MAX-MEAN DISPERSION PROBLEM

Universidad de La Laguna

Diseño y Análisis de Algoritmos

Este documento representa la solución del problema MAX-MEAN DISPERSION explicado con más detalle más adelante. En él se refleja la implementación de la solución, pseudocódigo de los algoritmos, las estructuras de clases seguidas y pruebas de ejecución.

Adrián Epifanio Rodríguez Hernández
alu0101158280@ull.edu.es

INDICE

1. Introducción

¿En qué consiste el Max-Mean Dispersion Problem?

El Max-Mean Dispersion Problem es una variante de un modelo de optimización clásico en el contexto de maximizar la diversidad de un conjunto de elementos. En particular este problema del cálculo de la dispersión media máxima modela varios problemas reales, como el control de la contaminación o la clasificación de una página web.

Objetivo de la práctica

Proponer, implementar y evaluar algoritmos constructivos y búsquedas por entornos para el Max-Mean Dispersion Problem.

Max-mean Dispersion Problem

Sea dado un grafo completo G = (V, E), donde V es el conjunto de vértices (|V| = n) y E es el conjunto de aristas (|E| = n(n − 1)/2. Cada arista (i, j) ∈ E tiene asociada una distancia o afinidad d(i, j). En el max-mean dispersion problem se desea encontrar el subconjunto de vértices S ⊆ V que maximiza la dispersión media dada por

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

Sistema de pruebas

Las características del computador con el que se han realizado las pruebas son:

- Procesador → *Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s)*
- Sistema operativo → *Microsoft Windows 10 Home*
- Memoria RAM instalada → *8 GB*
- Memoria Virtual → *11.8 GB*

2. Estructura de clases:

En este apartado se definirán las clases empleadas para la realización de la práctica, en ellas se mencionarán y definirán brevemente los métodos y los atributos de las mismas. Para ello emplearemos la notación de UML:

+ : Expresa que el atributo/método es público.

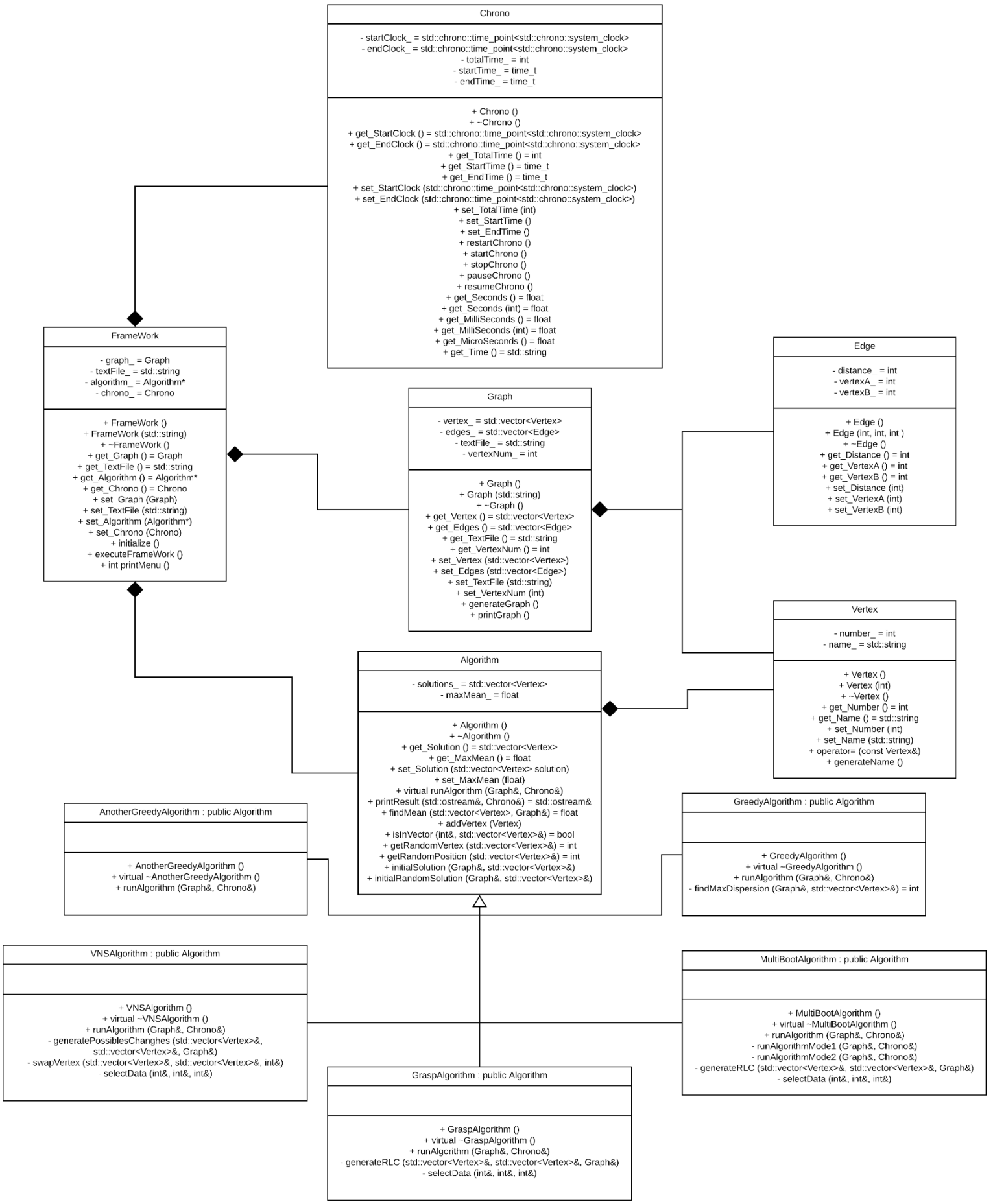
- : Expresa que el atributo/método es privado.

: Expresa que el atributo/método es protegido.

Para cada clase se hará una pequeña introducción de la misma seguida de una imagen con su respectivo diagrama en UML. A continuación, se definirán sus atributos y métodos. En el caso de las clases relacionadas con los tipos de algoritmos que buscan la solución (Greedy, Grasp, MultiBoost y VNP), además, al final de la misma estará el pseudocódigo de dicho algoritmo.

Por otro lado, también se incluye a modo de esquema y resumen un diagrama en UML que representa el conjunto de las clases, mostrando así la relación de herencia o inclusión entre ellas.

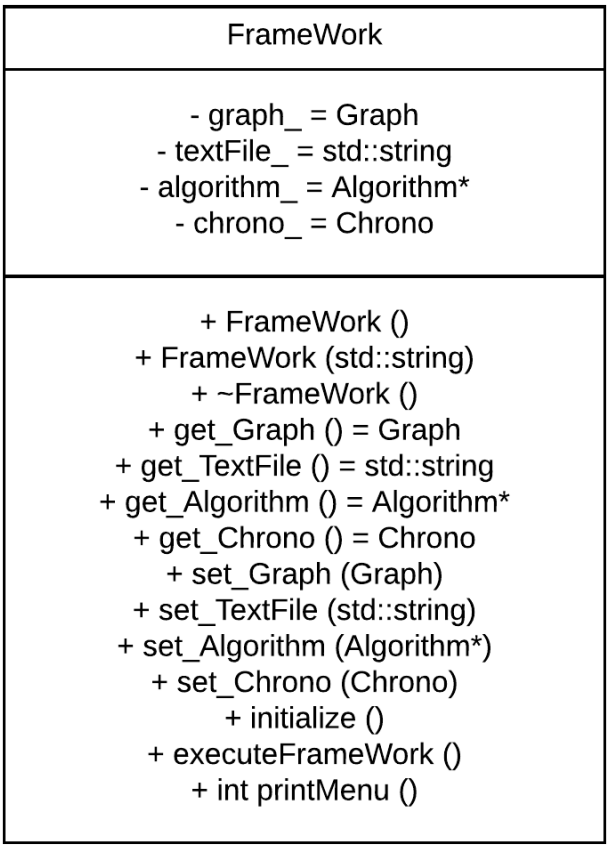
En el caso de las clases heredadas de la clase Algorithm, es decir en los distintos tipos de algoritmos de resolución en el resumen de la clase se incluirá que entendemos por una iteración en dicho algoritmo, cómo se obtiene la solución inicial y se explicará brevemente el algoritmo de búsqueda local empleado en cada clase.


Chrono
- startClock_ = std::chrono::time_point<std::chrono::system_clock>
- endClock_ = std::chrono::time_point<std::chrono::system_clock>
- totalTime_ = int
- startTime_ = time_t
- endTime_ = time_t
+ Chrono ()
+ ~Chrono ()
+ get_StartClock () = std::chrono::time_point<std::chrono::system_clock>
+ get_EndClock () = std::chrono::time_point<std::chrono::system_clock>
+ get_TotalTime () = int
+ get_StartTime () = time_t
+ get_EndTime () = time_t
+ set_StartClock (std::chrono::time_point<std::chrono::system_clock>)
+ set_EndClock (std::chrono::time_point<std::chrono::system_clock>)
+ set_TotalTime (int)
+ set_StartTime ()
+ set_EndTime ()
+ restartChrono ()
+ startChrono ()
+ stopChrono ()
+ pauseChrono ()
+ resumeChrono ()
+ get_Seconds () = float
+ get_Seconds (int) = float
+ get_MilliSeconds () = float
+ get_MilliSeconds (int) = float
+ get_MicroSeconds () = float
+ get_Time () = std::string
FrameWork
- graph_ = Graph
- textFile_ = std::string
- algorithm_ = Algorithm*
- chrono_ = Chrono
+ FrameWork ()
+ FrameWork (std::string)
+ ~FrameWork ()
+ get_Graph () = Graph
+ get_TextFile () = std::string
+ get_Algorithm () = Algorithm*
+ get_Chrono () = Chrono
+ set_Graph (Graph)
+ set_TextFile (std::string)
+ set_Algorithm (Algorithm*)
+ set_Chrono (Chrono)
+ initialize ()
+ executeFrameWork ()
+ int printMenu ()
Graph
- vertex_ = std::vector<Vertex>
- edges_ = std::vector<Edge>
- textFile_ = std::string
- vertexNum_ = int
+ Graph ()
+ Graph (std::string)
+ ~Graph ()
+ get_Vertex () = std::vector<Vertex>
+ get_Edges () = std::vector<Edge>
+ get_TextFile () = std::string
+ get_VertexNum () = int
+ set_Vertex (std::vector<Vertex>)
+ set_Edges (std::vector<Edge>)
+ set_TextFile (std::string)
+ set_VertexNum (int)
+ generateGraph ()
+ printGraph ()
Edge
- distance_ = int
- vertexA_ = int
- vertexB_ = int
+ Edge ()
+ Edge (int, int, int)
+ ~Edge ()
+ get_Distance () = int
+ get_VertexA () = int
+ get_VertexB () = int
+ set_Distance (int)
+ set_VertexA (int)
+ set_VertexB (int)
Vertex
- number_ = int
- name_ = std::string
+ Vertex ()
+ Vertex (int)
+ ~Vertex ()
+ get_Number () = int
+ get_Name () = std::string
+ set_Number (int)
+ set_Name (std::string)
+ operator= (const Vertex&)
+ generateName ()
Algorithm
- solutions_ = std::vector<Vertex>
- maxMean_ = float
+ Algorithm ()
+ ~Algorithm ()
+ get_Solution () = std::vector<Vertex>
+ get_MaxMean () = float
+ set_Solution (std::vector<Vertex> solution)
+ set_MaxMean (float)
+ virtual runAlgorithm (Graph&, Chrono&)
+ printResult (std::ostream&, Chrono&) = std::ostream&
+ findMean (std::vector<Vertex>, Graph&) = float
+ addVertex (Vertex)
+ isInVector (int&, std::vector<Vertex>&) = bool
+ getRandomVertex (std::vector<Vertex>&) = int
+ getRandomPosition (std::vector<Vertex>&) = int
+ initialSolution (Graph&, std::vector<Vertex>&)
+ initialRandomSolution (Graph&, std::vector<Vertex>&)
AnotherGreedyAlgorithm : public Algorithm
+ AnotherGreedyAlgorithm ()
+ virtual ~AnotherGreedyAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
GreedyAlgorithm : public Algorithm
+ GreedyAlgorithm ()
+ virtual ~GreedyAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
- findMaxDispersion (Graph&, std::vector<Vertex>&) = int
VNSAlgorithm : public Algorithm
+ VNSAlgorithm ()
+ virtual ~VNSAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
- generatePossiblesChanghes (std::vector<Vertex>&, std::vector<Vertex>&, Graph&)
- swapVertex (std::vector<Vertex>&, std::vector<Vertex>&, int&)
- selectData (int&, int&, int&)
MultiBootAlgorithm : public Algorithm
+ MultiBootAlgorithm ()
+ virtual ~MultiBootAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
- runAlgorithmMode1 (Graph&, Chrono&)
- runAlgorithmMode2 (Graph&, Chrono&)
- generateRLC (std::vector<Vertex>&, std::vector<Vertex>&, Graph&)
- selectData (int&, int&, int&)
GraspAlgorithm : public Algorithm
+ GraspAlgorithm ()
+ virtual ~GraspAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
- generateRLC (std::vector<Vertex>&, std::vector<Vertex>&, Graph&)
- selectData (int&, int&, int&)

2.1 Class FrameWork

En esta clase almacenamos el grafo generado por los datos del fichero de entrada y posteriormente se le pedirá al usuario que elija que algoritmo desea implementar para la resolución del mismo.

Diagrama UML de la clase FrameWork



Atributos de la clase FrameWork

- `graph_`: Objeto de la clase *Graph* en donde almacenaremos los datos de entrada.

- `textFile_`: Cadena de caracteres que contendrá el nombre del fichero de entrada con los datos iniciales.

- `algorithm_`: Puntero a un objeto de la clase Algorithm que será el encargado de ejecutar el algoritmo para calcular la solución al problema.

- `chorno_`: Objeto de la clase *Chrono* que se encargará de medir el tiempo que tarda la ejecución del algoritmo.

Métodos de la clase FrameWork

`+ Framework ()`: Constructor por defecto de la clase.

`+ Framework (string textfile)`: Constructor de la clase al que recibe y añade el nombre del fichero de entrada a los atributos del grafo.

`+ ~Framework ()`: Destruye el objeto framework.

`+ Graph get_Graph ()`: Método que devuelve el atributo *graph_*.

`+ get_TextFile ()`: Método que devuelve el atributo *textFile_*.

`+ get_Algorithm ()`: Método que devuelve el atributo *algorithm_*.

`+ get_Chrono ()`: Método que devuelve el atributo *chrono_*.

`+ set_Graph (Graph graph)`: Método que establece el atributo *graph_*.

`+ set_TextFile (string textFile)`: Método que establece el atributo *textFile_*.

`+ set_Algorithm (Algorithm* algorithm)`: Método que establece el atributo *algorithm_*.

`+ set_Chrono (Chrono chrono)`: Método que establece el atributo *chorno_*.

`+ initialize ()`: Método que inicializa el objeto FrameWork, inicializa el grafo y pide al usuario que indique que clase de algoritmo desea ejecutar para la resolución del problema.

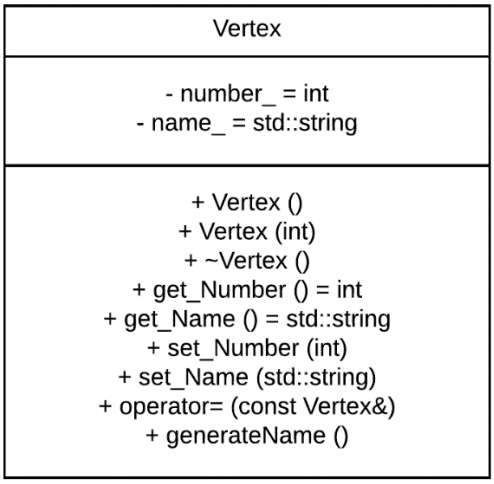
`+ executeFrameWork ()`: Método que se encarga de llamar a la ejecución del algoritmo, contabilizar el tiempo y llamar a la impresión del tiempo y los resultados del algoritmo

`+ printMenu ()`: Método que imprime un menú en el que se le solicita al usuario la selección del algoritmo que se desea emplear.

2.2 Class Vertex

Esta clase se emplea para almacenar un vértice con su nombre y numeración con la que será tratado en los algoritmos. En el caso concreto de nuestra práctica se creará un vector de vértices donde se almacenarán todos los vértices correspondientes a un determinado grafo.

Diagrama UML de la clase Vertex



Atributos de la clase Vertex

- `number_`: Número que se le asigna al vértice, su identificador en el programa.

- `name_`: Nombre del vértice en caso de tenerlo, en caso de no tener nombre se le asignaría como "Vertex X" donde X es el número identificatorio del vértice.

Métodos de la clase Vertex

`+ Vertex ()`: Constructor por defecto del vértice.

`+ Vertex (int number)`: Constructor al que se le pasa como parámetro el número que identifica al vértice.

`+ ~Vertex ()`: Destruye el objeto Vertex.

`+ get_Number ()`: Método que devuelve el atributo *number_*.

`+ get_Name ()`: Método que devuelve el atributo *name_*.

`+ set_Number (int number)`: Método que establece el atributo *number_*.

`+ set_Name (string name)`: Método que establece el atributo *name_*.

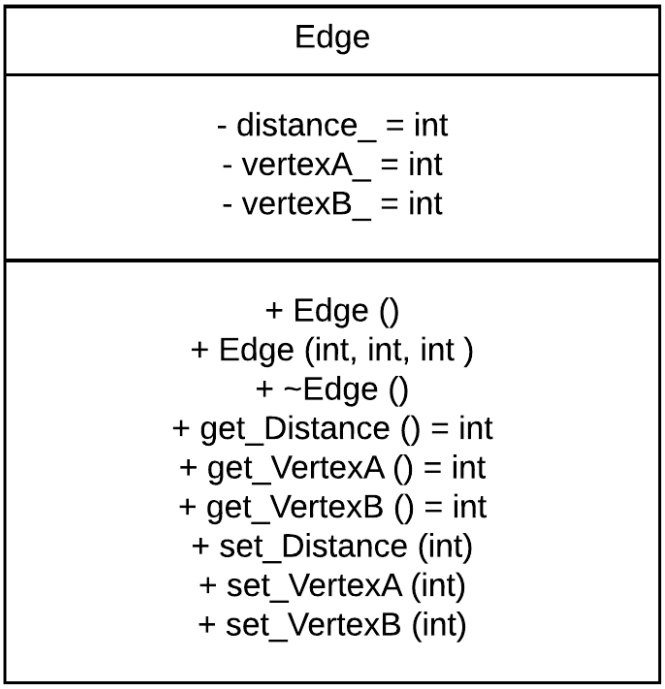
`+ operator= (const Vertex& vertex)`: Sobrecarga del operador de asignación.

`+ generateName ()`: Genera el nombre por defecto del vértice.

2.3 Class Edge

Esta clase se emplea para almacenar una arista, una arista es la línea que une un vértice con otro y tiene un valor como puede ser la distancia, coste, etc... En el caso particular de nuestra práctica emplearemos esta clase para almacenar todas las aristas pertenecientes a un determinado grafo.

Diagrama UML de la clase Edge



Atributos de la clase Edge

- `distance_`: Numero que almacena el coste en trasladarse del vértice A al vértice B.
- `vertexA_`: Número identificatorio del vértice de partida de la arista.
- `vertexB_`: Número identificatorio del vértice final de la arista.

Métodos de la clase Edge

`+ Edge ()`: Constructor por defecto de la clase arista.

`+ Edge (int vertexA, int vertexB, int distance)`: Constructor de la clase arista a la que se le pasa el vértice de partida, el final y el coste de la arista.

`+ ~Edge ()`: Destructor del objeto arista.

`+ get_Distance ()`: Método que devuelve el atributo *distance_*.

`+ get_VertexA ()`: Método que devuelve el atributo *vertexA_*.

`+ get_VertexB ()`: Método que devuelve el atributo *vertexB_*.

`+ set_Distance (int distance)`: Método que establece el atributo *distance_*.

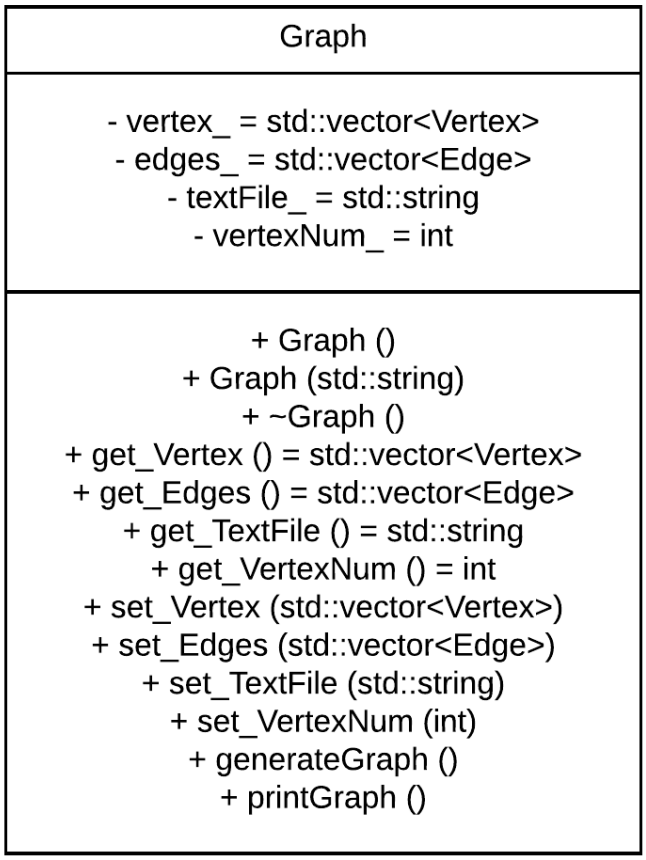
`+ set_VertexA (int vertexA)`: Método que establece el atributo *vertexA_*.

`+ set_VertexB (int vertexB)`: Método que establece el atributo *vertexB_*.

2.4 Class Graph:

Esta clase almacena los datos de un grafo, conjunto de vértices y aristas sobre los que se ejecutarán los algoritmos de búsqueda de la mejor solución.

Diagrama UML de la clase Graph



Atributos de la clase Graph

- `vertex_`: Vector de vértices donde se almacenarán todos los vértices pertenecientes al grafo.
- `edges_`: Vector de aristas donde se almacenarán todas las aristas pertenecientes al grafo.
- `textFile_`: Nombre del fichero de entrada del que se leen los datos para generar el grafo.
- `vertexNum_`: Número de vértices que posee el grafo.

Métodos de la clase Graph

`+ Graph ()`: Constructor por defecto de la clase arista.

`+ Graph (string textFile)`: Constructor de la clase que recibe como parámetro el nombre del fichero de entrada en el que se encuentran los datos del grafo.

`+ ~Graph ()`: Destructor de la clase Graph.

`+ get_Vertex ()`: Método de la clase que devuelve el vector de vértices vertex_.

`+ get_Edges ()`: Método de la clase que devuelve el vector de aristas edges_.

`+ get_TextFile ()`: Método de la clase que devuelve el atributo textFile_.

`+ get_VertexNum ()`: Método de la clase que devuelve el atributo vertexNum_.

`+ set_Vertex (vector<Vertex> vertex)`: Método de la clase que establece el vector de vértices vertex_.

`+ set_Edges (vector<Edge> edge)`: Método de la clase que establece el vector de aristas edges_.

`+ set_TextFile (string textFile)`: Método de la clase que establece el atributo textFile_.

`+ set_VertexNum (int vertexNum)`: Método de la clase que establece el atributo vertexNum_.

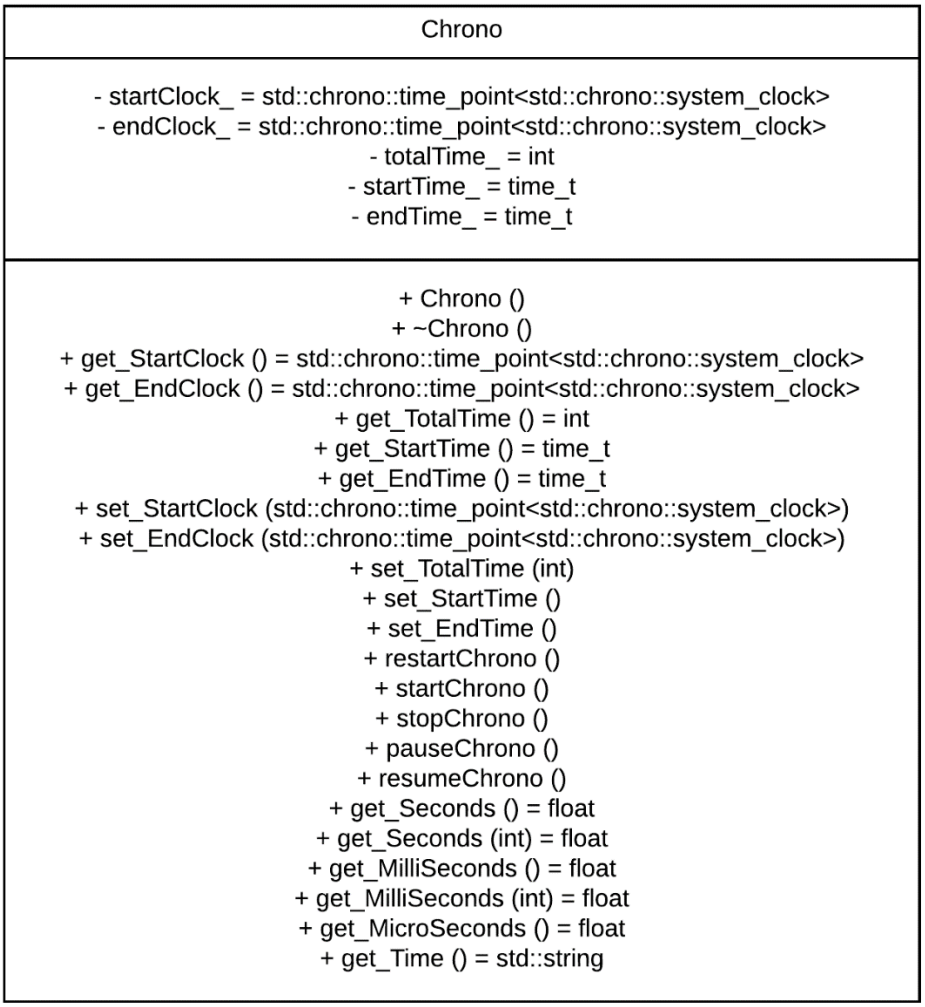
`+ generateGraph ()`: Genera el grafo leyendo los datos del fichero de entrada.

`+ printGraph ()`: Imprime el grafo por pantalla.

2.5 Class Chrono

Esta clase es empleada para crear un reloj en el programa y que este nos cronometre o calcule el tiempo que dura la ejecución de nuestros algoritmos. El cronómetro será inicializado antes de que empiece la ejecución del algoritmo y parado cuando acabe.

Diagrama UML de la clase Chrono



Atributos de la clase Chrono

- `startClock_`: Objeto de la librería *std::chrono* que almacenará el momento en que se inicia el programa.

- `endClock_`: Objeto de la librería *std::chrono* que almacenará el momento en que se finaliza el programa.

- `totalTime_`: Número de microsegundos transcurridos en ejecución.

- `startTime_`: Fecha y hora de comienzo.

- `endTime_`: Fecha y hora de finalización.

Métodos de la clase Chrono

+ `Chrono ()`: Constructor por defecto de un objeto Chrono.

+ `~Chrono ()`: Destruye el objeto Chrono.

+ `get_StartClock ()`: Devuelve el atributo *startClock_*.

+ `get_EndClock ()`: Devuelve el atributo *endClock_*.

+ `get_TotalTime ()`: Devuelve el atributo *totalTime_*.

+ `get_StartTime ()`: Devuelve el atributo *startTime_*.

+ `get_EndTime ()`: Devuelve el atributo *endTime_*.

+ `set_StartClock (startClock)`: Establece el atributo *startClock_*.

+ `set_EndClock (endClock)`: Establece el atributo *endClock_*.

+ `set_TotalTime (int totalTime)`: Establece el atributo *totalTime_*.

+ `set_StartTime ()`: Establece el atributo *startTime_*.

+ `set_EndTime ()`: Establece el atributo *endTime_*.

+ `restartChrono ()`: Resetea el objeto Chrono.

+ `startChrono ()`: Empieza a contabilizar el tiempo.

+ `stopChrono ()`: Para el Chrono y establece el tiempo total transcurrido.

+ `pauseChrono ()`: Pausa el Chrono y añade el tiempo total transcurrido al atributo *totalTime_*.

+ `resumeChrono ()`: Continúa la ejecución del Chrono en caso de que hubiese sido pausado.

+ `get_Seconds ()`: Devuelve el tiempo en segundos.

+ `get_Seconds (int decimalAmmount)`: Devuelve el tiempo en segundos con X cifras decimales donde X es valor pasado como parámetro.

+ `get_MilliSeconds ()`: Devuelve el tiempo en milisegundos.

+ `get_MilliSeconds (int decimalAmmount)`: Devuelve el tiempo en milisegundos con X cifras decimales donde X es valor pasado como parámetro.

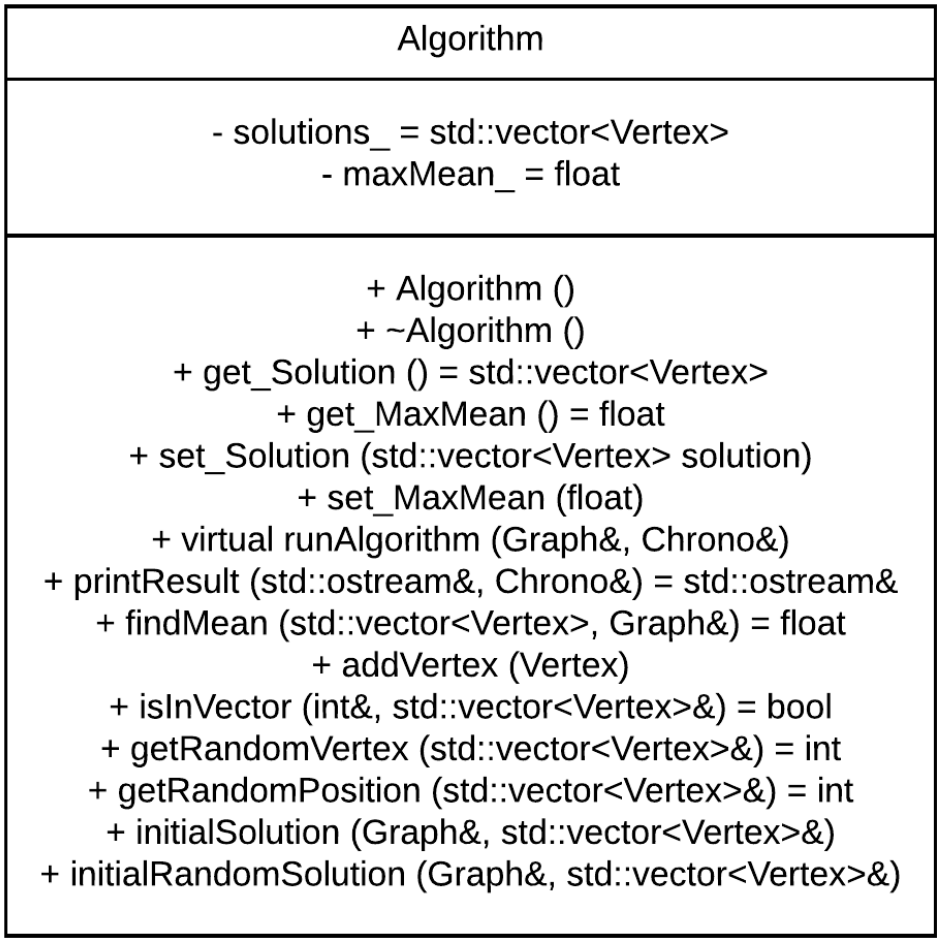
+ `get_MicroSeconds ()`: Devuelve el tiempo en microsegundos.

+ `get_Time ()`: Devuelve el tiempo en horas, minutos y segundos (Para operaciones largas).

2.6 Class Algorithm

Esta clase será la clase padre de los algoritmos de búsqueda que emplearemos "Greedy, AnotherGreedy, Grasp, MultiBoost y VNS". En ella se define una constante "STARTMEAN -9999999" que se empleará para inicializar la dispersión media al mínimo valor posible. Además, también se implementan los métodos que usan la mayoría de las clases con el fin de no tener código repetido como pueden ser los métodos que generan las soluciones iniciales, el comprobar si un determinado vértice se encuentra en un vector o el cálculo de la dispersión media.

Diagrama UML de la clase Algorithm



Atributos de la clase Algorithm

- `solutions_`: Vector de vértices en el que se almacenará la solución final de nuestro algoritmo.

- `maxMean_`: Flotante que contendrá el valor de la media de la solución con el fin de no calcularla por duplicado.

Métodos de la clase Algorithm

`+ Algorithm ()`: Constructor de la clase padre por defecto.

`+ virtual ~Algorithm ()`: Destructor de la clase padre por defecto.

`+ get_Solution ()`: Devuelve el atributo *solutions_* de la clase.

`+ get_MaxMean ()`: Devuelve el atributo *maxMean_* de la clase.

`+ set_Solution ()`: Establece el atributo *solutions_* de la clase.

`+ set_MaxMean ()`: Establece el atributo *maxMean_* de la clase.

`+ virtual runAlgorithm (Graph& graph, Chrono& chrono)`: Llama la función runAlgorithm correspondiente en función de que algoritmo se haya creado.

`+ std::ostream& printResult (std::ostream& os, Chrono& chrono)`: Muestra el resultado con la solución, el tiempo y la media del algoritmo.

`+ float findMean (std::vector<Vertex>& vertex, Graph& graph)`: Calcula la dispersión media de los vértices pertenecientes al vector que recibe como parámetro.

`+ addVertex (Vertex& newVertex)`: Añade un vértice a la solución final.

`+ bool isInVector (int number, std::vector<Vertex>& vertex)`: Comprueba si un determinado vértice está en el vector que recibe como parámetro.

`+ int getRandomVertex (std::vector<Vertex> vector)`: Devuelve el número asignado a un vértice seleccionado de forma aleatoria entre los pertenecientes al vector pasado como parámetro.

`+ int getRandomPosition (std::vector<Vertex>& vector)`: Devuelve la posición asignada a un vértice seleccionado de forma aleatoria entre los pertenecientes al vector pasado como parámetro.

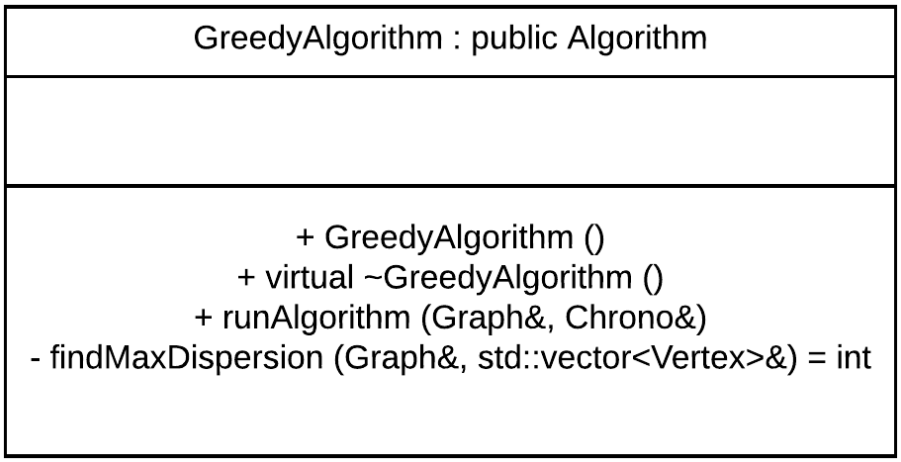
`+ initialSolution (Graph& graph, std::vector<Vertex>& vertex)`: Genera una solución inicial para los algoritmos mediante la búsqueda de la arista con mayor valor en todo el grafo.

`+ initialRandomSolution (Graph& graph, std::vector<Vertex>& solution)`: Genera una solución aleatoria inicial seleccionando dos nodos completamente al azar entre todos los pertenecientes al grafo.

2.7 Class GreedyAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica Greedy o voraz. Esta clase carece de atributos ya que los hereda de la clase padre. Este algoritmo trata de ir buscando el vértice que maximice la solución y comprobar si la solución junto con ese nuevo vértice es mejor que la solución sin él, en caso de serlo lo añade a la solución y sigue buscando. En este algoritmo la solución inicial se genera escogiendo la arista de nuestro grafo con mayor afinidad e introduciendo los dos vértices que la forman en la solución. Para este algoritmo una iteración consiste en buscar la arista que maximice nuestro conjunto de nodos en la solución y comprobar si esa nueva dispersión en la solución nueva es mejor que la anterior o no, en caso afirmativo la solución pasaría a ser la solución unida al vértice con mayor afinidad encontrado previamente.

Diagrama UML de la clase GreedyAlgorithm



Métodos de la clase GreedyAlgorithm

`+ GreedyAlgorithm ()` : Constructor vacío por defecto de la clase.

`+ virtual ~GreedyAlgorithm ()` : Destructor vacío y por defecto de la clase.

`+ runAlgorithm (Graph& graph, Chrono& chrono)` : Ejecuta el algoritmo greedy con el que se busca la solución al problema (ver pseudocódigo más adelante).

`- findMaxDispersion (Graph& graph, std::vector<Vertex>& vertex)` : Busca la arista vecina a alguno de los vértices del vector que tenga el mayor valor para maximizar posteriormente el resultado de la dispersión media.

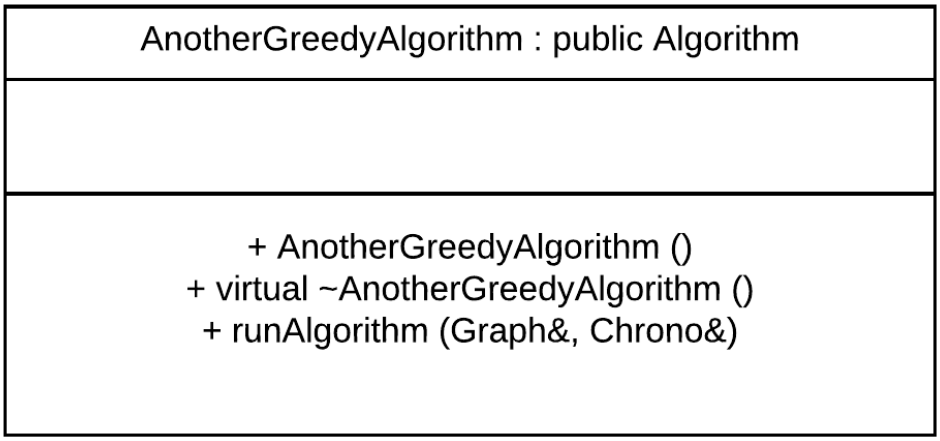
Pseudocódigo runAlgorithm

```
iniciar chrono
    S = S' = solución inicial (arista con mayor valor)
    repetir
        tmp = vértice que maximiza S'
        intoducir tmp en S'
        si md(S') > md(S) entonces:
            S = S'
    mientras (S' = S)
    guardar solución
    guardar media
    parar chrono
```

2.8 Class AnotherGreedyAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica Greedy o voraz. Esta clase carece de atributos ya que los hereda de la clase padre. Este algoritmo trata de ir buscando el vértice que maximice la solución y comprobar si la solución junto con ese nuevo vértice es mejor que la solución sin él, en caso de serlo lo añade a la solución y sigue buscando. En este algoritmo la solución inicial se genera escogiendo la arista de nuestro grafo con mayor afinidad e introduciendo los dos vértices que la forman en la solución. Para este algoritmo, una iteración consiste en ir recorriendo todo el vector de vértices del grafo e ir comprobando si la solución junto con cada uno de ellos mejora la solución o no, en caso afirmativo ese vértice pasaría a formar parte de la solución.

Diagrama UML de la clase AnotherGreedyAlgorithm



Métodos de la clase AnotherGreedyAlgorithm

`+ AnotherGreedyAlgorithm ()` : Constructor vacío por defecto de la clase.

`+ virtual ~AnotherGreedyAlgorithm ()` : Destructor vacío y por defecto de la clase.

`+ runAlgorithm (Graph& graph, Chrono& chrono)` : Ejecuta el algoritmo AnotherGreedy con el que se busca la solución al problema (ver pseudocódigo más adelante).

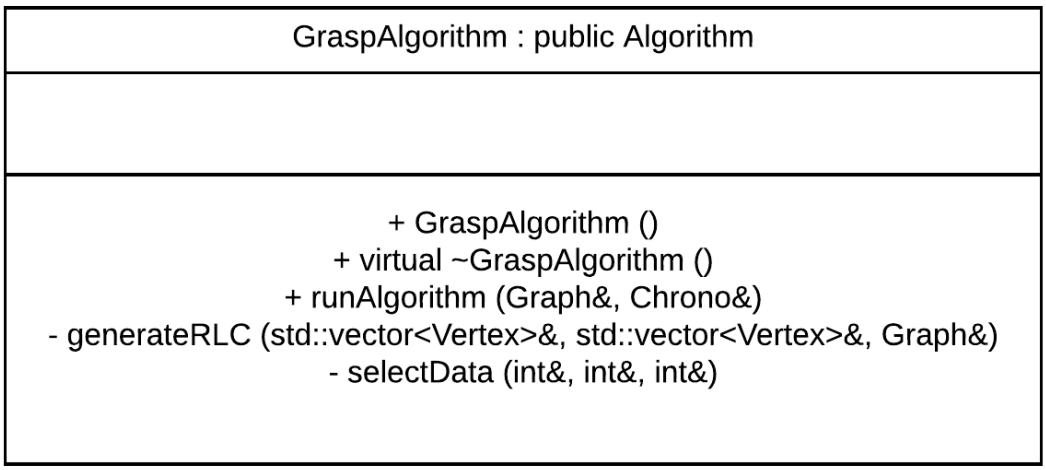
Pseudocódigo runAlgorithm

```
iniciar chrono
    S = solucion inicial (arista con mayor valor)
    repetir
        si i no está en solucion entonces
            S' = S ∪ i
            si md(S') > md(S) entonces:
                S = S'
    mientras (i < cantidad de vértices en grafo)
    guardar solución
    guardar media
    parar chrono
```

2.9 Class GraspAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica grasp. Esta clase carece de atributos ya que los hereda de la clase padre. Para este algoritmo necesitamos que el usuario nos indique por teclado el número de elementos que contendrá la LRC (Lista restringida de candidatos), el número de iteraciones que se ejecutará nuestro algoritmo y el tipo de parada, es decir, si queremos que cada iteración incremente el contador de parada o solo queremos que se incremente en caso de no encontrar una mejora en esa iteración. La solución inicial la generamos de la misma forma que en Greedy, es decir elegimos la arista de nuestro grafo con mayor afinidad e introducimos los dos vértices que la forman en la solución. En este algoritmo una iteración consiste en generar una LRC, extraer de ahí un vértice aleatorio y comprobar si dispersión en la unión de la solución con dicho vértice es mejor que la solución actual o no.

Diagrama UML de la clase GraspAlgorithm



Métodos de la clase GraspAlgorithm

`+ GraspAlgorithm ()`: Constructor vacío por defecto de la clase.

`+ virtual ~GraspAlgorithm ()`: Destructor vacío y por defecto de la calse.

`+ runAlgorithm (Graph& graph, Chrono& chrono)`: Ejecuta el algoritmo Grasp con el que se busca la solución al problema (ver pseudocódigo más adelante).

`- generateRLC (std::vector<Vertex>& RLC, std::vector<Vertex>& solution, Graph& graph, int& RLCSize)`: Genera una lista restringida de candidatos de forma aleatoria del tamaño que haya especificado el usuario en la que se encontrarán posibles vértices que añadir a la solucion.

`- selectData (int& RLCSize, int& iterations, int& stopMode)`: Imprime un menú en el que se le pedirán al usuario que introduzca algunos datos necesarios para la ejecución del algoritmo.

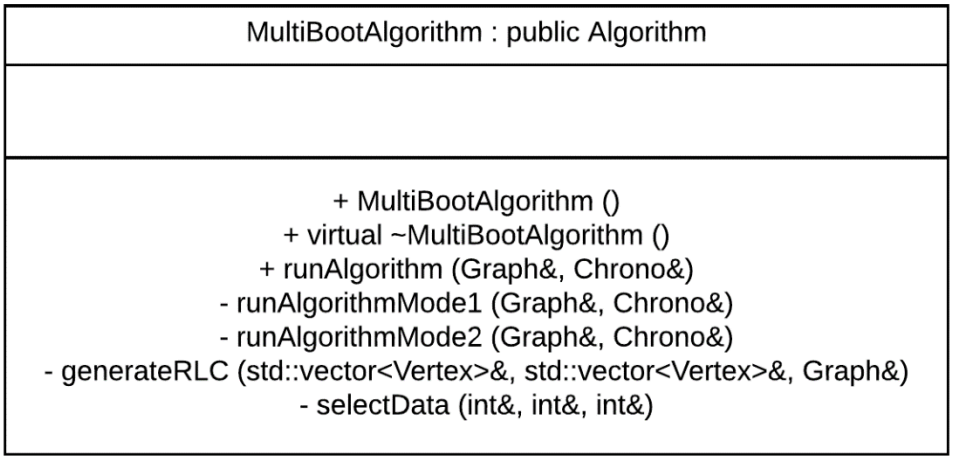
Pseudocódigo runAlgorithm

```
iniciar chrono
S = solucion inicial (arista con mayor valor)
repetir
    generar LRC
    v = vértice aleatorio de LRC
    S' = S ∪ v
    si md(S') > md(S) entonces:
        S = S'
mientras (cont < Iter)
guardar solucion
guardar media
parar chrono
```

2.10 Class MultiBootAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica multi-arranque. Esta clase carece de atributos ya que los hereda de la clase padre. Este algoritmo tenemos 2 modos de resolución, en el modo 1 generamos una solución inicial aleatoria (elegimos dos vértices completamente al azar) y aplicamos la búsqueda local de Grasp. En el modo 2 cogemos la misma solución inicial que para Grasp e implementamos una búsqueda local distinta a la de Grasp. Para este algoritmo necesitamos que el usuario nos indique por teclado el número de elementos que contendrá la LRC (Lista restringida de candidatos) dependiendo del modo elegido, el número de iteraciones que se ejecutará nuestro algoritmo y el tipo de parada, es decir, si queremos que cada iteración incremente el contador de parada o solo queremos que se incremente en caso de no encontrar una mejora en esa iteración. En este algoritmo en el modo 1 una iteración consiste en generar una LRC, extraer de ahí un vértice aleatorio y comprobar si dispersión en la unión de la solucion con dicho vértice es mejor que la solución actual o no. En el modo 2 una iteración consiste en elegir un vértice aleatorio que no pertenezca a la solución y probar a introducirlo en la solución, si la dispersión media mejora lo dejamos, en caso contrario no lo introducimos.

Diagrama UML clase MultiBootAlgorithm



Métodos de la clase MultiBootAlgorithm

`+ MultiBootAlgorithm ()`: Constructor vacío por defecto de la clase.

`+ virtual ~MultiBootAlgorithm ()`: Destructor vacío y por defecto de la clase.

`+ runAlgorithm (Graph& graph, Chrono& chrono)`: Ejecuta el algoritmo MultiBoot con el que se busca la solución al problema (ver pseudocódigo más adelante).

`- runAlgorithmMode1 (Graph& graph, Chrono& chrono)`: Ejecuta el algoritmo en modo 1.

- `runAlgorithmMode2 (Graph& graph, Chrono& chrono)`: Ejecuta el algoritmo en modo 2.

- `generateRLC (std::vector<Vertex>& RLC, std::vector<Vertex>& solution, Graph& graph, int& RLCSize)`: Genera una lista restringida de candidatos de forma aleatoria del tamaño que haya especificado el usuario en la que se encontrarán posibles vértices que añadir a la solucion.

- `selectData (int& RLCSize, int& iterations, int& stopMode)`: Imprime un menú en el que se le pedirán al usuario que introduzca algunos datos necesarios para la ejecución del algoritmo.

Pseudocódigo runAlgorithm1

```
iniciar chrono
S = solución aleatoria inicial
repetir
    generar LRC
    v = vértice aleatorio de LRC
    S' = S U v
    si md(S') > md(S) entonces:
        S = S'
mientras (cont < Iter)
guardar solución
guardar media
parar chrono
```

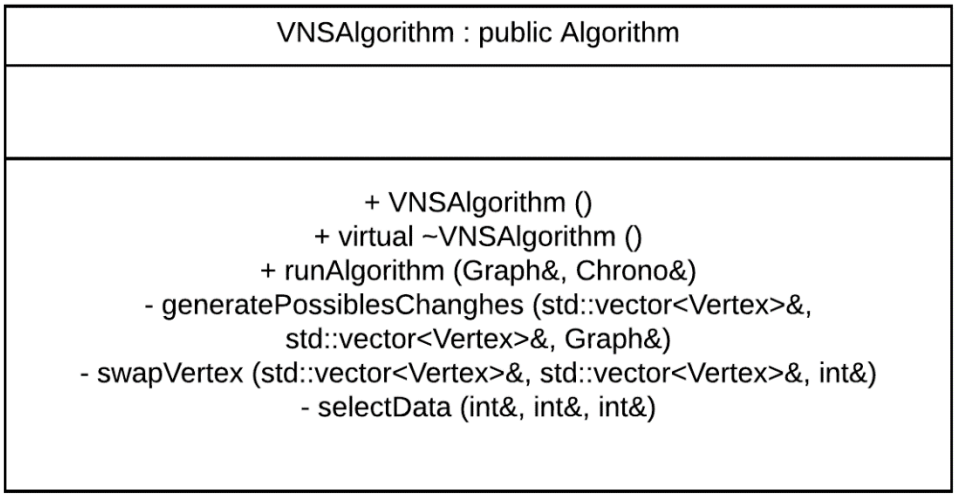
Pseudocódigo runAlgorithm2

```
iniciar chrono
S = solucion inicial (arista con mayor afinidad)
repetir
    k = vértice aleatorio no perteneciente a S
    S' = S U k
    si md(S') > md(S) entonces:
        S = S'
mientras (cont < Iter)
guardar solución
guardar media
parar chrono
```

2.11 Class VNSAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica VNS. Esta clase carece de atributos ya que los hereda de la clase padre. Para este algoritmo es necesario que el usuario nos indique el modo de parada y el número de iteraciones, pero además tendrá que indicar la forma que la que calculamos la solución inicial, es decir, VNS recibe como solución inicial la solución que nos da Grasp y a esta le vamos aplicando las búsquedas de entorno variables para intentar mejorar dicha solución. Entonces, como solución inicial tenemos la que nos dé el algoritmo de Grasp definido previamente. Para este algoritmo una iteración consiste en ir buscando soluciones vecinas dentro de un entorno que va variando el tamaño hasta encontrar soluciones que vayan mejorando nuestra solución final, es decir una iteración consistirá en generar todas las posibles soluciones vecinas dentro de su entorno y escoger una aleatoria, en caso de que sea mejor la actualizamos, en caso contrario no. Además, en caso de que no se actualice la solución aumentamos el entorno de la solución generando un entorno mayor para tener más posibilidades de encontrar una mejor solución de manera aleatoria si nos alejásemos más de la solución actual.

Diagrama UML de la clase VNSAlgorithm



Métodos de la clase VNSAlgorithm

`+ VNSAlgorithm ()`: Constructor vacío por defecto de la clase.

`+ virtual ~VNSAlgorithm ()`: Destructor vacío y por defecto de la clase.

`+ runAlgorithm (Graph& graph, Chrono& chrono)`: Ejecuta el algoritmo VNS con el que se busca la solución al problema (ver pseudocódigo más adelante).

`- genertePossiblesChanges (std::vector<Vertex>& possibilities, std::vector<Vertex>& solution, Graph& graph)`: Genera todas las posibles soluciones dentro del entorno.

- `swapVertex (std::vector<Vertex>& possibilities, std::vector<Vertex>& tempSolution, int& swapsNum)`: Intercambia un número de vértices de la solución con los posibles generados anteriormente para elegir de forma aleatoria así una solución perteneciente al entorno aleatoria.

- `selectData (int& RLCSize, int& iterations, int& stopMode)`: Imprime un menú en el que se le pedirán al usuario que introduzca algunos datos necesarios para la ejecución del algoritmo.

Pseudocódigo runAlgorithm

```
iniciar chrono
S = solucion inicial de Grasp
repetir
    p = posibles soluciones de entorno
    S' = elección de solucion aleatoria de p
    si md(S') > md(S) entonces:
        S = S'
        entorno = 1
    si no
        entorno++;
mientras (cont < Iter)
guardar solucion
guardar media
parar chrono
```

3. Tablas de pruebas

En este apartado hemos realizado diferentes pruebas probando distintos grafos y variando los parámetros introducidos a cada algoritmo en los casos que es posible para comparar las dispersiones medias de los distintos algoritmos en función de los parámetros introducidos por el usuario.

Para todos los algoritmos recogemos mínimo los siguientes datos:

- Problema / P → Identificador del problema.
- N → Número de vértices que contiene el grafo.
- Ejec → Número de ejecución del problema.
- MD → Dispersión media de la solución.
- CPU → Tiempo de ejecución del algoritmo en segundos.
- Solución → Conjunto de vértices que componen la solución al problema.

Para algunos algoritmos como Grasp, VNS o Multi-Arranque también podemos encontrar los siguientes datos en las tablas:

- Dif → Diferencia de la MD con la MD media entre todas las ejecuciones del algoritmo para ese problema.
- Stop → Método de parada (Nº iteraciones / Sin mejora), indica si el método de parada empleado en el algoritmo ha sido un número total de iteraciones (definidas en cada algoritmo) o un número de iteraciones en las que no haya habido mejora de la solución.
- LRC → Tamaño de la Lista Restringida de Candidatos.
- Iter → Número de iteraciones ejecutadas en el algoritmo.
- Modo → En el caso del algoritmo Multi-Arranque el modo de algoritmo empleado.
- K-max → Número máximo de entornos empleado en el algoritmo VNS.

3.1 Algoritmo Greedy

Problema	N	Ejec	MD	CPU	Solución
P1	10	1	10.1429	0.06074	0, 2, 4, 6, 7, 5, 8
P1	10	2	10.1429	0.05635	0, 2, 4, 6, 7, 5, 8
P1	10	3	10.1429	0.05707	0, 2, 4, 6, 7, 5, 8
P1	10	4	10.1429	0.05895	0, 2, 4, 6, 7, 5, 8
P1	10	5	10.1429	0.05516	0, 2, 4, 6, 7, 5, 8
P2	15	1	9.5	0.20248	1, 8, 10, 3
P2	15	2	9.5	0.21832	1, 8, 10, 3
P2	15	3	9.5	0.20785	1, 8, 10, 3
P2	15	4	9.5	0.20628	1, 8, 10, 3
P2	15	5	9.5	0.18586	1, 8, 10, 3
P3	20	1	12.8571	1.61066	0, 3, 18, 11, 8, 19, 7
P3	20	2	12.8571	1.72308	0, 3, 18, 11, 8, 19, 7
P3	20	3	12.8571	1.62641	0, 3, 18, 11, 8, 19, 7
P3	20	4	12.8571	1.61059	0, 3, 18, 11, 8, 19, 7
P3	20	5	12.8571	1.58957	0, 3, 18, 11, 8, 19, 7

3.2 Algoritmo Greedy 2

Problema	N	Ejec	MD	CPU	Solución
P1	10	1	9.57143	0.02682	0, 2, 4, 6, 7, 8, 9
P1	10	2	9.57143	0.02524	0, 2, 4, 6, 7, 8, 9
P1	10	3	9.57143	0.02671	0, 2, 4, 6, 7, 8, 9
P1	10	4	9.57143	0.02431	0, 2, 4, 6, 7, 8, 9
P1	10	5	9.57143	0.03055	0, 2, 4, 6, 7, 8, 9
P2	15	1	9	0.17096	1, 8, 6, 7
P2	15	2	9	0.17037	1, 8, 6, 7
P2	15	3	9	0.17309	1, 8, 6, 7
P2	15	4	9	0.17237	1, 8, 6, 7
P2	15	5	9	0.16061	1, 8, 6, 7
P3	20	1	8	0.91878	0, 3, 6, 11, 12, 18, 19
P3	20	2	8	1.05168	0, 3, 6, 11, 12, 18, 19
P3	20	3	8	0.98417	0, 3, 6, 11, 12, 18, 19
P3	20	4	8	0.93309	0, 3, 6, 11, 12, 18, 19
P3	20	5	8	1.00533	0, 3, 6, 11, 12, 18, 19

3.3 Algoritmo Grasp

P	N	Ejec	MD	CPU	Dif	Stop	LRC	Iter	Solución
P1	10	1	10.1429	0.29637	0.04645	Nº iteraciones	2	50	0, 2, 4, 6, 7, 8, 5
P1	10	2	10.125	0.20256	0.02855	Nº iteraciones	2	30	0, 2, 4, 6, 7, 8, 9, 5
P1	10	3	10.1429	0.28724	0.04645	Sin mejora	2	50	0, 2, 4, 6, 7, 5, 8
P1	10	4	10	0.32759	0.09645	Sin mejora	2	50	0, 2, 4, 6, 7, 8, 3, 5
P1	10	5	10	0.20113	0.09645	Sin mejora	2	30	0, 2, 4, 6, 7, 8, 3, 5
P1	10	6	10.1429	0.29163	0.04645	Sin mejora	3	50	0, 2, 4, 6, 7, 5, 8
P1	10	7	10.125	0.54217	0.02855	Sin mejora	3	80	0, 2, 4, 6, 7, 8, 9, 5
P1	10	8	10.1429	0.18323	0.04645	Nº iteraciones	3	50	0, 2, 4, 6, 7, 5, 8
P1	10	9	10.1429	0.26482	0.04645	Nº iteraciones	3	50	0, 2, 4, 6, 7, 5, 8
P1	10	10	10	0.51098	0.09645	Sin mejora	3	80	0, 2, 4, 6, 7, 8, 3, 5
P2	15	1	9.83333	1.24697	0.383331	Nº iteraciones	2	50	1, 8, 6, 10, 7, 3
P2	15	2	9	0.5034	0.449999	Nº iteraciones	2	30	1, 8, 6, 7
P2	15	3	9.5	0.97406	0.050001	Sin mejora	2	50	1, 8, 10, 3
P2	15	4	9	1.05551	0.449999	Sin mejora	2	50	1, 8, 7, 6
P2	15	5	9.83333	2.63303	0.383331	Sin mejora	2	100	1, 8, 6, 10, 7, 3
P2	15	6	9.5	0.81413	0.050001	Nº iteraciones	3	50	1, 8, 10, 3
P2	15	7	9.83333	2.04864	0.383331	Nº iteraciones	3	80	1, 8, 7, 10, 3, 6
P2	15	8	9.5	0.86588	0.050001	Sin mejora	3	50	1, 8, 10, 3
P2	15	9	9	0.96005	0.449999	Sin mejora	3	50	1, 8, 7, 6
P2	15	10	9.5	1.91368	0.050001	Sin mejora	3	100	1, 8, 10, 3
P3	20	1	12.8571	3.84663	0.53259	Nº iteraciones	2	50	0, 3, 11, 8, 19, 18, 7
P3	20	2	12.8571	6.65077	0.53259	Nº iteraciones	2	80	0, 3, 8, 18, 11, 19, 7
P3	20	3	11.6667	5.28228	0.65781	Sin mejora	2	50	0, 3, 8, 7, 14, 11, 18, 19, 17
P3	20	4	12	3.60231	0.32451	Sin mejora	2	50	0, 3, 12, 18, 11
P3	20	5	11.75	11.0672	0.57451	Sin mejora	2	100	0, 3, 13, 11, 18, 1, 19, 7
P3	20	6	12.8571	3.62671	0.53259	Nº iteraciones	3	50	0, 3, 18, 11, 8, 19, 7
P3	20	7	12	4.26181	0.32451	Nº iteraciones	3	80	0, 3, 18, 11, 12
P3	20	8	12	3.57714	0.32451	Sin mejora	3	50	0, 3, 12, 18, 11
P3	20	9	12.4	3.68149	0.07549	Sin mejora	3	50	0, 3, 11, 18, 8
P3	20	10	12.8571	8.58934	0.53259	Sin mejora	3	100	0, 3, 11, 19, 18, 8, 7

3.4 Algoritmo Multi-Arranque

Para las pruebas de este algoritmo en modo 1 hemos empleado la LRC de tamaño 2.

P	N	Ejec	MD	CPU	Dif	Stop	Modo	Iter	Solución
P1	10	1	11.7143	0.30472	0.5447	Nº iteraciones	1	50	0, 9, 7, 4, 6, 8, 5
P1	10	2	10.875	0.38168	0.2946	Sin mejora	1	50	1, 3, 4, 7, 9, 6, 8, 5
P1	10	3	14	0.17127	2.8304	Sin mejora	1	30	6, 7, 4, 5, 9, 8
P1	10	4	10.1429	0.08855	1.0267	Nº iteraciones	2	50	0, 2, 4, 6, 7, 5, 8
P1	10	5	10.1429	0.10917	1.0267	Sin mejora	2	50	0, 2, 4, 6, 7, 5, 8
P1	10	6	10.1429	0.06795	1.0267	Sin mejora	2	30	0, 2, 4, 6, 7, 8, 5
P2	15	1	8.25	1.11952	0.3382	Nº iteraciones	1	50	6, 8, 7, 11, 4, 1, 10, 3
P2	15	2	7.375	1.67512	1.2132	Sin mejora	1	50	5, 11, 10, 3, 6, 1, 8, 7
P2	15	3	7.57143	0.99409	1.01677	Sin mejora	1	30	13, 7, 8, 1, 14, 3, 6
P2	15	4	9	0.63842	0.4118	Nº iteraciones	2	50	1, 8, 7, 6
P2	15	5	9.83333	0.68582	1.24513	Sin mejora	2	50	1, 8, 7, 3, 6, 10
P2	15	6	9.5	0.37405	0.9118	Sin mejora	2	30	1, 8, 10, 3
P3	20	1	11.75	4.60335	0.0183	Nº iteraciones	1	50	0, 11, 13, 18, 3, 19, 1, 7
P3	20	2	9	7.43353	2.7317	Sin mejora	1	50	14, 11, 19, 16, 7, 4, 9, 8, 0, 17, 18
P3	20	3	12.8571	2.76075	1.1254	Sin mejora	1	30	3, 0, 11, 8, 19, 18, 7
P3	20	4	12.8333	2.76605	1.1016	Nº iteraciones	2	50	0, 3, 11, 18, 19, 8
P3	20	5	11.75	3.19921	0.0183	Sin mejora	2	50	0, 3, 13, 18, 7, 19, 11, 1
P3	20	6	12.2	1.55573	0.4683	Sin mejora	2	30	0, 3, 19, 18, 11

3.5 Algoritmo VNS

Para las pruebas de este algoritmo hemos empleado en Grasp una LRC de tamaño 2, como criterio de parada un nº global de iteraciones y 50 iteraciones.

P	N	Ejec	MD	CPU	Dif	Stop	k-max	Iter	Solución
P1	10	1	12.7143	0.50192	0.7858	Nº iteraciones	2	50	4, 6, 7, 5, 8, 9, 3
P1	10	2	11.2857	0.50269	0.6428	Sin mejora	2	50	0, 4, 6, 7, 8, 3, 5
P1	10	3	11.7143	0.51649	0.2142	Sin mejora	2	30	0, 4, 6, 7, 5, 8, 9
P1	10	4	13	3.87744	1.0715	Nº iteraciones	3	50	6, 7, 5, 8, 3, 4
P1	10	5	11	5.15366	0.9285	Sin mejora	3	50	0, 4, 6, 7, 8, 5, 9, 3
P1	10	6	11.8571	1.67239	0.0714	Sin mejora	3	30	4, 6, 7, 8, 5, 9, 1
P2	15	1	9	1.61878	0.3472	Nº iteraciones	2	50	1, 8, 6, 7
P2	15	2	9	1.73752	0.3472	Sin mejora	2	50	1, 8, 6, 7
P2	15	3	9.75	3.10822	0.4028	Sin mejora	2	100	8, 11, 4, 10
P2	15	4	9.5	2.91676	0.1528	Nº iteraciones	3	50	1, 8, 10, 3
P2	15	5	9	3.10327	0.3472	Sin mejora	3	50	1, 8, 6, 7
P2	15	6	9.83333	4.32261	0.48613	Sin mejora	3	100	1, 8, 6, 10, 3, 7
P3	20	1	12.8571	8.35793	0.5802	Nº iteraciones	2	50	0, 3, 19, 11, 18, 8, 7
P3	20	2	12.2	8.37576	0.0769	Sin mejora	2	50	0, 3, 18, 11, 19
P3	20	3	11.5714	17.0887	0.7055	Sin mejora	2	100	0, 3, 19, 11, 18, 1, 7
P3	20	4	12.2	10.0005	0.0769	Nº iteraciones	3	50	0, 3, 11, 18, 19
P3	20	5	12	7.51605	0.2769	Sin mejora	3	50	0, 3, 18, 11, 12
P3	20	6	12.8333	12.4662	0.5564	Sin mejora	3	100	0, 3, 11, 19, 18, 8