Víctor Cabrita Gómez
72355039Y - victorvcg@correo.ugr.es
Grupo 3 - Viernes

Problema del Agrupamiento con Restricciones

PRÁCTICA FINAL
Asexual Reproduction Optimizer
Algoritmo Memético ARO-LS
Algoritmo Híbrido ARO-ES

UNIVERSIDAD
DE GRANADA

17-05-2020

Índice general

1. Descripción del problema

El problema base consiste en agrupar un conjunto de elementos en X grupos diferentes según sus características. De esta forma se agruparán en distintos clústeres, los cuales tendrán su centroide u_i: un punto calculado como el punto medio de todos los elementos asociados a dicho clúster c_i.

$$\vec{u_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$$

Definimos la distacia media intra-clúster como la media de las distancias de cada elemento de un clúster a su centroide:
$$\bar{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \|\vec{x_j} - \vec{u_i}\|$$

Definimos la desviación general de la partición $C = c_1, ..., c_k$ como la media de todas las distancias medias intra-clúster:

$$\bar{C} = \frac{1}{k} \sum_{c_j \in C} \bar{c_i}$$

Lo especial del PAR es que añade el tratamiento de las restricciones, es decir, especifica que ciertos elementos deben o no deben ir juntos.

Estas restricciones se pueden tratar como restricciones fuertes o débiles, dependiendo la indispensabilidad de cumplirlas o no. Para nuestros algoritmos usaremos unas restricciones débiles, es decir, no será indispensable cumplirlas, si no que se ponderará el hecho de inclumplirlas junto con la distancia al centroide.

Definimos la $infeasibility$ (ïnfactibilidad") como el número de restricciones violadas.

De esta forma nuestro objetivo será minimizar la función

$$f = \bar{C} + (infeasibility * \lambda)$$

Siendo λ un parámetro de ponderación igual a la mayor distancia entre dos elementos del conjunto dividida entre el número de restricciones.

2. Algoritmos utilizados

El algoritmo utilizado en esta práctica será el *Asexual Reproduction Optimizer*. Para el cual, además de su versión original se propone crear un algoritmo memético que hibride entre ARO y búsqueda local y crear otro algoritmo que mejore el ARO original. Para el algoritmo mejorado se ha realizado una hibridación entre ARO y enfriamiento simulado.

Para todos los algoritmos no comparativos se presentan unos elementos comunes:

La solución se representará mediante un vector de enteros en el que el índice del vector indica a qué elemento del conjunto de datos hace referencia y el contenido de dicha posición del vector, a qué cluster pertenece el elemento.

La función objetivo es la función f relatada anteriormente: la suma entre la desviación general y el producto de la infactibilidad y lambda.

Se generará una solución inicial aleatoria de forma que ningún cluster quede vacío

La generación de nuevas soluciones se hará escogiendo un elemento aleatorio de la solución y moviéndolo de clúster para cada vecino creado, respentando siempre que no haya ningún clúster vacío.

El criterio de aceptación de una nueva solución es que disminuya el valor de la función objetivo de la solución actual, excepto para el algoritmo del enfriamiento simulado, en el que existe una probabilidad de que se acepte una solución peor a la actual para favorecer la exploración. Esta probabilidad depende de la temperatura, que se va enfriando con el tiempo, haciendo que cada vez sea más complicado aceptar soluciones peores.

Las funciones comunes a todos los algoritmos se encuentran en el fichero par.cpp y la gran mayoría serán explicadas a continuación (no se explicarán los getters y setters por ser obvia su función):

getLongestDistance() calcula la mayor distancia entre dos puntos y es indispensable para obtener λ. Recorre la matriz de elementos y calcula la distancia entre todos ellos quedándose con la mayor.

```
1 //Calculates the longest distance between every element in the set
2 const float Par::getLongestDistance(){
3     float maxDistance = 0;
4     for(int i = 0; i < nElements(); i++)
5         for(int j = i+1; j < nElements(); j++)
6             if(getDistance(this->elements[i], this->elements[j]) > maxDistance)
7                 maxDistance = getDistance(this->elements[i], this->elements[j]);
8     return maxDistance;
9 }
```

getDistance(a, b) calcula la distancia euclídea entre los puntos a y b.

```
//Calculates the distance between 2 elements in the set
const float Par::getDistance(vector<float> a, vector<float> b){
    float distance = 0;
    for(int i = 0; i < nAtributes(); i++)
        distance += pow((a[i] - b[i]), 2);
    return sqrt(distance);
}
```

nAtributes() es un getter que explicaremos por que tiene un poco más de misterio. Por atributos, nos referimos al número de elementos que tiene cada elementos del conjunto de datos, es decir, al número de columnas de nuestra matriz.

computeCentroid(cluster) calcula el valor del centroide del cluster indicado:

```
//Calculates the centroid's position
void Par::computeCentroid(int cluster){
    int nelements = 0;
    vector<float> newCentroid(nAtributes(),0);
    for(int i = 0; i < solution.size(); i++)
        if(solution[i] == cluster){
            nelements++;
            for(int j = 0; j < nAtributes(); j++)
                newCentroid[j] += this->elements[i][j];
        }
    for(int i = 0; i < newCentroid.size(); i++)
        newCentroid[i] /= nelements;
    setCentroid(cluster, newCentroid);
}
```

emptyCluster(cluster, solution) comprueba si el cluster indicado está vacío con los resultados de la solución pasada como argumento. Al pasar la solución a comprobar como argumento se permite mirar si algún hijo deja el clúster vacío sin tener que modificar la solución original:

```
//Checks if the cluster is empty
const bool Par::emptyCluster(int cluster, vector<int> solution){
    for(int i = 0; i < nElements(); i++)
        if(solution[i] == cluster)
            return false;
    return true;
}
```

emptyClusters(solution) hace la misma función que emptyCluster(cluster, solution), pero para todos los clústeres que tengamos. Muy útil cuando no quieres mirar solo si se queda vacío el clúster del elemento que mueves si no los clústeres de toda la solución.

```
//Checks if there is any empty cluster
const bool Par::emptyClusters(vector<int> solution){
    bool empty;
    for(int i = 0; i < nClusters(); i++){
        empty = true;
        for(int j = 0; j < nElements(); j++)
            if(solution[j] == i){
                empty = false;
                break;
```

```
            }
        if(empty)
            return true;
    }
    return false;
}
```

getAgregated(solution) calcula la función objetivo de la solución indicada al igual que en la fórmula explicada antes:

```
const float Par::getAgregated(vector<int> solution){
    return getDeviation(solution) + getInfeasibility(solution)*this->lambda;
}
```

getDeviation(solution) calcula la desviación de la solución dada. Primero recorre cada clúster y calcula la desviación intra-clúster como la media de las distancias de cada punto al centroide. Después hace la media de todas las desviaciones intra-clúster:

```
const float Par::getDeviation(vector<int> solution){
    float deviation = 0;
    float clusterDeviation;
    int clusterElements;
    for(int i = 0; i < nClusters(); i++){
        clusterDeviation = 0;
        clusterElements = 0;
        for(int j = 0; j < nElements(); j++)
            if(solution[j] == i){
                clusterDeviation += getDistance(this->elements[j], centroids[i]);
                clusterElements++;
            }
        deviation += clusterDeviation/clusterElements;
    }
    return deviation/centroids.size();
}
```

getInfeasibility(solution) calcula cuantas restricciones han sido violadas por la solución indicada. Para ello, recorre la lista de restricciones y, si ninguno de los elementos de la restricción está sin asignar (cluster -1), comprueba que si los dos elementos están en el mismo clúster la restricción sea de tipo false y que si están en distintos clústers la restricción sea de tipo true. Si esto pasase, significaría que se está violando esa restricción, por lo que se incrementaría el contador de infactibilidad:

```
const int Par::getInfeasibility(vector<int> solution){
    int infeasibility = 0;
    for(list<Restriction>::iterator it=this->restrictions.begin(); it != this->restrictions.
    end(); ++it)
        if((*it).a != -1 && (*it).b != -1)         //-1 Means that that position has no
    solution yet
            if((*it).type != (solution[(*it).a] == solution[(*it).b]))
                infeasibility++;
    return infeasibility;
}
```

La función lógica del segundo IF puede parecer algo confusa, pero tiene mucho sentido. Primero mira lo que hay en el paréntesis, es decir, mira si ambos elementos están asignados al mismo cluster. Si es *true*, significa que el tipo de la restricción debería ser *false* para ser violada, y si están en uno distinto, el tipo de la restricción debería ser *true*. En conclusión, el tipo de la restricción tiene que ser el contrario al resultado de comparar si

los elementos están asignados al mismo clúster o no.

Para las tres últimas funciones descritas se ha incluido una sobrecarga de las mismas que no requiere de argumentos. Si son llamadas de esta forma, usarán como solución la propia de la instancia.

3. Métodos de búsqueda

3.1. Asexual Reproduction Optimizer

ARO es un algoritmo generacional particular, ya que solo consta de un individuo en vez de tener una población y los hijos son generados por un único padre.

Como todos los algoritmos que hemos estudiado, empieza por crear una solución inicial aleatoria de forma que no deja ningún clúster vacío en su constructor:

```
ARO::ARO(Par instance, int max_eval){
    //Creates a random solution
    this->instance = instance;
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.modifySolution(i,i);
    for(int i = this->instance.nClusters(); i < this->instance.nElements(); i++)
        this->instance.modifySolution(i, rand()%this->instance.nClusters());
    this->instance.shuffleSolution();
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.computeCentroid(i);

    //Set the rest of the class atributes
    this->MAX_EVALUATIONS = max_eval;
}
```

Una vez hecho esto, genera la solución de la forma descrita a continuación: Mientras no se llegue al máximo de evaluaciones se generará un hijo o *bud* y si este es una mejor solución que el padre, se convierte en el nuevo padre:

```
void ARO::generateSolution(){
    vector<int> bud;
    int eval = 0;
    while(eval < MAX_EVALUATIONS){
        bud = generateBud();
        if(instance.getAgregated(bud) < instance.getAgregated()){
            instance.setSolution(bud);
            for(int i = 0; i < instance.nClusters(); i++)
                instance.computeCentroid(i);
        }
        eval++;
    }
}
```

Para generar cada *bud* simplemente se generará primero una larva y se combinará de forma aleatoria el padre con la larva. Para ello, se escogeran un numero aleatorio de elementos de la solución original y sustituirán a los de la larva. Finalmente, se aplicará una reparación a la larva para evitar que algún clúster quede vacío:

```
vector<int> ARO::generateBud(){
    vector<int> solution = getSolution();
```

```
    vector<int> larva = generateLarva();
    vector<int>r = random_selection(rand()%instance.nElements(), instance.nElements());
    for(int i = 0; i < r.size(); i++)
        larva[r[i]] = solution[r[i]];
    return reparation(larva);
}
```

La función random_selection(int k, int n) selecciona aleatoriamente k números del rango $[0, n)$, sin repeticiones.

```
//Choose k numbers from [0, n) without repeating
vector<int> random_selection(int k, int n){
    vector<int> range, selected;
    int position;
    for(int i = 0; i < n; i++)
        range.push_back(i);
    for(int i = 0; i < k; i++){
        position = rand()%range.size();
        selected.push_back(range[position]);
        range.erase(range.begin()+position);
    }
    return selected;
}
```

Cada larva es generada aplicándole una fuerte mutación por segmento al padre. Se generan dos números aleatorios, g y s. g es el primer punto del segmento y s es el tamaño del mismo. Como ya hemos visto en otros ejercicios, el segmento es circular, es decir, si se llega al final del vector solución sin haber llegado al final del tamaño del segmento, se seguirá modificando por el principio del vector solución. Para cada elemento del segmento se cambiará su clúster por otro seleccionado de forma aleatoria. No se comprobará la factibilidad de la solución, ya que después de combinar la larva con la solución original para generar el *bud* se hará un reparación, por lo que aplicar dos veces la reparación sería una pérdida de tiempo y de recursos.

```
vector<int> ARO::generateLarva(){
    vector<int> solution = getSolution();
    int g = rand()%solution.size();
    int s = rand()%solution.size();
    for(int i = 0; i < s; i++)
        solution[(i+g)%solution.size()] = rand()%instance.nClusters();
    return solution;
}
```

La reparación es la misma que en las otras prácticas, dada una solución comprueba si hay algún clúster vacío y en caso de haberlo, selecciona un elemento aleatorio de la solución y lo introduce en dicho clúster. Si se ha producido algún cambio, vuelve a repetir la operación, ya que podría haber un nuevo clúster vacío o podría haber más de uno vacíos desde un principio. Cuando compruebe que ya no hay ningún clúster vacío, devuelve el resultado.

```
vector<int> ARO::reparation(vector<int> solution){
    bool empty = true;
    //For each cluster, if one is empty, places an element in that cluster until no cluster is
     empty
    while(empty){
        empty = false;
        for(int i = 0; i < instance.nClusters(); i++){
            if(instance.emptyCluster(i, solution)){
                solution[rand()%solution.size()] = i;
                empty = true;
            }
```

```
        }
    }
    return solution;
}
```

3.2. Algoritmo memético ARO-LS

Para este algoritmo haremos una combinación memética entre búsqueda local y nuestra metaheurística. Primero, generaremos una solución inicial aleatoria como siempre:

```
AROLS::AROLS(Par instance, int max_eval, int k, int max_fails){
    //Creates a random solution
    this->instance = instance;
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.modifySolution(i,i);
    for(int i = this->instance.nClusters(); i < this->instance.nElements(); i++)
        this->instance.modifySolution(i, rand()%this->instance.nClusters());
    this->instance.shuffleSolution();
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.computeCentroid(i);

    //Set the rest of the class atributes
    this->MAX_EVALUATIONS = max_eval;
    this->INTERVAL = k;
    this->MAX_FAILS = max_fails*instance.nElements();
}
```

Para generar la solución, se irán contando las iteraciones de nuestro algoritmo ARO y cada *INTERVAL* iteraciones (en nuestro caso 10) se aplicará una Búsqueda Local Suave a nuestra solución, en lugar de ARO.

```
void AROLS::generateSolution(){
    vector<int> bud;
    int eval = 0;
    int iter = 1;
    while(eval < MAX_EVALUATIONS){
        //Applies local search every INTERVAL interval
        if(iter%INTERVAL == 0){
            eval += BLS();
        }
        //If not, applies the usual algorithm
        else{
            bud = generateBud();
            if(instance.getAgregated(bud) < instance.getAgregated()){
                instance.setSolution(bud);
                for(int i = 0; i < instance.nClusters(); i++)
                    instance.computeCentroid(i);
            }
            eval++;
        }
        iter++;
    }
}
```

Para la búsqueda local suave se genera una lista desordenada de los índices de todos los elementos de la solución, de forma que recorramos los mismos de forma no secuencial.
Después, para cada uno de esos índices se coloca dicho elemento en el clúster que mejor solución produzca,

siempre y cuando no deje ningún clúster vacío.
En caso de que haya sido imposible obtener una mejor solución moviendo dicho elemento a otro clúster, se dejará el elemento en el mismo clúster y se incrementará el contador de fallos. Cuando el contador de fallos llegue al máximo y se produzca un nuevo fallo o se haya aplicado el proceso sobre todos los índices de la lista, la función se detendrá, habiendo modificado la solución y devolviendo el número de evaluaciones realizadas.

```
//Applies BLS to the solution and return the number of evaluations done
int AROLS::BLS(){
    Par chromosome = instance;
    vector<int> RSI;
    for(int i = 0; i < chromosome.nElements(); i++)
        RSI.push_back(i);
    random_shuffle(RSI.begin(), RSI.end());
    int fails = 0;
    int i = 0;
    bool upgrade = true;
    vector<int> auxSolution;
    int bestCluster;
    float bestAgregated;
    float auxAgregated;
    int eval = 0;

    while((upgrade || fails < MAX_FAILS) && i < RSI.size()){
        upgrade = false;
        auxSolution = chromosome.getSolution();
        bestCluster = auxSolution[RSI[i]];
        bestAgregated = chromosome.getAgregated(auxSolution);
        for(int j = 0; j < chromosome.nClusters(); j++){
            auxSolution[RSI[i]] = j;
            auxAgregated = chromosome.getAgregated(auxSolution);
            eval++;
            if(!chromosome.emptyClusters(auxSolution) && auxAgregated < bestAgregated){
                bestCluster = j;
                bestAgregated = auxAgregated;
                upgrade = true;
            }
        }
        chromosome.modifySolution(RSI[i], bestCluster);
        if(!upgrade)
            fails++;
        i++;
    }
    for(int i = 0; i < chromosome.nClusters(); i++)
        chromosome.computeCentroid(i);
    this->instance = chromosome;
    return eval;
}
```

3.3. Algoritmo híbrido ARO-ES

Como idea para mejorar nuestro algoritmo, se propone una combinación con el algoritmo de enfriamiento simulado, de forma que se seguirá el esquema del enfriamiento simulado de la práctica anterior, pero en vez de generar un nuevo vecino moviendo un elemento de clúster, se generará creando un nuevo *bud*.

De esta forma tendremos el mismo algoritmo ARO, pero esta vez será capaz de aceptar *buds* con peores soluciones, atendiendo al esquema del enfriamiento simulado.

La temperatura inicial será $\frac{\mu * C(S_0}{-ln(\phi)}$, siendo $C(S_0)$ el coste de la solución inicial y μ y $\phi = 0.3$

La temperatura irá enfriando siguiendo el esquema de Cauchy modificado:
$T_{k+1} = \frac{T_k}{1+\beta * T_k}$; $\beta = \frac{T_0 - T_f}{M * T_0 * T_f}$
M es el número de enfriamientos a realizar, T_0 es la temperatura inicial explicada antes y T_f es la temperatura final, que para nuestra práctica será 10^{-3}.

La condición de enfriamiento $L(T)$ será que se haya generado un número máximo de vecinos *MAX_NEIGHBOURS* o cuando se hayan aceptado un número máximo de ellos *MAX_SUCCESS*.

La condición de parada será que se haya alcanzado el número máximo de evaluaciones *MAX_EVALUATIONS* o el número de éxitos sea 0, es decir, no haya una mejora.

Una vez definidos los puntos clave de nuestro enfriamiento simulado pasemos a entrar en detalle en la implementación.
Lo primero es, como siempre, generar una solución inicial en nuestro constructor:

```
AROES::AROES(Par instance, float ftemp, int max_eval){
    //Creates a random solution
    this->instance = instance;
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.modifySolution(i,i);
    for(int i = this->instance.nClusters(); i < this->instance.nElements(); i++)
        this->instance.modifySolution(i, rand()%this->instance.nClusters());
    this->instance.shuffleSolution();
    for(int i = 0; i < this->instance.nClusters(); i++)
        this->instance.computeCentroid(i);

    this->temperature = 0.3*this->instance.getAgregated()/-log(0.3);
    this->FINAL_TEMPERATURE = ftemp;
    this->MAX_NEIGHBOURS = 5*this->instance.nElements();
    this->MAX_SUCCESS = 0.1*MAX_NEIGHBOURS;
    float M = 100000/this->MAX_NEIGHBOURS;
    this->BETA = (this->temperature - this->FINAL_TEMPERATURE)/(M * this->temperature * this->
    FINAL_TEMPERATURE);
    this->MAX_EVALUATIONS = max_eval;
}
```

Después generaremos nuestra solución.
En nuestra implementación tendremos dos bucles, el externo que se ocupa de la condición de parada y de resetear los contadores de éxitos y vecinos generados; y el interno, que se ocupa de la condición de enfriamiento. Además usaremos la variable diff, la diferencia entre el valor de la función objetivo de nuestra solución y el vecino generado, para calcular la probabilidad que tiene el vecino de ser escogido en caso de ser peor que la solución actual.
Si alguna de las soluciones obtenidas es mejor que nuestra mejor solución (la almacenada en instance) la guardamos.

```
void AROES::generateSolution(){
    vector<int> solution(instance.getSolution());
    vector<int> bud;
    float diff;
    float r;
```

```
    int generated;
    int acepted;
    int evaluations = 0;
    do{
        generated = 0;
        acepted = 0;
        //Cooling speed
        while(generated < MAX_NEIGHBOURS && acepted < MAX_SUCCESS){
            //New solution generation
            bud = generateBud(solution);
            generated++;
            //Costs difference
            diff = instance.getAgregated(bud) - instance.getAgregated(solution);
            evaluations += 2;
            r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
            //Acceptance criteria
            if(diff < 0 || r <= exp(-diff/temperature)){
                solution = bud;
                acepted++;
                evaluations += 2;
                if(instance.getAgregated(solution) < instance.getAgregated()){
                    instance.setSolution(solution);
                    for(int j = 0; j < instance.nClusters(); j++)
                        instance.computeCentroid(j);
                }
            }
        }
        //Cooling mechanism
        this->temperature = this->temperature/(1+BETA*this->temperature);
    //Stop condition
    }while(evaluations < MAX_EVALUATIONS && acepted > 0);
}
```

4. Algoritmos de comparación

El algoritmo de comparación usado es el algoritmo greedy o voraz.

El algoritmo greedy se caracteriza por su eficiencia, pero no por su eficacia, es decir, es muy rápido, pero no nos asegura que la solución sea óptima. Ésto es así porque no tiene vista a futuro, se limita a escoger la mejor opción en cada momento.

A diferencia de los algoritmos de búsqueda planteados, el greedy no parte de una solución inicial, si no que va introduciendo cada elemento en el clúster que menos restricciones incumpla.

Parte de unos centroides aleatorios y un vector de índices también aleatorio, para no empezar a introducir los elementos en orden.

```
void greedy(Par& instance){
    //Shuffle the elements and create the initial centroids
    vector<int> solution(instance.nElements(), -1);
    vector<int> indexes;
    vector<float> auxCentroid(instance.nAtributes());
    vector<int>bestOptions;
    int bestOption;
    int minInf;
    float minDev;
    bool changes;
    for(int i = 0; i < instance.nElements(); i++)
        indexes.push_back(i);
    random_shuffle(indexes.begin(), indexes.end());
    for(int i = 0; i < instance.nClusters(); i++){
        for(int j = 0; j < auxCentroid.size(); j++)
            auxCentroid[j] = rand();
        instance.setCentroid(i, auxCentroid);

    }
```

Después, el algoritmo va cogiendo elementos del vector aleatorio y mira cuales son los clústeres en los que colocarlo significaría una menor infactibilidad.

```
    do{
        //Generates the solution
        for(vector<int>::iterator it = indexes.begin(); it != indexes.end(); ++it){
            //Calculates the minor infeasibility options
            minInf = numeric_limits<int>::max();
            for(int i = 0; i < instance.nClusters(); i++){
                solution[(*it)] = i;
                if(instance.getInfeasibility(solution) < minInf){
                    bestOptions.clear();
                    bestOptions.push_back(i);
                    minInf = instance.getInfeasibility(solution);
                }
```

```
                else if(instance.getInfeasibility(solution) == minInf)
                    bestOptions.push_back(i);
            }
```

De entre ellos, escoge al que supone, además, una menor desviación general.

```
            //Choose the best option with the lowest deviation
            minDev = numeric_limits<float>::max();
            bestOption = bestOptions[0];
            for(vector<int>::iterator itOpt = bestOptions.begin(); itOpt != bestOptions.end();
     ++itOpt){
                solution[(*it)] = (*itOpt);
                if(instance.getDeviation(solution) < minDev){
                    minDev = instance.getDeviation(solution);
                    bestOption = (*itOpt);
                }
            }
            solution[(*it)] = bestOption;
        }
        instance.setSolution(solution);
```

Cuando ha terminado de rellenar la solución, recalcula los centroides y vacía la solución para volver a empezar otra vez el ciclo, hasta que los nuevos centroides calculados sean los mismos que los anteriores.

```
        //Update the centroids
        changes = false;
        for(int i = 0; i < instance.nClusters(); i++){
            auxCentroid = instance.getCentroid(i);
            instance.computeCentroid(i);
            if(auxCentroid != instance.getCentroid(i))
                changes = true;
        }
    }while(changes);
}
```

5. Procedimiento y manual de usuario

Para la implementación se ha creado un código de 0 en C++.

La estructura del proyecto es la clásica en proyectos de C++, makefile incluido.

Para esta práctica se ha optado por un main en el que se crea un objeto de la clase PAR, que incluye todas las funciones relativas al problema, pero no los métodos de búsqueda. Los métodos de búsqueda son importados de sus propios ficheros donde incluyen el código de sus clases y el método random_selection es importado del fichero utils.cpp. En cada ejecución se obtendrán los resultados de los tres algoritmos usados para esa semilla y esos datos.

Para usar el proyecto se deberá hacer make y una vez hecho, ejecutar el siguiente comando:

./bin/par número_de_clústers conjunto_de_datos conjunto_de_restricciones semilla

Para el algoritmo base el número de repeticiones escogidas es de 600,000.

Para el algoritmo memético ARO-LS solo se necesitarán 70,000 evaluaciones, se aplicará la BLS cada 10 iteraciones y el número máximo de fallos será 0.1 · el número de elementos de la solución.

Para el algoritmo híbrido ARO-ES volveremos a las 600,000 evaluaciones y tendremos una temperatura final de 10^{-3}.

Se han utilizado 4 conjuntos de datos diferentes, que se encuentran en la carpeta data junto con sus restricciones.

Éstos son el conjunto de datos iris, ecoli, rand y newthyroid.

Iris contiene información sobre las características de los tres tipos de flor de iris, por lo tanto tiene 3 clústers.

Ecoli contiene medidas sobre características de celulas usadas para predecir la localización de proteínas específicas. Tiene 8 clústers.

Rand es un conjunto de datos generado en base a distribuciones normales que cuenta con 3 grupos (clústers) bien diferenciados.

Newthyroid contiene medidas sobre la glándula tiroides de 215 pacientes. Presenta 3 clases distintas, por lo que usarán 3 clústeres.

A partir de cada conjunto de datos se obtienen dos instancias distintas con dos conjuntos de restricciones: uno con el 10 % de las restricciones posibles y otro, con el 20 %.

Se puede usar cualquier semilla, ya que el algoritmo greedy no cicla ha ciclado en ninguna de las pruebas con semillas aleatorias, pero por mayor seguridad se recomienda usar las mismas que en el experimento (0, 1, 2, 3, 4).

6. Resultados y conclusiones

Como se ha comentado en el apartado anterior las semillas utilizadas han sido la 0, la 1, la 2, la 3 y la 4.

Con ellas hemos obtenido los siguientes resultados, que se pueden visualizar al final de la página.

Analizando nuestros resultados, comprobamos que iris, rand y newthyroid son conjuntos de datos muy simples, por lo que siempre llegan al óptimo todos los algoritmos, tarden más o menos. En este caso, sería mejor hacer un greedy o un búsqueda local con estos dos conjuntos de datos, ya que no nos merece la pena esperar para obtener un resultado si con menos tiempo y una implementación más sencilla podemos llegar al óptimo.

Ecoli como siempre nos aprieta un poco más las tuercas en cuanto al tiempo y a los resultados, pero gracias a él podemos observar las diferencias y similitudes entre los distintos algoritmos.

ARO consigue obtener unos buenos resultados en cuanto al valor de la función objetivo, pero en tiempo logra unos resultados bastante mediocres. Esto se debe principalmente a que necesita más evaluaciones que de la función objetivo que otros algoritmos probados en las anteriores prácticas. Aunque sigue teniendo mejor rendimiento en tiempo que los algoritmos generacionales.
Estos resultados son normales, ya que a grandes rasgos es un algoritmo que se basa en probar soluciones aleatorias hasta encontrar una buena. A pesar de que se conserve gran parte del ADN del padre en la solución, en el problema del clústering eso no asegura que los hijos sigan siendo una buena solución, ya que la mutación es considerable y al mover de clúster tantos elementos, aunque algunos permanezcan en el mismo, nada nos asegura haber empeorado la solución. Por ese motivo, para este problema, ARO es como probar soluciones aleatorias hasta dar con la correcta.
A pesar de ello, logra llegar a buenas soluciones, lo cual me ha sorprendido gratamente.

ARO-ES es mi intento de mejorar el algoritmo tras haber fracasado estrepitosamente al intentar crear una población sobre el algoritmo ARO base. La idea era probar si el aceptar de algunas soluciones peores de vez en cuando ayudaría a llegar antes a la solución o llegar a mejores soluciones.
El resultado es otro fracaso, ya que necesita las mismas evaluaciones que ARO para llegar a una solución que no resulte insultante, aunque sigue siendo la peor solución de todas las metaheurísticas implementadas hasta la fecha (a excepción de la búsqueda local y el greedy). Lo único bueno de este algoritmo es que reduce el tiempo de ejecución de ARO a la mitad.
A pesar de los resultados, creo que jugando con los hiperparámetros del enfriamiento simulado se podría obtener un mejor resultado.

Por último, ARO-LS ha sido el mejor de los algoritmos de esta práctica, ya que con solo 70,000 evaluaciones obtiene una de las mejores soluciones medias (siendo su solución con la semilla 2 la mejor solución obtenida en todo el año) y además en el segundo mejor tiempo a excepción de la búsqueda local y del greedy.
Esto lo coloca como la verdadera mejora del algoritmo ARO. Su éxito se puede deber a que realmente estamos realizando una búsqueda local con una gran mutación, lo que nos permite escapar de los óptimos locales y

mejorar rápidamente nuestra solución.

Por este motivo, si tuviese que elegir uno de los 3 algoritmos planteados para presentárselo a mi empresa, elegiría ARO-LS.

Tabla 6.2: 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
Greedy	0	0	0	0,02	35,88	15	41,62	0,67	0,76	0	0,76	0,02	x	x	x	x
BL	0,00	0,00	0,00	0,05	30,08	25,00	41,32	2,34	0,76	0,00	0,76	0,04	x	x	x	x
AGG-UN	0,67	0,00	0,67	73,93	21,56	119,20	24,76	344,75	0,72	0,00	0,72	73,97	2,67	0,00	2,67	147,24
AGG-SF	0,67	0,00	0,67	80,55	21,41	70,60	23,31	374,39	0,72	0,00	0,72	80,89	2,67	0,00	2,67	157,89
AGE-UN	0,67	0,00	0,67	36,28	21,25	62,40	22,92	163,17	0,72	0,00	0,72	36,16	2,67	0,00	2,67	70,52
AGE-SF	0,67	0,00	0,67	38,14	21,50	65,40	23,25	168,02	0,72	0,00	0,72	37,39	2,67	0,00	2,67	72,25
AM(10, 1)	0,67	0,00	0,67	9,63	22,67	93,20	25,17	8,95	0,72	0,00	0,72	9,75	2,67	0,00	2,67	13,28
AM(10, 0,1)	0,67	0,00	0,67	43,77	21,68	64,40	23,30	60,38	0,72	0,00	0,72	43,56	2,67	0,00	2,67	69,27
AM(10, 0,1mej)	0,67	0,00	0,67	40,92	22,29	99,40	24,96	60,30	0,72	0,00	0,72	41,07	2,67	0,00	2,67	65,46
BMB	0,67	0,00	0,67	0,48	23,92	109,00	26,84	15,42	0,72	0,00	0,72	0,40	13,84	6,00	14,05	1,24
ILS	0,67	0,00	0,67	0,23	22,27	59,00	23,85	8,36	0,72	0,00	0,72	0,22	13,84	6,00	14,05	0,55
ES	0,67	0,00	0,67	0,09	21,46	109,00	24,38	1,55	0,72	0,00	0,72	0,09	13,84	6,00	14,05	0,16
ILS-ES	0,67	0,00	0,67	2,08	22,58	93,00	25,08	15,99	0,72	0,00	0,72	2,00	13,84	6,00	14,05	5,34
ARO	0,67	0,00	0,67	13,85	23,01	87,20	25,35	43,23	0,72	0,00	0,72	12,63	13,26	28,00	14,25	21,62
ARO-LS	0,67	0,00	0,67	1,69	21,47	72,00	23,40	4,80	0,72	0,00	0,72	1,58	12,65	44,40	14,27	2,76
ARO-ES	0,67	0,00	0,67	6,80	28,32	220,20	34,23	21,71	0,72	0,00	0,72	6,36	13,84	6,00	14,05	10,81

Tabla 6.2: 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
Greedy	0	0	0	0,02	30,16	0	30,16	0,77	0,76	0	0,76	0,02	x	x	x	x
BL	0,00	0,00	0,00	0,02	28,98	1,60	29,70	2,20	0,76	0,00	0,76	0,02	x	x	x	x
AGG-UN	0,67	0,00	0,67	144,65	21,87	113,60	23,39	672,02	0,72	0,00	0,72	137,70	2,67	0,00	2,67	279,68
AGG-SF	0,67	0,00	0,67	154,07	21,35	120,60	22,97	713,78	0,72	0,00	0,72	147,17	2,67	0,00	2,67	287,36
AGE-UN	0,67	0,00	0,67	65,96	21,54	90,60	22,75	305,63	0,72	0,00	0,72	64,21	2,67	0,00	2,67	132,87
AGE-SF	0,67	0,00	0,67	67,35	21,50	110,60	22,98	313,51	0,72	0,00	0,72	66,37	2,67	0,00	2,67	145,62
AM(10, 1)	0,67	0,00	0,67	17,28	22,53	120,20	24,15	15,57	0,72	0,00	0,72	16,65	2,67	0,00	2,67	25,04
AM(10, 0,1)	0,67	0,00	0,67	81,72	21,81	133,80	23,60	115,55	0,72	0,00	0,72	78,78	2,67	0,00	2,67	138,33
AM(10, 0,1mej)	0,67	0,00	0,67	74,96	21,77	140,00	23,71	115,52	0,72	0,00	0,72	73,10	2,67	0,00	2,67	129,76
BMB	0,67	0,00	0,67	0,50	23,66	190,00	26,21	14,75	0,72	0,00	0,72	0,44	14,29	0,00	14,29	1,27
ILS	0,67	0,00	0,67	0,28	23,23	118,00	24,81	11,33	0,72	0,00	0,72	0,24	14,29	0,00	14,29	0,78
ES	0,67	0,00	0,67	0,08	19,58	278,00	23,31	2,17	0,72	0,00	0,72	0,08	13,55	41,00	14,30	0,25
ILS-ES	0,67	0,00	0,67	3,72	22,80	220,00	25,75	28,33	0,72	0,00	0,72	3,35	14,29	0,00	14,29	10,62
ARO	0,67	0,00	0,67	16,44	22,90	175,60	25,26	57,45	0,72	0,00	0,72	15,30	14,29	0,00	14,29	27,01
ARO-LS	0,67	0,00	0,67	2,19	21,91	148,80	23,91	6,54	0,72	0,00	0,72	2,05	13,45	61,60	14,57	3,75
ARO-ES	0,67	0,00	0,67	8,25	27,24	372,20	32,23	28,68	0,72	0,00	0,72	7,60	14,14	8,20	14,29	13,50

Tabla 6.1: Resultados obtenidos por el algoritmo ARO 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
Ejecución 1	0,67	0,00	0,67	13,76	22,49	82,00	24,69	43,46	0,72	0,00	0,72	12,53	13,84	6,00	14,05	21,58
Ejecución 2	0,67	0,00	0,67	13,76	22,49	82,00	24,69	43,46	0,72	0,00	0,72	12,53	13,84	6,00	14,05	21,58
Ejecución 3	0,67	0,00	0,67	14,12	24,97	120,00	28,19	43,09	0,72	0,00	0,72	12,65	10,82	116,00	15,06	21,86
Ejecución 4	0,67	0,00	0,67	13,86	22,60	80,00	24,74	42,97	0,72	0,00	0,72	12,74	13,84	6,00	14,05	21,55
Ejecución 5	0,67	0,00	0,67	13,77	22,52	72,00	24,45	43,19	0,72	0,00	0,72	12,69	13,84	6,00	14,05	21,54
Media	**0,67**	**0,00**	**0,67**	**13,85**	**23,01**	**87,20**	**25,35**	**43,23**	**0,72**	**0,00**	**0,72**	**12,63**	**13,23**	**28,00**	**14,25**	**21,62**

Tabla 6.1: Resultados obtenidos por el algoritmo ARO 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
Ejecución 1	0,67	0,00	0,67	16,58	22,71	161,00	24,87	57,16	0,72	0,00	0,72	15,20	14,29	0,00	14,29	26,86
Ejecución 2	0,67	0,00	0,67	16,58	22,71	161,00	24,87	57,16	0,72	0,00	0,72	15,20	14,29	0,00	14,29	26,86
Ejecución 3	0,67	0,00	0,67	16,38	24,06	204,00	26,79	58,78	0,72	0,00	0,72	15,44	14,29	0,00	14,29	27,37
Ejecución 4	0,67	0,00	0,67	16,30	22,67	187,00	25,18	57,14	0,72	0,00	0,72	15,42	14,29	0,00	14,29	26,90
Ejecución 5	0,67	0,00	0,67	16,35	22,37	165,00	24,58	56,99	0,72	0,00	0,72	15,26	14,29	0,00	14,29	27,08
Media	**0,67**	**0,00**	**0,67**	**16,44**	**22,90**	**175,60**	**25,26**	**57,45**	**0,72**	**0,00**	**0,72**	**15,30**	**14,29**	**0,00**	**14,29**	**27,01**

Tabla 6.1: Resultados obtenidos por el algoritmo ARO-LS 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
Ejecución 1	0,67	0,00	0,67	1,68	21,45	62,00	23,12	4,92	0,72	0,00	0,72	1,62	13,84	6,00	14,05	2,74
Ejecución 2	0,67	0,00	0,67	1,70	21,45	62,00	23,12	4,92	0,72	0,00	0,72	1,56	13,84	6,00	14,05	2,72
Ejecución 3	0,67	0,00	0,67	1,68	19,59	115,00	22,68	4,68	0,72	0,00	0,72	1,57	10,82	105,00	14,66	2,72
Ejecución 4	0,67	0,00	0,67	1,69	22,36	60,00	23,97	4,76	0,72	0,00	0,72	1,57	13,84	6,00	14,05	2,86
Ejecución 5	0,67	0,00	0,67	1,69	22,48	61,00	24,11	4,70	0,72	0,00	0,72	1,59	10,90	99,00	14,52	2,75
Media	**0,67**	**0,00**	**0,67**	**1,69**	**21,47**	**72,00**	**23,40**	**4,80**	**0,72**	**0,00**	**0,72**	**1,58**	**12,65**	**44,40**	**14,27**	**2,76**

Tabla 6.1: Resultados obtenidos por el algoritmo ARO-LS 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
Ejecución 1	0,67	0,00	0,67	2,18	21,88	145,00	23,82	6,53	0,72	0,00	0,72	2,04	14,29	0,00	14,29	3,76
Ejecución 2	0,67	0,00	0,67	2,17	21,88	145,00	23,82	6,55	0,72	0,00	0,72	2,05	14,29	0,00	14,29	3,76
Ejecución 3	0,67	0,00	0,67	2,17	21,86	169,00	24,13	6,55	0,72	0,00	0,72	2,05	13,55	41,00	14,30	3,74
Ejecución 4	0,67	0,00	0,67	2,26	21,96	148,00	23,95	6,55	0,72	0,00	0,72	2,05	14,29	0,00	14,29	3,77
Ejecución 5	0,67	0,00	0,67	2,18	21,98	137,00	23,81	6,50	0,72	0,00	0,72	2,05	10,82	267,00	15,70	3,73
Media	**0,67**	**0,00**	**0,67**	**2,19**	**21,91**	**148,80**	**23,91**	**6,54**	**0,72**	**0,00**	**0,72**	**2,05**	**13,45**	**61,60**	**14,57**	**3,75**

Tabla 6.1: Resultados obtenidos por el algoritmo ARO-ES 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
Ejecución 1	0,67	0,00	0,67	6,77	28,25	150,00	32,27	21,75	0,72	0,00	0,72	6,34	13,84	6,00	14,05	10,75
Ejecución 2	0,67	0,00	0,67	6,79	28,25	150,00	32,27	21,65	0,72	0,00	0,72	6,34	13,84	6,00	14,05	10,70
Ejecución 3	0,67	0,00	0,67	6,81	27,37	314,00	35,79	21,68	0,72	0,00	0,72	6,24	13,84	6,00	14,05	10,82
Ejecución 4	0,67	0,00	0,67	6,79	28,10	177,00	32,85	21,72	0,72	0,00	0,72	6,41	13,84	6,00	14,05	10,96
Ejecución 5	0,67	0,00	0,67	6,84	29,65	310,00	37,96	21,77	0,72	0,00	0,72	6,48	13,84	6,00	14,05	10,81
Media	**0,67**	**0,00**	**0,67**	**6,80**	**28,32**	**220,20**	**34,23**	**21,71**	**0,72**	**0,00**	**0,72**	**6,36**	**13,84**	**6,00**	**14,05**	**10,81**

Tabla 6.1: Resultados obtenidos por el algoritmo ARO-ES 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
Ejecución 1	0,67	0,00	0,67	8,13	27,29	252,00	30,67	28,68	0,72	0,00	0,72	7,53	14,29	0,00	14,29	13,54
Ejecución 2	0,67	0,00	0,67	8,11	27,29	252,00	30,67	28,54	0,72	0,00	0,72	7,64	14,29	0,00	14,29	13,55
Ejecución 3	0,67	0,00	0,67	8,14	28,72	516,00	35,64	28,65	0,72	0,00	0,72	7,59	13,55	41,00	14,30	13,38
Ejecución 4	0,67	0,00	0,67	8,70	27,36	338,00	31,89	28,90	0,72	0,00	0,72	7,56	14,29	0,00	14,29	13,57
Ejecución 5	0,67	0,00	0,67	8,16	25,53	503,00	32,27	28,64	0,72	0,00	0,72	7,68	14,29	0,00	14,29	13,44
Media	**0,67**	**0,00**	**0,67**	**8,25**	**27,24**	**372,20**	**32,23**	**28,68**	**0,72**	**0,00**	**0,72**	**7,60**	**14,14**	**8,20**	**14,29**	**13,50**