ATRテクニカルレポート表紙

［公開］

TR-M-0037

Movie Graph GUI

グレゴリ　モリ　　鈴木　良太郎

Gregory MORI　　Ryotaro SUZUKI

1998. 8.14

ATR 知能映像通信研究所

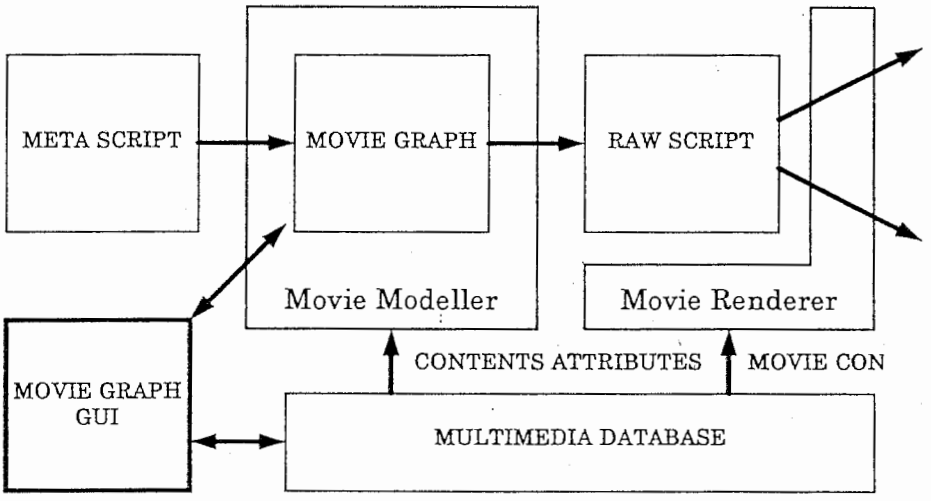
Movie Graph GUI

Gregory Mori, Ryotaro Suzuki
ATR Media Integration & Communications Research Laboratories

Introduction

The work that was done here at ATR involves a system for representing multimedia structure, and construction of multimedia compositions. The basis for this work is the Movie Graph data structure; it allows for the representation of a multimedia structure in a visual format that is easy to understand. The sub-systems that were worked on include the Movie Graph GUI, a program that allows creation and editing of Movie Graph structures in a visual way, the Component Database, which is a database for storing components that are used as data points within Movie Graphs, and finally the Movie Graph Distance, a way of measuring the difference between two Movie Graphs. Figure 1 shows how the system fits together.

Figure 1



Movie Graph GUI

Introduction

A Movie Graph is a representation of the structure of a multimedia composition. It allows simultaneous presentation of both the components in the composition and the timing layout of these pieces. The Movie Graph GUI is designed to allow simple editing of this structure. See paper for details on the structure...

Functions of the Interface

The GUI is a modal interface. The current mode is displayed at the bottom of the canvas holding the movie graph. In order to change the current mode, the user must click on one of the buttons from the right hand side of the GUI. The possible modes are adding mode, deleting mode, properties view mode, editing mode, reference mode, and content viewing mode. In addition to the mode changing buttons, there are buttons to perform saving and retrieval of movie graphs. It is not possible to open a movie graph, or create a new one, unless the previous graph is closed.

Adding mode

In adding mode, the user can click on nodes currently in the canvas in order to add new nodes to the graph. There are two *choice* selection boxes that affect the manner in which nodes are added to the existing graph. One box determines what type of movie graph node is added. The user can choose from among Movie, Define, Scene, Shot, and Symbol node types. The second box determines where the new node will be added relative to the node in the canvas that is clicked. If sibling is selected, the new node will be added on the same level as the node clicked in the canvas. If child is selected, the new node will be added as the first child of the clicked node.

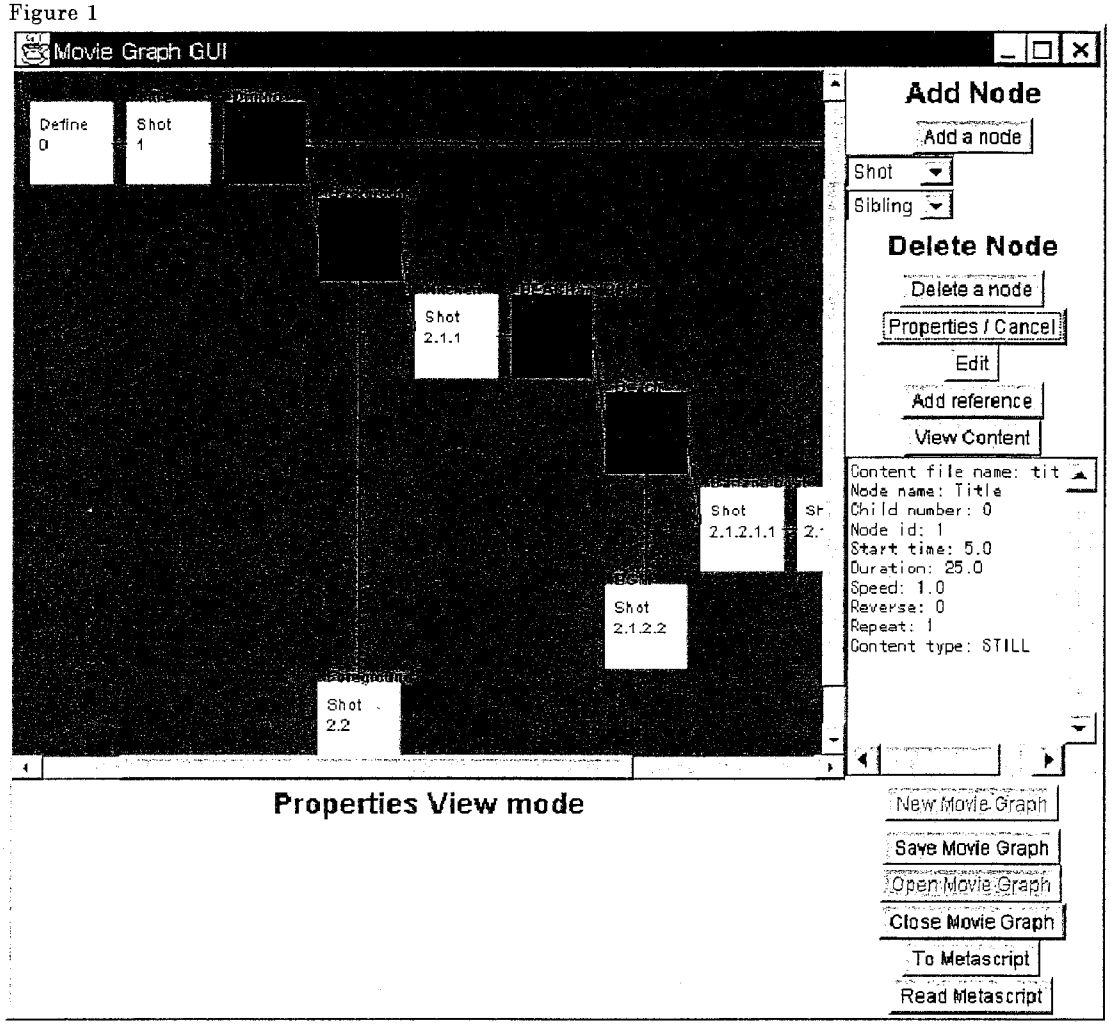
Deleting mode

When in deleting mode, if the user clicks on a node in the canvas it will be removed, along with all of its descendants.

Properties view mode

Properties view mode is used to display the numerical properties held in each node. When a node in the canvas is clicked, its numerical properties are displayed in the text area on the right hand side of the GUI. If the selected node references another node in the graph, that node is highlighted in yellow.

Figure 1



Editing mode

This mode is used to edit the aforementioned numerical properties of a node. When a node in the canvas is clicked, a dialog box containing fields for entering the numerical values appears. The type of dialog box that appears is dependent upon the type of node that is being edited, since different classes of nodes have different properties.

Reference mode

In reference mode, a reference to a previously defined node can be added. The GUI will first ask for the node to create the reference from, then the node to be referenced. The user can cancel the addition of the reference by clicking on the *properties/cancel* button.

Content viewing mode

This mode allows the user to see the content file of a shot node. A viewer based on Java Media Framework (JMF) is launched, and the content file is played.

Storage of graphs

The current graph can be stored persistently (ObjectStore for Java is used) by selecting the save button. The graph is saved under the name of the root movie node. A new graph can be created, or an existing one opened for editing, only if no graph is currently in the canvas. Select the close button to clear the canvas.

Metascript operations

A metascript representation of the current graph can be created by selecting the *to metascript* button. A dialog box asking for the file to print to will appear on the screen. In addition, one can read in a movie graph structure from an existing metascript file. This can be performed if there is no graph currently on the canvas.

Description of Important Source Files

BufferedMetascriptReader.java

This class is an extension of *BufferedReader* that is used when reading lines from metascript files. The reader skips lines that are denoted as comments (starting with '#'), and those that are blank.

DrawableNode.java

A DrawableNode holds all of the attributes of a node within a movie graph structure. It also specifies how it is to be drawn, and the position at which it should be laid out on the canvas. The DrawableNode also specifies how its attributes should be written to and read from a metascript.

DrawableDefineNode.java, DrawableSceneNode.java, DrawableShotNode.java, and DrawableSymbolNode.java

These are all descendants of DrawableNode, each specifying the attributes particular to a certain type of node.

DrawableMovieGraph.java

This holds a complete movie graph, stored as a collection of DrawableNodes. Various utility functions are also included, to allow addition and deletion of nodes, to find maximum X and Y positions of nodes in the graph, and to find the node at a given position on the canvas.

MovieGraphDisplay.java

This is the executable class. It sets up the GUI interface, adding appropriate buttons and a canvas for drawing the graph. All events in the canvas are interpreted, and the appropriate methods are called on the movie graph being edited.

NodeEditWindow.java

This class specifies how a particular node's properties are edited. A different panel of text fields for entering properties is displayed depending upon the type of node being edited.

How to compile

The source code is located in the directory ms74:/home/mori/MOVIEGRAPH-FINAL. Run the script build_it to create the JAR file moviegraph.jar.

How to install

Required elements:

- moviegraph.jar binary file, an archive of all the Java classes necessary to run the movie graph GUI
- Java interpreter, version 1.1.x
- Objectstore PSE version 1.2

The Java interpreter and Objectstore for windows95 can be found on mpc49 in "c:¥compressed software¥". Pseinst120.exe and jdk115-win32.exe are the files.

Place the moviegraph.jar file, and the necessary jar files from Objectstore in your CLASSPATH environment variable. Moviegraph.jar must be referenced directly, eg. */usr/xxx/moviegraph.jar*.

To run the Movie Graph GUI, type *java MovieGraphDisplay [database name].odb*

Component Database

Introduction

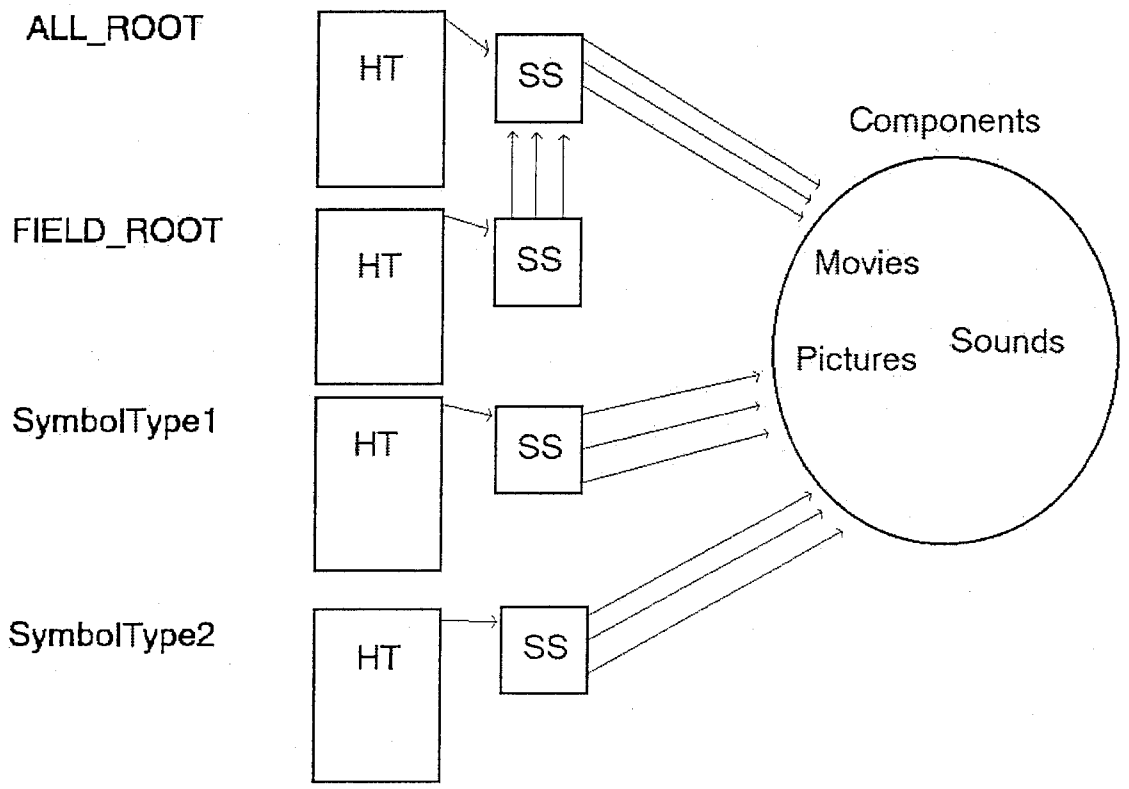
The multimedia component database allows for the storage of different types of multimedia-related components in an object oriented database. These components may be retrieved by means of queries of component type, component properties and their values, and by the presence of symbols. Symbols are attached to components in the database, and can be used to represent properties that may or may not be present, or may be present in differing cardinality, within the same class of component. One example is emotion data; in this model, one component may have zero or more emotions associated with it. Symbols can also be used to represent actors (or objects) within a particular component. Two components of differing class may each have a symbol of the same value. This could be the case with a movie clip and a sound clip that both express the emotion happiness. A query to find components expressing happiness would return both components.

Structure of the Database

In order to facilitate the types of query mentioned above, a variety of roots into the object-oriented database area maintained. There are two roots that point to hashtables, one hashtable is indexed by component class types (to allow queries by component type), and the other is indexed by field names of components. For each type of component in the database, one entry is made in the component class type hashtable. This entry is a SimpleSet containing references to all components in the database that are of this class. For every field name used by a component class type in the database, there is an entry in the field name hashtable. This entry is a SimpleSet of references to the aforementioned SimpleSets of components in classes using this field name. Note that the same field name may be repeated by different component class types.

In addition to the two static roots, there is a root for each type of symbol (such as ActorSymbol or EmotionSymbol) that is used by a component in the database. These roots point to hashtables that have one entry for each different value that a symbol of that type takes in the database. Each of these entries is a SimpleSet that holds all components in the database that have attached to them a symbol of that type taking that value. See figure 1 for a picture of the database.

Figure 1



Location of Files

The files for the component database are located on ms74, in the directory
/home/mori/jdk1.1.5/classes/componentDatabase
A jar archive of the files is located at
/home/mori/jdk1.1.5/classes/database.jar

Description of Source Files

Symbol.java:

A Symbol is an abstract class; subclasses of Symbol are used to represent specific types of optional properties that may occur in components in the database.

ActorSymbol.java:

An ActorSymbol is a Symbol that represents a role in a component, and the actor (i.e. person or object) cast into that role. Both the name of the object and the casting are stored as strings within the ActorSymbol.

EmotionSymbol.java:

An EmotionSymbol is a Symbol that represents an emotion that is conveyed by a component in the database. It has only one field, the emotion name, as a string.

ComponentDatabase.java:

This object represents a component database. A ComponentDatabase may be created from a filename that contains an existing database, or a new database may be created by passing an empty string to the constructor. Once created, a number of operations may be carried out, including insertion and deletion of components, modification of the symbols of existing components, printing the contents of the database, and searching for components by the methods mentioned in the introduction. Refer to the comments in the code for details on exactly how the structure of the database is maintained for insertion and deletion operations.

DatabaseComponent.java:

A DatabaseComponent is the abstract class of objects that may be stored in a ComponentDatabase. Classes that are to be stored in the database must implement the abstract methods outlined by this class. In particular, the addSymbol and removeSymbol methods are listed here so that the roots of the database may be properly updated when symbols are changed. Updating symbols is done by calling methods in the ComponentDatabase class, which in turn calls the protected addSymbol and removeSymbol methods.

There is also the facility to provide a GUI display for editing the contents of the component. This editor can be used by a GUI system to provide for the editing of different kinds of components. Sample DatabaseComponentDisplay classes are given in /home/mori/jdk1.1.5/classes/databaseGUI.

PictureDatabaseComponent.java:

An object that is used to store all of the properties associated with a still picture, such as the title, an explanation, and other properties.

PictureSize.java:

Used to represent the dimensions of a PictureDatabaseComponent.

ShotDatabaseComponent.java:

A representation of a multimedia shot. It could be a movie, a picture, or a sound. A generic representation of the possible properties of a shot.

SimpleSet.java and **SimpleSetItem.java**:

A SimpleSet is used to represent a set of SimpleSetItems, used so that the java objects may be made persistent.

Usage of Database

Objectstore PSE1.2 and java 1.1.x are required to run the component database, identical to that specified in the MovieGraphGUI documentation.

A component database can be used by means of the simple GUI contained in

/home/mori/jdk1.1.5/classes/databaseGUI

A driver program to run that GUI is located at

/home/mori/jdk1.1.5/classes/drivers/RunGUI.class

The usage is *java RunGUI*

A battery of drivers is located in the aforementioned directory, and two other useful drivers, one for printing a listing of all the components in the database (PrintContents) and one for printing a debug listing that is more representative of the structure of the database since all roots are included (PrintDatabase) are stored there. The usage is the same for both of them, *java Printxxx databasename.odb*.

Additional features may be added to the database by adding methods to ComponentDatabase.java, then adding driver programs, or GUI access, to run the methods. After modifying ComponentDatabase.java, run the script *build_it*, which is located in /home/mori/jdk1.1.5/classes. The script *build_it* will overwrite the jar archive with a new one.

Movie Graph Distance

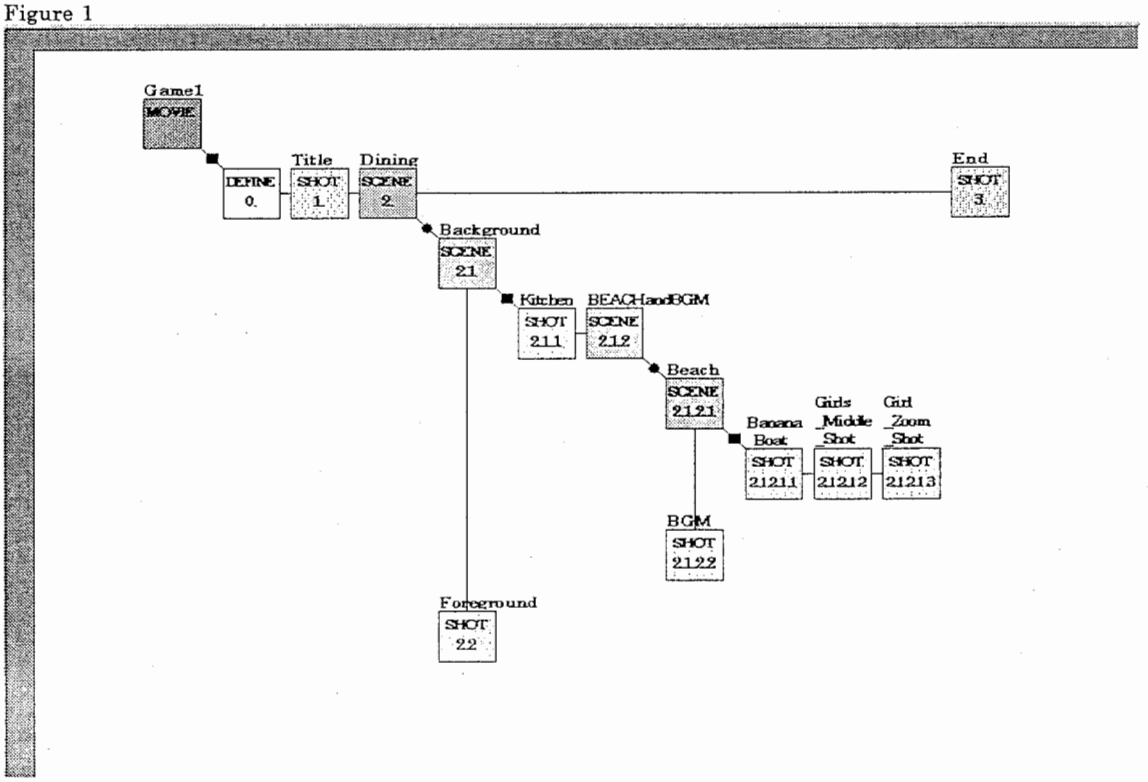
Introduction

After constructing Movie Graphs using the Movie Graph GUI, or by other means of creating metascript files that describe Movie Graphs, we would like to be able to measure the amount of similarity between Movie Graph structures. It is difficult to define what is meant by similarity; there are many ways to compare two graph structures. The method chosen has some degree of flexibility in it, some definitions may be changed to allow for different meanings of similarity. For example, the distance measuring can be tuned to look for sub-graphs, returning a high degree of similarity when one of the graphs analyzed is a sub-graph of the other.

Description of Method

The method for comparing Movie Graphs is based upon the notion of an axis within a graph. An axis is defined as the set of children of a particular node. In the case of Movie Graphs, there is an inherent ordering to the nodes in an axis, but this ordering is not a necessity for graphs to be measured under this distance function.

Figure 1



In the sample Movie Graph (Figure 1), examples of axes include the set {Define, Title, Dining, End}, the set {Banana Boat, Girls_Middle_Shot, Girl_Zoom_Shot}, and {Background, Foreground}.

Given two Movie Graphs (or any two graphs in general), and a distance function that compares two axes, the Movie Graph Distance is defined to be a matching or an assignment between the axes of the two graphs that minimizes the sum (may be a weighted sum of some variety) of the distances between all of the pairings of axes. In practice, the two graphs being compared do not necessarily have the same number of axes, so the distance function must be able to compare an axis to a null axis.

In the simple case of a straight sum of the distances between axes, we end up with an Assignment Problem to solve. There are two sets of *n* axes (fill the smaller set of axes with null axes to pad to the size of the larger), and *n^2* costs (differences) in making assignments between an axis in the first set to an axis in the second set. Each axis in the first set must be assigned to exactly one axis in the second. This version of the Movie Graph Distance is in fact a metric, providing that the distance function used to compare two axes is also a metric.

$$\text{SimpleMGD} = \underset{\sigma \in S(n)}{MIN}\left(\sum_{i=1}^{n} d\left(A_i, B_{\sigma(i)}\right)\right)$$

d is the distance function between two axes

A_i = i^{th} axis of graph A

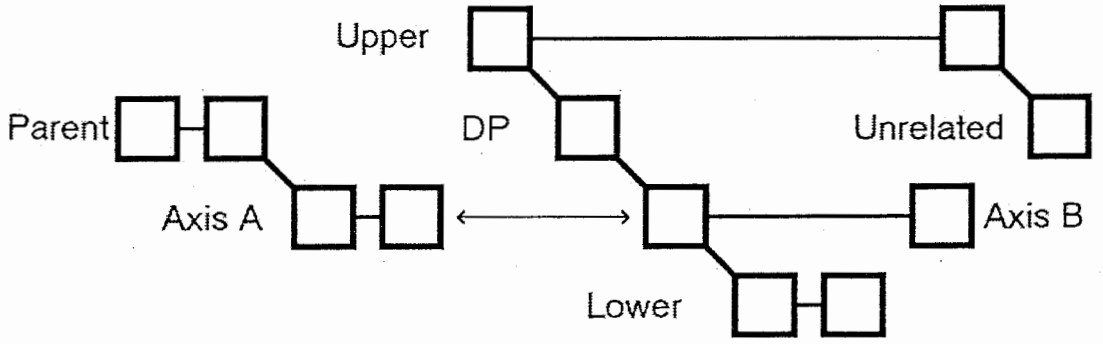
B_i = i^{th} axis of graph B

n is the number of axes in the larger (in terms of number of axes) of the two graphs

In the second version of the distance, we try to take into account parent-child relationships amongst axes. When making an assignment of one axis a1 (make real a-sub-1) in graph A to another axis (b-sub-1) in graph B, we consider the assignment of the parent of axis asub1. This pair of assignments is considered best if the parent of a-sub-1 was assigned to the parent of b-sub-1. Penalties to this assignment (reflected by an increase in cost) are imposed if the parent has been assigned somewhere else. The specific cases implemented by the second version of the Movie Graph Distance are (in increasing order of imposed penalty) direct parent, upper (the parent of asub1 was assigned to some axis that is an ancestor of b-sub1, but not its parent), unrelated (not an ancestor, but not a descendant either), and lower (a descendant of bsub1). In

Figure 2, when Axis A is matched with Axis B, the labels on the axes of the graph on the right specify the parent-matching case that occurs if Parent is matched to each of the other axes in the right-side graph.

Figure 2



By adding in this additional consideration of the structure of the axes within the graph, we end up with a distance that reflects the structural differences more emphatically. It is possible to measure these sorts of structural differences exclusively by defining the distance function to be identically equal to some constant, thereby ignoring the actual contents of the axes, and focusing entirely on the parent-child relationships.

This second version of the distance is represented in a linear programming type of problem; there is a function to minimize under a set of constraints.

give the equation of the function to minimize, plus some sample constraints

Function to minimize:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{P\text{-}C\_CASES} d(A_i, B_j) * A(i,j,k)$$

Subject to the constraints:

$$\forall i \in \{1..n\}, \sum_{j=1}^{n}\sum_{k=1}^{P\text{-}C\_CASES} A(i,j,k) = 1$$

$$\forall j \in \{1..n\}, \sum_{i=1}^{n}\sum_{k=1}^{P\text{-}C\_CASES} A(i,j,k) = 1$$

$A(i,j,k)$ is an entry in a 3-d matrix representing the assignment between axis i in graph A and axis j in graph B, subject to parent-child condition k

These two sets of constraints specify that each axis in graph A should only be assigned to one and only one axis in graph B.

There are also $4n^2$ equations similar to:

$$\forall i,j \in \{1..n\} \quad \sum_{k=1}^{P\_C\_CASES} A(p,j,k) + \sum_{n \in children(A_i)} A(i,n,1) \le 1$$

The axis numbered p in graph A is the parent of the axis numbered i. There are 4 different types of equations like this (one for each parent-child case). Each of these equations uses n^2 different combinations of i and j.

These equations specify that the parent-child case must be adhered to (i.e. the "upper" case must only be used if the parent of the axis in graph A really is assigned to an axis that "upper" of the axis in graph B).

The Distance Function Used

The distance function used is:

$$d(x,y) = \begin{cases} \left|\#repeated(x) - \#repeated(y)\right|^3, & TYPE(x) = symbol \wedge TYPE(y) = symbol \\ IMPORTANCE(x) + IMPORTANCE(y), & x = null \vee y = null \vee \left(TYPE(x) \neq TYPE(y)\right) \\ TF * ND + \left(TF - 1\right)^3, & TYPE(x) = shot \wedge TYPE(y) = shot \end{cases}$$

The "type" of an axis can be either null (*null*), all symbol (*symbol*), or a mixture of shot and symbol (*shot*).

$$IMPORTANCE(x) = IMP\_FACTOR * \left(\left(REPEAT\_CONSTANT * \#repeated(x)\right)^3 + PARALLEL\_IMP\_FACT * SMOOTH\_IT(depth(x)) * \left(SHOT\_CONST * \#shots(x) + SCENE\_CONST\right.\right.$$

$$TF = 1 + ALIGN\_FACTOR * \left|ALIGN(x) - ALIGN(y)\right|$$

The alignment of an axis is either 1 or 0, representing sequential vs. parallel alignment of shots, one of the most important distinguishing features of a movie graph.

$$ND = SCENE\_FACTOR * \left|\#scenes(x) - \#scenes(y)\right| + LENGTH\_FACTOR * \left|length(x) - length(y)\right|$$

The distance function used may be easily changed by modifying the function computeDifference. If only a change in the values of constants used is required, the constants text file may be edited.

How to Interpret the Output

The output that is returned by the Movie Graph Distance consists of a number representing the distance (minimum assignment), and the flow (or assignment between axes) that accomplishes that minimum distance. A single distance number by itself has little meaning; it is the comparison of differences between one input movie graph, and a battery of known movie graphs that holds more meaning. By looking at the rankings of the known graphs that are closest, or the relative differences between them, one can obtain an understanding of the nature of the structure that is being analyzed.

Software

How to use the system

The main program is called MovieGraphMetric, and requires four parameters – the names of two metascript files that represent the two Movie Graphs to be compared, and the names of two files for the output of the program, the first for the simple assignment problem, the second for the linear programming problem. There is a fifth, optional parameter, the name of a file from which to load constants. This file contains entries on a row by row basis, parameter name, and value. An example parameter file is given at:

c:¥Program Files¥DevStudio¥MyProjects¥metric¥constants.txt.

Files produced:

After running the program, two output files are created. One contains the data for the assignment problem, stored in DIMACS format for graph problems. The second is a linear programming problem, a system of equations, and an objective function to be optimized. Both files contain a comment at the top that gives a listing of the axes of the two graphs being compared, a key to reference the axes by number in the problem notation.

Once these problem files have been created, a solver must be run to produce an answer to each problem. These solvers are located on ms18.

For the standard assignment problem:

The software for the solver is located in ms18:/home/mori/CSAS/CSAS/

Run the program csa/prec_costs/csa_s to solve an assignment problem. This program reads from the standard input stream, so redirect input to come from the assignment

problem file created my MovieGraphMetric. Summary information on the result will be printed to the standard output stream. The file output.flow is also created; it contains more detailed information -- the exact assignments between axes which were made.

For the linear programming problem:

The software for the solver is located in ms18:/home/mori/solvers/lp_solve/lp_solve_2.0/ Run the program lp_solve to obtain a solution from one of the linear programming problem files created by MovieGraphMetric. Input should be redirected to read from the problem file in question, and output should be redirected to a file as well. The solution will be very long in most cases. The first line of the output will have the distance on it. The following lines will have linear programming variable names, and their values when the optimal solution (the distance) is obtained. The variables of note are those with value 1. Those with value 1 represent an assignment, those of value 0 represent no assignment. Use the UNIX command *grep* to find the lines with single 1s on them. Variables are named x*n*y*m*z*p*. This represents an assignment between axis *n* of graph 1, and axis *m* of graph 2. The z*p* portion refers to which of the parent assignment cases occurred.

Samples

The sample constant file subgraph-constants.txt contains a set of constants that can be used in detecting sub-graphs. The major thing that is changed by this constant file is setting the *importance* of any axis to 0. When an axis is compared against a null axis, the *importance* of the non-null axis is the difference. By setting all of these values to zero, we tend to look for a sub-graph. If graph A is a sub-graph of graph B, the Movie Graph Distance between the 2 graphs will be zero. If A is very close to being a sub-graph of B, a distance slightly higher will be returned. A set of sample data is kept in ms18:/home/mori/movie-graph-distance/samples/subgraph

Two useful script files are also kept in sub-directories there, asn/solve_asn and lp/solve_lp. These scripts take an *.asn* and *.lp* file as a parameter respectively, and redirect the output and interesting information to a pair of output files.

Location of Files

The files for MovieGraphMetric are stored on mpc49, in:

c:¥Program Files¥DevStudio¥MyProjects¥metric¥

There is a Developer Studio workspace file, metric.dsw in that directory.