MAXIMUM DIVERSITY PROBLEM

Universidad de La Laguna

Diseño y Análisis de Algoritmos

Este documento representa la solución del MAXIMUM DIVERSITY PROBLEM 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 Maximum Diversity Problem?

En este proyecto se trata el problema de la "Máxima diversidad" (Maximum Diversity ´ Problem (MDP)), uno de los problemas pertenecientes a los problemas del tipo de optimización combinatoria tan útiles en la actualidad.

Maximum Diversity Problem

En el Maximum diversity problem se desea encontrar el subconjunto de elementos de diversidad máxima de un conjunto dado de elementos. Sea dado un conjunto S = {s1, s2, . . . ,sn} de n elementos, en el que cada elemento si es un vector si = (si1,si2, . . . ,siK). Sea, asimismo, dij la distancia entre los elementos i y j. Si m < n es el tamaño del subconjunto que se busca el problema puede formularse como:

$$Max \quad z = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

Sujeto a:

$$\sum_{i=0}^{n-1} x_i = m$$

$$x_i \epsilon \{0,1\} \qquad i = 1, 2, \dots, n$$

Donde:

$$x_i = \begin{cases} 1 & si\ s_i \in\ a\ la\ ecuación \\ 0 & en\ caso\ contrario \end{cases}$$

La distancia dij depende de la aplicación real considerada. En muchas aplicaciones se usa la distancia euclídea. Así:

$$d_{ij} = d(s_i, s_j) = \sqrt{\sum_{r=1}^{K} (s_{ir} - s_{jr})^2}$$

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.

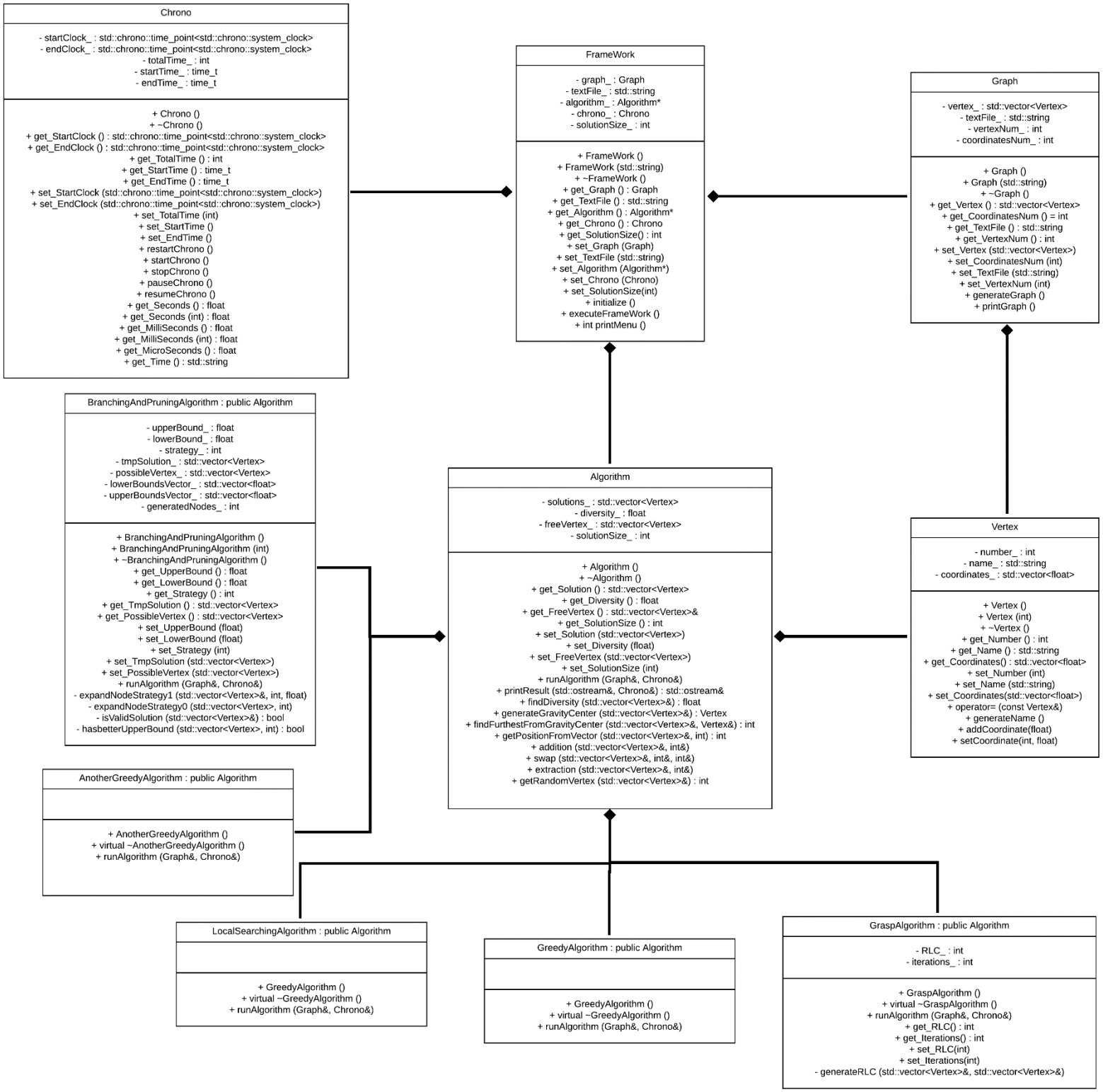
- : Expresa que el atributo/método es privado.

: 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, LocalSearch y BranchingAndPruning), 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.

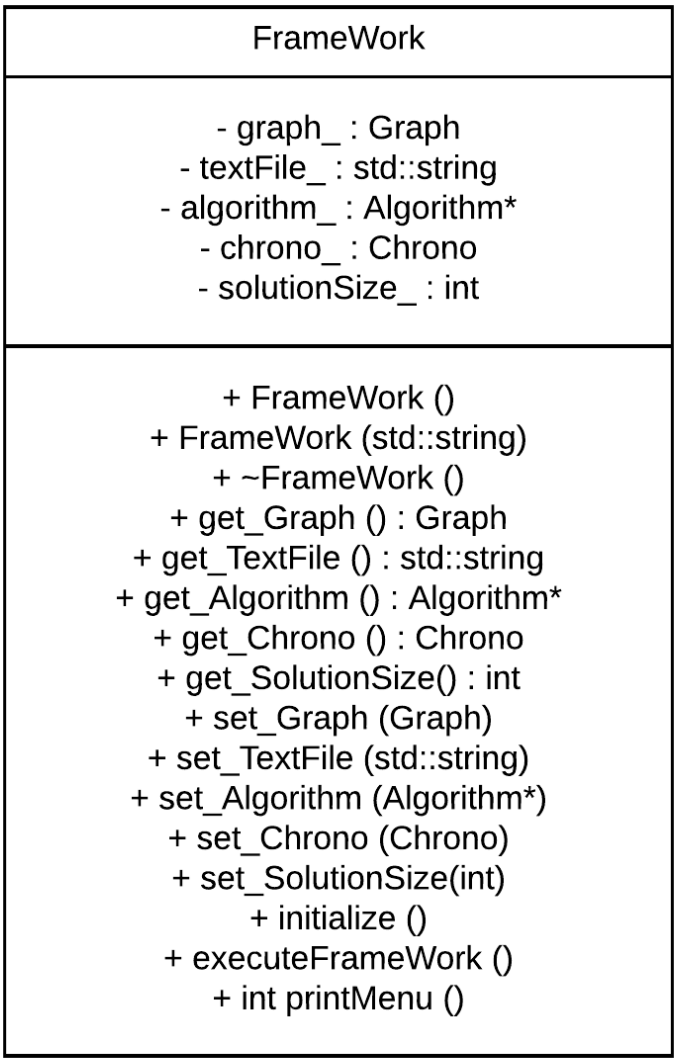
En el caso de las clases heredadas de la clase Algorithm, es decir en los distintos tipos de algoritmos de resolución en el resumen de la clase se incluirá que entendemos por una iteración en dicho algoritmo, cómo se obtiene la solución inicial y se explicará brevemente el algoritmo de búsqueda local 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
- solutionSize_ : int
+ FrameWork ()
+ FrameWork (std::string)
+ ~FrameWork ()
+ get_Graph () : Graph
+ get_TextFile () : std::string
+ get_Algorithm () : Algorithm*
+ get_Chrono () : Chrono
+ get_SolutionSize() : int
+ set_Graph (Graph)
+ set_TextFile (std::string)
+ set_Algorithm (Algorithm*)
+ set_Chrono (Chrono)
+ set_SolutionSize(int)
+ initialize ()
+ executeFrameWork ()
+ int printMenu ()
Graph
- vertex_ : std::vector<Vertex>
- textFile_ : std::string
- vertexNum_ : int
- coordinatesNum_ : int
+ Graph ()
+ Graph (std::string)
+ ~Graph ()
+ get_Vertex () : std::vector<Vertex>
+ get_CoordinatesNum () = int
+ get_TextFile () : std::string
+ get_VertexNum () : int
+ set_Vertex (std::vector<Vertex>)
+ set_CoordinatesNum (int)
+ set_TextFile (std::string)
+ set_VertexNum (int)
+ generateGraph ()
+ printGraph ()
BranchingAndPruningAlgorithm : public Algorithm
- upperBound_ : float
- lowerBound_ : float
- strategy_ : int
- tmpSolution_ : std::vector<Vertex>
- possibleVertex_ : std::vector<Vertex>
- lowerBoundsVector_ : std::vector<float>
- upperBoundsVector_ : std::vector<float>
- generatedNodes_ : int
+ BranchingAndPruningAlgorithm ()
+ BranchingAndPruningAlgorithm (int)
+ ~BranchingAndPruningAlgorithm ()
+ get_UpperBound () : float
+ get_LowerBound () : float
+ get_Strategy () : int
+ get_TmpSolution () : std::vector<Vertex>
+ get_PossibleVertex () : std::vector<Vertex>
+ set_UpperBound (float)
+ set_LowerBound (float)
+ set_Strategy (int)
+ set_TmpSolution (std::vector<Vertex>)
+ set_PossibleVertex (std::vector<Vertex>)
+ runAlgorithm (Graph&, Chrono&)
- expandNodeStrategy1 (std::vector<Vertex>&, int, float)
- expandNodeStrategy0 (std::vector<Vertex>, int)
- isValidSolution (std::vector<Vertex>&) : bool
- hasbetterUpperBound (std::vector<Vertex>, int) : bool
Algorithm
- solutions_ : std::vector<Vertex>
- diversity_ : float
- freeVertex_ : std::vector<Vertex>
- solutionSize_ : int
+ Algorithm ()
+ ~Algorithm ()
+ get_Solution () : std::vector<Vertex>
+ get_Diversity () : float
+ get_FreeVertex () : std::vector<Vertex>&
+ get_SolutionSize () : int
+ set_Solution (std::vector<Vertex>)
+ set_Diversity (float)
+ set_FreeVertex (std::vector<Vertex>)
+ set_SolutionSize (int)
+ runAlgorithm (Graph&, Chrono&)
+ printResult (std::ostream&, Chrono&) : std::ostream&
+ findDiversity (std::vector<Vertex>&) : float
+ generateGravityCenter (std::vector<Vertex>&) : Vertex
+ findFurthestFromGravityCenter (std::vector<Vertex>&, Vertex&) : int
+ getPositionFromVector (std::vector<Vertex>&, int) : int
+ addition (std::vector<Vertex>&, int&)
+ swap (std::vector<Vertex>&, int&, int&)
+ extraction (std::vector<Vertex>&, int&)
+ getRandomVertex (std::vector<Vertex>&) : int
Vertex
- number_ : int
- name_ : std::string
- coordinates_ : std::vector<float>
+ Vertex ()
+ Vertex (int)
+ ~Vertex ()
+ get_Number () : int
+ get_Name () : std::string
+ get_Coordinates() : std::vector<float>
+ set_Number (int)
+ set_Name (std::string)
+ set_Coordinates(std::vector<float>)
+ operator= (const Vertex&)
+ generateName ()
+ addCoordinate(float)
+ setCoordinate(int, float)
AnotherGreedyAlgorithm : public Algorithm
+ AnotherGreedyAlgorithm ()
+ virtual ~AnotherGreedyAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
LocalSearchingAlgorithm : public Algorithm
+ GreedyAlgorithm ()
+ virtual ~GreedyAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
GreedyAlgorithm : public Algorithm
+ GreedyAlgorithm ()
+ virtual ~GreedyAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
GraspAlgorithm : public Algorithm
- RLC_ : int
- iterations_ : int
+ GraspAlgorithm ()
+ virtual ~GraspAlgorithm ()
+ runAlgorithm (Graph&, Chrono&)
+ get_RLC() : int
+ get_Iterations() : int
+ set_RLC(int)
+ set_Iterations(int)
- generateRLC (std::vector<Vertex>&, std::vector<Vertex>&)

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.

- `solutionSize_`: Entero que almacena el número de elementos que contendrá la solución final de nuestro 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_*.

`+ get_SolutionSize ()`: Método que devuelve el atributo solutionSize_.

`+ 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_*.

`+ set_SolutionSize (int chrono)`: Método que establece el atributo *solutionSize_*.

`+ 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.

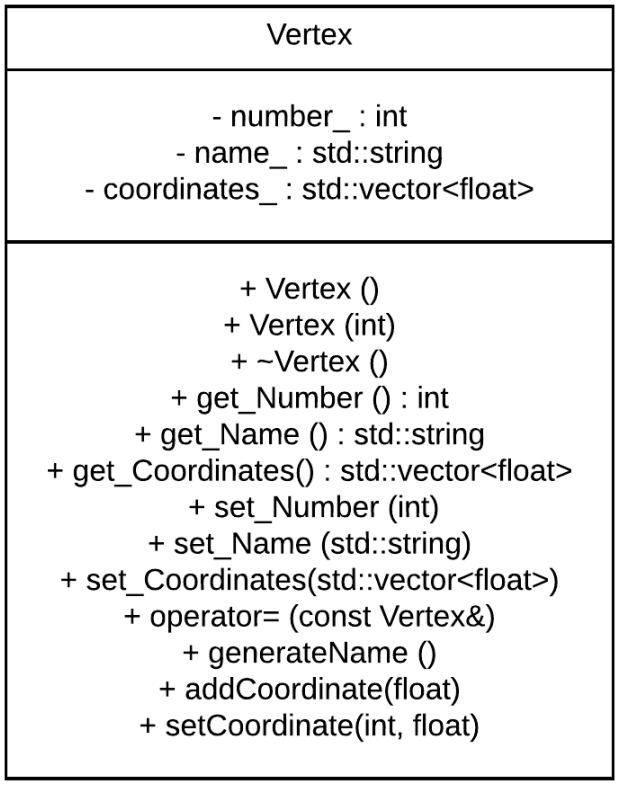
`+ executeFrameWork ()`: Método que se encarga de llamar a la ejecución del algoritmo, contabilizar el tiempo y llamar a la impresión del tiempo y los resultados del algoritmo

`+ 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. Cada vértice además de almacenar su nombre y número identificador contendrá un vector de números flotantes correspondientes a las coordenadas del vértice en el espacio.

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.

- `coordinates_`: Vector de coordenadas del vértice en el espacio (x, y, z, t, s…).

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_*.

+ `get_Coordinates ()`: Método que devuelve el atributo *coordinates_*.

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

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

+ `set_Coordinates (std::vector<float> name)`: Método que establece el atributo *coordinates_*.

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

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

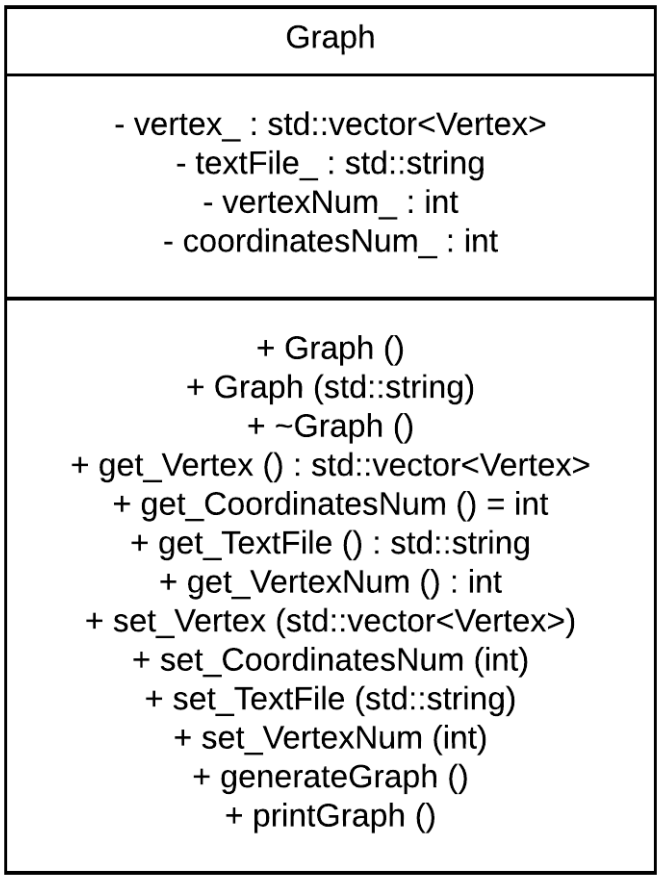
+ `addCoordinate (float coordinate)`: Método que añade una coordenada al final del vector de coordenadas del vértice.

+ `setCoordinate (int position float coordinate)`: Método que establece una coordenada en la posición del vector de coordenadas dada del vértice.

2.3 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.
- `coordinatesNum_`: Número de coordenadas que posee cada vértice.

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_.

`+ get_Coordinates ()`: Método de la clase que devuelve el atributo coordinatesNum_.

`+ 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_.

`+ set_CoordinatesNum (int coordinatesNum)`: Método de la clase que establece el atributo coordinatesNum_.

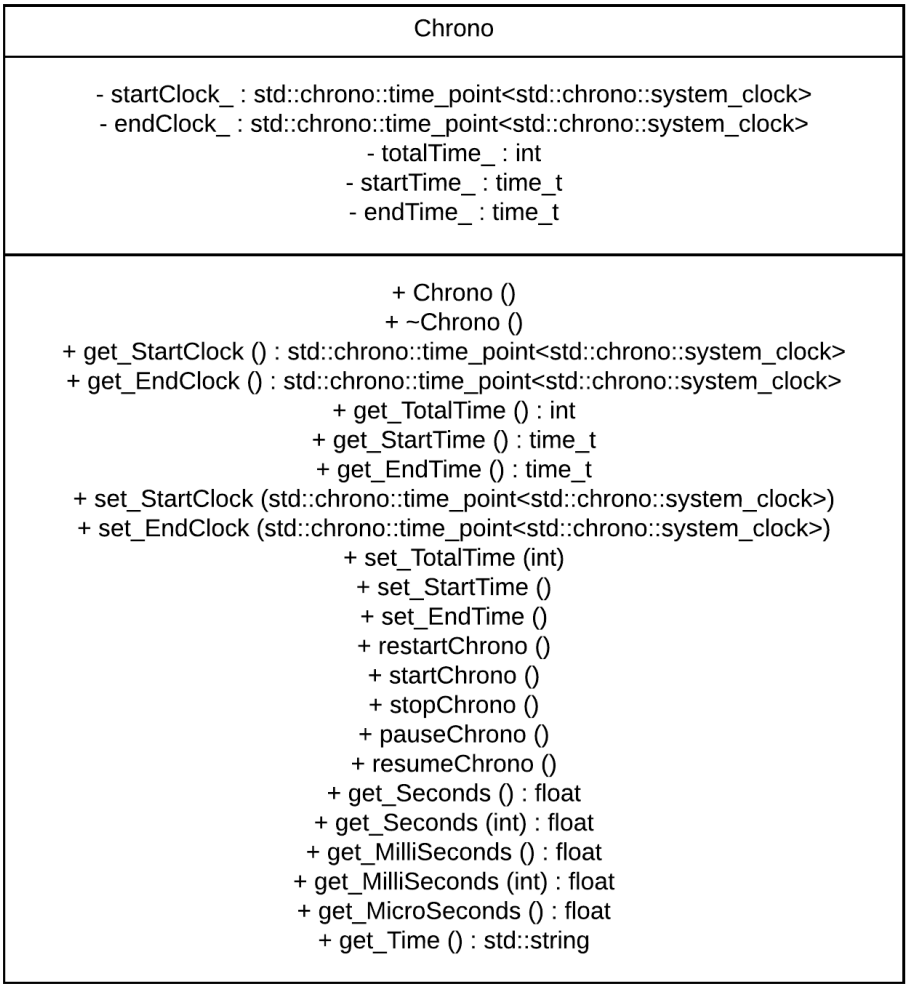
`+ generateGraph ()`: Genera el grafo leyendo los datos del fichero de entrada.

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

2.4 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.

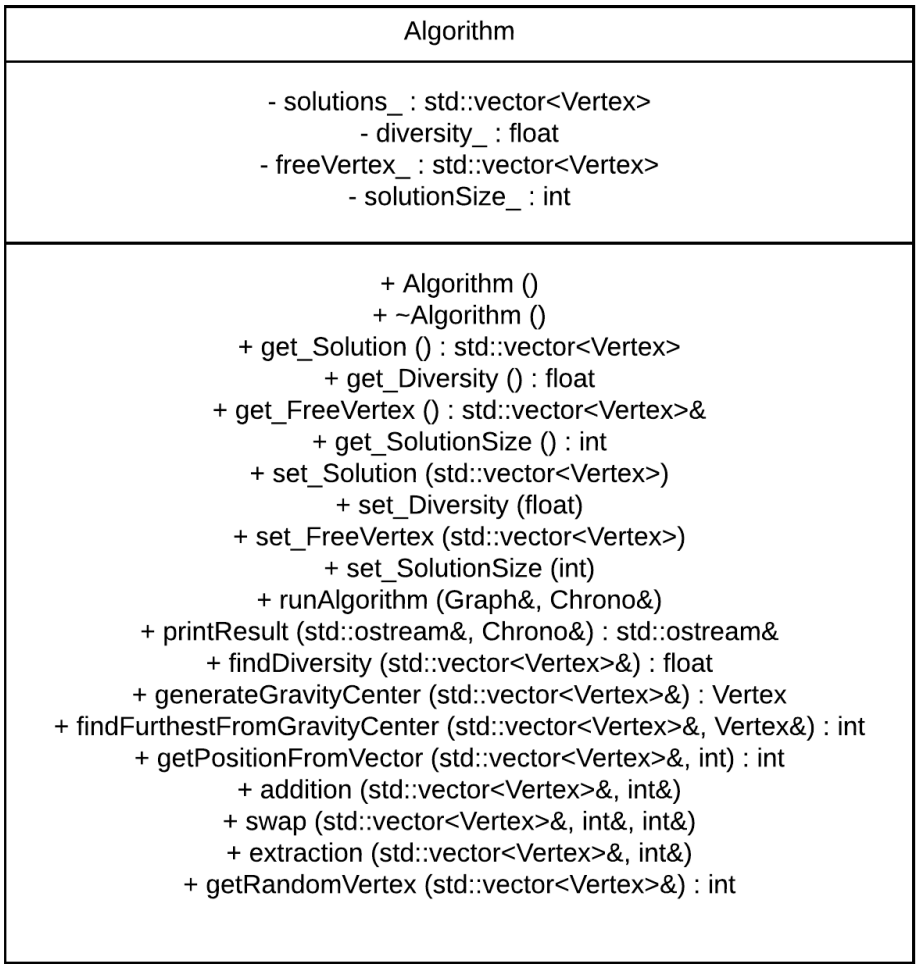
+ `get_MicroSeconds ()`: Devuelve el tiempo en microsegundos.

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

2.5 Class Algorithm

Esta clase será la clase padre de los algoritmos de búsqueda que emplearemos "Greedy, AnotherGreedy, Grasp, LocalSearch y BranchingAndPruning". En ella 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 diversidad de un conjunto de vértices.

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.

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

- `freeVertex_`: Vector de vértices externos a la solución el que se almacenarán los vértices que en cada momento no pertenecen ni a la solución global ni a la temporal.

- `solutionSize_`: Tamaño del vector de soluciones de nuestro algoritmo, es decir, número de elementos que contendrá la solución que nos deben proporcionar los algoritmos de búsqueda.

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_Diversity ()`: Devuelve el atributo *diversity_* de la clase.

`+ get_FreeVertex ()`: Devuelve el atributo *freeVertex_* de la clase.

`+ get_SolutionSize ()`: Devuelve el atributo *solutionSize_* de la clase.

`+ set_Solution (std::vector<Vertex>)`: Establece el atributo *solutions_* de la clase.

`+ set_Diversity (float)`: Establece el atributo *diversity_* de la clase.

`+ set_FreeVertex (std::vector<Vertex>)`: Establece el atributo *freeVertex_* de la clase.

`+ set_SolutionSize (int)`: Establece el atributo *solutionSize_* 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.

`+ findDiversity (std::vector<Vertex>& vertex)`: Calcula la diversidad entre los vértices pertenecientes al vector que recibe como parámetro.

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

`+ generateGravityCenter(std::vector<Vertex>& vector)`: Devuelve el punto que equivale al centro de gravedad entre los vértices pertenecientes al vector pasado como parametro.

`+ findFurthestDromGravityCenter(std::vector<Vertex>& vector, Vertex& vertex)`: Devuelve el vértice perteneciente al vector de vértices más alejado del punto de gravedad "vertex".

+ `getPositionFromVertex (std::vector<Vertex>& vector, int vertexNum)`: Devuelve la posición asignada al vértice perteneciente al vector con identificador = "vertexNum".

+ `addition (std::vector<Vertex>& vertex, int& vertexNum)`: Genera estructura de entorno de unión de un vértice no perteneciente a la solución a la solución actual extrayéndolo y eliminándolo del atributo de freeVertex_ y añadiéndolo al vector pasado como parámetro.

+ `swap (std::vector<Vertex>& vertex, int& vertexNum, int& freeVertexNum)`: Genera estructura de entorno de intercambio extrayendo y eliminando del atributo freeVertex_ el vértice "freeVertexNum" y añadiéndolo al vector pasado como parámetro. Después extrae del vector pasado como parámetro el vértice "vertexNum" y lo añade al atributo freeVertex_.

+ `extraction (std::vector<Vertex>& vertex, int& vertexNum)`: Genera estructura de entorno de extracción, extrae del vector pasado como parámetro el vértice "vertexNum" y lo añade al atributo freeVertex_.

+ `get_RamdomVertex (std::vector<Vertex>& vertex)`: Escoge de manera aleatoria un vértice del vector pasado como parámetro y devuelve la posición de dicho vértice en ese vector.

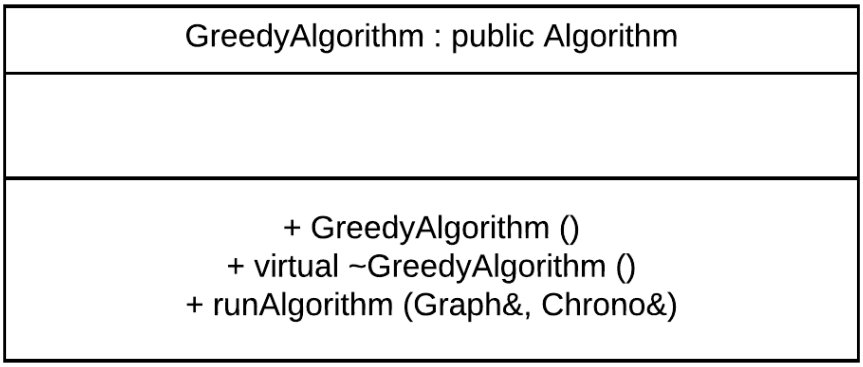
2.6 Class GreedyAlgorithm

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 diversidad de la solución escogiendo el vértice más alejado del centro de gravedad. En este algoritmo la solución inicial se genera escogiendo el vértice más alejado del centro de gravedad generado por el conjunto el grafo completo. Para este algoritmo una iteración consiste en buscar un vértice que añadir nuestro conjunto de nodos en la solución hasta que el tamaño de la solución sea el solicitado por el usuario anteriormente.

El centro de gravedad de un conjunto de elementos X = {si : i ∈ I} con I ⊆ {1, 2, . . . , n} se define como:

$$centro(X) = \frac{1}{|X|}(\sum_{i\in I} s_{i1}\,,\ \sum_{i\in I} s_{i2}\,, \ldots, \sum_{i\in I} s_{iK})$$

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).

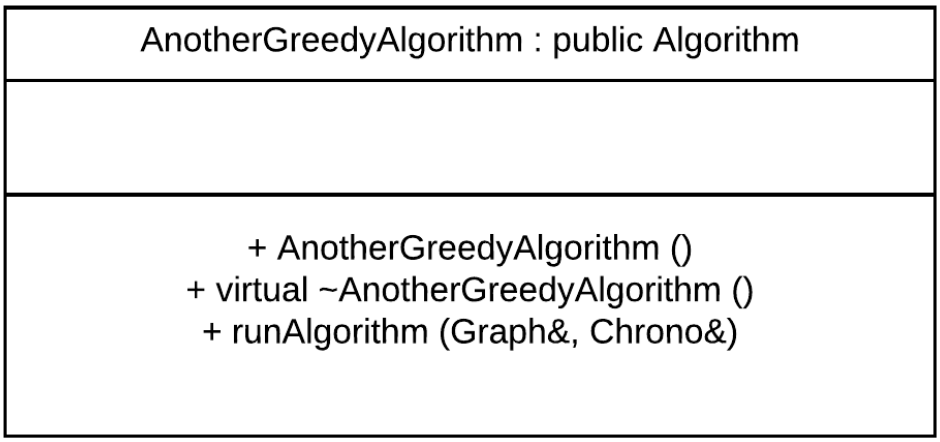
Pseudocódigo runAlgorithm

```
iniciar chrono
    FreeVertex = Conjunto Vértices Grafo
    S = ∅
    Obtener Sc = centro(FreeVertex)
    Repetir
        Obtener S* ∈ a FreeVertex más alejado de Sc
        S = S U {S*}
        FreeVertex = FreeVertex – {S*}
        Obtener Sc = centro(S)
    mientras (|S| < solSize)
    guardar solución
    guardar media
    parar chrono
```

2.7 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. En este algoritmo se introducen los primeros m elementos libres en a la solución y luego se va comprobando si la diversidad de esa solución es mejor o peor que la diversidad intercambiando cada nodo de la solución temporal con cada nodo no perteneciente a la solución actual, en caso de que mejore dicha diversidad realizamos el cambio, en caso contrario seguimos probando con el siguiente nodo. Para este algoritmo, una iteración consiste en ir recorriendo todo el vector de vértices libres y probar a intercambiar un nodo de la solución actual con todos los posibles vértices libres 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).

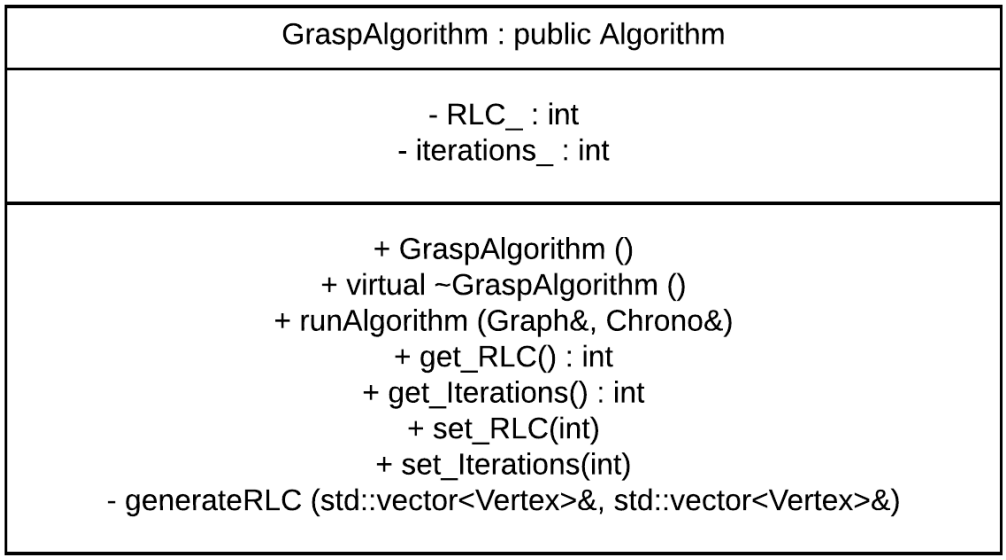
Pseudocódigo runAlgorithm

```
iniciar chrono
FreeVertex = Conjunto Vértices Grafo
Repetir
  S[i] = FreeVertex[i]
  FreeVertex = FreeVertex – {FreeVertex[i]}
Mientras (i < m)
D = diversidad(S)
Repetir
  Repetir
       Tmp = S
       Tmp = tmp – {tmp[i]}
       Tmp = tmp + {FreeVertex[j]}
       D2 = diversidad(tmp)
       Si (D2 > D)
            S[i] <-> FreeVertex[j]
            D = D2
       Fin si
       j++
  Mientras (j < |freeVertex|)
       i++
mientras (i < |S|)
guardar solución
guardar media
parar chrono
```

2.8 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 contiene los atributos "RLC" que corresponde al tamaño de la lista restringida de candidatos e "iteration" que corresponde al número de iteraciones que realizará el algoritmo antes de finalizar. 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. La solución inicial la generamos rellenando la solución con elementos al azar de nuestro conjunto de vértices. Para este algoritmo una iteración consiste en generar una LRC, extraer de ahí un vértice aleatorio y comprobar si dispersión en el intercambio de ese vértice aleatorio extraído de la LRC con un vértice aleatorio de la solución actual mejora la diversidad del conjunto, en caso afirmativo se realiza el intercambio, en caso contrario se deja como está y se repite el proceso.

Diagrama UML de la clase GraspAlgorithm



Atributos de la clase GraspAlgorithm

- `RLC_`: Número de elementos que contendrá la lista restringida de candidatos empleada por el algoritmo.

- `iterations_`: Número de iteraciones que realizará el algoritmo antes de parar.

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).

`+ get_RLC ()` : Devuelve el atributo *RLC_* de la clase.

`+ get_Iterations ()` : Devuelve el atributo *iterations_* de la clase.

`+ set_RLC (int)` : Establece el atributo *RLC_* de la clase.

`+ set_Iterations (int)` : Establece el atributo *iterations_* de la clase.

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

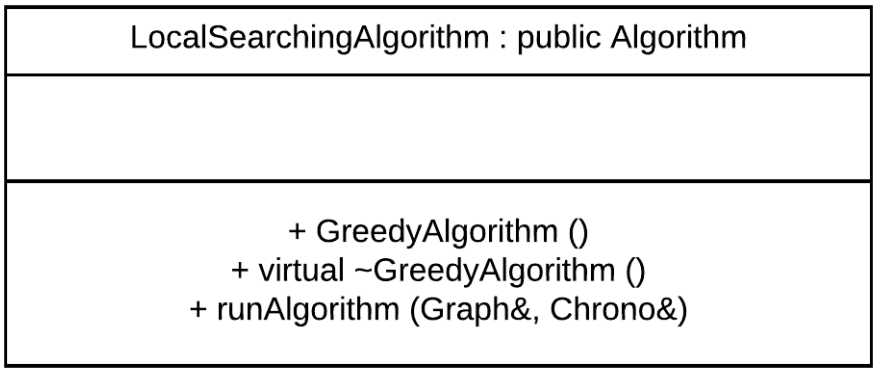
Pseudocódigo runAlgorithm

```
iniciar chrono
FreeVertex = Conjunto Vértices Grafo
Repetir
  X = random(FreeVertex)
  S[i] = FreeVertex[x]
  FreeVertex = FreeVertex – {FreeVertex[x]}
Mientras (i < m)
D = diversidad(S)
Repetir
  Repetir
       Tmp = S
       list = generarRLC
       X = random(S)
       Y = random(list)
       Tmp = tmp + {FreeVertex[y]}
       FreeVertex = FreeVertex + {tmp[x]}
       FreeVertex = FreeVertex – {FreeVertex[y}
       Tmp = tmp – {tmp[x]}
       D2 = diversidad(tmp)
       Si (D2 > D)
            S[x] <-> FreeVertex[y]
            D = D2
       Fin si
mientras (counter < iterations)
guardar solución
guardar media
parar chrono
```

2.9 Class LocalSearchAlgorithm

Clase hija de Algorithm, en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica búsqueda local. Para ello generamos la solución inicial introduciendo los primeros nodos del grafo hasta llegar al tamaño de solución necesario. Luego comprobamos a hacer un intercambio de cada nodo perteneciente a la solución con todas sus vecinas y en caso de que mejore la solución actual la actualizamos, en caso contrario seguimos iterando hasta finalizar la comprobación de todas y cada una de las vecinas de la solución inicial.

Diagrama UML clase LocalSearchAlgorithm



Métodos de la clase LocalSearchAlgorithm

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

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

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

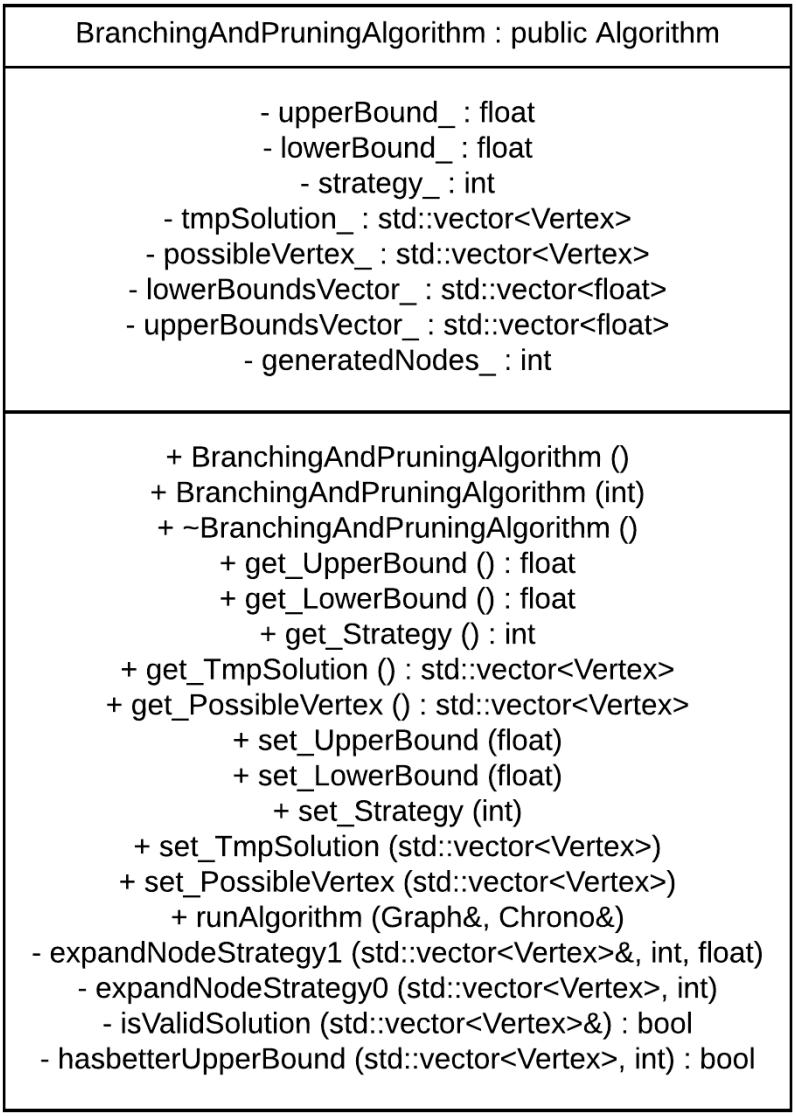
Pseudocódigo runAlgorithm

```
iniciar chrono
FreeVertex = Conjunto Vértices Grafo
Repetir
     S[i] = Freevertex[i]
     FreeVertex = Freevertex – {FreeVertex[i]}
     I++
Mientras (i < |S|)
Repetir
     S* = S
     Repetir
          S*[i] = Freevertex[j]
          Si (Diversidad(S*) > Diversidad(S)
               S[i] = Freevertex[j]
          Fin si
          J++
     Mientras (j < |FreeVertex|)
     I++
Mientras (i < |S|)
```

2.10 Class BranchingAndPruningAlgorithm

Clase hija de Algorithm. en ella se implementa la resolución del algoritmo de búsqueda de la mejor solución mediante una técnica ramificación y poda (Branch and Prune). Los algoritmos Branch and Prune intentan eliminar de la búsqueda las ramas del "árbol" de exploración que se sabe que no van a obtener una mejor solución a la mejor ya existente. La decisión de Podar (eliminar de la exploración una parte del árbol de soluciones) se realiza en base a unas cotas que se calculan durante la exploración y que sirven para indicarnos si vale la pena o no continuar por el camino que se lleva. En caso de no satisfacer los criterios para seguir por la rama de exploración se desecha (poda) y se continúa con la exploración por otra rama válida aun no explorada del árbol.

Diagrama UML de la clase BranchingAndPruningAlgorithm



Atributos de la clase BranchingAndPruningAlgorithm

- `upperBound_`: Flotante que almacena el valor de la cota superior.
- `lowerBound_`: Flotante que almacena el valor de la cota inferior.
- `tmpSolution_`: Vector de nodos que contiene la solución temporal al problema.
- `possibleVertex_`: Vector de nodos que contiene los nodos no pertenecientes en el momento a la solución al problema.
- `lowerBoundsVector_`: Vector de flotantes donde se almacenan las cotas inferiores en cada momento.
- `upperBoundsVector_`: Vector de flotantes donde se almacenan las cotas superiores en cada momento.
- `generatedNodes_`: Número de nodos que expande el algoritmo.

Metodos de la clase BranchingAndPruningAlgorithm

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

`+ BranchingAndPruningAlgorithm (int)`: Constructor parametrizado al que se le pasa la estrategia a utilizar.

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

`+ get_UpperBound ()`: Devuelve el atributo *upperBound_* de la clase.

`+ get_LowerBound ()`: Devuelve el atributo *lowerBound_* de la clase.

`+ get_Strategy ()`: Devuelve el atributo *strategy_* de la clase.

`+ get_TmpSolution ()`: Devuelve el atributo *tmpSolution_* de la clase.

`+ get_PossibleVertex ()`: Devuelve el atributo *possibleVertex_* de la clase.

`+ set_UpperBound (float)`: Establece el atributo *upperBound_* de la clase.

`+ set_LowerBound (float)`: Establece el atributo *lowerBound_* de la clase.

`+ set_Strategy (int)`: Establece el atributo *strategy_* de la clase.

`+ set_TmpSolution (std::vector<Vertex>)`: Establece el atributo *tmpSolution_* de la clase.

`+ set_PossibleVertex (std::vector<Vertex>)`: Establece el atributo *possibleVertex_* 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).

`- expandNodeStrategy1 (std::vector<Vertex>& tmp, int pos, float diversity)`: Calcula la solución final mediante la estrategia1 (expandiendo el nodo mas profundo)

`- expandNodeStrategy0 (std::vector<Vertex>& tmp, int pos)`: Calcula la solución final mediante la estrategia0 (expandiendo el nodo con la cota superior más baja)

`- isValidSolution (std::vector<Vertex>& tmp)`: Comprueba si la solución pasada como parámetro es una solución factible para el problema.

`- hasBetterUpperBound (std::vector<Vertex>& tmp, int pos)`: Comprueba si la solución pasada como parámetro tiene una mejor cota superior que la actual.

Pseudocódigo expandNodeStrategy

```
iniciar chrono
si (|tmp| > |Solucion objetivo|)
     return
fin si
si no si(|tmp| <|Solucion objetivo|)
     newTMP = tmp
     newTMP = newTMP + {possileVertex[pos]}
     newDiversity = diversidad(newTMP)
     si (newDiversity >= diversity)
          repetir
               expandNodeStrategy(newTMP, pos + 1,
newDiversity)
          i++
          mientras (i < |possibleVertex|)
     fin si
fin si no si
si no
     si (solución valida & diversity > lowerBound_)
          lowerBound_ = diversity
          tmpSolution = tmp
     fin si
fin si no
```

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.
- M → Número de elementos que contiene la solución.
- K → Número de coordenadas que poseen los vértices que contiene el grafo.
- Ejec → Número de ejecución del problema.
- Z → Diversidad 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 o BranchingAndPruning también podemos encontrar los siguientes datos en las tablas:

- LRC → Tamaño de la Lista Restringida de Candidatos.
- Iter → Número de iteraciones ejecutadas en el algoritmo.
- LowerBound → Cota inferior aplicada.

3.1 Algoritmo Greedy

Problema	N	Ejec	K	M	Z	CPU	Solución
max_div_15_2	15	1	2	2	11.8592	0.00022	{ 8, 6 }
max_div_15_2	15	2	2	3	25.7262	0.00046	{ 8, 6, 3 }
max_div_15_2	15	3	2	4	48.4139	0.00031	{ 8, 6, 3, 10 }
max_div_15_2	15	4	2	5	73.5619	0.00065	{ 8, 6, 3, 10, 1 }
max_div_20_2	20	1	2	2	8.51033	0.00044	{ 17, 18 }
max_div_20_2	20	2	2	3	21.9961	0.00033	{ 17, 18, 8 }
max_div_20_2	20	3	2	4	39.5682	0.00066	{ 17, 18, 8, 2 }
max_div_20_2	20	4	2	5	61.2393	0.00057	{ 17, 18, 8, 2, 12 }
max_div_30_2	30	1	2	2	11.6571	0.00055	{ 8, 27 }
max_div_30_2	30	2	2	3	28.9443	0.00088	{ 8, 27, 1 }
max_div_30_2	30	3	2	4	52.7712	0.00079	{ 8, 27, 1, 10 }
max_div_30_2	30	4	2	5	80.9102	0.001	{ 8, 27, 1, 10, 12 }
max_div_15_3	15	1	3	2	13.2732	0.00044	{ 11, 8 }
max_div_15_3	15	2	3	3	30.3241	0.00051	{ 11, 8, 4 }
max_div_15_3	15	3	3	4	59.7638	0.00041	{ 11, 8, 4, 10 }
max_div_15_3	15	4	3	5	94.7487	0.00056	{ 11, 8, 4, 10, 13 }
max_div_20_3	20	1	3	2	11.8003	0.00049	{ 12, 13 }
max_div_20_3	20	2	3	3	30.8727	0.00045	{ 12, 13, 7 }
max_div_20_3	20	3	3	4	56.5347	0.00058	{ 12, 13, 7, 2 }
max_div_20_3	20	4	3	5	92.8297	0.00158	{ 12, 13, 7, 2, 16 }
max_div_30_3	30	1	3	2	13.0737	0.00064	{ 16, 6 }
max_div_30_3	30	2	3	3	33.8423	0.00073	{ 16, 6, 23 }
max_div_30_3	30	3	3	4	63.5184	0.00101	{ 16, 6, 23, 13 }
max_div_30_3	30	4	3	5	99.5088	0.0011	{ 16, 6, 23, 13, 14 }

3.2 Algoritmo Greedy 2

Problema	N	Ejec	K	M	Z	CPU	Solución
max_div_15_2	15	1	2	2	11.8592	0.00021	{ 8, 6 }
max_div_15_2	15	2	2	3	25.9285	0.00036	{ 4, 1, 6 }
max_div_15_2	15	3	2	4	49.8268	0.00097	{ 6, 5, 0, 8 }
max_div_15_2	15	4	2	5	78.3885	0.00143	{ 6, 8, 0, 1, 5 }
max_div_20_2	20	1	2	2	7.71416	0.00043	{ 19, 2 }
max_div_20_2	20	2	2	3	21.2313	0.00077	{ 18, 2, 1 }
max_div_20_2	20	3	2	4	40.0023	0.00125	{ 2, 1, 8, 18 }
max_div_20_2	20	4	2	5	61.7459	0.00198	{ 2, 8, 10, 17, 1 }
max_div_30_2	30	1	2	2	10.11	0.00069	{ 10, 1 }
max_div_30_2	30	2	2	3	28.9443	0.00109	{ 8, 27, 1 }
max_div_30_2	30	3	2	4	52.7712	0.00185	{ 8, 27, 1, 10 }
max_div_30_2	30	4	2	5	80.9102	0.0031	{ 8, 1, 27, 10, 12 }
max_div_15_3	15	1	3	2	11.4012	0.00025	{ 9, 4 }
max_div_15_3	15	2	3	3	28.7327	0.00073	{ 9, 8, 4 }
max_div_15_3	15	3	3	4	54.3316	0.00117	{ 1, 4, 3, 11 }
max_div_15_3	15	4	3	5	91.5969	0.00223	{ 8, 9, 11, 4, 6 }
max_div_20_3	20	1	3	2	10.7295	0.00033	{ 7, 13 }
max_div_20_3	20	2	3	3	29.4688	0.00071	{ 10, 13, 16 }}
max_div_20_3	20	3	3	4	54.3872	0.00172	{ 3, 12, 7, 2 }
max_div_20_3	20	4	3	5	92.8298	0.0026	{ 12, 2, 7, 16, 13 }}
max_div_30_3	30	1	3	2	12.576	0.00096	{ 23, 7 }
max_div_30_3	30	2	3	3	30.8815	0.00124	{ 4, 23, 5 }
max_div_30_3	30	3	3	4	63.5184	0.0023	{ 6, 13, 16, 23 }
max_div_30_3	30	4	3	5	95.4865	0.00402	{ 4, 23, 13, 6, 16 }

3.3 Algoritmo Grasp

Problema	N	Ejec	K	M	Iter	|LRC|	Z	CPU	Solución
max_div_15_2	15	1	2	2	10	2	9.46527	0.00242	{ 7, 3 }
max_div_15_2	15	2	2	2	10	3	10.0414	0.0032	{ 13, 1 }
max_div_15_2	15	3	2	2	20	2	8.67594	0.0036	{ 9, 0 }
max_div_15_2	15	4	2	2	20	3	11.6972	0.00309	{ 1, 6 }
max_div_15_2	15	5	2	3	10	2	27.2312	0.00236	{ 6, 1, 0 }
max_div_15_2	15	6	2	3	10	3	24.1396	0.00192	{ 14, 1, 4 }
max_div_15_2	15	7	2	3	20	2	25.4472	0.00657	{ 10, 8, 0 }
max_div_15_2	15	8	2	3	20	3	27.2313	0.0042	{ 6, 0, 1 }
max_div_15_2	15	9	2	4	10	2	49.399	0.00387	{ 1, 6, 9, 0 }
max_div_15_2	15	10	2	4	10	3	47.1572	0.00327	{ 9, 10, 0, 1 }
max_div_15_2	15	11	2	4	20	2	43.5005	0.00554	{ 5, 0, 1, 3 }
max_div_15_2	15	12	2	4	20	3	47.8032	0.00622	{ 1, 6, 4, 9 }
max_div_15_2	15	13	2	5	10	2	75.222	0.00488	{ 14, 1, 0, 3, 10 }
max_div_15_2	15	14	2	5	10	3	72.4429	0.00365	{ 6, 0, 5, 1, 11 }
max_div_15_2	15	15	2	5	20	2	73.8484	0.00815	{ 10, 4, 0, 1, 6 }
max_div_15_2	15	16	2	5	20	3	75.3876	0.00802	{ 14, 3, 0, 10, 8 }
max_div_20_2	20	1	2	2	10	2	6.70812	0.00024	{ 13, 18 }
max_div_20_2	20	2	2	2	10	3	6.14044	0.00267	{ 14, 8 }
max_div_20_2	20	3	2	2	20	2	6.50696	0.00496	{ 9, 6 }
max_div_20_2	20	4	2	2	20	3	5.97719	0.00505	{ 4, 18 }
max_div_20_2	20	5	2	3	10	2	20.016	0.00089	{ 7, 2, 8 }
max_div_20_2	20	6	2	3	10	3	18.4362	0.00351	{ 16, 13, 8 }
max_div_20_2	20	7	2	3	20	2	19.0516	0.00585	{ 2, 1, 8 }
max_div_20_2	20	8	2	3	20	3	20.6419	0.00765	{ 2, 10, 1 }
max_div_20_2	20	9	2	4	10	2	32.5798	0.00169	{ 2, 13, 19, 1 }
max_div_20_2	20	10	2	4	10	3	35.0965	0.00393	{ 1, 5, 2, 18 }
max_div_20_2	20	11	2	4	20	2	26.7192	0.00782	{ 4, 14, 7, 19 }
max_div_20_2	20	12	2	4	20	3	38.7187	0.01121	{ 18, 1, 19, 2 }
max_div_20_2	20	13	2	5	10	2	53.2278	0.00414	{14, 17, 18, 19, 6}
max_div_20_2	20	14	2	5	10	3	52.7518	0.00515	{ 14, 9, 8, 18, 19 }
max_div_20_2	20	15	2	5	20	2	53.3331	0.0101	{ 7, 2, 17, 19, 11 }
max_div_20_2	20	16	2	5	20	3	56.5237	0.01266	{ 12, 2, 10, 6, 1 }
max_div_30_2	30	1	2	2	10	2	8.9584	0.00339	{ 29, 9 }
max_div_30_2	30	2	2	2	10	3	8.64868	0.0042	{ 0, 1 }
max_div_30_2	30	3	2	2	20	2	8.64868	0.01045	{ 1, 0 }
max_div_30_2	30	4	2	2	20	3	8.64868	0.00713	{ 0, 1 }
max_div_30_2	30	5	2	3	10	2	26.7139	0.00435	{ 1, 0, 27 }
max_div_30_2	30	6	2	3	10	3	24.047	0.00637	{ 20, 1, 0 }
max_div_30_2	30	7	2	3	20	2	26.7139	0.01445	{ 27, 0, 1 }
max_div_30_2	30	8	2	3	20	3	22.6025	0.00856	{ 1, 10, 0 }
max_div_30_2	30	9	2	4	10	2	38.9056	0.00633	{ 10, 0, 1, 2 }
max_div_30_2	30	10	2	4	10	3	42.6795	0.00818	{ 17, 29, 0, 9 }
max_div_30_2	30	11	2	4	20	2	35.9066	0.01121	{ 19, 1, 0, 2 }

max_div_30_2	30	12	2	4	20	3	42.7328	0.0127	{ 22, 0, 1, 12 }
max_div_30_2	30	13	2	5	10	2	59.7266	0.00873	{ 0, 26, 7, 10, 1 }
max_div_30_2	30	14	2	5	10	3	70.179	0.00882	{ 10, 20, 22, 1, 0 }
max_div_30_2	30	15	2	5	20	2	66.7545	0.01591	{ 18, 1, 10, 25, 0 }
max_div_30_2	30	16	2	5	20	3	61.9382	0.01556	{ 13, 26, 0, 1, 7 }

Problema	N	Ejec	K	M	Iter	|LRC|	Z	CPU	Solución
max_div_15_3	15	1	3	2	10	2	13.2732	0.00237	{ 11, 8 }
max_div_15_3	15	2	3	2	10	3	10.8978	0.00214	{ 12, 4 }
max_div_15_3	15	3	3	2	20	2	12.1612	0.00549	{ 10, 4 }
max_div_15_3	15	4	3	2	20	3	9.50544	0.00547	{ 4, 3 }
max_div_15_3	15	5	3	3	10	2	18.3902	0.00293	{ 0, 6, 1 }
max_div_15_3	15	6	3	3	10	3	26.1821	0.00233	{ 9, 14, 8 }
max_div_15_3	15	7	3	3	20	2	26.4914	0.00542	{ 10, 11, 0 }
max_div_15_3	15	8	3	3	20	3	29.6509	0.00579	{ 11, 6, 0 }
max_div_15_3	15	9	3	4	10	2	51.2794	0.00337	{ 3, 12, 8, 6 }
max_div_15_3	15	10	3	4	10	3	50.9389	0.00446	{ 10, 9, 0, 4 }
max_div_15_3	15	11	3	4	20	2	54.1176	0.00761	{ 11, 2, 4, 3 }
max_div_15_3	15	12	3	4	20	3	54.5893	0.00799	{ 6, 11, 4, 0 }
max_div_15_3	15	13	3	5	10	2	92.6239	0.00458	{ 11, 8, 6, 4, 3 }
max_div_15_3	15	14	3	5	10	3	84.2269	0.00523	{ 13, 7, 11, 4, 6 }
max_div_15_3	15	15	3	5	20	2	92.6239	0.00806	{ 3, 4, 6, 11, 8 }
max_div_15_3	15	16	3	5	20	3	90.2634	0.0084	{ 11, 8, 6, 12, 4 }
max_div_20_3	20	1	3	2	10	2	10.1731	0.00306	{ 11, 0 }
max_div_20_3	20	2	3	2	10	3	9.9954	0.00314	{ 11, 2 }
max_div_20_3	20	3	3	2	20	2	9.77499	0.00537	{ 7, 3 }
max_div_20_3	20	4	3	2	20	3	10.1731	0.0047	{ 11, 0 }
max_div_20_3	20	5	3	3	10	2	20.3364	0.00432	{ 3, 0, 2 }
max_div_20_3	20	6	3	3	10	3	25.5657	0.00449	{ 7, 3, 2 }
max_div_20_3	20	7	3	3	20	2	24.5707	0.00737	{ 13, 17, 0 }
max_div_20_3	20	8	3	3	20	3	26.3877	0.00644	{ 11, 0, 3 }
max_div_20_3	20	9	3	4	10	2	41.9411	0.00632	{ 19, 0, 11, 18 }
max_div_20_3	20	10	3	4	10	3	48.8698	0.00478	{ 16, 3, 2, 0 }
max_div_20_3	20	11	3	4	20	2	48.8698	0.01214	{ 0, 16, 3, 2 }
max_div_20_3	20	12	3	4	20	3	46.2861	0.00927	{ 18, 11, 7, 2 }
max_div_20_3	20	13	3	5	10	2	78.0716	0.00716	{ 10, 11, 3, 0, 2 }
max_div_20_3	20	14	3	5	10	3	77.6117	0.0066	{ 0, 13, 17, 12, 6 }
max_div_20_3	20	15	3	5	20	2	65.4856	0.01325	{ 0, 17, 2, 3, 4 }
max_div_20_3	20	16	3	5	20	3	82.498	0.01294	{ 11, 2, 12, 3, 8 }
max_div_30_3	30	1	3	2	10	2	9.94051	0.00359	{ 7, 1 }
max_div_30_3	30	2	3	2	10	3	10.1673	0.00433	{ 21, 4 }
max_div_30_3	30	3	3	2	20	2	10.8952	0.00852	{ 8, 7 }
max_div_30_3	30	4	3	2	20	3	9.54675	0.01108	{ 14, 0 }
max_div_30_3	30	5	3	3	10	2	27.0254	0.00574	{ 4, 8, 0 }
max_div_30_3	30	6	3	3	10	3	28.9518	0.00572	{ 8, 0, 7 }

max_div_30_3	30	7	3	3	20	2	28.9518	0.01192	{ 0, 8, 7 }
max_div_30_3	30	8	3	3	20	3	23.3734	0.01755	{ 21, 23, 12 }
max_div_30_3	30	9	3	4	10	2	52.6825	0.00715	{ 25, 16, 0, 8 }
max_div_30_3	30	10	3	4	10	3	53.7078	0.00754	{ 0, 3, 10, 7 }
max_div_30_3	30	11	3	4	20	2	50.6869	0.01979	{ 0, 7, 5, 4 }
max_div_30_3	30	12	3	4	20	3	54.3971	0.02274	{ 7, 0, 5, 14 }
max_div_30_3	30	13	3	5	10	2	84.3043	0.01092	{ 23, 5, 29, 0, 8 }
max_div_30_3	30	14	3	5	10	3	69.7177	0.01017	{ 8, 12, 27, 2, 0 }
max_div_30_3	30	15	3	5	20	2	80.7149	0.0262	{ 21, 4, 26, 7, 5 }
max_div_30_3	30	16	3	5	20	3	84.8643	0.03083	{ 0, 4, 6, 7, 10 }

Conclusiones GRASP:

Después de múltiples ejecuciones usando el algoritmo GRASP he llegado a la conclusión de que para estos problemas en concreto realmente no es necesaria una lista restringida de candidatos de tamaño grande, lo que realmente mejora el resultado es un mayor número de iteraciones ya que con 10 o 20 no siempre la solución obtenida es una solución eficiente o próxima a la óptima si lo comparamos con los resultados de ramificación y poda. Desde mi punto de vista para los conjuntos de datos que hemos probado es mas eficiente usar una LRC de tamaño 2 o 3 pero usar al menos 100 iteraciones para que la solución ya sea mas próxima a la óptima ya que en algunos casos la diferencia es bastante notable, es cierto que GRASP no nos aporta la solución óptima, pero si tratamos de encontrar una solución factible y eficiente se debería incrementar el número de iteraciones para un mejor resultado.

Respecto a los tiempos medios de ejecución no se perciben grandes diferencias de una ejecución a otra por lo que no nos es posible sacar conclusiones fiables, lo único que se puede percibir es que se nota un ligero aumento en los tiempos de ejecución cuanto mayor es el tamaño del problema ya que a la hora de elegir los mejores candidatos posibles e introducirlos en la LRC tiene que explorar mas vértices de nuestro grafo.

3.4 Algoritmo LocalSearch

Problema	N	Ejec	K	M	Z	CPU	Solución
max_div_15_2	15	1	2	2	11.8592	0.0006	{ 6, 8 }
max_div_15_2	15	2	2	3	23.8878	0.0008	{ 10, 6, 8 }
max_div_15_2	15	3	2	4	49.4832	0.00141	{ 6, 1, 5, 0 }
max_div_15_2	15	4	2	5	76.7606	0.00386	{ 4, 6, 5, 1, 0 }
max_div_20_2	20	1	2	2	6.70812	0.00038	{ 13, 18 }
max_div_20_2	20	2	2	3	14.1058	0.00061	{ 1, 9, 19 }
max_div_20_2	20	3	2	4	35.177	0.00124	{ 6, 1, 19, 2 }
max_div_20_2	20	4	2	5	57.8684	0.00361	{ 6, 1, 13, 18, 8 }
max_div_30_2	30	1	2	2	8.01641	0.00081	{ 4, 14 }
max_div_30_2	30	2	2	3	18.3259	0.00127	{ 5, 10, 12 }
max_div_30_2	30	3	2	4	48.8703	0.00442	{ 1, 5, 8, 27 }
max_div_30_2	30	4	2	5	64.4112	0.00599	{ 2, 4, 6, 8, 27 }
max_div_15_3	15	1	3	2	9.56224	0.00046	{ 6, 9 }
max_div_15_3	15	2	3	3	22.8951	0.00087	{ 9, 11, 6 }
max_div_15_3	15	3	3	4	50.3368	0.00145	{ 9, 11, 7, 6 }
max_div_15_3	15	4	3	5	75.9184	0.00226	{ 5, 9, 11, 0, 4 }
max_div_20_3	20	1	3	2	11.0048	0.00097	{ 19, 12 }
max_div_20_3	20	2	3	3	30.2691	0.00174	{ 11, 7, 13 }
max_div_20_3	20	3	3	4	53.1229	0.00285	{ 7, 10, 13, 11 }
max_div_20_3	20	4	3	5	81.1084	0.0047	{ 8, 3, 7, 11, 13 }
max_div_30_3	30	1	3	2	12.576	0.00098	{ 23, 7 }
max_div_30_3	30	2	3	3	20.3736	0.00287	{ 4, 8, 16 }
max_div_30_3	30	3	3	4	49.7089	0.00321	{ 4, 6, 9, 16 }
max_div_30_3	30	4	3	5	89.7069	0.00817	{ 0, 4, 6, 8, 16 }

3.5 Algoritmo BranchingAndPruning

e) Completar la tabla de resultados 6 usando como cota inferior la suministrada por el algoritmo constructivo voraz descrito en la figura 1. Usar como estrategia de ramificación la que expande el nodo más profundo (es decir, siguiendo una estrategia de búsqueda en profundidad)

Problema	N	Ejec	K	M	Z	CPU	Solución	LowerBound
max_div_15_2	15	1	2	2	11.8592	0.00055	{ 6, 8 }	11.859
max_div_15_2	15	2	2	3	27.3727	0.00486	{ 0, 6, 8 }	25.726
max_div_15_2	15	3	2	4	49.8268	0.04329	{ 0, 5, 6, 8 }	48.413
max_div_15_2	15	4	2	5	79.1295	0.23733	{ 0, 3, 5, 6, 8 }	73.561
max_div_20_2	20	1	2	2	8.51033	0.00123	{ 17, 18 }	8.5103
max_div_20_2	20	2	2	3	21.9961	0.01343	{ 8, 17, 18 }	21.996
max_div_20_2	20	3	2	4	40.0023	0.12072	{ 1, 2, 8, 18 }	39.568
max_div_20_2	20	4	2	5	63.6517	0.90884	{ 1, 8, 13, 17, 18 }	61.239
max_div_30_2	30	1	2	2	11.6571	0.00199	{ 8, 27 }	11.657
max_div_30_2	30	2	2	3	28.9443	0.04811	{ 1, 8, 27 }	28.944
max_div_30_2	30	3	2	4	52.7712	0.56163	{ 1, 8, 10, 27 }	52.771
max_div_30_2	30	4	2	5	80.9102	5.40241	{ 1, 8, 10, 12, 27 }	80.910
max_div_15_3	15	1	3	2	13.2732	0.00076	{ 8, 11 }	13.273
max_div_15_3	15	2	3	3	31.8685	0.00787	{ 4, 6, 11 }	30.324
max_div_15_3	15	3	3	4	59.7638	0.05676	{ 4, 8, 10, 11 }	59.763
max_div_15_3	15	4	3	5	96.0858	0.32105	{ 3, 4, 8, 11, 13 }	94.748
max_div_20_3	20	1	3	2	11.8003	0.00152	{ 12, 13 }	11.800
max_div_20_3	20	2	3	3	30.8727	0.01872	{ 7, 12, 13 }	30.872
max_div_20_3	20	3	3	4	56.6903	0.15154	{ 2, 12, 13, 16 }	56.534
max_div_20_3	20	4	3	5	92.8297	1.12394	{ 2, 7, 12, 13, 16 }	92.829
max_div_30_3	30	1	3	2	13.0737	0.00243	{ 6, 16 }	13.073
max_div_30_3	30	2	3	3	34.2905	0.04446	{ 5, 16, 23 }	33.842
max_div_30_3	30	3	3	4	63.702	0.64497	{ 5, 13, 16, 23 }	63.518
max_div_30_3	30	4	3	5	99.592	7.52609	{ 5, 13, 14, 16, 23 }	99.508

(f) Completar la tabla de resultados 6 usando como cota inferior la suministrada por el algoritmo constructivo que has propuesto. Usar como estrategia de ramificación la que expande el nodo más profundo (es decir, siguiendo una estrategia de búsqueda en profundidad).

Problema	N	Ejec	K	M	Z	CPU	Solución	LowerBound
max_div_15_2	15	1	2	2	11.8592	0.00075	{ 6, 8 }	11.8592
max_div_15_2	15	2	2	3	27.3727	0.00571	{ 0, 6, 8 }	25.9285
max_div_15_2	15	3	2	4	49.8268	0.04158	{ 0, 5, 6, 8 }	49.8268
max_div_15_2	15	4	2	5	79.1295	0.34145	{ 0, 3, 5, 6, 8 }	78.3885
max_div_20_2	20	1	2	2	8.51033	0.00333	{ 17, 18 }	7.71416
max_div_20_2	20	2	2	3	21.9961	0.017458	{ 8, 17, 18 }	21.2313
max_div_20_2	20	3	2	4	40.0023	0.245740	{ 1, 2, 8, 18 }	40.0023
max_div_20_2	20	4	2	5	63.6517	0.92467	{ 1, 8, 13, 17, 18 }	61.7459
max_div_30_2	30	1	2	2	11.6571	0.00347	{ 8, 27 }	10.11
max_div_30_2	30	2	2	3	28.9443	0.06661	{ 1, 8, 27 }	28.9443
max_div_30_2	30	3	2	4	52.7712	0.78185	{ 1, 8, 10, 27 }	52.7712
max_div_30_2	30	4	2	5	80.9102	6.72847	{ 1, 8, 10, 12, 27 }	80.9102
max_div_15_3	15	1	3	2	13.2732	0.00107	{ 8, 11 }	11.4012
max_div_15_3	15	2	3	3	31.8685	0.00997	{ 4, 6, 11 }	28.7327
max_div_15_3	15	3	3	4	59.7638	0.05746	{ 4, 8, 10, 11 }	54.3316
max_div_15_3	15	4	3	5	96.0858	0.34705	{ 3, 4, 8, 11, 13 }	91.5969
max_div_20_3	20	1	3	2	11.8003	0.00245	{ 12, 13 }	10.7295
max_div_20_3	20	2	3	3	30.8727	0.01754	{ 7, 12, 13 }	29.4688
max_div_20_3	20	3	3	4	56.6903	0.34585	{ 2, 12, 13, 16 }	54.3872
max_div_20_3	20	4	3	5	92.8297	0.72450	{ 2, 7, 12, 13, 16 }	92.8298
max_div_30_3	30	1	3	2	13.0737	0.00245	{ 6, 16 }	12.576
max_div_30_3	30	2	3	3	34.2905	0.04446	{ 5, 16, 23 }	30.8815
max_div_30_3	30	3	3	4	63.702	0.87548	{ 5, 13, 16, 23 }	63.5184
max_div_30_3	30	4	3	5	99.592	5.81443	{ 5, 13, 14, 16, 23 }	95.4865

(g) Completar la tabla de resultados 6 usando como cota inferior la suministrada por el algoritmo GRASP. Usar como estrategia de ramificación la que expande el nodo más profundo (es decir, siguiendo una estrategia de búsqueda en profundidad). LRC de tamaño 2 y 20 iteraciones.

Conclusiones BranchingAndPruning:

Problema	N	Ejec	K	M	Z	CPU	Solución	LowerBound
max_div_15_2	15	1	2	2	11.8592	0.00175	{ 6, 8 }	8.67594
max_div_15_2	15	2	2	3	27.3727	0.00435	{ 0, 6, 8 }	25.4472
max_div_15_2	15	3	2	4	49.8268	0.03759	{ 0, 5, 6, 8 }	43.5005
max_div_15_2	15	4	2	5	79.1295	0.25537	{ 0, 3, 5, 6, 8 }	73.8484
max_div_20_2	20	1	2	2	8.51033	0.00254	{ 17, 18 }	6.50696
max_div_20_2	20	2	2	3	21.9961	0.01454	{ 8, 17, 18 }	19.0516
max_div_20_2	20	3	2	4	40.0023	0.158466	{ 1, 2, 8, 18 }	26.7192
max_div_20_2	20	4	2	5	63.6517	0.9761	{ 1, 8, 13, 17, 18 }	53.3331
max_div_30_2	30	1	2	2	11.6571	0.00349	{ 8, 27 }	8.64868
max_div_30_2	30	2	2	3	28.9443	0.03741	{ 1, 8, 27 }	26.7139
max_div_30_2	30	3	2	4	52.7712	1.5912	{ 1, 8, 10, 27 }	35.9066
max_div_30_2	30	4	2	5	80.9102	7.58665	{ 1, 8, 10, 12, 27 }	66.7545
max_div_15_3	15	1	3	2	13.2732	0.00115	{ 8, 11 }	12.1612
max_div_15_3	15	2	3	3	31.8685	0.01754	{ 4, 6, 11 }	26.4914
max_div_15_3	15	3	3	4	59.7638	0.06859	{ 4, 8, 10, 11 }	54.1176
max_div_15_3	15	4	3	5	96.0858	0.293	{ 3, 4, 8, 11, 13 }	92.6239
max_div_20_3	20	1	3	2	11.8003	0.00172	{ 12, 13 }	9.77499
max_div_20_3	20	2	3	3	30.8727	0.001472	{ 7, 12, 13 }	24.5707
max_div_20_3	20	3	3	4	56.6903	0.31154	{ 2, 12, 13, 16 }	48.8698
max_div_20_3	20	4	3	5	92.8297	4.74394	{ 2, 7, 12, 13, 16 }	65.4856
max_div_30_3	30	1	3	2	13.0737	0.00483	{ 6, 16 }	10.8952
max_div_30_3	30	2	3	3	34.2905	0.06666	{ 5, 16, 23 }	28.9518
max_div_30_3	30	3	3	4	63.702	0.81497	{ 5, 13, 16, 23 }	50.6869
max_div_30_3	30	4	3	5	99.592	9.61403	{ 5, 13, 14, 16, 23 }	80.7149

Después de múltiples ejecuciones usando el algoritmo de ramificación y poda he llegado a la conclusión de que para estos problemas en concreto no siempre es necesaria una cota inferior ajustada ya que la diferencia en tiempo de cómputo es mínima. Todas las ejecuciones han sido realizadas utilizando como cota inferior la suministrada por los algoritmos descritos anteriormente. En el caso de GRASP se ha empleado para todas las cotas inferiores las generadas en GRASP con una lista restringida de candidatos de tamaño 2 y 20 iteraciones.

He podido observar que ya que ramificación y poda es un algoritmo que genera la solución exacta al problema, en todos los casos la solución es la misma, es decir, le apliquemos la cota que sea, la única variación va en función de la cota que provoca que nos genere una cantidad de nodos determinada u otra.