Practica 2.b
Técnicas de Búsqueda basadas en Poblaciones para el Problema del Agrupamiento con Restricciones

Curso 2019-2020

Tercer Curso del Grado en Ingeniería Informática

Metaheurísticas

Rafael Vázquez Conejo
DNI: 49137372B
Email: rafavazquez99@correo.ugr.es
Grupo Prácticas 1, miércoles.

1. Índice

2. Descripción del Problema del PAR

El problema del Agrupamiento con Restricciones (PAR) consiste en una generalización del agrupamiento clásico, permitiendo la incorporación de un nuevo tipo de información (restricciones) al proceso de agrupamiento. Nuestro problema trata de optimizar la clasificación de un conjunto de datos recibidos “X” con “n” instancias cada uno en particiones C del mismo, de manera que se minimice la desviación general y se cumplan las restricciones establecidas en el conjunto de restricciones R.

Es decir, para decidir que cluster $c \in C = c_1, \ldots, c_{1k}$ le asignamos a cada instancia de nuestro conjunto de datos X tenderemos en cuenta que la distancia de nuestra instancia al cluster sea lo más mínima posible, del mismo modo el número de restricciones que incumplimos al asignar dicha instancia al cluster sea lo menor posible.

Por lo tanto respecto a las restricciones de instancia podemos establecer que dada una pareja de instancias se establece una restricción del tipo Must-Link (ML), si estas instancias debe pertenecer al mismo cluster, o del tipo Cannot-Link (CL), si estas no pueden pertenecer al mismo cluster. Respecto a las restricciones de distancia, las instancias separadas por una distancia mayor a una dada deben pertenecer a diferentes clusters, a su vez las instancias separadas por una distancia menor que una dada deben pertenecer al mismo cluster.

Existen dos variantes del problema respecto al modo de interpretar las restricciones. Si todas las restricciones deben satisfacerse en la partición C de nuestro conjunto de datos X, nos encontramos ante Restricciones fuertes (Hard), o si buscamos que la partición C de nuestro conjunto de datos X debe minimizar el número de restricciones incumplidas pero puede incumplir algunas de ellas, este es el caso de Restricciones débiles (Soft). En nuestro caso nos encontramos ante la segunda variante del problema, restricciones débiles.

El valor que debemos buscar minimizar para encontrar el mejor cluster posible para cada instancia de X, es decir, nuestra función de valoración será:

$$f = \vec{C} + (\text{infeasability} * \lambda)$$

$$\overrightarrow{\mu_i} = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} \overrightarrow{x_j} \qquad \overline{c_i} = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} \|\overrightarrow{x_j} - \overrightarrow{\mu_i}\|_2 \qquad \overline{C} = \frac{1}{k} \sum_{c_i \in C} \overline{c_i}$$

$$infeasibility = \sum_{i=0}^{|ML|} \mathbb{1}(h_C(\overrightarrow{ML_{[i,1]}}) \neq h_C(\overrightarrow{ML_{[i,2]}})) + \sum_{i=0}^{|CL|} \mathbb{1}(h_C(\overrightarrow{CL_{[i,1]}}) = h_C(\overrightarrow{CL_{[i,2]}}))$$

Los parámetro de estas fórmula son los siguientes:

- $\vec{C}$, la desviación general de la partición C, se calcula como la media de las desviaciones intra-cluster, siendo a su vez la distancia media intra-cluster c_i la media de las distancias de las instancias que conforman el cluster con su centroide .
- infeasability, es el número de restricciones que se incumplen en C.
- λ, cociente entre la máxima distancia existente en el conjunto de datos y el número de restricciones presentes en el problema, |R|.

3. Descripción de los elementos y algoritmos comunes de los algoritmos.

3.1 Representación de los datos

Comentar que el lenguaje de programación que he utilizado es C++.

Tanto los datos de entrada “X” como las restricciones las he almacenado en un vector de vectores del tipo “double”. Todas las estructuras que he utilizado en la resolución del problema han sido vectores de la clase “vector”, ya que en un primer análisis del problema me parecieron una forma útil de representación de los datos y no de un alto coste computacional, en muchos casos podría haber sustituido un vector por el uso de una lista, pero debido a tener más conocimientos y practica con los vectores he preferido su utilización.

Los datos que he empleado en ambos algoritmos son:

Un vector de vectores del tipo “size_t” en el que almaceno la población generada, es decir, un vector en el que cada posición corresponde a una posible solución a nuestro problema. Cada posición será denominada como cromosoma, y los elementos que forman cada uno de estos se denominan genes. Dicho de otra manera, cada cromosoma es una posible solución a nuestro problema formada por genes que indican el cluster asignado a cada dato del conjunto de datos de entrada “X”.
También defino una variable de tipo “int” llamada “evaluaciones”, la cual me permite llevar la cuenta del número de evaluaciones que realizo sobre mi población, cuyo valor se incremento cada vez que llamo a mi función objetivo.
Mi solución, si nos referimos al cluster asignado para cada instancia, la represento en un vector de “int” en el cual cada valor “i” corresponde al cluster asignado a la instancia “i” del conjunto de datos, llamado “clusters”. De forma que si la posición “i” vale 2, significa que esta instancia está asignada al cluster 2. Esta solución la obtengo valorando las posibles soluciones de mi población y tomando aquella que posee la mejor función objetivo.

3.2. Calcular Violaciones

A continuación establezco una serie de funciones que me ayudarán al calculo de la función objetivo. La primera de éstas me permite obtener el número total de restricciones que se incumplen en el cromosoma o individuo que estamos evaluando.
La función comprueba que si dos elementos están en el mismo cluster y no deberían, según las restricciones, aumenta el número de violaciones. Lo aumenta también si dos elementos no están en el mismo cluster y según las restricciones deberían.

3.3 Mayor Distancia

Presento esta función de nuevo porque en la práctica anterior realicé de forma incorrecta el cálculo. Esta función también es necesaria para la obtención de la función evaluación, esta nos permite obtener tanto la desviación general cómo la mayor distancia de una instancia a un cluster, necesaria para la obtención de λ. Esta hace uso de “*clusters*” para saber el cluster asignado a cada instacia y de “centroides” gracias a la función obtenerCentroides() obtengo los centroides de cada cluster.
En pseudocógido, el algoritmo es el siguiente:

```
FUNCIÓN distanciaMedia(centroides, individuo):
        cluster_actual = 0
        suma_distancia = 0
```

distancia_por_cluster (vector para almacenar la distancia media de cada cluster)
HACER
PARA cada instancia en el conjunto de datos i
SI el cluster de la instancia = cluster_actual
PARA cada componente de la instancia
realizo la suma de la diferencia de cada instancia al centroide
FIN-PARA
FIN-SI
FIN-PARA
distancia = $\sqrt{(centroides[cluster\ actual][componente] - datos[instancia][componente])^2}$
Divido suma_distancia entre el total de elementos del cluster actual
Añado suma distancia al vector distancia_por_cluster
suma_distancia = 0
Incremento el valor de cluster_actual en una unidad
MIENTRAS cluster_actual != total cluster
media = 0
PARA cada distancia en distancia_por_cluster
media más la distancia_por_cluster[distancia]
FIN-PARA
Divido la media entre el número total de clusters
DEVUELVO la distancia maxima, max
FIN

3.4 Total de Violaciones

La última función para poder implementar la función de evaluación, esta recorre las restricciones y devuelve el número total de restricciones establecidas. Es una función sencilla que simplemente suma una unidad si encontramos un 1 o un -1 en las restricciones. Su pseudocógido es sencillo y ya fue proporcionado en la práctica anterior.

3.5 Función Objetivo

La evaluación de la solución se realiza según la formula establecida anteriormente :

$$f = \vec{C} + (infeasability * \lambda)$$

Recordando que $\vec{C}$ será la desviación general que contiene nuestra variable “media”, infeasability es el número de restricciones violadas y nuestra λ será la mayor distancia entre el número total de restricciones en el conjunto. Una vez hemos establecido las funciones anteriores simplemente debemos llamarlas de forma correcta. En pseudocógido, el algoritmo es el siguiente:

Función funcionObjetivo():
mayor_distancia = distanciaMaxima()
violaciones_realizadas = calcularRestricciones(individuo)
n_violaciones = totalViolaciones(individuo)
media = distanciaMedia(centroides, individuo)
landa = cociente mayor_distancia entre n_violaciones
valoración =+ violaciones_realizadas * landa
Devuelvo valoración con la valoración de nuestra solución

Fin

3.6 Generación de soluciones aleatorias

La función “generarPoblacion()” nos permite generar una población formada por cromosomas que poseen soluciones aleatorias a nuestro problema, de manera que cada cromosoma será un vector que en cada posición tendrá un valor de {0, …, k}, siendo k el número total de cluster, definiendo el cluster asociado a cada gen. Esta función recibe como parámetro una variable de tipo “int” que especifica el tipo de algoritmo que ejecutaremos. En el caso de algoritmo Genético:

Si modo = 0 → Estacionario con operador de cruce uniforme.
Si modo = 1 → Estacionario con operador de cruce por segmento fijo.
Si modo = 2 → Generacional con operador de cruce uniforme.
Si modo = 3 → Generacional con operador de cruce por segmento fijo.

En el caso de algoritmo Memético:
Si modo = 0 → AM-(10, 1.0)
Si modo = 1 →AM-(10, 0.1)
Si modo = 2 →AM-(10, 0.1mej)

En pseudocógido, el algoritmo es el siguiente:

```
FUNCIÓN generarPoblacion(modo):
        PARA cada posible solución hasta el tamaño máximo de población establecido
                PARA cada cromosoma
                        Añadir cada gen con un valor aleatorio de [0, n_cluster-1]
                FIN-PARA
        FIN-PARA
        Ejecuto el algoritmo correspondiente (genético o memético)
        Busco la mejor_solución entre los cromosomas obtenidos
        DEVOLVER mejor_solucion
FIN
```

3.6 Operador de Selección

Como operador de seleccioón en los algoritmos genéticos se ha utilizado torneo binario. Se seleccionan de forma aleatoria dos candidatos de la población y se compara su valoración, gracias a la función objetivo definida anteriormente. Aquel cromosoma que sea más prometedor (valoración menor) pasará a ser parte de la población intermedia y el otro será ignorado. Su pseudocódigo es el siguiente:

```
FUNCIÓN torneoBinario(poblacion):
        min = 99999999
        mejor = 0
        cromosoma_1 ← EnteroAleatorio(0, tamañoPoblacion – 1)
        cromosoma_2 ← EnteroAleatorio(0, tamañoPoblacion – 1)          Repetir MIENTRAS
                                                                cromosoma_1 = cromosoma_2
        Compruebo que los cromosomas tomados no sean el mismo
        PARA cada cromosoma de los 2 obtenidos
                valoración = funcionObjetivo(poblacion[cromosoma])
                //Me quedo con el que tenga mejor valoración de los dos
                SI valoración < min
                        min = valoración
                        mejor = cromosoma
                FIN-SI
        FIN-PARA
        DEVOLVER población[mejor]
```

FIN

3.7 Operador de cruce Uniforme

El operador de cruce uniforme nos permite crear un nuevo cromosoma o individuo (hijo) gracias a la combinación de dos individuos (padres) dados. Se combinan seleccionando la mitad de genes de un padre y la mitad del otro. Recibirá como parámetros los dos padres que combinamos. En pseudocógido, el algoritmo es el siguiente:

```
FUNCIÓN cruceUniforme(padre_1, padre_2)
        PARA cada gen de la mitad de genes que forman a un padre
                Añado genes seleccionados de forma aleatoria, Randint(0, padre.size() - 1)
        FIN-PARA
        PARA cada gen del total de genes del cromosoma padre
                //De esta forma añado la mitad de un padre y la mitad del otro
                SI es un gen de los seleccionado anteriormente
                        Añado al hijo el gen correspondiente tomándolo del padre_1
                SINO
                        Añado al hijo el gen correspondiente tomándolo del padre_2
                FIN-SINO
        FIN-PARA
        DEVOLVER evitarClusterVacio(hijo)
FIN
```

La función “evitarClusterVacio()” es una función que comprueba que ningún cluster queda sin ningún elemento asignado. Si detecta que algún cluster está “vacío” toma un elemento aleatorio del total y le asigna ese cluster.

3.8 Operador de cruce por Segmento fijo

El operador de cruce uniforme nos permite crear un nuevo cromosoma o individuo (hijo) gracias a la combinación de dos individuos (padres) dados. Se combinan seleccionando un segmento continuo de características y copiandolo sin modificación en el hijo. Los genes restantes por asignar combinan de manera uniforme características de ambos padres. Recibirá como parámetros los dos padres que combinamos. En pseudocógido, el algoritmo es el siguiente:

```
FUNCIÓN cruceSegmentoFijo(padre_1, padre_2)
        PARA cada gen desde un gen de inicio aleatorio hasta un tamaño de segmento aleatorio
                //Si llego a la última posición del vector y no he completado el tamaño del segmento
                //sigo por el comienzo del vector
                Añado posicion_segmento de los genes siguientes al gen inicial
        FIN-PARA
        PARA cada gen del total de genes del cromosoma padre
                SI el gen no es uno de los que pertenece a posicion_segmento
                        Añado el gen a los restantes
                SINO
                        Añado al hijo el gen correspondiente tomándolo del padre_1
                FIN-SINO
        FIN-PARA
        //Genero un vector con un tamaño igual a la mitad del total de genes restantes
        //Este vector “seleccion” contiene restantes.size()/2 genes elegidos de manera aleatoria
        PARA cada gen del total de genes del cromosoma padre
                SI el gen esta dentro de los genes restantes
```

```
                    SI el gen está dentro del vector de genes “seleccion”
                        Añado al hijo el gen correspondiente tomándolo del padre_1
                    SINO
                        Añado al hijo el gen correspondiente tomándolo del padre_2
                    FIN-SINO
                FIN-SI
        FIN-PARA
        DEVOLVER evitarClusterVacio(hijo)
FIN
```

3.9 Operador de mutación uniforme

Este operador nos permite mutar un gen. Es un algoritmo sencillo que recibe como parámetro el cromosoma hijo obtenido. Su función es generar dos número aleatorios, uno que determina una posición en el hijo, es decir un gen aleatorio del hijo, y otro que corresponde a un nuevo cluster que se le asignará a ese gen elegido aleatoriamente. El único factor a tener en cuenta es que este nuevo cluster seleccionado aleatoriamente debe ser distinto al cluster previamente asignado a nuestro gen aleatorio.

Si aplicamos o no la mutación sobre un cromosoma de la población, está determinado por la probabilidad de mutación establecida y el número de genes que forman los cromosomas. En pseudocógido, el algoritmo es el siguiente:

```
FUNCION mutar(poblacion, posiciones)
        PARA cada cromosoma de la población
                SI dado(probabilidad_de_mutacion * número_de_genes_cromosoma)
                        Añadir a mutados el cromosoma actual mutado por mutación uniforme
                        Añadir a posiciones la posición del cromosoma actual que hemos mutado
                FIN-SI
        FIN-PARA
        DEVOLVER mutados
FIN
```

La función dado() devuelve “true” si nuestro valor de probabilidad es mayor que un valor aleatorio obtenido mediante la función Rand(), que devuelve un número aleatoria real en el intervalo [0, 1].

4. Descripción Algoritmo Genético

Comenzaremos con los algoritmos genéticos presentando su esquema de evolución. Como ya he mencionado en el apartado anterior, encontramos 2 versiones de nuestro algoritmo genético. La primera tiene un esquema generacional con estilismo (AGG) y la otra un esquema estacionario (AGE).

Algoritmo Genético Generacional (AGG)

Una vez definidos todos los operadores y funciones anteriores, la implementación del algoritmo AGG es sencilla. Para su comprensión debemos aclarar las siguientes cuestiones:

- La probabilidad de cruce es de 0.7. Calcularemos a priori el número de cruces que, según la esperanza matemática, deben producirse. Concretamente este número esperado de cruces es "Número de cromosomas de la población / 2 * 0.7". En cada iteración del algoritmo se realizarán el mismo número de cruces y siempre en las mismas posiciones.
- Respecto a las mutaciones, su realización será similar a la manera del apartado anterior. Siempre realizamos el mismo número de mutaciones. En este caso las mutaciones también son calculadas a priori como: Número de mutaciones = "Número de genes por cromosoma * 0.001 (que es la probabilidad de mutación)". En cada mutación se seleccionarán de manera aleatoria tanto el cromosoma como el gen a mutar.
- Este algoritmo incorpora elitismo. En cada nueva población generada se incluirá la mejor solución de la población anterior. Si esta mejor solución no sobrevive en la nueva población se sustituye la peor solución de la nueva población por la mejor de la población anterior.

Finalmente nuestro algoritmo recibirá por parámetro la población generada aleatoriamente. El pseudocódigo es el siguiente:

```
FUNCIÓN algoritmoGeneticoGeneracional(poblacion)
        nueva_poblacion = poblacion
        MIENTRAS evaluaciones < 100000 HACER
                hijos = cruzarGeneracional(nueva_poblacion)
                mutados = mutarAG(nueva_poblacion, posiciones)
                //Añado las mutaciones a la población
                 PARA cada posición que he mutado
                        nueva_poblacion[posicion] = mutados[posicion]
                FIN-PARA
                mejor_cromosoma = reemplazamiento(nueva_poblacion, 1)
                PARA cada posición del cromosoma hijo
                        nueva_poblacion[posiciones_hijos[posicion]] = hijos[posicion]
                        SI el mejor_cromosoma ha sido mutado
                                elitista = true
                        FIN-SI
                FIN-PARA
                SI elitista
                        peor_cromosoma = sustituciones(hijos, -1)
                        nueva_poblacion[peor_cromosoma] = poblacion[mejor]
```

```
            FIN-SI
        FIN-MIENTRAS
        DEVOLVER población
FIN
```

Defino a continuación el algoritmo de *cuzarGeneracional()* presentado anteriormente:

```
FUNCIÓN cruzarGeneracional(poblacion)
        esperanza = probabilidad_cruce_agg (0.7) * padres_generacional (50)
        PARA contador = 0 hasta contador igual a esperanza, contador++
            Añado padres que decido por torneo binario, torneoBinario(poblacion)
        FIN-PARA
        esperanza = esperanza / 2
        //Aplico a continuación uno de los operadores de cruce
        //es decir aplico cruce Uniforme o por segmento fijo según sea AGG-UN o AGG-SF
        PARA cromosoma = 0 hasta un numero igual a la esperanza, cromosoma++
            Añado hijos (por UN o SF) cruzando el cromosoma padre y cromosoma padre +
                                                                              esperanza
        FIN-PARA
        DEVOLVER hijos
FIN
```

Respecto a la función *reemplazamiento()* es una función sencilla que recibe por parámetros la población y un modo, este modo puede ser -1 ó 1. Si es 1 la función devuelve el mejor cromosoma de la población, para ello evalúa todos los cromosomas de la población y se queda con el que obtenga mejor valoración (es decir la función objetivo más pequeña). Si recibe -1 hace lo mismo pero en este caso se quedará con el cromosoma que peor valoración obtenga.

Algoritmo Genético Estacionario (AGE)

En cada iteración de este algoritmo se seleccionan dos padres, ambos se cruzan, mutan y dan lugar a dos hijos. Dichos hijos competirán a continuación para entrar en la población. Este algoritmo es más sencillo que el anterior y tenemos que tener en cuenta las siguientes cuestiones:

- La probabilidad de cruce en este caso es de 1, es decir, los dos padres seleccionados siempre cruzan.
- La selección de los dos padres la realizamos por medio de torneo binario.
- La probabilidad de mutación es la misma que la anterior: 0.001, y calculamos el número de cormosomas de la misma forma: “cantidad genes por cromosoma * 0.001”.
- Cuando hemos generado ambos hijos esto sustituyen a los dos peores cromosomas de la población actual, si son mejores que ellos.

Finalmente presentamos el pseudocódigo:

```
FUNCIÓN algoritmoGeneticoEstacionario(poblacion)
```

```
        nueva_poblacion = poblacion
        MIENTRAS evaluaciones < 100000 HACER
                hijos = cruzarEstacionario(nueva_poblacion)
                mutados = mutarAG(nueva_poblacion, posiciones)
                //Añado las mutaciones a la población
                 PARA cada posición que he mutado
                        nueva_poblacion[posicion] = mutados[posicion]
                FIN-PARA
                //Compruebo que los hijos son mejores que los peores y sustituyo
                peor_cromosoma = reemplazamiento(nueva_poblacion, 1)
                SI funcionObjetivo(nueva_poblacion[peor]) > funcionObjetivo(hijos[0])
                        nueva_poblacion[peor] = hijos[0]
                FIN-SI
                peor_cromosoma = reemplazamiento(nueva_poblacion, 1)
                SI funcionObjetivo(nueva_poblacion[peor]) > funcionObjetivo(hijos[1])
                        nueva_poblacion[peor] = hijos[1]
                FIN-SI
        FIN-MIENTRAS
        DEVOLVER población
FIN
```

Defino a continuación el algoritmo de *cuzarEstacional()* presentado anteriormente:

```
FUNCIÓN cruzarEstacional(poblacion)
        esperanza = probabilidad_cruce_age (1) * padres_estacionarios (2)
        PARA contador = 0 hasta contador igual a esperanza, contador++
                Añado padres que decido por torneo binario, torneoBinario(poblacion)
        FIN-PARA
        //Aplico a continuación uno de los operadores de cruce
        //es decir aplico cruce Uniforme o por segmento fijo según sea AGG-UN o AGG-SF
        PARA cromosoma = 0 hasta un numero igual a la esperanza, cromosoma++
                Añado hijos (por UN o SF) cruzando los dos padres obtenidos
        FIN-PARA
        DEVOLVER hijos
FIN
```

5. Descripción Algoritmos Meméticos

Un algoritmo memético es el resultado de hibridar un algoritmo genético generacional junto con una búsqueda local. Concretamente, en nuestro caso, hemos mezclado el código AGG desarrollado anteriormente, junto con una búsqueda local suave.

Uno de los principales problemas de estos algoritmos es establecer cuando realizamos la búsqueda local y sobre que cromosomas de la población se aplicará. Se han desarrollado dos variantes fundamentales en esta práctica.

- La primera recibe una probabilidad de aplicar la búsqueda local suave a un cromosoma. Encontramos dos experimentos realizados: uno con una probabilidad de 1, es decir, se le aplicará a todos los cromosomas, y otro con una probabilidad de 0.1, s aplicará a un subconjunto de los cromosomas.
- La segunda variante aplica búsqueda local a un porcentaje * N mejores soluciones de la población. El porcentaje es de 0.1.

En ambos casos se ejecuta la búsqueda local cada 10 generaciones de AGG.

Finalmente podemos observar que no hemos aplicado la misma búsqueda local que en la anterior práctica. Esta nueva busqueda local suave recibe un cromosoma y selecciona elementos aleatorios del mismo y les asigna a estos elementos el mejor valor posible que pueden tomar actualmente. Cada vez que no producimos un cambio en el cromosoma decimos que este falla y aumentamos el contador de fallos, lo que nos evita desperdiciar evaluaciones de la función objetivo, ya que al llegar al máximo de fallos nos detenemos. Siendo el máximo de fallos 0.1 * número de genes por cromosoma.

En todos los casos comienzo con la función *algoritmoMemetico()* que recibe como parámetro la población generada aleatoriamente gracias a *generarPoblacion()*, ya definido anteriormente. Esta función es idéntica al *algoritmoGeneticoGeneracional(poblacion)*, definida en el apartado anterior. La única diferencia es que cada 10 iteraciones se llama a la función *optimizar(nueva_población, modo)*. Esta función aplicará la búsqueda local suave según la variante que estemos ejecutando.

```
FUNCIÓN optimizar(población, modo)
    poblacion ← RandomShuffle({1, …, n})
    SWITCH(modo)
        CASO AM-(10, 1.0)
            PARA cada cromosoma de la población
                población[cromosoma] =
                                busquedaLocalSuave(población[cromosoma])
            FIN-PARA
        BREAK
        CASO AM-(10, 0.1)
            PARA cada cromosoma de la población
                SI dado(0.1)
                    población[cromosoma] =
                                busquedaLocalSuave(población[cromosoma])
                FIN-SI
            FIN-PARA
```

```
            BREAK
            CASO AM-(10, 0.1mej)
                mejor_cromosoma = reemplazamiento(poblacion, 1)
                población[mejor_cromosoma] =
                                busquedaLocalSuave(población[mejor_cromosoma])
            BREAK
        FIN-SWITCH
        DEVOLVER población
FIN
```

Finalmente definiremos el pseudocódigo de la función *busquedaLocalSuave()*, ya definida anteriormente:

```
FUNCIÓN busquedaLocalSuave(cromosoma)
        mejora = true
        fallos = 0
        valoración = Evalua.funcionObjetivo(cromosoma)
        genes = número de genes que forman el cromosoma
        contador = 0
        MIENTRAS (mejora = true || fallos < MAX_FALLOS * genes) && contador < genes
            mejora = false
            vecino  = mutacionUniforme(cromosoma)
            valoracion_vecino = Evalua.funcionObjetivo(vecino)
            SI valoracion_vecino < valoracion
                mejora = true
                cromosoma = vecino
                valoracion = valoracion_vecino
            SINO
                Incremento fallos en 1
            FIN-SINO
            Incremento contador en 1
            SI evaluaciones ==  100000
                BREAK
            FIN-SI
        FIN-MIENTRAS
        DEVOLVER cromosoma
FIN
```

6. Breve manual de usuario

Como ya he mencionada anteriormente la práctica está realizada en el lenguaje de programación C++. Para su realización he utilizado el guión de la práctica así como las diapositivas del seminario.

El código se encuentra dividido en 6 archivos .cpp, situados en la carpeta “bin” con su correspondiente “.h” en la carpeta “include”. Los datos descargados de la web correspondientes a los conjuntos de datos y restricciones se sitúan en la carpeta “datos”.

A continuación detallaré la información contenida en cada archivo:

random.cpp/.h: este fichero contiene el mismo código que el publicado en la web. Para utilizar esta clase, utilizo desde el “main” la función “Set_random” inicializando la semilla a 49137372.

utiles.cpp/.h: en estos archivos he definido los métodos para leer los datos de entrada, tanto del conjunto de datos como el de la matriz de restricciones. Métodos que utilizaré en el “main” para inicializar los vectores de datos y restricciones. La lectura la he realizado gracias a “fstream”

AG.cpp/.h: como es de esperar en este archivo están definidos los métodos necesarios para ejecutar los diferentes algoritmos genéticos implementados que ya hemos mencionado.

AM.cpp/.h: en este caso encontramos los métodos necesarios para ejecutar los diferentes algoritmos meméticos desarrollados.

evaluacion.cpp/.h: encontramos las diferentes funciones que nos permiten ejecutar la función objetivo, además se encarga de ir incrementando el número de evaluaciones realizadas.

main.cpp: es el programa principal desde el que se llama a los métodos anteriormente descritos. Su estructura de forma resumida es:

```
        Set_random(49137372)
        datos = util.leerArchivoMatriz(“conjunto de datos”)
        restricciones = util.leerArchivoMatriz(“conjunto de restricciones”)
//Encontramos variables definidas en el main que nos permite acceder a cada conjunto de datos,
//de esta forma si queremos utilizar el cnjunto de datos de iris → util.leerArchivoMatriz(“iris”)
        //Inicializo la busqueda local
        AG genetico(n_cluster, datos, restricciones);
        //inicializo greedy
        AM memetico(n_cluster, datos, restricciones);

//Tras esto llamaré a las funciones para ejecutar ambos algoritmos
```

He realizado un makefile que se encarga de la compilación de todos los archivos.
Para lanzar el programa se debe lanzar desde una terminal situada en la carpeta raíz los comandos:

- make
- ./PAR

7. Experimentos y análisis de resultados

Antes de analizar los resultados obtenidos voy a explicar los conjunto de datos y restricciones que se han utilizado. Respecto a los conjuntos de datos:

- Iris: Posee información sobre las características de tres tipos de flores Iris, por lo tanto tendremos 3 clases, es decir, el número de clusters es 3.
- Ecoli: contiene características sobre diferentes tipos de células utilizadas para predecir la localización de ciertas proteínas. Tiene 8 clases, es decir, número de clusters es 8.
- Rand: Es un conjunto de datos artificial, se encuentra formado por 3 clusters bien diferenciados en base a distribuciones normales.
- Newthy Roid: contiene medidas cuantitativas tomadas sobre la glándula tiroides de 215 pacientes. Presenta 3 clases, es decir, número de clusters es 3.

Respecto al conjunto de restricciones, para cada conjunto de datos tenemos 2 conjuntos de restricciones generadas aleatoriamente, correspondientes al 10% y 20% del total de restricciones posibles.

Mencionar que los tiempos reflejados en la tabla están en segundos y los he obtenido gracias a la librería “chrono” y “ctime”. Las semillas utilizadas en cada ejecución comenzando con la Ejecución 1 hacia la 5, han sido: “49137372”, “491373”, “4913”, “49”, “100”.

He utilizado para el memético cruce uniforme debido a que, aunque mis resultados con AGG-SF sean levemente mejores que AGG-UN, al probar la hibridar con ambos casos, mis resultados son mejores en los algoritmos meméticos si utilizo cruce uniforme.

Importante añadir que al realizar la práctica me he dado cuenta de que cometía un error al obtener la distancia máxima y las distancias en la práctica anterior. Debido a falta de tiempo no he podido actualizar mis resultados de la anterior práctica y corregirlos, por ello los presentados no son del todo correctos en el caso del greedy y la búsqueda local.

Solamente adjunto las tablas que contienen la media de los resultados, la demás tablas se encuentran en el archivo “tablas_experimentos” situada en la misma carpeta. La causa es la gran cantidad.

Analizados los conjuntos de datos y restricciones, pasamos a mostrar las tablas para cada algoritmo:

Tabla 6.12: Resultados globales en el PAR con 10% de restricciones

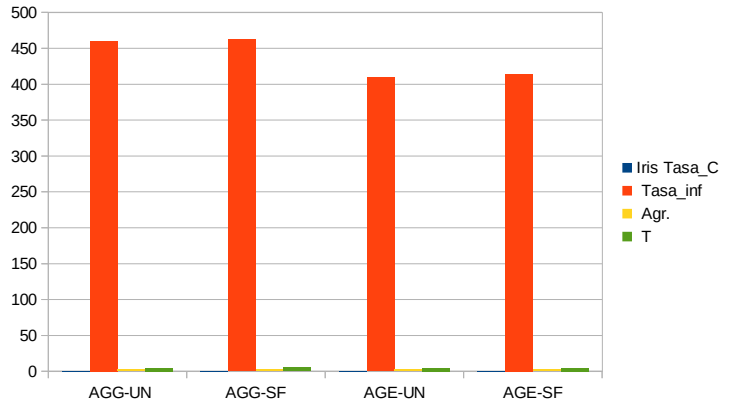
	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,66	48,60	0,78	0	17,99	535,80	25,55	0,01	0,55	0,00	0,55					
BL	1,72	0,00	1,72	1,02	35,51	49,20	36,16	207,36	1,40	0,00	1,40					
AGG-UN	0,29	460,40	2,87	4,89	7,52	1782,40	52,65	18,71	0,4	460,40	3,33	4,79	2,02	1124,80	39,64	8,65
AGG-SF	0,3	462,40	2,88	5,25	7,51	1780,60	52,59	20,02	0,41	465,20	3,36	5,28	2,02	1122,80	39,57	9,19
AGE-UN	0,29	409,60	2,58	4,7	7,48	1701,60	50,56	18,3	0,4	414,80	3,03	4,64	2,05	1055,00	37,34	8,27
AGE-SF	0,29	414,40	2,61	4,71	7,46	1697,00	50,42	18,34	0,4	414,40	3,03	4,65	2,05	1048,20	37,11	8,3
AM-(10, 1.0)	0,18	133,00	0,93	4,86	7,5	982,80	32,39	18,73	0,23	123,80	1,01	4,84	2,67	331,80	13,76	8,55
AM-(10, 0,1)	0,27	375,96	2,37	4,88	7,49	1588,88	47,72	18,82	0,37	375,72	2,75	4,84	2,16	936,52	33,48	8,59
AM-(10, 0,1mej)	0,18	157,80	1,06	5,00	7,45	1053,00	34,11	19,33	0,25	156,80	1,25	5,06	2,58	400,20	15,97	8,81

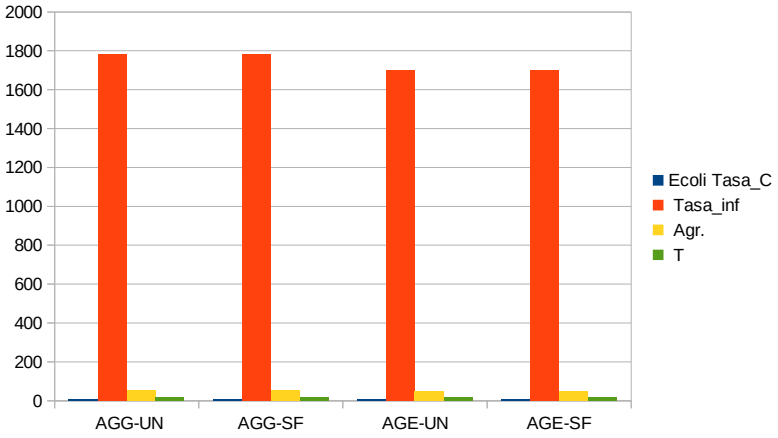
Comentar que las altas valoraciones obtenidas tanto en Ecoli como Newthy Roid son elevadas debido a que el valor de la mayor distancia presente en el conjunto es muy elevada. En el caso de Ecoli la distancia entre los puntos más separados es de 150.99 y en el caso de Newthy Roid de 84.11.

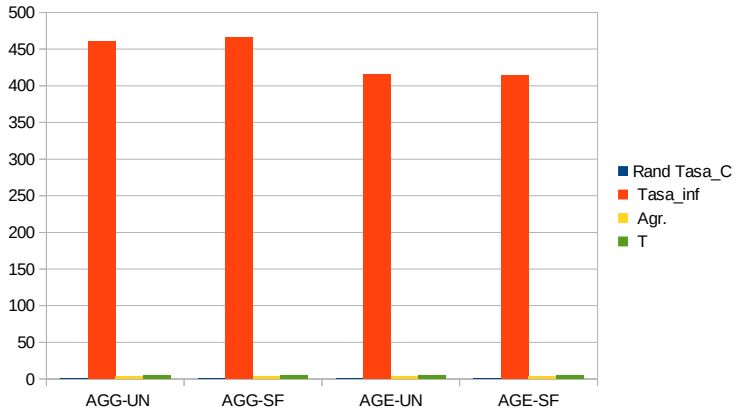
Dado que debemos analizar una gran cantidad de algoritmos y datos, vamos a estructurar el análisis por bloques, en los que se compararán y explicarán distintos aspectos de los experimentos realizados:

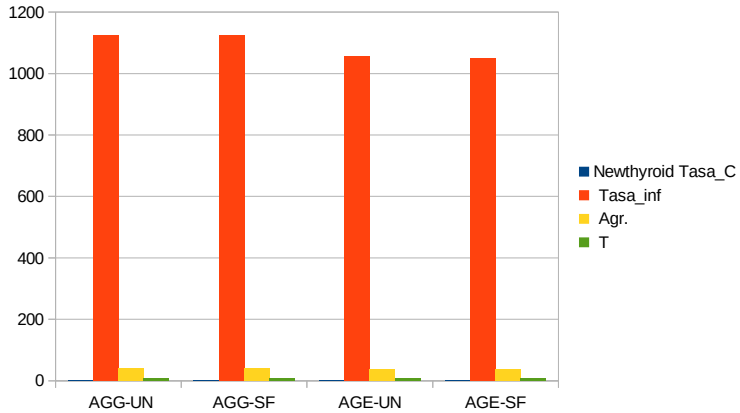
- Valoración general de los algoritmos genéticos: podemos observar que los resultados obtenidos por este tipo de metaheurística no son de una gran calidad. La causa de esto es debido a dos motivos: la principal causa es que por norma general los algoritmos genéticos requieren de un gran número de generaciones para poder llegar a converger correctamente, quizás el número de evaluaciones establecido no es lo suficientemente grande.

- Generacional vs Estacionario: Compararemos los dos tipos de esquemas de evolución frente a este problema. Las tablas muestran que los resultados mediante el estacionario son mejores que las obtenidas mediante el generacional. La alta presencia presión selectiva, junto con el elitismo propio de un estacionario (ya que recordemos que este tipo de variante elimina los cromosomas peores, conservando los mejores), producen que funcione bien en nuestro problema. Respecto a los tiempos también encontramos diferencia, siendo mejor los del estacionario, pero tampoco es muy amplia dicha diferencia.



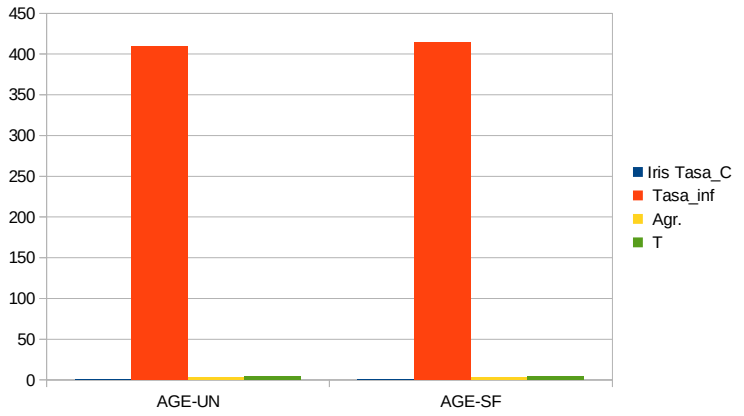


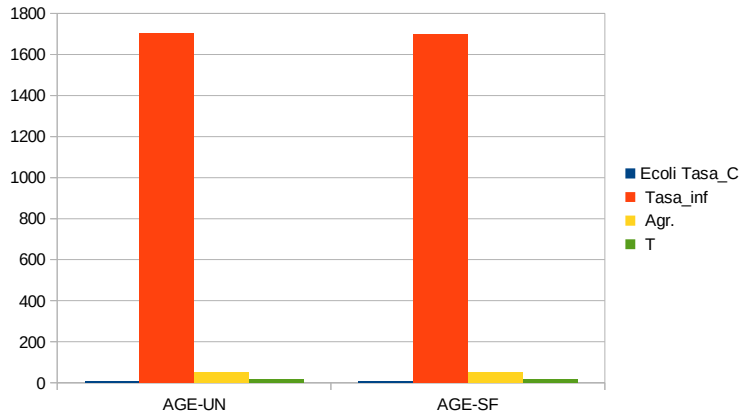


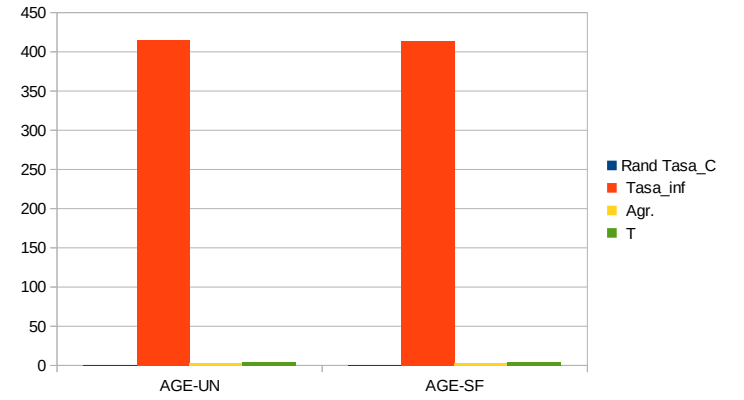


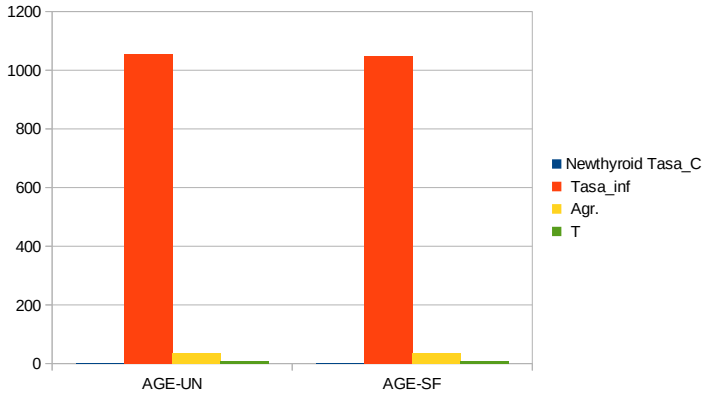
Podemos observar en estas gráficas, que presentan cada conjunto de dato con los resultados de cada variante del algoritmo genético, como es notable la mejor de los algoritmos estacionarios frente a los generacionales.

- Operadores de cruce: observamos en los resultados la importancia de los operadores empleados para el algoritmo, en concreto la importancia del operador de cruce se ve bien reflejado en las tablas. Podemos observar que obtenemos mejores resultados cuando utilizamos un cruce por segmento fijo en lugar de cruce uniforme.



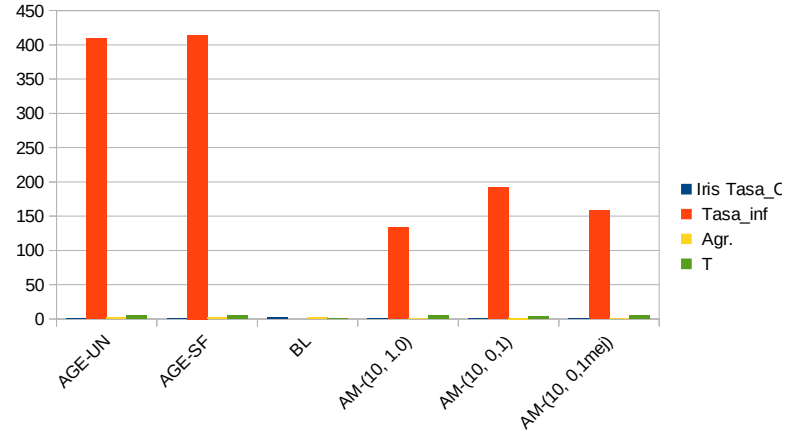


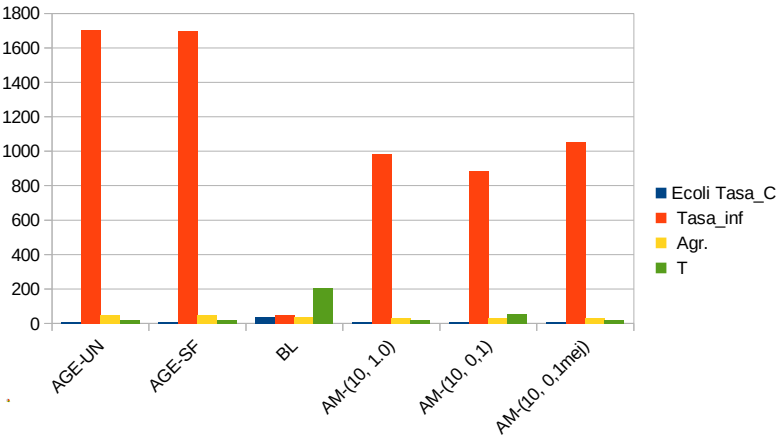


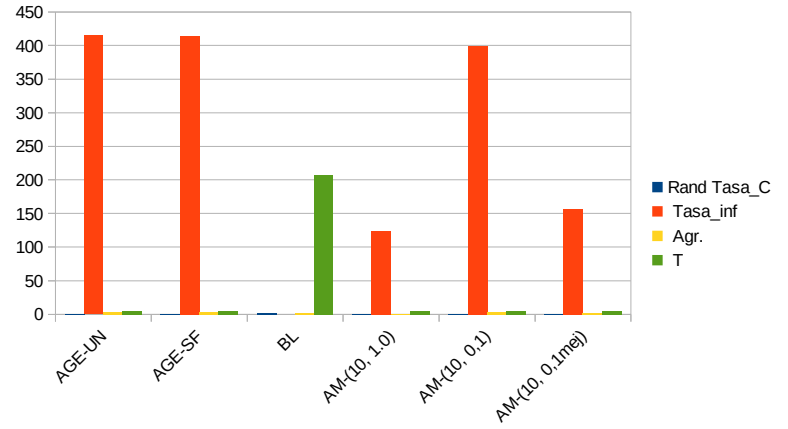


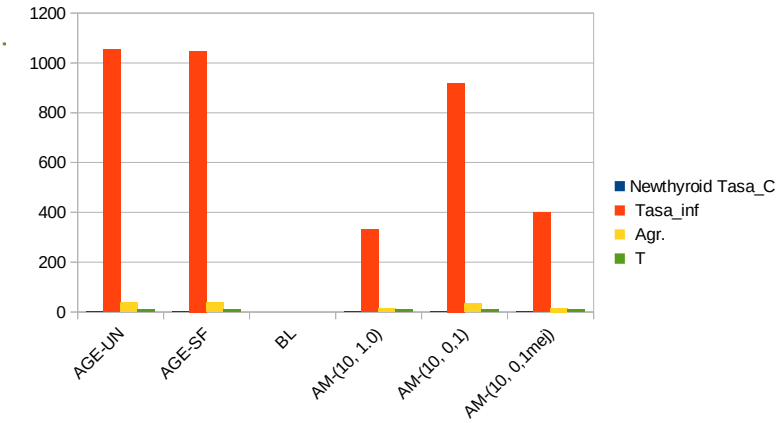
Presentamos las graficas de los 4 conjunto de datos, comparando los resultados estacionarios del cruce uniforme frente el cruce por segmento. Solamente presento los resultados con el estacionario porque como vimos antes obtengo mejores resultados. Podemos observar que es difícil observar la diferencia entre ellos.

- Algoritmos Meméticos: por norma general los algoritmos meméticos suelen mejorar los resultados obtenidos por los algoritmos genéticos y a los obtenidos con la búsqueda local, ya que estos son una hibridación de ambos (aunque en nuestro caso no hallamos utilizado la misma búsqueda local que en la práctica anterior). En nuestro caso así ha sido, en todos los tipos de algoritmo meméticos hemos obtenido mejores resultados. Respecto a los tiempos de ejecución, estos son razonables, después de todo la búsqueda local implementada es rápida.



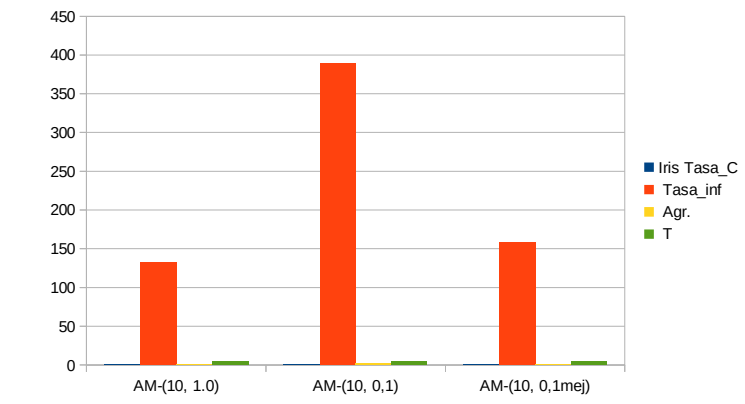


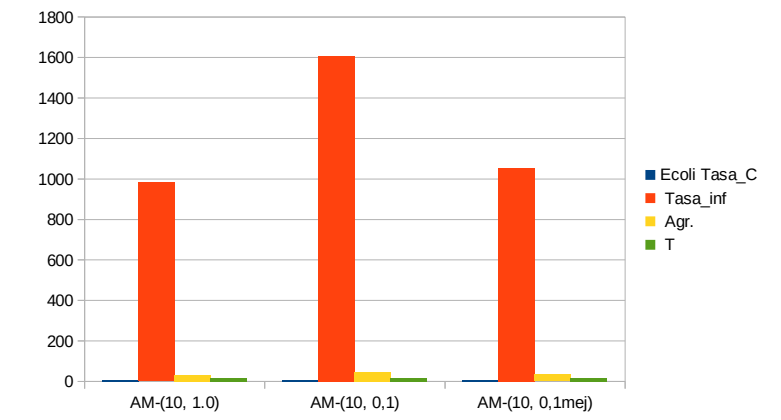


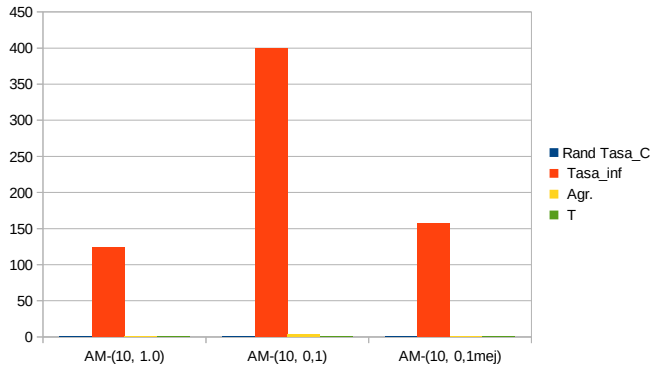


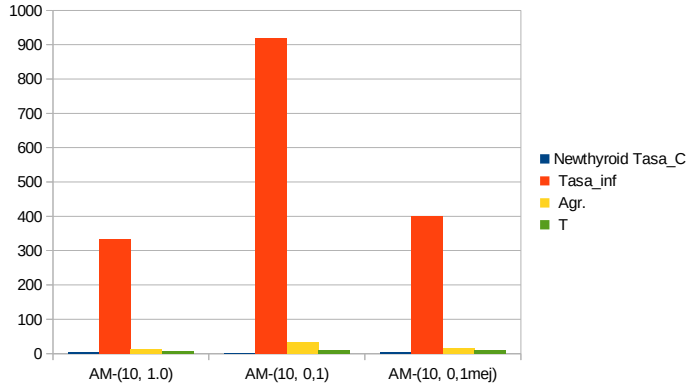
Presentamos las gráficas que nos permiten comparar los resultados de las búsqueda local y algoritmo genético (solo mostramos el estacional, ya que es en el que mejor resultado hemos obtenido como ya mencionamos anteriormente). Las gráfica nos permiten observar con facilidad la mejor de los algoritmos mémeticos en prácticamente todos los casos. Recordar que los datos de la búsqueda local son los obtenidos en la práctica anterior, datos que como he comentado tienen errores, por ello no nos son de especial utilidad en nuestra comparación.

- Comparación entre las distintas variantes del memético: el mejor de los tres meméticos es aquel que aplica la búsqueda local sobre todas las soluciones. Es cierto que este tipo en ocasiones podría fallarnos y dar lugar a una convergencia prematura, pero frente este problema no he tenido ese problema. Aplicar la búsqueda sólo a los mejores no tiene porque dar buenos resultados, puede que todos sean puntos cercanos al óptimo local.



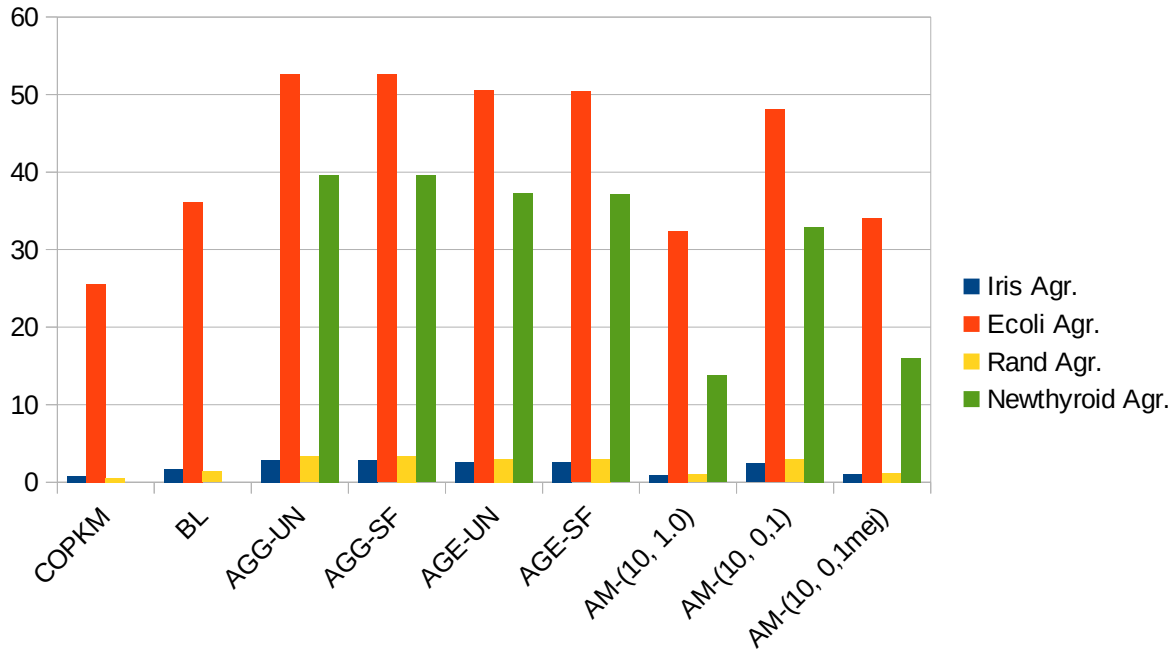






Presentamos las gráficas que nos permiten comparar los resultados obtenidos por cada variante de los algoritmos meméticos. Es notable la mejora de los resultados con la búsqueda local realizada sobre todas las soluciones.

No ha sido el caso, pero la aleatoriedad puede llegar a proporcionar mejores resultados porque permite una mayor diversidad, aplicando la búsqueda local a distintos puntos del espacio de búsqueda y manteniendo diversidad de otras soluciones. Por esto AM-(10, 0.1) podría llegar a proporcionar muy buenos resultados.

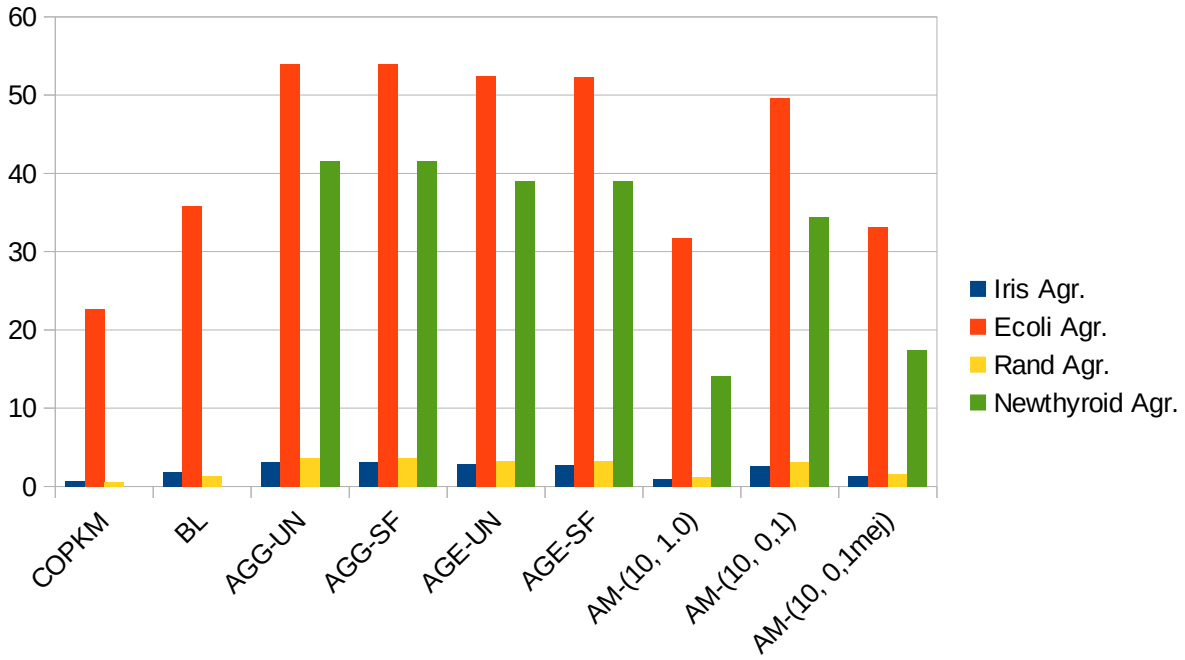


Presentamos una gráfica que muestra en conjunto todas las agregaciones obtenidas por cada algoritmo en cada conjunto. Podemos observar que nuestra función objetivo nos devuelve las mejores valoraciones con el algoritmo greedy y con el algoritmo memético AM-(10, 1.0). Los peores resultamos son los obtenidos por los algoritmos genéticos generacionales.

Tabla 6.16: Resultados globales en el PAR con 20% de restricciones

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,64	0,00	0,64	0	20,33	327,00	22,65	0,01	0,55	0,00	0,55	0				
BL	1,79	0,00	1,79	1,53	35,66	18,00	35,78	290,84	1,40	0,00	1,40	1,39				
AGG-UN	0,3	940,40	3,09	5,88	7,49	3567,80	53,96	24,57	79,92	939,20	3,57	5,89	2,05	2266,40	41,63	11,1
AGG-SF	0,29	936,80	3,08	6,28	7,48	3565,00	53,92	25,96	0,4	940,20	3,57	6,28	2,04	2266,20	41,62	11,63
AGE-UN	0,29	858,60	2,84	5,81	7,47	3447,80	52,38	24,36	0,4	844,20	3,24	5,56	2,02	2115,80	38,98	10,9
AGE-SF	0,29	845,80	2,80	5,71	7,48	3438,20	52,27	24,37	0,39	837,80	3,21	5,71	2,03	2119,60	39,05	10,84
AM-(10, 1.0)	0,18	272,00	0,99	6,03	7,64	1848,00	31,71	24,64	0,23	268,40	1,14	5,97	2,5	668,00	14,17	10,98
AM-(10, 0,1)	0,27	786,60	2,61	6,16	7,36	3241,60	49,59	25,25	0,38	795,80	3,06	6,08	2,19	1847,60	34,46	11,28
AM-(10, 0,1mej)	0,20	382,60	1,34	6,13	6,45	1966,60	33,09	25,26	0,27	391,60	1,59	6,09	2,49	856,40	17,45	11,38

Al aplicarlo los mismos algoritmos a los mismos conjuntos de datos, pero con un conjunto de restricciones del 20% observamos que obtenemos las mismas reflexionas que hemos comentado con anterior



Presentamos una gráfica con la misma estructura que la anterior pero esta vez frente a un conjunto de 20% de restricciones. La estructura es más o menos similar.

Finalmente añadir que como experimento personal añadí una mayor frecuencia al número de veces que realizo la busqueda local en los algoritmos meméticos, lo cual me producía resultados con 0 de infeasibility.

Antes de dar una valoración final volver a comentar que los resultados presentados del greedy y busqueda local, de la práctica anterior, no son correctos. He realizado pruebas para ver que valores aproximados deberíamos obtener con estos algoritmos, y en el caso de la búsqueda local los resultados deberían ser mejores, ya que me ha disminuido bastante la distancia. De manera opuesta, los resultados del greedy deberían ser peores, ya que las distancias han aumentado en este caso.

Conclusión final. ¿Merece la pena utilizar un genético frente a la búsqueda local de la primera práctica? Nos centramos en estos dos algoritmos, ya que son los dos con los que mejores resultados obtenemos, ya que si mi greedy fuera correcto tendría unos resultados peores. En mi opinión merece más la pena la búsqueda local, ya que su implementación es más sencilla, su tiempo de

ejecución es menor y en caso de que su implementación fuera correcta, obtendríamos unos resultados similares. No obstante, si mejorásemos los parámetros del genético (evaluaciones, probabilidad) es probable que obtengamos mejores resultados a costa de un tiempo mayor.

8. Bibliografía

La bibliografía que he utilizado para desarrollar la práctica ha sido básicamente la que aparece en la web de la asignatura, tanto el guión como las diapositivas del seminario 3 de prácticas sobre los poblemas PAR y MDP.

También he obtenido un gran información sobre el uso de vectores de http://www.cplusplus.com/