Práctica 3:
Técnicas de Búsqueda basadas en Trayectorias para el Problema de la Mochila Cuadrático (QKP)

Curso 2023-24
Tercer Curso del Grado en Ingeniería Informática

Problema: Mochila Cuadrática
Algoritmos: BMB,ILS,ES,ILS-ES,BL

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

```
Función PesoTotal(_vector_peso, _solucion):
    peso = 0
    Para cada i desde de 0 a _solucion.size()-1 hacer:
        Si _solucion[i]==1 entonces:
            peso += _vector_peso[i]
    Devolver peso
```

```
Función generarVecinos(solucion):

    vecinos = [] //Vector de vectores
    Para cada i desde de 0 a solucion.size()-1 hacer:
        Si solución[i] == 1 entonces:              //si objeto i  esta en la solucion
            Para cada j desde de 0 a solucion.size()-1 hacer:
                Si solucion[j] == 0:               //si objeto2 no esta en la solucion
                    vecino = []  //vector
                    vecino.push_back(i)
                    vecino.push_back(j)
                    vecinos.push_back(vecino)
Devolver vecinos
```

```
Función generarSolucionAleatoria(_vector_peso,_capacidad_mochila):

        espacio_disponible = _capacidad_mochila
        elementos_disponibles(_vector_peso.size(),1) //Inicializado todos los elementos a 1
        numero_objetos_disponibles = elementos_disponibles.size()
        solucion(_vector_peso.size(),0) //Inicializamos todos los elementos a 0

        Mientras numero_objetos_disponibles != 0 hacer:
            Para cada i desde de 0 a _vector_peso.size()-1 hacer:
                Si elementos_disponibles[i]== 1 y _vector_peso[i]>espacio_disponible
entonces:
                    elementos_disponibles[i] = 0
                    numero_objetos_disponibles = numero_objetos_disponibles - 1

            Si numero_objetos_disponibles > 0 entonces:
                encontrado = Falso
                elemento_aleatorio = -1

                Mientras !encontrado hacer:
                    elemento_aleatorio = get(0 ,_vector_peso.size()-1)
                    Si elementos_disponibles[elemento_aleatorio] == 1 entonces:
                        encontrado = Verdadero
```

```
                elementos_disponibles[elemento_aleatorio] = 0
                solucion[elemento_aleatorio] = 1
                numero_objetos_disponibles = numero_objetos_disponibles - 1
                espacio_disponible = espacio_disponible - _vector_peso[elemento_aleatorio]
    Devolver poblacion

————————————————————————————————————————————————————————————————————————
Función beneficioProducidoSolucion_Factorizado(_vector_beneficio, _matriz_beneficios,
_vector_peso, eliminado, asignado, beneficio, _solucion):

    beneficioAux = beneficio
    beneficioAux = beneficioAux - _vector_beneficio[eliminado]
    beneficioAux = beneficioAux + _vector_beneficio[asignado]

    Para cada i desde 0 hasta _vector_peso.size() - 1 hacer:

        Si _solucion[i] = 1 entonces:

            Si i no es igual a eliminado entonces:
                Si i < eliminado entonces:
                    beneficioAux = beneficioAux - (_matriz_beneficios[i][eliminado] * 2)
                Si i > eliminado entonces:
                    beneficioAux = beneficioAux - (_matriz_beneficios[eliminado][i] * 2)

            Si i != asignado y i != eliminado entonces:
                Si i < asignado entonces:
                    beneficioAux = beneficioAux + (_matriz_beneficios[i][asignado] * 2)
                Si i >asignado entonces:
                    beneficioAux = beneficioAux + (_matriz_beneficios[asignado][i] * 2)

    Devolver beneficioAux

————————————————————————————————————————————————————————————————————————
Función PesoTotal_Factorizado(_vector_peso, eliminado, asignado, peso):

    pesoAux = peso
    pesoAux = pesoAux - _vector_peso[eliminado]
    pesoAux = pesoAux + _vector_peso[asignado]

    Devolver pesoAux
```

2.3. Búsqueda Local

Esta búsqueda local a diferencia de la implementada en la práctica 1 , realiza un máximo de 4500 iteraciones y parte desde una solución la cual le pasamos por argumento , esta solución será la que modificaremos.
Esta BI también le pasaremos un argumento por referencia “beneficio” , en el cual , tras la ejecución, quedará almacenado el beneficio de la solución resultado, este es un paso útil a la hora de mejorar la eficiencia de BMB e ILS , debido a que evitará tener que calcular el beneficio , tras cada ejecución de BL.
El método de exploración de vecinos consiste básicamente , en generar todos los vecinos e ir controlando mediante la variable “contadorVecinosExplorar” , que queden más vecinos sin explorar. En el caso en el que no queden más vecinos la BL parará .
La función que genera los vecinos , está indicada en el apartado anterior ,debido a que es un operador común para todos los algoritmos.

```
 Función busquedadLocal(_vector_peso, _vector_beneficio, _matriz_beneficios,
_capacidad_mochila, Solucion, beneficio):

            cromosoma = Solucion
            iteracionesAux = 0
            contadorVecinosExplorar = 0
            posible_vecino = Verdadero
            vecinos = []

            //Calculo benefico de la solución en la que estamos
            beneficio_actual = beneficioProducidoSolucion(_vector_beneficio,
        _matriz_beneficios, _vector_peso, cromosoma)
        //Itero sobre la solución respetando las iteraciones y controlando los vecinos
        visitados, en el caso de que ninguno de los vecinos mejore el se sale del bucle.
            Mientras iteracionesAux < 4500 y posible_vecino hacer:

                solucionCandidata = cromosoma
                espacio_disponible = _capacidad_mochila - PesoTotal(_vector_peso,
        cromosoma)
               //Este if es útil generar los vecinos de las nuevas soluciones
                Si contadorVecinosExplorar = 0 entonces:
                    vecinos = generarVecinos(solucionCandidata)
                    shuffle(vecinos)
               //producimos un vecino
                solucionCandidata[vecinos[contadorVecinosExplorar][0]] = 0
                solucionCandidata[vecinos[contadorVecinosExplorar][1]] = 1

               //Vemos el espacio disponible en la mochila del vecino
               espacio_disponible = espacio_disponible +
               _vector_peso[vecinos[contadorVecinosExplorar][0]]

               espacio_disponible = espacio_disponible -
               _vector_peso[vecinos[contadorVecinosExplorar][1]]

               //cálculo de manera eficiente del beneficio producido por el vecino
               beneficio_candidata =
               beneficioProducidoSolucion_Factorizado(_vector_beneficio,
               _matriz_beneficios, _vector_peso, vecinos[contadorVecinosExplorar][0],
               vecinos[contadorVecinosExplorar][1], beneficio_actual, cromosoma)

               //Vemos si el vecino es mejor y actualizamos en el caso de que lo sea
                Si beneficio_actual < beneficio_candidata y espacio_disponible >= 0
        entonces:
                    cromosoma = solucionCandidata
                    contadorVecinosExplorar = 0
                    iteracionesAux = iteracionesAux + 1
                    beneficio_actual = beneficio_candidata
               //Si no es mejor miramos si quedan vecinos, en el caso de no quedar más
        vecinos el bucle pararía
                Sino si contadorVecinosExplorar = vecinos.size() - 1 entonces:
                    posible_vecino = Falso
               //Aumenta el contador para explorar el siguiente vecino
                Sino:
                    contadorVecinosExplorar = contadorVecinosExplorar + 1
                    iteracionesAux = iteracionesAux + 1

            Solucion = cromosoma
            beneficio = beneficio_actual
```

2.4. 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 todos los beneficios a pares entre los objetos que hay en la solución.

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

    beneficio = 0
    // Calcular el beneficio total sumando los beneficios de los objetos seleccionados
    Para cada objeto i en solucion :
        Si solucion[i] == 1 entonces:
            beneficio += vector_beneficio[i]
            Para cada objeto j en solucion:
                Si solucion[j] == 1 entonces:
                    Si i<j entonces:
                     beneficio += matriz_beneficios[i][j]
                    Si i>j entonces:
                    beneficio += matriz_beneficios[j][i]

    Devolver beneficio
```

3. Algoritmo de comparación.

Greddy en la mochila cuadrática puede encontrar de manera rápida una solución cercana a la óptima sin realizar muchas iteraciones , por lo cual proporciona buenos tiempos de ejecución , esto se debe a la exploración reducida de vecinos.

Búsqueda local parte de una solución y dependiendo de esta lograremos obtener mejores o peores fitness , si partimos de una solución que no es buena , cabe la posibilidad de que BL se quede atrapada en óptimos locales impidiendo llegar a una solución cercana a la óptima , sin embargo , si partimos de una buena solución es mucho más probable que BL se acerque a una solución óptima.

En BL se puede dar el problema de que no consigamos vecinos que mejoren nuestra solución , esto provocará que BL termine y no podamos conseguir una solución que se acerque al óptimo.

AGG parte de una población (conjunto de soluciones) , sobre estas soluciones aplicamos mecanismos de selección , cruce y mutación , estos mecanismos serán un punto clave para obtener mejores soluciones. Se selecciona un número de cromosomas igual al tamaño de la población(50) , sobre los seleccionados aplicamos cruces y mutaciones. Tras ser mutados y cruzados , se reemplaza el peor de los seleccionados , por el mejor de la solución antigua.
AGG es un algoritmo bastante costoso y no garantiza encontrar la solución óptima , sin embargo tiene la capacidad de explorar grandes espacios de soluciones y de escapar de los óptimos locales.

AGE parte también de una población , sobre la que también aplicaremos mecanismos de selección , cruce y mutación , a diferencia de AGG , en AGE se selecciona solo una parte de la población en cada generación (2 cromosomas) y sobre estos dos cromosomas seleccionados aplicaremos la mutación y el cruce. Tras realizar la mutación y el cruce , se realiza el reemplazo de las dos mejores soluciones entre las dos peores soluciones de la población y los dos cromosomas (ya mutados y cruzados).
AGE mantiene la población más estable debido a que explora el espacio de soluciones lentamente , esto tiene una ventaja debido a que en determinadas situaciones podremos evitar perder buenas soluciones , sin embargo cuando no enfrentamos a problemas que requieren de una exploración rápida , AGE no sería tan adecuado.

AMs son básicamente una mezcla de AGG y BL , es decir , de las 90000 iteraciones que van a hacer los AMs , un porcentaje serán realizadas con BL y otro porcentaje con AGG , es básicamente esto lo que provoca unos menores tiempos en los AMs , debido a la rapidez de BL , pero a su vez peores resultados que AGG , debido a que BL produce peores fitness que AGG.

Si comparamos los AMs entre ellos podemos observar que AM-ALL consigue peores resultados que AM-RAND y AM-BEST , esto es debido a que AM-ALL gasta muchas más iteraciones en BL que AM-RAND y AM-BEST , aparte muchas de estas iteraciones se van a destinar a mejorar cromosomas que ofrecen muy poco fitness, en AM-RAND podemos destinar también iteraciones en mejorar cromosomas que ofrecen muy poco fitness , pero a diferencia de AM-ALL es que AM-RAND va a aplicar BL sobre muchos menos cromosomas.
AM-BEST sin embargo es idóneo debido a que aplica BL de una manera controlada y sobre los mejores cromosomas , provocando de esta manera que las iteraciones que se realizan en BL sean útiles para conseguir mejores fitness invirtiendo menos iteraciones.

4. Descripción de los algoritmos de búsqueda implementados.

En este apartado explico los algoritmos implementado , con el objetivo de obtener un vector solución el cual maximice el beneficio y cumpla con la capacidad máxima de la mochila

4.1 ES

```
Funcion Cauchy(tkmenos1, temperaturaInicial, temperaturaFinal, MaxVecinos, iteraciones):

        B = (temperaturaInicial - temperaturaFinal) / ((iteraciones / MaxVecinos) *
        temperaturaInicial * temperaturaFinal)
        tk = tkmenos1 / (1 + B * tkmenos1)

    Devolver tk


-------------------------------------------------------------------------------------
Funcion ES(_vector_peso, _vector_beneficio, _matriz_beneficios, _capacidad_mochila):

            // Genero solución inicial y la almaceno en la mejor solución
            solucion = generarSolucionAleatoria(_vector_peso, _capacidad_mochila)
            mejor_solucion = solucion

            // Calculo el beneficio de la solución inicial
            beneficio = beneficioProducidoSolucion(_vector_beneficio, _matriz_beneficios,
        _vector_peso, solucion)
            beneficio_mejor_solucion = beneficio

            // Calculo el peso de la solución inicial
            peso = PesoTotal(_vector_peso, solucion)
            peso_mejor_solucion = peso

            // Inicializo la temperatura inicial y final
            temperaturaInicial = (0.1 * beneficio) / (-log(0.3))
            temperatura = temperaturaInicial
            temperaturaFinal = 0.001

            // Inicializo las variables de control
            evaluaciones = 0
            Nexitos = 1
            Nvecinos = 0
            MaxVecinos = 5 * _vector_peso.size()
            MaxExitos = 0.1 * MaxVecinos

            // Ajusto la temperatura final
            Mientras temperaturaFinal >= 0.001 hacer:
                temperaturaFinal = temperaturaFinal / 10

            contador = 0
            Mientras evaluaciones < 90000 y Nexitos > 0 hacer:

                Nexitos = 0
```

```
Nvecinos = 0
vecinos = []
contador = 0

// Bucle interno para explorar vecinos
Mientras Nexitos < MaxExitos y Nvecinos < MaxVecinos y evaluaciones < 90000
hacer:
            vecino = solucion

            Si contador = 0 entonces:
                vecinos = generarVecinos(solucion)
                shuffle(vecinos)

            vecino[vecinos[contador][0]] = 0
            vecino[vecinos[contador][1]] = 1

             beneficioVecino =
        beneficioProducidoSolucion_Factorizado(_vector_beneficio,
        _matriz_beneficios, _vector_peso, vecinos[contador][0],
        vecinos[contador][1], beneficio, solucion)

             pesoVecino =
        PesoTotal_Factorizado(_vector_peso, vecinos[contador][0],
        vecinos[contador][1], peso)

             incremento = beneficioVecino - beneficio
             Nvecinos = Nvecinos + 1
             evaluaciones = evaluaciones + 1
             contador = contador + 1

             Si (incremento > 0 o Random(0, 10) * 0.1 <= exp(incremento /
    temperatura)) y pesoVecino <= _capacidad_mochila entonces:

                          Nexitos = Nexitos + 1
                        //Inicializamos contador a 0 para que  genere los
                        vecinos de la nueva solución
                          contador = 0
                          solucion = vecino
                          beneficio = beneficioVecino
                          peso = pesoVecino

                          Si beneficioVecino > beneficio_mejor_solucion
        entonces:
                              mejor_solucion = solucion
                              beneficio_mejor_solucion = beneficioVecino
                              peso_mejor_solucion = pesoVecino

        //En el caso en el que se ha explorado todo el entorno , forzamos al
        algoritmo a terminar , para ello establecemos Vecinos = MaxVecinos
        para que salga del bucle interno y Nexitos = 0 para que salga del
        bucle externo

             Si contador >= vecinos.size() entonces:
                 Nexitos = 0
                 Nvecinos = MaxVecinos

FIN bucle interno
 // Actualizo la temperatura
temperatura = Cauchy(temperatura, temperaturaInicial, temperaturaFinal,
MaxVecinos, 90000)

FIN bucle externo
Devolver mejor_solucion
```

En mi código tengo otra función ES , que lleva a cabo exactamente le mismo procedimiento , pero tiene como tope de iteración 4500 y le pasamos por referencia un “vector<int>solucion” y un ” int beneficio” , donde al final de la ejecución quedarán almacenados los resultados , esta función es adaptada para su uso en ILS-ES.

4.2 BMB

```
Funcion BMB(_vector_peso, _vector_beneficio, _matriz_beneficios, _capacidad_mochila):

    mejor_beneficio = 0
    mejor_solucion(_vector_peso.size(),0) //vector inicializado a 0
    i = 0

    Mientras i < 20 hacer:

        beneficio = 0
        solucion = generarSolucionAleatoria(_vector_peso, _capacidad_mochila)

       busquedadLocal(_vector_peso, _vector_beneficio, _matriz_beneficios,
       _capacidad_mochila, solucion, beneficio)

        Si beneficio > mejor_beneficio entonces:
            mejor_beneficio = beneficio
            mejor_solucion = solucion

        i = i + 1

    Devolver mejor_solucion
```

4.3 ILS

```
Funcion mutacion(_vector_peso, _vector_beneficio, _matriz_beneficios, _capacidad_mochila,
&solucion):

        solucionAux = solucion
        elementoAleatorio = Random(0, _vector_peso.size() - 1)

        indices = []  //vector de enteros
        indicesAux = []  // vector de enteros

        //voy almacenando en indices e indicesAux para poder realizar al final la
asignación
        //Caso en el que no llega al final
        Si elementoAleatorio + 20 <= _vector_peso.size() entonces:
                i = elementoAleatorio
                Mientras i < elementoAleatorio + 20 hacer:
                        indices.push_back(i)
                        indicesAux.push_back(i)
                        i = i + 1

        //Caso en el que llega al final
        Sino:
                sobrantes = (elementoAleatorio + 19) - _vector_peso.size() - 1
                tope = 20 - sobrantes

                i = 0
                Mientras i < tope hacer:
                        indices.push_back(elementoAleatorio + i)
                        indicesAux.push_back(elementoAleatorio + i)
                        i = i + 1
```

```
                j = 0
                Mientras j < sobrantes hacer:
                        indices.push_back(j)
                        indicesAux.push_back(j)
                        j = j + 1


        //barajamos índices
        shuffle(indices)

        //Modifico la solución
        Para i desde 0 hasta indices.size() - 1 hacer:
                solucion[indicesAux[i]] = solucionAux[indices[i]]

        peso = PesoTotal(_vector_peso, solucion)

        Mientras peso > _capacidad_mochila hacer:
                elemento = Random(0, indicesAux.size() - 1)
                Si solucion[indicesAux[elemento]] = 1 entonces:
                        solucion[indicesAux[elemento]] = 0
                        peso = peso - _vector_peso[indicesAux[elemento]]
```

```
Funcion ILS(_vector_peso, _vector_beneficio, _matriz_beneficios, _capacidad_mochila):

        S = generarSolucionAleatoria(_vector_peso, _capacidad_mochila)
        mejor_beneficio = 0

        BL o ES

        mejor_solucion = S

        i = 1

        Mientras i < 20 hacer:

                solucion_mutada = mejor_solucion

                mutacion(_vector_peso, _vector_beneficio, _matriz_beneficios,
                _capacidad_mochila, solucion_mutada)

                beneficios_mutada = 0

                BL o ES

                Si beneficios_mutada > mejor_beneficio entonces:
                        mejor_beneficio = beneficios_mutada
                        mejor_solucion = solucion_mutada

                i = i + 1

        Retornar mejor_solucion
```

5. 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 , el scripts que automatiza la obtención de resultados, 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 la carpeta build , include(cabeceras) , src(.cpp) y CmakeList.txt

No se hace uso de ningún framework de metaheurística.

PROCEDIMIENTO :

1) Nos situamos en **software/FUENTES/build**
2) Ejecutamos la siguiente orden :
   make
   Si hemos modificado el CmakeList.txt , **hacemos cmake ..**
3) Nos situamos en la carpeta **software/BIN** y lanzamos el ejecutable.
   ./ejecutable_P3 data_QKP_2324/data/jeu* algoritmo semilla
   En LÉEME viene explicado como lanzar el ejecutable a detalle.

5. Experimentos y análisis de resultados

Ejecutamos todos los 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 indica el algoritmo que queremos usar y debe ser alguna de las siguientes opciones:

“1”--> BMB
“2”--> ILS
“3”--> ES
“4”--> BL
“5”--> ILS-ES

2º Argumento: Semilla

Este scripts , ejecutará “BIN/tiempo_fitness” 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 cada uno de los algoritmos con **semilla 15 ,** obtenemos los siguientes resultados**:**

BMB			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	911932.10	4.7701530000
200	100	736256.90	1.5179640000
200	50	448665.00	2.0399953000
300	25	385135.44	4.6635388888
200	75	378595.30	1.8244033000
100	100	252795.66	0.3243990000
200	25	230576.66	1.6635158888
100	75	160817.60	0.4128094400
100	50	110521.90	0.4645911000
100	25	57026.30	0.3900261000

ILS			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	867928.10	2.1092960000
200	100	684889.60	0.8732546000
200	50	412294.90	1.0474924000
300	25	363717.33	2.2928088888
200	75	320576.60	0.9986084000
100	100	228641.33	0.2459127555
200	25	216770.22	0.9562137777
100	75	147077.10	0.2953047500
100	50	100399.70	0.3536231000
100	25	51820.40	0.2749764900

ES			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	867515.80	0.7227124000
200	100	684030.70	0.1533654300
200	50	411476.00	0.2210917500
300	25	363428.11	0.6294267777
200	75	319291.30	0.1958359300
100	100	228481.11	0.0226568555
200	25	216164.55	0.1852056222
100	75	146545.20	0.0255466030
100	50	100072.30	0.0295895400
100	25	51474.00	0.0245080830

ILS-ES			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	866794.00	10.2605290000
200	100	684720.80	2.8101920000
200	50	412098.90	3.8295420000
300	25	362820.00	9.3682811111
200	75	320111.20	3.3515090000
100	100	228696.00	0.4368962222
200	25	216407.44	3.1432933333
100	75	146981.30	0.5085761000
100	50	100367.40	0.5791763000
100	25	51875.20	0.4774963000

BL			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	867912.60	0.9868479000
200	100	684721.30	0.1972789500
200	50	411940.40	0.2911162800
300	25	363679.33	0.8376455555
200	75	320055.50	0.2742201500
100	100	228495.88	0.0209653688
200	25	216330.22	0.2030477555
100	75	146604.50	0.0253812320
100	50	100162.20	0.0286739300
100	25	51685.80	0.0231427460

RESULTADOS GLOBALES

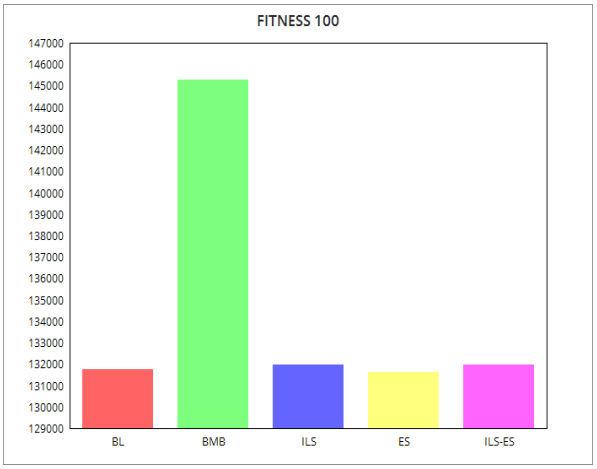
TAMAÑO 100		
ALGORITMO	FITNESS	TIEMPO
BL	131737.095	0.0245408192
AGG cruce propio	161029.305	0.6941639583
BMB	145290.365	0.39795641
ILS	131984.6325	0.2924542739
ES	131643.1525	0.02557527038
ILS-ES	131979.975	0.5005362305

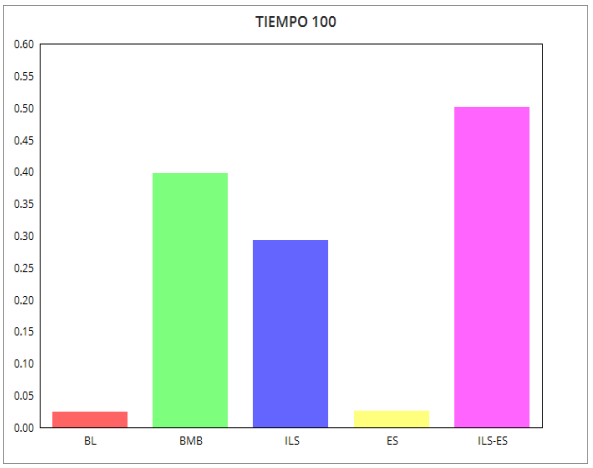
TAMAÑO 200		
ALGORITMO	**FITNESS**	**TIEMPO**
BL	**408261.855**	**0.2414157839**
AGG cruce propio	**545916.9675**	**2.037879056**
BMB	**448523.465**	**1.761469622**
ILS	**408632.83**	**0.9688922944**
ES	**407740.6375**	**0.188874683**
ILS-ES	**408334.585**	**3.283634083**

TAMAÑO 300		
ALGORITMO	**FITNESS**	**TIEMPO**
BL	**615795.965**	**0.9122467278**
AGG cruce propio	**758310.635**	**9.811861444**
BMB	**648533.77**	**4.716845944**
ILS	**615822.715**	**2.201052444**
ES	**615471.955**	**0.6760695889**
ILS-ES	**614807**	**9.814405056**

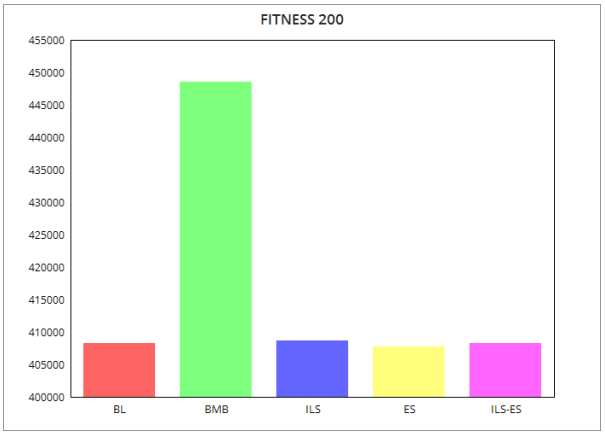
GRÁFICAS POR TAMAÑO

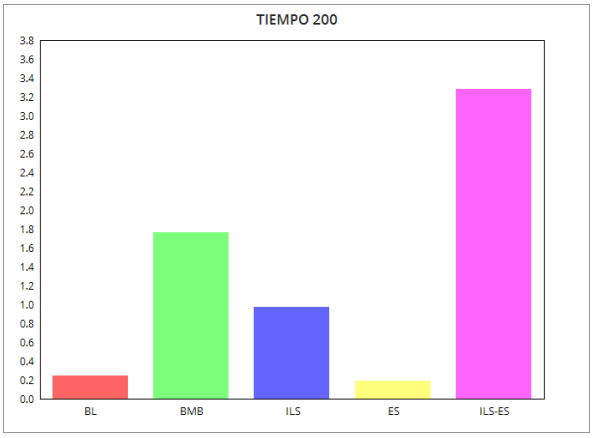
TAMAÑO 100:



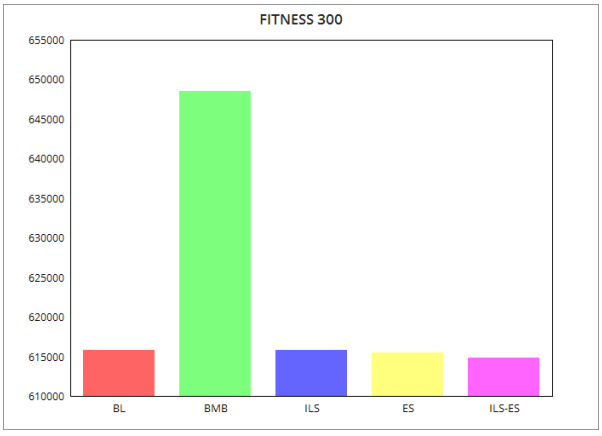


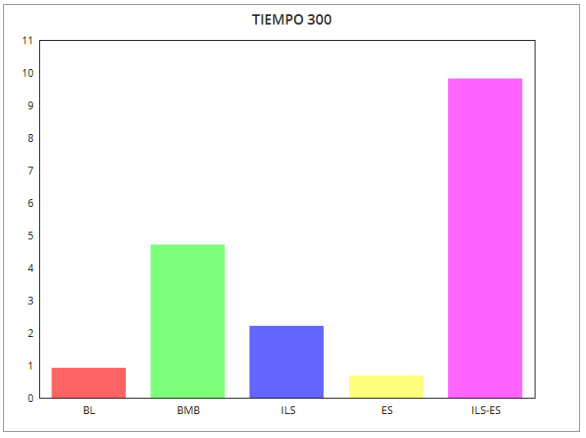
TAMAÑO 200:





TAMAÑO 300:





Observando las gráficas y tablas generadas , podemos determinar que no hay ningún algoritmo de esta práctica que supere al mejor de la práctica anterior(AGG con cruce propio).

Si hablamos de los algoritmos de esta práctica , observamos como BMB , es el que por diferencia ofrece mejor fitness , esto es debido a que este algoritmo aplica la exploración desde varias soluciones iniciales aleatorias , lo que nos permite explorar diversas regiones del espacio de búsqueda de una manera más global y sobre cada una de estas soluciones se aplica la BL y finalmente se queda con la mejor de todas. Todo esto provoca que BMB tenga menos probabilidad de quedarse atrapado en óptimos locales y más probabilidad de obtener una mejor solución.

Podemos observar cómo ILS , ofrece resultados bastante peores que BMB , esto es debido a que ILS genera una única solución inicial , por lo tanto cualquier exploración depende de esta y de la mutación que a esta se le aplique , pudiendo quedar atrapado en óptimos locales con mayor facilidad , si la mutación no provoca un cambio efectivo en la solución actual. Sin embargo , aunque es prácticamente inapreciable , ILS consigue mejores resultados que BL , esto se debe a que como he comentado anteriormente , ILS tiene la posibilidad de explorar diferentes regiones del espacio de soluciones gracias a las mutaciones.

Si comparamos ES con BL , debería dar mejores resultados ES , debido a que este tiene la capacidad de escapar con mayor facilidad de óptimos locales , sin embargo podemos observar en las gráficas y en las tablas , que a pesar de los resultados son prácticamente iguales , es BL el que ofrece mejores resultados.

Como consecuencia cuando aplicamos ILS con diferentes algoritmos (ES o BL) , tambien salen mejores resultados en ILS_BL que en ILS_ES , aunque en este caso los resultados también son prácticamente iguales , sin embargo los tiempo de ejecución de ILS_ES son mayores.