

Contents

Overview of the DAG Matcher Design.....	1
The DAG class	1
How to derive a class from the DAG class?	2
The DAGNodePtr class	2
EXAMPLE: How to integrate a new DAG-based class	2

Overview of the DAG Matcher Design

The best way of understanding this code is to think how one is supposed to create his own DAG classes. The main goal in the design of the project is to facilitate this task by avoiding code redundancy among the different DAG-derived classes. It is important to notice that several classes, such as the DAGDatabase, VisualDAG, and etc. need to work with DAG objects. Within the present design and through the use of virtual functions, classes like the DAGDatabase do not require explicit knowledge of the kind of DAG they are handling. They will discover the particular class of a DAG at run time.

The DAG class

The DAG class derives from the LEDA `GRAPH<DAGNode*, double>` class. This means that a DAG is a LEDA graph in which the information contained in a node is only a pointer to a DAGNode object. In addition, the edges of this graph can have a weight represented by a *double*. Having said this, it is straightforward to conclude that all the functions associated with a LEDA graph can be used with a DAG. Thus, to iterate through the DAG's nodes, we can use the well-known set of LEDA macros for this purpose. E.g.

```
void DAG::MyFunc()
{
    leda_node v;
    forall_nodes(v, *this)
    {
        DAGNode* p = inf(v);

        // do something with the info contained in *p.
    }
}
```

Since we usually need the information in a DAGNode object, instead of just a simple pointer to this object, there is a DAG member function, which simplifies the task of accessing this info given a `leda_node` parameter:

```
DAGNode* DAG::GetNode(leda_node v)
```

Then, we can re write the previous code as...

```
void DAG::MyFunc()
{
    String str;
    leda_node v;

    forall_nodes(v, *this)
    {
        str = GetNode(v)->GetNodeLbl();
    }
}
```

```

        // do something with the node's label
    }
}

```

Important point: there are situations in the code in which two nodes that belong to different graphs contain a pointer to the same DAGNode object. This is handled in a way that should not create any problem, but it is still good to know that this is the case.

How to derive a class from the DAG class?

The simplest way is to take a look at the ShockGraph class and the SGNode class, which are classes derived from DAG and DAGNode respectively. Otherwise, there is a sort of rule of thumb to create your own DAG-derived class and DAGNode-derived class. First you will need to provide an implementation for all the pure virtual member functions (those that have a prototype that looks like: `int Func(int, int) = 0`). Then, you can take a look at all the virtual (but not pure virtual) member functions of DAG and DAGNode to see which ones of them have a behaviors that is not the right one for your needs. These virtual functions are called by all the classes that use DAGs, and so you over right any of them in your derived classes to implement the particularities of your new DAG-derived class.

The important points to understand are:

- Both DAG and DAGNode work together so you need to derive a new class from each one of them.
- All the pure virtual functions need to be implemented or it will not be possible to create an instance of these classes.
- Virtual functions give a hint of the kind of functionality that can be change with out affecting classes that uses DAG objects.
- You can use the ShockGraph class and the SGNode class to see an example of how to implement the pure virtual functions.

The DAGNodePtr class

This class derives from SmartPtr, which is simply a pointer that keeps track of the number of pointers that point to the same object. When this number goes to zero, the object is destroyed.

There are two important points to remember about this class:

1. We can use it as if it were a DAGNode pointer (DAGNode*).
2. It's meant to avoid unnecessary copies of DAGNode's, but it requires a smart use of the keyword **const** to accomplish this result.

EXAMPLE: How to integrate a new DAG-based class

How to integrate a new DAG-based class within the existing framework for indexing and matching.

1) Any new graph class must derive from the class "DAG"

e.g.

```
class GestureGraph : public DAG
{
    ...
};
```

2) This new DAG will contain nodes. The particular attributes of each node and all the functions related to these attributes **MUST** go in a node class that must derive from "DAGNode".

e.g.

```
class GGNode : public DAGNode
{
public:
    double m_dSign;
    double m_dScale;

    virtual DAGNode* CreateObject() const;

public:

    GGNode(NODE_LABEL lbl = "") : DAGNode(lbl) { }
    GGNode(const GGNode &sb)          { GGNode::operator=(sb); }

    virtual void Clear();
    virtual DAGNode& operator=(const DAGNode& rhs);
    virtual void Print(ostream& os = cout) const;
    virtual istream& Read(istream& is);
    virtual ostream& Write(ostream& os) const;
};
```

3) The derived classes from DAG and DAGNode have a several virtual functions that can be overloaded. Look for them.

4) The DAG and DAGNode are abstract classes because they have a pure virtual function that must be implemented in its derived classes. This function is:

WITHIN DAG

```
virtual DAG* CreateObject() const;
DAGNodePtr GestureGraph::CreateNodeObject(NODE_LABEL lbl) const;
DAGNodePtr GestureGraph::ReadNode(istream& is) const;
String GestureGraph::ClassName() const;
```

IN DAGNode

```
virtual DAGNode* CreateObject() const;
```

And their implementations should be:

```
//! Creates a new shock graph object.
DAG* GestureGraph::CreateObject() const
{
    return new GestureGraph;
}

//! Creates a new gesture graph node object.
DAGNodePtr GestureGraph::CreateNodeObject(NODE_LABEL lbl) const
{
    return new GGNode;
```

```

}

/*!
    Creates a new GGNode and fill its contents with the current
    information in the stream.
*/
DAGNodePtr GestureGraph::ReadNode(istream& is) const
{
    DAGNodePtr node(new GGNode);

    node->Read(is);

    return node;
}

/*! Returns the class name
String GestureGraph::ClassName() const
{
    return "GestureGraph";
}

/*! Creates a new node
DAGNode* GGNode::CreateObject() const
{
    return (DAGNode*) new GGNode();
}

```

5) Look at the files GestureGraph.h and GGNode.h for a complete example. In addition, a bigger example can be found in ShochGraph.h and SGNode.h.

6) One important function to overload in the DAG-based class is:

```
virtual double NodeDistance(leda_node g1Node, const DAG& g2, leda_node g2Node) const;
```

and / or ...

```
virtual double NodeSimilarity(leda_node u, const DAG& from, leda_node v) const;
```

i.e., one can simple be the inverse of the other. For matching, NodeSimilarity is called.

6) For example, to create a graph whose nodes are blobs and ridges, one must create a graph class, say, "BlobRidgeGraph", and a node into class, say, "BRNode". This must be done following the steps 1 - 5.