CUGAR - Graph Clustering and Visualization Framework

University of Leipzig

Contributors:
R. Speck
A.-C. Ngonga Ngomo

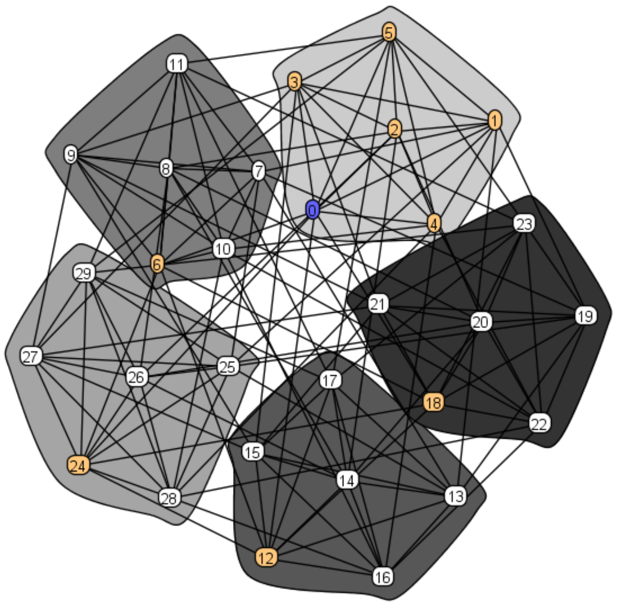


Figure 1: A clustered 6-5-partite clique.

January 25, 2011

Contents

1 Introduction

This manual is intended as a guide for using the Graph Clustering and Visualization Framework (CUGAR), a tool and framework for graph clustering. It presents the functionality of CUGAR in its current version. In addition, it provides insight in the architecture of the underlying framework, so as to enable any user to extend it as required for his purposes.

1.1 Purpose

Our main aim while developing CUGAR was to provide a lightweight framework that allows rapid integration and testing of novel graph clustering algorithms. Roughly speaking, CUGAR in its current version allows to (1) visualize input graphs stored in several formats, (2) cluster the input graph using any of the algorithms integrated, (3) integrate novel algorithms very efficiently as required (section 7.1.1), (4) explore the resulting clustering and (5) store the resulting clustering for further processing.

The toolkit is intended for both beginners and experts: CUGAR provides an easy-to-use graphical interface for users that not yet familiar with graph clustering. In addition, CUGAR's modular architecture allows the rapid integration of new clustering algorithms. Ergo, experts are provided with means to (1) efficiently develop, (2) visualize the results and (3) evaluate the quality of (new) clustering algorithms for graphs. The user interface was designed to be as intuitive as possible, so as to allow the easy manipulation of nodes and clusters during the exploration of the input graph and resulting clustering. In addition, the tool implements several metrics that provide a numeric impression of the quality of the clustering achieved by the algorithm applied.

The visualization component is based on the prefuse visualization toolkit[1], graphs automatically generated by graph models based among others on the jung framework [2] and the Lucene search engine[3], that supports a fast search over nodes. CUGAR implements the clustering algorithms Affinity Propagation, BorderFlow, Chinese Whispers, k-Nearest Neighbours and MCL in its current version. Further algorithms will be integrated soon. CUGAR is open-source and available under the BSD License.

1.2 Requirements

The only requirement to run the application is an installed JRE[4] version 1.6 upwards.

[1]`http://prefuse.org`

[2]`http://jung.sourceforge.net`

[3]See `http://lucene.apache.org/java/1_4_3` for all options.

[4]`http://www.java.com/de/download/manual.jsp`

2 Using CUGAR

2.1 Launching

To start the application, double-click the cugar.jar file or type the following in your terminal: *java -jar cugar.jar*. Should you get *Out of Memory errors*, increase the maximum heap size for your java virtual machine (JVM) via the *-Xmx* option[5].

After you have started the application, you should see the main panel as shown in figure below.

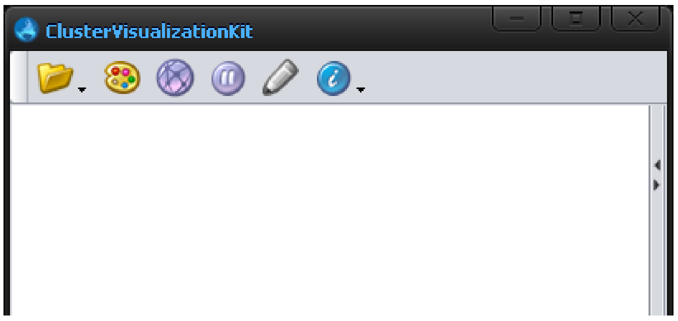


Figure 2: Welcome screen

2.2 Loading data

The first step in using the application is to read data. Two types of data can be loaded into CUGAR: user-generated data (i.e., a file) or automatically generated data for tests purposes.

2.2.1 Loading user-generated data

Currently, the CUGAR supports delimiter separated text files and the XML-format GraphML as input format for user-generated data. To import such data, use the toolbar or the corresponding keyboard shortcut as displayed in figure 4. After you have selected "Open File" from toolbar menu, you will see the dialog shown in 3. By using this dialog, you can choose the type of files to import, i.e., whether a delimiter separated or a GraphML file should be imported. The default "Open File" dialog is set to import delimiter separated data files as they are more commonly used than GraphML. The file extension is used to determine the file delimiter (3.2).

[5] `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`

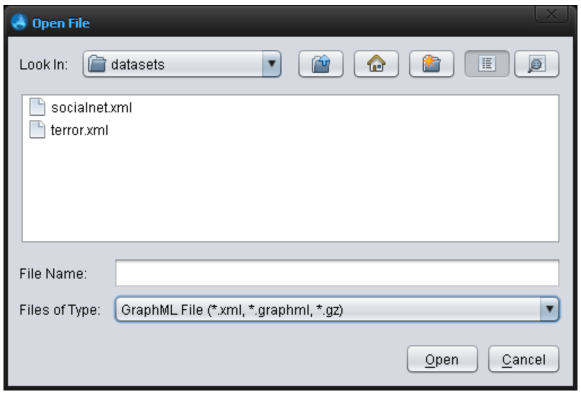


Figure 3: Open File Dialog

2.2.2 Loading automatically generated data

This menu item allows you to load automatically generated data instead of user-generated data. The basic intention behind this feature was to allow users to view the results of algorithms on small graphs without having to encode a graph manually.
After selecting "Open Graph", you will see the submenu as shown in figure 4. This submenu allows you to choose from various types of graphs to load, including the a topped tetrahedron, random n-k partite cliques (i.e., graphs that contains k cliques of n nodes, with $3 \leq n \leq 15$, $2 \leq k \leq 14$ and $k \leq n$), graphs generated by using the binomial model of Erdős and Rényi (n=200, p = 2.4/n) and some others that comes from the prefuse and jung library.

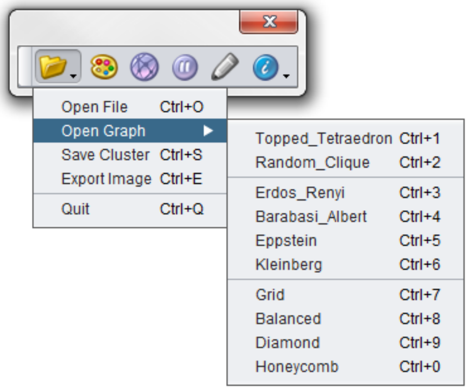
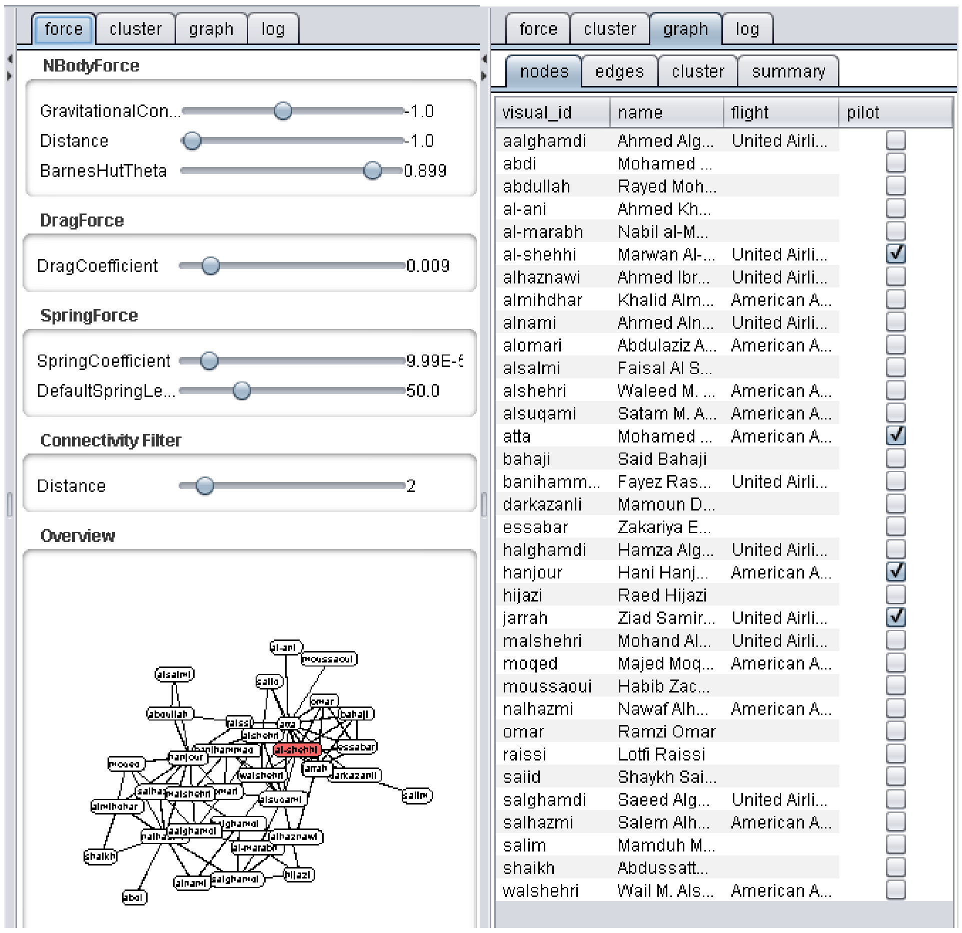


Figure 4: CURAR's toolbar



NBodyForce

GravitationalCon...	-1.0
Distance	-1.0
BarnesHutTheta	0.899

DragForce

DragCoefficient	0.009

SpringForce

SpringCoefficient	9.99E-
DefaultSpringLe...	50.0

Connectivity Filter

Distance	2

visual_id	name	flight	pilot
aalghamdi	Ahmed Alg...	United Airli...	☐
abdi	Mohamed ...		☐
abdullah	Rayed Moh...		☐
al-ani	Ahmed Kh...		☐
al-marabh	Nabil al-M...		☐
al-shehhi	Marwan Al-...	United Airli...	✓
alhaznawi	Ahmed Ibr...	United Airli...	☐
almihdhar	Khalid Alm...	American A...	☐
alnami	Ahmed Aln...	United Airli...	☐
alomari	Abdulaziz A...	American A...	☐
alsalmi	Faisal Al S...		☐
alshehri	Waleed M. ...	American A...	☐
alsuqami	Satam M. A...	American A...	☐
atta	Mohamed ...	American A...	✓
bahaji	Said Bahaji		☐
banihamm...	Fayez Ras...	United Airli...	☐
darkazanli	Mamoun D...		☐
essabar	Zakariya E...		☐
halghamdi	Hamza Alg...	United Airli...	☐
hanjour	Hani Hanj...	American A...	✓
hijazi	Raed Hijazi		☐
jarrah	Ziad Samir...	United Airli...	✓
malshehri	Mohand Al...	United Airli...	☐
moqed	Majed Moq...	American A...	☐
moussaoui	Habib Zac...		☐
nalhazmi	Nawaf Alh...	American A...	☐
omar	Ramzi Omar		☐
raissi	Lotfi Raissi		☐
saiid	Shaykh Sai...		☐
salghamdi	Saeed Alg...	United Airli...	☐
salhazmi	Salem Alh...	American A...	☐
salim	Mamduh M...		☐
shaikh	Abdussatt...		☐
walshehri	Wail M. Als...	American A...	☐

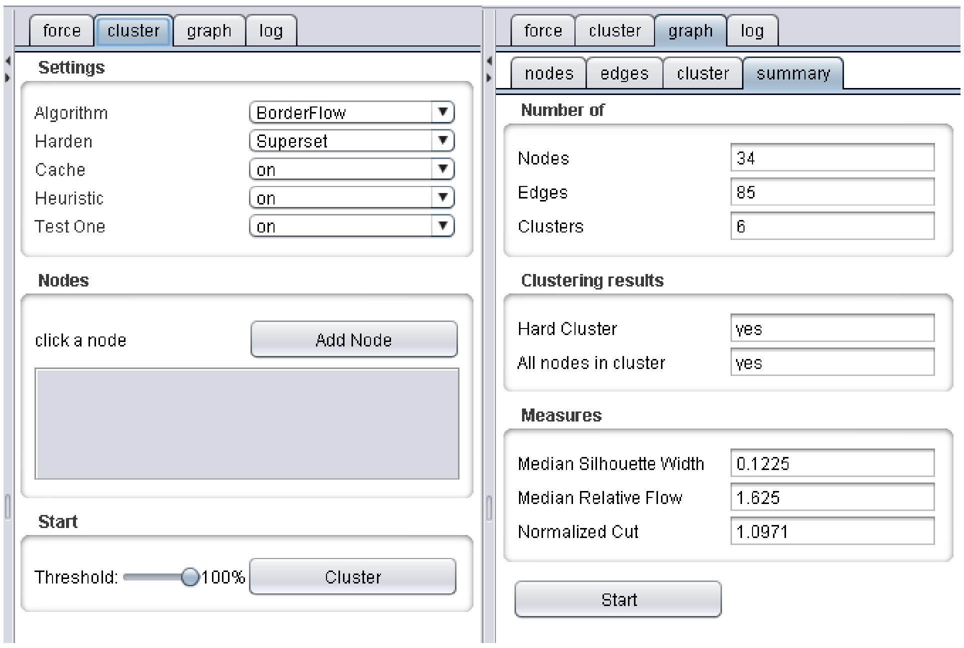
Figure 5: Force configurations (left) and clickable node table (right).

2.3 Configuring

After you have loaded your data, you may want to configure the force layout to display your graph in a fashion suitable to your taste. To achieve this, use the sliders (figure 5 - left) in the three boxes NBodyForce, DragForce and SpringForce. You can also change the Connectivity Filter, this entry controls the maximal distance between nodes that are displayed and the node that is currently selected. Obviously, pushing this value to more than 3 when a large graph is loaded might lead to longer processing time before a stable graph is displayed. To stop the animation, please use the "Play/Pause" button on the menu bar.

One of the drawback of displaying nodes based on the surroundings of the node that is currently selected is that nodes that are at an infinite distance (i.e., not connected) of the currently selected node never get displayed. To guarantee that the user retains an overview of the graph, the interface provide an overview window below the sliders. To ensure that all nodes are displayed, we added the grid button that allows to display all nodes are once. Another way to select a node and thus to display the surroundings of the node is to use the clickable nodes table (figure 5 - right) or to select all nodes in a cluster use the clickable cluster table.

2.4 Clustering



Settings

Algorithm	BorderFlow
Harden	Superset
Cache	on
Heuristic	on
Test One	on

Nodes

Start

Threshold: 100%

Number of

Nodes	34
Edges	85
Clusters	6

Clustering results

Hard Cluster	yes
All nodes in cluster	yes

Measures

Median Silhouette Width	0.1225
Median Relative Flow	1.625
Normalized Cut	1.0971

Figure 6: Clustering configurations (left) and graph statistic (right).

The main aim of the CUGAR is to enable users to test clustering approaches. To choose a clustering algorithm to process the data you loaded, go to the cluster tab (figure 6 - left), pick one of the algorithms (section 4) from the "Settings" box and select/set the parameters you need for your clustering process. CUGAR was designed in such a way that each algorithm has its own clustering configuration. For example, the BorderFlow and k-Nearest Neighbours algorithms allow to select the nodes that are used as seed for clustering; they also provide a threshold slider, which sets the percentage of the maximal connectivity that a node can maximally have to be used as seed (obviously, moving the slider to 0 causes all nodes to be used as seeds). The threshold slider for Chinese Whispers sets the percentage of the maximal edge weight. After having configured your clustering algorithm, press the "Cluster" button to start the clustering process. Depending on the size of the input data and the algorithm, the clustering might take a while. Please be patient.

After the completion of the clustering, CUGAR allows you to navigate through the clusters by using mouse actions on the main panel. By using the color button, you can turn the display of the clusters to greyscale or color. Finally, the "Save clusters" menu item under the folder icon allows to save the results of the clustering for further processing or evaluation.

3 Supported data formats

CUGAR supports various file formats, which are explained below. Furthermore, it is possible to open such graphs directly from the menu bar in the application as described in section 2.2.2. The use of edge weights is possible in all file types. If no weights are set, the weight value is set to 1. Because most clustering algorithms are defined in such a way that they do not support edges that begin and end at the same node, edges with the same node as source and target are ignored when reading the input data. In addition, nodes without edges are ignored.

3.1 GraphML/XML

GraphML is a XML format supporting graph structure and typed data schemas for both nodes and edges. For more information about this format, see the GraphML home page[6]. The only restriction of CUGAR when reading GraphML is that it does not support the mix of weighted and unweighted edges within one graph.

3.2 CSV/SSV/Tab/Txt

CUGAR supports an edge list file format in which each line is an edge. In a line the first term is a source node, second term is a target node and the last term is a weight. We use delimiter separated values, each term in a line has to be separated dependents on the used file format extension as shown in the following table.

File Extension	Separator
csv	comma
ssv	space
tab/txt	tabulator

[6]`http://graphml.graphdrawing.org/`

4 Supported algorithms

CUGAR is intended to enable beginners to utilize exisiting algorithms and developers to test their new algorithms. The current version of CUGAR implements the clustering algorithms described in the following.

4.1 Affinity Propagation

Affinity Propagation takes as input measures of similarity between pairs of data points [2]. Real-valued messages are exchanged between data points until a high-quality set of exemplars and corresponding clusters gradually emerges.

4.2 BorderFlow

BorderFlow is a general-purpose graph clustering algorithm [3]. It uses solely local information for clustering and achieves a soft clustering of the input graph. The standard definition of clusters is that they have a maximal intra-cluster density and inter-cluster sparseness. When considering a graph as the description of a flow system, this definition of a cluster implies that a cluster X can be understood as a set of nodes such that the flow within X is maximal while the flow from X to the outside is minimal. The idea behind BorderFlow is to maximize the flow from the border of each cluster to its inner nodes (i.e., the nodes within the cluster) while minimizing the flow from the cluster to the nodes outside of the cluster.

4.3 Chinese Whispers

Chinese Whispers is a randomized graph-clustering algorithm, which is time-linear in the number of edges [1]. Intuitively, the algorithm works as follows in a bottom-up fashion: First, all nodes get different classes. Then the nodes are processed for a small number of iterations and inherit the strongest class in the local neighborhood. This is the class whose sum of edge weights to the current node is maximal. In case of multiple strongest classes, one is chosen randomly. Regions of the same class stabilize during the iteration and grow until they reach the border of a stable region of another class. Note that classes are updated immediately: a node can obtain classes from the neighborhood that were introduced there in the same iteration.

4.4 k-Nearest Neighbours

In pattern recognition, the k-nearest neighbours algorithm (k-NN) was orginally a method for classifying objects based on closest training examples in the feature space. k-NN is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is deferred until classification. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of its nearest neighbor. k-NN can be used for seed-based clustering simply by stating that each of the $k-1$ nearest neighbors of an input seed are the elements of a cluster of size k (seed + neighbors). Clusters that happen to containe the same elements are merged to one cluster.

4.5 MCL

The basic idea underlying the MCL algorithm [6] is that dense regions (i.e., clusters) in sparse graphs correspond with regions in which the number of k-length paths is relatively large. Thus, random walks of length k have a higher probability to begin and end in the same dense region than for other paths. The algorithm starts uses a stochastic matrix M to represent the input graph. Then, it alternates two operations (expansion and inflation) to compute the set of transition probabilities until M does not change substantially. The result is a complete and does not contain overlapping clusters.

5 Measurements

Several means for assessing the quality of a clustering have been defined in the past. In this section, we describe those measurements that are computed after the completion of each clustering (figure 6 - right). These metrics can be easily extended as required by the user. In the following, we will assume that $G = (V, E, \omega)$ is a weighted directed graph with a set of vertices V, a set of edges E and a weighing function ω, which assigns a positive weight to each edge $e \in E$. V can be partitioned into k subsets $X_1, ..., X_k$. Furthermore, we define $\Omega(X_1, X_2) = \sum_{x_1 \in X_1, x_2 \in X_2} \omega(x_1 x_2)$ as the function that assigns the total weight of the edges from a subset $X_1 \subset V$ to a subset $X_2 \subset V$.

The toolkit implements the median of average silhouette width, the median of the relative flow and the normalized cut measures.

5.1 Median of average silhouette width

The silhouette measures of a set is given by the following equation[4]

$$s(x) = \frac{b(x) - a(x)}{max\{a(x), b(x)\}}$$

where $a(x)$ is the average similarity between vertex $x \in X$ and all other vertices in subset X, and $b(x)$ is the average similarity between vertex $x \in X$

and all other vertices in the neighbour subsets to X.

We implemented the median of the average silhouette width S_{med} to provide users of the CUGAR with a quick evaluation of their clustering with respect to the the partitioning of the input data:

$$S_{med} = \begin{cases} \overline{S}_{\frac{k+1}{2}} & \text{if k is odd} \\ \frac{1}{2}(\overline{S}_{\frac{k}{2}} + \overline{S}_{\frac{k}{2}+1}) & \text{if k is even} \end{cases}$$

with a sorted list $\overline{S}$ of the average silhouette width of all X_i, $i \in \{1, ..., k\}$, $\overline{s}(X_i) = \frac{\sum_{x \in X_i} s(x)}{|X_i|}$

5.2 Average relative flow

As another mean to asses the quality of a clustering, we implemented the median of the relative flow[3], which is defined as:

$$F(X) = \frac{\Omega(b(X), X)}{\Omega(b(X), n(X))}$$

where $b(X)$ is the set of border vertex of X and $n(X)$ is the set of direct neighbors of X.

5.3 Normalized cut

Any subset of vertices $X \subset V$ creates a cut, which is a partition of V into two disjoint subsets X and $V \setminus X$. The size of a cut X of graph G is defined as $cut(X) = \omega(X, V \setminus X)$ and measures the weight of edges that have to be eliminated in order to obtain the two components X and $V \setminus X$. The normalized cut[5] normalizes the cut measurement with the total sum of all vertex degrees over all vertices in a subset X and is defined as

$$ncut(X_1, ..., X_k) = \frac{1}{2} \sum_{i=1}^{k} \Omega(X_i, V \setminus X_i)/vol(X_i)$$

where $vol(X) = \sum_{x \in X} d_x$ is the volume of subset X and $d_x = \sum_y \omega(xy)$ is the degree of vertex $x \in X$.

6 Hardening

Many clustering algorithms return overlapping (i.e., BorderFlow) clusters or clusterings that do not cover all input data (i.e., DBScan, RNSC). CUGAR provides means to ensure that each clustering do not contain overlapping clusters and are complete, both characteristics being desirable for several types of applications. Currently, the toolkit implements two different hardening strategies: MaxQuality and SuperSet. Both require the following input data:

1. a graph $G = (V, E, \omega)$
2. a set $\Psi = \{X_1, ..., X_k\}$ of k subsets of V with
3. $\forall i \in \{1, ..., k\} X_i \neq \emptyset$.

A hardening process consists of transforming the set Ψ of k subsets to a set $\Psi^{'}$ of $k^{'}$ subsets such that $\forall i, j \in \{1, ..., k^{'}\} X_i^{'} \cap X_j^{'} = \emptyset$.

6.1 Hardening with MaxQuality strategy

The MaxQuality strategy applies functions f and f', quality functions, that assigns each subset X a value $f(X) \geq 0$.

1. remove all $X_i \in \Psi$ with $|X_i| = |V|$ from Ψ
2. put in a set S_{max} all $X_i \in \Psi$ with maximal $f(X_i)$
3. if S_{max} contains h subsets such that $\bigcap_{i \in \{1,...,h\}} X_i^{'} = \emptyset$ then put all these h subsets to $\Psi^{'}$, remove these h subsets from Ψ and go to step 5
4. else while $S_{max} \neq \emptyset$ do
   (a) find subsets in S_{max} they share any vertex, put these in $S_{max}^{'}$ and remove them from S_{max} and from Ψ
   (b) if $S_{max}^{'}$ is empty then copy all subsets from S_{max} to $\Psi^{'}$ and goto setp 5.
   (c) take all shared vertices in $\bigcap_{X \in S_{max}^{'}} X = R$ and remove them from all subsets $X \in S_{max}^{'}$ such that $\bigcap_{X' \in S_{max}^{'}} X^{'} = \emptyset$
      i. if $|R| = |\bigcup_{X' \in S_{max}^{'}} X^{'}|$ then put R to $\Psi^{'}$
      ii. else assign each vertex $v \in R$ to a subset $X^{'} \in S_{max}^{'}$ such that $f'(X')$ is maximal and put these $X^{'}$ to $\Psi^{'}$
   (d) set $S_{max}^{'} = \emptyset$
5. remove all vertices $\bigcup_{X' \in \Psi'} X^{'}$ from subsets in Ψ
6. remove empty sets in Ψ
7. if $\Psi \neq \emptyset$ go to step 2
8. assign each vertex that isn't in a subset to a subset $X^{'} \in \Psi^{'}$ so that $f'(X^{'})$ is maximal.

6.1.1 MaxQuality with relative flow function

The implementation uses relative flow for the quality function f and the flow from node to set $(\Omega(v, X_i))$ for f' as described in section 5.

6.1.2 MaxQuality with silhouette function

The implementation uses silhouette for the quality function f and the flow from node to set $(\Omega(v, X_i))$ for f' as described in section 5.

6.2 Hardening with SuperSet strategy

The SuperSet strategy works as follows:

1. Discard all subsets X_i such that $\exists X_j : X_j \subset X_i$.
2. Order all remaining X_j into a list $L = \{\lambda_1, ..., \lambda_m\}$ in descending order with respect to the number of seeds that led to their computation. Formally, let $\mu(\lambda_i) \subseteq V$ be the set of seeds of a subset λ_i. Then
$$i < j \Rightarrow |\mu(\lambda_i)| \geq |\mu(\lambda_j)|.$$
3. Let k be the smallest index such that the union of all λ_i with $i \leq k$ equals V:
$$\bigcup_{i=1}^{k} \lambda_i = V \wedge \forall j < k : \bigcup_{i=1}^{j} \lambda_i \subset V.$$
Discard all $\lambda_z \in L$ with $z > k$.
4. Re-assign each v' to the subset(s) X' such that
$$X' = \operatorname*{arg\,max}_{X \in \{\lambda_1, ..., \lambda_k\}} \frac{\sum_{v \in X} w(v', v)}{|X|}.$$
5. Return the new subsets.

7 Developers section

This section goes beyond presenting the features of CUGAR and shows how to extend the framework. It is intended for advanced users that aim at integrating their own algorithm or measures in the suite. The developers section is divided into four subsections. The first gives an introduction to the packages and describes how to extend CUGAR by integrating own algorithms. Then the following sub sections lists the dependencies, bugs and change log.

To compile CUGAR use the cugar.xml ant script in the root folder.

7.1 Packages

CUGAR contains two main packages. The *bf* package, which contains the implementations of the BorderFlow and the kNN algorithm and the *cugar* package, which contains the core of the framework. All details of the interfaces, methods and classes implemented in CUGAR can be found in the Javadoc of the project[7]. The entry point of CUGAR is in the *cugar.gui.Main* class.

7.1.1 The cugar package

This package contains the framework per se and is surely the more interesting package for developers. Figure 7 gives an overview of all subpackages contained in the cugar package.

1. **cugar.cluster**: Contains an abstract clustering algorithm class from which all clustering algorithm adapters or new clustering algorithm has to inherit. These clustering algorithms (adapted or new) are loaded into the application using a plugin system. The loading process is carried out by the context class that also implements the simple plugin system. Furthermore, an evaluation classes is included to evaluate the mentioned results.

2. **cugar.data**: This package contains the model class of the application and some classes to loaded user-generated or automatically generate data.

3. **cugar.harden**: In this package are all hardening strategies and an interface for graphs that is used by this strategies.

4. **cugar.gui**: Contains the graphical user interface and a logging system.

[7] `http://borderflow.sourceforge.net/doc/index.html`

5. **cugar.visual**: Includes all display and visualization classes depending on the prefuse library.

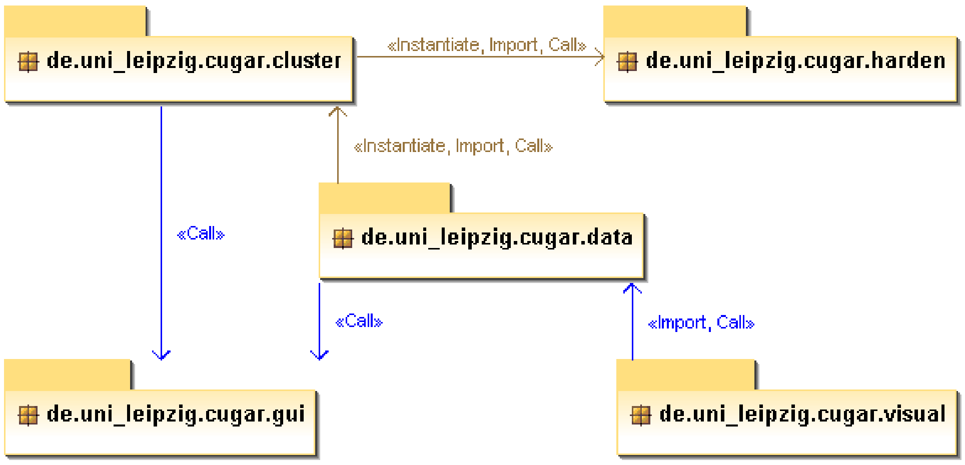

de.uni_leipzig.cugar.cluster
«Instantiate, Import, Call»
de.uni_leipzig.cugar.harden
«Instantiate, Import, Call»
«Call»
de.uni_leipzig.cugar.data
«Call»
«Import, Call»
de.uni_leipzig.cugar.gui
de.uni_leipzig.cugar.visual

Figure 7: Package cugar.*.

cugar.cluster

This package contains the adapter classes (*ClusterAlgorithmBorderFlow*, *ClusterAlgorithmKNN*, ...) for all supported clustering algorithms. Each of these adapter class is inherited from the abstract *ClusterAlgorithm* class and implements at least the abstract *clustering* method and the *getName* method as shown in figure 9. These adapters are managed by the *ClusterContext* class once they had loaded with a simple java class loader plugin system. You can add an algorithm via the plugin system without recompiling CUGAR or you can add your algorithm by recoding the curarDemo method in entry point as shown in figure 8.

The example in figure 8 explicates how to implement and integrate an adapter (ClusterAlgorithmFoobar) for self-developed algorithm (Foobar).

In the 1st step of the integration, the four String parameters of the *clustering* method comes from the member String arrays *config_A* ,..., *config_D* and can be used to provide possible paramaterizations of the novel algorithm. For example, the implementation of kNN uses *config_A = new String[]{"k","1",...,"100"}* for the parameter k. It is important to notice that the first element of the array is the label shown in CUGAR for the parameter. A user can choose the other elements of each array from a JComboBox in the application. These are the values that are sent back to the *clustering* method as parameters so that they can be used for configurations.

The general is pretty simple, all you need to do is:

1. Extend ClusterAlgorithm class by overriding the member String arrays to set new configuration options and implement the abstract *clustering* method and the *getName* method.

```java
public class ClusterAlgorithmFoobar extends ClusterAlgorithm {

  public ClusterAlgorithmFoobar(){
    config_A = new String []{"label_A","value_1␣","value_2"};
  }
  @Override
  protected Table clustering(String[] seeds, double threshold,
      String valuesA, String valuesB, String valuesC, String valuesD) {

    Foobar.loadFile( filename , getSeparator ());
    List<Set<String>> clusters = Foobar.clusters(valuesA);
    Table table = getTable();
    for( Set <String > cluster : clusters )
      table.set(table.addRow(), CLUSTER_COLUMN_NAME, cluster);
    return table ;
  }
  @Override
  public String getName() {
    return "Foobar";
  }
}
```

2(a). Integrate the adapter by modifying the *cugar.gui.Main.cugarDemo* method which starts CUGAR.

```java
public static void cugarDemo(){
  ...
  ClusterContext cc = model.getClusterContext();
  cc.addAlgorithm("Foobar", new ClusterAlgorithmFoobar());
  ...
}
```

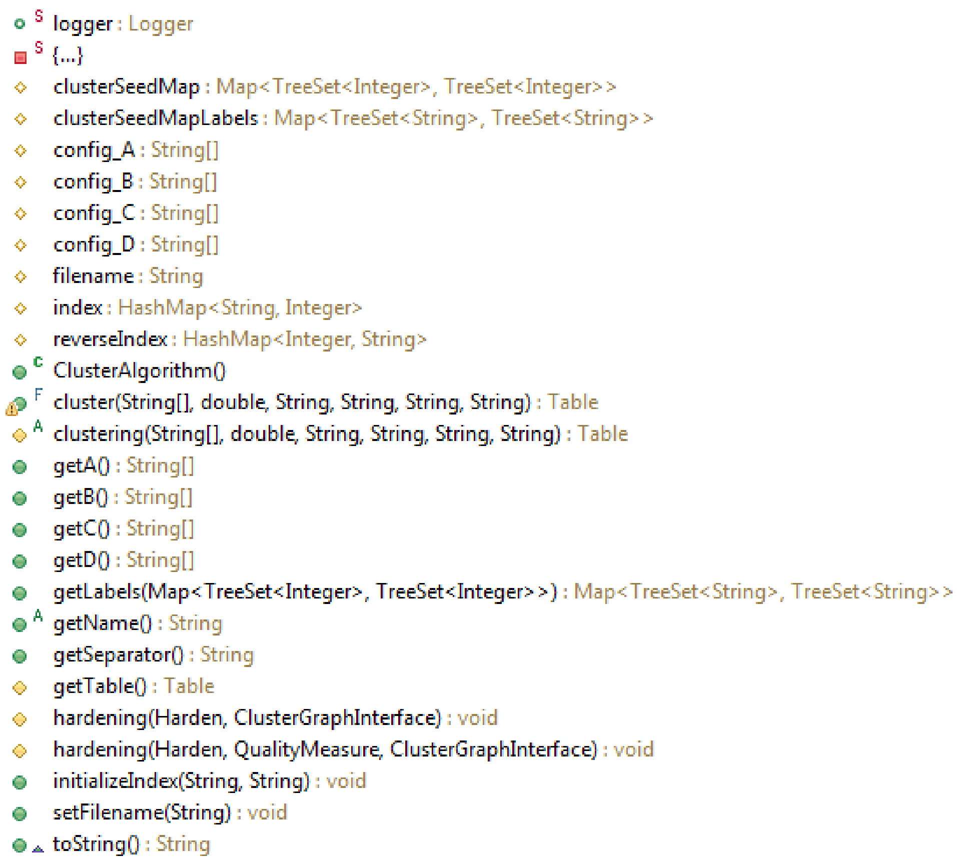
2(b). Integrate the adapter via the plugin system by creating a jar file, which contains the ClusterAlgorithmFoobar.class file and depositing this jar in the plugin folder of CUGAR. It is important that your class file is implemented in the *cugar.cluster* package.

Figure 8: Integration of a new algorithm

The String array parameter *seeds* gets all nodes that were added in Nodes table (figure 5-left) to use as seeds. If no seeds added to the table, the seeds parameter will be *null* and the double parameter *threshold* gets a value between 0 and 1 selected by the user from the threshold slider.
The return statement is a *prefuse.data.Table* with a specified column which holds the cluster data sets. The column name *CLUSTER_COLUMN_NAME*

and data type *CLUSTER_COLUMN_NAME_TYPE* are defined in *ClusterTableSettings* interface and are set to "Cluster" and TreeSet of String. The *getTable()* method gets a valid table instance.
The String member variable *filename* contains the path to a delimiter separated text file, the used separator get back by *getSeparator* method.
It is possible to add more informations to the returned *prefuse.data.Table* instance, which will be displayed in the cluster table under the graph tab in the application.

In 2(a). the first parameter of *addAlgorithm* method is the name of your algorithm shown in the application and has to be different from already existing names. The second parameter is your algorithm instance. By starting the application all algorithm adapters will be instantiated.



```
logger : Logger
{...}
clusterSeedMap : Map<TreeSet<Integer>, TreeSet<Integer>>
clusterSeedMapLabels : Map<TreeSet<String>, TreeSet<String>>
config_A : String[]
config_B : String[]
config_C : String[]
config_D : String[]
filename : String
index : HashMap<String, Integer>
reverseIndex : HashMap<Integer, String>
ClusterAlgorithm()
cluster(String[], double, String, String, String, String) : Table
clustering(String[], double, String, String, String, String) : Table
getA() : String[]
getB() : String[]
getC() : String[]
getD() : String[]
getLabels(Map<TreeSet<Integer>, TreeSet<Integer>>) : Map<TreeSet<String>, TreeSet<String>>
getName() : String
getSeparator() : String
getTable() : Table
hardening(Harden, ClusterGraphInterface) : void
hardening(Harden, QualityMeasure, ClusterGraphInterface) : void
initializeIndex(String, String) : void
setFilename(String) : void
toString() : String
```

Figure 9: Abstract ClusterAlgorithm class.

cugar.data

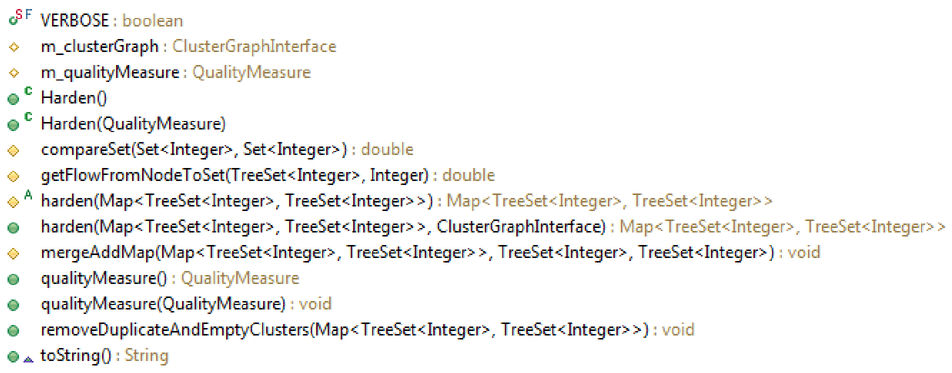
This package provides a *Model* class which has a instance of a *ClusterContext* class to get access to the current used algorithm (compare figure 10). Also, it provides classes to read or generate data and the resulting graph is stored into a *prefuse.data.Graph* instance.

The *Model* class is the only class that needs to be accessed by external classes to read input data by means of the integrated *EdgeListGraphReader* or *GraphMLReaderMod*, to generate graphs by using the *GraphGenerator* and to assign tasks to the currently chosen algorithm.

TODO ...

Figure 10: Package cugar.data, cugar.cluster and cugar.harden.

cugar.harden



```
VERBOSE : boolean
m_clusterGraph : ClusterGraphInterface
m_qualityMeasure : QualityMeasure
Harden()
Harden(QualityMeasure)
compareSet(Set<Integer>, Set<Integer>) : double
getFlowFromNodeToSet(TreeSet<Integer>, Integer) : double
harden(Map<TreeSet<Integer>, TreeSet<Integer>>) : Map<TreeSet<Integer>, TreeSet<Integer>>
harden(Map<TreeSet<Integer>, TreeSet<Integer>>, ClusterGraphInterface) : Map<TreeSet<Integer>, TreeSet<Integer>>
mergeAddMap(Map<TreeSet<Integer>, TreeSet<Integer>>, TreeSet<Integer>, TreeSet<Integer>) : void
qualityMeasure() : QualityMeasure
qualityMeasure(QualityMeasure) : void
removeDuplicateAndEmptyClusters(Map<TreeSet<Integer>, TreeSet<Integer>>) : void
toString() : String
```

Figure 11: Abstract *Harden* class.

This package contains the hardening strategies (described in section 6). The hardening techniques implemented in this package can be used by all other algorithms implemented within CUGAR.

To implement your own hardening strategy you can inherit from the abstract *Harden* class and implement the abstract *harden* method. The first parameter is a map of TreeSets that contains clusters as key and seeds as value. The second parameter is a data structure of the graph that is used by the strategies. The returned map is the hardening result.

The *ClusterAlgorithm* class described in 7.1.1 contains a member variable that can be used as map for hardening and some other methods that supports develops by using their hardening strategy (e.g., the hardening based on integer and the *ClusterAlgorithm* class has members for indexing and reverse indexing, this members can be initialized with the *initializeIndex* method).

7.1.2 The bf package

This package contains the BorderFlow and the k-Nearest Neighbors algorithms (described in section 4), which were available from the beginning of this project on. The BorderFlow algorithm was developed independently from CUGAR and integrated in agreement with its main developer.

7.2 Dependencies

7.2.1 Prefuse

ClusterVisualizationKit uses the prefuse toolkit and therefore the prefuse library beta-20071021 has to be included in your lib folder.
`http://prefuse.org`

7.2.2 Lucene

The prefuse toolkit supports a fast search with the Lucene search engine and we use it certainly. The library of Lucene search engine version 1.4.3 has to be included in your lib folder also.
`http://lucene.apache.org/java/1_4_3`

7.2.3 Jung

The "Java Universal Network/Graph Framework" is used to generate random graphs.
`http://jung.sourceforge.net`

7.3 Bugs

The application contain no critical bugs (i.e., bug that lead to a system failure) as far as we are awate. Still some minor bugs are still included in the library we utilize as listed in the subsequent section. Please do not hesitate to contact us if you have any helpful tips or comments on how to avoid these bugs. Furthermore, any feedback on unknown bugs will be greatly appreciated.

java.lang.IllegalArgumentException

An IllegalArgumentException with an invalid row index is thrown after a clustering task sometimes.

7.4 Change Log

Version - beta 0.6 (25. January 2010)

Added	graph models
Added	some clustering algorithms
Added	clickable cluster table
Added	plugin system
Fixed	some small bugs
Fixed	manual

Version - beta 0.4 (9. August 2010)

Added	toolbar, java look and feel
Added	measurements
Added	mcl algorithm
Fixed	some small bugs

Version - beta 0.3 (7. July 2010)

Added	java doc to source
Added	graph generator for cliques and Erdos-Renyi
Added	harden for all algorithms
Added	split package cvk.data to package cvk.data.cluster
Fixed	covered draw of clusters
Fixed	ClusterAlgorithm class hierarchy
Fixed	some small bugs

Version - beta 0.2 (23. Jun 2010)

Added	image export
Added	graphml and delimiter file support
Added	load example graphs from menu
Added	log to file
Added	choice of a node from node list
Added	edge list in graph statistic
Added	knn algorithm
Fixed	ConcurrentModificationException
Fixed	some small bugs

Version - beta 0.1 (31. May 2010)

First release.

References

[1] C. Biemann. Chinese whispers - an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of the HLT-NAACL-06 Workshop on Textgraphs-06*, New York, USA, 2006.

[2] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[3] Axel-Cyrille Ngonga Ngomo and Frank Schumacher. Border flow a local graph clustering algorithm for natural language processing. In *Proceedings of the 10th International Conference on Intelligent Text Processing and Computational Linguistics (CICLING 2009)*, pages 547–558, 2009. Best Presentation Award.

[4] Peter Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, 1987.

[5] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:888–905, 2000.

[6] S. van Dongen. *A Cluster algorithm for graphs.* PhD thesis, Centrum voor Wiskunde en Informatica, 2000.