Práctica 2:
Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática

Curso 2023-24
Tercer Curso del Grado en Ingeniería Informática

Problema: Mochila Cuadrática
Algoritmos: AGG,AGE,AGM

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 y operadores similares

Comunes:

NOTA: Las siguientes funciones , reciben nombres diferentes según el fichero donde se encuentren , pero su contenido es el mismo:

Ejemplo : “**poblacionInicial**” , en AGG se llama “**poblacionInicial_AGG**” y en AGE se llama “**poblacionInicial_AGE**” , AM utiliza todas las funciones de AGG.

```
GENERA UNA POBLACIÓN DE 50 CROMOSOMAS-------------------------------------------------------

Función poblacionInicial(_capacidad_mochila, _vector_peso):

    i = 0
    poblacion = [] //Es un vector de vectores

    Mientras i < 50 hacer:

        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(),1) //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]

        poblacion.push_back(solucion)
        i = i + 1

    Devolver poblacion
```

CALCULA LOS BENEFICIOS DE LA POBLACIÓN----------

```
Función beneficiosPoblacion(_poblacion, _vector_beneficio, _matriz_beneficios,
_vector_peso):

    beneficios_poblacion = []  //Vector donde almacenamos los beneficios de cada cromosoma

    Para cada i desde de 0 a _poblacion.size()-1 hacer:

       beneficio = beneficioProducidoSolucion(_vector_beneficio, _matriz_beneficios,
       _vector_peso, _poblacion[i])

        beneficios_poblacion.push_back(beneficio)

    Devolver beneficios_poblacion
```

CALCULA EL PESO TOTAL DE UN CROMOSOMA----------

```
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
```

INDICA SI UN CROMOSOMA ES MEJOR QUE OTROS DOS (UTIL PARA SELECCIÓN)----------

```
Función esMejor(indice_solucion, indice_solucion1, indice_solucion2, beneficios_poblacion):

    beneficio_solucion = beneficios_poblacion[indice_solucion]
    beneficio_solucion1 = beneficios_poblacion[indice_solucion1]
    beneficio_solucion2 = beneficios_poblacion[indice_solucion2]

    Si beneficio_solucion >= que beneficio_solucion1 y beneficio_solucion
>=beneficio_solucion2 entonces:

        Devolver Verdadero

    Sino:

        Devolver Falso
```

Similares:

En este apartado , introduzco las funciones de selección , de mutación y de cruce , estas funciones no llegan a ser del todo iguales, por las siguientes razones:

- La principal razón es que mis funciones de selección ,mutación , iteran sobre la población al completo , y dependiendo del algoritmo que utilicemos , estas funciones iteran sobre la población de una manera u otra.
- Con el objetivo de optimizar el código , se crea un vector de beneficios de la población inicial , este vector nos es de utilidad para comparar el beneficio de

diferentes cromosomas de la población sin necesidad de tener que calcularlo nuevamente. Este vector lo tendremos que ir actualizando , en el caso de AGE solo será necesario en el momento del reemplazo , mientras que en AGG la actualización del vector de beneficios se llevará a cabo en la selección , en el cruce , en la mutación y en el reemplazo.

SELECCIÓN (En el caso de AGG devuelve un bool , que indica si se ha seleccionado el mejor de la población , en AGE no es necesario devolverlo)----------
El AGE , sería suprimiendo cosas de AGG, bastaría con lo señalado en azul

```
Función torneos_AGG(_poblacion, posicion_mejor_solucion, beneficios_poblacion):
        i = 0
        _poblacionAux = []  //Vector de vectores
        encontrado_mejor = Falso
        beneficios_poblacionAux = []  //vector

        Mientras i < 50 o 2 hacer: //En caso de AGG es 50 y en caso de AGE 2
                elementoAleatorio = get(0, _poblacion.size() - 1)
                elementoAleatorio1 = get(0, _poblacion.size() - 1)
                elementoAleatorio2 = get(0, _poblacion.size() - 1)
                //Aseguramos que los cromosomas sean distintos
                Si elementoAleatorio != elementoAleatorio1 y elementoAleatorio !=
                elementoAleatorio2 y elementoAleatorio1 != elementoAleatorio2 entonces:

                        Si esMejor_AGG(elementoAleatorio, elementoAleatorio1,
                        elementoAleatorio2, beneficios_poblacion) entonces:

                                _poblacionAux.push_back(_poblacion[elementoAleatorio])

                                Si _poblacion[elementoAleatorio] =
                                _poblacion[posicion_mejor_solucion] entonces:

                                        encontrado_mejor = Verdadero

                                beneficios_poblacionAux.push_back(beneficios_poblacion[element
                                oAleatorio])

                        Sino ,Si esMejor_AGG(elementoAleatorio1, elementoAleatorio,
                        elementoAleatorio2, beneficios_poblacion) entonces:

                                _poblacionAux.push_back(_poblacion[elementoAleatorio1])

                                Si _poblacion[elementoAleatorio1] =
                                _poblacion[posicion_mejor_solucion] entonces:

                                        encontrado_mejor = Verdadero

                                beneficios_poblacionAux.push_back(beneficios_poblacion[element
                                oAleatorio1])

                        Sino ,Si esMejor_AGG(elementoAleatorio2, elementoAleatorio1,
                        elementoAleatorio0, beneficios_poblacion) entonces:

                                _poblacionAux.push_back(_poblacion[elementoAleatorio2])

                                Si _poblacion[elementoAleatorio2]=
                                _poblacion[posicion_mejor_solucion] entonces:

                                        encontrado_mejor = Verdadero

                                beneficios_poblacionAux.push_back(beneficios_poblacion[element
                                oAleatorio2])
```

```
                i = i + 1
        _poblacion = _poblacionAux
        beneficios_poblacion = beneficios_poblacionAux
        Devolver encontrado_mejor
```

CRUCE_2_PUNTOS----------

El AGE , sería suprimiendo cosas de AGG, bastaría con lo señalado en azul

```
Función cruce_AGG(padre1, indiceP1, padre2, indiceP2, _capacidad_mochila,
_vector_beneficio, _vector_peso, _matriz_beneficios, beneficios_poblacion):

        elementoAleatorio = 0
        elementoAleatorio1 = 0
        mayor = 0
        menor = 0

        Mientras elementoAleatorio==elementoAleatorio1 hacer:
                elementoAleatorio = get(0,padre1.size()- 1)
                elementoAleatorio1 = get(0,padre1.size()- 1)

        Si elementoAleatorio > elementoAleatorio1 entonces:
                mayor = elementoAleatorio
                menor = elementoAleatorio1
        Sino, Si elementoAleatorio1 > elementoAleatorio entonces:
                mayor = elementoAleatorio1
                menor = elementoAleatorio

        Para cada índice i desde menor hasta mayor hacer:
                swap(padre1[i],padre2[i])

        Mientras PesoTotal_AGG(_vector_peso, padre1)> _capacidad_mochila hacer:
                elementoEliminar = get(0,padre1.size()- 1)
                padre1[elementoEliminar] = 0

        Mientras PesoTotal_AGG(_vector_peso, padre2) > _capacidad_mochila hacer:
                elementoEliminar = get(0,padre1.size()- 1)
                padre2[elementoEliminar] = 0

        beneficios_poblacion[indiceP1] = beneficioProducidoSolucion_AGG(_vector_beneficio,
        _matriz_beneficios, _vector_peso, padre1)

        beneficios_poblacion[indiceP2] = beneficioProducidoSolucion_AGG(_vector_beneficio,
        _matriz_beneficios, _vector_peso, padre2)

        Devolver {padre1, padre2}
```

FUNCIONES USADAS PARA EJECUTAR CRUCE EN AGG _____

```
Función cruces_AGG(_poblacion, _capacidad_mochila, _vector_beneficio, _vector_peso,
_matriz_beneficios, beneficios_poblacion):
            numero_parejas_cruces = (_poblacion.size() / 2) * 0.68
            i = 0
            j = 0

            Mientras i < numero_parejas_cruces hacer:

                cruce_producido = cruce_AGG(_poblacion[j], j, _poblacion[j+1], j+1,
                _capacidad_mochila, _vector_beneficio, _vector_peso, _matriz_beneficios,
                beneficios_poblacion)

                 i = i + 1
                 j = j + 2
```

MUTACIÓN----------

```
Función mutacion_AGG(_poblacion, _capacidad_mochila, _vector_beneficio, _vector_peso,
_matriz_beneficios, beneficios_poblacion):

        cromosomas = _poblacion.size()
        numero_mutaciones = cromosomas * 0.08
        i = 0

        Mientras i < numero_mutaciones hacer:
                indicePoblacion = get(0, _poblacion.size()- 1)
                mutacion_producida = _poblacion[indicePoblacion]
                encontrado = Falso
                mutacion_aux = mutacion_producida //para poder restablecer valor

                Mientras !encontrado hacer:

                        elementoAleatorio = get(0, _vector_peso.size()- 1)

                        Si mutacion_producida[elementoAleatorio]==0 entonces:
                                mutacion_producida[elementoAleatorio] = 1
                        Sino:
                                mutacion_producida[elementoAleatorio] = 0

                        Si PesoTotal_AGG(_vector_peso, mutacion_producida) <=
                        _capacidad_mochila entonces:
                                _poblacion[indicePoblacion] = mutacion_producida
                                encontrado = Verdadero
                                i = i + 1
                        Sino:
                                mutacion_producida = mutacion_aux


                beneficios_poblacion[indicePoblacion]=beneficioProducidoSolucion_AGG(_vector
                _beneficio, _matriz_beneficios, _vector_peso, _poblacion[indicePoblacion])

Función mutacion_AGG(_poblacion, _capacidad_mochila, _vector_peso):

        dado1 = get(0,1) //Dado asociado al primer cromosoma
        dado2 = get(0,1) //Dado asociado al segundo cromosoma

        Si dado1 < 0.08:
                mutacion_producida = _poblacion[0]
                encontrado = Falso
                mutacion_aux = mutacion_producida

                Mientras !encontrado hacer:

                        elementoAleatorio = get(0, _vector_peso.size()- 1)

                        Si mutacion_producida[elementoAleatorio]==0 entonces:
                                mutacion_producida[elementoAleatorio] = 1
                        Sino:
                                mutacion_producida[elementoAleatorio] = 0

                        Si PesoTotal_AGG(_vector_peso, mutacion_producida) <=
                        _capacidad_mochila entonces:
                                _poblacion[indicePoblacion] = mutacion_producida
                                encontrado = Verdadero
                        Sino:
                                mutacion_producida = mutacion_aux

        Si dado2 < 0.08:
                mutacion_producida = _poblacion[1]
                encontrado = Falso
                mutacion_aux = mutacion_producida
```

```
Mientras !encontrado hacer:

        elementoAleatorio = get(0, _vector_peso.size()- 1)

        Si mutacion_producida[elementoAleatorio]==0 entonces:
                mutacion_producida[elementoAleatorio] = 1
        Sino:
                mutacion_producida[elementoAleatorio] = 0

        Si PesoTotal_AGG(_vector_peso, mutacion_producida) <=
        _capacidad_mochila entonces:
                _poblacion[indicePoblacion] = mutacion_producida
                encontrado = Verdadero
        Sino:
                mutacion_producida = mutacion_aux
```

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

NOTA: Las siguiente función , recibe nombres diferentes según el fichero donde se encuentre, pero su contenido es el mismo:

En AGG se llama " **beneficioProducidoSolucion_AGG**" .
En AGE se llama "**beneficioProducidoSolucion_AGE**"
En BL se llama "**beneficioProducidoSolucion_BL**"
AM utiliza todas las funciones de AGG.

```
CÁLCULO FITNESS-------------------------------------------------------------------------------
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.

4. 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.

4.1. Búsqueda local

A partir del cromosoma pasado por parámetro 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 _____

```
Función PesoTotal_BL(_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_BL(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 busquedaLocal(_vector_peso, _vector_beneficio, _matriz_beneficios,
_capacidad_mochila, poblacion , posicion_cromosoma , beneficio_poblacion , iteraciones ,
iteraciones_globales):

    //utilizamos función "generarSolucionAleatoria"

    cromosoma = poblacion[posicion_cromosoma]
    iteracionesAux = 0
    contadorVecinosExplorar = 0
    posible_vecino = verdadero
    vecinos = [] //Vector de vectores

    Mientras iteracionesAux < _vector_peso.size() y (iteracionesAux + iteraciones_globales)
<     90000 y posible_vecino hacer:

            solucionCandidata = cromosoma
             espacio_disponible = _capacidad_mochila -
       PesoTotal_BL(_vector_peso,cromosma)


            //Esto se producirá en la solución inicial y cuando encontremos una solución
       mejor.
             Si contadorVecinosExplorar == 0:
                 vecinos = generarVecinos_BL(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 i desde de 0 a solucionCandidata.size()-1 hacer:
                 Si solucionCandidata[i] == 1 entonces:
                     espacio_disponible -= _vector_peso[i]

            //Si conseguimos mejor solución actualizamos y esta solución no excede la
            capacidad de la mochila

            beneficio_candidata =
            beneficioProducidoSolucion_BL(_vector_beneficio,_matriz_beneficios,_vector_p
            eso,solucionCandidata)


            Si beneficio_poblacion[posicion_cromosoma] < benefico_candidata y
            espacio_disponible >= 0:
                 cromosoma = solucionCandidata
                 contadorVecinosExplorar = 0
                 iteracionesAux = iteracionesAux +1
                 beneficio_poblacion[posicion_cromosoma] = beneficio_candidata
                 poblacion[posicion_cromosoma] = cromosoma

            //En el caso de que la solución no sea mejor y no queden más vecinos
             Sino, si contadorVecinosExplorar == vecinos.size()-1 entonces:
                 posible_vecino = falso

            //Si la solución no es mejor pero quedan vecinos disponibles
             Sino:
                 contadorVecinosExplorar = contadorVecinosExplorar + 1
                iteracionesAux = iteracionesAux+1
```

```
interacciones = iteracionesAux
```

4.2. Operador de cruce propuesto

El cruce propuesto , genera por cada gen un número aleatorio entre 1 y 10 , si este es menor que o igual a 5 , se cruzaría el gen entre los padres

El AGE , sería suprimiendo cosas de AGG, bastaría con lo señalado en azul

```
Función cruce_AGG_propuesto(padre1, indiceP1, padre2, indiceP2, _capacidad_mochila,
_vector_beneficio, _vector_peso, _matriz_beneficios, beneficios_poblacion):

        probabilidad = 5
        i = 0
        Mientras i < padre1.size() hacer:

                elementoAleatorio = get(1,10)

                Si elementoAleatorio <= probabilidad entonces:
                        swap(padre1[i],padre2[i])
                i++


        Mientras PesoTotal_AGG(_vector_peso, padre1)> _capacidad_mochila hacer:
                elementoEliminar = get(0,padre1.size()- 1)
                padre1[elementoEliminar] = 0

        Mientras PesoTotal_AGG(_vector_peso, padre2) > _capacidad_mochila hacer:
                elementoEliminar = get(0,padre1.size()- 1)
                padre2[elementoEliminar] = 0

        beneficios_poblacion[indiceP1] = beneficioProducidoSolucion_AGG(_vector_beneficio,
        _matriz_beneficios, _vector_peso, padre1)

        beneficios_poblacion[indiceP2] = beneficioProducidoSolucion_AGG(_vector_beneficio,
        _matriz_beneficios, _vector_peso, padre2)

        Devolver {padre1, padre2}
```

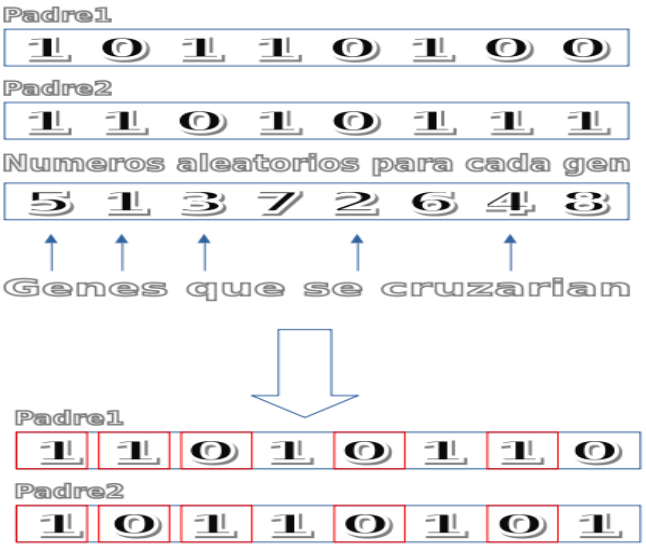
FUNCIONES USADAS PARA EJECUTAR CRUCE PROPUESTO EN AGG _____

```
Función cruces_AGG_propuesto(_poblacion, _capacidad_mochila, _vector_beneficio,
_vector_peso, _matriz_beneficios, beneficios_poblacion):
            numero_parejas_cruces = (_poblacion.size() / 2) * 0.68
            i = 0
            j = 0

            Mientras i < numero_parejas_cruces hacer:
```

```
cruce_producido = cruce_AGG_propuesto(_poblacion[j], j, _poblacion[j+1], j+1,
_capacidad_mochila, _vector_beneficio, _vector_peso, _matriz_beneficios,
beneficios_poblacion)

 i = i + 1
 j = j + 2
```



4.3. AGG

FUNCIONES USADAS _____

La función encontrarMejorSolucion , calcula cual es la posicion del cromosoma de la población , que produce un mejor fitness.

```
Función encontrarMejorSolucion_AGG(beneficios_poblacion)
        mejor = 0
        posicion = 0
        Para cada i desde 0 a beneficios_poblacion.size()-1 hacer:
                Si beneficios_poblacion[i] >= mejor
                        mejor = beneficios_poblacion[i]
                        posicion = i
        Devolver posicion
```

La función mejorSolucionNuevaPoblacion_AGG sirve para saber si existe un cromosoma en la nueva población que obtenga un mejor fitness que el mejor cromosoma de la antigua población.

```
Función mejorSolucionNuevaPoblacion_AGG(poblacionAux, poblacion, vector_beneficio,
vector_peso, matriz_beneficios, posicion_mejor_solucion, beneficios_poblacion):

        beneficioMejorSolucionAntigua = beneficioProducidoSolucion_AGG(vector_beneficio,
        matriz_beneficios, vector_peso, poblacionAux[posicion_mejor_solucion])

        Para cada i desde 0 a 50-1 hacer:
        Si beneficios_poblacion[i] >= beneficioMejorSolucionAntigua:
                Devolver verdadero
```

```
        Devolver falso
```

La función reemplazo_AGG busca el cromosoma de la población que produzca un menor fitness , y este es sustituido por el cromosoma que produce mejor fitness de la población Antigua , también actualiza el vector beneficios_poblacion , el cual almacena los fitness de cada uno de los cromosomas , para evitar calcularlo continuamente.

```
Funcion reemplazo_AGG(poblacionAux, poblacion, vector_beneficio, vector_peso,
matriz_beneficios, posicion_mejor_solucion, beneficios_poblacion):

        posicion_peor_solucion = -1
        peor_solucion = 9999999

        Para cada i desde 0 a poblacion.size()-1 hacer:
                beneficio = beneficios_poblacion[i]
                Si beneficio < peor_solucion:
                        peor_solucion = beneficio
                        posicion_peor_solucion = i
        //Actualizamos poblacion
        poblacion[posicion_peor_solucion] = poblacionAux[posicion_mejor_solucion]
        //Actualizamos beneficios de la población
        beneficios_poblacion[posicion_peor_solucion] =
        beneficioProducidoSolucion_AGG(vector_beneficio, matriz_beneficios, vector_peso,
        poblacion[posicion_peor_solucion])
```

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

        poblacion = poblacionInicial_AGG(capacidad_mochila, vector_peso)
        poblacionAux = poblacion
        beneficios_poblacion = beneficiosPoblacion_AGG(poblacion, vector_beneficio,
        matriz_beneficios, vector_peso)
        iteraciones = 0

        Mientras iteraciones < 90000:

                posición_mejor_solucion = encontrarMejorSolucion_AGG(beneficios_poblacion)
                encontrado = torneos_AGG(poblacion, posición_mejor_solucion,
                beneficios_poblacion)
                cruces_AGG(poblacion, capacidad_mochila, vector_beneficio, vector_peso,
                matriz_beneficios, beneficios_poblacion)
                mutacion_AGG(poblacion, capacidad_mochila, vector_beneficio, vector_peso,
                matriz_beneficios, beneficios_poblacion)
                conseguirMejorSolucion = mejorSolucionNuevaPoblacion_AGG(poblacionAux,
                poblacion, vector_beneficio, vector_peso, matriz_beneficios,
                posición_mejor_solucion, beneficios_poblacion)

                Si !encontrado y !conseguirMejorSolucion entonces:
                        reemplazo_AGG(poblacionAux, poblacion, vector_beneficio, vector_peso,
                        matriz_beneficios, posición_mejor_solucion, beneficios_poblacion)

                poblacionAux = poblacion
                iteraciones += 50


        beneficio = 0
        mejorSolucion = [] //Vector

        Para cada i desde de 0 a 50-1 hacer:
```

```
            beneficioSolucion = beneficios_poblacion[i]
            Si beneficioSolucion >= beneficio:
                   mejorSolucion = poblacion[i]
                   beneficio = beneficioSolucion

      Imprimir beneficio
      Devolver mejorSolucion
```

4.4. AGE

FUNCIONES USADAS _____

Esta función devuelve un vector de pares de vectores de enteros(solución) y entero(fitness), este resultado será útil para el reemplazo.

```
funcion mejorSolucionNuevaPoblacion_AGE(_poblacionAux, _poblacion, _vector_beneficio,
_vector_peso, _matriz_beneficios, beneficios_poblacion):
    aux = []  //vector
    vecSol = [] //vector de pares de vectores de enteros y enteros ,
vector<pair<vector<int>,int>>

    posicion_peor_solucion = -1
    posicion_segunda_peor_solucion = -1

    peor_solucion = 9999999
    segunda_peor_solucion = 9999999

//Encontramos peor solución
    Para cada i desde de 0 a poblacionAux.size()-1  hacer:
        beneficio = beneficios_poblacion[i]
        Si beneficio < peor_solucion entonces:
            peor_solucion = beneficio
            posicion_peor_solucion = i

//Encontramos segunda peor solución
    Para cada i desde de 0 a poblacionAux.size()-1  hacer:
        beneficio = beneficios_poblacion[i]
        si beneficio >= peor_solucion y beneficio < segunda_peor_solucion y i !=
       posicion_peor_solucion entonces:
            segunda_peor_solucion = beneficio
            posicion_segunda_peor_solucion = i

//Almacenamos en variables los beneficios de la peor solucion , de la segunda peor
solucion, y de los dos cromosomas de la poblacion que tenemos. Los dos cromosomas que
obtenemos de los torneos
    beneficioMejorSolucionAntigua1 = beneficios_poblacion[posicion_peor_solucion]
    beneficioMejorSolucionAntigua2 = beneficios_poblacion[posicion_segunda_peor_solucion]
    beneficioNuevaPoblacion1 = beneficioProducidoSolucion_AGE(_vector_beneficio,
_matriz_beneficios, _vector_peso, _poblacion[0])
    beneficioNuevaPoblacion2 = beneficioProducidoSolucion_AGE(_vector_beneficio,
_matriz_beneficios, _vector_peso, _poblacion[1])

//Almacenamos estos beneficios en un vector y ordenamos
    aux.push_back(beneficioMejorSolucionAntigua1)
    aux.push_back(beneficioMejorSolucionAntigua2)
    aux.push_back(beneficioNuevaPoblacion1)
    aux.push_back(beneficioNuevaPoblacion2)

    ordenar(aux, descendente)
//Booleanos propios de cada uno de los cromosomas , estos son útiles para en el caso de que
//dos de estos cromosomas tengan el mismo beneficio , se puedan identificar ambas
posiciones.
//Se puede ver el funcionamiento a continuación.
```

```
    encontradoSA1 = falso
    encontradoSA2 = falso
    encontradoNP1 = falso
    encontradoNP2 = falso

//Explicación del primer if:
       //Si el fitness más grande es igual a beneficioMejorSolucionAntigua1, inserto
       //en el vecSol , _poblacionAux[posicion_peor_solucion] (cromosma equivalente al
       beneficioMejorSolucionAntigua1 ) y la posición que tenemos que cambiar en la
       población (la peor)
    si aux[0] == beneficioMejorSolucionAntigua1 entonces:
        vecSol.push_back({_poblacionAux[posicion_peor_solucion], posicion_peor_solucion})
        encontradoSA1 = verdadero
        beneficios_poblacion[posicion_peor_solucion] = beneficioMejorSolucionAntigua1

    sino si aux[0] == beneficioMejorSolucionAntigua2 entonces:
       vecSol.push_back({_poblacionAux[posicion_segunda_peor_solucion],
       posicion_peor_solucion})
        encontradoSA2 = verdadero
        beneficios_poblacion[posicion_peor_solucion] = beneficioMejorSolucionAntigua2

    sino si aux[0] == beneficioNuevaPoblacion1 entonces:
        vecSol.push_back({_poblacion[0], posicion_peor_solucion})
        encontradoNP1 = verdadero
        beneficios_poblacion[posicion_peor_solucion] = beneficioNuevaPoblacion1

    sino si aux[0] == beneficioNuevaPoblacion2 entonces:
        vecSol.push_back({_poblacion[1], posicion_peor_solucion})
        encontradoNP2 = verdadero
        beneficios_poblacion[posicion_peor_solucion] = beneficioNuevaPoblacion2


    si aux[1] == beneficioMejorSolucionAntigua1 y !encontradoSA1 entonces:
       vecSol.push_back({_poblacionAux[posicion_peor_solucion],
       posicion_segunda_peor_solucion})
        beneficios_poblacion[posicion_segunda_peor_solucion] =
beneficioMejorSolucionAntigua1

    sino si aux[1] == beneficioMejorSolucionAntigua2 y !encontradoSA2 entonces:
       vecSol.push_back({_poblacionAux[posicion_segunda_peor_solucion],
       posicion_segunda_peor_solucion})
        beneficios_poblacion[posicion_segunda_peor_solucion] =
beneficioMejorSolucionAntigua2

    sino si aux[1] == beneficioNuevaPoblacion1 y !encontradoNP1 entonces:
        vecSol.push_back({_poblacion[0], posicion_segunda_peor_solucion})
        beneficios_poblacion[posicion_segunda_peor_solucion] = beneficioNuevaPoblacion1

    sino si aux[1] == beneficioNuevaPoblacion2 y !encontradoNP2 entonces:
        vecSol.push_back({_poblacion[1], posicion_segunda_peor_solucion})
        beneficios_poblacion[posicion_segunda_peor_solucion] = beneficioNuevaPoblacion2


    retornar vecSol
```

Esta función reemplaza en la población según el argumento datos , el cual es producido por la función anterior .

```
funcion reemplazo_AGE(_poblacionAux, _poblacion, datos):
    _poblacionAux[datos[0].segundo] = datos[0].primero
    _poblacionAux[datos[1].segundo] = datos[1].primero
```

```
funcion aplicarAlgoritmo_AGE(_numero_objetos, _capacidad_mochila, _vector_beneficio,
_vector_peso, _matriz_beneficios):

        poblacion = poblacionInicial_AGE(_capacidad_mochila, _vector_peso)
        poblacionAux = poblacion

        beneficios_poblacion = beneficiosPoblacion_AGE(poblacion, _vector_beneficio,
        _matriz_beneficios, _vector_peso)

        iteraciones = 0
        mientras iteraciones < 90000 hacer:
                torneos_AGE(poblacion, beneficios_poblacion)
                cruce_AGE(poblacion[0], poblacion[1], _capacidad_mochila, _vector_peso)
                mutacion_AGE(poblacion, _capacidad_mochila, _vector_peso)
               vecSol = mejorSolucionNuevaPoblacion_AGE(poblacionAux, poblacion,
               _vector_beneficio, _vector_peso, _matriz_beneficios, beneficios_poblacion)
                reemplazo_AGE(poblacionAux, poblacion, vecSol)
                poblacion = poblacionAux
                iteraciones = iteraciones + 2


    beneficio = 0
        mejorSolucion = [] //Vector

        Para cada i desde de 0 a 50-1 hacer:
               beneficioSolucion = beneficios_poblacion[i]
               Si beneficioSolucion >= beneficio:
                      beneficio = beneficioSolucion
                      mejorSolucion = poblacion[i]

        Imprimir beneficio
        Devolver mejorSolucion
```

4.5. AM

Utiliza todas las funciones de AGG excepto una función necesaria en AGM-Best para poder sacar los mejores , su pseudocódigo se muestra a continuación y también se muestra la aplicación de cada uno de los algoritmos meméticos.

La función calcula la posición del cromosoma de la población que produce mayor fitness , descartando una determinada posición y condicionando a que el beneficio que encontremos sea menor o igual al último beneficio encontrado , para evitar que siempre coja el mismo cromosoma, el funcionamiento se puede observar en la aplicación de los algoritmos.

```
funcion encontrarMejorSolucion_AMM(_beneficios_poblacion,
posicion_excluida,beneficio_ultima_solucion):
    mejor = 0
    posicion = 0
    para i desde 0 a _beneficios_poblacion.size()- 1 hacer:
        si _beneficios_poblacion[i] >= mejor y _beneficios_poblacion[i] <=
        beneficio_ultima_solucion y i != posicion_excluida entonces:
             mejor = _beneficios_poblacion[i]
             posicion = i
    retornar posicion
```

EL algoritmo para aplicar AM-ALL , AM-BEST y AM-RAND , es muy parecido solo cambia el fragmento de búsqueda local , en el pseudocódigo , se indica las partes propias de cada algoritmo , el resto de código es igual.

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

        poblacion = poblacionInicial_AGG(capacidad_mochila, vector_peso)
        poblacionAux = poblacion
        beneficios_poblacion = beneficiosPoblacion_AGG(poblacion, vector_beneficio,
        matriz_beneficios, vector_peso)
        iteraciones = 0
        generacion = 0

        Mientras iteraciones < 90000:


//AM-ALL************************************INICIO*****************************************
                si generacion % 10 == 0 entonces:
                        para i desde 0 a 50-1 y iteraciones < 90000 hacer:

                                iteracionBL = 0
                                        //BL Rellena iteracionesBL
                                busquedadLocal(_vector_peso, _vector_beneficio,
                                _matriz_beneficios, _capacidad_mochila, poblacion, i,
                                beneficios_poblacion, iteracionBL, iteraciones)

                                iteraciones = iteraciones + iteracionBL
//AM-ALL************************************FIN*****************************************
//AM-RAND***********************************INICIO*****************************************
                si generacion % 10 == 0 entonces:
                        para i desde 0 a (poblacion.size()*0.1)-1 y iteraciones < 90000
hacer:

                                iteracionBL = 0
                                indice = Random::get(0,49)
                                         //BL Rellena iteracionesBL
                                busquedadLocal(_vector_peso, _vector_beneficio,
                                _matriz_beneficios, _capacidad_mochila, poblacion, indice,
                                beneficios_poblacion, iteracionBL, iteraciones)

                                iteraciones = iteraciones + iteracionBL
//AM-RAND***********************************FIN*****************************************
//AM-BEST***********************************INICIO*****************************************
                si generacion % 10 == 0 entonces:

                        posicion_excluida = 9999
                        beneficio_ultima_solucion = INFINITY
                        indicesMejores = [] //vector
                        para i desde 0 a (poblacion.size()*0.1)-1 hacer:


                                posicion_producida=encontrarMejorSolucion_AMM(beneficios_pobla
                                cion,posicion_excluida,beneficio_ultima_solucion);
                                indicesMejores.push_back(posicion_producida);
                                posicion_excluida = posicion_producida;
                                beneficio_ultima_solucion=
                                beneficio_poblacion[posicion_producida]
```

```
                    para i desde 0 a (poblacion.size()*0.1)-1 y iteraciones < 90000
hacer:

                            iteracionBL = 0
                                    //Voy metiendo a BL los indices de los mejores
                                    //BL Rellena iteracionesBL
                            busquedadLocal(_vector_peso, _vector_beneficio,
                            _matriz_beneficios, _capacidad_mochila, poblacion,
                            indicesMejores[i], beneficios_poblacion, iteracionBL,
                            iteraciones)

                            iteraciones = iteraciones + iteracionBL
//AM-BEST**********************************FIN*****************************************

            posición_mejor_solucion = encontrarMejorSolucion_AGG(beneficios_poblacion)
            encontrado = torneos_AGG(poblacion, posición_mejor_solucion,
            beneficios_poblacion)
            cruces_AGG(poblacion, capacidad_mochila, vector_beneficio, vector_peso,
            matriz_beneficios, beneficios_poblacion)
            mutacion_AGG(poblacion, capacidad_mochila, vector_beneficio, vector_peso,
            matriz_beneficios, beneficios_poblacion)
            conseguirMejorSolucion = mejorSolucionNuevaPoblacion_AGG(poblacionAux,
            poblacion, vector_beneficio, vector_peso, matriz_beneficios,
            posición_mejor_solucion, beneficios_poblacion)

            Si !encontrado y !conseguirMejorSolucion entonces:
                    reemplazo_AGG(poblacionAux, poblacion, vector_beneficio, vector_peso,
                    matriz_beneficios, posición_mejor_solucion, beneficios_poblacion)

            poblacionAux = poblacion
            iteraciones += 50
            generacion += 1


    beneficio = 0
    mejorSolucion = [] //Vector

    Para cada i desde de 0 a 50-1 hacer:
            beneficioSolucion = beneficios_poblacion[i]
            Si beneficioSolucion >= beneficio:
                    mejorSolucion = poblacion[i]
                    beneficio = beneficioSolucion

    Imprimir beneficio
    Devolver mejorSolucion
```

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_P2 data_QKP_2324/data/jeu* algoritmo semilla
   En LÉEME viene explicado como lanzar el ejecutable a detalle.

6. 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”--> AGG cruce dos puntos
“2”--> AGE cruce dos puntos
“3”--> AM-ALL
“4”--> AM-RAND
“5”--> AM-BEST
“6”--> AGG cruce propio
“7”--> AGE cruce propio

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**:**

AGG cruce dos puntos			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	1017143.00	9.8897800000
200	100	856960.70	1.8980771000
200	50	515066.20	2.1379580000
200	75	478432.90	1.7559780000
300	25	414663.22	9.5158544444
100	100	268171.66	0.6604705555
200	25	250223.11	2.0040300000
100	75	180964.60	0.6697568000
100	50	122264.90	0.6477283000
100	25	59084.00	0.6395402000

AGG cruce propuesto			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	1073592.50	10.0607040000
200	100	874514.30	2.0138240000
200	50	539859.50	2.1745470000
200	75	511065.30	1.8680730000
300	25	443028.77	9.5630188888
100	100	278290.22	0.7089573333
200	25	258228.77	2.0950722222
100	75	180388.30	0.7026645000
100	50	124552.30	0.6951761000
100	25	60886.40	0.6698579000

AGE cruce dos puntos			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	1011445.30	13.4191700000
200	100	846745.20	2.6957720000
200	50	501833.90	2.9482900000
200	75	493459.20	2.4920500000
300	25	404484.22	12.6761666666
100	100	269712.33	1.0747096666
200	25	243367.77	2.8325122222
100	75	173925.40	1.0417329000
100	50	116071.50	1.0330970000
100	25	58072.90	1.0425739000

AGE cruce propuesto			
TAMAÑO	**DENSIDAD**	**FITNESS**	**TIEMPO**
300	50	1073917.00	13.7696400000
200	100	888239.20	2.9147920000
200	50	534923.60	3.1605890000
200	75	511545.70	2.6667400000
300	25	434269.11	13.1483777777
100	100	275340.88	1.1594883333
200	25	259239.55	3.0473355555
100	75	185458.40	1.1452137000
100	50	123500.10	1.1107500000
100	25	61378.20	1.1087776000

AM-ALL			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	948748.30	5.0998990000
200	100	808144.50	2.3261860000
200	50	481292.00	2.7223290000
200	75	430465.60	2.1516770000
300	25	382445.88	4.9088700000
100	100	263113.55	0.9237883333
200	25	235283.33	2.4989555555
100	75	172676.00	0.9315107000
100	50	113978.80	0.9422513000
100	25	57426.50	0.9098117000

AM-RAND			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	981313.00	4.8157470000
200	100	826801.90	2.2095520000
200	50	499850.80	2.5521480000
200	75	461867.10	2.0342710000
300	25	402957.66	4.6909944444
100	100	272527.00	0.8420706666
200	25	244359.66	2.4027644444
100	75	175615.10	0.8164477000
100	50	118574.90	0.8315079000
100	25	58788.60	0.8041203000

AM-BEST			
TAMAÑO	DENSIDAD	FITNESS	TIEMPO
300	50	982885.40	4.8214400000
200	100	839252.50	2.2714160000
200	50	495647.10	2.5678010000
200	75	451748.40	2.0771950000
300	25	401484.66	4.6509311111
100	100	271273.22	0.8319294444
200	25	244590.33	2.4422588888
100	75	175846.60	0.7935253000
100	50	116466.70	0.8271937000
100	25	58178.60	0.7774032000

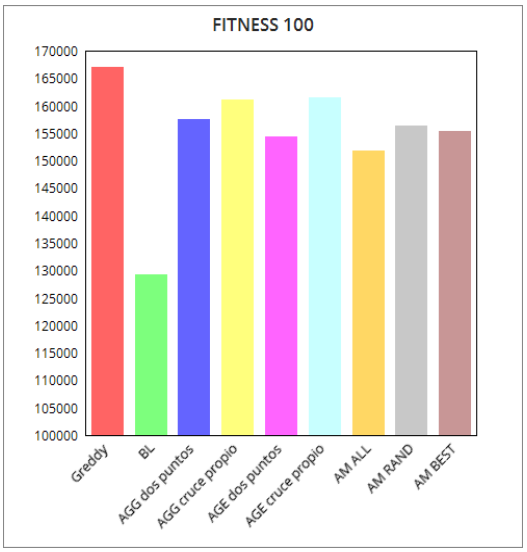
RESULTADOS GLOBALES

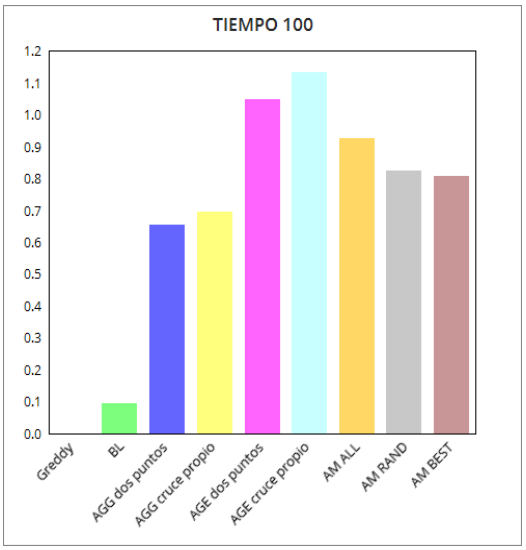
TAMAÑO 100		
ALGORITMO	FITNESS	TIEMPO
Greddy	167085.195	0.001378035325
BL	129323.1025	0.09636061203
AGG dos puntos	157621.29	0.6543739639
AGG cruce propio	161029.305	0.6941639583
AGE dos puntos	154445.5325	1.048028367
AGE cruce propio	161419.395	1.131057408
AM ALL	151798.7125	0.9268405083
AM RAND	156376.4	0.8235366416
AM BEST	155441.28	0.8075129111

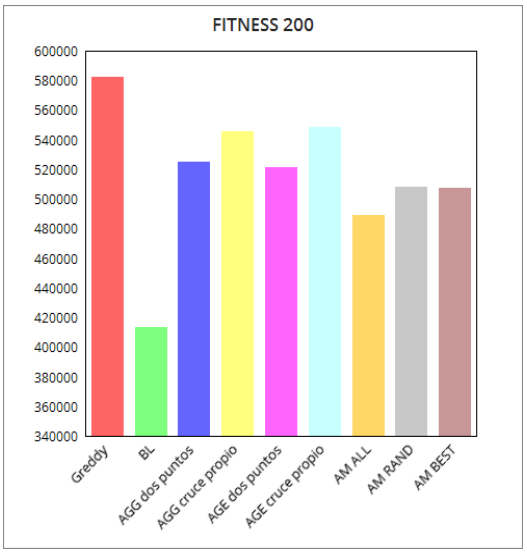
TAMAÑO 200		
ALGORITMO	**FITNESS**	**TIEMPO**
Greddy	582108.4925	0.0080455274
BL	413705.74	1.074759275
AGG dos puntos	525170.7275	1.949010775
AGG cruce propio	545916.9675	2.037879056
AGE dos puntos	521351.5175	2.742156056
AGE cruce propio	548487.0125	2.947364139
AM ALL	488796.3575	2.424786889
AM RAND	508219.865	2.299683861
AM BEST	507809.5825	2.339667722

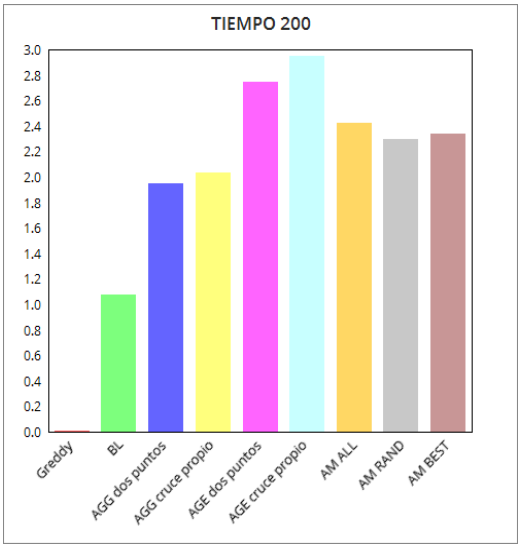
TAMAÑO 300		
ALGORITMO	**FITNESS**	**TIEMPO**
Greddy	817997.55	0.01063837365
BL	603304.57	6.744855889
AGG dos puntos	715903.11	9.702817222
AGG cruce propio	758310.635	9.811861444
AGE dos puntos	707964.76	13.04766833
AGE cruce propio	754093.055	13.45900889
AM ALL	665597.09	5.0043845
AM RAND	692135.33	4.753370722
AM BEST	692185.03	4.736185556

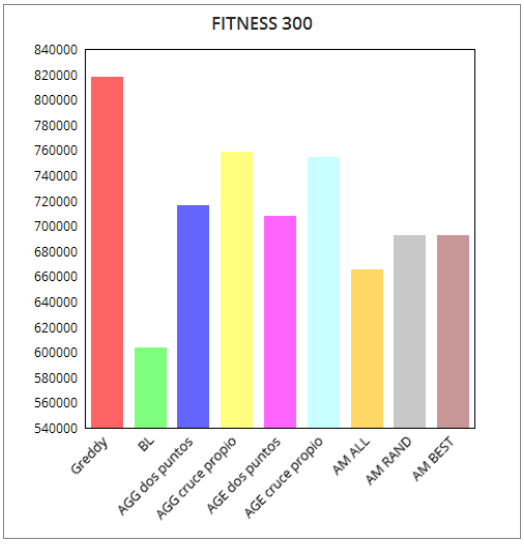
GRÁFICAS POR TAMAÑO

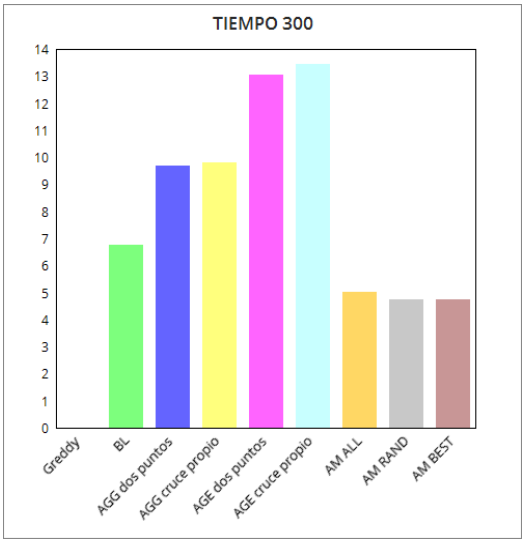




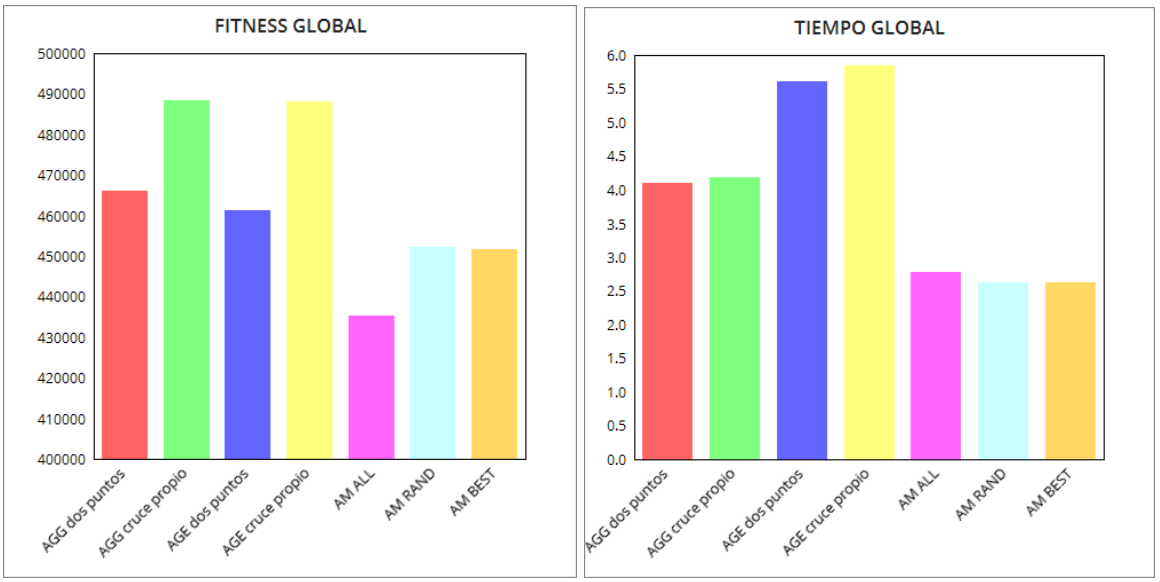








GRÁFICAS GLOBALES



Si observamos los gráficos individuales según el tamaño , no podemos ver realmente cómo se comportan los algoritmos , ante un conjunto de datos de cualquier tamaño.
Es por ello que he realizado estos gráficos globales donde podemos ver el comportamiento de los algoritmos con total claridad.

Partimos de la base de que Greddy produce mejor fitness que cualquier algoritmo realizado en esta práctica y que BL para cualquier tamaño es más rápido y produce menos fitness que AGG.

En este gráfico se puede observar que tanto AGG con cruce propio y AGE con cruce propio son los que mejores fitness obtienen , mientras que los AMs por lo general son los que menos fitness obtienen , a pesar de que estos fitness se aproximan a los de AGG con cruce de dos puntos.

Como ya sabemos los 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.

Si utilizamos el cruce propio , vemos que el fitness mejora ,pero a su vez los tiempos empeoran un poco , esto es debido a que el cruce de dos puntos solo necesita elegir dos números aleatorios para encontrar la parte que se quiere cruzar , mientras que mi cruce necesita producir tantos números aleatorios como genes existan para poder comparar estos números aleatorios con una probabilidad , produciendo de esta manera un mayor tiempo de ejecución.