Chapter 1

Graph Representation

D. Dominguez-Sal
Universitat Politècnica de Catalunya, Spain

V. Muntés-Mulero
Universitat Politècnica de Catalunya, Spain

N. Martínez-Bazán
Universitat Politècnica de Catalunya, Spain

J. Larriba-Pey
Universitat Politècnica de Catalunya, Spain

ABSTRACT

In this chapter, we review different graph implementation alternatives that have been proposed in the literature. Our objective is to provide the readers with a broad set of alternatives to implement a graph, according to their needs. We pay special attention to the techniques that enable the management of large graphs. We also include a description of the most representative libraries available for representing graphs.

INTRODUCTION

Graphs are a mathematical abstraction to represent a set of entities and their relationships. These entities are represented as the nodes of the graph and the edges correspond to the relationships among these entities.

The use of graphs is very appropriate for many datasets where we want to obtain information about how the entities are related or how the topology of the network evolves. For example, if we want to represent the data associated to a social network we may represent the users as nodes of the graphs and draw friendship relations among those users.

The analysis of this topology can be exploited to find communities of users with similar interests or find influential people (i.e. authorities) in the network.

Another common scenario is travelling routes, where the nodes are the destinations and the edges are the communication routes among them. In this scenario, some important questions are related to obtaining the shortest or cheapest route from a certain spot to a different place. Also, building routes to visit a large set of destinations in the minimum travel time or finding travel routes that may have an important impact in the overall travelling structure in case of malfunction.
In order to analyze the previous problems and others, it is necessary to efficiently represent graphs in computers. Graph data structures should be able to compute common operations such as finding the neighbors of a node, checking if two nodes are connected by an edge, or updating the graph structure in a short time. However, no data structure is optimal for all graph operations and hence depending on the applications one representation may be preferable than another. Here, we review some of the most relevant graph representation solutions found in the literature.

The chapter is organized as follows. First, we describe simple data structures for representing graphs, which are especially adequate for implementations without big spatial or temporal constraints. Next, we review advanced techniques for improving the in-memory graph representations and compress the graph when performance is important. Then, we describe some distributed graph techniques that aim at providing highly scalable architectures for graph processing. We describe the approach by the Resource Description Framework (RDF) community to describe graph datasets. Finally, we review some of the graph software libraries that are available for representing graphs, showing DEX as an example of the internals of a modern graph database.

**CLASSICAL STRUCTURES FOR REPRESENTING GRAPHS**

In this section, we introduce some of the most popular data structures for representing graphs. They are important because some of the techniques described later in the chapter are based on these data structures. We summarize the representations described in this section in Figure 1, depicting the four different graph representations for a sample graph.

**Adjacency Matrix**

A graph \( G = (V, E) \) is commonly modeled as the set of nodes (also called vertices, \( V \)) plus an associated set of edges \( E \). Edges can be expressed as pairs of the two nodes that are connected. We can represent this neighboring information with the aid of a bidimensional array \( m \) of \( n \times n \) boolean values, in which \( n = |V| \). The indexes of the array correspond to the node identifiers of the graph, and the boolean junction of the two indices indicates whether the two nodes are connected through an edge in the graph. If the graph is undirected, then the matrix is symmetric: \( m_{ij} \) and \( m_{ji} \) have the same value, which indicates that there is an edge between nodes \( i \) and \( j \). If the graph is directed, \( E_i, \ldots, E_j \) indicates an edge that is going from the node \( i \) to \( j \), and \( m_{ij} \) indicates an edge going from the node \( j \) to \( i \). A variant of the adjacency matrix representation for weighted graphs is to substitute the boolean entries by integers, and these integers can be used to encode the weights of the edges.

Although this implementation is very efficient for adding/removing edges or checking if two nodes are connected because the operation is immediate, it has three important drawbacks. First, it takes a quadratic space with respect to the number of nodes, independently of the number of edges, and hence it is not adequate if the graph is sparse, which tends to be the case. Second, the addition of nodes is expensive because a new matrix is reallocated and the contents copied to the new structure. Finally, the operation to retrieve all the neighboring nodes takes linear time with respect to the number of vertices in the graph. Since the most used operation for traversals is obtaining the neighbors of one node, matrix arrays are not adequate for traversing large graphs.