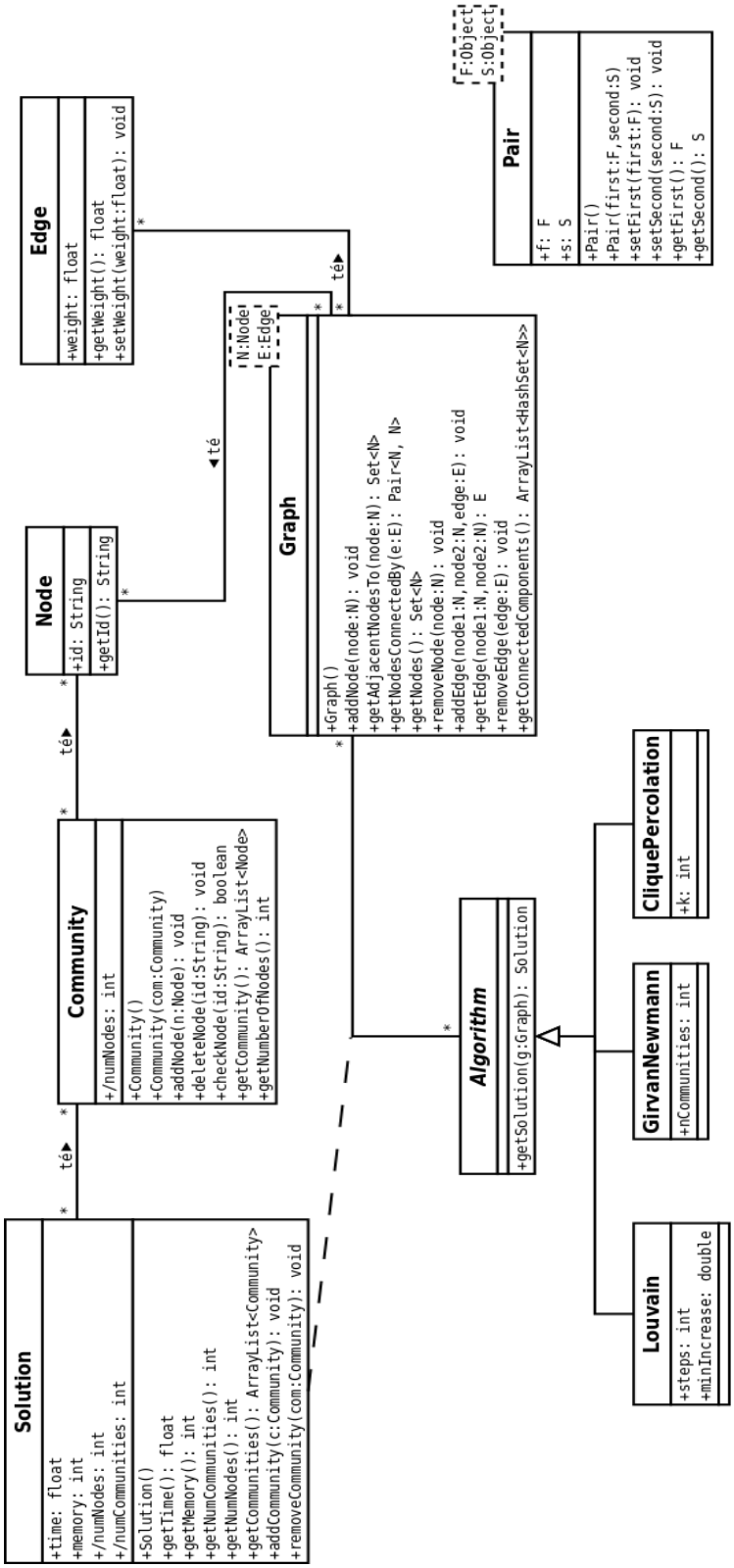
INDEX

DIAGRAMA


Edge
+weight: float
+getWeight(): float
+setWeight(weight:float): void
Pair
F:Object
S:Object
+f: F
+s: S
+Pair()
+Pair(first:F,second:S)
+setFirst(first:F): void
+setSecond(second:S): void
+getFirst(): F
+getSecond(): S
Node
+id: String
+getId(): String
Graph
N:Node
E:Edge
+Graph()
+addNode(node:N): void
+getAdjacentNodesTo(node:N): Set<N>
+getNodesConnectedBy(e:E): Pair<N, N>
+getNodes(): Set<N>
+removeNode(node:N): void
+addEdge(node1:N,node2:N,edge:E): void
+getEdge(node1:N,node2:N): E
+removeEdge(edge:E): void
+getConnectedComponents(): ArrayList<HashSet<N>>
Community
+/numNodes: int
+Community()
+Community(com:Community)
+addNode(n:Node): void
+deleteNode(id:String): void
+checkNode(id:String): boolean
+getCommunity(): ArrayList<Node>
+getNumberOfNodes(): int
Solution
+time: float
+memory: int
+/numNodes: int
+/numCommunities: int
+Solution()
+getTime(): float
+getMemory(): int
+getNumCommunities(): int
+getNumNodes(): int
+getCommunities(): ArrayList<Community>
+addCommunity(c:Community): void
+removeCommunity(com:Community): void
Algorithm
+getSolution(g:Graph): Solution
CliquePercolation
+k: int
GirvanNewmann
+nCommunities: int
Louvain
+steps: int
+minIncrease: double
té
té
té
té

CLASSES COMPARTIBLES:

- Comunity
- Graph
- Node
- Pair
- Edge
- Solution
- Algorithm
- Algorithm louvain
- Algorithm Clique
- Algorithm Girvan Newman
- Estadístiques
- Funció d'afinitat

JUSTIFICACIÓ CLASSES NO COMPARTIDES

- Estadístiques: No les hem compartit ja que cada grup volia obtenir diferents estadístiques. A més la part comuna ja s'obté al crear una solució (Solution).

- Funció d'afinitat: No s'ha arribat a compartir ja que L'únic que teniem en comú era la transformació de criteris i accions en afinitats (arestes del graf). Com cada grup tracta de forma diferents els criteris, no suposa cap aportació compartir la classe.

Grup 1

Community: (responsable Adrián Rambal)

// **class that represents a community of Nodes**

private HashMap<String, **Node**> com;

// Pre: true

// Post: Creates an empty community

public Community();

// Creates a copy of a community

// Pre: true

// Post: the community from the implicit parameter is now a copy of com

public Community(Community com);

// Adds a node to a community

// Pre: true

// Post: the community now contains the Node n (has one more Node)

public void addNode(Node n);

// Delete a Node from the community

// Pre: true

// Post: if the node with identifier ‘id’ is in the community it is deleted, otherwise the //community is not changed.

public void deleteNode(string id);

// Checks whether or not a Node is in the community

// Pre: true

// Post: returns true if a Node with identifier ‘id’ is in the community, else returns false

public bool checkNode(string id);

//Returns the collection of Nodes that belong to the community

// Pre: true

// Post: returns an ArrayList of all the Nodes that form the community

public ArrayList<Node> getCommunity();

// Returns the size (number of Nodes) of the community

// Pre: true

// Post: return the number of Nodes of the community. (0 if it is empty)

public int getNumberOfNodes();

Algorithm: (responsable David Ramal)

//Given a graf, returns a solution depending on the selected algorithm.

// Pre: true;

// Post: generates a solution;

public abstract solution getSolution(Graph g);

LouvainAlgorithm (responsable Alex Osés)

//Class representing Louvain's Algorithm.

// number of passes for one level computation

private int steps;

//a new pass is computed if the last one has generated an increase greater than
//minIncrease

private double minIncrease;

//Pre: True;

//Post: returns a solution according to Louvain's algorithm.

public solution getSolution(Graph g);

Grup 2

Classe Graph (responsable Víctor Antón)

(public class Graph <N extends Node, E extends Edge>)

Membres:

/**For every node, there will be a map containing the nodes it's connected to and its corresponding edges. */ private HashMap<N, HashMap<N, E>> graph;

Mètodes:

/** Add a new disconnected node (without edges to any node)

*@param node The node to be added

*/

//Pre: true

//Post: the node passed as parameter has been added to the graph with no initial connections.

public void AddNode (N node);

/** Returns a set of nodes the given node is connected (adjacent) to. */

//Pre: true

//Post: the adjacent nodes to node have been returned.

public Set<N> GetAdjacentNodesTo (N node);

```
/** Returns the first occurrence of the two nodes connected by the edge e . */

//Pre: true

//Post: a pair containing the two nodes connected by e has been returned.
public Pair<N, N> GetNodesConnectedBy(E e);

/** Returns a set of all the nodes in the graph.  */

//Pre: true

//Post: all the nodes in the graph have been returned.

public Set<N> GetNodes();


/** Remove a node and all its connections.

* @param node The node to be removed.

 */

//Pre:  the graph contains the node passed as parameter

//Post: the node passed as parameter and all the edges connected to it have been removed
public void RemoveNode(N node);

/**

* Add a new edge between two nodes.

* @param node1 The first node to be connected.

* @param node2 The second node to be connected.

* @param edge The edge that will connect node1 and node2.

 */

//Pre: true

//Post: the edge passed as parameter has been added conecting node1 and node2
public void AddEdge(N node1, N node2, E edge);


/** Returns the edge between node1 and node2.  Returns null in case it doesn't exist. */
//Pre: true
```

```
//Post:  the edge connecting node1 and node2 has been returned. Null returned in case node1 and  //            node2 aren't connected.

public E GetEdge(N node1, N node2);



/** Removes a given edge from the graph.

   * @param edge The edge that will be removed.

 */

//Pre: true

//Post: in case it exists in the graph, the edge passed as parameter has been removed from the graph.

public void RemoveEdge(E edge);



/** Returns the connected components in the graph. */

//Pre: true

//Post: returns a list of the connected components in the graph.
public ArrayList< HashSet<N> > GetConnectedComponents();
```

Classe Node (responsable Joan Fons)

(public abstract class Node)

Membres:

Mètodes:

/** Returns a uniquely identifying id for this node. */

//Pre: true

//Post: the id of the node has been returned.

```
public abstract String GetId ();
```

Classe Pair (responsable Joan Fons)

(public class Pair<F, S>)

Membres:

```
private F f;    //First element, of type F
private S s;    //Second element, of type S
```

Mètodes:

```
/** Creates an empty pair (first and second element will be null)*/

//Pre: true

//Post: the pair has been initialized. first and second are null.

public Pair ();


/** Creates a pair given the first and second elements. */

//Pre: true

//Post: the pair's been initialized. first and second are the first and second parameter respectively.

public Pair (F first, S second);


/** Sets the first element to the given F object */

//Pre: true

//Post: the first component of the pair is  the passed parameter
public void SetFirst (F first);


/** Sets the second element  to the given S object*/

//Pre: true

//Post: the second component of the pair is the passed parameter
public void SetSecond (S second);


/** Returns the first element */

//Pre: true

//Post: the first component of the pair has been returned

public F GetFirst ();


/** Returns the second element */

//Pre: true
```

```
//Post: the second component of the pair has been returned
public S GetSecond ();
```

Classe Edge (responsable Abraham Cortes)

(public abstract class Edge)

Membres:

Mètodes:

```
/** Returns the weight of the edge. */

//Pre: true

//Post: the weight of the edge has been returned
public abstract float GetWeight();


/** Sets the weight of the edge to the given weight. */

//Pre: true

//Post: the weight of the edge is the value passed as parameter
public abstract void SetWeight(float weight);
```

Classe GirvanNewman (responsable Aina Soler)

(public class GirvanNewman extends Algorisme)

Membres:

Mètodes:

/** Returns Solution of the GirvanNewman applied to the Graph graph. */

//Pre: true

//Post: the weight of the edge is the value passed as parameter
public Solucio GetSolution(Graph graph);

/** Returns Solution of the GirvanNewman applied to the Graph graph with nCommunities or more
* communities.

* @param graph The Graph taken as input, which must have a number of nodes >= nCommunities.

* @param nCommunities The minimum number of communities the solution must have.

*/

//Pre: the number of nodes of the graph must be greater than the second parameter (nCommunities)
//Post: a Solucio containing nCommunities or more communities obtained via GirvanNewman algorithm // have been returned.

public Solucio GetSolution(Graph graph, int nCommunities);

GRUP 3

Solution (responsable Rubén Marías)

Membres:

private float time;

private int memory;

private ArrayList<**Community**> communities;

Mètodes:

Pre: true

Post: new Solution created, with no communities

public Solution();

Pre: True

Post: The return value represents the time (ms) taken to obtain the solution.

public float GetTime();

Pre: True

Post: The return value represents the memory (bytes) used to obtain the solution.

public int GetMemory();

Pre: True

Post: Returns the number of communities that form the solution.

public int GetNumCommunities();

Pre: True

Post: Returns the number of nodes that form the solution.

public int GetNumNodes();

Pre: True

Post: Returns all the communities of the solution.

public ArrayList<**Community**> GetCommunities();

Pre: True.

Post: c has been added to the solution.

public void AddCommunity(Community c);

Pre: True.

Post: c has been removed from the solution.

public void RemoveCommunity(Community c);

Classe Clique (responsable Pau Oliver)

Membres:

Mètodes:

Pre: 2 <= k < 5

Post: The **Solution** returned is the result of applying the Fast Clique Percolation algorithm to the given **Graph**, considering k-cliques.

public Solution GetSolution(**Graph** g, int k);