Graph Generator Design Document

Parameters

The Graph Generator will take in a series of parameters that will be used to create test graphs. These test graphs will be used as input for the AFGMiner algorithm in order to fully assess this algorithm as to what which types of graphs it works well for as well as defining domain areas where this algorithm will succeed.

The tests done will involve manipulating the input parameters and observing the associated changes in output.

The necessary parameters are used to (a) create the test graphs or (b) create the AFGMiner configuration file.

The parameters inputted to Graph Generator will be the following (all numbers are integers unless noted. Distributions are U,E,G,or P):

Parameters for Graph Generation

- Size of attribute superset [1, ∞)
- # of nodes [1, ∞)
- # of EFG's [1, ∞)
- Attribute Distribution (Uniform, Exponential, Gaussian, Poisson)
- Attribute Weight Distribution (Uniform, Exponential, Gaussian, Poisson)
- Node Weight Distribution (Uniform, Exponential, Gaussian, Poisson)
- Edge Weight Distribution (Uniform, Exponential, Gaussian, Poisson)
- Node Out-Degree Distribution (Uniform, Exponential, Gaussian, Poisson)

Parameters for AFGMiner Configuration File

- Maximum attributes per pattern [1, ∞)
- Maximum forward edges per pattern [0, ∞) (0=∞)
- Maximum backward edges per pattern [0, ∞) (0=∞)
- Maximum Gap [0, ∞)
- MinSupport [0,1] REAL NUMBER
- Maximum # of Nodes per pattern [1, ∞)
- Threads? [0, ∞)

The output measurements will be:

- # of subgraphs mined total?
- # of subgraphs above minSupport
- # of n-subgraphs above minSupport (n = number of edges)
- Largest support value

Distributions

Uniform (p(x) = $\frac{1}{B-A}$)

- A = Minimum (Positive Integer)
- B = Maximum (Positive Integer)

Exponential (p(x) = $\lambda e^{-\lambda x}$)

- λ = rate parameter [0,∞)

Gaussian (p(x) = $ae^{\frac{-(x-b)^2}{2c^2}}$)

- a = height/max probability (0,1]
- b = center/mode [0,∞)
- c = width/standard deviation (0,∞)

Poisson (p(x) = $\frac{e^{-\lambda}\lambda^x}{x!}$)

- λ = mean

Classes

Main

- Gather input, control flow of program
- Will have option to gather input via prompts or file

ConfigFileCreator

- Holds data necessary to create config file
- Will be responsible for creating the file to be used

UniformDist

- Creates numbers necessary for uniform distribution
- int max
- int min

ExponentialDist

- Creates numbers necessary for Exponential distribution
- double rateParameter (controls speed of decay)

GaussianDist

- Creates numbers necessary for Gaussian distribution
- double height
- double center
- double width

PoissonDist

- Creates numbers necessary for a Poisson distribution
- double mean

Graph

- Vector<ExecutionFlowGraph> flowGraphs

File Creator

- Creates the input file to be run in AFGMiner

We will use the graph implementation already present in AFGMiner to represent graphs. Therefore these do not need to be created but a summary is put below for reference. The exception is EFGVertexWithOutDegree which will be the same as in AFGMiner except with outDegree added:

EFGVertexWithOutDegree

- int nodeWeight
- int outDegree
- Bitset attributes //If i-th element of bitset is 1, i-th attribute exists
- LinkedHashMap <int index, double weight> attrWeights

EFGEdge

- EFGVertex fromNode
- EFGVertex toNode
- double edgeWeight

ExecutionFlowGraph

- LinkedHashMap<Pair<Long,Long>, EFGEdge> edges
- LinkedHashMap <Long, EFGVertex> nodes

Additional Data Constraints

To make the graphs, there are a few constraints that must be met to ensure proper execution

- $\Sigma nodeWeight > \Sigma attrWeight$

Input File Format

This file is for data inputted to the graph generator via a passed in file, rather than via prompts.

- Size of attribute superset [1, ∞) - for graph creation
- # of nodes [1, ∞) - for graph creation
- # of EFG's [1, ∞) - for graph creation
- Attribute Distribution (U,E,G,P)
- Necessary Parameters
    - If U
        - Min
        - Max
    - If E
        - Rate Parameter
    - If G
        - Height
        - Center
        - Width
    - If P
        - Mean
- Attribute Weight Distribution (U,E,G,P)
- Necessary Parameters (Same as previous)
- Node Distribution (U,E,G,P)
- Necessary Parameters (Same as previous)
- Node Weight Distribution (U,E,G,P)
- Necessary Parameters (Same as previous)
- Edge Weight Distribution (U,E,G,P)
- Necessary Parameters (Same as previous)
- Node Out-Degree Distribution (U,E,G,P)
- Necessary Parameters (Same as previous)
- Maximum attributes per pattern [1, ∞)
- Maximum forward edges [0, ∞) (0=∞)
- Maximum backward edges [0, ∞) (0=∞)
- Maximum Gap [0, ∞)
- Threads used to Mine [0,∞)
- MinSupport [0,1]
- Maximum # of Nodes per pattern [1, ∞)

Example Input File

Ignore \n, space, tab, comments

```
5      //Number of attributes
10     //Number of nodes
2      //Number of EFGs
U      //Attribute Distribution (Uniform)
1      //Min 1 attribute per node
5      //Max 5 attributes per node
E      //Attribute Weight Distribution (Exponential)
1.5    //Rate Parameter = 1.5
P      //Node Distribution (Poisson)
5      //Mean number of Nodes per EFG
G      //Node Weight Distribution (Gaussian)
0.6    //Height of 0.6 (Largest Probability)
2      //Center of 2 (2 is most likely weight)
1      //Width of 1 (68% of data will be within 1 unit from 2)
P      //Edge Weight Distribution (Poisson)
2      //Mean Edge Weight
U      //Node Out-Degree Distribution (Uniform)
0      //Min 0 out-edge per node
4      //Max 4 outward edges per node
5      //Max attributes per node
4      //Max forward edges
2      //Max backward edges
1      //Max Gap
1      //Number of Threads to Mine with
0.4    //MinSupport
6      //Max Nodes per pattern
```

Algorithm Flow

The Graph Generator will begin by prompting for input. Either it can be passed in by argument or entered in via prompts. This way scripts can be run for efficient tests but also makes the program accessible to be used for future tests.

After gathering all necessary parameters, graphs will be generated.This process is described below.
Graph Creation

Creating Execution Flow Graphs (EFGs)

1. A number of LinkedHashMaps, mapping a Long index with a node, equal to the number of EFG's will be created. Each of these LinkedHashMaps will be placed in a Vector called the Graph Vector.

Creating Nodes

1. A LinkedHashMap will be created, mapping a numerical ID with an EFGVertex. There will be a total of entries in this map equal to the inputted number of nodes.
2. Every node will be given a weight, determined by the applicable inputted distribution
3. Every node will be given an Attribute set. This is done through the creation of a Bitset of length equal to the size of the attribute superset (inputted). If i-th bit is 1, then that node has the attribute matching the i-th index of the attribute superset. The attachment of attributes to nodes is determined by the Attribute Distribution.
4. For every node, an out-degree determined by the Node Out-Degree distribution will be assigned.
5. Each attribute a given node has will be inserted into a LinkedHashMap, mapping the index of the attribute within the node to the double weight (determined by the associated distribution). Which LinkedHashMap (EFG) the node will be placed in will be determined by the Node distribution inputted.

Creating Edges

1. For every node, a number of edges equal to the given node's out-degree will be created with the concerned node as the from-vertex.
2. THE TO-NODE FOR EVERY EDGE WILL BE DETERMINED RANDOMLY?????
3. For every edge, a weight will be assigned according to the Edge-weight distribution
4. A LinkedHashMap will be created that will map the Pair <FromNode, ToNode> to the edge's weight.

Input Graph File Generation

For each EFG in the Graph Vector, a series of lines will be produced, representing the EFG. This file is described below

- Number of EFG's
- Per EFG:
    - Number of Vertices (in EFG)
    - Per Vertex:
        - Vertex ID
        - Weight of vertex
        - Number of Attributes
        - Per Attribute:
            - <Attribute ID> <Weight>
    - Number of Edges
        - <From Vertex> <To Vertex> <Weight>

The Vertex ID will be a concatenation of the Index of the LinkedHashMap the vertex is in and the vertex's index within that map. (e.g. if a vertex was in the third vertex in the second LinkedHashMap, the ID would be 12, 1 for index of map, 2 for index within map). The attribute ID will be the index within the Bitset. (e.g. if an attribute was the fourth attribute in the set it's ID would be 3)

An example file is shown below with 2 EFGs, 3 Vertices, 3 Attributes, and 1 edge

```
2            //2 EFGs
2            //2 vertices in first EFG
00           //Vertex ID of the first vertex in the first EFG
1            //Weight of 1
2            //2 attributes attached to vertex 00
1 2          //Attribute 1 with weight of 2
0 1.2        //Attribute 0 with weight of 1.2
01           //Vertex ID of the second vertex in the first EFG
5            //Weight of 5
1            //1 attribute attached to vertex 01
2 3          //Attribute 2 with weight 3
1            //1 edge
01 00 2      //Edge from vertex 01 to vertex 00 with weight 2
1            //1 Vertex in second EFG
10           //Vertex ID of the first vertex in the second EFG
4            //Weight of 4
3            //3 attributes attached to vertex 10
0 1          //Attribute 0 with weight of 1
1 1          //Attribute 1 with weight of 1
2 1          //Attribute 2 with weight of 1
0            //0 edges
```

Input Configuration File Generation

The configuration file will now be produced. This file is created directly from the input given along with dummy information necessitated by AFGMiner. Information that is always the same is italicized. The file format is displayed below:

- *DB address*
- *DB name*
- *Connection Port*
- *User Name*
- *Prefix of dataset*
- *EFG bytecode Boolean*
- Max Attributes per pattern
- *Minimum Hotness*
- Max Forward Edges per pattern
- Max Backward Edges per pattern
- Maximum gap value
- *# of threads for loading*
- # of threads for mining
- Minimum Support Level
- Max vertices per pattern

Output

These graphs will then be run through the AFGMiner algorithm. The data obtained from these program executions are as follows:

Total Subgraphs - the total number of subgraphs that are mined (including ones below the minSupport value)

- Count will be added to AFGMiner

Number of Hot Subgraphs - the total number of subgraphs that have been found that exceed the minSupport value

- Already Calculated by AFGMiner

Number of Hot n-Subgraphs - the total number of subgraphs for each n number of edges (eg. 0-edge graphs i.e. nodes)

- Already calculated by AFGMiner

Largest Support Value - the subgraph that is found to have the largest support value

- Method will be added to AFGMiner

Output Format

The Output will be of the following form:

Total Subgraphs
Largest Support Value
Number of Hot Subgraphs
Number of Hot n-Subgraphs

An example is shown below:

```
TOTAL SUBGRAPHS: 100
LARGEST SUPPORT VALUE: 0.65
NUMBER OF HOT SUBGRAPHS: 40
NUMBER OF HOT 0-EDGE SUBGRAPHS: 4
NUMBER OF HOT 1-EDGE SUBGRAPHS: 16
NUMBER OF HOT 2-EDGE SUBGRAPHS:  20
```