resumir en:

- Una población de soluciones candidatas a un problema dado, cada una codificada de acuerdo

a un esquema de representación elegido.

- Una función *fitness* que asigna un valor numérico a cada cromosoma (solución) de una población para medir su calidad como candidato a solución del problema.
- Un conjunto de operadores que se aplicarán a los cromosomas para crear una nueva población. Estos suelen incluir:
  - Operador de Cruce: Dos cromosomas padres recombinan sus genes para producir una o más soluciones hijas.
  - Mutación: Uno o varios genes de una solución se modifican de forma aleatoria.

Una explicación más completa y detallada de lo relativo a AGs se puede encontrar en [28]. En dicho capítulo también presentan algunos artículos y libros donde encontrar aplicaciones de AGs exitosas para la optimización de problemas combinatorios, entre ellas se destaca [29], la cual lista algunas de las referencias más útiles y accesibles que podrían ser de interés para gente experimentando con metaheurísticas, como podrían ser el problema del viajante de comercio, problemas relacionados con grafos, problema de la mochila en forma binaria, etc.

Cualquier algoritmo que siga un ciclo reproducción-recombinación en una población de estructuras se puede denominar un AG en el sentido más amplio del término. Sin embargo, desde el trabajo de Holland (1975) para clasificar cualquier algoritmo como un AG en un sentido más estricto, se debía demostrar que su comportamiento de búsqueda reflejaba lo que Holland llamaba un "paralelismo implícito". Eshelman [8] comprueba que CHC, un algoritmo genético no tradicional, ciertamente muestra paralelismo implícito. Incluso justifica que algunas de las características que parecerían descalificarlo como un verdadero GA no solo no lo descalifican, sino que lo hacen más potente que un AG tradicional. Eshelman describe en detalle los componentes (prevención de incesto, cruce HUX, reinicios de población...) y las características de CHC, lo contrasta con un AG tradicional, aporta una justificación teórica de sus diferencias y provee algunas comparaciones empíricas. Un ejemplo donde se aplique el algoritmo CHC para resolver un problema lo podemos encontrar en el artículo [32], donde se estudia el uso de tres estrategias de memoria explícita combinadas con el algoritmo CHC para resolver distintas instancias del problema del viajante de comercio dinámico.

Llamaremos **problemas de optimización costosos o *expensive*** (EOP) a aquellos problemas que requieran costes elevados o incluso inalcanzables para evaluar candidatos a soluciones. Existen una gran cantidad de EOPs relativos a aplicaciones en el mundo real; además, con el progreso de la sociedad y problemas emergentes como las *smart cities*, la era del *big data*, etc., la resolución más eficiente de EOPs se ha vuelto cada vez más esencial para prosperar en distintos campos. Sin embargo, debido al coste tan elevado de cálculo, es complicado para los algoritmos de optimización encontrar una solución satisfactoria a dichos EOPs. Por ello, la EC se ha adoptado en gran medida para resolver EOPs, llevando a un campo de investigación de rápido crecimiento. Hasta la fecha, se han conducido varias investigaciones sobre el uso de EC para EOP y han alcanzado un éxito considerable [17] [15] [31] [34]. El concepto de EOP se suele mencionar junto con algunos conceptos del problema similares, como podría ser un problema de optimización a gran escala. En [17] se hacen referencia y se explican varios de estos conceptos.

Sin embargo, si bien somos capaces de encontrar una gran cantidad de investigaciones para EOPs, debemos notar que la vasta mayoría se refieren a problemas con parámetros reales, es decir, que pertenecen a los casos continuos. Esto implica que para el ámbito de los problemas combinatorios *expensive* nos encontramos con una sequía de estudios al respecto y, por tanto, algoritmos desarrollados para optimizar su resolución. Un ejemplo de este tipo de problemas lo podemos encontrar en el artículo [21], donde se emplea una fusión de algoritmos de optimización basados en

la biología y modelos de *Deep Learning*. Para recalcar la necesidad de algoritmos pensados para problemas combinatorios *expensive* podemos mencionar el artículo [26], donde los propios autores reconocen que estuvieron obligados a reducir en gran medida el número de evaluaciones de los algoritmos utilizados, lo que impedía el uso de tests estadísticos para estudiar el nivel de significación de las diferencias encontradas, debido a los elevados tiempos de ejecución para cada simulación.

Los AG son algoritmos de búsqueda potentes que pueden ser aplicados a un amplio rango de problemas. Generalmente, el establecimiento de los parámetros se realiza antes de ejecutar un AG y esta configuración se mantiene constante durante toda la ejecución. Sin embargo, es interesante considerar el uso parámetros auto-adaptativos. Es decir, tomar un enfoque en el que el control de los parámetros de un AG pueda ser codificado dentro de los cromosomas de cada individuo. Así, los valores son totalmente dependientes del mecanismo de evolución y del contexto del problema. Pellerin et al. [23] realizan un estudio sobre este tipo de comportamiento y obtienen resultados que indican un enfoque prometedor de desarrollo de AGs con parámetros auto-adaptativos que no requieran que el usuario los pre-ajuste. Podemos también encontrar una breve clasificación de los distintos métodos de auto-adaptabilidad propuestos en la literatura: empírico y adaptativo [7] [19]:

- Enfoque empírico: Se basan solo en la experimentación y en la observación . Miden el rendimiento de las AG por ensayo y error, a la vez que se modifican los parámetros del algoritmo [7]. Por lo tanto, es necesario encontrar buenos valores para dichos parámetros antes de ejecutar el algoritmo, y estos valores se mantendrán constantes durante la ejecución.

- Enfoque adaptativo. Nos encontramos con dos categorías:

  - Parámetros adaptativos limitados: Analiza los efectos aislados de uno o dos parámetros teniendo en cuenta el resto [18].
  - Parámetros auto-adaptativos: Refiriéndose a las técnicas que permite la evolución o adaptación de diversos parámetros de AG durante su ejecución [13] [24].

Como ya se ha indicado, los AGs tienen gran capacidad de adaptarse a la estructura del espacio de soluciones y controlar el equilibrio entre búsquedas locales y globales, incluso sin ajustar sus parámetros, como serían la probabilidad de cruce o mutación. Obviamente, ningún algoritmo es perfectamente adecuado para todos los problemas de acuerdo al teorema *Not free lunch*, pero un determinado tipo de cruce puede llegar a resolver un gran número de problemas. Por ello, se han propuesto una gran cantidad de variantes de AGs en las que se modifican el tipo de cruce a lo largo de estos últimos años. Chaturvedi y Sharma en el capítulo *A Modified Genetic Algorithm Based on Improved Crossover Array Approach* [3] estudiaron la introducción de un AG con un operador de cruce mejorado de tal forma que se eligiese dinámicamente el tipo de operador de cruce que se iba a utilizar en el problema. Cheng y Gen [4] propusieron un nuevo operador de cruce para el problema del viajante de comercio con el objetivo de mejorar el AG tanto en tiempo como en precisión mediante el uso de relaciones de precedencia local entre los genes para obtener candidatos de posibles *edges* y relaciones de precedencia global entre los genes para obtener al mejor entre los candidatos. Quiobing y Lixin [27] propusieron un nuevo mecanismo de cruce para AG con cromosomas de longitud variable para problemas de optimización de caminos.

Los métodos de selección de padres son básicos dentro de los AGs, ya que determinan las posibilidades de los individuos de influir en las generaciones siguientes. Dado que en los AGs el operador que produce diversidad es el cruce, es necesaria una adecuada selección de los individuos para poder conseguir un adecuado equilibrio entre exploración y explotación. En este trabajo se utilizarán dos métodos:

- Una Selección por Torneo clásica, donde se eligen de forma aleatoria cierta cantidad de individuos de la población y solo elegimos el mejor.
- Emparejamiento Variado inverso (*Negative Assortative Mating*, NAM) [9]. Está orientado a generar diversidad, eligiéndose un elemento de forma aleatoria y otro grupo de individuos; de este grupo elegiremos como segundo padre a aquel elemento más diferente del primero.

En los AG estacionarios se debe determinar los individuos que serán reemplazados por los nuevos individuos generados. Esta elección puede determinar el nivel de presión selectiva que se exige a los individuos para ser introducidos o permanecer en la población, y la pérdida de diversidad que se introduce conforme el proceso de búsqueda avanza. Se han propuesto numerosas estrategias de reemplazo, algunas como introducir alta presión selectiva, como sería el reemplazar el peor elemento de la población [12], otras para mantener una alta diversidad, como serían los métodos de multitud o *crowding* [20] que buscan que las nuevas soluciones reemplacen las soluciones parecidas a ellas, o un equilibrio entre ambas.

El problema de la mochila cuadrático (QKP, *quadratic knapsack problem*) binario fue introducido por Gallo et al. [11] y resulta una generalización del clásico problema de la mochila (KP, *knapsack problem*), donde el beneficio que aporta cada elemento deja de ser solo individual, pudiendo aportar más valor si se combina con otro elemento. Aunque el QKP no se ha estudiado de forma tan intensiva como el problema de la asignación cuadrática, nos podemos encontrar con numerosos artículos a lo largo de estos últimos años relativos a este problema. Como cabría esperar debido a su generalidad, el QKP tiene un rango de aplicaciones muy amplio. Witzgall [36] presentó un problema que surge en telecomunicaciones cuando se debe seleccionar un número de localizaciones para las estaciones de satélites de forma que el tráfico global entre estas estaciones se maximice y la restricción de presupuesto se respete. Este problema resulta ser un QKP. Nos encontramos con modelos similares al considerar la localización de aeropuertos, estaciones de tren o terminales de manejo de carga [30]. Ferreira et al. [10] consideran un problema de diseño VLSI (*Very Large Scale Integration*) donde los grafos de gran tamaño necesitan ser descompuestos en grafos de más pequeños de tamaño manejable. El problema de optimización asociado para un único subgrafo puede ser reconocido como un QKP de minimización. Pisinger [25] realizó un estudio sobre los límites superiores presentados en la literatura y muestra la estanqueidad relativa de los diversos límites, proporcionando resultados y un estudio experimental donde se comparan estos resutalos con respecto a la potencia y el esfuerzo computacional.

Por último, se debe mencionar que se ha intentado seguir una guía de buenas prácticas (artículo [16]), aunque hay algunos puntos que no se pueden cumplir por motivos explicados más adelante. Podemos resumir lo utilizado de dicha guía en:

- **Validación de los resultados**: En tanto que no existen benchmarks que podamos utilizar para este trabajo, no podemos comparar nuestros resultados con otros algoritmos. Sin embargo, se realizarán tests estadísticos no paramétricos ([6]) entre las diferentes versiones del algoritmo que vamos a desarrollar para poder asegurar que realmente se están produciendo mejoras.
- **Análisis de los componentes y ajuste de parámetros de la propuesta**: Este análisis y justificación se realizará a lo largo de este trabajo.
- **Utilidad del algoritmo propuesto**: Como ya se ha indicado anteriormente y se volverá a indicar más tarde, no hay algoritmos que estén orientados a resolver problemas de optimización combinatorios *expensive*, por lo que este proyecto no es solo útil, sino que también supone una novedad.

Capítulo 4: Contexto matemático

Los algoritmos usan operadores matemáticos con el objetivo de alcanzar una solución a los problemas a los que son aplicados y también es necesario obtener el óptimo. La prueba de este hecho puede ser realizada gracias a la teoría de optimización del Análisis Numérico. Nos centraremos en esta teoría, aportando resultados que nos permitan saber si un punto es el óptimo de una función y su relación con los algoritmos básicos de optimización global.

La programación no lineal es un área de las matemáticas aplicadas que involucra problemas de optimización cuando las funciones son no lineales. Nuestro objetivo es introducir este problema y revisar las condiciones generales de optimalidad, que son la base de muchos algoritmos por sus soluciones. Tras esto, aportaremos numerosas nociones relativas al rendimiento de los algoritmos en términos de convergencia, orden de convergencia y comportamiento numérico.

Cabe mencionar que aunque el problema abordado en este trabajo implica la maximización del valor de la solución, se utilizará la notación convencional y usual para tratar este tipo de problemas, es decir, la minimización del valor de la solución. Esto se debe a que, al fin y al cabo, son problemas equivalentes. Una forma sencilla de transformar un problema de maximización en uno de minimización es cambiar el símbolo de los valores. Por ello, en tanto que ambos problemas son equivalentes, se mantendrá la notación de minimizar la función objetivo.

4.1. Definición del problema

En primer lugar, daremos una definición a nuestro problema.

Definición 4.1 *Consideramos el problema de calcular el valor de un vector de variables de decisión $x \in \mathbb{R}^n$ que minimiza la función objetivo $f : \mathbb{R}^n \to \mathbb{R}$ donde x pertenece a un conjunto factible de soluciones $\mathcal{F} \in \mathbb{R}^n$. Consideramos el siguiente problema:*

$$\min_{x \in \mathcal{F}} f(x) \tag{4.1}$$

Nota: Llamaremos **conjunto factible** al espacio de soluciones, es decir, al conjunto de todos los puntos posibles de un problema de optimización que satisface las restricciones del problema.

Ahora, presentamos dos casos de ello:

- El conjunto factible de soluciones $\mathcal{F}$ es todo el espacio $\mathbb{R}^n$. En este caso, el problema es el siguiente:

  $$\min_{x \in \mathbb{R}^n} f(x) \tag{4.2}$$

  Diremos que el problema 4.1 es no restringido. De forma general, el problema 4.1 es no restringido si $\mathcal{F}$ es un conjunto abierto.

- El conjunto factible de soluciones está descrito por restricciones de desigualdad y/o igualdad en las variables de decisión:

$$\mathcal{F} = \{x \in \mathbb{R}^n : g_i(x) \leq 0, i = 1, ..., p; h_j(x) = 0, j = 1, ..., m\} \tag{4.3}$$

Entonces, el problema 4.1 se convierte en:

$$\begin{array}{c} min_{x \in \mathbb{R}^n} f(x) \\ g(x) \leq 0 \\ h(x) = 0 \end{array} \tag{4.4}$$

donde $g : \mathbb{R}^n \to \mathbb{R}^p$ y $h : \mathbb{R}^n \to \mathbb{R}^m$. En este caso, diremos que el problema es restringido.

El problema 4.1 es no lineal cuando al menos una de las funciones del problema es no lineal, es decir, f,g_i, $i = 1, ..., p$, h_j, $j = 1, ..., m$ es no lineal en x.

Normalmente, asumimos que en un problema del tipo 4.1 el número de condiciones de igualdad, m, es menor que el número de variables, n; en otro caso, el conjunto factible de soluciones será el vacío, a no ser que haya dependencia en las restricciones. Si solo hay condiciones de igualdad, el problema se llamará "**problema no lineal con restricciones de igualdad**". Equivalentemente, se tiene el caso de que solo aparezcan condiciones de desigualdad.

En lo siguiente asumiremos que nuestras funciones f,g,h son diferenciables y continuas en $\mathbb{R}^n$. Además, cuando f sea una función convexa y el conjunto $\mathcal{F}$ sea también convexo, al problema se le llamará "**problema convexo no lineal**". Particularmente, $\mathcal{F}$ es convexo si las funciones que nos aportan las restricciones de desigualdad son convexas y las funciones de las restricciones de igualdad son afines.

La convexidad nos permite añadir estructura a estos problemas y poder explotarlo desde un punto de vista teórico y computacional, esto se debe a que si f es una función convexa cuadrática y g,h son afines, entonces tenemos que tratar con un problema de programación cuadrática. Sin embargo, nos centraremos en problemas no lineales generales, sin asumir convexidad.

Definición 4.2 *Un punto un $x^* \in \mathcal{F}$ es un **solución global** del problema 4.1 si $f(x^*) \leq f(x)$, $\forall x \in \mathcal{F}$. El punto es una **solución global estricta** si $f(x^*) < f(x)$, $\forall x \in \mathcal{F}$, $x \neq x^*$.*

La existencia de soluciones globales se debe a la compacidad de $\mathcal{F}$, en relación con el teorema de Weierstrass:

Teorema 4.1 (Teorema de Weierstrass) *Sea $a, b \in \mathbb{R}$ con $a < b$ y sea $f : [a, b] \to \mathbb{R}$ una función continua. Entonces, el intervalo $f([a, b])$ es cerrado y acotado.*

Una consecuencia directa para los problemas sin restricciones es que una solución global existe si el conjunto $\mathcal{L}^\alpha = \{x \in \mathbb{R}^n \leq \alpha\}$ es compacto para un α finito.

Definición 4.3 *Un punto $x^* \in \mathcal{F}$ es una **solución local** del problema del problema 4.1 si existe x^* en un vecindario abierto $\mathcal{B}_{x^*}$ de x^* tal que $f(x^*) \leq f(x)$, $\forall x \in \mathcal{F} \bigcap \mathcal{B}_{x^*}$. Además, es una **solución local estricta** si $f(x^*) < f(x)$, $\forall x \in \mathcal{F} \bigcap \mathcal{B}_{x^*}$, $x \neq x^*$.*

Determinar una solución global a un problema de este tipo es normalmente una tarea complicada. Los algoritmos que resuelven este tipo de problemas se usan para alcanzar óptimos locales. Incluso en aplicaciones prácticas obtener este tipo de soluciones puede ser también algo bueno.

Ahora introduciremos **notación**:

- Dado un vector $y \in \mathbb{R}^n$, definimos su **traspuesta** por y'. Esta definición puede ser extendida a las matrices $A \in \mathbb{R}^n \times \mathbb{R}^n$.
- Dada una función $h : \mathbb{R}^n \to \mathbb{R}$, denotamos el vector **gradiente** por $\nabla h(x) = \left(\frac{\partial f(x)}{\partial x_1}, ..., \frac{\partial f(x)}{\partial x_n} \right)$ y la matriz **Hessiana** se denota por

$$\nabla^2 h(x) = \begin{pmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{pmatrix}$$

- Dada una función vector $w : \mathbb{R}^n \to \mathbb{R}^q$, denotamos la matriz de $n \times q$ cuyas columnas son $\nabla w_j(x)$, $j = 1, ..., q$ con $\nabla w(x)$.
- Sea y un vector, $y \in \mathbb{R}^q$, denotamos la **norma Euclídea** por $||y||$. Supongamos que sus componentes son y_i, $i = 1, ..., q$ y sea A una matriz cuyas columnas sean a_j, $j = 1, ..., q$. Sea $\mathcal{K} \subset \{1, ..., q\}$ un subconjunto de índices. Denotamos por $y_\mathcal{K}$ al subvector de y con componentes y_i tales que $i \in \mathcal{K}$ y por $A_\mathcal{K}$ a la submatriz de A compuesta por las columnas a_j con $j \in \mathcal{K}$.

4.2. Condición de Optimalidad

Nuestras soluciones locales deben satisfacer algunas condiciones de optimalidad necesarias. Si nos referimos a problemas como 4.2, tenemos uno de los resultados más conocidos del Análisis Numérico clásico:

Proposición 4.1 *Sea x^* una solución local al problema 4.2, entonces*

$$\nabla f(x^*) = 0 \tag{4.5}$$

Además, si f es continuamente 2-diferenciable, entonces

$$y' \nabla^2 f(x^*) y \geq 0, \forall y \in \mathbb{R}^n \tag{4.6}$$

Nota: Diremos que una función es continuamente i-diferenciable si es diferenciable i veces y dichas diferenciales son continuas.

Si tenemos problemas como 4.1, la mayoría de las condiciones necesarias de optimalidad usadas en el desarrollo de los algoritmos asumen que una solución local debe satisfacer algunas de estas condiciones para evitar alcanzar casos degenerados. Estas condiciones se suelen llamar "**calificaciones de restricción**" y, entre ellas, la más simple y más comúnmente usada es el requisito de restricción de independencia lineal:

Definición 4.4 *Sea $\hat{x} \in \mathcal{F}$. Decimos que la restricción de desigualdad g_i está **activa** en $\hat{x}$ si $g_i(\hat{x}) = 0$. Denotamos por $\mathcal{F}_a(\hat{x})$ al conjunto de índices de las restricciones de desigualdad activas en $\hat{x}$:*

$$\mathcal{F}_a(\hat{x}) = \{i \in \{1, ..., p\} : g_i(\hat{x}) = 0\} \tag{4.7}$$

Nótese que en la definición anterior podemos concluir que las restricciones de igualdad h_j son activas en $\hat{x}$. La restricción de independencia lineal se satisface si el gradiente de las restricciones activas son linealmente independientes.

Bajo las condiciones de independencia lineal asumidas, los problemas de restricciones como 4.1 pueden resolverse usando la función Lagrangiana generalizada:

$$L(x, \lambda, \mu) = f(x) + \lambda' g(x) + \mu' h(x) \tag{4.8}$$

donde $\lambda \in \mathbb{R}^p$, $\mu \in \mathbb{R}^m$ son multiplicadores de Lagrange. El siguiente resultado nos dan información sobre la existencia de estos multiplicadores.

Proposición 4.2 *Sea x^* una solución local del problema 4.1 y supongamos que la independencia lineal se satisface en x^*. Entonces, los multiplicadores de Lagrange $\lambda^* \geq 0$, μ^* existe de tal forma que:*

$$\nabla_x L(x^*, \lambda^*, \mu^*) = 0 \tag{4.9}$$

y

$$\lambda'^* g(x^*) = 0$$

Además, si f, g, h son continuamente 2-diferenciables, entonces:

$$y' \nabla_x^2 L(x^*, \lambda^*, \mu^*) y \geq 0, \forall y \in \mathcal{N}(x^*)$$

donde

$$\mathcal{N}(x^*) = \{y \in \mathbb{R}^n : \nabla g'_{\mathcal{F}_a} y = 0; \nabla h(x^*)' y = 0\}$$

Si $x \in \mathcal{F}$ satisface una condición de optimalidad suficiente, entonces dicho punto es una solución local al problema 4.1. Para problemas generales, las condiciones de optimalidad suficientes pueden ser establecidas bajo la asunción de que las funciones problemas son continuamente 2-diferenciables, así que tenemos condiciones de suficiencia de segundo orden. Estas condiciones cambian si el problema a tratar es 4.2 o 4.1. Por lo tanto, los siguientes resultados mostrarán cuándo x^* son óptimos locales en ambos casos.

Proposición 4.3 *Asumimos que x^* satisface la condición 4.5. También asumiremos que*

$$y' \nabla^2 f(x^*) y > 0, \forall y \in \mathbb{R}^n, y \neq 0$$

es decir, se asume que $\nabla^2 f(x^*)$ es definida positiva. Entonces, x^* es una solución local estricta del problema 4.2.

Proposición 4.4 *Asumimos que $x^* \in \mathcal{F}$ y λ^*, μ^* satisfacen las condiciones de 4.9. Asumimos también que*

$$y' \nabla_x^2 L(x^*, \lambda^*, \mu^*) y > 0, \forall y \in \mathcal{P}(x^*), y \neq 0 \tag{4.10}$$

donde

$$\mathcal{P}(x^*) = \{y \in \mathbb{R}^n : \nabla g'_{\mathcal{F}_a} y \leq 0, \nabla h(x^*)' y = 0; \nabla g_i(x^*)' y = 0, i \in \mathcal{F}_a(x^*), \lambda_i^*\}$$

Entonces, x^* es una solución local estricta del problema 4.1.

Nótese también que $\mathcal{N}(x^*) \subseteq \mathcal{P}(x^*)$ y la igualdad se da si $\lambda_i^* > 0, \forall i \in \mathcal{F}_a(x^*)$.

Una característica importante y principal de los problemas convexos es que una solución global (estricta) del problema es también una solución local (estricta). Además, cuando f es (estrictamente) convexa, las funciones g_i son también convexas y las h_j son afines, entonces las condiciones de optimalidad necesarias dadas en términos de primeras derivadas parciales son suficientes para que un punto x^* sea una solución global (estricta).

Las condiciones de optimalidad son esenciales para problemas no lineales. Si conocemos la existencia de un óptimo global, entonces el método más común de obtenerlo es el siguiente:

1. Encontrar todos los puntos que satisfacen las condiciones necesarias de primer orden.
2. Tomas el óptimo global como el punto con el valor más bajo dado por la función objetivo
3. Si la función problema es 2-diferenciable, entonces comprueba la condición necesaria de segundo orden y elimina los puntos que no la satisfagan.
4. Para el resto de puntos comprobaremos la condición de suficiencia de segundo orden para encontrar el mínimo local.

Tenemos que destacar que este método no funciona en casos prácticos excepto por casos simples, esto se debe a que tenemos que calcular la solución de un sistema de ecuaciones dado por $\nabla f(x) = 0$ y este sistema es normalmente no trivial. Entonces, ¿dónde son importantes estas condiciones? Las condiciones de optimalidad son importantes en el desarrollo y análisis de algoritmos.

Un algoritmo que intenta resolver un problema dado por 4.1 genera una secuencia de soluciones factibles $x^k, k = 0, 1, ...$ y normalmente finaliza cuando se satisface un criterio de parada. Este criterio se suele basar en la satisfacción de condiciones de optimalidad necesarias dentro de una tolerancia prefijada. Adicionalmente, estas condiciones normalmente sugieren cómo mejorar la solución actual, con lo que la siguiente debería encontrarse más cercana al óptimo.

Así, las condiciones de optimalidad necesarias nos dan la base para el análisis de convergencia de algoritmos. Por lo tanto, las condiciones de suficiencia juegan un papel importante en el análisis del orden de convergencia.

4.3. Rendimiento de los algoritmos

4.3.1. Convergencia y Orden de Convergencia

Definición 4.5 *Sea $\Omega \subset \mathcal{F}$ el subconjunto de puntos que satisfacen las condiciones necesarias de optimalidad del problema 4.1. Un algoritmo se detiene cuando un punto $x^* \in \Omega$ es calculado. Por lo tanto, a este conjunto se le llamará* ***conjunto objetivo****.*

Un ejemplo de este conjunto para el problema sin restricciones podría ser $\Omega = \{x \in \mathbb{R}^n : \nabla f(x) = 0\}$; mientras que para el problema restringido este conjunto será el conjunto de puntos que satisfacen la condición 4.9.

Definición 4.6 *Sea $x^k, k = 0, 1, ...$ la secuencia de puntos generados por un algoritmo. Entonces, el algoritmo es* ***globalmente convergente*** *si un punto límite x^* de x^k existe tal que $x^k \in \Omega$ para cualquier punto de inicio $x^0 \in \mathbb{R}^n$.*

Diremos que el algoritmo es ***localmente convergente*** *si la existencia de* $x^* \in \Omega$ *solo puede ser establecida si el punto inicial,* x^0*, pertenece a un vecindario de* Ω*.*

La definición de convergencia establecida anteriormente es la más débil que asegura que un punto x^k arbitrariamente cercano a Ω puede ser obtenido con un k suficientemente grande.

En el caso de no tener restricciones, esto implica

$$\lim_{k \to \infty} ||\nabla f(x^k)|| = 0$$

El requerimiento más fuerte de convergencia nos muestra que la secuencia x^k converge a un punto $x^* \in \Omega$.

Ahora mostraremos cómo se puede definir el orden de convergencia de un algoritmo. Así, podemos asumir por simplicidad que los algoritmos generan una secuencia x^k que converge a un punto $x^* \in \Omega$. El concepto más empleado en términos de convergencia es el Q-orden de convergencia, el cual considera el cociente entre dos iteraciones sucesivas dado por

$$\frac{||x^{k+1} - x^*||}{||x^k - x^*||}$$

Entonces, podemos definir el siguiente tipo u orden de convergencia.

Definición 4.7 *El orden de convergencia es Q-lineal si existe una constante* $r \in (0, 1)$ *tal que*

$$\frac{||x^{k+1} - x^*||}{||x^k - x^*||} \leq r,$$

para cualquier k *suficientemente grande.*

Definición 4.8 *El orden de convergencia es Q-superlineal si*

$$\lim_{k \to \infty} \frac{||x^{k+1} - x^*||}{||x^k - x^*||} = 0$$

Definición 4.9 *El orden de convergencia es Q-cuadrático si*

$$\frac{||x^{k+1} - x^*||}{||x^k - x^*||^2} \leq R$$

para cualquier k *suficientemente grande y donde* $R > 0$ *es una constante.*

4.3.2. Comportamiento numérico

A pesar del rendimiento teórico que puedan tener los algoritmos, otro aspecto importante es el comportamiento práctico. En efecto, si tenemos una gran cantidad de operaciones algebraicas por operación, es posible superar una tasa de convergencia rápida. Tenemos numerosas medidas para evaluar el comportamiento numérico. Sin embargo, si la carga computacional (operaciones algebraicas por iteración) no es despreciable, entonces podemos usar algunas medidas dadas por el número de iteraciones, número de evaluaciones de funciones objetivo, etc.

Medir el rendimiento de los algoritmos es importante para problemas no lineales de gran escala. El término "gran escala" depende de la máquina que se encargue de los datos, pero ese tipo de problema son normalmente problemas sin restricciones que verifican que $n \geq 1000$, donde n es el número de variables. Sin embargo, un problema con restricciones se considerará ser un problema de gran escala cuando el número de variables es $n \geq 100$ y cuando la suma de las condiciones es 100 o mayor. Uno de los problemas más importantes es el trasladar algoritmos eficientes en problemas a pequeña escala a problemas de gran escala.

4.4. Programación no lineal sin restricciones

En esta sección consideraremos algoritmos que traten de resolver el siguiente problema sin restricciones

$$\min_{x \in \mathbb{R}^n} f(x) \tag{4.11}$$

donde $x \in \mathbb{R}^n$ es el vector de variables de decisión y $f : \mathbb{R}^n \to \mathbb{R}$ es la función objetivo. Es lógico pensar que si somos capaces de resolver este problema, el procedimiento para ello puede ser ajustado a problemas con restricciones porque solo necesitaremos un conjunto abierto factible de soluciones $\mathcal{F}$, y un punto inicial $x^0 \in \mathcal{F}$.

Asumimos por simplicidad que nuestra función f es continuamente 2-diferenciable en $\mathbb{R}^n$. Además, asumiremos lo siguiente para asegurarnos de la existencia de una solución del problema 4.11: $\mathcal{L}^0 = \{x \in \mathbb{R} : f(x) \leq f(x^0)\}$ es compacto para algún $x^0 \in \mathbb{R}^n$.

4.4.1. Algoritmos de Optimización sin restricciones

Ahora presentaremos algunos modelos de algoritmos que serán usados para resolver los problemas previos. Estos tipos de algoritmos están caracterizados por generar una secuencia de puntos, $\{x^k\}$, empezando por un punto inicial x^0, usando la siguiente iteración

$$x^{k+1} = x^k + \alpha^k d^k \tag{4.12}$$

donde d^k es la dirección de búsqueda y α^k es el tamaño de paso junto con d^k. En este método tenemos dos parámetros a modificar: la dirección de búsqueda y el tamaño del paso, por lo tanto, dependiendo de como variamos estos datos, obtendremos diferentes métodos y esto afectará a las propiedades de convergencia. El tamaño de paso afecta a la convergencia global, mientras que la dirección de búsqueda afecta a la convergencia local.

El siguiente resultado nos da una relación entre convergencia y dirección de búsqueda:

Proposición 4.5 *Sea $\{x^k\}$ la secuencia generada por 4.12. También asumimos:*

1. *$d^k \neq 0$ si $\nabla f(x^k) \neq 0$.*
2. *$\forall k$ tenemos $f(x^{k+1}) \leq f(x^k)$.*
3. $$\lim_{k \to \infty} \frac{\nabla f(x^k)' d^k}{||d^k||} = 0 \tag{4.13}$$
4. *$\forall k$ con $d^k \neq 0$, tenemos $\dfrac{|\nabla f(x^k)' d^k|}{||d^k||} \geq c||\nabla f(x^k)||$ con $c > 0$.*

Entonces, tenemos que, o bien, existe $\hat{k}$ *tal que* $x^{\hat{k}} \in \mathcal{L}^0$ *y* $\nabla f(x^{\hat{k}}) = 0$*, o bien, se genera una secuencia infinita tal que:*

1. $\{x^k \in \mathcal{L}^0\}$.
2. $\{f(x^k)\}$ *converge.*
3. $$\lim_{k \to \infty} ||\nabla f(x^k))|| = 0 \tag{4.14}$$

La tercera condición implica que solo necesitamos una subsecuencia que tenga un punto límite en Ω. Este resultado nos da información sobre cómo es la convergencia en términos de la dirección de búsqueda.

Procedemos a describir dos métodos conocidos como algoritmos de Búsqueda Lineal y métodos basados en el gradiente.

4.4.2. Algoritmos de Búsqueda Lineal

Estos algoritmos están caracterizados por determinar el tamaño de paso α^k junto con la dirección de búsqueda d^k. El objetivo es elegir un tamaño de paso que asegure la convergencia de 4.12. Una elección puede ser elegir dicho tamaño de paso tal y como se describe en la siguiente ecuación:

$$\alpha^k = arg\min_{\alpha} f(x^k + \alpha d^k)$$

La ecuación anterior puede ser resumida en la siguiente idea: el tamaño de paso es el valor que minimiza la función objetivo junto con una dirección dada. Sin embargo, en esta situación es posible que el mínimo no pueda ser alcanzado porque la función necesita tener propiedades que nos permitan calcular dicho mínimo, por lo tanto, una búsqueda lineal no tiene por qué suponer la mejor solución computacionalmente hablando.

Por ello, tenemos que usar métodos de aproximación. Uno de ellos es calcular el gradiente de la función. En estos casos podemos asumir que

$$\nabla f(x^k)' d^k < 0, \forall k \tag{4.15}$$

Los algoritmos de búsqueda lineal más simples están proporcionados por Armijo, cuyo pseudocódigo se puede encontrar en Algoritmo 1.

La elección inicial de ∇^k se debe a la dirección d^k porque garantizamos que, en un número finito de pasos, α^k tiene un valor tal que $f(x^{k+1}) < f(x^k)$, por lo que las condiciones de convergencia de la proposición se encuentran satisfechas.

Esta búsqueda lineal encuentra un tamaño de paso que satisface la condición de decrecimiento suficiente de la función objetivo y, consecuentemente, el desplazamiento suficiente de la secuencia actual.

En algunos casos, necesitamos considerar que los algoritmos de búsqueda lineal no necesitan información sobre las derivadas. En estos casos, la condición 4.15 no puede ser verificada. Por lo tanto, la dirección d^k no tiene por qué ser descendiente.

Otra posible modificación al algoritmo propuesto podría ser sustituir la condición de parada por la siguiente, que no tiene información sobre la derivada:

$$f(x^k + \alpha d^k) \leq f(x^k) - \gamma \alpha^2 ||d^k||^2 \tag{4.16}$$

Algorithm 1 Algoritmo de Búsqueda Lineal de Armijo

procedure (Busqueda_Lineal)($\delta \in (0,1), \gamma \in (0,1/2), c \in (0,1)$)
  Elegir ∇^k tal que
$$\nabla^k \geq c\frac{|\nabla f(x^k)' d^k|}{||d^k||^2}$$
  $\alpha = \nabla^k$
  $N \leftarrow n$
  if $f(x^k + \alpha d^k) \leq f(x^k) + \gamma\alpha\nabla f(x^k)' d^k$ **then**
    $\alpha^k = \alpha$ y parar
  else
    Establecer $\alpha = \delta\alpha$ y volver al paso anterior.
  end if
end procedure

4.4.3. Métodos de Gradiente

El método de gradiente o método del descenso más rápido (*steepest-descent*) se considera un método básico de entre todos los algoritmos de optimización no restringidos. Tiene la regla básica de establecer $d^k = -\nabla f(x^k)$ en 4.12. Solo necesita información sobre la primera derivada y es importante debido a que el coste computacional y el almacenamiento disponibles son limitados.

Este método es un ejemplo de convergencia global, porque si usamos un algoritmo de búsqueda local adecuado, podemos establecer un resultado de convergencia global. El problema es su orden de convergencia, el cual es bajo y esto es la razón por la que este algoritmo no se suele usar solo. El esquema de este algoritmo es el siguiente:

Algorithm 2 Método de Gradiente

procedure (Gradiente)
  Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$
  while $\nabla f(x^k)! = 0$ **do**
    Establecer $d^k = -\nabla f(x^k)$ y encuentra un tamaño de paso α^k usando 1
    Establecer $x^{k+1} = x^k - \alpha^k \nabla f(x^k)$ y $k = k + 1$.
  end while
end procedure

La elección inicial del tamaño de paso como la dada por Armijo es importante, ya que puede afectar al comportamiento del algoritmo. En términos de convergencia, el siguiente resultado nos asegura la convergencia sin necesitar asumir la compacidad del conjunto $\mathcal{L}^0$.

Nota: Sea $f : \mathbb{R}^n \to \mathbb{R}^m$. f se dice **Lipschitziana** si existe $K > 0$ tal que
$$||f(x) - f(y)|| \leq K||z - y||, \forall x, y \in \mathbb{R}^n$$

Proposición 4.6 *Si ∇f es Lipschitziana, continua y f está acotada inferiormente , entonces la secuencia generada por 2 satisface la tesis de 4.5.*

Como conclusión, con este resultados podemos comprobar que el gradiente es bueno en términos de convergencia global. Sin embargo, solo podemos probar la convergencia lineal. Desde un punto de vista práctico, el orden de convergencia es muy pobre y depende del número de condiciones de la matriz Hessiana.

4.4.4. Métodos de Gradiente Conjugado

Este algoritmo es muy popular debido a su simplicidad y sus bajos requerimientos computacionales. En efecto, solo necesita saber sobre las derivadas de primer orden. La idea principal es que la minimización de funciones cuadráticas estrictamente convexas en $\mathbb{R}^n$ como la siguiente

$$f(x) = \frac{1}{2}x'Qx + \alpha' x \tag{4.17}$$

donde Q es una matriz simétrica definida positiva, puede ser dividida en n minimizaciones sobre $\mathbb{R}$. Esto se puede hacer utilizando n direcciones, $d^0, ..., d^{n-1}$ conjugadas con respecto de la matriz Hessiana Q. Junto con cada dirección, se realiza una búsqueda lineal.

El siguiente algoritmo (Algoritmo 3) muestra todo lo mencionado anteriormente.

Se debe notar que el valor de α^k que se utilizará en el tercer paso es el que minimiza la función $f(x^k + \alpha d^k)$. Además, el cociente está bien definido porque para dos direcciones distintas tenemos $(d^j)Qd^i = 0$, con i, j tales que $i \neq j$.

Algorithm 3 Algoritmo de direcciones conjugadas para funciones cuadráticas

1: **procedure** (Direcciones_Conjugadas) (direcciones Q-conjugadas $d^0, ..., d^{n-1}$)
2: Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$
3: **while** $\nabla f(x^k)! = 0$ **do**
4: Establecer $\alpha^k = \dfrac{\nabla f(x^k)' d^k}{(d^k)' Q d^k}$
5: Establecer $x^{k+1} = x^k - \alpha^k \nabla f(x^k)$ y $k = k + 1$.
6: **end while**
7: **end procedure**

En este algoritmo, las direcciones provienen de los datos, mientras que en el algoritmo del gradiente conjugado se calculan de forma iterativa usando la siguiente regla:

$$d^k = \begin{cases} -\nabla f(x^k) & si \quad k = 0 \\ -\nabla f(x^k) + \beta^{k-1} d^{k-1} & si \quad k \geq 1 \end{cases}$$

El escalar β^k se elige para reforzar la conjugación entre las direcciones. La opción más común es la propuesta de Fletcher-Reeves:

$$\beta_{FR} = \frac{||\nabla f(x^{k+1})||^2}{||\nabla f(x^k)||^2} \tag{4.18}$$

Sin embargo, la fórmula de Polak-Ribiére también puede usarse:

$$\beta_{PR} = \frac{\nabla f(x^{k+1})'(\nabla f(x^{k+1}) - \nabla f(x^k))}{||\nabla f(x^k)||^2} \tag{4.19}$$

Ambas fórmulas son iguales en el caso cuadrático y diferentes cuando se trate de cualquier otro caso.

El esquema del gradiente conjugado se presenta en Algoritmo 4.

Algorithm 4 Algoritmo del gradiente conjugado

1: **procedure** (Gradiente_Conjugado)
2: Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$
3: **while** $\nabla f(x^k)! = 0$ **do**
4: Calcular β^{k-1} usando 4.18 o 4.19 y establecer la dirección usando

$$d^k = \begin{cases} -\nabla f(x^k) & si \quad k = 0 \\ -\nabla f(x^k) + \beta^{k-1} d^{k-1} & si \quad k \geq 1 \end{cases}$$

5: Encontrar α^k usando un algoritmo de búsqueda lineal que satisfaga las condiciones de Wolfe.
6: Establecer $x^{k+1} = x^k + \alpha^k d^k$ y $k = k + 1$.
7: **end while**
8: **end procedure**

Las condiciones de Wolfe mencionadas en 4 son las siguientes:

$$f(x^k + \alpha d^k) \leq f(x^k) + \gamma \alpha \nabla f(x^k)' d^k \tag{4.20}$$

que es la misma que se usó en la Búsqueda Lineal de Armijo, con la siguiente condición siendo más fuerte

$$|\nabla f(x^k + \alpha d^k)' d^k| \leq \beta |\nabla f(x^k)' d^k| \tag{4.21}$$

donde $\beta \in (\gamma, 1)$ y γ se encuentra en el mismo intervalo que antes.

La forma más simple de asegurar las propiedades de convergencia global en este método es aplicando reinicios periódicos junto con dirección de descenso más rápido. Aún así, el reinicio puede también suceder si alguno de los términos cuadráticos se pierden y pueden causar, o bien, que el método sea ineficiente, o bien, elecciones de caminos sin sentido.

El mecanismo de reinicio se realiza cada n iteraciones o si se da la siguiente condición:

$$|\nabla f(x^k)' \nabla f(x^{k+1})| > \delta ||\nabla f(x^{k-1})||^2$$

con $0 < \delta < 1$.

4.4.5. Métodos de Newton

Este método se considera uno de los algoritmos más potentes para resolver problemas de optimización sin restricciones. La aproximación cuadrática de la función objetivo en un vecindario de la solución actual, x^k, considerada es la siguiente

$$q^k(s) = \frac{1}{2} s' \nabla^2 f(x^k) s + \nabla f(x^k)' s + f(x^k)$$

donde necesitamos saber las derivadas de primer y segundo orden de la función objetivo en la iteración número k. Este algoritmo también necesita calcular una dirección, d_N, y en ese caso, se obtiene como un punto estacionario de la aproximación anterior y es la solución del sistema dado por

$$\nabla^2 f(x^k) d_N = -\nabla f(x^k) \tag{4.22}$$

Por lo tanto, la matriz $\nabla^2 f(x^k)d_N$ es no singular, tiene una inversa y la dirección es $d_N = -(\nabla^2 f(x^k))^{-1}\nabla f(x^k)$.

El esquema básico algorítmico está definido por la iteración

$$x^{k+1} = x^k - (\nabla^2 f(x^k))^{-1}\nabla f(x^k) \tag{4.23}$$

La calidad de este método se debe al hecho de que si el punto de partida x^0 está próximo a la solución x^*, entonces la secuencia de puntos generado por la ecuación anterior converge a x^* de forma superlineal o cuadrática (si la Hessiana es continuamente Lipschitziana en un vecindario de la solución).

Sin embargo, este método tiene algunas desventajas. Una de ellas es la singularidad de la matriz $\nabla^2 f$ porque en el caso de ser singular, el método no puede definirse. Otra desventaja está relacionada con el punto inicial, x^0. Puede ser tal que la secuencia generada por 4.23 no converge, pero puede ocurrir la convergencia a un punto máximo.

Debido a estos hechos, el método de Newton necesita algunos cambios para asegurar la convergencia global a la solución. Un método de Newton convergente debería generar una secuencia de puntos $\{x^k\}$ con las siguientes características:

- Admite un punto límite.
- Cualquier otro punto límite pertenece a $\mathcal{L}$ y es un punto estacionario de f.
- Ningún punto límite es un punto máximo de f.
- Si x^* es un punto límite de $\{x^k\}$ y $\nabla^2 f(x^*)$ es definida positiva, entonces la convergencia es, al menos, superlineal.

Hay dos enfoques para diseñar un método de Newton convergente globalmente: un enfoque con búsqueda lineal y un enfoque con regiones de confianza.

Modificaciones de Búsqueda Lineal en el Método de Newton

la adaptación del método a este enfoque es el control del tamaño de paso junto con d_N tal que 4.23 se convierte en

$$x^{k+1} = x^k - \alpha^k[\nabla^2 f(x^k)]^{-1}\nabla f(x^k) \tag{4.24}$$

donde α^k se elige con una buena búsqueda local. Un ejemplo puede ser inicializar $\Delta^k = 1$ en Algoritmo 1. Adicionalmente a este cambio, la dirección d_N puede ser perturbada para asegurar la convergencia global del algoritmo. La forma más fácil de realizar este cambio es usar la dirección del descenso más rápido siempre cuando d_N^k no satisface alguna de las condiciones de convergencia.

Un posible esquema de dicho algoritmo se presenta en Algoritmo 5.

En el tercer paso, se toma la dirección del descenso más rápido si $\nabla f(x^k)' d_N^k \geq 0$. Otra posible modificación del método de Newton podría ser aquella que toma la dirección de la curvatura negativa, es decir, $d^k = -d_N^k$. Esta modificación se puede hacer si las siguientes dos condiciones se cumplen:

1. $|\nabla f(x^k)' d^k| \geq c_1 ||\nabla f(x^k)||^q$

2. $||d^k||^p \leq c_2||\nabla f(x^k)||$

Algorithm 5 Método de Newton con Búsqueda Lineal

procedure (ABLNewton) $(c_1 > 0, c_2 > 0, p \geq 2, q \geq 3)$
  Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$
  while $\nabla f(x^k)! = 0$ **do**
    if $\exists d_N^k$ solución de $\nabla^2 f(x^k) d_N^k = -\nabla f(x^k)$ y satisface

$$\nabla f(x^k)' d_N^k \leq -c_1||\nabla f(x^k)||^q, ||d_N^k||^p \leq c_2||\nabla f(x^k)||$$

    then
      Establecer la dirección $d^k = d_N^k$
    else
      $d^k = -\nabla f(x^k)$
    end if
    Encontrar α^k usando 1
    Establecer $x^{k+1} = x^k + \alpha^k d^k$ y, después, $k = k + 1$
  end while
end procedure

Una segunda modificación es perturbar la matriz Hessiana con una matriz definida positiva Y^k y ahora la solución provendría de resolver el sistema $(\nabla^2 f(x^k) + Y^k)d = -\nabla f(x^k)$.

Las modificaciones comunes de los métodos de Newton se basan en el decremento monótono de los valores de la función objetivo. Con estos cambios la región de convergencia del método puede aumentar más de lo esperado, pero una secuencia convergente en este conjunto puede no ser una secuencia monótonamente descendente de los valores de la función objetivo.

La siguiente modificación está basada en las condiciones de búsqueda lineal y mantiene la propiedad de convergencia global. En esta modificación, usaremos una regla no monótona. Por lo tanto, el método obtenido se presenta en Algoritmo 6.

Los métodos de Búsqueda Lineal estudiados hasta ahora convergen a puntos satisfaciendo solo las condiciones de optimalidad de primer orden necesarias de la ecuación 4.11. Esto se debe a que el método de Newton no explota toda la información obtenida en la segunda derivada. Es posible obtener una convergencia más fuerte si usamos el par de direcciones (d^k, s^k) y una búsqueda curvilinear, es decir,

$$x^{k+1} = x^k + \alpha^k d^k + (\alpha^k)^{\frac{1}{2}} s^k \tag{4.25}$$

donde d^k es una dirección del método de Newton y s^k es una dirección que incluye información de curvatura negativa con respecto a $\nabla^2 f(x^k)$. Con esta idea y algunas modificaciones a la Búsqueda Lineal de Armijo, se pueden crear algoritmos globalmente convergente que además puedan satisfacer las condiciones de optimalidad de segundo orden necesarias para la ecuación 4.11.

Modificaciones de Regiones de Confianza en el Método de Newton

Este tipo de algoritmos tienen su iteración principal como muestra la siguiente ecuación

$$x^{k+1} = x^k + s^k$$

Algorithm 6 Método de Newton no monótono

procedure (NewtonNM) ($c_1 > 0, c_2 > 0, p \geq 2, q \geq 3$ y M entero)
  Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$
  while $\nabla f(x^k)! = 0$ **do**
    if $\exists d_N^k$ solución de $\nabla^2 f(x^k) d_N^k = -\nabla f(x^k)$ y satisface

$$\nabla f(x^k)' d_N^k \leq -c_1 ||\nabla f(x^k)||^q, ||d_N^k||^p \leq c_2 ||\nabla f(x^k)||$$

then
      Establecer la dirección $d^k = d_N^k$
    else
      $d^k = -\nabla f(x^k)$
    end if
    Encontrar α^k usando 1, tal que

$$f(x^k + \alpha d^k) \leq \max_{0 \leq j \leq J} \{f(x^{k-j})\} + \gamma \alpha \nabla f(x^k)' d^k$$

con $J = m(k)$
    Establecer $x^{k+1} = x^k + \alpha^k d^k$ y, después, $k = k + 1$
    Establecer $m(k) = \min\{m(k-1) + 1, M\}$
  end while
end procedure

donde el paso s^k se obtiene minimizando la forma cuadrática q^k de la función objetivo en una región de confianza del espacio $\mathbb{R}^n$. La región de confianza se define como una norma l_p del paso s. Lo más común es elegir la norma Euclidiana, con la cual, en cada iteración, s^k es la solución de

$$\min_{s \in \mathbb{R}^n} \frac{1}{2} s' \nabla^2 f(x^k) s + \nabla f(x^k)' s$$

donde $||s||^2 \leq (a^k)^2$ con a siendo el radio de la región de confianza. Otra opción consistiría en realizar un cambio de escala a la condición previa de la siguiente forma: $||D^k s||^2 \leq (a^l)^2$. Por simplicidad, asumiremos que la matriz $D^k = I$, es decir, la matriz identidad en el espacio adecuado.

Estos algoritmos se caracterizan por la siguiente idea: cuando la matriz $\nabla^2 f(x^k)$ es definida positiva, entonces el radio a^k tiene que ser lo suficientemente grande que el minimizador de 4.4.5 no tenga restricciones y el paso dado por Newton sea un entero. Además, a^k se actualiza en cada iteración y su instrucción de actualización depende de la proporción ρ^k entre la reducción de los valores de la función objetivo $f(x^k) - f(x^{k+1})$ y la reducción esperada $f(x^k) - q^k(s^k)$.

El siguiente algoritmo (Algoritmo 7) presenta las ideas anteriores y también garantiza la satisfacción de las condiciones de optimalidad necesarias para 4.11 en su tercer paso.

Si f es además continuamente 2-diferenciable, entonces la secuencia del algoritmo anterior, $\{x^k\}$, tiene un punto límite que satisface las condiciones de optimalidad necesarias de primer y segundo orden para la ecuación 4.11. Además, si $\{x^k\}$ converge a un punto en el que la matriz Hessiana $\nabla^2 f$ es definida positiva, entonces el orden de convergencia es superlineal.

El esfuerzo computacional de este algoritmo es el subproblema de las regiones de confianza. Debido a este hecho, se han ido desarrollando cada vez más algoritmos para resolverlo. Sin embargo, no necesitamos una solución exacta para esta ecuación. Para probar que la convergencia global del algoritmo es suficiente verificar que el valor $q^k(s^k)$ es menor que el valor en un punto de Cauchy,

que es el punto que minimiza el modelo cuadrático en la región de confianza.

Algorithm 7 Método de Newton basado en Regiones de Confianza

procedure (NewtonRegionConfianza) $(0 < \gamma_1 \leq \gamma_2 < 1, 0\delta_1 < 1 \leq \delta_2)$
  Establecer $x^0 \in \mathbb{R}^n$, $k = 0$ y $a^0 = 0$.
  Encontrar $s^k = arg\,\text{mín}_{\{} ||s|| \leq a^k\} q^k(s)$
  if $f(x^k) == q^k(s^k)$ **then**
    Parar
  else
    Calcular la proporción

$$\rho^k = \frac{f(x^k) - f(x^k + s^k)}{f(x^k) - q^k(s^k)}$$

    if $\rho^k \geq \gamma_1$ **then**
      Actualizar $x^{k+1} = x^k + s^k$
    else
      Actualizar $x^{k+1} = x^k$
    end if
    Actualizar el radio a^k

$$a^{k+1} = \begin{cases} \delta_1 a^k & si \quad \rho^k < \gamma_1 \\ a^k & si \quad \rho^k \in\in [\gamma_1, \gamma_2] \\ \delta_2 a^k & si \quad \rho^k > \gamma_2 \end{cases}$$

  y establecer $k = k + 1$, entonces volver al paso 3.
  end if
end procedure

Métodos de Newton Truncados

Los métodos de Newton necesitan calcular la solución de un sistema lineal de ecuaciones en cada iteración. Si echamos un vistazo a problemas a gran escala, resolver este sistema en cada iteración puede resultar demasiada carga computacionalmente hablando. Además, la solución exacta cuando x^k está lejos de una solución y $||\nabla f(x^k)||$ es grande no es necesaria. Debido a este hecho, se han propuesto numerosos métodos que calculan una solución aproximada a este sistema con un orden de convergencia bueno, es decir, si $\widetilde{d_N^k}$ es una solución aproximada del sistema, entonces la medida de precisión es dada por el residual de la ecuación de Newton, que sería

$$r^k = \nabla^2 f(x^k)\widetilde{d_N^k} + \nabla f(x^k)$$

Si podemos controlar este residual, entonces podemos probar la convergencia superlineal.

Proposición 4.7 *Si $\{x^k\}$ converege a una solución y si*

$$\lim_{k \to \infty} \frac{||r^k||}{||\nabla f(x^k)||} = 0$$

entonces $\{x^k\}$ converge de forma superlineal.

Estos métodos solo requieren de operaciones matriciales-vectoriales, por lo que son adecuados para problemas a gran escala. Al siguiente algoritmo se le conoce como el método de Newton truncado y se necesita que la matriz Hessiana sea definida positiva. Este método se obtiene aplicando un esquema de gradiente conjugado para encontrar una solución óptima del sistema 4.22.

Generaremos los vectores d_N^i, que se van a ir aproximando a la dirección d_N^k y se detendrá cuando ocurra uno de los siguientes casos:

- El residual r^i verifica que $||r^i|| \leq \epsilon^k$, para $e^k > 0$.
- Si se ha encontrado una dirección de curvatura negativa, es decir, la dirección conjugada s^i verifica que $(s^i)'\nabla^2 f(x^k)s^i \leq 0$.

Este algoritmo tiene el mismo resultado para convergencia que el método de Newton. Por simplicidad, eliminaremos las dependencias en la iteración k y estableceremos $H = \nabla^2 f(x^k)$ y $g = \nabla f(x^k)$. Este método genera las direcciones conjugadas s^i y los vectores p^i que aproximan la solución del sistema de Newton $\widetilde{d_N^k}$. Esto se puede encontrar representado en el Algoritmo 8.

Algorithm 8 Algoritmo de Newton Truncado

1: **procedure** (NewtonRegionConfianza) (k, H, g y $\eta > 0$ escalar)
2: Establecer $i = 0, p^0 = 0, r^0 = -g, s^0 = r^0$ y

$$\epsilon = \eta||g|| \text{ mín}\left\{\frac{1}{k+1}, ||g||\right\}$$

3: **while do**
4: **end while**
5:

$$a_N = \begin{cases} \delta_1 a^k & si \quad \rho^k < \gamma_1 \\ a^k & si \quad \rho^k \in\in [\gamma_1, \gamma_2] \\ \delta_2 a^k & si \quad \rho^k > \gamma_2 \end{cases}$$

6: **end procedure**

Si aplicamos este algoritmo para encontrar una solución aproximada al sistema en el cuarto paso de 5 o 6, entonces obtendremos la versión monótona truncada del método de Newton.

Métodos Quasi-Newton

Estos métodos han sido introducidos con el objetivo de diseñar algoritmos eficientes que no necesiten la evaluación de derivadas de segundo orden. Por tanto, han establecido d^k como la solución de

$$B^k d = -\nabla f(x^k) \tag{4.26}$$

donde B^k es una matriz definida positiva simétrica de tamaño $n \times n$ que se ajusta de forma iterativa para que la dirección d^k tienda a aproximar la dirección del método de Newton. A la fórmula anterior se le conoce como **fórmula quasi-Newton** y su inversa es

$$d^k = -H^k \nabla f(x^k) \tag{4.27}$$

Ambas matrices se modifican en cada iteración como una correción de la anterior, es decir, $B^{k+1} = B^k + \Delta B^k$, y lo mismo pasa para H^k. Definimos las siguientes dos cantidades:

$$\delta^k = x^{k+1} - x^k \qquad \gamma^k = \nabla f(x^{k+1}) - \nabla f(x^k)$$

En caso de que f sea una función cuadrática, entonces la ecuación quasi-Newton sería

$$\nabla^2 f(x^k)\delta^k = \gamma^k \tag{4.28}$$

por lo tanto, ΔB^k (lo mismo para ΔH^k) se elige de la siguiente forma

$$(B^k + \Delta B^k)\delta^k = \gamma^k \tag{4.29}$$

La regla de actualización de H^k está dada por

$$\Delta H = \frac{\delta^k(\delta^k)'}{(\delta^k)'\gamma^k} - \frac{H^k\gamma^k(H^k\gamma^k)'}{(\gamma^k)'H^k\gamma^k} + c(\gamma^k)'H^k\gamma^k v^k(v^k)' \tag{4.30}$$

donde

$$v^k = \frac{\delta^k}{(\delta^k)'\gamma^k} - \frac{H^k\gamma^k}{(\gamma^k)'H^k\gamma^k}$$

y $c > 0$ es un escalar. El algoritmo que vamos a mostrar ahora fue creado por Broyden, Flecher, Goldfarb y Shanno. Es un método de búsqueda lineal en el que el tamaño de paso es α^k se obtiene mediante un algoritmo de búsqueda lineal. El esquema del algoritmo se presenta en Algoritmo 9.

Algorithm 9 Algoritmo BFGS Inverso Quasi-Newton

1: **procedure** (InversaBFGSQuasiNewton)
2: Establecer $x_0 \in \mathbb{R}^n, H^0 = I$ y $k = 0$
3: **while** $\nabla f(x^k)! = 0$ **do**
4: Establecer la dirección $d^k = -H^k\nabla f(x^k)$
5: Encontrar α^k mediante una búsqueda lineal que satisfaga las condiciones de Wolfe 4.20 y 4.21
6: Actualizar

$$\begin{aligned} x^{k+1} &= x^k + \alpha^k d^k \\ H^{k+1} &= H^k + \Delta H^k \end{aligned} \tag{4.31}$$

con ΔH^k dada por la regla 4.30 con $c = 1$.
7: Establecer $k = k + 1$
8: **end while**
9: **end procedure**

Distinguimos dos casos en relación a las propiedades de convergencia: caso convexo y caso no convexo. Cuando no tenemos convexidad, si existe una constante ρ tal que para cada k se verifica la siguiente condición

$$\frac{||\gamma^k||^2}{(\gamma^k)'\delta^k} \leq \rho \tag{4.32}$$

entonces la secuencia de puntos generada por el algoritmo satisface

$$\lim_{k\to\infty} inf||\nabla f(x^k)|| = 0$$

que es la condición débil de 4.5. Para el caso convexo, la desigualdad anterior se mantiene. El siguiente resultado nos dará información sobre el orden de convergencia de este algoritmo.

Proposición 4.8 *Sea* $\{B^k\}$ *una secuencia de matrices no singulares y sea* $\{x^k\}$ *la secuencia dada por*

$$x^{k+1} = x^k - (B^k)^{-1}\nabla f(x^k)$$

También supondremos que $\{x^k\}$ *converge a un punto* x^* *donde* $\nabla^2 f(x^*)$ *es también no singular. Entonces, la secuencia* $\{x^k\}$ *converge de forma superlineal a* $x^* \Leftrightarrow$

$$\lim_{k\to\infty} \frac{||[B^k - \nabla^2 f(x^*)](x^{k+1} - x^k)||}{||x^{k+1} - x^k||} = 0$$

Este algoritmo está pensado para problemas de pequeña escala, ya que para los de gran escala, el almacenar una matriz como serían B^k o H^k ocasionaría problemas de almacenamiento. Por lo tanto, para esos problemas la información se obtiene de las últimas iteraciones.

Métodos sin derivadas

Estos métodos no calculan explícitamente las derivadas de f. Son adecuados para cuando, o bien, el gradiente de la función objetivo no puede ser calculado, o bien, cuando es computacionalmente costoso. Sin embargo, si queremos probar las propiedades de convergencia, necesitaremos suponer que f es continuamente diferenciable.

De entre todos estos algoritmos, los dos más importantes son los algoritmos de búsqueda de patrones (PSA, *pattern-search algorithm*) y los algoritmos de búsqueda lineal sin derivadas (DFLSA, *derivative-free line-search algorithm*). Estos algoritmos son similares, y su diferencia reside en las suposiciones que hacen sobre el conjunto de direcciones y en la regla que usan para encontrar el tamaño de paso junto con las direcciones. Denotamos $\mathcal{D}^k = \{d^1, ..., d^r\}$ con $r \geq n+1$ como el conjunto de direcciones y suponemos que son unitarias.

También asumimos el siguiente hecho para los PSA: las direcciones $d^j \in \mathcal{D}^k$ son la j-ésima columna de la matriz $B\Gamma^k$ con B una matriz no singular con coeficientes reales de tamaño n y $\Gamma^k \in \mathcal{M} \subset \mathbb{Z}^{n\times r}$, donde $\mathcal{M}$ es un conjunto finito de matrices integrales tales que su rango coincide con el número de filas que tienen (*full row-rank*).

Esta suposición nos da una idea para entender que el PSA itera sobre x^{k+1} en una red (*lattice*) racional centrada en x^k. Sea $\mathcal{P}^k$ el conjunto de candidatos para la siguiente iteracion, es decir,

$$\mathcal{P} = \{x^{k+1} : x^k + \alpha^k d^j, d^j \in \mathcal{D}^k\}$$

En este conjunto se conoce el patrón y se elige el tamaño de paso para preservar la estructura algebraica en la siguiente iteración, por lo que tenemos $f(x^k + s^k) < f(x^k)$ con $s^k = \alpha^k d^j$.

Para DLFSA haremos la siguiente suposición: las direcciones $d^j \in \mathcal{D}^k$ son la j-ésima columna de una matriz B^k de tamaño $n \times r$ con rango n. En este caso, no hay suposiciones adicionales y solo tiene que verificar la reducción de la función objetivo.

Los dos siguientes algoritmos muestran una versión del PSA (Algoritmo 10) y del DFLSA (Algoritmo 11).

Algorithm 10 Algoritmo de Búsqueda de Patrones

procedure (PSA) ($\mathcal{D}^{\|}$ satisfaciendo la suposición previa, $\tau \in \{1,2\}, \theta = \frac{1}{2}$)
  Establecer $x_0 \in \mathbb{R}^n, \Delta^0 > 0$ y $k = 0$
  Comprobar la convergencia
  if $\exists j \in \{1,\dots,r\}$:

$$f(x^k + \alpha^k d^j) < f(x^k), \alpha^k = \Delta^k$$

then
    Establecer $x^{k+1} = x^k$, $\Delta k + 1 = \tau \Delta^k$, $k = k + 1$ e ir al paso 3.
  else
    Establecer $x^{k+1} = x^k$, $\Delta k + 1 = \theta \Delta^k$, $k = k + 1$
  end if
end procedure

Algorithm 11 Algoritmo de Búsqueda Lineal sin Derivadas

procedure (DFLSA) ($\mathcal{D}^{\|}$ satisfaciendo la suposición previa, $\gamma > 0$, $\theta \in (0,1)$)
  Establecer $x_0 \in \mathbb{R}^n, \Delta^0 > 0$ y $k = 0$
  Comprobar la convergencia
  if $\exists j \in \{1,\dots,r\}$:

$$f(x^k + \alpha^k d^j) \leq f(x^k) - \gamma(\alpha^k)^2, \alpha^k \geq \Delta^k$$

then
    Establecer $x^{k+1} = x^k$, $\Delta k + 1 = \alpha^k$, $k = k + 1$ e ir al paso 3.
  else
    Establecer $x^{k+1} = x^k$, $\Delta k + 1 = \theta \Delta^k$, $k = k + 1$
  end if
end procedure

Ambos algoritmos generan una secuencia que verifica la condición débil de convergencia de 4.5, es decir,

$$\lim_{k \to \infty} inf ||\nabla f(x^k)|| = 0$$

En este caso, tomamos un índice, j, en PSA tal que

$$f(x^k + \alpha^k d^j) = \min_{i: d^i \in \mathcal{D}^k} f(x^k + \alpha^k d^i) < f(x^k)$$

también mantenga la premisa de 4.5. Esto también ocurre en DFLSA, porque solo tenemos que tomar un tamaño de paso, α^k, tal que

$$f\left(x^k + \frac{\alpha^k}{\delta} d^k\right) \geq \max\left\{ f(x^k + \alpha^k d^k), f(x^k) - \gamma \left(\frac{\alpha^k}{\delta}\right)^2 \right\}, \delta \in (0,1)$$

Ambos esquemas encuentran un tamaño de paso que les permita comprobar la convergencia del algoritmo. En estos algoritmos no necesitamos tener el gradiente de f, por lo que tenemos que comprobar la siguiente condición:

$$\sqrt{\frac{\sum_{i=1}^{n+1} (f(x^i) - \overline{f})^2}{n+1}} \leq tolerancia \tag{4.33}$$

donde $\overline{f} = \frac{1}{n+1}\sum_{i=1}^{n+1} f(x^i)$ y $\{x^i : i = 1, \ldots, n+1\}$ incluye el punto actual y los n puntos anteriormente generados junto con las n direcciones.

Como resultado de esta teoría, tenemos algoritmos que nos permitan generar secuencias e puntos en $\mathbb{R}^n$ que converjan a los puntos óptimos y, bajo ciertas circunstancias, pueden ser el óptimo global. Estos métodos son el principio de la gran cantidad de algoritmos presentados en la literatura. Estos algoritmos proponen formas de alcanzar un óptimo global de funciones basadas en una idea inspirada en la naturaleza.

Incluso si la gran cantidad de algoritmos que presentaremos en las próximas secciones, el diseño de cada uno es similar a cómo esta teoría propone el movimiento, es decir, metaheurísticas basando su movimiento en una dirección dada por un vector y el criterio de parada basado en condiciones que normalmente presentan la optimalidad del mejor individuo de la población.

4.5. Comparaciones por parejas

Las comparaciones por parejas son el tipo de tests estadísticos más simples que un investigador puede aplicar en el contexto de un estudio experimental. Estos tests son útiles para comparar el rendimiento de dos algoritmos cuando se aplican a un conjunto de problemas común. En el análisis de múltiples problemas, se requiere un valor por cada par de algoritmo-problema (frecuentemente un valor medio de varias ejecuciones).

En esta sección, nos centraremos en primer lugar en un procedimiento fácil y potente, que puede proporcionar una primera impresión sobre la comparación: el Test de Signos. Entonces, introduciremos el uso del test de posiciones con signos de Wilcoxon, como un ejemplo de un test no paramétrico para comparaciones estadísticas por parejas.

4.5.1. Test de signos

Una forma popular de comparar el rendimiento general de los algoritmos es contar el número de casos en los que un algoritmo es el ganador (tiene el mejor rendimiento) de entre todos. Algunos autores también utilizan este conteo en estadística inferencial, con un test conocido como Test de signos. Si amso algoritmos comparados son, como se asume en la hipótesis nula, equivalentes, cada uno debería ganar en aproximadamente $n/2$ de n problemas.

El número de victorias se distribuye de acuerdo a una distribución binomial: para un mayor número de casos, el número de victorias está bajo la hipótesis nula de acuerdo a $n(n/2, \sqrt{n}/2)$, lo que permite el uso del Z-test: si el número de victorias es, al menos, $n/2 + 1.96 \cdot \sqrt{n}/2$, entonces el algoritmo es significativamente mejor con $p < 0.05$.

4.5.2. Test de posiciones con signos de Wilcoxon

Este test se utiliza para responder a la siguiente pregunta: ¿Dos muestras representan 2 poblaciones diferentes? Es un procedimiento no paramétrico utilizado en situaciones de testeo de hipótesis, involucrando un deiseño con dos muestras. Esto es análogo al t-test por parejas en procedimientos estadísticos no paramétricos; por lo tanto, es un test por parejas cuyo objetivo es detectar diferencias significativas entre dos muestras,es decir, el comportamiento de dos algoritmos.

Definición 4.10 *Test de Wilcoxon Sea d_i la diferencia entre la puntuación del rendimiento de los algoritmos en el problema i-ésimo de entre n problemas (si esta puntuación se pueden representar en distintas posiciones, pueden ser normalizados en el intervalo [0,1], con el objetivo de no darle prioridad a ningún problema). Las diferencias se ordenan según sus valores absolutos; en caso de empates, se puede aplicar alguno de los métodos disponibles en la literatura (ignorar empates, asignar la posición más elevada, calcular todas las posibilidades y hacer la media de los resultados obtenidos para cada aplicación del test...), aunque recomendamos el uso de hacer la media de las posiciones (por ejemplo, si dos diferencias están empatadas en la asignación de posiciones 1 y 2, entonces se le asignará la posición 1.5 a ambas).*

Sea $\mathbb{R}^+$ la suma de las posiciones para los problemas tales que el primer algoritmo mejoró al segundo, y $\mathbb{R}^-$ la suma de las posiciones del caso contrario. Las posiciones de $d_i = 0$ se dividen a partes iguales entre ambas sumas; si hay un número impar de estas, una de ellas se ignora:

$$\begin{aligned} R^+ &= \sum_{d_i>0} \text{rank}(d_i) + \frac{1}{2}\sum_{d_i=0} \text{rank}(d_i) \\ R^- &= \sum_{d_i<0} \text{rank}(d_i) + \frac{1}{2}\sum_{d_i=0} \text{rank}(d_i) \end{aligned} \tag{4.34}$$

Sea $T = \text{mín}(R^+, R^-)$. si T es menor o igual que el valor de la distribución de Wilcoxon para n grados de libertad, se rechaza la hipótesis nula de igualdad; esto significa que dicho algoritmo supera al otro, con el p-valor asociado.

El test de Wilcoxon es más sensible que el t-test. Asume conmensurabilidad de diferencias, pero solo de forma cualitativa: mayores diferencias contarán más, lo que es probablemente lo que se desea, pero las magnitudes absolutas se ignoran. Desde un punto de vista estadístico, este test es más seguro ya que no asume una distribución normal. Además, los valores atípicos (*outliers*) tienen menos efecto en el test de Wilcoxon que en el t-test. El test de Wilcoxon asume diferencias continuas d_i; por lo tanto, no deberían ser redondeadas a uno o dos decimales, ya que esto reduciría la potencia del test en caso de un número elevado de empates.

4.6. Múltiples comparaciones con un método de control

Una de las situaciones más frecuentes donde el uso de proceedimientos estadísticos es necesario es el análisis conjunto de resultados obtenidos por distintos algoritmos. Los grupos de diferencias entre estos métodos (también llamados bloques) se suelen asociar con los problemas con los que se trabaja en el estudio experimental. Cuando nos referimos a test de comparaciones múltiples, un bloque se compone de tres o más sujetos o resultados, cada uno de ellos correspondiendo a la evaluación de rendimiento del algoritmo sobre el problema.

En el análisis por parejas, si se intenta extraer una conclusión que involucre más de una comparación por parejas, obtendremos un error acumulado procedente de su combinación. En términos estadísticos, estamos perdiendo el control en el error producido por familia (*Family-Wise Error Rate*, FWER), definido como la probabilidad realizar descubrimientos falsos de entre todas las hipótesis cuando se usan los test múltiples por parejas. La verdadera significación estadística de combinar comparaciones por parejas viene dada por:

$$
\begin{aligned}
p &= P(\text{Rechazar } H_0 | H_0 \text{ cierto}) = 1 - P(\text{Aceptar } H_0 | H_0 \text{ cierto}) \\
&= 1 - P(\text{Aceptar } A_k = A_i, i = 1, ..., k-1 | H_0 \text{ cierto}) \\
&= 1 - \prod_{i=1}^{k-1} P(\text{Acceptar} A_k = A_i | H_0 \text{ cierto}) \\
&= 1 - \prod_{i=1}^{k-1} [1 - P(\text{Rechazar } A_k = A_i | H_0 \text{ cierto})] \\
&= 1 - \prod_{i=1}^{k-1} (1 - p_{H_i})
\end{aligned}
$$

Por lo tanto, un test de comparación por parejas, tal como el test de Wilcoxon, no debería usarse para realizar varias comparaciones involucrando un conjunto de algoritmos, porque el FWER no está controlado.

Esta sección está dedicada a describir el uso de diversos procedimientos para comparaciones múltiples considerando un método de control. En este sentido, un método de control puede ser definido como el algoritmo más interesante para el investigador del estudio experimental (normalmente este suele ser la nueva propuesta). Por lo tanto, su rendimiento será contrastado con el resto de algoritmos del estudio.

4.6.1. Test múltiple de signos

Dado un algoritmo de control etiquetado, el test de signos para múltiples comparaciones nos permite destacar aquellos cuyos rendimientos son estatísticamente distintos cuando comparados con el algoritmo de control. Este procedimiento es el siguiente:

1. Representaremos por $x_{i,1}$ y $x_{i,j}$ los rendimientos del algoritmo de control y el algoritmo j-ésimo en el i-ésimo problema.

2. Calculamos las diferencias de signos $d_{i,j} = x_{i,j} - x_{i,1}$. Esto es, emparejar cada rendimiento con el control y, en cada problema, restar el rendimiento del algoritmo de control al rendimiento del j-ésimo algoritmo.

3. Sea r_j el número de diferencias, $d_{i,j}$ que tienen el signo con la menor frecuencia de aparición con un emparejamiento de un algoritmo con el de control.

4. Sea M_1 la respuesta media de una muestra de resultados del algoritmo de control y M_j la respuesta media de una muestra de los resultados del algoritmo j-ésimos. Aplicaremos una de las siguientes reglas de decisión:

   - Para contrastar $H_0 : M_j \geq M_1$ frente $H_1 : M_j < M_1$, rechazaremos H_0 si el número de símbolos negativos es menor o igual al valor crítico de R_j para $k-1$ (número de algoritmos excluyendo el de control), n (número de problemas), y la tasa de error del experimento elegida.
   - Para contrastar $H_0 : M_j \leq M_1$ frente $H_1 : M_j > M_1$, rechazaremos H_0 si el número de símbolos positivos es menor o igual al valor crítico de R_j para $k-1$, n, y la tasa de error del experimento elegida.

4.6.2. Los test de Friedman, Posiciones Alineadas de Friedman y Quade

El test de Friedman es un análogo no paramétrico del análisis paramétrico ANOVA. Puede usarse para responder a la siguiente pregunta: en un conjunto de $k \geq 2$ muestras de n datos cada una, ¿hay al menos dos muestras que representen poblaciones con distintas medias? El test de Friedman es análogo a repetidas medidas ANOVA en procedimientos estadísticos no paramétricos; por lo tanto, es un test de múltiples comparaciones cuyo objetivo es detectar diferencias significativas entre el comportamiento de dos o más algoritmos.

La hipótesis nula del test de Friedman establece la igualdad de medias entre las poblaciones. La hipótesis alternativa se define como la negación de la hipótesis nula, por lo que es no direccional.

El primer paso para calcular el test estadístico es convertir los resultados originales en posiciones. Esto se calcula utilizando el siguiente procedimiento:

1. Reunir los resultados observados para cada par algoritmo-problema.
2. Por cada problema i, ordena los valores de 1 (mejor resultado) a k (peor resultado). Denotaremos estas posiciones como r_i^j $(1 \leq k)$.
3. Para cada algoritmo j, hacemos la media de las posiciones obtenidas en todos los problemas para calcular la posición final $R_j = \dfrac{1}{n}\sum_i r_i^j$.

Por ello, ordena los algoritmos de cada problema por separado; el algoritmo con mejor rendimiento debería obtener la posición 1, el segundo la posición 2, etc. En caso de empate, se recomienda calcular la media de las posiciones. Bajo la hipótesis nula, que establece que todos los algoritmos se comportan de forma similar (por lo tanto sus posiciones R_j deberían ser iguales) la estadística de Friedman F_j se puede calcular como

$$F_j = \frac{12n}{k(k+1)}\left[\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right] \tag{4.35}$$

que se distribuye según una distribución χ^2 con $k-1$ grados de libertad, donde n y k son lo suficientemente grandes (normalmente, $n > 10$ y $k > 5$). Para menor número de algoritmos y problemas, los valores críticos ya han sido calculados.

Iman y Davenport propusieron una derivación del estadístico de Friedman en tnato que esta última métrica suele producir un efecto conservativo no deseado. El estadístico propuesto es

$$F_{ID} = \frac{(n-1)\chi_F^2}{n(k-1)-\chi_F^2} \tag{4.36}$$

que se distribuye de acuerdo a una F-distribución con $k-1$ y $(k-1)(N-1)$ grados de libertad.

Un inconveniente del esquema de posiciones empleado por el test de Friedman es que solo permite comparaciones dentro del mismo conjunto. Cuando el número de algoritmos a comparar es pequeño, esto no puede ser una desventaja, en tanto que las comparaciones en el mismo conjunto pueden no ser significantes.

En el método de posiciones alineadas para el test de Friedman, el valor de una localización se calcula como la media de los rendimientos alcanzados por todos los algoritmos y problemas.

Las diferencias resultantes, que mantienen sus identidades con respecto al problema y la combinación de algoritmos a los que pertenecen, son entonces ordenados de 1 a $k \cdot n$ relativos entre sí.

Este esquema de posiciones es el mismo que se emplea en procedimientos de comparación múltiple que emplean muestras idependientes, como sería el test de Kruskal-Wallis. Las posiciones asignadas a las observaciones alineadas se llaman posiciones alineadas.

El test estadístico de Friedman de Posiciones Alineadas se puede definir como:

$$F_{AR} = \frac{(k-1)\left[\sum_{j=1}^{k}\hat{R}_j^2 - \frac{kn^2}{4}(kn+1)^2\right]}{\frac{kn(kn+1)(2kn+1)}{6} - \frac{1}{k}\sum_{i=1}^{n}\hat{R}_i^2} \tag{4.37}$$

donde $\hat{R}_i$ es igual a la posición total del problema i-ésimo y $\hat{R}_t$ es la posición total del j-ésimo algoritmo.

Finalmente, introduciremos un último test para realizar comparaciones múltiples: el test de Quade. Este test, en contraste al test de Friedman, toma en cuenta el hecho de que algunos problemas son más difíciles o que las diferencias registradas en la ejecución de varios algoritmos sobre ellos son mayores (el test de Friedman considera todos los problemas iguales en términos de importancia). Por lo tanto, las posiciones se calculan calculados en cada problema pueden ser escalados dependiendo de las diferencias observadas en el rendimiento de los algoritmos, obteniendo, como resultado, un análisis de *ranking* con pesos de las muestras de resultados.

El procedimiento empieza encontrando las posiciones r_i^j de la misma forma que el test de Friedman. El siguiente paso requiere los valores originales del rendimiento de los algoritmos x_i^j. Las posiciones se asignan a los propios problemas de acuerdo con el tamaño del rango de la muestra en cada problema. El rango de la muestra dentro de los prolemas i es la diferencia entre las mayores y menores observaciones en dicho problema:

$$\text{Rango en un problema} : i = \max_j x_i^j - \min_j x_i^j \tag{4.38}$$

Obviamente, hay n rangos de muestra, uno por cada problema. Asignando la posición 1 al problema con el menor rango, la posición 2 al problema con el segundo menor rango... Se usan las posiciones medias en caso de empate. Sean $Q_1, Q_2, ..., Q_n$ las posiciones asignadas a los problemas $1, 2, ..., N$, respectivamente.

Finalmente, la posición de problema Q_i se multiplica por la diferencia entre la posición dentro del problema i, r_i^j, y la posición media dentro de los problemas, $(k+1)/2$, para obtener el producto S_i^j, donde

$$S_i^j = Q_i\left[r_i^j - \frac{k+1}{2}\right] \tag{4.39}$$

es un estadístico que representa el tamaño relativo de cada observación dentro de un problema, ajustado para reflejar la significancia relativa del problema en el que aparece. Además, definimos S_j como la suma de cada algoritmo, $S_j = \sum_{i=1}^{n} S_i^j$, para $j = 1, 2, ...k$.

Por conveniencia, y para establecer una relación con el test de Friedman, también usaremos *rankings* sin ajustes de medias,

$$W_i^j = Q_i\left[r_i^j\right] \tag{4.40}$$

y la posición media para el j-ésimo algoritmo, T_j, dado por

$$T_j = \frac{W_j}{n(n+1)/2} \tag{4.41}$$

donde $W_j = \sum_{i=1}^{n} W_i^j$, para $j = 1, 2, ..., k$.

Se deben establecer algunas definiciones para calcular el test estadístico, F_Q. Sean los términos A y B

$$\begin{aligned} A &= \frac{n(n+1)(2n+1)k(k+1)(k-1)}{72} \\ B &= \frac{1}{n}\sum_{j=1} kS_j^2 \end{aligned} \tag{4.42}$$

El test estadístico, F_Q, es

$$F_Q = \frac{(n-1)B}{A-B} \tag{4.43}$$

que se distribuye de acuerdo a la F-distribución con $k-1$ y $(k-1)(n-1)$ grados de libertad. Cuando se calcula el estadístico, nótese que, si $A = B$, debemos considerar que el punto se encuentra en la región crítica de la distribución estadística.

Para cada uno de estos test, una vez se hayan calculado los propios estadísticos, es posible calcular un p-valor mediante aproximaciones normales. Si hallamos la existencia de diferencias significantes, podemos proceder con el procesamiento *post-hoc* para caracterizar estas diferencias.

4.6.3. Procedimientos *post-hoc*

El principal inconveniente para los test de Friedman, Iman-Davenport, Friedman *Aligned* y Quade es que solo pueden detectar diferencias significativas sobre la comparación múltiple en su totalidad, siendo incapaz de establecer comparaciones propias entre algunos de los algoritmos considerados. Cuando el objetivo de la aplicación de tests múltiples es el realizar una comparación considerando un método de control y un conjunto de algoritmos, se puede definir una familia de hipótesis, relacionados con el método de control. Entonces, el uso de un test *post-hoc* puede conducir a la obtención de un p-valoro que determine el grado de rechazo de cada hipótesis.

Una familia de hipótesis es un conjunto de hipótesis de comparación lógicamente interrelacionadas tales que, en $1 \times N$ comparaciones, compara los $k-1$ alogoritmos del estudio (excluyendo el de control) con el método de control, mientras que en $N \times N$ comparaciones, considera las $k(k-1)/2$ posibles comparaciones entre algoritmos. Por lo tanto, la familia estará compuesta de $k-1$ o $k(k-1)/2$ hipótesis, respectivamente, tales que pueden ser ordenadas por su p-valor, desde el menor hasta el mayor.

El p-valor de cada hipótesis en la familia se puede obtener mediante una conversión de posiciones calculados por cada test mediante el uso de una aproximacicón normal. El test estadístico para comparar el i-ésimo y el j-ésimo algoritmo, z depende del procedimiento no paramétrico principal utilizado:

- **Test de Friedman**

$$z = (R_i - R_j)/\sqrt{\frac{k(k+1)}{6n}} \tag{4.44}$$

  donde R_i y R_j son la posiciones medias por el test de Friedman de los algoritmos comparados.

- **Test de Friedman *Aligned***

$$z = (\hat{R}_i - \hat{R}_j)/\sqrt{\frac{k(n+1)}{6}} \tag{4.45}$$

donde $\hat{R}_i$ y $\hat{R}_j$ son las posiciones medias por el test de Posiciones Alineadas de Friedman para los algoritmos comparados.

- **Test de Quade**

$$z = (T_i - T_j)/\frac{k(k+1)(2n+1)(k-1)}{18n(n+1)} \tag{4.46}$$

donde $T_i = \frac{W_i}{n(n+1)/2}$, $T_j = \frac{W_j}{n(n+1)/2}$ y W_i y W_j son las posiciones sin el ajuste de medias por el test de Quade de los algoritmos comparados.

Sin embargo, estos p-valores no son adecuados para comparaciones múltiples. Cuando un p-valor es considerado en un test múltiple, refleja la posibilidad de error de una cierta comparación, pero no toma en consideración el resto de comparaciones pertenecientes a la familia. Si k algoritmos se comparan y en cada comparación el nivel de significancia es α, entonces en una comparación individual la probabilidad de no hacer un error de Tipo 1 (rechazar una hipótesis nula cierta) es (1-α), y la posibilidad de no hacer un error de Tipo 1 en $k-1$ comparaciones es $(1-\alpha)^{k-1}$. Por lo tanto, la posibilidad de hacer uno o más errores de tipo 1 es $1-(1-\alpha)^{k-1}$.

El z-valor en todos los casos se usa para encontrar la probabilidad correspondiente p-value de la tabla de la distribución normal $N(0,1)$, el cual es entonces comparado con un nivel de significancia, α, apropiado. Los tests *post-hoc* se diferencian en la forma en la que ajustan el valor de α para compensar las comparaciones múltiples.

A continuación, definiremos un conjunto de procedimientos *post-hoc* y explicaremos cómo calcular los APVs (*Analysis of Partial Variance*) dependiendo del procedimiento *post-hoc* usado en el análisis. La notación usada para describir el cálculo de las APVs es la siguiente:

- Índices i y j corresponden a una comparación en concreto o una hipótesis en una familia de hipótesis, de acuerdo con un orden incremental de sus p-valores. El índice i siempre se refiere a la hipótesis cuyo APV se está calculando y el índice j se refiere a otra hipótesis de en la familia.

- p_j es el p-valor obtenido por la j-ésimo hipótesis.

Los procedimientos del ajuste de p-valores pueden ser clasificados en distintos casos:

- **Un paso**:

  - El procedimiento de Bonferroni-Dunn: Ajusta los valores de α en un único paso mediante su división entre el número de comparaciones realizadas $(k-1)$. Este procedimiento es el más simple, pero el menos potente.
    Bonferroni APV_i: mín$\{v, 1\}$, donde $v = (k-1)p_i$.

- ***Step-down***:

  - El procedimiento de Holm: Ajusta el valor de α en una forma *step-down*. Sean $p_1, p_2, ..., p_{k-1}$ p-valores ordenados (de menor a mayor), tal que $p_1 \leq p_2 \leq \cdots \leq p_{k-1}$, y sean $H_1, H_2, ..., H_{k-1}$ las hipótesis correspondientes. El procedimiento de Holm rechaza H_1 a H_{i-1} si i es el menor entero tal que $p_i > \alpha(k-1)$. Este procedimiento empieza con el p-valor con mayor significancia. Si p_1 es menor que $\alpha/(k-1)$, se rechaza la hipótesis correspondiente y procedemos a comparar p_2 con $\alpha/(k-2)$. Si la segunda hipótesis es rechazada, el test

procede con la tercera, y así sucesivamente. En cuanto alguna hipótesis nula no pueda ser rechazada, el resto de hipótesis se mantienen también.
Holm APV_i: mín$\{v, 1\}$, donde $v = \text{máx}\{(k-j)p_j : l \leq j \leq i\}$

- El procedimiento de Holland: Ajusta el valor de α en una forma *step-down*, como el método de Holm. Rechaza H_1 a H_{i-1} si i es el menor entero tal que $p_i > 1-(1-\alpha)^{k-i}$
  Holland APV_i: mín$\{v, 1\}$, donde $v = \text{máx}\{1-(1-p_j)^{k-j} : l \leq j \leq i\}$
- El procedimiento de Finner: Ajusta el valor de α en una forma *step-down*, como el método de Holm y el método de Holland. Rechaza H_1 a H_{i-1} si i es el menor entero tal que $p_i > 1-(1-\alpha)^{(k-1)/i}$
  Finner APV_i: mín$\{v, 1\}$, donde $v = \text{máx}\{1-(1-p_j)^{(k-1)/j} : l \leq j \leq i\}$

▪ ***Step-up***

- El procedimiento de Hochberg: Ajusta el valor de α de una forma *step-up*. Funciona comparando el mayor p-valor con α, el siguiente mayor p-valor con $\alpha/2$, y así sucesivamente, hasta que se encuentre una hipótesis que se pueda rechazar. Todas las hipótesis con menor p-valor son rechazadas también.
  Hochberg APV_i: máx$\{(k-j)p_j : (k-1) \geq j \geq i\}$
- El procedimiento de Hommel: Este es un procedimiento más complicado que el resto, funciona encontrando el mayor j para el que $p_{n-j+k} > k\alpha/j$ para todo $k = 1, ..., j$. Si no existe tal j, se rechazan todas las hipótesis; en otro caso, se rechazan todas que cumplan $p_i \leq \alpha/j$.
  Hommel APV_i: Compruebése el algoritmo para el APV de Hommel en el Algoritmo 12

Algorithm 12 Método para calcular el APV del test de Hommel

```
1:  procedure (HommelAPV)
2:      Establecer APV_i = p_i ∀i
3:      for each j = k−1, k−2, ..., 2 (en dicho orden) do
4:          Establecer B = ∅
5:          for each i, i > (k−1−j) do
6:              Calcular el valor c_i = (j · p_i)/(j+i−k+1)
7:              Establecer B = B ⋃ c_i
8:          end for
9:          Encontrar el menor valor c_i en B → c_min
10:         if APV_i < c_min then
11:             Establecer APV_i = c_min
12:         end if
13:         for each i, i ≤ (k−1−j) do
14:             Establecer c_i = mín(c_min, j · p_i)
15:             if APV_i < c_i then
16:                 Establecer APV_i = c_i
17:             end if
18:         end for
19:     end for
20: end procedure
```

- El procedimiento de Rom: Es una modificación del procedimiento de Hochberg para aumentar su potencia. Se comporta de la misma forma que el procedimiento de Hochberg,

excepto que los α-valores se calculan a través de la expresión:

$$\alpha_{k-i} = \left[\sum_{j=1}^{i-1} \alpha^j - \sum_{j=1}^{i-2} \binom{i}{k} \alpha_{k-1-j}^{i-j} \right] / i \tag{4.47}$$

donde $\alpha_{k-1} = \alpha$ y $\alpha_{k-2} = \alpha/2$
Rom APV_i : máx$\{(r_{k-j})p_j : (k-1) \geq j \geq i\}$, donde r_{k-j} pueden obtenidos a partir de la ecuación 4.47

- **Dos pasos**

  - El procedimiento de Li: Se propuse un procedimiento de rechazo en dos pasos:
    1. Rechazar todas las H_i si $p_{k-1} \leq \alpha$. En otro caso, aceptar las hipótesis asociadas a p_{k-1} e ir al siguiente paso.
    2. Rechazar las H_i restantes con $p_i \leq (1 - p_{k-1})/(1-\alpha)\alpha$.
       Li APV_i: $p_i/(p_i + 1 - p_{k-1})$

4.6.4. Estimación de contraste

La estimación de contrate (*Contrast Estimation*) basada en medias puede ser usada para estimar la diferencia entre rendimiento de dos algoritmos. Asume que las diferencias esperadas entre rendimiento de los algoritmos son iguales a través de los problemas. Por lo tanto, el rendimiento de los algoritmos está reflejado por las magnitudes de las diferencias entre ellos en cada dominio.

El interés de este test se basa en la estimación de contrate entre las medias de las muestras de resultados considerando todas las comparaciones por parejas. El test obtiene una diferencia cuantitativa calculada usando las medias entre los dos algoritmos en múltiples problemas, procediendo como sigue:

1. Para cada par de k algoritmos en el experimento, calculamos la diferencia entre el rendimiento de los dos algoritmos en cada uno de los n problemas. Esto es, calculamos las diferencias

   $$D_{i(u,v)} = x_{iu} - x_{iv} \tag{4.48}$$

   donde $i = 1, ..., n$; $u = 1, ..., k$; $v = 1, ..., k$.

2. Encontrar la media para cada conjunto de diferencias (Z_{uv}, que puede ser considerado como un *estimador no ajustado* de las medias del algoritmo u y v, $M_u - M_v$). En tanto que $Z_{uv} = Z_{vu}$, solo se necesita calcular Z_{uv} en los casos donde $u < v$. También, nótese que $Z_{uu} = 0$.

3. Calcular la media de cada conjunto de medias no ajustadas teniendo el mismo primer subíndice, m_u:

   $$m_u = \frac{\sum_{j=1}^{k} Z_{uj}}{k}, \qquad u = 1, ..., k \tag{4.49}$$

4. El estimador de $M_u - M_v$ es $m_u - m_v$, donde $u, v \in [1, k]$.

Estos estimadores se pueden entender como una medida de rendimiento global avanzada. Aunque este test no puede aportar una probabilidad de error asociada con el rechazo de la hipótesis nula de igualdad, es especialmente útil para estimar en cuánta cantidad un algoritmo tiene un mejor rendimiento que otro.

Capítulo 5: Problema y Diseño Experimental

En este capítulo se presenta el problema que se aborda con el algoritmo creado, así como las necesidades que surgen por el interés de identificar el aporte de este frente al panorama actual del campo en términos de rendimiento, las soluciones que existen para satisfacer dicha necesidad y cuál de ellas es la más adecuada.

5.1. Descripción del problema

El problema que se va a abordar en este proyecto es el de la optimización de problemas de tipo combinatorio.

Definición 5.1 *Un **problema de optimización combinatoria** se define como aquel en el que el conjunto de soluciones posibles es discreto. Es decir, se trata de un problema de optimización que involucra una cantidad finita o numerable de soluciones posibles.*

Este tipo de problemas se diferencia de los problemas de optimización continuos, en los cuales el conjunto de soluciones posibles es infinito e incontable.

Dentro del campo de la optimización combinatoria es común que la mayoría de los procesos de resolución de problemas no puedan garantizar la solución óptima, incluso dentro del contexto del modelo que se esté utilizando. Sin embargo, la aproximación al óptimo suele ser suficiente para resolver los problemas en la práctica.

Con el fin de poder estudiar más a fondo el comportamiento de los distintos algoritmos que se han estado desarrollando era necesario elegir un problema determinado con el que trabajar. En nuestro caso, se ha elegido la generalización del problema comúnmente llamado "problema de la mochila" (*Knapsack Problem* (KP)): ***Quadratic Knapsack Problem* (QKP)**.

Antes de definir el problema, justificaremos por qué se ha elegido este problema. La primera razón, y posiblemente la más importante, es la ausencia de *benchmarks* para problemas combinatorios *expensive*, por lo que nos hemos visto obligados a crear el nuestro propio para un futuro. Ante este panorama también debemos encontrar un problema específico adecuado sobre el que trabajar desde cero y que resulte de interés. Como el objetivo inicial estaba relacionado con redes neuronales, buscamos un problema que puede tener representación binaria (`1` para la elección de elementos, `0` para el caso contrario). En este sentido, el QKP cumple con este requisito, además tiene interés añadido debido a que es un problema con restricciones y constituye una alternativa moderna de un problema clásico; además de que podemos generar instancias de este problema con distintos tamaños. Además, es un problema que tiene muchas aplicaciones en el mundo real en situaciones donde los recursos con distintos niveles de interacción tienen que distribuirse entre distintas tareas, por ejemplo, asignar compañeros de equipo a distintos proyectos donde las contribuciones de cada

miembro se consideran de forma individual y por parejas. Por lo tanto, teniendo en cuenta la falta de *benchmarks* y referencias, el QKP resulta ser una buena opción, ya que es un problema difícil, costoso y moderno.

5.1.1. Quadratic Knapsack Problem

Se procede a definir en profundidad dicho problema. En primer lugar, se tienen n elementos donde cada elemento j tiene un peso entero positivo w_j. Adicionalmente, se nos da una matriz de enteros no negativos de tamaño $n \times n$, $P = \{p_{ij}\}$, donde p_{jj} es el beneficio asociado a elegir el elemento j y $p_{ij} + p_{ji}$ es el beneficio que se alcanza si ambos elementos i, j, con $i < j$ son seleccionados. Consideramos que una combinación de elementos es una solución a QKP cuando peso el total (la suma del peso de todos los elementos seleccionados) no superan la capacidad máxima de la mochila dada, c. Así, el problema consiste el maximizar el beneficio total sin sobrepasar la capacidad máxima.

Por conveniencia en la notación, establecemos que $N = \{1, \ldots, n\}$ denotará el conjunto de elementos. Representando la lista de elementos de forma binaria, x_j, para indicar si el elemento j ha sido seleccionado (su valor será 0 si no ha sido seleccionado, 1 en caso contrario), el problema podrá ser formulado de la siguiente forma:

$$\begin{aligned} \text{maximizar} \sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j \\ \text{sujeto a} \sum_{j \in N} w_j x_j \leq c \\ x_j \in \{0, 1\}, j \in N \end{aligned} \tag{5.1}$$

Sin pérdida de generalidad, podemos suponer que:

- $\max_{j \in N} w_j \leq c < \sum_{j \in N} w_j$
- La matriz de beneficios es simétrica, es decir, $p_{ij} = p_{ji}$, $\forall j > i$.

Una vez definido el problema, es fácil ver por qué es una versión generalizada del KP. KP se puede obtener a partir de QKP si $p_{ij} = 0$, para todo $i \neq j$. También se considera una versión restringida del *Quadratic 0-1 Programming Problem* (QP), el cual se define como 5.1 sin la restricción de capacidad.

Como uno cabría esperar, debido a su generalidad, el QKP tiene un amplio espectro de aplicaciones. Witzgall [36] presentó un problema que surge en telecomunicación cuando un número de localizaciones para satélites tienen que ser seleccionados, tales que el tráfico global entre estas estaciones se maximice y la limitación de presupuesto se cumpla; este problema resulta ser un QKP.

5.1.2. Datos del problema

Utilizaremos 97 archivos de datos generados aleatoriamente de (QKP) *instances*, los cuales se pueden distribuir de forma que se indica en la Tabla 5.1

Se entiende como "densidad" al porcentaje de beneficios combinados positivos, es decir, $p_{ij} > 0$. Particularmente QP tendría densidad 0 %.

Ahora bien, todos los archivos tienen el mismo formato, lo cual resulta útil para definir funciones capaces de obtener los datos más relevantes para nuestros algoritmos. El formato que siguen los archivos es el siguiente:

- La referencia de la instancia: Su nombre.
- El número de variables (n)
- Los coeficientes lineales de la función objetivo p_{jj}
- Los coeficientes cuadráticos p_{ij}: representados en líneas
- Una línea en blanco
- 0 si la restricción es de tipo $\leq$, lo cual siempre va a ocurrir ya que estamos considerando instancias QKP.
- La capacidad c de la mochila.
- Los coeficientes de capacidad/peso, w_j.
- Algunos comentarios.

Tabla 5.1: Datos del Problema

Número de variables	Densidad	Número de archivos
n = 100	25 %	10
	50 %	10
	75 %	10
	100 %	9
n = 200	25 %	9
	50 %	10
	75 %	10
	100 %	10
n = 300	25 %	9
	50 %	10

5.2. Diseño Experimental

5.2.1. Criterio de Parada

Se han elegido las instancias mencionadas anteriormente ya que también se proporcionaban algunos resultados de otros algoritmos, por lo que resultaba conveniente a la hora de comprobar si los resultados obtenidos por nuestros algoritmos bases eran competitivos. Ya que, en el caso de que no lo fuesen, tendríamos que buscar otros algoritmos base sobre los que trabajar. Sin embargo, nos encontramos con un problema, esto es, el criterio de parada presentado en estos casos es el tiempo. Adicionalmente, el tiempo de parada también depende del número de elementos, n, de forma que se tiene:

- Para $n = 100$, se tienen **5 segundos** de ejecución.

- Para $n > 100$, en nuestro caso, $n = 200$ y $n = 300$, se tienen **30 segundos** de ejecución.

Como se ha indicado antes, utilizar el tiempo de ejecución como criterio de parada resulta un problema. Esto se debe a que no es un criterio de parada fiable, ya que depende de la capacidad de computación de cada ordenador. No es comparable la velocidad de los ordenadores actuales con la velocidad de los ordenadores de dentro de 10 años, de la misma no podemos comparar el rendimiento de un ordenador de hace 10 años con respecto a uno actual. E incluso dentro de los ordenadores de la misma generación, dependerá de las características propias de cada computador. Por lo que se llega a la conclusión de que si se quiere que los resultados obtenidos en este trabajo puedan ser usados como referencia, o incluso si se quiere recrear el trabajo, en un futuro se debe cambiar el criterio de parada a algo portable, a algo que sea independiente de cuándo se produzca el experimento. Por ello, se ha decidido cambiar el criterio de parada a un número de iteraciones máximo. Este criterio sí es portable, ya que independientemente de qué tipo de computador se utilice para obtener los resultados siempre se van a obtener los mismos resultados utilizando los mismos parámetros.

Primero debemos indicar lo que entenderemos por iteraciones. Denotaremos como iteraciones al número de veces que se repite el procedimiento completo un determinado algoritmo, en nuestro caso se va a traducir en lo siguiente:

- Para el algoritmo *Random*, el número iteraciones será el número de veces que generemos una solución de forma aleatoria.
- Para los algoritmos genéticos, una iteración puede suponer un número distinto de evaluaciones. En los algoritmos genéticos al número de iteraciones se llama también generaciones.

Dicho esto, para elegir el número de iteraciones máximo que se iba a utilizar se tomó como referencia el tiempo establecido anteriormente. En tanto se tenía elegir un número de iteraciones como criterio de parada, se decidió estudiar a qué equivalía actualmente el criterio de parada por tiempo establecido. Para ello, mediante una variable `contador`, se ejecutaron todos los archivos con el tiempo como criterio de parada y se mostraba por pantalla el número de iteraciones que se había alcanzado. Tras realizar una media de todos estos datos y redondearlo, obtenemos que el nuevo criterio de parada es **90000 iteraciones** para todos los archivos. El que el número de iteraciones máximo no dependa del número de elementos como lo hacía el tiempo de ejecución máximo se puede justificar en tanto que se necesita más tiempo para realizar todos los cálculos si aumenta el número de datos.

Además, para asegurarnos que realmente ambos criterios de parada eran equivalentes, se ejecutaron todos los archivos usando el algoritmo base con criterio de parada por iteraciones y por tiempo. Nótese que no solo se almacenan las soluciones finales, si no que también se almacenan las soluciones intermedias llegado a ciertos porcentajes de la ejecución. Tras esto, se compararon ambos resultados y se pudo comprobar que, efectivamente, eran equivalentes.

Por lo tanto, se puede decir con seguridad que los cambios que apliquemos a un criterio de parada en específico se puede traducir a un cambio en el otro criterio de parada. Esto cabe la pena destacarlo ya que, recordemos, el objetivo de este trabajo es crear un algoritmo útil y competitivo para tratar con problemas *expensive*, por lo que queremos obtener buenos resultados en un tiempo muy reducido.

Para simular de forma eficiente y suficiente esta reducción de tiempo, se propuso el siguiente cambio con respecto al tiempo como criterio de parada:

- En vez de ejecutar los archivos con $n = 100$ durante 5 segundos, lo reduciremos a 25ms.

- En vez de ejecutar los archivos con $n > 100$ durante 30 segundos, lo reduciremos a 150ms.

Realizamos un cálculo básico para comprobar qué proporción de la ejecución estaremos realizando con estos nuevos tiempos:

$$\frac{25 \cdot 100ms}{5 \cdot 1000ms} = 0.5 \qquad \frac{150 \cdot 100ms}{30 \cdot 1000ms} = 0.5$$

Por lo tanto, obtenemos que solo utilizaremos el 0.5 % inicial de cada ejecución, lo que podemos traducir en que nuestro nuevo criterio de parada serán **450 iteraciones**.

Con el fin de comprobar que esta reducción había sido suficiente y necesaria, se ejecutan de nuevo todos los archivos con el algoritmo base ahora con 450 iteraciones y comparamos los resultados con los obtenidos anteriormente con 90000 iteraciones. Podemos ver que, efectivamente, se han obtenido resultados mucho peores.

Finalmente, ya hemos establecido nuestro criterio de parada escalable y tenemos unos resultados base que utilizar. A partir de esto, nuestro objetivo será mejorar estos resultados lo máximo posible.

5.2.2. Parámetros

La ejecución del programa no requiere de ningún tipo de parámetro que se tenga que introducir manualmente. Todos los parámetros que se nombren a continuación vienen definido dentro del código del programa `main` como constantes globales o que dependen del propio problema.

En primer lugar, hablemos sobre las constantes globales:

- `NEVALUACIONESMAX`: Es el número de iteraciones máximas, mantendremos su valor a 450 por lo explicado en el apartado anterior.
- `NTRIES`: Para que nuestros resultados no sean muy dependientes de la aleatoriedad, es necesario ejecutar cada archivo cierta cantidad de veces y obtener la media. Esta media será verdaderamente el resultado que guardaremos y compararemos con el resto. Para que puedan ser comparados estadísticamente los resultados, pero a la vez manteniendo un tiempo de ejecución razonable, se establece su valor a 50.
- `INITSEED`: En relación con la constante anterior, es necesario no utilizar siempre la misma semilla de aleatoriedad, ya que eso resultaría en obtener siempre los mismos resultados, por lo que ejecutarlos varias veces sería una pérdida de tiempo. Así pues, tendremos que utilizar distintas semillas para cada ejecución. Además, por conveniencia, es recomendable que las semillas utilizadas sean comunes a todas las sucesivas ejecuciones de los distintos algoritmos (una vez más, para asegurarnos que la aleatoriedad afecta a todos los algoritmos de forma similar) y que siempre se obtengan los mismos resultados aún si ejecutamos el mismo archivo varias veces. Por ello, una solución es establecer un valor de semilla inicial arbitrario y utilizar este para generar el resto de semillas.
- `numcro`: Es el tamaño de la población para los AGs. En tanto que el objetivo es obtener buenos resultados en pocas iteraciones, no nos podemos permitir tener un tamaño de población excesivamente grande; ya que a mayor población, más evaluaciones deberán realizarse por generación y dará lugar a que se realicen muy pocos cambios de generación (lo que significa pocas iteraciones del algoritmo). Por ello, estableceremos un tamaño de la población lo suficientemente pequeño para que esto no resulte un inconveniente, pero que no se considere una micropoblación: nos decantaremos por un tamaño de 10 individuos.

En definitiva, a modo de resumen, podemos rápidamente visualizar en la Tabla 5.2 los distintos parámetros con los que trabajaremos, cuáles son sus usos y sus valores.

Tabla 5.2: Resumen de parámetros utilizados

Parámetros	Resumen	Valor
`NEVALUACIONESMAX`	Criterio de parada: Número de Iteraciones Máximas para cada algoritmo	450
`NTRIES`	Número de veces que se va a ejecutar el algoritmo por cada archivo	50
`INITSEED`	Semilla inicial para poder generar el resto de semillas que utilizaremos para los algoritmos	5
`EvaluacionMAX`	Criterio de parada: Número de Iteraciones Máximas dicho algoritmo	`NEVALUACIONESMAX`
`Semilla`	Semilla de aleatoriedad para determinada ejecución del algoritmo	Valor generado aleatoriamente por `INITSEED`
`numcro`	Número de elementos de la población de soluciones	10

Capítulo 6: Algoritmos de Referencia

En este capítulo se presentan los dos algoritmos que se utilizaran como base para desarrollar un algoritmo competitivo para problemas *expensive*, se describirán con sus referencias y se indicarán algunas características; aunque se explicarán de forma más detallada los componentes propios de cada uno en el siguiente capítulo.

6.1. Algoritmo Genético Estacionario Uniforme

En primer lugar, debemos explicar brevemente la importancia de los Algoritmos Genéticos (AG) y, posteriormente, justificar la elección de su versión Algoritmo Genético Estacionario Uniforme (AGEU) (*Steady-State Genetic Algorithm*, SSGA).

Los seres vivos son solucionadores de problemas de forma natural. Exhiben una versatilidad que ponen en evidencia hasta a los mejores programas. Esta observación es especialmente humillante para los informáticos, que necesitan utilizar meses e incluso años de esfuerzos intelectuales en un algoritmo, mientras que estos organismos obtienen sus habilidades a través de los aparentemente indirectos mecanismos de evolución y selección natural.

Los investigadores más pragmáticos observan el notable poder de la evolución como algo que simular. La selección natural elimina uno de los mayores inconvenientes en el diseño de software: especificar de antemano todas las características de un problema y las acciones que dicho programa tendría que tomar para tratar con ellas. Aprovechando los mecanismos de evaluación, los investigadores pueden ser capaces de "reproducir" programas que resuelvan problemas incluso cuando nadie pueda comprender enteramente su estructura. Efectivamente, estos llamados **algoritmos genéticos** han demostrado la habilidad de hacer avances en el diseño de sistemas complejos.

Los AGs hacen posible explorar un rango mucho más amplio de posibles soluciones a un problema que programas convencionales. Además, en los estudios realizados sobre la selección natural de programas bajo condiciones controladas bien entendidas, los resultados prácticos alcanzados pueden aportar cierto conocimiento sobre los detalles de cómo la vida y la inteligencia evolucionan en el mundo natural.

El funcionamiento de un AG viene dado por el siguiente pseudocódigo (Algoritmo 13)

En el caso de la versión AGEU, su pseudocódigo viene representado en 14. Claramente sigue la misma estructura, pero presenta dos especificaciones:

- Estacionario (E): En relación con el operador de selección. Se enfrentan las soluciones hijas con las de la población de la generación anterior y mantenemos las mejores.
- Uniforme (U): En relación con el cruce de las soluciones. Las soluciones hijas van a tener de partida los elementos comunes a ambas soluciones padres. El resto de los elementos de los

Algorithm 13 Algoritmo Genético

procedure (AG)($EMax > 0, nelem > 0, pcruce \in [0,1], pmut \in [0,1]$)
  Generar una población inicial aleatoria
  Calcular la función *fitness* de cada individuo
  `generacion` $= 0$
  while `generacion` <EMax **do**
    Calcular el número de parejas a formar para el cruce $\rightarrow ncruce = pcruce * nelem$
    Seleccionar aleatoriamente con repetición $4 * ncruce$ soluciones de la población
    Aplicar torneo de 2 en 2 soluciones del conjunto anterior y almacenar solo la mejor $\rightarrow$ `padres`
    for $i \in [0, ncruce]; i+ = 2$ **do**
      Generar 2 hijos cruzando `padres`$[i]$ y `padres`$[i+1]$
      Aplicar el Operador de Reparación sobre ambos hijos
      Calcular la función *fitness* de cada hijo
      Almacenar dichos hijos $\rightarrow$ `hijos`
    end for
    Calcular el número de soluciones a mutar $\rightarrow nmut = pmut * nelem$
    Mutar $nmut$ soluciones distintas
    Calcular la función *fitness* de las nuevas soluciones
    Aplicar el Operador de Selección sobre la población actual e `hijos`
    `generacion` = `generacion`+1
  end while
end procedure

hijos se obtienen de forma que para algunos elementos $\texttt{hijo}_i$ los obtiene de $\texttt{padre}_i$ y el resto los obtiene del otro padre. El objetivo de esto es preservar selecciones prometedoras.

También se estuvieron barajando otra versión estudiada durante la asignatura de Metaheurísticas:

- Generacional (G): En relación con el operador de selección. Sustituimos la antigua generación por la nueva.

La versión generacional puede ocasionar que se pierda la mejor solución hasta el momento, lo que impediría que se pudiese seguir una búsqueda profundizando en dicha solución. Como tenemos pocas iteraciones, no nos podemos permitir buenas soluciones sin ningún tipo de garantía, así que no sería un buen enfoque inicial.

A efectos prácticos, se ha usado los resultados y el análisis que realicé en el trabajo "Problemas con técnicas basadas en poblaciones" de la asignatura Metaheurísticas (curso 2021-2022), donde se debía comparar experimental y teóricamente los distintos algoritmos genéticos y meméticos. En dicho trabajo se llega a la conclusión de que la mejor opción es utilizar AGEU. En tanto que no es el mismo problema, no podemos garantizar que este sea el caso para este problema con solo esa información, pero sabemos que es un buen punto de partida.

6.1.1. Pseudocódigo

A efectos prácticos, este pseudocódigo (Algoritmo 14) se diferenciará del anterior (Algoritmo 13) en el cruce, ya que solo cruzaremos dos padres no necesitaremos el parámetro *pcruce*, y que en el operador de selección especificaremos que es estacionario.

Como se ha dicho al principio de este capítulo, una explicación más detallada junto con el pseudocódigo de cada una de las componentes será dada en el siguiente capítulo.

Algorithm 14 Algoritmo Genético Estacionario Uniforme

procedure (AG)($EMax > 0, nelem > 0, pmut \in [0, 1]$)
    Generar una población inicial aleatoria
    Calcular la función *fitness* de cada individuo
    `generacion` = 0
    while `generacion` <EMax **do**
        Seleccionar aleatoriamente 4 soluciones de la población sin repetición 2 a 2
        Aplicar torneo de 2 en 2 soluciones del conjunto anterior y almacenar solo la mejor → `padres`
        Generar 2 hijos cruzando $\texttt{padres}_1$ y $\texttt{padres}_2$
        Aplicar el Operador de Reparación sobre ambos hijos
        Calcular la función *fitness* de cada hijo
        Almacenar dichos hijos → `hijos`
        Calcular el número de soluciones a mutar → $nmut = pmut * nelem$
        Mutar $nmut$ soluciones distintas
        Calcular la función *fitness* de las nuevas soluciones
        Aplicar el Operador de Selección Estacionario sobre la población actual e `hijos`
        `generacion` = `generacion`+1
    end while
end procedure

6.1.2. Componentes

Operador de Reparación

Cuando se realiza el cruce de dos soluciones pueden ocurrir dos casos:

- El resultado del cruce sea una solución factible.
- El resultado del cruce no sea una solución factible.

Más adelante en este capítulo se explicarán cómo son los cruces y se entenderá por qué es posible que el resultado de un cruce sea una solución no factible. En este caso, no es lógico desechar al "hijo", por lo que debemos "arreglarlo" para que se vuelva una solución. Ese es el objetivo del Operador de Reparación.

En nuestro caso lo vamos a aplicar en ambos casos (que el "hijo" sea solución o no). Si el "hijo" no es solución factible es obvio el por qué necesitamos aplicarlo. Si el "hijo" es solución factible lo aplicaremos para asegurarnos que no hay huecos libres, es decir, que no hay elementos adicionales que podrían introducirse adicionalmente. Esto último se hace ya que queremos maximizar el valor de la solución, por lo que si queremos que sea mínimamente competente para cuando intentemos introducirlas en la población.

Entonces seguimos el siguiente proceso, ilustrado en el pseudocódigo 15:

- Si no es solución factible (su peso supera al peso máximo): Se deberán eliminar elementos del "hijo" hasta que constituya una solución factible. La forma de eliminar elementos será usando

Greedy, es decir, se eliminarán los elementos con menor proporción *valor_acumulado/peso*. Esta lógica viene dada por un intento de eliminar el máximo peso posible sin reducir mucho el valor final cuando se vuelva una solución.

- Si es solución factible (su peso no supera al peso máximo): Se buscará, utilizando Greedy, un elemento para introducir. En este caso, nos interesa encontrar el elemento con mayor proporción *valor_acumulado/peso*, ya que eso nos permitiría potencialmente aumentar significativamente el valor total (evaluación de la función *fitness*) de la solución. Esto es, al tener en cuenta el peso puede llegar a resultar en que seamos capaces de introducir más elementos, pudiendo superar finalmente el valor que se tendría si se introdujese el elemento con el mayor valor acumulado pero no permitiese introducir más elementos. Este proceso se repetirá hasta que no sea capaz de introducir ningún otro elemento en la solución.

Algorithm 15 Operador de Reparación

```
1: procedure (Operador Reparación)(hijo)
2:     Calcular el peso total de hijo → pesoHijo
3:     if pesoHijo >c then
4:         while pesoHijo >c do
5:             Eliminar elemento usando Greedy
6:         end while
7:     else
8:         anadido = true
9:         while anadido do
10:            anadido = Añadir elemento usando Greedy
11:        end while
12:    end if
13: end procedure
```

Téngase en cuenta que al eliminar y añadir elementos del "hijo", se debe recalcular su peso total.

Operador de Cruce

El cruce es un operador genético usado para variar los cromosomas de una generación a otra. Dos soluciones obtenidas de la población con anterior se cruzarán con el objetivo de producir una descendencia superior.

Nos encontramos con distintos tipos de cruces básicos:

- **Cruce en un punto**: Dados dos padres, se le asignan los elementos de $\texttt{padre}_1$ a $\texttt{hijo}_1$ y de $\texttt{padre}_2$ a $\texttt{hijo}_2$ hasta cierto cromosoma elegido con anterioridad. A partir de dicho cromosoma, cambiaremos la asignación de forma que $\texttt{hijo}_1$ hereda de $\texttt{padre}_2$ e $\texttt{hijo}_2$ hereda de $\texttt{padre}_1$. Un ejemplo de este tipo de cruce podría ser:

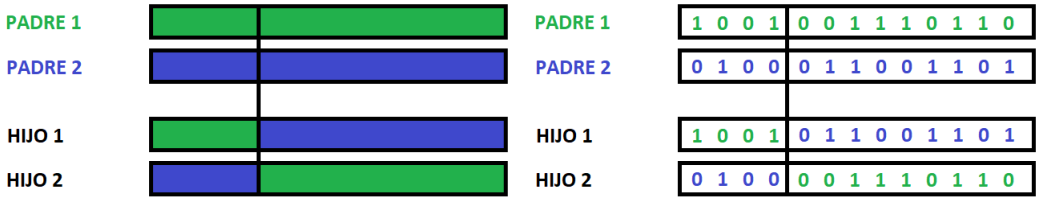

Figura 6.1: Cruce en un punto

- **Cruce en dos puntos**: Sigue la misma lógica que el anterior, solo que elegimos dos puntos a partir de los cuales se cambian los elementos de qué padre se asignan a cada hijo. Un ejemplo de este tipo de cruce podría ser:

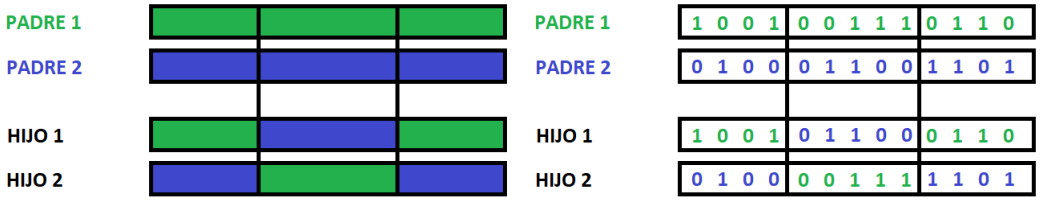

Figura 6.2: Cruce en dos puntos

- **Cruce uniforme**: En este caso, en cada cromosoma se elige de forma aleatoria de qué padre lo hereda, cumpliéndose que si un hijo hereda cierto cromosoma de un padre, el otro hijo deberá heredar el mismo cromosoma del otro padre. Un ejemplo de este tipo de cruce sería:

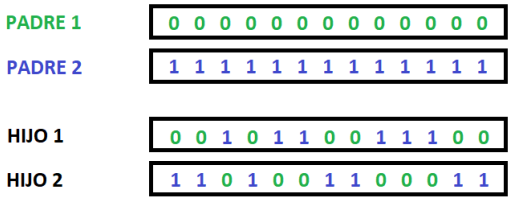

Figura 6.3: Cruce Uniforme

El cruce de dos soluciones buenas no tiene por qué siempre dar lugar a una solución mejor o igual de buena. Sin embargo, si los padres son buenas soluciones, la probabilidad de tener un hijo bueno es elevada; en el caso de que el hijo no sea una buena solución, será eliminado durante el periodo de reemplazo.

En nuestro problema es posible que el cruce de dos soluciones no de lugar a una solución factible. Esto se debe a la elección aleatoria de qué cromosomas elegir, no estamos teniendo en cuenta el peso que se está alcanzando al asignar cada elemento; por lo que es totalmente posible que al asignar los elementos a cada hijo se sobrepase la capacidad máxima, dejando por ello de ser una solución factible.

En nuestro caso, realmente utilizamos una mezcla de cruce en un punto y cruce uniforme. Esto es, vamos a asignarle a cada hijo la mitad de cada uno de los padres, pero esta asignación será aleatoria: desordenamos el orden de los índices y lo partimos por la mitad. Esto viene representado en el pseudocódigo 16.

Algorithm 16 Cruce Uniforme

```
1: procedure (Cruce Uniforme)(padre_1, padre_2)
2:     Desordenar los índices que indican la posición de cada elemento
3:     for i in 0..n do
4:         if i <n/2 then
5:             hijo_1[indice[i]] = padre_1[indice[i]]
6:             hijo_2[indice[i]] = padre_2[indice[i]]
7:         else
8:             hijo_1[indice[i]] = padre_2[indice[i]]
9:             hijo_2[indice[i]] = padre_1[indice[i]]
10:        end if
11:    end for
12: end procedure
```

Mutación

Los primeros intentos de mezclar computación y evolución no progresaron porque pusieron énfasis en los textos de biología del momento y confiaban más en la mutación que en el cruce para generar nuevas combinaciones de genes.

La mutación consiste en modificar al azar una muy pequeña parte del cromosoma de los individuos, y permite alcanzar zonas del espacio de búsqueda que no estaban cubiertas por los individuos de la población actual. La mutación sola de por sí generalmente no permite avanzar en la búsqueda de una solución, pero nos garantiza que la población no va a evolucionar hacia una población uniforme que no sea capaz de seguir evolucionando.

De forma similar a lo explicado en el Operador de Reparación, una vez que obtengamos la nueva solución debemos comprobar si podemos introducir más elementos, con el fin de maximizar el valor que puede llegar a tener. Esto también lo haremos siguiendo la misma lógica: ir introduciendo los genes con mayor proporción $valor\_acumulado/peso$.

El pseudocódigo de nuestra implementación de la mutación viene expresada en Algoritmo 17.

Operador de Reemplazo Estacionario

Una vez que hemos generado nuevas soluciones a partir del cruce de soluciones de la población existente, es necesario establecer un criterio sobre qué soluciones se mantienen o insertan en la población para la siguiente generación. En esta versión del Operador de Reemplazo el criterio que se sigue es el enfrentamiento de al población actual con las soluciones hijas, la población resultante estará compuesta de aquellos con mayor valor al calcular su función *fitness*.

En concreto, para lo que se aplica en el algoritmo base, AG, tenemos en cuenta que solo generamos 2 soluciones hijas antes de enfrentarlas con la población. Por ello, se ha optado por un método simplificado de enfrentamiento que consiste en encontrar cuáles son las 2 peores soluciones de la población actual (es decir, aquellas soluciones de la población actual con menor valor *fitness*) y comprobar si las hijas son mejores o no. Se puede seguir su esquema en el pseudocódigo 18. Usaremos los índices de forma que indique un orden de valores *fitness*, por ejemplo, dados $\texttt{padre}_1$ y $\texttt{padre}_2$, se cumplirá que *fitness*($\texttt{padre}_1$) >*fitness*($\texttt{padre}_2$). Además, por conveniencia en la notación, se entenderá que $\texttt{solucion}_i$ >$\texttt{solucion}_j$ significa que el valor *fitness* de $\texttt{solucion}_i$ es mayor al de $\texttt{solucion}_j$.

Algorithm 17 Mutación

```
1: procedure (Mutación)(poblacion, prob_mut)
2:     Calcular el número de cromosomas que mutarán → nmut = n*prob_mut
3:     Almacenar de forma aleatoria sin repetición nmut cromosomas de poblacion → mutacion
4:     for i in 0..nmut do
5:         Elegir dos genes con distinto valor de forma aleatoria
6:         if Al intercambiar los genes sigue siendo válido then
7:             Intercambiar los genes
8:         else
9:             Volver al paso anterior
10:        end if
11:        anadido = true
12:        while anadido do
13:            anadido = Añadir elemento usando Greedy
14:        end while
15:        Calcular el valor de la función fitness
16:    end for
17: end procedure
```

Más adelante en la sección de "Componentes de CHC" de este capítulo se presenta la versión más generalizada de este tipo de enfrentamiento.

Algorithm 18 Operador de Reemplazo Estacionario

```
1: procedure (Op Reemplazo Estacionario)
2:     Calcular los 2 peores padres de la población actual → padre_1, padre_2.
3:     if hijo_1 >padre_1 && hijo_2 >padre_2 then
4:         Intercambiar ambas soluciones de la población por ambos hijos
5:     else if hijo_1 >padre_2 then
6:         Intercambiar la peor solución de la población por el mejor hijo
7:     else
8:         No hacer nada, ya que los dos hijos son peores que las peores soluciones de la población
9:     end if
10: end procedure
```

6.2. CHC

El algoritmo CHC utiliza un método de selección elitista que, combinada con un mecanismo de prevención de incesto y un método para obligar que la población diverja cada vez que converge, permite el mantenimiento de la diversidad de la población. Este algoritmo se ha utilizado de forma exitosa en el pasado para problemas de optimización estáticos.

El algoritmo CHC (*Cross-generational elitist selecition, Heterogeneous recombination and Cataclysmic mutation*) propuesto por Eshelman utiliza un método de selección elitista combinado con un cruce altamente disruptivo para promover la diversidad de la población. La principal característica de este algoritmo es su capacidad de prevenir la convergencia de la población, algo que, como luego comprobaremos, será útil en nuestro problema.

Originalmente, cuando la población converge se pueden tomar dos acciones:

- Reiniciar la población entera de forma aleatoria con excepción de la mejor solución
- Reiniciar la población utilizando la mejor solución como base y generando el resto realizando modificaciones sobre esta.

Sin embargo, esto es solo útil cuando se tienen bastantes evaluaciones. En nuestro problema realizar esto resultaría en una pérdida de tiempo e iteraciones importantes, por lo tanto, no se tendrá en cuenta.

6.2.1. Pseudocódigo

Algorithm 19 Algoritmo CHC

```
1: procedure (AG)(EMax > 0, nelem > 0)
2:     Generar una población inicial aleatoria
3:     Calcular la función fitness de cada individuo
4:     generacion = 0
5:     threshold = n/4
6:     while generacion <EMax do
7:         Calcular el número de parejas a formar para el cruce → ncruce = pcruce * nelem
8:         Desordenar los elementos de la población actual y comprobar si cumplen la condición
       de prevención de incesto (distancia de Hamming >threshold) de dos en dos
9:         if hamming >threshold then
10:            Almacenar las soluciones → parejas
11:        end if
12:        if parejas = ∅ then
13:            if threshold ≠ 0 then
14:                threshold = threshold-1
15:            end if
16:        else
17:            for i∈ [0, parejas.size()]; i=i+2 do
18:                Generar 2 hijos cruzando parejas[i] y parejas[i + 1]
19:                Aplicar el Operador de Reparación sobre ambos hijos
20:                Calcular la función fitness de cada hijo
21:                Almacenar dichos hijos → hijos
22:            end for
23:            Aplicar el Operador de Selección sobre la población actual e hijos
24:        end if
25:        generacion = generacion+1
26:    end while
27: end procedure
```

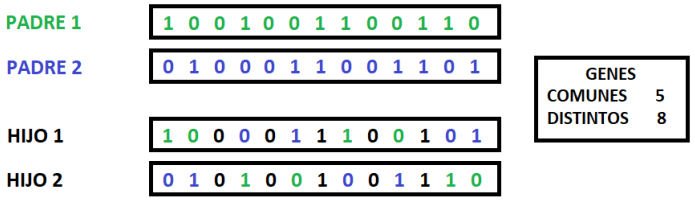

Figura 6.4: Cruce HUX

6.2.2. Componentes

Operador de reparación

Se utilizará el mismo que se ha presentado anteriormente en los componentes de AG. Véase el pseudocódigo 15.

Cruce HUX

El cruce HUX (*Half Uniform Crossover*) se caracteriza por, dados dos cromosomas, asignarle a los resultados del cruce en primer lugar los genes comunes a ambos padres y el resto de los genes serán repartidos a partes iguales entre ambos padres. Es decir, exactamente la mitad de los genes no coincidentes se intercambian en los hijos.

A continuación se detallará en forma de pseudocódigo (20) el comportamiento de este tipo de cruce. Aunque primero se mostrará un ejemplo de este tipo de cruce en Figura 6.4.

Algorithm 20 Cruce HUX

```
1: procedure (Cruce HUX)(padre_1, padre_2)
2:     Asignar los genes comunes de los padres a ambos hijos
3:     Desordenar los índices que indican la posición de cada gen restante → Supongamos tamaño
   m ≤ n
4:     for i in 0..m do
5:         if i <m/2 then
6:             hijo_1[indice[i]] = padre_1[indice[i]]
7:             hijo_2[indice[i]] = padre_2[indice[i]]
8:         else
9:             hijo_1[indice[i]] = padre_2[indice[i]]
10:            hijo_2[indice[i]] = padre_1[indice[i]]
11:        end if
12:    end for
13: end procedure
```

Reemplazo Elitista

Es una generalización del Operador de Reemplazo Estacionario del AG (Algoritmo 18). En este caso tenemos que enfrentar toda la población de hijos (con tamaño menor igual al tamaño de la población) contra toda la generación anterior. Por ello, no podemos seguir el método de obtener

los x peores elementos de la población para enfrentarlos con los hijos; así que optaremos por otro razonamiento.

Se ha optado por, a grandes rasgos, juntar ambas poblaciones (generación actual y descendientes) en una sola y ordenarlas en base a su valor *fitness*. De esta forma, al final tenemos todas nuestras soluciones ordenadas de mejor a peor y solo tenemos que quedarnos con las n para formar la nueva población.

Este comportamiento se puede ver reflejado en su pseudocódigo (Algoritmo 21)

Algorithm 21 Enfrentamiento CHC

1: **procedure** (Enfrentamiento)(*poblacion*, *hijos*)
2: Calcular la función *fitness* de `poblacion` e `hijos`
3: Unir ambas poblaciones → `pobTotal`
4: Unir los valores de ambas poblaciones → `valorTotal`
5: Ordenar los elementos de `pobTotal` según `valorTotal` (orden descendente)
6: Asignar a la población actual los n primeros elementos de `pobTotal`
7: **end procedure**

Capítulo 7: Componentes de la propuesta

En este capítulo se presentan los distintos componentes que se han ido desarrollando como modificaciones de los distintos componentes de los algoritmos de referencia. Es importante mencionar que en este capítulo no solo se harán mención de aquellas modificaciones que han resultado fructuosas, si no también aquellas que no han mejorado los resultados de versiones anteriores. Estas últimas se conformarán su propia sección aparte al final del capítulo. Se describirán detalladamente además de usar pseudocódigo para representarlos.

Téngase en cuenta que solo se explicará brevemente el por qué de estas modificaciones, ya que entraremos en este tema en más profundidad en el siguiente capítulo.

7.1. Histórico

Al tener tan pocas evaluaciones y con un tamaño tan pequeño de la población no podemos permitirnos una exploración del vecindario de las distintas soluciones en direcciones que en el momento se podrían considerar erróneas, es decir, que ocasionarían que empeorasen las soluciones. Por ello, se plantea un sistema capaz de "recordar" buenos elementos y malos elementos de la solución; considerándose "buenos elementos" aquellos que aparecen en gran medida en las mejores soluciones y no se encuentran en las peores, y "malos elementos" aquellos que aparecen en gran medida en las peores soluciones y no se encuentran en las mejores. A esto lo llamaremos **histórico**.

De esta forma, podemos guiar la exploración de vecindario hacia aquellos elementos que han demostrado ser "buenos" y alejarlos de aquellos que han demostrado ser "malos". Esta lógica la podemos aplicar en el Operador de Reparación y/o en la Mutación. Esto se haría sustituyendo la lógica Greedy de tener en cuenta el ratio *valor_acumulado/peso* a la hora de añadir o eliminar elementos por un fomento del uso de los datos del histórico: se intentarán añadir de forma aleatoria entre los mejores elementos y se intentarán eliminar de forma aleatoria entre los peores elementos.

Sin embargo, no podemos permitir que esto guíe toda la ejecución, tenemos que decidir un par de cosas:

- ¿Cuándo se para de recopilar datos para generar el histórico? Si recopilamos información de pocas iteraciones puede ocasionar que los datos obtenidos no sean representativos, ya que es posible que no se hayan alcanzado en el momento suficientes soluciones buenas. Si recopilamos información de demasiadas iteraciones puede suponer que los datos obtenidos no sean representativos, ya que la población hubiese podido converger por lo que no hay ningún elemento que se pudiese considerar "bueno" o "malo"; además de que si se dejan pocas ejecuciones para el uso del histórico no se va a poder alcanzar mucha mejora.
- ¿Durante cuántas ejecuciones deberíamos utilizar el histórico? En otras palabras, ¿una vez obtenido los datos del histórico deberíamos utilizarlos hasta el final del algoritmo? La res-

puesta a esto es: no se debería. Esto es porque no podemos esperar que los datos obtenidos mediante el histórico sean del todo fiables, es decir, podrían estar guiando la solución hasta un máximo local, por lo que las soluciones de la población convergerían rápidamente.

Por lo tanto, llegamos a la conclusión de que no podemos utilizar únicamente el histórico. Por ello, lo que haremos será intercalar varias etapas de recopilación de datos y uso del histórico generado. Durante la recopilación de datos se procederá de la misma forma que lo haría si esta modificación no estuviese añadida, con la excepción de que se irá guardando en un archivo las distintas soluciones que estamos alcanzando para luego hacer el estudio de sus elementos. De las 450 iteraciones totales iremos alternando fases cada 50 iteraciones, de esta forma combinamos en una proporción 4-5 uso del histórico y comportamiento usual. Este esquema se ve representado de forma genérica, por simplicidad, en Algoritmo 22.

Algorithm 22 Estructura General del Histórico

```
1:  procedure (Histórico)
2:      RecDatos = True
3:      for i = 0; i <NEVALUACIONES; ++i do
4:          if i%50==0 && i≠ 0 then /* Puntos de cambio */
5:              if i%100 then /* Empezamos a recopilar datos */
6:                  RecDatos = True
7:                  Eliminar todos los datos actuales del histograma
8:                  Recopilar las soluciones de la población actual
9:              else
10:                 RecDatos = False
11:             end if
12:         end if
13:         if RecDatos then
14:             Ejecutar los componentes de la manera original
15:             Recopilar las nuevas soluciones
16:         else
17:             Ejecutar los componentes usando los datos del histórico
18:         end if
19:     end for
20: end procedure
```

Las modificaciones realizadas al Operador de Reparación y a la Mutación se verán representadas respectivamente en Algoritmo 23 y Algoritmo 24.

7.2. GRASP

En el capítulo 6 se ha explicado por qué seguimos un enfoque *Greedy* en el Operador de Reparación. Sin embargo esto tiene un problema, y esto es que es bastante probable que siempre se estén eligiendo los mismos pocos elementos cada vez que se usa este componente. Si realmente ocurre eso, ocasionaría que la población converja y la evolución se estanque; por ello, debemos encargarnos de encontrar otra alternativa con la que podamos aumentar la diversidad de la población.

Sin embargo, la diversidad en la población es un arma de doble filo. La diversidad es necesaria para explorar el espacio de soluciones, pero puede causar que no finalice el proceso de exploración.

Algorithm 23 Histórico en Operador de Reparación

procedure (Historico_OR)(*hijo*)
    Calcular el peso total de `hijo` → `pesoHijo`
    if `pesoHijo >c` **then**
        Eliminar de forma aleatoria los peores elementos según el histórico hasta que `pesoHijo` deje de sobrepasar `c`
        Si no es posible lo anterior y sigue dándose la condición, eliminar de forma aleatoria los elementos que no aparecen en el histórico hasta que `pesoHijo` deje de sobrepasar `c`
        Si no es posible lo anterior y sigue dándose la condición, eliminar de forma aleatoria los mejores elementos según el histórico hasta que `pesoHijo` deje de sobrepasar `c`
    else
        Añadir de forma aleatoria los mejores elementos según el histórico sin que `pesoHijo` sobrepase `c`
        Añadir de forma aleatoria los elementos que no aparecen en el histórico sin que `pesoHijo` sobrepase `c`
        Añadir de forma aleatoria los peores elementos según el histórico sin que `pesoHijo` sobrepase `c`
    end if
end procedure

Algorithm 24 Histórico en Mutación

procedure (Historico_Mutacion)(*solucion*)
    `sustituido = False`
    if `!sustituido` **then** /*Establecer una lista de prioridades sobre qué mutar y con qué tipo de elementos (una vez que se haya sustituido, se establece `sustituido = True` y no entra en el resto de elementos de la lista)*/
        Sustituir peor 1 por mejor 0
        Sustituir 1 sin información por mejor 0
        Sustituir peor 1 por 0 sin información
        Sustituir 1 sin información por 0 sin información
        Sustituir mejor 1 por mejor 0
        Sustituir peor 1 por peor 0
        Sustituir mejor 1 por 0 sin información
        Sustituir 1 sin información por peor 0
        Sustituir mejor 1 por peor 0
    end if
    Aplicar Operador de Reparación con Histórico para rellenar más las soluciones
end procedure

Si este es el caso, entonces no se le dedicará suficiente tiempo a la fase de explotación, que es esencial para obtener soluciones de mejor calidad. Por lo tanto, utilizaremos un operador capaz de introducir cierta diversidad a la población a la vez que asegura cierta calidad. Esto es, utilizaremos el operador GRASP (*Greedy Randomized Adaptive Search Procedure*).

El GRASP vendrá presentado en Algoritmo 25. De forma simplificada, GRASP utiliza de base el algoritmo Greedy para obtener cierta cantidad de mejores elementos, eligirá aleatoriamente uno de ellos y lo devolverá. Lo que se entiende como "mejor elemento" para GRASP es lo mismo que para lo que se propuso durante la justificación del operador Greedy. Además, la cantidad de elementos a considerar dependerá del valor aportado por el mejor elemento; en concreto, solo se considerarán los elementos cuyo valor relativo (*valor_acumulado/peso*) varíe en un 10 % del valor acumulado del mejor elemento, es decir:

- Si se necesitan añadir elementos, nos quedaremos con los elementos que tengan un valor relativo mayor que $0.9 * valor\_relativo\_mayor$.
- Si se necesitan eliminar elementos, nos quedaremos con los elementos que tengan un valor relativo menor que $1.1 * valor\_relativo\_menor$.

Algorithm 25 Operador GRASP

```
1:  procedure (GTRASP)(solucion, min)
2:      if min then
3:          Almacenar en indices los elementos activados en solucion
4:      else
5:          Almacenar en indices los elementos desactivados en solucion
6:      end if
7:      Calcular el valor relativo de indices → valores
8:      encontrado = false
9:      if min then
10:         Ordenar indices según valores en orden ascendente
11:         for i ∈ 1..size(indices)&&!encontrado do
12:             if valores_i >1.1·valores_0 then
13:                 encontrado = true
14:                 Eliminar los elementos de indices desde i hasta size(indices)
15:             end if
16:         end for
17:     else
18:         Ordenar indices según valores en orden descendente
19:         for i ∈ 1..size(indices)&&!encontrado do
20:             if valores_i <0.9·valores_0 then
21:                 encontrado = true
22:                 Eliminar los elementos de indices desde i hasta size(indices)
23:             end if
24:         end for
25:     end if
26:     Elegir aleatoriamente algún elemento de indices
27: end procedure
```

7.3. Cruce Intensivo

Para aumentar la explotación de buenas soluciones se propone hacer una modificación en el cruce que realizamos. En vez de utilizar el cruce uniforme, que hemos establecido que a cada hijo se le asigna la mitad de los genes de cada uno de los padres (aunque sea de forma aleatoria), en este caso se le asignará un mayor porcentaje de genes del mejor de los padres.

Para ello, lógicamente tendremos que calcular cuál de los padres es la mejor solución (tiene mayor valor). En este caso, no nos sirve aleatorizar una sola lista de índices, ya que la separación no va a ser igualitaria. Por lo que tendremos que hacer las separaciones de genes para ambos hijos. Una vez hecho esto, se procede de igual forma que en los anteriores cruces, es decir, se le asigna a cada hijo los genes de los padres correspondientes y, posteriormente, se le aplica el operado de reparación a ambos hijos. Estoy viene representado en Algoritmo 26.

Por conveniencia, vamos a volver a utilizar la notación que implica que $\texttt{padre}_1$ tiene un valor más elevado que $\texttt{padre}_2$.

Algorithm 26 Cruce Intensivo

```
1:  procedure (Cruce Intensivo)(padre_1, padre_2, porcentaje)
2:      Desordenar los índices (indices_1, indices_2) que indican la posición de cada elemento
3:      nelem = n*porcentaje
4:      for i in 0..n do
5:          if i <nelem then
6:              hijo_1[indice_1[i]] = padre_1[indice_1[i]]
7:              hijo_2[indice_2[i]] = padre_1[indice_2[i]]
8:          else
9:              hijo_1[indice_1[i]] = padre_2[indice_1[i]]
10:             hijo_2[indice_2[i]] = padre_2[indice_2[i]]
11:         end if
12:     end for
13: end procedure
```

7.4. Operador NAM

El operador de Emparejamiento Variado Inverso (*Negative Assortative Mating*) o NAM [9] está orientado a generar diversidad. Esta es una alternativa a cómo se eligen los individuos de la población que se van a cruzar. La lógica de este operador es parecida a la prevención de incesto que se propone en el algoritmo CHC, pero sin ser tan restrictiva. Se necesita que los padres sean distintos para conseguir que los hijos aporten diversidad a la población si llegan a ser introducidos. Una optativa a esto es el operador NAM.

La implementación de este operador viene representado en el Algoritmo 27. Lo que se hace en este operador es elegir a una solución utilizando el Torneo Binario propio de los algoritmos genéticos. Para la segunda solución, primero, debemos obtener algunas soluciones de la población distintas a la primera obtenida para el cruce. Una vez hecho esto, elegimos como el segundo padre a la solución del segundo grupo más diferente al primer padre; esto es, se calculará la distancia de Hamming entre el primer padre y las soluciones del segundo grupo, y se elegirá como segundo padre aquella solución con mayor distancia de Hamming.

Algorithm 27 Operador NAM

1: **procedure** (Operador NAM)
2: Aplicar `Torneo Binario` para obtener al primer padre → $\texttt{padre}_1$
3: Elegir aleatoriamente sin repetición un subconjunto de la población distinta a $\texttt{padre}_1$ → `indices`.
4: Calcular la distancia de Hamming entre $\texttt{padre}_1$ y cada elemento de `indices`
5: Establecer como $\texttt{padre}_2$ la solución de `indices` con la mayor distancia de Hamming a $\texttt{padre}_1$
6: **end procedure**

7.5. Operador Reemplazo: *Crowding* Determinístico

Otra forma de intentar mantener cierta diversidad en la población es mediante la modificación del Operador de Reemplazo. En vez de siempre eliminar las peores soluciones para introducir otras mejores, se sustituirán individuos de la población similares a las nuevas soluciones si estas últimas son mejores. Esto da lugar que se permitirá mantener las peores soluciones en la población siempre que estén aportando diversidad a la población.

Este operador [20] viene representado en el Algoritmo 28. En definitiva, el objetivo de este operador será comprobar si las nuevas soluciones (hijas) son mejores que las soluciones de la población actual más cercanas a ellas. Si las soluciones hijas constituyen una mejora, entonces sustituirán a las soluciones más cercanas en la población.

Algorithm 28 Operador de Reemplazo por Cercanía

1: **procedure** (Reemplazo Cercanía)
2: **for** i in 0..`numHijos` **do**
3: Calcular distancia de Hamming de $\texttt{hijo}_i$ a todos los elementos de la población
4: Elegir la solución con la menor distancia de Hamming a $\texttt{hijo}_i$ → `solucion`
5: **if** $\texttt{valor}_{solucion}$ < $\texttt{valor}_{hijo_i}$ **then**
6: Sustituir `solucion` por $\texttt{hijo}_i$
7: **end if**
8: **end for**
9: **end procedure**

7.5.1. Versión intensiva

En esta versión, adicionalmente a modificar el operador de reemplazo de forma que las nuevas soluciones sustituirán a las más parecidas siempre que las mejoren, se compararán el resto de soluciones similares (tienen una distancia de Hamming menor que determinado umbral). La idea de esta versión es aumentar aún más la diversidad mediante la eliminación de soluciones peores parecidas a aquellas que ya tenemos.

Si se logra introducir la nueva solución en la población se comprobará si hay más soluciones cercanas. En caso negativo no se hará nada más al respecto. Sin embargo, en caso positivo, nos quedaremos únicamente con la mejor solución entre todas las consideradas (la nueva y las parecidas en la población), eliminando el resto de la población. Como el tamaño de la población debe ser constante, estas vacantes serán cubiertas por soluciones totalmente aleatorias que no sean parecidas

a la solución que se ha conseguido mantener en la población. Este funcionamiento viene representado en 29.

Algorithm 29 Operador de Reemplazo por Cercanía intensivo

```
1: procedure (Reemplazo Cercanía)(δ_H)
2:     for i in 0..numHijos do
3:         Almacenar los individuos de la población tal que su distancia de Hamming a hijo_i sea menor que δ_H → indCercanos
4:         Calculamos el elemento de indCercanos con mayor valor
5:         El resto de elementos de indCercanos se sustituye por soluciones generadas aleatoriamente
6:         while La solución generada aleatoriamente sea cercana a una existente do
7:             Sustituirla por otra solución generada aleatoriamente
8:         end while
9:     end for
10: end procedure
```

Capítulo 8: Parte Experimental (In progress)

Adicionalmente, como se ha comentado antes, cada algoritmo requiere de sus propios parámetros. Aunque explicaremos en más profundidad cada algoritmo en los siguientes capítulos, es necesario que establezcamos ahora cuáles son dichos parámetros y qué valores se han utilizado. Para ello, primero vamos a resumir en la siguiente tabla (??) qué parámetros usa cada algoritmo y después explicaremos qué es cada uno y qué valor tienen asignado.

Tabla 8.1: Parámetros utilizados por cada algoritmo

Algoritmo	EvaluacionMAX	Semilla	Nº Elementos	Probabilidad mutación
Random	**Sí**	**Sí**	No	No
AGEU	**Sí**	**Sí**	**Sí**	**Sí**
GACEP	**Sí**	**Sí**	**Sí**	**Sí**
CHC	**Sí**	**Sí**	**Sí**	No
GACEPCHC	**Sí**	**Sí**	**Sí**	**Sí**
GACEP3103	**Sí**	**Sí**	**Sí**	**Sí**

- `EvaluacionMAX`: Es el número de evaluaciones máximas para dicho algoritmo. Su valor será el de `NEVALUACIONESMAX`.

- `Semilla`: Como se ha indicado anteriormente, es necesario establecer una semilla de aleatoriedad en cada ejecución del algoritmo. Por lo tanto, el valor de este parámetro será el de la semilla generada por `INITSEED`.

- `Nº Elementos`: Número de soluciones que va a constituir una población. Es necesario tener una población pequeña con el fin de utilizar el menor número de evaluaciones posible, pero suficiente como para poder trabajar con cierto margen. Por lo tanto, estableceremos el tamaño de población a 10.

- `Probabilidad mutación`: En algunos algoritmos intentaremos modificar un poco alguna solución en cada iteración. Este valor establece el porcentaje de soluciones que la población que mutará. Genéricamente este valor suele ser del 0.1, por lo que también lo utilizaremos en nuestro problema. Correspondería que solo mutaría una solución de la población por cada generación.

Tabla 8.2: Solo Ranking

	GACEPwGRASP	GACEPwoGRASP
1	**1.309278**	1.690722
2	**1.154639**	1.845361
3	**1.185567**	1.814433
5	**1.154639**	1.845361
10	**1.051546**	1.948454
20	**1.092784**	1.907216
30	**1.123711**	1.876289
40	**1.061856**	1.938144
50	**1.082474**	1.917526
60	**1.072165**	1.927835
70	**1.103093**	1.896907
80	**1.103093**	1.896907
90	**1.103093**	1.896907
100	**1.103093**	1.896907

8.1. Algoritmos de Referencia experimentación

8.2. Resultados versión expensive

8.3. Incorporación del histórico

8.4. Uso de GRASP

8.5. Operador de Cruce Intensivo

8.6. Estudio de la diversidad

8.7. Incrementando la diversidad (con nuevo reemplazo)

8.8. Tablas de prueba

Tabla 8.3: Solo valores

	GACEPwGRASP	GACEPwoGRASP
F01	203870	**203917**
F02	**88229.5**	86893.8
F03	**202940**	202056
F04	**235610**	234528
F05	238439	**238931**
F06	**81325.7**	78698.4
F07	**10867.2**	10611.2
F08	**67669.3**	67007.1
F09	239982	**241977**
F10	**26642.1**	26107.5
F11	**20217.6**	20195.7
F12	**58173.8**	58108.8
F13	**3967.14**	3841.06
F14	**53183.3**	52534.5
F15	62310.9	**62661.5**
F16	**38963**	38120.5
F17	**15641.2**	15454.7
F18	**22181.6**	21423.6
F19	**37575.9**	36782.8
F20	94259.5	**94302.4**
F21	**89232.8**	88189.9
F22	**110642**	108332
F23	**37247.4**	36961.9
F24	110026	**110489**
F25	**60821.5**	57749
F26	**18066.4**	17239.7
F27	**55937.8**	55209.6
F28	**58848.7**	57800.7
F29	**73569.8**	72183.1
F30	**151250**	151073
F31	191323	**191744**
F32	**104160**	101933
F33	**66522.7**	65749.4
F34	**80481.7**	78826.7
F35	**31122.7**	30323.9
F36	**153157**	150580
F37	**120369**	118139
F38	**21816.6**	21744.5
F39	**112427**	111748
F40	**379570**	371906
F41	958934	**960695**
F42	**315558**	310368
F43	**32375.5**	31956
F44	106229	**107874**
F45	**813622**	804278
F46	**44144.1**	43930.7
F47	**726019**	711466
F48	**813949**	805644
F49	**653176**	639675
F50	**50543**	49255.1
F51	**211429**	206394
F52	**244518**	243532

Tabla 8.4: Valores+Ranking Final

	GACEPwGRASP	GACEPwoGRASP
F01	203870	**203917**
F02	**88229.5**	86893.8
F03	**202940**	202056
F04	**235610**	234528
F05	238439	**238931**
F06	**81325.7**	78698.4
F07	**10867.2**	10611.2
F08	**67669.3**	67007.1
F09	239982	**241977**
F10	**26642.1**	26107.5
F11	**20217.6**	20195.7
F12	**58173.8**	58108.8
F13	**3967.14**	3841.06
F14	**53183.3**	52534.5
F15	62310.9	**62661.5**
F16	**38963**	38120.5
F17	**15641.2**	15454.7
F18	**22181.6**	21423.6
F19	**37575.9**	36782.8
F20	94259.5	**94302.4**
F21	**89232.8**	88189.9
F22	**110642**	108332
F23	**37247.4**	36961.9
F24	110026	**110489**
F25	**60821.5**	57749
F26	**18066.4**	17239.7
F27	**55937.8**	55209.6
F28	**58848.7**	57800.7
F29	**73569.8**	72183.1
F30	**151250**	151073
F31	191323	**191744**
F32	**104160**	101933
F33	**66522.7**	65749.4
F34	**80481.7**	78826.7
F35	**31122.7**	30323.9
F36	**153157**	150580
F37	**120369**	118139
F38	**21816.6**	21744.5
F39	**112427**	111748
F40	**379570**	371906
F41	958934	**960695**
F42	**315558**	310368
F43	**32375.5**	31956
F44	106229	**107874**
F45	**813622**	804278
F46	**44144.1**	43930.7
F47	**726019**	711466
F48	**813949**	805644
F49	**653176**	639675
F50	**50543**	49255.1
F51	**211429**	206394
F52	**244518**	243532

Tabla 8.5: Valores+Ranking Final + nProblemas

		GACEPwGRASP	GACEPwoGRASP
n = 100	F01	203870	**203917**
	F02	**88229.5**	86893.8
	F03	**202940**	202056
	F04	**235610**	234528
	F05	238439	**238931**
	F06	**81325.7**	78698.4
	F07	**10867.2**	10611.2
	F08	**67669.3**	67007.1
	F09	239982	**241977**
	F10	**26642.1**	26107.5
	F11	**20217.6**	20195.7
	F12	**58173.8**	58108.8
	F13	**3967.14**	3841.06
	F14	**53183.3**	52534.5
	F15	62310.9	**62661.5**
	F16	**38963**	38120.5
	F17	**15641.2**	15454.7
	F18	**22181.6**	21423.6
	F19	**37575.9**	36782.8
	F20	94259.5	**94302.4**
	F21	**89232.8**	88189.9
	F22	**110642**	108332
	F23	**37247.4**	36961.9
	F24	110026	**110489**
	F25	**60821.5**	57749
	F26	**18066.4**	17239.7
	F27	**55937.8**	55209.6
	F28	**58848.7**	57800.7
	F29	**73569.8**	72183.1
	F30	**151250**	151073
	F31	191323	**191744**
	F32	**104160**	101933
	F33	**66522.7**	65749.4
	F34	**80481.7**	78826.7
	F35	**31122.7**	30323.9
	F36	**153157**	150580
	F37	**120369**	118139
	F38	**21816.6**	21744.5
	F39	**112427**	111748
	F40	**379570**	371906
	F41	958934	**960695**
	F42	**315558**	310368
	F43	**32375.5**	31956
	F44	106229	**107874**
	F45	**813622**	804278
	F46	**44144.1**	43930.7
	F47	**726019**	711466
	F48	**813949**	805644
	F49	**653176**	639675
	F50	**50543**	49255.1
	F51	**211429**	206394
	F52	**244518**	243532
	F53	**227361**	226204
	F54	**193634**	189500

Tabla 8.6: Ranking Mejor-Peor

	AGEU	GACEP3103_5_4-2	GACEPCHC	GACEPMutacion	GACEPTotal	GACEPc50	GACEPorGRASP
1	3.463918	**6.773196**	2.896907	3.463918	3.463918	3.659794	4.278351
2	3.391753	**6.742268**	2.845361	3.391753	3.391753	3.649485	4.587629
3	3.360825	**6.680412**	2.56701	3.360825	3.360825	3.896907	4.773196
5	3.505155	**6.340206**	1.876289	3.505155	3.505155	3.917526	5.350515
10	3.690722	5.391753	1.329897	3.690722	3.690722	3.886598	**6.319588**
20	3.561856	4.489691	4.71134	3.231959	3.412371	3.386598	**5.206186**
30	3.530928	4.561856	4.082474	2.798969	3.71134	3.5	**5.814433**
40	2.427835	5.324742	3.247423	2.396907	4.28866	4.06701	**6.247423**
50	2.376289	5.572165	2.917526	2.469072	4.226804	4.314433	**6.123711**
60	2.340206	**5.963918**	2.896907	2.57732	4.154639	4.108247	5.958763
70	2.43299	**6.489691**	2.917526	2.515464	3.979381	3.984536	5.680412
80	2.365979	**6.582474**	3.072165	2.510309	3.979381	3.891753	5.597938
90	2.278351	**6.840206**	3.164948	2.56701	3.948454	3.809278	5.391753
100	2.350515	**6.840206**	3.134021	2.546392	3.938144	3.819588	5.371134

Tabla 8.7: Wilcoxon

	AGEU	GACEP3103	GACEPCHC	Mutacion	GACEPTotal	c50	orGRASP	Error Acumulado(%)
AGEU	1	3.54E-17	0.655466	0.009625	6.79E-07	1.21E-06	0	26.49081
GACEP3103	**3.54E-17**	1	**1.64E-17**	**3.77E-17**	**3.77E-17**	**2.72E-17**	**2.78E-16**	26.49081
GACEPCHC	0.655466	1.64E-17	1	0.735187	0.001855	0.003562	9E-10	26.49081
Mutacion	**0.009625**	3.77E-17	0.735187	1	1.53E-06	3.46E-06	0	26.49081
GACEPTotal	**6.79E-07**	3.77E-17	**0.001855**	**1.53E-06**	1	0.233648	0	26.49081
GACEPc50	**1.21E-06**	2.72E-17	**0.003562**	**3.46E-06**	0.233648	1	0	26.49081
GACEPorGRASP	**0**	2.78E-16	**9E-10**	**0**	**0**	**0**	1	26.49081

Tabla 8.8: Comparaciones múltiples

Algoritmos vs GACEP3103	**Original**	**Holm**	**Hommel**	**Hochberg**
AGEU	3.54446E-17	**1.42E-16**	**7.09E-17**	**7.54E-17**
GACEPCHC	1.63701E-17	**9.82E-17**	**5.03E-17**	**7.54E-17**
GACEPMutacion	3.7691E-17	**1.42E-16**	**7.54E-17**	**7.54E-17**
GACEPTotal	3.7691E-17	**1.42E-16**	**7.54E-17**	**7.54E-17**
GACEPc50	2.7223E-17	**1.36E-16**	**5.65E-17**	**7.54E-17**
GACEPorGRASP	2.78282E-16	**2.78E-16**	**2.78E-16**	**2.78E-16**

Capítulo 9: Conclusiones

Bibliografía

[1] T. Back y H.-P. Schwefel. "Evolutionary Computation: An Overview". En: *Proceedings of IEEE International Conference on Evolutionary Computation*. 105 citations (Crossref) [2023-05-31]. Mayo de 1996, págs. 20-29. DOI: 10.1109/ICEC.1996.542329.

[2] Quang-Thanh Bui. "Metaheuristic Algorithms in Optimizing Neural Network: A Comparative Study for Forest Fire Susceptibility Mapping in Dak Nong, Vietnam". En: *Geomatics, Natural Hazards and Risk* 10.1 (ene. de 2019), págs. 136-150. ISSN: 1947-5705, 1947-5713. DOI: 10.1080/19475705.2018.1509902. (Visitado 18-05-2023).

[3] Stuti Chaturvedi y Vishnu P. Sharma. "A Modified Genetic Algorithm Based on Improved Crossover Array Approach". En: *Advances in Data and Information Sciences*. Ed. por Mohan L. Kolhe y col. Lecture Notes in Networks and Systems. Singapore: Springer, 2019, págs. 117-127. ISBN: 9789811302770. DOI: 10.1007/978-981-13-0277-0_10.

[4] Runwei Cheng y Mitsuo Gen. "Crossover on Intensive Search and Traveling Salesman Problem". En: *Computers & Industrial Engineering*. 16th Annual Conference on Computers and Industrial Engineering 27.1 (sep. de 1994), págs. 485-488. ISSN: 0360-8352. DOI: 10.1016/0360-8352(94)90340-9. (Visitado 04-06-2023).

[5] Matheus Bernardelli de Moraes y Guilherme Palermo Coelho. "A Diversity Preservation Method for Expensive Multi-Objective Combinatorial Optimization Problems Using Novel-First Tabu Search and MOEA/D". En: *Expert Systems with Applications* 202 (sep. de 2022), pág. 117251. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2022.117251. (Visitado 02-06-2023).

[6] Joaquín Derrac y col. "A Practical Tutorial on the Use of Nonparametric Statistical Tests as a Methodology for Comparing Evolutionary and Swarm Intelligence Algorithms". En: *Swarm and Evolutionary Computation* 1.1 (mar. de 2011). Test Estadisticos No paramétricos
Mirar la sección 3 y 4 para rellenar el capítulo de Contexto Matemático respecto a los test estadísticos no paramétricos, págs. 3-18. ISSN: 22106502. DOI: 10.1016/j.swevo.2011.02.002. (Visitado 20-05-2023).

[7] A.E. Eiben, R. Hinterding y Z. Michalewicz. "Parameter Control in Evolutionary Algorithms". En: *IEEE Transactions on Evolutionary Computation* 3.2 (jul. de 1999), págs. 124-141. ISSN: 1941-0026. DOI: 10.1109/4235.771166.

[8] Larry J. Eshelman. "The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination". En: *Foundations of Genetic Algorithms*. Vol. 1. Elsevier, 1991, págs. 265-283. ISBN: 978-0-08-050684-5. DOI: 10.1016/B978-0-08-050684-5.50020-3. (Visitado 03-06-2023).

[9] Carlos Fernandes y col. "A Study on Non-random Mating and Varying Population Size in Genetic Algorithms Using a Royal Road Function". En: (abr. de 2001).

[10] C. E. Ferreira y col. "Formulations and Valid Inequalities for the Node Capacitated Graph Partitioning Problem". En: *Mathematical Programming* 74.3 (sep. de 1996), págs. 247-266. ISSN: 1436-4646. DOI: 10.1007/BF02592198. (Visitado 04-06-2023).

[11] G. Gallo, P. L. Hammer y B. Simeone. "Quadratic Knapsack Problems". En: *Combinatorial Optimization*. Ed. por M. W. Padberg. Mathematical Programming Studies. Berlin, Heidelberg: Springer, 1980, págs. 132-149. ISBN: 978-3-642-00802-3. DOI: 10.1007/BFb0120892. (Visitado 04-06-2023).

[12] David E. Goldberg y Kalyanmoy Deb. "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms". En: *Foundations of Genetic Algorithms*. Vol. 1. Elsevier, 1991, págs. 69-93. ISBN: 978-0-08-050684-5. DOI: 10.1016/B978-0-08-050684-5.50008-2. (Visitado 17-06-2023).

[13] John J. Grefenstette. "Optimization of Control Parameters for Genetic Algorithms". En: *IEEE Transactions on Systems, Man, and Cybernetics* 16.1 (ene. de 1986), págs. 122-128. ISSN: 2168-2909. DOI: 10.1109/TSMC.1986.289288.

[14] S. Islam. *MATHEMATICAL PROGRAMMING*. Vol. 1. Ene. de 2020, pág. 700. ISBN: 978-620-0-46432-3.

[15] Y. Jin. "A Comprehensive Survey of Fitness Approximation in Evolutionary Computation". En: *Soft Computing* 9.1 (ene. de 2005), págs. 3-12. ISSN: 1433-7479. DOI: 10.1007/s00500-003-0328-5. (Visitado 04-06-2023).

[16] Antonio LaTorre y col. "A Prescription of Methodological Guidelines for Comparing Bio-Inspired Optimization Algorithms". En: *Swarm and Evolutionary Computation* 67 (dic. de 2021). 43 citations (Crossref) [2023-05-31], pág. 100973. ISSN: 2210-6502. DOI: 10.1016/j.swevo.2021.100973. (Visitado 31-05-2023).

[17] Jian-Yu Li, Zhi-Hui Zhan y Jun Zhang. "Evolutionary Computation for Expensive Optimization: A Survey". En: *Machine Intelligence Research* 19.1 (feb. de 2022). Definición EOP ¿Ejemplos?, págs. 3-23. ISSN: 2731-538X, 2731-5398. DOI: 10.1007/s11633-022-1317-4. (Visitado 05-10-2022).

[18] J. Lis. "Parallel Genetic Algorithm with the Dynamic Control Parameter". En: *Proceedings of IEEE International Conference on Evolutionary Computation*. Mayo de 1996, págs. 324-329. DOI: 10.1109/ICEC.1996.542383.

[19] Fernando Lobo y David Goldberg. "An Overview of the Parameter-Less Genetic Algorithm". En: (ene. de 2008).

[20] Samir W. Mahfoud. "Crowding and Preselection Revisited". En: *Parallel Problem Solving from Nature*. [TLDR] This paper considers the related algorithms, crowding and preselection, as potential multimodal function optimizers and examines the ability of the two algorithms to preserve diversity, especially multi-modal diversity. 1992. (Visitado 17-06-2023).

[21] Aritz D. Martinez y col. "Lights and Shadows in Evolutionary Deep Learning: Taxonomy, Critical Methodological Analysis, Cases of Study, Learned Lessons, Recommendations and Challenges". En: *Information Fusion* 67 (mar. de 2021). 12 citations (Crossref) [2023-05-31] Neuroevolución, págs. 161-194. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2020.10.014. (Visitado 31-05-2023).

[22] Z. Michalewicz, Thomas Baeck y D.B. Fogel. *Handbook of Evolutionary Computation*. Boca Raton: CRC Press, ene. de 1997. ISBN: 978-0-367-80248-6. DOI: 10.1201/9780367802486.

[23] Eric Pellerin, Luc Pigeon y Sylvain Delisle. "Self-Adaptive Parameters in Genetic Algorithms". En: *Defense and Security*. Ed. por Belur V. Dasarathy. Orlando, FL, abr. de 2004, pág. 53. DOI: 10.1117/12.542156. (Visitado 04-06-2023).

[24] Tuan Pham. *Competitive Evolution: A Natural Approach to Operator Selection*. Vol. 956. Ene. de 1994, pág. 60. ISBN: 978-3-540-60154-8. DOI: 10.1007/3-540-60154-6_47.

[25] David Pisinger. "The Quadratic Knapsack Problem—a Survey". En: *Discrete Applied Mathematics* 155.5 (mar. de 2007), págs. 623-648. ISSN: 0166218X. DOI: 10.1016/j.dam.2006.08.007. (Visitado 26-10-2022).

[26] Javier Poyatos y col. "EvoPruneDeepTL: An Evolutionary Pruning Model for Transfer Learning Based Deep Neural Networks". En: *Neural Networks* 158 (ene. de 2023). 1 citations (Crossref) [2023-05-31]
Ejemplo concreto de redes neuronales en las que se vieron obligaron a bajar el número, págs. 59-82. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2022.10.011. (Visitado 31-05-2023).

[27] Zhang Qiongbing y Ding Lixin. "A New Crossover Mechanism for Genetic Algorithms with Variable-Length Chromosomes for Path Optimization Problems". En: *Expert Systems with Applications* 60 (oct. de 2016), págs. 183-189. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2016.04.005. (Visitado 04-06-2023).

[28] Colin Reeves. "Genetic Algorithms". En: *Handbook of Metaheuristics*. Vol. 146. Sep. de 2010, págs. 109-139. DOI: 10.1007/978-1-4419-1665-5_5.

[29] Colin R. Reeves. "Feature Article—Genetic Algorithms for the Operations Researcher". En: *INFORMS Journal on Computing* 9.3 (ago. de 1997), págs. 231-250. ISSN: 1091-9856. DOI: 10.1287/ijoc.9.3.231. (Visitado 31-05-2023).

[30] J. M. W. Rhys. "A Selection Problem of Shared Fixed Costs and Network Flows". En: *Management Science* 17.3 (nov. de 1970), págs. 200-207. ISSN: 0025-1909, 1526-5501. DOI: 10.1287/mnsc.17.3.200. (Visitado 04-06-2023).

[31] Songqing Shan y G. Gary Wang. "Survey of Modeling and Optimization Strategies to Solve High-Dimensional Design Problems with Computationally-Expensive Black-Box Functions". En: *Structural and Multidisciplinary Optimization* 41.2 (mar. de 2010), págs. 219-241. ISSN: 1615-1488. DOI: 10.1007/s00158-009-0420-2. (Visitado 04-06-2023).

[32] Anabela Simões y Ernesto Costa. "CHC-Based Algorithms for the Dynamic Traveling Salesman Problem". En: *Applications of Evolutionary Computation*. Ed. por Cecilia Di Chio y col. Vol. 6624. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, págs. 354-363. ISBN: 978-3-642-20524-8 978-3-642-20525-5. DOI: 10.1007/978-3-642-20525-5_36. (Visitado 13-05-2023).

[33] Jim Smith y T. C. Fogarty. "Operator and Parameter Adaptation in Genetic Algorithms". En: *Soft Computing* 1.2 (ene. de 1997). ISSN: 1432-7643. DOI: 10.1007/s005000050009. (Visitado 31-05-2023).

[34] Yoel Tenne y col., eds. *Computational Intelligence in Expensive Optimization Problems*. Vol. 2. Adaptation Learning and Optimization. Berlin, Heidelberg: Springer, 2010. ISBN: 978-3-642-10700-9 978-3-642-10701-6. DOI: 10.1007/978-3-642-10701-6. (Visitado 04-06-2023).

[35] Z. Tu e Y. Lu. "A Robust Stochastic Genetic Algorithm (StGA) for Global Numerical Optimization". En: *IEEE Transactions on Evolutionary Computation* 8.5 (oct. de 2004), págs. 456-470. ISSN: 1089-778X. DOI: 10.1109/TEVC.2004.831258. (Visitado 31-05-2023).

[36] C. Witzgall. "Mathematical Methods of Site Selection for Electronic Message Systems (EMS)". En: *NASA STI/Recon Technical Report N* 76 (jun. de 1975), pág. 18321. (Visitado 02-06-2023).