Implementation of Directed Multigraphs in Java

Jaanus Pöial
University of Tartu, Estonia
jaanus@cs.ut.ee

ABSTRACT

An implementation of directed multigraphs is introduced for teaching course on algorithms and data structures with Java.

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. INTRODUCTION

There are different implementations of the graph abstract data type depending on programming paradigm used and problems to be solved. When teaching algorithms and data structures with Java we need a reference implementation for graphs that has good features from both object oriented and algorithmic viewpoint. Object oriented style puts emphasis on reusable components, modeling and decomposition, algorithmic approach concentrates more on complexity issues, memory management, etc. For example, a linked list in Java is a part of standard API, in practice it is preferred to reuse this implementation instead of inventing a new one. On the other hand, implementing a linked list is an important skill when learning about data structures. This is not a real contradiction, we can always instruct students to produce their own linked list implementation from scratch.

In case of graphs, we have implemented basically the same idea in two ways. The first is a direct memory structure that reinvents list operations, the second is more object oriented and reuses 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.

Our program can be used both as reference implementation for graphs and API extension for solving graph problems.

2. BASIC STRUCTURE FOR DIRECTED MULTIGRAPHS

(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) .

The basic structure can be expressed as follows:

1. The graph is represented by an object that contains list of vertices.

2. Each vertex is represented by an object that contains list of outgoing edges.

3. Each edge is represented by an object that contains (pointer to) its end-vertex.

As we can see this is very similar to the adjacency list structure used in [1] 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).

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 mostly on `LinkedList` class from `java.util` package, many 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.

3. REFERENCES

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