A graph is a data structure composed of dots (i.e., vertices) and lines (i.e., edges). The dots and lines of a graph can be organized into intricate arrangements. A graph’s ability to denote objects and their relationships to one another allows for a surprisingly large number of things to be modeled as graphs. From the dependencies that link software packages to the wood beams that provide the framing to a house, most anything has a corresponding graph representation. However, just because it is possible to represent something as a graph does not necessarily mean that its graph representation will be useful. If a modeler can leverage the plethora of tools and algorithms that store and process graphs, then such a mapping is worthwhile. This article explores the world of graphs in computing and exposes situations in which graphical models are beneficial.

The Bits and Pieces of the Dots and Lines

A model is a representation of some aspect of reality. Many models can be thought of as a collection of objects, such as people or concepts, and the relationships that exist between them, such as friendships or subclasses. Such objects and relations form a network. Graphically, an object in a network can be denoted by a dot, and a relationship can be denoted by a line. A structure formed by dots and lines is known as a graph – the mathematical term for a network [1]. The most common type of graph is the simple graph. An example is diagrammed in Figure 1. In a simple graph there are a set of vertices (dots) and a set of edges (lines), where edges are undirected and connect two unique vertices (that is, there are no loops), and no two edges exist between the same pair of vertices.

Despite the title of this article, dots and lines are not the only components in a graph modeler’s toolkit. There are many more bits and pieces in the world of graphs. In practice, rarely are vertices and edges the only data...
FIGURE 2. There are numerous types of graphs. Many of the formalisms described can be mixed and matched in order to provide the modeler the expressivity necessary to capture the essential features of a domain.

contained within a graph. For instance, sometimes it is useful to have a name associated with a vertex or a weight and direction associated with an edge. From primitive dots and lines various bits and pieces can be added to yield a more flexible, more expressive graph. Figure 2 diagrams a collection of different graph types while a short summary of each graph type illustrated is provided below:

- **Half-edge graph**: a unary edge (i.e., an edge that “connects” one vertex). It has limited practical application and is primarily discussed in mathematics.
- **Multi-graph**: There are many situations in which it is desirable to have multiple edges between the same two vertices.
- **Simple graph**: the prototypical graph, where an edge connects two vertices and no loops are allowed.
- **Weighted graph**: used to represent strength of ties or transition probabilities.
- **Vertex-labeled graph**: Most every graph makes use of labeled vertices (e.g., an identifier).
- **Semantic graph**: used to model cognitive structures such as the relationship between concepts and the instances of those concepts [2]
- **Vertex-attributed**: used in applications where it is desirable to append non-relational metadata to a vertex
- **Edge-labeled graph**: used to denote the way in which two vertices are related (e.g., friendships, kinships, etc.)
- **Directed graph**: orders the vertices of an edge to denote edge orientation.
- **Hypergraph**: generalizes a binary edge whereby an edge connects an arbitrary number of vertices [3]
- **Undirected graph**: the typical graph that is used when the relationship is symmetric (e.g., friendship).
- **Resource description framework graph**: a graph standard developed by the World Wide Web consortium that denotes vertices and edges by uniform resource identifiers [4]
- **Edge-attributed graph**: used in applications where it is desirable to append non-relational metadata to an edge.
- **Pseudo graph**: used to denote a reflexive relationship.

Note that in many cases, these bits and pieces can be used in combination with one another, that is, they are not necessarily mutually exclusive. The list presented is not the complete space of all graph types, nor are the terms generally accepted in all domains. Many of these structures have been rediscovered in different domains and under different names. The important point is that there are numerous graph types and, consequently, there are systems and algorithms that exist to store and process them.

A common graph type supported by most graph systems is the directed, labeled, attributed, multi-graph – also known as a “property graph.” Graphs of this form allow for the representation of labeled vertices, labeled edges and attribute metadata (properties) for both vertices and edges. The property graph is common because modelers can express other types of graphs by simply abandoning or adding particular bits and pieces. For example, not allowing loops or multiple edges between two vertices generates a simple graph. Not allowing vertex/edge attributes generates a standard semantic graph. Restricting the vertex/edge labels to Uniform Resource Identifiers (URIs) generates a Resource Description Framework (RDF) graph (allowing for a few additional technicalities). Adding a weight attribute to an edge generates a weighted graph. The various graph types and the morphisms...
that yield one graph type from another are diagrammed in Figure 3. Note the location of the property graph within this diagram. Finally, while it is possible to model a hypergraph in a property graph, it comes at the expense of using vertices in the property graph to denote both vertices and edges in the hypergraph. For this reason, there are specialized hypergraph systems, such as HyperGraphDB. For the remainder of this article, the more common property graph and its supporting technologies are discussed.

FIGURE 3. The property graph is a convenient structure because it contains most of the bits and pieces used in graph modeling. Simple morphisms of the property graph yield other common graph structures. Thus, graph systems that support the property graph data model also, implicitly support other graph types.

Preserving Dots and Lines

The computer science community has recently seen an explosion of database technologies. For decades, the relational database of Codd’s relational algebra has been the primary storage and query mechanism for large data sets [5]. However, with the continued growth of data and an increasingly variegated application landscape, new databases have emerged. In this space, no database is seen as the single solution to all problems. Instead, each database attempts to solve a particular data management issue. Itemized below are short descriptions of recent database types:

- **Document database**: These databases have the “document” as their atomic entity. Such objects are semi-structured and usually represented in XML (Extended Markup Language) or JSON (JavaScript Object Notation). A document can be retrieved by means of pattern matching a query document (that is, a semi-populated document) against all the documents contained in the database. The benefit of this model is that these databases scale horizontally with relative ease. This ability is due to the fact that documents lack references between one another. The drawback is that data is not interrelated and thus, cross-database analyses are costly. For many web applications the document database is a very suitable solution that supports data scale and a convenient symmetry between the document structure and processing languages such as those that natively support XML and/or JSON. Examples of such databases include MongoDB and CouchDB.

- **Key/value store**: This family of databases is focused on scaling large amounts of data over a large number of machines and, in turn, supporting heavy read/write loads. Most of the databases in this class were inspired by Amazon’s Dynamo [6]. A popular open-source key/value store is Tokyo Cabinet.

- **Triple/quad store**: Triple/quad stores were developed to support the demands of the Semantic Web/Web of Data/Linked Data community. These databases are optimized for storing and querying data represented according to the Resource Description Framework (RDF) [4]. Typical use cases include description logic reasoning [7] and SPARQL-based graph pattern matching [8]. AllegroGraph is a high-performance quad store with a large suite of extensions and features.

- **Column store**: Most column stores are modeled after Google’s BigTable database [9]. A big table is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a time-stamp. Real-world services implemented with BigTable include GoogleAnalytics and GoogleEarth. Cassandra is a popular open-source column store.
**Graph database**: Graph databases are optimized for the efficient processing of dense, interrelated datasets. In these databases, the atomic entity is the graph as a whole. The typical data model is the property graph. By supporting the interrelation of data, graph databases allow for fast traversals along the edges between vertices [10]. A popular graph database of this form is Neo4j.

There are numerous databases in this growing space that were not mentioned. Moreover, there are other database types not mentioned. It is out of the scope of this article to explore this space in depth. The interested reader is directed to related discussions, blog posts and presentations that are made freely available on the Internet. Of particular relevance to this article are the graph database and the property graph data model. Figure 4 diagrams a property graph containing people, their articles and a university. In this particular domain model, each vertex has a name property and a type property. Edges denote both a directionality and a relationship type (that is, an edge label). Moreover, it is possible to also include properties on an edge to further refine the way in which two vertices are related (for example, Josh started attending RPI in 2007).

A consequence of the flexibility of a graph is that other related data can be represented as graph structures along with the domain model. A typical case of the use of such graph extensions is the endogenous index. An index is usually a tree-structure that allows for the fast look-up of elements within a collection. If there were no indices into a collection, then to determine if a particular element had a particular property, each element in the collection would have to be examined. Assuming that the cost of a linear scan of this kind is $O(n)$, where $n$ is the number of elements, an index provides the ability to partition the elements into increasingly fine-grained bins and thus to reduce the lookup cost to $O(\log_2 n)$ in most cases. While an index creates more data (the tree structure), it makes up for this cost by greatly increasing the speed of element retrieval. Figure 5 shows a name-property index over the example graph diagrammed in Figure 4. Together, the domain model and the index of the domain model are seen as a single atomic entity. Searching for an element and moving between elements are accomplished by a unified framework: the graph traversal.

**Jumping from Dot to Dot**

The first aspect of using a graph is creating a graph. Once a graph has
been created, it can be subjected to algorithms that quantify aspects of its structure, alter its structure or solve problems that are a function of its structure. At the root of any of these algorithms is the graph traversal [10]. A graph traversal is a walk along the elements of a graph – from vertex to edge to vertex, etc. As this walk proceeds, aspects of the graph can be saved or manipulated and in general, an algorithm can be computed. In principle, any of the data models and databases presented in the previous section (and including typical relational databases) can be used to represent and process a graph. However, when traversing a graph is the ultimate use case for a graph data set, then a graph database is the optimal solution, an import point. A graph database is optimized for graph traversals because elements (vertices and edges) maintain direct references to their adjacent elements. It is this design choice that makes traversing a graph structure within a graph database fast and efficient.

To get a better understanding of how graph traversals work, the examples in this section will be expressed in terms of a graph programming language called Gremlin. In Gremlin, moving over vertices and edges is analogous in many ways to moving through the directory structure of a local file system. To demonstrate, a naive friend-of-a-friend query is represented as follows:

```
./outE[@label='friend']/inV[outE[@label='friend']/inV
```

Reading from left to right, this expression states

- Start at the root vertex (., that is, the vertex to evaluate the expression on).
- Traverse to all the outgoing edges of the root vertex (/outE).
- Filter out all edges that are not labeled “friend” ([@label='friend']).
- For all those friend-labeled edges, go to their incoming/head vertices (/inV).
- For all the friends of the root vertex, get their outgoing edges (/outE).
- Filter out all edges that are not labeled “friend” ([@label='friend']).
- For all those friend-labeled edges, go to their incoming/head vertices (/inV).

At the end of this expression, the resultant vertices are the friends of the friends of the root vertex. Figure 6a diagrams the traversal, where the grey vertices are the returned vertices. This example is naive because in many cases, it is important to retrieve the root vertex’s friends of friends that are not also its friends. In such situations, the traverser must remember if a located friend-of-friend is not already a friend. In order to calculate the friend-of-a-friend, the friends must be determined first. Therefore, it is possible to save this information for later use. This idea is diagrammed in Figure 6b and the Gremlin expression is presented below, where the variable $x$ references the friends of the root vertex.

```
./outE[@label='friend']/inV[g:assign('$x')]/
    outE[@label='friend']/inV[g:except($x)]
```

An important aspect of working with property graphs is that the edges are typed/labeled. The standard suites of graph algorithms found in most graph/network-theory textbooks are not immediately useful for property graphs [11] because most graph algorithms have been developed for unlabeled graphs. When vertices can be related in many different ways and vertices can represent various types of objects, the meaning of the rankings, paths and other features returned by standard graph algorithms are ambiguous. However, by interpreting a path through a graph as an edge, it is possible to express standard graph algorithms on property graphs [12]. The previously presented Gremlin expression followed a path from the root vertex to its friends’ friends.
This path can be considered a virtual (that is, inferred, derived) edge. From the perspective of this expression, a new implicit graph is created over the graph’s vertices that only contains edges labeled “friend-of-a-friend.” This idea is diagrammed in Figure 7. As such, this virtual graph is equivalent to an unlabeled graph because all edges having the same meaning. Therefore, all the standard graph algorithms can be meaningfully applied to this derived graph – for example, the shortest path between person A and person B through their friends of friends. The benefit of edge-labeled graphs such as property graphs is that there are as many types of rankings, scorings and so forth as there are types of paths that exist between the elements of the graph.

Conclusion

The concept of a graph was introduced in the late 19th century. During the many decades that followed, the world of graphs was primarily left to the toiling of mathematicians. In the last few decades, the sociology, physics and computer science communities introduced a suite of algorithms and insightful realizations about the nature of graphs found in the real world. Moreover, the increasingly large volume of data made available by the Internet has yielded datasets that reflect the graphs found in our technological and social systems. To satiate the need to handle and process these large-scale graphs, graph databases have come to the forefront. To make use of the graphs beyond simply representing their explicit structure, graph traversal frameworks and algorithms have been developed in order to shape graphs by driving the evolution of the entities that they model – for example, humans and their relationships to one another and the objects of their world [13].

Resources Mentioned in the Article

Resources Mentioned in the Article, continued


Systems and Languages Mentioned in the Article

AllegroGraph: www.franz.com/agraph/allegrograph/
Cassandra: http://cassandra.apache.org/
CouchDB: http://couchdb.apache.org/
Gremlin: http://gremlin.tinkerpop.com/
HyperGraphDB: www.kobrix.com/hgdb.jsp
JSON: www.json.org
MongoDB: www.mongodb.org/
Neo4j: http://neo4j.org/
Tokyo Cabinet: http://1978th.net/tokycabinet/
XML: www.w3.org/XML