Implementation of Directed Multigraphs in Java

In this paper we discuss some problems of implementation of graphs in object oriented programming languages. We have chosen directed multigraphs as the most general type of graphs and Java as a programming language. Java is often used as the first programming language in teaching computer science. Unfortunately, sometimes complexity of algorithms remains hidden in object oriented framework, and it is not obvious which methods are preferred.

We propose an implementation of graphs that is similar to adjacency list structure, introduced, for example, in [GT98] but our model does not duplicate any references that makes graph updates more efficient.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representations—*Object representation*; E.1 [**Data**]: Data Structures—*Graphs*; D.3.3 [**Software**]: Programming Languages—*Java*

1. MOTIVATION

Graphs and graph algorithms are significant topics in university degree courses like "Computer Science" or "Algorithms and Data Structures". There are different implementation strategies for the graph abstract data type depending on programming paradigm used and problems solved.

We have chosen Java as the first programming language in teaching instead of Pascal that was used for many years. This change has caused some difficulties in introducing classical data structures and mathematical "objects" (like graphs). This is mainly a problem of second programming course that needs to cover themes both from object oriented programming and algorithms and data structures. Object oriented style puts emphasis on modeling and decomposition, algorithmic approach concentrates on effective implementation, memory management, etc. For example, "linked list" in Java is a part of standard API, it is preferred to reuse this implementation instead of inventing a new one. On the other hand, implementing a "linked list" data structure is an important skill when learning data structures and algorithms. This is not a real contradiction, we can always instruct students to produce their own "linked list" from scratch, but sometimes such an approach confuses the audience.

In case of graphs, we have implemented basically the same idea in two ways. First is direct "translation" from Pascal pointer structure, second is more object oriented trying to reuse existing API. The result is somewhat amazing: direct implementation instead of delegation is much shorter. On the other hand, it is less transparent and more sensitive to later changes. As these implementations have the same complexity, we introduce only the second approach in this article and leave the "direct" way mentioned above as an exercise for potential programming class.

This program can be used both as reference implementation for graphs and API extension for solving graph problems. Some methods are more recommended to write efficient programs, some are introduced to make this implementation as complete as possible.

2. BASIC STRUCTURE FOR DIRECTED MULTIGRAPHS

In object oriented approach we need to decompose our domain terms into classes: "graph", "vertex", "edge", etc. Each class has its own responsibilities, implemented via methods. Starting from method descriptions it is not easy to find an underlying structure. We can spend hours creating lists with "useful" and "universal" method signatures (people having ADT experience tend to love this), but still have no idea about efficient way to implement these operations.

Let us start from the basic structure.

(Multi)graph $G = (V, E)$ consists of final set of vertices V and final (multi)set of edges $E \subset V \times V$.

We allow multiple edges between the same pair of vertices, even loops, also we consider all pairs ordered (edges are directed). In case we need an undirected edge (arc) $\{u, v\}$ we model this using two directed edges (u, v) and (v, u) .

Basic structure can be expressed in following three sentences.

1. The graph is represented by object that contains list of vertices.
2. Each vertex is represented by object that contains list of outgoing edges.
3. Each edge is represented by object that contains pointer to its end-vertex.

As we can see this is very similar to the adjacency list structure used in [GT98] but not symmetrical: we do not pay attention to incoming edges (all edges are outgoing edges for some vertex and only edge itself knows where it goes).

This structure has the following benefits:

1. Each object (graph, vertex, edge) has exactly one representative, all updates (insert, delete) made to the graph are "local" (only one of the lists is changed).
2. Graph is scalable and dynamic (we do not use static containers).
3. This representation is complete (we can implement all operations, but not all operations are efficient, e.g. iterator over incoming edges).
4. Most algorithms use outgoing edges only. If, for example, we need to count incoming edges in topological sort algorithm, we can use the vertex workfield.

3. IMPLEMENTATION

The following classes and methods are implemented. `get-` and `set`-methods have no side-effects, they only read and write corresponding fields. Constructors integrate new objects to existing structure. All objects contain identifier (used in search methods) and two workfields: one for objects and another for numerical results. The workfields have no importance now, we could introduce these later in subclasses.

Mostly semantics of methods is obvious, few additional explanations follow. Full source code and `javadoc`-documentation are available on authors web-page.

3.1 Class Graph

Graph contains a list of vertices. Main vertex operations are delegated to this list.

```
Graph (Comparable id)

public Comparable getId()
public void setId (Comparable c)

public List getVertexList()
public void setVertexList (List l)
```

```
public Object getGObject()
public void setGObject (Object o)

public int getGInfo()
public void setGInfo (int i)

public Iterator vertices()
public Iterator edges()

public String toString()
public Object clone() throws CloneNotSupportedException

public boolean insertVertex (Vertex v)
public Vertex findVertex (Comparable id)
public boolean removeVertex (Vertex v)

public boolean insertEdge (Edge e, Vertex from, Vertex to)
public Edge findEdge (Comparable id)
public boolean removeEdge (Edge e)

public Iterator edgesBetween (Vertex from, Vertex to)
```

From these methods `removeVertex` is not efficient, it removes vertex together with both outgoing and incoming edges, but incoming edges are available only through filtering all edges. Iterator `edges` is complex, it iterates over list of lists.

3.2 Class Vertex

Vertex contains a list of outgoing edges. Most operations on edges are delegated to this list. In addition, vertex contains reference to the graph it belongs to.

```
Vertex (Comparable id, Graph g)

public Comparable getId()
public void setId (Comparable c)

public List getEdgeList()
public void setEdgeList (List l)

public Graph getGraph()
public void setGraph (Graph g)

public Object getVObject()
public void setVObject (Object o)

public int getVInfo()
public void setVInfo (int i)

public Iterator outEdges()
public Iterator inEdges()

public String toString()

public boolean insertOutEdge (Edge e, Vertex to)
public Edge findOutEdge (Comparable id)
public boolean removeOutEdge (Edge e)
```

3.3 Class Edge

Edge contains references to its end-vertices.

```
Edge (Comparable id, Vertex from, Vertex to)

public Comparable getId()
public void setId (Comparable c)

public Vertex getToVert()
public void setToVert (Vertex v)

public Vertex getFromVert()
public void setFromVert (Vertex v)

public Object getEObject()
public void setEObject (Object o)

public int getEInfo()
public void setEInfo (int i)

public String toString()
public void changeFromVert (Vertex v)
public void reverse()
```

Method `changeFromVert` is introduced to make structural change in the graph: moving current edge to another list of outgoing edges.

4. DISCUSSION

Mathematical "objects", ADT and classical data structures are not easy to explain and implement using object oriented approach, partially because there are so many possibilities to build corresponding OO-models. There is no single solution for complex structures like graphs (the other interesting topic related to implementation of classical data structures in Java is influence of programmers previous experience).

Power of polymorphism often hides complexity issues, we do not know the "price" of components we use. Starting everything from scratch is sometimes a good idea, but usually dangerous and time consuming. Maybe it is useful to introduce complexity information into method declaration, starting, for example, from special `javadoc`-comments. Informal descriptions already appear in standard Java API documentation but do we need something stronger and more formal?

Our implementation of directed multigraphs relies on `LinkedList` class from `java.util` package, most of operations are finally delegated to some `LinkedList` object. On the other hand, if search speed becomes crucial, we can easily replace `LinkedList` to something more efficient, only four lines of code need to be rewritten.

REFERENCES

[GT98] M.T.Goodrich, R.Tamassia. Data Structures and Algorithms in Java. John Wiley and Sons, 1998, 738 pp.