Práctica 1:
Técnicas de Búsqueda Local y Algoritmos Greedy

Curso 2023-24
Tercer Curso del Grado en Ingeniería Informática

Problema: Mochila Cuadrática
Algoritmos: Greedy y Búsqueda Local

José Antonio Zamora Reyes

Metaheurística , Grupo 1 (Martes, 17:30-19:30)

UNIVERSIDAD
DE GRANADA

ÍNDICE

1. Descripción del problema

El problema de la mochila cuadrática (QKP) es un problema del cual partimos con un número de objetos **(n)** ,que tiene asociados unos pesos**(wi)**, y una capacidad máxima **(W)**.

Consiste en generar una solución de objetos de manera que se obtenga un beneficio total máximo , siempre cumpliendo que la sumatoria de pesos de los objetos seleccionados sea menor o igual a la capacidad máxima de la mochila.

La solución vendrá representada por un vector binario**(X)** , indicando con 1 los objetos seleccionados y con 0 los no seleccionados.

$$\text{X debe cumplir que} \left(x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i \leq W \right)$$

A diferencia del problema de la mochila clásica ,que solo trabaja con beneficios individuales de objetos , en el problema de la mochila cuadrática contamos , aparte de con beneficios individuales de objetos **(pi)** , con beneficios combinados entre objetos **(pij)**.

Los beneficios combinados entre objetos estarán representados por una matriz y los beneficios individuales por un vector.

El contar tanto con beneficios individuales como con combinados, indica que el beneficio de un objeto depende tanto de él mismo como de los objetos ya asignados a la solución.

Buscamos maximizar:

$$\max \left(\sum_{i=1}^{n} p_i \cdot x_i + \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} p_{ij} \cdot x_i \cdot x_j \right)$$

En el problema de la mochila cuadrática es NP-completo ,no podemos garantizar que con más de 100 objetos podamos alcanzar el óptimo .

2. Descripción de la aplicación de algoritmos empleados en el problema

2.1. Descripción del esquema de representación de soluciones

Para representar las soluciones a los algoritmos utilizamos un vector<int> de “n” componentes , siendo “n” el número de objetos cuyo contenido serán 0 y 1.

El “0” en una determinada posición del vector indica que el objeto correspondiente a esa posición no ha sido seleccionado y el “1” indicará todo lo contrario , que el objeto fue seleccionado.

2.2. Operadores comunes

Las acciones que hacen en común los algoritmos son algunas como:

-Sumar beneficios de pares de objetos

Esto se realizará en el momento de incorporar un nuevo objeto a la solución en el caso de greddy, para poder determinar el incremento de beneficio que este produce en la solución si lo utilizamos

En el caso de Búsqueda Local para saber cual es el beneficio que nos ha aportado un vecino.

-Ver el espacio disponible en la mochila

Para determinar qué objetos de los todavía no seleccionados caben en la mochila.

2.3. Función objetivo

El objetivo es conseguir la solución con mejor fitness posible , es decir aquella cuyo beneficio total sea máximo.

$$\max\left(\sum_{i=1}^{n} p_i \cdot x_i + \sum_{i=1}^{n} \sum_{j=1,\, j\neq i}^{n} p_{ij} \cdot x_i \cdot x_j\right)$$

Para ello es necesario el uso de la función ***beneficioProducidoSolucion*** , esta función devuelve un entero el cual será la sumatoria de todos los beneficios individuales de los objetos seleccionados más la todos los beneficios a pares entre los objetos que hay en la solución.

La función **beneficioProducidoSolucion** , será útil ,

En el caso de greedy para saber cual es el fitness de la solución que el algoritmo ha generado.
En el caso de búsqueda local será útil , aparte de para saber el fitness de la solución final , para saber si las soluciones que se van generando por mutaciones ,son mejores que la que tenemos por el momento.

```
-----------------------------------------------------------------------------------------
Función beneficioProducidoSolucion(vector_beneficio, matriz_beneficios, vector_peso,
solucion):

    beneficio = 0
    // Calcular el beneficio total sumando los beneficios de los objetos seleccionados
    Para cada objeto1:
        Si el objeto1 está en la solución:
            beneficio += beneficio individual objeto1
            Para cada objeto2:
                Si objeto2 esta en la solucion:
                    beneficio += beneficio del para objeto1 y objeto 2 //matriz_beneficios

    Devolver beneficio

-----------------------------------------------------------------------------------------
```

3. Descripción de los algoritmos de búsqueda implementados.

En este apartado explico los algoritmos implementados con el objetivo de conseguir un vector solución que maximice el beneficio y cumpla con la capacidad de la mochila.

3.1. Greedy

Este algoritmo devuelve un vector el cual será la solución, para rellenar este vector el primer paso será elegir el objeto el cual tenga mayor beneficio individual y quepa en la mochila.

Una vez elegido el objeto con mayor beneficio individual , se actualizará el espacio disponible en la mochila.

Una vez actualizado el espacio disponible de la mochila , se descartarán los objetos cuyos pesos son mayores que el espacio disponible.

A continuación iremos metiendo los objetos restantes según el beneficio que estos puedan dar a la solución actual , de modo que meteremos el objeto que más incrementa el beneficio total de la solución, tomando en cuenta tanto el beneficio individual , como el beneficio a pares con los objetos que ya están en la mochila.

Siempre que se incorpora un nuevo objeto a la solución , el algoritmo actualizará nuevamente el espacio disponible de la mochila y descartará los objetos que tras la incorporación del nuevo objeto ya no tengan espacio en la mochila.

```
Función Greedy(numero_objetos, capacidad_mochila, vector_beneficio, vector_peso,
matriz_beneficios):

    solucion = Vector de tamaño igual a la longitud de vector_peso, inicializado con ceros
    espacio_disponible = capacidad_mochila
    contador_disponibles = longitud de vector_peso
    posibles = Vector de tamaño igual a la longitud de vector_peso, inicializado con unos

    beneficio_mayor = 0
    indice_beneficio_mayor = -1
    existe_objeto = Falso

    // Buscar el objeto con el mayor beneficio por unidad de peso que quepa en la mochila

    Para cada objeto:

        Si el objeto es posible:

            Calcular beneficio_mayor_actual como el beneficio del objeto dividido por su
peso

            Si beneficio_mayor_actual es mayor o igual que beneficio_mayor y el peso del
            objeto cabe en la mochila disponible:

               Actualizar beneficio_mayor, indice_beneficio_mayor y existe_objeto

    // Si se encontró un objeto válido, añadirlo a la solución

    Si existe_objeto:
        Decrementar contador_disponibles
        Marcar el objeto como seleccionado en la solución   //Meter un 1
       Desmarcar el objeto en el vector posibles           //Meter un 0
        Actualizar espacio_disponible

    // Continuar seleccionando objetos mientras queden disponibles y espacio en la mochila

    Mientras contador_disponibles no sea cero:

        mayor_valor_potencial = 0.0
        indice_mayor_valor_potencial = -1

        // Desmarcar objetos que no quepan en la mochila

        Para cada objeto:

            Si el objeto no cabe en la mochila y está marcado como posible:
                Desmarcar el objeto del vector posibles y decrementar contador_disponibles

        // Si aún quedan objetos posibles, buscar el que tenga mayor "valor potencial"
        Si contador_disponibles no es cero:

           _valor_potencial = 0.0
```

```
        Para cada objeto1:
           Si el objeto1 es posible:
               //Calcular el _valor_potencial del objeto1
                _valor_potencial se inicializa con el beneficio individual del objeto

                Para cada objeto2:

                    Si el objeto2 está en la solución


                           _valor_potencial += beneficio del par objeto1 y objeto2


                _valor_potencia /= peso objeto1
                 Si el _valor_potencial es mayor que mayor_valor_potencial:
                     Actualizar mayor_valor_potencial e indice_mayor_valor_potencial

        // Añadir el objeto con el mayor "valor potencial" a la solución

        Decrementar contador_disponibles
        Marcar el objeto como seleccionado en la solución    //Meter un 1
       Desmarcar el objeto en el vector posibles            //Meter un 0
        Actualizar espacio_disponible


Devolver solucion
```

3.2. Búsqueda local

Este algoritmo devuelve un vector el cual será la solución , está solución será al inicio elegida de manera aleatoria.
De esta primera solución generamos todos sus posibles vecinos , los cuales iremos probando de una manera aleatoria si consiguen una mejor solución.

En el caso de que se consiga una mejor solución con algún vecino , la solución será actualizada y se repite exactamente el mismo proceso.

Si no se encuentra ninguna mejor solución, la solución actual se mantendrá y será la que devuelva el algoritmo.

FUNCIONES USADAS _____

```
//Calcula de dos soluciones cual es mejor
Función esMejor(solucion, solucion_mejor, vector_peso, vector_beneficio, matriz_beneficios,
capacidad_mochila):

       //Utiliza la función "beneficioProducidoSolucion"
       beneficio_solucion = beneficioProducidoSolucion(vector_beneficio,
       matriz_beneficios, vector_peso, solucion)
       beneficio_mejor_solucion = beneficioProducidoSolucion(vector_beneficio,
       matriz_beneficios, vector_peso, solucion_mejor)

       Si beneficio_mejor_solucion > beneficio_solucion:
              Devolver Verdadero
       Sino:
              Devolver Falso
```

```
-----------------------------------------------------------------------------------------

Función generarVecinos(solucion):

    vecinos = vector de vectores
    Para cada objeto1:
        Si solución[indiceObjeto1] es igual a 1:           //si objeto1 esta en la solucion
            Para cada objeto2:
                Si solucion[indiceObjeto2] es igual a 0:   //si objeto2 no esta en la
solucion
                    vecino = vector
                    Agregar indiceObjeto1 al vecino
                    Agregar indiceObjeto2 al vecino
                    Agregar vecino a vecinos

    Devolver vecinos
-----------------------------------------------------------------------------------------


Función generarSolucionAleatoria(_vector_peso, _capacidad_mochila):

    espacio_disponible = _capacidad_mochila
    //que esté inicializada a 1 indica que todos los elementos están disponibles
inicialmente
    elementosDisponibles = vector con tamaño igual al número de objetos , inicializado a 1
    numeroObjetosDisponibles = longitud de elementosDisponibles
    solucion = vector con tamaño igual al número de objetos , inicializado a 0

    Mientras numeroObjetosDisponibles no sea igual a 0:

        Para cada objeto:
              Si elementosDisponibles[indiceObjeto] es igual a 1 y
              _vector_peso[indiceObjeto] es mayor que espacio_disponible:
                       elementosDisponibles[indiceObjeto] = 0
                       Disminuir numeroObjetosDisponibles en 1

        Si numeroObjetosDisponibles es mayor que 0:

            encontrado = falso
            elementoAleatorio = -1

            Mientras !encontrado :

                elementoAleatorio = generarNúmeroAleatorio entre 0 y número objetos
                Si elementosDisponibles[elementoAleatorio] es igual a 1:
                    encontrado = verdadero

            elementosDisponibles[elementoAleatorio] = 0
            solucion[elementoAleatorio] = 1
            Disminuir numeroObjetosDisponibles en 1
            espacio_disponible -= _vector_peso[elementoAleatorio]

    Devolver solucion
```

```
Función busquedaLocal(_vector_peso, _vector_beneficio, _matriz_beneficios,
_capacidad_mochila):

    //utilizamos función "generarSolucionAleatoria"

    solucion = generarSolucionAleatoria(_vector_peso, _capacidad_mochila)
    contadorIteraciones = 0
    contadorVecinosExplorar = 0
    posible_vecino = verdadero
    vecinos = vector de vectores

    Mientras contadorIteraciones sea menor que 90000 y posible_vecino sea verdadero:
        espacio_disponible = _capacidad_mochila
        solucionCandidata = solucion

       //Esto se producirá en la solución inicial y cuando encontremos una solución mejor.
        Si contadorVecinosExplorar es igual a 0:
            vecinos = generarVecinos(solucionCandidata)
            mezclar(vecinos)
       //Probamos un vecino aleatorio
        solucionCandidata[vecinos[contadorVecinosExplorar][0]] = 0
        solucionCandidata[vecinos[contadorVecinosExplorar][1]] = 1

       //Comprobamos el espacio de la mochila
        Para cada objeto:
            Si objeto está en solucionCandidata:   //solucionCandidata[indiceObjeto] = 1
                espacio_disponible -= peso de objeto

       //Si conseguimos mejor solución actualizamos y esta solución no excede la capacidad
       de la mochila
       Si esMejor(solucion, solucionCandidata, _vector_beneficio, _vector_peso,
       _matriz_beneficios, _capacidad_mochila) es verdadero y espacio_disponible es mayor
       o igual a 0:
            solucion = solucionCandidata
            contadorIteraciones = 0
            contadorVecinosExplorar = 0

       //En el caso de que la solución no sea mejor y no queden más vecinos
        Sino, si contadorVecinosExplorar es igual a la longitud de vecinos - 1:
            posible_vecino = falso

       //Si la solución no es mejor pero quedan vecinos disponibles
        Sino:
            Incrementar contadorIteraciones en 1
            Incrementar contadorVecinosExplorar en 1


    Devolver solucion
```

---

4. Procedimiento Considerado para desarrollar la práctica.

Esta práctica ha sido realizada utilizando C++ , he utilizado algunas librerías como iostream , string , vector , chrono ,ctime ..etc. También , se ha hecho uso del archivo random.hpp , el cual nos facilita el trabajo con la aleatoriedad.

En cuanto a la estructura utilizada para la organización de archivos , toda la práctica se encuentra en una carpeta raíz llamada “software”, sobre la que tenemos a su vez dos principales directorios “BIN” y “FUENTES”.

En la carpeta “BIN” encontramos los ejecutables de la práctica , una carpeta “data_QKP_2324/data” que contiene todos los fichero con los que trabajamos y un fichero de texto “LEEME.txt” que contiene información útil acerca del contenido de la carpeta.

En la carpeta “FUENTES” donde tenemos los programas “mochilaCuadratica1.cpp” y “random.cpp”.

No se hace uso de ningún framework de metaheurística.

PROCEDIMIENTO:

1) Nos situamos en **software**
2) Ejecutamos la siguiente orden :
   g++ FUENTES/mochilaCuadratica1.cpp -o /BIN/mochilaCuadratica1EX
3) Nos situamos en la carpeta **software/BIN** y lanzamos el ejecutable.
   ./mochilaCuadratica1EX data_QKP_2324/data/jeu* algoritmo semilla
   En LÉEME viene explicado como lanzar el ejecutable a detalle.

5. Experimentos y análisis de resultados

Ejecutamos ambos algoritmos sobre el conjunto de ficheros de datos. Para ello hacemos uso del scripts “BIN/creadorEjecutables.sh” , a este debemos pasarle dos argumentos:

> **1º Argumento**: Este debe ser “1” , que indica que el algoritmo que vamos a aplicar es greddy ,o “2” indicando que el algoritmo es Búsqueda Local.
>
> **2º Argumento:** Semilla

Este scripts , ejecutará “BIN/mochilaCuadratica1EX” con los 97 ficheros de datos , originando un archivo de texto cuyo contenido será los valores fitness y tiempo para cada una de las ejecuciones , también contendrá una media tanto del tiempo como del fitness para cada grupo de ficheros(jeu_100_25 , jeu_100_50 , jeu_200_25 …etc).

Ejecutando este scripts para ambos algoritmos con **semilla 5 ,** obtenemos los siguientes resultados**:**

GREEDY

TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	1157833.10	0.0104538140
200	100	928801.70	0.0092428600
200	50	569572.80	0.0079513760
200	75	556901.70	0.0073863170
300	25	478162.00	0.0108229333
100	100	284351.88	0.0009197413
200	25	273157.77	0.0076015566
100	75	190104.40	0.0016783851
100	50	130794.60	0.0017315660
100	25	63089.90	0.0011824489

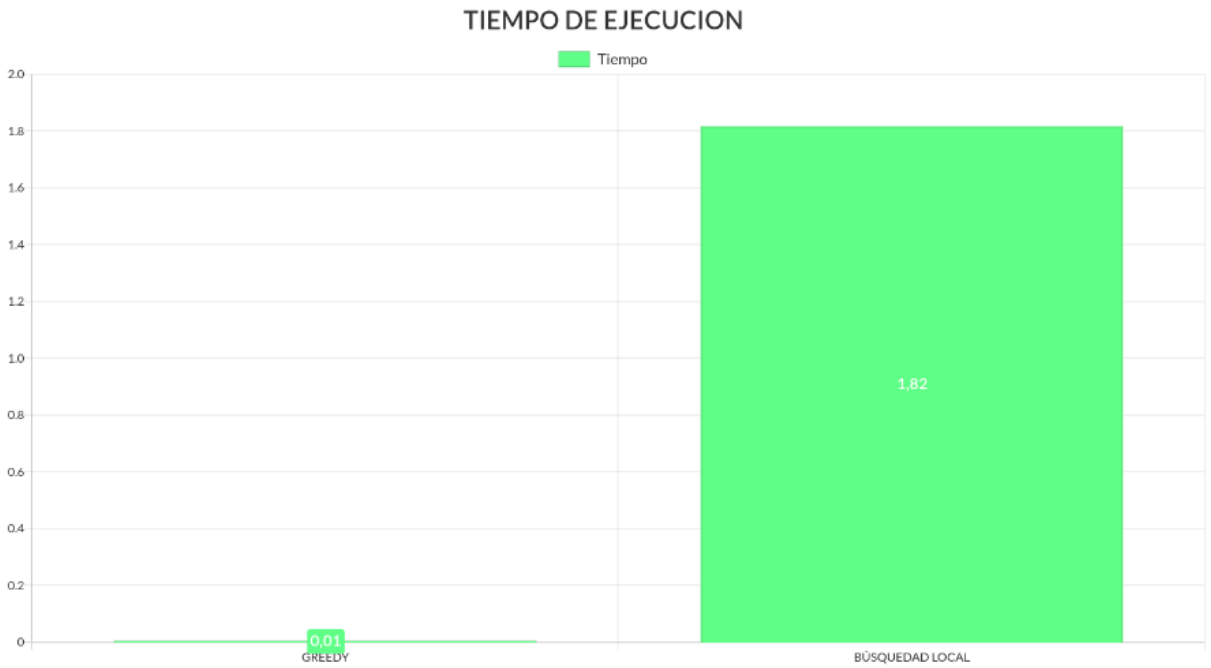
BÚSQUEDA LOCAL

TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	854460.70	6.8860650000
200	100	692132.10	0.8807273000
200	50	424004.30	1.5005247000
300	25	352148.44	6.6036467777
200	75	318623.90	1.1007971000
100	100	229897.11	0.0672666111
200	25	220062.66	0.8169880000
100	75	142945.00	0.1184443070
100	50	95127.90	0.1073719100
100	25	49322.40	0.0923596200

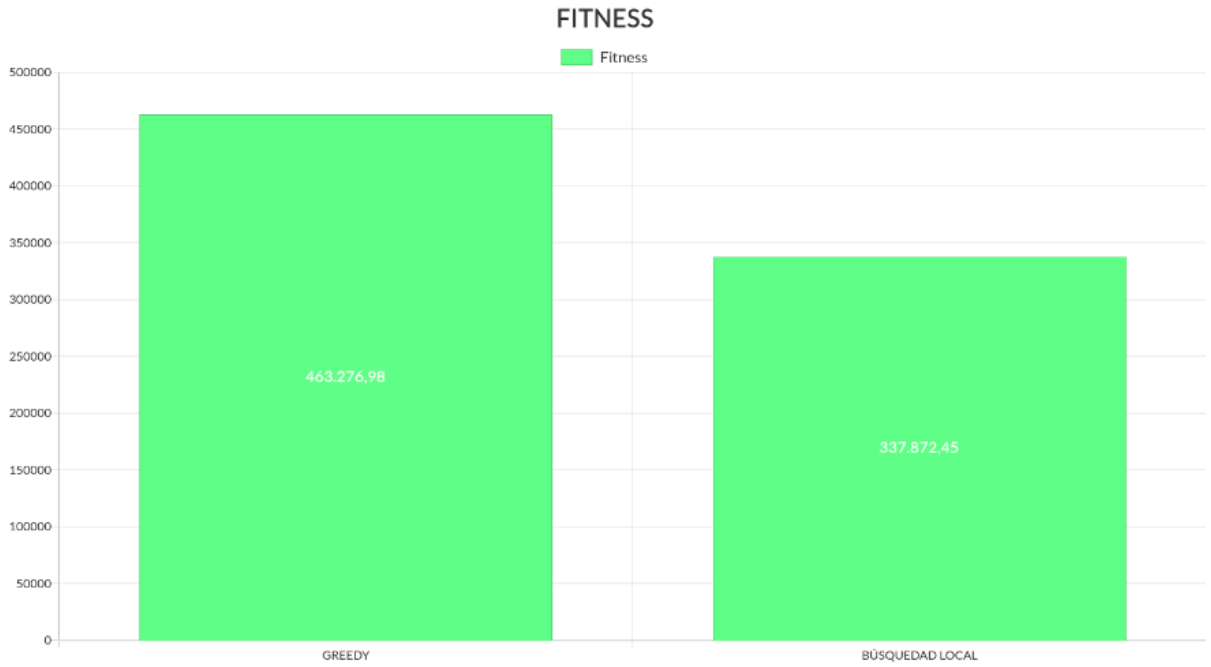
RESULTADOS GLOBALES

ALGORITMO	FITNESS	TIEMPO
GREEDY	463276,985	0.00589709982
BÚSQUEDA LOCAL	337872,451	1.81741913300

GRÁFICAS:



En cuanto al tiempo de ejecución podemos observar que Greedy es infinitamente mejor que Búsqueda Local .



En cuanto al fitness de media , se puede observar que Greedy , es también mejor que Búsqueda Local , a pesar de que la diferencia no es tan grande como pasa con el tiempo de ejecución.

En conclusión Greedy consigue obtener una mejor solución y en un tiempo mejor , esto se debe a que este algoritmo parte de una solución inicialmente vacía la cual nosotros vamos rellenando según los beneficios , lo que nos va a producir mayor rapidez y fitness , mientras que la Búsqueda local parte de una solución aleatoria , esto puede provocar que el algoritmo se quede atrapado en óptimos locales.
La búsqueda local solo puede mejorar llevando a cabo mutaciones , las cuales nos pueden llevar mucho tiempo para dar con la mejor posible solución.