Diplomarbeit

Design and Implementation of an Extensible Query System for Graphs

Paul Holleis
Universität Passau
12. Januar 2004

Erstgutachter: Prof. Dr. Franz J. Brandenburg
Zweitgutachter: Prof. Dr. Ulrik Brandes

Lehrstuhl für Informatik
Fakultät für Mathematik und Informatik
Universität Passau

Abstract

Many problems in computer science and in other fields can be expressed as the need to retrieve information from data. Often, the data is discrete and can be represented as a graph. The system presented in this thesis allows the user to enter a query - in a graphical and user friendly way - that will extract information from such a graph. The result is displayed in form of subsets of graph elements, tables of arbitrary objects or by further attributing the graph.

A large part of the work concentrates on capabilities and restrictions of this query language. It is shown that all operations from relational algebra as well as `SQL` queries can be simulated. The language is not Turing complete, but can be extended easily.

The strength of the system lies in its powerful capabilities at the same time offering a quick and intuitive way of formulating queries.

It is built as a plug-in for a new graph editor called *Gravisto*. Care was taken to separate the system as much as possible to allow quick adaption to other systems.

Acknowledgements

It goes without saying that I owe a dept of gratitude to Professor Dr. Brandenburg who accepted my proposals for the subject of this thesis and even sacrificed some of his spare time to have a look at it. I am also grateful to Professor Dr. Brandes, who took on the task to review the thesis as well and showed some interest in it from the start.

Many suggestions and constructive criticism has been provided by the assistants of the chair for theoretical computer science. Especially Mr. Michael Forster has never hesitated to clarify misunderstandings, avoid fruitless discussions and find solutions that satisfy everyone.

My reserved attitude towards the typesetting program LaTeX 2_ε was tremendously soothed by Mr. Richard Kuntschke, who with only few well-chosen words, shared a lot of his knowledge with me. Mr. Thomas Zimmermann helped me not to loose track of the overall structure and never stopped offering to proofread the thesis.

Special thanks deserves Ms. Corinna Seitz who spent much time finding errors, identifying uncomprehensible sentences and encouraged me more than once *not* to throw my computer out of the eighth floor.

The typesetting program LaTeX 2_ε has been used in conjunction with the XEmacs text editor to write this thesis. All figures have been created using Microsoft Visio Professional 2002 or via a screen shot utility. TogetherSoft Together 6.0 helped designing `UML` diagrams.

The *QUOGGLES* system has been implemented in the programming language Java, using the JDK 1.4.0 of Sun Microsystems, Inc. Eclipse has been used as a programming environment.

A CD-ROM is attached to this thesis that contains the implementation of the *QUOGGLES* system, many example queries and the example graphs mentioned in the text. The *Gravisto* system ([Pas02]) in its version from January 9, 2004, is also included on the CD-ROM.

Contents

Chapter 1

Introduction / Motivation

1.1 The Need for Knowledge

Not too long ago, the major problem in handling larger amounts of data has been the shortage of storage space. These days, even the smallest hand-held computers can save an amount of information for which whole rooms had to be filled in the past. One of the consequences of this phenomenon is that the focus in research has shifted slightly and concentrates more on the efficient and reliable retrieval of information stored in some sort of database.

Graph data structures are used in many research areas including geographical systems and social networks. The central tasks of the latter is to identify nodes that seem to be more important than others in the same graph according to some characteristics.

With the advance of `XML`, `RDF`, `ODMG` and other structures based on graphs but nevertheless describing inherently different data, more and more branches of scientific research are in need of systems collecting and extracting data from large sources. Hypertext documents, the World Wide Web itself, multimedia content and e-business applications all are modelled or draw on graph structures.

Many applications using graph data and graph algorithms can also be found in the field of bio-informatics concerning visualization and querying of large graphs (showing reactivity of certain proteins and molecules, for instance). New projects are started on a regular basis (for example [Pro03]) and numbers of biological databases (like [Gro03], [ABa03]) are on the increase. An example graph has even been used for the Graph Drawing Contest at the Graph Drawing Conference 2003 ([OMMA01]).

This need for knowledge has also inspired the organizers of the Graph Drawing Conference 2002: Although it may have been a little bit disappointing for Joe Marks (MERL, Cambridge), who in 2002 had the interesting and promising idea to organize an interactive contest at the annual Graph Drawing Conference. The fastest way to solve small riddles on medium sized graphs should have been found. Unfortunately, the number of spectators exceeded the number of participants by 100 percent.

The idea has been to motivate people to come up with ways or tools to quickly extract information from a graph. Possible attempts might have included pre-computation, visualization, animation or special querying methods. This thesis tries to follow the last approach, designing an easy-to-

use query language.

As the duration of the conference dinner imposes a limit on the time available to solve the problems, the design goals include a quick way to create queries. The type of problems being probably as manifold as the menu, the language must be powerful enough to express complex queries, at the same time being simple and of manageable size. Since unforseen tasks should best have been foreseen, a most important factor in the design of any system has been the ease of extending it. An experienced and fast programmer should be able to add desired routines as easily and quickly as possible.

This thesis shows that the call to participate at this interactive contest has produced at least some hundred pages of thoughts in this thesis only.

The program presented in the following chapters fulfils all mentioned requirements and provides a graphical, largely extensible system for queries on graphs.

The structure of the thesis is as follows:

Chapter 1: Several terms are defined that will be used throughout the rest of the thesis. This includes graph theoretical definitions, terms used for the implementation and a description of the *Gravisto* system used as a graph visualization and editor toolkit.

Chapter 2: An overview of the features of the system is given, followed by an illustrated example.

Chapter 3: The query language itself is presented. The chapter splits into the presentation of basic boxes that represent the basic components of the language and compound boxes that represent operations that can be simulated using basic boxes. Several examples of how to use the query language can be found in this chapter.

Chapter 4: The power of the language is examined in detail in the fourth chapter. It is proven that the language is relational complete and it is shown how `SQL` queries can be translated into the query language. The chapter is concluded with the statement that the language is not Turing-complete.

Chapter 5: This chapter describes possible further work and hints at query optimization and application of the system to a whole database of graphs. It also shows one possible direction in which the *QUOGGLES* system could be extended in Section 5.1.3. After the the system is compared to other approaches in similar domains.

Chapter 6: A brief summary of the thesis is given in the last chapter.

Appendix A: Implementation details are presented in the first part of the appendix. The way of how to extend the system is particularly emphasized.

Appendix B: A concise manual of the *QUOGGLES* system can be found in the second part of the appendix.

1.2 Preliminaries

This section concentrates on the definition and presentation of terms and concepts used throughout the thesis. Readers familiar with graphs as data structures can skip Section 1.2.1. After a short treatment of what and how information can be stored using graphs in Section 1.2.2, the *Gravisto* system is introduced as far as it is necessary for its treatment in this thesis. For this, the reader is supposed to be acquainted with the basics of object-oriented design and know about concepts like classes and interfaces.

1.2.1 Graph Theoretic Terms

The data structure for which the *QUOGGLES* system provides a query language is called a graph. This structure and several important terms concerning graphs are defined next. For a more detailed description see the appropriate chapters of any book on data structures, for example the exhaustive treatment in [AHU87], [OW02], [Jun99] and [Meh84] or [CH94].

Definition 1.1 (Undirected Graph)

An **undirected graph** $\widehat{G} = (N, E)$ is a finite non-empty set N of objects called **nodes** (also called **vertices**) together with a (possibly empty) set E of unordered pairs $\{n_1, n_2\}$ of distinct nodes of N called **(undirected) edges**. □

Definition 1.2 (Directed Graph, Digraph)

A **directed graph** $\overrightarrow{G} = (N, E)$, also called a **digraph** for short, is a finite non-empty set N of objects called **nodes** (also called **vertices**) together with a (possibly empty) set E of ordered pairs (n_1, n_2) of distinct nodes of V called **(directed) edges** (or **arcs**). □

Definition 1.3 (Graph, Graph Element)

A **graph** is either a directed or an undirected graph. Following the *Gravisto* data structure, the set of edges E of a graph can also contain directed *and* undirected edges at the same time.

The union of the set of nodes and the set of edges is called the set of **graph elements**. □

These definitions do not allow multiple edges (since, for a graph $G = (N, E)$, E is a set).

Definition 1.4 (Simple Graph)

A graph $G = (N, E)$ is **simple** if no edge is a pair of the form (n, n) or $\{n, n\}$ (such edges are called often self-loops). □

In this thesis, all graphs are considered to be simple unless stated otherwise.

Definition 1.5 (Out-Edge, In-Edge, Source Node, Target Node)

In a graph $G = (N, E)$, an edge $e = (n, n') \in E$ is an **out-edge** (also called outgoing edge) of node $n \in N$. It is an **in-edge** (also called incoming edge) of node $n' \in N$.

For an edge $e = (n, n') \in E$, n is the **source node** and n' the **target node** of e. □

For undirected graphs, these terms are undefined.

Definition 1.6 (Incident Node / Edge)

In a digraph $\overrightarrow{G} = (N, E)$, a node $n \in N$ is **incident to an edge** $e \in E$ if e is an out-edge or an in-edge of n.

In an undirected graph $\widehat{G} = (N, E)$, a node $n \in N$ is **incident to an edge** $e \in E$ if a node $n\prime \in N$ exists such that $e = \{n, n'\}$.

In a graph $G = (N, E)$, a node $n \in N$ is **incident to an edge** $e \in E$ if one of the previous two conditions holds for n.

In a graph $G = (N, E)$, an edge $e \in E$ is **incident to a node** $n \in N$ if n is incident to edge e. □

Definition 1.7 (Out-Neighbor, In-Neighbor, Neighbor)

In a digraph $\overrightarrow{G} = (N, E)$, a node n' is an **out-neighbor** of a node n, if there exists an edge $e = (n, n') \in E$. A node n is an **in-neighbor** of a node n', if there exists an edge $e = (n, n') \in E$.

In a graph $G = (N, E)$, a node n' is a **neighbor** of a node n, if there exists an edge $e \in E$ and a node $n' \in N$ for which hold one of the following conditions:

- $e = (n, n')$
- $e = (n', n)$
- $e = \{n, n'\}$ □

Definition 1.8 (In-Degree, Out-Degree)

In a digraph $\overrightarrow{G} = (N, E)$, the **in-degree** / **out-degree** of a node $n \in N$ is the number of in-edges / out-edges of n.

In a graph $G = (N; E)$, the **in-degree** / **out-degree** of a node $n \in N$ is the number of in-edges / out-edges of n plus the number of undirected edges incident to n. □

For undirected graphs, these terms are undefined.

Definition 1.9 (Degree)

In a graph $G = (N, E)$, the degree of a node $n \in N$ is the number of edges incident to n. □

Thus, for a digraph, the degree is the sum of in- and out-degree.

The query graphs created in the system will not support recursion. Therefore, there will not be any cycles in those graphs:

Definition 1.10 (Path, Cycle, Acyclic Graph)

In a graph $G = (N, E)$, a **directed path** of length l is a sequence of directed edges

$$[(n_1, n_2), (n_2, n_3) \ldots, (n_{l-1}, n_l)]$$

where an edge has the same target node as the source node of the subsequent edge.

An **undirected path** uses only undirected edges and a **path** in general may use directed and undirected edges. Directed edges must always be traversed from their source to their target node.

A **circle** is either a directed path where the source node of the first edge coincides with the target node of the last edge or an undirected path where each node appears exactly twice.

A graph that does not contain any cycle is said to be **acyclic**. □

1.2.2 Information in Graphs

A graph is a structure widely used to represent a number of distinct entities and their relations. Such information can always also be stored in many other ways like for example tables. However, the standard graphical representation of a graph makes it much easier for a human to grasp the content expressed by this structure. Nodes are displayed as small rectangles or circles while edges connect their source and target node by lines.

Because of them being very useful for displaying information, graphs appear in the most specialized research areas as well as in every day life. Bio-chemical analysts use them to show how molecules and substances react with each other in transcriptional regulation networks (as in [OMMA01]). On the other hand, probably everybody has already come across some type of graph: The relations between people on a genealogical table, the stations of a subway network or the matching of teams in a competition all are mainly displayed as some sort of graph. Even the World Wide Web, any intranet or organizational structures can be seen as graphs.

In small graphs, it is relatively easy to present detailed information as well as give a good idea of its overall structure. Figure 2.2 shown an example of that. With a growing number of nodes and edges, the layout of the graph gets more and more complicated. Since nodes have some minimum size, the person displaying the graph soon has to decide whether to concentrate on information local to nodes and edges or to emphasize the structure and type of graph. Imagine a graph visualizing the telecommunication network in the United States of America. No drawing of size less than a few square meters could show names and address of all stations. It is more interesting to see the extension of the network and identify regions of low coverage.

This means that there are two types of information contained within graphs: Local attributes, associated with nodes and edges making specific statements of the use and meaning of these element are considered details. All of them can only be sensibly displayed in small graphs. Structural information needs are more distant view of the graph. The way nodes are connected via edges can reveal important properties of the graph. This includes geometric properties that use information at the nodes and edges as does the telecommunication network. A graph may exhibit certain features important for the application that uses it. Special layout algorithms can only operate on trees, bipartite graphs show the existence of a sensible partition of their nodes.

Graph theoretical experts can find nodes that seem to be more important than others (in fact, much effort is put into the research of network analysis for larger graphs).

When visualizing the latter characteristics of a graph, detailed information located as attributes at nodes and edges can hardly be made visible. This thesis suggests the following approach: Display the whole graph in one window. Special knowledge or findings help to show the global structure. When needing local information, it is rarely the case that all attributes of all graph elements should be displayed as this would require too much space. The purpose of the system presented in the chapters that follow is to present a means to query the graph. This extracts information that can then be shown using multiple strategies. If, for instance, the phone number of a specific telecommunication company's headquarter should be retrieved, the corresponding node can be identified and this attribute displayed. If the whole subgraph of the telecommunication network of one single company is the result of the query, all corresponding nodes and edges can be highlighted in the graph, the number of nodes and edges can be calculated and so on.

Thus, while keeping the general idea of the structure of the graph, details can be extracted and visualized.

The system presented in this paper is especially suitable for retrieving local information or sets of graph elements that match certain criteria concerning their attributes and the graph theoretical vicinity. Other approaches exist (for example [SÖÖ99], [Kar03]), that are optimized for queries using paths (i.e. sequences of edges) as base elements.

The main part of this thesis introduces a query language. Its implementation is a plug-in for the *Gravisto* system which is briefly presented in the next section.

1.2.3 The Gravisto System

The *Gravisto* system has evolved from a larger programming practical ("Projektstudium") at the University of Passau in the academic year 2001/02. A team of six students (including the author of this thesis) carefully designed it to, in the long run, replace the *Graphlet* system ([Pas90]), started also at the University of Passau. *Graphlet* is a toolkit for graph editors and graph algorithms, drawing from the `GTL`, a `C++` library for working with graph data structures. The program has, however, undergone several changes and additions in the last decade which rendered it hard to use for developers other than those who designed it. Thus, a new project was initiated.

The name *Gravisto* is short for "Graph Visualization Toolkit" ([Pas02]). Written in Java, extensibility as one of its major design goals and a clean data structure make it a powerful tool to create, edit, visualize and work with graphs.

The Graph Data Structure

For detailed information about the design of the system, refer to the online and JavaDoc documentation of *Gravisto* that can be found in [Pas02].

The `UML` diagram in Figure 1.1 shows the design of the graph data structure that is used in the *Gravisto* system and shall be briefly presented next.

The design mirrors the structure of graphs, providing interfaces `Graph` and `GraphElement`. Since edges and nodes are graph elements (cf. Definition 1.3), interfaces `Node` and `Edge` are introduced, extending `GraphElement`. For all those interfaces exist abstract classes that provide default implementations for most of the methods. The *Gravisto* system ships with an adjacent list

implementation of graph, using the classes `AdjListNode`, `AdjListEdge` and `AdjListGraph`. A slightly more optimized version of the graph class can be found in `OptAdjListGraph`. Other implementations based on different structures are expected to be written in the future.

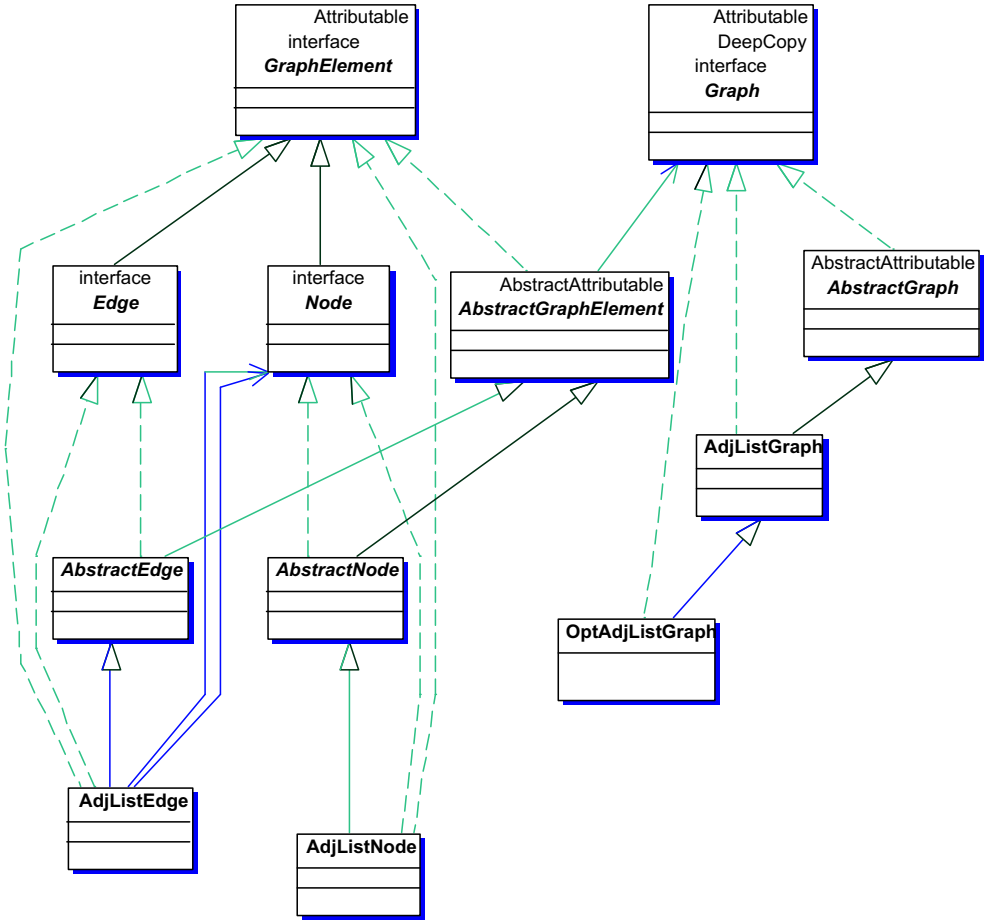


Figure 1.1: Graph Data Structure

One additional common property of graphs and graph elements in the *Gravisto* system must be stressed: Both, `Graph` and `GraphElement` extend the interface `Attributable`. The design of attributes is very powerful and one of the reasons why *Gravisto* is so extensible. Since this apparatus is not needed in its whole extension for the purpose of this thesis, only some important aspects shall be highlighted:

To separate the core graph data structure from any other information, graph and graph element interfaces are kept as small as possible. The interface `Attributable` has been designed to enable objects to hold data in form of attributes. A graph might hold its creation date, a small description of its contents and the names of its authors as attributes. Sensible information to be stored at nodes or edges include labels and graphical descriptions like shape or color. For another approach of associating information with graphs, see [MN99].

To be able to hold this variety of data types, a generic interface `Attribute` and two different types implementing it have been introduced: On the left hand side of Figure 1.2, basic attributes for all primitive types that Java provides can be seen: `StringAttribute`, `IntegerAttribute`, etc. The `CompositeAttribute` serves as base class for special user-defined attributes. All these

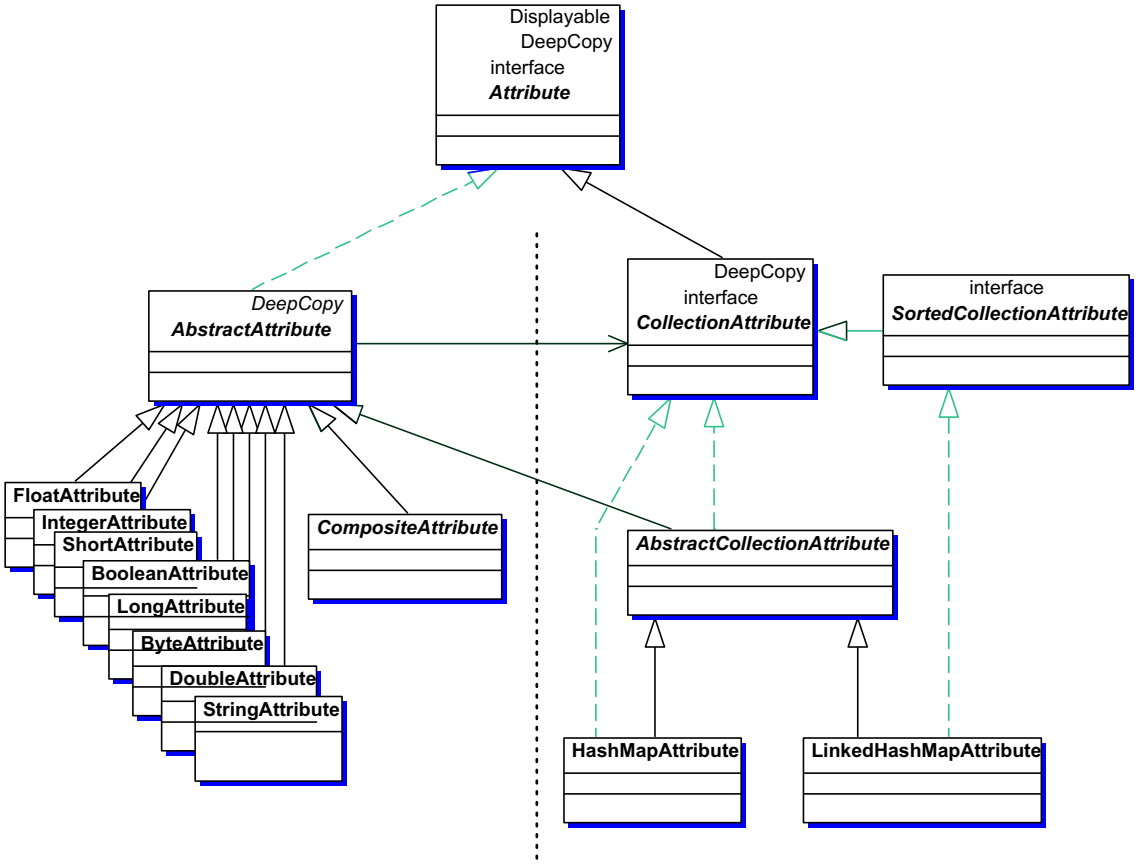


Figure 1.2: Attribute Design

extend `AbstractAttribute` that provides default implementations of most methods as well as some auxiliary functions.

The classes and interfaces to the right of the vertical dotted line in Figure 1.2 are used to create an attribute hierarchy. `CollectionAttribute`s can hold basic attributes as well as other `CollectionAttributes`. Two implementations are provided, building on a normal and a linked list implementation of a hash map. This system enables the user to attach a whole tree of attributes to an `Attributable`.

All `Attribute`s have an id by which they can be identified. To every attribute in the attribute tree, there exists exactly one path from the root (which has the empty string as id) to the attribute. The **path** of an attribute is the concatenation of all ids of attributes on that path, in that order, separated by a special character. Assuming this character is the dot ".", a valid path would be "graphics.coordinates.x". In the standard *Gravisto* system, this describes a `DoubleAttribute` holding the horizontal coordinate of a node in the graphical display.

The linked list implementation of `CollectionAttribute` is necessary for attributes that must be saved in a particular order like the control points of a spline representation of an edge. Important for the implementation of the *QUOGGLES* system is that *Gravisto* bases much of its functionality on plug-ins. This means that developers can write their own tailored-made attributes, tools, visualization components, algorithms and much more, adding them to the system at run-time without having to recompile the whole program. Appendix A provides more information about that.

1.2.4 Technical Terms

This section concisely describes a few technical terms used throughout this thesis. Most of them denote Java specific classes and names that are used implementing more general concepts ([SM02]). Readers familiar with the Java programming language might want to skip this section.

`Collection`: The generic term to describe any group of objects, known as its *elements*. This includes sets and lists.

`Iterator`: Every `Collection` defines an `Iterator` that can be used to access each element. If the `Collection` defines an order on its elements (as do lists, for instance), the `Iterator` reflects this order. For unordered `Collection`s, the `Iterator` returns the elements in no particular order.

`Number`: The generic term (in the language of object orientated programming called the super class) of all objects representing any kind of number. This may be integers as well as roman numerals. All numbers provide methods to get their value in form of a whole or floating point number. This allows arithmetic do be done.

`String`: A sequence of arbitrary length consisting of arbitrary characters. In this thesis, occurrences of `String`s are written enclosed in quotation marks as in "`this is a string`". Therefore, two objects "`12`" and `12` are completely different. The first denotes a `String`, the second a `Number`. There exist several methods to convert `String`s to `Number`s and vice versa, though.

`Boolean`: An object that can has only two states: It can either represent one of the boolean values true or false.

`Combo Box`: A combination of a text field an a drop down box. The user can choose from a list of predefined values or (if the combo box is editable) enter some text. Frequently used in graphical user interfaces.

Chapter 2

Overview

2.1 Description

The pipeline system is well known. In the world of the Unix operation system and its derivatives, it is heavily used in bash processing. It is applied in the following scenarios: Some data is produced by an operation o_1. This data shall directly be processed by an operation o_2 which, in turn, passes its output to operation o_3 that produces the final result. An example would be the task of finding the occurrences of the word "hidden" in all files in the current directory:

```
cat * | grep "hidden" | less
```

The first operation `cat *` concatenates all contents of all files. This text is passed to the `grep "hidden"` command that filters those lines that contain the desired string. `less` shows the output and lets the user scroll through it. This system saves the work to save intermediate results that are not interesting to the user.

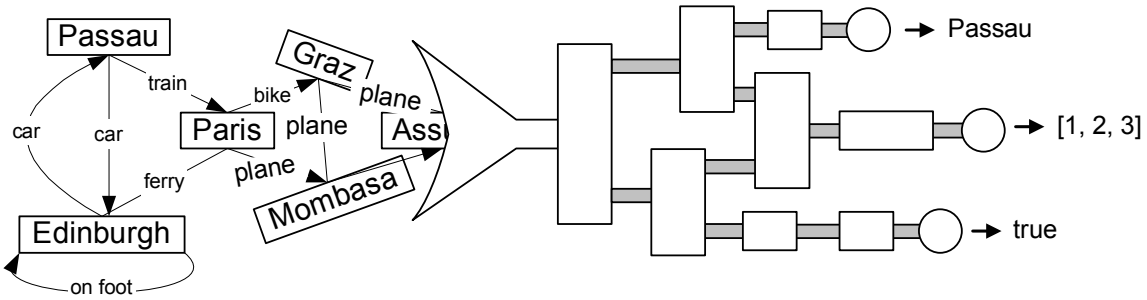


Figure 2.1: The Idea of the System

The basic idea of the *QUOGGLES* system is to apply this general approach in information processing to graphs. Simple but powerful operations are designed. The set of graph elements from the graph on which the query is run act as source. Data then flows through the pipeline and is processed by operations called boxes. Since the notion of one single pipeline is quite restrictive, a directed acyclic graph is used instead. It consists of the boxes that represent operations with an arbitrary (finite) number of inputs and outputs. Data then flows along the edges of the query graph.

A graphical user interface helps the user create, change and execute query graphs.

The result of a simple pipelined sequence of operations is the output of the last operation. Using graphs as an extension of the pipeline principle, several outputs are possible. The user is not always interested in all of them. Some intermediate results, on the other hand, might be important as well. Thus, some way must be provided for the user to specify which outputs of which boxes in the query graph should be observed: A special type of box that can be connected to all outputs of all boxes is introduced solely for that purpose. It records the data that passes it.

The final outcome of the query is displayed in a result table. Each box of that special type contributes to one column in that table. Thus, there is no need for a relationship of cells of one row.[1]

In the result table, any cell or column can be selected. Graph elements displayed in such a cell or column are highlighted in the graph editor's view, improving the use and interpretation of the query results.

Most of this thesis will be concerned with introducing the query language and examine its possibilities. The whole system is referred to as the *QUOGGLES* system. The name is short for

Queries **O**n **G**raphs: A **G**raphcial **L**argely **E**xtensible **S**ystem

This name reflects the core aspects of the system.

Everything described in this thesis has been implemented using the programming language Java. The implementation is realized as a plug-in for the *Gravisto* system described in Section 1.2.3. The CD-ROM accompanying the thesis contains this system in the version from January 10, 2004. The latest release can always be found at [Pas02]. Most example queries and graphs mentioned in the text are also included on the CD-ROM.

2.1.1 Main Features of the System

- Operate on the graph data structure of the *Gravisto* system and at the same time staying general enough to cope with different data structures.

  The *Gravisto* system has been introduced in Section 1.2.3. The *QUOGGLES* system is designed to use and access the data structure presented there. Except from the modelling of graph elements and attributes, the implementation draws only on few essential features (like selections). Otherwise, that would make it impossible to use the system in different environments. A stand-alone version is also feasible that fetches its input graphs from some different source than the graph editor.

- Supply the user with a graphical interface to enter the query.

  One of the advantages of the *QUOGGLES* system is that the constructs of the language are visually represented and can be easily accessed. The user can click on single components and create a query using the mouse only.

- Provide the possibility to load and save queries and subqueries.

  Important for the work with the system is that queries do not have to be recreated from scratch every time. Thus, some way to serialize and parse queries must be provided.

[1] Some operations preserve some linkage between their input and their output, though. This means that some columns can be related to others. Chapter 4 provides more detail about that phenomenon.

Since queries are themselves graphs, an extended version of the GML graph data description language is used ([MH95]).

- Careful design of elements / operations of which a query consists.

  The invention of a new language always places a strong emphasis on a careful design of its components. They must be simple enough to be the basic constructs and at the same time complex enough to provide the language with enough power to express a minimum of queries. All components have been extensively tested for their necessity and usefulness.

- Automatic check whether the query graph is valid (design- and run-time). It is especially important for a sensible graphical user interface to show whenever (and where) a query graph is not valid. The design-time check includes the visualization of misplaced boxes while the run-time check finds out about run-time type errors. Helpful error messages simplify the search for errors.

- Any (intermediate) result can be used as input to another query.

  All data that is present anywhere in the query graph can be displayed and used for further processing. This is especially useful for debugging purposes.

- Ease of extending the system (i.e. introduce more powerful, specialized or optimized operations).

  The system as it is provides means to solve a remarkably large amount of problems. There is much room for extensions. This may mean specializations (i.e. providing full-fledged operations for simple tasks) and optimizations but also the introduction of more powerful paradigms (see the discussion on features not yet implemented in Section 2.1.2).

- Information exchange between graph editor and query system.

  Information provided by a graph editor (for example via a selection) is accessible by the query system. In addition to that, (intermediate) results from the *QUOGGLES* system can be displayed (graph elements resulting from a query can be marked in the result table and are highlighted in the graph editor).

2.1.2 Features Remaining Future Work

- Optimize queries and their execution.

  Optimization is an interesting and important processing step. Unfortunately, it is a very broad and time-consuming field of research. Even for concepts like relational algebra with only a small set of operations and no extensions, much work has been invested (see [KE01] for a basic treatment and an extensive list of references on this subject). It needs even more effort to examine systems like the *QUOGGLES* system with many operations and nearly unlimited opportunities to extends the language. Therefore, apart from some comments in Section 5.1.1, a more detailed discussion is left out.

- Introduce recursion / loops

  The introduction of recursion outside the box operators leads to several issues: Although the language would become more powerful, much of the plainness of the queries would vanish. The query graphs would be much harder to layout and definitely more complicated to understand. It would no longer be possible to guarantee termination.

Developers can make use of recursion inside boxes, though. This seems to be powerful enough without complicating the system too much. Nevertheless, the ongoing discussions on the new SQL 3 standard (which provides recursion techniques) ([KE01] or [Mel96]) has shown that recursion still is worth examination.

- Optimize query execution for several graphs (preprocessing, fingerprints, etc.)

  The *QUOGGLES* system is designed to execute a query on one single graph only. It does not constitute a major task to sequentially execute the same query on several graphs. Several approaches exist, however, that can cope with querying whole databases of graphs much better. These heavily use preprocessing and fingerprints to be able to quickly reduce the number of candidates. Any work in this direction is neglected in this thesis (but cf. Section 5.1.2).

- Introduce an extensive help system

  Although most of the operators and the general use of the *QUOGGLES* system is easy to understand, a context-sensitive help system would be highly appreciated. Especially the display of exact definitions of operations and step-by-step instructions on how to eliminate error messages would simplify the use. Such a system is much work, though, that does not contribute to the theoretical value of the program and only pays off if the *QUOGGLES* system is going to be used by a broader public.

2.2 Introductory Example

Figure 2.2 shows a graph consisting of six nodes and eleven edges. It is (a very small and artificial) part of a graph showing large cities in the world and the possible traffic connections between them. Thus, Graz can be reached from Passau by train, while the only direct way from Paris to Mombasa is by plane. Passau and Assuan are not directly connected, but there exists a path via Graz. Edges without arrows describe bi-directional connections. A plane can thus move freely between Graz and Mombasa, while Paris does not allow any incoming planes.

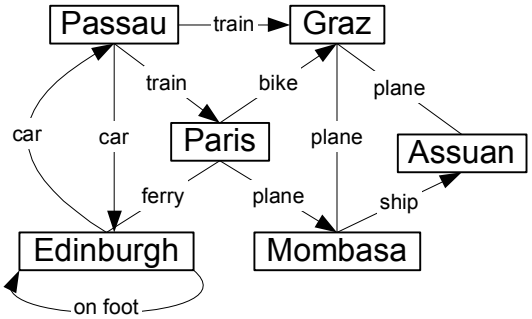


Figure 2.2: Example Graph (City Connections)

What information can now be extracted from this graph?

Example 2.1: Some readers might wonder what possibilities they would have to leave Passau if they decided to pay the university for which this thesis has been created a visit. Assume the node labelled "Passau" to be the input to the query system. Queries in the *QUOGGLES* system are graphs consisting of boxes. One such graph is shown in Figure 2.3.

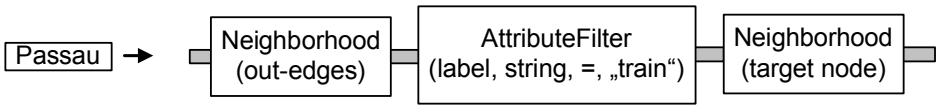


Figure 2.3: Example Query Graph 1 a)

As is illustrated, the node "Passau" is the input to a query graph that resembles a pipeline with three dilations (called boxes). Intuitively speaking, the input node will enter the pipeline to the left hand side, get transformed on the way and some information will exit to the right. To see what actually happens, the *QUOGGLES* system provides means to specify points in the query graph where the data that is currently flowing at that point is displayed.

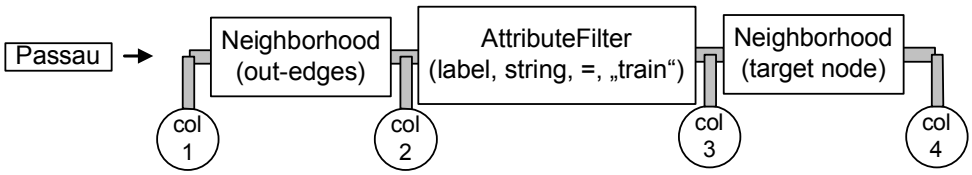


Figure 2.4: Example Query Graph 1 b)

In Figure 2.4, the four points that might be of interest are marked by bulb-like objects labelled "col. 1" to "col. 4". When the query is executed, the data that flows at those four points will be displayed in a so-called result table. As can be guessed, the labels "col. 1" to "col. 4" refer to columns one to four of that table.

Executing the query yields the result table shown in table 2.1.

col 1	col 2	col 3	col 4
Node (Passau)	Edge (train)	Edge (train)	Node (Graz)
	Edge (train)	Edge (train)	Node (Paris)
	Edge (car)		

Table 2.1: Result Table of the Query Graph in Figure 2.4

Now it is easy to understand what happens when executing the query. Column one of the result table shows - as expected - the node "Passau" itself. This node passes a box the label of which reveals that the out-edges of the node are calculated, confirmed by the second column of the result table. Intuitively, the second box acts like a set of bars in a pipeline. Only those edges are allowed to pass that have a label equal to the string "train". At this point (column three), all departing train connections from Passau are still in the pipeline. The last box gets the target nodes from those edges and produces the nodes Graz and Paris, as can be seen in the fourth column of the result table. The result of the query therefore answers the question: "*Which cities are directly reachable from Passau by train?*"

Example 2.2: Another example shall be demonstrated that is concerned with a task applicable to arbitrary graphs, independent of any labels:

"*Show all nodes of a graph sorted according to their out-degree*"

To solve this problem, assume that the set of graph elements of the graph displayed in Figure 2.2 is the input to the query graph. First of all, only the nodes are of interest. Therefore, a box is needed that works as a filter that only nodes can pass. A box titled "GetGraphElements(nodes)" can be used for this purpose and is added to the pipeline.

The data currently in the pipeline is the list of nodes of the input graph. This list will be needed twice: The out-degrees must be calculated and it must be sorted. Therefore the pipeline splits, duplicating the list of nodes (a real pipeline would of course split the data in half). At one pipe, a `GetProperty_Box(out-degree)` is attached that will get the out-degrees of all nodes that flow into this box. The subquery up to this point and the data that is flowing is presented in Figure 2.5.

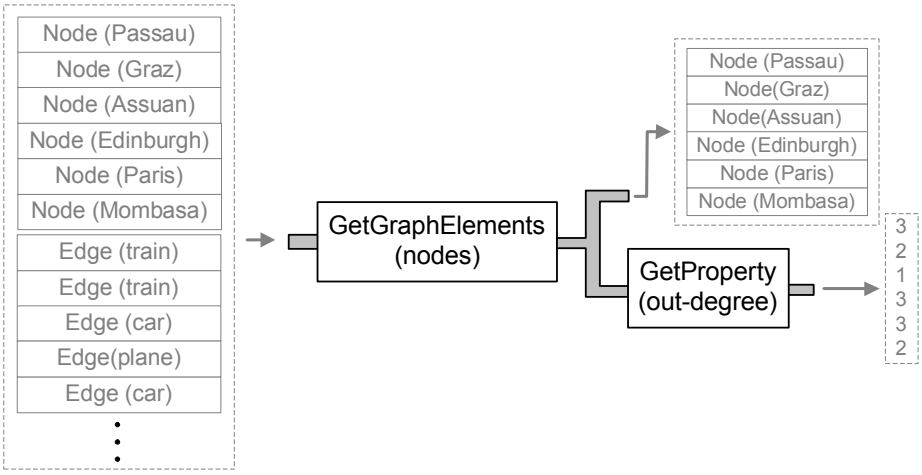


Figure 2.5: Example Query Graph 2 a)

The search for a box that can do the sorting successfully comes across the `SortBy_Box`. This box takes two parameters m and n, specifying the number of input collections m that are going to be sorted according to n other input collections. Finding that this box perfectly fits the needs with $m = 1$ (the list of nodes shall be sorted) and $n = 1$ (it shall be sorted according to the list of out-degrees), it is added to the pipeline. This `SortBy_Box(1, 1)` reunites the data stream into one output.

Figure 2.6 shows the final query, extended by three bulbs "col. 1", "col. 2" and "col. 3". The corresponding result table is shown in table 2.2.

The nodes in the third column are sorted according to the values of their out-degree as can be verified with the data in the first two columns. Nodes with the same out-degree are sorted in random order. In the example, a stable sorting algorithm has been applied. Thus, those elements are kept in the same order as they have been in the input.

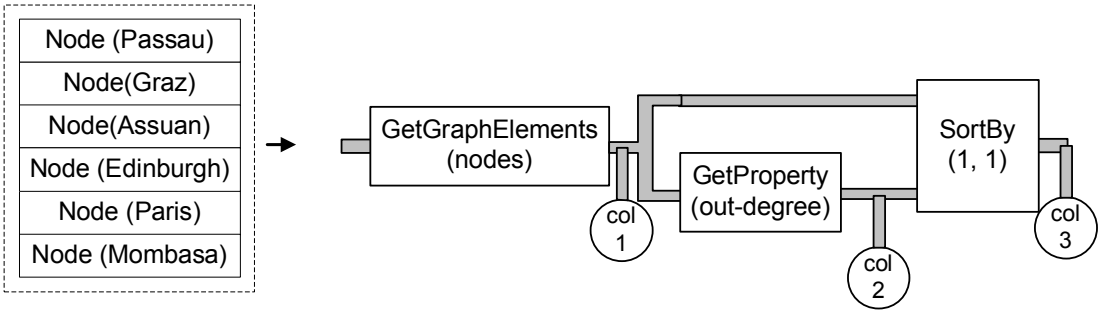


Figure 2.6: Example Query Graph 2 b)

col 1	col 2	col 3
Node (Passau)	3	Node (Assuan)
Node(Graz)	2	Node(Graz)
Node(Assuan)	1	Node(Mombasa)
Node (Edinburgh)	3	Node (Passau)
Node (Paris)	3	Node (Edinburgh)
Node (Mombasa)	2	Node (Paris)

Table 2.2: Result Table of the Query Graph of Figure 2.6

Chapter 3

The Query Language

In this chapter, syntax and semantics of the query language specifically designed for this work are presented. After the introduction and definition of some important terms in Section 3.1, Section 3.2 concentrates on the language elements.

The chapter also contains several examples how to use of the language.

3.1 Basic Terminology

Definition 3.1 (Current Graph)

The graph on which the query will be executed is called the **current graph**. □

The query is expressed via a directed, acyclic graph. Data is passed along its edges from the source to the target node. At the nodes, the fundamental components of the query are situated: Operations are executed on the incoming data. In the context of the query graph, those operations are called ***boxes***:

Definition 3.2 (Box)

A **box** is an operation. It is of the form

$$op : [in_1, in_2, \ldots, in_i, p_1, p_2, \ldots, p_s] \rightarrow [out_1, out_2, \ldots, out_o]$$

where $i \in \mathbb{N}$ is the **number of inputs** and $o \in \mathbb{N}_0$ the **number of outputs**. $in_1, \ldots, in_i$ represent dynamic inputs that are generated while the query is executed. There is a finite number s of design-time **parameters** $p_1, \ldots, p_s$ to the operation, the value of which must be known before the query is run.

To guarantee greatest flexibility, the number of inputs i and outputs o can depend on values of the parameters. □

Some boxes generate data without needing any input. These are called ***input box***es:

Definition 3.3 ((Standard) Input Box)

An **input box** is a box with no inputs, i.e. $i = 0$.

Although these boxes can appear anywhere in the system, there is one that is always present in the system. It can neither be removed nor added by the user. It is called the **standard input box** and is used to generate the input to the query. The role of this box is special since it accesses the ***current graph*** and produces the set of its graph elements as output. □

The implementation provides a second special input box that produces the set of *selected* graph elements as output. It is not included as a ***standard input box*** since the system does not necessarily run accompagnied by a graph editor like *Gravisto* that supports selections.

An other example of an ***input box*** is one that generates constants of specific types useful to the query (cf. the `Constant_Box` defined in Section 3.2.1 on page 24).

> **Remark:** In most example query graphs, the ***standard input box*** is omitted to save space. If not stated otherwise, an input bearing to the left border without being connected to a box, is implicitly connected to the ***standard input box***.

One goal during the design of the query language has been to provide a simple and user-friendly way of entering a query. To accomplish this, a way of graphically composing it has been chosen. Therefore, every box needs some kind of graphical representation that can be used to display it:

Definition 3.4 (Box Representation)

A **box representation** is an arbitrary graphical representation of a ***box***. It is good style, though, to comply to some defined look-and-feel so that all box representations have at least some similarity. □

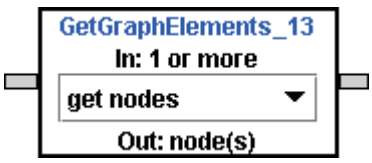


Figure 3.1: Standard Box Representation

Figure 3.1 shows a standard box representation. The name of the box is "`GetGraphElements_Box`", it has one input, one output and one parameter. The representation shows the name of the box (including a number, uniquely identifying the box within the query graph), information about the input and output types and provides means to show and

edit the parameter (here using a combo box). Docking points for input and output connectors are present in form of the small shaded rectangles to the left and right of the main shape.

The structure that connects the boxes forming the query is called the ***query graph***. It is formally defined as follows:

Definition 3.5 (Query Graph)

The **query graph** $QG(B, E)$ is defined as a directed acyclic graph. B is a set of boxes (cf. Definition 3.2). E is a set of directed edges. Each connects a **source box** with a **target box**. Edges represent pipelines through which information can be passed between boxes.

To be able to traverse the query graph correctly, an edge has to carry some information about its source and target:

- **Output Index:** The index of the output to which it is attached at the source box.
- **Input Index:** The index of the input to which it is attached at the target box.

There are two restrictions on the set of edges that can be incident to a box:

- For each output of a box b, b can have no or one outgoing edge.
- For each input of a box b, b can have no or one incoming edge. □

A query graph in the *QUOGGLES* system will simply be called "query" if no confusions are to be expected.

One term that will frequently be used is defined as follows:

Definition 3.6 (Free Input / Free Output)

Assume a box has i inputs and o outputs. In some query graph, this box has i_{edge} in-edges and o_{edge} out-edges. All $i - i_{edge}$ inputs and $o - o_{edge}$ outputs that do not have a corresponding edge are **free inputs** and **free outputs** respectively. □

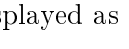
The presentation of most boxes will be accompanied by an illustration. An example input is displayed to the left, the box in the center and the output that the box will calculate from the input is displayed to the right. An object o is represented in the following way, according to its type:

String: o is displayed as "*string value of o*"

Boolean: o is displayed as *true* or *false*

Node: o is displayed as 

, where i is a number uniquely identifying the node

Edge: o is displayed as 

, where i is a number uniquely identifying the edge

Collection: o is displayed using a dashed box. Inside the box, the elements of o are displayed according to their type one below the other, the first element being the topmost one. Figure 3.2 shows a collection that contains three collections. Written as lists, this would be equal to

$$[\ [e4], [e1, e2], [e1, e3]\ ]$$

Objects the type of which has not yet been mentioned (like numbers) are displayed by their string representation.

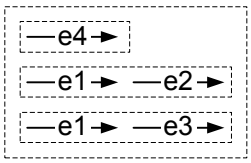


Figure 3.2: Representation of a Collection in Examples

3.2 Boxes

The basic components of the query language used by the *QUOGGLES* system are boxes. Therefore, there exist several types of them and each fulfils a special task. Although the system is meant to be easily extensible, this work specializes its examination on a restricted set of boxes. The power of the language induced by them is examined in Chapter 4.

3.2.1 Basic Boxes

The following list describes the boxes needed for the query language to exhibit some crucial power. They constitute the basic constructs of the language and are an integral part of the system. All operations represented by boxes mentioned in Section 3.2.3 can be simulated by a combination of several basic boxes.

The implementation of these boxes can be found in package `quoggles.stdboxes`.

> **Remark:** If not stated otherwise, boxes that take a list (i.e. an ordered collection) as input will work on its elements in the order imposed by that list. For unordered collections, the order will be defined by the iterator this collection provides (iterators are described in Section 1.2.4).
> All boxes accepting a collection will also accept an object that is not a collection. They treat this input as a list of size one, containing exactly this one object. If the result of the application of the box operation would yield a collection containing exactly one element, the output will be that element (and not the collection).

The examples following all draw on data from the example graph shown in Figure 3.3.

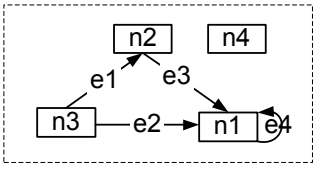


Figure 3.3: Small Example Graph

Boxes Generating Data

The boxes that belong to this category have no inputs. They generate output by directly accessing the ***current graph*** or by using information given to them via parameters.

- **StandardInput_Box** As defined in Definition 3.3, this box cannot be added or removed by the user. It provides the the graph elements of the ***current graph***.

Input: *none*

Output: A collection of all `GraphElement`s of the current graph. The element order depends on the current graph's data structure.

- **Constant_Box**
  Used to introduce constants into the query (see Figure 3.4 for an example).

  Input: *none* (except for the first parameter having value *collection*; the input must then be a natural number)

  Output: A value the type and value of which are specified by two parameters.

  Parameter 1: The user can choose between several types:
  - *integer:* A (positive or negative) whole number.
  - *double:* A floating point number.
  - *boolean:* A boolean value (i.e. either true or false).
  - *string:* An arbitrary text (without line breaks).
  - *empty:* The special value `QConstants.EMPTY`.
  - *collection:* The special value `QConstants.EMPTY`. The input number specifies the size of the collection to create. Elements are initialized to the special value `QConstants.EMPTY`.

  Parameter 2: A special component that allows the user (only) to enter a value compatible with the type specified by the first parameter. A boolean value can be set using a check box, for example. The last two possible values of the first parameter do not allow editing this parameter.

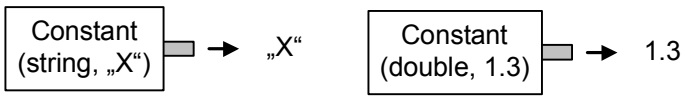


Figure 3.4: Two `Constant_Box`es

Boxes Gathering Information

All boxes mentioned in this subsection extract information from the input elements.

- **GetAttributeValue_Box**
  Used to retrieve attributed information from graph elements (cf. Figure 3.5 for an illustration).

  Input: A collection of `Attributable`s (cf. definition in 1.2.3 on page 1.2.3).

  Output: A collection of values of `Attribute`s (the path to the desired attributes is specified by the parameter). The i^{th} element of the input collection will produce the i^{th} element of the output collection.

  Parameter: The path (a string) to the desired attribute.

If an input graph element does not have the specified attribute, a special value (`QConstants.EMPTY`) is inserted instead of an `Attribute`'s value. This is done to ensure the box's ***relating*** behavior, which will be defined in Definition 4.7.

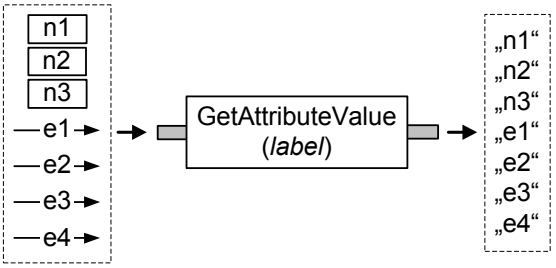


Figure 3.5: `GetAttributeValue_Box`

- **Neighborhood_Box**
  Used to gain information about the graph theoretical vicinity of graph elements (Figure 3.6 illustrates this).

  Input: A collection of objects. The accepted type of those objects depends on the value of the parameter:
  - *neighbors / inc. edges / in-edges / out-edges:* Objects must be `Node`s.
  - *inc. nodes / source node / target node:* Objects must be `Edge`s.

  Output: A collection of the same size as the input collection. The content depends on the value of the parameter.

  Parameter: The user can choose between several options:
  - *neighbors:* For each input node n, all nodes n_o are put in a set for which an edge between n and n_o exists.
  - *inc. edges:* For each input node n, all edges that have n as source or target node are put in a list.
  - *in-edges:* For each input node n, all undirected edges that have n as source or target node and all directed edges that have n as target node are put in a list.
  - *out-edges:* For each input node n, all undirected edges that have n as source or target node and all directed edges that have n as source node are put in a list.
  - *inc. nodes:* Source and target node of each input edge are put in a list.
  - *source nodes:* Retrieves the source node of each input edge.
  - *target nodes:* Retrieves the target node of each input edge.

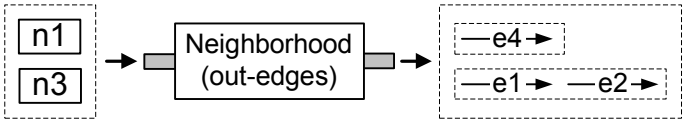


Figure 3.6: `Neighborhood_Box`

Filter Boxes

As the name indicates, the purpose of filter boxes is to remove elements from an input collection that do not match certain criteria.

The next box needs the notion of a subquery that, for any input, results in one single boolean value:

Definition 3.7 (Predicate Subquery)

Let $Q = (V, E)$ be a query in the *QUOGGLES* system. A query $Q' = (V', E')$ is called a **predicate subquery** of Q, if all of the following conditions hold:

- V' does not contain any ***standard input box***es
- $V' \subset V$ (the containment is strict since a query always contains at least one ***standard input box***)
- $\forall e \in E' : source(e) \in V' \wedge target(e) \in V'$, where $source(e)$ and $target(e)$ represent the source and target node of an edge e
- $|\{b \mid b \in V',\ b \text{ is a } \texttt{BoolPredicateEnd\_Box}\}| = 1$, i.e. there must be exactly one `BoolPredicateEnd_Box`, identifying the boolean result of the subquery. This box is defined in this subsection. □

> **Remark:** In the current implementation, the last requirement is slightly relaxed, removing the need for the existence of a `BoolPredicateEnd_Box` in certain circumstances: Suppose there is no `BoolPredicateEnd_Box` in a ***predicate subquery*** $G' = (V', E')$. If there is only one box $b \in V'$ that has free outputs and the number of free outputs is one, a `BoolPredicateEnd_Box` can be added automatically to b at this free output. Then G' complies with the stricter requirement. After execution of the query, the additional `BoolPredicateEnd_Box` is removed.

The `ComplexFilter_Box` uses this ***predicate subquery*** as a predicate according to which input elements are filtered:

- **ComplexFilter_Box**
  Used to filter input elements according to a complex predicate.

  The `ComplexFilter_Box` needs an extension of Definition 3.2: To be able to apply the operation associated with this box, in addition to the inputs, knowledge of the subquery attached to its second output is necessary. The box executes the subquery and uses the result to calculate its output.

  Input: A collection of objects of arbitrary types.

Output 1: A collection of elements that passed the predicate test (the test is applied in a predicate subquery attached at the second output).

Output 2: Each element from the input collection appears at this output, one after the other. This output is designed to be the starting point of a ***predicate subquery*** that results in a boolean value. If, for input element g, this subquery evaluates to true, g is put into the output collection at the first output. g is ignored otherwise.

Parameter: *none*

An example of how to use this box is given after the introduction of the next box in Example 3.1.

- **BoolPredicateEnd_Box**
  This box is not a filter itself, but is used only in conjunction with a `ComplexFilter_Box`.

  It is needed to unambiguously identify the boolean result of a predicate subquery: A query may have several ***free outputs***, i.e. outputs that are not connected to other boxes. If such a query is used as a predicate, a `BoolPredicateEnd_Box` must be attached to the output that holds the desired result. In addition to that, this box also converts any input to a boolean value according to the following rule:

  An object evaluates to false, if one of the following holds:

  - it is an empty `Collection`
  - it is a non-empty `Collection` and one of its elements evaluates to false
  - it is of type `Boolean` object and its value is `false`
  - it is a `Number` object and its value is `0.0` or `NaN` (not a number)
  - it is `null`
  - it is the special value `QConstants.EMPTY`
  - its value as a string (i.e. what a call to `toString()` yields) is equal to "false" (ignoring case)

  If the objects satisfies none of those conditions, the result is the boolean value `true`.

  Input: Any object.

  Output: *none* (though the result of the internal conversion of the input value to a boolean value is used by the system)

  Parameter: none

Example 3.1: An example use of a `ComplexFilter_Box` *cb* and a `BoolPredicateEnd_Box` is given in Figure 3.7. A set of four nodes is the input to *cb*. At the second (predicate) output, a ***predicate subquery*** is attached that will test each node. The predicate first will get all out-edges of a given node and then use the implicit conversion to a boolean value done by the `BoolPredicateEnd_Box`. The results of the `Neighborhood_Box(out-edges)` is displayed for each node. Node `n1` for example has one out-edge, node `n2` two and node `n4` has no out-edges at all. The

`BoolPredicateEnd_Box` will convert every non-empty list to the value true. Therefore, only the first three nodes pass the test and will be accepted by *cb*. That means that they will be put into a list that is available at *cb*'s output number one (as shown in the figure). The predicate can be much more complex, as long as there is exactly one `BoolPredicateEnd_Box` that specifies the result of the test.

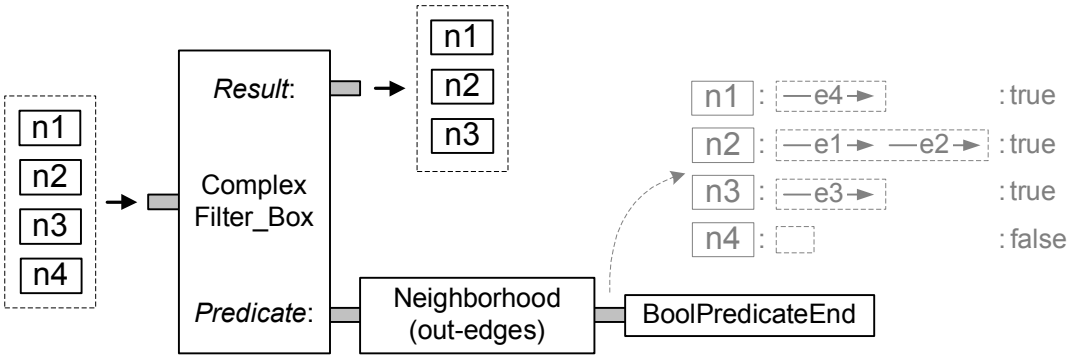


Figure 3.7: `ComplexFilter_Box` and `BoolPredicateEnd_Box`

Boxes Operating on Collections

Most data is present in the *QUOGGLES* system in the form of collections. Thus, operations that operate on collections are needed, especially some that act as conversion functions must be provided.

- **ListOperations1_Box**
  Used to work on a collection containing elements of arbitrary type. The number in the name indicates that this box works on a single input collection. The box will be extended later to work on several inputs. These inputs will represent a single relation, however. Thus, the name keeps its intuitive meaning. The `ListOperations2_Box` operates on two input collections and is presented afterwards. Since the first parameter specifies the operation which is applied, the description of this box is categorized according to its value:

  Input: A collection of objects of arbitrary type.

  Output: A collection the content of which is determined by the parameter.

  Parameter: The parameter may have one of the following values:

  – *flatten:* The output is a flat structure of the input collection (this flattening is applied recursively). No duplicate elimination is performed.
  Example: If the input is a list of the form

  ```
  [ [A1, A2], B, [C1, A1, C2] ]
  ```

  the resulting list will look like this:

  ```
  [ A1, A2, B, C1, A1, C2 ]
  ```

- *reverse:* The output is the input collection with the order of its elements reversed.
- *remove empty:* The input collection is traversed and a copy of it is returned where all elements that are equal to the special value `QConstants.EMPTY` have been removed.
- *make distinct:* All duplicates are removed from the input collection. Duplicates are defined in terms of the standard Java `boolean Object.equals(Object obj)` method. This means that, by default, two lists are equal if they contain the same elements in the same order and two `Number`s are equal if and only if they have the same type (`Integer`, `Double`, ...) and their values are equal. The objects under comparison can implement their own semantics, however.

Example 3.2: A common use of the `ListOperations1_Box(flatten)` is presented in Figure 3.8. To get a list of all out-edges of a set of nodes, the output of a `Neighborhood_Box(out-edges)` is not exactly what is wanted because it is a list of sets. The `ListOperations1_Box(flatten)` produces the desired list (note that although this list could theoretically contain duplicates, it will not contain any edge twice since an edge has exactly one source node).

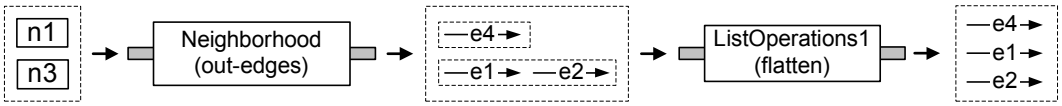


Figure 3.8: `ListOperations1_Box(flatten)`

Definition 3.8 (Homogenous Collection / Homogenous Type)

A collection is **homogenous**, if it fulfills one of the following conditions:

- all elements have the same type
- all elements are instances of `Collection`
- all elements are instances of `GraphElement`
- all elements are instances of `Attribute`
- all elements are instances of `Attributable`
- all elements are instances of `Number`
- all elements are instances of `String`

If a collection is homogenous, the type mentioned in this list is the **homogenous type**. □

- **ListOperations2_Box**
  Used to operate on collections containing elements of arbitrary type. The number in the name distinguishes this box from the one presented in the last item. It indicates that this box has two inputs, whereas the `ListOperations1_Box` works on a single input collection only. The extensions described in Chapter 4 will add functionality using more than two inputs. Since these inputs then represent *two relations*, the number still has an appropriate meaning. Again, the description of this box is categorized according to the value of the first parameter:

  Input 1: A ***homogenous*** collection.

  Input 2: A ***homogenous*** collection having the same ***homogeneous type*** as the collection at input 1.

  Output: A ***homogenous*** collection having the same ***homogeneous type*** as the input collections. Its content is determined by the parameter. No duplicate elimination is done. If set operations are desired, a `ListOperations1_Box(make distinct)` must be added.

  Parameter: The parameter may have one of the following values:

  - *union:* The output is a list containing the elements of the first input collection and the elements of the second input collection.
  - *list minus:* The output is a list containing all elements of the first input collection that are not contained in the second collection. If an element occurs m times in the first collection and n times in the other, it is contained $m - n$ times in the result collection.

  It may not be relied upon a certain order of the elements.

Example 3.3: Figures 3.9 and 3.10 show examples of the `ListOperations_Box` with the two possible parameter values *union* and *list minus*. It can be seen how duplicates in the input collections are treated.

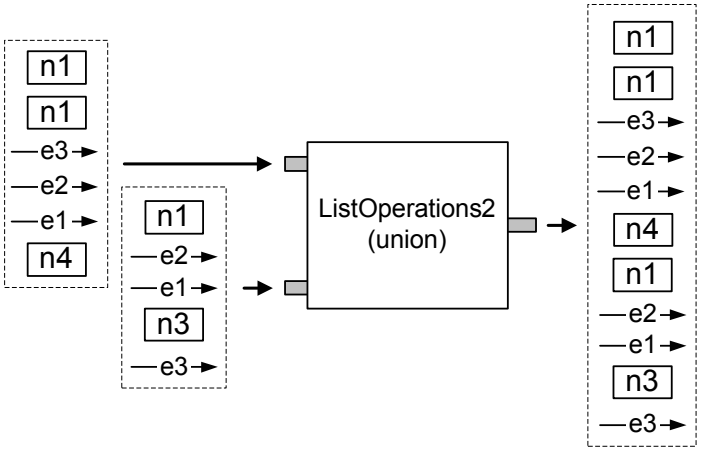


Figure 3.9: `ListOperations2_Box(union)`

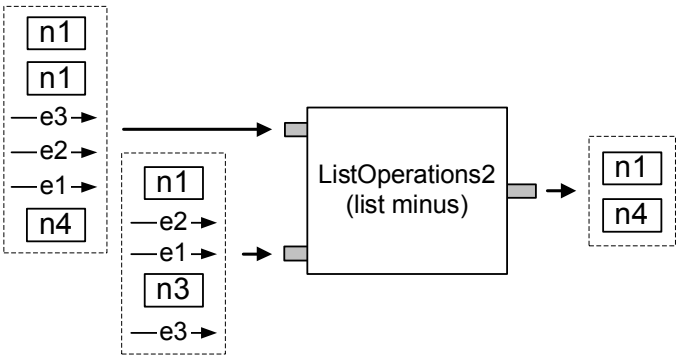


Figure 3.10: `ListOperations2_Box(list minus)`

- **Sort_Box**
  This box sorts the input collection.

  Input: A collection containing objects of the same type.

  Output: A list containing all elements of the input collection in ascending or descending order (as specified by the parameter).

  Parameter: The parameter may have one of the following values:

  - *asc.:* The output list is sorted in ascending order.
  - *desc.:* The output list is sorted in descending order.

  The sorting is done using the natural order of the elements. This means that objects implementing the Java interface `Comparable` are compared to each other using their implementation of the `int compareTo(Object)` method. The implementation of the box uses the string representation of all objects if one of them does not implement this interface. Strings are compared by the lexicographical order defined by method `int String.compareTo(Object)`.

  This definition leads to a common misunderstanding: Let N be a set of nodes. A graph editor might display some label near every node. Giving N as input to a `Sort_Box` does not necessarily sort it according to the value of node labels. Most probably (depending on the data structure used), the nodes will not implement the Java interface `Comparable`. Thus, the sorting algorithm uses their string representation as sorting criterium. This representation will most likely be a string describing a memory address since no custom `String toString()` method is implemented. The order of the nodes thus depends on the implementation of the data structure or is random. To sort a set of nodes according to their label values, a `SortBy_Box` can be used that will be introduced on page 60.

  Strictly speaking, the parameter specifying the ascending or descending sort order is not necessary: Since the sorting method used is not guaranteed to be stable, a descending order can be obtained sorting the collection in ascending order and reversing the list using a `ListOperations1_Box(reverse)` as described on page 28. For a sorting algorithm to be stable means that elements that are considered equal are not reordered and keep their relative order. If a stable method was used, two equal object would stay in the given order while reversing the list would swap them.

  The operation of the `Sort_Box` could have been put into the `ListOperations1_Box`. In Section 4.1.1, it will be enhanced with functionality that justifies the choice to put it into

a separate box. In fact, a new box called `SortBy_Box` will be introduced subsuming the functionality of the `Sort_Box`.

An important input restriction is that all elements of the input collection must be of the same type. One could argue that two elements that cannot be compared via their `int compareTo(Object)` method could be compared using their string representations. This leads to strange orders, however: Consider a list of three elements: The integer 13 the string "5" and the integer 4. A sorting algorithm might compare 13 to "5". Choosing the lexicographical comparison that yields

$$13 < \text{“5”}$$

Comparing "5" and 4 results in

$$\text{“5”} > 4$$

The last necessary step is the integer comparison of 13 and 4:

$$13 > 4$$

This gives a resulting order of $[4, 13, 5]$ which is neither lexicographically correct nor a sensible order on numbers.

The implementation of that box checks if all elements are of the same type and implement a `int compareTo(Object)` method. If not, all elements are compared using their string representation.

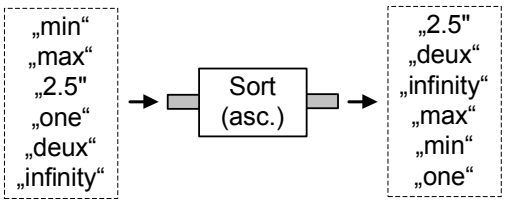


Figure 3.11: `Sort_Box`

Figure 3.11 illustrates the function of the `Sort_Box`. All elements are string values and are therefore sorted lexicographically (in ascending order).

- **Arithmetic_Box**
  This box provides two different types of operations: Aggregate functions and the four fundamental operations of arithmetics.

  Aggregate Functions:

  Input: A collection of `Number`s (or objects that can automatically be converted to numbers[1]). In case the parameter has the value *count*, a collection containing arbitrary objects is allowed.

  Output: A number representing the result of applying the operation specified by the parameter on the input collection. All operations ignore any occurrences of the special value `QConstants.EMPTY`.

[1] Two attempts are made to convert an object that is not a `Number` to a number, both using a string representation of the value of the object: First, it is tried to parse the string as an integer, and, if that does not lead to success, it is read as a floating point number. If both attempts fail, the object cannot be converted.

Parameter: The parameter sets the operation that is applied on the input collection:

- *sum:* The output is the sum of all elements.
- *count:* The output is the number of elements.
- *max:* The output is the element with the highest of all values.
- *min:* The output is the element with the lowest of all values.

Arithmetic Operations:

Input 1: A `Number` or an object that can automatically be converted to a number.

Input 2: A `Number` or an object that can automatically be converted to a number.

Output: A `Number` representing the result of applying the operation specified by the parameter on the inputs.

Parameter: The parameter sets the operation that is applied on the inputs:

- *plus (+):* The inputs are added.
- *minus (−):* The second input is subtracted from the first.
- *times (∗):* The inputs are multiplied.
- *divide (/):* The first input is divided by the second (the result is truncated to a whole number if both elements are of type integer).

Examples 3.4: Figure 3.12 presents the two possible applications of a `Arithmetic_Box`. The version on the left hand side has one input since the

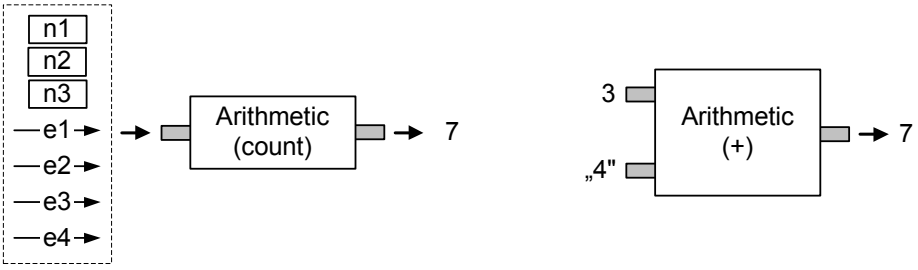


Figure 3.12: `Arithmetic_Box`

parameter specifies an aggregate function. The result is simply the number of elements contained in the input collection. The box on the right calculates the sum of the two input numbers. Since both input values are of type integer, the result is an integer, too. If one or both of them are floating point numbers, the box yields a double value. The same holds for the aggregate function *sum*: If all elements are integers, an integer is returned, a double value otherwise.

This box expects numbers as input. To be more intuitive and easier to use, the implementation performs implicit conversions. Therefore, the input string "4" in the second example is successfully converted to a natural number.

- **SizeOf_Box**

  Remark: The functionality of the `Arithmetic_Box(count)` is needed very often. To save space and since it renders several query graphs and algorithms more intuitive to read, an extra box is introduced at this point called `SizeOf_Box`. It has no parameters, takes an arbitrary collection as input and produces the number of elements contained in this collection as output. It adds no new functionality, though

Comparison Boxes

Boxes in this section compare their inputs to some value or with each other. They therefore always return an object of type boolean.

- **CompareTwoValues_Box**
  Used to check the relationship between two objects.

  Input 1: An arbitrary object.

  Input 2: An arbitrary object.

  Output: A boolean value that is the result of applying a binary comparison operation on the two inputs. The type of operation is specified by parameters.

  Parameter 1: The type to which both inputs are converted before they are compared or the type of comparison rule that is used. The following options are available:
  - *string:* The string representations are compared.
  - *integer:* Inputs are converted to whole numbers (i.e. `Long`s).[2]
  - *floating:* Inputs are converted to floating point numbers (i.e. `Double`s).[3]
  - *boolean:* Inputs are converted to boolean values (see the description of the `BoolPredicateEnd_Box` on page 27 for a more detailed explanation of this conversion).
  - *via equals:* The comparison uses the Java method `boolean Object.equals(Object obj)` (see the corresponding remark on page 29 for more details).
  - *==:* Inputs are compared via the Java `==` operator. That means that two objects are equal if and only if they represent one and the same object.

  Parameter 2: The following comparison operations are available, having their apparent meaning: *equal (=)*, *unequal (≠)*, *less (<)*, *less or equal (≤)*, *greater (>)*, *greater or equal (≥)*.

[2] The conversion uses the standard Java method `java.lang.Long.parseLong(String s), [SM02].`
[3] The conversion uses the standard Java method `java.lang.Double.parseLong(String s), [SM02].`

If the first parameter is one of *boolean*, *via equals* or ==, only the first two operations are allowed (equality and inequality).

It should be noted that the number of available comparison operations can be reduced to *equal* and *less* if boxes are introduced that represent the boolean *not* and *and* operations. Boxes carrying this functionality will be introduced shortly.

Example 3.5: Figure 3.13 shows two examples of the use of this box. The reason for the two results being different is the implicit conversion to floating point numbers in the top and strings in the bottom box.

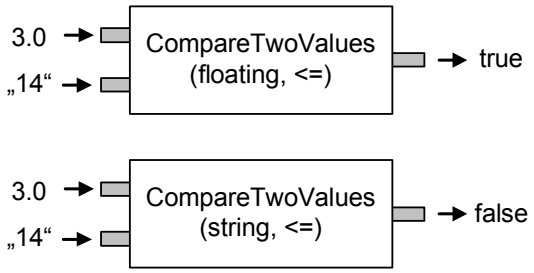


Figure 3.13: `CompareTwoValues_Box`

- **InstanceOf_Box**
  This box yields true if and only if the input object has some specified type.

  Input: An arbitrary object.

  Output: True if the input object is an instance of a certain type, false otherwise. The type is specified via a parameter.

  Parameter: States the desired type. It can be one of the following:
  - *node:* All instances of classes implementing interface `Node` yield true.
  - *edge:* All instances of classes implementing interface `Edge` yield true.
  - *graphelement:* All instances of classes implementing interface `GraphElement` yield true. All `Node`s and `Edge`s automatically implement this interface.
  - *boolean:* All `Boolean` objects yield true.
  - *string:* All `String` objects yield true.
  - *number:* All `Number` objects yield true.
  - A fully quantified class name can be entered that will be used as type.

Examples 3.6: Figure 3.14 presents two possible applications of a `Instanceof_Box`. The input is a collection (*true*) but is not an instance of `Node` (*false*).

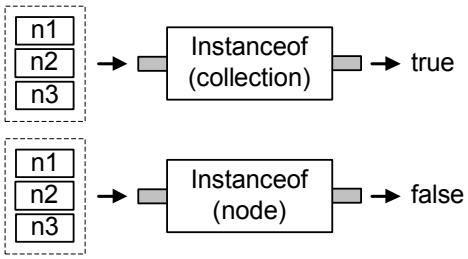


Figure 3.14: `Instanceof_Box`

The box in this basic variant cannot test if an input is a collection holding only elements of some specific type t. To test this, the input collection must be pushed through a `ComplexFilter_Box` with the test for the type t as its predicate subquery (i.e. an `Instanceof_Box(`t`)`. The size (`SizeOf_Box`) of the (filtered) result of the `ComplexFilter_Box` must then be compared (`CompareTwoValues_Box(integer, =)`) to the size of the original collection. The collection contains only elements of type t if and only if these sizes are equal. Such a test (where t is set to the class `String`) is illustrated in Figure 3.15.

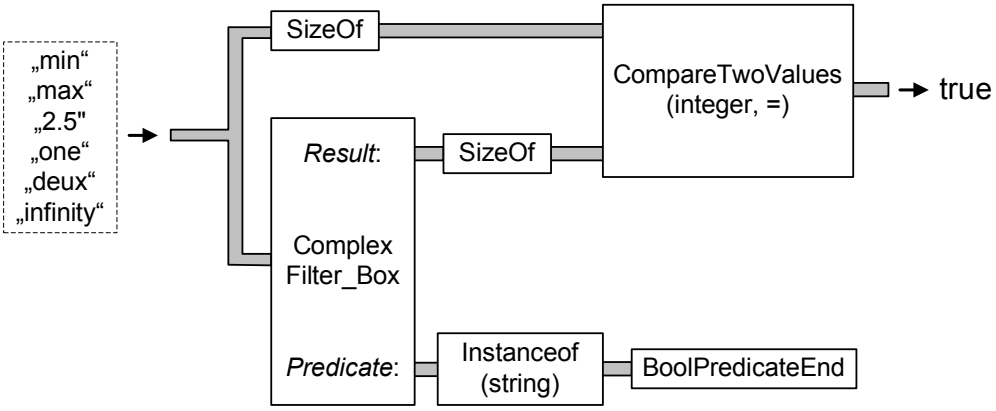


Figure 3.15: A Query Using an `Instanceof_Box`

Identity Operation Boxes

The boxes presented in this subsection do not change their inputs. They serve different purposes.

The result of executing a query must be shown in some way. Since a query does not only yield sets of graph elements, mere highlighting in the ***current graph*** is not sufficient. The presentation in tabular form is much more appropriate for which serve the next definitions:

Definition 3.9 (Cell View)

The **cell view** of an object o, in short **cv(o)**, is a string inductively defined as follows:

- If o is an instance of class `Node`: Distinguish two cases:
  - o has a label l. Let s be a string representation of l: $cv(o) =$ "Node(s)""
  - o does not have a label: $cv(o) =$ "Node"
- If o is an instance of class `Edge`: Distinguish two cases:
  - o has a label l. Let s be a string representation of l: $cv(o) =$ "Edge(s)""
  - o does not have a label: $cv(o) =$ "Edge"
- If o is the special value `QConstants.EMPTY`: $cv(o) =$ "-"
- If $o = [o_1, o_2, \ldots, o_n]$ is a `Collection`: $cv(o) = [cv(o_1), cv(o_2), \ldots, cv(o_n)]$
- In any other case, $cv(o)$ is a string representation of o □

Definition 3.10 (Result Table)

While executing a query, data is passed from box to box. Special boxes (the `Output_Box`es that are presented next) are used to mark inputs or outputs of boxes that are of interest. The data that is present at those places is displayed in a table, called the **result table**. Every such marked place will fill one column in the table.

Assume that an object o should be displayed in some column *col*, the cells of which are named $c_1, c_2, \ldots$. The column is defined as follows, distinguishing two cases:

- If o is a collection of size n, *col* has n cells. For $i \in \{1, 2, \ldots, n\}$, let c_i be the i^{th} cell. The content of c_i is the ***cell view*** of the i^{th} element of o.
- If o is not a collection, *col* consists of one cell. The content of that cell is the ***cell view*** of o. □

After those definitions, one of the most important boxes is defined. It is used to specify the edges in the query graph through which the data of interest is flowing. This data is then displayed by the system in the ***result table***.

- **Output_Box**

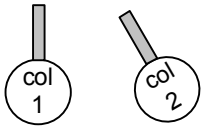


Figure 3.16: Two Representations of an `Output_Box`

This box just passes its input to its output. It can be attached to any input and output of any box, no matter if another box has already been attached there. When executing the

query, data flows along the edges of the query graph. The sole purpose of an `Output_Box` is to mark the place where the system should show the data present at that place. After executing a query, the system displays a ***result table*** showing the result of the query. Data provided by an `Output_Box` is shown in one column. The number of that column is specified by a parameter.

As has been described in the subsection on the `ComplexFilter_Box` on page 26, one and the same box can be executed several times during one execution of a query. The predicate subquery of a `ComplexFilter_Box`, for instance, is executed n times for a collection of n elements as input.

During one run of a query, an `Output_Box` that is executed several times does not reset the data it saved. Instead, it accumulates it by putting each element that passes through it into a list: Assume that an `Output_Box` is connected to the second (predicate) output of a `ComplexFilter_Box`. Let the input to the `ComplexFilter_Box` be a list of n elements. The `Output_Box` subsequently gets each element of that list as input. It saves all of them into a list. At the end of the execution, the `Output_Box` will pass the list of all n elements to the ***result table***. If $n = 1$, the only element of the list is passed to the ***result table*** (not a list containing this one element).

Input: An arbitrary object.

Output: The input object.

Parameter: A natural number specifying the column in the ***result table*** where the input data is shown. Every `Output_Box` has a unique number in a query.

To emphasize this special purpose, the ***box representation*** of this box looks different. Figure 3.16 shows two possibilities used in this thesis. Input and output are located at the same position.

Example 3.7: The query graph shown in Figure 3.17 is the same as has been used in Figure 3.7 with an additional `SizeOf_Box(count)` attached to the first output of the `ComplexFilter_Box`. The input is again a set of four nodes. Between every pair of consecutive boxes, an `Output_Box` has been inserted. Table 3.1 shows the ***result table*** of that query graph.

col 1	col 2	col 3	col 4	col 5
Node (n1)	Node (n1)	[Edge (e4)]	Node (n1)	3
Node (n2)	Node (n2)	[Edge (e1), Edge (e2)]	Node (n2)	
Node (n3)	Node (n3)	[Edge (e3)]	Node (n3)	
Node (n4)	Node (n4)	[ ]		

Table 3.1: Result Table of the Query Graph Shown in Figure 3.17

The `Output_Box(2)` only gets one of the four input nodes at a time. Nevertheless, after the `ComplexFilter_Box` has finished execution, all four nodes have passed through it. Therefore, columns one and two of the ***result table*** are

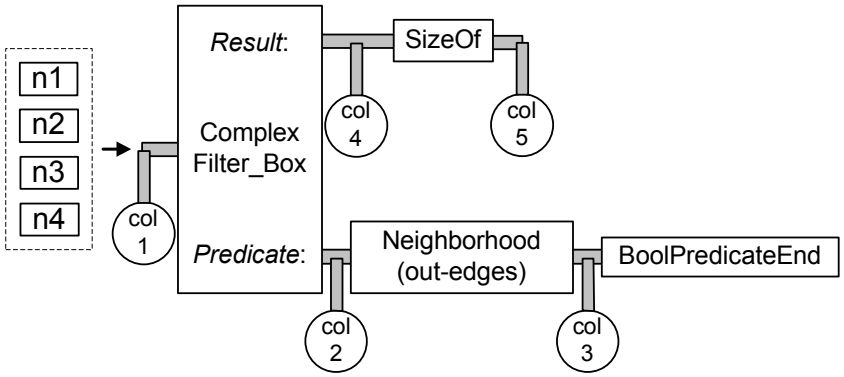


Figure 3.17: Example using several `Output_Boxes`

equal. In column four, node `n4` does not appear any more since it did not pass the predicate test. The last column shows the number of (input) nodes that are source nodes of some edges.

- **TwoSplitConnector_Box**
  Used to duplicate the input.

  Input: An arbitrary object.

  Output 1: A copy of the input object.

  Output 2: A copy of the the input object.

  Parameter: *none*

  The ***box representation*** of this box looks different from those of the other boxes as well. Figure 3.18 shows two possibilities, the first of which is used in this thesis. The figure also illustrates the function of this box.

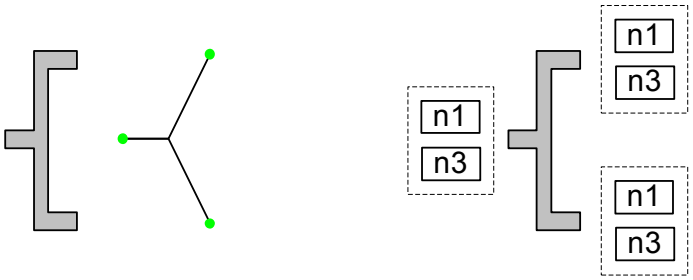


Figure 3.18: Two Representations of a `TwoSplitConnector_Box`

Calculating Boxes

The following boxes are used to perform an operation on their input(s) that can best be described as "calculating".

- **Arithmetic_Box**
  This box as already been described one page 32.

- **BooleanNot_Box**
  The boolean negation operation is implemented by this box.

  Input: A boolean object (no implicit conversion is done).

  Output: A boolean object having the negated boolean value of the input object.

  Parameter: *none*

Special Boxes

The boxes presented here do not fit in one of the categories already described.

- **SubQuery_Box**
  Used to put a whole subquery into one box.

  Input(s): Number and type depend on the associated subquery.

  Output(s): Number and type depend on the associated subquery.

  Parameter: The (absolute) path to a filename containing a saved query. When executing this box, the query loaded from that file will be executed.

  This box works as follows:

  Having created a query in the *QUOGGLES* system, the user can select any part of it (excluding the standard input box) and save it to a file. The file format used in the implementation is the `GML` format developed at the University of Passau, [MH95]. To save information about the state of the query boxes, the boxes are serialized into an `XML` string that is saved as an attribute of the associated nodes (cf. page 115). One or several inputs of boxes are not connected to another box at that moment, i.e. are free. These are marked, close to the appropriate box representations, in the selection with tags $\{i_1, i_2, \ldots i_m\}$.

  Loading this query into a `SubQuery_Box` means that this box gets equipped with m inputs that represent the free inputs of the subquery tagged with $\{i_1, i_2, \ldots i_m\}$, in that order.

  All `Output_Box`es are tagged with $\{o_1, o_2, \ldots o_n\}$ in ascending parameter value order. The `SubQuery_Box` then has n outputs corresponding to the tagged outputs in the given order.

  When executing this `SubQuery_Box`, all inputs are redirected to the corresponding inputs of boxes of the subquery. Analogous to that, data from the `Output_Box`es within the subquery is redirected to the outputs of the `SubQuery_Box`.

  Thus, this box is an abbreviation for a larger query graph. It is very useful if the same subgraph appears several times in a query. This subgraph can then be replaced by one single box. It might also be of assistance for an optimization step.

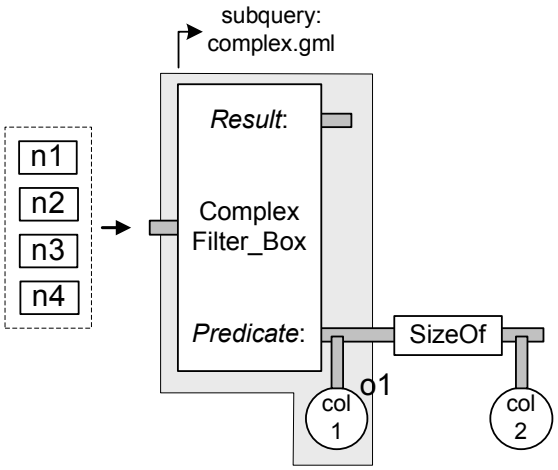
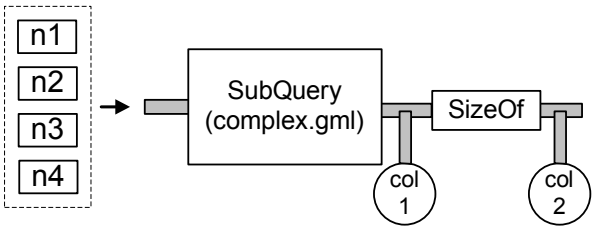


Figure 3.19: Marked Boxes will be put into a `SubQuery_Box`

The reason that this box is nevertheless not redundant is the following:

Assume a list of nodes $[n_1, n_2, n_3, n_4]$ is the input to a `ComplexFilter_Box` cfb with an `Output_Box` ob at its second (predicate) output. A box that is connected with ob will get four times one element as input. First n_1, then n_2, n_3 and finally n_4. A `SizeOf_Box` would therefore yield $[1, 1, 1, 1]$ as output. This is displayed in Figure 3.19, including the corresponding ***result table***.



col 1	col 2
Node (n1)	1
Node (n2)	
Node (n3)	
Node (n4)	

Figure 3.20: A Query Using a `SubQuery_Box`

If, on the other hand, cfb and ob are put into a `SubQuery_Box` sb (as is indicated in Figure 3.19), the corresponding output of sb would be the list $[n_1, n_2, n_3, n_4]$. This would make the `SizeOf_Box`, connected to the first output of sb (that corresponds to the second output of cfb), produce one single number: 4. The query graph and the its ***result table*** is shown in Figure 3.20.

That difference is due to the `SubQuery_Box` always executing its entire subquery. Therefore it has accumulated all intermediate results (four elements) to one (list of four elements).

This behavior is necessary at some points (see for example the procedure that calculates the cartesian product of two collections on page 76).

3.2.2 Example Using Basic Boxes

The next pages illustrate the creation of a query solving the following problem (the ***current graph*** is the one displayed in Figure 2.2):

"*Which cities can directly be reached from a given city by train?*"

To correspond to the data found in the graph, the question must be translated to: "Which nodes are connected to a given node with an out-edge labelled "train"?"

Answering the question requires four steps:

1. Get all out-edges from the given node,
2. filter desired edges by ...
3. ...retrieving their label and comparing them to the string "train" and
4. get the target node of the found edges.

As an example, assume the input is the node labelled "Passau". These steps can easily be transformed into a query graph:

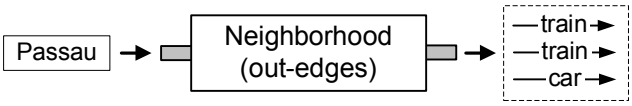


Figure 3.21: Basic Boxes Example, Step 1

1. All out-edges can be retrieved using a `Neighborhood_Box(out-edges)`. As shows Figure 3.21, having the node labelled "Passau" as input, the output is a list of three edges.

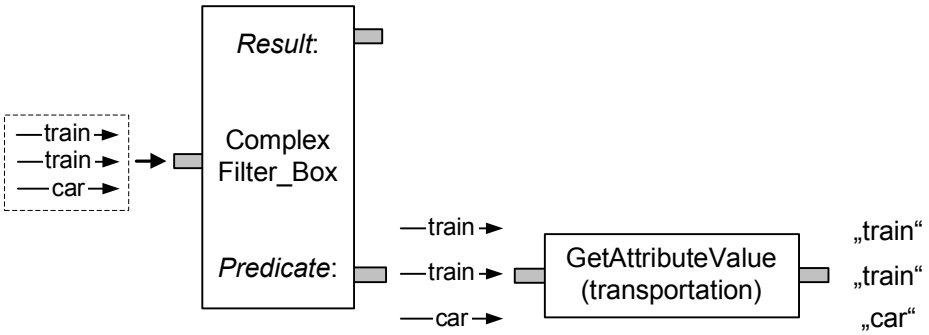


Figure 3.22: Basic Boxes Example, Step 2

2. Not all those edges contribute to the solution. They must be filtered. Therefore, a `ComplexFilter_Box` is used. The required condition uses the labels of the edges. Thus, a `GetAttributeValue_Box` is added as the first box in the predicate query (second output of the `ComplexFilter_Box`). As is indicated in Figure 3.22, the three edges each are the input to the predicate query and lead to strings "train", "train" and "car", respectively.

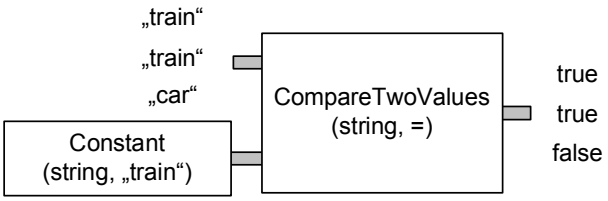


Figure 3.23: Basic Boxes Example, Step 3a

3. Comparing these strings with the constant string value "train" requires two boxes: A `Constant_Box(string, "train")` that generates the constant and a `CompareTwoValues_Box(string, =)` that performs the comparison. These two boxes are shown in Figure 3.23 together with the output they produce for the three strings: boolean values true, true and false.

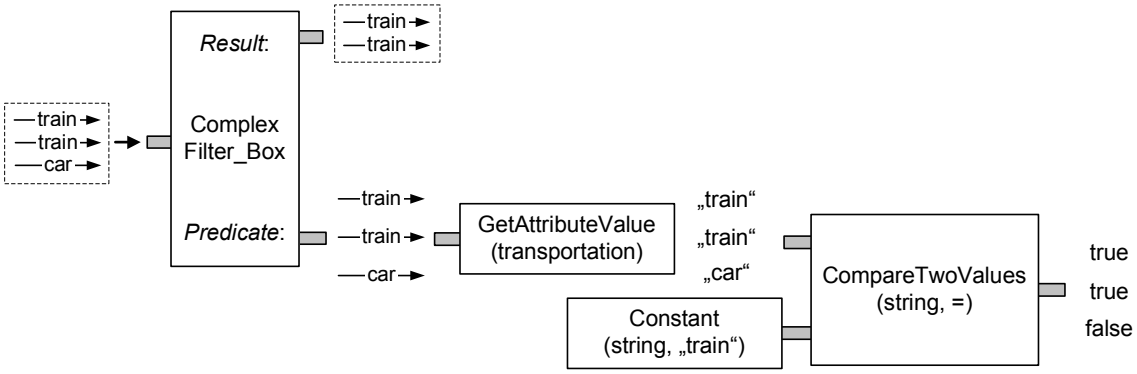


Figure 3.24: Basic Boxes Example, Step 3b

The query developed so far is portrayed in Figure 3.24. The inputs and outputs are shown as well. The out-edges of the input node are filtered and a list of those that are correctly labelled is present at the first output of the `ComplexFilter_Box`.

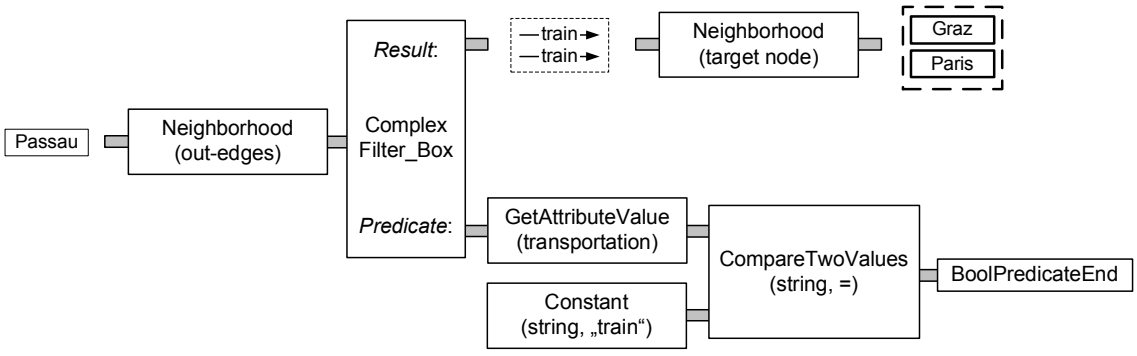


Figure 3.25: Basic Boxes Example, Step 4

4. The last step is to retrieve the target nodes of the found edges. A `Neighborhood_Box(target node)` serves this purpose. The output of this box is a list of the two nodes labelled "Graz" and "Paris", as is shown in Figure 3.25.

The whole query with an output box added to the place where the result is expected depicts Figure 3.26.

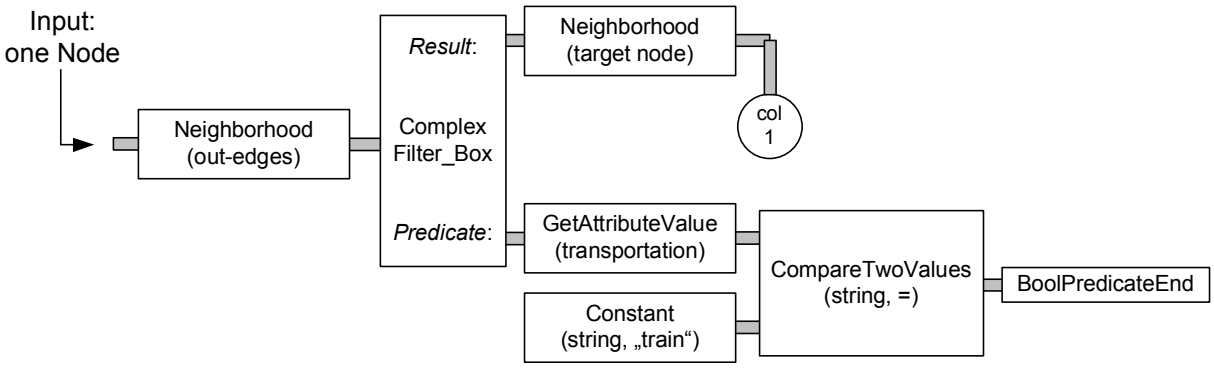


Figure 3.26: Basic Boxes Example

This query gets more concise if a `ValueCompare_Box(string, =, "train")` is used, replacing the two boxes `Constant_Box(string, "train")` and `CompareTwoValues_Box(string, =)`. This box is one of those that are not necessary for the power of the language since they can be replaced by several basic boxes. They contribute to the readability and ease of creating queries, however, since some combinations of basic boxes appear quite often. The five boxes `ComplexFilter_Box`, `GetAttributeValue_Box(`*`attrPath`*`)`, `CompareTwoValues_Box(`*`type`*`, `*`compOper`*`)`, `Constant_Box(`*`type`*`, `*`value`*`)` and `BoolPredicateEnd_Box` can be replaced by a single `AttributeFilter_Box(`*`attrPath`*`, `*`type`*`, `*`compOper`*`, `*`value`*`)`. The resulting graph is the one presented in the introductory example in Chapter 2 on page 15.

The next section introduces some useful boxes that can either serve cosmetic purposes or can be replaced by basic boxes. They are called auxiliary and compound boxes.

3.2.3 Auxiliary Boxes

The boxes described in this section do not introduce any new power to the system. In contrast to the boxes presented in the following section, they cannot be replaced by a set of basic boxes. They are used for layout or clarifying purposes only.

The next box can be found in package `quoggles.auxboxes`.

- **OneOneConnector_Box** This box is not necessary at all for the theory, but is heavily used in praxis. It represents an identity operation and is used to layout the query graph. Its ***box representation*** is a line between two end points with specified coordinates. In the graphical user interface, these coordinates can be changed by dragging the endpoints (or a point on the line close to one of the endpoints) using an input device like the mouse.

  Input: An arbitrary object.

  Output: The input object.

  Parameter: *none*

- **SplineConnector_Box**
  Used for the same purpose as the previous box. Instead of a straight line, the box representation of this box is a spline with an arbitrary number of control points.

  Input: An arbitrary object.

  Output: The input object.

  Parameter: *none* (number and position of control points do not influence the semantics of the box and are therefore not modelled as parameters)

3.2.4 Compound Boxes

This section describes a few useful boxes that will be used throughout the rest of this document. They are, however, only abbreviations for some combination of basic boxes. Thus, this part also serves as a collection of simple examples of how to use the basic boxes.

With the exception of the `SplitNConnector_Box`, all these boxes are implemented and can be found in package `quoggles.auxboxes`.

- **SplitNConnector_Box:** A box that can be used to replicated n times. It replaces a binary tree of $n-1$ `TwoSplitConnector_Box`es without introducing any more power to the language.

  Input: An arbitrary object.

  Outputs 1 to *n*: The input object.

  Parameter: *none*

- **ListOperations1_Box**
  This box has already been introduced as a basic box on page 28. Two possible parameter values are now added. The new functionality can be simulated using only basic boxes.

  Input: A collection of size equal to one.

  Output: The element contained in the input collection.

  Parameter: *unpack:* Having this parameter, the box restricts its possible input to collections of size one. The output is the only element contained in the input collection.

  A `ListOperations1_Box(unpack)` *lo* can be replaced by a `ComplexFilter_Box` *cb*. All outedges of *lo* are attached to the second output of *cb* instead. The first output is not needed. This exploits the property of *cb* to produce each element of the input collection at its second output. The only difference in its use is that the *QUOGGLES* system will complain if the subquery attached to *cb* does not contain exactly one `BoolPredicateEnd_Box`. Attaching one at some free output (generating one with a `TwoSplitConnector_Box` if necessary) will get rid of that problem.

- **ListOperations2_Box**
  This box has also been already introduced as a basic box on page 30. One additional parameter value (*intersect*) is added.

  Input 1: A ***homogenous*** collection.

Input 2: A ***homogenous*** collection having the same ***homogeneous type*** as the collection at input 1.

Output: A collection representing the intersection of the two input collection.

Parameter: *intersect:* The output is a list containing all elements contained in both, the first and the second input collection. If an element occurs m times in one collection and n times in the other, it is contained $min(m, n)$ times in the result collection.

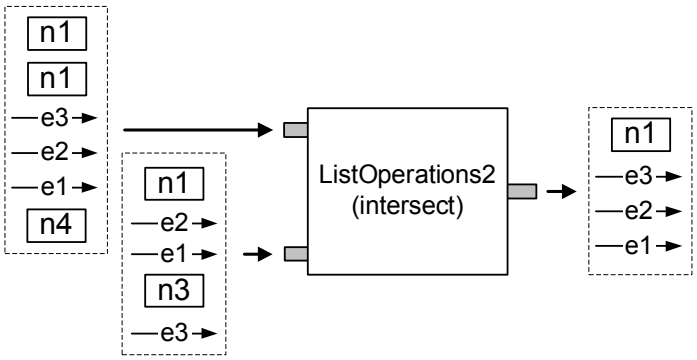


Figure 3.27: `ListOperations2_Box(intersect)`

Figure 3.27 shows an example use of the box with the parameter *intersect*. The intersection of two lists can be expressed as:

$$L_1 \cap L_2 = L_1 \setminus (L_1 \setminus L_2)$$

where

$$L_1 \setminus L_2 = \{\underbrace{a, a, \ldots, a}_{k \text{ times}} \mid a \in L_1 \ \wedge \ k = max\{0,\ \text{count}(a, L_1) - \ \text{count}(a, L_2)\}\}$$

and $count(a, L)$ is the number of times a appears in L.

Proof: It can easily be shown that for every $a \in L_1 \cup L_2$

$$count(a, L_1 \cap L_2) = count(a, L_1 \setminus (L_1 \setminus L_2))$$

Thus, this box can be simulated using two `ListOperations1_Box`es with parameter *list minus* which implement the "\" operation.

- **ValueCompare_Box**
  Compares an input object with a constant that can directly be specified by a parameter.

  Input: An arbitrary object.

  Output: *true* if the comparison defined by the parameter yields true, *false* otherwise.

  Parameters 1 / 2: See parameters of the `CompareTwoValues_Box` on page 34.

  Parameter 3: A constant that can be converted to the type specified by the first parameter.[4]

[4]The implementation uses a box representation that provides a neat possibility to enter values for that third parameter. The *Gravisto* system provides so-called `ValueEditComponents` that are small plug-ins used to provide means to specify values of certain types. There exist, for instance, special components to enter strings and floating or integral numbers.

Let par_1, par_2 and par_3 be the values of the three parameters. This box is not more than an abbreviation of the combination of a `CompareTwoValues_Box(`par_1`, `par_2`)` and a `Constant_Box(`par_1`, `par_3`)`. The latter box provides the data for the second input of the `CompareTwoValues_Box`. This is illustrated in Figure 3.28, where the `ValueCOmpare_Box` to the left can be replaced by the two boxes to the right.

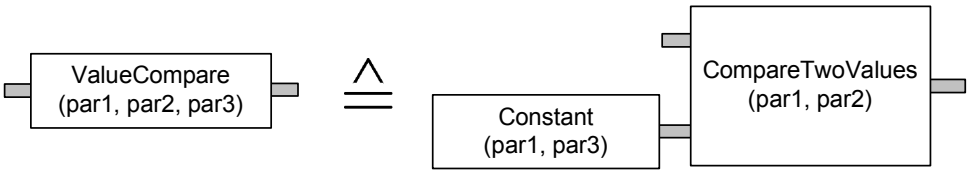


Figure 3.28: `ValueCompare_Box` and its Basic Replacement

- **Arithmetic_Box**
  This basic box (cf. page 32) is extended with one additional parameter value:

  Input: A collection of `Number`s (or objects that can automatically be converted to numbers).

  Output: A `Number` equal to the average value of all numbers in the input collection.

  Parameter: *avg:* The output is the sum of all elements divided by the number of the elements.

  The simulation with a sequence of two basic `Arithmetic_Box`es is straightforward.

- **AttributeFilter_Box:** Removes objects from the input collection the attribute of which does not comply with a specified condition.

  Input: A collection of `Attributable`s.

  Output: A sublist of the input collection.

  From an input collection, it lets pass only those elements e for which hold the next three conditions:

  - e must be an `Attributable` (i.e. designed to have attributes)
  - e must have the attribute a
  - The specified comparison of the value of a with the constant must yield true.

  Parameter 1: The path to an attribute.

  Parameters 2 / 3 / 4: See parameters of the `ValueCompare_Box` on page 46.

  Let the parameter values of this box be par_1, par_2, par_3 and par_4. This box replaces a combination of a `ComplexFilter_Box` with a sequence of a `GetAttributeValue_Box(`par_1`)` and a `ValueCompare_Box(`par_2`, `par_3`, `par_4`)` as predicate.

  This simulation is illustrated in Figure 3.29.

- **GetGraphElements_Box:** This box removes any objects from the input collection that is not a node, an edge or a graph element, depending on the parameter.

  Input: A collection of arbitrary objects.

  Output: A sublist of the input collection containing only objects of a certain type. This type is specified by the parameter.

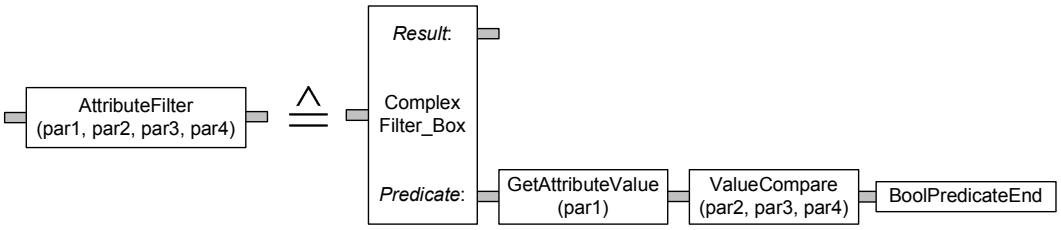


Figure 3.29: `AttributeFilter_Box` and its Basic Replacement

Parameter: The type of allowed objects. The user can choose between `Node`, `Edge` and `GraphElement`.

The box can simulated by a `ComplexFilter_Box` with an `InstanceOf_Box(type)` where *type* represents the type specified by the parameter, as illustrated in Figure 3.30.

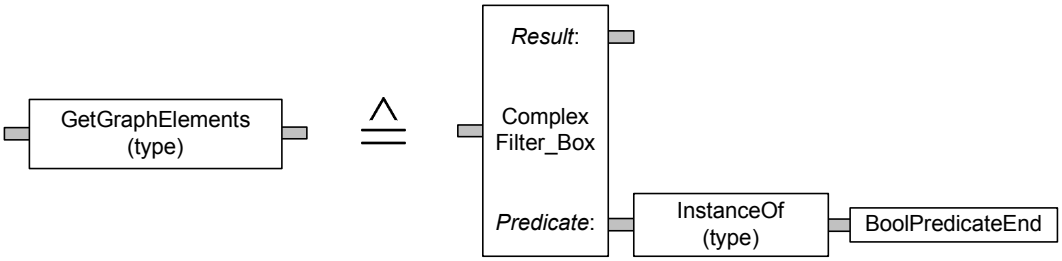


Figure 3.30: `GetGraphElements_Box` and its Basic Replacement

- **GetProperty_Box:** This box is used to get local information from graph elements. It is meant to be extended with whatever properties are present in the graph data structure. For now, it can retrieve the in-degree, out-degree and degree of a node.

  Input: A collection of `Node`s.

  Output: A list of numbers, depending on the parameter.

  Parameter: One of *in-degree*, *out-degree* and *degree*, having the meaning as defined in Section 1.2.

  This output can also be achieved via a sequence of a `Neighborhood_Box` with the appropriate parameter ("in-edges" / "out-edges" / "inc. edges") and a `SizeOf_Box`.

- **BooleanOp_Box:** Operations like $\wedge, \vee, \text{XOR}, \ldots$ can be applied to two boolean input values.

  Input 1: A boolean value.

  Input 2: A boolean value.

  Output: A boolean value, calculated from the two input values according to the value of the parameter.

  Parameter: One of the boolean binary operations *AND*, *OR*, *XOR* and *NOR*.

  To show that all those operations can be simulated via basic boxes, it suffices to present a query graph for $\wedge$. Together with the boolean not operation (see the `BooleanNot_Box` on

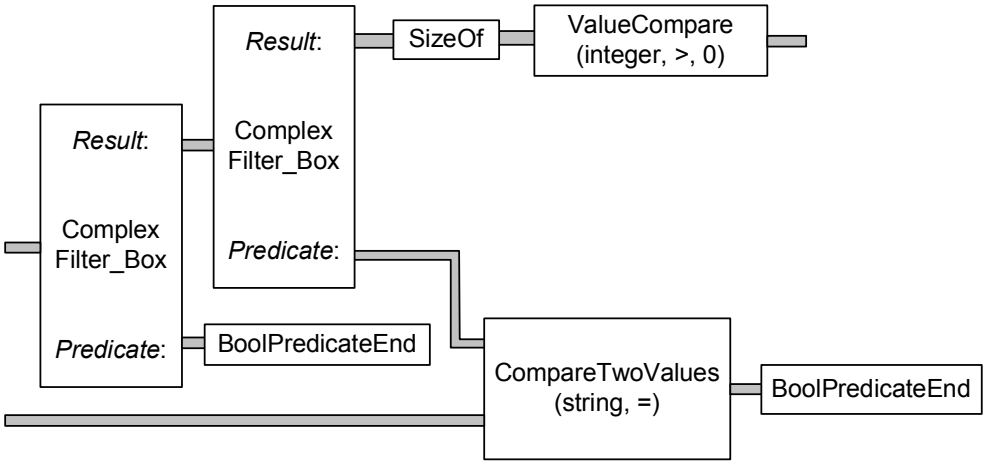


Figure 3.31: Simulation of the $\wedge$ Operation

page 40), it forms a basis for boolean formulas and all other operations can be expressed through them. $\vee$, for instance, can be written as $a \vee b \equiv \neg\,(\neg\, a \wedge \neg\, b)$.

Figure 3.31 shows the query graph necessary to simulate the $\wedge$ operation. Assume two boolean inputs a and b (at the left hand side of the figure). The first `ComplexFilter_Box` ensures that a is true. Otherwise, the input to the second `ComplexFilter_Box` would be the empty list, leading to an empty list as output. The combination of the `SizeOf_Box` and `ValueCompare_Box(integer, >, 0)` would yield false. If a is true, the input to the `CompareTwoValues_Box(string, =)` are "true" and the string value of b. Only if b evaluates to true, the comparison yields true. Thus, the result is *true* if and only if both a and b are true.

- **InducedSubgraph_Box:** An important operation is to get the subgraph induced by a set of graph elements. Although that seems to be a complicated procedure, it is possible to simulate it using several boxes. The definition of an induced subgraph is as follows:

Definition 3.11 (Induced Subgraph)

Let ge be as set (or a list) of graph elements belonging to one graph G. The **induced subgraph** G_{ind} of ge consists of the graph elements e for which one of the following properties holds:

1. e is an element in ge
2. e is an edge, the source and target node of which are contained in ge.
3. e is a source or a target node of an edge contained in ge. □

Input: A list of graph elements.

Output: A set of graph elements containing all graph elements of the subgraph induced by the input elements.

Figure 3.32 shows how to get the induced subgraph of a set of graph elements ge that are passed as input at the left hand side. The data present at the edges marked with numbers 1 to 4 is:

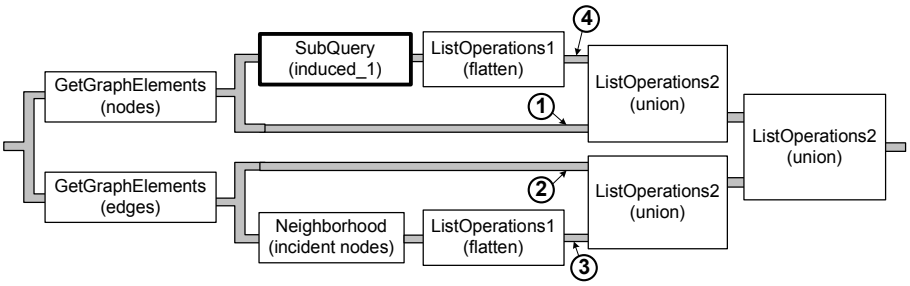


Figure 3.32: Simulation of the Induced Subgraph Operation, Part 1

(1) All nodes contained in ge are passed on (first property).

(2) All edges contained in ge are passed on (first property).

(3) Source and target nodes of edges in ge (third property).

(4) Edges the source and target nodes of which are contained in ge (second property).

The fourth part is done in a `SubQuery_Box(induced_1)`, the contents of which shows Figure 3.33. This query graph is hard to layout in a readable way. The idea behind it is quite easy, though: The `Neighborhood_Box(incident edges)` produces a list of edge *sets*. To iterate over all edges, the nested collection is flattened and a `ComplexFilter_Box` is used. The predicate ensures that only those edges are passed on that have source and target nodes contained in the input set ge. At the first output of the `ComplexFilter_Box`, all desired edges are present.

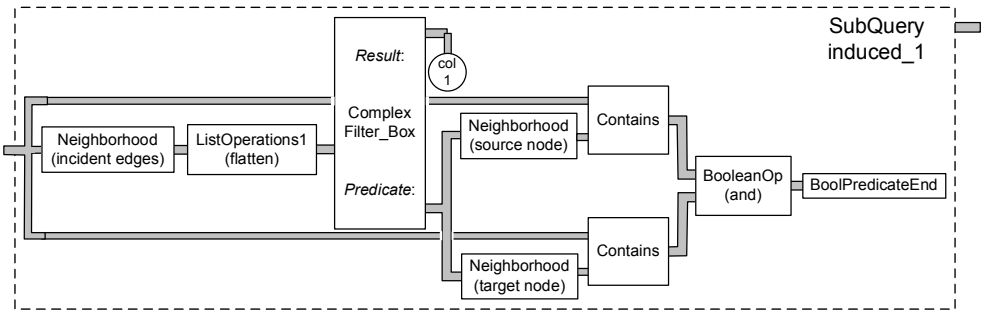


Figure 3.33: Induced Subgraph Operation, Part 2 (Subquery "induced_1")

Remark: Another definition of an induced subgraph repeats the second step after the third. This adds edges from G the source and target nodes of which are contained in ge or G_{ind} and at least one of them has been added in the third step. This slightly complicates the task of simulating the operation and increases the number of boxes needed thus making the query graph even harder to read.

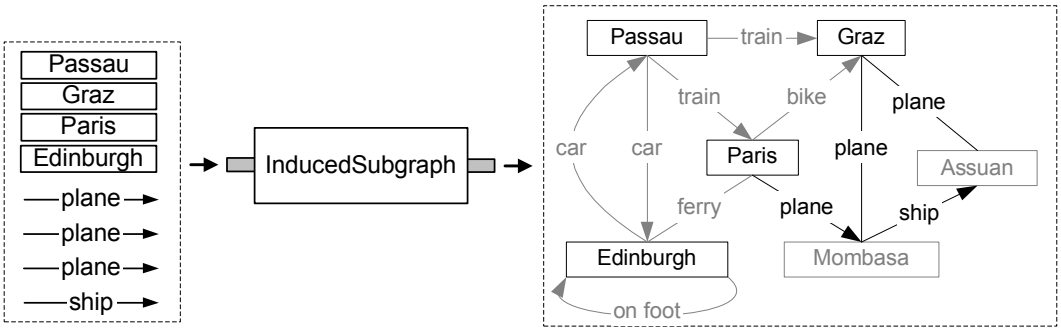


Figure 3.34: `InducedSubgraph_Box`

Example 3.8: From the graph presented in Figure 2.2, a sublist of graph elements are taken as the input to the `InducedSubgraph_Box` as shown in Figure 3.34. The result is the whole set of graph elements. Thus, all European cities and all train and ship connections induce the whole traffic system.

The next few boxes are important as well, but their basic counterpart is straightforward. Thus, the description of how to simulate them is omitted.

- **Equals_Box:**

  Input 1: An arbitrary object.

  Input 2: An arbitrary object.

  Output: A boolean value indicating whether or not the two input objects are considered equal. The definition of being equal is adopted from the implementation of the Java method `java.lang.Object.equals(Object o)`.

  Parameter: *none*

- **Contains_Box:**

  Input: An arbitrary object (although it will be a collection most of the time).

  Output: The boolean value *true* if and only if both inputs are equal or the first is a collection and the second input object is contained in this collection. *false* otherwise.

  Parameter: *none*

- **MakeTrue_Box:**

  Inputs 1 to *n*: n times an arbitrary object. The number of inputs can be specified by the parameter.

  Output: The boolean value *true*.

  Parameter: An integer $n \in \mathbb{N} \setminus \{0\}$, specifying the number of inputs of this box.

  This box is barely useful if not used to connect the predicates of two `ComplexFilter_Box`es to generate a cartesian product (see its use on page 76).

- **InterpretAsBoolean_Box:**

  Input: An arbitrary object.

  Output: The boolean interpretation of the input object:

  An object evaluates to *false*, if one of the following holds:

  - it is an empty `Collection`
  - it is a non-empty `Collection` and one of its elements evaluates to *false*
  - it is of type `Boolean` and its value is *false*
  - it is a `Number` and its value as a double is `0d` or `NaN`
  - it is `null`
  - its lower case string representation is equal to *false*

  Parameter: *none*

  This is the same way as the `BoolPredicateEnd_Box` internally processes its input.

Chapter 4

The Power of the Language

In this section, the type of problems that can be solved using the system introduced in the previous section (and those which cannot be tackled) are examined more carefully. Because of its extensibility, the real power of the query language is only restricted by the power of the programming language used. Boxes with arbitrary functionality can be introduced. Therefore, everything that can be programmed can be expressed via one single box.

It is much more interesting, however, to inspect the capabilities of the system given a few carefully chosen boxes with well defined semantics.

These boxes have been presented as "basic boxes" in Section 3.2.1. This restricted system will be compared with other approaches to graph querying methodologies like relational algebra and `SQL`. It turns out that some small extensions have to be added to the basic *QUOGGLES* system because the relational algebra view of graphs is quite different from the one presented here, but otherwise, the language is fully capably of simulating queries from such systems.

4.1 Comparison to Relational Algebra

When designing and examining query languages, what is the standard with which it must compete? They work on many different data structures and use completely disjoint paradigms. Besides the intuitive opinion of a user whether or not a language is of some use (to her), some standard must be defined.

One of the most renown query (and data manipulation) language is `SQL`. Most papers concerned with the power of `SQL` draw a comparison to relational algebra. Some concentrate on the safe tuple calculus or the safe domain calculus. It can be shown, however, that those three define the same relations, i.e. are equivalent. The reason why relational algebra plays such an important role can be found in [Cod72] and [Ull82] as well as [AU79]:
Codd fixes relational algebra as the

> "minimum capability of any reasonable query language"

and Ullman defines:

> "A language that can (at least) simulate [safe] tuple calculus, or equivalently, relational algebra or [safe] domain calculus, is said to be **complete**."

After a short definition and description of relation algebra, Section 4.1.2 shows the link between relation algebra and the *QUOGGLES* system. This lays the foundation of the proof given in Section 4.1.3 that the query language presented in this thesis subsumes the power of relational algebra and therefore is ***relational complete***.

4.1.1 Relational Algebra

Definition 4.1 (Set)

A **set** is an unordered collection of objects in which a given element is allowed at most once. □

Definition 4.2 (Tuple)

An ***n*-tuple**, written as $(o_1, o_2, \ldots, o_n)$ is a construct holding n elements that can be identified by a unique index i where $1 \leq i \leq n$.

n is the **arity** of the tuple. □

Definition 4.3 (Relation)

A **relation** R is a ***set*** of ***tuple***s. All tuples must have the same ***arity*** r.

The term arity is extended for relations: **arity(R)** $= r$.

The list of the i^{th} elements in all tuples is called the i^{th} ***component*** of R. If these components are named, the name of the i^{th} component is called the i^{th} ***attribute*** of R. □

Definition 4.4 (Relational Algebra)

Relational algebra is defined in terms of the following minimal set of six operations. Let R and S be two relations.

1. **Renaming:**
   - $\rho_V(R)$: renames relation R to V
   - $\rho_{B \leftarrow A}(R)$: renames attribute A in relation R to B
2. **Selection:** $\sigma_{cond}(R) = \{\tau \in R \mid cond(\tau) \text{ is true}\}$
3. **Projection:** $\pi_{i_1,i_2,\ldots,i_m}(R) = \{c_{i_1}, c_{i_2}, \ldots, c_{i_m}\}$ where, for $i \in \{i_1, i_2, \ldots, i_m\}$, c_i denotes the i^{th} component of R
4. **Set Union:** $R \cup S = \{\tau \mid \tau \in R \vee \tau \in S\}$
5. **Set Difference:** $R \setminus S = \{\tau \mid \tau \in R \wedge \tau \notin S\}$
6. **Cartesian Product:** $R \times S = \{\tau * \sigma \mid \tau \in R \wedge \sigma \in S\}$ where

$$(a_1, a_2, \ldots, a_m) * (b_1, b_2, \ldots, b_n) = (a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n)$$

□

4.1.2 Relational View of Graphs

Several approaches exist that model tree or graph structures for database systems. Even for relational systems, many different strategies are subject to discussion, especially in the field of research concerned with XML. Documents in XML format can be viewed as graphs. [FTS00], [FMST01], [MFK+00], [Süß01] and [SZF01] can be seen as introductory literature.

However, those tend to be optimizations with respect to a smooth integration with other data as well as to compatibility with external tools for graph data structures.

Since these aspects are not of interest for the following examination, a simple and straightforward model is used that is still complex enough to serve as a basis for the comparison of expressive powers. An object-oriented approach is chosen since it best matches the model used by the *QUOGGLES* system. Since the database should work on the same data as the *QUOGGLES* system, it contains only one graph for one query. An extension to several graphs is discussed in Section 5.1.2.

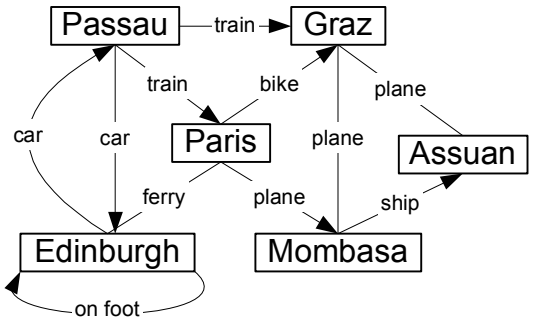


Figure 4.1: Example Graph (City Connections)

The model uses three relations, presented in the form of tables. Keys are underlined in the schema:

- A **GraphElement relation** associates nodes or edges with attributes that are common to both (i.e. have the same name and the same domain):

  `GraphElement(`<u>`node_or_edge_id`</u>`, ge_attr_1, ge_attr_2, ..., ge_attr_`k`)`

  for $k \in \mathbb{N}$. *node_or_edge_id* is either a node or an edge id. Node ids and edge ids have different domains. *ge_attr_i* (for $i \in \{1, 2, \ldots, k\}$) are attributes that nodes and edges can have. These are labels, colors, weights, etc.

Node	node_id	city	color
	n1	Passau	red
	n2	Edinburgh	red
	n3	Paris	red
	n4	Mombasa	green
	n5	Graz	green
	n6	Assuan	green

Table 4.1: Example Node Relation

- A **Node relation** associates an id and several attributes with nodes:

  `Node(node_id, n_attr_1, n_attr_2, ..., n_attr_`n`)`

  for $n \in \mathbb{N}$. The *n_ attr_ i* (for $i \in \{1, 2, \ldots, n\}$) are node specific attributes (which especially means that they do not appear in the **GraphElement** relation) like shape, size, coordinates, etc.

- An **Edge relation** associates an id and several attributes with edges:

  `Edge(edge_id, source, target, directed, e_attr_1, e_attr_2, ..., e_attr_`m`)`

  for $m \in \mathbb{N}$. *source* and *target* are node identifiers (foreign keys into the **Node relation**). The *directed* attribute was added as a special attribute for two reasons: First, it is a good example of what is meant by attributes and second, the *Gravisto* graph data structure on which the *QUOGGLES* system is built regards it as a special attribute as well. The *e_ attr_ i* (for $i \in \{1, 2, \ldots, m\}$) are edge specific attributes like arrow type, form, bends, etc.

Edge	edge_id	source	target	directed	transportation
	e1	n1	n2	true	car
	e2	n1	n3	true	train
	e3	n1	n5	true	train
	e4	n2	n1	true	car
	e5	n2	n2	true	on foot
	e6	n3	n2	false	ferry
	e7	n3	n5	true	bike
	e8	n3	n4	true	plane
	e9	n4	n5	false	plane
	e10	n4	n6	true	ship
	e11	n5	n6	false	plane

Table 4.2: Example Edge Relation

Tables 4.1 and 4.2 give an example instance of the given relational data base schema. The graph saved this way is shown in Figure 4.1. Nodes and edges do not have common attributes. Thus, the **GraphElement relation** is empty and omitted.

4.1.3 Simulation of Relational Algebra Operations

In this section, it is shown how all operations that relational algebra offers can be simulated by the *QUOGGLES* system. In this context, simulation means that the result of applying the specified relational algebra operator is equivalent to the result of applying the constructed query in the *QUOGGLES* system. Equivalence can of course not necessarily mean syntactical equality. Since the *QUOGGLES* system does not build on any relational model of graphs, no relations are present in its data model. Thus, names of tables and attributes, for instance, only appear in relational algebra and are ignored elsewhere.

To be able to compare results, the result of a query in the *QUOGGLES* system is brought into tabular form aiming at a syntactical equivalence of the rows of these tables with the tuples of the result relation.

Data in the *QUOGGLES* system is present in the form of single objects or collections (containing single objects and / or collections themselves). The table view of such an object is defined inductively as follows.

First recall a few standard naming conventions for tables: A table can be partitioned into **rows** (horizontally) or **columns** (vertically). The intersection of a column and a row in a table is called a **cell**.

Definition 4.5 (Table View)

Let o be an arbitrary object.

If o is a collection, the **table view** of o is a table with one column: Let n denote the number of elements in o. For each $i \in \{1, 2, \ldots, n\}$, the i^{th} cell of that column contains a string representation of the i^{th} element of the collection. Thus, the order of the cells within the column is the order of the collection (if it defines one).

If o is a relation, the **table view** of o is a table: Let n be the number of tuples and r be the arity of the relation. The resulting table has n rows and r columns. Let $c_{i,j}$ be the cell that is the intersection of row i and column j. Then the value of $c_{i,j}$ is a string representation of the j^{th} component in the i^{th} tuple of the relation o.

Otherwise, the **table view** of o consists of a table with only one cell, containing a string representation of o. □

Introducing this view to the *QUOGGLES* system means that boxes that work on collections need to be be extended:

- `ListOperations1_Box`
  For values *flatten, reverse* and *remove empty*, the box remains unchanged. On the other hand, the operations *make distinct* and *unpack* are extended and another two are added (*listify* and *de-listify*). For the latter four operations, a second parameter is added to the box.

  1. **Parameter 1:** *listify*

     Inputs 1 to *n*: n collections. n is the value of the second parameter.

     Output: A list of tuples (lists) holding the elements of all input collections.
     The idea of this box is to view the input collections as n columns of a table (also refer to Definition 4.5 of the ***table view***). The output collection is a list of all rows of that table. More formally:
     Let s be the size of the largest of all n input collection. The result of applying this box operation is a list of s n-tuples (which are represented by lists that have n elements). The i^{th} n-tuple consists of the i^{th} elements of all n collections. If a collection does not have enough elements, the special value `QConstants.EMPTY` is inserted.

     Parameter 2: An integer n specifying the number n of inputs of the box.

     Figure 4.2 illustrates a use of this box.

2. **Parameter 1:** *unpack*
   This parameter value has already been described on page 45. The input has been restricted to collections of size one. Now the functionality is extended using a parameter to specify the number of elements (and therefore the number of outputs).

   Input: A collection holding n elements. n is the value of the second parameter.

   Outputs 1 to *n*: For $i \in \{1, 2, \ldots, n\}$, output i will consist of the i^{th} element of the input collection. n is the value of the second parameter.

   Parameter 2: An integer specifying the number n of inputs of the box.

   This box converts a collection of n elements into a relation with arity n containing exactly one tuple. In conjunction with a `ComplexFilter_Box`, it can be used revert the conversion from a relation to a collection of tuples done by a `ListOperations1_Box(listify,` n`)`. Since it is needed quite often this way, this functionality is assigned to the special parameter *de-listify* in the `ListOperations1_Box` described next.

3. **Parameter 1:** *de-listify:*

   Input: A collection holding n elements. All elements must be collections of the same size s. s must match the value of the second parameter.

   Outputs 1 to *n*: For $i \in \{1, 2, \ldots, s\}$, output i will be a list of all i^{th} elements of all n input collections.

   Parameter 2: An integer s specifying the number of outputs of the box.
   This box reverts the operation of a `ListOperations1_Box(listify,` s`)`. The given list of rows is converted back to a list of s columns, each holding n elements.

   Having this parameter, this box becomes a compound box. The operation associated with it can be simulated easily: Let $C = [c_1, c_2, \ldots, c_n]$ be the input collection. Each element c_i is a collection of size s. A `ComplexFilter_Box` is used that takes C as its input. The first output is not of interest. Therefore a `End_Box` is connected to it. At its second (predicate) output, a `ListOperations1_Box(unpack,` s`)` is attached. This box's outputs yield the desire result. The three boxes are saved as a subquery *sub*. The `SubQuery_Box(sub)` using *sub* as parameter has one input and s outputs and produces the desired result.

   Figure 4.2 shows how this box reverts the operation of a `ListOperations1-_Box(listify,` n`)`.

4. **Parameter 1:** *make distinct*

   Inputs 1 to *n*: n collections of the same size l. n is the value of the second parameter.

   Outputs 1 to *n*: n collections of the same size l. n is the value of the second parameter.
   All duplicates in the input are removed. For one input collection, the meaning is straightforward and has been explained on page 28.
   To see what is meant by "duplicates" for several inputs, suppose there are n input collections (i.e. the value of the second parameter is n). These collections must all be of the same size l.
   Consider all i^{th} elements of the n collections. For $i \in \{1, 2, \ldots, l\}$, these form l n-tuples. From the list of those tuples $\{n_1, n_2, \ldots, n_l\}$, all duplicates are removed.[1]

[1] Since duplicates are defined via the standard Java `boolean Object.equals(Object obj)` method, this means that two tuples are equal if and only if the i^{th} element of the first is equal to the i^{th} element of the second tuple.

Afterwards, the list of remaining tuples is converted back to n collections: The i^{th} element in the j^{th} tuple is the j^{th} element in collection number i. See also the definition of the ***table view*** on page 57.

Parameter 2: An integer n specifying the number of inputs (and outputs) of the box.

The function of this box can be more easily described by showing the way it can be simulated: The inputs represent a relation. This can be seen as a table. After converting this table of n columns to a list of s rows using a `ListOperations_Box(listify,` n`)`, the basic version of `ListOperations1_Box(make distinct)` introduced on page 28 removes any duplicate rows. A `ListOperations1_Box(de-listify,` n`)` converts this list back to a table, i.e. a relation consisting of n components.

This sequence is illustrated in Figure 4.2.

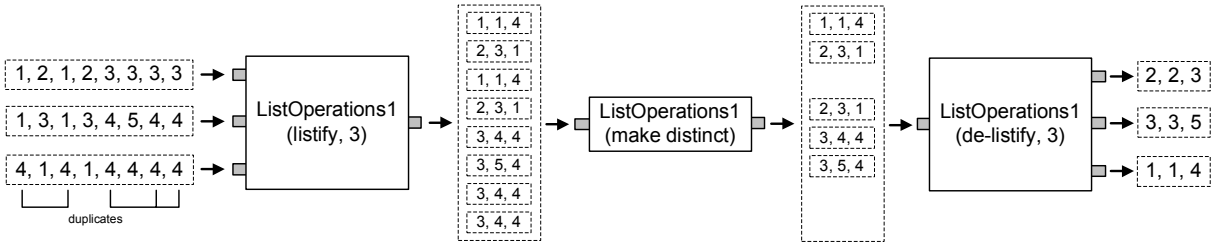


Figure 4.2: Simulation of a `ListOperations1_Box(make distinct,` n`)`

- `ListOperations2_Box`
  The basic version of this box has been introduced on page 30.

  Inputs 1 to *r*: r ***homogenous*** collections of equal size, all having the same ***homogeneous type***.

  Inputs *r+1* to *2·r*: r ***homogenous*** collections of equal size having the same ***homogeneous type*** as the collections at input 1.

  Outputs 1 to *r*: r ***homogenous*** collections of equal size, all having the same ***homogeneous type*** as the input collections.

  Parameter: The same as the basic version: *union*, *intersect* or *list minus*

  All operations are extended to operate on collections of tuples. Thus, a second parameter is added with which the user can specify the desired number of inputs. The new function of the box is to apply one of the operations *union*, *intersect* or *list minus* on two relations of equal arity r. The number of inputs must therefore be $2r$ and is thus always an even number. The number of outputs is r.

  The result of a `ListOperations2_Box(`*par*`,` $2r$`)` can also be obtained by compressing the two input relations (i.e. r collections each) into one list each by a `ListOperations1_Box(listify,` r`)`. The two collections produced this way are then processed by a basic `ListOperations2_Box(`*par*`)`. The result must then be converted back to r collections. This is done by a `ListOperations1_Box(de-listify,` r`)`.

  This sequence is shown in Figure 4.3.

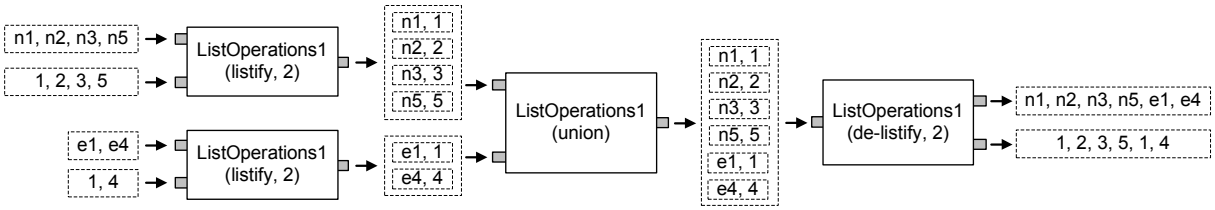


Figure 4.3: `Simulation of a ListOperations2_Box(union, 2 · 2)`

- `SortBy_Box`
  A new box is introduced called `SortBy_Box` that copes with collections of tuples instead of collections of objects. This box replaces the less powerful `Sort_Box` presented on page 31. It cannot be simulated in the same way as the precedent boxes because the `ListOperations1_Box(listify,` n`)` would yield a collection for which an appropriate sorting criterium lacks definition.

  The box is designed in a very general way: It takes one relation as input that is reordered according to the permutation resulting from sorting the second input relation. Thus, it can be used to simulate the `SQL` keyword `ORDER BY` in Chapter 4.2. When both inputs are one and the same collection, this box can be replaced by the basic `Sort_Box`.

  Inputs 1 to *s*: s collections of the same size representing the first relation. Those columns will be reordered. s is the value of the first parameter.

  Inputs *s* + *1* to *n*: $n - s$ collections of the same size representing the second relation. According to those, the first s input collections will be reordered. The elements of each collection must all be of the same type. s and n are the values of the first and second parameter, respectively.

  Outputs 1 to *s*: The first s input collections reordered according to the collections $s+1$ to n. s and n are the values of the first and second parameter, respectively.

  Parameter 1: An integer s specifying the number of input collections that are going to be reordered.

  Parameter 2: An integer a specifying the number of input collections used to get the permutation according to which the first s input collections are reordered.

  The box has $s + a$ inputs, where s and a are the values of the two parameters. It works as follows:

  The second $n - s$ input collections are treated as columns of a table. This table is sorted according to the first column. If some elements in the first column are considered equal, these are sorted according to the values in the second column and so on. The permutation π used to sort this table is saved.

  Permutation π is then applied to all of the first s input collections. The results are the outputs of the box.

Examples 4.1:

- If $C = \{c_1, c_2, \ldots, c_r\}$ is a set of collections forming a relation, this relation can be sorted by giving a `SortBy_Box(`r`, `r`)` twice the set as input: For $i \in \{1, 2, \ldots, r\}$, c_i will be provided to inputs with index i and $2 \cdot i$.
- As another example, consider a relation the first components of which are nodes and the second their labels. This can easily be sorted according to the degree of the nodes by using a `SortBy_Box(2, 1)`. Input one will be the collection of nodes and input two the collection of labels. Input three gets the output of a `GetProperty_Box(degree)` that got the collection of nodes as input. The `SortBy_Box(2, 1)` will have the sorted collections of nodes and labels as output.
- Figure 4.4 shows an example use of this box. Elements of larger collections are displayed horizontally to save space. The task is to sort the set of nodes of the example graph drawn in Figure 4.1 first according to their degree and then, i.e. if two nodes have the same degree, according to their out-degree. The parameters specify that one collection (first parameter, "1") is going to be sorted according to two other collections (second parameter, "2"). The first input is the set of nodes represented by their labels. Two `GetProperty_Box`es generate the list of degrees and out-degrees, respectively. The three collections are the input to the `SortBy_Box(1, 2)`. The result is a list containing the elements from the first input sorted according to the specification. In the figure, the degree and out-degree of each node is displayed below the node labels. It can be seen that the degrees have been sorted. There are three nodes with degree four. The node labelled "Graz" is the first of them in the list since its out-degree is smaller than that of the other ones.

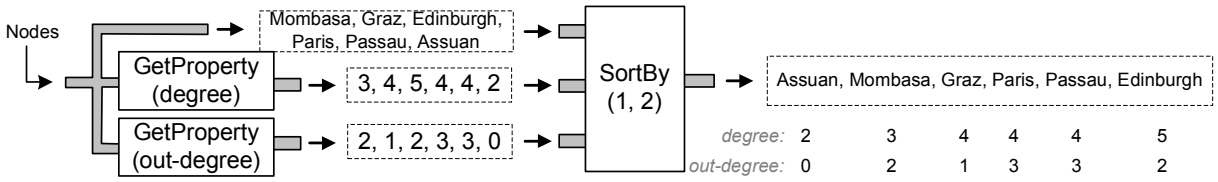


Figure 4.4: `SortBy_Box`

The reason why the collections according to which the input relation is sorted must each consist of elements of the same type has been explained in the description of the `Sort_Box` on page 31. The possibility mentioned and implemented there to choose the sorting operation according to the type of the elements is too expensive here. Thus, the input type restriction is sensible.

The next important step is to define what relational algebra means in this context.

Definition 4.6 (GraphRelAlg)

GraphRelAlg is the language for which holds:

- The three relations n, e and ge (specified by the **Node-**, **Edge-** and **GraphElement relation**s) defined in Section 4.1.2 are the basic objects of the language ***GraphRelAlg***.
- Any relation that can be formed from n, e, ge using the 6 operations defined for relational algebra is in the language ***GraphRelAlg***. □

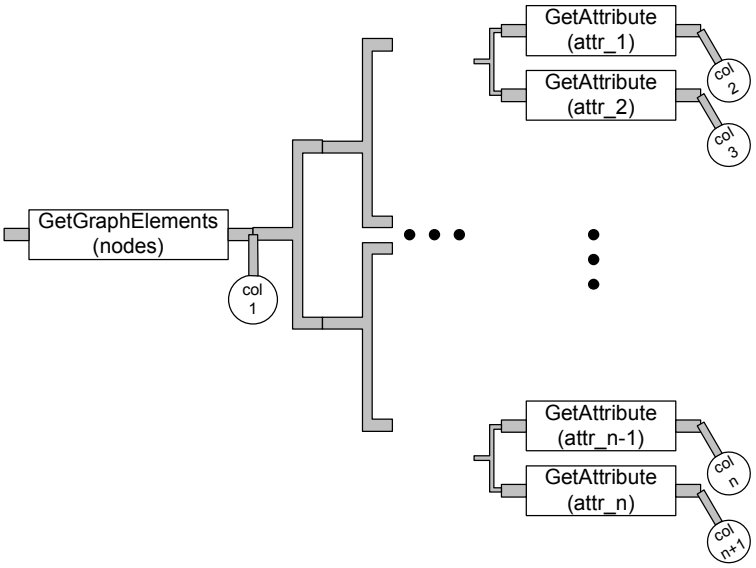


Figure 4.5: Simulation of the Node Table.

The first action in showing that the *QUOGGLES* system subsumes ***GraphRelAlg*** is to demonstrate that the three basic objects (i.e. relations) can be produced. The only deviation (in the present implementation) is that node and edge identification numbers are not explicitly represented. The objects that are used are implicitly identified as unique objects.

The relation

`Node(node_id, n_attr_1, n_attr_2, ..., n_attr_`n`)`

is the result of the query shown in Figure 4.5. Instead of using $n-1$ `TwoSplitConnector_Boxes`, a `SplitNConnector_Box(n-1)` that reproduces its input $n-1$-times would reduce the number of required boxes by about a half. But, as stated before, the purpose here is to show that a small set of standard boxes suffices.

The relation

`Edge(edge_id, source, target, directed, e_attr_1, e_attr_2, ..., e_attr_`m`)`

is the result of the query shown in Figure 4.6.

The relation

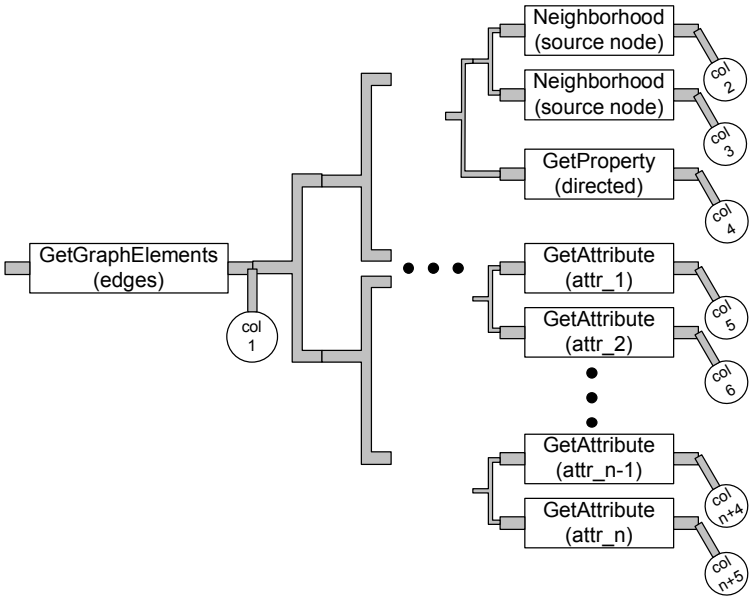


Figure 4.6: Simulation of the Edge Table.

`GraphElement(node_or_edge_id, ge_attr_1, ge_attr_2, ..., ge_attr_`k`)`

can be built in the same way using parallel composition of the two previously introduced queries and is therefore not explicitly demonstrated.

Since the notion of relations is not used in the model of the *QUOGGLES* system, the next definition helps interpreting the results of the *QUOGGLES* system as a set of relations.

Definition 4.7 (Relating Box)

For a box to be **relating** (i.e. a **relating box**), it must have at least one input and one output. Additionally, one of the following two conditions must hold for every possible input(s) and every possible value(s) of its parameter(s):

1. If at least one input is a collection of size $s > 1$, then

- all inputs must be collections of the same size s (observe that the type of elements is not important) and
- all outputs must be collections of the same size s as the input collections and
- for all $i \in \{1, 2, \ldots, s\}$ there must be a logical link between all i^{th} elements of the input and output collections.

2. If at least one input is a collection of size one or not a collection at all, then

- no input is a collection of size greater than one and
- no output is a collection of size greater than one and
- there must be a logical link between all input and output elements. □

Node 1	4
Node 2	
Node 3	
Node 4	

Table 4.3: A "Relation" Produced by a Non Relating Box

This definition results from the observation that the output of a non ***relating box*** cannot be part of the same relation as the inputs. Consider for example the `SizeOf_Box`, that produces an integer n representing the number of elements in the input collection. The ***table view*** of the input is a column with n entries. The ***table view*** of the output is a column with just one cell. Relational algebra can never produce such a table. In fact those two columns cannot be put into one and the same relation. The only way to do this is to add special values (`QConstants.EMPTY`) to the smaller column. This would, however, add a relationship between some values of the first column and those non `QConstants.EMPTY` values of the second. This relationship has not been intended by the box, since otherwise, it would have had to take care to insert some kind of `QConstants.EMPTY` values itself.

The third item in the two cases assures that a box really represents a relation between the input and the output. A box that produces some entirely unrelated, maybe random output is not treated as a ***relating box***. No sequence of operators of relational algebra could ever generate such a result.

The consequence is that whenever a *non* ***relating box*** appears, subsequent output is put into a different relation. This is important since applying the selection operation on a relation like the one shown in table 4.3 would yield unexpected and absurd results.

This definition is influenced by the ***table view*** of the inputs: The ***table view*** of a collection holding one element is equal to the ***table view*** of the element itself: It consists of a table having one cell containing the string representation of the element. This is the reason why the definition treats inputs that are collections of size greater than one specially.

Definition 4.8 (State of a Box)

The **state** of a box is a list of parameter values of the box's parameters. □

Definition 4.9 (Relating Configuration)

A **relating configuration** of a box b and its state s, in short $\mathbf{rc(b,s)}$, is a list of pairs that state for which combinations of indices b is a ***relating box***. A pair can either be of the form $(\mathrm{i}_k, \mathrm{o}_l)$, where k is an input index and l an output index, or $(\mathrm{o}_k, \mathrm{o}_l)$ where k and l are both output indices ($k \neq l$). If this configuration does not depend on the state of the box, the state is omitted and the **relating configuration** is written as $\mathbf{rc(b)}$. □

Definition 4.10 (Input / Output Sensitive Relating Box)

A box b with state s, *in* inputs and *out* outputs is **input / output sensitive relating** if

$$\emptyset \neq rc(b,s) \neq \{(\mathrm{i}_k, \mathrm{o}_l) \mid k \in \{1,2,\ldots,in\}, l \in \{1,2,\ldots,out\}) \cup \{(\mathrm{o}_k, \mathrm{o}_l) \mid k,l \in \{1,2,\ldots,out\})$$

i.e. if the restriction to some combination of inputs and outputs does yield a ***relating box*** and some other restriction not. □

Definition 4.11 (Parameter Sensitive Relating Box)

A box is said to be **parameter sensitive relating** if it depends on the values of (some of) its parameters (i.e. the state of the box) whether or not it is ***relating***. □

Examples 4.2:

- A `GetAttributeValue_Box` *gav* (see page 24) is ***relating*** since every input `Attributable` produces one output value. It has one input and one output. Thus, its ***relating configuration*** is:

$$rc(gav) = [(\mathrm{i}_k, \mathrm{o}_l) \mid 1 \leq k \leq \#inputs, 1 \leq l \leq \#outputs] = [(\mathrm{i}_1, \mathrm{o}_1)]$$

- Another ***relating*** box is the `TwoSplitConnector_Box` *tsc* since it just duplicates its input. It has one input and two outputs. Thus

$$rc(tsb) = [(\mathrm{i}_1, \mathrm{o}_1), (\mathrm{i}_1, \mathrm{o}_2)]$$

- A `ValueCompare_Box` *vcb* having one input and one output is *not* ***relating***, implicating

$$rc(vcb) = [\,]$$

  The reason is that two collections of size larger than one as input produce only one single value as output. This contradicts the definition of a relating box.

- A `ComplexFilter_Box` *cfb* relates the input (only) with its second output. Therefore,

$$rc(cfb) = [(\mathrm{i}_1, \mathrm{o}_2)]$$

  where the tuple $(\mathrm{i}_1, \mathrm{o}_1)$ is missing. This is an ***input / output sensitive relating*** box. No filter boxes can be ***relating*** since some input elements might be missing in the output.

- A `ListOperations2_Box(`*par*`,` *n*`)` *lob* has n inputs and $n/2$ outputs (n must be an even number). It is ***input / output sensitive relating*** since there are no input indices in its ***relating configuration***, independent of the values of the parameters:

$$rc(lob) = [(\mathrm{o}_k, \mathrm{o}_l) \mid 1 \leq k \leq n/2, 1 \leq l \leq n/2]$$

- An example for a ***parameter sensitive relating*** box is a `SubQuery_Box`. Depending on the value of the parameter (i.e. the file name of a saved subquery graph), the box can be ***relating***, ***input / output sensitive relating*** or even *non* ***relating***.

- A box b with two inputs, one output and

$$rc(b) = [(\mathrm{i}_1, \mathrm{o}_1), (\mathrm{i}_2, \mathrm{o}_1)]$$

is not part of the set of basic boxes. A sensible example for such a box is one that can be used for network flow graphs. Given a list of capacities of edges as first input and a list of the current amount of flow of the same edges, the output is a list of the currently remaining capacity of those edges.

Table 4.4 shows the ***relating configuration*** of all basic and compound boxes. Boxes without any inputs or outputs are omitted since their ***relating configuration*** is always empty. ***Parameter sensitive*** boxes are presented in tables 4.5 to 4.7. The `Arithmetic_Box` (table 4.5), for example, is *not* ***relating*** for all its aggregate operations and ***relating*** for its arithmetic operations. The `SubQuery_Box` cannot be categorized since it allows an infinite number of different parameter values.

Box	Relating Configuration	#in-/outputs
`AttributeFilter_Box`	$rc = [\,]$	1/1
`BooleanNot_Box`	$rc = [(i_1, o_1)]$	1/1
`BooleanOp_Box`	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
`CompareTwoValues_Box`	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
`ComplexFilter_Box`	$rc = [(i_1, o_2)]$	1/2
`Contains_Box`	$rc = [(i_2, o_1)]$	2/1
`Equals_Box`	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
`GetAttributeValue_Box`	$rc = [(i_1, o_1)]$	1/1
`GetGraphElements_Box`	$rc = [\,]$	1/1
`GetProperty_Box`	$rc = [(i_1, o_1)]$	1/1
`InducedSubgraph_Box`	$rc = [\,]$	1/1
`InterpretAsBoolean_Box`	$rc = [\,]$	1/1
`Instanceof_Box`	$rc = [(i_1, o_1)]$	1/1
`Neighborhood_Box`	$rc = [(i_1, o_1)]$	1/1
`OneOneConnector_Box`	$rc = [(i_1, o_1)]$	1/1
`SizeOf_Box`	$rc = [(i_1, o_1)]$	1/1
`Sort_Box`	$rc = [\,]$	1/1
`SortBy_Box`	$rc = [\,]$	m+n/m
`TrueMaker_Box`	$rc = [\,]$	n/1
`TwoSplitConnector_Box`	$rc = [(i_1, o_1), (i_1, o_2)]$	1/2
`ValueCompare_Box`	$rc = [\,]$	1/1
`ValueFilter_Box`	$rc = [\,]$	1/1

Table 4.4: ***Relating Configuration***s.

Before being able to present the algorithms that show that the *QUOGGLES* system is indeed able to simulate any operation from relation algebra, some basic functions are defined which are used in the algorithms that follow.

Parameter	Relating Configuration	#in-/outputs
sum	$rc = [\,]$	1/1
count	$rc = [\,]$	1/1
min	$rc = [\,]$	1/1
max	$rc = [\,]$	1/1
avg	$rc = [\,]$	1/1
plus	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
minus	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
times	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1
divide	$rc = [(i_1, o_1), (i_2, o_1)]$	2/1

Table 4.5: Relating Configuration of the `Arithmetic_Box`.

Parameter 1	Parameter 2	Relating Configuration	#in-/outputs
flatten	-	$rc = [\,]$	1/1
reverse	-	$rc = [\,]$	1/1
remove empty	-	$rc = [\,]$	1/1
make distinct	n	$rc = [\,]$	n/n
unpack	-	$rc = [(i_1, o_1)]$	1/1
listify	n	$rc = [(i_k, o_1) \mid k \in \{1, 2, \ldots, n\}]$	n/1
listify	n	$rc = [(i_k, o_1) \mid k \in \{1, 2, \ldots, n\}]$	n/1

Table 4.6: Relating Configuration of the `ListOperation1_Box`.

Definition 4.12 (Basic Functions)

In this definition, types are written in capital letters and should be self explanatory: `LIST(A)` means a list of objects of type `A`. The return type precedes the function name. A list of parameters follows in parenthesis.

`PARAMVALUE getParamValue(BOX b, INT i)`: Given a box b, the method returns the value of the i^{th} parameter. The first parameter has index 1.

`VOID setParamValue(BOX b, INT i, OBJECT o)`: Given a box b, the method sets the value of the i^{th} parameter to the value of o. The first parameter has index 1.

`(GRAPH, BOX) addCondition(GRAPH G, FORMULA F)`: The formula F is transformed into a query graph G_F. The graph created by concatenating G and G_F is returned together with the first non `Output_Box` to which a path from the input box in G exists. This is described in more detail on page 69.

`LIST(EDGE) getOutedges(BOX b)`: Returns all edges that have box b as source box (all edges in the query graph are directed).

`VOID deleteBox(GRAPH G, BOX b)`: Deletes the given box b from graph G,

`BOX addBox(GRAPH G, BOX b)`: Adds box b to the graph G and returns the added box. Returning the box is necessary since some implementations might add a copy of the given box b.

Parameter 2	Parameter 1	Relating Configuration	#in-/outputs
union	2n	$rc = [(o_k, o_l) \mid 1 \leq k \leq n, 1 \leq l \leq n]$	2n/n
intersect	2n	$rc = [(o_k, o_l) \mid 1 \leq k \leq n, 1 \leq l \leq n]$	2n/n
list minus	2n	$rc = [(o_k, o_l) \mid 1 \leq k \leq n, 1 \leq l \leq n]$	2n/n

Table 4.7: Relating Configuration of the `ListOperation2_Box`.

`EDGE addEdge(GRAPH G, BOX b1, INT i1, BOX b2, INT i2)`: Adds an edge between boxes b_1 (at output with index i_1) and b_2 (at input with index $_2$) in graph `G` and returns the added edge. Index counts start at 1.

`VOID retargetEdge(GRAPH G, EDGE e, BOX b, INT i)`: Changes the target box of the given edge e to b at input number i. If that is not possible in the data structure used, the following steps must be executed instead:

```
BOX source = e.getSource();
INT o = getOutIndex(e);
deleteEdge(e);
addEdge(G, source, o, b, i);
```

Thus, the edge is replaced by an edge having the same source but the new target. □

Next, the 6 operations defined for relational algebra have to be examined in some detail. For this, let R and S represent two relations in ***GraphRelAlg*** of arity r and s, respectively.

Operation 1: Renaming:

As stated before, table and columns names are not represented in the graph data model and therefore do not appear in the *QUOGGLES* system. In relational queries where renaming is not a mere output beautification, it may be used to distinguish several instances of one and the same table. This is implicit in the *QUOGGLES* system since every box gets its input independently from all others.

Operation 2: Selection

It must now be shown that the selection operation of relational algebra can be simulated. The general algorithm is followed by an illustrated example.

In this chapter, it is assumed, without loss of generality, that all `Output_Box`es do not have any out-edges. Therefore, their output is freely available. If an `Output_Box` had an out-edge, a `TwoSplitConnector_Box` would be added between the `Output_Box` and its out-neighbor to provide the free output.

Definition 4.13 (Current Relation)

The selection condition cannot use entries from several relations. The relation on which the condition is applied is called the **current relation**. □

Let $\sigma_C(R) = \sigma_{cond(\{c_1,c_2,\ldots,c_n\})}(R)$ be the selection operation. The set $\{c_1, c_2, \ldots, c_n\}$ specifies the set of indices of the components of the ***current relation*** R that are used within the condition statement. The condition is a formula using constants (numbers and strings) and built-in comparison operators $(=, \neq, >, \ldots)$, concatenating those via boolean operators $(\vee, \wedge)$.

Transformation of a Selection Condition into a Subquery

A selection condition is constructed using comparison and boolean operators forming a boolean formula. To ease the conversion, this formula is assumed to be in disjunctive normal form and simplified as much as possible. This implies that no comparison uses two constant values, the boolean values *true* and *false* do not occur and any boolean operator that is not a negation, $\vee$ or $\wedge$ operator, is equivalently expressed using only those three operations.

All non constant values occurring in the condition are references to some component in the ***current relation***. Let o be such a reference. In the *QUOGGLES* system, the data referred to by o is produced by an `Output_Box` b_o and is directly accessible. If o is needed several times, a corresponding amount of `TwoSplitConnector_Box`es is added to replicate it.

In the following description, a statement like "*o is the input of box b*" is an abbreviation for: Add an edge between the output of the box that produces o and the input of box b.

The conversion to a query graph is straightforward. It is done in two steps:

1. Convert the comparison operators:

   Let a comparison be the form $i_1 \, \varrho \, i_2$. A `CompareTwoValues_Box(`*par*`,` ϱ`)` *ct* is added to the query.

   If i_1 is a constant value, a `Constant_Box(`*type*`,` i_1`)` is used that provides the first input to *ct*. The parameter `type` specifies the type of i_1.

   If i_1 is not a constant value, it is directly used as first input of *ct*.

   The same holds for i_2 and the second input of *ct*. The parameter `par` must be the type of the i_1 and i_2.[2]

2. Convert the boolean formula:

   For the boolean negation operator, the `BooleanNot_Box` is used. A `BooleanOp_Box(`ϱ`)` simulates the binary operations where $\varrho \in \{\wedge, \vee\}$. The order in which the operations are converted follows the evaluation order of the formula, i.e. takes parentheses into account.

Unfortunately, this conversion produces a large graph. Besides taking up much space in the graphical representation of the query graph, the amount of edges will very probably render the graph quite confusing. It might be worthwhile to construct a box that better supports the display of a boolean formula. As a start, a box that combines an arbitrary amount of inputs using one

[2] The two types should match for the formula to be valid. In the event of them being different, the implementation applies an implicit conversion to the specified type on both of them (if applicable). This relieves the user of the task to add several conversion boxes.

or several boolean operations would yield a much clearer layout. However, this way suffices to show the feasibility of the approach.

Example 4.3: Consider the boolean formula

$$((o_1 > 10) \vee (o_1 = o_3)) \wedge (o_2 = 2)$$

where o_1, o_2 and o_3 are components of the ***current relation***. Figure 4.7 shows the query graph that represents this formula (the final `BoolPredicateEnd_Box` has been omitted).

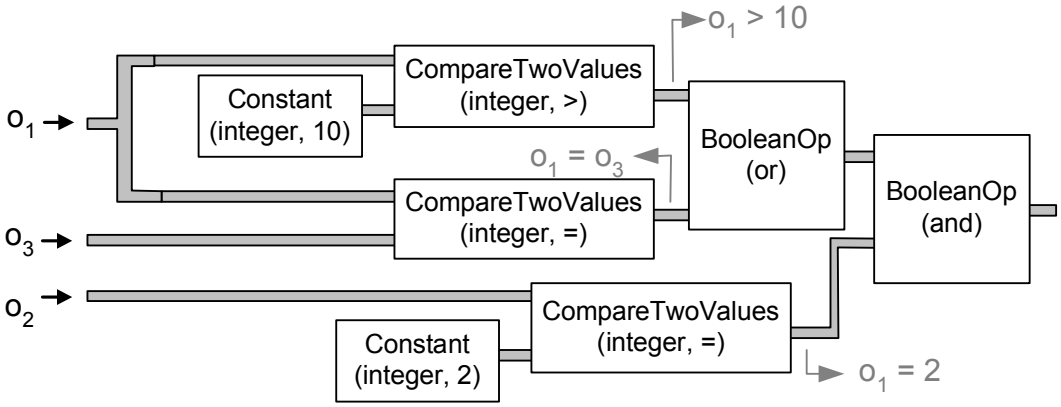


Figure 4.7: Graph Representation of an Example Boolean Formula

Simulation of the Selection Operation

Let r be the arity of relation R. There must be r `Output_Box`es $\{o_1, o_2, \ldots, o_r\}$ in the query graph that are responsible for creating the components of R.

Five steps are required to generate the new query graph:

1. A `ListOperations1_Box(listify,` r`)` ll is added to the query. Let $\{(b_1, o_1), (b_2, o_2), \ldots (b_r, o_r)\}$ a set of pairs specifying the box and output index of its out-edge for each of the `Output_Box`es. Remove all `Output_Box`es. For every $i \in \{1, 2, \ldots, r\}$, an edge is added from box b_i at output o_i to the i^{th} input of ll.

2. An edge is added that connects the output of ll with the input of a new `ComplexFilter_Box` cb.

3. At the first output of cb, an edge is added to a new `ListOperations1_Box(de-listify,` r`)` called ld. This reverts the operation of ll for all tuples that passed the predicate of cb.

4. An `Output_Box` is added at each of the r outputs of ld. These will generate the desired result.

5. The `ComplexFilter_Box` *cb* offers the possibility to add a predicate subquery at its second output. This subquery represents the selection condition C as has been described above. To generate the inputs for that condition subquery, a `ListOperations1_Box(unpack, r)` *lu* is added and connected to the second output of *cb*. During execution of the query, *lu* always gets one tuple of R (i.e. a list of r elements) as input. *lu* then distributes the elements of that tuple to r outputs. Not all outputs need to be used by the predicate subquery since only those components that are referenced in the selection condition C are of interest. In those cases, it is important that the predicate subquery is terminated by a `BoolPredicateEnd_Box` (see page 27).

The section about simulating a selection operation is concluded with an example:

Example 4.4: Let R be a relation of arity $r = 3$. The three components are named o_1, o_2 and o_3. To simulate the selection operation $\sigma_C(R)$, the condition C is transformed into a query graph G_{cond}. Then, the five steps just described are carried out and G_{cond} is added, creating the graph shown in Figure 4.8. The three arrows mark the outputs where the result of the selection operation can be accessed. The condition is the same used in the previous example (shown in Figure 4.7).

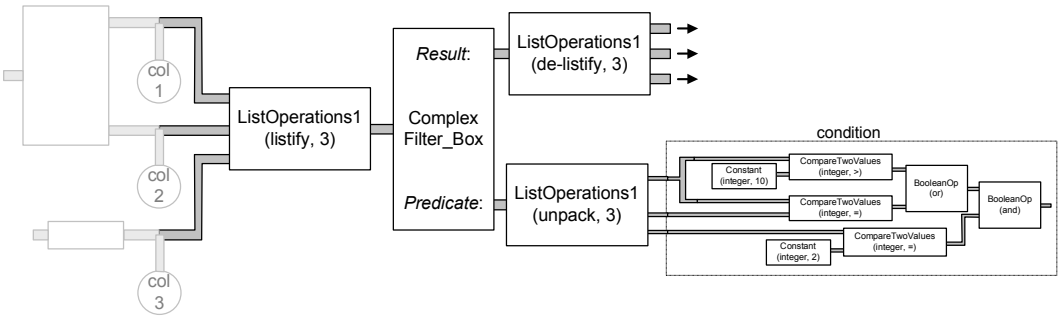


Figure 4.8: Simulation of an Example Selection Operation.

Operation 3: Projection

Let R be a relation with arity r. The projection $\pi_{c_1,\ldots,c_m}(R)$ of R onto components $c_1, \ldots, c_m$ (with $\forall i : 1 \leq c_i \leq r$ and $c_i \neq c_j$ for $i \neq j$) in relational algebra means choosing columns $c_1, \ldots, c_m$ (in this order) from relation R.

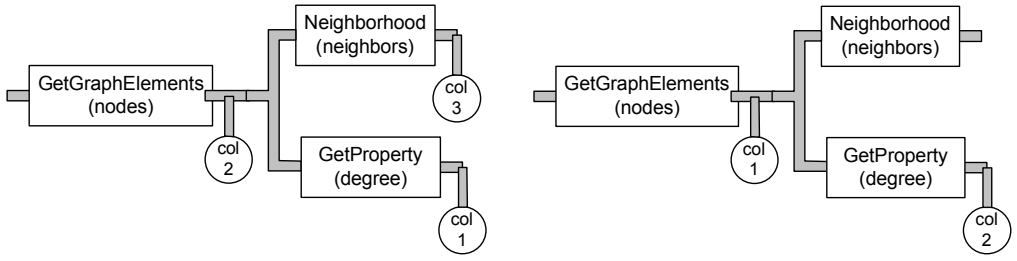
There is no combination of primitive boxes in the *QUOGGLES* system that could access, choose and reorder components (list elements) of an arbitrary relation (represented for example by a set of lists). However, since the representation of relations in the *QUOGGLES* system is designed in a special way, the projection operation can be simulated:

Let t_R be the ***table view*** of R. In the *QUOGGLES* system, the columns in t_R are determined by a special set of boxes (the `Output_Box`es). Therefore, no box has to cope with an input representing an entire relation. For the ***table view***, projection means to choose and reorder columns. Since these columns are produced by the `Output_Box`es, the projection can easily be managed:

All `Output_Box`es which are responsible for columns that are *not* used by the projection (i.e. those with an index $i \in (\{1, \ldots, r\} \setminus \{c_1, \ldots, c_m\})$ are removed from the query graph. The column parameters of the remaining boxes are adjusted to reflect the new order specified by the projection operation: The parameter of the box previously responsible for column i must be set to $\pi(i)$, where π denotes the permutation of the projection operator.

Informally speaking, the idea is to leave discarded information and order columns correctly right from the start instead of creating a relation and then projecting it onto some of its components.

The procedure that manages this transformation is presented as algorithm 4.1.



col. 1	col. 2	col. 3
4	Passau	[Paris, ...]
4	Graz	[Paris, ...]
4	Paris	[Edinburgh, ...]
2	Assuan	[Graz, ...]
3	Mombasa	[Paris, ...]
5	Edinburgh	[Paris, ...]

col. 1	col. 2
Passau	4
Graz	4
Paris	4
Assuan	2
Mombasa	3
Edinburgh	5

Figure 4.9: Simulation of an Example Projection Operation

Example 4.5: The query graph on the left hand side in Figure 4.9 produces the three columns shown in the table below the graph. For each node, its degree, its string representation and the list of its neighbors is calculated. This forms a relation R of arity three that has six tuples.

The projection operation $\pi_{2,1}(R)$ means selecting components two and one (in that order).

The application of algorithm 4.1 yields the query graph shown to the right of the original graph. Below that, the result is shown in a table.

Algorithm 4.1 Simulate a Projection Operation

Input: Graph G, projection specification as a list of indices $[c_1, c_2, \ldots, c_n]$
Output: Graph G' simulating G with the projection operation applied to it

$G' \leftarrow$ copy of G;

/* get `Output_Box`es that are of interest */
$outs \leftarrow \{o \mid o$ is a `Output_Box`in $G\}$;
$outs_{proj} \leftarrow \{o \in outs \mid \text{getParamValue}(o, 1) \in [c_1, c_2, \ldots, c_n]\}$;

/* remove all columns that are not mentioned in the projection */
for all $out \in outs \setminus outs_{proj}$ **do**
  deleteBox(G', out);
end for

/* reorder remaining columns according to projection */
for $i = 1$ to n **do**
  $out \leftarrow$ the only element in $\{o \in outs_{proj} \mid \text{getParamValue}(o, 1) = i\}$;
  setParamValue($out, 1, i$);
end for
return G';

Operation 4: Set Union:

The set union operation in relational algebra differs slightly from the same operation in set theory: It can only be applied to two relations having the same schema. The restriction that the arity of the unified relations must be the same is obvious since otherwise, the result would not be a relation any more. Since the elements of a component of any relation must also have the same type, the union operation only works on collections of the same type.

For collections in the *QUOGGLES* system, the restriction is not that tight. On reason for that is the generality of the data structure of *Gravisto*. Attributes of graph elements with different types can still have the same id (and path). This means, for instance, that one node can have a number as label and another node a string. Nevertheless, the operation of the `GetAttributeValue_Box` must still be allowed. On could argue that the union operation could work on the ***table view*** of the collections, i.e. on string representations. Type information would be lost, however. As a compromise on those two approaches, the union operation implemented in the *QUOGGLES* system defines a type compatibility as follows:

Definition 4.14 (Type Compatibility)

Two objects have **compatible types**, if they have the same type or are both instances of one of the following classes:

- `Number` (doubles, integers, ...)
- `Collection` (lists, sets, ...)
- `Attribute`
- `Attributable` (nodes, edges, ...)
- `String` □

In addition to all relations that relational algebra allows to be unified, the *QUOGGLES* system enables the union of collections that have ***compatible types***.

If applicable, the set union operation is defined in the following way: Let R and S be two relations of arity a.

$$R \cup S = \{x \mid x \in R \vee x \in S\}$$

The set union of R and S will be simulated in the *QUOGGLES* system in two steps: First of all, the tuples of R will be added to the tuples of S regardless of any duplicates (using a `ListOperations2_Box(union,` $2 \cdot a$`)`). After that, a `ListOperations1_Box(make distinct,` $2 \cdot a$`)` will be responsible for eliminating all duplicates, thus generating the desired result. The parameter a in both boxes specifies the number of inputs needed. This is the arity of the two relations R and S.

Algorithm 4.2 describes the steps needed to simulate the set union operation for two arbitrary relations in detail.

Example 4.6: The ***current graph*** is again the one shown in Figure 4.1. The query graph in Figure 4.10 produces two relations. One that relates all nodes with their label (`Output_Box`es 1 and 2) and one that associates all edges with their labels (`Output_Box`es 3 and 4).

Figure 4.11 shows the transformed graph, uniting the two relations. The ***table view*** of the result (produced by `Output_Box`es 1 and 2) consists of two columns associating all graph elements with their labels.

Operation 5: Set Difference:

Subtracting one relation S from another R, written as $R \setminus S$, removes all tuples from R that occur in S. Since R and S are relations (i.e. do not contain duplicates), there is no need to check for duplicates in the result of applying the operation.

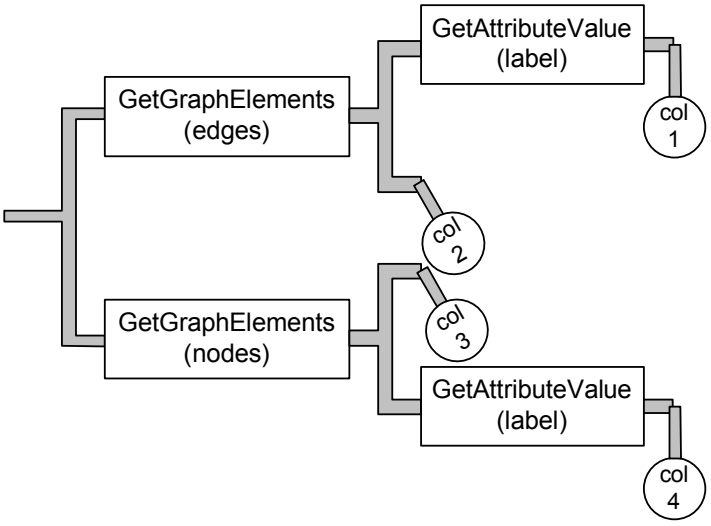


Figure 4.10: Starting Graph for Set Union Example

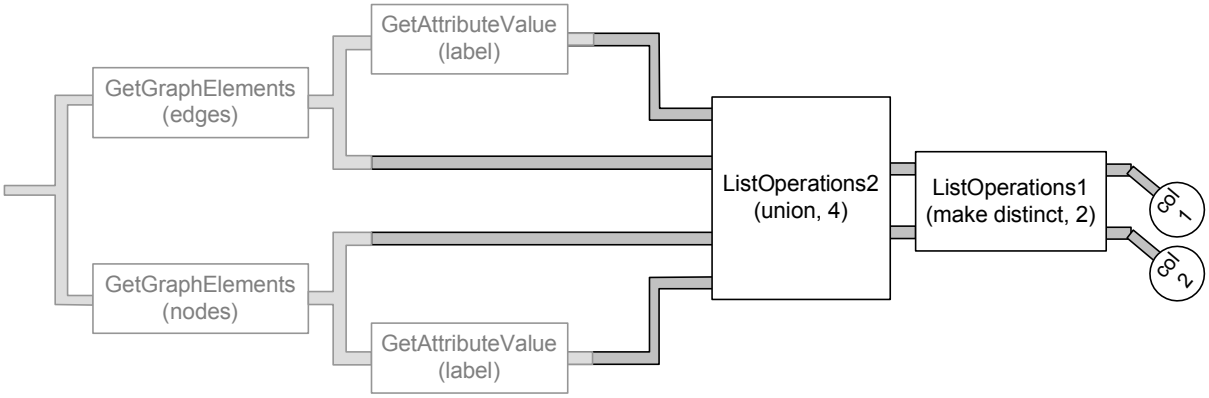


Figure 4.11: Simulation of an Example Set Union Operation

The simulation in the *QUOGGLES* system is easy since it follows the same procedure as the first step described in the simulation of operation 4 (set union). The `Output_Box`es that produce the two relations are connected to the inputs of a `ListOperations2_Box(list minus,` a`)` where a denotes the arity of R and S.

Algorithm 4.2 that shows how to simulate the set union method can be used for this operation as well. The only derivations are that it uses the parameter *list minus* instead of *union* and does not need the duplicate elimination box.

Remark: In contrast to the set union operation, $R \setminus S$ could be applied to two relations with different arity. The result will then of course always be the unchanged relation R. Since experience shows that those cases are mostly caused by some design error, the implementation of the *QUOGGLES* system assures that R and S share the same arity. This is not a restriction, however. The operation on relations with different arity can always be safely ignored.

Algorithm 4.2 Simulate the Set Union Operation

Input: Query graph G with $2 \cdot a$ `Output_Box`es representing two relations R and S of same arity a

Output: Query graph G' that produces the set union of R and S

$G' \leftarrow$ copy of G;

/* save in-edges of `Output_Box`es */
$[r_1, r_2, \ldots, r_a] \leftarrow$ list of `Output_Box`es that produce relation R;
$[s_1, s_2, \ldots, s_a] \leftarrow$ list of `Output_Box`es that produce relation S;
for all $o \in [r_1, r_2, \ldots, r_a] \cup [s_1, s_2, \ldots, s_a]$ **do**
  $edge_o \leftarrow$ in-edge of o;
end for

/* save indices for the output */
$[idx_1, idx_2, \ldots, idx_a] \leftarrow [getParamValue(r_i, 1) \mid 1 \leq i \leq a]$;

/* add all tuples of the relations together */
$unionBox \leftarrow$ new ListOperations2_Box(union, $2 \cdot a$);
addBox($G', unionBox$);

/* add a box that removes duplicates */
$distinctBox \leftarrow$ new ListOperations1_Box(make distinct, a);
addBox($G', distinctBox$);

/* connect unionBox and distinctBox; replace old `Output_Box`es */
for $i = 1$ to a **do**
  retargetEdge($G', edge_{r_i}, unionBox, i$);
  retargetEdge($G', edge_{s_i}, unionBox, i + a$);
  addEdge($G', unionBox, i, distinctBox, i$);

  /* remove output boxes that produced R and S */
  deleteBox(G', r_i);
  deleteBox(G', s_i);

  /* add output boxes to produce the result */
  $out_i \leftarrow$ new `Output_Box`(idx_i);
  addBox(G', out);
  addEdge($G', distinctBox, i, out_i, 1$);
end for
return G'

Operation 6: Cartesian Product:

Let R and S be two relations. The cartesian product of two relations means to combine each element from the first relation with each from the second, thus creating $arity(R) \cdot arity(S)$ tuples.

The simulation of this operation in the *QUOGGLES* system is demonstrated in two steps: First, it is shown how the special case that both relations have arity equal to one can be treated. Second, the general case is reduced to this simple case.

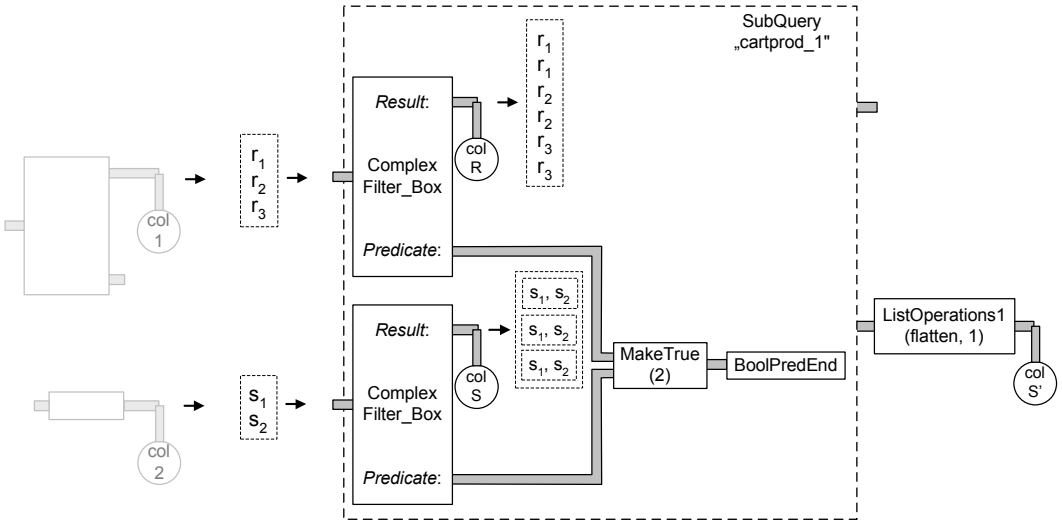


Figure 4.12: Simulation of the Cartesian Product Operator, Basic Case

Basic Case:
Let R and S be two relations, each of arity one, consisting of elements $[r_1, r_2, \ldots, r_r]$ and $[s_1, s_2, \ldots, s_s]$. Except for the trivial case where r and s are equal to one, the following must be considered:

There must be two `Output_Box`es that are responsible for creating those relations. Assume, without loss of generality, that these boxes do not have any out-edges. If they had, `TwoSplitConnector_Box`es would have had to be added between the `Output_Box` and their out-neighbor to provide an additional point of access. One `ComplexFilter_Box` is attached to each of those `Output_Box`es. At the outputs with index one of each of these `ComplexFilter_Box`es, new `Output_Box`es o_R and o_S are added. These will generate the result.

The basic idea is that the algorithm that executes the query will at some point reach one of the `ComplexFilter_Box`es. Let this be the box cf_1. Each element of its input collection appears once at the second output of cf_1. The outputs (with index two) of both `ComplexFilter_Box`es are connected by a `MakeTrue_Box(2)`mt (presented on page 51. Any other combination of boxes is possible that makes it necessary for the executing algorithm to execute the other `ComplexFilter_Box` cf_2. Assume for the sake of simplicity that mt has been chosen and a `BoolPredicateEnd_Box` terminates the predicate after that.

When the first element of the first input collection, r_1, reaches mt, mt cannot be executed until the second input has been provided. This implies that the second `ComplexFilter_Box` cf_2 must be executed and the first element of the input collection of cf_2 reaches mt. This is element s_1. Now, mt is executed, yielding the boolean value true. The elements pushed into the predicates by the `ComplexFilter_Box`es have therefore successfully passed the predicate and are put into the respective output collections. At this stage, the output would be

$$r_1 \ / \ s_1$$

The execution of `ComplexFilter_Box` cf_2 has not finished yet. After the predicate has been evaluated, the next element, s_2 is tested. This leads to the result

$$r_1 \ / \ s_2$$

o_r
r_1
r_1
$\vdots$
r_1
r_2
r_2
$\vdots$
r_2
$\vdots$
r_r
r_r
$\vdots$
r_r

o_s
$[s_1, s_2, \ldots, s_s]$
$[s_1, s_2, \ldots, s_s]$
$\vdots$
$[s_1, s_2, \ldots, s_s]$

Table 4.8: Table views of Output from `Output_Box`es o_r and o_s

and so on. After s steps, cf_2 has finished and the result up to this point is

$$r_1 \ / \ s_1 \text{ and } r_1 \ / \ s_2 \text{ and } \ldots \text{ and } r_1 \ / \ s_s$$

Next, cf_1 posts r_2 into the predicate subquery. The execution of cf_2 is repeated and the results

$$r_2 \ / \ s_1 \text{ and } r_2 \ / \ s_2 \text{ and } \ldots \text{ and } r_2 \ / \ s_s$$

are produced.

After $r \cdot s$ steps, cf_1 has finished its execution and all tuples

$$r_1 \ / \ s_1 \ldots r_r \ / \ s_r$$

have been put into the output lists of the `ComplexFilter_Box`es.

When those lists are put into the result table by the two `Output_Box`es o_R and o_S, the result will not be as expected, however. The reason is that the `ComplexFilter_Box` cf_2 reached by the algorithm after cf_1 is executed r times. Thus, r results reach the `Output_Box` o_S. As a consequence, o_S will have to accumulate data: After cf_2 has been executed for the first time, a list $[s_1, s_2, \ldots, s_s]$ is saved. After the second execution, the same list reaches o_s again and the result will be

$$[\ [s_1, s_2, \ldots, s_s], [s_1, s_2, \ldots, s_s]\ ]$$

The ***table view*** of the output of o_R will consist of $r \cdot s$ cells. The first s cells contain element r_1, the next s cells contain r_2 and so on. This is the desired behavior.

On the other hand, the ***table view*** of the output of o_S will consist of only r cells each containing the list $[s_1, s_2, \ldots, s_s]$. Both ***table view***s are shown in table 4.8.

To remedy this, all added boxes are saved as a subquery and a `SubQuery_Box` is added taking this subquery as parameter. This is illustrated in Figure 4.12. The `SubQuery_Box` has two inputs and two outputs. At the second output of this box, the list of lists from o_S is present. A `ListOperations1_Box(flatten)` can be added there leading to the desired output. An `Output_Box` o'_S attached there yields the table view as depicted in table 4.9. In this tables, one can clearly see the tuples of the cartesian product that have been created.

o_r	$o2_s$
r_1	s_1
r_1	s_2
$\vdots$	$\vdots$
r_1	s_s
r_2	s_1
r_2	s_2
$\vdots$	$\vdots$
r_2	s_s
$\vdots$	$\vdots$
r_r	s_1
r_r	s_2
$\vdots$	$\vdots$
r_r	s_s

Table 4.9: Desired Table views of Output from `Output_Box`es o_r and o_s'

General Case:

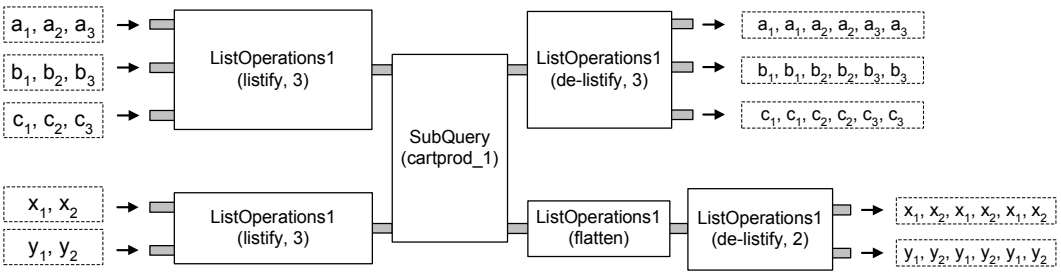


Figure 4.13: Simulation of the Cartesian Product Operator, General Case

Let R and S be two relations of arbitrary arity. Both relations will be compressed into one list each. These two lists will then be processed as has been described in the previous basic case. The compression is shown for relation R with arity $r > 1$. The same procedure can applied to S.

Let $[o_1, o_2, \ldots, o_r]$ be the list of `Output_Box`es that are responsible for creating R. As has been done in the basic step, assume that these boxes do not have any out-edges. A `ListOperations1_Box(listify,` r`)` called lo_1 is added to the query graph (this box is described on pages 28 and 57). Each box $o_i \in [o_1, o_2, \ldots, o_r]$ is connected to the i^{th} input of lo_1. This results in one list l_1 holding all tuples of relation R.

The same is done with relation S and a `ListOperations1_Box(listify,` s`)` called lo_2 producing l_2.

The two lists l_1 and l_2 are used as inputs to the procedure described in the basic case. This combines every tuple in l_1 with every tuple in l_2. It creates two lists l_1' and l_2'.

To obtain the desired result, a `ListOperations1_Box(unpack,` r`)` is added to the query and

gets l'_1 as its sole input. Its r outputs provide the entries of the requested cartesian product from relation R. Repeating this step for l'_2 completes the simulation.

Figure 4.13 shows an example. The tables in Figure 4.14 show the table views of example relations R and S and the result of applying the described simulation of the cartesian product $R \times S$:

r_1	r_2	r_3	s_1	s_2
a_1	b_1	c_1	x_1	y_1
a_2	b_2	c_2	x_2	y_1
a_3	b_3	c_3		

r'_1	r'_2	r'_3	s'_1	s'_2
a_1	b_1	c_1	x_1	y_1
a_1	b_1	c_1	x_2	y_2
a_2	b_2	c_2	x_1	y_1
a_2	b_2	c_2	x_2	y_2
a_3	b_3	c_3	x_1	y_1
a_3	b_3	c_3	x_2	y_2

Figure 4.14: Table Views of Two Input Relations and their Cartesian Product

4.2 Comparison to SQL

[KE01] gives a good introduction to SQL and serves as a list of further references.

In this thesis, SQL 92 is examined ([Cha76], [DD93]). The new SQL 3 standard is still not Turing-complete, but has added ways of introducing recursion into the queries. Since recursion is not an issue in the *QUOGGLES* system, the older standard is taken as model. All language constructs making SQL a data manipulation language are ignored. Only features for which it remains a pure query language are treated. Thus, keywords like *create, alter, drop* and *insert, delete, update* are not considered. [CG85] shows connections between relational algebra and SQL (to be more precise, translation techniques from SQL into relational algebra are examined).

This section shows that all SQL queries can be formulated in the *QUOGGLES* system using basic boxes (compound boxes are often used as an abbreviation). Section 4.3 will show that the system using queries that can be formulated with basic boxes is not Turing-complete.

It has been shown that SQL is relational complete. Apart from that, it has language constructs that make it even more powerful:

1. Aggregate Functions, keywords *count, sum, min, max, avg*
2. Arithmetic operations, keywords *+, -, *, /*
3. Sorting, keyword *order by*
4. Grouping, keyword *group by*

Languages supporting such features have been titled *more than complete* ([Cod72]). It will now be shown that the *QUOGGLES* system is more than complete in the same sense as SQL. This is done by simulating all four capabilities in the *QUOGGLES* system.

4.2.1 Aggregate Functions

An aggregate Function can be applied to a component of a relation, resulting in a single value. Functions supported by SQL are:

***count*:** Calculates the number of elements in the column. This method can also be applied to a whole relation (`COUNT(*)`). The result is of course the same when applying it to one of the components.

***sum*:** Calculates the sum of all elements in the column.

***min*:** Calculates the minimum element in the column.

***max*:** Calculates the maximum element in the column.

***avg*:** Calculates the average value of all elements in the column.

Since a component in the *QUOGGLES* system is a collection, the simulation of these functions is simple: An `Arithmetic_Box` with the appropriate parameter is used. The value of the parameter is exactly the name of the aggregate function (see the description of the box on pages 32 and 47).

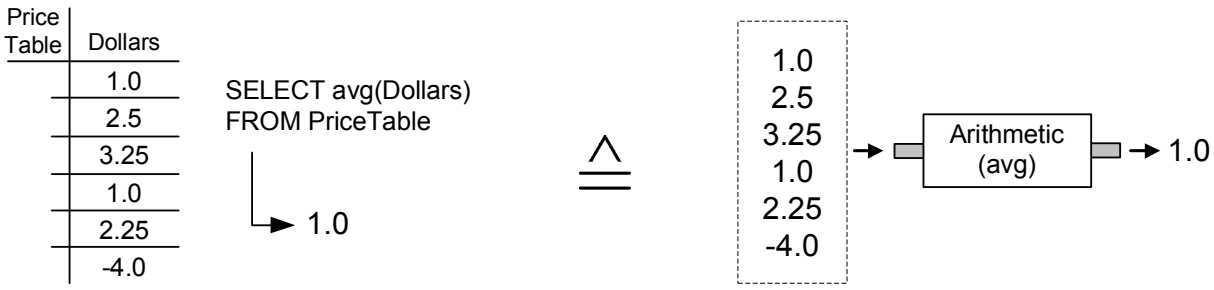


Figure 4.15: Simulation of an Aggregate Function.

Figure 4.15 shows the simulation with an example.

Aggregate Function play an important role in conjunction with the `GROUP BY` keyword presented in Section 4.2.4.

4.2.2 Arithmetic Operations

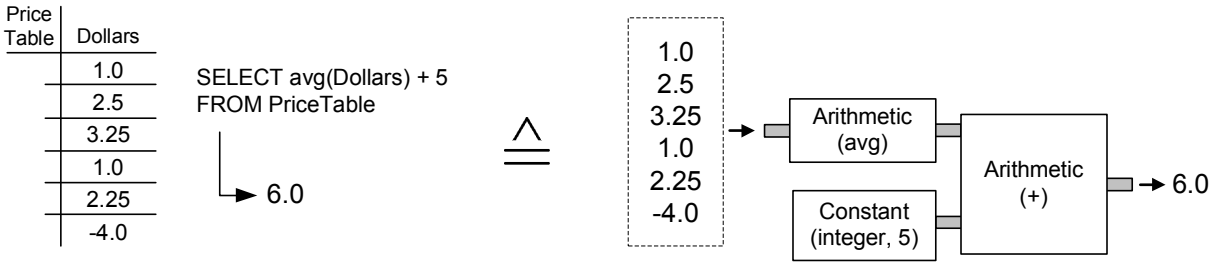


Figure 4.16: Simulation of an Arithmetic Operation.

The existence of arithmetic operations in any system where numbers occur as objects is often taken for granted. In relational algebra, however, even simple operations like addition of two numbers is not included. In `SQL`, arithmetic operations can be applied anywhere in the query where two numbers occur. These may result from constants or complex expressions like aggregate functions.

In the *QUOGGLES* system, these arithmetic operations can be applied to two numbers using an `Arithmetic_Box`. It takes the numbers as input and applies the operation specified by the parameter (see the description of the box on page 32).

Figure 4.16 shows the equivalence by means of an example.

4.2.3 Sorting

The keyword *ORDER BY* is used in `SQL` to specify that a relation should be sorted according to the components listed after the keyword. In addition to that, it can be specified whether the relation should be sorted according to the ascending or descending order of each component. The complete syntax is:

```
SELECT c_1, c_2, ..., c_m FROM ... WHERE ...
ORDER BY col_1 [asc|desc], col_2 [asc|desc], ..., col_n [asc|desc]
```

`c_1, c_2, ..., c_m` and `col_1, col_2, ..., col_n` are lists of component names of the relation generated by the `FROM` clause.

This statement directly translates into a query in the *QUOGGLES* system: A `SortBy_Box(`m`, `n`)`, described in detail on page 60, gets the collections representing column names `c_1, c_2, ..., c_m` as first m inputs and those representing names `col_1, col_2, ..., col_n` as second n inputs. The result will be the first m columns sorted according to the other columns. Figure 4.17 illustrates this. For an example, refer to Figure 4.4 on page 61.

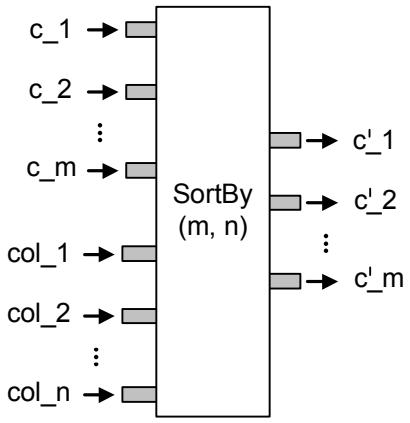


Figure 4.17: Simulation of the *ORDER BY* Keyword.

4.2.4 Grouping

`SQL` allows combining tuples of a relation according to the values of specified components. Let

$$R = \{r_1, r_2, \ldots, r_r\}$$

be a relation of arity r.

The syntax of the `GROUP BY` keyword in `SQL` is as follows:

```
SELECT ca_1, ca_2, ..., ca_m FROM R WHERE ...
GROUP BY col_1, col_2, ..., col_n
```

Without loss of generality, R is used as the relation on which the statement works. Of course, any expression yielding a relation can be used instead of R.

`col_1, col_2, ..., col_n` is a list of component names of R. `ca_1, ca_2, ..., ca_m` consists of component names contained in `col_1, col_2, ..., col_n` and names of aggregate functions. The list following the `SELECT` keyword is restricted to these two possibilities since this is the only way to ensure that no selected value changes within one group.

Consider the tables in Figure 4.18. The top left table represents the original relation. The top right table is grouped by the second (degree) component, the one at the bottom left is grouped

original relation:

node	degree	hidden
n1	3	false
n2	1	true
n3	3	true
n4	0	false
n5	2	true
n6	2	true

`GROUP BY` *degree*:

node	degree	hidden
n4	0	false
n2	1	true
n5	2	true
n6	2	true
n1	3	false
n3	3	true

`GROUP BY` *hidden*:

node	degree	hidden
n1	3	false
n4	0	false
n2	1	true
n3	3	true
n5	2	true
n6	2	true

`GROUP BY` *degree, hidden*:

node	degree	hidden
n4	0	false
n2	1	true
n5	2	true
n6	2	true
n1	3	true
n3	3	false

Figure 4.18: Three Examples of Different `GROUP BY` Statements

by the third (hidden) and the bottom right table is grouped by the second and third components (degree and hidden).

For every group, `SQL` generates one single tuple. Therefore, the selection clause can only list components that have been mentioned in the `GROUP BY` statement. Aggregate functions can of course be used as well, since they produce exactly one value per group.

How can this operation be simulated? The procedure is illustrated by an example. The ***current graph*** on which the query operates is shown in Figure 4.19. It shows eight cities.

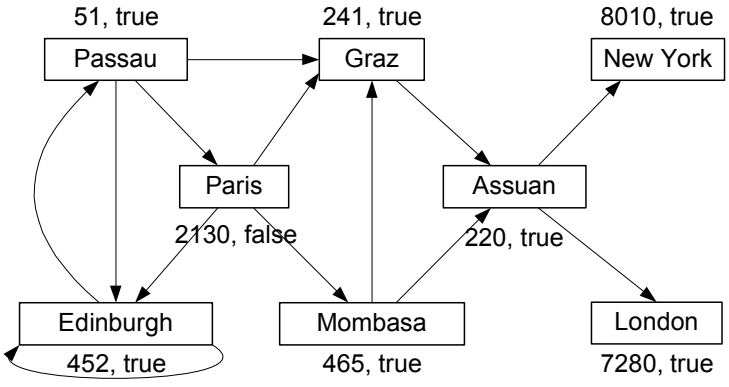
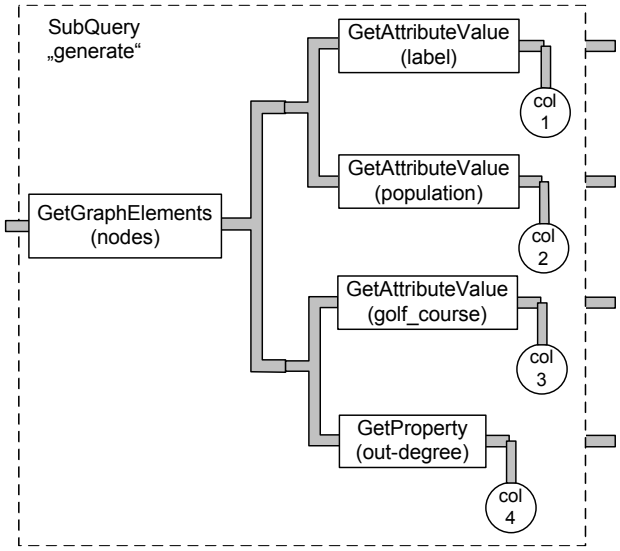


Figure 4.19: Example Graph

They all have the following four attributes:

- **label**: The name of the city
- **population**: An integer specifying the population of the city (in thousands; figures from 1999, [Mic03])

- **golf_course**: A boolean attribute stating whether or not the city has a certain type of golf course (artificial values)
- **out_degree**: The out-degree of the node representing the city in the graph



col 1	col 2	col 3	col 4
Passau	51	true	3
Graz	241	true	1
Assuan	220	true	2
Edinburgh	452	true	2
Paris	2130	false	3
Mombasa	465	true	2
New York	8010	true	0
London	7280	true	0

Figure 4.20: Generate an Example Relation

Figure 4.20 displays a small query graph that will be put into a `SubQuery_Box(generate)`. It generates a relation R of arity four. This relation is shown to the right of the graph.

The following `SQL` statement will be simulated:

```
SELECT sum(population), golf_course, out_degree
FROM R
GROUP BY golf_course, out_degree
```

col 2	col 3	col 4
15290	true	0
241	true	1
1137	true	2
51	true	3
2130	false	3

Table 4.10: Result of the `SQL` query

The result of this query is shown in Figure 4.10.

Call the projection of R onto the components by which it is grouped (attributes `golf_course` and `out_degree`)

$$R_G = \pi_{golf\_course,out\_degree}(R)$$

Grouping applied to R_G means eliminating all duplicates. Therefore, the first step in the simulation is to put components `golf_course` and `out_degree` into one list of tuples and apply a

duplicate elimination procedure on it. This is illustrated by Figure 4.21. In the remainder of this section, the tuples in R_G are called group representatives.

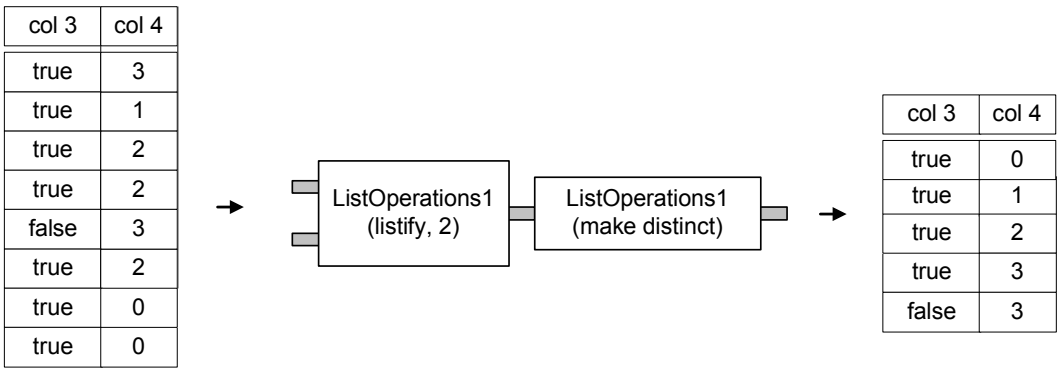


Figure 4.21: Generating the groups

The next steps will check all tuples of R. Let r be such a tuple. The idea is to calculate

$$r_G = \pi_{golf\_course,out\_degree}(r)$$

(where r is interpreted as a relation containing exactly one tuple) and compare that to the tuples in R_G. Let g be the number of tuples in R_G. g is the number of distinct groups. Imagine g buckets. Whenever r_G is equal to tuple number i in R_G, the tuple r is put into bucket i. This is exactly how an intuitive algorithm would work: Check all tuples and put all those belonging to one group into one bucket.

The next figure displays the query graph that is responsible for putting the tuples into buckets. It takes the three components of R as input and produces a list of tuples of R. Each tuple t will now be compared against each distinct representative of a group. These consist only of components `golf_course` and `out_degree`. Therefore, t must be projected onto those components as well. The resulting tuple is now compared. Those tuples that match with the group representative, are put into a list available at the first output of the `ComplexFilter_Box`. The figure illustrates that for the tuple $(true, 0)$ as representative. Matching tuples are:

$$(8010, true, 0) \text{ and } (7280, true, 0)$$

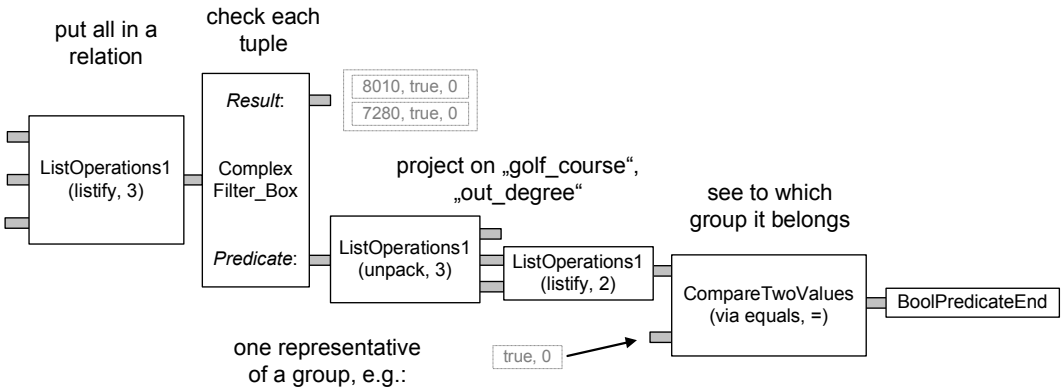


Figure 4.22: Divide Tuples into Buckets

This subquery is executed for each group representative. A `ComplexFilter_Box` will produce the group representatives. As has been described in the description of the simulation of the cartesian product operation in Section 4.1.3, this way all tuples will be compared to all group representatives.

The query resulting from the last two illustrations is put into a `SubQuery_Box(group)` with three inputs (the components of R) and one output as illustrated in Figure 4.23.

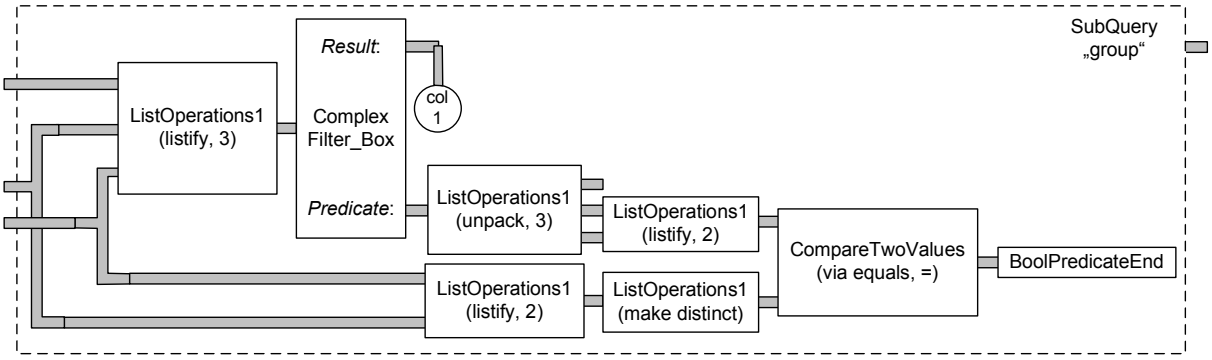


Figure 4.23: The `SubQuery_Box(group)`

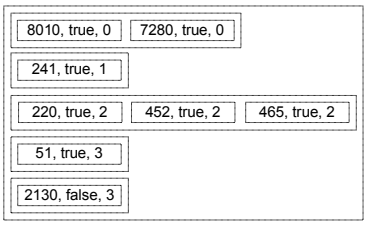


Figure 4.24: The Result of the `SubQuery_Box(group)`

When the query created so far is executed, the result shown in Figure 4.24 is produced. It is a list of five elements. Each element is a list containing the tuples that belong to one of the five groups. The group of $(true, 0)$ has two members, the $(true, 3)$ only one, for instance.

This intermediate result contains all information needed to generate the output of the `SQL` statement. The select part states that for each group, the first values of the entry (population) should be added together, and the second and third elements should be printed.

This can be done by iterating over this list with a `ComplexFilter_Box`. Each element appearing at the second output of this box is a list containing 3-tuples. A `ListOperations1(de-listify, 3)` splits these tuples and generates three lists containing all first, second and third elements respectively. Thus, the first list will hold all population entries for the current group. An example of the predicate getting the list

$$[\,[8010, true, 0], [7280, true, 0]\,]$$

as input is visualized in Figure 4.25. The first list generated by the `ListOperations1(de-listify, 3)` contains values 8010 and 7280. These are added to produce the final value. Elements in the second and third lists are values from components by which the relation has been grouped. Therefore, they contain several copies of one

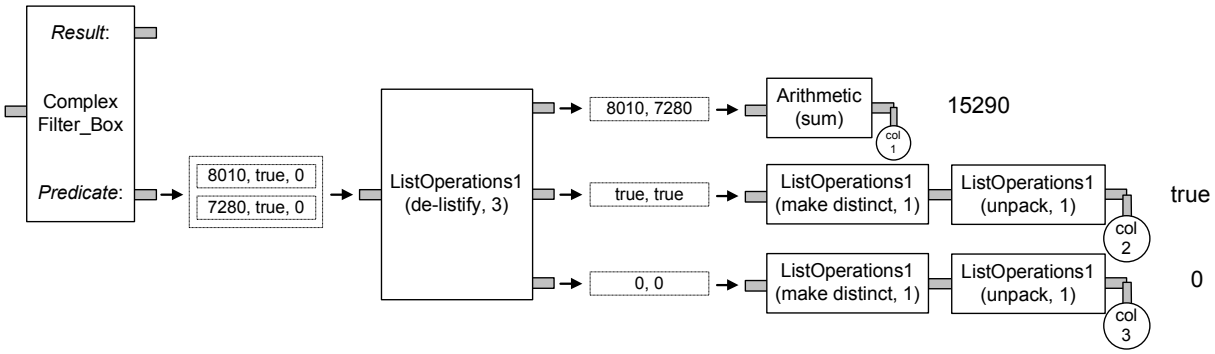


Figure 4.25: Generating the Output, Example

and the same value. A combination of a `ListOperations1_Box(make distinct, 1)` and `ListOperations1_Box(unpack, 1)` gets hold of this element. In the example, these values are "true" and "0".

The subquery generating the output is shown again in Figure 4.26 without any example outputs.

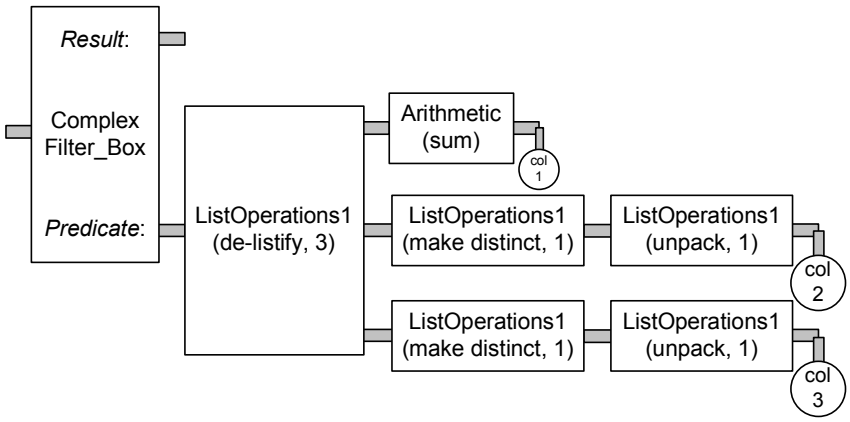


Figure 4.26: Generating the Output

Figure 4.27 is a screen shot of the *QUOGGLES* system with the query that generates relation R and simulates the `SQL` statement.

This concludes the section about simulating all operations of `SQL` that have not been already demonstrated in Section 4.1.3 about simulating relation algebra.

The next section proves that although the *QUOGGLES* system can answer many questions, it still cannot solve all computable problems.

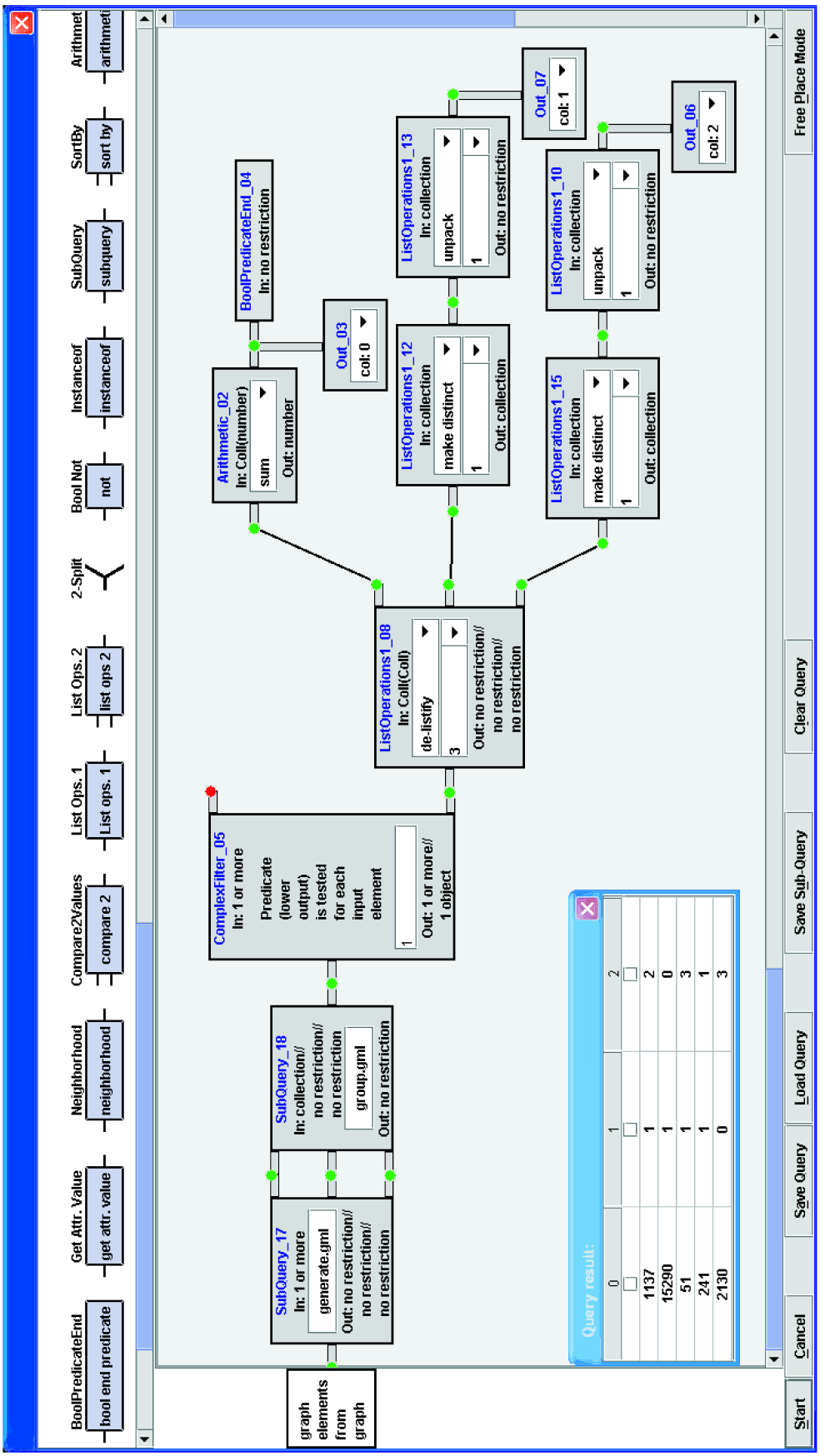


Figure 4.27: Screen Shot of the Query Simulating the GROUP BY Keyword

4.3 Turing-completeness

To be able to give a more general idea of the power of the *QUOGGLES* system, two terms are introduced, primitive recursive and μ-recursive functions. The relation of the system and those concepts is examined afterwards. This section follows the reasoning used in [MC99]. The authors present `OCL`, an extension for `UML` and show several considerations concerning the power of their language. For a more detailed treatment of formal languages, see [HA94] and [Sal78] besides the specific citation in the following text.

Turing machines are one possibility to characterize the general notion of computability. According to Church's Thesis, everything that is intuitively computable, is computable on a Turing machine, or **Turing-complete**. Instead of theoretical machine models, recursive functions can be used to formalize the concept of algorithms.

Definition 4.15 (Class of Primitive Recursive Functions, [Sch97])

The class of **primitive recursive functions** is inductively defined as follows: $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is primitive recursive, if

1. f is constant, a projection or the successor function $f(n) = n + 1$ or
2. f is composed of primitive recursive functions by
   - composition or
   - primitive recursion, i.e. $f(0, \ldots) = g(...)$ and $f(n + 1, \ldots) = h(f(n, \ldots), \ldots)$ where g and h are primitive recursive

Primitive recursion represents restricted iteration. The depth of recursion (i.e. number of iterations) is a number explicitly fixed in advance (cf. [Bro98]). This means that it describes functions that can be realized using for-loops. Schöning ([Sch97]) introduces so-called LOOP–programs that contain assignments and for-loops only. He uses this concept to proof the equivalence of programs using for-loops and primitive recursive functions. LOOP–programs will be defined later in this section.

By definition, no partial function can be primitive recursive. However, there exist even total functions that are Turing-complete but not primitive recursive. One example is the Ackermann function (cf. [Bro98]).

The extension of the class of primitive recursive functions by the μ-Operator leads to the definition of the class of μ-recursive functions:

Definition 4.16 (μ-Operator, [Sch97])

Let f be a $k + 1$-ary function. When applying the μ-operator on f, the resulting function is $g : \mathbb{N}^k \rightarrow \mathbb{N}$ where

$$g(x_1, \ldots, x_k) = min\{n \mid f(n, x_1, \ldots, x_k) = 0$$

and $f(m, x_1, \ldots, x_k)$ is defined for all $m < n$

Definition 4.17 (Class of μ-Recursive Functions)

The class of **μ-recursive functions** is the smallest class of (partial) functions that contains the basic functions (constant, projection and successor function) and is closed under application of primitive recursion and the μ-operator.

In analogy to primitive recursive function that are equal to LOOP–programs, μ-recursive functions descibe what can be realized by so-called WHILE–programs that are extensions of LOOP–programs ([Sch97]).

The rest of this section shows that the query language presented up to this point (called the basic system) is not Turing-complete or, equivalently, cannot simulate μ-recursive functions. With a small and sensible extension, it can create primitive recursive queries.

The crucial point in the question whether or not the *QUOGGLES* system using only basic or compound boxes is Turing-complete, is the existence or non-existence of some type of recursion. If there were no constructs allowing loops or recursion, there would be no chance for Turing-completeness.

However, there is the `ComplexFilter_Box`. It constitutes a loop with the number of iterations defined by the (size of the) input. Using the `Constant_Box(collection)`, a list can be created. The size of it dynamically depends on the input number. Using a `Constant_Box(integer,` n`)` as input, a list of length n can be generated. Therefore, a subquery can be executed n times, where n is an arbitrary natural number (that can even be set dynamically, i.e. depending on the number of graph elements in the ***current graph*** or a value of some attribute).

This seems to be convincing enough that looping techniques can indeed be used. There is, however, a catch to that: There is no way to pass information from one iteration step to the next. This renders general path queries impossible to formulate in the basic *QUOGGLES* system as is summarized in the following proposition:

Proposition 1: The problem "*Let n_1 and n_2 be two nodes. Is there a (directed) path from n_1 to n_2 of arbitrary length?*" is not solvable in the *QUOGGLES* system *for all graphs.*

Before this proposition is proven, it is shown that the problem can indeed be solved if the graph is known in advance:

Proposition 2: The problem "*Let n_1 and n_2 be two nodes. Is there a (directed) path from n_1 to n_2 of arbitrary length?*" can be solved in the *QUOGGLES* system *for a given graph G.*

Proof of Proposition 2: The crucial information when working on a specific graph is its number of nodes. The number of edges of a shortest path (in fact any path that does not visit any node several times) between two arbitrary nodes in a graph is always smaller than the number of nodes $n = |V|$ in the graph $G = (V, E)$. This means that it suffices to find all nodes that are accessible by traversing n edges starting from node n_1.
The case where $n_1 = n_2$ is easy found out. Otherwise, observe that, in the *QUOGGLES* system, the following sequence of boxes calculates the out-neighbors of a list of nodes:

`Neighborhood_Box(out-edges)` → `ListOperations1_Box(flatten)` →

`Neighborhood_Box(target node)` → `ListOperations1_Box(flatten)`

The flattening is necessary since the neighborhood boxes yield a collection of sets. A `Contains_Box` is then used to check whether this set of out-neighbors contains node n_2. The set is then passed to another copy of those four boxes, calculating the out-neighbors of them and so on. The results of the $n - 1$ `Contains_Box`es are concatenated by $n - 1$ `BooleanOp_Box`es with parameter *OR*. □

If the length of the wanted path is delimited, the problem can be solved for any input graph as well. Consider the following query:

Proposition 3: The problem "*Let n_1 and n_2 be two nodes. Is there a path (in the underlying undirected graph) from n_1 to n_2 of length at most 3 (i.e. one has to traverse at most three edges to get from n_1 to n_2)?*" can be solved in the *QUOGGLES* system *for all graphs.*

Proof of Proposition 3: To get the correct answer, three `Neighborhood_Box`es are needed, stepping three times from one node to an adjacent node, starting from n_1. If n_2 is encountered somewhere on that way, the answer is true, false otherwise. Figure 4.29 shows the query graph. Since the task of getting the set of distinct neighbors appears three times, it has been put into a separate subquery, shown in Figure 4.28. □

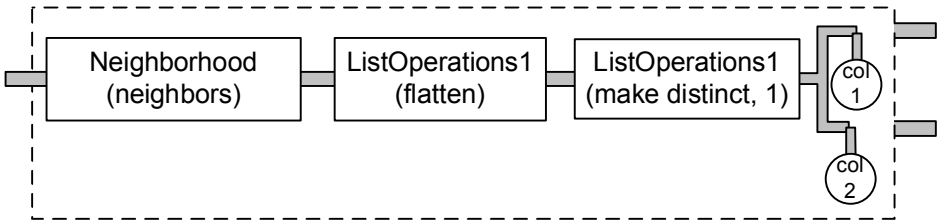


Figure 4.28: Subquery "distinctNeighbors"

Thus, if either the size of the graph is known or the maximal length of paths is fixed, such queries can be constructed.

After these considerations, the notion of LOOP–programs is introduced to be able to more formally characterize the power of the basic system:

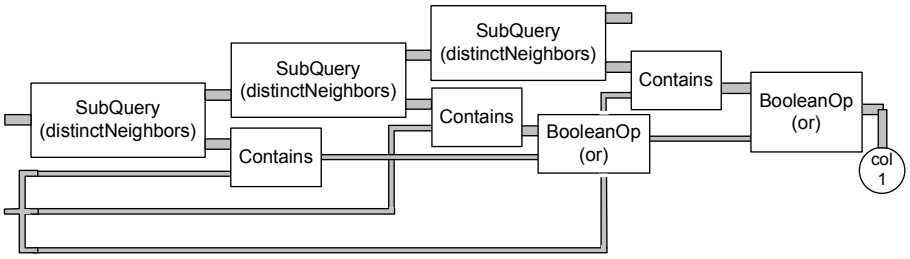


Figure 4.29: Existence of a Path of Length ≤ 3

Definition 4.18 (LOOP–Programs)

LOOP–programs are imperative programs with the following *syntax*:

`P ::=` $X \leftarrow X$ `+` C

| $X \leftarrow X$ `-` C

| `LOOP` X `DO` P `END`

| P `;` P

X `::=` x_0 | x_1 | x_2 | `...` `(variables)`

C `::= 0 | 1 | 2 | ...` `(constants)`

The *semantic* of LOOP-programs is as follows: The value assignment $x_i \leftarrow x_j$ `+` c is states that the new value of the variable x_i is the value of x_j plus the constant c. The value assignment $x_i \leftarrow x_j$ `-` c is the nonnegative subtraction, that is, if c `>` x_j then the new value of x_i is 0 otherwise the value of x_j minus c. A LOOP–program of the form P_1 `;` P_2 is means executing P_1 and P_2 consecutively. Finally a LOOP–program of the form `LOOP` x_i `DO` P `END` is interpreted as follows: The program P is executed n times, where n is the value of the variable x_i at the beginning (i.e. the change of the value of x_i within P does not affect the number of repetitions). □

Proposition 4: The basic system is not able to simulate ***LOOP–programs***.

> **Proof of Proposition 4:** The critical point in the definition of ***LOOP–programs*** is the existence of variables. The value of those variables can be altered in each iteration. Thus, information can be collected and passed between iterations. This is not possible in the *QUOGGLES* system. Since all boxes used in the predicate query of a `ComplexFilter_Box` are reset before each iteration, all data is lost after each execution. □

Using these characterization, the next proof is trivial:

> **Proof of Proposition 1:** The proof follows directly from Proposition 4 since, in one iteration step, the size of the graph theoretical environment of nodes that can be explored is fixed by some number not depending on the number n of nodes in the ***current graph***. This size depends on the number of `Neighborhood_Box`es in the query. Only the number of possible iterations can be set dynamically. Although a subquery can be executed n times, all information about the neighborhood of nodes is lost after each iteration. □

This statement implies that the basic system cannot be Turing-complete: ***LOOP–programs*** obviously are computable. Since the system cannot simulate those programs, it cannot generate

all Turing-computable functions. As has been stated in the beginning of this section, ***LOOP–programs*** represent the same class of functions as those that are primitive recursive. Therefore, the basic system is not even as powerful as the class of primitive recursive functions.

Those observations motivate the introduction of boxes that can change the values of attributes. The attributes associated with a graph or its graph elements can then act as variables:

- **ChangeAttribute_Box**
  Changes the value of a specified attribute of the input `Attributable` to the input value.

  Input 1: An `Attributable`.

  Input 2: An object the type of which must be compatible with the attribute specified by the parameter.

  Output: The `Attributable` from the first input. Its attribute (specified by the parameter) has the object from the second input as value. If the attribute has not previously existed, it has been created.

  Parameter 1: The type of the attribute. Besides being more type-safe, this parameter is necessary since the specified attribute will be created if it does not already exist.

  Parameter 2: The path to the attribute that should be created or changed.

  Figure 4.30 shows an example use. A node with label "n1" is passed as first input and gets its label changed to "Prague".

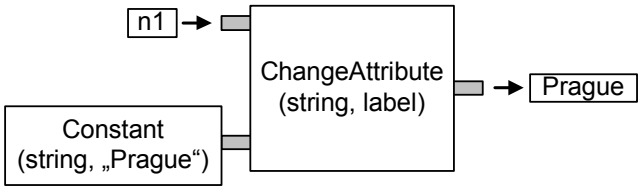


Figure 4.30: `ChangeAttribute_Box`

- **ChangeAttributes_Box**
  The same semantics as the `ChangeAttribute_Box` operating on a list of `Attributable`s and a list of attribute values.

  Input 1: A list of `Attributable`s.

  Input 2: A list of objects the type of which must be compatible with the attribute specified by the parameter.

  Output: The list of `Attributable`s from the first input. The attribute (specified by the parameter) of each `Attributable` has the corresponding object from the second input as value. If an `Attributable` did not have the attribute, it has been created.

  Parameter 1: The type of the attribute. Besides being more type-safe, this parameter is necessary since the specified attribute will be created if it does not already exist.

  Parameter 2: The path to the attribute that should be created or changed.

  As can be seen in Figure 4.31 it is very easy using this box to assign all nodes in a graph their out-degree as label. Some types of trivial conversions (in this example from integer to string) are done automatically.

Obviously, the function of this compound box can easily be simulated by the `ChangeAttribute_Box`, iterating through both input lists with two `ComplexFilter_Box`es.

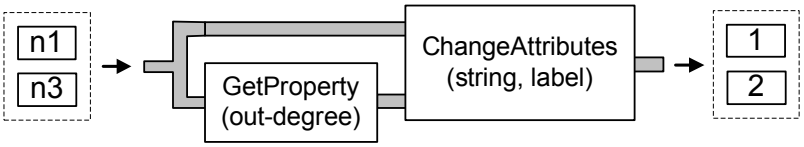


Figure 4.31: `ChangeAttributes_Box`

The language including those boxes is called the **extended system**.

Using those boxes, all features of ***LOOP–programs*** can be simulated in the *QUOGGLES* system. Arbitrary assignments are executed by `ChangeAttribute_Box`es, subqueries can be concatenated and the number of iterations of loops can depend on dynamic input yielding the following result:

Proposition 5: The ***extended system*** is equally powerful as primitive recursive functions.

Proof of Proposition 5: The proof follows immediately from the comparison to ***LOOP–programs***. □

This means that it must be possible to form a query that, for an arbitrary graph and one of its nodes as input, calculates the ***reach-set*** of the node in this graph:

Definition 4.19 (Reach-Set)

The **reach-set** $RS(n)$ of a node n is the set of nodes to which a directed path exists starting from node n

The next part shows the solution for this problem (called the RS-Query) and briefly explains the ideas behind it. Let $G = (N, E)$ be the ***current graph***. The ***reach-set*** of the node n labelled "Mombasa" will be found.

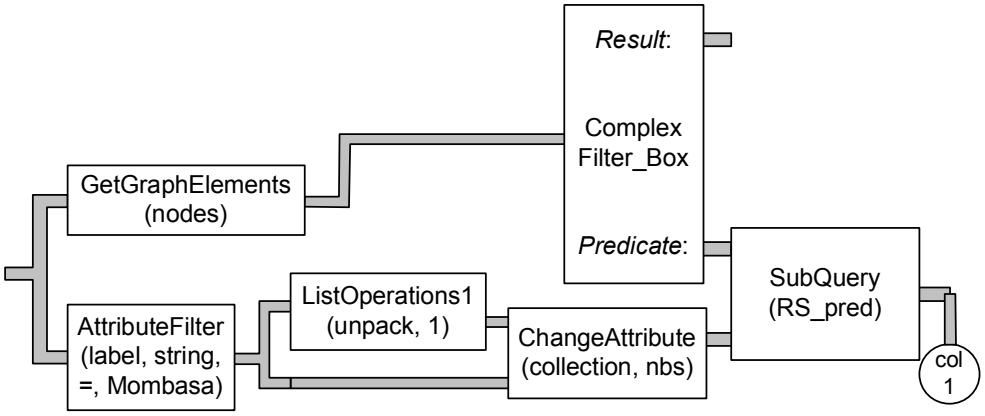


Figure 4.32: Input, Initialization and Iteration of the RS-Query

In Figure 4.32, two paths join each other in a `SubQuery_Box(RS_pred)`. The two boxes on top (`GetGraphElements_Box(nodes)` and `ComplexFilter_Box`) represent a loop with $|N|$ iterations. The lower boxes extract the node n for which $RS(n)$ will be computed (`AttributeFilter_Box(label, string, =, Mombasa)`) and initialize its attribute *nbs* to a list containing n. The `AttributeFilter_Box` produces a collection containing one element, therefore the `ListOperations1_Box(unpack, 1)` is needed.

The subquery called "RS_pred" needs two inputs. First, it must be connected to the predicate output of the `ComplexFilter_Box`. Otherwise, no iteration would be processed. Second, it must get node n as starting point.

The main task of this subquery is to calculate the set of out-neighbors of a set of nodes. Applying this procedure repeatedly will simulate a walk through the graph along its directed edges. After at most n steps (in fact after at most $n-1$ steps), the whole RS must have been visited. It is a breadth first search traversal of the graph. The nodes are visited similar to the waves that evolve from throwing a stone in the water. After every iteration, nodes that are farther away (in the graph theoretical way of the shortest paths being longer) are found.

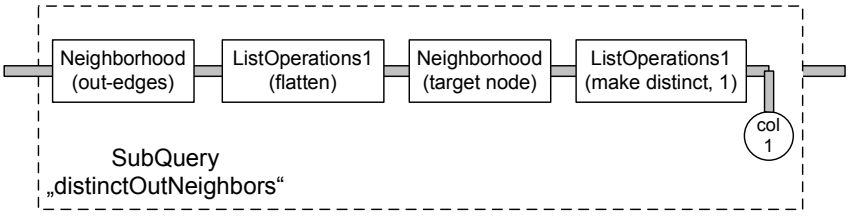


Figure 4.33: Calculation of the Set of Out-Neighbors

A subquery calculating out-neighbors for a collection of input nodes is shown in Figure 4.33.

The most important part is the subquery "RS_pred" depicted in Figure 4.34. The `GetAttributeValue_Box(nbs)` retrieves the attribute *nbs*. This attribute is a collection holding the nodes that represent the wavefront mentioned before. It is initialized to the node (or nodes) for which the RS is wanted. If the ***current graph*** is acylic and contains no loops, no node is present in this collection in two different iteration steps.

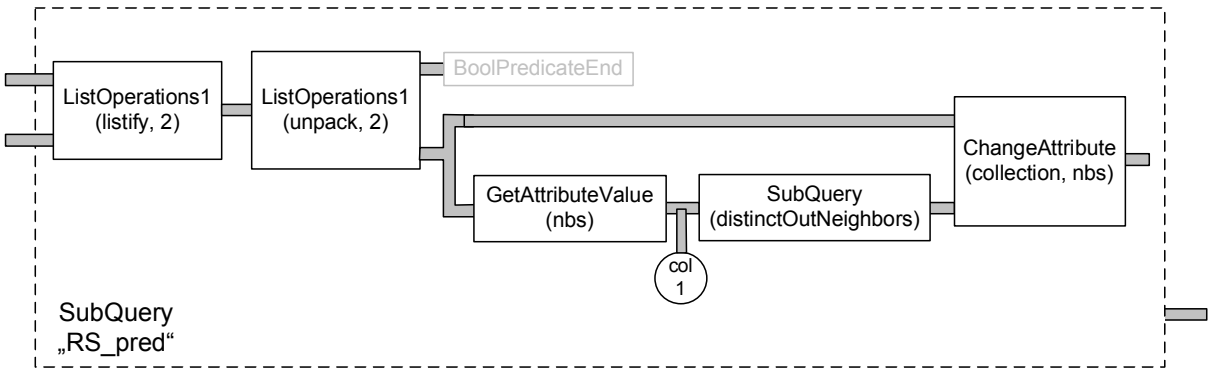


Figure 4.34: Accumulating the Out-Neighbors

For this collection, the out-neighbors are computed and the result is written back to the attribute. This is done by the combination of the `SubQuery_Box(distinctOutNeighbors)` and

the `ChangeAttribute_Box(collection, nbs)`. The latter needs the collection and node n as input.

The two boxes on the left hand side of the subquery represent a little trick that has to be applied in such circumstances: The subquery needs only node n to work on. As stated before, however, it must be connected to the `ComplexFilter_Box` to be executed $|N|$ times as is necessary. Therefore, a `ListOperations1_Box(listify, 2)` combines the output from the `ComplexFilter_Box` and node n whereas the `ListOperations1_Box(unpack, 2)` splits them again. The output of the `ComplexFilter_Box` (all nodes of the ***current graph***) is not of any interest. The `BoolPredicateEnd_Box` is attached only because the subquery is used as a predicate and therefore must have exactly one of those (cf. page 27).

Tables 4.11 and 4.12 show the ***result table***s of the *RS*-Queries starting with nodes "Mombasa" and "Paris", respectively.

col. 1	col. 2
Node (Mombasa)	[Node (Mombasa)] [Node (Graz), Node (Assuan)] [Node (Assuan)] - - -

Table 4.11: Result Table of *RS*-Query for Node "Mombasa"

col. 1	col. 2
Node (Paris)	[Node (Paris)] [Node (Mombasa), Node (Edinburgh), Node (Graz)] [Node (Graz), Node (Assuan), Node (Edinburgh), Node (Passau)] [Node (Assuan), Node (Edinburgh), Node (Passau), Node (Graz), Node (Paris)] [Node (Edinburgh), Node (Passau), Node (Graz), Node (Paris), Node (Assuan), Node (Mombasa)] [Node (Edinburgh), Node (Passau), Node (Graz), Node (Paris), Node (Assuan), Node (Mombasa)]

Table 4.12: Result Table of *RS*-Query for Node "Paris"

It can be seen that after three and five steps, respectively, no changes occur any more. The reason why the second query ("Paris") leads to the repeating of a list of all nodes is that the node labelled "Edinburgh" has a self-loop. Therefore, the moment it is found by the search (in the first step), it will always be included in the next step. This means that its neighbor ("Passau") will be included as well resulting in nodes "Graz" and "Passau" to be included and so on.

The next proposition will show why it is not possible to terminate the loop earlier, i.e. as soon as no or all nodes are contained in the list saved as attribute *nbs*.

Even equipped with the new `ChangeAttribute_Box`, the extended system is not Turing-complete. Another characterization of this completeness is the ability to express WHILE-programs. As [Sch97] shows, these programs are equivalent to μ-recursive functions.

Definition 4.20 (WHILE–Programs)

WHILE–programs are ***LOOP–programs*** with an additional construct:

`WHILE` $x_i \neq$ `0 DO` P `END`

Its meaning is apparent: Program `P` is repeatedly executed as long as the value of variable x_i is different from 0. (Obviously, the `LOOP` is no longer necessary since `LOOP` X `DO` P `END` can be simulated by `WHILE` $x_0 \neq$ `0 DO` x_0 `:=` x_0 `- 1 ;` P `END`.) □

This sections ends with the statement that the *QUOGGLES* system is not equivalent to the class of μ-recursive functions.

Proposition 6: The ***extended system*** is not Turing-complete.

> **Proof of Proposition 6:** Since every μ-recursive function is Turing-computable (see for example [Bro98]), it suffices to show ***WHILE–programs*** cannot be simulated in the query language. This is straightforward since the number of iterations of any loop in the *QUOGGLES* system is fixed in advance of any iteration and cannot depend on any data calculated during the iterations.
> The number of times a `ComplexFilter_Box` executes its predicate query is fixed by the number of elements contained in its input collection. No changes or computations done within the subquery can change this. Thus, no while loop can be encoded. □

Chapter 5

Future and Related Work

5.1 Future Work

This section summarizes some aspects that are subject to improvement or are thought to be introduced into the system at a later stage of the development. Especially two items of the list presented in Section 2.1.2 shall be restated:

5.1.1 Query Optimization

Query optimization is an issue as soon as the queried graphs have more than just a few dozen nodes and edges. Optimization techniques have been widely studied. Much effort has been put into methods of speeding up information retrieval in relational database systems. [KE01] presents some of the common strategies, [JK84] is also concerned with general aspects. More sophisticated and specific methods can be found in specialized articles like [CSSK94] and [CSSK95].

Concerning the *QUOGGLES* system, some aspects can be illuminated without going into detail or trying to formalize any strategies.

Several principles of standard databases can be adopted. Thus, it will always be faster if the number of elements contained in some intermediate result can be reduced before any costly operation like a cartesian product is computed. This is known as "early selection". "Early projection" is another means in relational databases. Data that is neither used in the output nor in immediate results, can be ignored from the start. This will not occur so often in the *QUOGGLES* system, however. Columns (collections) are rarely linked with each other and data that does not appear in the result table will therefore probably not be used beforehand. When using boxes of type `ListOperations1_Box(listify)`, unnecessary data can indeed be dragged along, though.

Several operations that are executed sequentially can be combined. Think of a query having two consecutive `AttributeFilter_Box`es. This means that the list of n input elements is traversed twice. The second traversal might or might not get a list with fewer elements. The worst case is $2 \cdot n$. Those two boxes could easily be concatenated forming a single box with the two conditions linked together by the boolean operator $\wedge$. The same applies to two filter boxes that are executed in parallel. They can be connected using $\vee$, although this case is harder to detect.

The following example illustrates both, the potential for and the problem with optimizations:

Example 5.1:

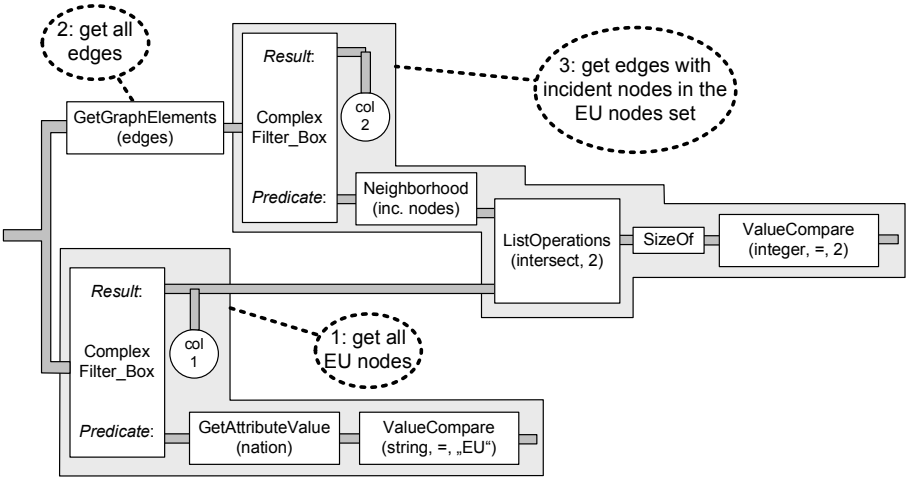


Figure 5.1: Example Query (More Basic Boxes), Possibility 1

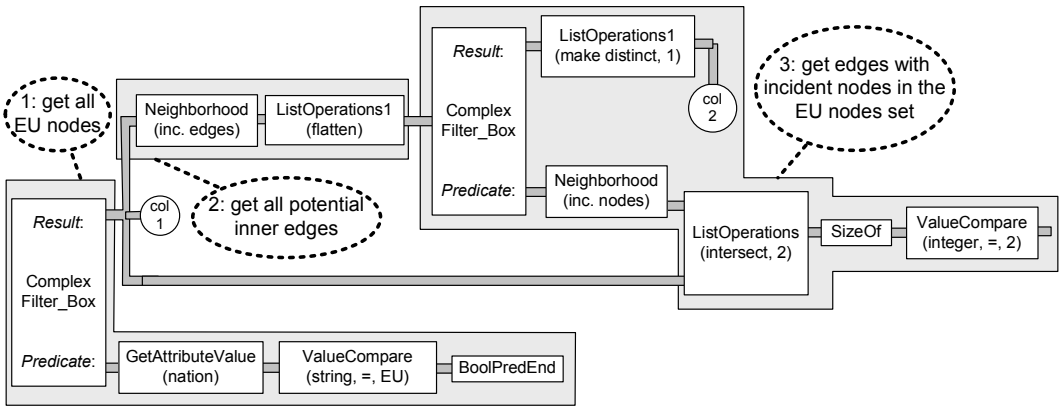


Figure 5.2: Example Query (More Basic Boxes), Possibility 2

Figure 5.1 shows an annotated query graph that solves the following problem: "*Show all inner-European connections*". The task is to identify the set C of all nodes representing European cities (via an attribute "nation") and find all edges the source and target nodes of which are contained in C. Put another way, find all edges that belong to the subgraph induced by nodes in C.

The query graph in Figure 5.1 closely follows the description given in the query statement. It finds all "European nodes" C and then checks all edges in the graph whether or not their source and target nodes are contained in C.

Another way to solve the same task is presented in Figure 5.2. After having identified the set C, only edges that are incident to a node in C are checked. The predicate is the same as in the approach before. A `ListOperations1_Box(flatten)` had to be introduced right after the `Neighborhood_Box(inc. edges)` since the latter produces (for a list of nodes as input) a

collection of edge sets. This implies that there can be duplicates in the resulting list. After having filtered the edges, a `ListOperations_Box(make distinct, 1)` ensures that there are no duplicates in the output (compare the `DISTINCT` keyword in `SQL` queries).

This will very probably be faster than the first proposal. Especially if the graph is dense, i.e. the number of edges is high, and the number of chosen input nodes is small. In this scenario, most of the edges will not even be touched by the second query graph. However, the equivalence of the two queries is by no means easy to see and optimization routines probably would have difficulties in finding the faster variant.

Another tuning opportunity can be seen in this latter example. Without doubt, the `ListOperations_Box(make distinct, 1)` is required in this query since the `Neighborhood_Box(inc. edges)` very probably produced duplicates. However, the `ComplexFilter_Box` treats all elements in its input collection the same. Therefore, duplicates are treated the same, i.e. either all or none of them are present in the output collection. This means that the `ListOperations_Box(make distinct, 1)` can as well be put right before the `ComplexFilter_Box`. Since the predicate is quite complex, the overhead on applying the duplicate elimination operation on the larger, unfiltered collection will pay off in nearly all circumstances. Although even in this rather obvious example, there exist graphs where this optimization step would in fact slow the system down: A graph for which the optimization would be counterproductive would have few connections between many European cities with each having lots of ways to emigrate to non European countries. More formal, a graph $G = (V, E)$ where the nodes in C have many edges to nodes in $V \setminus E$ but only few to nodes in E.

Situations like the one just describes complicates the optimization process. To be successful, the queries must probably be converted into some other structure for which strategies have already be found that work well on the average.

5.1.2 Queries on Several Graphs

The present system is designed to search or extract information from one single graph. For some applications, it is more important to search in a whole database for graphs that match certain criteria.

Obviously, the system could be used to apply one and the same query on every graph in the database and collect those that match or all graphs could be combined into one single graph, distinguishing them by a specific attribute or some other means. However, this technique will not yield satisfactory results.

There are existing approaches that can handle this kind of job much better. Most concepts use heavy preprocessing to be able to quickly reduce the search space. Examples are [Gro03] (a database of fingerprints of molecules), [MB99] and [GS02]. The latter generates an index that stores information for each graph and for each node. It assumes undirected and unlabelled edges but can be generalized. Nodes have some attribute (id) uniquely identifying them and some additional label. Let G be a graph in the database. From each node of G, all short paths (i.e. those that are no longer than some fixed value, for example four) are computed. For each label sequence, the corresponding lists of node ids are saved in a table. The number of distinct paths having the same label sequence is stored for each node in G. This is used as a hash-function and is referred to as *fingerprint*. The fingerprint of the query can then be compared to that of the graphs in the database, excluding many of them from a more detailed search in one quick run. The remaining graphs are candidates and are checked using the pre-built table.

Incorporating such strategies into the *QUOGGLES* system would probably result in a complete redesign.

5.1.3 Consecutive Execution of Algorithms

This part is put into the section about future work since several details have still to be examined. The functionality described here highly depends on the *Gravisto* system. Before all design decisions can be finalized, several issues have to be treated by the developers of *Gravisto*.

One of the advantages of the *Gravisto* toolkit is that algorithms of any sort can be written and easily plugged into the program. As is the case when designing some programming library, it serves reuse and simplicity if algorithms are kept small: Often, a complicated method can be split into several parts that are easier to understand, maintain and use. Those parts may well serve others to build different methods on top of them.

Assume there are several algorithms that produce graphs of some size, either randomly or according to some restrictions. It might be sensible to test a newly written layout algorithm on such graphs. However, it requires considerable work to carry out these plans. One has to write another algorithm that executes the graph generator and the layout algorithm sequentially. Besides having to implement the required interfaces etc. to comply with the *Gravisto* structure, every time a different graph generator routine is wanted, the code has to be adjusted, recompiled and the *Gravisto* editor restarted.[1]

Such tasks of having to execute one algorithm after the other, the second taking some output of the first as input are quite common. The pipeline system of the *QUOGGLES* system suggests itself as a remedy against the mentioned obstacles. First, the new `Algorithm_Box` is introduced:

- **Algorithm_Box**
  This box executes a specified algorithm. It can access all algorithms that have been registered with the *Gravisto* system.

  Input(s): The number of inputs depends on the number and type of parameters of the algorithm. The box has always at least one input. For every parameter of the algorithm, the box has one input. In addition to that, a "dummy" input is added defining the execution order of several algorithms (i.e. to be able to put them into a sequence and connect them to the ***standard input box***). If the algorithm has a parameter that takes a selection (class `SelectionParameter`), it is assumed that this input can be used instead and no "dummy" input is added. But this design will be subject to discussion.

  Output: The box has always at least one output. If the algorithm implements interface `CalculatingAlgorithm`, the output will be whatever result the algorithm produces. If it does not implement this interface, the algorithm is not designed to return some value (it works on the selection, the graph or attributes). Some way has to be provided to sequentially concatenate two algorithms. Therefore, a "dummy" output is added that is not designed to yield some sensible value but serves the only purpose to fix a computation order for the algorithms.

  In the current version of the implementation, the value of the very first parameter is returned.

[1] In future versions of *Gravisto* the last step might be unnecessary

Parameter: One of the algorithms registered at the *Gravisto* editor.

The algorithm specified by the parameter will be executed. If the algorithm needs parameters $p_1, p_2, \ldots, p_n$, the `Algorithm_Box` will need

$$p_1, p_2, \ldots, p_n, d$$

as input, where d is the optional "dummy" input.

To be able to use the `Algorithm_Box`, it must be possible to set its inputs. Since these must not be ordinary constants but classes implementing interface `Parameter`, a special input box needs to be added (another approach would be to extend the `Constant_Box` presented on page 24, but a separation of the basic system and this extension is desired):

- **CreateParameter_Box**
  This box creates a parameter of a specific type with a given value.

  Input: *none*

  Output: A parameter (i.e. an object implementing interface `Parameter`) having the specified type and value.

  Parameter 1: The (fully quantified) name of a class implementing interface `Parameter`. This type of parameter will then be created.

  Parameter 2: A special component that allows the user (only) to enter a value compatible with the type specified by the first parameter. A boolean value can be set using a check box, for example. Refer also to the description of the basic `Constant_Box` on page 24.

  Since it is important that dynamic values (i.e. values that have been calculated by some query and have not been specified by the user) can be used as parameters for algorithms, a box is introduced that converts objects to parameters (if possible). This box then probably makes the `CreateParameter_Box` redundant since it can be simulated by a combination of a `Constant_Box` and this box (if that can be done depends on the parameters that exist in the system).

- **ConvertToParameter_Box**
  This box creates a parameter of a specific type and sets the input as value.

  Input: *none*

  Output: A parameter (i.e. an object implementing interface `Parameter`) having the specified type and value.

  Parameter: The (fully quantified) name of a class implementing interface `Parameter`. This type of parameter will then be created.

- **ConvertFromParameter_Box**
  This box extracts the value of a parameter.

  Input: An object implementing interface `Parameter`.

  Output: The value of the input parameter.

  Parameter: *none*

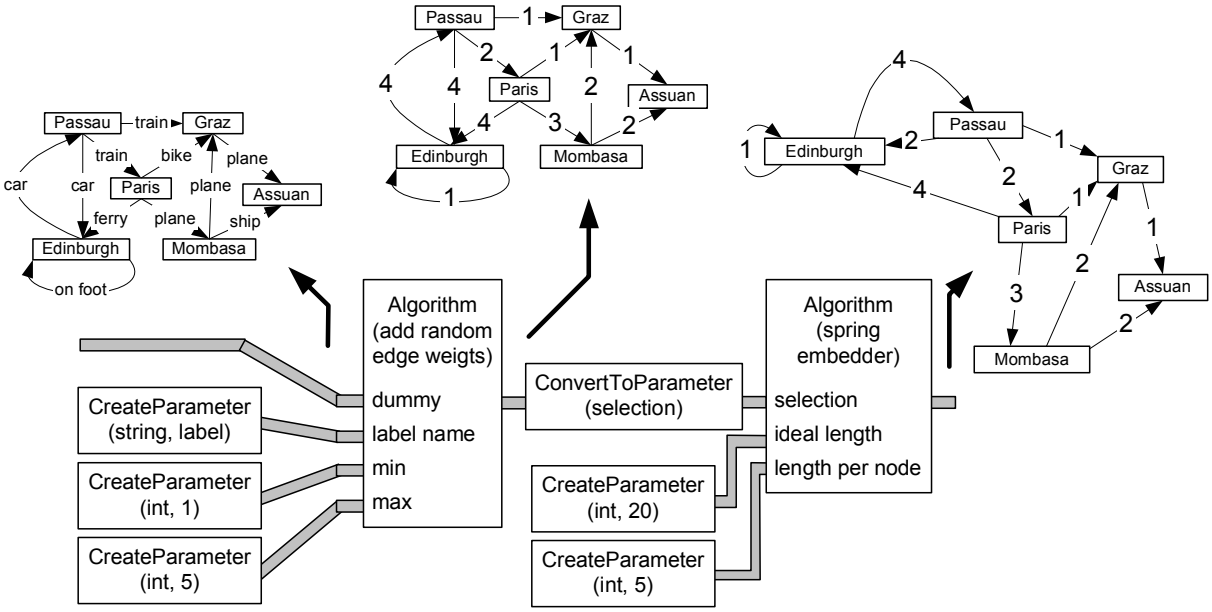


Figure 5.3: `Algorithm_Box`

Figure 5.3 shows how two algorithms can be concatenated. The first assigns random weights to all edges of the ***current graph***, the second uses a spring embedder to layout the graph. The first algorithm needs three parameters, the id of the attribute that will get the values assigned and the interval in which those values are allowed to be. The spring embedder needs three parameters, the last two of which specify ideal edge lengths. The first is a `SelectionParameter` that holds the nodes on which the layout algorithm is applied. Since the first algorithm gets all graph elements set as its "dummy" input and this never gets changed, the output (collection of graph elements) from the first `Algorithm_Box` needs only to be converted to a `SelectionParameter`. This can be done without any problems.

Every time the query is run, new random edge weights are assigned and the spring embedder produces a different result since it takes edge labels into account.

Besides everything that has been marked as being in an early version of development during the description of the functionality, storing the state of the `CreateParameter_Box`es requires saving the special `ValueEditComponent`s they use. This has not yet been implemented in the *Gravisto* system.

It also remains to be checked if it is worthwhile to go further in this direction. One of the next logical steps would be to introduce constructs like `IF ... THEN ... ELSE ...;` to be able to decide which algorithm should be called (a complex algorithm for small graphs, a special method if the result of the last algorithm is a tree, and so on). This quickly leads into research about graphical programming languages and will probably not be appropriate.

Nevertheless, the potential of the *QUOGGLES* system can be seen and the semantics of those boxes will become clearer as soon as the design of the corresponding components in the *Gravisto* editor has been finished.

5.2 Related Work

When classifying methods for querying data stored in some (graph) structure or database, three different approaches can be found:

First, the superiority of the SQL query language must still be accepted. Therefore, many researchers try to use this well analyzed way and either push SQL to communicate well with graphs or transform the graph data structure to comply with (nearly) ordinary SQL statements:

Many established but also new database applications such as hypertext applications ([Tom89]), geographic information systems ([Güt91]) and heterogeneous information integration, etc. require modelling and querying of graph data. [DM94] shows an application in genome research.

The interest in breaking the World Wide Web down to known structures and apply fast and reliable search strategies has grown as has the web itself. Much work is being put into that area. One branch of it concentrates on modelling the web as graphs ([MMM96], [SBH+00]). [ABS00] combines relational approaches, semi-structured data and XML.

In the last years, XML technologies have become famous and widely used. Although XQuery ([W3C03]) has become a W3C recommendation, much work still needs to be done to provide a user-friendly query language. Therefore, there is much ongoing research in the field of semi-structured data and XML: [Abi97], [AQa97] and [BDFS97] to mention but a few important references. The new SQL 3 standard includes an XML SQL extension that tries to store and access XML data in a more appropriate way.

In the XML context, a system called RDF has appeared. The Resource Description Framework (RDF, [DB03]) is a language for representing information about resources in the World Wide Web. There exists a RDF/XML syntax for writing RDF data, but the conceptual model is a graph. Besides storing RDF data, most application want to retrieve and select some specific information from it. There are several proposals for query languages, most of which are extensions or modifications of the known SQL concept and keep syntactical similarity. Examples are RQL ([Kar03]) and SQUISH ([Mil02]).

Database models like relational systems have long tradition and many results have been found to effectively use them. Object-oriented, functional and logical approaches are advancing. It is obvious that people try to exploit the advantages of known database systems for graph applications. In [GBP94] and [GPG90] as well as in [Gut94] and many more, graph-oriented object data base models are examined. Güting ([Güt91]) shows how to extend a spatial database system by graphs and in [BRS90], the SQL query language is extended to better fit retrieving data from graphs.

An interesting object-oriented graph data model and OQL like graph query language (cf. [AM98] for more information about OQL) is used by GOQL ([SÖÖ99]). In the domain of multimedia presentation graphs, the authors present their data model, a syntax that is very close to SQL queries and how they translate and implement their queries in an operator-based algebra called *O-Algebra*.

Second, a completely different approach has been chosen that is generally subsumed by the term "graphical query languages". The common ground for these languages is that they use some sort of graphical way to state the queries. Nearly all of those perform to a paradigm introduced 1975 by M. Zloof, [Zlo75]: *QBE, Query By Example*. Although the implementation can be quite

different, they share the same idea. This idea can most intuitively be described using a database of graphs. A query might now consist of two nodes with specific labels. They are connected by an edge with a regular expression as label. The query interpreter will then try to match the graphs in the database with this model according to some rules. [ÖW93] and [BCCL91] give brief overviews of visual database query languages while [ÖW89] is a more fundamental work on extending relational calculus. An example of such a language is *Noodle*, described in [MR93] or the *DAML Query Language* `DQL`, [Com03].

This graph matching has also been examined quite exhaustively and fast algorithms have been proposed. [SWG] provides an extensive and very good survey of existing graph matching algorithms and query languages. Basically, graphical query languages have evolved in two branches. One kind uses tables, nested forms, etc. to create an interface (examples are [ÖMÖ89] and [Bal95]). These build on a method called *Summary-Table-By-Example*. The other kind uses sample graphs, i.e. nodes and edges to express queries. A recent publication about the latter approach is [GS02], "*A Fast and Universal Method for Querying Graphs*". It uses fingerprints, i.e. small information items storing crucial characteristics of the graphs in the database, thus being remarkably fast in experiments.

Third, since some applications require inexact matching, some querying techniques allow the query condition to be blurred and retrieve data that (only) closely matches the given criteria. Methods, examples and explanations about how these systems are evaluated using metrics like precision and recall are described in [YN99].

It is hard, if not impossible, to classify the *QUOGGLES* system according to the systems just presented. It does not use a relational base, although it draws on some of its concepts and operators. Neither does it provide any example graphs for which matches are searched in the database, although it graphically represents its query in form of boxes. It definitely does not fit into the third category, however.

Two approaches can be found that better match the *QUOGGLES* system's type of query language:

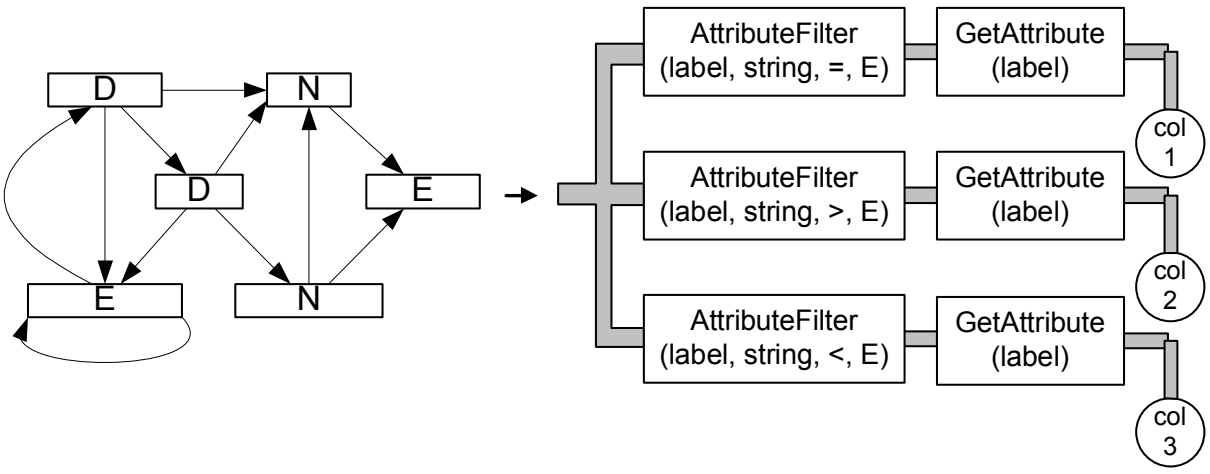
[MC95] presents a general purpose object query system and language called `QBI` (Query By Icon). The structure and characteristics of the underlying database is hidden from the user as much as possible. `QBI` presents data as classes of objects and (generalized) attributes. The idea is to provide the user with icons that represent those objects. A query is build by choosing appropriate icons and connecting them. This query is transformed into a triple holding the chosen object o, a selection condition c and a set of attributes a that should be presented as the result (*select-project paradigm*). In that, the query resembles strongly the `SQL` formulation `SELECT A FROM O WHERE C`. The representation of the query and the way it has been created differs considerably, however. Attribute representations are dragged into a special condition window where they can be concatenated by some valid connective like *is equal to*.

`QBI` also offers a natural language text interpretation of the query and images that visualize the function and type of objects. Of course, this can only be sensible for specific and well-known applications for which the system must then be adapted. Formulating more complex queries with nested conditions will prove more difficult because of the generality of the way the data is presented. The whole database must be presented by objects and the notion of generalized attributes (representing elementary properties in *entity relationship* ([KE01]) diagrams). The

identification of objects supported by images and the compatibility of objects and connectives being represented visually, relieves the user of any having to think about details of the data structure.

Thus, `QBI` surely is closer to the *QUOGGLES* system than the other languages, but it still follows too tightly the `SQL` specific idea of a `SELECT ... FROM ... WHERE ...`.

As a another approach, [Pro03] seems to have similar ideas as the author of this thesis. A student project is started at the Rijksuniversiteit Groningen, Netherlands, that is supposed to design a query system for biological gene networks. Nodes depict genes, proteins, dna, etc. while edges represent relations like transcription and protein interactions. The basic idea is that the source graph is processed by modules that can be exchanged, maybe concatenated and parameterized from outside. The main difference between this concept and the *QUOGGLES* system will probably be the granularity of the operations. Whereas there are several operations working on parts of graphs presented in this thesis, the transformations in [Pro03] will most likely work on whole graphs only. It remains to be seen how this project develops.



col 1	col 2	col 3
E	**N**	**D**
E	**N**	**D**

Figure 5.4: – *no comment* –

Chapter 6

Summary

In this thesis, the *QUOGGLES* system has been presented. It includes the design and implementation of a query language on graphs. The basic concept is an extended pipelining method: A set of operations (called boxes) with well-defined semantics is defined. These operations can be arbitrarily concatenated as long as the input and output specification allows for that. Having the collection of graph elements of some graph as input, information about nodes, edges and attributes flows through the pipeline and gets transformed along way. At every junction of two boxes, the data that is currently flowing at this point can be put into a result table. This maximizes the amount of information retrievable via one single query.

It has been possible to show that the language is relational complete (i.e. can express all operations of relational algebra and equivalent concepts) and therefore qualifies as a fully-fledged query language. In fact, the system can even simulate the queries of the famous relational database query language `SQL` which provides several additional features.

To finish the considerations about the power of the *QUOGGLES* system, it has been proven that, extending the basic "read-only" system by a box that can change the values of attributes, it coincides with the class of primitive recursive functions. Turing-completeness is neither wanted nor reached.

Important goals that have been achieved and distinguish this from other systems are its ease of extending the language and the smooth cooperation with a graph editor for visualization and information exchange purposes.

The possibility to extend the system has effectively been demonstrated by means of an example in section 5.1.3.

Appendix A

Implementation Details

A.1 Technical Terms

Throw and catch an `Exception`: Whenever an error occurs somewhere in the system, Java is designed to react by a special mechanism called "throwing an `Exception`". A multitude of types of `Exception`s are defined by Java itself. The implementation of the *QUOGGLES* system adds several new types, among others:

- `InvalidInputException`: Thrown if an object has been given to a box that cannot be handled by the box.
- `InvalidParameterException`: Thrown if a parameter of a box has a value that cannot be handled by the box.
- `InputNotSetException`: Thrown if a box is executed without all inputs having been set before.
- `BoxNotExecutedException`: Thrown if the outputs of a box are demanded without the box having been previously executed.

All these `Exception`s are subsumed by the generic `QueryExecutionException`.

A developer can expect the possible occurrence of an exception (called a checked exception) and handle it appropriately. This is called "catching an exception". All exceptions defined by the *QUOGGLES* system are checked and, if they cannot be handled otherwise, produce sensible error messages to the user.

`Stack`: This is a class implementing the standard stack structure. It allows methods `BOOLEAN empty()` (also called `isEmpty`), `push(OBJECT)`, `OBJECT pop()` and `OBJECT peek()` (often called `top`), having their familiar meaning.

A.2 System Architecture

Figure A.4 on page 129 shows an `UML` diagram of the general architecture of the system.

> **Remark:** The implementation uses a top level package called `quoggles`. To shorten names, all package, class and interface references are specified relative to that package. A class `quoggles.boxes.Box` will be written as `boxes.Box` instead. Exceptions like `QueryExecutionException` are contained in package `exceptions` and are not fully quantified in this chapter. In method listings, parameters and return types are not fully quantified, either.

The core class is `QMain`. It provides a system entry point via method `void main(String[] args)`. This allows starting and using the system separated from *Gravisto*.[1] Its other main function is to act as communication center. Other classes can call its methods to invoke operations on different parts of the system. It has very little functionality itself but delegates most of the work to the relevant classes. This provides a good separation of components forming logical units.

Main modules of the system are:

- The window where the query is created (`QDialog`)
- The class responsible for reacting to user input (`QGraphMouse`)
- A module assuring that the representation of the query graph and the query graph itself are kept synchronized (`QGraphConnect`)
- A set of methods working on the query graph itself (`QGraph`)
- The module for executing and resetting a query (`QRunQuery`)
- A dialog that displays the results of executing a query (`QResultDialog`)
- Auxiliary methods useful to several of the modules listed above (`QAuxiliary`)

A.2.1 Implementing the Gravisto Plug-in Concept

The *Gravisto* system requires plug-ins to create a class that implements `org.gravisto.plugin.GenericPlugin` or extends class `org.gravisto.plugin.AbstractGenericPlugin`. This implementation uses the latter approach for class `QPlugin`. The query system is implemented as an algorithm. Therefore, `QPlugin` provides one algorithm using those lines:

```
private QAlgorithm alg = new QAlgorithm();

public QPlugin() {
    this.algorithms = new Algorithm[1];
    this.algorithms[0] = alg;
}
```

[1] Using it as a stand-alone system requires the implementation of an input box that is capable of importing existing graphs.

Class `QAlgorithm` implements the required interface `org.gravisto.plugin.algorithm-.Algorithm` and provides an `void execute()` method. In this method, the core class `QMain` is used that displays the main dialog of the *QUOGGLES* system and sets the graph on which the query is be run. The rest of the work is delegated to the dialog.

```
if(qMain == null) {
    qMain = new QMain();
    qMain.setGraph(graph);
} else {
    qMain.setGraph(graph);
}
qMain.showQDialog(false);
```

This code fragment prevents the instantiation of the core class each time the query system is started.[2]

A.2.2 Query Graph Data Structure

Boxes (package `boxes`)

***Box*es** are simple objects that implement interface `boxes.IBox`. This interface contains several methods. Only a few of them are will have to be overridden when creating a box. The default implementation found in `boxes.Box` will suffice for most needs (Section A.4 explains which methods need to be implemented by hand). The methods can be divided into five groups:

- Input / Output specification

  `int[] getInputTypes()`: Specifies the allowed types of the inputs (and implicitly their number). Uses constants defined in `constants.ITypeConstants`.

  `int getNumberOfInputs()`: Sets the number of inputs of the box.

  `int[] getOutputTypes()`: Specifies the types of the outputs (and implicitly their number). Uses constants defined in `constants.ITypeConstants`.

  `int getNumberOfOutputs()`: Sets the number of outputs of the box.

- Execution

  `void setInputs(Object[]) throws InvalidInputException`: Provide the box with all its inputs. The inputs are provided in an array, the size of which is equal to the number of inputs. Throws an exception if the type of the input is inacceptable for this box.

  `boolean isInputSet()`: If this method returns true, the inputs have been set and the box is ready to be executed.

[2]Since the user can work on the *QUOGGLES* system and *Gravisto* at the same time, there is in general no need to restart the system several times. `QMain` observes the *Gravisto* system in a way that it is informed when another graph gets activated and the query is automatically run on that graph. Since the query runs on the same graph object as *Gravisto* works on, changes within the graph itself are immediately present in the *QUOGGLES* system.

`void setInputAt(Object, int) throws InvalidInputException:` Set the input of a specific input index. This method is necessary because a box with several inputs might not get all of its inputs set at the same time.

`boolean isInputSetAt(int):` Returns true if the input at the specified index has been set.

`void execute() throws QueryExecutionException:` Executes the box. This is only possible if all inputs have been set since the last call to `void reset()`. After successfully executing this method, the output of the box is ready to be retrieved.

`boolean hasBeenExecuted():` Returns true if the box has been successfully executed. This means that the outputs are valid.

`Object[] getOutputs() throws BoxNotExecutedException:` Gets the outputs of the box in an array the size of which is equal to the number of outputs. Throws an exception if the box has not yet been executed.

`Object getOutputAt(int) throws BoxNotExecutedException:` Retrieves the output at a specific output index.

`void reset():` Resets the box. Apart from internal consequences, this means that the box needs to be set its inputs and executed before any outputs can be collected.

The normal sequence in which these methods are called is

$$\text{setInputs} \longrightarrow \text{execute} \longrightarrow \text{getOutputs}$$

- Methods Concerned With Parameters

  `Parameter[] getParameters():` Returns an array of `org.gravisto.plugin.parameter.Parameter`s that the box defines.

  `void setDefaultParameters():` This method is called when a box is created. A box should can use that to set default values for its parameters.

  `void setParameters(Parameter[]):` Sets new parameters.

- Methods Used for the Graphical Representation

  `IBoxRepresentation getGraphicalRepresentation():` Returns an object that can display the box.

  `String getId():` Returns a string used as short description and for identification purposes.

  `void setBoxNumber(int):` Every box is associated with a number that uniquely identifies it in the current query.

- Specialized Methods

  `Node getNode():` Returns the node associated with the box.

  `void setNode(Node):` Sets the node associated with the box.

  The next four methods are used by special boxes that need to execute some subquery themselves. Examples are `SubQuery_Box`es and `ComplexFilter_Box`es:

  `boolean needsQueryRunner():` Returns true if the box needs to execute some subquery itself.

`void setQueryRunner(RunQuery)`: Sets an object that can execute subqueries.

`void setCurrentNodesTodo(Stack)`: The stack of nodes that still have to be executed. Since the box might execute boxes that already had been scheduled, this list must be updated.

`void setCurrentResult(ArrayList)`: Sets the list that contains the result of the query calculated so far. If any `Output_Box`es are executed, this result must be updated.

Graph and Attributes (package `querygraph`)

A query graph consists of boxes and edges. This has been implemented using the graph data structure that the *Gravisto* system provides. It uses nodes and edges to represent the graph. Therefore, a one to one mapping has been added to simulate the identity of nodes and boxes. This was achieved by creating a special attribute called `querygraph.BoxAttribute` that encapsulates a ***box***.

As an important feature of the system queries can be saved and loaded. Thus, some way to serialize query graphs and parse serialized data had to be implemented. The serializer of *Gravisto* writes a graph to a file using the `GML` format. This implementation could partly be used, but had to be extended to cope with the new attribute type.

In view of the probable adjustment of the *Gravisto* serializers and parsers to an `XML` format, the readers and writers have been extended to accept `XML` strings. Not only attributes have to be saved and loaded. Parameters define the state of each box. Among other things, this common requirement led to the design of `org.gravisto.plugin.Displayable`, an interface that both, `org.gravisto.attributes.Attribute` and `org.gravisto.plugin.parameter.Parameter` extend. This interface has been enriched with a method `String toXMLString()` that produces an `XML` representation of the attribute or the parameter, respectively.

The standard format for attributes looks like that, `CLASSNAME` being the fully quantified class name of the attribute and `VALUE` a (`XML`) string representation of the value:

```
<attribute classname=\"CLASSNAME\">
  <value>
    VALUE
  </value>
</attribute>
```

The quoted quotation mark is needed since the whole string is used as the value of an attribute. In `GML`, strings as attribute values must be put in quotation marks. Therefore, quotation marks inside the attribute value must be specially marked.

A standard implementation additionally encloses the value of the attribute into a `<![CDATA[ ...]]` section to prevent the string representation of the value from making the `XML` string unparsable.

The serialization of parameters looks similar, adding `XML` attributes for its name `NAME` and description `DESC`:

```
<parameter classname=\"CLASSNAME}\">
   name=\"NAME\" description=\"DESC\">
```

```
    <value>
      VALUE
    </value>
  </parameter>
```

The string value of a parameter can be as simple as "true" or "false" (for a `BooleanParameter`). Others are more complicated: The `OptionParameter` has been introduced since several parameters used in the *QUOGGLES* system have a predefined set of options to choose from. A combo box is an obvious and intuitive way to represent such a set of options. The string representation of this parameter includes an `options` tag that contains several option elements. The latter specify the type (class name) and value of each option. In addition to that, a `properties` element contains information about the selected option and whether or not user defined options can be added.

Thus, a query graph can be written in `GML` format: As every node is associated with a box, each has an `BoxAttribute` that must be serialized. Its `String toXMLString()` method returns an `XML` string containing the box's class name (`CLASSNAME`, the `XML` string representation of its parameters `PAR1`, `PAR2` ... `PARn` and information about position and size of the graphical representation (X, Y, W, H).

```
<box classname=\"CLASSNAME\"
  <parameters>
    PAR1 PAR2 ... PARn
  </parameters>
  <geometry x=\"X\" y=\"Y\" w=\"W\" h=\"H"/>
</box>
```

When reading such a `GML` file, the parser instantiates nodes, edges, attributes and parameters according to the specified class name and initializes them. For more details, consult the implementation of the

`org.gravisto.plugins.ios.importers.gml.GMLReader`

and the accompanying parser files.

Icons (package `icons`)

Creating a query graph, the user must be able to add and remove boxes. While the removal can be done by double clicking on the box, icons are provided that act as buttons. Double clicking on such a button will add a box of the respective type to the graph. Giving the icons of all boxes a common look and letting them display basic information of the box they represent adds to the intuitive handling of the system.

Figure A.1 shows the `UML` diagram of the icon interface and its implementing class.

Apart from being able to provide an arbitrary panel as icon, the *QUOGGLES* system provides a standard implementation that generates an icon for every type of box. Figure A.2 shows the icon that has been automatically generated for a `SortBy_Box`. The top label displays the name of the box as is specified in the `boxes.xml` file. The text in the center can be set in the constructor of

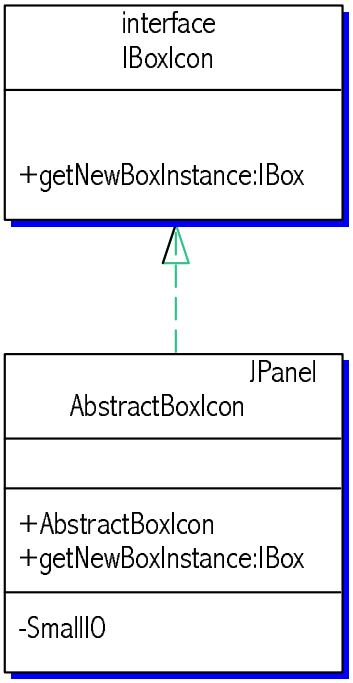


Figure A.1: Icon, Class Diagram

Figure A.2: Automatically Generated Icon

the icon itself. As can be seen, the number of inputs and outputs is schematically displayed by the number of lines to the left and right of the central rectangle.[3]

Box Representation (package `representation`)

The creation of a query graph means the addition of boxes and edges. This is most easily done using a graphical user interface. Icons have already been presented as the means to interact with the system and add boxes. As is customary in software and language design, the semantics and any graphical representation should be separated. This is reflected in the design of interface `representation.IBoxRepresentation` and the associated classes, as presented in Figure A.3.

A class implementing this interface must provide the following methods:

`IBox getIBox()`: Gets the box represented by this `IboxRepresentation`. The box is set via the constructor and cannot be changed to maintain the association.

`void updateGraphicalRep()`: This method is called whenever the state of the box changed, invalidating the graphical representation.

[3] These small lines are in fact instances of the nested class `SmallIO` that can be seen in the class diagram.

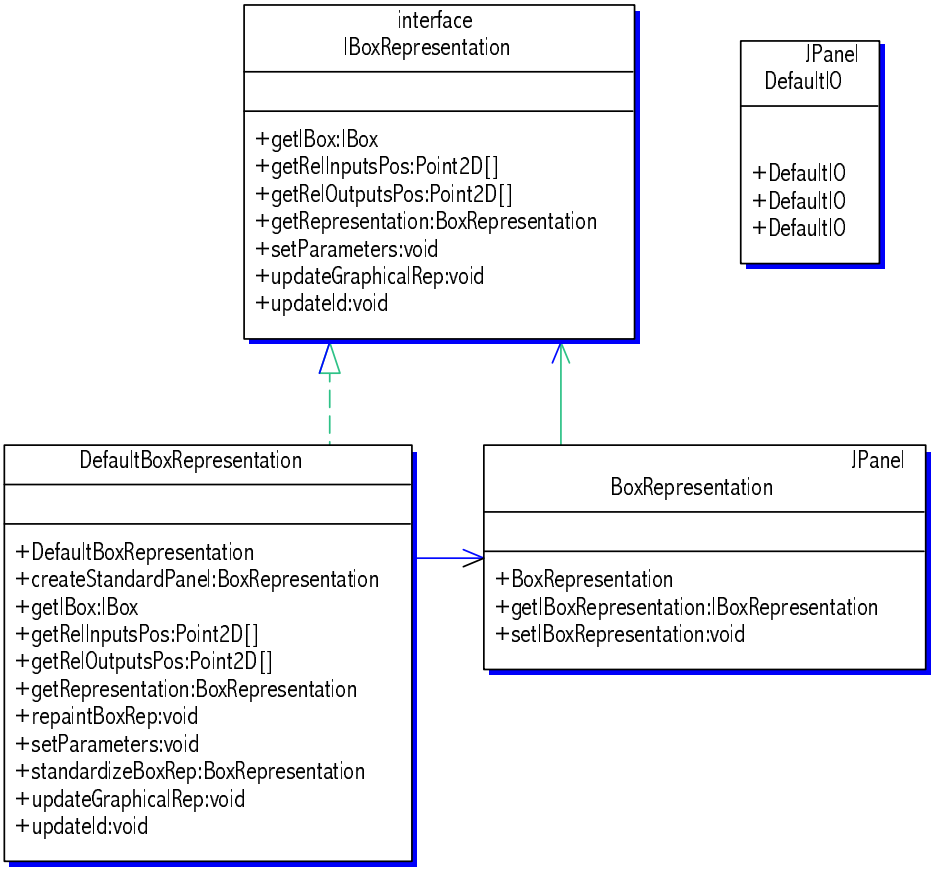


Figure A.3: Box Representation, Class Diagram

`void updateId()`: An optimization: If not the whole graphical representation must be updated but only the identification number changed.

`BoxRepresentation getRepresentation()`: Returns an instance of class `BoxRepresentation` that is a direct subclass of a Java Swing `JPanel`. This will be displayed by the system to represent a box.

`Point2D[] getRelInputsPos()`: For the input with index i, the returned array contains a point $p = (x, y)$ at index i. It specifies the relative position of where the input is displayed. This is used as docking points where the outputs of other boxes can be attached.

`Point2D[] getRelOutputsPos()`: The same as `getRelInputsPos()`, only for outputs.

`void setParameters(Parameter[])`: Many representations will provide some means of changing the parameters of the box (a combo for changing the value of an `OptionParameter`, for instance). This method is called to update the parameters.

All methods are implemented by class `representation.DefaultBoxRepresentation`. It is useful to extend this class when creating a new box representation. Without overriding any methods, this is the default behavior:

- Display all inputs as horizontal bars[4] distributed vertically to the left.

[4]Instances of class `representation.DefaultIO` are used to display those

- Display all outputs as horizontal bars, distributed vertically to the right.
- No parameters are visualized.

The class additionally provides useful final methods that makes creating box representation that fit into the general look-and-feel considerably easier:

`final BoxRepresentation standardizeBoxRep(BoxRepresentation, JComponent):` This method should be used by all implementors of specific box representations. It takes a `BoxRepresentation` as first parameter. This will most of the time be an empty `javax.swing.JPanel`. The method will make it look in a standard way, using the component provided as second parameter.

One possible use is to provide a `javax.swing.JComboBox` as second parameter. The method will then create a standard representation, using the combo box as central component. The combo box can then be used to display and provide means to change a parameter of the box. All box representation implement for this thesis use this technique.

`void updateInputOutput():` Can be called to synchronize the display of inputs and outputs if their specification changed (if, for example, a parameter is changed that affects the number of inputs of the box).

`final void repaintBoxRep(ActionEvent):` A call to this method tells the system that an event occurred that makes it necessary to repaint the box representation. The event is passed as a parameter. As an example, consider a user using a combo box to change the value of a parameter that controls the number of outputs. The change must result in the connections between this box and others to be revalidated.

A.3 Executing a Query

The process of interpreting a query and finding the result of it is called "executing a query" and refers to a call of method `runQuery`:

After having created a query, it is executed by a class implementing interface `auxiliary.RunQuery`. This interface requires only one method:

```
List runQuery(Graph qGraph, Collection sourceBoxes,
              boolean internalUseOnly, List curResult,
              Stack boxesTodo)
        throws QueryExecutionException
```

The parameters have the following meaning: `qGraph` is the query graph that should be executed. The collection of `sourceBoxes` contains boxes the boxes from which the execution starts. Normally, that will be the set of ***standard input box***es (see Definition 3.3 on page 20). The parameter is necessary since some boxes execute parts of a query graph. The boolean flag `internalUseOnly` indicates whether or not the current call to `runQuery` should produce any graphical output or not. Any boxes that process subqueries during execution of the main query will pass *true* for that parameter. The list `curResult` denotes the current ***result table*** (Definition 3.10) as a list of lists (columns). The first call of `runQuery` will pass an empty list, other calls will pass the current value on. An empty stack is the initial value of parameter `boxesTodo`. It

is passed on by other calls to `runQuery` and contains all boxes that have already been scheduled to be executed.

The method throws a `QueryExecutionException` indicating that some error has occurred during the execution.

Algorithm A.2 describes an implementation of that method. Several auxiliary methods need to be introduced before the way it works can be presented in detail.

`VOID pushOutputs(BOX b):` The given box b is assumed to have been executed. This method identifies the out-neighbors of b and pushes its outputs to the appropriate inputs of those neighbors. This method is called after a box has been executed.

`VOID pushOutputs(BOX b, INT i):` This method behaves like `VOID pushOutputs(BOX b)`. It pushes only the data that is read at the output with index i, however. It is used after execution of a `ComplexFilter_Box`. The predicate subquery at the output with index two has already been executed. Therefore, only the data at the first output needs to be passed on.

`VOID saveOutput(BOX b):` This method calculates the ***table view*** of the data from the output of b and saves it into the ***result table*** at column i. i is the value of the parameter of b, i.e. $i = \text{getParamValue}(b, 1)$.

`INT getInIndex(EDGE e):` From the given edge e, this function returns the associated ***input index***. It is the index of the input of e's target box (see Definition 3.5).

`INT getOutIndex(EDGE e):` From the given edge e, this function returns the associated ***output index***. It is the index of the output of e's source box (see Definition 3.5).

`VOID executeInputBox(BOX b):` This method is responsible to check the type of the input box b and initialize it according to that type. The ***standard input box***, for instance, gets to know the ***current graph***. A selection input box would get informed about the currently active selection in the ***current graph***. Afterwards, the box's `execute` method is called as is done with any other box.

`VOID remove(STACK s, OBJECT o):` This `remove` method extends the default stack operations. It enables deleting any element from the stack s if necessary and is mainly used in the sequence

```
stack.remove(obj);
stack.push(obj);
```

Those two commands move *obj* to the top of the stack. If it has not previously been on the stack, the `remove` method does nothing and the second command pushes *obj* onto the stack.

`BOOLEAN noChanceToExecute(BOX b):` Returns *true* if it can be found out that this box can definitely never be executed due to missing input(s). This can be if

Basic Idea:
Executing a query means executing each of the boxes contained in the query graph. The graph is traversed in a depth first search fashion. Whenever a box is reached that cannot be executed because of a missing input, the algorithm tries to get that input. This is done by following the corresponding edge backward, i.e. to its source box. This backtracking is repeated until a box can be executed and the depth first strategy takes over again. This is a common way to implement a topological sorting method on a graph. All query graphs have a topological sorting since they are directed and acyclic.

Details Executing Boxes:
The term "executing a box" means calling method `execute` on this box. Most boxes work according to the sequence

$$\text{setInputs} \longrightarrow \text{execute}$$

The query executing algorithm uses the more appropriate sequence

$$\text{execute} \longrightarrow \text{pushOutputs}$$

After having executed a box a, method `pushOutputs` sets the inputs of each box b to which an edge from a exists. Then, `execute` and `pushOutputs` can be applied on b. Following the strategy described in the previous paragraph, this repeats until all accessible boxes have been executed.

This system does not work for the very first box accessed by the algorithm, though. This must be an ***input box*** which is one of the following two boxes that need special treatment:

`Input_Box`**:** Since such a box does not have any inputs, it must get its data set directly from the system by calling a special method `VOID executeInputBox(BOX b)` (see description on page 120).

`Output_Box`**:** In addition to calling `execute` and `pushOutputs`, this box must publish its output to the ***result table***. This is done by method `saveOutputs` described at the beginning of Section A.3.

Algorithm `addInNeighbors` (A.1) is called on a box b whenever this box cannot be executed since one or several of its inputs are missing. It updates the stack of boxes that still need to be executed.

Those in-neighbors of the given box b that have not already been executed are put on top of the stack. If no such box can be found, an exception is thrown, indicating that some inputs cannot be set.

In the event of a `ComplexFilter_Box`'s second output is needed for b, method `setNotExecuted` (special to the `ComplexFilter_Box`) must be called (lines 4-6). The reason is the following:

Whenever the predicates of two `ComplexFilter_Box`es are joined in some way (cf. the simulation of the cartesian product on page 76), one of the two, call it *cb*, is executed several times without being reset. If it were reset, only the result of the last execution would be available. That means that, from the moment after the first execution, *cb* is treated as "executed". The boxes connected to the second (predicate) output of *cb* are reset, since their input changes every time *cb* is executed. As a consequence, algorithm `addInNeighbors` is applied on all of them. If method `setNotExecuted` is not called, *cb* will never be added to the stack and an error will be produced. The box connected to the second output of *cb* will not get its input.

Algorithm A.1 addInNeighbors(*box*, *boxesTodo*)

Input: A box *box*, stack *boxesTodo* of boxes already scheduled to be executed
Output: The updated stack

```
    foundNewInput ← false;
 2: for all e ∈ getInEdges(box) do
      nb ← e.getSource();
 4:   if (nb instanceof ComplexFilter_Box && getOutputIndex(e) == 2) then
        ((ComplexFilter_Box)nb).setNotExecuted();
 6:   end if/* Only add boxes that have not already been executed: */
      if (!nb.hasBeenExecuted()) then
 8:     remove(boxesTodo, nb);
        boxesTodo.push(nb);
10:     foundNewInput ← true;
      end if
12: end for
    if (!foundNewInput) then
14:   throw new QueryExecutionException(Missing Input);
    end if
16: return boxesTodo;
```

Details of the Query Execution Algorithm:
Algorithm A.2 shows an implementation of the `runQuery` method. It consists mainly of a loop processing boxes from the stack `boxesTodo` until it is empty. The boxes found in the list `sourceBoxes` are initially added to the stack.

Lines seven to eleven treat boxes that execute subqueries themselves. This is indicated by a *true* return value of method `needsQueryRunner()`. To be able to run queries, these boxes need a class implementing interface `auxiliary.RunQuery`, the current ***result table*** `result` and the stack of boxes `boxesTodo`.

After executing a box b and pushing its outputs to its successors in the query graph (lines 13 to 21), the statements in lines 21 to 27 add the candidate boxes to the stack: Those boxes that have gotten an input from b can no be examined. If they have all their inputs set, they can be executed in on e of the next iterations of the main loop. The predicate of a `ComplexFilter_Box` has already been executed by the `execute` method of the box itself. Therefore, the box attached to the second output of a `ComplexFilter_Box` must not be pushed on the stack.

When adding a box to the stack, it is not checked whether or not the box can be executed. One or more inputs may still not have been set yet. Therefore, a `InputNotSetException` can be be thrown and is caught in line 28. Algorithm A.1 is called on such a box b, assuring (if possible) that all boxes that provide inputs to b that have not yet been processed are executed.

Algorithm A.2 Execute a Query

Input: graph *qGraph*, collection *sourceBoxes*, list *curResult*, Stack *boxesTodo*,
Output: The result of the query as a table (a list of columns)

```
    boxesTodo.addAll(sourceBoxes);
 2: while (!boxesTodo.isEmpty()) do
      box ← boxesTodo.pop();
 4:   if (noChanceToExecute(box)) then
        throw new QueryExecutionException("Missing Inputs");
 6:   end if

      /* Provide necessary information for boxes that execute queries themselves */
      if (box.needsQueryRunner()) then
 8:     box.setQueryRunner(this);
        box.setCurrentResult(result);
10:     box.setCurrentBoxesTodo(boxesTodo);
      end if

12:   try { /* Catch all InputNotSetExceptions: */

      if (box instanceof Input_Box) then
14:     executeInputBox(box);
      else if (box instanceof Output_Box) then
16:     box.execute();
        saveOutput(box);
18:   else
        box.execute;
20:   end if

      pushOutputs(box);

      /* Add out-neighbors to the stack (not the predicates of ComplexFilter_Boxes) */
22:   for all e ∈ getOutEdges(box) do
        if (!(box instanceof ComplexFilter_Box && getOutIndex(e) == 2)) then
24:       remove(boxesTodo, e.getTarget());
          boxesTodo.push(e.getTarget());
26:     end if
      end for

28:   } catch (InputNotSetException inse) {
        boxesTodo ← addInNeighbors(box, boxesTodo);
30:   }

    end while
32: return curResult;
```

3]

A.4 Extensibility of the System

This section is intended to show how easy it is to add new features to the system. Independent of any specific implementation, the following steps are necessary for designing a new operation:

- Specify the type of the operation, i.e. number and type of inputs and outputs.
- Check all preconditions that have to be met by the inputs.
- Implement the application of the operation on the inputs.
- Specify number and type of parameters of the operation.

These items have to be translated into the design of the *QUOGGLES* system, which is straightforward. Since it is a graphical system, two additional steps have to taken care of: First, some means for adding boxes of the new type must be provided to the user. This is done via a clickable icon. Second, the graphical presentation of the box must be adapted to the needs of the operation (i.e. show parameters and an appropriate number of input / output connections ...). The last step, though requiring some time when programming from scratch, is largely automated and therefore fast to implement in reality.

There is no restriction on the types of operations that can be introduced via boxes. The possibilities range from simple boxes that retrieve some special attributes to complex procedures like layout algorithms. The latter example shows that it is also possible to use the query as a data manipulation language. This should be used with caution, though. Adding a box that allows changing attributes extremely extends the power of the system as is described in Section 4.3.

To summarize, there is little to no programming overhead. The steps are described in more detail in the next few pages. Refer to the chapter about implementation details (Chapter A) for a more in-depth explanation of the general design of the *QUOGGLES* system.

The system is very easy to extend. To create a new box, at most four steps are to be done: Create a sub package, create an icon (adds a new box to the query), create the box (implement the semantics) and optionally create a representation (a user defined way of displaying the box).

The next list illustrates are step-by-step instruction on introducing a new box to the system. Since all aspects are covered, the average process will take much fewer steps: A simple box can be built by subclassing `quoggles.boxes.AbstractBox` and `quoggles.icons.AbstractBoxIcon`. Adding it to `boxes.xml` will make it available to the system.

Suppose the box to be created is called “NewBox”.

First, **create a new sub package** in package `quoggles.auxboxes`. It should describe the type of the box, so call the package `quoggles.auxboxes.newbox`.

Icon

1. In this new package, **create a class called** `NewBox_Icon`. It must implement interface `IBoxIcon`, but it is more convenient to extend class `AbstractBoxIcon` (both in package `quoggles.icons`) that provides default implementation for setting the size etc. This icon will be displayed in the top bar of the system and serves as a button for creating new box instances.

2. Override / implement the method `getNewBoxInstance()` to return a new instance of class `NewBox_Box` that will be created in on of the next steps.

3. If superclass `AbstractBoxIcon` was used, the **label string** that will appear on screen for this icon can be set in the constructor:

```
label.setText(‘‘newbox’’);
adjustSize();
```

   The call to method `adjustSize()` assures that the box is large enough to display the new label.

4. Tell the system that there is a new box: Open the file `boxes.xml` in package `quoggles`. **Add a new XML element** "<box>" with subelements as specified in the DTD. The obligatory element "<main>" contains the fully quantified classname of the icon class `NewBox_Icon`:

```
<box>
  <name>Name of the Box</name>
  <main>quoggles.auxboxes.newbox.NewBox_Icon</main>
  <description>Some kind of description here</description>
  <version>0.0.1</version>
</box>
```

Box

1. In the new package, **create a class called** `NewBox_Box`. It must implement interface `IBox`, but it is more convenient to extend class `AbstractBox` (both in package `quoggles.boxes`) that provides default implementations for all methods. This will be the class that does all the work. The next steps assume that the new class extends class `AbstractBox`.

2. Specify the **number and types of inputs and outputs** of the box. This is done by overriding methods `getInputTypes()` and `getOutputTypes()` respectively. They return an array of type int and use constants defined in `quoggles.constants.ITypeConstants`. The number of inputs / outputs can be specified directly be overriding methods `getNumberOfInputs()` and `getNumberOfOutputs()` but this can be deduced by the sizes of the int arrays. *The default implementation assumes one input and output each of the most general type.*

3. The input to the box can be accessed via the field `inputs` (an array of `Objects`). If any type checking and processing of input is wanted, **override method** `setInputs(Object[])`. (Processing of inputs means actions like assigning it to local variables, converting it to special types etc. Any other processing should be done in the `execute()` method).

4. **Method** `execute()` **must be overridden** unless the box should act as the identity operator. This method very probably will use the `Object` array `inputs`, use all `parameters` defined and save the result of the execution into the `Object` array `outputs`.

5. If any user-defined state must be reset prior to the call sequence

   `setInputs(Object[])` → `execute()` → `Object[] getOutputs()`

   the `reset()` and / or `reset(int)` methods should be overridden.

6. If a **user-defined graphical representation** is wanted (as is necessary when using parameters), method `getGraphicalRepresentation()` needs to be overridden. Refer to the respective item later in this list when the use of parameters is treated. Essentially, a class implementing interface `quoggles.representation.IBoxRepresentation` must be returned. This class then implements method `getRepresentation()` that (basically) returns a `JPanel` used to display the box.

   Use of parameters:

7. In the constructor of the box class, **set the protected field** `parameters`, an `org.gravisto.plugin.parameter.Parameter` array, for example like this:

   ```
   parameters = new Parameter[]{ myParam\_1, myParam\_2 };
   ```

   For $i \in \{1, 2\}$, "myParam_i" is an instance of one of the large variety of parameters defined in `org.gravisto.plugin.parameter` and `quoggles.parameters`.

   Attention: *When accessing or changing parameters, it is always wise to access them through the `parameters` array, not through some local variable that might not be up-to-date.*

8. Method `setDefaultParameters()` is called just before a new box instance is displayed the very first time. Instead of overriding this method to specify the default values displayed in this event, it is preferred to set the correct default values right at the creation of the parameters. All parameters provide constructors where a default value can be specified.

9. When using parameters, most boxes need a way to let the user change the value of those. Therefore, the default graphical representation provided by the system is normally not sufficient. Method `getGraphicalRepresentation()` should be overridden and implemented as follows:

   ```
   public IBoxRepresentation getGraphicalRepresentation() {
     if (graphicalRep == null ||
         !(graphicalRep instanceof NewBox_Rep)) {

          graphicalRep = new NewBox_Rep(this);
     }
     return graphicalRep;
   }
   ```

   Attention: *Since this method might be called several times, it should be assured (as the implementation shown does) that a class is only instantiated the first time the method is called.*

   Class `NewBox_Rep` is the conventional name for a class that implements interface `quoggles.representation.IBoxRepresentation`. The next step describes it in more detail.

Representation

1. **Create a class called** `NewBox_Rep` implementing interface `IBoxRepresentation` or extending class `DefaultBoxRepresentation` (both in package `quoggles.representation`). This class will control how the box is displayed. When providing graphical means to change the values of parameters, this class might have to implement Swing interfaces like `java.awt.event.ActionListener` and / or `java.awt.event.FocusListener`.

2. The class can set the variable `graphicalRep` to an instance of `BoxRepresentation`. That in turn is just a subclass of `javax.swing.JPanel`. This code should be put into method `void updateGraphicalRep()`. It is called whenever the graphical representation of the box must be reloaded for some reason.[5]

   The easiest (and regarding the need for a homogeneous look and feel, the best) way to provide a graphical representation is to create a panel `myComponent` containing all user defined elements. This panel is then passed to method

   ```
   \method{BoxRepresentation standardizeBoxRep(javax.swing.JComponent)}
   ```

   The call is shown in the next line:

   ```
   graphicalRep = standardizeBoxRep(myComponent);
   ```

   This default implementation is used in all boxes provided by the current implementation and will fit the needs of most developers.

3. **Override the method** `setParameters(Parameter[], boolean)` and update the values of the user-defined graphical elements representing the parameters. Ignore the boolean parameter to the method, just pass it on to super's implementation. Example:

   ```
   public void setParameters(Parameter[] params, boolean fromBox) {
     super.setParameters(params, fromBox);
     comboBox.setSelectedItem(
       ((OptionParameter)parameters[0]).getValue());
    }
   ```

   If any of the parameters result in a change in the appearance of the box, a call to

   ```
   void updateGraphicalRep()
   ```

   should be used to update the graphical representation.

4. Add any functionality needed to react to user interaction. For example:

[5] The representation must be rebuilt, for instance, if the user changed the value of a parameter and this affects the appearance of the box. Changing the number of inputs / outputs of a `ListOperations_Box` is an example of such a behavior.

```
public void actionPerformed(ActionEvent e) {
  ((OptionParameter)parameters[0]).setValue(comboBox.getSelectedItem());
  updateInputOutput();
}
```

The method `updateInputOutput()` can be used to state that the input and / or output specification has been changed. This change will then be reflected in the respective labels. If the representation has to be rebuilt completely as a consequence of the change, `setParameters(Parameter[], boolean)` should be called which itself contains a call to method `void updateGraphicalRep()`.

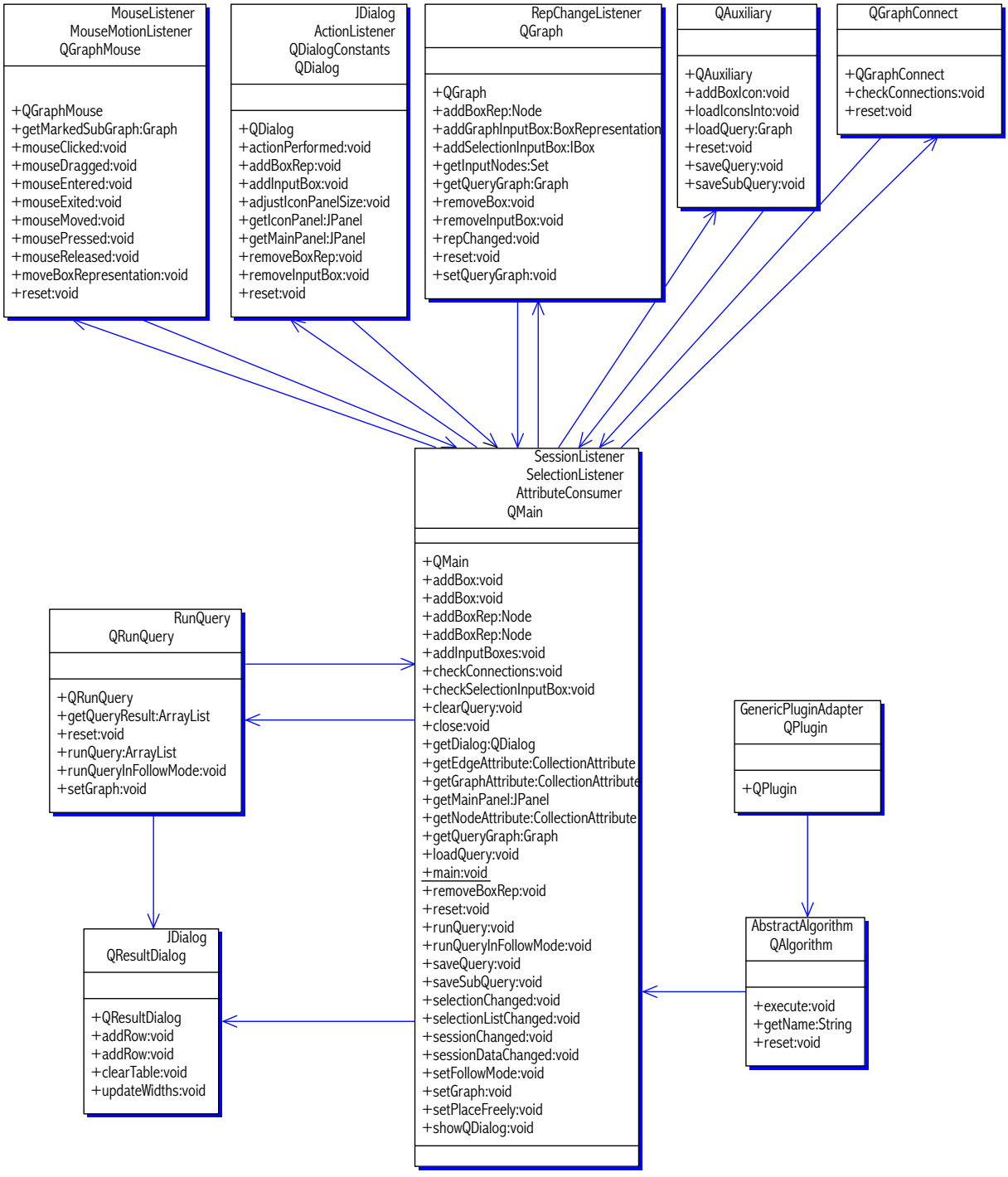


Figure A.4: Core System Architecture

Appendix B

Program Manual

B.1 Starting the System

This chapter gives a brief introduction on how to use the *QUOGGLES* system. As it is primarily designed as a plug-in for the *Gravisto* system, having a working version of it is assumed. Start the plug-in manager and, if *QUOGGLES* is not already contained in the list, use the *Search for plugins ...* or *Add plugin ...* buttons to register it in the system. If the first possibility is chosen, the *QUOGGLES* plug-in description must be located in the class path. For a more detailed description on how to load plug-ins into the *Gravisto* system consult the online documentation at [Pas02].

Once the plug-in has been registered, the menu item *Plugin* should contain an entry *QUOGGLES*. If everything has correctly loaded (the system starts by reading a list of available boxes from an `XML` file called *boxes.xml*), a dialog window should appear.

This window consists of four parts (see the screen shot on page 89). In the lower part, there are a number of buttons with which the user can control the execution and loading / saving of queries. The left part of the dialog is reserved for the (***standard***) ***input box***(es). The one providing all graph elements of the ***current graph*** should be displayed in that area. The top ribbon is occupied by iconic representations of all available boxes. The `Output_Box` will be located to the very left, followed by all basic boxes. Separated by a larger gap the compound boxes like the `AttributeFilter_Box` and `SizeOf_Box` can be found. The last icons (after having scrolled the *icon panel* to the very right) belong to boxes that extend the system beyond a simple query language: `ChangeAttribute_Box` and `Algorithm_Box`. The largest part of the dialog is covered by an empty space. This is called the *main panel*. In this panel, the query will be created.

The *Cancel* button in the bottom left corner closes the window (the current query is lost if it has not been explicitly saved).

B.2 Creating Queries

The icons in the icon panel on top act like buttons. Double-clicking on one of them with the left mouse button adds an instance of this box to the query panel. Boxes will normally automatically snap to the next free input / output.

If the position of a box should be changed, it can easily be done by dragging the box with the mouse. Observe that the box will not move as long as the mouse cursor has not come nearer to

another free input / output than to the current one.

Green and red filled circles (*marks*) indicate whether two boxes (and maybe an additional `Output_Box`) are correctly connected or not.

To remove a box from the main panel, simple double click on it. To remove several boxes in one step, draw a selection rectangle around those boxes and double click on one of the marked ones. This can be done by pressing and holding the left mouse button at some free place in the main panel (i.e. where no box is located). Drag the mouse over the boxes that should be marked and release the button. A rectangle is drawn indicating the area that will be selected. Only boxes that are located entirely inside the selection rectangle are marked.

Individual boxes can be marked by clicking on the box while holding the *shift* or *control* key. Selecting non-connected areas can also be managed by drawing a selection rectangle while holding the *shift* or *control* key.

The parameters of boxes can intuitively be changed using the corresponding components inside the box representations.

The button labelled *Clear Query* can be used to remove all boxes from the main panel.

Sometimes it is useful to move a box representation some distance without it being automatically snapped to the next free input / output. This can be achieved by activating the *free place mode* by clicking on the *Free Place Mode* button located in the bottom right corner. This deactivates the automatic placement of boxes as long as the mode is not deactivated by clicking on the button again.

Several boxes can be moved in one step: Mark all boxes that should be moved, click on one of them and drag the boxes to the desired location. Observe that the automatic placement feature has been switched off in that case.

Clicking anywhere but on a selected box removes an active selection.

Whenever two boxes should be connected that cannot be dragged towards each other, `OneOneConnector_Box`es can be used. There is no explicit icon in the icon panel for this box. Clicking on any red mark (not belonging to a `OneOneConnector_Box`) starts a `OneOneConnector_Box`. If *free place mode* is not switched on, the other end of the connector will probably snap to the next free input / output at once.

B.3 Saving and Loading Queries

The query currently displayed in the main panel (i.e. excluding any ***standard input box***es) can be saved into a `GML` file. Pressing the *Save Query* button (always!) opens a file chooser dialog where the name of the file into which the query should be saved can be entered.

The button *Save Sub-Query* is responsible for saving only the selected part of the query. The second difference to the normal *Save Query* button is that additional information is stored. The purpose of this feature is to save subqueries that will later be loaded into a `SubQuery_Box`. When selecting several boxes, all free inputs and all `Output_Box`es are tagged with $i_1, i_2, \ldots$ and $o_1, o_2, \ldots$ respectively. This information will be saved together with the subquery since it fixes the order in which the inputs and outputs of the `SubQuery_Box` getting this subquery as parameter will appear. The tags are sorted by the vertical (i.e. y-) coordinates of the inputs / outputs.

Principally, it is possible to load a query into a `SubQuery_Box` that has not been saved via the *Save Sub-Query* button. This means, however, that the order of the inputs and outputs is chosen by random and can change when loading the same query several times.

Loading a query using the *Load Query* button opens a file chooser dialog. If a file has been chosen and the contained query graph can successfully be loaded, the current query will be removed from the main panel and the new graph is added.

Saved queries can also be loaded into the *Gravisto* system. Since the names of the boxes and their geometrical properties are saved as normal attributes, the standard *Gravisto* view will display a readable version of it. No functionality is connected to those node, of course.

B.4 Executing Queries

After having created or loaded a query, it can be executed. Clicking on the *Start* button will initiate the corresponding algorithm described in Section A.3.

Only boxes to which a path in the underlying undirected query graph from one of the ***standard input box***es exists are executed. Others will be ignored.

Whenever a query has been successfully executed, another small dialog pops up, showing the ***result table*** of the current query. This is connected to the *Gravisto* editor in the following way: Selecting any cell in the table results in all graph elements that are displayed in this cell (maybe nested within collections of collections) being marked in the *Gravisto* view. The checkbox on top of each column in the ***result table*** can be used to select all cells in one column. Observe that the very moment a cell containing a graph element has been selected, the second ***standard input box*** appears.

B.5 FAQ

The following list answers frequently asked questions:

When scrolling the main panel, the input box(es) do not move? This does not influence the connection between the standard input box and the next box as long as the corresponding marker is green.

The query does not work as expected? Most problems can be solved by looking at the type of some box's output. Some errors occur because the box yields a collection of sets (of outedges, for example) and not a collection (of edges) as maybe expected. try adding some `Output_Box`es at critical places, moving those boxes that cause the problems aside.

A box cannot be (re)moved since other boxes are always marked instead? Sometimes, if there are several boxes close together, the bounding box of some boxes prevent that smaller ones can be marked. The solution is to temporarily move the neighboring boxes aside.

How can the input box at the left hand side of the window be removed? The ***standard input box*** can neither be removed nor replaced.

Sometimes, a second input box appears below the standard input box? The implementation features a second ***standard input box*** that produces the set of graph elements selected in the ***current graph***. This box appears only if there is an active selection in the corresponding *Gravisto* view that displays the ***current graph***. After the execution of a query, the *QUOGGLES* system removes any selection to enable the user to select the desired parts in the ***result table***.

The space in the main panel does not suffice for the query? There should be enough room for any human generated query in the main panel! If the query reaches the top of the window, try dragging the whole query some distance down and connect it to the input box with a `OneOneConnector_Box`. If that does not help, consider moving some part of the query into a `SubQuery_Box`.

When creating a `OneOneConnector_Box`, it suddenly disappears? Try drawing it from left to right.

How to start a connector at an output having already an `Output_Box`? Currently, the only way is to temporarily move away the `Output_Box`, drag the `OneOneConnector_Box` and move `Output_Box` back.

Can a query be saved into different formats? Currently, only an extended `GML` format is supported.

Can the contents of a saved subquery be added to the current query? Currently, no. Except for loading the subquery into a `SubQuery_Box`.

Despite of a large query having been executed, the result table is empty? There are three possibilities for this: First, it might be that there are no `Output_Box`es present in the query. Only data marked by such a box is displayed in the ***result table***. Second, all `Output_Box`es are located at boxes that are not executed. Remember that boxes not reachable from any ***standard input box*** and not necessary for the execution of other boxes are not executed at all. And third, it might be that all present `Output_Box`es have empty lists as the data they present in the ***result table***.

Executing the same query several times yields different results? If boxes like the `ChangeAttribute_Box` that can initiate changes in the ***current graph***, subsequent execution of queries will indeed probably yield different results. Another possibility is that the second ***standard input box*** has been used and the selection in the ***current graph*** has changed since the last execution of the query.

After having written a new box, why is it not available? Refer to Section A.4. The new box must be entered into the box description file *boxes.xml*.

Bibliography

[ABa03] Attwood, T. K. ; Bradley, P. ; Flower et al., D. R.: PRINTS and its Automatic Supplement, prePRINTS. In: *Nucleic Acids Research* 31 (2003), Nr. 1, S. 400–402

[Abi97] Abiteboul, S.: Querying Semi-Structured Data. In: *Lecture Notes in Computer Science* 1186 (1997), January, S. 1–18

[ABS00] Abiteboul, S. ; Buneman, P. ; Suciu, D.: From Relations to Semi-Structured Data and XML. In: *Data on the Web* (2000)

[AHU87] Aho, A. V. ; Hopcroft, J. E. ; Ullman, J. D.: *Data Structures and Algorithms.* Reading, Massachusetts : Addison-Wesley Publishing Company, 1987

[AM98] Arocena, G. ; Mendelzon, A.: WebOQL: Restructuring Documents, Databases and Webs. In: *IEEE ICDE* (1998)

[AMO93] Ahuja, R. K. ; Magnanti, T. L. ; Orlin, J. B.: *Network Flows: Theory, Algorithms, and Applications.* New Jersey : Prentice Hall, Inc., 1993

[AQa97] Abiteboul, S. ; Quass, D. ; McHugh et al., J.: The Lorel Query Language for Semi-Structured Data. In: *International Journal on Digital Libraries (IJDL)* 1 (1997), April, Nr. 1, S. 68–88

[AU79] Aho, A. V. ; Ullman, J. D.: *Universality of Data Retrieval Languages.* Sixth ACM Symposium on Principles of Programming Languages (POPL79, proceedings), 1979. – 110–117 S

[Bal95] Balkir, N. H.: *VISUAL.* Cleveland, Comp. Eng. and Sci. Dept., Case Western Reserve University, Diplomarbeit, 1995

[BCCL91] Batini, C. ; Catarci, T. ; Costabile, M. F. ; Levialdi, S.: Visual Query Systems. In: *Technical Report, Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza"* No.04.91 (1991)

[BDFS97] Buneman, P. ; Davidson, S. B. ; Fernandez, M. ; Suciu, D.: Adding Structure to Unstructured Data. In: *Database Theory-ICDT'97, 6th International Conference* (1997), January, S. 336–350

[Bro98] Broy, M.: *Grundlagen der Informatk 2.* 2. Auflage. New York, Berlin, etc. : Springer-Verlag, 1998

[BRS90] Biskup, J. ; Räsch, U. ; Stiefeling, H.: An Extension of SQL for Querying Graph Relations. In: *Computer Languages* 15 (1990), Nr. 2, S. 65–82

[CG85] Ceri, S. ; Gottlob, G.: Translating SQL Into Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. In: *IEEE Transactions on Software Engineering* 11 (1985), April, Nr. 4

[CH94] Clark, J. ; Holton, D. A.: *Graphentheorie. Grundlagen und Anwendungen.* Heidelberg, Berlin : Spektrum Akademischer Verlag, 1994

[Cha76] Chamberlin, D. D.: SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control. In: *IBM Journal of Research and Development* 20 (1976), November, Nr. 6, S. 560–575

[Cod72] Codd, E. F. ; Rustin, R. (Hrsg.): *Relational Completeness of Data Base Sublanguages.* New Jersey : Prentice Hall, Inc., 1972

[Com03] Committee, DAML J. *DAML Query Language (DQL).* `http://www.daml.org/dql`. April 2003

[CSSK94] Chaudhuri ; Surajit ; Shim ; Kyuseok: Including Group-By in Query Optimization. In: *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), S. 354–366

[CSSK95] Chaudhuri ; Surajit ; Shim ; Kyuseok: An Overview of Cost-based Optimization of Queries with Aggregates. In: *Bulletin of the Technical Committee on Data Engineering (IEEE)* 18 (1995), September, Nr. 3, S. 3–9

[DB03] D. Becket, W3C. *Resource Description Framework (RDF).* `http://www.w3.org/TR/rdf-primer/`. 2003

[DD93] Date, C. J. ; Darwen, H.: *A Guide to The SQL Standard.* Third Edition. Reading, Massachusetts : Addison-Wesley Publishing Company, 1993

[DM94] Durbin, R. ; Mieg, J. T.: The ACEDB Genome Database. In: *Computational Methods in Genome Research* (1994), S. 45–56

[FMST01] Fernandez, M. F. ; Morishama, A. ; Suciu, D. ; Tan, W-C.: *Publishing Relational Data in XML: The SilkRoute Approach.* Proc. 9th International World Wide Web Conference, 2001

[FTS00] Fernandez, M. F. ; Tan, W-C. ; Suciu, D.: *SilkRoute: Trading between Relations and XML.* Proc. 9th International World Wide Web Conference, 2000

[GBP94] Gyssens, M. ; Bussche, V. D. ; Paradaens, J.: A Graph-Oriented Object Database Model. In: *IEEE TKDE* (1994)

[GPG90] Gyssens, M. ; Paradaens, J. ; Gucht, D. V.: A Graph-Oriented Object Database Model. In: *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (1990), S. 417–424

[Gro03] The Bioinformatics Group. *The PRINTS Fingerprint Database.* `http://www.bioinf.man.ac.uk/dbbrowser/PRINTS`. 2003

[GS02] Giugno, R. ; Shasha, D.: GraphGrep: A Fast and Universal Method for Querying Graphs. In: *Proceeding of the International Conference in Pattern recognition (ICPR)* (2002), August

[Güt91] Güting, R.: Extending a Spatial Database System by Graphs and Object Class Hierarchies. In: *Proceedings of the International Workshop on Database Management Systems for Geographical Applications* (1991), May

[Gut94] Guting, R.: GraphDB: Modelling and Querying Graphs in Databases. In: *VLDB* (1994), S. 297–308

[HA94] Hopcroft, J. E. ; Aho, A. V.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Reading, Massachusetts : Addison-Wesley Publishing Company, 1994

[JK84] Jarke, M. ; Koch, J.: Query Optimization in Database Systems. In: *ACM Computing Surveys* 16 (1984), June, Nr. 2, S. 111–152

[Jun99] Jungnickel, D.: *Graphs, Networks and Algorithms.* New York, Berlin, etc. : Springer-Verlag, 1999

[Kar03] Karvounaraki, G. *RQL: The RDF Query Language.* `http://139.91.183.30:9090/RDF/RQL/Design.html`. 2003

[KE01] Kemper, A. ; Eickler, A.: *Datenbanksysteme – Eine Einführung.* 4., überarbeitete und erweiterte Auflage. München, Wien : R. Oldenbourg Verlag, 2001

[MB99] Messmer, B. T. ; Bunke, H.: A Decision Tree Approach to Graph and Subgraph Isomorphism Detection. In: *Pattern Recognition* 32 (1999), S. 1979–1998

[MC95] Massari, A. ; Chrysanthis, P. K.: Visual Query of Completely Encapsulated Objects. In: *Proceedings, Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management* (1995), March, S. 18–25

[MC99] Mandel, L. ; Cengarle, M. V.: On the Expressive Power of the Object Constraint Language. (1999), February

[Meh84] Mehlhorn, K.: *Data Structures and Algorithms.* Bd. 1/2. New York, Berlin, etc. : Springer-Verlag, 1984

[Mel96] Melton, J.: An SQL3 Snapshot. In: *Proceedings of the IEEE Twelfth International Conference on Data Engineering* (1996), S. 666–672

[MFK+00] Manulescu, I. ; Florescu, D. ; Kossmann, D. ; Xhumari, F. ; Olteanu, D.: *Agora: Living with XML and Relational.* Proc. 26th VLDB Conference, 2000

[MH95] M. Himsolt, University of Passau. *Graph Modelling Language.* `http://infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html`. 1995

[Mic03] Microsoft Corporation. *Encarta 2003.* `http://Encarta.msn.de`. 2003

[Mil02] Miller, L. *RDF Squish Query Language and Java Implementation.* `http://ilrt.org/discovery/2001/02/squish/`. 2002

[MMM96] Mendelzon, AM. O. ; Mihaila, G. A. ; Milo, T.: Querying the World Wide Web. In: *Proceedings of PDIS'96* (1996)

[MN99] Mehlhorn, K. ; Näher, S. *The LEDA Platform of Combinatorial and Geometric Computing*. `http://www.algorithmic-solutions.info/enleda.html`. 1999

[MR93] Mumick, I. S. ; Ross, K. A.: Noodle: A Language for Declarative Querying in an ObjectOriented Database. In: *Proc. DOOD Conference* (1993)

[OMMA01] Shen Orr, S. S. ; Milo, R. ; Mangan, S. ; Alon, U.: Network Motifs in the Transcriptional Regulation Network of Escherichia Coli. In: *Nature Genetics, Advanced Online Publication* (2001)

[ÖMÖ89] Özsoyoğlu, G. ; Matos, V. ; Özsoyoğlu, Z. M.: Query Processing Techniques in the Summary-Table-By-Example Database Query Language. In: *ACM TODS* 14 (1989), December, Nr. 4

[ÖW89] Özsoyoğlu, G. ; Wang, H.: A Relational Calculus with Set Operators, its Safety and Equivalent Graphical Languages. In: *IEEE Software Eng.* 15 (1989), September, Nr. 9

[ÖW93] Özsoyoğlu, G. ; Wang, H.: ExampleBased Graphical Database Query Languages. In: *IEEE Computer* (1993), May, S. 25–38

[OW02] Ottmann, T. ; Widmayer, P.: *Algorithmen und Datenstrukturen*. 4. Auflage. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002

[Pas90] University of Passau, Theoretical Computer Science Group. *Graphlet*. `http://www.infosun.fmi.uni-passau.de/Graphlet`. 1990

[Pas02] University of Passau, Theoretical Computer Science Group. *Gravisto, Graph Visualization Toolkit*. `http://www.gravisto.org`. 2002

[Pro03] Student Project: *"Computational Genomics of Prokaryotes": A Graph Query Language for Biological Networks*. `http://www.rug.nl/informatica/onderzoek/programmas/svcg/`. 2003. – Ongoing Student Project at the Rijksuniversiteit Groningen, Netherlands

[PS85] Preparata, F. P. ; Shamos, M. I.: *Computational Geometry: An Introduction*. New York, Berlin, etc. : Springer-Verlag, 1985

[Sal78] Salomaa, A.: *Formale Sprachen*. New York, Berlin, etc. : Springer-Verlag, 1978

[SBH+00] Sheng, L. T. ; Balkir, L. ; Al Hamdani, N. H. ; Özsoyoğlu, Z. M. ; Özsoyoğlu, G.: Query Processing Techniques for Multimedia Presentations. In: *Journal of Multimedia Tools and Applications* (2000), February

[Sch97] Schöning, U.: *Theoretische Informatik - kurzgefaßt*. Third Edition. Spektrum Akademischer Verlag, 1997

[Sed88] Sedgewick, R.: *Algorithms*. 2. Auflage. Reading, Massachusetts : Addison-Wesley Publishing Company, 1988

[SM02] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition, v1.4.0 API Specification.* `http://java.sun.com/j2se/1.4/docs/api/index.html`. Oktober 2002

[SÖÖ99] Sheng, L. ; Özsoyoğlu, Z. M. ; Özsoyoğlu, G.: A Graph Query Language and Its Query Processing. In: *IEEE, Trans. On Knowledge and Data Engineering 2* (1999)

[Süß01] Süss, C.: *An Approach to the Model-Based Fragmentation and Relational Storage of XML-Documents.* 13. GI-Workshop Grundlagen von Datenbanken, 2001

[SWG] Shasha, D. ; Wang, J. T. ; Guigno, R.: Algorithmics and Applications of Tree and Graphs Searching.

[SZF01] Süss, C. ; Zukowsky, U. ; Freitag, B.: *Data Modelling and Relational Storage of XML-based Teachware.* Vienna, Austria : Proceedings Informatik, 2001

[Tom89] Tompa, F.: A Data Model for Flexible Hypertext Database Systems. In: *ACM Transactions on Information Systems* 7 (1989), July, Nr. 1, S. 85–100

[Ull82] Ullman, J. D.: *Principles of Database Systems.* Rockville, Maryland : Computer Science Press, 1982

[W3C03] W3C. *XQuery: A W3C Recommendation.* `http://www.w3.org/XML/Query`. 2003

[YN99] Yates, R. B. ; Neto, B. R.: *Modern Information Retrieval.* Reading, Massachusetts : Addison-Wesley Publishing Company, 1999

[Zlo75] Zloof, M.: Query By Example. In: *AFIPS Conference Proceedings* 44 (1975), S. 431–432

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, dass ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 12. Januar 2004 .....

Paul Holleis