Representation of graphs for storing in relational databases

Mikhail Urubkin[1,*], *Vasiliy* Galushka[1], *Vladimir* Fathi[1], *Denis* Fathi[1], and *Alla* Gerasimenko[1]

[1]Don state technical university, chairs Computing systems and information security, 1, Gagarin square, 344000, Rostov-on-Don, Russia

Abstract. The article is devoted to the problem of representing graphs in the form that is most suitable for their recording in relational databases and for subsequent efficient extracting and processing. The article analyzes various ways to describe graphs, such as adjacency and, incidence matrices, and adjacency lists. Each of them is reviewed from the point of view of their compliance with normal forms to assess the possibility of using a particular method when developing databases for storing graphs. It is shown that for such a task, each of these methods has a large number of disadvantages that lead to low efficiency of both data storing and processing. The article suggests the way to represent graphs in the form of a relational list of edges corresponding to the third normal form and allowing to eliminate the disadvantages of other methods.

1 Introduction

Graphs are a set of vertices and connections between them, called edges, and are abstract mathematical objects that provide a universal way to describe various structures and systems. Methods of processing graphs using standard algorithms are the basis of many modern information systems [1, 2]. Graph theoretic models are most popular in the study of communication networks, chemical and genetic structures, electrical circuits, and other systems with a network structure.

At the same time, the software part of any information system has two aspects: data structures and algorithms for processing them [3]. This principle is reflected in many modern information technologies, in particular, in object-oriented programming and databases. These aspects are interrelated. Algorithms are always designed to work with a specific data structure, and the choice of the way to represent the data significantly affects the methods of the data processing and the effectiveness of these methods.

For some special cases of graphs, for example, trees, there are effective ways to represent them for writing to relational database tables that take into account the structure of such graphs and their other features [4, 5]. However, these methods are not applicable for other types of graphs. Instead, you can use one of the universal ways to represent graphs, such as the adjacency matrix.

*Corresponding author: mishanya005@ya.ru

The problem with this is that such representation methods do not comply with the principles of the data storage in relational databases [6, 7], which results in the need to perform additional transformations of the data when it is transferred from the database to the application for further processing. This raises the challenge of developing a table structure for storing graphs and corresponding methods for processing them in order to reduce the overhead of converting data from one format to another.

2 Forms of the graph representation

In discrete mathematics, there are currently several ways to represent graphs. These primarily include adjacency matrices, incidence matrices, and adjacency lists.

For Graph G shown in Figure 1, consider its representation using each of these specified methods.

The classical adjacency matrix is a square matrix *A* of size *n*, in which the value of the element a_{ij} is equal to the number of edges from the *i*th vertex of the graph to the *j*th vertex. It is more common to define the adjacency matrix element a_{ij} as the edge length from the ith vertex to the *j*th vertex.

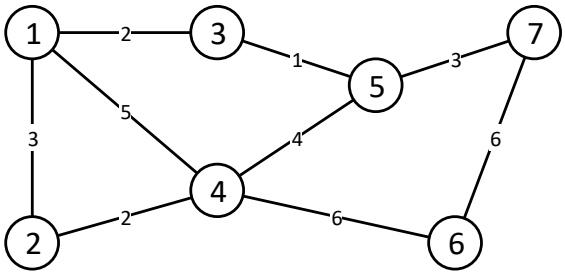


Fig. 1. Example of a Graph.

Another form of the graph representation is the incidence matrix, which specifies the relationships between graph elements: edges and vertices. The matrix columns correspond to edges, and the rows correspond to vertices. A non-zero value in a matrix cell indicates the relationship between a vertex and an edge (their incidence).

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	3	2	5	0	0	0
v_2	3	0	0	2	0	0	0
v_3	2	0	0	0	1	0	0
v_4	5	2	0	0	4	6	0
v_5	0	0	1	4	0	0	3
v_6	0	0	0	6	0	0	6
v_7	0	0	0	0	3	6	0

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
v_1	1	1	1	0	0	0	0	0	0
v_2	1	0	0	1	0	0	0	0	0
v_3	0	1	0	0	1	0	0	0	0
v_4	0	0	1	1	0	1	1	0	0
v_5	0	0	0	0	1	1	0	1	0
v_6	0	0	0	0	0	0	1	0	1
v_7	0	0	0	0	0	0	0	1	1

Fig. 2. a) adjacency matrices; b) incidence matrices.

The definitions and figures show the advantages and disadvantages of each method. It should be noted that incidence matrices are not suitable for the use with weighted graphs, so only adjacency matrices will be considered in this article.

The main advantage of the adjacency matrix is the low computational cost of performing operations with edges: adding, deleting, and evaluating the length [8]. These

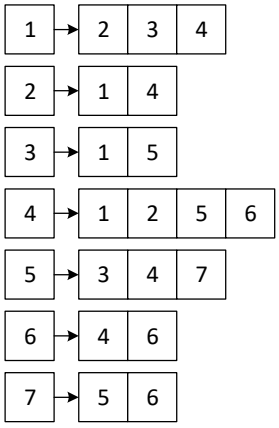
operations are performed in a constant time $O(1)$, and when implemented programmatically, they require a simple reference to a cell in a two-dimensional array.

On the other hand, similar operations with vertices that affect their number (adding and deleting them) require changing the size of the very matrix, which leads to the use of many more computational resources than simply entering a value in a cell of a two-dimensional array when adding an edge, regardless of the programming language.

Another disadvantage of the adjacency matrix is that the efficiency of the memory consumption depends on the density of a graph — if each vertex has a relatively small number of edges, then there will be many zero values in the adjacency matrix, for which the memory must also be allocated.

The next way to represent graphs is an adjacency list, according to which the graph is represented as array L, each element of which denotes vertex i and contains another array L_i with the numbers of vertices j connected to the i-th vertex. There are several approaches to the software implementation of adjacency lists: using collections, step arrays, or other dynamic data structures depending on the programming language, as well as representation as a table with 2 columns, where the first column indicates the vertex number and the second column specifies all adjacent to it (Fig. 3).

The adjacency list is efficient in terms of storing — among all the methods, it requires a minimum amount of the memory, since it contains only really existing connections. In this case, the operations of adding or deleting both the vertex and the edge are also performed in a constant time $O(1)$.



vertex	adjacent vertices
1	2, 3, 4
2	1, 4
3	1,5
4	1, 2, 5, 6
5	3, 4, 7
6	4, 6
7	5, 6

Fig. 3. Forms of Recording the Adjacency List.

The adjacency list is easy for a person to understand and write programs for processing it, but its effective implementation is possible only in high-level programming languages (C#, Java, Python) with the built-in support for dynamic data types, which negatively affects the performance of such programs in comparison, for example, with the C++ programming language.

These advantages and disadvantages are true when implementing a particular way of representing graphs in traditional algorithmic programming languages. However, the modern approach to designing information systems requires dividing them into modules, each of which performs its own function: data storing, processing, and displaying for the user. At the same time, for the data storage there are always used such databases, for the design of which a large number of formal requirements must be met, and the assessment of compliance with these requirements is made on the basis of normal forms.

3 Assessing the methods of graph representation based on the normal forms

The normal form is a relation feature in the relational data model that characterizes it from the point of view of redundancy, which can potentially lead to logically erroneous results of sampling or changing data [6].

The database is considered normalized if its tables comply with at least the third normal form. Let's consider the described above methods for representing graphs in terms of their normalization.

The first normal form requires each relation attribute to store an atomic value. In addition, the rows in the table should not depend on each other, should not be repeated, and their order should not affect the understanding of the information.

The adjacency matrix satisfies the atomicity requirement, since each of its cells stores a single value, but the order of rows in it is important, and according to the definition, each row must display the adjacency of the next vertex to all the others. The rows of the adjacency matrix for an undirected graph are also not independent. When adding an edge from the *i*-th vertex to the *j*-th vertex or changing its length, you must make changes to the cells in 2 different rows: a_{ij} and a_{ji}. Therefore, the adjacency matrix is not in the first normal form.

Similarly, it does not contain an adjacency list either, since when it is written as a table, its second column contains a list of vertices adjacent to the vertex specified in the first column, that is, a non-atomic value.

It should be noted specifically that all the methods listed above for representing graphs (except for the table-oriented view of the adjacency list) do not have a fixed quantity of columns. Their quantity depends either on the total number of vertices, or on the number of connections of each vertex to the others. In this regard, a formal record of these methods in the form of a relation diagram is not possible and therefore is not given. An even bigger disadvantage in terms of storing graphs in the database is the need to change the quantity of columns when adding or removing vertices. Although this functionality is available in any database management system, its using is highly unrecommended due to the impaired performance and increased complexity of the program code for changing the table structure.

4 Relational List of Edges

These disadvantages lead to the extreme inefficiency when any of the described methods of the graph representation is used for storing graphs in relational databases [9, 10], and therefore it is necessary to develop a new method that satisfies the following requirements:

- compliance with the 3rd normal form;
- minimizing the amount of memory used, the absence of empty cells;
- the ability to present a wide range of graphs: oriented/non-oriented, weighted/unweighted;
- simplicity of software implementation of processing algorithms.

As mentioned earlier, the closest way to the 3rd normal form of graph representation is a tabular representation of the adjacency list. The scheme describing his relationship is:

$$S_L = \{v, adj\_vs\}, \quad (1)$$

where *v* is the vertex number,

adj_vs — list (array) of adjacent vertices.

It can be seen from the scheme that the only drawback of this presentation method is the non-atomicity of the *adj_vs* attribute. To eliminate it, it is necessary to split each tuple into

several independent ones as follows: in each tuple of the relation L for the vertex v_i, one of the vertices adjacent to v_i must be presented. The other adjacent to it must be presented in another tuple. As a result, each row of the table created for a given relation will store information about the connection of two vertices, that is, about the edge of the graph.

The relational list of edges will be called Relation *R*, with the following scheme [9, 10]:

$$S_R = \{id, v1, v2, len\}, \quad (2)$$

where *id* is the surrogate primary key of the relation,

v1 is the number of one of the vertices,

v2 is the number of the vertex with which vertex *v1* is connected,

len is the length of the edge connecting vertices *v1* and *v2*.

Thus, each tuple of this relation is a certain edge of the graph. The only attribute in this relation which doesn't characterize the edge is *id*. It is added to the relation to serve as its primary key and takes values [0, 1, 2 ...], thus ensuring that each row is unique. In addition, the *id* value can be considered the number of a particular edge.

In theory, the *id* attribute is optional. Instead of it, you can create a composite primary key from the pair of attributes *v1* and *v2*. However, this key will only be unique for graphs where there can be just one connection between two vertices. It also does not provide uniqueness for graphs with multiple loops. In this regard, it is preferable to use an artificially designed key, called a surrogate key, which, being independent of any real data, in addition to the guaranteed uniqueness, also provides immutability and has a smaller size.

An example of a table corresponding to the relation R for the previously considered graph is shown in Figure 4.

R

id	v1	v2	len
0	1	2	3
1	1	3	2
2	1	4	5
3	2	1	3
4	2	4	2
5	3	1	2
6	3	5	1
7	4	1	5
8	4	2	2

R (Continuation)

id	v1	v2	len
9	4	5	4
10	4	6	6
11	5	3	1
12	5	4	4
13	5	7	3
14	6	4	6
15	6	7	6
16	7	5	3
17	7	6	6

Fig. 4. Representation of the relation R.

The relation in question complies with the first normal form: each attribute of it stores only one value, that is, it is atomic, the rows in the table representing this relation are independent and will not be repeated, since each of them represents a separate edge and has its own unique identifier, and the order of the rows is also not important.

This relation is in the second normal form too. This form requires the relation to comply with the first normal form, and that all its non-key attributes depend only on the primary key. Given that the *id* key is the edge identifier, all other attributes (*v1, v2, len*) depend on it.

The third normal form requires the relation to be in the second normal form and each of its non-key attributes to be non-transitively dependent on the primary key. There are no transitive dependencies in the given scheme of Relation *R*: *v1* does not depend on *v2*, since

any vertex can be connected to any other, *len* (the edge length) also does not depend on the numbers of vertices that it connects. Therefore, Relation R in question is in the third normal form, which means it works better for the implementation in the relational database than adjacency matrices or adjacency lists.

4 Conclusion

Representation of a graph as the relation described in this article allows you storing it effectively in a database, and also has other advantages over other ways of representing graphs. These advantages, first of all, include the minimum possible amount of the memory required for storing the graph, high speed of adding/removing vertices or edges, ease of processing the stored information by means of queries, and the ability to represent any types of graphs, including directed, undirected, cyclic, etc.

The reviewed method of the graph representation can be used when developing information systems with a distributed architecture, in which one of the relational systems for the database management is responsible for the data storage.

References

1. O.V. Klyuchnikova, S.S. Kadilin, Inzenernyj vestnik Dona **2** (2013) ivdon.ru/magazine/archive/n2y2013/1666/
2. S.V Astanin, N.V. Dragnysh, N.K. Zhukovskaya, Inzenernyj vestnik Dona **4** (2012) ivdon.ru/magazine/archive/n4p2y2012/1434/
3. S.S. Skiena, *The Algorithm Design Manual: Data Structures for Graphs* (Springer, New York, 2008)
4. V.N. Kasyjanov, V.A. Yevstigneyev, *Graphs in programming: processing, visualization and application* (BKHV-Peterburg, SPb, 2013)
5. Ch. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, *48th Annual Southeast Regional Conference, ACM SE* (2010)
6. T.S. Karpova, *Databases: models, development, implementation* (Piter, SPb, 2011)
7. M. Schlichtkrull, T. Kipf, P. Bloem, R. van den Berg and I. Titov, *15th International Conference*, Springer, 593-607 (2018)
8. K.Mi. Lee, K.My. Lee, Applied Mechanics and Materials Trans Tech Publications **241**, 3165–3170 (2012)
9. V. Benzaken, E. Contejean and S. Dumbrava, *ESOP 2014: Programming Languages and Systems*, Springer, 189-208 (2014)
10. A.V. Markin, Building queries and programming on SQL (DIALOG-MIFI, Moscow, 2008)