Metaheurísticas

Alberto Estepa Fernández – 31014191W
albertoestep@correo.ugr.es
Gr. Prácticas MH2 - Jueves 17:30h
Estudiante del Doble Grado de Ingeniería Informática y Matemáticas

Práctica 2.a: Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Máxima Diversidad

2019/2020

UNIVERSIDAD
DE GRANADA

27 de abril de 2020

Índice

1. Descripción del problema …… 3
2. Descripción de la aplicación de los algoritmos empleados ……. 4
3. Descripción de la estructura de los métodos ……. 13
4. Procedimiento considerado para desarrollar la práctica …… 17
5. Experimentos y análisis de resultados ……19
6. Referencias bibliográficas ……. 29

Descripción del problema

El Problema de la Máxima Diversidad (Maximun Diversity Problem en inglés, MDP) es un problema de optimización combinatoria consistente en seleccionar un subconjunto M de m elementos (|M|=m) de un conjunto inicial S de n elementos (obviamente, n>m) de forma que se maximice la diversidad entre los elementos escogidos.

Además de los n elementos (e_i, i=1,...,n) y el número de elementos a seleccionar m, se dispone de una matriz D=(d_{ij}) de dimensión nxn que contiene las distancias entre ellos.

En este caso la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Matemáticamente el problema consistiría en maximizar la función:

$$\text{Maximizar} \quad z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Sujeto a} \quad \sum_{i=1}^{n} x_i = m$$

$$x_i = \{0, 1\}, \quad i = 1, \ldots, n.$$

donde x es el vector solución al problema.

Se utilizarán 30 casos seleccionados de varios de los conjuntos de instancias disponibles en la MDPLIB (http://www.optsicom.es/mdp/), 10 pertenecientes al grupo GKD con distancias Euclideas, n=500 y m=50 (GKD-c_11_n500_m50 a GKD-c_20_n500_m50), 10 del grupo MDG con distancias reales en [0,1000] , n=500 y m=50 (MDG-b_1_n500_m50 a MDG-b_10_n500_m50); y 10 del grupo MDG con distancias enteras en {0,10}, n=2000 y m=200 (MDG-a_31_n2000_m200 a MDG-a_40_n2000_m200).

Descripción de la aplicación de los algoritmos empleados

En este apartado describiremos las consideraciones comunes a los distintos algoritmos. Incluimos la representación de las soluciones y la función objetivo. Se ha usado C++ para la realización del código de la práctica.

A) Representación de las soluciones

Debido a que ahora estamos trabajando con técnicas de búsqueda basadas en poblaciones el esquema de representación de una solución la hemos basado en representación binaria en vez de representación en enteros. El esquema resultante es el siguiente:

```
struct solucion{
        vector<bool> vector;
        double coste;
        bool evaluada;
};
```

Aquí el vector 'vector' contiene un 1 en los índices de los puntos que forman la solución y un 0 en los demás (ahora es un vector de tamaño n en vez de tamaño m como en la práctica anterior). Los datos de donde hemos sacado las distancias se han guardado en una matriz simétrica de tamaño nxn. Por otro lado, hemos construido una clase para almacenar la población:

```
class poblacion{
        vector<solucion> vector;
        double max_coste;
        int mejor_solucion;
        int n_individuos;

        poblacion();
};
```

Esta clase permite guardar los individuos de la población en un instante dado, además guardamos también el índice en el vector de de soluciones de la mejor solución actualmente en la población (para evitar cálculos posteriormente), así como la diversidad de ésta solución.

En los algoritmos meméticos necesitaremos la representación dada por índices enteros, así que sólo en este caso y de forma momentánea usaremos la siguiente representación:

```
struct solucion_ints
        vector<int> vector;
        double coste;
};
```

B) Función objetivo

Se han mantenido las siguientes funciones, adaptándolas si fuese necesario a la representación binaria- Esta primera función calcula, dado un conjunto de elementos y un elemento concreto, la distancia acumulada entre el elemento y el cada uno de los elementos del conjunto.

Parámetros: elemento, conjunto y matriz de distancias
Resultado: suma acumulada de distancias del elemento al conjunto.
distanciaAUnConjunto*{*
    suma = 0
    Para cada i ∈ conjunto hacer:
        suma = matriz[elemento][i]
    Fin para cada
    Devolver suma
}

La siguiente función calcula, dado un conjunto de elementos que forma la solución, la distancia entre los elementos del conjunto, es decir, el coste de la solución.

Parámetros: conjunto_solución y matriz de distancias
Resultado: coste de la solución.
costeSolucion*{*
    coste = 0
    Para cada i ∈ conjunto hacer:
        coste = coste + distancia acumulada de i al conjunto_solución
    Fin para cada
    Devolver coste / 2
}

Aún así se han incorporado la funcione de evaluación de una solución para hacerlo más entendible, y permitir el evitar hacer cálculos innecesarios modificando la bandera ‘evaluada’ cuando sea necesario.

Parámetros: solución (s) y matriz de distancias
Resultado: calcula la diversidad entre los elemento de la solución y pone el flag evaluada a true.
evaluarSolucion*{*
    coste de s = costeSolucion()
    flag evaluada = true
    Devolver coste de s
}

C) Operadores comunes

La estructura de los algoritmos es común, es decir todas disponen de operadores de mutación, cruce, selección de individuos, inicialización de la población, reemplazo de la población y evaluación de la población. Aunque alguno de ellos presenta diferentes modificaciones según el algoritmo que implementemos. Vamos a incluir a continuación todas las diversas funciones que hemos usado y en el apartado siguiente solo distinguiremos la estructura de búsqueda de los algoritmos.

Inicialización de la población

Hemos realizado una inicialización aleatoria de la población para todos los algoritmos implementados.

Parámetros: población (p), número de individuos de p (n_i), número de elementos a seleccionar en una solución (n) y matriz de distancias
Resultado: población inicializada
inicializarPoblacion*{*
    Dimensionalizar el vector de p de soluciones a n_i
    PARA CADA i ∈ {0, número de individuos de p - 1} HACER:
        aplicar solucionAleatoria(p.vector[i], matriz_distancias, n)
    FIN PARA CADA
    número de individuos de p = n_i
}

Como vemos hemos usado la función siguiente para inicializar cada solución:

Parámetros: solución a rellenar (s), matriz de distancias (nxn), n.º de elementos a seleccionar en la solución (m)
Resultado: Rellena una solución con valores aleatorios.
soluciónAleatoria*{*
    número de elegidos = 0
    s.evaluada = false
    inicializar s.vector = vector de tamaño n a false todos los elementos
    MIENTRAS número de elegidos < m HACER:
        calcular número aleatorio
        SI s.vector[número aleatorio] no está elegido (está a false) HACER:
            s.vector[número aleatorio] = true
            número de elegidos = número de elegidos + 1
    FIN MIENTRAS
}

Básicamente rellenamos una solución seleccionando los elementos aleatoriamente hasta que sea factible dicha solución. Además modificamos los elementos de control como el flag ‘evaluada’.

Evaluación de la población

Hemos realizado una evaluación de la población común para todos los algoritmos implementados. Simplemente llamamos a la función objetivo de cada solución y actualizamos los datos si procede.

<u>Parámetros:</u> población (p), número de evaluaciones (n_v) y matriz de distancias.
<u>Resultado:</u> calculamos la diversidad de cada individuo y actualiza los valores de control de la pobl.
evaluarPoblacion*{*
    PARA CADA i ∈ {0, número de individuos de la p - 1} HACER:
        SI la solución i de p no está evaluada HACER:
            evaluarSolucion(·);
            n_v = n_v + 1
            SI diversidad de la solución i de p > máxima diversidad HACER:
                mejor_solución de p = i
                max_coste de p = diversidad de la solución i de p
            FIN SI
        FIN SI
    FIN PARA CADA
}

Selección de individuos

Hemos incluido dos versiones, una para algoritmos generacionales y otra para los estacionarios. Veamos primero la selección para algoritmos generacionales:

<u>Parámetros:</u> población donde elegiremos los individuos (p_a), población nueva resultante (p_n).
<u>Resultado:</u> Método de selección de individuos de la población.
seleccionarIndividuos*{ //Caso generacional*
    Dimensionalizar el vector de p_n de soluciones al mismo tamaño que el vector de p_a.
    p_n.max_coste = 0
    p_n.n_individuos = p_a.n_individuos
    PARA CADA i ∈ {0, número de individuos de la p - 1} HACER:
        ganador = torneoBinario(p_a)
        p_anvector[i] = p_a.v[ganador]
        SI p_n.vector[i] > p_n.max_coste HACER:
            p_n.mejor_solucion = i
            p_n.max_coste = p_n.vector[i].coste
        FIN SI
    FIN PARA CADA
}

Como vemos, en este esquema generacional, se aplican tantos torneos como individuos existen en la población genética, incluyendo los individuos ganadores en la población de padres.

<u>Parámetros:</u> población donde elegiremos los individuos (p), padre1 y padre2 que son soluciones de la población que van a ser los padres(se incluyen porque serán actualizados por referencia).
<u>Resultado:</u> Método de selección de individuos de la población.
seleccionarIndividuos*{ //Caso estacionario*
    hijo1 = torneoBinario(p)
    HACER:
        hijo2 = torneoBinario(p)
    MIENTRAS hijo1 sea igual que hijo2
    padre1 = p.vector[hijo1]
    padre2 = p.vector[hijo2]
}

Como vemos, en este esquema estacionario, se aplican solo dos torneos para elegir los dos padres se van a cruzar.

Hemos usado como hemos visto el torneo binario como método de selección, que escoge dos individuos de la población de forma aleatoria y escoge el mejor de ellos:

<u>Parámetros:</u> población donde elegiremos los individuos (p)
<u>Resultado:</u> Índice que representa un individuo de la población.
torneoBinario*{*
    índice1 = numero_aleatorio entre 0 y número de individuos de p
    índice2 = numero_aleatorio entre 0 y número de individuos de p
    SI p.vector[índice1].coste > p.vector[índice2].coste HACER:
        DEVOLVER índice1
    SINO HACER:
        DEVOLVER índice2
    FIN SI
}

Cruce

Hemos incluido dos versiones, una de cruce uniforme y otra de cruce posición. Veamos primero el cruce uniforme:

Parámetros: padre1 y padre2 que son soluciones de la población y matriz de distancias.
Resultado: Solución cruzada.
cruceUniforme*{*
Inicializamos la solución hijo como una solución permitida.
hijo.evaluada = false
nª a seleccionar = 0
PARA CADA i ∈ {0, padre1.vector.size() - 1} HACER:
SI padre1.vector[i] está a true → nª a seleccionar = nª a seleccionar + 1 FIN SI
SI padre1.vector[i] y padre2.vector[i] están a true HACER:
hijo.vector[i] = true
FIN SI
SI padre1.vector[i] y padre2.vector[i] están a false HACER:
hijo.vector[i] = false
FIN SI
EN OTRO CASO HACER:
aleatorio ∈ {true, false}
hijo.vector[i] = aleatorio
FIN EN OTRO CASO
FIN PARA CADA
repararSolucion(hijo, nª a seleccionar y matriz de distancias)
}

En este caso, si coinciden los valores de los padres, heredamos dicho valor al hijo, si no, elegimos si el hijo incluye el elemento de forma aleatoria. Como vemos, para éste cruce puede ocurrir que los hijos resultantes no sean soluciones válidas (factibles), por lo que tenemos que utilizar una función de reparación:

Parámetros: solución a reparar (s), n.º a seleccionar, y matriz de distancias .
Resultado: Solución reparada.
repararSolucion*{*
n.º seleccionados = 0
PARA CADA i ∈ {0, padre1.vector.size() - 1} HACER:
SI s.vector[i] = true HACER: n.º seleccionados = n.º seleccionados + 1 FIN SI
FIN PARA CADA
MIENTRAS n.º seleccionados > n.º a seleccionar HACER:
i_max_coste = elegirMayorContribucionSeleccionados(·)
s.vector[i_max_coste] = false
n.º seleccionados = n.º seleccionados - 1
FIN MIENTRAS
MIENTRAS n.º seleccionados < n.º a seleccionar HACER:
i_max_coste = elegirMayorContribucionNoSeleccionados(·)
s.vector[i_max_coste] = true
n.º seleccionados = n.º seleccionados + 1
FIN MIENTRAS
}

Vemos que usamos dos funciones para ésto: elegirMayorContribucionSeleccionados(·), que elige el elemento de los seleccionados que aporta mayor diversidad y elegirMayorContribucionNoSeleccionados(·) que elige el elemento de los no seleccionados que aporta mayor diversidad. Su implementaciones son sencillas y no creo que sea muy importante incluirlas en dicha memoria. Simplemente recorremos los elementos correspondientes y aplicar la distanciaAUnConjunto de cada elemento y hacer las comprobaciones. Para más información, se encuentran en cualquier *_uniforme.cpp.

Por otro lado veamos el cruce posición.

Parámetros: padre1 y padre2 que son las soluciones a cruzar.
Resultado: hijo1 e hijo2 que son las soluciones resultantes.
crucePosicion{
hijo1 = padre1
hijo2 = padre2
hijo1.evaluada = false
hijo2.evaluada = false
Inicializamos a false un vector (s) de booleanos del tamaño del vector padre1.
Creamos tambien el vector s1 y s2.
PARA CADA i ∈ {0, padre1.vector.size() - 1} HACER:
SI padre1.vector[i] = padre2.vector[i] HACER:
hijo1.vector[i] = padre1.vector[i]
hijo2.vector[i] = padre2.vector[i]
SINO HACER:
s[i] = true
s1.insertar(padre1.vector[i])
FIN SI
FIN PARA CADA
Barajar s1
s2 = s1
Barajar s2
j = 0
PARA CADA i ∈ {0, padre1.vector.size() - 1} HACER:
SI s[i] = true HACER:
hijo1.vector[i] = s1[j]
hijo2.vector[i] = s2[j]
j = j + 1
FIN SI
FIN PARA CADA
}

Básicamente, si coincide un valor para los dos padre, se mantiene en el hijo y si no copiar el de algún padre (da igual cual, pero siempre el mismo) y guardar los valores en un vector, reordenarlo y asignarlo a un hijo, y lo mismo con el otro hijo.

Mutación

Para el operador de mutación, tenemos la suerte que usaremos el mismo operador para todos los algoritmos implementados. La diferencia será la probabilidad de mutación o sobre que elementos de la población aplicarlos.

<u>Parámetros:</u> solución a mutar (s), n.º de evaluaciones de la función objetivo, y matriz de distancias.
<u>Resultado:</u> Solución mutada.
mutarSolucion*{*
HACER: n.º aleatorio1= aleatorio entre 0 y s.vector.size() MIENTRAS n.º aletorio1 = true
HACER: n.º aleatorio2= aleatorio entre 0 y s.vector.size() MIENTRAS n.º aletorio2 = false
s.vector[nº aletorio1] = true
s.vector[nº aletorio2] = false
SI s.evaluada = true HACER:
antigua_diversidad = distanciaAUnConjunto(nº aleatorio2, s.vector, m) –
matriz_distancias[nº aletorio2][nº aleatorio1]
nueva_diversidad = distanciaAUnConjunto(nº aleatorio1, s.vector, m)
s.coste = s.coste + nueva_diversidad – antigua_diversidad
n.º evaluaciones = n.º evaluaciones + 1
FIN SI
}

Como vemos se trata de elegir dos índices aleatorios contrarios (uno a 1 y otro a 0) y cambiarlos y por supuesto actualizar los datos (ésto último lo hacemos para simplificar a continuación la función de evaluación).

Reemplazo de la población

De nuevo existirán dos tipos de reemplazo de la población según el algoritmo que estemos usando. Por un lado para los algoritmos generacionales tenemos:

Parámetros: población que se actualizará (p_a), población que pasará a ser actual (p_n).
Resultado: población actualizada
reemplazarPoblacion*{*
índice peor solución = p_n.mejor_solucion
coste_minimo = p_n.max_coste
SI p_n.max_coste < p_a.max_coste (Si la población nueva es peor que la anterior) HACER:
PARA CADA j ∈ {0, padre1.vector.size() - 1}
SI p_n.vector[j].coste < coste_minimo HACER:
índice peor solución = j
coste_mínimo = p_n.vector[j].coste
FIN SI
FIN PARA CADA
i = p_a.mejor_solucion
mejor solución = p_a.vector[i]
p_a.swap(p_n)
p_a.vector[índice peor solución] = mejor solución
SINO
p_a.swap(p_n)
p_a.max_coste = p_n.max_coste
p.a.mejor_solucion = p_n.max_coste
FIN SI
}

En esta función lo que hacemos es comprobar si la población nueva es peor que la anterior, en ese caso busco el peor individuo de la población nueva, guardamos el mejor individuo de la población actual e intercambio las poblaciones pero guardando la mejor solución en la población que ahora es la actual, pero en el lugar del peor individuo que ya habíamos calculado. En el caso de que la población nueva es mejor que la anterior, solo intercambiamos y actualizamos los datos.

Por último incluimos el mismo método pero para algoritmos estacionarios. Difiere en que ahora solo debemos incluir los 2 hijos generados, sustituyéndolos por los 2 peores. Es una función mucho más simple pues sólo necesitamos ordenar la población sobrecargando el operador < como en la P1. Además ahora al ordenar el vector de soluciones tendremos como primer individuo el mejor y podemos usar operaciones de la STL para dicha función de forma sencilla.

Parámetros: población a actualizar (p), hijo1 e hijo2 que son las soluciones a incluir
Resultado: población actualizada.
reemplazarPoblacion*{*
p.insertar(hijo1) y p.insertar(hijo2)
ordenar(p por coste)
p.redimensionar(p.vector.size() - 2)
p.max_coste = p.vector[0].coste
}

Descripción de la estructura de los métodos

Para ésta práctica disponemos de siete algoritmos para la búsqueda de la solución. En este apartado describiremos la estructura de dichos métodos y las operaciones relevantes. Podemos diferenciar varios grupos que forman los algoritmos:

TABLA CARACTERÍSTICAS ALGORITMOS	Selección y Reemplazo Generacional	Selección y Reemplazo Estacionario	Cruce uniforme	Cruce posición
AGG_uniforme	X		X	
AGG_posicion	X			X
AGE_uniforme		X	X	
AGE_posicion		X		X
AM_tipo0	X		X	
AM_tipo1	X		X	
AM_tipo2	X		X	

A) Algoritmos Genéticos (AGs)

Corresponden a los cuatro primeros de la tabla anterior. Distinguimos dos grandes grupos en este apartado: los generacionales (AGGs), que usan técnicas generacionales y elitistas; y los estacionarios (AGEs), en los que en cada iteración únicamente seleccionamos dos padres que, tras ser cruzados y mutados, compiten por entrar de nuevo en población. Veamos el pseudocódigo de dichos algoritmos, por un lado los algoritmos genéticos generacionales:

Parámetros: matriz de distancias, n.º elementos a seleccionar (n) y n.º máx. de evaluaciones.
Resultado: solución_calculada y coste de la solución.
AGG{
Incializamos parámetros(n.º de evaluaciones = evaluaciones = 0, n.º individuos = 50, prob. de mutación = 0.001, prob. de cruce = 0.7)
poblacion_actual = inicializarPoblacion(·)
evaluarPoblacion(·)
MIENTRAS n.º de evaluaciones < n.º máx. de evaluaciones HACER:
generaciones = generaciones + 1
poblacion_nueva = seleccionarIndividuos(poblacion_actual)
cruce(poblacion_nueva)
mutarPoblacion(poblacion_nueva)
evaluarPoblacion(poblacion_nueva)
poblacion_actual = reemplazarPoblacion(poblacion_nueva)
FIN MIENTRAS
DEVOLVER poblacion_actual.vector[mejor_solucion] y coste de ésta.

Hay dos algoritmos de los anteriores expuestos que se basan en esta estructura y éstos son el AGG_uniforme (que usa el operador de cruce uniforme) y el algoritmo AGG_posicion (que usa el operador de cruce posición).

Veamos ahora el pseudocódigo de los algoritmos genéticos estacionarios:

<u>Parámetros:</u> matriz de distancias, n.º elementos a seleccionar (n) y n.º máx. de evaluaciones.
<u>Resultado:</u> solución_calculada y coste de la solución.
AGE{
Incializamos parámetros(n.º de evaluaciones = evaluaciones = 0, n.º individuos = 50, prob. de mutación = 0.001, prob. de cruce = 0.7)
poblacion_actual = inicializarPoblacion(·)
evaluarPoblacion(·)
ordenar(poblacion)
MIENTRAS n.º de evaluaciones < n.º máx. de evaluaciones HACER:
generaciones = generaciones + 1
padre1 y padre2 = seleccionarIndividuos(poblacion_actual)
hijo1 e hijo2 = cruce(padre1 y padre2)
hijo1 e hijo2 = mutarPoblacion(hijo1 e hijo2)
evaluarSolucion(hijo1)
evaluarSolucion(hijo2)
n.º de evaluaciones = n.º de evaluaciones + 2
poblacion_actual = reemplazarPoblacion(poblacion_actual, hijo1, hijo2)
FIN MIENTRAS
DEVOLVER poblacion_actual.vector[mejor_solucion] y coste de ésta.

Hay dos algoritmos de los anteriores expuestos que se basan en esta estructura y éstos son el AGE_uniforme (que usa el operador de cruce uniforme) y el algoritmo AGE_posicion (que usa el operador de cruce posición).

B) Algoritmos Meméticos (AMs)

Los algoritmos meméticos implementados consisten en hibridar el algoritmo generacional de cruce uniforme (AGG_uniforme) con la búsqueda local que desarrollamos en la Práctica 1. Es decir se mantendrá el esquema evolutivo de reemplazo generacional pero ahora aplicamos una búsqueda local a algunos individuos de la población tras unas generaciones. Se han estudiado las tres hibridaciones:

- AM-(10,1.0): Cada 10 generaciones, se aplica la búsqueda local sobre todos los cromosomas de la población. Lo que nosotros hemos llamado AM_tipo0,
- AM-(10,0.1): Cada 10 generaciones, se aplica la búsqueda local sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con probabilidad 0.1 para cada cromosoma. Lo que nosotros hemos llamado AM_tipo1.
- AM-(10,0.1mej): Cada 10 generaciones, aplicar la BL sobre los 0.1·N mejores cromosomas de la población actual (N es el tamaño de ésta). Lo que nosotros hemos llamado AM_tipo2.

Así el pseudocódigo del algoritmo es análogo al pseudocódigo del algoritmo de AGG a diferencia que justo después de reemplazar la población incluimos la función memetizar, es decir:

<u>Parámetros:</u> matriz de distancias, n.º elementos a seleccionar (n), n.º máx. de evaluaciones y tipo de algoritmo memético.
<u>Resultado:</u> solución_calculada y coste de la solución.
***AM**{*
Incializamos parámetros(n.º de evaluaciones = evaluaciones = 0, n.º individuos = 50, prob. de mutación = 0.001, prob. de cruce = 0.7)
poblacion_actual = inicializarPoblacion(·)
evaluarPoblacion(·)
MIENTRAS n.º de evaluaciones < n.º máx. de evaluaciones HACER:
generaciones = generaciones + 1
poblacion_nueva = seleccionarIndividuos(poblacion_actual)
cruce(poblacion_nueva)
mutarPoblacion(poblacion_nueva)
evaluarPoblacion(poblacion_nueva)
poblacion_actual = reemplazarPoblacion(poblacion_nueva)
SI generaciones % 10 es igual a 0 HACER:
memetizar(…, tipo de algoritmo memético)
FIN MIENTRAS
DEVOLVER poblacion_actual.vector[mejor_solucion] y coste de ésta.
}

La gran similitud entre los algoritmos meméticos que teníamos que implementar hizo que simplemente usemos esta función (memetizar) para distinguir entre un algoritmo y otro. Incluímos el pseudocódigo de dicha función:

```
Parámetros: población (p), tipo de algoritmo memético (tipo), n.º máx. de evaluaciones de la
        búsqueda local, evaluaciones (e), probabilidad, y matriz de distancias (m).
Resultado: población resultante una vez hecha la búsqueda local.
memetizar{
    SI tipo = 0 HACER:
        PARA CADA i ∈ {0, p.n_individuos – 1} HACER:
            e = e + busquedaLocal(m, p.vector[i], n.º máx. de evaluaciones de la b.l.)
            evaluarSolucion(p.vector[i], m)
            SI p.vector[i].coste > p.max_coste HACER:
                p.max_coste = p.vector[i].coste
                p.mejor_solucion = i
            FIN SI
        FIN PARA CADA
    SI NO SI tipo = 1 HACER:
        n.º de memetizaciones = p.n_invididuos * probabilidad
        PARA CADA i ∈ {0, n.º de memetizaciones – 1} HACER:
            n.º aleatorio ∈ {0, p.n_invididuos – 1}
            e = e + busquedaLocal(m, p.vector[n.º aleatorio], n.º máx. de ev. de la b.l.)
            evaluarSolucion(p.vector[n.º aleatorio], m)
            SI p.vector[n.º aleatorio].coste > p.max_coste HACER:
                p.max_coste = p.vector[n.º aleatorio].coste
                p.mejor_solucion =  n.º aleatorio
            FIN SI
        FIN PARA CADA
    SI NO SI tipo = 2 HACER:
        mejor = p.mejor_solucion
        e = e + busquedaLocal(m, p.vector[mejor], n.º máx. de ev. de la b.l.)
        evaluarSolucion(p.vector[mejor], m)
        p.max_coste = p.vector[mejor].coste
    FIN SI
}
```

Recordamos que la búsqueda local implementada en la Práctica 1 devolvía el número de evaluaciones que había realizado en su ejecución. Por otro lado hemos podido implementar el tipo 2 de dicha forma ya que N (número individuos de la población) es igual a 10, así N*0.1 es 1, y solo tenemos que hacer la búsqueda sobre ese mejor.

Como bien sabemos, la búsqueda local implementada en la práctica anterior necesitaba de una representación en enteros de la solución. Para ello como hemos comentado, creamos la estructura solucion_ints para éste fin y simplemente hemos creado dos funciones pasarDeIntsABits(·) y pasarDeBitsAInts(·), las cuales pasan de una solución en representación en enteros a representación binaria y de una solución en representación binaria a representación en enteros respectivamente.

Procedimiento considerado para desarrollar la práctica

Todo el código se ha implementado en C++. El procedimiento para el desarrollo de la práctica ha seguido las explicaciones dadas en la memoria. El código ha sido implementado por mí siguiendo el Seminario 3 de la asignatura, el guión de la práctica y las explicaciones dadas en clase.

Está estructurado en una serie de carpetas: BIN (donde se encontraran los ejecutables), entrada (donde se encuentran las ficheros de datos proporcionados), resultados (donde se encuentran las tablas pedidas con los resultados de las ejecuciones), FUENTES (donde se encuentran los dos ficheros creados para la práctica: AGG_uniforme.cpp donde se incluye el algoritmo genético generacional con cruce uniforme, AGG_posicion.cpp donde se incluye el algoritmo genético generacional con cruce posición, AGE_uniforme.cpp donde se incluye el algoritmo genético estacionario con cruce uniforme, AGE_posicion.cpp donde se incluye el algoritmo genético estacionario con cruce posición y AM.cpp donde se incluye el algoritmo memético implementado) y un makefile.

El makefile realiza toda la ejecución. Dispone de varias reglas entre las que destacamos:

- make clean: limpia los ejecutables creados anteriormente
- make: ejecuta los dos algoritmos con algunos de los problemas de ejemplo.
- make Resultados: realiza las 210 ejecuciones necesarias para obtener los datos

El fichero makefile ha ejecutado con opciones de optimización -O2.

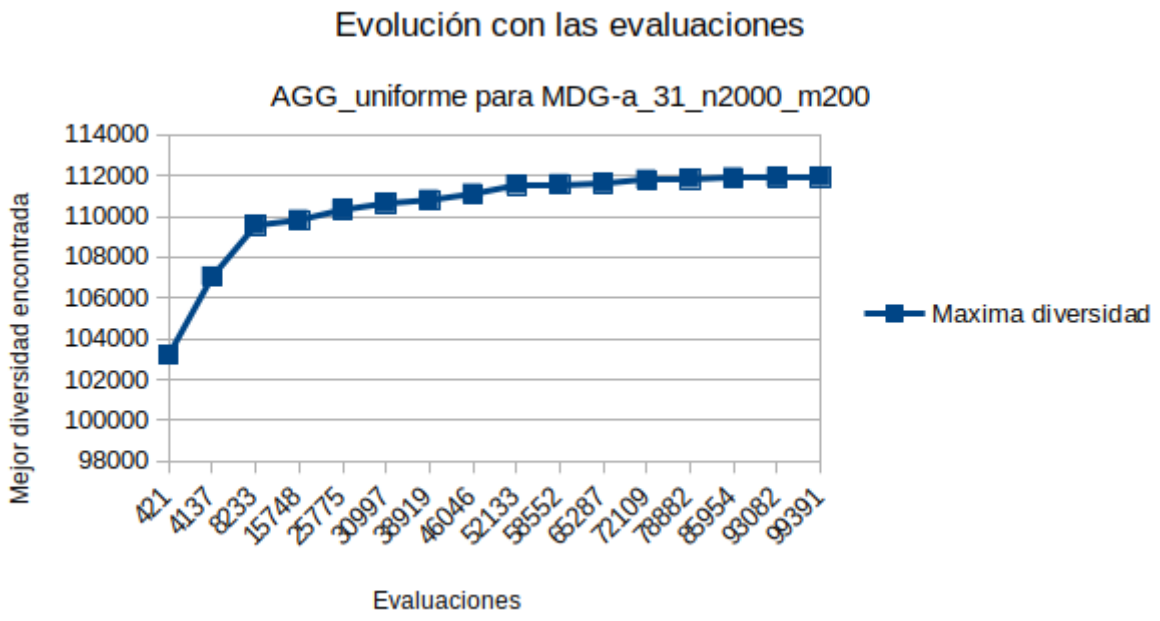
Todos los ficheros debe ejecutarse introduciendo un valor de semilla. Este valor será introducido en el Makefile en mi caso y se puede cambiar en la variable del makefile SEMILLA. Además los algoritmos meméticos necesitan ejecutarse introduciendo también el valor numérico que indican el tipo de algoritmo que ejecutamos (valor perteneciente a {0, 1, 2}).

Los resultados obtenidos se deben a que el PC usado para dichas ejecuciones consta de:

- Sistema operativo Ubuntu 18.10
- Memoria de 7885 M
- Procesador Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

Hemos usado los parámetros pedidos en el enunciado: En los algoritmos AGs el tamaño de la población es de 50 cromosomas. La probabilidad de cruce es 0,7 en el AGG y 1 en el AGE (siempre se cruzan los dos padres). La probabilidad de mutación (por gen) es de 0,001 en ambos casos. Mientras que para el caso de los algoritmos AMs, el tamaño de la población del AGG_uniforme es de 10 cromosomas. Las probabilidades de cruce y mutación son 0,7 y 0,001 (por gen) en ambos casos. Se detendrá la ejecución dela BL aplicada sobre un cromosoma bien cuando no se encuentre mejora en todo el entorno o bien cuando se hayan evaluado 400 vecinos distintos en la ejecución.

Sin embargo hemos fijado el criterio de parada en 50.000 evaluaciones en vez de 100.000 evaluaciones. Esto es porque la mejora en la diversidad obtenida con respecto al número de evaluaciones a partir de 50.000 evaluaciones es casi nula para todos los algoritmos. Hemos dibujado ésto que reflejamos en la siguiente gráfica, tomando como modelo el algoritmo de AGG_uniforme con el problema de MDG-a_31_n2000_m200:



Como vemos, aunque obtenemos cierta mejora, el coste computacional y en tiempo que suponía ejecutar los 7 algoritmos con todas las instancias ha influido en dicha decisión.

Experimentos y análisis de resultados

A continuación plasmamos los datos recogidos por las ejecuciones propuestas para los siete algoritmos:

A) **AGG_uniforme**

AGG_uniforme				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18226,5	19587,12891	6,95	4,96527
GKD-c_12_n500_m50	17665,7	19360,23633	8,75	5,76152
GKD-c_13_n500_m50	17831,6	19366,69922	7,93	5,80473
GKD-c_14_n500_m50	17870,5	19458,56641	8,16	5,72931
GKD-c_15_n500_m50	17835,3	19422,15039	8,17	5,65694
GKD-c_16_n500_m50	18074,4	19680,20898	8,16	5,20529
GKD-c_17_n500_m50	17986,1	19331,38867	6,96	5,96743
GKD-c_18_n500_m50	17859,6	19461,39453	8,23	5,83463
GKD-c_19_n500_m50	17959,5	19477,32813	7,79	4,98309
GKD-c_20_n500_m50	17832	19604,84375	9,04	4,90718
MDG-b_1_n500_m50	750959	778030,625	3,48	5,04212
MDG-b_2_n500_m50	753767	779963,6875	3,36	4,92418
MDG-b_3_n500_m50	756022	776768,4375	2,67	4,62276
MDG-b_4_n500_m50	755408	775394,625	2,58	4,93484
MDG-b_5_n500_m50	751993	775611,0625	3,05	5,56855
MDG-b_6_n500_m50	756157	775153,6875	2,45	5,1374
MDG-b_7_n500_m50	759278	777232,875	2,31	4,92538
MDG-b_8_n500_m50	751768	779168,75	3,52	4,86483
MDG-b_9_n500_m50	757179	774802,1875	2,27	5,0193
MDG-b_10_n500_m50	762686	774961,3125	1,58	5,33289
MDG-a_31_n2000_m200	111545	114139	2,27	283,597
MDG-a_32_n2000_m200	109949	114092	3,63	270,564
MDG-a_33_n2000_m200	111260	114124	2,51	283,683
MDG-a_34_n2000_m200	110299	114203	3,42	286,789
MDG-a_35_n2000_m200	111099	114180	2,70	288,847
MDG-a_36_n2000_m200	110865	114252	2,96	282,319
MDG-a_37_n2000_m200	110703	114213	3,07	288,973
MDG-a_38_n2000_m200	109998	114378	3,83	288,876
MDG-a_39_n2000_m200	111790	114201	2,11	293,63
MDG-a_40_n2000_m200	110797	114191	2,97	294,175

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	4,56302738561465
Media Tiempo:	98,8880213333333

B) **AGG_posición**

AGG_posicion				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18173,1	19587,12891	7,22	1,00243
GKD-c_12_n500_m50	17683,4	19360,23633	8,66	0,917061
GKD-c_13_n500_m50	17739,6	19366,69922	8,40	0,929092
GKD-c_14_n500_m50	17823	19458,56641	8,41	0,928663
GKD-c_15_n500_m50	17675,8	19422,15039	8,99	0,939613
GKD-c_16_n500_m50	17901,7	19680,20898	9,04	1,02756
GKD-c_17_n500_m50	17882,4	19331,38867	7,50	0,953943
GKD-c_18_n500_m50	17521,8	19461,39453	9,97	0,958588
GKD-c_19_n500_m50	17718,8	19477,32813	9,03	0,9488
GKD-c_20_n500_m50	17794,2	19604,84375	9,24	0,95895
MDG-b_1_n500_m50	751852	778030,625	3,36	0,965117
MDG-b_2_n500_m50	737389	779963,6875	5,46	1,00491
MDG-b_3_n500_m50	754674	776768,4375	2,84	0,919125
MDG-b_4_n500_m50	754884	775394,625	2,65	0,932485
MDG-b_5_n500_m50	753617	775611,0625	2,84	0,975174
MDG-b_6_n500_m50	759430	775153,6875	2,03	1,04056
MDG-b_7_n500_m50	741181	777232,875	4,64	0,952942
MDG-b_8_n500_m50	740043	779168,75	5,02	1,06294
MDG-b_9_n500_m50	742112	774802,1875	4,22	0,999866
MDG-b_10_n500_m50	755836	774961,3125	2,47	0,956713
MDG-a_31_n2000_m200	106418	114139	6,76	9,75718
MDG-a_32_n2000_m200	105494	114092	7,54	9,06436
MDG-a_33_n2000_m200	105956	114124	7,16	9,63326
MDG-a_34_n2000_m200	106365	114203	6,86	8,34459
MDG-a_35_n2000_m200	106118	114180	7,06	7,98553
MDG-a_36_n2000_m200	106067	114252	7,16	8,33028
MDG-a_37_n2000_m200	106411	114213	6,83	8,33431
MDG-a_38_n2000_m200	106541	114378	6,85	8,6557
MDG-a_39_n2000_m200	106302	114201	6,92	10,1856
MDG-a_40_n2000_m200	105931	114191	7,23	9,78738

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	6,41150124645679
Media Tiempo:	3,64842406666667

C) AGE_uniforme

AGE_uniforme				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18255,6	19587,12891	6,80	1,92972
GKD-c_12_n500_m50	17762,9	19360,23633	8,25	1,74018
GKD-c_13_n500_m50	17789,6	19366,69922	8,14	1,68491
GKD-c_14_n500_m50	17779,9	19458,56641	8,63	1,72569
GKD-c_15_n500_m50	17799,3	19422,15039	8,36	1,6325
GKD-c_16_n500_m50	18115,9	19680,20898	7,95	1,73134
GKD-c_17_n500_m50	17942,9	19331,38867	7,18	1,76674
GKD-c_18_n500_m50	17828,5	19461,39453	8,39	1,62417
GKD-c_19_n500_m50	17851,6	19477,32813	8,35	1,80643
GKD-c_20_n500_m50	17889,2	19604,84375	8,75	1,76334
MDG-b_1_n500_m50	755370	778030,625	2,91	1,73296
MDG-b_2_n500_m50	758928	779963,6875	2,70	1,97013
MDG-b_3_n500_m50	763443	776768,4375	1,72	1,90218
MDG-b_4_n500_m50	759394	775394,625	2,06	1,75929
MDG-b_5_n500_m50	758386	775611,0625	2,22	1,72435
MDG-b_6_n500_m50	760479	775153,6875	1,89	1,79495
MDG-b_7_n500_m50	774552	777232,875	0,34	1,84643
MDG-b_8_n500_m50	758492	779168,75	2,65	1,6573
MDG-b_9_n500_m50	753424	774802,1875	2,76	1,71337
MDG-b_10_n500_m50	755509	774961,3125	2,51	1,8142
MDG-a_31_n2000_m200	112762	114139	1,21	306,555
MDG-a_32_n2000_m200	112860	114092	1,08	401,712
MDG-a_33_n2000_m200	112538	114124	1,39	416,56
MDG-a_34_n2000_m200	112713	114203	1,30	388,31
MDG-a_35_n2000_m200	112418	114180	1,54	374,437
MDG-a_36_n2000_m200	112473	114252	1,56	373,975
MDG-a_37_n2000_m200	112593	114213	1,42	413,93
MDG-a_38_n2000_m200	112476	114378	1,66	414,532
MDG-a_39_n2000_m200	112628	114201	1,38	402,86
MDG-a_40_n2000_m200	113012	114191	1,03	388,615

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	3,87122007809997
Media Tiempo:	130,560206

D) AGE_posición

AGE_posicion				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18236,6	19587,12891	6,89	1,67444
GKD-c_12_n500_m50	17710,8	19360,23633	8,52	1,77906
GKD-c_13_n500_m50	17883,3	19366,69922	7,66	1,70371
GKD-c_14_n500_m50	17893,3	19458,56641	8,04	1,74639
GKD-c_15_n500_m50	17815,7	19422,15039	8,27	1,77245
GKD-c_16_n500_m50	18112,2	19680,20898	7,97	1,74344
GKD-c_17_n500_m50	17973,6	19331,38867	7,02	1,73815
GKD-c_18_n500_m50	17787,8	19461,39453	8,60	1,81859
GKD-c_19_n500_m50	17949,9	19477,32813	7,84	1,57127
GKD-c_20_n500_m50	17880,1	19604,84375	8,80	1,67296
MDG-b_1_n500_m50	760988	778030,625	2,19	1,90844
MDG-b_2_n500_m50	766715	779963,6875	1,70	1,76632
MDG-b_3_n500_m50	759045	776768,4375	2,28	1,82656
MDG-b_4_n500_m50	763253	775394,625	1,57	1,87798
MDG-b_5_n500_m50	758790	775611,0625	2,17	1,80204
MDG-b_6_n500_m50	759060	775153,6875	2,08	1,7521
MDG-b_7_n500_m50	759081	777232,875	2,34	1,53766
MDG-b_8_n500_m50	763104	779168,75	2,06	1,62241
MDG-b_9_n500_m50	754764	774802,1875	2,59	1,59143
MDG-b_10_n500_m50	756684	774961,3125	2,36	1,58385
MDG-a_31_n2000_m200	111231	114139	2,55	36,2953
MDG-a_32_n2000_m200	111217	114092	2,52	35,5452
MDG-a_33_n2000_m200	111454	114124	2,34	32,3229
MDG-a_34_n2000_m200	111669	114203	2,22	33,5517
MDG-a_35_n2000_m200	111312	114180	2,51	31,8708
MDG-a_36_n2000_m200	111084	114252	2,77	31,8687
MDG-a_37_n2000_m200	111332	114213	2,52	31,2058
MDG-a_38_n2000_m200	111554	114378	2,47	33,1409
MDG-a_39_n2000_m200	111344	114201	2,50	34,3674
MDG-a_40_n2000_m200	111502	114191	2,35	33,3483

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	4,19007490613725
Media Tiempo:	12,266875

E) AM_tipo0

AM_tipo0				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18189,1	19587,12891	7,14	2,20847
GKD-c_12_n500_m50	17775,5	19360,23633	8,19	1,8395
GKD-c_13_n500_m50	17910,9	19366,69922	7,52	2,1308
GKD-c_14_n500_m50	17848,4	19458,56641	8,27	2,07628
GKD-c_15_n500_m50	17762,4	19422,15039	8,55	2,09405
GKD-c_16_n500_m50	18157	19680,20898	7,74	1,76865
GKD-c_17_n500_m50	17965,2	19331,38867	7,07	1,68763
GKD-c_18_n500_m50	17868,3	19461,39453	8,19	1,65759
GKD-c_19_n500_m50	17954,4	19477,32813	7,82	1,82604
GKD-c_20_n500_m50	17869,1	19604,84375	8,85	1,82994
MDG-b_1_n500_m50	767114	778030,625	1,40	2,01859
MDG-b_2_n500_m50	768962	779963,6875	1,41	2,05638
MDG-b_3_n500_m50	770222	776768,4375	0,84	2,08833
MDG-b_4_n500_m50	757235	775394,625	2,34	1,83164
MDG-b_5_n500_m50	763381	775611,0625	1,58	1,70959
MDG-b_6_n500_m50	757639	775153,6875	2,26	1,72697
MDG-b_7_n500_m50	768726	777232,875	1,09	1,78416
MDG-b_8_n500_m50	763623	779168,75	2,00	2,03045
MDG-b_9_n500_m50	771547	774802,1875	0,42	1,80669
MDG-b_10_n500_m50	769071	774961,3125	0,76	1,64828
MDG-a_31_n2000_m200	112305	114139	1,61	186,367
MDG-a_32_n2000_m200	111326	114092	2,42	192,071
MDG-a_33_n2000_m200	112074	114124	1,80	202,087
MDG-a_34_n2000_m200	111779	114203	2,12	198,422
MDG-a_35_n2000_m200	111349	114180	2,48	204,634
MDG-a_36_n2000_m200	111779	114252	2,16	174,886
MDG-a_37_n2000_m200	112576	114213	1,43	160,898
MDG-a_38_n2000_m200	112359	114378	1,77	176,832
MDG-a_39_n2000_m200	110959	114201	2,84	175,191
MDG-a_40_n2000_m200	112425	114191	1,55	175,238

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	3,7869511327179
Media Tiempo:	62,814867666667

F) AM_tipo1

AM_tipo1				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18102,7	19587,12891	7,58	2,88597
GKD-c_12_n500_m50	17714,3	19360,23633	8,50	2,52986
GKD-c_13_n500_m50	17901,8	19366,69922	7,56	2,68289
GKD-c_14_n500_m50	17770,3	19458,56641	8,68	2,78389
GKD-c_15_n500_m50	17726,2	19422,15039	8,73	3,45754
GKD-c_16_n500_m50	18059,9	19680,20898	8,23	3,62872
GKD-c_17_n500_m50	17903,8	19331,38867	7,38	2,63839
GKD-c_18_n500_m50	17869,2	19461,39453	8,18	3,35194
GKD-c_19_n500_m50	17990,1	19477,32813	7,64	3,13542
GKD-c_20_n500_m50	17742,1	19604,84375	9,50	2,87305
MDG-b_1_n500_m50	739879	778030,625	4,90	2,79329
MDG-b_2_n500_m50	758557	779963,6875	2,74	2,71083
MDG-b_3_n500_m50	769008	776768,4375	1,00	2,86316
MDG-b_4_n500_m50	747108	775394,625	3,65	2,69224
MDG-b_5_n500_m50	756453	775611,0625	2,47	2,91609
MDG-b_6_n500_m50	759841	775153,6875	1,98	2,85289
MDG-b_7_n500_m50	767065	777232,875	1,31	2,85709
MDG-b_8_n500_m50	763586	779168,75	2,00	2,98998
MDG-b_9_n500_m50	756026	774802,1875	2,42	2,58042
MDG-b_10_n500_m50	758390	774961,3125	2,14	3,19731
MDG-a_31_n2000_m200	109893	114139	3,72	156,631
MDG-a_32_n2000_m200	110956	114092	2,75	146,671
MDG-a_33_n2000_m200	109873	114124	3,72	150,201
MDG-a_34_n2000_m200	110150	114203	3,55	151,414
MDG-a_35_n2000_m200	110683	114180	3,06	150,806
MDG-a_36_n2000_m200	111269	114252	2,61	148,302
MDG-a_37_n2000_m200	110876	114213	2,92	154,362
MDG-a_38_n2000_m200	109607	114378	4,17	154,951
MDG-a_39_n2000_m200	109648	114201	3,99	149,641
MDG-a_40_n2000_m200	111516	114191	2,34	158,226

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	4,64793546538365
Media Tiempo:	52,654199

G) AM_tipo2

AM_tipo2				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18119,5	19587,1289	7,49	3,19506
GKD-c_12_n500_m50	17777,1	19360,2363	8,18	3,00057
GKD-c_13_n500_m50	17961,2	19366,6992	7,26	4,84257
GKD-c_14_n500_m50	17914,2	19458,5664	7,94	6,67661
GKD-c_15_n500_m50	17723,5	19422,1504	8,75	6,65177
GKD-c_16_n500_m50	18137,2	19680,209	7,84	6,32688
GKD-c_17_n500_m50	17726,6	19331,3887	8,30	5,99507
GKD-c_18_n500_m50	17743,6	19461,3945	8,83	6,0989
GKD-c_19_n500_m50	17940,8	19477,3281	7,89	5,62711
GKD-c_20_n500_m50	17866	19604,8438	8,87	6,17066
MDG-b_1_n500_m50	766245	778030,625	1,51	6,19519
MDG-b_2_n500_m50	747473	779963,688	4,17	6,07123
MDG-b_3_n500_m50	760036	776768,438	2,15	5,1173
MDG-b_4_n500_m50	759150	775394,625	2,10	4,53199
MDG-b_5_n500_m50	765420	775611,063	1,31	5,62677
MDG-b_6_n500_m50	770666	775153,688	0,58	6,00744
MDG-b_7_n500_m50	758006	777232,875	2,47	4,92442
MDG-b_8_n500_m50	770143	779168,75	1,16	6,40485
MDG-b_9_n500_m50	765472	774802,188	1,20	4,85526
MDG-b_10_n500_m50	766572	774961,313	1,08	4,95976
MDG-a_31_n2000_m200	110707	114139	3,01	180,599
MDG-a_32_n2000_m200	111519	114092	2,26	183,776
MDG-a_33_n2000_m200	110302	114124	3,35	192,279
MDG-a_34_n2000_m200	112091	114203	1,85	180,534
MDG-a_35_n2000_m200	112183	114180	1,75	187,068
MDG-a_36_n2000_m200	110935	114252	2,90	216,677
MDG-a_37_n2000_m200	110948	114213	2,86	226,257
MDG-a_38_n2000_m200	110935	114378	3,01	202,011
MDG-a_39_n2000_m200	110916	114201	2,88	215,154
MDG-a_40_n2000_m200	111118	114191	2,69	199,057

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Media Desv:	4,18757680395868
Media Tiempo:	69,7563803333333

Conclusiones:

Lo primero que podemos observar, dados los resultados anteriores es que todos métodos llegan a soluciones bastante notables. Si juntamos y resumimos la información anterior sobre desviaciones y tiempos de todos los algoritmos, tenemos:

	Algoritmo						
	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM2
Desv.	4,6	6,4	3,9	4,2	3,8	4,6	4,2
Tiempo (s.)	98,9	3,6	130,6	12,3	62,8	52,7	69,8

Hay bastante información que destacar: por un lado lo que más destaca es el tiempo de ejecución de los algoritmos con cruce uniforme con respecto a los algoritmos con cruce posición. Ésto se debe a que al cruzar los individuos obtenemos soluciones no factibles (soluciones que no cumplen las restricciones impuestas, es decir obtenemos soluciones con mayor número de elementos elegidos al problema del permitido, o con menor). Dichas soluciones no factibles tenemos que repararlas, y para éste proceso tenemos por cada solución que buscar los elementos que contribuyen de mayor grado en la diversidad de la solución, lo que como cabe esperar, es bastante costoso computacionalmente. Ésto provoca que el algoritmo pueda ganar diversidad, pero como vemos el coste es elevado. Podríamos ser menos restrictivos y elegir elementos aleatorios dentro de la solución, en vez de elegir una mayor contribución. Sin embargo hemos comprobado que, aunque bajen considerablemente los tiempos de ejecución, el resultado es bastante peor. El cruce posición por su parte no necesita reparación, por lo que los costes computacionales se reducen, como ya vemos.

Otro de los aspectos a destacar es la efectividad de los problemas AGEs frente a los AGGs. Ésto puede ser debido a la gran presión selectiva que provocamos en el modelo estacionario al sustituir los dos peores cromosomas de la población, lo que provoca una convergencia rápida del algoritmo.

Metiéndonos de lleno en los algoritmos meméticos, recordamos que hemos usado una implementación basada en AGGu + búsqueda local. Hemos elegido el algoritmo de AGGu como base para los algoritmos meméticos porque como vemos, entre los resultados que hemos obtenido de los dos AGGs, es el mejor.

Entre los tres tipos de algoritmos meméticos que hemos implementado, el que mejor resultados obtiene es el tipo 0, es decir, el que aplica la búsqueda local sobre todos los cromosomas de la población. Ésto es porque, como recordamos, con la búsqueda local que implementamos en la práctica 1, obtuvimos resultados bastante buenos, por lo que cabe esperar que a mayor veces se aplique sobre los individuos de la población, mejores sean los resultados.

Recordamos los valores que obtuvimos con los mismos problemas en la práctica anterior solo con la búsqueda local:

Media Desv:	3,4782723
Media Tiempo:	11,3907252

Vemos que ninguno de los algoritmos nuevos implementados consigue mejorar a dicho algoritmo por si solo, incluso a pesar del mayor coste de computación que tienen estos nuevos

algoritmos. Sin embargo el algoritmo memético que aplica la búsqueda local sobre todos los cromosomas de la población consigue obtener resultados bastante análogos.

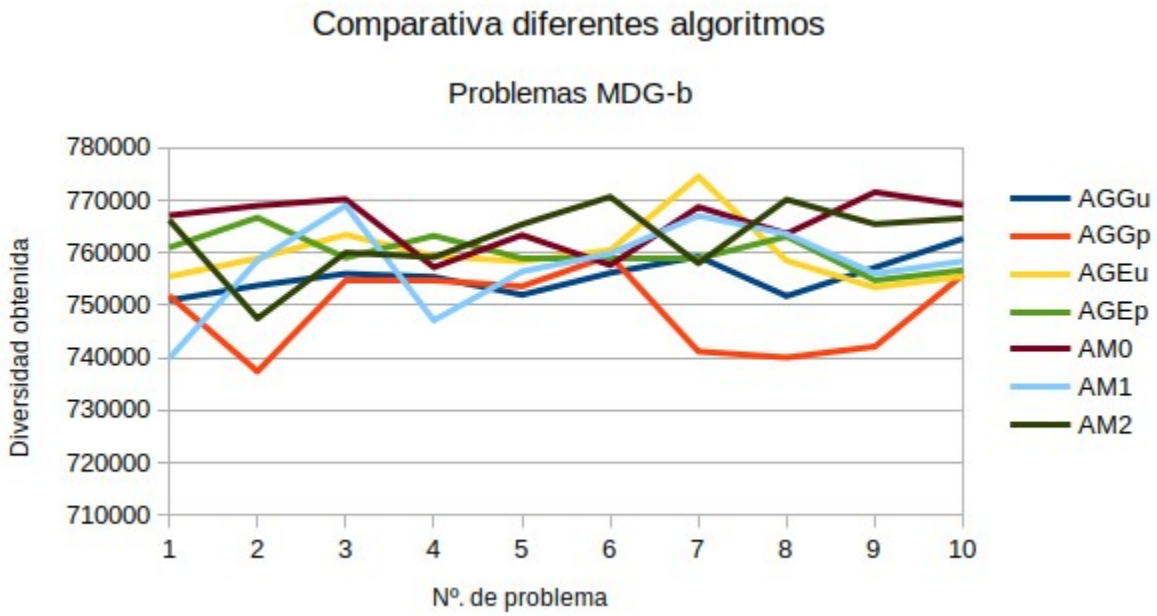
Destacamos también que se aprecia un coste en tiempo bastante grande cuando aumentamos el tamaño de los datos, como podemos apreciar en los problemas MDG-a_x_n2000_m200 con respecto a los anteriores.

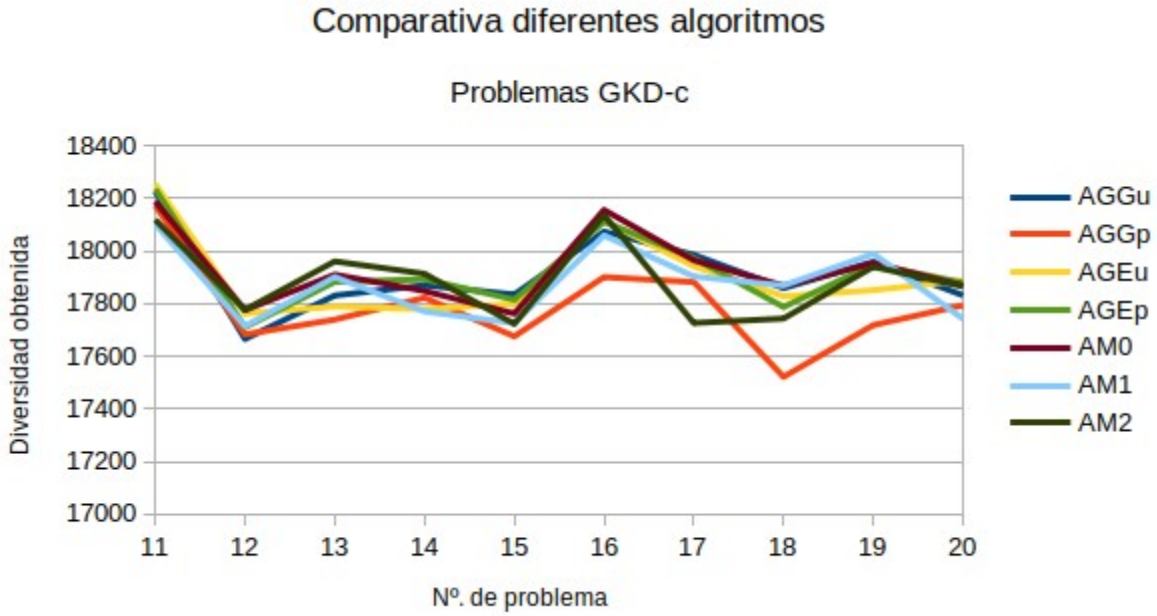
Desgranando los resultados y juntando los resultados obtenidos en una sola tabla, podemos analizar uno a uno los resultados obtenidos para cada problema con todos los algoritmos implementados. Es la tabla que justamente tenemos debajo.

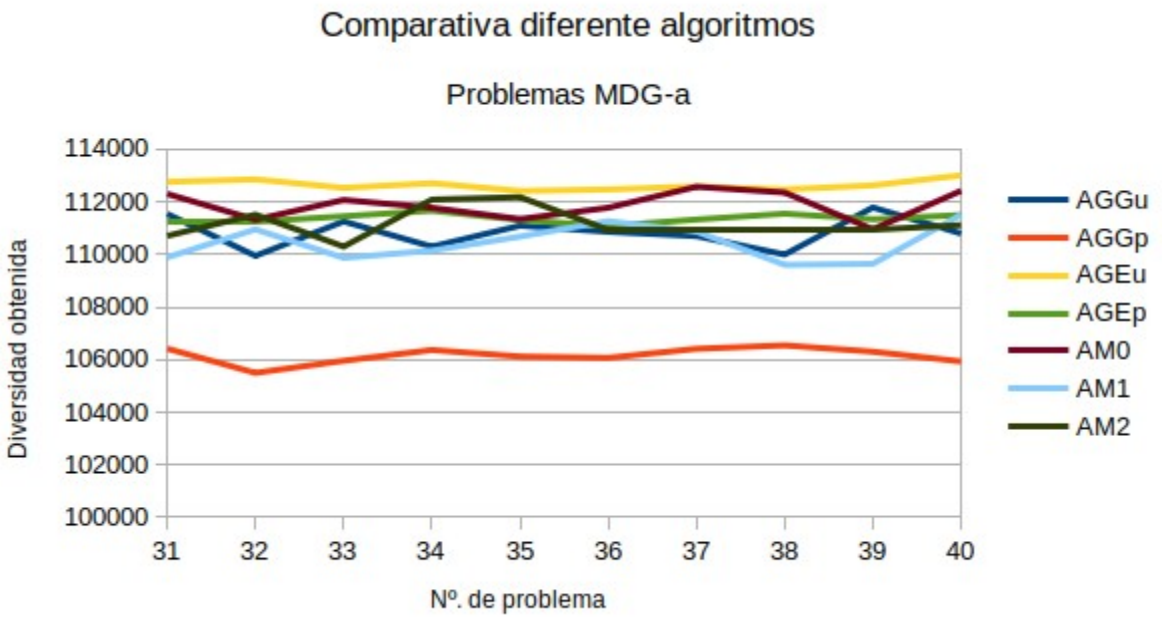
	Algoritmo						
	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM2
Caso	Diversidad						
GKD-c_11_n500_m50	18226,5	18173,1	18255,6	18236,6	18189,1	18102,7	18119,5
GKD-c_12_n500_m50	17665,7	17683,4	17762,9	17710,8	17775,5	17714,3	17777,1
GKD-c_13_n500_m50	17831,6	17739,6	17789,6	17883,3	17910,9	17901,8	17961,2
GKD-c_14_n500_m50	17870,5	17823	17779,9	17893,3	17848,4	17770,3	17914,2
GKD-c_15_n500_m50	17835,3	17675,8	17799,3	17815,7	17762,4	17726,2	17723,5
GKD-c_16_n500_m50	18074,4	17901,7	18115,9	18112,2	18157	18059,9	18137,2
GKD-c_17_n500_m50	17986,1	17882,4	17942,9	17973,6	17965,2	17903,8	17726,6
GKD-c_18_n500_m50	17859,6	17521,8	17828,5	17787,8	17868,3	17869,2	17743,6
GKD-c_19_n500_m50	17959,5	17718,8	17851,6	17949,9	17954,4	17990,1	17940,8
GKD-c_20_n500_m50	17832	17794,2	17889,2	17880,1	17869,1	17742,1	17866
MDG-b_1_n500_m50	750959	751852	755370	760988	767114	739879	766245
MDG-b_2_n500_m50	753767	737389	758928	766715	768962	758557	747473
MDG-b_3_n500_m50	756022	754674	763443	759045	770222	769008	760036
MDG-b_4_n500_m50	755408	754884	759394	763253	757235	747108	759150
MDG-b_5_n500_m50	751993	753617	758386	758790	763381	756453	765420
MDG-b_6_n500_m50	756157	759430	760479	759060	757639	759841	770666
MDG-b_7_n500_m50	759278	741181	774552	759081	768726	767065	758006
MDG-b_8_n500_m50	751768	740043	758492	763104	763623	763586	770143
MDG-b_9_n500_m50	757179	742112	753424	754764	771547	756026	765472
MDG-b_10_n500_m50	762686	755836	755509	756684	769071	758390	766572
MDG-a_31_n2000_m200	111545	106418	112762	111231	112305	109893	110707
MDG-a_32_n2000_m200	109949	105494	112860	111217	111326	110956	111519
MDG-a_33_n2000_m200	111260	105956	112538	111454	112074	109873	110302
MDG-a_34_n2000_m200	110299	106365	112713	111669	111779	110150	112091
MDG-a_35_n2000_m200	111099	106118	112418	111312	111349	110683	112183
MDG-a_36_n2000_m200	110865	106067	112473	111084	111779	111269	110935
MDG-a_37_n2000_m200	110703	106411	112593	111332	112576	110876	110948
MDG-a_38_n2000_m200	109998	106541	112476	111554	112359	109607	110935
MDG-a_39_n2000_m200	111790	106302	112628	111344	110959	109648	110916
MDG-a_40_n2000_m200	110797	105931	113012	111502	112425	111516	111118

Vemos que salvo casos concretos, en general los algoritmos meméticos mejoran a todos los algoritmos genéticos. Ésto es resultado de aplicar la búsqueda local que como hemos discutido mejora los resultados obtenidos, sin embargo cabe destacar que no son para nada significativos estas mejoras y en algunos casos específicos algunos algoritmos genéticos son mejores que los meméticos. Puede explicarse porque a la búsqueda local le proporcionamos demasiadas pocas evaluaciones por cromosoma y no consigue elaborar una búsqueda exhaustiva en el entorno. Otra de las explicaciones de éstos casos concretos es que se juega con muchos números aleatorios y la semilla que elijamos para un determinado algoritmo puede determinar el resultado obtenido, ya que no podemos estudiar todos las soluciones posibles. Éstos ejemplos son una clara muestra de como un algoritmo puede ser más efectivo frente a un caso de estudio particular y no serlo en otro.

Para visualizarlo correctamente, hemos realizado gráficas comparativas para cada una de las tres clases de problemas. Adjuntamos dichas gráficas a continuación y en ellas podemos apreciar lo discutido anteriormente:







Queda así claro que tal vez una estrategia evolutiva para estos problemas no es el mejor procedimiento, ya que obtenemos tiempos de ejecución mucho más elevados que los dos algoritmos implementados en la práctica anterior y los resultados obtenidos ésta vez no consiguen mejorar los obtenidos por los de la otra práctica.

Referencias bibliográficas

No he utilizado libros pero si he encontrado algunos links útiles para ciertas funciones del proyecto. A continuación los muestro:

- Para todo el tema de la implementación usé la documentación de la stl de C++ http://www.cplusplus.com/reference/stl/
- Para el tema de la generación de números aleatorios y fijar la semilla, me basé en implementaciones vistas en https://stackoverflow.com
- PRADO y página web de la asignatura.