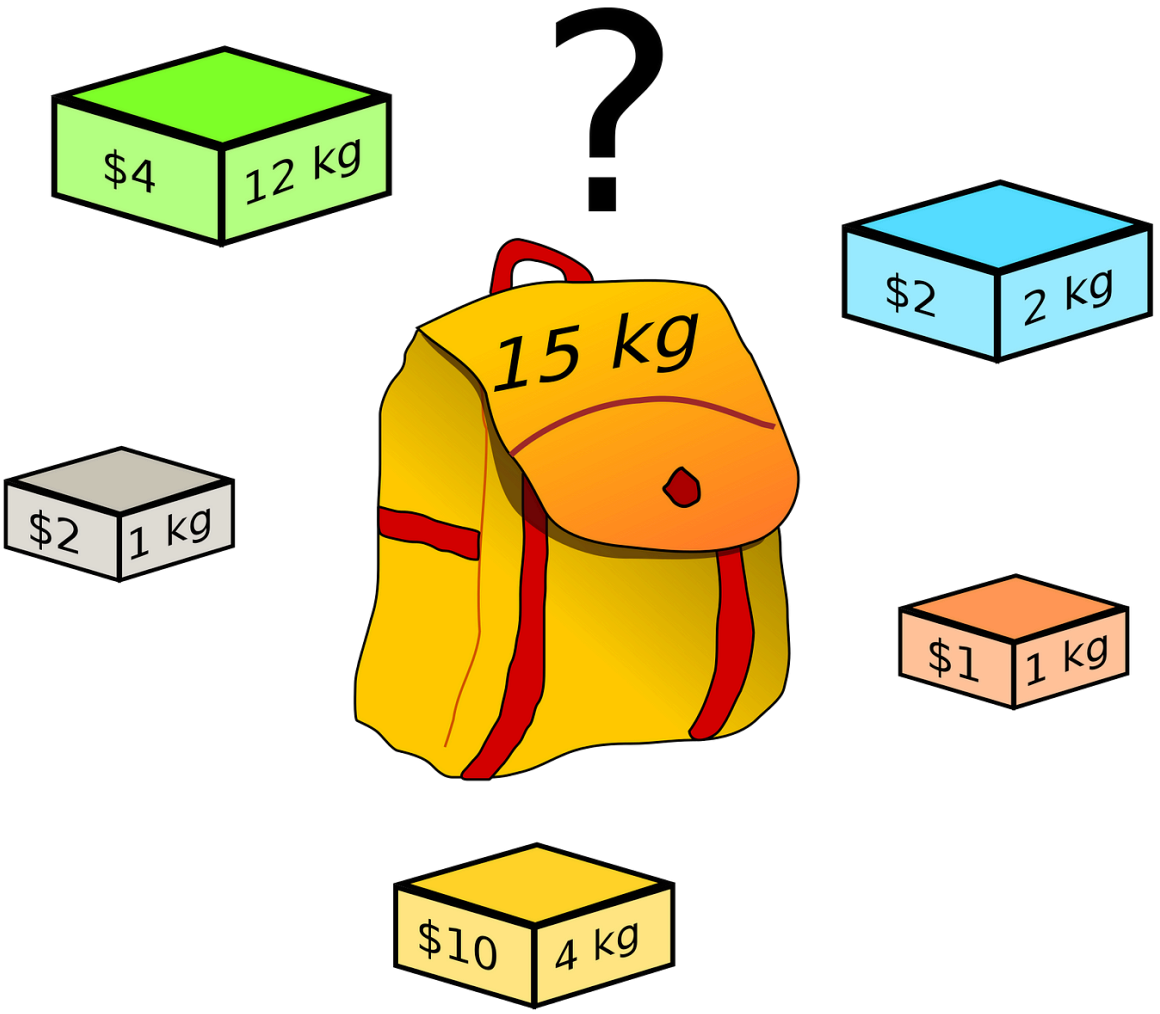
# Práctica 1
# Resolución del Problema de la Mochila Cuadrático mediante los algoritmos Greedy y de Búsqueda Local(Primer Mejor)




**Santiago Herron Mulet**
**Metaheurística Grupo de Prácticas 3**

# ÍNDICE


**1. Descripción del problema. 4**
a. Planteamiento: 4
b. Formulación matemática: 5
i. Función a maximizar f(x): 5
ii. Sujeta a: 5
c. Propiedades y complejidad: 5
d. Algoritmos de resolución: 5
i. Algoritmos exactos: 5
ii. Algoritmos heurísticos: 6
**2. Descripción de los ámbitos generales de los algoritmos Greedy y Búsqueda Local. 6**
a. Greedy y BLPM como algoritmos heurísticos. 6
i. Naturaleza heurística. 6
ii. Búsqueda incremental. 6
iii. Evaluación Local. 6
iv. Riesgo de óptimos locales. 6
v. Ventajas y desventajas. 7
b. Clase Objeto: 7
i. Atributos: 7
ii. Métodos: 7
iii. Descripción: 8
c. Clase Mochila. 8
i. Atributos: 8
ii. Métodos: 8
iii. Descripción: 9
iv. Función Fitness: 9
d. Otras implementaciones comunes. 9
i. Lectura desde ficheros: 9
ii. Cálculo de tiempos de ejecución: 10
**3. Greedy 10**
a. Cabecera. 10
i. Salida: 10
ii. Parámetros: 10
b. Heurística. 11
c. Función de ordenación según ratio beneficio - peso. 11
d. Algoritmo. 12
**4. Búsqueda Local Primero Mejor. 13**
a. Cabecera: 13
i. Salida: 13
ii. Parámetros: 13
b. Heurística: 13
c. Función de Generación de Vecindario: 13
d. Asignación inicial. 14

e. Algoritmo: 14
**5. Extra - Función Generación de Vecindario por doble intercambio. 16**
**6. Extra - Búsqueda Local Mejor Vecino 17**
**7. Procedimiento para desarrollar la práctica. 18**
a. Organización del proyecto. 18
b. Manual de usuario: 19
i. Compilación: 19
ii. Ejecución: 19
iii. Salida: 19
iv. Scripts: 19
**8. Experimentos y análisis de resultados: 20**
a. Recogida de datos. 20
b. Resultados: 20
i. Greedy 20
ii. Búsqueda Local Primero Mejor 21
iii. Búsqueda Local Primero Mejor con generación de vecinos por doble intercambio. 21
iv. Búsqueda Local Mejor. 22
v. Búsqueda Local Mejor con generación de vecinos por doble intercambio. 22
c. Resumen de Resultados: 23
i. Tamaño 100: 23
ii. Tamaño 200: 23
iii. Tamaño 300: 24
d. Evolución número de instancias frente a beneficio. 24
e. Análisis de resultados - Greedy vs BL. 26
f. BL vs BL-DOBLE vs BL-MEJOR vs BL-MEJOR-DOBLE. 27

# 1. Descripción del problema.

## a. Planteamiento:

El problema de la mochila cuadrática (QKP) es un problema de optimización combinatoria NP-completo que busca maximizar el beneficio obtenido al seleccionar un conjunto de objetos para colocar en una mochila con capacidad limitada. Cada objeto tiene un peso, un valor y un beneficio adicional que se obtiene si se seleccionan dos objetos específicos. El objetivo es encontrar la combinación de objetos que maximice el beneficio total sin exceder la capacidad de la mochila.

## b. Formulación matemática:

### i. Función a maximizar f(x):

$$\text{maximize} \left\{ \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{n} \sum_{j=1, i \neq j}^{n} P_{ij} x_i x_j : x \in X, x \text{ binary} \right\}$$

### ii. Sujeta a:

$$\text{subject to } X \equiv \left\{ x \in \{0,1\}^n : \sum_{i=1}^{n} w_i x_i \leq W; x_i \in \{0,1\} \text{ for } i = 1, \ldots, n \right\}$$

donde:
p_i: Valor del objeto i.
p_ij: Beneficio adicional obtenido al seleccionar los objetos i y j juntos.
w_i: Peso del objeto i.
W: Capacidad de la mochila.
x_i: Variable binaria que indica si el objeto i se selecciona (x_i = 1) o no (x_i = 0).

## c. Propiedades y complejidad:

El QKP es un problema NP-completo, lo que significa que no existe un algoritmo conocido que pueda resolverlo de manera eficiente para todas las instancias. A medida que aumenta el tamaño del problema, la complejidad computacional para encontrar la solución óptima aumenta exponencialmente.

### d. Algoritmos de resolución:

Existen diversos algoritmos para resolver el QKP, los cuales se pueden dividir en dos categorías principales:

#### i. Algoritmos exactos:

Estos algoritmos garantizan encontrar la solución óptima, pero pueden ser muy lentos para problemas de gran tamaño.

#### ii. Algoritmos heurísticos:

Estos algoritmos no garantizan encontrar la solución óptima, pero son mucho más rápidos que los algoritmos exactos.

## 2. Descripción de los ámbitos generales de los algoritmos Greedy y Búsqueda Local.

### a. Greedy y BLPM como algoritmos heurísticos.

#### i. Naturaleza heurística.

Tanto el algoritmo Greedy como el algoritmo de Búsqueda Local Primero Mejor (BLPM) son algoritmos heurísticos. Esto significa que no garantizan encontrar la solución óptima al problema de la mochila cuadrática (QKP). En cambio, ofrecen soluciones aproximadas que pueden ser de buena calidad, pero no siempre son las mejores posibles.

#### ii. Búsqueda incremental.

Ambos algoritmos construyen la solución de forma incremental, explorando el espacio de soluciones paso a paso. En cada paso, se evalúan diferentes opciones y se selecciona la que se considera más prometedora para mejorar el valor de la función objetivo.

#### iii. Evaluación Local.

Los algoritmos heurísticos basan su toma de decisiones en la evaluación de la información local. Es decir, solo consideran el impacto inmediato de una decisión en la solución actual, sin tener en cuenta el efecto global que podría tener en el problema completo.

### iv. Riesgo de óptimos locales.

Debido a la naturaleza local de la evaluación, tanto el algoritmo Greedy como el BLPM pueden quedar atrapados en óptimos locales. Un óptimo local es una solución que es mejor que sus vecinos inmediatos, pero no es la mejor solución posible en el espacio de soluciones completo.

### v. Ventajas y desventajas.

Los algoritmos heurísticos ofrecen una serie de ventajas, como su simplicidad, velocidad y eficiencia en comparación con los métodos exactos. Sin embargo, también tienen la desventaja de no garantizar la obtención de la solución óptima y de ser susceptibles a quedar atrapados en óptimos locales.

## b. Clase Objeto:

### i. Atributos:

indice: Un entero que representa el índice único del objeto.
b_individual: Un entero que representa el valor individual del objeto.
b_interdependencias: Un vector de enteros que representa los beneficios adicionales de seleccionar el objeto junto con cada uno de los demás objetos.
peso: Un entero que representa el peso del objeto.
b_acumulado: Un entero que representa el valor acumulado del objeto, que se calcula sumando el valor individual del objeto y los beneficios adicionales de seleccionarlo junto con todos los objetos que comparten mochila con este.

### ii. Métodos:

Objeto().
Objeto(int i, int bi, vector<int> b_inter, int p).
getIndice().
getB_Individual().
getB_Acumulado().
getB_Interdependencias(int j).
getPeso().
aniadeB_Acumulado(int b).
operator==(const Objeto& other).

### iii. Descripción:

La clase "Objeto" representa un objeto individual en el problema de la mochila cuadrática. Cada objeto tiene un índice único, un valor individual, un conjunto de beneficios adicionales que se obtienen al seleccionarlo junto con otros objetos, un peso y un valor acumulado.
El valor acumulado del objeto se calcula sumando el valor individual del objeto y los beneficios
adicionales de seleccionarlo junto con todos los objetos de la mochila actual. Esto permite una evaluación eficiente del valor de una solución al problema de la mochila cuadrática.

## c. Clase Mochila.

### i. Atributos:

objetos: Un vector de objetos que representa los objetos que se pueden incluir en la mochila.
solucion: Un vector de booleanos que indica si un objeto está incluido (True) o no (False) en la mochila.
beneficio: Un entero que representa el beneficio total de la solución actual.

### ii. Métodos:

Mochila().
Mochila(vector<Objeto> objetos, vector<bool> asignacion).
getObjeto().
getAsignacion().
size().
getFitness().
getPeso().
operator==(const Mochila& other).
swap(int i, int j, int capacidad).

### iii. Descripción:

La clase Mochila representa una mochila en el problema de la mochila cuadrática (QKP). Cada mochila tiene un conjunto de objetos que se pueden incluir en ella, una solución que indica qué objetos están seleccionados y un beneficio total que se calcula en función de los objetos seleccionados.

### iv. Función Fitness:

La función Fitness de la clase Mochila calcula el beneficio total de la solución actual, es decir, la suma de los beneficios individuales de los objetos seleccionados más los beneficios adicionales obtenidos al seleccionar pares de objetos. Esto es, en definitiva, la función a maximizar por nuestros algoritmos.
Pseudocódigo:

```
Función Fitness():
   beneficio = 0
   Para cada objeto en el conjunto de objetos:
      Si el objeto está asignado a la mochila:
         beneficio += obtenerBeneficioIndividual(objeto)
         Para cada otro objeto en el conjunto de objetos:
            beneficio +=
obtenerBeneficioInterdependencias(objeto, otroObjeto) *
asignacion[otroObjeto]
   Devolver beneficio
```

## d. Otras implementaciones comunes.

### i. Lectura desde ficheros:

Se ha implementado una función de lectura para obtener los datos del problema desde un archivo externo. El programa espera dos argumentos como mínimo: el algoritmo a utilizar y la ruta del archivo. Luego, lee el número de objetos, la capacidad de la mochila, los pesos, los beneficios individuales y los beneficios por interdependencias desde el archivo. Estos datos se utilizan para crear un vector de objetos que representan los elementos del problema, que luego se utilizan en el algoritmo seleccionado para resolver el problema de la mochila cuadrática.

### ii. Cálculo de tiempos de ejecución:

Para calcular los tiempos de ejecución de los algoritmos, se utiliza la biblioteca <chrono>. Después de cargar los datos del problema desde el archivo, se ejecuta el algoritmo correspondiente: ya sea el algoritmo Greedy o uno de los algoritmos de Búsqueda Local (BL). Se toma el tiempo de inicio antes de la ejecución del algoritmo y el tiempo de finalización después de que el algoritmo haya terminado su ejecución. Luego, se calcula la diferencia entre estos dos tiempos para obtener el tiempo de ejecución total del algoritmo.

# 3. Greedy

## a. Cabecera.

### i. Salida:

El algoritmo devuelve un vector de bool del tamaño del número de objetos. Si un índice está a true significa que el objeto del mismo índice fue seleccionado para formar parte de la mochila solución.

### ii. Parámetros:

1. vector de objetos.
2. Tamaño.
3. Capacidad de la mochila (W)

## b. Heurística.

La heurística aplicada para este algoritmo voraz es muy sencilla. Simplemente, en cada instante, se elige incluir en la mochila aquel objeto que, no superando la capacidad actual de la mochila, tenga mayor ratio beneficio - peso. Este beneficio se calculará sumando el beneficio actual del objeto, más el beneficio por interdependencias de ese objeto con todos los objetos ya seleccionados.

## c. Función de ordenación según ratio beneficio - peso.

Para esta función de ordenación se eligió la metodología de ordenación quicksort por su buena eficiencia promedio.
Pseudocódigo:

```
Función compare_ratios(obj1, obj2):
        ratio_obj1 = obj1.getB_Acumulado() / obj1.getPeso()
        ratio_obj2 = obj2.getB_Acumulado() / obj2.getPeso()
        Devolver ratio_obj1 < ratio_obj2

Función particion(objeto, bajo, alto):
        pivote = objeto[alto]
        i = bajo - 1
        Para j desde bajo hasta alto - 1:
        Si compare_ratios(objeto[j], pivote):
                Incrementar i
                Intercambiar objeto[i] con objeto[j]
        Intercambiar objeto[i + 1] con objeto[alto]
        Devolver i + 1

Función quicksort(objeto, bajo, alto):
```

```
                Si bajo < alto:
                pi = particion(objeto, bajo, alto)
                quicksort(objeto, bajo, pi - 1)
                quicksort(objeto, pi + 1, alto)

        Función OrdenGlobal(objeto):
                n = tamaño de objeto
                quicksort(objeto, 0, n - 1)
```

### d. Algoritmo.

El algoritmo comenzará ordenando el vector de objetos. A continuación entrará en un bucle en el que seguirá hasta que el vector de objetos esté vacío. En el bucle comprobará si el objeto más prometedor cabe. Si cabe lo incluye en la asignación y lo elimina del vector, si no cabe lo elimina.

```
Función greedy(objeto, n, W, peso_final):
   vector<bool> v(n)
   capacidad = W
   OrdenGlobal(objeto)
   Mientras objeto no esté vacío:
      Si capacidad >= objeto[tamaño de objeto - 1].getPeso():
         capacidad -= objeto[tamaño de objeto - 1].getPeso()
         v[objeto[tamaño de objeto - 1].getIndice()] = verdadero
         Para cada objeto[j] en objeto:
            Si j < tamaño de objeto - 1:

objeto[j].aniadeB_Acumulado(objeto[j].getB_Interdependencias(objeto[t
amaño de objeto - 1].getIndice()))
         Eliminar el último objeto de objeto
         OrdenGlobal(objeto)
      Sino:
         Eliminar el último objeto de objeto
   Devolver v
```

# 4. Búsqueda Local Primero Mejor.

## a. Cabecera:

### i. Salida:

El algoritmo devuelve un vector de bool del tamaño del número de objetos. Si un índice está a true significa que el objeto del

mismo índice fue seleccionado para formar parte de la mochila solución.

### ii. Parámetros:

1. Mochila actual. (El algoritmo se puede llamar desde soluciones parciales).
2. Tamaño.
3. Capacidad W.
4. Máximo número de vecinos que se pueden explorar.
5. Referencia al número de instancias exploradas.
6. Referencia a bool que indica si se ha llegado a óptimo local.
7. Referencia a función de generación de vecinos.

## b. Heurística:

Primero hay que definir el concepto de vecino:
Definiremos vecino a partir de la configuración de su vector solución, o sea, de los objetos que incluya o no en la mochila.
Para nuestra aproximación, diremos que una mochila es vecina de otra cuando entre las dos la única diferencia sea una permutación entre un objeto seleccionado y otro no. Obviamente, solo se generarán vecinos factibles, es decir, que acaten la restricción de no superar la capacidad de la mochila.
La búsqueda local, para la mochila actual, genera su vecindario. Selecciona un vecino aleatorio hasta que éste mejore la mochila actual. Repetir el proceso hasta que en un vecindario no mejore la mochila actual o hasta que se llegué al máximo de vecinos expandidos.

## c. Función de Generación de Vecindario:

```
Función VecinosPermutacion(actual: Mochila, W: entero) -> vector<pair<entero, vector<vector<entero>>>>:
    vecinos = nuevo vector<pair<entero, vector<vector<entero>>>>()
    objeto = actual.getObjeto()
    v = actual.getAsignacion()

    Para cada índice i en rango(0, objeto.size()):
        Si v[i] es verdadero:
            Para cada índice j en rango(0, objeto.size()):
                Si no v[j] y i != j:  // Evitar intercambiar un objeto consigo mismo
                    Si actual.getPeso() - objeto[i].getPeso() + objeto[j].getPeso() <= W:
```

```
                    // Se puede realizar el intercambio sin exceder la
capacidad
                    vecinos.push_back({objeto[i].getIndice(),
{{objeto[j].getIndice()}}})

  Devolver vecinos
```

## d. Asignación inicial.

```
Función solucionInicial(objeto, n, capacidad):
  s = lista de tamaño n, inicializada con todos los elementos a falso
  o = copia de la lista de objetos
  Mientras o no esté vacía:
    Mezclar(o)
    Si capacidad >= peso del último objeto en o y s[indice del último
objeto en o] es falso:
      capacidad -= peso del último objeto en o
      s[indice del último objeto en o] = verdadero
      Eliminar el último objeto en o
    Sino:
      Eliminar el último objeto en o
  Devolver s
```

## e. Algoritmo:

```
Función busquedaLocal(actual: Mochila, n: entero, W: entero,
max_instancias: entero, n_instancias: referencia entero,
                  maximo_local: referencia booleano, Vecinos:
Función(VecinosPermutacion) -> vector<pair<entero,
vector<vector<entero>>>>) -> vector<booleano>:
  mejor_asignacion = actual.getAsignacion()
  mejor_fitness = actual.getFitness()
  maximo_local = falso
  n_instancias = 0

  Mientras n_instancias < max_instancias y no maximo_local hacer:
    vecinos = Vecinos(actual, W)
    Random::shuffle(vecinos)

    Para cada vecino en vecinos hacer:
      Para cada intercambio en Random::shuffle(vecino.second)
hacer:
        v = actual.getAsignacion()
        Intercambio(v, vecino.first, intercambio)
```

```
            nueva = Mochila(actual.getObjeto(), v)

            Si nueva.getFitness() > mejor_fitness entonces:
                mejor_asignacion = v
                mejor_fitness = nueva.getFitness()
                actual = nueva
                maximo_local = falso
                romper  // Salir del bucle de intercambios
            Fin Si
        Fin Para

        Si mejor_fitness no cambió entonces:
            maximo_local = verdadero  // No se encontró mejora en esta
iteración
        Fin Si

        Si no maximo_local entonces:
            romper  // Salir del bucle de vecinos
        Fin Si
    Fin Para

    Incrementar n_instancias
  Fin Mientras

  Devolver mejor_asignacion
Fin Función
```

# 5. Extra - Función Generación de Vecindario por doble intercambio.

A modo de trabajo extra se desarrolló la siguiente función de generación de vecindario:

```
Función VecinosDobleIntercambio(actual: Mochila, capacidad: entero) ->
vector<pair<entero, vector<vector<entero>>>>:
  vecinos = VecinosPermutacion(actual, capacidad)
  objeto = actual.getObjeto()
  v = actual.getAsignacion()

  Para cada vecino en vecinos hacer:
    Para cada grupo en vecino.second hacer:
      objeto_actual = vecino.first
      objeto_cero = grupo[0]
```

```
        capacidad_actual = capacidad - objeto[objeto_actual].getPeso() +
objeto[objeto_cero].getPeso()

        Para cada k en rango(0, objeto.size()) hacer:
            Si no v[k] y k != objeto_actual y k != objeto_cero entonces:
                nuevo_peso = capacidad_actual + objeto[k].getPeso()
                Si nuevo_peso >= 0 y nuevo_peso <= capacidad entonces:
                    Si tamaño(grupo) < 2 entonces:
                        grupo.push_back(k)
                        vecino.second.push_back(grupo)
                    Fin Si
                Fin Si
            Fin Si
        Fin Para
    Fin Para
  Fin Para

  Devolver vecinos
Fin Función
```

Esta función añade a la primera, en la que se buscaba cambiar la configuración de la asignación cambiando un objeto que estaba en la asignación por otro que no, otro objeto si cabe. Es decir, si se decide usar esta función para generar el vecindario se obtendrán vecinos que, respecto a la asignación original, cambiarán en que un objeto de los que estaba a 1 estará a 0 y después un objeto o dos objetos que estaban a 0 estén a 1. Esto dará pie a una ligera mejora del algoritmo al generar vecindarios diferentes, más amplios. Además también puede arreglar el problema de empezar el algoritmo por una primera asignación deficiente en cuanto a objetos.
Sin embargo, el uso de este algoritmo supone un incremento del costo computacional.

# 6. Extra - Búsqueda Local Mejor Vecino

Se desarrolló una nueva función de búsqueda local pero está vez buscando en cada vecindario el mejor vecino.

```
Función busquedaLocalMejor(actual: Mochila, n: entero, W: entero,
max_instancias: entero, n_instancias: referencia entero,
```

```
                    maximo_local: referencia booleano, Vecinos:
Función(VecinosPermutacion) -> vector<pair<entero,
vector<vector<entero>>>>) -> vector<booleano>:
  mejor_asignacion = actual.getAsignacion()
  mejor_fitness = actual.getFitness()
  capacidad = W
  n_instancias = 0
  objeto = actual.getObjeto()
  maximo_local = falso

  Mientras n_instancias < max_instancias y no maximo_local hacer:
    vecinos = Vecinos(actual, capacidad)
    mejora_encontrada = falso
    mejor_mochila = actual // Mantener un seguimiento de la mejor mochila
encontrada en esta iteración

    Para cada vecino en vecinos hacer:
      Para cada intercambio en vecino.second hacer:
        v = actual.getAsignacion()
        Intercambio(v, vecino.first, intercambio) // Función para aplicar
intercambio
        nueva = Mochila(objeto, v)

        Si nueva.getFitness() > mejor_fitness entonces:
          mejor_asignacion = v
          mejor_fitness = nueva.getFitness()
          mejor_mochila = nueva // Actualizar la mejor mochila encontrada
          mejora_encontrada = verdadero
        Fin Si
      Fin Para
    Fin Para

    Si mejora_encontrada entonces:
      actual = mejor_mochila // Actualizar el estado de la mochila solo si se
encontró una mejora
    Sino:
      maximo_local = verdadero // No se encontró mejora en esta iteración
    Fin Si

    Incrementar n_instancias
  Fin Mientras

  Devolver mejor_asignacion
Fin Función
```

Esta implementación solo se mueve entre las mejores soluciones de cada vecindario. Esto supone una mejora en cuanto al beneficio final pero hace que sea más costoso que la versión original.

# 7. Procedimiento para desarrollar la práctica.

## a. Organización del proyecto.

Para comenzar el desarrollo de la práctica, lo primero fue tomar las decisiones a nivel de estructuras. Con ese fin, se crearon las clases Objeto y Mochila ya explicadas.
A continuación se decidió la estructura de ficheros. Usando un fichero para el main, otro para el desarrollo de las funciones del algoritmo greedy y otro para las funciones bl.
Por último, con el fin de facilitar el uso del software, se desarrolló un Makefile para el compilado automático de las fuentes y diversos bash scripts para generación de múltiples resultados para posteriormente analizarlos.

## b. Manual de usuario:

### i. Compilación:

Con la orden make.
Se genera un ejecutable llamado QKP.

### ii. Ejecución:

Los argumentos del programa son.

1. ./$ejecutable
2. $Algoritmo (G para greedy o BL para búsqueda local).
3. $fichero de entrada de datos
4. (opcional) $semilla (Para greedy no es necesaria, para bl, si no se indica, no se fijará semilla).

### iii. Salida:

Se mostrará por pantalla lo siguiente:

1. W: capacidad de la mochila.
2. Peso de la mochila final.
3. Beneficio final.
4. Asignación (unos y ceros)
5. Última línea: $beneficio $milisegundos

### iv. Scripts:

Importante: si se desea utilizar alguno de los scripts se deben comentar todas las salidas de texto por pantalla menos la última

línea que saca el beneficio y el tiempo y volver a compilar con make.

Cada uno de esos scripts ejecuta el programa con el algoritmo correspondiente para cada uno de los archivos datos de entrada. Para aquellos con tamaño y densidad comunes, hace la media del beneficio y del tiempo y guarda los resultados en ficheros en formato csv.
Se desarrollaron 5 scripts:

1. ScriptGreedy.sh
2. ScriptBL.sh
3. ScriptBL-DOBLE.sh (extra)
   algoritmo BL con vecinosDobleIntercambio.
4. ScriptBL-MEJOR.sh (extra)
   algoritmo BL mejor vecino
5. ScriptBL-MEJOR-DOBLE.sh (extra, no se recomienda su ejecución pues tarda demasiado)
   algoritmo BL mejor vecino con vecinosDobleIntercambio.

# 8. Experimentos y análisis de resultados:

## a. Recogida de datos.

Para la recogida de datos experimentales se usaron los scripts mencionados usando la semilla **33**. Posteriormente, se organizaron en tablas.

## b. Resultados:

### i. Greedy

| Resultados algoritmo greedy | | | |
|---|---|---|---|
| Tamaño | Densidad | Fitness Medio | Tiempo Medio |
| 100 | 25 | 62994 | 0 |
| 100 | 50 | 130766 | 0 |
| 100 | 75 | 190036 | 0 |
| 100 | 100 | 255929 | 0 |
| 200 | 25 | 245760 | 2 |
| 200 | 50 | 569501 | 2 |

| | | | |
|---|---|---|---|
| 200 | 75 | 557095 | 2 |
| 200 | 100 | 928975 | 2 |
| 300 | 25 | 431144 | 6 |
| 300 | 50 | 1157976 | 9 |

ii. Búsqueda Local Primero Mejor

| Resultados Algoritmo BL | | | |
|---|---|---|---|
| Tamaño | Densidad | Fitness Medio | Tiempo Medio |
| 100 | 25 | 52650 | 102 |
| 100 | 50 | 96299 | 129 |
| 100 | 75 | 144459 | 129 |
| 100 | 100 | 214603 | 71 |
| 200 | 25 | 196312 | 1418 |
| 200 | 50 | 425702 | 1641 |
| 200 | 75 | 340975 | 1686 |
| 200 | 100 | 658963 | 1693 |
| 300 | 25 | 324532 | 6291 |
| 300 | 50 | 863514 | 8577 |

iii. Búsqueda Local Primero Mejor con generación de vecinos por doble intercambio.

| Resultados algoritmos BL-DOBLE | | | |
|---|---|---|---|
| Tamaño | Densidad | Fitness Medio | Tiempo Medio |
| 100 | 25 | 63336 | 114 |
| 100 | 50 | 130274 | 145 |
| 100 | 75 | 189301 | 95 |
| 100 | 100 | 252068 | 89 |
| 200 | 25 | 244909 | 2131 |
| 200 | 50 | 563786 | 2826 |
| 200 | 75 | 551098 | 4158 |

| | | | |
|---|---|---|---|
| 200 | 100 | 914399 | 2653 |
| 300 | 25 | 428830 | 10892 |
| 300 | 50 | 1141011 | 12361 |

iv. Búsqueda Local Mejor.

| Resultados algoritmo BL-MEJOR | | | |
|---|---|---|---|
| Tamaño | Densidad | Fitness Medio | Tiempo Medio |
| 100 | 25 | 60910 | 117 |
| 100 | 50 | 120402 | 150 |
| 100 | 75 | 173098 | 136 |
| 100 | 100 | 241271 | 88 |
| 200 | 25 | 234370 | 1655 |
| 200 | 50 | 525723 | 1892 |
| 200 | 75 | 475650 | 2012 |
| 200 | 100 | 847882 | 1611 |
| 300 | 25 | 399176 | 8855 |
| 300 | 50 | 1053699 | 10827 |

v. Búsqueda Local Mejor con generación de vecinos por doble intercambio.

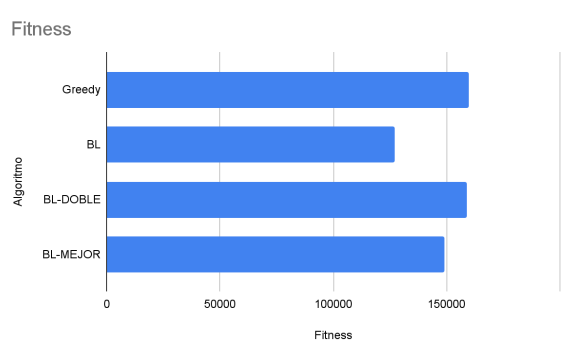
| Resultados algoritmo BL-MEJOR-DOBLE | | | |
|---|---|---|---|
| Tamaño | Densidad | Fitness Medio | Tiempo Medio |
| 100 | 25 | 63341 | 1416 |
| 100 | 50 | 130600 | 2439 |
| 100 | 75 | 188336 | 1383 |
| 100 | 100 | 253107 | 966 |
| 200 | 25 | 244675 | 62845 |
| 200 | 50 | 565358 | 94980 |

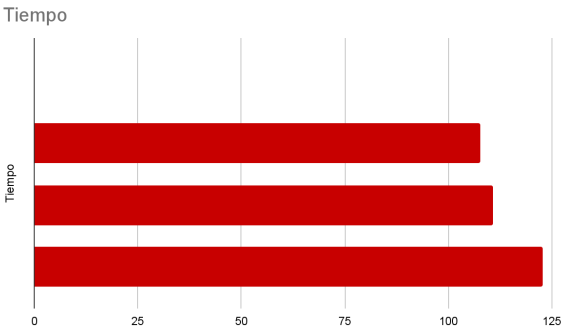
| | | | |
|---|---|---|---|
| 200 | 75 | 550930 | 114933 |
| 200 | 100 | 913951 | 135273 |

## c. Resumen de Resultados:

### i. Tamaño 100:

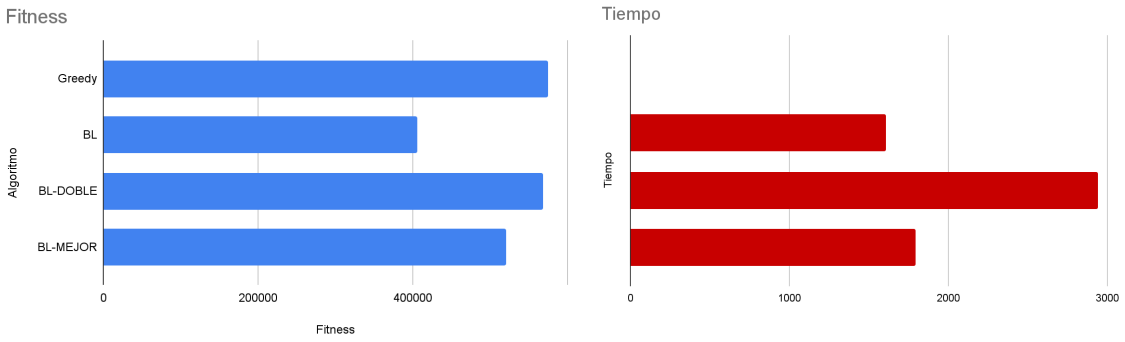
| Resumen algoritmos Tamaño 100 | | |
|---|---|---|
| Algoritmo | Fitness | Tiempo |
| Greedy | 159931,25 | 0 |
| BL | 126977,75 | 107,75 |
| BL-DOBLE | 158744,75 | 110,75 |
| BL-MEJOR | 148920,25 | 122,75 |



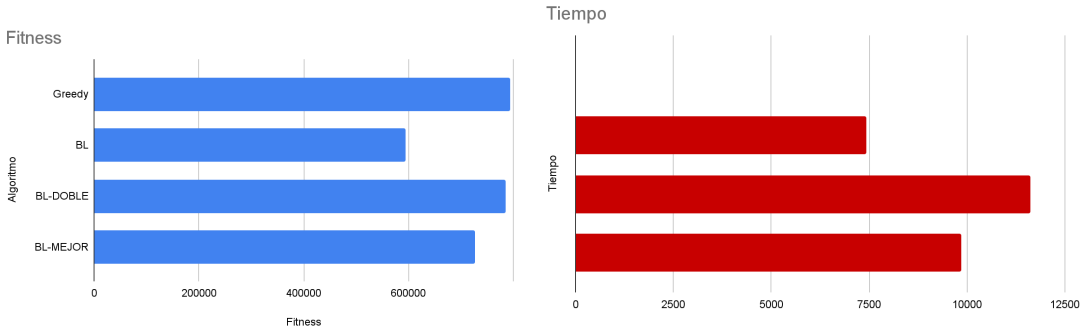

### ii. Tamaño 200:

| Resumen algoritmos tamaño 200 | | |
|---|---|---|
| Algoritmo | Fitness | Tiempo |
| Greedy | 575332,75 | 2 |
| BL | 405488 | 1609,5 |
| BL-DOBLE | 568548 | 2942 |
| BL-MEJOR | 520906,25 | 1792,5 |




iii. Tamaño 300:

| Algoritmo | Fitness | Tiempo |
|---|---|---|
| Greedy | 794560 | 7,5 |
| BL | 594023 | 7434 |
| BL-DOBLE | 784920,5 | 11626,5 |
| BL-MEJOR | 726437,5 | 9841 |



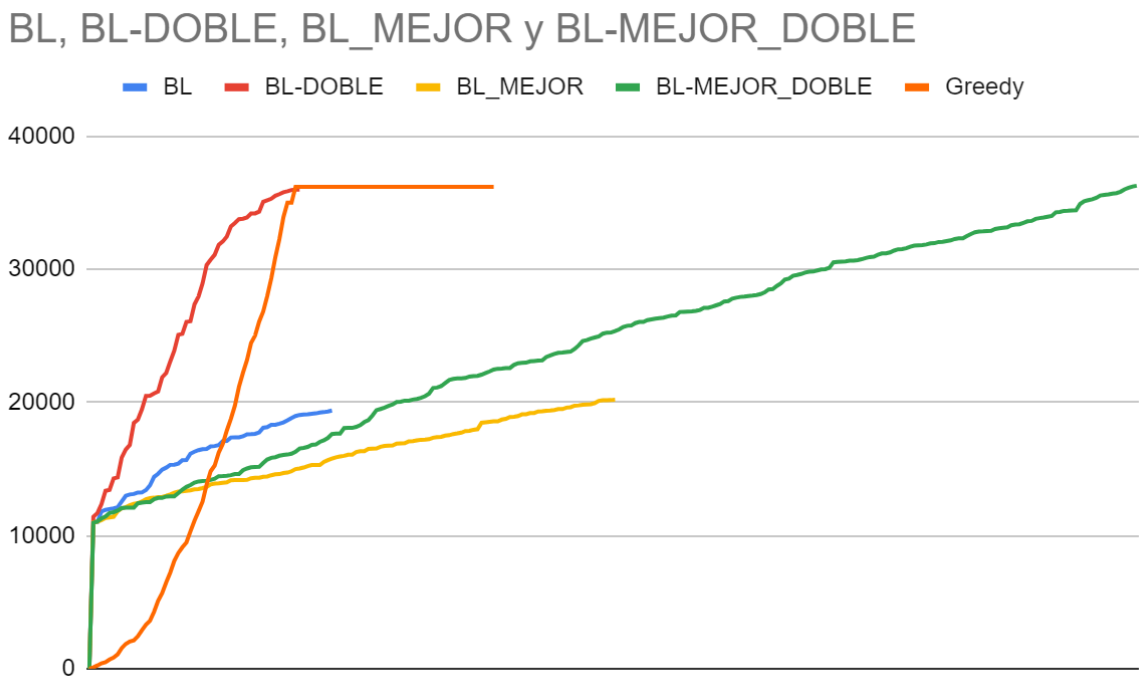

## d. Evolución número de instancias frente a beneficio.

Para observar como van evolucionando las soluciones parciales de los algoritmos (característica incremental) se sacaron los datos parciales de una ejecución en concreto y se observaron mediante la siguiente gráfica de convergencia.
Ejecución: ./QKP $(algoritmo) bin/data/jeu_100_25_1.txt 33

Datos:
Grafica Convergencia

Gráfica:

Eje x: número de instancias.
Eje y: beneficio.




Se puede observar que los algoritmos BL crecen muy rápido al principio debido a la solución inicial. A partir de ahí, podemos observar que hay diferencias entre los algoritmos bl que usan la generación de vecinos normal (1 por 1) y la doble (1 por 2) ya que BL y BL-MEJOR al tener vecindarios más acotados, se estancan antes al caer en óptimos locales y los algoritmos BL-DOBLE y BL-MEJOR-DOBLE generan instancias hasta valores de beneficios más altos, es decir, los óptimos locales en los que caen son mejores. También se puede observar similitudes y diferencias entre los algoritmos de búsqueda local primero mejor y búsqueda local mejor. Los primeros, dan pie a generar crecimientos considerables de forma más repentina haciendo que la mejora, por lo general sea más rápida. Sin embargo los algoritmos de búsqueda local mejor mantienen un crecimiento más constante haciendo que su beneficio crezca más lento pero consiguiendo, por norma general, beneficios finales iguales o mejores.

Por otro lado, el algoritmo greedy, evalúa todos los objetos (en este caso 100). Llena y llena la mochila hasta llegar al límite o cerca de la capacidad momento en el que se estanca al no poder incluir más objetos porque no caben.

## e. Análisis de resultados - Greedy vs BL.

A la vista de las tablas y gráficas resumen, es más que obvio que los algoritmos de búsqueda local no pueden competir con el algoritmo greedy. Esto es tanto por el lado del fitness (beneficio) como por el tiempo de cómputo.

El algoritmo greedy se destaca por su simplicidad y eficiencia. Su enfoque de selección de objetos basado únicamente en la maximización del beneficio por unidad de peso puede conducir a soluciones subóptimas en algunos casos, pero en el contexto del problema QKP, su capacidad para generar soluciones cercanas a óptimas es sorprendente. La naturaleza voraz del algoritmo greedy le permite tomar decisiones rápidas y consistentes, lo que resulta en soluciones que, aunque no sean necesariamente óptimas, son altamente competitivas.

Por otro lado, los algoritmos de búsqueda local, como su nombre lo indica, se basan en la exploración y refinamiento iterativo de soluciones en un espacio de búsqueda local. Estos algoritmos pueden encontrar soluciones óptimas locales al enfocarse en la mejora incremental a partir de una solución inicial, pero su desempeño puede verse limitado por la naturaleza de la exploración local. En el caso del QKP, donde la búsqueda local se enfrenta a un espacio de soluciones vasto y complejo, es posible que se quede atrapada en óptimos locales y no pueda explorar soluciones alternativas que puedan ser más competitivas.

Además, la eficacia de los algoritmos también puede verse afectada por la estructura específica del problema y las características de las instancias utilizadas en el experimento. Si las instancias del problema presentan ciertas propiedades que favorecen la estrategia del algoritmo greedy, como la presencia de objetos con ratios beneficio-peso muy dispares, el rendimiento del algoritmo greedy puede ser aún más notable en comparación con los algoritmos de búsqueda local.

En resumen, el dominio del algoritmo greedy sobre los algoritmos de búsqueda local en el problema QKP puede atribuirse a su capacidad para generar soluciones competitivas de manera eficiente, su enfoque directo en la maximización del beneficio y las características específicas del problema y las instancias utilizadas. Si bien los algoritmos de búsqueda local ofrecen un enfoque más sofisticado para la optimización iterativa, su desempeño puede verse comprometido por la complejidad del espacio de búsqueda y la posibilidad de quedar atrapados en óptimos locales.

## f. BL vs BL-DOBLE vs BL-MEJOR vs BL-MEJOR-DOBLE.

BL-MEJOR-DOBLE no se considerará significativamente en el análisis debido a su tiempo de cómputo extremadamente alto, lo que hace inviable su ejecución para tamaños grandes, a pesar de su mejora en el fitness.

- BL: Produce resultados consistentes con un fitness decente, aunque es el peor en comparación con el resto en este aspecto. Sin embargo, su tiempo de cómputo es el más corto.
- BL-DOBLE: Experimenta una mejora significativa en el fitness, llegando casi al nivel del algoritmo greedy. Sin embargo, esto va acompañado de un aumento considerable en el tiempo de cómputo, especialmente notable en tamaños grandes, donde es el más lento entre BL, BL-DOBLE y BL-MEJOR.
- BL-MEJOR: Mejora el fitness respecto a BL, aunque no alcanza los niveles de BL-DOBLE, quedando ligeramente por debajo. En cuanto al tiempo de cómputo, por lo general es más lento que BL, pero más rápido que BL-DOBLE.

  Además, surge una curiosidad: para tamaños pequeños, BL-MEJOR es más lento que BL-DOBLE, pero para tamaños grandes, BL-DOBLE es indiscutiblemente más lento que BL-MEJOR. Esto podría deberse al aumento en el costo de generar vecinos con el doble intercambio a medida que aumenta el tamaño del problema.

Los diferentes enfoques de los algoritmos de búsqueda local (BL) y sus variantes (BL-DOBLE, BL-MEJOR y BL-MEJOR-DOBLE) conducen a diferencias significativas en los resultados y las consecuencias.

En primer lugar, BL se centra en una estrategia de búsqueda local simple, donde se evalúa y se adopta la primera solución mejor que la actual. Esta estrategia, aunque puede ser eficiente en términos de tiempo de ejecución, limita la capacidad de explorar soluciones más óptimas que pueden encontrarse más lejos en el espacio de búsqueda.

Por otro lado, BL-DOBLE y BL-MEJOR-DOBLE amplían el alcance del vecindario de búsqueda, considerando más soluciones potenciales. Esto conlleva una mejora en la calidad de la solución al permitir la exploración de un espacio de búsqueda más amplio. Sin embargo, este aumento en la complejidad del vecindario conlleva un incremento significativo en el tiempo

de cómputo, ya que se requiere más tiempo para evaluar y explorar las soluciones potenciales adicionales.

Por último, BL-MEJOR y BL-MEJOR-DOBLE adoptan una estrategia más exhaustiva al buscar la mejor solución dentro del vecindario. Esto puede llevar a mejoras adicionales en la calidad de la solución, ya que se consideran múltiples vecinos antes de tomar una decisión. Sin embargo, esta exhaustividad también resulta en un aumento adicional en el tiempo de ejecución, ya que se necesita más tiempo para evaluar y seleccionar la mejor solución.

En resumen, las diferencias en los resultados y las consecuencias entre los algoritmos BL y sus variantes se deben principalmente a la forma en que exploran y evalúan el espacio de soluciones. BL ofrece una solución rápida pero potencialmente subóptima, mientras que BL-DOBLE ofrecen mejora en la calidad de la solución a expensas de un mayor tiempo de cómputo.
BL-MEJOR proporciona un equilibrio entre eficiencia y calidad de solución. La elección del algoritmo más adecuado depende de los requisitos específicos del problema y de los recursos computacionales disponibles.

Trabajo Realizado por Santiago Herron Mulet.