PROFI

Project number:	FP6-511572
Project acronym:	PROFI
Title:	Perceptually-relevant Retrieval Of Figurative Images

Deliverable No: D7.2: Implementation of Layout Indexing

Short description:
We implemented the algorithm for layout indexing described in deliberable D7.1. For a given query and a large database, the problem of layout indexing is formulated as efficiently selecting candidate models, which have similar layouts as the query. In our framework we represent the layout of an image with multiple shapes as as a graph whose nodes correspond to shapes and whose edges show relations between the shapes. In this representation the problem of layout indexing is transformed into that of graph-based indexing. Our graph indexing algorithm is based on a powerful characterization of graphs using laplacian matrices. We draw on an important theorem from spectral graph theory to show that our graph characterization can be used to retrieve similar graphs or subgraphs from the database. Emperical evaluation of the algorithm on an extensive set of recognition trials demonstrates both the robustness and efficacy of the overall approach.

Due month:	M24
Delivery month:	M24
Lead partner:	Utrecht University
Partners contributed:	Utrecht University
Classification:	PU

Project funded by the European Community under the "Information Society Technologies" Programme

1 Results

1.1 Introduction

The objective of work package (WP) 7 is defined as fast selection of candidate images, which have similar layouts as the query. The implementation described in this report is based on the layout indexing framework developped for WP 7. Our framework starts by representing images as graphs, where the vertices correspond to shapes and the edges show the relations between the shapes. This representation enables us to formulate the problem of layout indexing as that of indexing based on graph structures. In many applications especially in information retrieval, the computational complexity of graph matching is further compounded by the fact that query graphs are to be matched against a large graph database. Therefore, when working with graphs, indexing is a challenging task and can be defined as the problem of efficiently selecting a small set of database graphs, which share a subgraph with the query.

Our framework represents the topological structure of a graph as well as that of its subgraphs as vectors in which the components correspond to the sorted laplacian eigenvalues of the graph or subgraphs. Our motivation for this representation comes from the fact that laplacian matrices are more informative about graphs than other matrix types such as adjacency matrix. Thus, by using the laplacian spectrum as a signature, we capture the graph topology to a large extent. The signature of a graph is invariant under the reorderings of its vertices. This, in turn, allows us to compare the signatures of a large number of graphs without solving the computationally expensive correspondence problem between their vertices. Given the laplacian spectrum of a principal submatrix of a matrix, we draw on an important theorem from spectral graph theory to show that our graph characterization can be used to retrieve similar graphs or subgraphs from large database systems. Given a query graph, a voting schema ranks database graphs into an indexing hypothesis to which a final matching process can be applied.

In the following subsection, we will summarize our indexing framework. The details can be found in deliverable D7.1.

1.2 Indexing by Laplacian Spectra

Our framework encodes the topology of a graph through its laplacian spectrum. Laplacian matrix $L(G)$ of graph G is computed as $L(G) = D(G) - A(G)$, where $D(G)$ is the degree matrix and $A(G)$ is the adjacency matrix for G. The spectrum of a graph's laplacian matrix is obtained from its eigendecomposition. More formally, the eigendecomposition of a laplacian matrix is $L(G) = P\Lambda P^T$, where $\Lambda = \text{diag}\,(\lambda_1, \lambda_2, \ldots, \lambda_{|V|})$ is the diagonal matrix with the eigenvalues in increasing order and $P = (p_1|p_2|\ldots|p_{|V|})$ is the matrix with the ordered eigenvectors as columns. The laplacian spectrum is the set of eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_{|V|}\}$.

In our framework, we define the signature of a graph as the sorted eigenvalues of its laplacian matrix. To compute the similarity between two graphs, we compute the Euclidean distance between their signatures, which is inversely proportional to the structural similarity of the graphs. For a given query, retrieving its similar graphs, therefore, can be reduced to a nearest neighbor search among a set of points. Unfortunately, this formulation cannot deal with occlusion or segmentation errors as two graphs may share similar structures up to only some level. Although adding or removing edges changes the laplacian spectrum, the spectrum of the subgraphs that survive such alteration will not be affected. Our indexing mechanism,

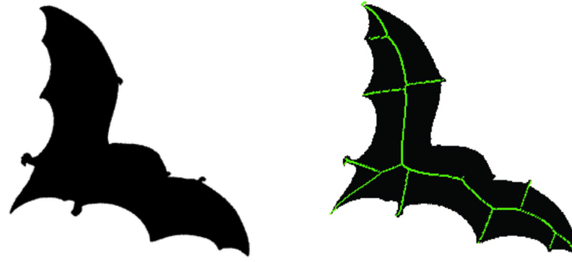

Figure 1: Left: a view of a bat. Right: the shock graph constructed from the medial axis and superimposed on the left image.

therefore, cannot depend on the signature of the whole graph only. Instead, we will combine the signatures of the subgraphs in the framework.

For a given database graph $G = (V, E)$, we compute the signatures of each subgraph of G in our algorithm. In this process, we gradually increase the size of the subgraphs. Since the sorted eigenvalues are invariant under consistent re-orderings of the graph's vertices, it is sufficient to compute the spectrum of permutation-similar matrices once. Associated with each signature in the system is a pointer to the corresponding graph or subgraph in the database. At runtime, we first generate the signature of each subgraph of the query. Given a query signature s_q, we then retrieve its nearest neighbors of the same size from the database through a nearest neighbor search. Each neighbor of s_q retrieved from the database gets a vote whose value is inversely proportional to the distance from s_q. Thus, as a result, each signature of the query generates a set of votes. Moreover, we weigh the votes according to the size of the subgraphs corresponding to the signatures, i.e., the bigger the size, the more weight the vote receives.

After performing a nearest neighbor search around the query signatures, we compute the weights of the votes between the query and the database graphs having at least one signature as the nearest neighbor of the query. We then sort the database graphs based on these weights. In this process, we only add the sufficiently high-support database graphs to the indexing hypothesis. Since a small number of structurally different graphs may also share the same laplacian spectrum, each graph in the hypothesis should still be verified by some matching algorithm. Despite the fact that such graphs may exist in the indexing hypothesis, the number of them is very small. In addition, as shown in deliverable 7.1, not only do isomorphic graphs share the same signature, non-isomorphic but similar graphs or subgraphs have close signatures in the vector space. The database, therefore, can be pruned without losing structurally similar graphs.

1.3 Experimental Evaluation

To examine the fitness of the new indexing framework, we have performed a number of experiments using an extensive set of recognition trials in the domain of 2-D and 3-D object recognition. We first perform our experiments using silhouettes. For a given shape, its silhouette is represented by an undirected shock graph. The graph is then constructed from the discrete skeleton. An illustration of this type of graph is given in Figure 1, where the left portion shows an input image taken from the database, while the right portion presents the constructed shock graph superimposed on top of the image. We used the MPEG-7 dataset CE-Shape-1 part B for this representation type. The database consists of 70 classes

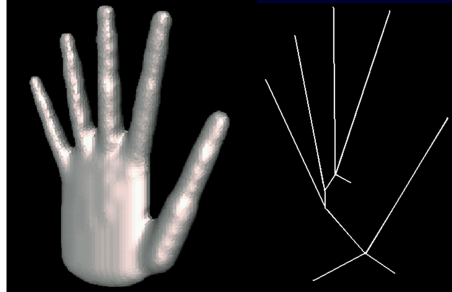

Figure 2: The Reeb graph constructed for the object on the left is shown on the right.

and 20 shapes per class. We also conduct our experiments in the domain of 3D object recognition using Reeb graphs. These graph representations allow for topological properties to be represented in a coarse sense. The second database used in the experiments consists of Reeb graphs constructed for the McGill 3D Shape Benchmark. The database consists of 420 objects classified in 19 classes. The right of Figure 2 shows a Reeb graph constructed for the object shown in the left.

We first represent each object in each database as a graph. Given a graph, we compute the signatures for each of its subgraphs and populate the resulting signatures in the vector space. In our experimental setup, we applied the following leave-one-out procedure to the datasets for evaluation. We initially remove the first graph from the database and use it as a query for the remaining database graphs. The graph is then put back in the database, and the procedure is repeated with the second graph from the database, etc., until all database graphs have been used as a query.

There exist several performance measures to assess the quality of a retrieval system or indexing mechanism. Precision and recall are two well-known examples. In some applications high precision is necessary, meaning that the relevant items that are returned must be at the top of the ranking. In some other applications, however, high recall is preferred, meaning that false negatives are to be avoided (the returned result must contain all or the most relevant objects). A good indexing system should, in fact, perform well according to both of these two measures. We conducted two sets of experiments to cover both scenarios. In the first experiment, the class of the query should be determined quickly (best match must appear high in the ranking). In the second experiment, all the objects belonging to the query class should be returned in a small candidate set.

The results for these experiments are presented in Table 1. For comparison purposes, the same experiments were also conducted with the adjacency spectrum representation. For each query, the database graphs are ranked in decreasing order of the vote weights. In in 37.3% of the cases, the highest-weight database graph belongs to the correct shape class for shock graphs and this ratio is 29.6% for Reeb graphs. Moreover, the average position of the closest matching graph among the highest-weight candidates is 7.4 and 4.3 for shock and Reeb graphs, respectively. In addition, the worst position of the closest matching graph is 12 for shock graphs, while this number is 9 for Reeb graphs. These results present that to determine the correct class of the query, more than 99% and 97% of shock and Reeb graph datasets can be pruned by our indexing mechanism.

While the previous evaluation method is suitable for classification tasks, in some retrieval applications, however, it is a prerequisite to retrieve all images from the database that belong

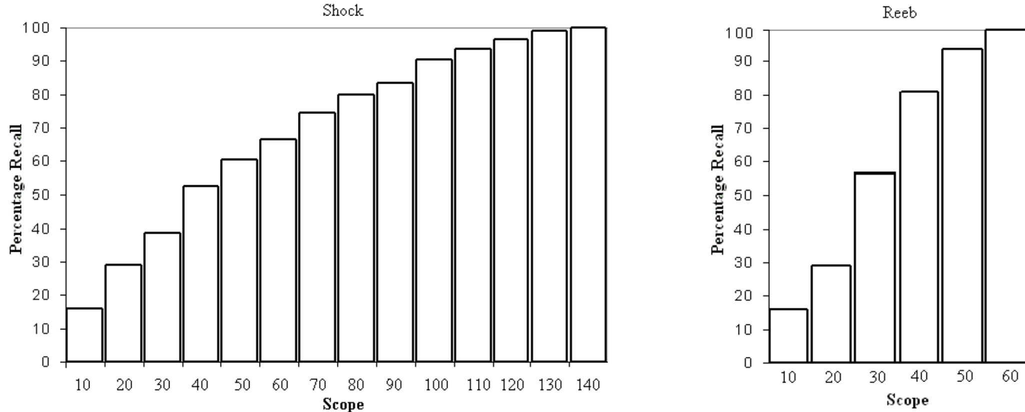

Figure 3: Percentage recall values for various top ranked highest-weight candidate graphs for shock graphs of MPEG-7 and Reeb graphs of McGill datasets.

Criteria	HW(%)	AP	WP	PC(%)	PI(%)
Shock, Lapl.	37.3%	7.4	12	99%	90%
Reeb, Lapl.	29.6%	4.3	9	97%	87%
Shock, Adj.	18.0%	19.1	23	97%	82%
Reeb, Adj.	17.3%	10.2	22	94%	78%

Table 1: Results for indexing systems constructed using eigenvalues for Laplacian and Adjacency matrices. HW: Percentage of highest-weighted graph belonging to the same class as the query, AP: Average position of the closest matching graph from the query class, WP: Worst position of the closest matching graph from the query class, PC: Percentage of database that can be pruned to determine the right query class, PI: Percentage of database that can be pruned to retrieve all instances of the query.

to the query class. In the second experiment, the system's performance is evaluated by computing the total number of retrieved images that is necessary to retrieve the entire query class (maximum minimal scope). Our results show that the first 134 of the candidate return set always contains all the graphs belonging to the query class for shock graphs; this number is 54 for Reeb graphs. This indicates that for this task our framework prunes more than 90% and 87% of the shock and Reeb graph datasets, respectively. In other words, the recall in each dataset is 100% if the scope is set to the first 10% and 13% of the sorted candidate models for shock and Reeb graphs respectively. In Figure 3, we show percentage recall values for various scopes for shock and Reeb graph datasets.

2 Layout Indexing Software

The layout indexing software consists of two subprograms; one for creating distinct subgraphs and their laplacian spectra, the other for computing the similarity in accordance with our local indexing formulation. We will provide the documentation for each program below.

2.1 Creation of subgraphs and laplacian spectra

For a given graph, this code finds the distinct subgraphs and computes their laplacian spectra. Before describing the functions, we will introduce the file format for an input graph.

Input Graph Format: The input format shows information for vertices and edges. Each vertex and edge have some attributes. The comment character is "%". A sample input file is shown below.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The file starts with the number of vertices.
% Then for each vertex, we list its index and attributes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
0 [-1, -1, -1]
1 [-1, -1, -1]
2 [-1, -1, -1]
3 [-1, -1, -1]
4 [-1, -1, -1]
5 [-1, -1, -1]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% We now write down the number of edges
% and for each edge, we list the source vertex, target vertex, and both directional
% and topological attributes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
5
0 1 [ (SN, 0.5) (OL, 1.0)]
%edge between 0 and 1.
%Direction: SouthNorth with weight 0.5. Topology: OverLap with weight 1.0:
0 2 [( EW, 0.3) (OL, 1.0)]
1 2 [(SN , 0.4) (OL, 1.0)]
1 3 [ (NE, 0.2) (OL, 1.0)]
3 0 [ (NW, 0.7) (OL, 1.0)]
```

In the current version, the code does not take into account vertex attributes. Topological information in the edges are not used either. These fields are kept in the file format for future use. For an edge, there are 4 main directions: East-West (EW), South-North (SN), North-East (NE), North-West (NW). For a given file in the above format, the following program computes its subgraphs and for each subgraph it finds laplacian eigenvalues.

Function Prototypes

- void Show-copyright();
- char* Extract-file(const char*, const char*);
- void Process(const char*);

- int Is-white-comment(const char*);
- char* Get-dir(const char*, int);
- float Get-weight(const char*, int);
- void Create-subgraph(const char*, const int&, edge**, const int&);
- void Count-subgraphs(int*, const int&, int, const int&, const char*, edge**, const int&);
- void Generate-subgraph-file(int*, const int&, const char*, edge**, const int&);
- int Is-repeated(int*, const int&, const int&);
- int Is-connected(int*, const int&, edge**, const int&, char*);
- void Print-adjacency(edge**, int);
- void Is-path-notype(const int&, const int&, edge**, const int&, int&, int*, int*, const int&);
- void Is-path-with-type(const int&, const int&, edge**, const int&, int&, int*, int*, const int&, char*, char*);
- void Print-array(int*, int);
- void Count-subgraphs-2(int *array, const int&, int&, const char*, edge**, const int&, int&);
- void Get-prefix(const int&, char*, char*);
- void Compute-eigen(const char*, int&, int&);
- void Create-type-dirs();
- void Move-type-dirs();
- int Get-subgraph-no(const char*);
- int is-same-graph(const char*, const DiagonalMatrix& L-E, const int&, const int&, int&);

Public Types

```
typedef struct {
 int present;
 char dir[20], top[20];
 float dir-weight, top-weight;
} edge;
```

2.1.1 Function Documentation

void Show-copyright();
Displays the copyright information on the screen. This function is called when insufficient number of parameters is given.

char* Extract-file(const char* string1, const char* string2);
Extracts the second string from the first one given as parameters. For example, let string1 = "DATABASE/sample-input1.DAT", string2 ="DATABASE/" Then, the function returns "sample-input1.DAT"

void Process(const char* in-file);
For a given input file, this code reads the graph information, allocates space, and initializes its adjacency matrix. Creation of subgraphs is called from here.

int Is-white-comment(const char* buffer);
Checks if buffer is a white character or starts with the comment %. If so 1, otherwise 0 is returned.

char* Get-dir(const char* buffer, int pos);
Finds and returns the type of an edge. If pos is 1 then directional edge type, if pos is 2 then topological edge type is returned.

float Get-weight(const char* buffer, int pos);
Finds and returns the weight of an edge. If pos is 1 then directional edge weight, if pos is 2 then topological edge weight is returned.

void Create-subgraph(const char* in-file, const int& curr-size, edge adjacency, const int& no-vert);**
Main sub-program to create subgraphs of size curr-size. no-vert shows the number of vertices in the graph. adjacency is the adjacency matrix representation of the graph. Generate-subgraph-file() and Count-subgraphs-2() are called from here.

void Generate-subgraph-file(int* array, const int& size, const char* in-file, edge adjacency, const int& no-vert);**
Given a subgraph file, this code first checks if it has been produced before. If not, it then writes the subgraph to the file, whose name is computed using in-file and size.

void Count-subgraphs-2(int *array, const int& size, int& current-index, const char* in-file, edge adjacency, const int& no-vert, int& done);**
A recursive program to compute all subgraphs of a given graph. When all subgraphs are computed "done" becomes 1, and the code quits.

int Is-repeated(int* array, const int& size, const int& no-vert);
For controlling purposes, the code checks if one vertex is used more than once in the subgraph. Returns 1, if one vertex is repeated. Returns 0, otherwise.

int Is-connected(int* array, const int& size, edge adjacency, const int& no-vert, char* edge-type);**

Finds if the subgraph is connected by checking if the graph contains a path between each vertex pair. If so, it returns 1 and returns 0 if the subgraph is disconnected.

void Print-adjacency(edge adjacency, int no-vert);**
Prints an $n \times n$ 2-dimensional array with type "edge" on the screen. The value of n is specified by the second parameter. This is created for debugging purposes.

void Is-path-notype(const int& s-vertex, const int& t-vertex, edge adjacency, const int& no-vert, int& found, int* color, int* array, const int& size);**
Checks if there is a path between vertices s-vertex and t-vertex without taking into account edge types. If such a path is present in the graph then "found" becomes 1. Otherwise "found" is set to 0. This code is called from Is-connected().

void Is-path-with-type(const int& s-vertex, const int& t-vertex, edge adjacency, const int& no-vert, int& found, int* color, int* array, const int& size, char* prev-type, char* edge-type);**
Checks if there is a path between vertices s-vertex and t-vertex such that edge types on the path are the same. If such a path is present in the graph then "found" becomes 1. Otherwise "found" is set to 0. This code is called from Is-connected().

void Print-array(int* array, int size);
Prints a one dimentional array to the screen. This code is created for debugging purposes.

void Get-prefix(const int& size, char* pre-size, char* pre-file);
Inserts zero in front of the filename so that the length of the files becomes equal to each other. Final names are written in "pre-file".

void Compute-eigen(const char* in-file, int& same, int& reset);
Reads the graph written in "in-file" and computes the eigenvalues for its Laplacian matrix using publicly-available Newmat library. For comparison purposes, this code also computes the eigenvalues for the Adjacency matrix. The eigenvalues are appended at the end of the file.

void Create-type-dirs();
Includes a bunch of system calls to create subdirectories for storing the subgraphs with eigenvalues.

void Move-type-dirs();
Includes a bunch of system calls to move subdirectories storing the subgraphs.

int Get-subgraph-no(const char* file-name);
Finds and returns the subgraph number from the file-name.

int is-same-graph(const char* file, const DiagonalMatrix& L-E, const int& no-vert, const int& subgraph-no, int& reset);
Checks if the graph has been encoded in the system by comparing its eigenvalues to those listed in DiagonalMatrix L-E. If so the code returns 1 and returns 0 otherwise.

2.1.2 User Documentation

We used Newmat, a C++ library to manipulate a different types of matrices using standard matrix operations for computing the eigenvalues. For comparison purposes, the code computes eigenvalues for both laplacian and adjacency matrices. The details about this library can be found at http://www.robertnz.net/nm_intro.htm. The user needs to give an input directoty that contains the list of graphs in the specified format (see Section 2.1). When the executable is called only, the following appears:

```
$ ./Debug/create_subgraphs\

usage: ./Debug/create_subgraphs  [input_directory]

 where
      input_directory     Directory that contains input files.


Copyright (C) 2006 by M. Fatih Demirci.

 This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the the License, or
at your option) any later version.
```

The following is the output of the code when it is used with a proper parameter.

```
$ ./Debug/create_subgraphs\  input
Processing input1.DAT
Processing input2.DAT
Processing input3.DAT
Processing input4.DAT
Processing input5.DAT
Processing input1.DAT
Processing input2.DAT
Processing input3.DAT
Processing input4.DAT
Processing input5.DAT

Subgraphs with their eigenvalues are computed succesfully. They can be found in Sub_Graphs/
```

The program processes each file twice, one for untyped subgraphs and the other for typed subgraphs. Recall that edge types are given in the input files. Typed graphs are graphs where no edge type is taken into consideration. Similarly, in typed subgraphs, edge types are used to form subgraphs. Both typed and untyped subgraphs with eigenvalues are kept under Sub_Graphs/.

2.2 Computing the distance using laplacian spectrum

Once the subgraphs are computed, we compute the distance between pairs of graphs using our local indexing approach decribed in deliverable 7.1. In the following, we first give the input file format and describe the subprograms used in the software. Note that the input files to this program is produced automatically by the subgraph creation tool described in the previous subsection.

Input Graph Format: The file starts with the number of vertices, then for each vertex its index and one attribute is listed. For internal use, there is one 0 and the number of edges are shown. For each edge, we have source and target vertices and the type of the edge indicated. If "NT " is used as the edge type then edge types are not considered to form the subgraphs. Finally, we list both laplacian and adjacency eigenvalues. Sample file for a subgraph is shown below. Since this file format is produced automatically, no comment character is defined.

```
6

0 0.1
1 0.1
2 0.1
3 0.1
4 0.1
5 0.1

0

7

0 1 NT
0 5 NT
1 2 NT
1 4 NT
2 3 NT
3 4 NT
4 5 NT

0 Lapl_eigen: 0.000000 Adj_eigen: -2.414214
1 Lapl_eigen: 1.000000 Adj_eigen: -1.000000
2 Lapl_eigen: 2.000000 Adj_eigen: -0.414214
3 Lapl_eigen: 3.000000 Adj_eigen: 0.414214
4 Lapl_eigen: 3.000000 Adj_eigen: 1.000000
5 Lapl_eigen: 5.000000 Adj_eigen: 2.414214
```

Function Prototypes

- void Show-copyright();
- char *Extract-file(const char*, const char*);
- int Get-no-files(const char*);
- void Compute-distances(const char*, const char*, const char*, double&, double&, int&);
- int Get-no(const char*);
- void Get-matrices(const char*, double*, double*);
- void Vector-dist(double*, double*, double*, double*, double&, double&, int);
- int Get-size-no-type(const char*, const char*);
- #define MAX(x,y) ((x) > (y) ? (x) : (y)) /* returns the maximum of two numbers */
- #define MIN(x,y) ((x) < (y) ? (x) : (y)) /* returns the minimum of two numbers */

2.2.1 Function Documentation

void Show-copyright();
Displays the copyright information on the screen. This function is called when insufficient number of parameters is given.

char* Extract-file(const char* string1, const char* string2);
Extracts the second string from the first one given as parameters. For example, let string1 = "DATABASE/sample-input1.DAT", string2 = "DATABASE/" Then, the function returns "sample-input1.DAT".

int Get-no-files(const char* file);
Returns the number of lines in "file".

void Compute-distances(const char* query, const char* model, const char* input-dir, double& lalp-dist, double& adj-dist, int& flag-done);
For a given pair of files, query and model, this code computes the distance between using their laplacian the adjacency eigenvalues. The distances are returned with lalp-dist and adj-dist. For graphs with several distinct subgraphs, the Hassdorrf distance is computed.

int Get-no(const char* file);
Returns the number of vertices in a file in the input format.

void Get-matrices(const char* file, double* A, double* L);
For a graph written in "file", this code extracts its adjacency and laplacian matrices. The matrices are written in double *A and double *L.

void Vector-dist(double* A1, double* L1, double* A2, double* L2, double& curr-lap, double& curr-adj, int size);
Computes the Euclidean distance between A1 and A2, and between L1 and L2. The distances

are returned with curr-lap and curr-adj. This code is called from Compute-distances();

int Get-size-no-type(const char* file, const char* input-dir);
Returns the size of a subgraph with name "file" located under "input-dir".

2.2.2 User Documentation

The code takes 2 parameters to run. First parameter is a list of model files, each appearing in one line. The second parameter is the directory that contains the model files. The output is an $n \times n$ distance matrix, where n is the number of model files. When the executable is called only, the following appears:

```
$ ./Debug/compute_distance\

2 arguments needed.

usage: ./Debug/compute_distance  [model list] [directory, where each file in model
list with different sizes to be found]


Copyright (C) 2006 by M. Fatih Demirci.

 This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the the License, or
(at your option) any later version.
```

Assume the code is called for 4 model files (2x2-898127.grp, 2x2-920416.grp, TRN-1042176.grp, TRN-1055897.grp) given in the first parameter.

```
$ ./Debug/compute_distance\  list.lst Sub_Graphs/
Processing 2x2_898127.grp v.s. 2x2_898127.grp
Processing 2x2_898127.grp v.s. 2x2_920416.grp
Processing 2x2_898127.grp v.s. TRN_1042176.grp
Processing 2x2_898127.grp v.s. TRN_1055897.grp
Processing 2x2_920416.grp v.s. 2x2_898127.grp
Processing 2x2_920416.grp v.s. 2x2_920416.grp
Processing 2x2_920416.grp v.s. TRN_1042176.grp
Processing 2x2_920416.grp v.s. TRN_1055897.grp
Processing TRN_1042176.grp v.s. 2x2_898127.grp
Processing TRN_1042176.grp v.s. 2x2_920416.grp
Processing TRN_1042176.grp v.s. TRN_1042176.grp
Processing TRN_1042176.grp v.s. TRN_1055897.grp
Processing TRN_1055897.grp v.s. 2x2_898127.grp
Processing TRN_1055897.grp v.s. 2x2_920416.grp
Processing TRN_1055897.grp v.s. TRN_1042176.grp
Processing TRN_1055897.grp v.s. TRN_1055897.grp
```

Once the program finishes processing the input files, it creates two $n \times n$ distance matrices computed using laplacian and adjacency eigenvalues. These distance matrices have the same format, which starts with the list of model files, and writes in each row the query name and its distance to each of the models. One sample distance matrix is shown below.

```
2x2_898127.grp
2x2_920416.grp
TRN_1042176.grp
TRN_1055897.grp

2x2_898127.grp: 1.000000 0.125000 0.500000 0.500000
2x2_920416.grp: 0.125000 1.000000 0.500000 0.500000
TRN_1042176.grp: 0.500000 0.500000 1.000000 0.250000
TRN_1055897.grp: 0.500000 0.500000 0.250000 1.000000
```

To properly compute different types of recognition rates, e.g., nearest neighbor, first and second tier, etc., the distance between one query and itself is written as 1.0 on a scale between 0 and 1. This makes us ignore each match between a query and itself.

3 Deviations from plan

Even though deliverable D7.1 was produced later than planned (swapped with deliverable D6.1), namely project month M21, deliverable D7.2 is produced as originally planned (M24).