April 2017

Lot 5 of Data Analytics FY16/17

Final Technical Report: TR UoK/Dstl 2017/1

Dover: Scalable Algorithms for Graph Mining in Java

Rob Baker, Alena Frankel, Peter Rodgers

Abstract

The Dover project addresses two related variants of graph mining. One is to look for subgraphs (or behaviours of interest) present in a target graph, which corresponds to the subgraph isomorphism problem. The other is to find motifs (over-represented subgraphs) in large graphs, which involves a number of algorithms, including graph isomorphism. The time complexity of algorithms to perform these isomorphisms means that exact solutions are infeasible in the general case. Hence, we provide algorithms to solve exact and approximate solutions to both of these variants.

Subgraph isomorphism (finding pattern graphs in larger target graphs) finds significant practical application in pattern recognition where searching for matching subgraphs is used for Machine Learning. Various applications of motifs have been proposed, however the widest use is found in Bioinformatics where frequent graph searching algorithms look for occurrences of subgraphs that are greater than might be expected from chance.

A principle focus of this work has been on scalability. The work targets graphs in excess of one million nodes and ten million edges. The implementation emphasises efficiency and we provide sample performance indicators for the implemented algorithms on both synthetic and real-world graphs. Dover is implemented in pure Java for ease of integration with other systems.

1. Introduction

This technical report is the final deliverable from the Dstl tender for Lot 5 of Data Analytics FY16/17, which ran October 2016 to April 2017. The bid document was titled “Scalable Subgraph Isomorphism” and the project completed by teams at the University of Kent and Roke Manor Research Limited. The result of the work is open source software and data, which we call “Dover”. The project tackled two related but separate tasks in Graph Mining: firstly, finding motifs in graphs and secondly finding structures of interest in graphs. The principle challenge was in the size of data, with a specified minimum 1 million nodes and 10 million edges. This is significantly larger than addressed by previous systems [Coc10]. Our solution to this scaling problem is efficient implementation of approximate solutions, so that by restricting search we can get some, but not all structures of interest and by sampling the graph we can get a representative indication of motifs. As we discuss below, we show capability significantly beyond the required size. Another aspect to the project was that of dealing with behaviour graphs, that is graphs that show change over time. Our methods were firstly implemented over standard graphs, then behaviour graph functionality was added.

Subgraph isomorphism finds significant practical application in pattern recognition [FPV14] where searching for matching subgraphs is used for Machine Learning and image recognition. Motif finding is most often found in analysis of bioinformatics data [Bai09] where frequent graph mining is the process of discovering motifs, or occurrences of subgraphs that are overrepresented in the graph. Motifs have been applied to crime pattern analysis [DM15], YouTube spamming [CHC12] and social networks [Fan12].

This document includes sample profiling data, all such data was run on Java 8, jre 1.8.0_91 on Windows 10. The hardware was a Dell XPS i7 6700HQ with 32GB RAM and a 1TB SSD.

Throughout this document we outline code snippets and examples to ease use, further integration and development. Section 2 describes the software system and installation details. Section 3 gives a user guide to both the graphical interface and command line functions. Section 4 details the underlying data structures used. Section 5 describes the fundamental algorithms implemented. The description of how these are used in various graph mining methods is given in Section 5.

2. Software Access and Installation

The software is implemented in Java 8, jre 1.8.0_91, compiled using Eclipse Neon.1a (4.6.1) on Windows 10. Whilst we expect the code to be portable, no rigorous testing beyond these specifications has been performed. All code written for this project is released under GPL 3. All external libraries and code is under GPL or less restrictive open source licence.

Git repository: `https://github.com/peterrodgers/dover.git`

Get a copy of the code via:

```
git clone https://github.com/peterrodgers/dover.git
```

The DisplayGraph source code is packaged with the dover code. The following libraries are included in the repository and their jars should be added in Eclipse via `Project Properties -> Java Build Path -> Libraries:`

- commons-cli
- Jama
- jsoup

Sample data can be downloaded from https://www.cs.kent.ac.uk/projects/dover/

Create a jar file via: `buildjar.bat` found in the top level directory.

Create the javadocs via: `document.bat` found in the top level directory.

3. User Guide

To install Dover, download dover.jar. There are two interfaces to the system, command line and GUI.

3.1. Command Line

There is a command line interface to Dover. It does not provide as much visual feedback as the GUI, but still allows most of the functionality to be performed.

It is strongly recommended that you allow the JVM to have access more RAM when running Dover. For example, if you are running on a 16GB RAM we recommend a command of the type: `java -Xmx14g -jar dover.jar`. This amount will vary depending on the amount of RAM available to you. As the `-Xmx` value is system dependent, it is not shown in the below examples.

It is also possible to view a manual page: `java -jar dover.jar -h`. Specifying no Dover parameters will instead load the GUI, e.g. `java -Xmx14g -jar dover.jar`. This is a useful way to load the GUI with specific JVM options.

3.1.1. Exact Subgraph Isomorphism

To run the exact subgraph isomorphism algorithm, use the following parameters:

```
-s <arg>   Runs the subgraph isomorphism algorithm. Specifies the location of the
           target graph.
-p <arg>   Specifies the location of the pattern graph.
```

For example: `java -jar dover.jar -s targetgraph -p patterngraph`

3.1.2. Approximate Subgraph Isomorphism

To run the approximate subgraph isomorphism algorithm, use the following parameters:

```
-S <arg>        Runs the approximate subgraph isomorphism algorithm. Specifies the
                location of the target graph.

-p <arg>        Specifies the location of the pattern graph.

-n <arg>        Specifies the number of nodes in the enumerated subgraphs. A larger
                number will allow more matches, but increases the runtime.

--subspernode <arg>

                Specifies the number of subgraphs enumerated per node. This will
                define the number of subgraphs tested based on the number of nodes in
                the target graph.
```

For example: `java -jar dover.jar -S targetgraph -p patterngraph -n 5 --subspernode 5`

3.1.3. Exact Motif Finding

To run the exact motif finder, use the following parameters:

```
-m <arg>              Runs the exact motif finding algorithm. Specifies the location
                      of the target graph.

--minsize <arg>       Specifies the minimum size of motifs to be found.

--maxsize <arg>       Specifies the maximum size of motifs to be found.

--saveall             Optional. Specifies is every example of a motif is to be
                      found. Warning: This can take considerable time and disk
                      space.
```

For example: `java -jar dover.jar -m targetgraph --minsize 4 --maxsize 5` or `java -jar dover.jar -m targetgraph --minsize 4 --maxsize 5 --saveall`

3.1.4. Approximate Motif Finding

To run the approximate motif finder, use the following parameters:

```
-M <arg>              Runs the approximate motif finding algorithm. Specifies the
                      location of the target graph.

--minsize <arg>       Specifies the minimum size of motifs to be found.

--maxsize <arg>       Specifies the maximum size of motifs to be found.

--subspernode <arg>   Specifies the number of subgraphs enumerated per node. This
                      will define the number of subgraphs tested based on the number
                      of nodes in the target graph.

--attempts <arg>      Specifies the number of attempts to enumerate a subgraph. 20
                      is normally sensible for subgraphs of size < 8.

--clusters <arg>      Specifies the number of clusters to group motifs into.

--iterations <arg>    Specifies the number of iterations to run kMedoids for.
                      Increasing this number will increase the runtime.
```

For example: `java -jar dover.jar -M targetgraph --minsize 4 --maxsize 5 --subspernode 5 --attempts 20 --clusters 5 --iterations 2`

3.1.5. Adjacency List to Dover Converter

To run the converter, use the following parameters:

```
-c <arg>      Runs the converter. Specifies the location of the adjacency list.

-n <arg>      Specifies the number of nodes in the graph.

-e <arg>      Specified the number of edges in the graph

-d            Optional. Indicates if the graph is directed.
```

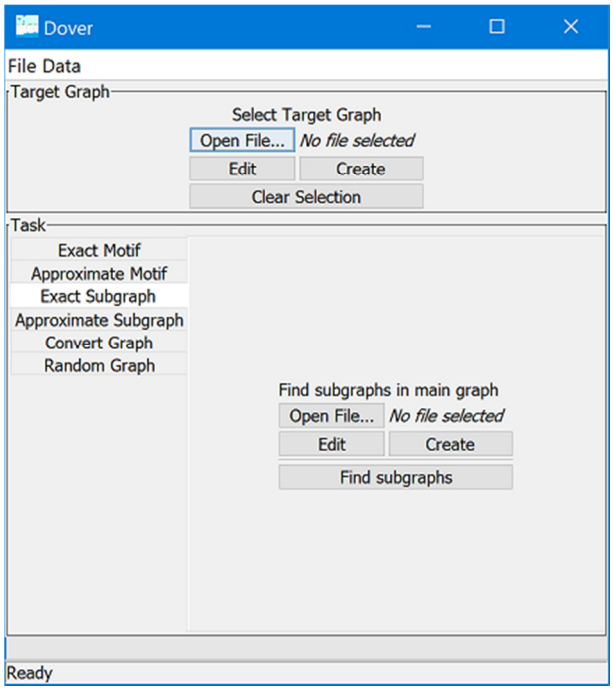
For example: `java -jar dover.jar -c adjlist.adj -n 30 -e 40`

3.2. Graphical User Interface

3.2.1. Selecting a Target Graph

Select a target graph by clicking **Open File** in the target graph panel. This will allow a directory to be selected contain the various buffers and `.info` file. This graph can be cleared by clicking **Clear Selection**.

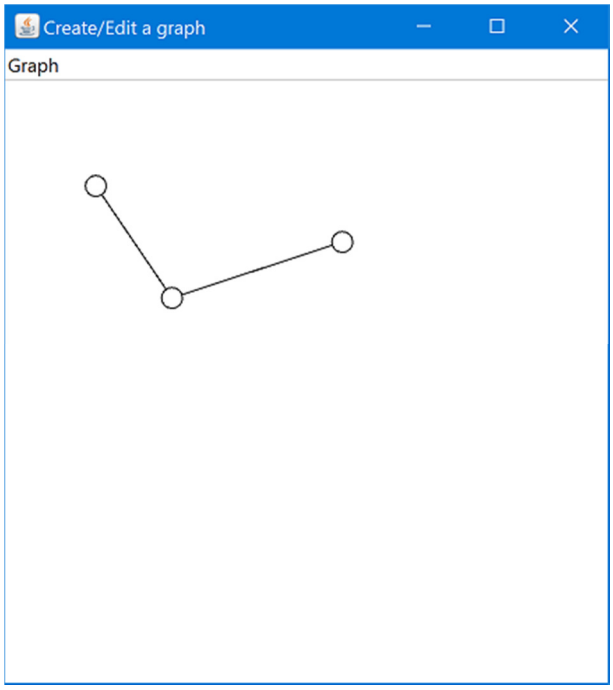
It is possible to **Create** and **Edit** a target graph. Editing a graph is inadvisable if the graph contains more than around 30 nodes. Creating and editing a graph uses the Graph Editor.

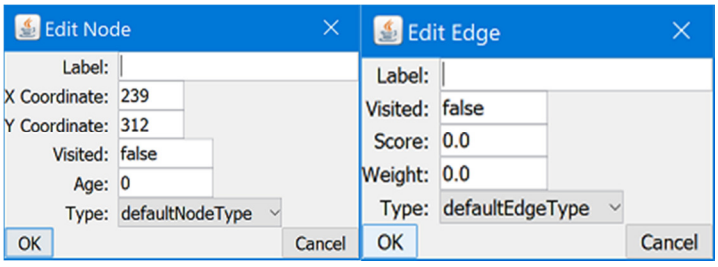


3.2.2. Graph Editor

The graph editor allows the modification or creation of graphs. Double click to create a node, and right-click and drag to create an edge between nodes. Double left clicking on a node or edge will display an editor box - this can be used to specify a label of a node or edge, along with an age. It is also possible to change the type of an edge to `timeEdge` if you wish to perform analysis on behaviour graphs.

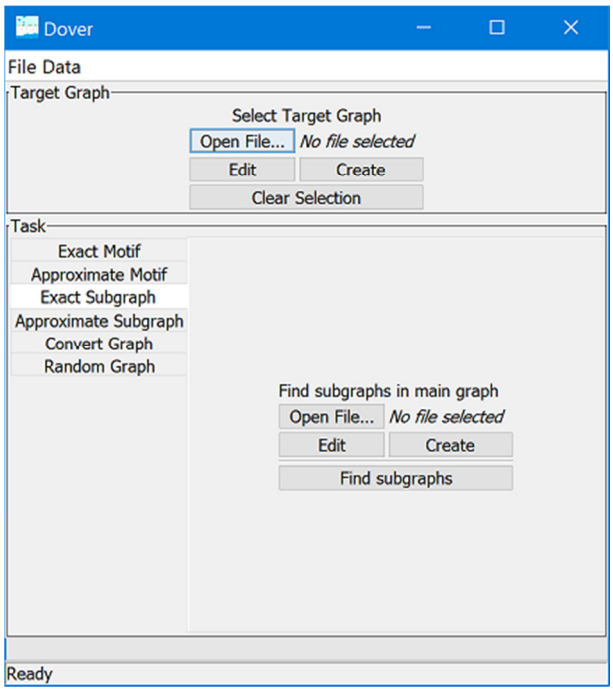
Under the **File** menu, there is an option to save this graph, as well as two variations of a spring embedder to automatically draw the graph (one is animated, the other is not). This layout method aims to display the various time slices (if relevant) in a sensible manner.





3.2.3. Exact Subgraph Isomorphism

To run the exact subgraph isomorphism algorithm, navigate to the **Exact Subgraph** tab. Here, use the **Open Fil**e button to select a pattern graph. As with the target graph, it is possible to **Create** or **Edit** a pattern graph. Once complete, click **Find subgraphs** and the algorithm will run. When complete, a message will be displayed to the user.



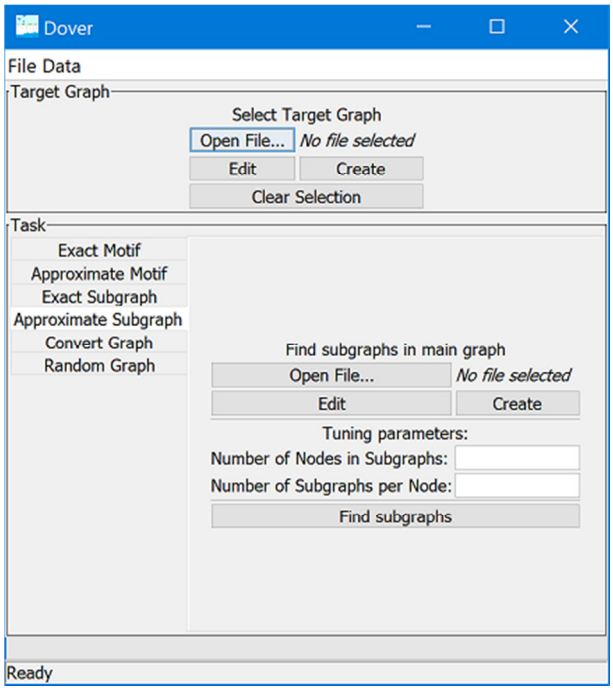
3.2.4. Approximate Subgraph Isomorphism

To run the approximate subgraph isomorphism algorithm, navigate to the **Approx Subgraph** tab. Here, use the **Open File** button to select a pattern graph. As with the target graph, it is possible to **Create** or **Edit** a pattern graph.

There are a number of tuning parameters for this algorithm.

- The **Number of Nodes in Subgraphs** parameter specifies the number of nodes in the enumerated subgraphs. A larger number will potentially allow more matches, but increases the runtime. This must be a positive integer.
- The **Number of Subgraphs per Nodes** parameter specifies the number of subgraphs enumerated per node. This will define the number of subgraphs tested based on the number of nodes in the target graph.

Once complete, click **Find subgraphs** and the algorithm will run. When complete, a message will be displayed to the user.



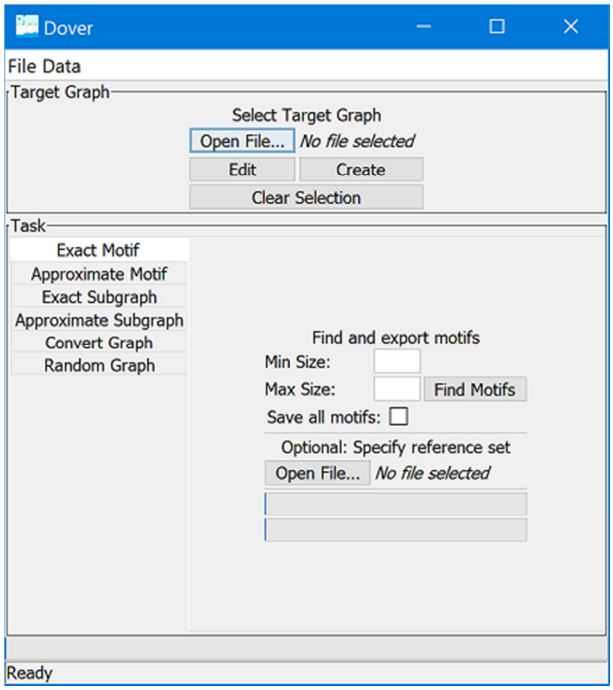
3.2.5. Exact Motif Finder

To run the exact motif finder algorithm, navigate to the **Exact Motif** tab.

There are a number of tuning parameters for this algorithm:

- The **Min Size** and **Max Size** parameters specify the sizes of the motifs to find - these numbers are inclusive.
- The **Save all motifs** options allows the user to save every example of the found motifs. Beware that this adds considerable time to the running of the algorithm (especially when not using an SSD) and can occupy large amounts of disk space.
- It is also possible to specify a particular reference set to use in comparison the the target graph. This can be set using the **Open File** button, and is optional. If the user does not specify a reference set, the target graph will be rewired several times to use instead.

Finally, when all parameters have been set, the **Find Motifs** button will run the algorithm. This process can take a considerable amount of time, so two progress bars provide updates to the user. Once complete, a message will be displayed and (if supported) a web browser will be opened displaying the results.



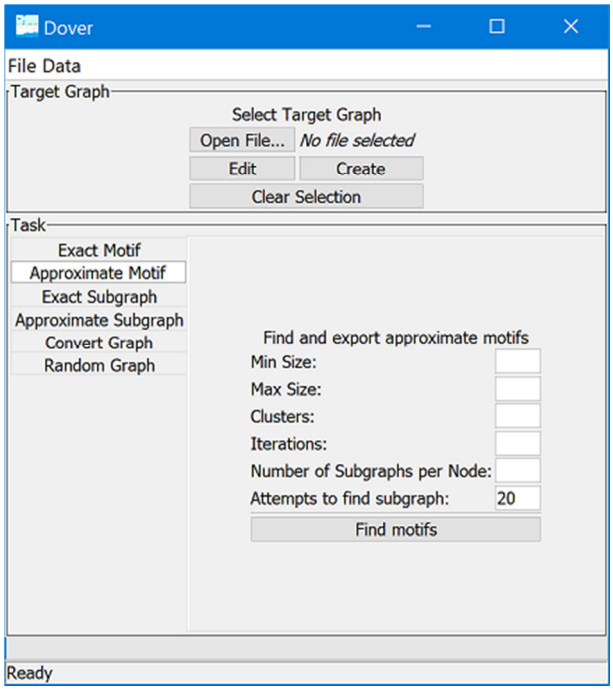
3.2.6. Approximate Motif Finder

To run the approximate motif finder algorithm, navigate to the **Approx Motif** tab.

There are a number of tuning parameters for this algorithm:

- The **Min Size** and **Max Size** parameters specify the sizes of the motifs to find - these numbers are inclusive.
- The **Clusters** parameter specifies how many groups the found motifs will be assigned to. The more groups, the increased likelihood that wildly differing graphs will be stored separately. However, a number of these groups may be empty.
- The **Iterations** parameter specifies the number of times the reassignment of motifs will be completed. It is advisable to use a number greater than 1, but each increasing the number of iterations will increase the runtime of the algorithm.
- The **Number of Subgraphs per Nodes** parameter specifies the number of subgraphs enumerated per node. This will define the number of subgraphs tested based on the number of nodes in the target graph.
- The **Attempts to find subgraphs** parameter has a default value of 20. This determines how many attempts will be run to enumerate a particular subgraph. If the number of subgraphs enumerated is considerably less than `Number of Subgraphs per Nodes * Number of Nodes`, then increasing this number may be of value.

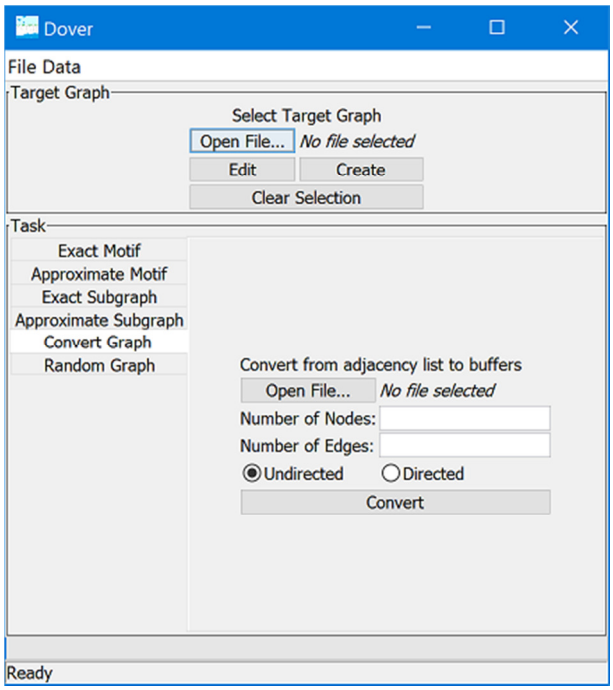
Finally, when all parameters have been set, the **Find Motifs** button will run the algorithm. Once complete, a message will be displayed and (if supported) a web browser will be opened displaying the results.



3.2.7. Adjacency List to Dover Converter

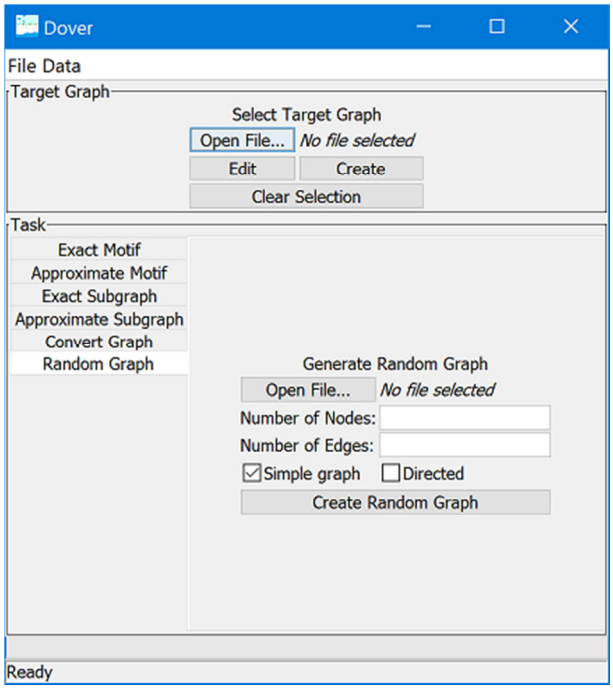
It is also possible to convert a graph from an adjacency list to the Dover format. Here, use the **Open File** button to select an adjacency list. Parameters are:

- **Number of Nodes** and **Number of Edges** specify the number of nodes and edge in the adjacency list.
- Selecting **Undirected** or **Directed** specifies if the graph has directed or undirected edges. Finally, when all parameters have been set, the **Convert** button will build the graph in Dover format.



3.2.8. Random Graph

The GUI allows the creation of a random graph. This can be useful when testing the system. Use the **Open File** button to select a location to save the graph. The **Number of Nodes** and **Number of Edges** parameters specify the number of nodes and edge in the new graph. Checking the **Simple Graph** parameter will not create any parallel edges in the graph, while checking **Directed** will create directed edges. Finally, when all parameters have been set, the **Create Random Graph** button will build the random graph.



4. Underlying Data Structures

Standard object-oriented graph storage models cannot manage the size of data required in this project. Informal testing of existing data structures (using the DisplayGraph package integrated into this project) showed performance limited to less than 1,000,000 edges.

Hence, a more efficient storage mechanism is required. Java provides the ByteBuffer data structure. Here, the number of bytes stored is specified by the programmer, and the can be bytes directly accessed as various basic data types, such as bytes, chars, integers or longs. Access to data is fast and the ByteBuffers can be stored and loaded from hard disk at high speed. To store items such as nodes and edges, the items must be serialized and items stored one after the other in the buffer. The disadvantages to ByteBuffers are two-fold: firstly, time for implementation is greater as the representation of items in the ByteBuffers needs to be carefully defined and access must be coded at a low level; secondly removing and changing is difficult when using this serial representation.

We use 5 ByteBuffers to store information (Figure 1, below). These are for: nodes, edges, node labels, edge labels, and connections. For Nodes and Edges, we interleave item attributes through the ByteBuffer. Hence, each node takes 28 bytes, and each edge takes 20 bytes. ByteBuffers are limited in size to `MAX_INT` bytes, which is 2,147,483,647, This indicates a top limit of approximately 76 million nodes and 100 million edges. We have not explored this limit due to RAM limitations.

Whilst node and edges are fixed length, labels and connectivity information are variable length.

Node attributes are:

- The start of label information in the node label buffer and the number of chars present
- Four attributes for connection information for nodes. This is also stored in edges, see below. The redundancy here speeds up algorithms, as it means it is constant time complexity to find a neighbour of a node. The connection buffer stores pairs: each pair is the connecting edge index and the node at the other end of the edge: both are stored for speed reasons. The connections into this node are stored first where this node is `node2` in an edge, followed by the connections out (where this node is `node1` in an edge). To allow access to both of these lists separately, the node has a two connection start attributes, and two degree attributes: in degree and out degree. Self-sourcing edges appear in both lists.
- Weight, a user definable attribute
- Type, a user definable attribute
- Age, which has a particular use in behaviour graphs

Edge Attributes are

- Two nodes: `node1` and `node2`, specifying the nodes at either end of the edge by their indexes. We note that having `node1` and `node2` specified means the graph can be seen as directed.
- The start of label information in the node label buffer and the number of chars present
- Weight, a user definable attribute
- Type, a user definable attribute
- Age, which has a particular use in behaviour graphs

Creating the FastGraphs is via factory methods that create new graphs. The constructors should not be used by application programmers. Use of the data structures is via accessor methods, so no direct access to ByteBuffers by application programmers is required. Nodes and edges are identified by integers, and getters are provided all attributes including labels. Some editing of attributes by setters is provided, but this is limited to fixed length attributes. To edit labels, nodes and edges, a new FastGraph must be created from the existing one, which can be an expensive operation, hence if multiple edits are required, we advise performing as many as possible at the same time. See the javadocs for the various methods provided.

Here is a code snippet to create a graph consisting of a triangle with 3 nodes and 3 edges, and perform some simple operations:

```
// define the nodes that will appear in the new graph
LinkedList<NodeStructure> addNodes = new LinkedList<NodeStructure>();
NodeStructure ns0 = new NodeStructure(0,"ns0", 1, (byte)1, (byte)0);
NodeStructure ns1 = new NodeStructure(1,"ns1", 1, (byte)1, (byte)0);
NodeStructure ns2 = new NodeStructure(2,"ns2", 1, (byte)0, (byte)0);
addNodes.add(ns0);
addNodes.add(ns1);
addNodes.add(ns2);

// define the edges that will appear in the new graph
// the nodes must be defined first, as the last two arguments give the node indexes
LinkedList<EdgeStructure> addEdges = new LinkedList<EdgeStructure>();
EdgeStructure es0 = new EdgeStructure(0,"es0", 1, (byte)0, (byte)0, 0, 1);
EdgeStructure es1 = new EdgeStructure(1,"es1", 2, (byte)0, (byte)0, 0, 2);
EdgeStructure es2 = new EdgeStructure(2,"es2", 2, (byte)0, (byte)0, 1, 2);
addEdges.add(es0);
addEdges.add(es1);
addEdges.add(es2);

// use a factory to create the new graph
FastGraph g = FastGraph.structureFactory("new graph",(byte)0,addNodes,addEdges,false);

// iterate through all the nodes, and output the labels of those with type 1
for(int n = 0; n < g.getNumberOfNodes(); n++) {
  if(g.getNodeType(n) == 1) {
    System.out.println(g.getNodeLabel(n));
  }
}

// iterate through all the edges, and output the labels of those with weight 2
for(int e = 0; e < g.getNumberOfEdges(); e++) {
  if(g.getEdgeWeight(e) == 2) {
    System.out.println(g.getEdgeLabel(e));
  }
}
```

The DisplayGraph package is included in the distribution. It allows visualization of small graphs. The limit for comprehensible visualization is around 30 nodes. This command creates a DisplayGraph:

```
// visualize the graph. This creates a new DisplayGraph and window to show it in
g.displayFastGraph();
```

Graphs can be saved and loaded. The default directory is `data/` under the current working directory, although other locations can be specified:

```
// this will create a new directory if it does not already exist
// It contains the ByteBuffers and information about the graph
g.saveBuffers(null, "test");

// This creates a new graph from the saved data
FastGraph loadGraph = FastGraph.loadBuffersGraphFactory(null, "test");
```

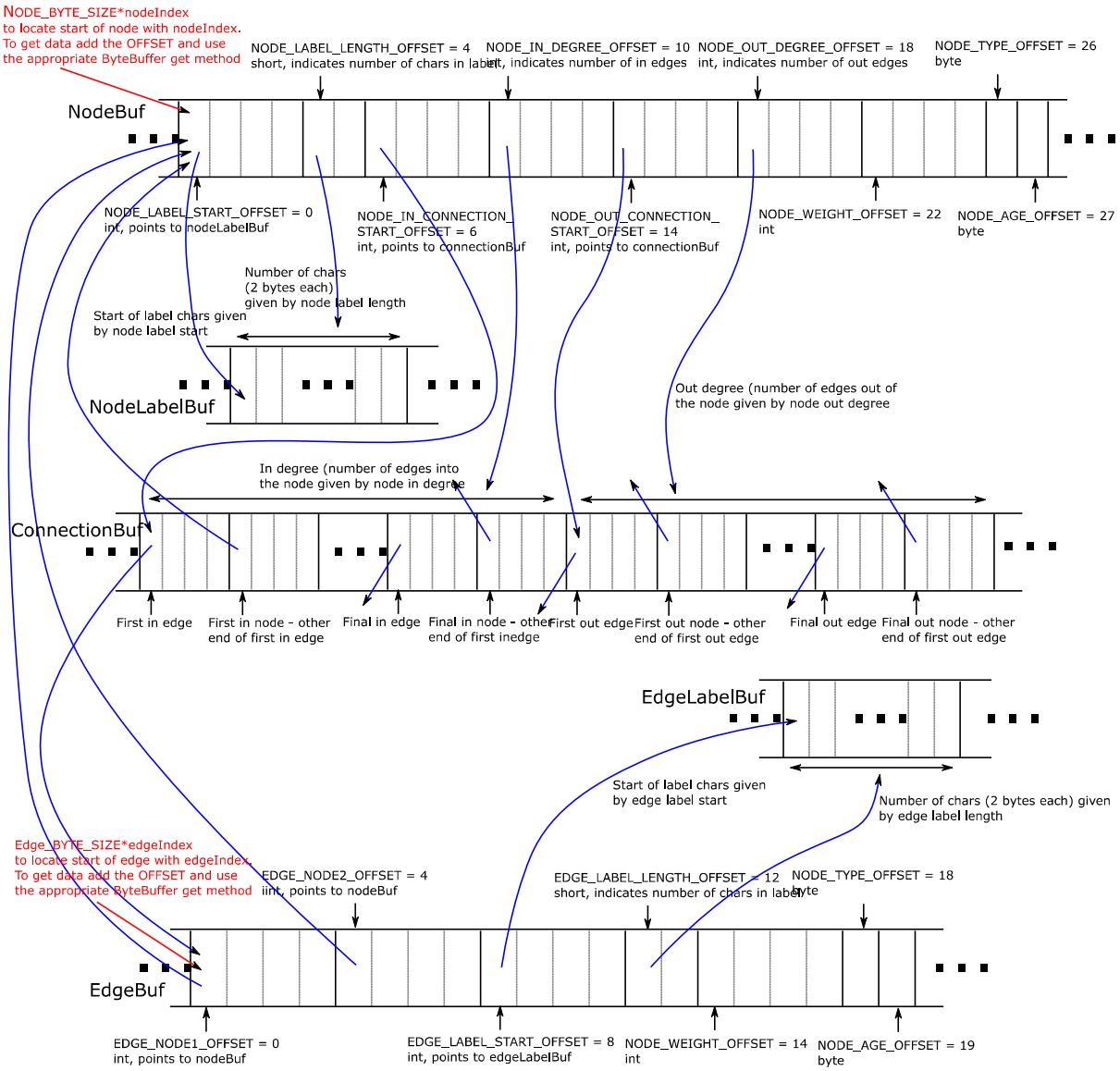


Figure 1: ByteBuffer Structure

Behaviour graphs, which model changes in data over time are implemented by the use of the “age” attribute of nodes and edges and the introduction of time edges. Age 0 is assumed to be the earliest generation, and age 0 is always the first generation in a behaviour graph. To model nodes that persist into the next generation, they are connected to the corresponding node in the previous generation by a “time edge”. These are edges with a reserved type `TIME`, currently assigned 127. These edges are treated differently to normal edges in the algorithms, as explained later. Each node and edge in the time slice is given an age one greater than the previous – all nodes and edges in a time slice have the same age attribute. Note that this is a misuse of the term age as the oldest items have age 0, and those in later time slices have greater ages.

5. Fundamental Algorithms

These algorithms are utilized in the Graph Mining methods given in Section 6. All algorithms assume an undirected graph.

5.1. Graph Isomorphism

Graph Isomorphism is the topological equality of graphs. We implement a version that requires the input graphs to be connected. The method first tests cheap inequality measures in the graphs. If these pass then a brute force comparison is executed to see if there is a mapping between all nodes.

The cheap tests initially performed (in order) are:

1. Node count – the number of nodes must be equal
2. Edge count – the number of edges must be equal
3. Node degree profile – the number of nodes with particular degree must be equal
4. Eigenvalue – the eigenvalues of the graphs must be equal

In terms of performance, for randomly generated graphs with equal nodes and edges, the node degree profile filters most non-isomorphic graphs. The below table shows the number of non-exponential tests carried out (nodes and edges were equal by generation, so are not listed). Here "Isomorphic" (when the two graphs are equal) and "Fail by Brute Force" added up form the number of times the exponential brute force algorithm is applied. The low number for "Fail by Brute Force" is typical, as well over 99% of non-isomorphic graphs are caught by the cheap tests in nearly all normal circumstances.

Nodes	Edges	Isomorphic	Fail by Degree Profile	Fail by Eigenvalue	Fail by Brute Force	Average time per test
4	3	61.7%	38.3%	0%	0%	9.4E-6s
5	6	26.3%	71%	2.6%	0%	1.85E-5s
6	9	7.9%	81.4%	10.7%	0%	2.32E-5s
7	12	1.2%	89.2%	9.6%	0%	2.69E-5s
8	16	0.07%	94.5%	5.4%	0%	3.17E-5s

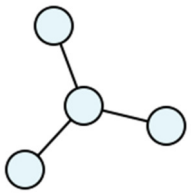
The brute force method initially calculates the possible matches in the second graph for each node in the first graph. This is the set of nodes in the second graph with the same degree as the node in the first graph. An exponential time (on the number of nodes in the graph) backtracking search is then performed, taking a node `n1` from the first graph and attempting to find a match from its matching set of nodes. For each node in the second graph, `n2`, the neighbours of `n1` and `n2` must correspond. That is, if a node is a neighbour of `n1` and it has an assigned match in the other graph, the matching node must be a neighbour of `n2`. Similarly for neighbours of `n2`. If a match is found, the next node in the first graph is checked for possible matches. If no matches can be found for `n1`, backtracking occurs, and the last previously matched node in the first graph is checked for other matches. The process ends successfully when all nodes are assigned a match. A failure to find a solution occurs when all possible node matches have been tried without success.

An example of use of the isomorphism algorithm is:

```
FastGraph g1 = FastGraph.randomGraphFactory(10,20,1,true,false);
FastGraph g2 = FastGraph.randomGraphFactory(10,20,2,true,false);
ExactIsomorphism ei = new ExactIsomorphism(g1);
boolean result = ei.isomorphic(g2);
```

Note that the `ExactIsomorphism` class is created with one of the graphs to be tested as a parameter to the constructor. The class stores profiling information for this graph. Hence, if one graph (`g1`) is to be tested against a number of other graphs, then there is a performance

increase if an instance of the class is created once with `g1` and the `isomorphic(gX)` method called on the graphs to be tested against. The matching nodes can be found with `getLastMatch()` which should only be called after `isomorphic(gX)` has been called. This returns an integer array where the index are the nodes in the first graph and the value at a index is the matching node in the second graph.



Subgraphs can be classified using a serialized form. The serialized forms are one way of performing all of the cheap isomorphism tests given above as two serialized forms can be directly compared. If they are different then the two graphs are not isomorphic. However, note that if the serialized forms are the same, the two graphs may not necessarily be isomorphic and a brute force check is still needed.

The serialized form is structured in the following way:

```
Nodes, Edges, Node Degree Profile, Eigenvalue, Relative Time Profile
```

For example, the graph above would be represented as:

```
4,3[0, 3, 0, 1][-1.732051, 0.0, 0.0, 1.732051][1,3]
```

In this case, the graph has 4 nodes, 3 edges and 0 nodes with 0 degree, 3 nodes with 1 degree, and so on. Then follows the eigenvalues for this graph, before the relative time profile. In this case, there is 1 node at the lowest age value, with 3 nodes at the next one. This allows graphs which span, say, ages 0 and 1, to have the same key as one that spans 2 and 3.

5.2. Subgraph Isomorphism

Subgraph isomorphism is the process of finding instances of a *pattern graph* in a *target graph*. The algorithm is used for graph and behaviour matching, see Sections 6.1 and 6.2. It is an NP-Complete problem, and so worst case solutions have exponential time complexity.

The method we use applies a brute force backtracking search to find all possible matches of the pattern in the target. Initially, potential target node matches in the pattern graph are identified. A node in the pattern graph is a potential match with a node in the target graph if the pattern node has the same or less degree than the target node. A backtracking search is then performed, taking a node `np` from the pattern and attempting to find a match from its matching set of target nodes. For each node in the target, `nt`, the neighbours of `np` and `nt` must correspond. That is, if a node is a neighbour of `np` and it has an assigned match in the other graph, the matching node must be a neighbour of `nt`. Similarly for neighbours of `nt`. If a match is found, the next node in the first graph is checked for possible matches. If no matches can be found for `np`, backtracking occurs, and the last previously matched node in the first graph is checked for other matches. If all nodes in the target are matched, the matching nodes are stored and the process continues. The process ends when all nodes have been tested.

An example of use of the isomorphism algorithm where a pure topological search is made, so that the nodes and edges are not restricted from matching any others in the target except for topological matching is:

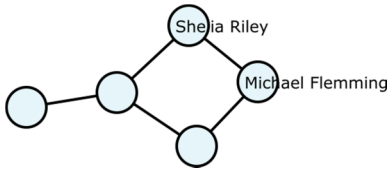
```
FastGraph target = FastGraph.randomGraphFactory(40, 200, 999, true, false);
FastGraph pattern = FastGraph.randomGraphFactory(10, 20, 1, true, false);
ExactSubgraphIsomorphism esi = new ExactSubgraphIsomorphism(target,pattern,null, null);
boolean result = esi.subgraphIsomorphismFinder();
LinkedList<SubgraphMapping> mappings = esi.getFoundMappings();
```

Where `SubgraphMapping` is a class that holds single mapping from the target to the pattern. `getFoundMappings()` should only be called after `subgraphIsomorphismFinder()` has been run.

Some timing information for randomly generated graphs for such unrestricted node and edge matches is shown in the table below:

Pattern Nodes	Pattern Edges	Target Nodes	Target Edges	Matches Found	Total Time
8	11	30	104	59654	1.31s
8	11	38	168	165882	18.8s
10	18	30	104	44	4.23s
10	18	38	168	1920	0.868s
12	21	30	104	10	0.03s
12	21	38	168	16004	3.05s
14	27	30	104	0	0.677s
14	27	38	168	5125	0.728s

The subgraph isomorphism constructor, `ExactSubgraphIsomorphism`, can be given node and edge comparators. In the above the default comparators are used to allow nodes and edges of any type and label to be matches (via `null` in the last two arguments). However, use of comparators can greatly reduce the search required in the graph, permitting subgraph isomorphisms to be found in very large graphs. If, for example, nodes are restricted to matching nodes with the same label, using the `SimpleNodeLabelComparator` and similarly `SimpleNodeLabelComparator`, then searching large graphs is feasible. Here we have a particular pattern involving two named nodes. Where there are labels in the pattern, these node and edge comparators only consider nodes and edges in the target with the same name. Pattern nodes and edges without labels are free to match any node and edge in the target.



Applying this to the 1.5 million node, 10 million edge graph derived from SNAP data [SNA16] gives these results:

Pattern Nodes	Pattern Edges	Target Nodes	Target Edges	Matches Found	Total Time
5	5	1500000	10000000	24	13.95s

5.3. Graph Rewiring

Motif finding (Sections 6.3 and 6.4) requires a reference graph to test the data graph against. The reference graph can be provided by the user, but in many cases such a reference graph is not available. Hence a random graph needs to be generated. This is usually based on the data graph. Typical approaches [WR06] attempt to keep similar characteristics of the data graph whilst randomising the connectivity. We adapt this approach by rewriting the graph but keeping the degree profile of the node set as it was in the original data graph. Rewiring is the process of taking one end of an edge and assigning it to a different node. If another edge of the newly assigned node is then removed to be reassigned elsewhere, the degree of the node is maintained, whilst changing the connectivity of the graph. In one iteration, all edges are rewired using this process, with the last rewiring move being to attach an edge to the first node from which an edge was removed. The effect is to change the connectivity of the graph radically, whilst maintaining the same number of nodes with a particular degree in the newly rewired graph. We restrict an edge's end point to moving just once in an iteration. The typical effect is to change over 99% of the edge connectivity in the graph on a single iteration.

A reference graph for behaviour graphs is created by taking the zero generation of the data graph, rewiring it, then creating new generations through a random node and edge deletion/creation mechanism.

5.4. Graph Sampling

There are numerous methods for sampling subgraphs from a larger graph and many have bias towards particular subgraphs. The method used is a neighbourhood sampling method which randomly builds a subgraph in a combined depth and breadth approach.

The user must first define how many subgraphs per node are to be generated and the number of attempts the system may take to build a subgraph.

The method takes each node in turn. From the node, a random neighbour is chosen and added to the subgraph along with the starting node. From then, one of the nodes in the subgraph is chosen at random and one of the neighbours of this node is chosen at random. If there are no neighbours, or the chosen neighbour is already in the subgraph, then another node and neighbour is picked at random instead. If the system continues to pick invalid nodes, up to the limit defined by the user, then the subgraph is abandoned.

Once the required number of nodes have been added to the subgraph, then any edges connecting these nodes are induced. This subgraph is now complete and stored for further use.

Nodes	Edges	Subgraph Size	Subgraphs Per Node	Attempts	Completion Time (s)	Number of subgraphs
1,500,000	10,000,000	4	5	10	**56**	**7,500,000**
1,500,000	10,000,000	8	5	10	**123**	**7,499,353**

5.5. Graph Edit Distance Approximation

Graph edit distance algorithms calculate the minimum number of changes required to one graph so that it is isomorphic to the second. As the basis of an approximate motif finding method, graph edit distance is suitable because it makes no assumptions about the properties of the graphs. Whilst exact graph edit distance has an exponential time [ARR15] complexity in the number of graph nodes, making it infeasible for larger graphs, inexact graph edit distance sacrifices accuracy for speed, so is a viable option.

Alternative measures, such as Graph Hashing and Eigenvalues were rejected because the output of these measures is not necessarily similar for similar graphs. Node and edge label

comparison was considered too simplistic. Random walks and subtree matching were examined, but they only have an indirect relevance to graph isomorphism and so their conversion to a graph similarity measure for this project would have been too time consuming.

Our approximation of graph edit distance is used to cluster motifs in the approximate motif finder. The clustering method is described in Section 5.6. To determine the difference between two graphs, one can use Graph Edit Distance. For performance reasons, in this system, an approximation to that calculation is performed. The difference in node degree profiles are compared to determine how “distant” one graph is to another. The node degree profile is a count of the number of nodes with a particular degree.

In the above example, the node profiles are shown in the table below. These two graphs have a difference of 6, and therefore unlikely to be assigned the same group.

Nodes with Degree	0	1	2	3	
Graph 1	0	3	2	0	
Graph 2	0	0	3	2	
Difference	**0**	**3**	**1**	**2**	**Total: 6**

In another example, two graphs appear isomorphic, except one is connected by time edges. This would be represented as follows:

Age	0			1			
Nodes with Degree	**0**	**1**	**2**	**0**	**1**	**2**	
Graph 1	0	0	4	0	0	0	
Graph 2	0	2	0	0	2	0	
Difference	**0**	**2**	**4**	**0**	**2**	**0**	**Total: 8**

For a 4-node subgraph, this approximation can complete just over 2million comparisons per second.

A previous implementation of approximate graph edit distance, developed by integrating code in the Graph Matching Toolkit was discarded as too slow for the size of data in this project.

Sample code for running the comparison score is as follows:

```
FastGraph g1 = FastGraph.loadBuffersGraphFactory(null, "g1");
FastGraph g2 = FastGraph.loadBuffersGraphFactory(null, "g2");
KMedoids km = new KMedoids(g1, 1, 1);
double result = km.comparisonScore(g1, g2); //compare graphs
```

The result of the comparison is stored in `result`.

If the graphs being compared have time edges dividing generations, then (a) the time edges are not considered in the degree calculations and (b) the degree profiles are split into generation degree profiles, rather than aggregated over the whole graph.

5.6. k-medoids

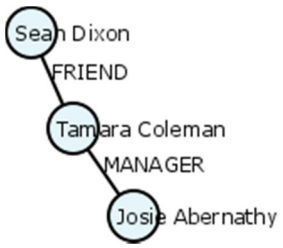
k-medoids is a clustering method which assigns points to various groups based upon the centre of that cluster. It is distinguished from many other clustering methods by not being reliant on the data points having a position in metric space. Rather, it uses a distance measure being applied to pairs of data points. When applied to the approximate motif finding, the algorithm will determine which motifs belong into which clusters based on their graph edit distance to the existing "central" motif (mediod). This process is iterative, with group assignment followed by mediod discovery (where the most central motif in a group is potentially re-assigned) repeated a number of times (specified by the user). For this project the distance measure applied was the graph edit distance approximation described in Section 5.5.

6. Graph Mining Methods

In this section we bring together the methods described in Section 5 into practical graph mining techniques usable on large graphs. We describe both exact and approximate variations of both graph matching and motif finding. We also show how these methods are adapted for behaviour graphs, which represent changes to the data over time, as described in Section 4.

6.1. Exact Structure/Behaviour Matching

The exact subgraph isomorphism algorithm determines all the subgraphs that exist with the target graph that are isomorphic to the pattern graph, using the specified comparators. The algorithm determines which nodes and edges are contained with the target graph and returns a list of node mapping – i.e. which nodes in the target graph match to which nodes in the pattern graph. From this, a subgraph can be built. Full details of the isomorphic algorithm are in Section 5.2. The user may specify a comparator by labelling a node or edge, or by writing a specific one for the user's needs. This algorithm is efficient at discovering all subgraphs where nodes are labelled, otherwise it can potentially return a large number of matching subgraphs. In the example below, a 3-node 2-edge graph is designed with the end nodes being `Sean Dixon` and `Josie Abernathy`. All 5 examples of this subgraph are found in 7 seconds, with one being shown below.



Nodes	Edges	Subgraphs Size	Time (s)	Results
2,700,000	11,200,000	3 nodes, 2 edges	7	**5**

Example code for running the exact subgraph isomorphism algorithm is as follows:

```
FastGraph g1 = FastGraph.loadBuffersGraphFactory(null, "g1"); //load graph 1
FastGraph g2 = FastGraph.loadBuffersGraphFactory(null, "g2"); //load graph 2
SimpleNodeLabelComparator snlc = new SimpleNodeLabelComparator(g1, g2);
SimpleEdgeLabelComparator selc = new SimpleEdgeLabelComparator(g1, g2);

ExactSubgraphIsomorphism esi = new ExactSubgraphIsomorphism(g1, g2, snlc, selc);

boolean result = esi.subgraphIsomorphismFinder();
```

If the `result` were `true`, then use `esi.getFoundMappings();` to return a `LinkedList<SubgraphMapping>` which contains a list of all the found subgraphs.

Behaviour graphs, where time slices are are present, have time edges connecting the nodes in adjacent generations (see Section 4). To find subgraphs in behaviour graphs, an edge comparator is used (for example, `TimeEdgeComparator`) to ensure that time edges match with only time edges and that non-time edges do not match with time edges.

6.2. Approximate Structure/Behaviour Matching

The approximate subgraph matching algorithm takes a sample of subgraphs from the target set and compares them with the given pattern graph. The number of enumerated subgraphs is specified by the user, along with the desired target graph. For each of these enumerated subgraphs, the exact subgraph isomorphism algorithm is run (see above), with a label comparator. This comparator may be changed by the user to a custom one written for the user's requirements (for example a comparator that ensures a particular edge is a time edge). Once all subgraphs have been compared, the result are output. The approximate subgraph isomorphism algorithm can run quicker than the exact version where a node will have potential matches with a large number of target nodes. The approximate method is also tuneable by specifying more or fewer sampled subgraphs.

Example code for running the approximate subgraph isomorphism algorithm is below:

```
FastGraph g1 = FastGraph.loadBuffersGraphFactory(null, "g1");
FastGraph g2 = FastGraph.loadBuffersGraphFactory(null, "g2");
SimpleNodeLabelComparator snlc = new SimpleNodeLabelComparator(g1, g2);
SimpleEdgeLabelComparator selc = new SimpleEdgeLabelComparator(g1, g2);

ApproximateSubgraphIsomorphism asi = new ApproximateSubgraphIsomorphism(g1, g2, 4, 1,
snlc, selc);
int count = asi.subgraphIsomorphismFinder();
```

As with the exact matching (Section 6.1), when behaviour graphs are the target graph the an edge comparator such as `TimeEdgeComparator` can be used to ensure that time edges match with only time edges and that non-time edges do not match with time edges.

6.3. Exact Motif Finding

Exact motif finding compares the occurrences of a particular motif in the dataset compared to how often that motif would be expected under normal conditions. To generate a reference graph for the normal conditions, the existing graph is rewired several times and subgraphs are enumerated for each rewiring (or the user may specify a reference set). These motifs are then grouped into particular classes, by using the serialized form defined in Section 5.2, which distinguishes between topologically equal motifs that have different time slice structure, so accounting for behaviour graphs. Each class may then contain several buckets, each bucket holding isomorphic motifs. Once assigned, the only one example of each bucket is saved along with a statistical analysis of the found motifs. This process repeats for each required size of motif, and the rewired reference graphs are saved for future use.

The process then repeats using the target graph. It follows a similar manner to the reference set, in that a large number of motifs are found and classified. Unlike the reference set, the user may choose to save every motif found and these are saved when discovered and classified.

Once both the reference and target graphs have been analysed, the system then compares the statistics for both sets. Motifs are ordered by their z-score, a measure of statistical significance, and displayed in HTML format. This allows a user to view an example of each motif type, along with various statistics. If the user requested the saving of all motifs, these are also displayed in this output.

Nodes	Edges	Subgraphs Tested	Size of Motifs	Time	Unique Results
2,000	10,000	19,000	4 & 5	611s	**389**
1,500,000	10,000,000	15,000,000	7 & 8	~15h	**349,000**

Example code for running the exact motif finder algorithm is as follows:

```
FastGraph g1 = FastGraph.loadBuffersGraphFactory(null, "g1");
int minNum = 4, maxNum = 6; //size of motifs
ExactMotifFinder emf = new ExactMotifFinder(g1, new MotifTaskDummy(), false);
emf.findMotifsReferenceSet(10,minNum,maxNum); //reference set
emf.findMotifsRealSet(minNum,maxNum); //real set
emf.compareMotifDatas(minNum,maxNum); //comparison
```

6.4. Approximate Motif Finding

The approximate motif finding algorithm using the k-medoids algorithm using an approximation of graph edit distance as described in Section 5.6. A number of subgraphs are enumerated for the various sizes of motifs required. These are then clustered by k-medoids into groups. The groups are then output in HTML format, with every example of motif in each group displayed. The runtime of this method can be compared to the exact motif finder. While this approximate algorithm runs ten times faster, it assigns the motifs into much broader groups. For behaviour graphs, the graph edit distance approximation is modified as described in Section 5.5.

Nodes	Edges	Subgraphs Tested	Size of Motifs	Time	Groups
2,000	10,000	19,000	4 & 5	67s	**4**

Example code for the running of the approximate motif finder algorithm is as follows:

```
FastGraph g1 = FastGraph.loadBuffersGraphFactory(null, "g1");
int minSize = 4, maxSize = 6;
int numOfClusters = 4, iterations = 2;
int subsPerNode = 5, attempts = 20;
KMedoids km = new KMedoids(g1, numOfClusters, iterations);
EnumerateSubgraphNeighbourhood esn = new EnumerateSubgraphNeighbourhood(g1);
HashSet<FastGraph> subs = new HashSet<FastGraph>();
for(int i = minSize; i <= maxSize; i++) {//build a list of potential subgraphs
  subs.addAll(esn.enumerateSubgraphs(i, subsPerNode, attempts));
}

ArrayList<FastGraph> subgraphs = new ArrayList<FastGraph>(subs);
ArrayList<ArrayList<FastGraph>> clusters = km.cluster(subgraphs);

km.saveClusters(clusters);
```

7. Future Work

We note that an alternative ByteBuffer representation, where each item attribute is stored in a separate ByteBuffer would increase the current limit on the number edges by a factor of around 10, making billion edge graphs feasible. This might be considered a more flexible and extendable representation, providing a mechanism to add new attributes by adding further ByteBuffers. The downside of this representation is the need to manage a large number of ByteBuffers. For further scalability, it would be entirely feasible to store attributes across multiple ByteBuffers, perhaps by using a long integer identifier for items. Further sophistication could make dynamic behaviour feasible, by e.g. modelling storage on database heap files. At this point, one might make a strong engineering case to move to a dedicated database storage mechanism. A simple database such as a key-value store (e.g. redis) maps well to the current implementation. An Entity Relational database such as Gaffer naturally stores graphs. Standard relational databases could be used, but we regard the mapping from graph to tables as not particularly intuitive. We note that use of an external database system would mean Dover was no longer a pure Java system, which may reduce portability and adversely impact ease of integration with other systems. A reduction in memory requirements may be achieved by efficient behaviour graph storage. At the moment repeated nodes and edges are duplicated through multiple generations. Where the behaviour graph does not have much variation between generations, a more compact representation might be considered. This, however, has a significant impact on the algorithm design.

Whilst it might be possible to convert to external systems for some of the functionality discussed in this section, it would also be possible to extend the current system with functionality. Extending the current system might be required where algorithms on large graphs are being applied. Here the algorithms would be implemented within Dover for greatest efficiency. However, for implementation speed, Dover could have integration modules added to connect with other graph analysis, data extraction and visualisation tools, such as Mamba,. This would allow the system to be combined with other products to harness the benefits and specialities of each.

The isomorphism, sampling and graph edit distance algorithms are limited by the time available for implementation and have been tuned to the particular applications. More efficient and general algorithms are certainly possible and many variations are given in the literature, such as for motifs: [SFS03], and for subgraph isomorphism: [FPV14]. A relatively low cost improvement is to modify the current algorithms for directed graphs. Given the underlying data representation already supports directed graphs, this would not be overly problematic.

Significant work on network analytics for security in large network graphs has been performed, looking, at, for example, malware detection in the cloud [CNW10]. It would be possible to integrate such metrics, along with other graph characteristics such as graph width, articulation points and measure of dynamic behaviour. In addition, the current Dover system would have more effective analysis capability if greater graph querying were added, for instance matching restrictions on number and type of neighbours and how they change over time, definition of matching paths and path lengths, including discovery of connectivity between nodes. Whilst these functions are present in other software systems. The benefit of integration into Dover is that they would perform quickly in an environment that can be easily installed on a secure, standalone machine.

8. Conclusions

We have described open source Java graph mining software for motif finding and structure discovery in graphs to fulfil Lot 5 of Data Analytics FY16/17. The work scales well beyond the current state of the art in both of these tasks and we can demonstrate both motif finding and structure discovery in graphs beyond the required minimum of 1 million nodes and 10 million edges. The project was exploratory in nature and the initial work here has the potential to be built on further, as current software can be expanded, as well as integrated with other systems. More details on the project and code can be found at: https://www.cs.kent.ac.uk/projects/dover/

References

[ARR15] Z. Abu-Aisheh, R. Raveaux, J.Y. Ramel and P. Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. In 4th International Conference on Pattern Recognition Applications and Methods 2015.

[Bai09]. T.L. Bailey et. al. MEME Suite: tools for motif discovery and searching. Nucleic Acids Research May 2009 doi: 10.1093/nar/gkp335

[CHC12] D. O'Callaghan, M. Harrigan, J. Carthy and P. Cunningham. Network analysis of Recurring YouYube Spam Campaigns. arXiv preprint arXiv:1201.3783. 2012.

[CNW10] D.H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos, C. Polonium: Tera-scale Graph Mining for Malware Detection. In 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining., 2010, July

[Coc10] https://www.biostars.org/p/1779/. Visited 2 September 2016.

[DM15] T. Davies, E. Marchione. Event Networks and the Identification of Crime Pattern Motifs. PLoS ONE 10(11): e0143638. doi:10.1371/journal.pone.0143638. 2015.

[FPV14] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. International Journal of Pattern Recognition and Artificial Intelligence 28(01). 2014.

[Fan12] W. Fan. Graph pattern matching revised for social network analysis. In Proceedings of the 15th ACM International Conference on Database Theory, pp. 8-21..

[SNA16] Stanford Large Graph Database. https://snap.stanford.edu/data/. Visited 1st September 2016.

[SFS03] M. De Santo, P. Foggia, C. Sansone and M. Vento, A Large Database of Graphs and its use for Benchmarking Graph Isomorphism Algorithms, Pattern Recognition Letters, 24(8): 1067-1079, May 2003. doi:10.1016/S0167-8655(02)00253-2.

[WR06] S. Wernick and F. Rasche. FANMOD: a tool for fast network motif detection. Bioinformatics, 22(9):1152–1153, 2006.