QUOGGLES: Query On Graphs – A Graphical Largely Extensible System

Paul Holleis[1] and Franz J. Brandenburg[2]

[1] University of Munich, 80333 Munich, Germany
[2] University of Passau, 94030 Passau, Germany

Abstract. We describe the query and data processing language *QUOGGLES* which is particularly designed for the application on graphs. It uses a pipeline-like technique known from command line processing, and composes its queries as directed acyclic graphs. The main focus is on the extensibility and the ease of use. The language permits queries that select a distinguished subgraph, e.g., the set of all green nodes with degree at least d or the set of edges whose endnodes have a neighbor which has exactly one neighbor. It is SQL complete, however, it cannot describe paths of arbitrary length; otherwise NP-hard problems like Hamilton path could directly be expressed. *QUOGGLES* also enables the user to concatenate queries with algorithms, e.g. with graph drawing algorithms, which are then applied to the selected subgraph.

1 Introduction

Graphs are frequently used to represent discrete data with objects as nodes and (binary) relations represented as edges. A relational database can be seen as a graph with n-ary relations, which can be modelled by hyperedges or by bipartite graphs. Often, a user has a special view on the data. In terms of graphs this means a distinguished subgraph, which is described by a collection of nodes, edges and attributes. This is particularly true for huge graphs such as the WWW or communication networks, from which the user selects a particular section.

This scenario coincides with the theme of Category C of the 10th Graph Drawing Contest 2002 [1]. The initiator Joe Marks had posted example graphs and wanted on-line answers on questions like "what is the largest wheel of green and blue nodes". At GD 2002, nobody could answer this question on-line. This was the starting point for quoggles ("QUeries On Graphs: A Graphical Largely Extensible System"), a plug-in for the graph visualization toolkit Gravisto developed at the University of Passau [2]. It is fully described in [3]. Gravisto associates graph elements with a hierarchy of attributes. These are addressed by queries and used for further computations like comparisons and sorting. The query language is capable of simulating relational algebra and SQL. However, it cannot express transitive closures and the existence of paths of arbitrary length. *QUOGGLES* itself is fully graphical and composes its queries in terms of directed acyclic graphs, which are automatically drawn by a simple algorithm.

Here we give a short description of the language and illustrate its use by some examples. For details we refer to [3] and [2].

J. Pach (Ed.): GD 2004, LNCS 3383, pp. 465–470, 2004.

2 Description of the Language

The query language of *QUOGGLES* consists of a set of fundamental operations which can be combined to form more complex operations. Every operation has i inputs, p parameters and o outputs. For maximal generality these numbers can depend on the values of the parameters. The resulting language has the full power of relational algebra and `SQL` (the proof can be found in [3]); however it cannot express paths of arbitrary length.

QUOGGLES is fully graphical. Every operation has a **box** as graphical representation as shown in Fig. 1. The box includes the name of the operation, values for its parameters and is numbered consecutively in the order of the creation within the query. It has i incoming lines on the left hand side for the input and o outgoing lines on the right for the output. These act as connection points to other boxes.

In the graphical representation, queries are composed from operations as directed acyclic graphs, combining inputs and outputs of boxes of operations in an appropriate way, possibly observing intermediate results using `Output` boxes. See, e.g., Fig. 2 for an example which is explained later in more detail.

For the evaluation of a query, the pipeline idea is used, which is well known from Unix command line and batch file processing. *QUOGGLES* applies and extends this general approach in information processing to graphs.

The set of graph elements from a graph acts as a source for each query. Data is processed as it flows through the pipelined operations. Since the notion of one single linear pipeline is quite restrictive, a directed acyclic graph can be built instead. It is constructed from operations with any finite number of inputs and outputs. Data flows along the edges of such a query graph. A query can then be evaluated using a topological ordering. A graphical user interface helps to create, change, execute and debug query graphs.

2.1 Basic Operations

In this section we describe a set of basic operations necessary to generate a sensible range of queries. This includes input operations, filters, general purpose and

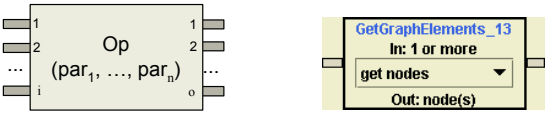
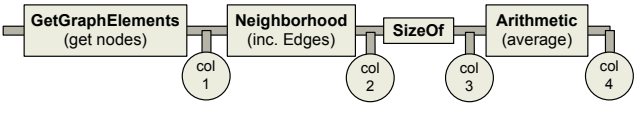


Fig. 1. Graphical representations of a general and a sample operation.



col 1	col 2	col 3	col 4
n1	[e4]	1	1
n2	[e1, e2]	2	
n3	[e3]	1	
n4	[ ]	0	

Fig. 2. This query computes the average degree of the nodes of a graph.

graph specific operations. Every operation receives one element or a collection of elements as input, checks and transforms the element(s) and outputs its result. Such elements can be nodes and edges, numbers and strings, e.g., nodes and edge labels, or tuples of such elements. In the implementation a collection is a list of Java objects. The following categories of operations are available:

Input Operations. Input boxes have no inputs and are used to create specific constants, such as text labels or numerical values or access external information like saved graphs.

Graph Specific Operations. For the navigation through graphs it suffices to provide an operation that accesses the neighborhood of a graph element and one that returns associated attribute(s):

The `Neighborhood` operation accesses elements in the graph theoretical neighborhood from input graph elements. Possible parameter values include *neighbors*, *incoming* or *outgoing edges* and *source* or *target nodes*.

The `GetAttributeValue` operation is used to retrieve attributed information from graph elements, such as node and edge identifiers or their labels. Graphical attributes like shape and size can also be queried.

General Query Operations. Most of the data processed by *QUOGGLES* is present as lists of elements. Hence, operations on collections are common. These include *flatten*, which converts a nested to a flat collection, *reverse*, which reverses lists and *make distinct*, which removes duplicates and *sort* for sorting a list using a string representation of the objects. The *union* and *intersection* boxes take two input lists and compute the set union and intersection, respectively.

Further general operations use the textual representation of elements in the input collection, count elements, compute the average, do arithmetic, comparisons and boolean operations or check the type of an input. Figure 3 shows the `CompareTwoValues` operation that compares its two inputs according to some specified relation.

There is a special `TwoSplitConnector` operation that duplicates the input and thus enables producing queries beyond that of simple linear pipelining.

Since it is not always clear that the sinks of the query graph are the (only) places that should contribute to the query result, `Output` boxes specify which part of the data present somewhere in the query pipeline should contribute to its result. They can also be used to check intermediate results.

Figure 2 shows an example query. The table on the right shows its output if the small graph displayed in Fig. 4 is used as input to the query. The data in the first column is retrieved from the first `output` box ('col 1'). It shows a list of

Fig. 3. Comparing two inputs using the <= relation and two different orders.

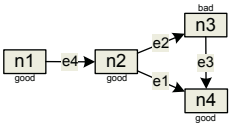
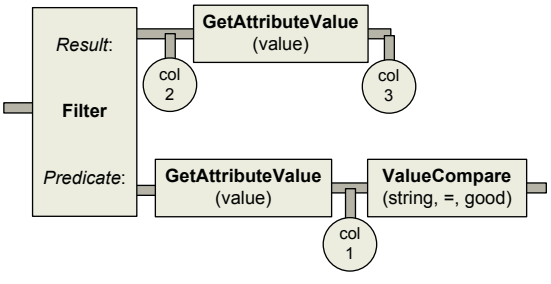


Fig. 4. A small graph used as sample input to queries.

all nodes of the graph. The `Neighborhood` operation produces a list that holds (lists of) all incident edges of the corresponding nodes. The third columns shows the sizes of these edge lists in the second column, i.e., the number of incident edges of each node. The `Arithmetic` box then computes the average degree of all nodes of the input graph ('col 4'). The selection of elements according to some predicate is the most frequently used operation in database systems. Here, the `Filter` operation retrieves those objects from an input collection of arbitrary objects that meet the condition specified by a predicate subquery. The predicate can be an arbitrary query. Its output will be interpreted as a boolean value. Empty collections or the value zero will for example be converted to *false*.

Figure 5 shows a query that filters all graph elements that have an attribute called *value* with value equal to *good*. The table on the right shows the result for the example graph displayed in Fig. 4. Since edges do not have such an attribute, column one displays a "-" for them. Nodes *n1, n2, n4* match the criterium (which can be verified by examining their attribute value in column three).



col 1	col 2	col 3
good	n1	good
-	n2	good
bad	n4	good
-		
-		
good		
good		
-		

Fig. 5. Find all nodes from the small graph on the left that have outgoing edges.

To ensure reusability of queries and a reasonable size of queries, it is important that any query or part thereof can be saved as a subquery for later use. This is shown in Fig. 6 where the calculation of the average node degree of a graph is saved. This subquery can then be used to get all nodes that have a degree larger than average. The query and the result ('col 3') is shown in Fig. 7.

3 Application in Graph Drawing

Good layouts of graphs can often not be achieved by generic graph drawing algorithms alone. A certain degree of interactivity can be necessary or at least helpful. However, tweaking algorithms for those special uses can be time con-

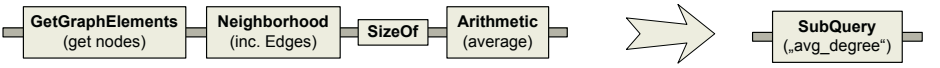


Fig. 6. The query on the left is saved and can later be used in a `Subquery` box.

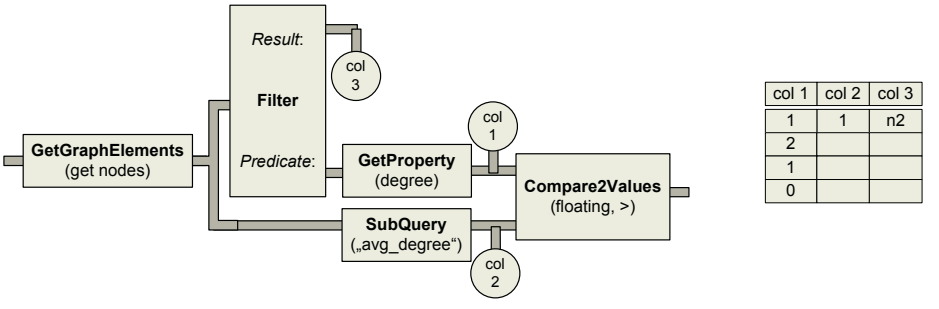


Fig. 7. Using a `Subquery` box, nodes with high degree (*n2*) can easily be found.

suming or even impossible if third-party programs are used. *QUOGGLES* enables the user to layout different parts of the graph differently.

Subgraphs that should be drawn in some special way can be retrieved by using queries. Then these sets can be further processed by applying layout algorithms on them. This renders it extremely easy to, e.g., quickly test which type of centrality best serves to find a good drawing. Nodes with a high centrality value can be drawn, e.g., more central than others.

Figure 8 shows a generic example query that finds a certain set of 'important' nodes. This might be done, e.g., by using the query shown in Fig. 7 as a subquery. This set is drawn using a spring embedder algorithm ('spring') with a small value as parameter ('10') indicating that nodes will be close together. All other nodes (retrieved using the *set minus* operation) are drawn using an algorithm that places nodes on a circle with a rather large radius ('75'). The algorithms directly work on the data structure. The result of the query (as specified by the circular box titled 'col 1') is the set of special nodes. This helps to manually adjust the relative placement of the two layouted subgraphs. This query has been applied to a random placement of nodes of a small graph producing the layout shown in Fig. 8.

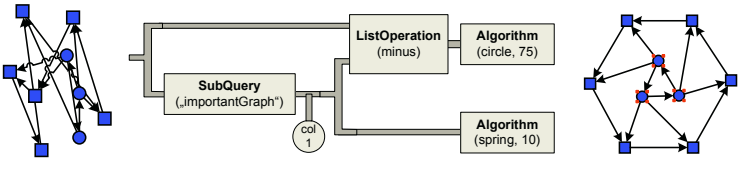


Fig. 8. A query used to apply two layout algorithms to different parts of the graph.

4 Conclusion

The *QUOGGLES* system is an implementation of a query language specifically designed to retrieve information from graphs. It combines general and graph specific operations using an extended pipeline principle. It can been shown that the system is relational complete and even provides similar functionality as SQL 92. An intuitive user interface is provided. Its extensibility renders it especially useful for semi-automatic graph processing. As an example, we showed how to apply its feature to include algorithms in query processing to address layouting and graph drawing problems. Fig. 9 shows a screen dump of the system.

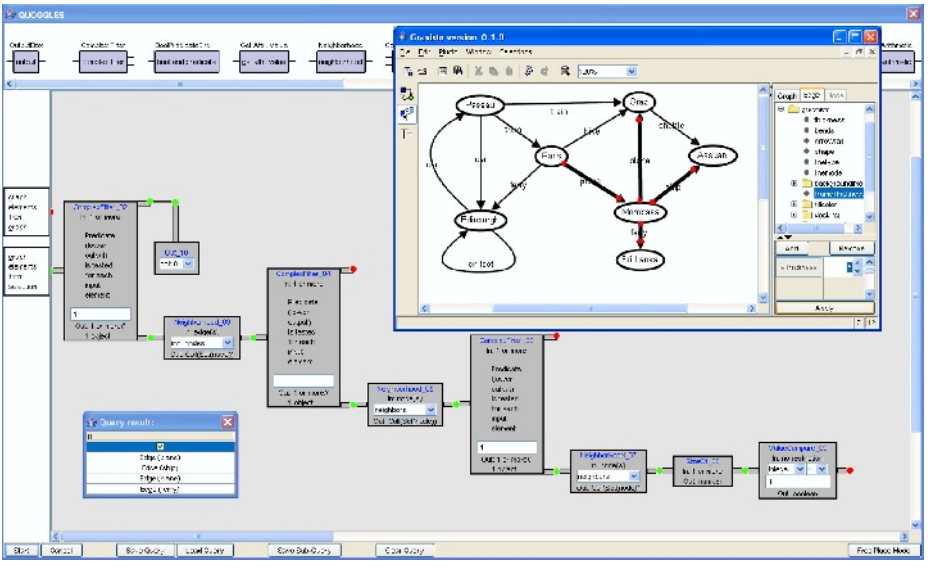


Fig. 9. A screen dump of Gravisto with an example graph and the *QUOGGLES* system after executing the introductory query: "Get the set of edges whose endnodes have a neighbor which has exactly one neighbor".

References

1. Brandenburg, F.J.: Graph-drawing contest report. Proceedings Graph Drawing 2002, LNCS 2528 (2002) 376–379
2. University of Passau: Gravisto. http://www.gravisto.org/ (2002)
3. Holleis, P.: Design and implementation of an extensible query language on graphs. Diploma thesis (2004)