METAHEURÍSTICAS (GRUPO 1)
UNIVERSIDAD DE GRANADA

PRÁCTICA 2:
PROBLEMA DE LA MÁXIMA DIVERSIDAD. TÉCNICAS DE BUSQUEDA BASADAS EN POBLACIONES.

Juan Miguel Gómez Daza (75928603Z)

juanmigd@correo.ugr.es

Índice

1. Formulación del problema

El *Problema de la Máxima Diversidad* o *MDP* (Maximum Diversity Problem) consiste en escoger un subconjunto de m de elementos de un conjunto principal de n elementos (por tanto n > m), de forma que sea lo más diverso posible.

En realidad existen diferentes variantes del problema según definamos la diversidad. En nuestro caso vamos a desarrollar la variante *MaxSum*, donde la diversidad la calculamos como la suma de distancias entre cada par de elementos seleccionados.

Para conocer las distancias entre los distintos elementos del conjunto principal se dispone de una matriz *D* de dimensión *nxn* donde d_{ij} es la distancia entre los elementos *i* y *j*. La matriz *D* es apriori triangular para no almacenar datos redundantes ya que $d_{ij} = d_{ji}$.

La solución que debemos encontrar es un subconjunto de *m* elementos. Una posible representación para las soluciones sería un vector x binario de tamaño n, que indica con un *1* en la posición *i* si el elemento *i*-esimo forma parte de la solución, *0* en caso contrario. Con esta representación definimos la variante *MaxSum* matemáticamente como:

$$\text{Maximize} \quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Subject to} \quad \sum_{i=1}^{n} x_i = m,$$

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

2. Componentes comunes

Consideraciones

Las variables definidas en la clase son:

- Tamaño del conjunto principal: **int n**
- Tamaño del conjunto de seleccionados: **int m**
- Matriz de distancias: **double distancias[n][n]**, para simplificar el cálculo de las distancias esta matriz genera como simetrica.
- Matriz de población: **bool poblacion[pop_size][n]**
- Matriz de generación: **bool generacion[pop_size][n]**
- Matriz del fitness: **double fitness[pop_size]**, almacena el fitness de la poblacion para ahorrar llamadas a evaluar.

Como esquema de representación de soluciones he utilizado una representación binaria, esto es un vector booleano de tamaño *n*, en el que cada posicion *i* del vector indica con true si se escoge *i* o false en caso contrario. El principal problema de esta representación es validar la solución, ya que solo serán correctas aquellas que tengan *m* posiciones del vector a *true*. Con esta representación nuestra función objetivo es encontrar *X* tal que:

$$maximizar \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$sujeto\ a \sum_{i=1}^{n} x_i = m$$

$$x_i = \{0, 1\},\ i = 1,\dots, n$$

En pseudocódigo esta función a maximizar es:

```
calcularDispersion(solucion: vector de bool): número real
    seleccionados = []
    para cada i en solucion
        si i == true
            seleccionados.insertar(i)
    si seleccionados.tam() == m
        para cada i en seleccionados
            j = siguiente(i)
            mientras j <= seleccionados.ultimo
                dispersion += distancia(i,j)
                j = siguiente(i)
    devolver dispersion
```

Esta función valora como de dispersa es una solución. El objetivo será encontrar aquel conjunto seleccionados que nos de el mayor valor de dispersión que calcula esta función. Esta función es también llamada la función fitness, y nos limita el número de iteraciones que hay en los algoritmos basados en poblaciones.

Operadores comunes.

```
inicializarPoblacion():
      para cada i < poblacion.tam()
            para cada j < poblacion[i].tam()
                  poblacion[i][j] = false

      para cada i < poblacion.tam()
            n_selec = 0
            mientras n_selec < m
                  indice = random(0,poblacion[i].tam())
                  si poblacion[indice] == false
                        poblacion[indice] = true
                        n_selec++

      para cada i < poblacion.tam()
            fitness[i] = calcularDispersion(poblacion[i])
```

```
obtenerCrontribucion(indice: entero, solucion: vector bool): numero real
      contribucion = 0
      para cada i < solucion.tam()
            si solucion[i] == true
                  contribucion += distancia(indice,i)
      devolver contribucion

seleccionar():
      //AGEs generacion.tam() = 2
      //Resto generacion.tam() = poblacion.tam()
      para cada i < generacion.tam()
            candidato1 = random(0,poblacion.tam())
            candidato2 = random(0,poblacion.tam())
            si fitness[candidato1] > fitness[candidato2]
                  generacion[i] = poblacion[candidato1]
            si no
                  generacion[i] = poblacion[candidato2]

cruzar():
      //En AGEs prob_cruce = 1 y 0.7 en el resto
      n_cruces = generacion.tam() * prob_cruce * 0.5
      para cada i < n_cruces
            indice = i*2
            //cruce_posicion en AGs que usan el otro cruce
            hijo1 =
cruce_uniforme(generacion[indice],generacion[indice+1])
            hijo2 =
cruce_uniforme(generacion[indice],generacion[indice+1])
            generacion[indice] = hijo1
```

```
            generacion[indice+1] = hijo2


//Para AGGs y AMs
mutar():
      n_mutac = prob_mutac * n
      para cada i < n_mutac
            indice = random(0,poblacion.tam())
            gen1 = random(0,n)
            gen2 = random(0,n)
            mientras generacion[indice][gen1] == generacion[indice][gen2]
                  gen2 = random(0,n)
            intercambiar(generacion[indice][gen1],
            generacion[indice][gen2])




//Mutacion de los AGEs
mutar():
      para cada i < generacion.tam() //generacion.tam() = 2
            si random(0.0,1.0) < prob_mutar //prob_mutar = 0.1
                  gen1 = random(0,n)
                  gen2 = random(0,n)
                  mientras generacion[indice][gen1] ==
generacion[indice][gen2]
                        gen2 = random(0,n)
                  intercambiar(generacion[indice][gen1],
                  generacion[indice][gen2])
```

```
/*  Si tenemos en cuenta que usamos una representacion binaria:

SI NO SE COMPARTE EL GEN, El gen en un padre será 1 y en el otro 0 ->
Esto es lo miso que escoger aleatoriamente 1 o 0, ya que escogeriamos
aleatoriamente un padre o otro. SI SE COMPARTE EL GEN, El gen sera el de
los dos.

1- El algoritmo va a rellenar aleatoriamete el hijo para rellenar las
posiciones donde no se comparte el gen.

2- A posteriori algoritmo va a cambiar los genes que coincidan en ambos
padres al que corresponda.

3- Por ultimo llama al reparador.

El resultado es el mismo pero mas eficiente porque genera menos
aleatorios si utiliza los bits del aleatorio como genes no compartidos.
Importante por que aquí esta el cuello de botella */

cruce_uniforme(padre1: vector bool, padre2: vector bool): vector bool
      hijo = [], contribuciones = []
      //Rellenar hijo
      mientras !hijo.completo()
            aleatorio = random()
            para cada bit en aleatorio
                  hijo.insertar(bool(bit))
      //Insertar genes comunes
      para cada i < n
            si padre1[i] == padre2[i]
                  hijo[i] = padre1[i]
      //Contamos elementos seleccionados
      v = 0
      para cada i < n
            si hijo[i] == true
                  v++
```

```
      //Reparamos
      si v < m
            para cada  i < n
                  si !hijo[i]
                        contribuciones[i] = calcularContribucion(i,hijo)
                  mientras v < m
                        mci = max_element(contribuciones)
                        contribuciones[mci] = true
                        v++
                        para cada j < contribuciones.tam()
                              si !hijo[j]
                                    contribuciones[j] += distancia[j][mci]

      si no
            si v > m
                  para cada  i < n
                        si hijo[i]
                              contribuciones[i] =
calcularContribucion(i,hijo)
                        mientras v > m
                              mci = max_element(contribuciones)
                              contribuciones[mci] = false
                              v--
                              para cada j < contribuciones.tam()
                                    si hijo[j]
                                          contribuciones[j] -=
distancia[j][mci]


      devolver hijo
```

```
cruce_posicion(padre1: vector bool, padre2: vector bool): vector bool
      hijo = vector(n)
      resto = []
      indices = []

      para cada i < n
            //Insertar genes comunes
            si padre1[i] == padre2[i]
                  hijo[i] = padre1[i]
            si no
                  //Almacenar genes del padre1 y los huecos del hijo
                  resto.insertar(padre1[i])
                  indices.insertar(i)

      barajar(resto)

      //Insertar resto de genes del primer padre
      para cada i < resto.tam()
            hijo[indices[i]] = resto[i]

      devolver hijo
```

3. Algoritmos

Pseudocódigo de AGGs y AGEs

```
algoritmoGenetico(): vector de bool
      inicializarPoblacion()
      //eval_por_it = 50 -> AGG; eval_por_it = 2 -> AGE;
      para cada i < (100000/eval_por_it)
            seleccionar()
            cruzar()
            mutar()
            remplazar()

      // En AGEs estan ordenados por fitness y se coge el primero
      mfi = max_element(fitness)

      devolver poblacion[mfi]//Funcion de remplazar de AGGs y AMs

//Funcion para AGGs y AMs
remplazar():
      mfi = max_element(fitness) //mfi indice del individuo de mayor
fitness
      generacion[random(0,generacion.tam()] = poblacion[mfi]
      poblacion = generacion
      barajar(poblacion) //Importante sobretodo en AMs

      //Actualizar el fitness de los individuos de la nueva población
      para cada i < poblacion.tam()
            fitness[i] = calcularDispersion(poblacion[i])
```

```
//Funcion de remplazar de AGEs: los mantiene ordenados
remplazar():
      fcf = calcularDispersion(generacion[0])
      scf = calcularDispersion(generacion[1])

      indice = lower_bound(fitness, fcf)
      si indice != null
            para cada i < indice-1
                  fitness[i-1] = fitness[i]
                  poblacion[i-1] = fitness[i]
            poblacion[indice] = generacion[0]
            fitness[indice] = fcf

      indice = lower_bound(fitness, scf)
      si indice != null
            para cada i < indice-1
                  fitness[i-1] = fitness[i]
                  poblacion[i-1] = fitness[i]
            poblacion[indice] = generacion[1]
            fitness[indice] = sfc
```

Pseudocódigo de AMs

Los algoritmos meméticos parten de los AGG con cruze uniforme, por tanto las funciones de inicilizarPoblacion, seleccionar, cruzar, mutar y remplazar serán exacatamente las mismas.

```
algoritmoGenetico(): vector de bool
    inicializarPoblacion()
    //eval_por_it = 51 ≈ 50 -> AM10-0.1 y AM10-0.1mejor
    //eval_por_it = 60 -> AM10-1.0 -> ¡Por error usé 52!

    para cada i < (100000/eval_por_it)
        seleccionar()
        cruzar()
        mutar()
        remplazar()
        si i mod 10 == 0
            //En AM10-0.1mejor se ordena por fitness la poblacion
            //Tambien ordenaría el vector fitness
            ordenar(poblacion,fitness) //SOLO en AM10-0.1mejor
            //tam_sub = 1.0 -> AM10-1.0 y tam_sub = 0.1 -> AM10-0.1
            para cada i < (poblacion.tam() * tam_sub)
                //BusquedaLocal solo llama una vez a la funcion
                evaluar (evaluar es la funcion calcularDispersion)
                poblacion[i] = busquedaLocal(poblacion[i])
                fitness[i] = calcularDispersion(poblacion[i])

    mfi = max_element(fitness)

    devolver poblacion[mfi]
```

```
busquedaLocal(solucion: vector de booleanos): lista
   solucion = convertirAConjuntoEnteros(solucion)
   dispersion = calcularDispersion() //Unica llamada a evaluar
      mientras item_a_sacar <= sol_sin_sacar.ultimo y it < MAX
         ordenados = ordenarSeleccionados(solucion)
         mientras delta<0 y item_a_sacar<=sol_sin_sacar.ultimo y it<MAX
            item_a_sacar = siguiente(ordenados)
            cont_a_sacar = calculaContribucion(item_a_sacar,solucion)
            sol_sin_a_sacar = solucion.sin(item_a_sacar)
            mientras delta<0 y item_a_insertar<=resto.ultimo y it<MAX
               item_a_insertar = siguiente(resto)
               cont_a_insertar =  calculaContribucion(item_a_insertar,
               ...sol_sin_a_sacar)
               delta = cont_a_sacar - cont_a_insertar
               it++
          si delta > 0
             solucion.intercambiar(item_a_sacar,item_a_insertar)
             dipersion += delta
   devolver convertirACodificacionBinaria(solucion)
```

```
convertirAConjuntoEnteros(solucion: vector de bool): conjunto enteros
      sol_enteros = []
      para cada i < solucion.tam()
            si solucion[i] == true
                  sol_enteros.insertar(i)


convertirACodificacionBinaria(solucion: conjunto enteros): vector de bool
      sol_bool = [false, ... , false]
      para cada i en solucion
            sol_bool[i] = true
```

4. Análisis

Informacion del repositorio

En la carpeta *out* se encontrarán los resultados obtenidos automaticamente con el script de la carpeta raiz. También se compilan automaticamente los fichero fuente haciendo *make* sobre el directorio raiz y con la ordena *make clean* se borran los ejecutables. Para replicar los resultados hay que usar la semilla *531*, que en esta practica si ha sido fijada. Los programas tienen dos argumentos de entrada el primero el fichero de datos de entrada y el segunda la semilla. Un ejemplo de ejecucion es:

```
> ./agg-uniforme data\MDG-a_1_n500_m50.txt 531
```

Casos de problemas

Para el análisis tenemos tres tipos de casos de problemas: MDG-a es el primero con un tamaño n=500 y m=50, MDG-b es el segundo con n=2000 y m=200 y el último es MDG-c con un tamaño n=3000 y una m=(de 300 a 600). Por cada caso de problemas tenemos 10 matrices de distancias aleatorias, que seran leidas por el programa y que seran utilizadas por el algoritmo para generar una solucion.

Nuestro análisis se basa en la ejecución de estos ejemplos, de los que disponemos de las soluciones óptimas, y con los que podremos comparar nuestros resultados para hacer nuestro análisis. Así por tanto tendremos una tabla con 4 columnas: la primera el nombre del ejemplo, la segunda los resultados del coste obtenido, la tercera es la desviación respecto a la solución optima y la última es el tiempo que tarda la función en encontrar la solución.

Resultados obtenidos

AGG-uniforme

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7683.6900	1.92	10150191.00
MDG-a_2_n500_m50	7657.2500	1.47	9829622.00
MDG-a_3_n500_m50	7675.0500	1.09	11019512.00
MDG-a_4_n500_m50	7678.9900	1.17	9896829.00
MDG-a_5_n500_m50	7740.0600	0.20	10893071.00
MDG-a_6_n500_m50	7652.0600	1.56	10032742.00
MDG-a_7_n500_m50	7571.0100	2.58	9912819.00
MDG-a_8_n500_m50	7628.9500	1.57	9699257.00
MDG-a_9_n500_m50	7724.3800	0.59	9112732.00
MDG-a_10_n500_m50	7663.9400	1.50	10365087.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	10969300.0000	2.93	199095038.00
MDG-b_22_n2000_m200	10999800.0000	2.54	199745140.00
MDG-b_23_n2000_m200	10996300.0000	2.69	199762802.00
MDG-b_24_n2000_m200	10997100.0000	2.60	190432824.00
MDG-b_25_n2000_m200	10989200.0000	2.72	198110207.00
MDG-b_26_n2000_m200	11015100.0000	2.45	184646723.00
MDG-b_27_n2000_m200	11015000.0000	2.57	210240765.00
MDG-b_28_n2000_m200	11005100.0000	2.44	231211107.00
MDG-b_29_n2000_m200	11064600.0000	2.06	203363184.00
MDG-b_30_n2000_m200	11012600.0000	2.51	189145069.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24118900	3.08	566015808.00
MDG-c_2_n3000_m300	24051300	3.43	521172979.00
MDG-c_8_n3000_m400	42219900	2.80	534495041.00
MDG-c_9_n3000_m400	42345300	2.52	521525889.00
MDG-c_10_n3000_m400	42346500	2.60	483938286.00
MDG-c_13_n3000_m500	65609900	2.10	511228891.00
MDG-c_14_n3000_m500	65534100	2.16	508963152.00
MDG-c_15_n3000_m500	65631100	2.03	481518622.00
MDG-c_19_n3000_m600	93942900	1.77	525999349.00
MDG-c_20_n3000_m600	93848200	1.88	620320702.00

Media Desv:	2.12
Media Tiempo: (us)	246061448.00
Media Tiempo: (seg)	246.061448

AGG-posicion

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7654.9000	2.28	480452.00
MDG-a_2_n500_m50	7547.6800	2.88	509040.00
MDG-a_3_n500_m50	7519.0400	3.10	508002.00
MDG-a_4_n500_m50	7610.6400	2.05	482490.00
MDG-a_5_n500_m50	7407.0100	4.49	478647.00
MDG-a_6_n500_m50	7527.7800	3.16	479164.00
MDG-a_7_n500_m50	7562.0100	2.70	487009.00
MDG-a_8_n500_m50	7566.4100	2.38	520020.00
MDG-a_9_n500_m50	7596.5400	2.23	529452.00
MDG-a_10_n500_m50	7632.0600	1.91	483483.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	10628300.0000	5.94	4498592.00
MDG-b_22_n2000_m200	10622000.0000	5.89	4340460.00
MDG-b_23_n2000_m200	10609100.0000	6.11	4384085.00
MDG-b_24_n2000_m200	10618400.0000	5.96	4432720.00
MDG-b_25_n2000_m200	10621100.0000	5.98	4340042.00
MDG-b_26_n2000_m200	10611900.0000	6.03	4344922.00
MDG-b_27_n2000_m200	10611400.0000	6.14	4362865.00
MDG-b_28_n2000_m200	10613800.0000	5.91	4345274.00
MDG-b_29_n2000_m200	10637800.0000	5.84	4299174.00
MDG-b_30_n2000_m200	10635000.0000	5.86	4443383.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	23460100	5.72	15115769.00
MDG-c_2_n3000_m300	23489800	5.68	13505233.00
MDG-c_8_n3000_m400	41404000	4.68	25190945.00
MDG-c_9_n3000_m400	41305700	4.91	25698751.00
MDG-c_10_n3000_m400	41418000	4.73	21219578.00
MDG-c_13_n3000_m500	64386800	3.92	33002240.00
MDG-c_14_n3000_m500	64332900	3.95	35777888.00
MDG-c_15_n3000_m500	64466100	3.77	38203881.00
MDG-c_19_n3000_m600	92365000	3.42	67665293.00
MDG-c_20_n3000_m600	92421900	3.37	2843734623.00

Media Desv:	4.37
Media Tiempo: (us)	105595449.23
Media Tiempo: (seg)	105.5954492

AGE-uniforme

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7619.4900	2.74	767915.00
MDG-a_2_n500_m50	7570.8300	2.58	877573.00
MDG-a_3_n500_m50	7503.0800	3.30	784187.00
MDG-a_4_n500_m50	7647.0300	1.59	765555.00
MDG-a_5_n500_m50	7589.3300	2.14	952097.00
MDG-a_6_n500_m50	7572.4500	2.59	795928.00
MDG-a_7_n500_m50	7544.8000	2.92	776986.00
MDG-a_8_n500_m50	7685.3600	0.85	809021.00
MDG-a_9_n500_m50	7543.3800	2.92	814087.00
MDG-a_10_n500_m50	7556.4300	2.88	812394.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	11106400.0000	1.71	13693755.00
MDG-b_22_n2000_m200	11110100.0000	1.57	13794178.00
MDG-b_23_n2000_m200	11080900.0000	1.94	11891150.00
MDG-b_24_n2000_m200	11045700.0000	2.17	13650715.00
MDG-b_25_n2000_m200	11074100.0000	1.96	12921480.00
MDG-b_26_n2000_m200	11131900.0000	1.42	12929545.00
MDG-b_27_n2000_m200	11131200.0000	1.54	13289784.00
MDG-b_28_n2000_m200	11041900.0000	2.11	12398470.00
MDG-b_29_n2000_m200	11077700.0000	1.94	12841550.00
MDG-b_30_n2000_m200	11085100.0000	1.87	13683626.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24532000	1.41	37355224.00
MDG-c_2_n3000_m300	24482500	1.70	40212205.00
MDG-c_8_n3000_m400	42846400	1.36	58007991.00
MDG-c_9_n3000_m400	42771100	1.53	56340880.00
MDG-c_10_n3000_m400	42870200	1.39	53228285.00
MDG-c_13_n3000_m500	66220000	1.18	76223612.00
MDG-c_14_n3000_m500	66218400	1.14	75835012.00
MDG-c_15_n3000_m500	66398400	0.89	83153777.00
MDG-c_19_n3000_m600	94723500	0.95	103087192.00
MDG-c_20_n3000_m600	94689400	1.00	118970907.00

Media Desv:	1.84
Media Tiempo: (us)	28055502.70
Media Tiempo: (seg)	28.0555027

AGE-posicion

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7606.4300	2.90	458592.00
MDG-a_2_n500_m50	7531.2000	3.09	455315.00
MDG-a_3_n500_m50	7549.2100	2.71	462085.00
MDG-a_4_n500_m50	7583.7100	2.40	462868.00
MDG-a_5_n500_m50	7576.6400	2.30	464691.00
MDG-a_6_n500_m50	7543.1200	2.97	498577.00
MDG-a_7_n500_m50	7602.4900	2.18	489452.00
MDG-a_8_n500_m50	7576.1300	2.25	465656.00
MDG-a_9_n500_m50	7482.3600	3.70	454895.00
MDG-a_10_n500_m50	7546.0000	3.01	469109.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	10958300.0000	3.02	4142286.00
MDG-b_22_n2000_m200	11016900.0000	2.39	4178278.00
MDG-b_23_n2000_m200	10988100.0000	2.76	3974655.00
MDG-b_24_n2000_m200	11014900.0000	2.44	3925349.00
MDG-b_25_n2000_m200	10997800.0000	2.64	3975926.00
MDG-b_26_n2000_m200	11035700.0000	2.27	3829929.00
MDG-b_27_n2000_m200	11019800.0000	2.53	3908943.00
MDG-b_28_n2000_m200	10995700.0000	2.52	4053709.00
MDG-b_29_n2000_m200	10990200.0000	2.72	3783485.00
MDG-b_30_n2000_m200	10954800.0000	3.02	3711837.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24282100	2.42	8611386.00
MDG-c_2_n3000_m300	24268200	2.56	8321861.00
MDG-c_8_n3000_m400	42557800	2.02	17436660.00
MDG-c_9_n3000_m400	42531700	2.09	15942423.00
MDG-c_10_n3000_m400	42486300	2.28	17078672.00
MDG-c_13_n3000_m500	65872500	1.70	28351554.00
MDG-c_14_n3000_m500	65775700	1.80	28700188.00
MDG-c_15_n3000_m500	65841700	1.72	27971402.00
MDG-c_19_n3000_m600	94246900	1.45	43510025.00
MDG-c_20_n3000_m600	94133500	1.58	45460966.00

Media Desv:	2.45
Media Tiempo: (us)	9518359.13
Media Tiempo: (seg)	9.518359133

AM-10-0.1

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7601.1100	2.97	11662399.00
MDG-a_2_n500_m50	7572.8000	2.56	11152020.00
MDG-a_3_n500_m50	7702.5300	0.73	10884183.00
MDG-a_4_n500_m50	7676.0900	1.21	10876183.00
MDG-a_5_n500_m50	7605.7200	1.93	10696260.00
MDG-a_6_n500_m50	7645.2700	1.65	12066444.00
MDG-a_7_n500_m50	7680.3200	1.18	11175787.00
MDG-a_8_n500_m50	7679.5700	0.92	12896846.00
MDG-a_9_n500_m50	7587.1200	2.35	11927930.00
MDG-a_10_n500_m50	7617.8900	2.09	10871114.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	11070200.0000	2.03	254436971.00
MDG-b_22_n2000_m200	11030600.0000	2.27	246952259.00
MDG-b_23_n2000_m200	11072600.0000	2.01	253921309.00
MDG-b_24_n2000_m200	11057200.0000	2.07	255226253.00
MDG-b_25_n2000_m200	11083500.0000	1.88	256161603.00
MDG-b_26_n2000_m200	11107700.0000	1.63	247732349.00
MDG-b_27_n2000_m200	11012300.0000	2.59	264192140.00
MDG-b_28_n2000_m200	11056000.0000	1.99	253778253.00
MDG-b_29_n2000_m200	11044800.0000	2.23	255396143.00
MDG-b_30_n2000_m200	11046600.0000	2.21	236716595.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24316600	2.28	631500892.00
MDG-c_2_n3000_m300	24267900	2.56	629200904.00
MDG-c_8_n3000_m400	42618400	1.89	653891328.00
MDG-c_9_n3000_m400	42515200	2.12	654893030.00
MDG-c_10_n3000_m400	42512200	2.22	668473568.00
MDG-c_13_n3000_m500	65929000	1.62	749235330.00
MDG-c_14_n3000_m500	65808200	1.75	763053962.00
MDG-c_15_n3000_m500	65967700	1.53	768540952.00
MDG-c_19_n3000_m600	94197400	1.50	850940046.00
MDG-c_20_n3000_m600	94243200	1.46	853029552.00

Media Desv:	1.91
Media Tiempo: (us)	328716086.83
Media Tiempo: (seg)	328.7160868

AM-10-0.1 mejor

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7656.9300	2.26	10926189.00
MDG-a_2_n500_m50	7645.5900	1.62	11907215.00
MDG-a_3_n500_m50	7596.9800	2.09	10961346.00
MDG-a_4_n500_m50	7625.8400	1.86	11596717.00
MDG-a_5_n500_m50	7709.9800	0.58	12384263.00
MDG-a_6_n500_m50	7606.5000	2.15	16193005.00
MDG-a_7_n500_m50	7640.6100	1.69	16403216.00
MDG-a_8_n500_m50	7696.1400	0.71	14418430.00
MDG-a_9_n500_m50	7562.1300	2.68	14478257.00
MDG-a_10_n500_m50	7654.7300	1.61	10247243.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	11098900.0000	1.78	331276469.00
MDG-b_22_n2000_m200	11097200.0000	1.68	340017323.00
MDG-b_23_n2000_m200	11090200.0000	1.86	272607185.00
MDG-b_24_n2000_m200	11134400.0000	1.39	264852829.00
MDG-b_25_n2000_m200	11065300.0000	2.04	281210688.00
MDG-b_26_n2000_m200	11090600.0000	1.79	225529600.00
MDG-b_27_n2000_m200	11047300.0000	2.29	258873013.00
MDG-b_28_n2000_m200	11020700.0000	2.30	272068960.00
MDG-b_29_n2000_m200	11054300.0000	2.15	269703260.00
MDG-b_30_n2000_m200	11084200.0000	1.88	255040509.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24336600	2.20	612668023.00
MDG-c_2_n3000_m300	24330400	2.31	706373295.00
MDG-c_8_n3000_m400	42648700	1.82	680809540.00
MDG-c_9_n3000_m400	42756600	1.57	699646854.00
MDG-c_10_n3000_m400	42684900	1.82	660183091.00
MDG-c_13_n3000_m500	66102400	1.36	724916887.00
MDG-c_14_n3000_m500	66126400	1.27	719164165.00
MDG-c_15_n3000_m500	65949100	1.56	646356031.00
MDG-c_19_n3000_m600	94495100	1.19	770276074.00
MDG-c_20_n3000_m600	94335500	1.37	861459079.00

Media Desv:	1.76
Media Tiempo: (us)	332751625.20
Media Tiempo: (seg)	332.7516252

AM-10-1.0

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-a_1_n500_m50	7643.4500	2.43	23996023.00
MDG-a_2_n500_m50	7582.4000	2.44	22956684.00
MDG-a_3_n500_m50	7645.1800	1.47	23378317.00
MDG-a_4_n500_m50	7689.5900	1.04	22371604.00
MDG-a_5_n500_m50	7631.6500	1.59	24475600.00
MDG-a_6_n500_m50	7637.0500	1.76	25164532.00
MDG-a_7_n500_m50	7713.2300	0.75	26614430.00
MDG-a_8_n500_m50	7699.0200	0.67	24450354.00
MDG-a_9_n500_m50	7671.8300	1.26	23176651.00
MDG-a_10_n500_m50	7639.0100	1.82	25015013.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-b_21_n2000_m200	11082500.0000	1.92	592539994.00
MDG-b_22_n2000_m200	11094200.0000	1.71	614029617.00
MDG-b_23_n2000_m200	11151900.0000	1.31	645273584.00
MDG-b_24_n2000_m200	11071000.0000	1.95	636728316.00
MDG-b_25_n2000_m200	11046300.0000	2.21	656037960.00
MDG-b_26_n2000_m200	11078700.0000	1.89	541647172.00
MDG-b_27_n2000_m200	11143200.0000	1.44	595409907.00
MDG-b_28_n2000_m200	11012300.0000	2.37	516951958.00
MDG-b_29_n2000_m200	11069800.0000	2.01	570155555.00
MDG-b_30_n2000_m200	11107200.0000	1.67	517707279.00

Caso	Coste obtenido	Desv	Tiempo (us)
MDG-c_1_n3000_m300	24409000.0000	1.91	1493655506.00
MDG-c_2_n3000_m300	24399900.0000	2.03	1448175913.00
MDG-c_8_n3000_m400	42779000.0000	1.52	1972248315.00
MDG-c_9_n3000_m400	42781800.0000	1.51	2096227116.00
MDG-c_10_n3000_m400	42717000.0000	1.75	2132342980.00
MDG-c_13_n3000_m500	66040900.0000	1.45	2936227694.00
MDG-c_14_n3000_m500	66164900.0000	1.22	3568570402.00
MDG-c_15_n3000_m500	66066200.0000	1.38	3337638931.00
MDG-c_19_n3000_m600	94413200.0000	1.28	3551448301.00
MDG-c_20_n3000_m600	94401900.0000	1.30	3780471728.00

Media Desv:	1.64
Media Tiempo: (us)	1081502914.53
Media Tiempo: (seg)	1081.502915

Comparación de resultados

Algoritmo	Desv Media	Tiempo Medio (us)	Tiempo Medio (s)
Greedy	9.22	797745.80	0.80
BusquedaLocal	2.96	3503571.93	3.50
AGG-uniforme	2.12	246061448.00	246.06
AGG-posicion	4.37	105595449.23	105.60
AGE-uniforme	1.84	28055502.70	28.06
AGE-posicion	2.45	9518359.13	9.52
AM-10-0.1	1.91	328716086.83	328.72
AM-10-0.1mejor	1.76	332751625.20	332.75
AM-10-1.0	1.64	1081502914.53	1081.50

Estudio de la desviación media

Algoritmo	MDG-a	MDG-b	MDG-c
Greedy	12.15	9.05	6.46
BusquedaLocal	2.25	3.39	3.25
AGG-uniforme	1.37	2.55	2.44
AGG-posicion	2.72	5.96	4.42
AGE-uniforme	2.45	1.82	1.26
AGE-posicion	2.75	2.63	1.96
AM-10-0.1	1.76	2.09	1.89
AM-10-0.1mejor	1.73	1.91	1.65
AM-10-1.0	1.52	1.85	1.53

Estudio de los tiempos de ejecución (seg)

Algoritmo	MDG-a	MDG-b	MDG-c
Greedy	0.00	0.32	2.07
BusquedaLocal	0.07	1.62	8.83
AGG-uniforme	10.09	200.58	527.52
AGG-posicion	0.50	4.38	311.91
AGE-uniforme	0.82	13.11	70.24
AGE-posicion	0.47	3.95	24.14
AM-10-0.1	11.42	252.45	722.28
AM-10-0.1mejor	12.95	277.12	708.19
AM-10-1.0	24.16	588.65	2631.70

Como lectura general podemos decir que los algoritmos implementados en esta práctica, los algoritmos basados en poblaciones, mejoran nuestros resultados en cuanto a calidad de la solución, solo en el caso de AGG con cruce basado en posición la solución empeora y más adelante estudiaremos los motivos. Por otra parte, los tiempos medios de ejecución de estos algoritmos se disparan: hasta 70 veces más lentos en el peor de los casos de los Algoritmos Genéticos y mas de 300 veces en el caso de los Algoritmos Meméticos.

Teniendo en cuenta estos datos generales observamos que el cruce es un factor determinante para obtener una buena solución, y en este caso el mejor es el cruce uniforme. Tanto para los algoritmos generacionales como para los estacionarios este cruce obtiene mejores resultados que el basado en posición e intuyo el motivo. El uniforme utiliza la información de ambos padres por igual favoreciendo la diversidad, mientras que el cruce basado en

posición favorece a un padre y genera dos hijos muy parecidos (deberia haber hecho las llamadas al cruce_posicion la primera favoreciendo al padre1 y otra favoreciendo al padre2 para que los padres no fueran tan parecidos aunque tampoco llegarian a la calidad de los cruce uniforme). Los tiempos favorecen al cruce basado en posicion y es que en el uniforme la funcion de reparar es cuello de botella y consume la mayor parte del tiempo. Sin embargo la diferencia de tiempos entre los cruces no creo que sea un gran problema para estos tamaños de problemas. Para los algoritmos meméticos por tanto este será el cruce que se use.

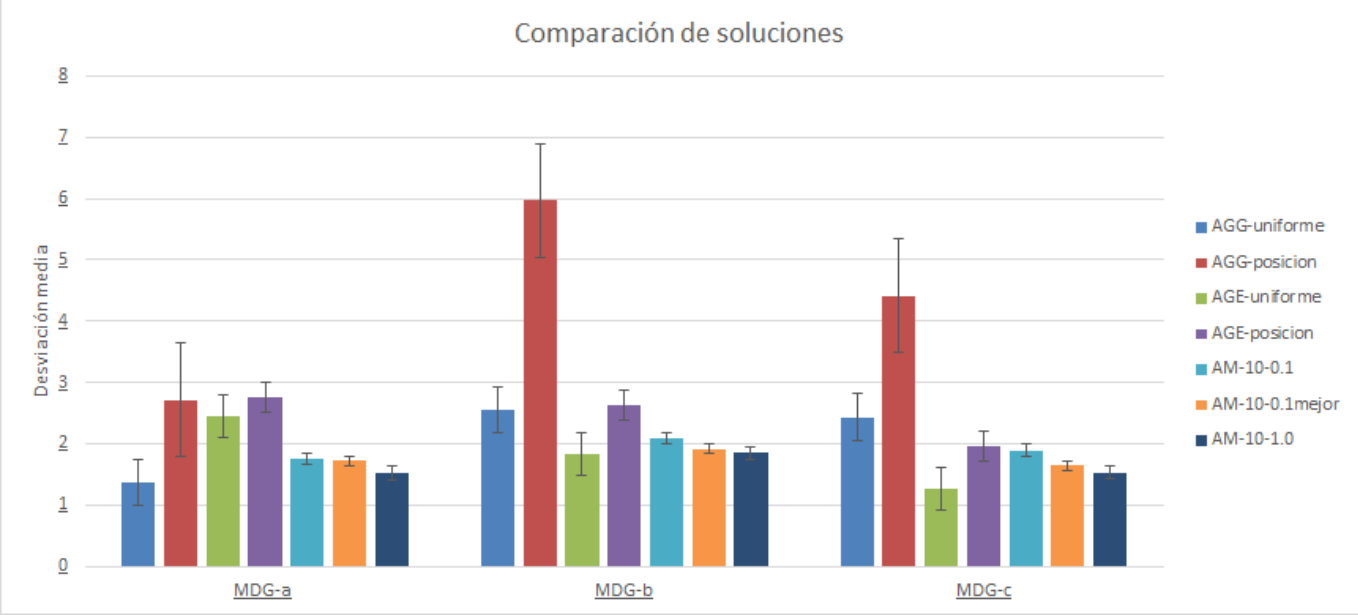

Figura 1. Comparación de desviaciones.

Respecto a los algoritmos estacionarios me ha sorprendido bastante los resultados obtenidos. En general mejoran a los generacionales tanto en las soluciones como en los tiempos, viendo la *Figura 1* observamos que el comportamiento depende del tamaño del dataset, y es en los dataset más grandes donde se comporta aún mejor. Si consultamos la *Figura 2* vemos que son los que mejor se comportan en tiempo. Asi que el comportamiento elitista en este caso nos ayuda a obtener unas soluciones mejores para este numero de iteraciones.

Por ultimo los algoritmos meméticos que de media se comportan muy bien, pero son lentisimos, asi que quizas se podrían haber optimizado más. Como he dicho de media se comportan muy bien respecto a la solución y no hay grandes diferencias entre los 3, además los errores son muy pequeños asi que podemos decir que son muy estables y se generarán buenas soluciones para cualquier dataset.

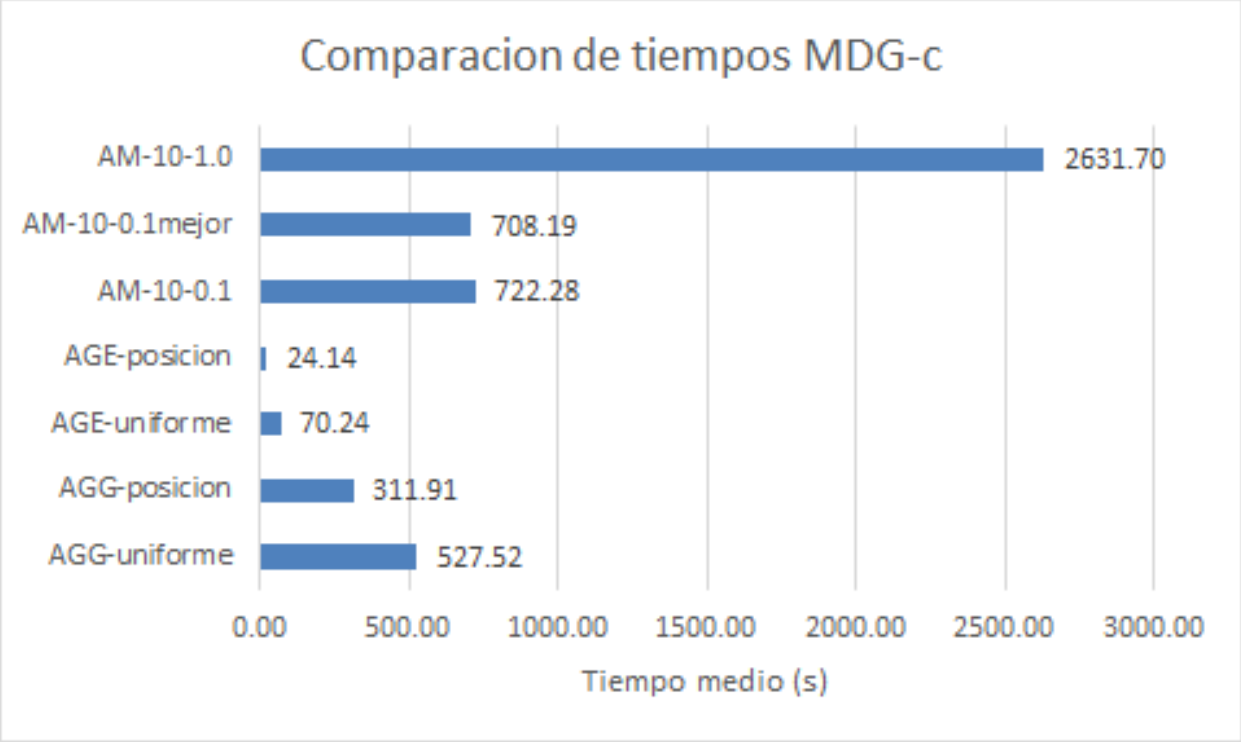

Figura 2. Tiempos MDG-c

El mayor problema de estos algoritmos es el tiempo, son lentisimos y no son recomendables cuando tenemos exigencias en el tiempo. En concreto el AM-10-1.0 ha sido el algoritmo más lento aunque era algo esperado la lentitud de estos algoritmos, ya que parten de los AGGs que ya son lentos y ademas hacen una busqueda local, en el caso de AM-10-1.0 a toda la poblacion cada 10 iteraciones.

Para terminar si tuviera que quedarme con uno de estos algoritmos sería el AGE-posicion, por que me ha sorprendido mucho que obtenga la mejor solución con los MDG-c y ademas obtiene la solución bastante rapido. Asi que sería un buen sustituto a la busqueda local si las exigencias de tiempos lo permiten. Como algoritmo a descartar el AGG-posicion que obtiene resultados peores que la busqueda local.