



# Practica 8 - DAA . Max-mean dispersión problem

## Introducción

El problema de la máxima dispersión media es un problema que parte de los siguientes elementos: \* Un grafo  $G = (V, E)$  donde: --  $V$  = Conjunto de vértices de longitud  $|V|$ . --  $E$  = Conjunto de aristas que interconectan los vértices/nodos anteriores. \* Una **función objetivo** ,  $f(S)$ , por la cual se evalúa la optimalidad de la solución encontrada tal que: 

-- Donde  $S$  = Conjunto de vértices que representan una solución al problema *Max-mean dispersión*.

--  $|S|$  = Longitud del conjunto.

--  = Sumatorio de los costes (representado por la función  $d(i,j)$ ) de todas aquellas aristas que interconectan los vertices/nodos seleccionados y almacenados en la solución  $S$ .

Para la búsqueda de una solución óptima al problema al problema expuesto anteriormente, se diseña e implementan 4 algoritmos de búsqueda distintos: \* Una búsqueda **voraz constructiva** \* Una búsqueda **voraz destructiva** \* Una búsqueda **GRASP** (*Greedy Randomized Adaptive Search Procedure*) \* Una búsqueda **MultiArranque** \* Una búsqueda **VNS** (*Variable Neighborhood Search*)

El fundamento de estos algoritmos y su implementación en el código se explicarán más adelante.

## Lectura del Grafo

Antes de empezar con la búsqueda de una solución al problema hay que cargar los datos del grafo a la memoria de nuestro programa. Previo a explicación de cómo se carga, el formato estándar elegido para representar nuestro grafo en un fichero de texto plano es el que se presenta en el siguiente ejemplo (los comentarios al lado no deben estar y son solo a modo explicativo):

```
4 //Número de vértices
-4 //d(1, 2) = d(2, 1)
-4 //d(1, 3) = d(3, 1)
6 //d(1, 4) = d(4, 1)
6 //d(2, 3) = d(3, 2)
10 //d(2, 4) = d(4, 2)
-7 //d(3, 4) = d(4, 3)
```

La lectura y carga de ficheros con el formato previo es tarea de la clase **Grafo**:

```
class Grafo {
...
public:
    ~Grafo();
    explicit Grafo(const Grafo& other) = delete; //Prohibido copia
    explicit Grafo(std::istream& input, bool debug = VERBOSE); //Nuevo
...
};
```

Por simplificar el código, se han eliminado aquellas partes que no son intereses de explicar ahora, pero si más adelante.

Antes de empezar, es importante hacer notar que, como estamos programando en C++, y para evitar el mayor número de

problemas en la gestión de objetos y punteros en este lenguaje, todos los constructores han de ser explícitos, de forma que no se ejecute nada de forma implícita (y, por lo tanto, no prevista por nosotros). Los constructores copias, por lo general, excepto cuando de verdad sean imprescindibles, son borrados. Esta filosofía de trabajo de clases se seguirá viendo en el resto del código de nuestro programa. Ante la incertidumbre de si vamos a tener que refactorizar creando una clase hija, los métodos o propiedades de la clase privados no existen.

Volviendo a la explicación, nuestro **Grafo** solo se carga desde el constructor y este recibe 2 argumentos: 1) Una instancia *std::istream*, esto es, una tubería de entrada por el cual nuestro programa pueda leer del fichero. Esta tubería, en nuestro código, se crea creando una instancia *std::ifstream* con los siguientes argumentos:

```
//argv[1] debe contener la ruta a 1 fichero válido.  
std::ifstream input (argv[1], std::ifstream::in | std::ifstream::binary);
```

Y dado que *std::ifstream* hereda de *std::istream* podemos pasarlo a nuestra clase **Grafo** directamente:

```
Grafo* grafo = new Grafo(input);
```

Una vez tenemos abierto nuestro fichero con el formato presentado anteriormente, al leer la primera fila, que se corresponde con el número de vértices, seteamos la propiedad que guarda el nº de vértices en nuestro grado:

```
class Grafo {  
...  
protected:  
    size_t _numVertices = -1; //recordar size_t es unsigned  
...  
};
```

E inicializamos y guardamos todos los vértices con los que trabajaremos justo después en la siguiente propiedad de nuestra clase:

```
class Grafo {  
...  
protected:  
    std::map<const int, Vertice*> _vertices; //key = id vertice, value = Clase vertice  
...  
};
```

Después de leer la primera fila, se sigue leyendo y creando las estructuras de datos explicadas justo delante, así hasta que se alcanza el final de línea del fichero.

Para trabajar con los vértices, en nuestro programa trabajamos con la clase **Vertice**: (contenido dentro de la clase Grafo a causa de la relación de dependencia entre ambas)

```

class Vertice {
protected:
    int _id = -1;
    std::shared_ptr<const Grafo*> _grafo;
    std::map<const int, const Arista> _aristas; // id vertice_vecino -> coste_viaje
public:
    ~Vertice();
    explicit Vertice(const Vertice& other) = delete; //Constructor copia prohibido
    explicit Vertice(const std::shared_ptr<const Grafo*>&, const int);
    //explicit Vertice(const std::shared_ptr<const Grafo*>&, const Vertice& other); // = delete;

    const int getId() const { return this->_id; }

    void addVecino(const int id, const Arista& arista);
    const std::map<const int, const Arista> getAristas() const { return this->_aristas; }
};

```

La clase es bastante autoexplicativa. El grafo tiene un conjunto de vértices y cada vértice tiene un conjunto de aristas que le conectan con otros vértices. Por supuesto, en nuestro programa también trabajamos con una clase **Arista**:

```

class Arista {
public:
    explicit Arista(); //Inválido
    explicit Arista(int idNodo1, int idNodo2, int coste); //Nuevo
    explicit Arista(const Arista& other); //Copia
    int idNodo1 = -1;
    int idNodo2 = -1;
    int coste = 0;

    friend const bool operator ==(const Arista& a1, const Arista& a2) { return ((a1.idNodo1 == a2.idNodo1) && (a1.idNodo2 == a2.idNodo2) && (a1.coste == a2.coste)); }
    friend const bool operator !=(const Arista& a1, const Arista& a2) { return !(a1==a2); }
    friend bool operator <(const Arista& a1, const Arista& a2) { return a1.coste < a2.coste; }
    friend bool operator >(const Arista& a1, const Arista& a2) { return a1.coste > a2.coste; }
    const bool operator <(int otroCoste) { return coste < otroCoste; }
    const bool operator >(int otroCoste) { return coste > otroCoste; }
};

```

La filosofía de esta clase es que sea como mi dato básico (al nivel de int, double, float, ...) y por ello que la marque como un *Struct* y no tenga ni un solo método pero sí muchas sobrecargas de operadores.

Una vez hemos explicado cómo funciona la carga de un Grafo y las estructuras de datos que usamos para guardar esos datos, procedemos a explicar los algoritmos de búsqueda que iteran sobre ello en busca del valor máximo de la dispersión media.

## Busqueda solución óptima

Los algoritmos que enumeramos anteriormente, todos ellos, heredan de una clase base abstracta **BusquedaMD**, con la siguiente estructura:

```

class BusquedaMD {
protected:
    size_t _iteracion = 0;
    size_t _bestIteracion = 0;

    std::unique_ptr<const Grafo*> _grafo;

    SolucionMD* generarSolucionAleatoria(const Grafo*);
public:
    explicit BusquedaMD(const Grafo* grafo);
    explicit BusquedaMD(const BusquedaMD& other) = delete;

    double funcionObjetivo(const Grafo* grafo , const SolucionMD* const S) const; //md
    SolucionMD* busquedaMejor(SolucionMD* sol = nullptr) ;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) = 0;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) = 0;
    virtual const bool condicionParada(SolucionMD*& S,SolucionMD* Sx) = 0;
};

```

Esta clase abstracta, sólo implementa 2 métodos:

1) La primera, **busquedaMejor** que es desde la cual se va a llamar al resto de métodos (que las hijas deben implementar) 2) La segunda, **funcionObjetivo**, que guarda la función objetivo explicado en la introducción.

**busquedaMejor:**

```

SolucionMD* BusquedaMD::busquedaMejor(SolucionMD* sol )
{
    const Grafo* grafo = *this->_grafo.get();
    SolucionMD* S = (sol == nullptr) ? primeraMejorSolucion(*this->_grafo.get()) : sol;
    SolucionMD* Sx = nullptr;
    do {
        Sx = algoritmo(grafo, S);
        Sx->setScore(this->funcionObjetivo(grafo, Sx));
    } while(condicionParada(S,Sx));
    return S;
}

```

**funcionObjetivo:**

```

double BusquedaMD::funcionObjetivo(const Grafo* grafo , const SolucionMD* const S ) const //core
{
    std::unordered_set<Arista, AristaHash, AristaEqual> visitados;
    double sumatorio = 0;
    size_t numNodos = S->getListaVertices().size();
#pragma omp parallel for
    for (int i = 0; i < numNodos; i++) {
        int id1 = (*S)[i];
        for (int j = 0, id2 = (*S)[j]; j < numNodos; j++, id2 = (*S)[j]) {
            const auto aristas = grafo->getVertices().at(id1)->getAristas();
            const auto it = aristas.find(id2);
#pragma omp critical
            {
                if (it != aristas.end())
                    if (visitados.insert(it->second).second)
                        sumatorio += it->second.coste;
            }

        }
    }

    visitados.clear();
    return (sumatorio / numNodos);
}

```

De lo mostrado anteriormente, cabe destacar 2 cosas: 1) Los **"#pragma omp"** son directivas de la librería OpenMP para programación paralela (el uso de esta librería debe notificarse al compilador pasándole la opción `"/openmp"`). De forma que el `for` marcado con `"#pragma omp parallel for"` sirve para notificar que el `for` justo delante puede y debe ejecutarse en múltiples hilos de ejecución y `"#pragma omp critical"` sirve para notificar que lo que sigue es una zona crítica y que el hilo que entre dentro bloqueará al resto. 2) Más importante, la clase que se retorna en `"busquedaMejor"`, **SolucionMD**, es la estructura planteada para guardar las soluciones a nuestro problema:

```

class SolucionMD {
protected:
    std::set<int> _vertices;
    //std::vector<Arista> _solucion;
    double _score = -INFINITY;
public:
    explicit SolucionMD(){} //Nuevo vacio
    explicit SolucionMD(const int& vertice1, const int& vertice2); //Nuevo
    explicit SolucionMD(const SolucionMD& other); //Copia

    const std::set<int>& getListVertices() const { return this->_vertices; }

    const bool addVertice(const int& idVertice);
    const bool removeVertice(const int& idVertice);

    const double getScore() const { return this->_score; }
    void setScore(const double& mdm) { this->_score = mdm; }

    const size_t size() const { return this->_vertices.size(); }
    const bool isPresent(const int& idVert) const { return (this->_vertices.find(idVert) != this->_ver

    const int operator [](const int& indice) const;
    friend const bool operator ==(const SolucionMD& s1, const SolucionMD& s2);
    friend SolucionMD* operator +(const SolucionMD& s, const int& idVert);
    friend SolucionMD* operator -(const SolucionMD& s, const int& idVert);
};

```

"\_vertices" guarda los id de los vértices de nuestra solución "\_score" guarda el resultado de la funcionObjetivo de nuestra solución. El operador [] devuelve el id del nodo en índice i de la lista "\_vertices"El operador + **crea una copia** de la solución actual y le añade (con "addVertice") el id pasado como argumento(**devolviendo error si ya existía**) El operador -, por lo tanto, hace lo contrario, **crea una copia** de la solución actual y le elimina el id(con "removeVertice") pasado como argumento (**devolviendo error si no existía previamente**) Los operadores son útiles para reducir la longitud de nuestro código. El resto de métodos, el propio nombre del método es autoexplicativo.

Ya hemos definido la mayoría de las estructuras que se usarán en los algoritmos , ahora vamos a explicar la propia implementación de los algoritmos:

## Algoritmo Voraz Constructivo

Nuestro algoritmo Voraz Constructivo se implementa en la clase **BusquedaMDVorazConstructiva** , la cual hereda de la anteriormente nombrada **BusquedaMD** :

```

class BusquedaMDVorazConstructiva : public BusquedaMD {
public:
    explicit BusquedaMDVorazConstructiva(const Grafo* grafo);
    explicit BusquedaMDVorazConstructiva(const BusquedaMDVorazConstructiva& other) = delete;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) override;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) override;
    virtual const bool condicionParada(SolucionMD*& S, SolucionMD* Sx) override;
};

```

Es importante recordar la estructura y secuencia de llamadas de **"busquedaMejor"** explicada anteriormente porque el resto de clases:

Antes de entrar en el bucle (a modo de preProcesamiento) se llama a **"primeraMejorSolucion"** . Después, dentro del bucle (a modo de procesamiento) se llama a **"algoritmo"** y , para comprobación de la condición de parada del bucle (también a modo de postProcesamiento) se llama finalmente a **"condicionParada"** .

Volviendo a la explicación de nuestro algoritmo, las 3 fases anteriores se pueden resumir:

1) **primeraMejorSolucion** : Devuelve como primer solución aquella que se consigue de buscar la arista (y por lo tanto, los 2 nodos que se conectan con ella) con mayor coste de nuestro Grafo "grafo". 2) **algoritmo** : Ejecuta el propio algoritmo greedy, por el cual, desde la solución inicial conseguida anteriormente, se va iterando por los nodos vecinos de los nodos en la solución, añadiéndoles a la solución actual para comprobar si mejora la solución. Si no la mejora, se descarta y se busca el siguiente vecino. Si la mejora, se marca como solución posible. En ambos casos, se itera por todos los nodos vecinos con la esperanza de encontrar una que lo mejore. Por lo tanto, al final, pueden pasar 2 cosas: 2.1 No se encuentra ningún nodo que mejore la solución actual, la solución inicial es óptima y se devuelve. 2.2 Se encuentra un nodo que mejore la solución actual, la solución inicial es sustituida por la mejor y se devuelve. 3) **condicionParada** : Si la solución devuelta anteriormente en la iteración es distinto a la solución inicial, el bucle continua. Sino, si la fase previa no logró mejorar la solución, se detiene el bucle.

## Algoritmo Voraz Alternativo

Implementado en la clase **BusquedaMDVorazAlternativa**:

```
class BusquedaMDVorazAlternativa : public BusquedaMD {
public:
    explicit BusquedaMDVorazAlternativa(const Grafo* grafo);
    explicit BusquedaMDVorazAlternativa(const BusquedaMDVorazAlternativa& other) = delete;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) override;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) override;
    virtual const bool condicionParada(SolucionMD*& S, SolucionMD* Sx) override;
};
```

Es exactamente igual que el Voraz Constructivo pero invertido. La solución inicial parte con todos los nodos del grafo añadido, luego, en el algoritmo, se van eliminando aquellos nodos que mejoran la solución y la condición de parada es la misma, hasta que no exista una iteración sin mejora no se detiene.

## Algoritmo GRASP

Implementando en la clase **BusquedaMDGRASP** :

```

class BusquedaMDGRASP : public BusquedaMD {
protected:
    const size_t _itSinMejora = 0;
    size_t _countItSinMejora = 0;
    size_t _lrcSize = 0;
    float _lrcDelta = 0;

    virtual SolucionMD* faseConstructiva(const Grafo*, SolucionMD* S);
    virtual int* construirLRC(const Grafo*, SolucionMD* S);

public:
    explicit BusquedaMDGRASP(const Grafo* grafo, const float delta, const size_t itSinMejora = 500);
    explicit BusquedaMDGRASP(const BusquedaMDGRASP& other) = delete;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) override;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) override;
    virtual const bool condicionParada(SolucionMD*& S, SolucionMD* Sx) override;
};

```

En el constructor de la búsqueda GRASP, se recibe 2 argumentos adicionales, "*delta*" que se refiere a **qué porcentaje del tamaño original de nodos** tendrá la LRC (Lista Rrestringida de Ccandidatos) y "*itSinMejora*" que hace referencia a lo que el propio nombre indica, el número de iteraciones sin detectar una mejora (por defecto a 500) para que la condición de parada se evalúe a true y se salga del bucle. Ahora, describiremos qué hacemos en cada fase:

1) **primeraMejorSolucion** : Sería el hueco para un preprocesamiento, por ejemplo, descartar aquellos nodos con todas las aristas negativas, pero por culpa de una filosofía inicial con la estructura de nuestros datos, la clase "Grafo" que no admite ningún cambio ni copia, no fué posible 2) **algoritmo** : Dentro de esta fase hay 2 subfases: 2.1 **faseConstructiva** : Dentro de esta fase, a partir de una solución inicial previa (que si es la primera vez será vacía) se empieza añadiendo un vértice aleatoria no presente ya en la solución. Con esa nueva solución se forma una LRC de tamaño (*numVertices \* delta*) que contendrá, ordenado de mayor a menor afinidad, esto es, de mayor a menor incremento en la función objetivo, los ids no presentes en la solución. De esta LRC se selecciona uno al azar y se le añade a la solución. 2.2 Esta solución luego se pasa a otra clase encargada de las búsquedas por entorno local, la clase **BusquedaEntornoLocal** :

```

class BusquedaEntornoLocal {
    const Grafo* _grafo;
    const SolucionMD* _solucion;
    const BusquedaMD* _busqueda;
public:
    BusquedaEntornoLocal() = delete;
    BusquedaEntornoLocal(const BusquedaEntornoLocal&) = delete;
    BusquedaEntornoLocal(const Grafo* grafo, const SolucionMD* solucion, const BusquedaMD* busqueda) :

        SolucionMD* operator()();
};

```

La clase es bastante sencilla y adhoc para refactorizar código. Básicamente lo que hace **BusquedaEntornoLocal** son 2 *movimientos*. 1) Primer movimiento, a partir de una solución inicial, remueve el vértice de la solución con menor afinidad, aquella que produce menor pérdida de función objetivo sacarla 2) Segundo movimiento, insertar otro nodo no presente en la solución inicial que mejore la solución inicial. Estos 2 movimientos se repiten en bucle hasta que en el 2º movimiento no se logra encontrar una solución mejor.

Volviendo a la clase **BusquedaMDGRASP**, la condición de parada implementada en "**condicionParada**" es que ocurran *itSinMejora* iteraciones sin una mejora



# Algoritmo MultiArranque

```
class BusquedaMDMultiArranque : public BusquedaMD {
protected:
    const size_t _itSinMejora = 0;
    size_t _countItSinMejora = 0;
public:
    explicit BusquedaMDMultiArranque(const Grafo* grafo, const size_t itSinMejora = 500);
    explicit BusquedaMDMultiArranque(const BusquedaMDMultiArranque& other) = delete;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) override;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) override;
    virtual const bool condicionParada(SolucionMD*& S, SolucionMD* Sx) override;
};
```

Este algoritmo es mucho más sencillo que los anteriores, en cada fase se realiza lo siguiente:

1) **primeraMejorSolucion** : Se crea una solución enteramente aleatorio (tanto el tamaño y los nodos elegidos) y se devuelve 2) **algoritmo**: A esa solución aleatoria se le pasa a **BusquedaEntornoLocal** para que la mejore mediante una búsqueda local (explicada anteriormente, una búsqueda que analiza las soluciones posibles inmediatamente vecinas). 3) **condicionParada** : La condición de parada es que exista unas 500 iteraciones sin una sola mejora.

## Algoritmo VNS

Terminamos la documentación explicando en qué consiste el algoritmo **VNS (Variable Neighborhood Search)**. Este algoritmo se implementa en la clase **BusquedaMDVNS**

```
class BusquedaMDVNS : public BusquedaMD {
protected:
    const size_t _itSinMejora = 0;
    size_t _countItSinMejora = 0;

    std::vector<SolucionMD*>* _entornos = nullptr;
    size_t _maxEntornosSize = UINT_MAX;

    std::vector<SolucionMD*>* generarEntornos(const Grafo* grafo, const SolucionMD* S);
    SolucionMD* agitarEntorno(const Grafo* grafo, const SolucionMD* const entorno);
public:
    ~BusquedaMDVNS() { this->_entornos->clear(); delete this->_entornos; }
    explicit BusquedaMDVNS(const Grafo* grafo, const size_t itSinMejora = 50, const size_t maxEntornosSize = 20);
    explicit BusquedaMDVNS(const BusquedaMDVNS& other) = delete;

    virtual SolucionMD* primeraMejorSolucion(const Grafo* grafo) override;
    virtual SolucionMD* algoritmo(const Grafo* grafo, const SolucionMD* const S) override;
    virtual const bool condicionParada(SolucionMD*& S, SolucionMD* Sx) override;
};
```

Esta clase recibe en el constructor, nuevamente, 2 argumentos adicionales, '*itSinMejora*' o el número de repeticiones de **algoritmo** sin devolver una mejora que va a realizar antes de detener el algoritmo y '*maxEntornosSize*' que setea (por defecto 20) el tamaño máxima de la lista (*\_entornos*) que va a guardar todos esos entornos. Nuevamente, algoritmo dividido en 3 fases: 1)

**primeraMejorSolucion** : Se genera una solución totalmente aleatoria (como en MultiArranque) y con esa solución, se generan todos los entornos posibles con centro cada uno de los nodos presentes en la solución (todos los que quepan en una lista de *maxEntornosSize*). 2) **algoritmo** : En esta fase, se inicia otro bucle donde se itera por cada uno de los entornos generados previamente que estén dentro de la lista *\_entornos*, por cada entorno, se le hace un *"agitado"*, que consiste, brevemente, en "agitar" el entorno para sacar un vértice y meter otro no presente, de forma que el entorno se desplaza de forma aleatoria, permitiéndonos hacer saltos y poder escapar de algún óptimo local (problema que se sucede en todos los algoritmos previos). De este nuevo entorno se obtiene una solución y esta es pasada a **BusquedaEntornoLocal** para hacer una búsqueda local dentro del entorno de la solución y devolver la mejor solución local encontrada si existiese. 3) Como en las anteriores (excepto en las Greedy), el criterio de parada es que se produzca número de iteraciones sin mejora = *itSinMejora*.

# Análisis

## Especificaciones máquina

### CPU

- Nombre : Intel Core i7-6700k
- Frecuencia (base-turbo) : 4 Ghz - 4.2 GHz
- Núcleos: 4
- Hilos: 8
- Arquitectura: Skylake (64 bits)
- Caché L1 : 64 KB (por núcleo)
- Caché L2: 256 KB(por núcleo)
- Caché L3: 8192 KB
- Interfaz de memoria: DDR4-1866/2133, DDR3L-1333/1600

### Memoria

- Capacidad: 16 GB
- Anchura : 128 bits
- Frecuencia: 2400 MHz

## Algoritmo Voraz

Problema	N	Ejecución	md	CPU
max-mean-div-10.txt	10	1	10.1429	6.1665 ms
max-mean-div-10.txt	10	2	10.1429	4.984 ms
max-mean-div-10.txt	10	3	10.1429	3.1842 ms
max-mean-div-10.txt	10	4	10.1429	4.0077 ms
max-mean-div-10.txt	10	5	10.1429	3.2423 ms
max-mean-div-15.txt	15	1	9.5	2.6585 ms
max-mean-div-15.txt	15	2	9.5	2.9053 ms
max-mean-div-15.txt	15	3	9.5	2.8281 ms
max-mean-div-15.txt	15	4	9.5	3.4658 ms

Problema	N	Ejecución	md	CPU
max-mean-div-15.txt	15	1	9.5	2.9696 ms
max-mean-div-20.txt	20	1	12.8571	13.8072 ms
max-mean-div-20.txt	20	2	12.8571	17.7031 ms
max-mean-div-20.txt	20	3	12.8571	16.5597 ms
max-mean-div-20.txt	20	4	12.8571	16.5597 ms
max-mean-div-20.txt	20	5	12.8571	14.7064 ms

## Algoritmo Voraz Alternativo

Problema	N	Ejecución	md	CPU
max-mean-div-10.txt	10	1	14	10.6744 ms
max-mean-div-10.txt	10	2	14	10.0406 ms
max-mean-div-10.txt	10	3	14	10.9923 ms
max-mean-div-10.txt	10	4	14	11.2049 ms
max-mean-div-10.txt	10	5	14	12.1292 ms
max-mean-div-15.txt	15	1	9.83333	112.69 ms
max-mean-div-15.txt	15	2	9.83333	92.1789 ms
max-mean-div-15.txt	15	3	9.83333	95.9177 ms
max-mean-div-15.txt	15	4	9.83333	93.5733 ms
max-mean-div-15.txt	15	5	9.83333	92.3261 ms
max-mean-div-20.txt	20	1	12.8571	456.408 ms

Problema	N	Ejecución	md	CPU
max-mean-div-20.txt	20	3	12.8571	452.744 ms
max-mean-div-20.txt	20	3	12.8571	464.504 ms
max-mean-div-20.txt	20	4	12.8571	451.607 ms
max-mean-div-20.txt	20	5	12.8571	459.484 ms

## Algoritmo GRASP

Problema	N	LRC	Ejecución	md	CPU
max-mean-div-10.txt	10	2	1	11.7143	118.654 ms
max-mean-div-10.txt	10	4	2	12.8	138.885 ms
max-mean-div-10.txt	10	6	3	12.8	110.685 ms
max-mean-div-10.txt	10	8	4	12.8	104.504 ms
max-mean-div-10.txt	10	10	5	12.8	121.029 ms
max-mean-div-15.txt	15	3	1	9.5	164.413 ms
max-mean-div-15.txt	15	6	2	9.5	160.036 ms
max-mean-div-15.txt	15	9	3	9.5	154.824 ms
max-mean-div-15.txt	15	12	4	9.75	281.264 ms
max-mean-div-15.txt	15	15	5	9.33333	152.61 ms
max-mean-div-20.txt	20	4	1	12.8571	253.001 ms
max-mean-div-20.txt	20	8	2	5	205.059 ms
max-mean-div-20.txt	20	12	3	12.8571	244.123 ms

Problema	N	LR	Ejecución	md	CPU
max-mean-div-20.txt	20	10	5	12.8571	219.043 ms
max-mean-div-20.txt	20	20	5	12.8571	295.155 ms

## Algoritmo MultiArranque

Problema	N	Ejecución	md	CPU
max-mean-div-10.txt	10	1	12.8	87.2331 ms
max-mean-div-10.txt	10	2	12.8	79.8559 ms
max-mean-div-10.txt	10	3	12.8	101.708 ms
max-mean-div-10.txt	10	4	12.8	105.76 ms
max-mean-div-10.txt	10	5	12.8	159.646 ms
max-mean-div-15.txt	15	1	7	176.762 ms
max-mean-div-15.txt	15	2	7	145.13 ms
max-mean-div-15.txt	15	3	7	182.526 ms
max-mean-div-15.txt	15	4	5	141.299 ms
max-mean-div-15.txt	15	5	9.5	528.75 ms
max-mean-div-20.txt	20	1	10.75	754.504 ms
max-mean-div-20.txt	20	2	5	202.781 ms
max-mean-div-20.txt	20	3	5	200.683 ms
max-mean-div-20.txt	20	4	10.75	232.49 ms
max-mean-div-20.txt	20	5	12.8571	353.553 ms

## Algoritmo VNS

Problema	N	K(max)	Ejecución	md	CPU
max-mean-div-10.txt	10	5	1	14	91.4151 ms
max-mean-div-10.txt	10	10	2	14	221.354 ms
max-mean-div-10.txt	10	15	3	14	203.791 ms
max-mean-div-10.txt	10	20	4	14	192.107 ms
max-mean-div-10.txt	10	25	5	12.8	180.76 ms
max-mean-div-15.txt	15	5	1	9.5	158.746 ms
max-mean-div-15.txt	15	10	2	9.75	616.786 ms
max-mean-div-15.txt	15	15	3	9.75	472.275 ms
max-mean-div-15.txt	15	20	4	9.75	522.093 ms
max-mean-div-15.txt	15	52	5	9.75	600.191 ms
max-mean-div-20.txt	20	5	1	13.1667	632.736 ms
max-mean-div-20.txt	20	10	2	13.1667	981.466 ms
max-mean-div-20.txt	20	15	3	13.1667	1256.86 ms
max-mean-div-20.txt	20	20	4	13.1667	1980.76 ms
max-mean-div-20.txt	20	25	5	13.1667	1379.01 ms