



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



ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍAS INFORMÁTICA
Y DE TELECOMUNICACIÓN



Facultad de Ciencias

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 para el caso continuo de este tipo de problemas; 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, exploitation, diversity

Abstract

There are very interesting problems which their computational cost is excessively high (these kind of problems will be referred as expensive problems), such as the neural network optimization problems, therefore it is compelling to develop algorithms that are able to obtain competitive solutions in few evaluations. Furthermore, specific algorithms for the continuous case of these problems can be found in the literature; but that is not the case for the combinatorial 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, 18 de mayo 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 18 de mayo de 2023.

Los directores:

Daniel Molina Cabrera

Francisco Herrera Triguero

Agradecimientos

Poner aquí agradecimientos...

Índice general

1. Introducción	1
2. Estimación y Presupuesto	3
2.1. Planificación	3
2.1.1. Planificación Base	3
2.1.2. Planificación Final	4
2.2. Presupuesto	5
3. Repaso bibliográfico	7
4. Contexto matemático	9
4.1. Definición del problema	9
4.2. Condición de Optimalidad	11
4.3. Rendimiento de los algoritmos	13
4.3.1. Convergencia y Orden de Convergencia	13
4.3.2. Comportamiento numérico	14
4.4. Programación no lineal sin restricciones	15
4.4.1. Algoritmos de Optimización sin restricciones	15
4.4.2. Algoritmos de Búsqueda Lineal	16
4.4.3. Métodos de Gradiente	17
4.4.4. Métodos de Gradiente Conjugado	18
4.4.5. Métodos de Newton	19
5. Problema y Diseño Experimental	29
5.1. Descripción del problema	29
5.1.1. Quadratic Knapsack Problem	30
5.1.2. Datos del problema	30
5.2. Diseño Experimental	31
5.2.1. Criterio de Parada	31

5.2.2. Parámetros	33
6. Algoritmos de Referencia	35
6.1. Algoritmo Genético Estacionario Uniforme	35
6.1.1. Pseudocódigo	36
6.1.2. Componentes	37
6.2. CHC	41
6.2.1. Pseudocódigo	42
6.2.2. Componentes	42
7. Componentes de la propuesta	45
8. Parte Experimental (In progress)	47
8.1. Algoritmos de Referencia experimentación	48
8.2. Resultados versión expensive	48
8.3. Incorporación del histórico	48
8.4. Uso de GRASP	48
8.5. Operador de Cruce Intensivo	48
8.6. Estudio de la diversidad	48
8.7. Incrementando la diversidad (con nuevo reemplazo)	48
9. Conclusiones	49

Índice de figuras

6.1. Cruce en un punto	38
6.2. Cruce en dos puntos	39
6.3. Cruce Uniforme	39
6.4. Cruce HUX	43

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. Los algoritmos aproximados se pueden dividir en heurísticas y metaheurísticas, estas últimas son en las que vamos a estar más interesados. Las metaheurísticas suelen ser procedimientos iterativos que guían una heurística subordinada de búsqueda, combinando de forma inteligente distintos conceptos para explorar y explotar adecuadamente el espacio de búsqueda. Así, las metaheurísticas son una familia de algoritmos aproximados más generales que las heurísticas (que son dependientes del problema a tratar) y aplicables a una gran variedad de problemas de optimización, tanto continuos como combinatorios.

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 [Metaheuristic algorithms in optimizing neural network: a comparative study for forest fire susceptibility mapping in Dak Nong, Vietnam]; 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 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, como ya hemos comentado anteriormente, podría ser el uso de este tipo de algoritmos para optimizar redes neuronales. 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 evaluaciones; para ello, compararemos los resultados obtenidos del algoritmo propuesto con los resultados pertenecientes a otros tipos de algoritmos conocidos pero no especializados en este tipo de problemas.

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:

Cuadro 2.1: Planificación Base

Resumen	Tareas	Duración
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

Cuadro 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.

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 se 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:

$$500h \cdot 25 \frac{\text{€}}{h} + 1300\text{€} + 50h \cdot 0.443 \frac{\text{€}}{h} = 13822.15\text{€}$$

En resumen, el **presupuesto de este proyecto queda fijado en 13822.15€.**

Capítulo 3: Repaso bibliográfico

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ donde x pertenece a un conjunto factible de soluciones $\mathcal{F} \in \mathbb{R}^n$. Consideramos el siguiente problema:

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

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:

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

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:

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

Entonces, el problema 4.1 se convierte en:

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

donde $g : \mathbb{R}^n \rightarrow \mathbb{R}^p$ y $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$. 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, $f, g_i, i = 1, \dots, p, h_j, j = 1, \dots, m$ 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 $x^* \in \mathcal{F}$ 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 $f : [a, b] \rightarrow \mathbb{R}$ 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 $\mathcal{L}^\alpha = \{x \in \mathbb{R}^n : f(x) \leq \alpha\}$ 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 $f(x^*) \leq f(x)$, $\forall x \in \mathcal{F} \cap \mathcal{B}_{x^*}$. Además, es una **solución local estricta** si $f(x^*) < f(x)$, $\forall x \in \mathcal{F} \cap \mathcal{B}_{x^*}, x \neq x^*$.

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 $h : \mathbb{R}^n \rightarrow \mathbb{R}$, denotamos el vector **gradiente** por $\nabla h(x) = \left(\frac{\partial f(x)}{\partial x_1}, \dots, \frac{\partial f(x)}{\partial x_n} \right)$ y la matriz **Hessiana** se denota por

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

- Dada una función vector $w : \mathbb{R}^n \rightarrow \mathbb{R}^q$, denotamos la matriz de $n \times q$ cuyas columnas son $\nabla w_j(x)$, $j = 1, \dots, q$ 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 , $i = 1, \dots, q$ y sea A una matriz cuyas columnas sean a_j , $j = 1, \dots, q$. Sea $\mathcal{K} \subset \{1, \dots, q\}$ 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*

$$\nabla f(x^*) = 0 \quad (4.5)$$

Además, si f es continuamente 2-diferenciable, entonces

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

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} :*

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

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:

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

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:*

$$\nabla_x L(x^*, \lambda^*, \mu^*) = 0 \quad (4.9)$$

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*

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

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 $x^k, k = 0, 1, \dots$ 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 $x^k, k = 0, 1, \dots$ 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

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

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

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

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

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

donde $x \in \mathbb{R}^n$ es el vector de variables de decisión y $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 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

$$x^{k+1} = x^k + \alpha^k d^k \quad (4.12)$$

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.

$$\lim_{k \rightarrow \infty} \frac{\nabla f(x^k)' d^k}{\|d^k\|} = 0 \quad (4.13)$$

4. $\forall k$ con $d^k \neq 0$, tenemos $\frac{|\nabla f(x^k)' d^k|}{\|d^k\|} \geq c \|\nabla f(x^k)\|$ 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.

$$\lim_{k \rightarrow \infty} \|\nabla f(x^k)\| = 0 \quad (4.14)$$

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

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

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:

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

Algorithm 1 Algoritmo de Búsqueda Lineal de Armijo

```

1: procedure (Busqueda_Lineal)( $\delta \in (0, 1), \gamma \in (0, 1/2), c \in (0, 1)$ )
2:   Elegir  $\nabla^k$  tal que

$$\nabla^k \geq c \frac{|\nabla f(x^k)' d^k|}{\|d^k\|^2}$$

3:    $\alpha = \nabla^k$ 
4:    $N \leftarrow n$ 
5:   if  $f(x^k + \alpha d^k) \leq f(x^k) + \gamma \alpha \nabla f(x^k)' d^k$  then
6:      $\alpha^k = \alpha$  y parar
7:   else
8:     Establecer  $\alpha = \delta \alpha$  y volver al paso anterior.
9:   end if
10: 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

```

1: procedure (Gradiente)
2:   Establecer  $x^0 \in \mathbb{R}^n$  y  $k = 0$ 
3:   while  $\nabla f(x^k) \neq 0$  do
4:     Establecer  $d^k = -\nabla f(x^k)$  y encuentra un tamaño de paso  $\alpha^k$  usando 1
5:     Establecer  $x^{k+1} = x^k - \alpha^k \nabla f(x^k)$  y  $k = k + 1$ .
6:   end while
7: 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. f se dice **Lipschitziana** si existe $K > 0$ tal que

$$\|f(x) - f(y)\| \leq K \|x - y\|, \forall x, y \in \mathbb{R}^n$$

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

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

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, \dots, 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  $d^0, \dots, d^{n-1}$ )
2:   Establecer  $x^0 \in \mathbb{R}^n$  y  $k = 0$ 
3:   while  $\nabla f(x^k) \neq 0$  do
4:     Establecer  $\alpha^k = \frac{\nabla f(x^k)'d^k}{(d^k)'Qd^k}$ 
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:

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

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

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

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

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

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  $\nabla f(x^k) \neq 0$  do
4:     Calcular  $\beta^{k-1}$  usando 4.18 o 4.19 y establecer la dirección usando

```

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

```

5:     Encontrar  $\alpha^k$  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:

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

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

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

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

$$\nabla^2 f(x^k) d_N = -\nabla f(x^k) \quad (4.22)$$

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

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

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

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

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

```

1: procedure (ABLNewton) ( $c_1 > 0, c_2 > 0, p \geq 2, q \geq 3$ )
2:   Establecer  $x^0 \in \mathbb{R}^n$  y  $k = 0$ 
3:   while  $\nabla f(x^k) \neq 0$  do
4:     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
5:       Establecer la dirección  $d^k = d_N^k$ 
6:     else
7:        $d^k = -\nabla f(x^k)$ 
8:     end if
9:     Encontrar  $\alpha^k$  usando 1
10:    Establecer  $x^{k+1} = x^k + \alpha^k d^k$  y, después,  $k = k + 1$ 
11:  end while
12: 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,

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

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

1: **procedure** (NewtonNM) ($c_1 > 0, c_2 > 0, p \geq 2, q \geq 3$ y M entero)

2: Establecer $x^0 \in \mathbb{R}^n$ y $k = 0$

3: **while** $\nabla f(x^k) \neq 0$ **do**

4: **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

5: Establecer la dirección $d^k = d_N^k$

6: **else**

7: $d^k = -\nabla f(x^k)$

8: **end if**

9: 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 $J = m(k)$

10: Establecer $x^{k+1} = x^k + \alpha^k d^k$ y, después, $k = k + 1$

11: Establecer $m(k) = \min\{m(k-1) + 1, M\}$

12: **end while**

13: **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

```

1: procedure (NewtonRegionConfianza) ( $0 < \gamma_1 \leq \gamma_2 < 1, 0\delta_1 < 1 \leq \delta_2$ )
2:   Establecer  $x^0 \in \mathbb{R}^n$ ,  $k = 0$  y  $a^0 = 0$ .
3:   Encontrar  $s^k = \arg \min_{\|s\| \leq a^k} q^k(s)$ 
4:   if  $f(x^k) == q^k(s^k)$  then
5:     Parar
6:   else
7:     Calcular la proporción

```

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

```

8:     if  $\rho^k \geq \gamma_1$  then
9:       Actualizar  $x^{k+1} = x^k + s^k$ 
10:    else
11:      Actualizar  $x^{k+1} = x^k$ 
12:    end if
13:    Actualizar el radio  $a^k$ 

```

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

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

```

14:   end if
15: 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 $\{x^k\}$ converge a una solución y si

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

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 $\epsilon^k > 0$.
- 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 \tilde{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

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

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

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

- 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

$$B^k d = -\nabla f(x^k) \quad (4.26)$$

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

$$d^k = -H^k \nabla f(x^k) \quad (4.27)$$

Ambas matrices se modifican en cada iteración como una corrección 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:

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

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

$$\nabla^2 f(x^k) \delta^k = \gamma^k \quad (4.28)$$

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

$$(B^k + \Delta B^k) \delta^k = \gamma^k \quad (4.29)$$

La regla de actualización de H^k está dada por

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

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** $\nabla f(x^k) \neq 0$ **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

$$\begin{aligned} x^{k+1} &= x^k + \alpha^k d^k \\ H^{k+1} &= H^k + \Delta H^k \end{aligned} \quad (4.31)$$
 - 7: Establecer $k = k + 1$
 - 8: **end while**
 - 9: **end procedure**
-

Distinguiamos 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

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

entonces la secuencia de puntos generada por el algoritmo satisface

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

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 $x^* \Leftrightarrow$

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

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 $\mathcal{D}^k = \{d^1, \dots, d^r\}$ 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 iteración, 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 DFLSA 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

```

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

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

```

Algorithm 11 Algoritmo de Búsqueda Lineal sin Derivadas

```

1: procedure (DFLSA) ( $\mathcal{D}^{\parallel}$  satisfaciendo la suposición previa,  $\gamma > 0, \theta \in (0, 1)$ )
2:   Establecer  $x_0 \in \mathbb{R}^n, \Delta^0 > 0$  y  $k = 0$ 
3:   Comprobar la convergencia
4:   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
5:     Establecer  $x^{k+1} = x^k, \Delta k + 1 = \alpha^k, k = k + 1$  e ir al paso 3.
6:   else
7:     Establecer  $x^{k+1} = x^k, \Delta k + 1 = \theta \Delta^k, k = k + 1$ 
8:   end if
9: end procedure

```

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

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

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:

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

donde $\bar{f} = \frac{1}{n+1} \sum_{i=1}^{n+1} f(x^i)$ y $\{x^i : i = 1, \dots, n+1\}$ 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 .

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. 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 $N = \{1, \dots, n\}$ 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:

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

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 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 , 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.

Cuadro 5.1: Datos del Problema

Número de variables	Densidad	Número de archivos
n = 100	25 %	10
	50 %	10
	75 %	10
	100 %	10
n = 200	25 %	10
	50 %	10
	75 %	10
	100 %	10
n = 300	25 %	10
	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 o evaluaciones. Denotaremos como iteraciones al número de veces que se evalúa 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, el número de iteraciones será el número de generaciones con las que trabajaremos.

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 esta reducción del 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.

Haciendo cálculos, 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 evaluaciones 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. De forma más o menos arbitraria, se ha establecido 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.

En definitiva, podemos resumir los parámetros, para lo que se usan y sus valores en la Tabla 5.2.

Cuadro 5.2: Resumen de parámetros utilizados

Parámetros	Resumen	Valor
NEVALUACIONESMAX	Criterio de parada: Número de Evaluaciones 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 Evaluaciones Máximas dicho algoritmo	NEVALUACIONESMAX
Semilla	Semilla de aleatoriedad para determinada ejecución del algoritmo	Valor generado aleatoriamente por INITSEED

Capítulo 6: Algoritmos de Referencia

En este capítulo se presentan los dos algoritmos que se utilizarán 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).

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 12)

En el caso de la versión AGEU, su pseudocódigo viene representado en 13. 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 12 Algoritmo Genético

```

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

```

hijos se obtienen de forma que para algunos elementos **hijo** _{i} los obtiene de **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.

6.1.1. Pseudocódigo

A efectos prácticos, este pseudocódigo (Algoritmo 13) se diferenciará del anterior (12) 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 13 Algoritmo Genético Estacionario Uniforme

```

1: procedure (AG)( $EMax > 0, nelem > 0, pmut \in [0, 1]$ )
2:   Generar una población inicial aleatoria
3:   Calcular la función fitness de cada individuo
4:   generacion = 0
5:   while generacion < EMax do
6:     Seleccionar aleatoriamente 4 soluciones de la población sin repetición 2 a 2
7:     Aplicar torneo de 2 en 2 soluciones del conjunto anterior y almacenar solo la mejor  $\rightarrow$ 
       padres
8:     Generar 2 hijos cruzando padres1 y padres2
9:     Aplicar el Operador de Reparación sobre ambos hijos
10:    Calcular la función fitness de cada hijo
11:    Almacenar dichos hijos  $\rightarrow$  hijos
12:    Calcular el número de soluciones a mutar  $\rightarrow nmut = pmut * nelem$ 
13:    Mutar nmut soluciones distintas
14:    Calcular la función fitness de las nuevas soluciones
15:    Aplicar el Operador de Selección Estacionario sobre la población actual e hijos
16:    generacion = generacion+1
17:  end while
18: 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 pueda seguir considerándose una solución.
- El resultado del cruce no constituya el espacio de soluciones.

Más adelante en este capítulo se explicarán cómo son los cruces y se entenderá por qué es posible que el resultado de un cruce no sea una solución. 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 es obvio el por qué necesitamos aplicarlo. Si el “hijo” es solución 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 14:

- Si no es solución (su peso supera al peso máximo): Se deberán eliminar elementos del “hijo” hasta que constituya una solución. 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 (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 14 Operador de Reparación

```

1: procedure (Operador Reparación)(hijo)
2:   Calcular el peso total de hijo  $\rightarrow$  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.

Cruce Uniforme

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 *padre*₁ a *hijo*₁ y de *padre*₂ a *hijo*₂ hasta cierto cromosoma elegido con anterioridad. A partir de dicho cromosoma, cambiaremos la asignación de forma que *hijo*₁ hereda de *padre*₂ e *hijo*₂ hereda de *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:

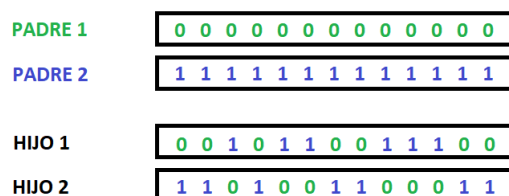


Figura 6.3: Cruce Uniforme

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. 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 válida.

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 15.

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

Algorithm 15 Cruce Uniforme

```

1: procedure (Cruce Uniforme)(padre1, padre2)
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:       hijo1[indice[i]] = padre1[indice[i]]
6:       hijo2[indice[i]] = padre2[indice[i]]
7:     else
8:       hijo1[indice[i]] = padre2[indice[i]]
9:       hijo2[indice[i]] = padre1[indice[i]]
10:    end if
11:  end for
12: end procedure

```

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 16.

Algorithm 16 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

```

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 la 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 17. Usaremos los índices de forma que indique un orden de valores *fitness*, por ejemplo, dados **padre₁** y **padre₂**, se cumplirá que $fitness(\text{padre}_1) > fitness(\text{padre}_2)$. Además, por conveniencia en la notación, se entenderá que $\text{solucion}_i > \text{solucion}_j$ significa que el valor *fitness* de solucion_i es mayor al de solucion_j .

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

Algorithm 17 Operador de Reemplazo Estacionario

```

1: procedure (Op Reemplazo Estacionario)
2:   Calcular los 2 peores padres de la población actual  $\rightarrow$  padre1, padre2.
3:   if hijo1 > padre1 && hijo2 > padre2 then
4:     Intercambiar ambas soluciones de la población por ambos hijos
5:   else if hijo1 > padre2 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 selection, 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 18 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  $\rightarrow n_{cruce} = p_{cruce} * nelem$ 
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  $\rightarrow$  parejas
11:    end if
12:    if parejas =  $\emptyset$  then
13:      if threshold  $\neq$  0 then
14:        threshold = threshold-1
15:      end if
16:    else
17:      for i  $\in$  [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  $\rightarrow$  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

```

6.2.2. Componentes

Operador de reparación

Se utilizará el mismo que se ha presentado anteriormente en los componentes de AG. Véase el pseudocódigo 14.

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

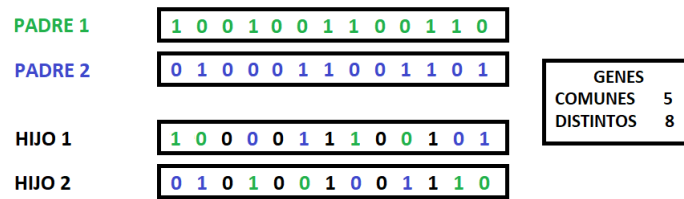


Figura 6.4: Cruce HUX

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 (19) el comportamiento de este tipo de cruce. Aunque primero se mostrará un ejemplo de este tipo de cruce en Figura 6.4.

Algorithm 19 Cruce HUX

```

1: procedure (Cruce HUX)(padre1, padre2)
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:       hijo1[indice[i]] = padre1[indice[i]]
7:       hijo2[indice[i]] = padre2[indice[i]]
8:     else
9:       hijo1[indice[i]] = padre2[indice[i]]
10:      hijo2[indice[i]] = padre1[indice[i]]
11:    end if
12:  end for
13: end procedure

```

Enfrentamiento

Es una generalización del Operador de Reemplazo Estacionario del AG (Algoritmo 17). 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 20)

Algorithm 20 Enfrentamiento CHC

```
1: procedure (Enfrentamiento)(poblacion, hijos)  
2:   Calcular la función fitness de poblacion e hijos  
3:   Unir ambas poblaciones  $\rightarrow$  pobTotal  
4:   Unir los valores de ambas poblaciones  $\rightarrow$  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 componentes, es decir, las funcionalidades propias de los algoritmos que se utilizarán como base (presentados en el capítulo anterior). Se describirán detalladamente además de usar pseudocódigo para representarlos.

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.

Cuadro 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.

- 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)

Capítulo 9: Conclusiones

