Graph Edit Distance Project

Roman Tekhov

Objectives

Since graphs are widely used in order to represent data (e.g. process models) it might be important to determine and measure the level of similarity between individual graphs. In the field of text processing there is a concept of string edit distance, which represents a minimal number of edit operations that have to be performed on the first string to obtain the second. Each operation is assigned a certain cost and the edit distance represents the sum of costs of all individual edit operations (a minimal sum from all possible variants). First objective of this work is to try to implement the same approach for graph comparison, i.e. develop an algorithm which would be able to determine the edit distance between two input graphs. The second objective is to develop a small application which would allow providing input graph data, evaluating the algorithm and displaying results for verification and analysis.

Graph edit distance finding algorithm

Definitions

Suppose we have two graphs, a source and a destination, that need to be compared. The goal is to obtain the destination graph by performing the so-called edit operations on the source graph. Possible operations are: insertion and deletion of nodes and edges and substitution of nodes. Each operation has some predefined cost. Node/edge insertion/deletion costs can theoretically be set to some static values but node substitution has to take into account the similarity of node labels, i.e. substitution of nodes with more similar labels has a smaller cost than substitution of nodes with more distinct labels.

Let us define *node edit path (NEP)* as a mapping between a source node and a destination node involving a set of edit operations. In case of node insertion or deletion the source or destination nodes might be empty. This mapping might possibly include more than one edit operation because node insertions, deletions and substitutions might trigger the edit operations on the adjacent edges as well. E.g. if the node is deleted then we also have to delete all edges connected to it. The primary node edit operation and the secondary edge edit operations are all included in a single NEP and the total cost of this path is the sum of costs of all its operations.

Let *graph edit path* (*GEP*) be a set of individual node edit paths between the two graphs. I.e. GEP is a mapping of multiple source nodes to corresponding destination nodes involving edit operations. The edit path is said to be complete if it includes node edit paths for all source and destination nodes, otherwise it is partial. The total cost of the edit path is the sum of costs of all its node edit paths.

Edit operations are performed one by one until the source graph is fully transformed into the destination graph or until the minimal possible edit path cost exceeds some predefined upper limit which means that the process has failed and there is no possible edit path with an acceptable cost.

Algorithm in detail

At each step the algorithm chooses the best partial edit path known so far (which has the smallest cost) and extends it by mapping the next source node in all possible ways. If all source nodes are mapped then it adds insertions of all remaining destination nodes. All these new extended paths will be considered for further extension later. The process might be represented logically as a tree structure where each node represents a partial edit path. At each stage the best leaf node is chosen and extended further to the next level. Technically all candidate edit paths are simply held in a priority queue sorted by their costs in ascending order.

Input: source graph *S*, destination graph *D*, edit costs, maximum acceptance limit *L*.

Output: edit path from *S* to *D* or the limit exceeded error.

```
FindGed(S, D, L) {
        Q = empty priority queue; // Sorts edit paths by total cost
        initialize(S, D, Q);
        while(true) {
                e = extract(Q); // First edit path from queue which has the smallest cost
                if (e is complete) {
                        return e;
                }
                if (e.cost > L) {
                        return error; // Acceptance limit exceeded
                }
                extend(e, S, D, Q);
        }
}
initialize(S, D, Q) {
        m = random node of S;
        for each n in D {
                se = substitution of m and n edit path;
                add se to Q;
        }
        de = deletion of m edit path;
        add de to Q;
}
extend(e, S, D, Q) {
        if(not all S nodes mapped in e) {
                m = next S node;
                for each n in D {
                        se = e + substitution of m and n NED; // Extend e with substitution
                        add se to Q;
```

```
                }
                de = e + deletion of m NED; // Extend e with deletion
                add de to Q;
        } else {
                for each n in D { // All remaining D nodes
                        ie = e + insertion of n NED; // Extend e with insertion
                        add ie to Q;
                }
        }
}
```

During its initialization stage the algorithm chooses one random source node and maps it to each destination node thus producing n small partial edit paths where n is the number of destination nodes. All these paths are added to queue. Then the algorithm calculates another small edit path for the chosen node deletion and also adds it to queue. Point of that stage is to obtain the initial set of partial edit paths in order to proceed with the main loop.

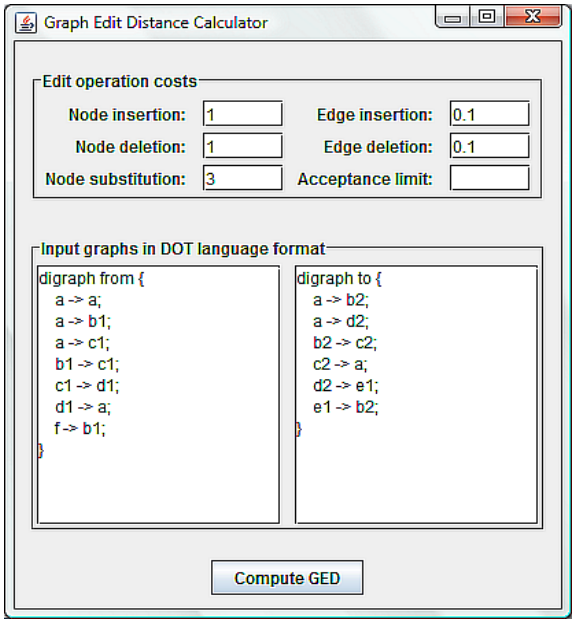
At each stage of the main loop the algorithm extracts the first edit path from queue (the one with the smallest cost known so far). If the path is already complete then it is returned as output. If the path's cost exceeds the maximum acceptable limit then the process terminates and reports an error. If the best path is not yet complete, limit is not exceeded and there are still some unmapped source nodes left then the path is extended like so:

1. Choose the next unmapped source node;
2. Create substitution node edit paths between this chosen node and each unmapped destination node.
3. For each such substitution NED create a new graph edit path which represents a copy of extracted minimal path + NED and add it to queue.
4. Create a new graph edit path which represents a copy of extracted minimal path + chosen node deletion NED and add it to queue.

If all source nodes have been mapped then the algorithms extends the extracted edit path with insertion node edit paths for each remaining destination node and adds it to queue.

GedFinder application

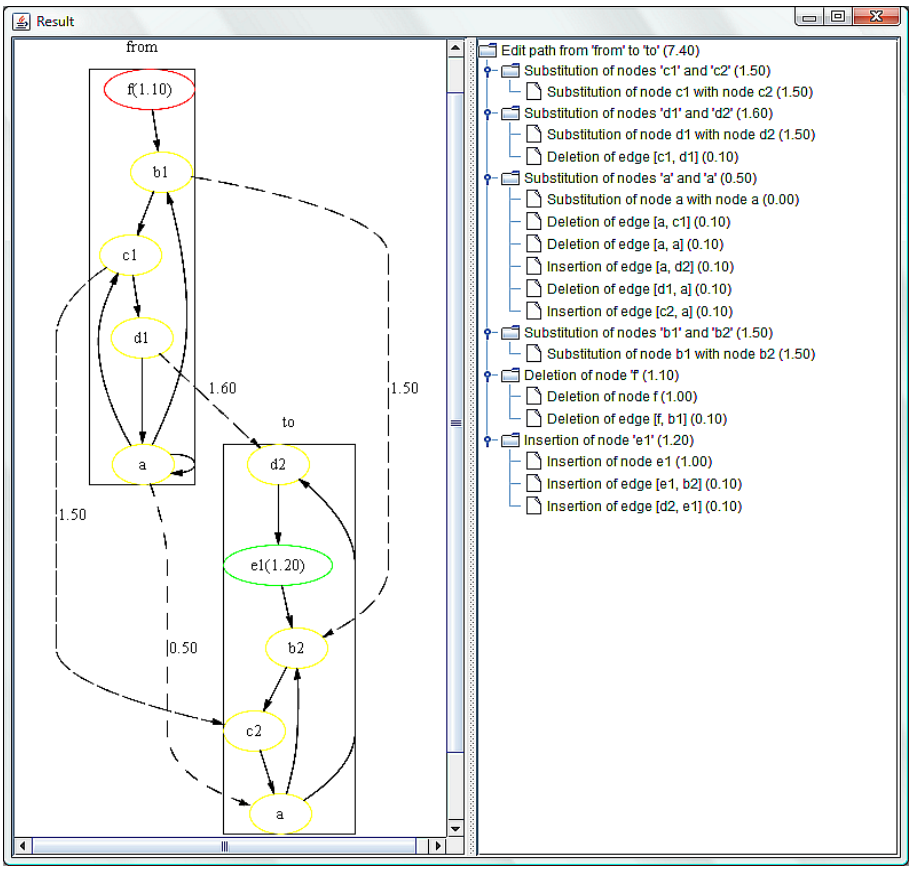
The program is written in Java. The graphical user interface is built using Swing. The main window allows inputing cost values for edit operations and for the maximum acceptance limit and the two graphs in DOT language format. The left graph will be treated as the source and the right graph will be the destination. The “Compute GED” button triggers the calculation process and will be disabled until the output is returned.



DOT format conversion is done using Grappa parsing mechanism. The problem is that the algorithm needs to access neighbor nodes frequently and Grappa's *Node* class doesn't provide any efficient way of doing so. For that reason Grappa nodes are wrapped in *DecoratedNode* instances which hold the adjacency lists for every node. These lists allow to access neighbor nodes in a convenient and fast way. The *GraphConverter* class is responsible for DOT conversion delegation to Grappa, building of adjacency lists and wrapping into decorated structures.

The *EditPathFinder* class contains the algorithm implementation and returns an instance of *EditPath* class as a result. *EditPath* contains references to individual *NodeEditPath* instances which in turn reference their underlying *EditOperation* objects. The *EditOperation* class is a common abstraction and has a specific subclass for each type of edit operation: *NodeDeletion*, *NodeInsertion*, *EdgeDeletion*, *EdgeInsertion*, *NodeSubstitution*. Each class holds references to its relevant nodes and this information is used later to get a graphical representation of edit operations.

In case of successful output the result window is shown which contains a list of all edit operations and a graphical representation of edit path. This presentation includes both input graphs where deleted nodes are shown as red, inserted nodes are shown as green and all the rest (substituted) nodes are shown as yellow with additional dashed edges connecting them. The *GraphConverter* class produces the combined and supplemented graph and Grappa's *GrappaPanel* class is used to do the actual graphical rendering. The problem is that Grappa does not provide any layout information (all nodes are positioned in the same location) and doing this by hand would require a complex algorithm of its own. That is why external layout tools are used to delegate the problem to. “Dot” is used for directed graphs and “neato” is used for undirected graphs. Both these programs are included in the GraphViz package and have to be on the machine's path when the application runs. The output for the previous input looks like this:



The file sample-graph.txt contains the example graphs text in DOT format.

Resources

GraphViz - http://www.graphviz.org/ (contains information about DOT language and allows to download the required programs).