Manual for Galib : A Graph Algorithm Library

Network Dynamics and Simulation Science Laboratory
Virginia Tech, Blacksburg, VA 24073

July 20, 2016

Galib is a library of graph algorithms for the analysis of large scale labeled graphs. There are some other existing graph libraries such as LEDA, Boost and NetworkX. However, the analysis of many large graphs that arise in practice lead to the following new challenges, which limits the utility of these libraries.

- The graphs we consider (especially social contact graphs) have millions of nodes and edges, which leads to significant space-time challenges – these graphs barely fit in main memory, and super-linear time algorithms become infeasible. This necessitates approximation algorithms for these problems, but most of these libraries only have exact implementations (e.g., of the clustering coefficient). Additionally, there is a significant overhead due to the powerful and generic graph data structure, and new light-weight methods are needed.
- All of these libraries only deal with unlabeled graphs. A key feature of the graphs we study is that they are labeled – nodes have attributes such as demographic information (e.g., age, immunity level, gender, and household income), and edges have attributes such as contact time, nature of contact, location, etc. Many interesting analysis can be done using such labels. Further, these graphs are time-varying, and this cannot be captured easily in these libraries.
- Most applications do not deal with either the dynamics of the graphs or the dynamical processes defined on the graphs (e.g., spread of epidemics). This aspect has received very limited attention, and there is no support for such computations in any of the available libraries.

Galib provides efficient implementations of various classical and new graph measures that are motivated by the analysis of social contact graphs and disease dynamics on such graphs, including (i) analysis of the subgraph structure and relational labeled graph queries, e.g., counts and relative frequencies of subgraphs such as cliques and stars, and (ii) graph measures related to disease dynamics on social contact networks, e.g., *vulnerability* of a node, which is defined as the probability that the node gets infected; also, there are subgraph queries motivated by disease dynamics, which are also provided as part of Galib . For some of these algorithms, Galib implements sampling based approximation algorithms known in literature, but with error guarantees that

can be controlled by the users, in contrast to exact implementations provided in other libraries, leading to significant speedups. The data structures and algorithms included in Galib are carefully tuned to be capable for running on large graphs containing up to millions of nodes. Galib is written in C++.

Below is a list of the tools included in Galib . Definitions and descriptions of how to run them follow the list.

1. cgraph – A collection of various graph operations
2. cneighbor – Find the distribution of the number of common neighbors of the pairs at a given distance
3. component – Count the number of connected components and their sizes
4. shuffle – Shuffle the edges of a graph
5. dshuffle – Degree-assortative shuffle of the edges of a graph
6. gengraph – Generate graph using various models: Chung-Lu, $G(n, p)$, etc.
7. likelihood – Compute likelihood of a given graph
8. ClusterCo – Compute exact and approximate clustering coefficients
9. streamcc – Streaming version for computing approximate clustering coefficients
10. streamccT – Streaming version for computing approximate clustering coefficients and enumerate sampled triangles
11. parse – A distributed-memory parallel algorithm to count overlapped subgraphs isomorphic to a given template
12. spath – Computes the distribution of unweighted distance between two nodes
13. CliqueCount – Count the number of cliques of a given size in the graph
14. CliqueCompression
15. betweenness – Computes betweenness centrality
16. DegreeDistribution – Computes the degree distribution
17. RoDistribution – Computes the ro distribution
18. sssp – Computes single source shortest path using Dijkstra's algorithm in a graph
19. spt - Computes shortest path tree in a graph
20. mst – Computes minimum spanning tree using Prim's algorithm in a graph

21. wsp – Computes average shortest path distance, weighted diameter and shortest path distance distribution of a graph

22. rmnode – Removes some specified nodes from a graph and creates another new graph

23. rmnodedegree – Removes the nodes having degrees greater than or equal to some input degree from a graph and creates another new graph

24. rmnodepercentage – Removes topmost $x\%$ nodes (having higher degrees) from a graph and creates another new graph

25. rmedge – Removes some specified edges from a graph and creates another new graph

26. ggraph complete – Generate a complete graph

27. ggraph chain – Generate a chain or path graph

28. ggraph star – Generate a star graph

29. ggraph cycle – Generate a cycle or circle graph

30. ggraph wheel – Generate a wheel graph

31. ggraph btree – Generate a binary tree graph

32. ggraph grid – Generate a grid graph

33. ggraph torus – Generate a torus graph

34. bfsForest – Generate BFS forest from input graph

35. bfsPart – Generate BFS-style partitions

36. compGraph – Generate component graphs from the input graph

37. graphical – Check whether a degree sequence is graphical

38. ParCC – Parallel (MPI)implementation of determining Clustering Coefficients

39. Euel2gph – External memory conversion from edge list to graph (Galib) format

1 Galib Graph File Format

The graph is formatted as adjacency lists in a text file. The first line in the file is n, the number of vertices in the graph, followed by n blocks of data – one block for each node v. The first line of each block contains ID of the node v and degree $d(v)$ of this node. The first line of each block is followed by $d(v)$ lines; each such line contains ID of a neighbor u of v, weight (contact duration) of the edge (u, v), and label of the edge (u, v). For example, consider the graph given in Figure 1. Assume that edge $(3, 55)$ has weight 4 and label 2 and the rest of the edges have weight 1 and label 0. Then the corresponding graph file is shown in Table 1.

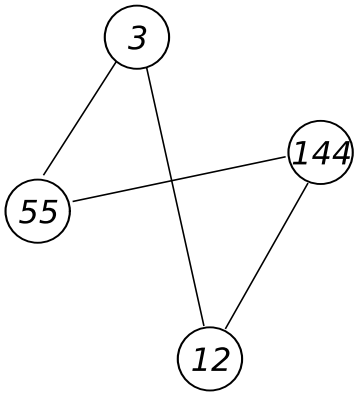

Figure 1: A simple graph with 4 nodes and 4 edges.

Table 1: Galib file format of a simple graph shown in Figure 1.

4			
55	2		
	3	1	0
	144	1	0
3	2		
	55	4	2
	12	1	0
144	2		
	55	1	0
	12	1	0
12	2		
	3	1	0
	144	1	0

In Galib Git (see next section), some sample graphs in Galib format can be found in subdirectory "/Graphs/Sample/".

Table 2: UEL file format of a simple graph shown in Figure 1.

3	0	12	0	1
3	2	55	2	4
55	0	144	0	1
144	0	12	0	1

2 UEL Graph File Format

The graph $G = (V, E)$ is formatted as undirected edge lists (UEL) in a text file with file extension ".uel". There are total $|E|$ number of lines in the file, where $|E|$ is the total number of edges in the graph. Each edge $(u, v) \in E$ is written in one line in a 5-tuple format as follows: $< u\ label_u(u, v)\ v\ label_v(u, v)\ weight(u, v) >$. Here, $weight(u, v)$ is the weight (contact duration) of the edge, $label_u(u, v)$ is the label of the edge (u, v) with respect to u. Typically, $label_u(u, v) = label_v(u, v)$. In some cases, such as contact network (where $label(u, v)$ represents contact type), $label_u(u, v)$ and $label_v(u, v)$ can be different. For example, consider the graph given in Figure 1. Assume that edge $(3, 55)$ has weight 4 and label 2 and the rest of the edges have weight 1 and label 0. Then the corresponding graph file is shown in Table 2.

3 Galib Git Location

Galib package can be found at NDSSL Git location:

https://ndsslgit.vbi.vt.edu/software-galib/galib.git

Below are the main subdirectories of Galib Git:

- Bin – contains the executable binaries of the Galib modules
- Cfg – contains configurations files. A configuration file contains input to a Galib module.
- Docs – contains various documents related to Galib
- Graphs – contains graph files
- Input – contains various input files other than the graph files
- Logs – contains log files
- Output – contain output files
- Scripts – contains various perl and bash scripts
- Src – contains C++ source files of the Galib modules

The binary files may not be included in the Git. However, you can generate them by compiling the source using the Makefile in Src subdirectory.

4 Running Specifications

We use a contact graph called Miami when discussing the time and space cost of various tools in Galib . The reason is that graph analysis always involves with large time and space consuming. Therefore, we hope to build the intuition on when and where to use Galib , by providing some instances of performance analysis on moderate sized graphs. The specifications of the networks are in Table 3:

Table 3: Two networks: Miami and Internet

Graph	Nodes	Edges	Average Degree
Miami	2,092,147	105,396,252	50.38
Internet	22,963	48,436	4.22

5 Description of the parameters

See Table 4 for descriptions of commonly used parameters. Other parameters are with the function descriptions.

Table 4: Description of the parameters

Parameter	**Description**
input graph/output graph/template graph	graph files in Galib format
config file	configuration file for the modules
sampling ratio	ratio of sample size to the original problem size ranging from 0 to 1, to compute an approximate solution (depends on the problem to be computed)
sample size	number of vertices or edges or other properties of the original graph to be sampled (depends on the problem to be computed)

6 Detail descriptions of the modules

6.1 cgraph

This program performs various small tasks on an undirected graph such as checking consistency, counting component, compute various statistics, etc. (c stands for check, compute, count, convert, consistency, component). Use different switch for different operation.

Usage:

```
cgraph  -cdnuCDLE  <input graph>
```

The program takes total 2 parameters as input.

1. `-cdnuCDLE`: These are switches.
   (a) `-c`: It checks for graph consistency, i.e., if a given input file contains a valid undirected graph.
   (b) `-d`: It measures degree statistics and outputs number of vertices, number of edges, average degree, maximum degree and minimum degree.
   (c) `-n`: It produces count file for EpiFast, must be with -u (not included yet).
   (d) `-u`: It produces "uel" or undirected edge list file (not included yet).
   (e) `-C`: It counts component (not included yet).
   (f) `-DL`: It lists of node degrees in $< id, deg >$ format.
   (g) `-DE`: For each edge (u, v), it outputs one line containing $< deg(u), deg(v) >$ where $deg(u) <= deg(v)$.
   (h) `-DS`: It outputs degree sequence in $< n, d_1, d_2, \ldots, d_n >$ format.
2. `<input graph>`: This is the input graph in Galib format.

Running time:

It takes about 3 min 29 seconds to read the Miami graph in and check the graph with the "-c" option.

6.2 cneighbor

Find the distribution of the number of common neighbors of the pairs of nodes at a given distance. Here, distance means the number of edges on the shortest path between the two nodes. For a given distance d (1 or 2), find the pairs of nodes such that the distance between two nodes in each pair is d and determine the number of common neighbors between the nodes in each such pairs. Then output distribution $f(x)$, where $f(x)$ is the number of pairs having x common neighbors. Note that for $d > 2$, the number of common neighbors is always zero for any pair of nodes.

Usage:

```
cneighbor  <input graph>  <distance>  <sample size>
```

The program takes total 3 parameters as input.

1. `<input graph>`: This is the input graph in Galib format.
2. `<distance>`: It is an integer denoting the distance (either 1 or 2).

3. `<sample size>`: It denotes the number of pairs of vertices used in computing $f(x)$.

Running time:

It takes about 3 min 22 seconds to read the Miami graph in and compute the common neighbors. Parameters used are $d = 2$ and $< sample\ size >= 100000$.

6.3 component

This program counts the total number of connected components in the given graph and the size of each component. Size of a component is the number of nodes in the component.

Usage:

```
component  <input graph>
```

It takes only 1 parameter as input.

1. `<input graph:>` This is the input graph in Galib format.

Running time:

It takes about 3 min 8 seconds to read the Miami graph in and compute the components.

6.4 shuffle

An "edge shuffle" operation replaces two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$. Such a shuffle is allowed only if it does not create any parallel edges or loop. This program repeatedly picks a pair of random edges (uniformly at random) and swaps the end points. This operation is continued until a given fraction of the edges are shuffled. The individual node's degree remains same after shuffling.

Usage:

```
shuffle <input graph>  <output graph>  <fraction>
```

The program takes total 3 command line arguments as input.

1. `<input graph>`: It is the input graph in Galib format.
2. `<output graph>`: It is the output graph name where the new graph will be written after edge shuffling.

3. `<fraction>`: The parameter's data type is double, ranging $0 \leq fraction \leq 1$. The program shuffles (<fraction> $\times 100$)% of the total edges of the <input graph> and writes the resultant graph in the file name <output graph>.

Running time:

It takes about 14 min 34 seconds to read the Miami graph and shuffle with a <fraction> of 0.99.

6.5 dshuffle

In degree-assortative shuffle of the edges of a graph, the degrees of the end points of the edges are maintained. Let $d(v)$ be the degree of the node v. Shuffle of the two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ are allowed only if $d(u_1) = d(u_2)$ and $d(v_1) = d(v_2)$ (in addition to the conditions of the regular shuffle). As the degree of old neighbor is the same as new neighbor, it ensures that for any given nodes, the degrees of its neighbors remain unchanged.

Usage:

```
dshuffle <input graph>  <output graph>  <fraction> <check_graph>
```

The program takes total 4 command line arguments as input.

1. `<input graph>`: It is the input graph in Galib format.
2. `<output graph>`: It is the output graph name where the new graph will be written after edge shuffling.
3. `<fraction>`: The parameter's data type is double, ranging $0 \leq fraction \leq 1$. The program shuffles (<fraction> $\times 100$)% of the total edges of the <input graph> and writes the resultant graph in the file name <output graph>.
4. `<check_graph>`: Here, <check_graph> is "Y/N". "Y" means "Yes", check the consistency of the new graph after shuffling. "N" means "No", do not check for consistency after shuffling.

Running time:

It takes about 6 min 41 seconds to read the Miami graph and shuffle with a <fraction> of 0.1.

6.6 gengraph

This program generates random graphs using various models: Chung-Lu, $G(n, p)$, etc. The current implementation can generate Chung-Lu graphs only. Other models have not been implemented yet. From a given degree sequence, Chung-Lu model generates a graph such that the expected degree of each node is same as it's degree in the given sequence.

Usage:

```
gengraph <deg_seq file> <output graph> [<in_uel file>]
```

The program always takes the first 2 command line arguments as input and the 3rd parameter is optional.

1. `<deg_seq file>`: This is the input file containing a given degree sequence. The format of the file is: $< n, d_1, d_2, \ldots, d_n >$, where n is the total number of nodes and d_i is the degree of the i-th node.

2. `<output graph>`: It is the output graph name where the new graph will be written.

3. `[<in_uel file>]`: This is an optional input parameter. A graph file in undirected edge list (uel) format, can be specified. If specified, the distribution of contact durations in this input graph will be used to randomly generate contact duration for the output graph. If not specified, an unweighted graph (with contact duration 1 for each edge) will be generated.

6.7 likelihood

Likelihood of a graph, $G = (V, E)$, is defined as a summation of a quantity for all edges $(u, v) \in E$ in a graph. The quantity is the multiplication of the degree of vertex u and v. If $L(G)$ is the likelihood of graph G, and $d(u)$ denotes the degree of vertex u, then

$$L(G) = \sum_{(u,v)\in E} d(u)d(v) \tag{6.1}$$

Usage:

```
likelihood  <input graph>
```

The program takes a graph as the only input and outputs the likelihood.

1. `<input graph>`: It is the input graph in Galib format.

6.8 ClusterCo

This program computes exact and approximate clustering coefficients (CC) of the nodes and the distribution of CC. Clustering coefficient is a measurement of a vertex to quantify the possibility of forming triangles among it and any two of its neighbors and represents the likelihood of existence of tightly knit groups in the graph. In an unweighted graph, CC of a vertex v can be compute by Eq. 6.2

$$CC_v = \frac{NE(v)}{\binom{d(v)}{2}} \tag{6.2}$$

Here, $NE(v)$ is the number of edges among neighbors of node v and $d(v)$ is the degree of v.

Usage:

```
ClusterCo  <config file>
```

The config file contains input and output specification as shown below:

1. `Graph_file:` It is the input graph in Galib format, i.e., "../inputgraph/portland.gph".
2. `Output_file:` It is the output file name without extension. Extension will be added by the program, i.e., "../outputfiles/port-CC"
3. `Buckets:` The number of buckets in histogram, i.e., 50.
4. `Actual:` This option denotes whether to compute exact histogram. 0 means "No", 1 means "Yes".
5. `Approx:` This option denotes whether to compute approximate histogram. 0 means "No", 1 means "Yes".
6. `Error:` Specify the error bound for approximate hist, i.e., 0.01.
7. `All_node:` This option denotes whether to list approximate cc of all nodes. 0 means "No", 1 means "Yes".
8. `Gnuplot_path:` It is the path of the executable binary gnuplot. It must include the slash '/' at the last, i.e., "/home/simsci1/share/bin/".
   (a) If it is in the current directory use: "./".
   (b) If you want the OS to search the path by following PATH variable, keep it empty.
   (c) If you don't want to plot, instead of specifying path, use: "NOPLOT".
   (d) If you want to change the appearance of the plotted graph, its' titles, etc., edit the file "<output_file>.plt" and run gnuplot on your own.

The followings are the output files:

1. Actual histogram data : <output_file>.hst
2. Approx histogram data : <output_file>.aht
3. List of CC for all nodes : <output_file>.lst
4. Plot file used by gnuplot : <output_file>.plt
5. Plotted eps file : <output_file>.eps

The above sample configuration can be found in "/CFG/cluster.cfg".

6.9 streamcc

This program computes clustering coefficients (CC) of some sampled nodes and based on the clustering coefficients of those nodes, it computes approximate distribution (histogram) of CC. It can not compute exact CC. However, the benefit of using streamcc over ClusterCo is that it does not store the entire graph in the memory; rather it makes several passes through the graph file and only stores a small portion of the graph in the memory at any given moment. Thus streamcc is very useful for large graphs that cannot fit into memory. The sample size is assumed to be $O(B \lg n)$, where B and n are the number of buckets and the number of nodes respectively.

Usage:

```
streamcc  <config file>
```

The input and output specification for streamcc also given in a config file. A sample configuration can be found in "/CFG/streamcc.cfg".

6.10 streamccT

Very similar to streamcc and also need to run in the same way. Only difference is that it enumerates the sampled triangles, in addition to computation of CC distribution.

6.11 parse

ParSE (parallel subgraph enumeration) is an approximated algorithm to count the number of non-induced subgraphs which are isomorphic to a given template. The template is partitioned to two sub-templates and counted locally in each computing nodes using color coding approximation. Then the counts of the original template is acquired by aggregating the sub-template's counts. The code uses SGI MPI Library for communication. The program outputs the number of counts and running time to the standard output. Refer to "./Cfg/parse.conf" and "./Scripts/parse.qsub" for more usage information.

Usage:

```
mpiexec_mpt -n 16 parse parse.conf
```

Running time:

It takes about 61 min to read the Miami graph in and do the subgraph count. Number of processors is 400. Template has 6 node which consists of 2 triangles connected with an edge.

6.12 spath

This program computes the distribution of unweighted shortest path distance between two nodes. It computes exact or approximate (chosen by the user) distribution $f(x)$, where $f(x)$ is the number of pairs of the nodes with distance between them x.

Usage:

```
spath  <config file>
```

The input and output specification for spath is given in a config file. A sample configuration file can be found in "/home/NDSSL/projects/CINET/Cfg/shortest_path.cfg" in shadowfax.

6.13 CliqueCount

This program counts the number of cliques of a given size in the graph. A clique C in a graph $G = (V, E)$ is a subset of nodes, $C \subset V$, such that for any two nodes $u, v \in C$, we have $(u, v) \in E$.

Usage:

```
clique  <input graph>  <size>
```

It takes total 2 command line arguments as input.

1. `<input graph>`: It is the input graph in Galib format.
2. `<size>`: This is the size of the clique. The program finds the number of cliques C in the input graph such that $|C| = size$.

6.14 betweenness

Computes betweenness distribution of the nodes. Three sampling method can be selected, which are uniformly sampling, sampling proportional to degree, or no sampling(precise computation).

Usage:

```
betweenness  <config file>
```

"/Cfg/betweenness.conf" is a sample configuration file. Upto 3 files will be output according to configurations. The output files are:

1. The first output file is the raw data file, i.e., betweenness for each node.
2. This file contains the distribution of the betweenness.
3. The third one is a gnuplot script file.

Running time:

It takes about 39 hours to read the graph in and do the betweenness calculation. Graph used for testing is Miami. 1% out of 2 million nodes are sampled in the computing.

6.15 DegreeDistribution

The program computes the distribution on the nodes degree, i.e., the program maintains a count of number of nodes having the same degree. The program outputs a gnuplot file, showing the degree distribution. Multiple contact graph files can be given as input.

Usage:

```
degree  <config file>
```

The input and output specification is given in a config file. A sample configuration file can be found in the path "/home/NDSSL/projects/CINET/Cfg/degree.cfg" in shadowfax.

6.16 RoDistribution

The program computes a distribution of nodes based on the Ro values, i.e. the program maintains a count of number of nodes having the same Ro. Ro is calcaulated using the formula $Ro(v) = \sum_{c \in S(v)}(1 - e^{-\tau * w(c)}$, where $S(v)$ is a set of outgoing edges from node v and τ value is an input. The program outputs a gnuplot file, that shows the Ro distribution. Multiple contact graph files can be given as input. Multiple tau values also can be given as input.

Usage:

```
ro  <config file>
```

The input and output specification is given in a config file. A sample configuration file can be found in the path "/home/NDSSL/projects/CINET/Cfg/ro.cfg" in shadowfax.

6.17 Single Source Shortest Path (sssp)

This program computes weighted shortest path distance from a source node to all other nodes in an undirected graph using Dijkstra's algorithm. A shortest path is a simple path from the source node to a destination node with minimum weighted distance.

Usage:

```
sssp  <graph_name> <source_node> <output_file>
```

It takes the following 3 command line arguments as input.

1. `<graph_name>`: It is an undirected weighted input graph in Galib format. It can be either connected or disconnected graph.

2. `<source_node>`: It is the source node from where the shortest path distances will be computed. It is an integer input. There are two options for the $< source\_node >$:

   (a) Typically, the $< source\_node >$ is a node in the graph. If the graph is connected, then the shortest path distances from the source node to all other nodes are computed and saved in the $< output\_file >$ in the format $< node\_id >< distance\_from\_source\_node >$. If the graph is disconnected, then the shortest path distances from the source node to all nodes in the connected component containing the source node will be computed and the result is saved in the $< output\_file >$. The other nodes, which are not connected to the source node, are ignored in this case. If the source node itself is isolated from the rest of the graph, then it is detected, the following message is shown in the terminal: "Graph is disconnected. The source node (...) is isolated from the rest of the graph ..." and no output file has been generated.

   (b) The user can also specify -1 for source node. If so, the source node is selected randomly. If the graph is disconnected, then one source node is selected randomly in every connected component (except the isolated nodes), the corresponding shortest path distances of the nodes in that component are computed and results are saved in output files. In this case, there may be multiple output files, one for each connected component (except the isolated nodes). If every node in the graph is isolated, then it is detected, and the following message is shown in the terminal: "Every node in the graph is isolated ..." and the program exits without generating any output file.

3. `<output_file>`: It is the output filename. If an undirected disconnected weighted graph contains total 6 components and 2 are isolated nodes among them, "op.txt" is given as input for the $< output\_file >$ parameter and the $< source\_node >$ input is -1, then total 4 output files will be generated: "op.txt", "op1.txt", "op2.txt" and "op3.txt". If the $< output\_file >$ contains no extension (no dot)

in the name, e.g. "op", then the output files will be: "op", "op1", "op2" and "op3" for the above example. If the $< source\_node >$ input is some specific node in the graph that is not isolated, then only 1 output file will be generated containing the shortest path distances of all the nodes in the component containing the source node. A node's shortest path distance from the source node is written in the format: $< node\_id >< distance\_from\_source\_node >$ in the output file.

6.18 Shortest Path Tree (spt)

This program computes shortest path tree (or forest) from a source node to all other nodes in an undirected graph using Dijkstra's algorithm and output the edges in the tree. A shortest path tree is a tree where the source node is the root and all other nodes in that tree are the other nodes in that component that are reachable from the source node or root. A path from the root to any node in the tree is the shortest path from the source node to that node in the graph. This is very similar to the single source shortest path distance program. Here, just the edges are given as output in "uel" or "undirected edge list" file format.

Usage:

```
spt  <graph_name> <source_node> <output_graph_name>
```

It takes the following 3 command line arguments as input.

1. `<graph_name>`: It is an undirected weighted input graph in Galib format. It can be either connected or disconnected graph.

2. `<source_node>`: It is the source node from where the shortest path tree will be computed. It is an integer input. There are two options for the $< source\_node >$:

   (a) Typically, the $< source\_node >$ is a node in the graph. If the graph is connected, then the shortest path tree from the root (source node) to all other nodes are computed and saved in the $< output\_graph\_name >$ in the format $< u >< label(u,v) >< v >< label(u,v) >< weight(u,v) >$ such that there is an edge (u, v) in the shortest path tree with $label(u, v)$, $weight(u, v)$ and u is the parent of v. If the graph is disconnected, then the shortest path tree from the source node to all nodes in the connected component containing the source node will be computed and the result is saved in the $< output\_graph\_name >$. The other nodes, which are not connected to the source node, are ignored in this case. If the source node itself is isolated from the rest of the graph, then it is detected, the following message is shown in the terminal: "Graph is disconnected. The source node (...) is isolated from the rest of the graph ..." and no output file has been generated.

(b) The user can also specify -1 for source node. If so, the source node is selected randomly. If the graph is disconnected, then one source node is selected randomly in every connected component (except the isolated nodes), the corresponding shortest path tree in that component is computed and results are saved in output files. In this case, there may be multiple output files, one for each connected component (except the isolated nodes). If every node in the graph is isolated, then it is detected, and the following message is shown in the terminal: "Every node in the graph is isolated ..." and the program exits without generating any output file.

3. `<output_graph_name>`: It is the output filename. As the output is written in "uel" or "undirected edge list" format, it is expected that the $< output\_graph\_name >$ will contain ".uel" at the end of the filename. If the $< output\_graph\_name >$ does not contain ".uel" at the end, then it is appended at the end of the filename. For example, if an undirected disconnected weighted graph contains total 6 components and 2 are isolated nodes among them, "op.uel" or "op" is given as input for the $< output\_graph\_name >$ parameter, the $< source\_node >$ input is -1, then there will be total 4 output files: "op.uel", "op1.uel", "op2.uel" and "op3.uel". If the $< source\_node >$ input is some specific node in the graph that is not isolated, then only 1 output file will be generated for the shortest path tree of the component containing the source node. Every edge (u, v) in the shortest path tree with $label(u, v)$ and $weight(u, v)$, where u is the parent of v, is written in the output file in the following format: $< u >< label(u, v) >< v >< label(u, v) >< weight(u, v) >$.

6.19 Minimum Spanning Tree (mst)

This program computes the minimum spanning tree (or forest) of an undirected graph using Prim's algorithm and output the edges in the tree. A minimum spanning tree of a connected undirected graph is an acyclic subset of edges that connects all the vertices such that the summation of the weights of edges is minimized. There may be multiple minimum spanning trees in a disconnected graph. In that case, one tree is written in one file. Here, the edges of the tree are given as output in "uel" or "undirected edge list" file format.

Usage:

```
mst  <graph_name> <source_node> <output_graph_name>
```

It takes the following 3 command line arguments as input.

1. `<graph_name>`: It is an undirected weighted input graph in Galib format. It can be either connected or disconnected graph.

2. `<source_node>`: It is the source node from where the minimum spanning tree will be computed. It is an integer input. The total weight of edges in the minimum

spanning tree is same for a connected component irrespective of the source node. There are two options for the $< source\_node >$:

(a) Typically, the $< source\_node >$ is a node in the graph. If the graph is connected, then the minimum spanning tree from the root (source node) to all other nodes are computed and saved in the $< output\_graph\_name >$ in the format $< u >< label(u, v) >< v >< label(u, v) >< weight(u, v) >$ such that there is an edge (u, v) in the minimum spanning tree with $label(u, v)$, $weight(u, v)$ and u is the parent of v. If the graph is disconnected, then the minimum spanning tree from the source node to all nodes in the connected component containing the source node will be computed and the result is saved in the $< output\_graph\_name >$. The other nodes, which are not connected to the source node, are ignored in this case. If the source node itself is isolated from the rest of the graph, then it is detected, the following message is shown in the terminal: "Graph is disconnected. The source node (...) is isolated from the rest of the graph ..." and no output file has been generated.

(b) The user can also specify -1 for source node. If so, the source node is selected randomly. If the graph is disconnected, then one source node is selected randomly in every connected component (except the isolated nodes), the corresponding minimum spanning tree in that component is computed and results are saved in output files. In this case, there may be multiple output files, one for each connected component (except the isolated nodes). If every node in the graph is isolated, then it is detected, and the following message is shown in the terminal: "Every node in the graph is isolated ..." and the program exits without generating any output file.

3. `<output_graph_name>`: It is the output filename. As the output is written in "uel" or "undirected edge list" format, it is expected that the $< output\_graph\_name >$ will contain ".uel" at the end of the filename. If the $< output\_graph\_name >$ does not contain ".uel" at the end, then it is appended at the end of the filename. For example, if an undirected disconnected weighted graph contains total 6 components and 2 are isolated nodes among them, "op.uel" or "op" is given as input for the $< output\_graph\_name >$ parameter, the $< source\_node >$ input is -1, then there will be total 4 output files: "op.uel", "op1.uel", "op2.uel" and "op3.uel". If the $< source\_node >$ input is some specific node in the graph that is not isolated, then only 1 output file will be generated for the minimum spanning tree of the component containing the source node. Every edge (u, v) in the minimum spanning tree with $label(u, v)$ and $weight(u, v)$, where u is the parent of v, is written in the output file in the following format: $< u >< label(u, v) >< v >< label(u, v) >< weight(u, v) >$.

6.20 Weighted Shortest Path (wsp)

This program computes average shortest path distance, weighted diameter and distribution of shortest path distances for an undirected weighted graph in Galib format, with the option of approximate and exact calculation. The input graph is assumed to be connected. For the average shortest path distance, distances between all pairs of nodes are calculated and average is taken. When approximate calculation is done, only some sample set of pairs of nodes are considered and average is taken and for exact calculation, all pairs are considered. Weighted diameter is the maximum distance (in terms of weight) between any pair of nodes. The distribution of shortest path distances computes the distribution of shortest path distances among pairs of nodes by maintaining the count of the pairs of nodes having the same weighted distance.

Usage:

```
wsp  <graph_name> -awd <output_file> <#buckets> -ae
```

The program always takes total 5 command line arguments as input.

1. `<graph_name>`: The $< graph\_name >$ is an undirected weighted input graph in Galib format. It must be a connected graph. The scenario for disconnected graph is not handled here.

2. `-awd`: They are switches and provide the option of calculating any combination of the average shortest path distance, weighted diameter and distribution of shortest path distance to the user. "-" at the beginning of the second parameter, followed by at least one character from "*awd*", is required. Any combination of "*awd*" is allowed with the beginning of "-". For example, only for average shortest path calculation, it should be "-a", for both weighted diameter and distribution calculation, it should be "-wd" or "-dw", and for computing all the 3 measures, it can be "-awd" or any other valid combination.

   (a) `a`: It denotes average shortest path distance calculation. Output is given to terminal.

   (b) `w`: It denotes weighted diameter calculation. Output is given to terminal.

   (c) `d`: It denotes distribution of shortest path calculation. Output is saved in file.

3. `<output_file>`: It is the output filename. The average shortest path distance and weighted diameter output are given at the terminal. Output is saved in file only for the distribution of shortest path distances. If the user does not want to calculate the distribution (wants only to calculate any combination of average shortest path and weighted diameter), still the $< output\_file >$ input is mandatory. If the input for $< output\_file >$ is "distribution", then corresponding graph showing the plot of weight of the shortest path distance versus the number of pairs of nodes, is plotted in file "distribution.eps". Here, we plot the

midpoint of the bucket for weighted shortest path distance and the number of pairs of nodes in normalized form. There are some other auxiliary output files. The distribution of shortest path distances are saved in file "distribution.aht" for approximate calculation and in "distribution.hst" for exact calculation, the plot was generated using "gnuplot" commands, and these commands are saved in file "distribution.plt" and the corresponding log file for gnuplot is saved in "distribution_plot.log".

4. `<#buckets>`: It is an integer input denoting the number of buckets. The value must be greater than or equal to 2.

5. `-ae`: These are switches and provide the option of approximate and exact calculation to the user. "-" at the beginning of the fifth parameter, followed by at least one character from "*ae*", is required. Any combination of "*ae*" is allowed with the beginning of "-". Only for approximate calculation, it should be "-a", for only exact calculation, it should be "-e" and for both approximate and exact calculation, it should be "-ae" or "-ea".

   (a) `a`: It denotes approximate calculation. If n is the total number of nodes and B is the total number of buckets, then total $min(n, 8B \ln n)$ number of iterations (one single source shortest path is run at every iteration) are used for calculating the approximate result. At every iteration, the source node is selected randomly.

   (b) `e`: It denotes exact calculation. Every node is selected as source node and single source shortest path is run, thus making total n iterations.

6.21 Remove nodes (rmnode)

The purpose of this program is to generate new graph from an input graph after removing nodes specified in another input file. By removing nodes, we mean that the nodes along with their adjacent edges will be removed. The remaining graph after node removal will be written in the output file.

Usage:

```
rmnode <graph_name> <filename_containing_nodes> <output_graph_name>
```

It takes total 3 command line arguments as input. Among them, 2 are input filenames and 1 is output filename.

1. `<graph_name>`: It is the input graph name from where the nodes will be removed. It is an undirected graph in Galib format.

2. `<filename_containing_nodes>`: It is the input file name containing the nodes to be removed. It may contain some nodes that do not exist in the original graph. In this case, they are not removed as they do not exist in the original graph and

the list of these nodes are saved in a log file. If a node is requested to remove multiple times, then only the first request is served and the other requests are ignored.

3. `<output_graph_name>`: It is the output file name where the new graph will be written. The new graph after node removal is written in Galib format and the $<$ *output_graph_name* $>$ parameter is expected to contain ".gph" at the end. If not, then ".gph" is added to the end of the output filename. Two new files will be created as output. If "afternoderemoval" or "afternoderemoval.gph" is the input for $<$ *output_graph_name* $>$, then "afternoderemoval.gph" is the new graph generated after node removal and "afternoderemoval.log" is the log file containing the nodes that were requested to remove but failed (the nodes that do not exist in the original graph). If the new graph is an empty graph, then the following message is shown in the terminal: "Empty graph. Nothing to write in file" and no ".gph" file has been generated. Some message will also be displayed in the terminal for the user showing how many nodes were requested to remove and how many of them were actually removed.

6.22 Remove high-degree nodes: degree $\geq$ x (rmnodedegree)

The purpose of this program is to generate new graph from an input graph after removing nodes having degrees higher than or equal to a specified input number. By removing nodes, we mean that the nodes along with their adjacent edges will be removed. The remaining graph after node removal will be written in the output file.

Usage:

```
rmnodedegree <graph_name> <degree> <output_graph_name>
```

It takes total 3 command line arguments as input.

1. `<graph_name>`: It is the input graph name from where the nodes will be removed. It is an undirected graph in Galib format.

2. `<degree>`: It is an integer input denoting that all the nodes having degrees higher than or equal to this number will be removed from the graph.

3. `<output_graph_name>`: It is the output file name where the new graph will be written. The new graph after node removal is written in Galib format and the $<$ *output_graph_name* $>$ parameter is expected to contain ".gph" at the end. If not, then ".gph" is added to the end of the output filename. Two new files will be created as output. If "afternoderemoval" or "afternoderemoval.gph" is the input for $<$ *output_graph_name* $>$, then "afternoderemoval.gph" is the new graph generated after node removal and "afternoderemoval.log" is the log file containing the nodes that were removed from the original graph. Some message will also be displayed in the terminal for the user showing how many nodes were actually removed and the new graph contains how many nodes.

6.23 Remove high-degree nodes: top x% (rmnodepercentage)

The purpose of this program is to generate new graph from an input graph after removing the topmost $x\%$ nodes having higher degrees. By removing nodes, we mean that the nodes along with their adjacent edges will be removed. The remaining graph after node removal will be written in the output file.

Usage:

```
rmnodepercentage <graph_name> <x> <output_graph_name>
```

It takes total 3 command line arguments as input.

1. `<graph_name>`: It is the input graph name from where the nodes will be removed. It is an undirected graph in Galib format.

2. `<x>`: It is a float input in the range $(0, 100)$. The topmost $x\%$ nodes having higher degrees are removed from the original graph.

3. `<output_graph_name>`: It is the output file name where the new graph will be written. The new graph after node removal is written in Galib format and the $<$ *output_graph_name* $>$ parameter is expected to contain ".gph" at the end. If not, then ".gph" is added to the end of the output filename. Two new files will be created as output. If "afternoderemoval" or "afternoderemoval.gph" is the input for $<$ *output_graph_name* $>$, then "afternoderemoval.gph" is the new graph generated after node removal and "afternoderemoval.log" is the log file containing the nodes that were removed from the original graph. Some message will also be displayed in the terminal for the user showing how many nodes were removed and the new graph contains how many nodes.

6.24 Remove edges (rmedge)

The purpose of this program is to generate new graph from an input graph after removing edges specified in another input file. By removing edge, we mean that the edge is removed from both end nodes' adjacency lists. The remaining graph after edge removal will be written in the output file.

Usage:

```
rmedge <graph_name> <filename_containing_edges> <output_graph_name>
```

It takes total 3 command line arguments as input. Among them, 2 are input filenames and 1 is output filename.

1. `<graph_name>`: It is the input graph name from where the edges will be removed. It is an undirected graph in Galib format.

2. `<filename_containing_edges>`: It is the input file name containing the edges to be removed. It may contain some edges that do not exist in the original graph. In this case, they are not removed as they do not exist in the original graph and the list of these edges are saved in a log file. If an edge is requested to remove multiple times, then only the first request is served, the other requests are ignored and saved in the log file.

3. `<output_graph_name>`: It is the output file name where the new graph will be written. The new graph after edge removal is written in Galib format and the $< output\_graph\_name >$ parameter is expected to contain ".gph" at the end. If not, then ".gph" is added to the end of the output filename. Two new files will be created as output. If "afteredgeremoval" or "afteredgeremoval.gph" is the input for $< output\_graph\_name >$, then "afteredgeremoval.gph" is the new graph generated after edge removal and "afteredgeremoval.log" is the log file containing the edges that were requested to remove but failed (the edges that do not exist in the original graph or were requested multiple times to remove). If the new graph is an empty graph, then the following message is shown in the terminal: "Empty graph. Nothing to write in file" and no ".gph" file has been generated. Some message will also be displayed in the terminal for the user showing how many edges were requested to remove and how many of them were actually removed.

6.25 Generate Complete Graph (ggraph complete)

This program generates a new complete graph in Galib format. The number of nodes in the graph must be greater than or equal to 1.

Usage:

```
ggraph complete <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `complete`: The first parameter must be "complete" that tells the program to generate a new complete graph.

2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.

3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 1$. For $n < 1$, an error message is shown in the terminal and no output file will be generated.

6.26 Generate Chain or Path Graph (ggraph chain)

This program generates a new chain or path graph in Galib format. The number of nodes in the graph must be greater than or equal to 2.

Usage:

```
ggraph chain <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `chain`: The first parameter must be "chain" that tells the program to generate a new chain graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 2$. For $n < 2$, an error message is shown in the terminal and no output file will be generated.

6.27 Generate Star Graph (ggraph star)

This program generates a new star graph in Galib format. The number of nodes in the graph must be greater than or equal to 1.

Usage:

```
ggraph star <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `star`: The first parameter must be "star" that tells the program to generate a new star graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 1$. For $n < 1$, an error message is shown in the terminal and no output file will be generated.

6.28 Generate Cycle or Circle Graph (ggraph cycle)

This program generates a new cycle graph in Galib format. The number of nodes in the graph must be greater than or equal to 3.

Usage:

```
ggraph cycle <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `cycle`: The first parameter must be "cycle" that tells the program to generate a new cycle graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 3$. For $n < 3$, an error message is shown in the terminal and no output file will be generated.

6.29 Generate Wheel Graph (ggraph wheel)

This program generates a new wheel graph in Galib format. The number of nodes in the graph must be greater than or equal to 4.

Usage:

```
ggraph wheel <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `wheel`: The first parameter must be "wheel" that tells the program to generate a new wheel graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 4$. For $n < 4$, an error message is shown in the terminal and no output file will be generated.

6.30 Generate Binary Tree Graph (ggraph btree)

This program generates a new binary tree graph in Galib format. The number of nodes in the graph must be greater than or equal to 3.

Usage:

```
ggraph btree <output_graph_name> <#nodes>
```

It takes total 3 command line arguments as input.

1. `btree`: The first parameter must be "btree" that tells the program to generate a new binary tree graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#nodes>`: The total number of nodes (n) in the graph. In this case, $n \geq 3$. For $n < 3$, an error message is shown in the terminal and no output file will be generated.

6.31 Generate Grid Graph (ggraph grid)

This program generates a new grid graph of dimension ($row \times column$) in Galib format. The number of rows and columns must be greater than or equal to 2.

Usage:

```
ggraph grid <output_graph_name> <#row> <#col>
```

It takes total 4 command line arguments as input.

1. `grid`: The first parameter must be "grid" that tells the program to generate a new grid graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. The dimension of this graph will be ($\#row \times \#col$). It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#row>`: The number of rows in the graph must be ≥ 2. For $row < 2$, an error message is shown in the terminal and no output file will be generated.
4. `<#col>`: The number of columns in the graph must be ≥ 2. For $col < 2$, an error message is shown in the terminal and no output file will be generated.

6.32 Generate Torus Graph (ggraph torus)

This program generates a new torus graph of dimension ($row \times column$) in Galib format. The number of rows and columns must be greater than or equal to 3.

Usage:

```
ggraph torus <output_graph_name> <#row> <#col>
```

It takes total 4 command line arguments as input.

1. `torus`: The first parameter must be "torus" that tells the program to generate a new torus graph.
2. `<output_graph_name>`: It is the output graph name which will contain the new graph in Galib format. The dimension of this graph will be ($\#row \times \#col$). It is expected to contain ".gph" at the end of the filename. No output file has been generated for wrong input.
3. `<#row>`: The number of rows in the graph must be ≥ 3. For $row < 3$, an error message is shown in the terminal and no output file will be generated.
4. `<#col>`: The number of columns in the graph must be ≥ 3. For $col < 3$, an error message is shown in the terminal and no output file will be generated.

6.33 BFS forest (bfsForest)

bfsForest module is intended for generation of BFS forest from an input graph. A BFS forest consists of one or more BFS tree. A BFS tree is, informally, a tree obtained from BFS traversal on a connected component of the given graph. A graph may be connected itself or may have multiple connected components. For the first case, BFS forest contains a single tree and for the later case, we will have multiple BFS trees from the input graph.

Usage:

```
bfsforest <graph-file name> <source node> <output-file>
```

It takes total 3 command line arguments as input.

1. `graph-file name`: It is the input graph file in Galib format.
2. `Source node`: It is the root or source vertex from where BFS traversal should start on. If an user provides such a vertex, then the module will return a BFS tree starting from that source vertex. Otherwise, if the user wants all the trees (forest) to be generated automatically from BFS traversal on the given input graph, he/she should put a value of -1 for the parameter.
3. `output-file`: It specifies the file-name (possibly with path) into which output tree will be written. If the forest contains more than one tree, then the same directory will contain all the trees named after the original output-file name with numbering.

6.34 BFS Partition (bfsPart)

Given, an input graph and number of partitions, the module conduct BFS traversal on the graph and generate partitions based on the traversal. By the process, it tends to keep neighboring elements in the same partition. Each partition contains a disjoint set of vertices (core vertices). Besides, the partition include one extra layer of vertices (i.e. neighbors of core vertices). For certain applications (determining Cluster-Co), this overlapping extra layer of vertices is necessary. For each partition, the module generates a output-file (with extension '.lst') specifying the core vertices of the given partition and also a graph file (with extension '.gph') containing the cores and extra layers.

Usage:

```
bfspart <graph-file name> <number of partitions> <output-file name>
```

It takes total 3 command line arguments as input.

1. `graph-file name`: It is the input graph file in Galib format.
2. `number of partitions`: It is the intended number of partitions user want to have from the given input graph.
3. `output-file`: It specifies the file-name (possibly with path) into which output partitions will be written. For multiple partitions, the same directory will contain all the partitions named after the original output-file name with numbering. Obviously for each partitions, there are two output files (.lst and .gph).

6.35 Component Graphs (compGraph)

compGraph module generates component graphs from an input graph. If the graph is fully connected, then the component graph is basically the same input graph. But, when the graph contains multiple connected components, then the module generates graph file for each of the components. At the same time, the module also facilitates the users for learning different information of interest about the graph, such as number of connected component, size of each components etc.

Usage:

```
compgraph  -cnd <graph-file name> <output-file name>
```

It takes total 3 command line arguments as input.

1. `cnd`: These are switches.
    - -c: check whether the graph is connected
    - -n: the number of connected component

- -d: distribution of the component sizes

2. `graph-file name:` It is the input graph file in Galib format.

3. `output-file name:` It specifies the file-name (possibly with path) into which output component graphs will be written. For multiple components, the same directory will contain all the components named after the original output-file name with numbering. So, for each component, there is one output file (extension .gph).

6.36 Check if a degree sequence is graphical (graphical)

graphical module checks whether a degree sequence is graphical. Informally, a degree sequence is graphical if a graph can be constructed satisfying the degree information. Alternatively, a graphical sequence is a sequence of numbers which can be the degree sequence of some graph. This modules uses Havel-Hakimi theorem to check whether the sequence of numbers (degrees) is graphical.

Usage:

```
graphical  <degree sequence file>
```

It takes one command line argument as input.

1. `degree sequence file:` It is the name of input degree-sequence file. The file contains number of degrees followed by that many integer numbers (degrees).

Post-execution:
After running the module, a message will be displayed on standard output indicating whether the input sequence is graphical ('Graphical') or not ('Not Graphical').

6.37 Parallel Clustering Coefficient (ParCC)

ParCC module provides a faster computation of local clustering coefficients (cluster-co) of vertices as well as average clustering coefficient of the input graph using parallel computing environment. Local cluster-co is the measure of how heavily neighbors of a particular vertex are connected. Global (average) cluster-co provides an average measure of cluster-co over the whole graph.

Usage:

```
mpirun -np <number of processor> .\parcc <graph-file name>  <output-file>
```

As this is a parallel module, the execution command also specifies some parallel execution variables like number of processors. Also, this sort of specification is dependent on particular clusters where it is being executed. `mpirun -np <number of processor>` is needed when it is running from terminal. For, qsub job submission, syntax is a bit differnt. `<number of processor>` specifies intended number of processors to be used for the execution of the module. The module uses two input parameters.

1. `graph-file name:` It is the input graph file in Galib format.

2. `output-file:` It is the name of the file into which local cluster-co will be written.

Post-execution:
After running the module, output-file will contain local cluster-co information. The file is structured as having two number in each line- first one is the vertex and second one is its cluster-co. Number of lines is, definitely, equal to number of vertices of the input graph. Global cluster-co will be displayed on standard output. As usual, some informative messages will also be displayed on standard output.

6.38 External Memory Graph Conversion, uel to gph (Euel2gph)

This modules converts graph format using external memory instead of loading the whole graph into memory. *Euel2gph* converts edge list format into galib graph format. For precise specification of edge list format and galib graph format, see the earlier section of this manual.

Usage:

```
Euel2gph <graph-file name>  <output-file>
```

The module takes two parameters from command line.

1. `graph-file name:` It is the input file in undirected edge list format (uel).

2. `output-file:` It is the name of the output file where the converted graph will be written in Galib format.

Post-execution:
After running the module, output-file will be written on specified output file. As usual, some informative messages will also be displayed on standard output.