The VFLib Graph Matching Library, version 2.0

Pasquale Foggia `foggiapa@unina.it` March 2001

This document describes the **VFLib** graph matching library, developed at the *Intelligent Systems and Artificial Vision Lab.* (*SIVALab*) of the University of Naples "Federico II".

Contents

1 Introduction

This document describes the **VFLib** graph matching library, developed at the *Intelligent Systems and Artificial Vision Lab.* (*SIVALab*) of the University of Naples "Federico II".

The library has been developed mainly to test a new graph matching algorithm, named **VF**, and to compare it with other known algorithms, like Ullmann's algorithm and the algorithm by Schmidt and Druffel (see section 5 (References)).

By no means VFLib attempts at providing a general graph library; it addresses only the problems of exact graph isomorphism and graph-subgraph isomorphism. If you need to solve other graph problems, you should look for a different tool, such as the *LEDA Library* `<http://www.algorithmic-solutions.com>`.

The root of VFLib is a graph matching program, named `grapp`, which was built in 1997 to compare the VF graph matching algorithm with Ullmann's. The `grapp` program sources (written in C++) have been made available on the World Wide Web since June 1997, and several researchers in different fields have downloaded it for their research activity.

The `grapp` program was limited to only graph isomorphism, though the paper describing the VF algorithm also presented its application to graph-subgraph isomorphism; another limitation was that the program did not make provisions for associating semantic attributes to the nodes and the edges of the graphs.

Successively, the code was re-organized as a separate library (for which the VFLib name was coined) to be linked with a main program, in order to make easy the reuse the matching code in different applications. At the same time, graph-subgraph isomorphism was added (both by means of VF and of Ullmann's algorithm) and a limited support for graph monomorphism was also introduced. Moreover, the data representation was modified to deal with node and edge attributes, and accordingly the algorithm implementations were changed to take advantage of this semantic information when available.

This version of VFLib (called 1.0) was never released to the public (mostly because it was completely undocumented), but it has been used extensively for other projects in our Lab, including the development of a Machine Learning system named **Gengis** (see section 5 (References)).

In 2000, in order to make the library work more efficiently on large graphs, the data structures used for representing the graphs have been changed. We have also modified for this purpose the VF algorithm, defining an improved version named VF2, which lowers the memory requirements from O(n^2) to O(n) with respect to the number of nodes in the graphs. Also, after the suggestion of a reviewer of a paper of ours describing VF, we have added support for the algorithm by Schmidt and Druffel.

These changes led to version 2.0 of VFLib, which we have made now available on the Internet. The documentation you are reading refers to VFLib 2.0; you may check the *SIVALab graph matching page*

`<http://amalfi.dis.unina.it/graph>` for further updates of this library. On that page you will also find the *Graph Database*, a huge collection of graphs aimed at becoming a common data set for benchmarking graph matching algorithms.

1.1 Copyright and distribution of the library

The VFLib distribution is

1.2 Contacting the authors

In addition to the Web page cited above (*http://amalfi.dis.unina.it/graph*), you can contact the authors of the library using the following address:

Prof. Mario Vento
Dip. di Informatica e Sistemistica
Univ. di Napoli "Federico II"
Via Claudio, 21
I-80125 Napoli ITALY

2 Installing VFLib

The VFLib library is distributed in source format, packed and compressed in a `.tar.gz` or in a `.zip` format file. This means that the first step for installation is to unpack the archive, but probably you have already done this, if you are reading this documentation!

Now, you have to compile the library, which requires a C++ compiler. The library has been developed under the Linux operating system, using the `egcs` compiler. However, the code does make use only of standard C++ features, so it should work on other operating systems with any recent compiler.

For Unix systems like Linux we have provided a Makefile that automates the compilation process. You need to edit the file `Makefile` in order to set the name of your compiler and the relevant compile options (the file provided assumes that the compiler is invoked as `g++` and enables full code optimization); after that you only have to issue the command `make` to build the library.

This will produce a library file named `libvf.a` in the `lib` subdirectory. You will need to link this library to your programs for using the graph matching library. You must also remember, when compiling C++ files that use the library, to add the `include` directory, created when unpacking the library archive, to the compiler include path.

For example, suppose you have a file named `my_prog.cc` which makes use of the library, and suppose that you have unpacked and built VFLib under the directory `/usr/local/vflib/`. Then the command to produce an object file `my_prog.o` would be:

```
        g++ -c -I/usr/local/vflib/include my_prog.cc
```

where the `-c` flag tells the compiler to stop after producing the object file, and the `-I` flag adds the specified directory to the include path.

The command to produce an executable would be:

```
        g++ -o my_prog my_prog.o -L/usr/local/vflib/lib -lvf -lstdc++ -lm
```

where the `-o` flag specifies the name of the executable (the default would be `a.out`), the `-L` flag adds the `lib` directory to the library search path, and the three `-l` flags indicate the libraries to link to the program (`-lvf` represents the VFLib library file).

In section 3 (Using VFLib: a quick tour) you will find some example programs that make use the library.

If you use a non-Unix system, please refer to your compiler documentation (or ask a local guru) for building the library and linking it to your programs. Notice that on some compilers it may be necessary to change the source filenames extension from `.cc` to `.cpp` to compile them in C++ mode (as opposed to C).

3 Using VFLib: a quick tour

VFLib deals with *simple directed graphs*, i.e. the graphs cannot contain self-loops, edge *(i,j)* is considered different from edge *(j,i)*, and between two nodes there is at most one edge.

Nodes are identified through values of type `node_id`, defined in the header `argraph.h`, which corresponds to unsigned 16-bit integers. The first node has an id equal to 0. The value `NULL_NODE` (corresponding to 2^16 - 1, i.e. 65535) is used as a null node id, so the valid ids are from 0 to 65534, and the graph can contain at most 65535 nodes. Edges are identified through a pair of node ids.

The library has been expressly designed to perform only one operation, i.e. graph matching, and all the classes and the data structures have been chosen in such a way to make this operation fast, especially on large graphs.

Unfortunately, this means that the library interface can be a little bit cumbersome with respect to some tasks which are instead quite simple with other libraries.

In the following you will find a short description, together with code fragments,

Class	Header file	Relation	Algorithm
`SDState`	`sd_state.h`	isomorphism	Schmidt-Druffel
`UllState`	`ull_state.h`	isomorphism	Ullmann
`VFState`	`vf_state.h`	isomorphism	VF
`VF2State`	`vf2_state.h`	isomorphism	VF2
`UllSubState`	`ull_sub_state.h`	graph-subgr. isom.	Ullmann
`VFSubState`	`vf_sub_state.h`	graph-subgr. isom.	VF
`VF2SubState`	`vf2_sub_state.h`	graph-subgr. isom.	VF2
`VFMonoState`	`vf_mono_state.h`	monomorphism	VF
`VF2MonoState`	`vf2_mono_state.h`	monomorphism	VF2

Table 1: Matching algorithms

```
// Insert the four nodes
for(i=0; i<4; i++)
  ed.InsertNode(NULL); // The inserted node will have index i.
                       // NULL stands for no semantic attribute.

// Insert the edges
for(i=0; i<4; i++)
  for(j=0; j<4; j++)
    if (i!=j)
          ed.InsertEdge(i, j, NULL); // NULL stands for no sem. attribute.


// Now the Graph can be constructed...
Graph g(&ed);

// To be continued...
```

In a later subsection, some other `ARGLoader` implementations will be presented which can be used to load a graph from a file.

3.2 Choosing the matching algorithm

In order to perform the matching, you have to choose which kind of relation you are interested in (e.g. isomorphism), and which algorithm you want to use. To make this choice, you need to create an instance of a class derived from the `State` class, which represents the starting point in the search space of the algorithm. Table 1 (Matching algorithms) presents the available choices.

Once you have chosen the right class, you need to create an instance of it, passing to the constructor a pointer to the two graphs being matched. If the matching relation is graph-subgraph isomorphism or monomorphism, the role of the two graph is not symmetric. In this case, the first constructor parameter must be the smallest of the two graphs.

For example, suppose you need to perform a graph-subgraph isomorphism, and decide to use the VF2 algorithm. The code needed to initialize the search state is:

```
#include "argraph.h"
#include "argedit.h"
#include "vf2_sub_state.h"

int main()
  { ARGEdit small_ed, large_ed;
    //... some code here should construct the graphs ...
        Graph small_graph(&small_ed), large_graph(&large_ed);

    // Create the initial state of the search space
        VF2SubState s0(&small_graph, &large_graph);

    //... to be continued ...
```

3.3 Finding the first matching between two graphs

In order to start the matching process, you need to call the `match` function declared in `match.h` . Actually there are two overloaded `match` functions; the first stops after a matching is found, while the other iterates over all the possible different matchings between the two graphs.

If you are interested only in the first matching found, you need to declare two arrays of `node_id` to store the pairs of corresponding nodes; the dimension of each array must be at least equal to the number of nodes in the smallest of the two graphs.

The function to be used has one input parameter, i.e. a pointer to the initial state, described in the previous subsection. The function returns a `bool` value which is `false` if no matching has been found, and has three output parameters: the number of matched nodes (which for the matching relations implemented up to now is always equal to the number of nodes of the first graph), and the two arrays of matched node ids. Corresponding elements of the two arrays contain the ids of nodes paired by the matching algorithm.

For example, the following code would invoke the matcher and print the result:

```
#include <argraph.h>
#include <match.h>

#define MAXNODES 200

int main()
{
//... here the two graphs and the inistial state
//    should be created ...

  int n;
  node_id ni1[MAXNODES], ni2[MAXNODES];

  if (!match(&s0, &n, ni1, ni2))
    { printf("No matching found!\n");
      return 1;
    }

  printf("Found a matching with %d nodes:\n", n);
  int i;
```

```
    for(i=0; i<n; i++)
      printf("\tNode %hd of graph 1 is paired with node %hd of graph 2\n",
                 ni1[i], ni2[i]);
    return 0;
  }
```

3.4 Finding all the matchings between two graphs

In case you need to examine all the matching between two graphs, you have to use a different version of the `match` function.

This version needs a *match visitor*, that is a callback function which is called everytime a matching is found. The match visitor has four parameters: the number of nodes in the matching, the two arrays of `node_id` that represent the nodes paired by the algorithm, and an user-provided void pointer which can be used to pass some other useful information to the visitor (for example, a file where the matchings should be stored).

The visitor, after processing the current matching, must return a `bool` value: if the value is `false`, the next matching is searched for; else, the search stops.

The `match` function takes as input parameters a pointer to the initial state, a function pointer to the match visitor, and an optional `void *` (which defaults to `NULL`) that will be passed to the visitor. The return value of the `match` function is the number of examined matchings.

As an example, suppose that you want to save all the matchings on a text file. The code needed to perform this task is:

```
#include <argraph.h>
#include <match.h>

bool my_visitor(int n, node_id ni1[], node_id ni2[], void *usr_data)
  { FILE *f = (FILE *)usr_data;

    // Prints the matched pairs on the file
        int i;
        for(i=0; i<n; i++)
          fprintf(f, "(%hd, %hd) ", ni1[i], ni2[i]);
        fprintf(f, "\n");

        // Return false to search for the next matching
        return false;
  }

int main()
  { // ... here goes the code to build the graphs
    // and to create the initial search state 's0'...

    // Create the output file
        f=fopen("output.txt", "w");

        match(&s0, my_visitor, f);

        fclose(f);
        return 0;
    }
```

3.5 Adding and using node/edge attributes

Semantic attributes are handled by the Graph class through `void *` pointers, and so any built-in or user defined type can be used for node or edge attributes.

The use of pointers has the advantage of making possible to manage the attributes efficiently, avoiding unnecessary copies, and allowing different graphs to share the memory areas used for the attributes. On the other hand, especially in a language like C++ that does not support automatic garbage collection, there is the problem of the ownership of the pointed data structure: who is responsible for allocating the attribute, for cloning it (i.e. allocating a new copy of it) when necessary, and for deallocating it?

The easiest answer for the user would be: the Graph class. But this choice would have implied excessive limitations to the flexibility of the attribute management.

For example, in many situations it would be reasonable to have the attributes dynamically allocated on the heap, with a separated attribute instance for each node/edge. However there are cases where only a very limited set of attribute values are possible, and so a great memory saving can be obtained by preallocating these values (possibly with static allocation) and sharing the pointers among all the nodes/edges of the graphs. Giving the Graph class the responsibility for the allocation/deallocation of the attributes would imply that only one allocation policy must be chosen, at the price of an efficiency loss in the cases where this policy is not the best one.

So, the rules we have followed in the design of the library are:

- `ARGLoader` objects are responsible for allocating the attributes when the graph data is generated or read; the user that writes his/her own `ARGLoader` is the most indicated person to decide which allocation policy is the best. `ARGLoader` objects should not be concerned with attribute clonation or deallocation; they simply pass the pointers to the `Graph` object that is built. Note that the `ARGEdit` class does not perform attribute allocation; it simply receives the attribute pointers from the caller.
- `Graph` objects by default do not deal with attribute allocation, clonation or deallocation. However, it is possible to register a node destroyer and an edge destroyer (i.e. an object that knows how to deallocate a node/edge attribute) within each graph.
- Everything else is upon the shoulders of the library user, which has to understand when an attribute must be cloned or deallocated, and to perform these operations iterating over the nodes and the edges. In particular, notice that when multiple graphs are built using the same `ARGLoader`, it is usually necessary to clone the node/edge attributes.

In order to avoid the need of pointer casting when dealing with attributes, the library provides a template class `ARGraph<N,E>`, derived from `Graph`, for graphs with node attributes of type `N` and edge attributes of type `E`.

How are attributes employed in the matching process? The user can provide an object of the graph class with a *node comparator* and an *edge comparator*. These are objects implementing the `AttrComparator` abstract class, which has a `compatible` method taking two attribute pointers, and returning a `bool` value that is `false` if the corresponding nodes or edges are to be considered incompatible and must not be paired in the matching process. In this way, the search space can be profitably pruned removing semantically undesirable matchings. Notice that the matching algorithm uses the comparators of the first of the two graphs used to construct the initial state.

Now let us turn to a practical example. Suppose that our nodes must represent points in a plane; we will associate with each node an attribute holding its cartesian coordinates. For simplicity, the edges will have no attributes. Suppose we have a class `Point` to represent the node attributes:

```
class Point
  { public:
      float x, y;
      Point(float x, float y)
        { this->x=x;
          this->y=y;
        }
  };
```

Now, if we want to allocate the attributes on heap, we need a destroyer class, which is an implementation of the abstract class `AttrDestroyer`. In our example, the destroyer could be:

```
class PointDestroyer: public AttrDestroyer
  { public:
       virtual void destroy(void *p)
         { delete p;
         }
  };
```

We will also need a comparator class for testing two points for compatibility during the matching process. A comparator is an implementation of the abstract class `AttrComparator`. Suppose that we consider two points to be compatible if their euclidean distance is less than a threshold:

```
class PointComparator: public AttrComparator
{ private:
    double threshold;

public:
  PointComparator(double thr)
    { threshold=thr;
    }
  virtual bool compatible(void *pa, void *pb)
    { Point *a = (Point *)pa;
      Point *b = (Point *)pb;
      double dist = hypot(a->x - b->x, a->y - b->y);
           // Function hypot is declared in <math.h>

      return dist < threshold;
    }
};
```

We can build two graphs with these attributes using the `ARGEdit` class:

```
int main()
  { ARGEdit ed1, ed2;
```

```
        ed1.InsertNode( new Point(10.0, 7.5) );
        ed1.InsertNode( new Point(2.7, -1.9) );
        ed1.InsertEdge(1, 0, NULL);
        // ... and so on ...

        ARGraph<Point, void> g1(&ed1);
        ARGraph<Point, void> g2(&ed2);


        // Install the attribute destroyers
        g1.SetNodeDestroyer(new PointDestroyer());
        g2.SetNodeDestroyer(new PointDestroyer());

        // Install the attribute comparator
        // This needs to be done only on graph g1.
        double my_threshold=0.1;
        g1.SetNodeComparator(new PointComparator(my_threshold));

        VFSubState s0(&g1, &g2);

        // Now matching can begin...
```

Notice that the attribute destroyers and comparators have to be allocated on heap with `new`; once they are installed they are owned by the graph, which will `delete` them when they are no longer needed. So it is an error to share a destroyer or a comparator across graphs, as is to use a static or automatic variable for this purpose.

Historical note: Previous versions of the library (before 2.0.5) used simple functions instead of full objects for deallocating or comparing attributes. These functions were installed using the `SetNodeDestroy`, `SetEdgeDestroy`, `SetNodeCompat` and `SetEdgeCompat` methods. While these methods are still supported for backward compatibility, we warmly recommend to use the new object-oriented approach, which provides greater flexibility. For example, with the old approach it would have been quite difficult to obtain something equivalent to a point comparator; the threshold would have had to be either a compile-time costant or a global variable, with obvious drawbacks.

3.6 Loading and saving graphs

In order to load and save graphs from files, we have provided two classes derived from `ARGLoader`. The first, `BinaryGraphLoader`, is devoted to unattributed graphs, and adopts the binary file format employed for the *Graph Database* `<http://amalfi.dis.unina.it/graph>`. The second, `StreamARGLoader`, is a template class which employs C++ input/output streams and overloaded stream operators to read and write graphs from/to text files. Both the classes are defined in `argloader.h` .

In section 4 (VFLib class reference) a detailed description of the file formats will be presented. Here we will show some examples of the use of these two classes.

Let us start with `BinaryGraphLoader`. The constructor of this class requires an `istream` object, which must be open using the `ios::binary` mode:

```
    #include <iostream>
```

```
#include <fstream>

using namespace std;

#include "argraph.h"
#include "argloader.h"


int main()
  { // First, open the file in binary mode
    ifstream in("graph.bin", ios::in | ios::binary);

    // Second, create a BinaryGraphLoader
    BinaryGraphLoader loader(in);

    // Now a graph can be constructed
    Graph g(&loader);

    // ... to be continued ...
```

A graph, represented either as an object of class `Graph`, or as an `ARGLoader`, can be saved in the same format using the static method `write`.

```
// ... suppose that a graph g has been constructed ...

// Open a binary file
ofstream out("outfile.bin", ios::out | ios::binary);

// Write the graph
GraphBinaryLoader::write(out, g);

// Close the file
out.close();

// ... to be continued ...
```

Remember that this class does not consider node/edge attributes!

Now, let us turn our attention to `StreamARGLoader<N,E>`. This is a template class for reading and writing graphs with node attributes of type `N` and edge attributes of type `E`.

As we have said previously, it is responsibility of the `ARGLoader` to allocate the data structures for storing the attributes. `StreamARGLoader` perform this task with the help of an `Allocator` instance. `Allocator` is an abstract template class which provide an `Allocate` method for creating a new attribute. The template class `NewAllocator` is an implementation of `Allocator` that uses the `new` operator for allocating the attribute. Both `Allocator` and `NewAllocator` are defined in `allocpool.h`.

Besides creating a suitable allocator, you need to override the operator >> `for reading graphs and the operator` << `for writing, if the types of your attributes do not already support stream IO`.

For example, let us reconsider the case in which the node attributes represent points in the plane, and there are no edge attributes.

First, you have to define the attribute classes, with the corresponding I/O operators. Notice that you will need an empty class for the edge attribute.

```
#include <iostream>
#include <fstream>

#include <argraph.h>
#include <argloader.h>
#include <allocpool.h>


//
// Define class Point, with its I/O operators
//
class Point
  { public:
      float x, y;
  };

istream& operator>>(istream& in, Point &p)
  { in >> p.x >> p.y;
    return in;
  }

ostream& operator<<(ostream& out, Point &p)
  { out << p.x << ' ' << p.y;
    return out;
  }

//
// Define class Empty, with its I/O operators
//
class Empty
  {
  };

istream& operator>>(istream& in, Empty &)
  { // Do nothing!
    return in;
  }

ostream& operator<<(ostream& out, Empty &)
  { // Do nothing!
    return out;
  }
```

Now, you have first to create the two allocators (one for nodes and one for edges), and then a `Stream-ARGLoader` object. Notice that for the edges we have used a `NullAllocator`, which does no allocation at all!

```
int main()
  { // Create the allocators
```

```
    NewAllocator<Point> node_allocator;
    NullAllocator<Empty> edge_allocator;

    // Open the file
    ifstream in("graph.txt");


    // Create the ARGLoader
    StreamARGLoader<Point, Empty> loader(&node_allocator,
                                         &edge_allocator,
                                         in);

    // Build the graph
    ARGraph<Point, Empty> graph(&loader);

    //......
```

Similarly to the `GraphBinaryLoader` class, also `StreamARGLoader` provides a static `write` method to write out a graph in the corresponding format:

```
    // ... suppose that a graph g has been constructed ...

    // Open a text file
    ofstream out("outfile.txt");

    // Write the graph
    StreamARGLoader::write(out, g);

    // Close the file
    out.close();

    // ... to be continued ...
```

4 VFLib class reference

In this section, a short description of the VFLib classes, and of their public methods.

4.1 Abstract class `ARGLoader`

Defined in: argraph.h

Extends: -

Overview: This class is used to construct a Graph / ARGraph object. Implementations of this class could, for example, load the graph data from a file. Two implementation strategies are possible:

- the ARGLoader loads or generates the graph data in memory when it is allocated.
- graph data are loaded/generated when the program ask for it calling one of the methods of ARGLoader. In this case, the implementor may rely on the fact that the information for the nodes

(method `GetNodeAttr`) is requested once per node, and sequentially; the information for the edges (methods `OutEdgeCount` and `GetOutEdge`) is requested after all the node information has been collected, and in a sequential order depending on the start node of the edges.

4.2 Class `Graph` / `ARGraph_impl`

Defined in: argraph.h

Extends: -

Overview: This class is used to hold the graph information needed during the matching process. This class is designed for a compromise between fast access to the graph structure and attributes, and low memory occupation (in order to deal with large graphs). The actual name of the class is `ARGraph_impl`; `Graph` is a `typedef` which is often used for brevity.

Notice that graphs represented through this class are *immutable*, that is they cannot be changed (except for node/edge attribute values).

4.3 Class `ARGraph<Node,Edge>`

Defined in: argraph.h

Extends: Graph

Overview: This is a template class that adds static type checking to the node/edge attribute pointers used in the Graph class as 'void *'.

4.4 Class `ARGEdit`

Defined in: argedit.h

Extends: ARGLoader

Overview: An ARGEdit is a simple implementation of an ARGLoader which stores the graph in memory allowing the basic graph editing operations. The data structure is not optimized for fast access, e

The file is composed by a sequence of 16-bit words; the words are encoded in little-endian format (e.g., LSB first). The first word represents the number of nodes in the graph. Then, for each node, there is a word encoding the number of edges coming out of that node, followed by a sequence of words encoding the endpoints of those edges.

An example, represented in hexadecimal, follows:

```
03 00     Number of nodes (3)
00 00     Number of edges out of node 0 (0)
02 00     Number of edges out of node 1 (2)
00 00     Target of the first edge of node 1 (edge 1 -> 0)
02 00     Target of the second edge of node 1 (edge 1 -> 2)
01 00     Number of edges out of node 2 (1)
00 00     Target of the first (and only) edge of node 2 (edge 2 -> 0)
```

4.6 Class `StreamARGLoader<Node,Edge>`

Defined in: argloader.h

Extends: ARGEdit

Overview: This class allows to read (or write) an attributed graph from a text stream. The class relies on the stream insertion/extraction operators (<`lt;` and >>) to perform the input/output operations on the attributes.

The class parameters (`Node` and `Edge`) are the types of the node/edge attributes, which are stored internally through pointers. The attributes can be heap allocated, or a more efficient allocation strategy can be implemented by means of an Allocator object.

File format

On the first line there must be the number of nodes; subsequent lines will contain the node attributes, one node per line, preceded by the node id; node ids must be in the range from 0 to the number of nodes - 1. Then, for each node there is the number of edges coming out of the node, followed by a line for each edge containing the ids of the edge ends and the edge attribute. Blank lines, and lines starting with #, are ignored. An example file, where both node and edge attributes are ints, could be the following:

```
# Number of nodes
3
# Node attributes
0 27
1 42
2 13

# Edges coming out of node 0
2
0 1  24
0 2  73

# Edges coming out of node 1
```

```
1
1 3  66

# Edges coming out of node 2
0
```

4.7 Abstract class `AttrDestroyer`

Defined in: argraph.h

Extends: -

Overview: Concrete implementations of this class are used to deallocate node or edge attributes when no longer needed. A destroyer is installed with the `SetNodeDestroyer` and `SetEdgeDestroyer` methods of class `Graph`.

Notice that the attribute destroyers have to be allocated on heap with `new`; once they are installed they are owned by the graph, which will `delete` them when they are no longer needed. So it is an error to share a destroyer across graphs, as is to use a static or automatic variable for this purpose.

4.8 Abstract class `AttrComparator`

Defined in: argraph.h

Extends: -

Overview: Concrete implementations of this class are used to compare two node or edge attributes in order to test for their compatibility during the matching process. A comparator is installed using the `SetNodeComparator` and `SetEdgeComparator` methods of class `Graph`.

Notice that the attribute comparators have to be allocated on heap with `new`; once they are installed they are owned by the graph, which will `delete` them when they are no longer needed. So it is an error to share a comparator across graphs, as is to use a static or automatic variable for this purpose.

4.9 Class `FunctionAttrDestroyer`

Defined in: argraph.h

Extends: `AttrDestroyer`

Overview: This class implements an `AttrDestroyer` that invokes a function passed at construction time. It is used to ease the transition from the old style (pre 2.0.5) deallocation functions and the new `AttrDestroyer` objects.

4.10 Class `FunctionAttrComparator`

Defined in: argraph.h

Extends: `AttrComparator`

Overview: This class implements an `AttrComparator` that invokes a function passed at construction time. It is used to ease the transition from the old style (pre 2.0.5) compatibility functions and the new `AttrComparator` objects.

4.11 Abstract class `Allocator<T>`

Defined in: allocpool.h

Extends: -

Overview: An Allocator is an object that knows how to allocate a node/edge attribute of type `T`. Subclasses can be used to implement different allocation strategies.

4.12 Class `NullAllocator<T>`

Defined in: allocpool.h

Extends: Allocator<T>

Overview: A dummy allocator that always returns NULL.

4.13 Class `NewAllocator<T>`

Defined in: allocpool.h

Extends: Allocator<T>

Overview: A simple allocator that uses the `new` operator to perform the actual allocation.

4.14 Class `AllocationPool<T, CHUNK_SIZE>`

Defined in: allocpool.h

Extends: Allocator<T>

Overview: An allocation pool allocates together chunks of values of type T, instead of allocating one value at a time. This reduce the memory waste for small objects (since usually the memory actually consumed is always a multiple of 8 or 16, no matters the size of the object), and can be also quite faster.

The drawback is that the values cannot be disposed singularly. Actually, they are owned by the AllocationPool object, and are freed when the AllocationPool is disposed. Be careful to never deallocate the AllocationPool before you have finished with the objects allocated through it.

4.15 Abstract class `State`

Defined in: state.h

Extends: -

Overview: This is an abstract representation of the SSR current state. Implementations of this class define the actual matching behavior, differentiating the various kinds of matchings and the various algorithms. **Note:** respect to pre-2.0 versions of the library, class State assumes explicitly a depth-first search.

In Table 1 (Algorithms), a list of the implementations of this class can be found, with the indication of the corresponding matching types and algorithms.

4.16 Class `Dictionary<Key, Value>`

Defined in: dict.h

Extends: -

Overview: A simple dictionary, implemented through linked lists. Can be used to store/retrieve Key/Value pairs. It is not particularly efficient.

4.17 Class `DictionaryIterator<Key, Value>`

Defined in: dict.h

Extends: -

Overview: A DictionaryIterator is like a cursor that can be used to traverse the contents of a Dictionary.

5 References

1. D. C. Schmidt, L. E. Druffel. "A Fast Backtracking Algorithm for Test Directed Graphs for Isomorphism", Journal of the Assoc. for Computing Machinery, 23, pp. 433-445, 1976.

2. J. R. Ullmann. "An Algorithm for Subgraph Isomorphism", Journal of the Assoc. for Computing Machinery, 23, pp. 31-42, 1976.

3. L. P. Cordella, P. Foggia, C. Sansone, M. Vento. "Performance Evaluation of the VF Graph Matching Algorithm", Proc. of the 10th ICIAP, IEEE Computer Society Press, vol. 2, pp. 1038-1041, 1999.

4. P. Foggia, C. Sansone, M. Vento. "An Improved Algorithm for Matching Large Graphs", accepted for the 3rd IAPR-TC15 Workshop on Graph-based Representations, Ischia, 2001.

5. P. Foggia, R. Genna, M. Vento. "Symbolic vs. Connectionist Learning: An Experimental Comparison in a Structured Domain", IEEE Trans. on Knowledge and Data Engineering, vol. 13, n. 2, 2001.