**Developer Manual of GAIA (in C++)**

*This is the developer manual of a C++ implementation of "GAIA: graph classification using evolutionary computation" in Proceedings of the ACM SIGMOD International Conference on management of Data, pages 879-890, 2010.*

*authors: Ning Jin, Calvin Young, Wei Wang*

*affiliation: University of North Carolina at Chapel Hill, U.S.A.*

*implemented by: Ning Jin*

*contact: njin@cs.unc.edu*

**CONTENTS:**

**1. INTRODUCTION**

**2. CLASSES**

**3. RELATIONSHIP BETWEEN CLASSES**

**4. MEMBERS AND METHODS**

**5. WORKFLOW**

**1. INTRODUCTION**

GAIA is a discriminative subgraph pattern mining algorithm. The input of GAIA is composed of two sets of graphs, one is called the positive set and the other is called the negative set. The output of GAIA is one discriminative subgraph graph for each graph in the positive graph.

Discriminative subgraphs are subgraphs that are frequent in the positive set and infrequent in the negative set. The bigger the difference between the positive frequency and the negative frequency of a subgraph is, the more discriminative the subgraph is. GAIA uses an objective function, which is the logarithm of the ratio of the positive frequency to the negative frequency, to evaluate how discriminative each subgraph pattern is. The goal is to output subgraph patterns that are highly discriminative. For more information on intuition of discriminative subgraph mining and how the objective function is defined, please refer to our research paper listed at the beginning of this manual.

The key feature of GAIA is that it uses evolutionary computation to search for discriminative subgraph patterns and it achieves superior performance in terms of runtime efficiency, resulting pattern quality and classification accuracy.

The evolutionary computation used in GAIA can be considered as a heuristic search process which mimics the biological evolution. It considers the candidate subgraph patterns as the population and it measures the fitness of a pattern by its discrimination score. Patterns with higher scores fit better. The population is initialized with patterns with only one edge. During the evolution, the patterns compete with each other for two kinds of resource: memory space to store the patterns and CPU time to compute the offspring of the patterns, where the offspring of a pattern p is the set of patterns that are supergraphs of p with one more edge. The patterns with higher discrimination scores have a better chance of being stored in the memory and getting extended to larger patterns.

For more details on the design of GAIA, please refer to our research paper listed at the beginning of this manual.

**2. CLASSES**

This implementation was written in C++. This section lists the C++ classes in the implementation.

**graph (graph.h)**

Each instance of the graph class stores the information of one input graph, including the descriptive information about this graph, all the nodes and all the edges.

The descriptive information is not used for the mining process. It is used for debugging and locating the graph representation of a specific structure in the input.

Nodes are stored in a vector and each node label is stored as a short type integer. Each node has exactly one node label (please refer to the user manual for input file format). Node labels CANNOT be 0 because it causes problems when GAIA concatenates node labels to generate the string representation of a subgraph pattern. However, the ID of a node must start at 0 and each node must have a unique ID, otherwise GAIA may have unexpected and incorrect results.

Edges are stored as adjacency lists. Each node *v* has a vector to store incident edges. Each edge is stored as a pair of short type integers. The first element is the node ID of the node other than *v*; the second element is the edge label of the edge. Similar to node labels, edge labels cannot be 0.

**pattern (pattern.h)**

Each instance of the pattern class stores the information of one subgraph pattern.

It stores the adjacency matrix, CCAM code, all the embeddings, supporting graph IDs of the subgraph pattern.

It has the methods to extend, calculate the code of, calculate the discrimination score of and output the adjacency matrix of a subgraph pattern.

**extension (pattern.h)**

Each instance of the extension class stores the information of one extension edge added to a base subgraph pattern. It does NOT include the information of the base subgraph pattern.

**pattern\_index (pattern\_index.h)**

An instance of the pattern\_index class is used to keep track of the CCAM codes of subgraph patterns that have already been explored/generated by GAIA in order to avoid repetitive consideration of the same subgraph pattern.

Currently it is implemented with STL container set. Each entry is the CCAM code of one subgraph pattern. Hash\_set is a more efficient option for larger graph database or larger search space of subgraph patterns.

**candidate\_list (candidate\_list.h)**

Each instance of the candidate\_list class implements one candidate list for one positive graph as described in our paper. Each positive graph is associated with exactly one candidate list and each candidate list can be associated with only one positive graph. Negative graphs are not associated with candidate lists.

Each candidate list uses a vector with a fixed size to store candidate subgraph patterns for the associated positive graph. Each entry in the vector stores the pointer to one instance of the pattern class, which stores the information of one subgraph pattern.

It has the methods to randomly select one candidate subgraph pattern for extension, randomly pairing a newly generated pattern with an existing pattern, compare their scores and decide which of the two patterns will survive and be inserted back to the candidate list.

**feature (feature.h)**

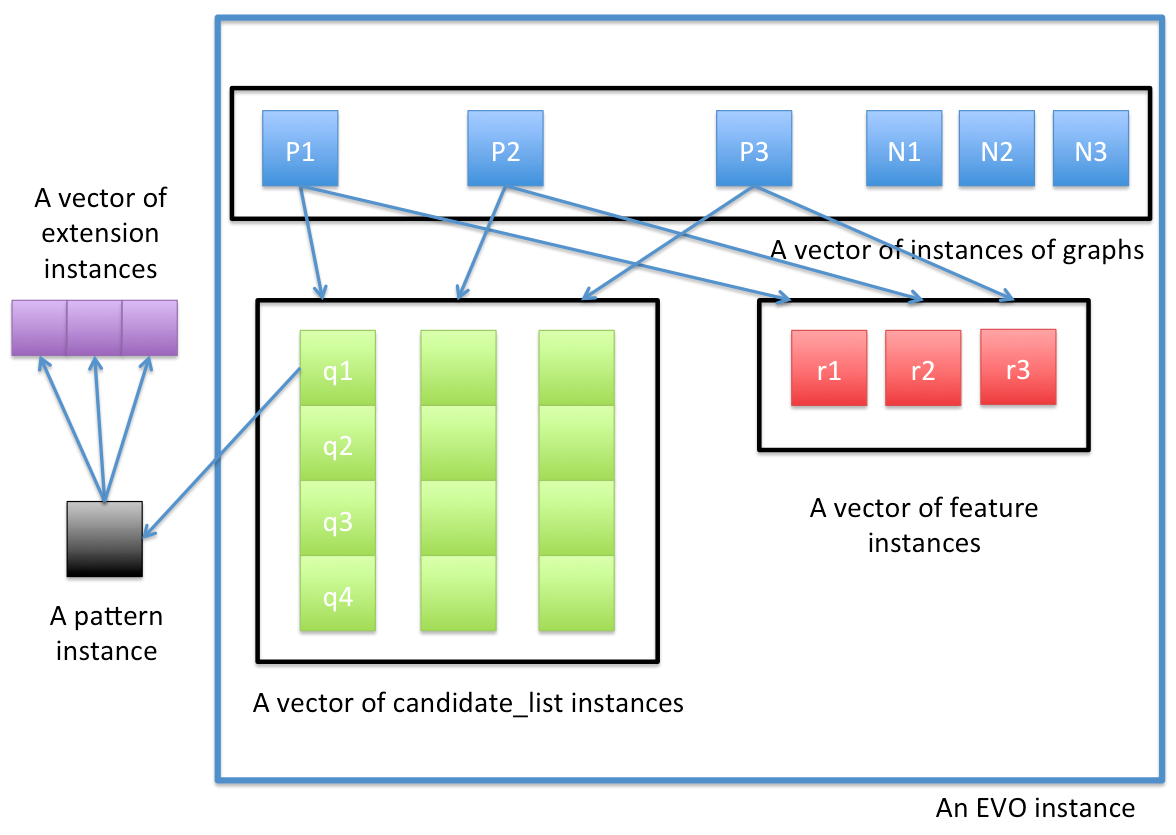
Each instance of the feature class is used to store information of the representative pattern (the most discriminative subgraph pattern found so far by GAIA for the positive graph) of one positive graph, including CCAM code, score, size and supporting graph IDs in both positive and negative sets.

This class can be extended to store more complex representative features, such as co-occurrence of subgraph patterns and approximate subgraph patterns. However, currently the implementation only supports rigid subgraph patterns as described in the paper.

**EVO (EVO.h)**

Each instance of the EVO class performs the evolutionary computation in GAIA to search for discriminative subgraphs. Each instance has its own storage space for input graphs, candidate subgraph patterns and resulting subgraph patterns. However, the instances share argument configuration, such as parameters, input file names and number of positive graphs. Such configuration is stored as external variables, defined in main.cpp, so that all instances of all classes can access them easily. This implementation is intended to run a single instance of EVO at a time. Therefore, developers need to be careful when they want to run multiple instances of EVO in multiple threads in parallel. The best way to enable execution of multiple instances of EVO in multiple threads at the same time is to store the argument configuration within each instance. Code changes need to be made accordingly.

**3. RELATIONSHIP BETWEEN CLASSES**



The figure above illustrates the relationship between classes.

An instance of the EVO class contains a vector of instances of the graph class, a vector of instances of the candidate\_list class and a vector of instances of the feature class. Each positive graph is associated with its own candidate\_list instance and its own feature instance. Negative graphs are not associated with candidate\_list instances or feature instances.

Each instance of the candidate\_list class contains a vector of pointers pointing to instances of the pattern class. Each entry points to exactly one instance of the pattern class. Each pattern instance is associated with a vector of instances of the extension class during extension operations.

**3. MEMBERS AND METHODS**

This section lists the non-trivial members and methods.

**members of the graph class**

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| info | string | descriptive information about a graph |
| nodes | vector<short> | list of nodes and their labels in a graph; nodes[i] corresponds to the node whose ID is i |
| adjList | vector<vector<pair<short, short> >\*> | adjacency list of each node; for each pair: first=node ID, second=edge label; adjList[i] corresponds to the node whose ID is i |

**member of the pattern class**

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| matrix | short\*\* | adjacency matrix representation of the pattern |
| size | int | number of nodes in the pattern |
| edge\_size | int | number of edges in the pattern |
| score\_precise | float | discrimination score of the pattern |
| score\_binned | int | discretized discrimination score of the pattern |
| code | vector<short> | CCAM code representation of the pattern |
| embeddings | vector<pair<short, vector<occ\*>\* > > | occurrences of the pattern in both positive set and negative set |
| pgids | vector<short> | IDs of the positive supporting graphs; the IDs in this vector are assumed to be sorted in ascending order |
| ngids | vector<short> | IDs of the negative supporting graphs; the IDs in this vector are assumed to be sorted in ascending order |

**members of the extension class**

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| is\_internal\_ext | bool | whether the extension is an internal extension; an internal extension is an extension that does not involve new node added to the base subgraph pattern |
| pgids | vector<short> | IDs of the positive supporting graphs; the IDs in this vector are assumed to be sorted in ascending order |
| ngids | vector<short> | IDs of the negative supporting graphs; the IDs in this vector are assumed to be sorted in ascending order |
| ext\_occs | vector<ext\_occ\*> | occurrences of the extension in both positive set and negative set |

**members of the candidate\_list class**

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| data | vector<pattern\*> | a vector of pointers to subgraph patterns being stored in the candidate list |
| length | int | the number of subgraph patterns currently stored in the candidate list; this is different from candidate list size, a parameter that specifies the maximum number of subgraph patterns that can be stored in a candidate list |

**members of the EVO class**

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| graphs | vector<graph\*> | input graphs to GAIA |
| pat\_idx | pattern\_index | patterns that have already been explored |
| graph\_features | vector<feature\*> | the representative subgraph pattern for each positive graph; the size of this vector equals the number of positive graphs; graph\_features[i] stores the representative subgraph pattern for graph whose ID is i |
| candidate\_lists | vector<candidate\_list\*> | the candidate list for each positive graph; the size of this vector equals the number of positive graphs; candidate\_lists[i] stores the candidate list for graph whose ID is i |
| feature\_updated | vector<bool> | feature\_updated[i] indicates whether the representative subgraph pattern has been updated for positive graph whose ID is i |
| edges | map<string, pattern\*> | 1-edge subgraph patterns that have been explored; it is needed because CCAM codes for subgraph patterns with only 1-edge are not calculated; for more details, please refer to the paper |

**methods of the pattern class**

vector<short> gen\_code()

*description: It generates and returns the CCAM code for the pattern.*

bool check\_embeddings(vector<graph\*>& graphs)

*description: It checks whether each embedding stored in the instance is truly an occurrence of this pattern. Input vector graphs are all input graphs. This method is used only for debugging.*

void get\_score()

*description: It updates the score of the current pattern.*

pattern\* gen\_new\_pattern(extension& ext, int code, pattern\_index& pat\_idx, vector<graph\*>& graphs)

*description: It generates a new subgraph pattern based on the current pattern and the extension ext. Inut pattern index is needed here to avoid generating the same subgraph pattern repeatedly. Input vector graphs are all input graphs.*

void collect\_ext(short gid, short par\_em\_gid, occ& occ1, short occ1\_id, vector<graph\*>& graphs)

*description: It collects all the possible extensions of this pattern in graph whose ID is gid. Input par\_em\_gid is the graph ID of the parent pattern and occ1\_id is the embedding ID of the parent pattern. Input vector graphs are all input graphs.*

vector<pattern\*>\* extend(vector<graph\*>& graphs, pattern\_index& pat\_idx)

*description: It generates and returns all possible patterns with one edge added to the current pattern. It first collects all the possible extensions by calling collect\_ext and then generate a new pattern for each extension by calling gen\_new\_pattern.*

**methods of the candidate\_list class**

int select\_duel()

*description: It randomly selects a pattern in the candidate list and return the index of this pattern. The selected pattern will be compared with a newly generated pattern in terms of discrimination scores. The pattern with the lower score will be discarded.*

bool insert(pattern\* p)

*description: It inserts a candidate subgraph pattern into the candidate list. If the candidate list is not full, it inserts the pattern p into an available entry; if the candidate list is already full, then it calls select\_duel to select an existing pattern q and compares the discrimination scores of p and q. If the score of p is higher, then p will replace q; if the score of q is higher, then p will be discarded. This method returns true if the pattern is successfully inserted; it returns false otherwise.*

pattern\* select\_extension()

*description: It randomly selects a subgraph pattern from the current candidate list for extension. The probability of a pattern being selected is proportional to its discretized discrimination score. It returns the selected pattern.*

void insert(vector<pattern\*>\* patterns)

*description: It inserts a vector of patterns into the candidate list by calling insert(p) to insert them individually.*

**methods of the EVO class**

bool read\_graphs()

*description: It reads the input graphs from files, assuming the file formats described in the user manual.*

void init\_edges()

*description: It initializes the pattern index for subgraph patterns with only one edge. The pattern index for patterns with only one edge is different from the pattern index for larger patterns because calculating CCAM code for a subgraph pattern with only one edge is inefficient when there are many occurrences of this pattern in a graph. Please refer to our research paper for more details. This method also initializes the candidate list of each positive graph with 1-edge patterns.*

void insert\_edge(int gid, short n1, short n2, short l1, short l2, short le)

*description: It inserts an occurrence of a 1-edge pattern into the pattern index for patterns with only one edge. Input gid is the ID of the graph where this edge occurs. Input n1 and n2 are the IDs of the two nodes of the edge; l1 and l2 are the labels of the two nodes. It assumes that n1 is less than n2. Input le is the edge label.*

void pick\_one\_grow(int gid)

*description: It randomly picks to extend one subgraph pattern from the candidate list of the graph whose ID is gid. It generates all possible one-edge extensions.*

bool distribute\_pattern(pattern\* p)

*description: It attempts to distribute a newly generated pattern p to a candidate list of a supporting graph of p. It tries to find a candidate list that is not full yet, if there are multiple qualified candidate lists, it inserts p into the list with the lowest sum of pattern scores (lower sum of pattern scores means the competition in that list is less fierce). If there are no qualified candidate list, then it tries to insert p into the list with the lowest sum of pattern scores. Please refer to the paper for more details.*

void evolution()

*description: It runs the evolutionary algorithm to search for discriminative subgraph patterns. It begins with the initialization of pattern index for 1-edge patterns and candidate lists. Then it calls pick\_one\_grow and distribute\_pattern iteratively either until there are no more candidate patterns for extension or until the number of iterations reaches a user-specified threshold.*

void feature\_selection(vector<feature\*>& features)

*description: It selects representative patterns to generate graph classification rules (please refer to the paper for how we generate graph classification rules with subgraph patterns). Although each representative subgraph pattern is the best pattern found by GAIA for a particular positive graph, the whole set of representative subgraph patterns do not necessarily lead to the best graph classifier. Therefore, there is one more step after the evolutionary computation to optimize the graph classifier. This method uses sequential coverage to select subgraph patterns.*

**5. WORKFLOW**

The figure below illustrates the overall workflow of GAIA with method calls. None of the methods is a static method of the class. I use class\_name.method\_name only to indicate in which class the method is defined.

