МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2019
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2019
Proceedings of the Forty-eighth Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 1–5, 2019

GRAPHSL: STANDARD LIBRARY FOR GRAPHS AND TREES*

Krassimir Manev, Mario Georgiev, Neli Maneva

Using libraries of standard subprograms is one of the traditional approaches for amelioration of the process of creating programs. Many libraries, including predefined abstract data types and corresponding methods, are crated for different programming languages and platforms. The most popular standard library for the GNU compilers of C/C++ language is STL. Unfortunately, despite its large volume, STL does not maintain classical abstract data types for presenting graph structures and trees. This paper describes an implementation of such data types.

1. Introduction. Programming with *abstract data types* (ADT) is an old approach, which has its modern form in the Object-Oriented Programming paradigm. Main profit of using this approach is that it separates the logic of the program, expressed in operation with the abstract types, from implementation of the operations themselves. In such a way the implementation of the ADT could be done better and better during the life cycle of the program without changing the implementation of the program logic.

Traditionally implementation of different ADT are collected in *libraries of standard subprograms*. Many libraries, including predefined abstract data types and corresponding methods, are created for different programming languages and platforms from years. During the years each such library is developed with including new objects and replacing used algorithms with better ones.

The most popular standard library for the GNU compilers of C/C++ language is the *Standard Template Library* STL. Unfortunately, despite its large volume and constantly increasing quality, STL does not maintain classical ADT for presenting graph structures and trees. Different researchers and developers are implementing their own libraries with appropriate for their purposes interfaces but there is still no standard for implementing graph structures and trees, despite the importance of these ADT and their widespread use.

This paper describes one possibility for standardization of a set of ADT for presenting graph structures and trees as classes of objects in C/C++ as well as implementation of corresponding interfaces that contain some basic operations with the proposed ADT. In Section 2 a very brief overview of the necessary terms and concepts is given. Section

*2010 Mathematics Subject Classification: G.2.2, E.1.
Key words: Abstract data types, data types and structures, procedures, functions and subroutines, graph algorithms.

3 defines the proposed system of classes for presenting graph structures and trees following the STL style. Section 4 describes a set of classical graph algorithms that was implemented for verification of the implementations of the ADT and comparing their quality with similar implementations programmed in kernel C. Section 5 contains some conclusions and remarks.

2. Terms and concepts. Each graph structure G is composed of set V of *vertices* and set E of *edges*. Each edge $e(u, w) \in E$ is connecting two vertices $u \in V$ and $w \in V$. In some graph structures edges are directed from one of the two vertices (called *begin* or *parent*) to the other (called *end* or *child*). Such structures are called *directed*. In other graph structures the edges have not an orientation – these structures are *undirected*. In this case the two vertices that the edge connects are called *neighbors*. In some graph structures it is possible to have more than one edge that links two vertices. Such structures are called *multigraphs*. When such repetition of edges is not permitted the structure is simply called *graph*. So we consider four abstract data types – *directed multigraph*, *directed graph*, *undirected multigraph* and *undirected graph*, the last one called simply *graph*.

When we introduce a graph structure in the computer memory we usually specify the structure graph or multigraph, the number N of the vertices usually denoted by the integers from 1 to N, the number M of the edges and which two vertices each of the edges connects. There are three most popular forms for defining the connections in the structure:

• *List of edges* consists of M couples of vertices – one couple for each edge of the structure. When the structure is directed the begin of the edge is listed before the end. When the structure is undirected the both vertices are listed in arbitrary order.

• Presentation with *Lists of neighbors/children* is composed of N lists of vertices – one list per vertex. In undirected case each list contains the neighbors of the corresponding vertex and in the directed case – its children.

• *Adjacency matrix* is $N \times N$ matrix G. The element $G[u][w]$ in the row u and column w of the matrix contains the number of edges that link vertex u with vertex w. In undirected case $G[u][w] = G[w][u]$, but in directed case the two values are not necessarily equal.

Sometime the objects from the life that are modeled with graph structures have different characteristics – called *length*, *weight*, *price*, etc., associated with the edges or the vertices of the structure. Here we will consider only graphs (multigraphs excluded) with integer numerical characteristics of the edges called *weight*, will denote the price of the edge (u, w) with $c(u, w)$ and will call such structure *weighted*.

Trees are special kind of graphs with very important role not only in the algorithmic in graphs but in informatics in general. Classical tree, defined as connected graph without circuits, is not appropriate for informatics because of the difficult from algorithmic point of view properties "connected" and "without circuits". That is why we use the much more appropriate *rooted tree* – a tree in which one of the vertices is chosen for *root*. A rooted tree is constructed inductively, starting with trivial tree $T(\{r\}, \emptyset)$ with a single vertex r as a root and no edges. The inductive operation, starting with a rooted tree $T(V, E)$ with root r, links a new vertex $w \notin V$ to some of the vertices $u \in V$ giving new rooted tree $T'(V \cup \{w\}, E \cup \{(u, w)\}$ with the same root. This operation makes the undirected rooted tree *explicitly directed* and gives us a possibility to call u *parent* of w

and w – *child* of u.

Rooted trees could be presented in the computer memory in any of the three universal presentations for graph structures listed above. But there is a much better presentation for the rooted trees – *List of parents*, which contains for each vertex its parent. As parent of the root we take 0.

For more detailed introduction in the topic we recommend some of the numerous textbooks in Algorithms (for example [1]) or Algorithms in Graphs (for example [2]).

3. GraphSL library. The most simple for defining as a class of objects is the Adjacency matrix:

```
class GraphAdjacencyMatrix {
private:
   int N, M; bool directed; vector<vector<int>> G,C;
public:
   GraphAdjacencyMatrix(int N, int M, bool directed);
   int getVNum(); int getENum(); bool getIsDirected();
   void connect(int u, int w);
   bool isConnected(int u, int w);
   vector<vector<int>>& getGraph();
};
```

Traditionally, the number of vertices will be always in the variable `N` and the number of edges in `M`. The fact that the structure is directed will be marked with `true` in the variable `directed` and the matrix itself will be in the container `G` of type `vector<vector<int>>`. When we have weighted graph or directed graph then the weight of the edge (u, w) is in `C[u][w]`. Methods `get*` return the mentioned in the name value. The method `connect` appends an edge from vertex `v` to vertex `w` when the structure is directed or an edge between `v` and `w` when the structure is undirected. Finally, the method `isConnected` checks if there is at least one edge connecting `v` and `w`.

For implementation of the two other presentations we introduced an auxiliary class `Edge` as follow:

```
class Edge {
public:
   int u, v, c;
   static Edge newEdge(int u, int v, int c);
   static Edge newEdge(int u, int v);
   Edge(int u, int v, int c);
   Edge(int u, int v); Edge();
};
```

Lists of neighbors/children class is defined and implemented as follows:

```
class GraphAdjacencyLists {
private:
   int N, M; bool directed; vector<vector<Edge>> G;
public:
   GraphAdjacencyLists(int N, int M, bool directed);
```

```
    int getVNum(); int getENum(); bool getIsDirected();
    void connect(int u, int v);
    void connect(int u, int v, int w);
    vector<vector<Edge>>& getGraph();
};
```

The purpose of the elements and the semantics of the methods here is not different from these in the class `GraphAdjacencyMatrix`.

The presentation List of edges is used inside programs rarely. But this is the presentation in which the graph structures are usually given to the input of programs. Our implementation is the following:

```
class GraphEdgeList {
private:
    int N, M; bool directed; vector<Edge> G;
public:
    GraphEdgeList(int N, int M, bool directed);
    int getVertices(); int getEdges();
    void addEdge(int u, int v, int c);
    void addEdge(int u, int v);
    bool isConnected(int u, int v);
    vector<Edge>&getGraph();
    GraphAdjacencyMatrix* transformToAdjacencyMatrix();
    GraphAdjacencyLists* transformToAdjacencyLists();
};
```

Beside the methods that we have in the previous two classes here two special methods are included that transform List of edges to more frequently used Adjacency matrix and Lists of neighbors/children.

Finally, our implementation of rooted tree is the class `RootedTree` as follows:

```
class RootedTree {
private:
    int N, root; vector<int> parent;
public:
    RootedTree(int N, int root); int getRoot();
    int getParent(int u); void connect(int u, int v);
};
```

For rooted tree the number of edges is always $N - 1$ that is why it is not necessary to keep the number of edges.

The full implementation of the classes is published in [3].

4. Verification. A set of some popular algorithms was implemented in order to check usability of the defined classes and their correctness, namely:

• Breadth first search approach and some of its derivatives (spanning tree, connected components, shortest path in not weighted graph structures);

• Depth first search and some of its derivatives spanning tree, connected components, finding bridges and articulation points, strongly connected components of directed structures, topological sort of DAG);

• Euler traversals in multigraps;

• Minimal/maximal spanning trees in weighted graph structures (algorithms of Prim and Kruskal);

• Shortest path in weighted graph structures (algorithms of Dijkstra, Bellman-Ford and Floyd-Warshall);

• Max flow in weighted graphs (algorithms of Ford-Fulkerson and Edmond-Karp).

Full implementations of these algorithms are published in [3].

For checking the correctness and effectiveness of the implemented algorithms a comparison of some of them with corresponding algorithms from [4], implemented in kernel C, was made. For each of the listed in Table 1 algorithms two test cases was run in both implementations on Intel Pentium G630 CPU working on 2.7 GHz. First test was a small graph with 10–20 vertices – just to verify that both implementations give equal results, and the second – graph with 2000 vertices and 1000000 edges. The results are summarized in Table 1.

Table 1. Comparing efficiency of the implementations

Task	N	M	Kernel C	GraphSL
BFS spanning tree	2000	1000000	0,60 sec	0,64 sec
DFS spanning tree	2000	1000000	0,58 sec	0,64 sec
Euler circuit	2000	1000000	4,32 sec	4,55 sec
Min spanning tree (Prim)	2000	1000000	0,76 sec	0,90 sec
Min spanning tree (Kruskal)	2000	1000000	1,46 sec	1,80 sec
One-to-all shortest path	2000	1000000	0,82 sec	0,82 sec

5. Conclusion. The experiments show no significant difference between the efficiency of the Kernel C and GraphSL implementation. Some amelioration in GraphSL could be obtained if implementation is made without using the STL containers.

REFERENCES

[1] T. H. Cormen, Ch. A. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms. MIT Press, 2009.

[2] R. Sedgwick. Algorithms in C++ Part 5: Graph Algorithms, 3rd Edition, Addison-Wesley Professional, 2002.

[3] М. Георгиев. Алгоритми и реализация на абстрактни класове за работа с графови структури и дървета. Бакалавърска теза, Нов Български Университет, София, 2019.

[4] Кр. Манев. Алгоритми в графи. Основни алгоритми. КЛМН, София, 2013.

Krassimir Manev
e-mail: kmanev@nbu.bg
Mario Georgiev
e-mail: mario.r.georgiev@gmail.com
New Bulgarian University
21, Montevideo Blvd
1618 Sofia, Bulgaria

Neli Maneva
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Acad. G. Bonchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: neman@math.bas.bg

GRAPHSL: СТАНДАРТНА БИБЛИОТЕКА ЗА ГРАФИ И ДЪРВЕТА

Красимир Манев, Марио Георгиев, Нели Манева

Използването на стандартни библиотеки от подпрограми е един от традиционните подходи за подобряване процеса на създаване на програми. Много библиотеки, включващи предефинирани абстрактни типове данни и съответните методи, са създадени за различни програмни езици и платформи. Най-популярната библиотека за компилаторите от серията GNU за езика C/C++ е STL. За съжаление, въпреки големия си обем, STL не поддържа класическите абстрактни типове за представяне на графови структури и дървета. Тази статия предлага една възможна стандартизация за такива типове данни.