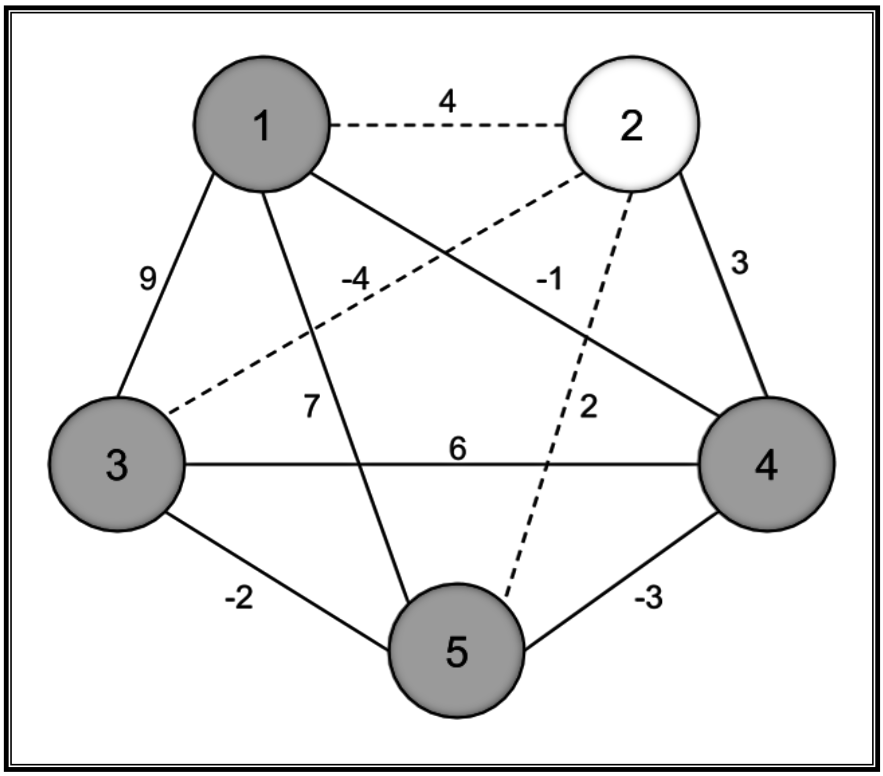
# PRACTICA 8. MAX-MEAN DISPERSION

Diseño y Análisis de Algoritmos




27 DE ABRIL DE 2020

CHRISTIAN TORRES GONZALEZ
Alu0101137902 – Universidad de La Laguna

# Practica 8. Max-mean dispersión

## Índice




27 DE ABRIL DE 2020

CHRISTIAN TORRES GONZALEZ

Alu0101137902 – Universidad de La Laguna

# 1º. Introducción

En esta primera sección del informe, se procederá, a realizar una breve introducción sobre el problema al que nos hemos enfrentado en esta práctica, llamado Max-Mean Dispersión, o Problema de la Dispersión Media Máxima.

Este problema se basa en un grafo formado por una serie de nodos, los cuales están todos conectados, por lo que podemos decir que estamos ante un grafo completo, además de conexo, ya que entre cada par de nodos siempre existe un camino que los conecta, independiente de la distancia a la que se encuentren. De esta forma, cada arista que conecta dos nodos (i, j), su valor de coste o de afinidad o de distancia, va a ser el mismo valor que el de la arista que conecta el nodo (j, i).

El objetivo de este problema es, dada la estructura de grafos comentada en el párrafo anterior, intentar buscar aquel conjunto de nodos que hacen que la dispersión media de ese conjunto de nodos sea máxima. Para saber la dispersión media de cada conjunto de nodos, se aplica la siguiente formula:

$$md(S) = \frac{\sum_{i,j \in S} d(i,j)}{|S|}$$

Para comprender como funciona esta fórmula, debemos comprender los diferentes términos que la componen.

- S: subconjunto de vértices S ∈ V, que maximiza la dispersión media.

**¡MUY IMPORTANTE, EN NUESTRO SUBCONJUNTO S, ¡SE GUARDAN NODOS! ¡NO ARISTAS!**

- **d (i, j)**: coste, distancia o afinidad (se utilizará un término u otro en función del problema con el que estemos trabajando, en nuestro caso usaremos distancia o afinidad) del nodo i, al nodo j.
- **|S|:** valor absoluto de S, es decir, cantidad de nodos que hay en S

Esto que hemos comentado, serían los términos que componen la formula, pero ¿Qué es lo que estamos haciendo en la formula?

- Tal y como se muestra, en la parte del numerador, nos encontramos con el sumatorio de todas las afinidades que hay entre todos los nodos que conforman nuestro subconjunto S.

Por ejemplo, si nuestro subconjunto S es [1 - 2 - 3], la parte del numerador correspondería con la suma de las distancias que unen los nodos **d(1, 2) + d(1, 3) + d(2, 3).**
Hay que recordar que no hace falta sumar **d(2, 1) o d(3, 1)** porque la distancia de esa arista es la misma, tal y como ya habíamos comentado.

- En segundo lugar, en la parte del denominador, tendríamos la cantidad de nodos que conforman nuestro subconjunto S.
- El resultado de esta división es lo que nos daría nuestro valor de dispersión media para ese conjunto de nodos. De esta manera, el objetivo de este problema sería intentar maximizar ese valor calculado, es decir, buscar que conjunto de nodos tengo que incluir en S, para que el valor que me retorna esa división sea el máximo

Esta sería la fórmula que utilizaremos en este problema para calcular la dispersión media, y decidir si ese conjunto de nodos es válido como solución que estamos buscando. La complejidad de todo esto no es aplicar la formula, ya que simplemente es sustituir número y sumar y dividir, sino que, por el contrario, la complejidad reside en saber buscar o en saber que nodo es el que hay que introducir en mi subconjunto para obtener la dispersión media máxima. Por ello, en las siguientes secciones de este informe, se abordarán las diferentes técnicas que se han utilizado para la resolución del problema.

# 2º. Estructura de directorios

En esta sección se explicará la estructura de directorios en la que ha sido almacenado el proyecto:

- **Src**: directorio en el que encontramos los ficheros de implementación de las clases, donde implementamos los métodos que hemos definido para cada clase
- **Include**: directorio donde se encuentran los ficheros de definición de cada clase, aquí solo se encuentra la definición, es decir, los ficheros que contienen que métodos y atributos va a requerir cada clase
- **Grafos**: en este directorio se encuentran almacenados los grafos que se suministran para poder trabajar con nuestro problema
- **Build**: en este fichero, se encuentra el ejecutable del programa, llamado "**algoritmos**", además también se encuentra la herramienta necesaria para la compilación de los archivos del programa y obtener el ejecutable mencionado.

También encontramos otros directorios como pueden ser ".git" o ".vscode" pero simplemente son ficheros que contienen archivos de configuración por lo que no los comentaremos, ya que no son de importancia

**¡Para la ejecución del programa, hay que ejecutar!**

**"./algoritmos ../Grafos/<grafo a utilizar>"**

# 3º. Algoritmos

En esta sección, se abordarán los diferentes algoritmos que se han utilizado para la resolución del problema. Se hará una introducción a cómo funciona el algoritmo, y a continuación, se muestra la implementación por la que hemos optado nosotros. Se indicará, además, mediante una referencia, la sección del informe a la que debemos acudir para visualizar la implementación en código empleada para cada algoritmo, ya que aquí se explicara de manera textual, lo que estamos haciendo, y es en otra sección del informe, donde se comenta la estructura de código utilizada.

## a) Algoritmo Voraz

En primera instancia, el primer algoritmo que debemos tratar es el Algoritmo Voraz.

El funcionamiento de este algoritmo se basa en intentar buscar en cada iteración, o en cada paso de ejecución, el valor optimo, o el valor que maximice nuestra función objetivo, de esta manera siempre estaríamos buscando la opción optima, en cada paso local, aunque puede ocurrir que esta no lleve a una solución óptima al problema.

Aplicado al problema de la dispersión media máxima, este seguiría la misma lógica. Es decir, en cada paso, buscamos cual es el nodo que mejor resultado aporta a nuestra función objetivo, para así añadirlo a nuestra solución.

Esta sería la lógica del algoritmo voraz utilizada para los problemas en un ámbito más generalizado. Procedemos ahora a comentar la lógica que he seguido para la implementación de este algoritmo.

En primer lugar, tendríamos una primera fase llamada preprocesamiento, esta fase podemos decidir si aplicarla o no. En nuestro caso realizamos esta primera fase, y lo que hacemos es buscar la arista cuya distancia es la máxima. También se pueden aplicar otras estrategias para la fase de preprocesamiento tal como puede ser seleccionar una arista aleatoria y partir de los nodos que forman esa arista. Una vez realizada esta primera fase, es cuando aplicamos el algoritmo que comentamos anteriormente, donde vamos a buscar cual es el nodo que maximiza el valor de dispersión, para saber cuál es ese nodo, estamos obligados a recorrer todos los nodos de nuestro grafo. Una vez sabemos cuál es ese nodo, lo introducimos en nuestro vector solución y lo sacamos de nuestro grafo, ya que, si no tenemos un registro de los nodos que hemos ido introduciendo en nuestra solución, se repetiría el nodo y entraríamos en un bucle infinito, ya que siempre nos cogería el mismo nodo.

Procederemos a explicar la lógica utilizada en nuestro programa paso a paso, o lo que es lo mismo, veremos el pseudocódigo utilizado para este algoritmo:

1. **Fase de preprocesamiento**: recorremos nuestro grafo buscando la arista con mayor distancia o afinidad.
2. Una vez obtenida la arista con mayor distancia, añadimos los dos nodos que une esa arista a nuestro vector solución, de esta forma, ya partiríamos de un vector inicial formado por dos nodos
3. En tercer lugar, vamos a necesitar otro vector auxiliar, que utilizaremos para comparar con el vector solución. El objetivo de esto es que, cuando nosotros buscamos el siguiente nodo que maximiza la dispersión media, puede ocurrir que ya no haya ningún nodo que mejore la solución, por lo que nuestro vector no cambiaría.
4. Tras igualar los vectores comentados en el paso anterior, entramos en nuestro bucle de repetición, hasta que se cumpla la siguiente condición: Estaremos iterando en nuestro bucle, hasta que no encontremos un nuevo nodo que mejore la solución, es decir, no saldremos del bucle mientras tengamos nodos que podamos incluir en nuestra solución, ya que estos la mejorarían.
5. Dentro del bucle que hemos iniciado, el primer paso que debemos ejecutar es igualar el vector auxiliar que hemos creado a nuestro **vector_solucion**
6. En segundo lugar, buscamos el nodo que maximiza la dispersión media, que teníamos antes de añadir ese nodo a la solución.
7. Una vez hemos obtenido la dispersión media con un nuevo nodo, comprobamos si esa nueva dispersión, es mejor que la dispersión que teníamos antes de añadir ese posible nodo a la solución. En caso de comprobar que es mejor, añadiríamos ese nodo a nuestra solución, y volveríamos a iterar sobre el bucle. Por el contrario, si no se cumple que esa dispersión es mejor que la anterior, no modificaríamos nuestro bucle, por lo que terminaríamos el algoritmo, ya que significa que ya no hay nodos posibles que me mejoren la solución.

El pseudocodigo utilizado para la implementación de este algoritmo ha sido el propuesto en el enunciado:

**Algoritmo constructivo voraz**

```
1: Seleccionar la arista (i, j) con mayor afinidad;
2: S = {i, j};
3: repeat
4:     S* = S;
5:     Obtener el vértice k que maximiza md(S ∪ {k});
6:     if md(S ∪ {k}) ≥ md(S) then
7:         S = S ∪ {k};
8: until (S* = S)
9: Devolver S*;
```

Tal y como se aprecia en la imagen, el código mostrado es el mismo descrito que en los pasos que se encuentran justo antes de la foto. Creamos nuestra solución inicial (fase de preprocesamiento). Mientras haya nodos que puedan mejorar nuestra solución, deberemos iterar en nuestro bucle hasta que no se modifique nuestra solución, significando que ya no hay ningún nodo más, que al añadirlo a nuestro vector solución, mejore la dispersión media hasta el momento.

En secciones posteriores de este informe, se encuentra el código implementado para la realización de este primer algoritmo (Mirar sección "**Implementación Algoritmo Voraz**").

Además, también, en secciones posteriores, se muestran una serie de tablas comparativas a los resultados obtenidos tras la ejecución de este algoritmo con los diferentes grafos propuestos en el enunciado.

## b) Algoritmo Nuevo

En este segundo ejercicio, se pide desarrollar un nuevo algoritmo voraz diseñado por el usuario, es decir de libre elección.

Para la implementación de este segundo algoritmo he decidido optar por una implementación similar a la del algoritmo voraz desarrollado para el apartado uno, lo que ahora cambiáramos nuestra función objetivo. Es decir, en vez de buscar el nodo con mayor dispersión media de los nodos candidatos que todavía no han sido introducidos en la solución, buscaremos el primer nodo que al añadirlo a nuestro vector solución, mejore la dispersión media que teníamos. La principal diferencia en esta segunda opción es que ahora en vez de buscar el máximo, buscaremos el primero que cumpla la condición de mejorar la dispersión media.

Es por este motivo, que la lógica seguida para este algoritmo es similar a la del algoritmo anterior pero solo cambiando nuestra condición de parada.

El pseudocodigo utilizado para este segundo algoritmo es prácticamente similar al del algoritmo voraz.

# c) Algoritmo Grasp

El Algoritmo Grasp, es un algoritmo que pertenece a la familia de los algoritmos Greedy. El nombre de Algoritmo Grasp viene de Greedy Randomized Adaptive Search Procedure.

Este algoritmo es una adaptación del algoritmo voraz(Greedy) que, a diferencia de él, cada una de las soluciones que proporciona este algoritmo, es construida de manera totalmente aleatoria.

El objetivo de introducir la aleatoriedad en este algoritmo es el de intentar reducir los tiempos de cómputo y de resolución del problema. Además, este tipo de algoritmos sigue una estructura esquematizada para la resolución de los problemas, independiente del ámbito al que pertenezca el problema. Esta estructura consta de cuatro fases:

- Preprocesamiento: fase inicial donde se inicializa nuestro vector de partida.
- Fase constructiva: fase en la que se selecciona el nodo candidato a formar parte de nuestro vector de soluciones.
- Postprocesamiento: fase en la que se comprueba si la solución obtenida en la fase constructiva es viable, es decir, cumple la función objetivo.
- Fase de actualización: fase en la que, si se cumple la fase de Postprocesamiento, se procede a actualizar nuestra solución, añadiendo el nuevo nodo candidato.

Para la aplicación de este algoritmo a nuestro problema se sigue con exactitud la lógica comentada anteriormente, es decir, inicializamos nuestro vector, seleccionamos un candidato (de manera totalmente aleatoria) a solución, comprobamos ese candidato a solución, actualizamos la solución.

Para la realización de este tercer algoritmo nos hemos basado en el pseudocodigo propuesto en las transparencias de clase, el cual podemos encontrar en la siguiente imagen.

```
Procedure GRASP
Begin
 Preprocesamiento
  Repeat
    Fase Constructiva(Solución);
    PostProcesamiento(Solución);
    Actualizar(Solución, MejorSolución);
  Until (Criterio de parada);
End.
```

Tal y como se aprecia en la imagen del pseudocodigo utilizado, tendríamos:

1. Fase de preprocesamiento: en esta fase inicializamos nuestro vector_inicial siguiendo la técnica utilizada para los algoritmos anteriores, donde buscamos la arista con mayor distancia, y es esa la arista de donde vamos a sacar los nodos de los que partiremos.
2. Fase constructiva: en esta fase obtendremos el nodo candidato a introducir en nuestra solución de manera aleatoria. Para la obtención de este nodo, utilizaremos lo que se denomina, una Lista Restringida de Candidatos, esta lista consiste en un vector auxiliar de un tamaño especificado por el usuario, donde iremos almacenando los nodos con mejor valor de dispersión media entre sí. Una vez hemos recorrido el vector de nodos restante a posibles candidatos, y hemos completado nuestra LRC, procedemos a seleccionar un nodo de los contenidos en nuestra LRC, de manera aleatoria, y será ese nodo el que utilizaremos como candidato a entrar en nuestra solución.
3. Postprocesamiento: en esta fase, procedemos a comprobar si el nodo obtenido en la fase anterior cumple los requisitos de la función objetivo para formar parte de nuestra solución.
4. Fase de actualización: se ejecuta cuando la fase de Postprocesamiento se lleva a cabo de manera satisfactoria, actualizando nuestra solución, añadiendo el nuevo nodo candidato a nuestra solución.

A diferencia de los algoritmos desarrollados anteriormente, aquí aparece una nueva restricción, y es nuestra condición de parada. Ahora, ya no pararemos cuando no se haya modificado nuestra solución, al igual que hacíamos en los algoritmos anteriores. Los criterios de parada en este algoritmo son dos:

- Numero de iteraciones: el algoritmo se estará ejecutando un numero de iteraciones introducido por el usuario.
- Numero de iteraciones sin mejora: el algoritmo se estará ejecutando durante un numero de iteraciones introducida por el usuario, pero ahora, solo aumentaremos las iteraciones, cuando el nodo seleccionado aleatoriamente, no mejora la dispersión media obtenida hasta el momento.

# d) Algoritmo Multiarranque

El Algoritmo Multiarranque, se caracteriza, por su forma de inicializar el vector de soluciones, de ahí su nombre Multiarranque, ya que tiene diversas formas de inicializar el vector de soluciones.

Dos formas validas que se podrían utilizar como mecanismo de arranque, serian:

- Selección de dos nodos totalmente aleatorios y ser esos dos nodos los que incluimos en nuestra solución inicial.
- Partir de la solución inicial propuesta por Grasp y cambiar el método de búsqueda local, es decir cambiar la forma en la que se busca el siguiente nodo candidato a formar parte de nuestra solución.

Estas dos formas de arranque comentadas son las dos implementaciones por las que puede optar para ejecutar nuestro algoritmo Multiarranque. En nuestro caso realizaremos la implementación de ambas maneras, es decir seleccionaremos dos nodos aleatorios y calcularemos nuestra solución usando Grasp, y partiremos de la solución inicial de la que parte Grasp y de ahí calcularemos las búsquedas locales.

De esta manera, tenemos dos formas de ver la ejecución de un algoritmo de búsqueda Multiarranque:

- **Método 1:**
  1º. Aplicamos preprocesamiento aleatorio
  2º. Aplicamos Grasp usando ese nuevo preprocesamiento aleatorio

- **Método 2:**
  1º. Aplicamos preprocesamiento de Grasp.
  2º. Aplicamos búsqueda local diferente.

Como bien ya dije, en la sección de "**Implementación Algoritmo Multiarranque",** desarrollaremos ambas opciones.

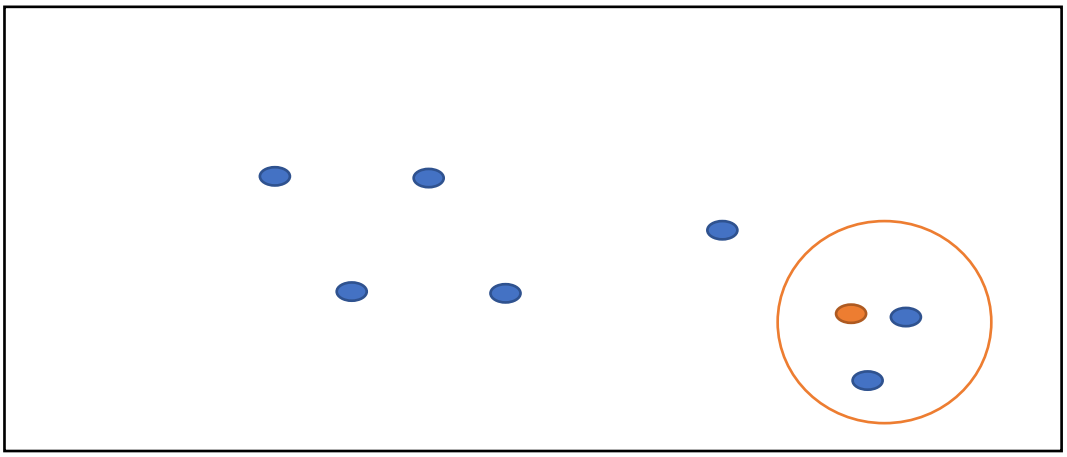
Comentar, que la única innovación en este nuevo algoritmo es en el preprocesamiento del **método 1**, donde seleccionaremos dos nodos de manera totalmente aleatoria y será, de esos dos nodos de donde partamos a buscar nuestra solución. Y la segunda innovación viene en el **método 2**, en la búsqueda local, ya que nos tenemos que inventar una nueva búsqueda del nodo candidato a formar parte de la solución.

En mi caso, para esa nueva búsqueda local, seguiremos la misma estrategia que utiliza el algoritmo Grasp, es decir, utilizaremos una posible lista de candidatos, pero en vez de almacenar los nodos con mejor dispersión media entre sí, aláncenos los nodos que tienen mejor dispersión media que la actual solución, y es de esos nodos de donde voy a obtener el nuevo posible candidato

## e) Algoritmo VNS

El Algoritmo VNS, se caracteriza, por su forma buscar soluciones, esto quiere decir que este algoritmo, parte de una solución encontrada de alguna manera, y a partir de ahí, intenta buscar otras posibles soluciones que puedan mejorar la solución que ya tenemos.

En nuestro caso, la aplicación que vamos a seguir para este algoritmo consiste en, partiremos de una solución inicial que nos proporciona el algoritmo Grasp, y es a partir de ahí, donde iremos buscando nuevas soluciones. Esto que denominamos como ir buscando nuevas soluciones, es lo que se llama "**Estructura de entorno"**, para entender esto de que es la estructura de entorno, nos basaremos en el siguiente ejemplo:



En este cuadro, partimos de una solución propuesta por Grasp (circulo naranja), y es a partir de esa solución de la que vamos a ir expandiendo en busca de otras soluciones que nos puedan mejorar la solución que ya tenemos. En cuanto a la forma de expandir en busca de nuevas soluciones, iteraremos un máximo de tres veces, es decir, seleccionaremos de manera aleatoria un nodo que no forme parte de nuestra solución, y lo cambiaremos por uno que si se encuentra dentro de nuestra solución. Evaluaremos nuestra función objetivo, y comprobaremos si esa nueva solución es mejor que la anterior y actualizaremos. Estaremos ejecutando estas búsquedas durante un numero de iteraciones determinado.

# 4º. Estructura de Clases

## - Carga y lectura del Grafo

Para la carga de nuestros grafos introducidos por consola, he diseñado una clase “Gestora_”, encargada de la lectura del fichero suministrado.

A medida que esta clase va leyendo los datos de nuestro fichero, se encarga de ir creando las aristas correspondientes a ese grafo y las va almacenando en un vector auxiliar. Una vez se han leído todos los datos de nuestro archivo, lo que quiere decir que ya se han creado todas las aristas que conforman nuestro grafo, realizo un paso adicional, que es el de ordenar esas aristas en función del nodo al que pertenecen.

Es decir, ordeno todas las aristas en función del orden de nodos, primero todas las aristas del nodo 1, luego todas las del 2, … Y dentro de cada nodo las ordeno también.

Este proceso que puede parecer laborioso, a la hora de crear el grafo, me ahorra bastante tiempo de computo ya que simplemente es almacenar las aristas en su correspondiente nodo.

```
18  #pragma once
19
20  #include <iostream>
21  #include <vector>
22  #include <fstream>
23  #include <string>
24
25  #include "arista.h"
26  #include "grafo.h"
27
28  using namespace std;
29
30  class Gestor_Archivos_
31  {
32      private:
33          ifstream archivo1;
34          ofstream archivo2;
35          vector<Arista_> vector_aristas;
36
37      public:
38          Gestor_Archivos_(string ,string ,Grafo_ &);
39
40          void cargar_datos(Grafo_ &);
41          void ordenar_aristas(int );
42          void mostrar_aristas(void);
43  };
```

En la imagen anterior, se aprecian todas las funciones y atributos que he descrito en la introducción de esta clase nueva que he creado. En cuanto a los atributos,

- Archivo1: archivo donde que contiene el fichero de entrada
- Archivo2: archivo de salida
- Vector<Aristas_>: vector donde se almacenan todas las aristas que se van creando a medida que se lee el archivo.

En cuanto a los métodos empleados, simplemente hay que destacar, el método cargar datos() se encarga de leer los datos del archivo y crear las aristas correspondientes. El segundo método, el método ordenar aristas(), es el método que comente que se encarga de ordenar las aristas en función de su nodo. Y finalmente, tenemos el método que se encarga de imprimir las aristas que se han leído a modo de comprobación.

## - Representación del Grafo

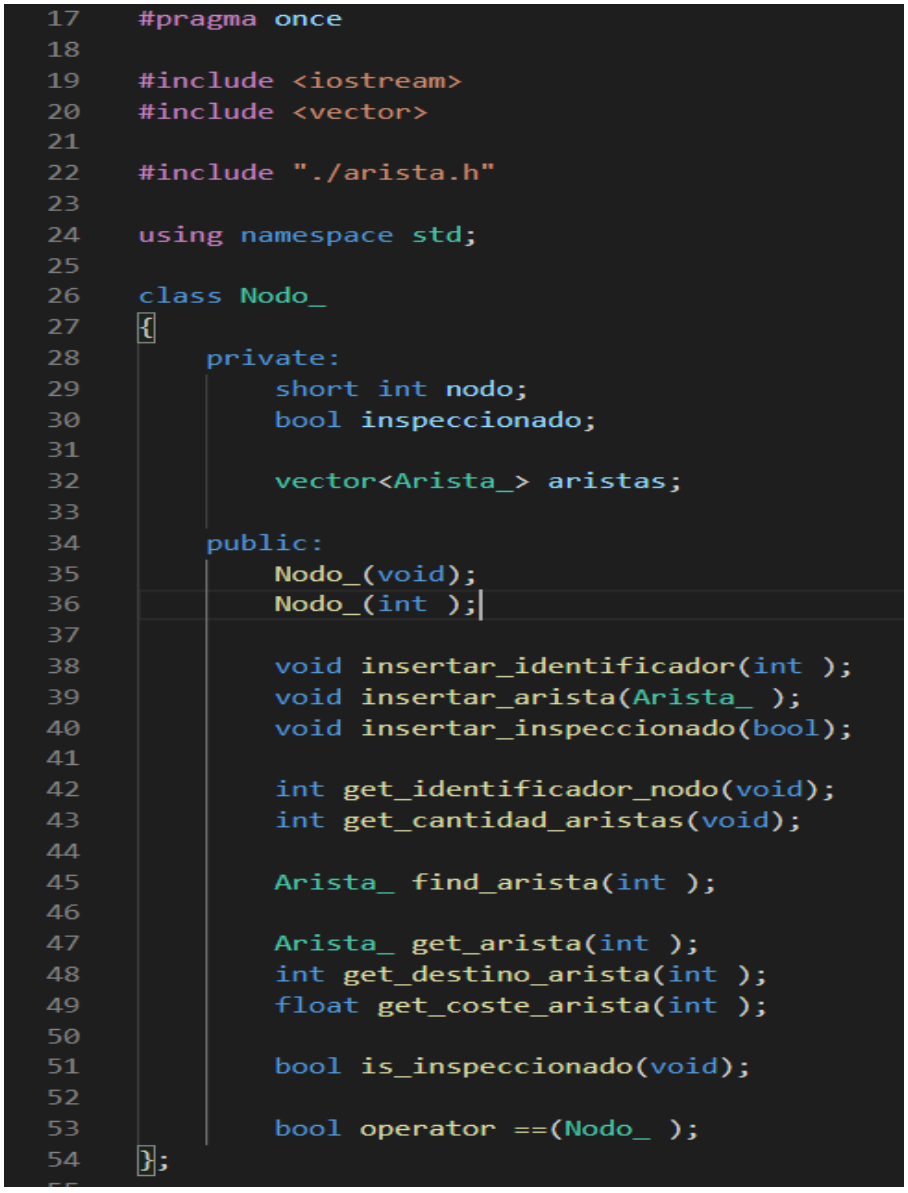
Para la representación del grafo con el que estamos trabajando, he decidido crear una clase grafo, que se encarga de almacenar los diferentes nodos que formar nuestro grafo.

```
17  #pragma once
18
19  #include <iostream>
20  #include <vector>
21  #include <fstream>
22
23  #include "nodo.h"
24  #include "arista.h"
25
26  using namespace std;
27
28  class Grafo_
29  {
30      public:
31          int numero_nodos;
32          vector<Nodo_> vector_nodos;
33          vector<Arista_> vector_aristas;
34
35      public:
36          Grafo_(void);
37          Grafo_(int , vector<Arista_> );
38
39          void construir_nodos(void);
40
41          void set_aristas(vector<Arista_> );
42          void insertar_nodo(Nodo_ );
43
44          int get_numero_nodos(void);
45          Nodo_ get_nodo(int );
46
47          void imprimir_grafo(void);
48          void imprimir_costes(void);
49
50          void eliminar_nodo(int );
51  };
52
```

En cuanto a la clase grafo, simplemente hay que destacar que el atributo más importante es el vector nodos, ya que es donde almacenamos todos los nodos que conforman nuestro grafo, junto a la información que cada nodo contiene. Es el único atributo con el que trabajaran los algoritmos. En cuanto a los métodos implementados, no hay que destacar nada, ya que son simplemente setter y gentes.

## - Representación de los nodos

Para la representación de los nodos, he desarrollado la clase Nodo, que se encarga de almacenar la información referente a un nodo, tal y como podemos comprobar en la imagen.



```
17  #pragma once
18
19  #include <iostream>
20  #include <vector>
21
22  #include "./arista.h"
23
24  using namespace std;
25
26  class Nodo_
27  {
28      private:
29          short int nodo;
30          bool inspeccionado;
31
32          vector<Arista_> aristas;
33
34      public:
35          Nodo_(void);
36          Nodo_(int );
37
38          void insertar_identificador(int );
39          void insertar_arista(Arista_ );
40          void insertar_inspeccionado(bool);
41
42          int get_identificador_nodo(void);
43          int get_cantidad_aristas(void);
44
45          Arista_ find_arista(int );
46
47          Arista_ get_arista(int );
48          int get_destino_arista(int );
49          float get_coste_arista(int );
50
51          bool is_inspeccionado(void);
52
53          bool operator ==(Nodo_ );
54  };
```

En cuanto a los atributos que encontramos en esta clase nodo, simplemente destacar nuestro vector de aristas, ya que es el más importante junto al atributo nodo, el primero es el vector donde almacenamos todas las aristas que salen de nuestro nodo en cuestión y el segundo simplemente es el identificador del nodo con el que estamos trabajando. En cuanto a los métodos implementados, ocurre igual que con la clase grafo, no hay nada que destacar, ya que simplemente son setter y getters.

## - Representación de las aristas

Para la representación de las aristas que conforman nuestro grafo, se ha desarrollado la clase arista, que simplemente almacena la información de una arista, es decir, el nodo del que parte esa arista, el nodo al que llega esa arista y la distancia de esa arista.

```
17  #pragma once
18
19  #include <iostream>
20  #include <vector>
21  #include <string>
22
23  using namespace std;
24
25  class Arista_
26  {
27      public:
28          int nodo_inicial;
29          int nodo_final;
30          float coste;
31
32      public:
33          Arista_(void);
34          Arista_(int , int , float );
35
36          int get_nodo_inicial(void);
37          int get_nodo_destino(void);
38          float get_coste_arista(void);
39
40          void set_nodo_inicial(int );
41          void set_nodo_final(int );
42          void set_coste(float );
43
44          bool operator ==(Arista_ );
45
46          void imprimir_arista(void);
47  };
48
```

En cuanto a los atributos utilizados en esta clase, tal y como ya comenté, almacenamos el nodo del que parte la arista, almacenamos el nodo al que llega la arista y almacenamos el coste o distancia de esa arista.

En cuanto a los métodos de esta clase, no hay que destacar ninguno, ya que al igual que ocurría en las clases anteriores, son simplemente setter y getters

# 4º. Implementación Algoritmo Voraz

Tal y como se comentó en la sección donde describimos el funcionamiento del Algoritmo Voraz y del pseudocodigo implementado, en este apartado, procederemos a comentar el código que se ha desarrollado para la implementación del pseudocodigo descrito.

### a) Fichero algoritmo_voraz.h

En primer lugar, veamos el fichero .h que se encarga de contener la definición de los atributos y métodos que utilizaremos en la clase.

include > algoritmo_voraz.h > Algoritmo_Voraz_ > vector_inicial

```
1   /**
2    * @copyright Universidad de La Laguna
3    * @copyright Escuela Superior de Ingeniería y Tecnología
4    * @copyright Grado en Ingeniería Informática
5    * @copyright Asignatura: Diseño y Analisis de Algoritmos (DAA)
6    * @copyright Curso: 3º Itinerario 1
7    * @copyright Práctica 8. Algoritmos
8    * @author: Christian Torres Gonzalez alu0101137902@ull.edu.es
9    * @description: Fichero de la clase Algoritmo_Voraz_, encargada de
10   * implementar el codigo que resuelve el problema de la maxima dispersion
11   * media utilizando la tecnica de Algoritmo Voraz
12   * @since 23/04/2020
13   * @file Fichero de implementacion de la clase Algoritmo_Voraz_
14   * @version 1.0.0
15   * @see https://github.com/ChristianTorresGonzalez/DAA_P8.git
16  */
17
18  #pragma once
19
20  #include <iostream>
21  #include <vector>
22
23  #include "./algoritmos.h"
24
25  class Algoritmo_Voraz_ : public Algoritmos_
26  {
27      private:
28          vector<Nodo_> vector_inicial;
29
30      public:
31          Algoritmo_Voraz_(Grafo_ );
32
33          void resolver_algoritmo(void);
34
35          float calcular_dispersion_maxima();
36  };
```

Empecemos hablando por los atributos que hemos definido para la clase:

- **Vector_inicial**: es el vector en el que iremos almacenado nuestra solución relativa. Es decir, en el primer paso descrito en pseudocodigo, es en este vector donde almacenamos el resultado de nuestra fase de preprocesamiento, y donde iremos añadiendo los nodos que nos van mejorando nuestra solución.
- **Vector_solución**: es el vector donde almacenaremos nuestra solución final. Si nos fijamos en nuestro pseudocódigo, este vector es el que se utiliza para comparar la condición de nuestro while. Al final de la ejecución, este vector se iguala a vector_inicial, para que vector_solucion contenga nuestra solución al problema.

En cuanto a los métodos definidos para esta clase, a parte de los métodos de los que ya disponemos en la clase padre, tenemos:

- **Algoritmo_Voraz (Grafo_ )**: constructor de la clase. Simplemente lo utilizamos para inicializar ambos vectores a un tamaño de 0, y, además, para almacenar el grafo que vamos a utilizar para trabajar.
- **void resolver_algoritmo(void)**: función a la que llamaremos desde nuestro archivo "main"(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo. Es decir, para buscar aquellos nodos, que cumplen nuestra función objetivo.
- **float calcular_dispersion_maxima(void):** este método es llamado dentro de nuestra función **resolver_algoritmo(),** y va a ser el encargado de buscar el nodo con valor de dispersión media máxima.

### b) Fichero algoritmo_voraz.cc

Una vez comentado, todos los atributos y funciones de los que disponemos en esta clase, pasemos a comentar la implementación de cada una de esas funciones. Recordar, que además de las funciones definidas en la propia clase, también disponemos de las funciones definidas en la clase Padre, lo que como esas funciones son comunes al resto de algoritmos, las hemos definido ahí, ya que son de uso común.

Ya que el constructor de nuestra clase simplemente se encarga de almacenar el grafo introducido, lo veremos reflejado en la imagen siguiente, pero no comentaremos nada.

```
Algoritmo_Voraz_::Algoritmo_Voraz_(Grafo_ grafo):
    Algoritmos_(grafo)
    {}
```

Ahora sí, procedamos a comentar la función **resolver_algoritmo().**

```
void Algoritmo_Voraz_::resolver_algoritmo(void)
{
    cronometro.start();
    Arista_ arista = calcular_arista_maxima();

    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);

    dispersion_media = calcular_dispersion_media(vector_inicial);

    while(comparar_vectores(vector_inicial, vector_solucion) == false)
    {
        vector_solucion = vector_inicial;
        float nueva_dispersion = calcular_dispersion_maxima();

        if (nueva_dispersion >= dispersion_media)
        {
            dispersion_media = nueva_dispersion;
            grafo.eliminar_nodo(vector_inicial[vector_inicial.size() - 1].get_identificador_nodo());
        }
        else
        {
            vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());
        }
    }

    cronometro.end();
}
```

Esta primera sentencia que vemos, que es **cronometro.start(),** es la línea que se encarga de iniciar el cronometro que utilizamos para contabilizar el tiempo de ejecución del algoritmo. También coincide con la última línea de esta misma función, la única diferencia, es que ahora paramos el cronometro, ya que el algoritmo ha terminado.

En primer lugar, tal y como habíamos dicho en la sección “Algoritmo Voraz” de este informe, indicábamos que lo primero que íbamos a hacer, era una fase inicial de preprocesamiento, en donde íbamos a buscar la arista con mayor distancia y almacenar los nodos correspondientes a esa arista. Ese procedimiento descrito es el que hacemos en esta primera sección de la función. Una vez obtenemos la arista con mayor distancia, almacenamos los nodos que une esa arista, y además procedemos a eliminarlos de nuestro grafo, para que no se

produzcan bucles infinitos. Recordar, tal y como definimos en la clase **Algoritmos_**, dijimos que almacenábamos el grafo ya que nos sería más fácil, puesto que esa implementación nos permite eliminar nodos del grafo sin que afecte al resto del problema.

```
Arista_ arista = calcular_arista_maxima();

vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));
grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);
```

En segundo lugar, una vez tenemos nuestro vector de partida inicializado, deberemos calcular la dispersión media del vector con esos nodos, ya que será la que utilizaremos posteriormente para comprobar si la nueva dispersión tras añadir otro nodo es mejor que la dispersión media sin añadirlo.

```
dispersion_media = calcular_dispersion_media(vector_inicial);
```

(La implementación de esta función, se describe en la sección de" Algoritmos_")

Como tercer y último paso, para finalizar con nuestro pseudocodigo, solo quedaría comentar que es lo que hacemos en nuestro bucle while():

```
while(comparar_vectores(vector_inicial, vector_solucion) == false)
{
    vector_solucion = vector_inicial;
    float nueva_dispersion = calcular_dispersion_maxima();

    if (nueva_dispersion >= dispersion_media)
    {
        dispersion_media = nueva_dispersion;
        grafo.eliminar_nodo(vector_inicial[vector_inicial.size() - 1].get_identificador_nodo());
    }
    else
    {
        vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());
    }
}
```

En primer lugar, igualamos nuestra solución a nuestro vector actual, esta asignación es necesaria, ya que es nuestra condición de parada de nuestro bucle. Este se debe a que solo pararemos cuando nuestro vector inicial no haya añadido ningún nodo nuevo, por lo que cuando comparamos nuestra condición en el while() dado que ambos vectores son iguales, la función **comparar_vectores()** retorna **true**, y saldríamos del bucle.

¿Pero qué hacemos dentro del bucle? Una vez hemos igualado ambos vectores, llamamos a la función **calcular_dispersion_maxima()**, que calcula para cada nodo restante, añadiéndolo al **vector_inicial**, su dispersión media y nos retorna el valor de dispersión máxima. Aquí podemos ver una de las grandes ventajas que nos proporciona el poder almacenar nuestro grafo e ir eliminando nodos a medida que los vamos incluyendo en nuestra solución, ya que cuando vamos a buscar los nodos restantes, ese conjunto de nodos restante no tiene nodos repetidos, por lo que no tenemos que hacer comprobaciones adicionales para ver si el nodo que queremos utilizar ya está introducido en nuestra solución.

Una vez finaliza la función en cuestión, y guardamos el valor de dispersión devuelto, solo quedaría comprobar, si añadimos el nuevo nodo, la dispersión mejora, o por el contrario no mejora, En caso de que mejore, en primer lugar, tendremos que actualizar la nueva dispersión media y eliminamos el nodo de nuestro grafo, para que no se repita. En el caso contrario de que la dispersión no mejore, simplemente tendríamos que quitar ese nodo candidato de nuestro vector solución.

Para terminar con este segundo fichero nos quedaría comentar el funcionamiento de esa función **calcular_dispersion_maxima()**. En esta función, una vez es llamada, lo primero que hacemos es declarar variables locales que necesitaremos para el cálculo del nodo que nos proporciona la mayor dispersión media. Una de estas variables es el nodo que tiene mayor dispersión media.

Tras esta declaración inicial, procedemos a introducir cada nodo de los que todavía no hemos introducido en nuestra solución, en nuestro vector_inicial, y calcular la dispersión media resultante. Una vez calculada esa dispersión media, solamente tenemos que comprobar si se cumple que esa dispersión es mejor que la dispersión calculada con el nodo anterior.

Estaremos repitiendo este bucle mientras queden nodos que inspeccionar en nuestro grafo. Si se cumple, simplemente tenemos que actualizar ese nodo, al **nodo_maximo**, que es el que nos proporciona la dispersión media máxima de los posibles candidatos. Finalmente introducimos el mejor nodo como posible candidato a nuestra solución final y retornamos la dispersión media obtenida con ese nuevo nodo.

En nuestra función principal, en caso de que esa dispersión retornada, no sea mejor que la que ya teníamos, simplemente tenemos que eliminar de nuestro vector_solucion ese nodo candidato que habíamos introducido.

```
float Algoritmo_Voraz_::calcular_dispersion_maxima()
{
    float dispersion = 0;
    Nodo_ nodo_maximo;

    for (int i = 0; i < grafo.vector_nodos.size(); i++)
    {
        vector_inicial.push_back(grafo.get_nodo(i));
        float nueva_dispersion = calcular_dispersion_media(vector_inicial);

        if (nueva_dispersion >= dispersion)
        {
            dispersion = nueva_dispersion;
            nodo_maximo = grafo.get_nodo(i);
        }

        vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());
    }

    vector_inicial.push_back(nodo_maximo);

    return dispersion;
}
```

En esta imagen, podemos ver reflejado el código utilizado en nuestra la función.

# 5º. Implementación Algoritmo Nuevo

Para este segundo algoritmo, como ya se ha comentado en la sección de definición del algoritmo, vamos a optar por una implementación similar a la del algoritmo del apartado 1 del enunciado, pero cambiado la forma en la que se elige nuestro nodo candidato. Como ya hemos comentado, ahora no buscaremos el nodo que proporcione mayor dispersión media, sino que buscaremos el primer nodo que mejore la dispersión media, sin que tenga que ser el nodo máximo.

Tras comentar esto, al igual que hicimos en el apartado anterior, comentaremos los archivos de implementación de la clase.

## a) Fichero algoritmo_nuevo.h

Como es evidente, en este archivo, se encuentra la definición de los atributos y métodos utilizados para el desarrollo de este algoritmo.

include > C algoritmo_nuevo.h > ...

```
/**
 * @copyright Universidad de La Laguna
 * @copyright Escuela Superior de Ingeniería y Tecnología
 * @copyright Grado en Ingeniería Informática
 * @copyright Asignatura: Diseño y Analisis de Algoritmos (DAA)
 * @copyright Curso: 3º Itinerario 1
 * @copyright Práctica 8. Algoritmos
 * @author: Christian Torres Gonzalez alu0101137902@ull.edu.es
 * @description: Fichero de la clase Algoritmo_Nuevo_, encargada de
 * implementar el codigo que resuelve el problema de la maxima dispersion
 * media utilizando la tecnica de Algoritmo Nuevo implementado por el alumno
 * @since 23/04/2020
 * @file Fichero de implementacion de la clase Algoritmo_Voraz_
 * @version 1.0.0
 * @see https://github.com/ChristianTorresGonzalez/DAA_P8.git
*/

#pragma once

#include <iostream>
#include <vector>

#include "./algoritmos.h"

class Algoritmo_Nuevo_ : public Algoritmos_
{
    private:
        vector<Nodo_> vector_inicial;

    public:
        Algoritmo_Nuevo_(Grafo_ );

        void resolver_algoritmo(void);

        float calcular_dispersion_maxima(float );
};
```

Empecemos comentando los atributos y métodos que encontramos en esta clase. En cuanto a los atributos, encontramos:

- **Vector_inicial:** es el vector en el que iremos almacenado nuestra solución relativa. Es decir, en el primer paso descrito en pseudocodigo, es en este vector donde almacenamos el resultado de nuestra fase de preprocesamiento, y donde iremos añadiendo los nodos que nos van mejorando nuestra solución.

En cuanto a los métodos definidos para esta clase, a parte de los métodos de los que ya disponemos en la clase padre, tenemos:

- **Algoritmo_Nuevo_ (Grafo_ ):** constructor de la clase. Simplemente lo utilizamos para inicializar ambos vectores a un tamaño de 0, y, además, para almacenar el grafo que vamos a utilizar para trabajar.
- **void resolver_algoritmo(void):** función a la que llamaremos desde nuestro archivo "main"(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo. Es decir, para buscar aquellos nodos, que cumplen nuestra función objetivo.
- **float calcular_dispersion_maxima(float):** este método es llamado dentro de nuestra función **resolver_algoritmo(),** y va a ser el encargado de buscar el primer nodo con valor de dispersión media mejor que la media que ya tenemos.

Como bien se introdujo en le explicación del algoritmo, la lógica de este algoritmo es prácticamente similar a la del algoritmo voraz, cambiando solo la función objetivo, es por esto por lo que la estructura de la clase es prácticamente similar a ambos algoritmos.

### b) Fichero algoritmo_nuevo.cc

Una vez comentado, todos los atributos y funciones de los que disponemos en esta clase, pasemos a comentar la implementación de cada una de esas funciones. Recordar, que además de las funciones definidas en la propia clase, también disponemos de las funciones definidas en la clase Padre, lo que como esas funciones son comunes al resto de algoritmos, las hemos definido ahí, ya que son de uso común.

Ya que el constructor de nuestra clase simplemente se encarga de almacenar el grafo introducido, lo veremos reflejado en la imagen siguiente, pero no comentaremos nada.

```
Algoritmo_Nuevo_::Algoritmo_Nuevo_(Grafo_ grafo):
    Algoritmos_(grafo)
    {}
```

Ahora sí, procedamos a comentar la función **resolver_algoritmo().**

```
void Algoritmo_Nuevo_::resolver_algoritmo(void)
{
    cronometro.start();
    Arista_ arista = calcular_arista_maxima();

    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);

    dispersion_media = calcular_dispersion_media(vector_inicial);

    while(comparar_vectores(vector_inicial, vector_solucion) == false)
    {
        vector_solucion = vector_inicial;
        float nueva_dispersion = calcular_dispersion_maxima(dispersion_media);

        if (nueva_dispersion >= dispersion_media)
        {
            dispersion_media = nueva_dispersion;
            grafo.eliminar_nodo(vector_inicial[vector_inicial.size() - 1].get_identificador_nodo());
        }
    }

    cronometro.end();
}
```

Esta primera sentencia que vemos, que es **cronometro.start(),** es la línea que se encarga de iniciar el cronometro que utilizamos para contabilizar el tiempo de ejecución del algoritmo. También coincide con la última línea de esta misma función, la única diferencia, es que ahora paramos el cronometro, ya que el algoritmo ha terminado.

En primer lugar, como ya hemos comentado, estamos siguiendo la misma estrategia que el algoritmo voraz, por lo que, al igual que ocurre en el otro algoritmo, indicábamos que lo primero que íbamos a hacer, era una fase inicial de preprocesamiento, en donde íbamos a buscar la arista con mayor distancia y almacenar los nodos correspondientes a esa arista. Ese procedimiento descrito es el que hacemos en esta primera sección de la función. Una vez obtenemos la arista con mayor distancia, almacenamos los nodos que une esa arista, y además procedemos a eliminarlos de nuestro grafo, para que no se produzcan bucles infinitos. Recordar, tal y como definimos en la clase **Algoritmos_**, dijimos que almacenábamos el grafo ya que nos sería más fácil, puesto que esa implementación nos permite eliminar nodos del grafo sin que afecte al resto del problema.

```
Arista_ arista = calcular_arista_maxima();

vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));
grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);
```

En segundo lugar, una vez tenemos nuestro vector de partida inicializado, deberemos calcular la dispersión media del vector con esos nodos, ya que será la que utilizaremos posteriormente para comprobar si la nueva dispersión tras añadir otro nodo es mejor que la dispersión media sin añadirlo.

```
dispersion_media = calcular_dispersion_media(vector_inicial);
```

(La implementación de esta función, se describe en la sección de" Algoritmos_")

Como tercer y último paso, para finalizar con nuestro pseudocodigo, solo quedaría comentar que es lo que hacemos en nuestro bucle while():

```
while(comparar_vectores(vector_inicial, vector_solucion) == false)
{
    vector_solucion = vector_inicial;
    float nueva_dispersion = calcular_dispersion_maxima(dispersion_media);

    if (nueva_dispersion >= dispersion_media)
    {
        dispersion_media = nueva_dispersion;
        grafo.eliminar_nodo(vector_inicial[vector_inicial.size() - 1].get_identificador_nodo());
    }
}
```

En primer lugar, igualamos nuestra solución a nuestro vector actual, esta asignación es necesaria, ya que es nuestra condición de parada de nuestro bucle. Este se debe a que solo pararemos cuando nuestro vector inicial no haya añadido ningún nodo nuevo, por lo que cuando comparamos nuestra condición en el while() dado que ambos vectores son iguales, la función **comparar_vectores()** retorna **true**, y saldríamos del bucle.

¿Pero qué hacemos dentro del bucle? Una vez hemos igualado ambos vectores, llamamos a la función **calcular_dispersion_maxima(float),** que se encarga de buscar cual es el primer nodo, de los que todavía no ha sido añadido a nuestra solución, que mejore la dispersión media hasta el momento. En este algoritmo, ya no recorremos todo el vector de nodos restantes en busca el nodo máximo, sino que ahora iteramos hasta que encontremos el primero que mejora nuestra dispersión media hasta el momento. Una vez se nos devuelve el valor, simplemente tenemos que comprobar si se cumple la condición de que esa dispersión media es mejor, y almacenamos el nodo en nuestro vector de solución, además de eliminarlo de nuestro grafo, donde se encuentran el resto de los nodos

que todavía no han sido almacenados. Aquí podemos ver una de las grandes ventajas que nos proporciona el poder almacenar nuestro grafo e ir eliminando nodos a medida que los vamos incluyendo en nuestra solución, ya que cuando vamos a buscar los nodos restantes, ese conjunto de nodos restante no tiene nodos repetidos, por lo que no tenemos que hacer comprobaciones adicionales para ver si el nodo que queremos utilizar ya está introducido en nuestra solución.

Una vez finaliza la función en cuestión, y guardamos el valor de dispersión devuelto, solo quedaría comprobar, si añadimos el nuevo nodo, la dispersión mejora, o por el contrario no mejora, En caso de que mejore, en primer lugar, tendremos que actualizar la nueva dispersión media y eliminamos el nodo de nuestro grafo, para que no se repita. En el caso contrario de que la dispersión no mejore, simplemente tendríamos que quitar ese nodo candidato de nuestro vector solución.

Para terminar con este segundo fichero nos quedaría comentar el funcionamiento de esa función **calcular_dispersion_maxima(float).** En esta función, una vez es llamada, lo primero que hacemos es declarar variables locales que necesitaremos para el cálculo del nodo que nos proporciona la mayor dispersión media. Para comprobar si ya hemos encontrado ese nodo que mejora la solución, utilizamos un bol que inicializaremos a false, ya que de momento el nodo no ha sido encontrado.

Tras esta declaración inicial, procedemos recorrer el vector de nodos candidatos, en busca del primer nodo que mejore la solución, es por ello por lo que la condición de parada de nuestro while(), ahora ya es diferente, debido a que pueden ocurrir dos opciones:

- Que encontremos el nodo que mejora la solución, por lo que terminamos el bucle y salimos.
- O que no encontremos un nodo que mejora la solución por lo que terminaríamos cuando nuestro iterador “i” ha recorrido todos los nodos de nuestro vector de candidatos.

De esto, el funcionamiento es exactamente el mismo que encontramos en la función del algoritmo voraz, añadimos un nuevo nodo, calculamos la nueva dispersión, comprobamos si mejora, en caso afirmativo retornamos, en caso negativo seguimos comprobando hasta llegar al final de nuestro vector de candidatos.

```
float Algoritmo_Nuevo_::calcular_dispersion_maxima(float dispersion_actual)
{
    float dispersion = dispersion_actual;
    bool encontrado = false;

    int i = 0;
    while (i < grafo.vector_nodos.size() && encontrado == false)
    {
        vector_inicial.push_back(grafo.get_nodo(i));
        float nueva_dispersion = calcular_dispersion_media(vector_inicial);

        if (nueva_dispersion >= dispersion)
        {
            dispersion = nueva_dispersion;
            encontrado = true;
        }
        else
        {
            vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());
        }

        i++;
    }

    return dispersion;
}
```

En esta imagen, podemos ver reflejado el código utilizado en nuestra la función.

# 6º. Implementación Algoritmo Grasp

Para este tercer algoritmo, como ya se ha comentado en la sección de definición del algoritmo, vamos a tener que crear un vector auxiliar donde iremos introduciendo los nodos y de donde seleccionaremos uno de manera aleatoria.

Tras comentar esto, al igual que hicimos en el apartado anterior, comentaremos los archivos de implementación de la clase.

## a) Fichero algoritmo_grasp.h

Como es evidente, en este archivo, se encuentra la definición de los atributos y métodos utilizados para el desarrollo de este algoritmo.

```
/**
 * @copyright Universidad de La Laguna
 * @copyright Escuela Superior de Ingeniería y Tecnología
 * @copyright Grado en Ingeniería Informática
 * @copyright Asignatura: Diseño y Analisis de Algoritmos (DAA)
 * @copyright Curso: 3º Itinerario 1
 * @copyright Práctica 8. Algoritmos
 * @author: Christian Torres Gonzalez alu0101137902@ull.edu.es
 * @description: Fichero de la clase Algoritmo_GRASP_, encargada de
 * implementar el codigo que resuelve el problema de la maxima dispersion
 * media utilizando la tecnica de Algoritmo Nuevo implementado por el alumno
 * @since 23/04/2020
 * @file Fichero de implementacion de la clase Algoritmo_GRASP_
 * @version 1.0.0
 * @see https://github.com/ChristianTorresGonzalez/DAA_P8.git
*/

#pragma once

#include <iostream>
#include <vector>

#include "./algoritmos.h"

class Algoritmo_GRASP_ : public Algoritmos_
{
    private:
        int opcion;
        int iteraciones;
        int size_lrc;
        vector<Nodo_> vector_inicial;
        vector<Nodo_> lrc;

    public:
        Algoritmo_GRASP_(Grafo_ , int , int , int );

        void resolver_algoritmo(int );
        void preprocesamiento();
        void fase_constructiva(Nodo_ &);
        bool post_procesamiento(Nodo_ &);
        void actualizar_solucion();

        vector<Nodo_> resolver_algoritmo(vector<Nodo_> &, int metodo);
        void fase_constructiva_multiarranque(Nodo_ &);
        bool post_procesamiento_multiarranque(Nodo_ &);
};
```

Empecemos comentando los atributos y métodos que encontramos en esta clase. En cuanto a los atributos, encontramos:

- **Vector_inicial:** es el vector en el que iremos almacenado nuestra solución relativa. Es decir, en el primer paso descrito en pseudocodigo, es en este vector donde almacenamos el resultado de nuestra fase de preprocesamiento, y donde iremos añadiendo los nodos que nos van mejorando nuestra solución.
- **Vector_lrc:** es el vector que utilizaremos en cada iteración para almacenar los nodos con mejor dispersión media entre sí, para posteriormente seleccionar uno de ellos de manera aleatoria.
- **Opción:** este atributo, se utiliza para indicar la condición de parada. Si recordamos de la sección de definición de este algoritmo, comentamos que teníamos dos nuevas condiciones de parada, pues esta variable, se utiliza para seleccionar una de las dos.
- **Iteraciones:** se utiliza para almacenar la cantidad total de iteraciones máxima que se ejecutara nuestro algoritmo.
- **Size_lrc:** se utiliza para almacenar el tamaño de nuestro vector lrc utilizado para almacenar nuestros candidatos.

En cuanto a los métodos definidos para esta clase, a parte de los métodos de los que ya disponemos en la clase padre, tenemos:

- **Algoritmo_Grasp_ (Grafo_ , int, int, int):** constructor de la clase. Simplemente lo utilizamos para inicializar los atributos definidos, y, además, para almacenar el grafo que vamos a utilizar para trabajar**.**
- **void resolver_algoritmo(int):** función a la que llamaremos desde nuestro archivo “main“(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo. Es decir, para buscar aquellos nodos, que cumplen nuestra función objetivo.
- **void preprocesamiento(void):** función utilizada para inicializar nuestro vector_inicial, con los nodos seleccionados, en función del criterio utilizado para la fase de preprocesamiento. En este caso ese criterio es seleccionar la arista de mayor distancia.
- **void fase_constructiva(void):** función utilizada para crear nuestro vector lrc, donde almacenaremos los nodos con mejor dispersión media entre sí, para posteriormente seleccionarlo de manera aleatoria.
- **bool postprocesamiento(void):** función utilizada para comprobar si el nodo que se ha seleccionado aleatoriamente como posible candidato, cumple las condiciones de la función objetivo.
- **void actualizar_solucion(void)**: función utilizada para actualizar la solución.

Además de las funciones comentadas, también podemos ver que hay otras tres funciones más al final del archivo, pero que no comentaremos de momento, ya que esas funciones se utilizan para la resolución de un algoritmo que explicaremos más adelante.

Como bien se introdujo en le explicación del algoritmo, la lógica de este algoritmo es prácticamente similar a la del algoritmo voraz, cambiando solo la función de selección. Ya que ahora en vez de seleccionar el mejor, seleccionaremos un nodo de manera aleatoria.

### b) Fichero algoritmo_grasp.cc

Una vez comentado, todos los atributos y funciones de los que disponemos en esta clase, pasemos a comentar la implementación de cada una de esas funciones. Recordar, que además de las funciones definidas en la propia clase, también disponemos de las funciones definidas en la clase Padre, lo que como esas funciones son comunes al resto de algoritmos, las hemos definido ahí, ya que son de uso común.

Ya que el constructor de nuestra clase simplemente se encarga de almacenar el grafo introducido, lo veremos reflejado en la imagen siguiente, pero no comentaremos nada.

```
Algoritmo_GRASP_::Algoritmo_GRASP_(Grafo_ grafo, int size, int opcion, int iteraciones):
    iteraciones(iteraciones),
    opcion(opcion),
    size_lrc(size),
    lrc(),
    Algoritmos_(grafo)
    {}
```

Ahora sí, procedamos a comentar la función **resolver_algoritmo().**

Previamente a comentar la función, tal y como se aprecia en la imagen, esta es bastante grande, por lo que descompondremos la explicación de toda esta función en partes más pequeñas, ya que será más fácil de explicar

```
33      void Algoritmo_GRASP_::resolver_algoritmo(int metodo)
34      {
35          cronometro.start();
36          vector_solucion.resize(0);
37          int repeticiones = 0;
38
39          if (metodo == 0)
40              preprocesamiento();
41
42          vector_solucion = vector_inicial;
43
44          dispersion_media = calcular_dispersion_media(vector_inicial);
45
46          if (opcion == 0)
47          {
48              while(repeticiones < iteraciones)
49              {
50                  lrc.resize(0);
51                  Nodo_ nodo;
52
53                  if (metodo == 0 || metodo == 1)
54                  {
55                      fase_constructiva(nodo);
56                      post_procesamiento(nodo);
57                      actualizar_solucion();
58                  }
59                  else if (metodo == 2)
60                  {
61                      fase_constructiva_multiarranque(nodo);
62                      post_procesamiento_multiarranque(nodo);
63                      actualizar_solucion();
64                  }
65
66                  repeticiones++;
67              }
68
69              cronometro.end();
70          }
71          else
72          {
73              while(repeticiones < iteraciones)
74              {
75                  lrc.resize(0);
76                  Nodo_ nodo;
77
78                  fase_constructiva(nodo);
79                  int salida = post_procesamiento(nodo);
80                  actualizar_solucion();
81
82                  if (salida == false)
83                      repeticiones++;
84              }
85
86              cronometro.end();
87          }
88      }
```

```
cronometro.start();
cronometro.end();
```

Esta primera sentencia que estamos viendo en la imagen, que es **cronometro.start() y cronometro.end ()**, se utilizan para registrar el tiempo que tarde el algoritmo en realizar la ejecución completa. La sentencia **cronometro.start()** se sitúa al inicio de cada código, para iniciar el cronometro y la sentencia **cronometro.end()** se sitúa en el final, utilizada para detener el cronometro y contabilizar el tiempo transcurrido.

Posterior a esta línea de iniciar el cronometro se encuentra la definición de variables locales que utilizaremos, que es el número de repeticiones, para contabilizar las veces que hemos iterado sobre nuestro bucle.

Esta sentencia que estamos viendo en la imagen, que es la que se encuentra en la siguiente posición a las líneas antores comentadas, se utiliza para detectar que algoritmo queremos utilizar. Para entenderlo mejor, tenemos que saber que el algoritmo que se pide en el siguiente apartado de la practica (**Algoritmo Multiarranque**), una de las posibles implementaciones, es modificar la fase de preprocesamiento, pero el resto de fases, es exactamente lo mismo que el Algoritmo Grasp, por lo que para no copiar y pegar el código de Grasp en nuestra clase Multiarranque, lo que hacemos es crear un objeto de tipo **Algoritmo_Grasp**, y es ese objeto el que utilizamos. Esta explicación viene porque este if que estamos viendo en la imagen viene por eso, porque en función de la opción que le pasemos, significara que queremos ejecutar solo el **Algoritmo_Grasp**, por lo que ejecutaremos la fase de preprocesamiento siguiendo la técnica de seleccionar la arista con mayor distancia, o por el contrario, que queremos ejecutar el **algoritmo_Multiarranque** usando Grasp, por lo que no hace falta ejecutar la fase de preprocesamiento de Grasp, porque ya se ha ejecutado la de Multiarranque.

```
if (metodo == 0)
    preprocesamiento();
```

Como en el resto de los algoritmos, igualamos nuestra solución inicial a nuestra solución y calculamos su dispersión media inicial.

Una vez realizadas las líneas anteriores, es cuando procedemos a ejecutar Grasp, en función del criterio de parada seleccionado.

```
if (opcion == 0)
{
    while(repeticiones < iteraciones)
    {
        lrc.resize(0);
        Nodo_ nodo;

        if (metodo == 0 || metodo == 1)
        {
            fase_constructiva(nodo);
            post_procesamiento(nodo);
            actualizar_solucion();
        }
        else if (metodo == 2)
        {
            fase_constructiva_multiarranque(nodo);
            post_procesamiento_multiarranque(nodo);
            actualizar_solucion();
        }

        repeticiones++;
    }

    cronometro.end();
}
```

La imagen superior muestra la ejecución del algoritmo Grasp, mediante el primer criterio de parada, que recordemos que es el de ejecutar Grasp un número determinado de veces (un número determinado de iteraciones) independientemente de que el nodo elegido sea válido o no.

Dentro de nuestro bucle while(), apreciamos un primer if, que utilizamos para comprobar el algoritmo que estamos ejecutado. Recordemos la explicación anterior cuando hablábamos de la fase de preprocesamiento. Recordemos que utilizaremos Grasp, para la ejecución de nuestro algoritmo Multiarranque por lo que habrá que diferenciar la ejecución de un algoritmo u otro. Tipos de ejecución

- **Opción = 0**, ejecución de algoritmo Grasp
- **Opción = 1 | 2**, ejecución de algoritmo Multiarranque

Una vez seleccionada la opción a ejecutar, simplemente nos limitamos a seguir el pseudocodigo que habíamos puesto en la imagen de la explicación del algoritmo(sección "**Algoritmo Grasp**")

La siguiente imagen que estamos viendo muestra la segunda versión en la que puede ser ejecutado el algoritmo Grasp, que recordemos que es el criterio de parada de iteraciones sin mejora. Esto quiere decir, que solo incrementaremos las iteraciones, cuando el nodo que seleccionamos en nuestra fase constructiva que posteriormente analizamos en la fase Postprocesamiento no es válido para entrar a la solución. Por lo que el código ejecutado en el interior de este while es similar al anterior, lo que ahora nuestra función Postprocesamiento retorna un bool para comprobar si el nodo es válido o no.

```
else
{
    while(repeticiones < iteraciones)
    {
        lrc.resize(0);
        Nodo_ nodo;

        fase_constructiva(nodo);
        int salida = post_procesamiento(nodo);
        actualizar_solucion();

        if (salida == false)
            repeticiones++;
    }

    cronometro.end();
}
```

En cuanto a la fase de preprocesamiento tal y como se aprecia en la imagen de abajo, el código es exactamente el mismo que el utilizado para los algoritmos Greedy, en donde seleccionamos la arista de mayor distancia y es de esa arista de donde sacamos los nodos con los que inicializaremos nuestra solución.

```
void Algoritmo_GRASP_::preprocesamiento()
{
    Arista_ arista = calcular_arista_maxima();

    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
    vector_inicial.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));

    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);
}
```

Pasamos ahora a la explicación de la fase constructiva, ya que es aquí donde aparece la primera diferencia respecto al algoritmo Greedy implementado anteriormente.

Tal y como se aprecia en la imagen superior, vamos a tener que recorrer todo nuestro vector de candidatos, ya que lo que vamos a hacer es ir rellenando nuestro vector LRC, con los mejores nodos en función de su dispersión media. En este caso, nuestro iterador sobre los nodos será "**i, j**". Una vez declarado esto, entramos a la ejecución de nuestro while. La condición de parada de nuestro while es una vez ya hemos recorrido todos los nodos de nuestro vector de candidatos que todavía no forman parte de nuestra solución.

```
void Algoritmo_GRASP_::fase_constructiva(Nodo_ &nodo)
{
    int i, j = 0;
    while(j < grafo.vector_nodos.size())
    {
        if (i < size_lrc)
        {
            lrc.push_back(grafo.get_nodo(j));
            i++;
            j++;
        }
        else
        {
            i = 0;
            bool cambiado = false;
            while(!cambiado || i < lrc.size())
            {
                vector_inicial.push_back(lrc[i]);
                int dispersion_anterior = calcular_dispersion_media(vector_inicial);
                vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());

                vector_inicial.push_back(grafo.get_nodo(j));
                int dispersion_siguiente = calcular_dispersion_media(vector_inicial);
                vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());

                if (dispersion_siguiente >= dispersion_anterior)
                {
                    lrc[i] = grafo.get_nodo(j);
                    cambiado = true;
                }

                i++;
            }
        }
    }

    nodo = lrc[rand() % lrc.size()];
}
```

Ahora bien, dentro del while se pueden dar dos situaciones:

- Todavía no hemos llenado toda nuestra LRC, por lo que simplemente iremos insertando los nodos que vamos analizando hasta que esta se llene.

```
if (i < size_lrc)
{
    lrc.push_back(grafo.get_nodo(j));
    i++;
    j++;
}
```

- La lista LRC ya está llena, por lo que ya no podemos insertar más elementos, sino que lo que vamos a hacer ahora es, comprobar si el nuevo nodo tiene mejor dispersión media que el que ya se encuentra dentro de nuestra LRC. Esto que hemos comentado de comparar los nodos, son las líneas que estamos viendo en la imagen inferior, donde calculamos la dispersión media con el primer nodo, luego calculamos la dispersión media con el segundo nodo y luego comparamos cual dispersión es mejor, y en función de ese sustituimos el nodo o no.

```
else
{
    i = 0;
    bool cambiado = false;
    while(!cambiado || i < lrc.size())
    {
        vector_inicial.push_back(lrc[i]);
        int dispersion_anterior = calcular_dispersion_media(vector_inicial);
        vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());

        vector_inicial.push_back(grafo.get_nodo(j));
        int dispersion_siguiente = calcular_dispersion_media(vector_inicial);
        vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());

        if (dispersion_siguiente >= dispersion_anterior)
        {
            lrc[i] = grafo.get_nodo(j);
            cambiado = true;
        }

        i++;
    }
}
```

Una vez hemos rellenado nuestra lista LRC, solo quedaría seleccionar un nodo de manera aleatoria de los introducidos en esa lista.

En cuanto a la fase de postprocesamiento, es en esta fase en la que comprobamos si el nodo seleccionado de manera aleatoria en la fase constructiva es válido, lo que significa que cumple la función objetivo. En este caso la función objetivo sigue siendo la misma que es la de comprobar si nuestra solución hasta el momento tiene mejor dispersión media que la solución sin ese nodo.

```
bool Algoritmo_GRASP_::post_procesamiento(Nodo_ &nodo)
{
    vector_inicial.push_back(nodo);

    float nueva_dispersion = calcular_dispersion_media(vector_inicial);

    if (nueva_dispersion >= dispersion_media)
    {
        dispersion_media = nueva_dispersion;
        grafo.eliminar_nodo(vector_inicial[vector_inicial.size() - 1].get_identificador_nodo());

        return true;
    }
    else
    {
        vector_inicial.erase(vector_inicial.begin() + vector_inicial.size());

        return false;
    }
}
```

El resultado de esta función como podemos comprobar es retornar un true o false en función de si el nodo es válido o no. Pero además de esto también se encarga de que, si el nodo es válido, lo añade a nuestra solución y lo elimina del grafo para que no se repita, y en caso de que no sea válido lo descarta.

En nuestra función principal, en la fase de actualización, simplemente tenemos que igualar nuestro vector_inicial, que es donde vamos almacenando nuestra solución parcial, a nuestra solución final.

```
void Algoritmo_GRASP_::actualizar_solucion()
{
    vector_solucion = vector_inicial;
}
```

# 7º. Implementación Algoritmo Multiarranque

Para este cuarto algoritmo, como ya se ha comentado en la sección de definición del algoritmo, vamos a tener dos posibles formas de implementarlo, en nuestro caso implementaremos ambas versiones y las expondremos en esta sección.

Tras comentar esto, al igual que hicimos en el apartado anterior, comentaremos los archivos de implementación de la clase.

### c) Fichero algoritmo_multiarranque.h

Como es evidente, en este archivo, se encuentra la definición de los atributos y métodos utilizados para el desarrollo de este algoritmo.

```
18  #pragma once
19
20  #include <iostream>
21  #include <vector>
22  #include <fstream>
23  #include <string>
24
25  #include "./algoritmos.h"
26  #include "./algoritmo_grasp.h"
27
28  class Algoritmo_Multiarranque_ : public Algoritmos_
29  {
30      private:
31          float dispersion;
32          vector<Nodo_> vector_inicial;
33
34      public:
35          Algoritmo_Multiarranque_(Grafo_ );
36
37          void resolver_algoritmo_metodo1(void);
38          void resolver_algoritmo_metodo2(void);
39
40          void preprocesamiento_metodo1(vector<Nodo_> &);
41          void preprocesamiento_metodo2(vector<Nodo_> &);
42  };
```

Empecemos comentando los atributos y métodos que encontramos en esta clase. En cuanto a los atributos, encontramos:

- **Vector_inicial:** es el vector en el que iremos almacenado nuestra solución relativa. Es decir, en el primer paso descrito en pseudocodigo, es en este vector donde almacenamos el resultado de nuestra fase de preprocesamiento, y donde iremos añadiendo los nodos que nos van mejorando nuestra solución.

En cuanto a los métodos definidos para esta clase, a parte de los métodos de los que ya disponemos en la clase padre, tenemos:

- **Algoritmo_Multiarranque_ (Grafo_):** constructor de la clase. Simplemente lo utilizamos para inicializar los atributos definidos, y, además, para almacenar el grafo que vamos a utilizar para trabajar**.**
- **void resolver_algoritmo_metodo1(void):** función a la que llamaremos desde nuestro archivo "main"(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo mediante el método 1, comentado en la sección de explicación de este algoritmo
- **void resolver_algoritmo_metodo2(void):** función a la que llamaremos desde nuestro archivo "main"(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo mediante el método 2, comentado en la sección de explicación de este algoritmo
- **void preprocesamiento_metodo1(void):** función utilizada para inicializar nuestro vector_inicial, mediante el criterio elegido, que en nuestro caso es seleccionar dos nodos de manera aleatoria.
- **void preprocesamiento_metodo2(void):** función utilizada para inicializar nuestro vector_inicial, que en este caso es el mismo que el del método Grasp y del Greedy.

Tal y como comentábamos en la sección referente al algoritmo Grasp, comentábamos que había una serie de funciones que implementábamos pero que las utilizaríamos en Grasp, sino que las utilizaríamos en Multiarranque, por lo que además de las funciones definidas aquí, también tenemos funciones que utilizaremos en el fichero de implementación del algoritmo Grasp, que son las que estamos viendo en la imagen inferior.

```
vector<Nodo_> resolver_algoritmo(vector<Nodo_> &, int metodo);
void fase_constructiva_multiarranque(Nodo_ &);
bool post_procesamiento_multiarranque(Nodo_ &);
```

### d) Fichero algoritmo_multiarranque.cc

Una vez comentado, todos los atributos y funciones de los que disponemos en esta clase, pasemos a comentar la implementación de cada una de esas funciones. Recordar, que además de las funciones definidas en la propia clase, también disponemos de las funciones definidas en la clase Padre, lo que como esas funciones son comunes al resto de algoritmos, las hemos definido ahí, ya que son de uso común.

Ya que el constructor de nuestra clase simplemente se encarga de almacenar el grafo introducido, lo veremos reflejado en la imagen siguiente, pero no comentaremos nada.

```
Algoritmo_Multiarranque_::Algoritmo_Multiarranque_(Grafo_ grafo):
    Algoritmos_(grafo)
    {}
```

Ahora sí, pasemos a comentar la función **resolver_algoritmo_metodo1().**

Como bien ya se adelantó en la explicación del Algoritmo_Grasp, mencionábamos que en Multiarranque, íbamos a definir un objeto de tipo Algoritmo_Grasp e iba a ser ese objeto el que íbamos a utilizar en vez de copiar y pegar todo el código de Grasp en este fichero. Es por eso por lo que, en la siguiente imagen, referente a la ejecución del **método 1,** se aprecia como, en primer lugar, llamamos al método preprocesamiento_metodo1(), que recordemos que es el que utilizamos para declarar los dos nodos aleatorios que van a constituir nuestra solución inicial, y ya es después, cuando llamamos a ejecutar Grasp, pasándole esa nueva solución inicial de la que partimos.

```
void Algoritmo_Multiarranque_::resolver_algoritmo_metodo1(void)
{
    cronometro.start();
    vector_solucion.resize(0);

    preprocesamiento_metodo1(vector_inicial);

    Algoritmo_GRASP_ algoritmo(grafo, 3, 1, 1000);
    vector_solucion = algoritmo.resolver_algoritmo(vector_inicial, 1);
    dispersion_media = algoritmo.dispersion_media;

    cronometro.end();
}
```

```
void Algoritmo_Multiarranque_::preprocesamiento_metodo1(vector<Nodo_> &vector_nodos)
{
    int nodo = rand() % grafo.get_numero_nodos();
    int arista = rand() % grafo.get_nodo(nodo).get_cantidad_aristas();

    Arista_ arista_inicial = grafo.get_nodo(nodo).get_arista(arista);

    vector_nodos.push_back(grafo.get_nodo(arista_inicial.get_nodo_inicial() - 1));
    vector_nodos.push_back(grafo.get_nodo(arista_inicial.get_nodo_destino() - 1));

    grafo.eliminar_nodo(arista_inicial.get_nodo_inicial());
    grafo.eliminar_nodo(arista_inicial.get_nodo_destino());
}
```

Tras la ejecución de esta primera fase de preprocesamiento, llamamos a la ejecución de nuestro algoritmo Grasp, pero no llamamos a la función de manera directa, sino que tenemos que llamar a una función adicional, que es a la que le pasamos nuestra nueva solución inicial. Es esta función encargada la encargada, de en primer lugar, almacenar nuestra nueva solución inicial, y posteriormente llamar a la ejecución del algoritmo Grasp de manera normal y corriente

```
vector<Nodo_> Algoritmo_GRASP_::resolver_algoritmo(vector<Nodo_> &vector_inicio, int metodo)
{
    cronometro.start();
    vector_inicial = vector_inicio;
    resolver_algoritmo(metodo);

    return vector_solucion;
    cronometro.end();
}
```

Esto sería todo lo que comentar en cuanto al método 1 de ejecución del algoritmo Multiarranque, por lo que pasaríamos a comentar el método 2.

Ahora sí, pasemos a comentar la función **resolver_algoritmo_metodo2().**

En primer lugar, recordemos que ahora nuestra fase de preprocesamiento consiste en aplicar la misma técnica que en el Algoritmo Grasp, es decir, seleccionar la arista con mayor distancia y utilizar los dos nodos que una esa arista.

```
void Algoritmo_Multiarranque_::preprocesamiento_metodo2(vector<Nodo_> &vector_nodos)
{
    Arista_ arista = calcular_arista_maxima();

    vector_nodos.push_back(grafo.get_nodo(arista.get_nodo_inicial() - 1));
    vector_nodos.push_back(grafo.get_nodo(arista.get_nodo_destino() - 1));

    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_inicial() - 1);
    grafo.vector_nodos.erase(grafo.vector_nodos.begin() + arista.get_nodo_destino() - 2);
}
```

Como bien ya comenté antes, en este método, la principal diferencia, aparecía en la forma de construir nuestra LRC, en nuestra fase constructiva, ya que ahora, la solución inicial es exactamente la misma que la de Grasp. Ahora, nuestra nueva fase constructiva consiste en lo siguiente:

```
void Algoritmo_Multiarranque_::resolver_algoritmo_metodo2(void)
{
    cronometro.start();
    preprocesamiento_metodo2(vector_inicial);

    Algoritmo_GRASP_ algoritmo(grafo, 5, 0, 1000);
    vector_solucion = algoritmo.resolver_algoritmo(vector_inicial, 2);
    dispersion_media = algoritmo.dispersion_media;

    cronometro.end();
}
```

Como bien ya he comentado anteriormente, mi nueva búsqueda local, va a consistir, en que ahora en vez de meter los mejores nodos entre sí, voy a introducir los nodos cuya dispersión media es mejor que la dispersión media que la actual, por lo que vamos a recorrer nuestro vector de candidatos, buscando aquellos nodos que tienen una mejor dispersión media que la solución actual. Si se cumplen que tienen mejor dispersión media que la actual, entonces los añadimos a nuestra LRC, hasta que esta se llene o hasta que se cumple que ya recorrimos todos los nodos y ya no hay ninguno más que la mejore. Una vez hemos o bien llenado nuestra LRC o bien hemos recorrido todos los nodos, es cuando seleccionamos uno de esos nodos de manera aleatoria, y se lo pasamos a la fase de postprocesamiento para ver si ese nodo es un posible candidato a solución. Si nos fijamos en nuestra fase_constructiva, hay dos opciones a la hora de retornar. Esto se debe a que, a la hora de retornar el nodo seleccionado, pueden ocurrir dos cosas, que se haya seleccionado un nodo de nuestra LRC, o que no se haya seleccionado ningún nodo de nuestra LRC, porque esta estaba vacía, si se da este caso, retornamos el nodo con identificador -1, para saber que no es un nodo valido y que hay que descartar esa solución.

```
bool Algoritmo_GRASP_::post_procesamiento_multiarranque(Nodo_ &nodo)
{
    if (nodo.get_identificador_nodo() != -1)
    {
        vector_inicial.push_back(nodo);
        float nueva_dispersion = calcular_dispersion_media(vector_inicial);
        dispersion_media = nueva_dispersion;
        grafo.eliminar_nodo(nodo.get_identificador_nodo());
    }
}
```

En este caso, en nuestra función de postprocesamiento es por lo comentado en el párrafo anterior, que aquí comprobamos el identificador del nodo, para saber si es un nodo valido o no, en caso de ser un nodo valido, como cumple nuestra función objetivo que ya habíamos definido en nuestra nueva fase constructiva, simplemente tenemos que añadir el nodo a nuestra solución, y actualizar la nueva dispersión media.

En cuanto a la ejecución del algoritmo Grasp, no comentaremos nada, ya que la explicación referente a ese algoritmo se ha realizado en la sección anterior.

# 8º. Implementación Algoritmo VNS

Para este quinto algoritmo, como ya se ha comentado en la sección de definición del algoritmo, vamos a partir de una solución inicial, la cual hemos obtenido mediante el algoritmo Grasp, una vez obtenida esta solución, aplicaríamos el algoritmo descrito. Este algoritmo simplemente consiste en ejecutar un número de veces el algoritmo seleccionando un nodo de los que no forma parte de nuestra solución, y cambiarlo por uno que tenemos en nuestra solución, y comparar las diferentes dispersiones medias.

Tras comentar esto, al igual que hicimos en el apartado anterior, comentaremos los archivos de implementación de la clase.

### a) Fichero algoritmo_vns.h

Como es evidente, en este archivo, se encuentra la definición de los atributos y métodos utilizados para el desarrollo de este algoritmo.

```
17  #pragma once
18
19  #include <iostream>
20  #include <vector>
21  #include <fstream>
22  #include <string>
23
24  #include "./algoritmos.h"
25  #include "./algoritmo_grasp.h"
26
27  class Algoritmo_VNS_ : public Algoritmos_
28  {
29      private:
30          float dispersion;
31
32      public:
33          Algoritmo_VNS_(Grafo_ );
34
35          void resolver_algoritmo(int , int , int );
36          void estructura_entorno(int , int , int );
37          void preprocesamiento();
38  };
```

Empecemos comentando los atributos y métodos que encontramos en esta clase. En cuanto a los atributos, encontramos:

- Para esta clase no hemos definido ningún atributo, ya que utilizaremos los definidos en la clase padre, por lo que no definimos ningún atributo nuevo.

En cuanto a los métodos definidos para esta clase, a parte de los métodos de los que ya disponemos en la clase padre, tenemos:

- **Algoritmo_VNS_ (Grafo_):** constructor de la clase. Simplemente lo utilizamos para inicializar los atributos definidos, y, además, para almacenar el grafo que vamos a utilizar para trabajar**.**
- **void resolver_algoritmo(int , int , int ):** función a la que llamaremos desde nuestro archivo "main"(archivo con la ejecución principal del programa) para resolver el problema utilizando este algoritmo.
- **void estructura_entorno(void):** función a la que llamaremos desde nuestro archivo desde resolver_algoritmo(), que se encarga de comprobar y de realizar el cambio de nodos entre los nodos de la solución y los nodos candidatos
- **void preprocesamiento (void):** función utilizada para inicializar nuestro vector_inicial, mediante el criterio elegido, que en nuestro caso es seleccionar el nodo con mayor distancia.

### b) Fichero algoritmo_vns.cc

Una vez comentado, todos los atributos y funciones de los que disponemos en esta clase, pasemos a comentar la implementación de cada una de esas funciones. Recordar, que además de las funciones definidas en la propia clase, también disponemos de las funciones definidas en la clase Padre, lo que como esas funciones son comunes al resto de algoritmos, las hemos definido ahí, ya que son de uso común.

Ya que el constructor de nuestra clase simplemente se encarga de almacenar el grafo introducido, lo veremos reflejado en la imagen siguiente, pero no comentaremos nada.

```
Algoritmo_Multiarranque_::Algoritmo_Multiarranque_(Grafo_ grafo):
    Algoritmos_(grafo)
    {}
```

Ahora sí, pasemos a comentar la función **resolver_algoritmo().**

Como bien ya se adelantó en la explicación del algoritmo, mencionábamos que íbamos a partir de la solución que habíamos calculado en Grasp, y a partir de esta solución, íbamos a crear nuestra estructura de entorno, que iba a ser la encargada de buscar otras soluciones vecinas y comprobar si eran mejor que la solución que ya teníamos. En esta función, simplemente, creamos un objeto Grasp y obtenemos la solución de la que partiremos.

```
void Algoritmo_VNS_::resolver_algoritmo(int iteraciones, int intercambios, int opcion)
{
    cronometro.start();
    vector_solucion.resize(0);

    preprocesamiento();

    Algoritmo_GRASP_ algoritmo(grafo, 3, 1, 1000);
    vector_solucion = algoritmo.resolver_algoritmo(vector_inicial, 1);
    dispersion_media = algoritmo.dispersion_media;
    grafo = algoritmo.grafo;
    estructura_entorno(iteraciones, intercambios, opcion);

    cronometro.end();
}
```

Tras la ejecución de esta primera fase de preprocesamiento, llamamos a la ejecución de nuestro algoritmo Grasp, pero no llamamos a la función de manera directa, sino que tenemos que llamar a una función adicional, que es a la que le pasamos nuestra nueva solución inicial. Es esta la función encargada, de en primer lugar, almacenar nuestra nueva solución inicial, y posteriormente llamar a la ejecución del algoritmo Grasp de manera normal y corriente

```
vector<Nodo_> Algoritmo_GRASP_::resolver_algoritmo(vector<Nodo_> &vector_inicio, int metodo)
{
    cronometro.start();
    vector_inicial = vector_inicio;
    resolver_algoritmo(metodo);

    return vector_solucion;
    cronometro.end();
}
```

Esto sería todo lo que tendríamos que comentar en cuanto a este método, ya que la verdadera aplicación del algoritmo reside en la función de estructura de entorno.

En esta segunda función, simplemente, vamos a seleccionar de manera aleatoria, un nodo de los posibles candidatos de nuestro grafo que todavía no ha sido seleccionado y lo intercambiaremos por uno de los nodos que forma parte de nuestra solución. El objetivo de esto es intentar buscar otras soluciones vecinas a

nuestra solución, para comprobar si existen soluciones mejores que nos lleven a la solución optima.

```
void Algoritmo_VNS_::estructura_entorno(int iteraciones, int intercambios, int opcion)
{
    if (opcion == 1)
    {
        for (int i = 0; i < iteraciones; i++)
        {
            bool encontrado = false;
            int busquedas = 0;

            while (busquedas < intercambios && encontrado == false)
            {
                vector<Nodo_> auxiliar = vector_solucion;
                Nodo_ nodo_candidato = grafo.get_nodo(rand() % grafo.vector_nodos.size());
                int posicion = rand() % auxiliar.size();

                Nodo_ nodo_extraido = auxiliar[posicion];
                auxiliar.erase(auxiliar.begin() + posicion);
                auxiliar.push_back(nodo_candidato);

                float disperion_nueva = calcular_dispersion_media(auxiliar);

                if (disperion_nueva >= dispersion_media)
                {
                    vector_solucion = auxiliar;
                    encontrado = true;
                    dispersion_media = disperion_nueva;
                    grafo.eliminar_nodo(nodo_candidato.get_identificador_nodo());
                    grafo.vector_nodos.push_back(nodo_extraido);
                }

                busquedas++;
            }
        }

    }
```

# 9º. Tablas comparativas

**El tiempo calculado ha sido expresado en segundos**

| Problema | N | Ejecución | Md | CPU |
|---|---|---|---|---|
| **ID1** | 10 | 1 | 10.1429 | 0.000152 |
| **ID1** | 10 | 2 | 10.1429 | 0.000151 |
| **ID1** | 10 | 3 | 10.1429 | 0.000154 |
| **ID1** | 10 | 4 | 10.1429 | 0.000154 |
| **ID1** | 10 | 5 | 10.1429 | 0.000154 |
| **ID2** | 15 | 1 | 9.5 | 0.000217 |
| **ID2** | 15 | 2 | 9.5 | 0.000218 |
| **ID2** | 15 | 3 | 9.5 | 0.000219 |
| **ID2** | 15 | 4 | 9.5 | 0.000218 |
| **ID2** | 15 | 5 | 9.5 | 0.000218 |
| **ID3** | 20 | 1 | 12.8571 | 0.000627 |
| **ID3** | 20 | 2 | 12.8571 | 0.000631 |
| **ID3** | 20 | 3 | 12.8571 | 0.000628 |
| **ID3** | 20 | 4 | 12.8571 | 0.000628 |
| **ID3** | 20 | 5 | 12.8571 | 0.000626 |

**(Tabla algoritmo voraz)**

| Problema | N | Ejecución | Md | CPU |
|---|---|---|---|---|
| **ID1** | 10 | 1 | 10.1429 | 0.000121 |
| **ID1** | 10 | 2 | 10.1429 | 0.000121 |
| **ID1** | 10 | 3 | 10.1429 | 0.000121 |
| **ID1** | 10 | 4 | 10.1429 | 0.000121 |
| **ID1** | 10 | 5 | 10.1429 | 0.000121 |
| **ID2** | 15 | 1 | 9.833 | 0.000231 |
| **ID2** | 15 | 2 | 9.833 | 0.000233 |
| **ID2** | 15 | 3 | 9.833 | 0.00023 |
| **ID2** | 15 | 4 | 9.833 | 0.00023 |
| **ID2** | 15 | 5 | 9.833 | 0.00023 |
| **ID3** | 20 | 1 | 8 | 0.000577 |
| **ID3** | 20 | 2 | 8 | 0.000577 |
| **ID3** | 20 | 3 | 8 | 0.000577 |
| **ID3** | 20 | 4 | 8 | 0.000577 |
| **ID3** | 20 | 5 | 8 | 0.000577 |

**(Tabla algoritmo voraz implementado por el alumno)**

| Problema | N | LRC | Ejecución | Md | CPU |
|---|---|---|---|---|---|
| **ID1** | 10 | 2 | 1 | 10 | 0.005745 |
| **ID1** | 10 | 2 | 2 | 10.1429 | 0.005535 |
| **ID1** | 10 | 2 | 3 | 10 | 0.00589 |
| **ID1** | 10 | 2 | 4 | 10.1429 | 0.005599 |
| **ID1** | 10 | 2 | 5 | 10.125 | 0.006119 |
| **ID1** | 10 | 3 | 1 | 10.1429 | 0.005603 |
| **ID1** | 10 | 3 | 2 | 10.1429 | 0.005517 |
| **ID1** | 10 | 3 | 3 | 10.1429 | 0.005824 |
| **ID1** | 10 | 3 | 4 | 10.125 | 0.006125 |
| **ID1** | 10 | 3 | 5 | 10.1429 | 0.005699 |
| **ID2** | 15 | 2 | 1 | 9.83333 | 0.008334 |
| **ID2** | 15 | 2 | 2 | 9.5 | 0.007666 |
| **ID2** | 15 | 2 | 3 | 8.85174 | 0.008941 |
| **ID2** | 15 | 2 | 4 | 9.83333 | 0.008243 |
| **ID2** | 15 | 2 | 5 | 9.5 | 0.007655 |
| **ID2** | 15 | 3 | 1 | 9.5 | 0.00773 |
| **ID2** | 15 | 3 | 2 | 9.5 | 0.007817 |
| **ID2** | 15 | 3 | 3 | 9.83333 | 0.008253 |
| **ID2** | 15 | 3 | 4 | 7.16667 | 0.008319 |
| **ID2** | 15 | 3 | 5 | 9.5 | 0.007652 |
| **ID3** | 20 | 2 | 1 | 11.0909 | 0.01575 |
| **ID3** | 20 | 2 | 2 | 12.8571 | 0.012638 |
| **ID3** | 20 | 2 | 3 | 12.8571 | 0.012874 |
| **ID3** | 20 | 2 | 4 | 12 | 0.01197 |
| **ID3** | 20 | 2 | 5 | 10.1429 | 0.012439 |
| **ID3** | 20 | 3 | 1 | 12.8571 | 0.01268 |
| **ID3** | 20 | 3 | 2 | 11.75 | 0.013138 |
| **ID3** | 20 | 3 | 3 | 11.75 | 0.013223 |
| **ID3** | 20 | 3 | 4 | 10.1429 | 0.012552 |
| **ID3** | 20 | 3 | 5 | 12.8571 | 0.012753 |

**(Tabla algoritmo Grasp)**

| Problema | N | Ejecución | Md | CPU |
|---|---|---|---|---|
| **ID1** | 10 | 1 | 12.7143 | 0.005557 |
| **ID1** | 10 | 2 | 9 | 0.006371 |
| **ID1** | 10 | 3 | 10 | 0.005723 |
| **ID1** | 10 | 4 | 11.2857 | 0.005427 |
| **ID1** | 10 | 5 | 13 | 0.005284 |
| **ID2** | 15 | 1 | 9.83333 | 0.035759 |
| **ID2** | 15 | 2 | 9.5 | 0.030696 |
| **ID2** | 15 | 3 | 9.5 | 0.029698 |
| **ID2** | 15 | 4 | 9.83333 | 0.03631 |
| **ID2** | 15 | 5 | 9.83333 | 0.036058 |
| **ID3** | 20 | 1 | 11 | 0.013116 |
| **ID3** | 20 | 2 | 9.1 | 0.014312 |
| **ID3** | 20 | 3 | 9.1 | 0.014.03 |
| **ID3** | 20 | 4 | 6.90909 | 0.01532 |
| **ID3** | 20 | 5 | 8.16667 | 0.016355 |

**(Tabla algoritmo Multiarranque)**

| Problema | N | K | Ejecución | Md | CPU |
|---|---|---|---|---|---|
| **ID1** | 10 | 2 | 1 | 11.5714 | 0.015922 |
| **ID1** | 10 | 2 | 2 | 12.7143 | 0.016016 |
| **ID1** | 10 | 2 | 3 | 12.7143 | 0.016096 |
| **ID1** | 10 | 2 | 4 | 12.7143 | 0.016579 |
| **ID1** | 10 | 2 | 5 | 11 | 0.017876 |
| **ID1** | 10 | 3 | 1 | 12.7143 | 0.018535 |
| **ID1** | 10 | 3 | 2 | 12.7143 | 0.018698 |
| **ID1** | 10 | 3 | 3 | 12.7143 | 0.018643 |
| **ID1** | 10 | 3 | 4 | 12.7143 | 0.018371 |
| **ID1** | 10 | 3 | 5 | 11.5 | 0.022333 |
| **ID2** | 15 | 2 | 1 | 9.83333 | 0.018491 |
| **ID2** | 15 | 2 | 2 | 14 | 0.014759 |
| **ID2** | 15 | 2 | 3 | 13 | 0.01463 |
| **ID2** | 15 | 2 | 4 | 9.14286 | 0.020307 |
| **ID2** | 15 | 2 | 5 | 9.14286 | 0.020342 |
| **ID2** | 15 | 3 | 1 | 9.83333 | 0.01836 |
| **ID2** | 15 | 3 | 2 | 9.83333 | 0.018737 |
| **ID2** | 15 | 3 | 3 | 9.83333 | 0.018469 |
| **ID2** | 15 | 3 | 4 | 14 | 0.014568 |
| **ID2** | 15 | 3 | 5 | 9.83333 | 0.018325 |
| **ID3** | 20 | 2 | 1 | 11.7 | 0.037088 |
| **ID3** | 20 | 2 | 2 | 12.8571 | 0.026606 |
| **ID3** | 20 | 2 | 3 | 12.8571 | 0.026699 |
| **ID3** | 20 | 2 | 4 | 12.4 | 0.021632 |
| **ID3** | 20 | 2 | 5 | 12.4 | 0.021612 |
| **ID3** | 20 | 3 | 1 | 12.8571 | 0.032105 |
| **ID3** | 20 | 3 | 2 | 12.625 | 0.039971 |
| **ID3** | 20 | 3 | 3 | 12.8571 | 0.032874 |
| **ID3** | 20 | 3 | 4 | 12.8571 | 0.032308 |
| **ID3** | 20 | 3 | 5 | 11.7 | 0.045542 |

**(Tabla algoritmo VNS)**

# 10º. Arquitectura utilizada

**Procesador:** Intel Core i9-9900K 8 Núcleos 16 Hilos

**Memoria RAM:** 16 Gb 3200Hz

**Sistema Operativo:** Windows 10