University of Groningen **Faculty of Mathematics and Natural Sciences**

Department of
Mathematics and
Computing Science

A
Graph Query Language

Edwin Dijkshoorn

August 2004

Supervisors

Prof. Dr. J.B.T.M. Roerdink.

Drs. D.W.J.Bosman

Prof. dr. ir. J.Bosch

RuG

Abstract:

Gene regulatory networks can be visualised by graphs, where nodes represent genes and edges represent interactions between genes.

This approach can aid in the analysis of large gene regulatory networks. A useful tool in analysing gene regulatory network represented as a graph is the ability to select elements of the network. For this purpose, a graph query language was developed.

A graph query language (GQL) is a tool that can assist the user to search within a graph for a certain sub-structure of that graph. In our case we also wanted to be able to search for properties of the nodes and edges, which is more like what is supported by a conventional database query language (filtering based on the properties of nodes and edges).

A query language and search algorithm has been developed which is able to find structures and properties within a large sparse graph.

Acknowledgments

Particular thanks go to Jos Roerdink, who put a large amount of energy and time into reading and analyzing this thesis and helped me to get it done in time.

And, of course, the contributions of Dinne Bosman, Evert-Jan Blom and Patrick Ogao are very much appreciated.

To my wife Gitta and to all the friends and supporters unmentioned but by no means unremembered, I give my most heartfelt thanks.

Contents

List of figures

List of examples

Chapter 1: Introduction

Genes, the basic units of heredity, are found in the cells of all living organisms, from bacteria to humans. They determine the physical characteristics an organism inherits and are composed of segments of deoxyribonucleic acid (DNA).

The information encoded within a gene directs the production of proteins. These are the compounds that are essential for the functioning of an organism. Experimental advances like DNA-microarrays provide a wealth of data that can be used to identify and visualize the underlying regulatory networks.

A gene regulatory network (also called a *GRN* or *genetic regulatory network*) is a collection of DNA segments and proteins in a cell which interact with each other and with other substances in the cell, thereby governing the rates at which genes in the network are transcribed into mRNA, which is the first step towards the creation of a protein.

Even in the simplest known organisms, i.e. bacteria, gene regulatory networks are only starting to be elucidated and existing knowledge still is very fragmented. However, for the two most fully characterized bacterial species *(E. coli*[1] *and B. subtilis*[2]*)* regulatory circuits are being unravelled at a fast pace, especially through analysis of large mutant collections by DNA-microarrays.

Dynamical modelling of those networks is becoming increasingly widespread, as people attempt to understand biological phenomena in their full complexity and make sense of the huge amount of experimental data[a].

[1] ***Escherichia coli*** (usually abbreviated to *E. coli*) is one of the main species of bacteria that live in the lower intestines of warm-blooded animals (including birds and mammals) and are necessary for the proper digestion of food. Its presence in groundwater is a common indicator of fecal contamination. ("Enteric" is the adjective that describes organisms that live in the intestines. "Fecal" is the adjective for organisms that live in feces, so it is often a synonym for "enteric.") The name comes from its discoverer, Theodor Escherich. It belongs among the Enterobacteriaceae, and is commonly used as a model organism for the bacteria in general.[c]

[2] ***Bacillus subtilis*** is a gram-positive, rod-shaped, aerobic bacterium that is commonly found in soil. An important property of *Bacillus subtilis* is its ability to form a tough, protective endospore, which allows it tolerate extreme environmental conditions. [c]

The research group *Scientific Visualisation and Computer Graphics* of the Institute *for Mathematics and Computing Science* is working with the Molecular Genetics group in developing a system that can comprehensively visualise a gene regulatory network. More details on this project are given in Chapter 2.

Understanding gene interactions is important for genetic research on organisms, *e.g. how does changing one gene affect the amount of protein another gene transcribes.*

One tool that can help to make sense of a large amount of visual data is selecting particular data with a query, *e.g. all genes interacting with protein C.* Since a gene regulatory network can be represented by a graph[1], the project partners wanted to use a graph query language for this purpose.

A graph query language (GQL) is a tool that can search within a graph for a certain sub-structure of that graph.

In our case we also wanted to be able to search for properties of the nodes and edges as well, which is more like a conventional database query language[2], (filtering on the properties of nodes and edges).

The first few weeks of designing and building the GQL were under the assumption that it would only be a small feature in the main program, to be constructed in the larger project. Therefore the fast implementation route with rapid prototyping was chosen. While in the writing stage, we realized the value of an accurate GQL function in a graph viewing application which deals with the massive amount of data that we use.

---

[1] a **graph** is a generalization of the simple concept of a set of points, called vertices or nodes, connected by links, also called edges or arcs. Depending on the applications, edges may or may not have a direction; edges joining a vertex to itself may or may not be allowed, and vertices and/or edges may be assigned labels. A numeric label is often called a "weight". If the edges have a direction associated with them (indicated by an arrow in the graphical representation) we have a **directed graph**. This means that it is possible to follow a path from one vertex to another, but not in the opposite direction. If there are no directed edges, the graph is an **undirected graph**. There may be more than one edge between two vertices (directed or undirected), a case which is known as a **multigraph**. [c]

[2] Database query languages are like programming languages. The person formulating the query is expected to understand the relevant rules for formulating the query, and to program the query according to the requirements. [c]

With this realization it was needed to make the GQL a more robust en fast feature with concerns for memory usage and performance.

Because of this we decided to treat the first partly finished (and working) version as a 'proof of concept' and we began designing and building a second more architecturally sound version.

Chapter 2: Problem description and problem domain

2.1 Problem domain

The GQL is part of a project called *Developing an Automated Gene Network Identification, Modelling, Visualization & Simulation System*, which in itself is a component of a larger research program on *Computational Genomics of Prokaryotes*, funded by the Netherlands Organization for Scientific Research (NWO), Program on Biomolecular Informatics (BMI).

This program aims to reconstruct the cellular processes, metabolic potential (metabolome) and gene regulatory networks of selected gram-positive Bacteria and Archaea[1], by *in silico* analysis of all proteins encoded by their chromosome. "Virtual cell" databases will be generated, consisting of modules that include the majority of genes and predicted encoded proteins, signalling and information pathways, transport systems, regulatory networks, metabolic routes, etc. *In silico* comparative genomics, simulation and visualization will provide a detailed picture of the distribution of the overall gene-pool among these organisms, the architecture of the metabolome and an inventory of both shared and unique genes and encoded properties of each species. This should lead to important advances in the understanding of prokaryote evolution, and will contribute to the prediction of their metabolic functions.

There are partners to the projects. These are:

1. Centre for Molecular and Bio-molecular Informatics (CMBI), University of Nijmegen (computational genomics, tool development)

2. Wageningen University, Laboratory of Microbiology (Archaea, control of gene expression, evolution)

3. University of Groningen, Molecular Genetics group (comparative genomics lactic acid bacteria and bacilli, regulatory networks, transcriptome analysis)

[1] The **Archaea** are one of the three major groups of living organisms, together with bacteria and eukaryota. They are prokaryotes, like bacteria, and were originally included among them. Their separate identity was discovered in the late 1970s by Dr. Carl Woese at the University of Illinois by genetic comparison. Originally they were termed the Archaebacteria, and the other prokaryotes the Eubacteria, but now there is a growing tendency to restrict the term bacteria to the latter and the names have adjusted accordingly. [c]

4. University of Groningen, Institute for Mathematics and Computing Science (gene regulatory network identification, dynamical system modelling, simulation, visualization).

The project undertaken by the two RuG research groups focuses on representing and simulating a gene regulatory network. The following paragraphs provide some background information on basic biological processes and techniques used to create models of gene regulatory networks.

2.1.1 DNA

Pieces of DNA are not single molecules. Rather, they are pairs of molecules, which entwine like vines to form a **double helix**.

Each number of a pair is a strand of DNA: a chemically linked chain of nucleotides, each of which consists of a sugar, a phosphate and one of four kinds of aromatic "bases". Because DNA strands are composed of these nucleotide subunits, they are polymers.

The diversity of the bases means that there are four kinds of nucleotides, which are commonly referred to by the identity of their bases. These are adenine (A), thymine (T), cytosine (C), and guanine (G).

In a DNA double helix, two polynucleotide strands come together through complementary pairing of the bases, which occurs by hydrogen bonding. Each base forms hydrogen bonds readily to only one other -- A to T and C to G -- so that the identity of the base on one strand dictates what base must face it on the opposing strand. Thus the entire nucleotide sequence of each strand is complementary to that of the other, and when separated, each may act as a template with which to replicate the other (middle and lower half of the illustration at the right).

Because pairing causes the nucleotide bases to face the helical axis, the sugar and phosphate groups of the nucleotides run along the outside and the two chains they form are sometimes called the "**backbones**" of the helix. In fact, it is chemical bonds between the phosphates and the sugars that link one nucleotide to the next in the DNA strand.

When an interesting piece of DNA has been isolated or identified, scientists often need to determine if the sequence of nucleotides in the fragment is related to known genes and to determine what kind of protein it might produce. The technology of determining the exact order of the building blocks is called *sequencing*. Since the late seventies sequencing

projects were started. The total output of those projects grew exponentially.

2.1.2 What is a Gene regulatory network?

Genes can be viewed as nodes in a network, with input being proteins such as transcription factors, and outputs being the level of gene expression. The node itself can also be viewed as a function which can be obtained by combining basic functions upon the inputs. These functions have been interpreted as performing a kind of information processing within cells which determine cellular behaviour. The basic drivers within cells are levels of some proteins, which determine both spatial (tissue related) and temporal (developmental stage) co-ordinates of the cell, as a kind of "cellular memory". The gene networks are only beginning to be understood, and it is a next step for biology to attempt to deduce the functions for each gene "node", to assist in building models of the behaviour of a cell.

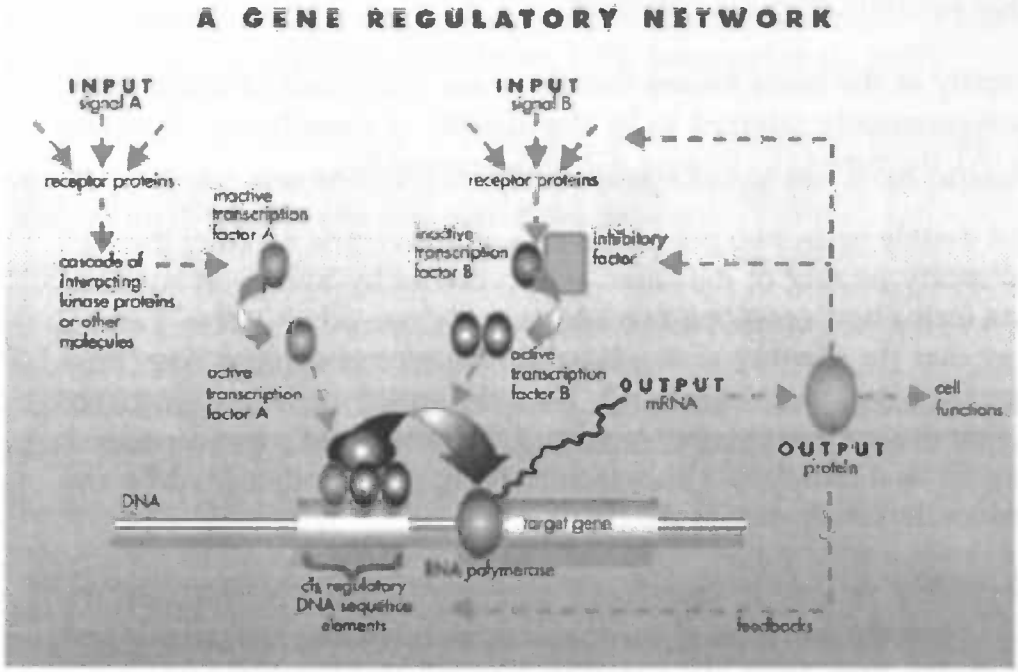


Figure 2-1: A gene regulatory network.

Scientist try to construct these models by doing experiments on bacteria, one of the most common experiments use DNA microarrays.

2.1.3 DNA microarrays[c]

A **DNA microarray** (also **DNA chip** or *gene chip* in common speech) is a piece of glass or plastic on which single-stranded pieces of DNA have been affixed in a microscopic array.

Machines use such chips to screen a biological sample for the presence of many genetic sequences at once. The affixed DNA segments are known as *probes*. Hundreds of identical probes are affixed at each point in the array to make the chips effective detectors.

Typically arrays are used to detect the presence of mRNAs that may have been transcribed from different genes and which encode different proteins. The RNA is extracted from many cells of a single type, then converted to cDNA and "amplified" in concentration by rtPCR. Fluorescent tags are chemically attached to the strands of DNA. A cDNA molecule that contains a sequence complementary to one of the single-stranded probe sequences will stick via *base pairing* to the spot at which the complementary probes are affixed. The spot will then fluoresce (or glow) when examined.

The glow indicates that cells in the sample had recently transcribed a gene that contained the probed sequence ("recently," because cells tend to degrade RNAs soon after transcribing them). The intensity of the glow depends on how many copies of a particular mRNA were present and thus roughly indicates the *activity* or *expression level* of that gene. So arrays in a sense paint a picture or "profile" of which genes in the genome are active in a particular cell type and under a particular condition.

Because most proteins remain of unknown function, and because many genes are active all the time in all kinds of cells, researchers usually use microarrays to make close comparisons. For example, an RNA sample from brain tumour cells might be compared to a sample from healthy neurons or glia. Probes that bind RNA in the tumour sample but not in the healthy one indicate genes that are uniquely associated with the disease. Typically in such a test, the two sample's cDNAs are tagged with two distinct colours, enabling comparison on a single chip. Researchers hope to find molecules that could be therapeutically targeted with drugs among the various proteins encoded by disease-associated genes.

Although the chips detect RNAs and not proteins, many scientists refer to these kinds of analysis as "expression analysis" or expression profiling. Since there are hundreds of thousands of distinct probes on an array, each can accomplish the equivalent of thousands of genetic tests in parallel.

Arrays have therefore dramatically accelerated many types of investigations including the understanding of gene regulatory networks.

2.1.4 Simulation

In addition to the experimental tools like DNA microarrays, new methods for modelling and simulation of gene regulatory networks are essential. When supported by intuitive methods and computer tools, modelling and simulation methods allow large and complex genetic regulatory systems to be analyzed. Based on knowledge of regulatory mechanisms and available expression data a model is constructed. The behaviour of the system can then be stimulated for a variety of experimental conditions. Based on the outcome of the comparison of the predictions and the observed behaviour an indication of the adequacy of the model can be given. If they don't match, and the experimental data is considered reliable, the model should be revised.

Computer simulation of genomic networks started more than three decades ago with simple, Boolean networks[1] to model interactions between genes. Though crude, these models capture a number of essential properties of real genomes (Kauffman, 1974; Somogyi et al., 1997).

In addition of simulating a complex gene regulatory network it is necessary to be able to interpret the results. One of the most promising ways to do this is the visualisation of the network.

2.1.5 Visualizing gene regulatory networks

One advantage of visualization is that a vast amount of information can be easily and rapidly interpreted. Visualization also enables one to perceive

---

[1] The following example illustrates how a Boolean network can model a GRN together with its gene products (the outputs) and the substances from the environment that affect it (the inputs). Stuart Kauffman was amongst the first biologists to use the metaphor of Boolean networks to model genetic regulatory networks.

1. Each gene, each input, and each output is represented by a node in a directed graph in which there is an arrow from one node to another if and only if there is a causal link between the two nodes.
2. Each node in the graph can be in one of two states: on or off.
3. For a gene, "on" corresponds to the gene being expressed; for inputs and outputs, "on" corresponds to the substance being present.
4. Time is viewed as proceeding in discrete steps. At each step, the new state of a node is a boolean function of the prior states of the nodes with arrows pointing towards it.

emergent and/or unanticipated properties that could have otherwise gone unseen, in what may be termed a visual discovery process by also facilitating the formation of hypotheses. By this process, any inherent problem or hidden pattern becomes immediately apparent.

One technique that can be used to visually represent gene regulatory networks is that of graph visualization. Graphs play a crucial role in conveying relationships between entities within a defined unifying context.

The program currently under development by the two RuG research groups aims to satisfy the need for such a visualisation.

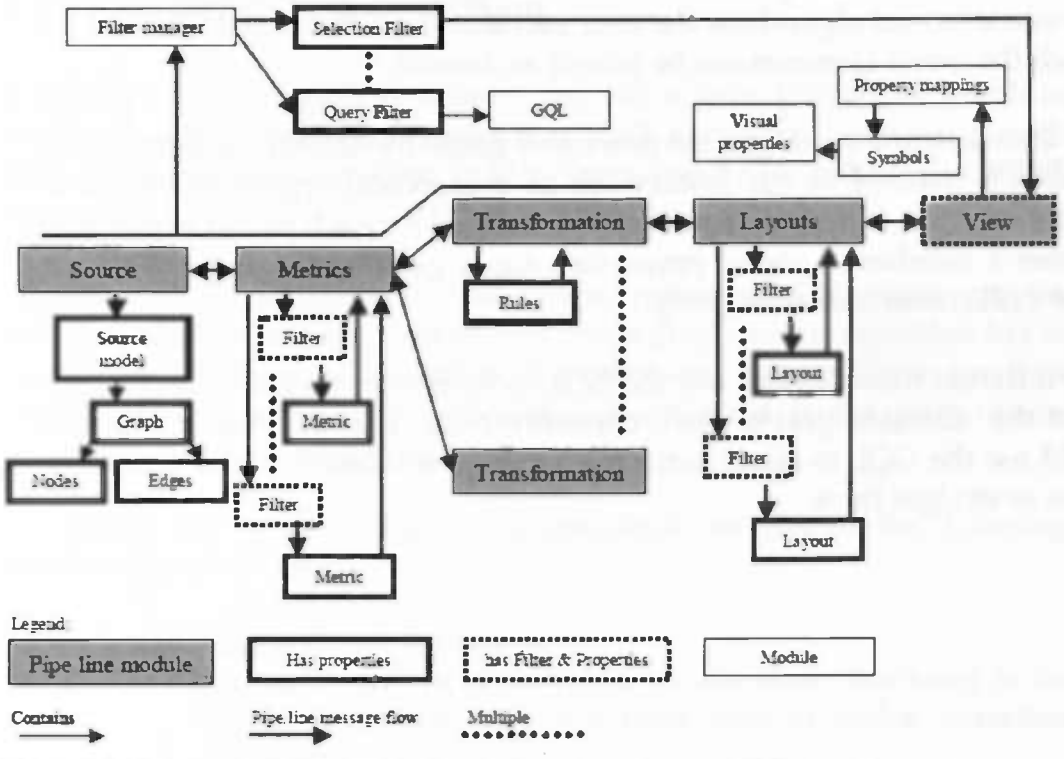


Figure 2-2: A flowchart showing the pipeline-based architecture of the main program.

2.1.6 Workings of the main program

The main program is a pipeline based program consisting of several modules.

The source module handles the consistency between the data source (file or database) and the source graph. For the database source all the entities contained in a biological network are retrieved by queries. From the query result a graph is constructed which is used for further processing.

The metrics module adds information to the nodes and edges of the graph based on structural information. Each type of information (e.g. gene name, confidence-value, expression profile, etc.) is added as a property to a graph element during calculation phase.

The transformation module takes a graph, containing nodes and edges modelled according to a certain data model as its input. The output is another graph, but now containing graph elements modelled in another data model.

The layouts module contains a number of layout filters. Each layout filter extracts a part of the input graph. The layout filter is then coupled to a layout module that positions the nodes and edges. In addition to automatic layout algorithms the user can also choose a manual layout in which the graph elements can be placed as desired.

The view module visualizes the processed graph by displaying symbols. A symbol is defined in our framework as a graphical representation of a model object. A different symbol can be defined for each model object and defines a number of visual properties, e.g. a gene rectangle symbol can have a fill colour visual property.

The different modules can use the GQL to select certain nodes and edges from the different graphs they currently hold. E.g. the metric module could use the GQL to select paths from one gene to another and represent these as straight lines.

2.2 Problem description

We want to be able to find sub-structures within a large graph representing a gene regulatory network. We also want to have control over what kind of nodes and edges are allowed or required within these sub-structures. To do this we need to be able to identify properties of the nodes and of the edges and filter on these properties.

As a solution to these requirements we have build the graph query language E-GLIDE and its corresponding graph-searching algorithm.

2.2.1 Requirements specification

A simple organism such as a bacterium has a gene interaction graph in excess of 4000 nodes. This means that if all nodes had interaction with each other (a fully connected graph) there would be $4000! = 1.8*10^{12673}$ edges.

Luckily gene regulatory networks can be represented as sparse graphs. Still this amounts to a large set of data that a graph query algorithm has to search for the right combination of nodes and edges. There for it is important to create a search method that is optimized, so the query can be done within a reasonable time.

To specify the work needed to be completed we created the following requirements specification.

- **Searching for graph structures.**
  Structures are the main thing we are after. We want to be able to find interesting structures such as cycles, branches and paths.

- **Testing of node and edge attributes.**
  Attributes are the values and properties of a node or edge. These include values such as unique node identifier and node value, and properties like number of incoming edges.
  This feature is closely related to the Jython integration (see below). It must be possible to 'ask' for specific node or edge attributes. Instead of giving static attribute requirements, the GQL should provide code hooks to test attribute requirements. Later on these code hooks can then be coupled to Jython.

- **Searching for multiple repetitions of structures.**
  Since it is very cumbersome to write large queries, our language must be able to accommodate for repetition of sub-structures.

- **Possibility for negation of structures.**
  Sometimes it is useful to disallow some graph-structure to be part of your answer. E.g. a path must be possible between node A and node B, but this path may not be part of a cycle. For this we want to use the negation.

- **The graph data structure should not be modified.**
  The graph we want to use our GQL on can be part of a temporary answer, which the user of the program will want to use in further investigation of the gene regulatory network. So we need to preserve the graph while searching it.

- **Possibility for Jython integration.**
  Jython is an implementation of the high-level, dynamic, object-oriented language Python integrated within the Java platform. It allows users to write simple or complex scripts that add functionality to applications.

- **Preferred user interaction.**
  In research done in advance of the GQL project a language known Glide was found. Glide is a language which allows the users to specify graph queries in a regular expression-like syntax.
  Since Glide solves a number of syntactic problems such as defining cycles and edges in a compact en transparent manner, we decided to use as much as possible from this language.
  Glide is writhen by Prof. Dennis Shasha and Rosalba Giugno.
  See section 3.2.1 on page 21, and Appendix C: Glide Language.

Chapter 3: Design and Implementation

3.1 Overall design: The big picture

We decided to use a pipeline based design for implementing the GQL, in part because the main program designed to visualise and analyse gene regulatory networks is also pipeline based, and because a pipeline is easily testable. The first part of the pipe is the query. The query is parsed and the resulting tree-structure is passed on to the graph query engine (GQE) which returns ether a tree that represents all matching structures, or the collection of nodes and edges contained in that tree.

The GQE has access to the target graph (this is the graph we want to query) and a collection of rules. Rules are integer-labelled restrictions to nodes and edges which can be defined by the user.

The GQL is designed work independently of the rest of the program. One of the classes, the Graph Interaction Module (GIM), is designed to serve as a handle to the target graph. By changing this class it is possible to integrate the GQL in any program using graphs.

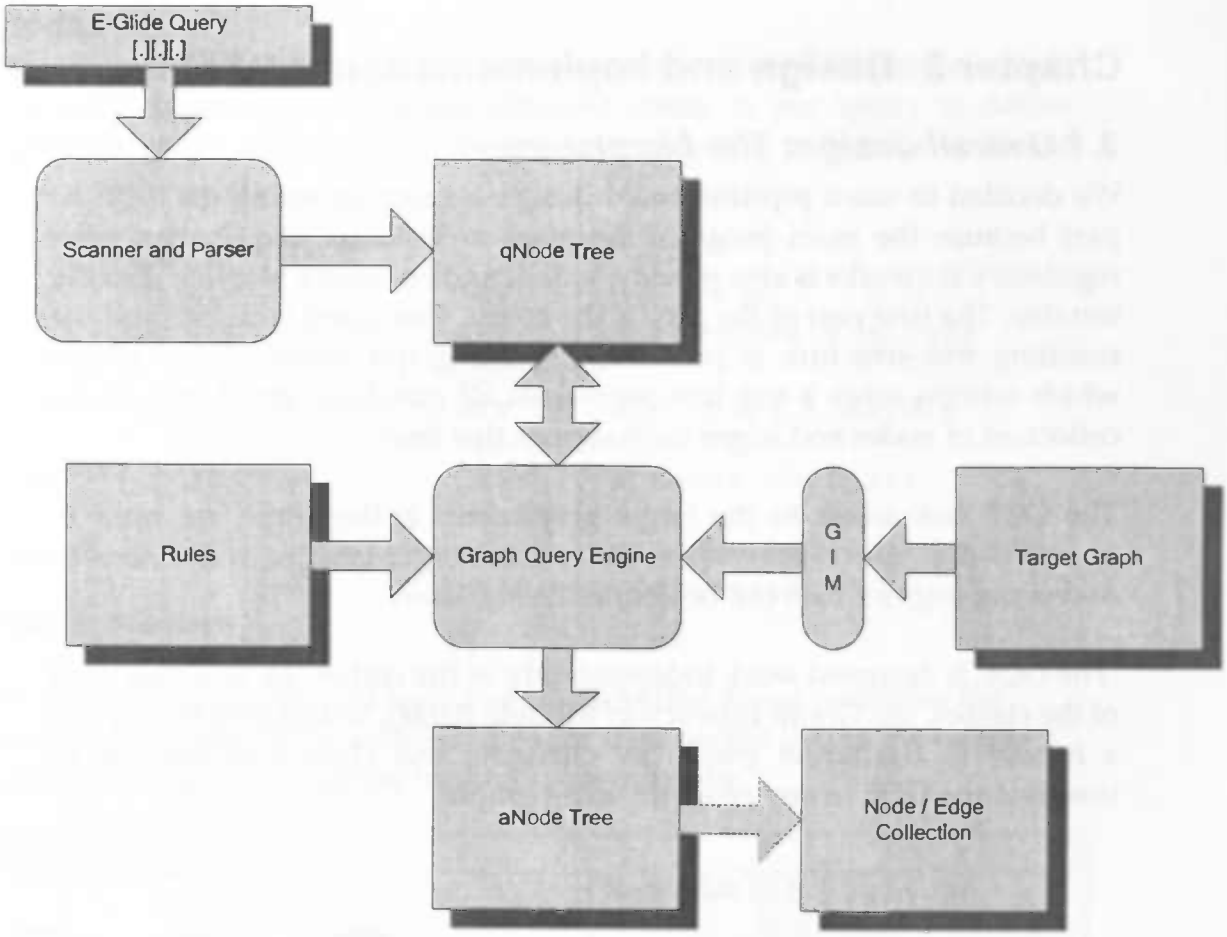


Figure 3-1 : A simple representation of the GQL program structure.

A query is writhen by the user in the E-Glide language. This query is parsed by the scanner and the parser which create a qNode-tree. The qNode-tree is then used by the graph query engine to search the target graph. The interface between the GQE and the target graph is provided by the GIM. Rules defined by the user are used to check the attributes of the target graph's nodes and edges. The result of a query is stored in an aNode-tree structure which contains all individual answers to the query; this structure can be converted to a collection of all nodes and edges that are part of an answer.

3.2 Language: E-GLIDE

3.2.1 Analysis

The language of a graph query tool is the part directly relevant to a user therefore it needs to be understandable, learnable and preferably as compact as possible.

Our research in existing GQL's brought us to the GQL known as GLIDE.

Glide (Graph LInear DEscription) is a language designed to express graphs. Glide represents a graph with a linear notation. The main idea is to represent a graph as a set of branches where each node is presented only once. The expressions in Glide, called regular graph expressions, allow the description of portions of graphs. Glide is designed to be as general as possible without compromising efficiency[b]

For a complete definition of Glide see Appendix C: Glide Language.

It became quickly apparent that the expressional power of the GLIDE language was insufficient for our needs. GLIDE is only able to search for structures within a graph, we however needed a GQL that also incorporated disallowing some graph structures and more detailed querying to the attributes of the nodes and edges.

We decided to adapt GLIDE to our own needs. This meant adapting the language to allow negation of structures, grouping of structures and adding the possibility for rules. Because of the extended features we decided to name our variant of GLIDE, E-GLIDE which stands for Extended Graph LInear DEscription). The differences with the original language will be discussed in section 3.2.3

3.2.2 Design / E-GLIDE Manual

The E-GLIDE language has 3 sorts of structures: Nodes, Edges and Collections. Nodes represent the nodes of the target graph and edges represent the edges of the target graph, whereas a collection is structure build out of nodes and edges.

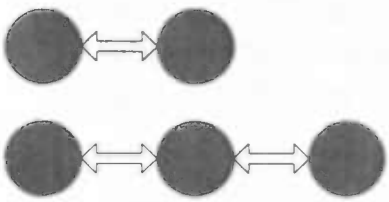
Nodes

A node in E-GLIDE is a structure that represents a part of the target graph we want to search for. We use different nodes in our query to define different graph structures in the target graph. We call these different nodes node-types.

.	(Point /dot)	: A Single node
?	(Question)	: Zero or One node
*	(Star)	: Zero or More nodes that are connected[1].
+	(Plus)	: One or More nodes that are connected.
(x)	(Exact)	: Exactly x nodes that are connected.
(x,y)	(MinMax)	: Minimal x nodes maximal y nodes that are connected.

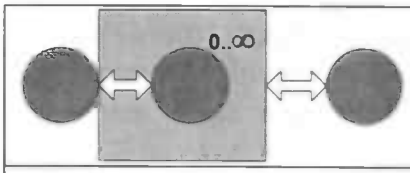
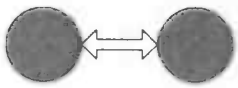
Table 1 : Node-types.

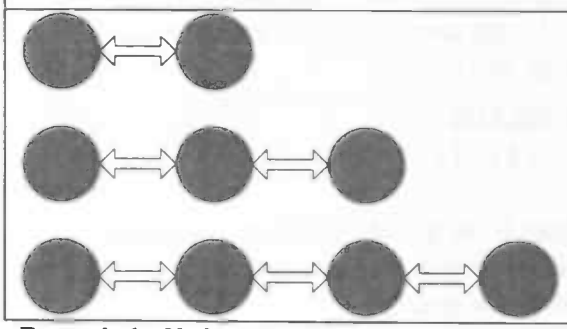
A node is always enclosed by '['and ']'. Note that in our examples the nodes are connected by undirected edges. This means that in the target-graph the edge can be either incoming ore outgoing.

EXAMPLES	
	**E-GLIDE: [.][.][.];** This is one of the most basic queries, search for three connected nodes.
	**E-GLIDE: [.][?][.];** Search for two nodes that may or may not be connected by[2] a single node.

[1] We use the word 'connected' to refer to nodes which appear in order as part of one branch, e.g. 'tree connected nodes' means that 3 nodes are consecutive and each node is linked to its predecessor by an edge.

[2] 'x and y Connected by z' means that two nodes x and y are connected to both ends of structure z.

	`E-GLIDE: [.][*][.];` Search for two nodes that are connected by 0 to ∞ nodes.
	`E-GLIDE: [(2)];` Search for two nodes that are connected. This is the same as: `[.][.];`

	`E-GLIDE: [(2,4)];`
	
	Search for a minimum of two connected nodes, and a maximum of four connected nodes.
	
	This is the same as:
	`[.][.][?][?]`

Example 1 : Node-types.

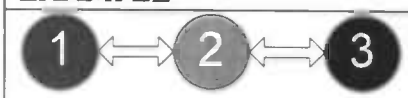
Node-rules

Nodes can be assigned rule-numbers; these numbers refer to rules provided by the user. A node-rule number is placed behind the node-type.

In our examples we use the following node-rules:

1. node colour is red.
2. node colour is green.
3. node colour is blue.

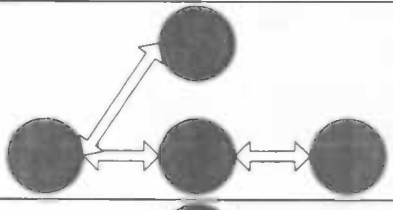
Note that the rules can be adapted to fit almost any graph. The different modules of the program being developed by the RuG research groups for instance can create and use rules to select attributes of the graph-nodes that are relevant to their functioning.
E.g. the metrics module could use biologically relevant data such as DNA sequence, while the layout algorithm could use graph data such as number of incoming paths (ingrad).

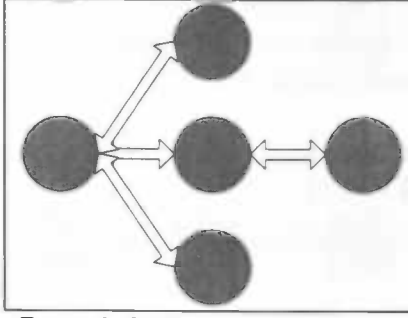
EXAMPLE	
	`E-GLIDE: [.1][.2][.3];` Search for three connected nodes using node-rules 1, 2 and 3 respectively.

Example 2 : Labelled nodes.

Branches

Nodes can have connections to other nodes. In our previous examples the number of connections for each node was at most two. When we want to have for example 3 nodes we use '(' and ')' to indicate we want to add a path to a node.

EXAMPLE	
	`E-GLIDE : [.]([.])[(2)];` Search for a node with two branches, one with two connected nodes, and one with one node.

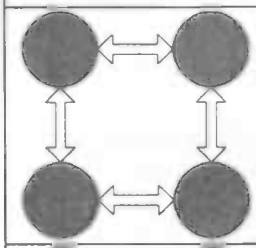
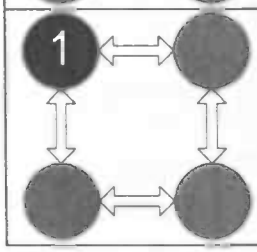
	`E-GLIDE :`
	`[.]([.])([.])[(2)];`
	
	Search for a node with tree branches, one with two connected nodes, two with one node.

Example 3 : Branches.

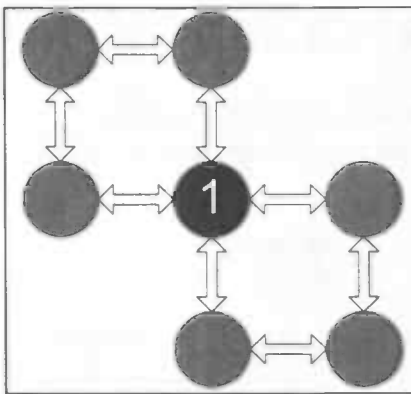
Cycles

If we want to search for a cycle we need to specify this, we do this by defining a cycle as a branch in which the first and the last node are connected by a special 'cycle-edge'. This means that we are searching for a connection between the first and the last node. If a Node is the beginning or the end for more then one cycle we separate the cycle-labels with a comma. Cycle-labels consist of the '%'and an integer number more then zero.

It is necessary to specify one cycle-labelled node 'downstream' from the other. This means that the first node has to be reachable from the second without going into a branch. In other words the first may not be between and '(' and ')' if the second is not also between those two same brackets.

EXAMPLE	
	**E-GLIDE: [.%1][(2)][.%1];** Search for a cycle of length 4.
	**E-GLIDE: [.1%1][(2)][.%1];** Search for a cycle of length 4. With the first node of the cycle using node-rule 1.

Example 4 : Cycles.

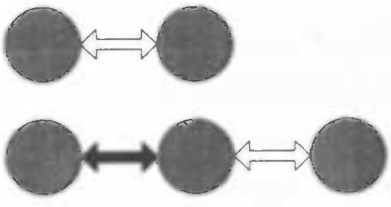
	**E-GLIDE :** **[.1%1,%2]([(2)][.%1])[(2)][.%2];** Search for a node using node-rule 1, and part of two cycles of length 4.

Example 5 : Branches and Cycles.

Edges

Edges can have edge-rules just as nodes can have node-rules. When an edge has a rule the edge is placed between two nodes. If an edge is placed ahead of a node that can have or has mutable instance, the edge-rule holds for all edges between those instances.

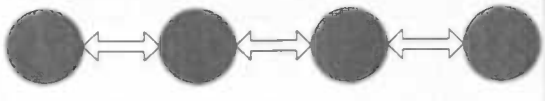
In our example we use the edge-rule 5: edge colour is red. We could also define a rule that defines this edge as a '**from** this node **to** that node'-edge thus creating a directed graph structure.

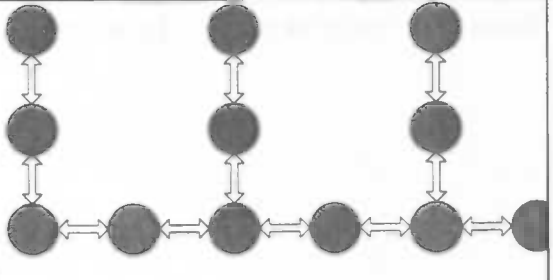
EXAMPLE	
	`E-GLIDE: [.]{5}[.][.];` Search for three connected nodes. The edge between the first two nodes must satisfy edge-rule 5.
	`E-GLIDE: [.]{5}[?][.];` Search for two nodes that may or may not be connected by a single node. If the node exists the edge between it and the first must satisfy edge-rule 5.
	`E-GLIDE: [.]{5}[(2)][.];` Search for four connected nodes, the first two edges must satisfy edge-rule 5.

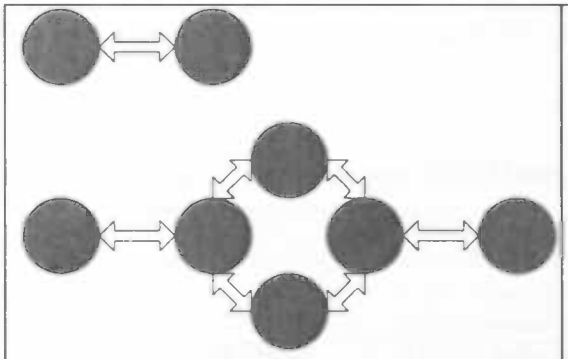
Example 6 : Edges.

Collections

Collections are sub-queries. They are defined by an E-GLIDE query between '<' and '>'. This part can now be used as a single node using the same notation as with the node-types. The type is placed behind the closing '>'.

EXAMPLE	
	`E-GLIDE: [.]<[.]>(2)[.];` Search for four connected nodes. This is the same as `[(4)];`

	`E-GLIDE:`
	`<[.]([(2)])[.]>(3);`
	
	Search for 3 times the occurrence of `[.]([(2)])[.]` which are connected.



```
E-GLIDE:
[.]<[.]([.%1])[.][.%1]>?[.
];
```

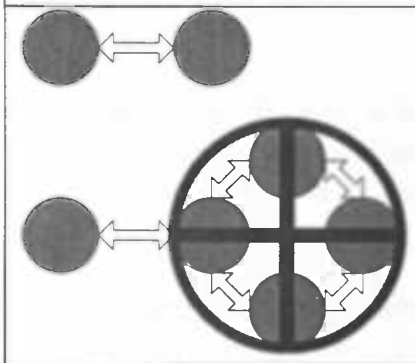
Search for two nodes that may or may not be connected by a cycle of length four.

Example 7: Collections.

Negation

Sometimes it is useful to disallow some graph-structure to be part of the answer. For this we use the negation. By adding a '-' to a collection this collection should not be possible to find from the point were we insert the negation.

EXAMPLE



```
E-GLIDE:
[.]-
<[.]([.%1])[.][.%1]>.[.];
```

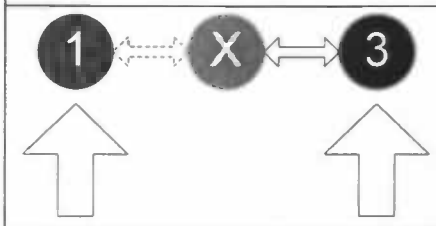
Search for two connected nodes, the second node may not be part of a cycle of length four.

Example 8: Negation.

No-show

The no-show option allows for nodes and edges not to be shown as part of the answer. A no-show is defined as a '!' placed directly after the opening '[' of a node or opening '{' of an edge.

EXAMPLE




```
E-GLIDE: [1]{!}[!.]{!}[2]
```

Search for two nodes connected by a single node. Only the [1]-node and the [2]-node are shown in the answer.

Example 9: Using a no-Show.

3.2.3 Differences with Glide

The E-GLIDE language differs from the original Glide language on a number of points. Glide only offers structural searches, while the implementation of rules allows E-GLIDE to search for attributes of the nodes. See Table 2 for a comparison of features.

Feature	Glide	E-GLIDE
Searching for structures	YES	YES
Defining rules	NO	YES
Allows cycles	YES	YES
Defining collections	NO	YES
Allows for nodes and edges not to be shown	NO	YES
Negation	NO	YES

Table 2: Comparison of Glide and E-GLIDE

3.2.4 Implementation

For the use in CUP (see section 3.3) we devised a formal grammar to describe the E-GLIDE language. A **formal grammar** is a way to describe a formal language, i.e., a set of finite-length strings over a certain finite alphabet.

A formal grammar consists of a set of rules for transforming strings. To generate a string in the language, one begins with a string consisting of only a single "start" symbol, and then applies the rules (any number of times, in any order) to this string. The language consists of all the strings that can be generated in this manner.

The formal grammar for E-GLIDE can be found in **Appendix A: A formal language definition for E-GLIDE**

Assume the alphabet consists of 'a' and 'b', the start symbol is 'S' and we have the following rules:

1. S → aSb
2. S → ba

then we can rewrite "S" to "aSb" by replacing 'S' with "aSb" (rule 1), and we can then rewrite "aSb" to "aaSbb" by again applying the same rule. This is repeated until the result contains only symbols from the alphabet. In our example we can rewrite S as follows: S → aSb → aaSbb → aababb. The language of the grammar then consists of all the strings that can be generated that way; in this case: ba, abab, aababb, aaababbb, etc.

Example 10: Generating a string using a formal grammar.

3.3 Scanning and parsing: JFlex and CUP

3.3.1 Analysis

In applications that require user input, this input has to be analysed. When the input consists of simple unstructured data a parse can be written by hand. However when data are complex structured, e.g. text in a text editor, or queries in a database program, data must be analysed. For this we use a combination of program, elements called a lexer and a parser.

A **lexical analyzer**, or **lexer** for short, performs a **Lexical analysis.** This is the process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called "lexical tokens", or just "tokens", which may be handled more easily by a parser.[c]

A lexer typically has two stages. The first stage is called the *scanner* and is usually based on a finite state machine. It reads through the input one character at a time, changing states based on what characters it encounters. If it lands on an accepting state, it takes note of the type and position of the acceptance, and continues. Eventually it lands on a "dead state," which is a non-accepting state which goes only to itself on all characters. When the lexical analyzer lands on the dead state, it is done; it goes back to the last accepting state, and thus has the type and length of the longest valid lexeme.

A lexeme, however, is only a string of characters known to be of a certain type. In order to construct a token, the lexical analyzer needs a second stage. This stage, the *evaluator*, goes over the characters of the lexeme to produce a *value*. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser.

A **parser** is a computer program or a component of a program that analyses the grammatical structure of an input, with respect to a given formal grammar, a process known as parsing. Parsers can be made both for natural languages and for programming languages. Programming language parsers tend to be based on context free grammars as fast and efficient parsers can be written for them. For example LALR parsers are capable of efficiently analysing a wide class of context free grammars[c].

Scanners and parsers are generally not written by hand, but generated by scanner- and parser generators[d]. We searched the internet for lexer and parser generators for JAVA and found among others JFlex and CUP, which were similar in syntax to the tools we used in previous assignments for the course compiler design. Since we could use the know-how of compiler design we decided to go with JFlex and CUP.

3.3.1.1 JFlex

JFlex[e] is a lexical analyzer generator for Java(tm), written in Java(tm). It is also a rewrite of the very useful tool JLex which was developed by Elliot Berk at Princeton University. They do not share any code though. JFlex is designed to work together with the LALR parser generator CUP.

3.3.1.2 Cup

The Java(tm) Based Constructor of Useful Parsers (CUP for short) is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java.

Using CUP involves creating a simple specification based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols).[f]

3.3.2 Design

The grammar for E-GLIDE is officially not a LALR grammar because it is ambiguous. However CUP can build a LALR-parser for this grammar by always choosing one production rule over another.

The data we acquire by parsing the query has to be stored, which is generally done in a tree structure. We made our own tree structure, built out of qNodes. The qNode (or query Node) is a structure that contains data about a part of the graph we are looking for. To create the tree we add some programming to the rules in our grammar. This programming is executed when the input of the parser is 'reduced'. Reducing the input string means simplifying the input by using the rules provided by the grammar of a language. When the tree is built we use it as input for the GQE.

Given the rules of Example 10

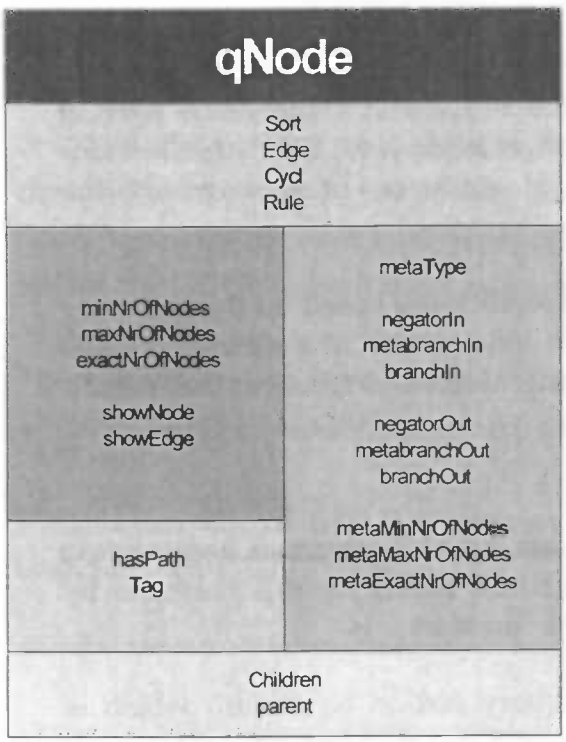
A string abab can be reduced to aSb

Example 11 : Reducing input.

3.3.3 Implementation

3.3.3.1 The qNode

The qNode and the initial data a qNode contains are created in the parsing process. This process builds the qNodes and combines them into a qNode-tree. This tree is used as input for the GQE.



A qNode consist of several sorts of data structures and data pointers. Some are used to define the qNode-tree others are used by the GQE to store temporary data.

Sort: Sort contains the type of the qNode. Its value can be

singleNode: This is the same as a '.' in E-GLIDE.

zeroOrMoreNode: This is the same as a '*' in E-GLIDE.

zeroOrOneNode: This is the same as a '?' in E-GLIDE.

oneOrMoreNode: This is the same as a '+' in E-GLIDE.

minMaxNode: This is the same as a '(x, y)' in E-GLIDE.

exactNrNode: This is the same as a '(x)' in E-GLIDE.

zeroNode: This type of qNode is used to indicate that there does not have to be a node in the answer. E.g. with the 'zero' of the zeroOrOneNode.

shadowNode: A shadowNode is a node that is used as beginning or end for a collection. It has no real value hence the name.

Edge: Holds the integer value of the edge-rule.
Cycl: Holds the integer values for the cycle edges.
Rule: Holds the integer value for the node-rule.

minNrOfNodes: If the qNode is a minMaxNode this holds the minimal number of nodes to be created.
maxNrOfNodes: If the qNode is a minMaxNode this holds the maximal number of nodes to be created.
exactNrOfNodes: If the qNode is an exactNrNode this holds the number of nodes that have to be created.
showNode: Boolean indicating if the graph-nodes found by way of this node should be shown in the final answer.
showEdge: Boolean indicating if the graph-edges found by way of this node should be shown in the final answer.

metaType: If the node is the first or last node of a collection this contains the type of the collection. The same values apply as for the sort field with exception of the shadow which does not exist for collections.

The following six values are used to identify the beginning end ending of branches, negations and collections. These values are primarily needed for copping of the branch since a copping algorithm needs to know where to start and stop copying.

negatorIn: Indicates that the node is the start of an negator collection.
metabranchIn: Indicates that the node is the start of a collection.
branchIn: Indicates that the node is the start of a branch.

negatorOut: Indicates that the node is the end of a negator-collection.
metabranchOut: Indicates that the node is the end of a collection.
branchOut: Indicates that the node is the end of a branch.

metaMinNrOfNodes: same as minNrOfNodes but now for the collection.
metaMaxNrOfNodes: same as maxNrOfNodes but now for the collection.
metaExactNrOfNodes: same as exactNrOfNodes but now for the collection.

Children: Holds the pointers to the children of this node
Parent: Hols a pointer to the parent of this node.

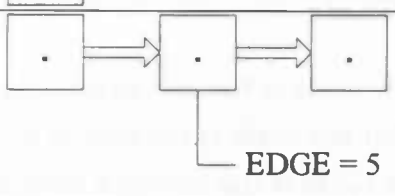
hasPath: used in building the tree, it indicates if this node already has a path that is not a branch, this is used to 'glue' qNode-structures together.
Tag: used in cloning the tree, If we clone a (part) of a qNode Tree and we want to keep track of a specific node within the tree we can use this field to tag a specific qNode.

3.3.3.2 The qNode-tree structure

The qNode-tree closely resembles the graph element it queries for. The GQE will use its structure as well as the node information while searching for matches in the target graph.
The information we provided in the E-GLIDE-query is distributed to the corresponding nodes and edges in the qNode tree. One extra element in the qNode-tree that is not present in the E-GLIDE-query is the shadow-node. Two of these nodes are used to 'hold' the collections, enclosing them so that the GQE can treat them as a single node.

The qNode-tree is a true tree, so it does not have cycles. The cycle-edges are stored in the qNode until used by the GQE. (See section 3.3.6.5)

Some examples of E-GLIDE queries and resulting qNode-trees can be found in Figure 3-2.

`[.];`	
`[.]{5}[.][.];`	

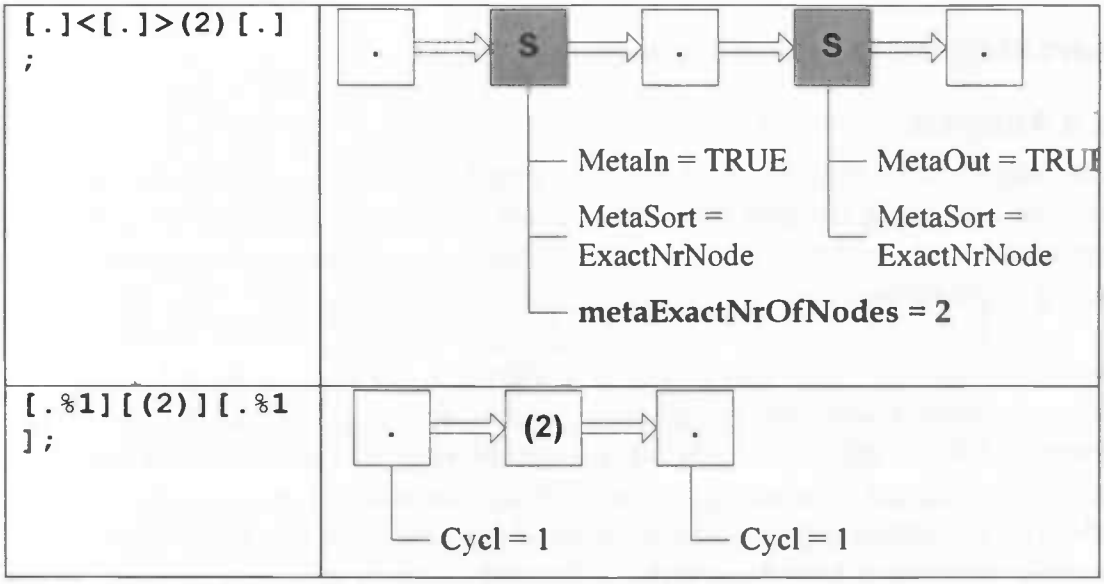


Figure 3-2: E-GLIDE to qNode.

Searching the graph: Graph query engine

3.3.4 Analysis

Many algorithms exist for searching in graphs, only a few of those are used for searching for sub-structures. Most of those algorithms concern themselves with finding cycles in a graph for the purpose of shortest-path finding algorithms etc.

We also looked at GraphGrep; this is a software package to search for a query graph in a database of graphs. Given a collection of graphs and a pattern graph, GraphGrep finds all the occurrences of the pattern in each graph. The pattern is a sub-graph and it can be also a tree, a path, or a node. The pattern is expressed as a list of nodes and a list of edges[g]. One of the early versions of GraphGrep (1.0-1.1) used Glide as input.

GraphGrep itself was not used because it is optimised for static (non-changing) graphs. Constantly updating the database would cost too much system time. We could not find any more GQL-like applications so we decided to build our own.

Before we can build a graph query algorithm we must ask ourselves the question: How fast can this algorithm be? Complexity theory tells us it is not possible to build an algorithm that will do *any* query in a timeframe most people would find acceptable.

For instance we could use a graph query to solve the 'travelling salesman problem[1]'. This is a NP-complete problem[h] which means that this problem (probably) cannot be solved in polynomial time[2].
See Figure 3-3.

[1] Given a number of cities and the costs of travelling from one to the other, what is the cheapest roundtrip route that visits each city and then returns to the starting city?

[2] In computational complexity theory, **Polynomial time** refers to the computation time of a problem where the time, $m(n)$, is no greater than a polynomial function of the problem size, n.

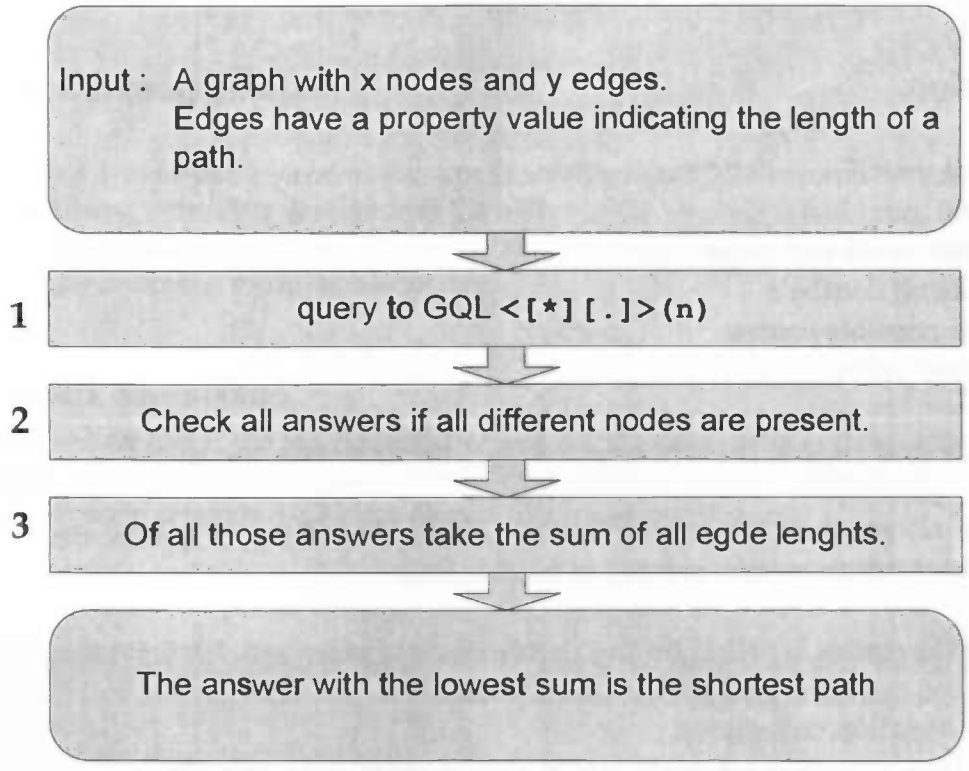


Figure 3-3: A GQL can solve the travelling salesman problem.

Steps 2 and 3 can be done in polynomial time (both n*n) so the NP-complete part must come from step 1 representing the GQL.

So we must be content with an algorithm that is at most $O\ (\ X^n)$ with n being the number of graph-elements and X being a random number > 1.

Given the qNode-tree provided by the parser, how must we proceed? A qNode has to be matched to a graph-node. And the following conditions must hold.

1: if provided, the rules for edge and node must hold.
2: if the qNode has children, the graph-node must have children that match the qNode's.
3: check if the qNode is the begin or an end of a cycle.

The accomplish this we designed the following procedures.

3.3.5 Design

We divided the GQL in four major parts:

1. Check node and edge
2. Make all combo's
3. Find possible routes
4. Create paths

Check node and edge is used to provide the interface with the rules.

The **create all combo's** algorithm creates all combinations of paths and graph-node children, which al have to be checked.

Find possible routes is called by the pathfinder to create from the children of a node the possible paths they can represent. It also handles the other part of the negation collections.

Create paths is the search algorithm, it checks the node-rules, the edge-rules, checks for cycles and handles a part of the negation collections.

There is also a **Graph interface module** of **GIM** that interacts with the target graph. This module is be specific for the target graph and provides operations on it like getting the children of a graph-node.

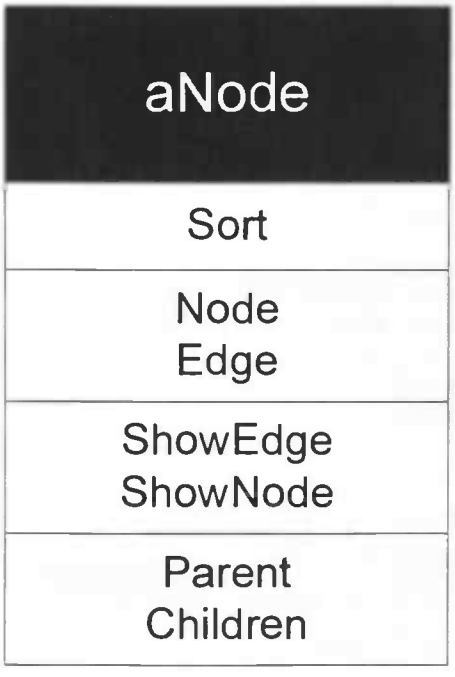
The following section gives an overview of the most important procedures in the GQE.

3.3.6 Implementation

After a qNode-tree is created this tree is used by the GQE to search the graph. An initializing procedure iterates over the nodes of the target graph offering them to the createPaths procedure. The createPaths procedure uses the nodes along with the qNode-tree to create an aNode-tree.

3.3.6.1 The aNode

The aNode tree is the structure that will contain the answer(s) to the query (if any). The aNode-tree consists of individual aNodes that are connected by child/parent relations. ANodes consist of a number of values:



Sort: An integer number which refers to the type of aNode we are using.

Node: A pointer to a graph-node[1] or NULL

Edge: A pointer to an graph-edge[2] or NULL

ShowEdge: a Boolean that tells us if the edge should be seen in the final answer.

ShowNode: a Boolean that tells us if the node should be seen in the final answer.

Parent: The parent node or NULL if does not exist.

Children: A vector containing the children of the aNode.

The answers the aNode tree contains can be used in 2 different ways:
1: The graph-nodes and graph-edges are collected from the tree, and deposited in a collection. This collection represents all graph-nodes and

[1] If we refer to a graph-node we mean a node that is part of the graph we are searching in.
[2] If we refer to a edge-node we mean a edge that is part of the graph we are searching in.

graph-edges that are part of the solution. However, the context of how these graph-nodes and graph-edges are part of a solution is lost in this process.

2: We keep the aNode-tree intact. Its nodes and the structure of the aNode tree represent the individual answers to the query. This information is largely kept by the structure but also by the Sort field. Using this information we can reconstruct individual answers to the query.

The Sort-field can have any of these values:

0	UNDEFINED	Every aNode starts its life as an undefined aNode. This value can not be in the final answer. If it is, the program has an error in its code.
101	AND_NODE	The AND_NODE can be found in an answer when a branch is encountered. It means that all of the children of this aNode are part of the same answer. See Figure 3-4
102	OR_NODE	The OR_NODE in contrast to the AND_NODE means that the children of this aNode are all different valid answers to the query. See Figure 3-5
103	ENDING	An aNode is an ENDING when it is the last aNode of a branch, or in other words is a leaf-node in the aNode tree.
104	EMPTY_OR_NODE	Not used.
105	EMPTY_AND_NODE	Sometimes the choice is between two or more sets of possible branches, (e.g. a node followed by branches A and B or branches X and Y. To facilitate the efficient representation of these options there is the EMPTY_AND_NODE. See Figure 3-6

404 BAD_NODE

If the one of the algorithm returns an aNode of the type BAD_NODE it means that it could not find a path that matched the qNode tree. Note that this is a very different type then the UNDEFINED aNode. The former represents an error in the program the later represents a message to the program.

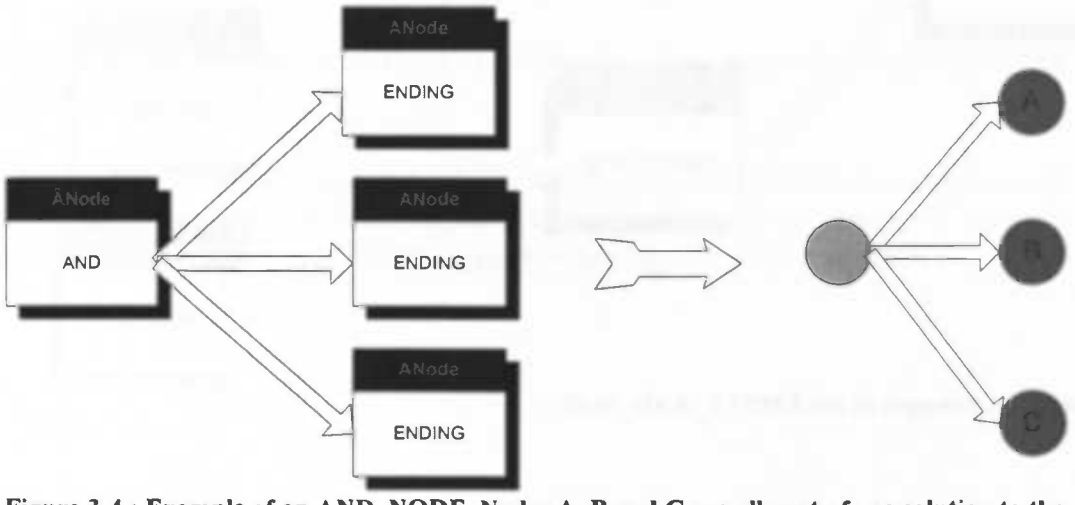


Figure 3-4 : Example of an AND_NODE. Nodes A, B and C are all part of one solution to the query.

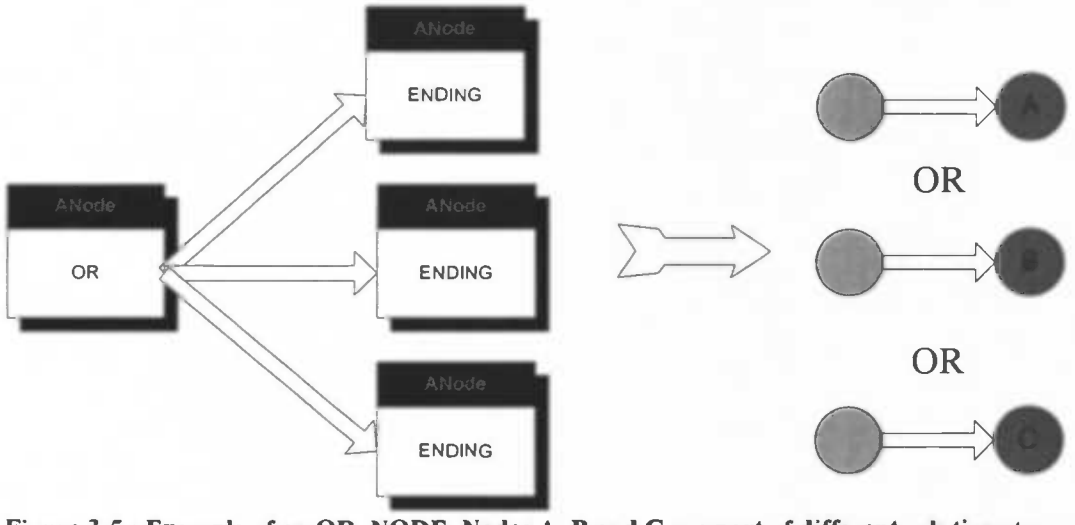


Figure 3-5 : Example of an OR_NODE. Nodes A, B and C are part of different solutions to the query.

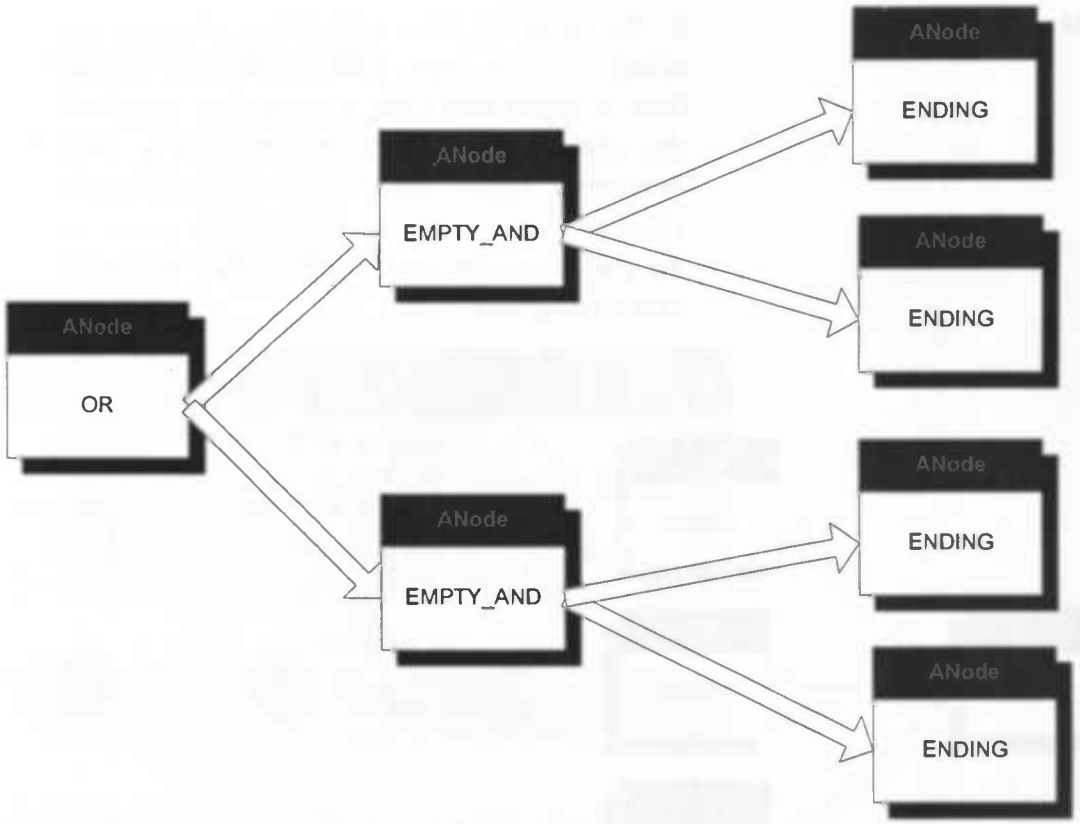


Figure 3-6: Example of the EMPTY_AND_NODE.

3.3.6.2 CheckNodeAndEdge

Input : **Object graph-node**
Object graph-edge

Output: **Boolean**

CheckNodeAndEdge is a small part of the program that provides the interface between the rules and the GQE. When checkNodeAndEdge is called, it refers to the rules the user of the program has defined elsewhere.

Note that the definition of rules is not part of the program, but is left to the user to implement.

CheckNodeAndEdge returns TRUE if the graph-node and the graph-edge pass the rules set for them, or if no rule is defined for them. It returns FALSE if the graph-node or the graph-edge do not pass the rule.

3.3.6.3 MakeAllCombos

Input : **Vector q**
Vector N1
Vector N2
HashMap H

Output: -

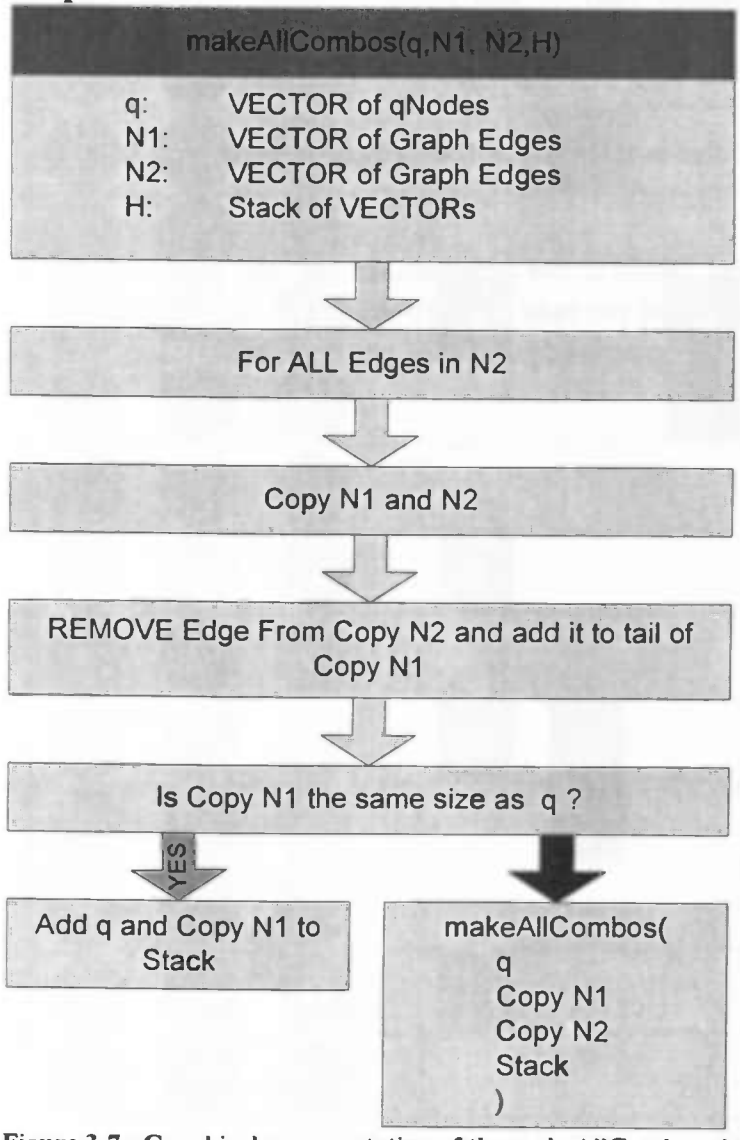


Figure 3-7 : Graphical representation of the makeAllCombos algorithm.

A qNode is always matched with a single graph-node / edge combination. If the graph-node has more then one child (represented by its edges) all possible combinations of qNode children and Graph-edges have to be considered by the GQE (see Figure 3-8). To find all these combinations we use the makeAllCombos algorithm.

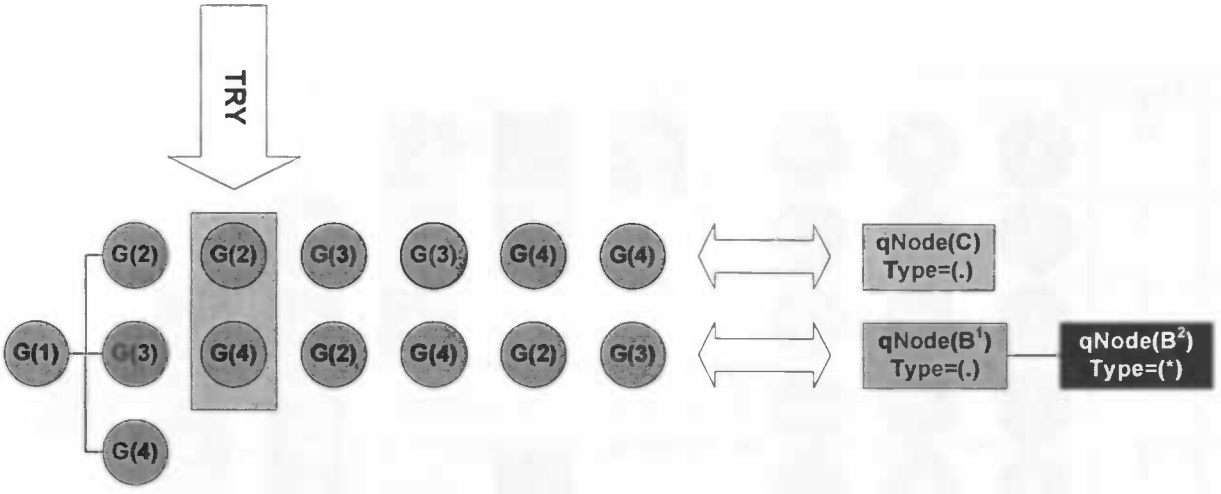


Figure 3-8 : Matching different graph-edges to a qNode tree.

MakeAllCombos takes three vectors (q, N1, N2) and adds to a stack (H[1]) the vectors that represent al possible combinations of graph-edges and qNodes.

When the algorithm is called by another procedure it is called with two vectors (q, N2) containing the qNodes and the graph-edges, while the third vector (N1) is empty. For correct functioning of makeAllCombos, the number of nodes in the first vector (q) may not exceed the number of edges in N2 in the initial call of the procedure.

The algorithm iterates over the edges stored in the vector containing the graph-edges (N2). It creates a copy of the vector minus the edge it is currently holding. The edge is added to the third vector (N3). It then calls itself recursively[2] for each graph-edge, thus building collections of edges using the third vector to store the order of edges. If the third vector has as many elements as the first the algorithm is done and pushes[3] the first and third vector on the stack (H).

Example 12 on page 46 shows the different stages of the makeAllCombos algorithm building the combinations.

[1] A **stack** is a data structure that works on the principle of Last In First Out (LIFO). This means that the last item put on the stack is the first item that can be taken off, like a physical stack of plates. A **stack-based** computer system is one that is based on the use of stacks, rather than being register based.

[2] **Recursion** is a way of specifying a process by means of itself. More precisely (and to dispel the appearance of circularity in the definition), "complicated" instances of the process are defined in terms of "simpler" instances, and the "simplest" instances are given explicitly.[c]

[3] **push**: an item is put on top of the stack, increasing the stack size by one..

	Depth	q	N2	N1	NEXT
A	0	A B C	X Y Z		B G L
B	1	A B C	Y Z	X	C E
C	2	A B C	Z	X Y	D
D	3	A B C		X Y Z	PUSH
E	2	A B C	Y	X Z	F
F	3	A B C		X Z Y	PUSH
G	1	A B C	X Z	Y	H J
H	2	A B C	Z	Y X	I
I	3	A B C		Y X Z	PUSH
J	2	A B C	X	Y Z	K
K	3	A B C		Y Z X	PUSH
L	1	A B C	X Y	Z	M O
M	2	A B C	Y	Z X	N
N	3	A B C		Z X Y	PUSH
O	2	A B C	X	Z Y	P
P	3	A B C		Z Y X	PUSH

Example 12 : The makeAllCombos Algorithm creating all possible combinations for a 3 edge graph-node and a 3 child qNode. The depth is the recursion depth. A 'PUSH' on the right side means that this combination is done and can be pushed onto the stack. The letters A..P represent the steps that are done next.

3.3.6.4 FindPossibleRoutes

Input : **Vector possibleNodes**
Object g
Object parentEdge

Output: **Vector**

FindPossibleRoutes is an algorithm that checks all the children of a qNode which are contained in the Vector posibleNodes and changes them into qNodes of the type singleNode.
This enables the findPaths algorithm to use the qNodes in building a path.

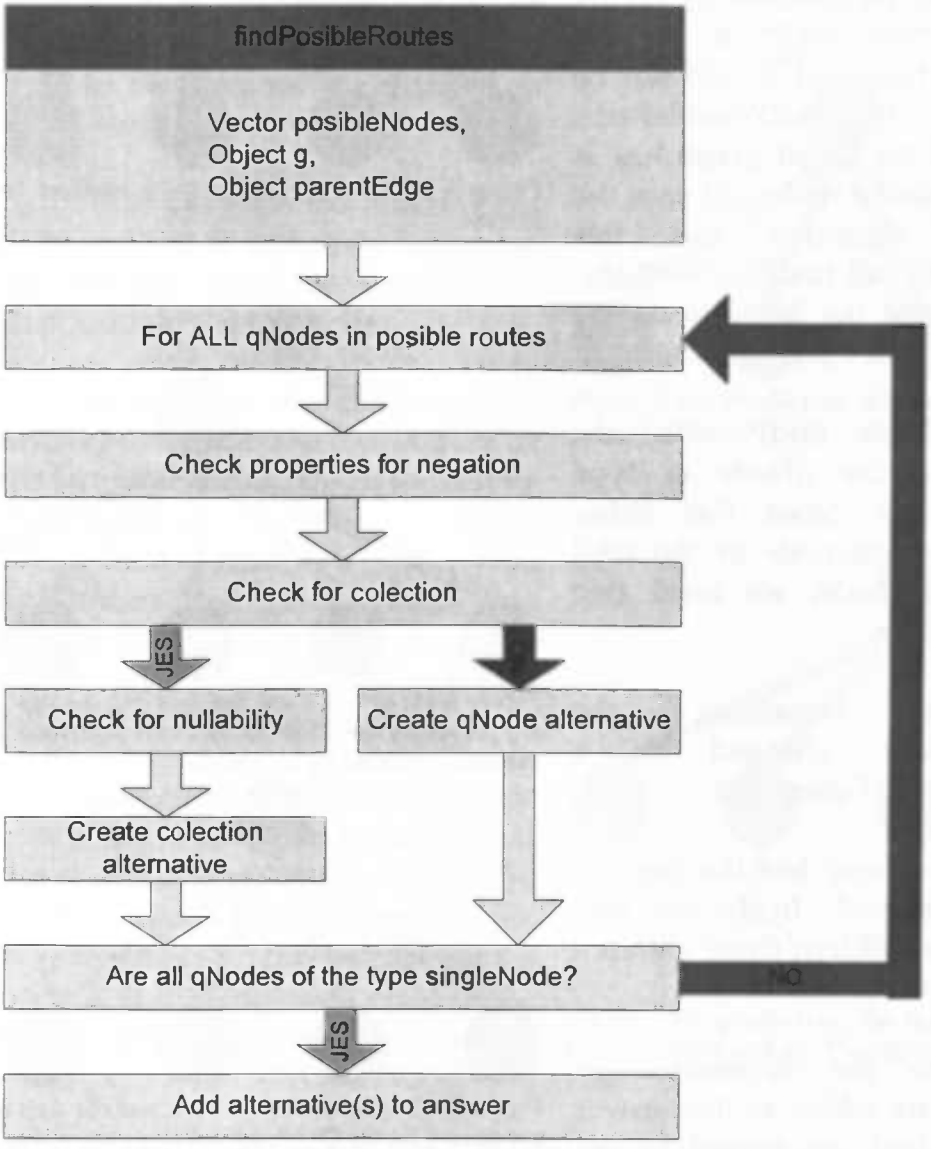


Figure 3-9 : The findPossibleRoutes algorithm.

The first part of the algorithm checks for the negator property in the qNodes. If it is found, a number of checks are made. One of those is checking if a negation branch has no children. In that case we can try the qNodes in the possibleRoutes vector minus the negator qNode.

The second part divides the qNodes into nodes which are collection and nodes that are ordinary nodes.

Let's first consider the case that children are all nodes (and not collections, we will discuss those later on).

Consider the E-GLIDE query `[.]([?])([*])([+])([2])[2, 3]` this results in a qNode-tree that can be visibly represented by Figure 3-10. The root node is of the single_node type and it will not be altered by the findPossiblePaths algorithm. If the target graph has at least one node the node will pass the createPaths algorithm and this algorithm will call findPossiblePaths. The children of the first qNode are contained in vector possibleNodes.

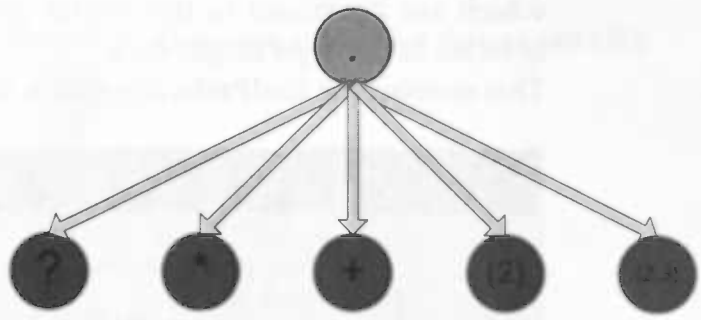


Figure 3-10: qNode-tree, the root is checked, the leaf nodes are not.

The first qNode findPossiblePaths encounters is the qNode of type zeroOrOneNode. Since this either describes a single_node or the total absence of a node, we need two possible answers.

The first containing the zeroOrOneNode changed to a single_node. See Figure 3-11.

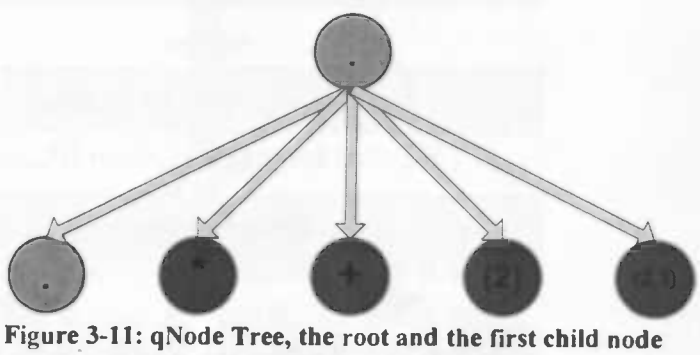


Figure 3-11: qNode Tree, the root and the first child node are parsed by the findPossiblePaths algorithm, a '?'-node has been replaced by a '.'-node.

The second answer has the zeroOr-OneNode removed. In the case that the node has children those children are added to the parent node.

Both answers are represented by vectors and are added to the answer vector after both are parsed by the findPossibleRoutes algorithm, which

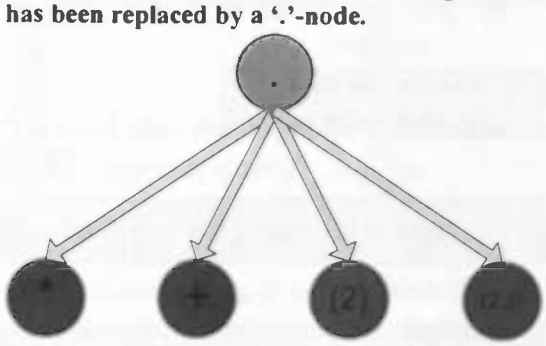


Figure 3-12: qNode-tree, the root and the first child node are parsed by the findPossiblePaths algorithm, a '?'-node has been deleted.

ignores child nodes of the type single_node.

The second child of the type zeroOrMoreNode is treated almost the same as the oneOrMoreNode. Here we also need two answers, one with the node changed into a single_node, one with the node removed. (See Figure 3-14)

Different from the zeroOrOne node is that in the first answer the oneOrMoreNode may occur a number of times. This case is handled by adding a copy of the node as a child to the instance we changed to single Node. See Figure 3-13

In the case this node has a number of children. This is solved by removing the children from the instance we changed into a single node.

This means that the children of a zeroOrMoreNode are found only at the last node of the branch formed by these nodes.
The third node is of type oneOrMoreNode. This means that there is always at least one instance of this node so we only need to return one vector containing the answer.

By transforming the oneOrMoreNode into a single_node we have satisfied the *at least one* part of the node. We need to address the *or more* part.
We can simply do this by copying the node, add this copy as a child to the just created single node, and change its type to zeroOrMoreNode. See Figure 3-15.

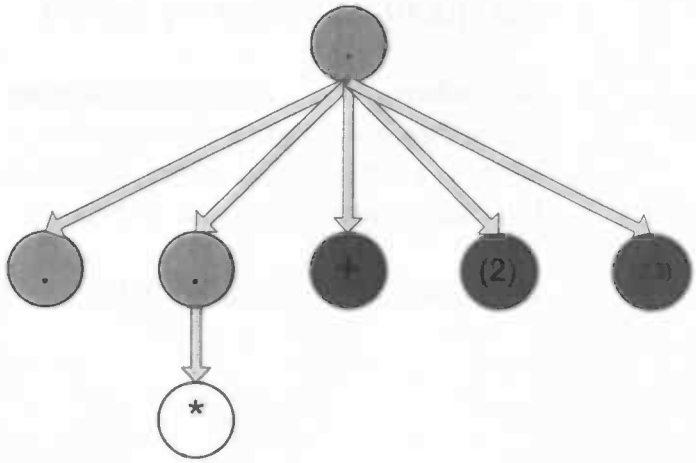


Figure 3-13: qNode-tree, The second node is changed from a '*'-node to a '.'-node.

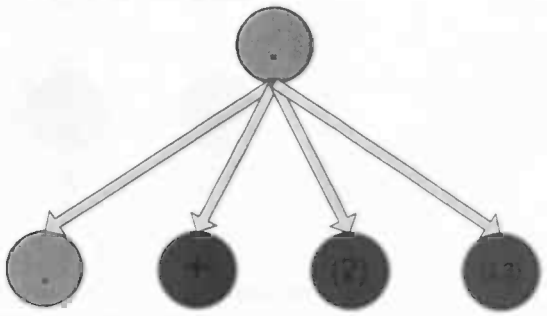


Figure 3-14: qNode-tree, the second node has been removed.

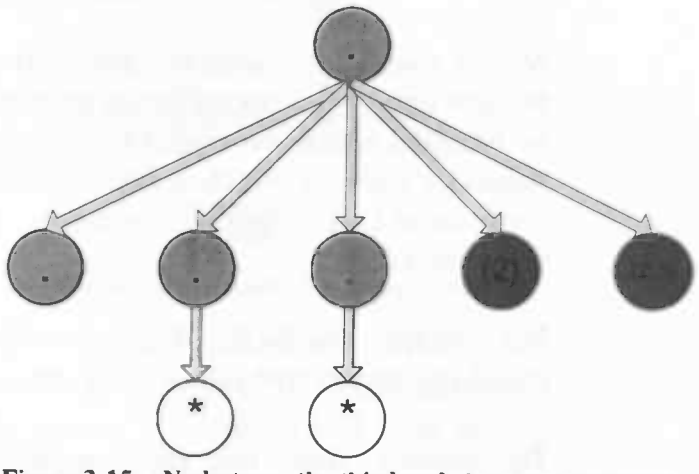


Figure 3-15: qNode-tree, the third node has been replaced by an '.'-node and an '*'-node is added as a child to that node.

The fourth and fifth node are actually shorthand notations. E.g. **[(3)]** in E-GLIDE is exactly the same as **[.][.][.]** and **[(2,4)]** is the same as **[.][.][?][?]**.

So when the findPossibleRoutes algorithm encounters those nodes it simply changes the short notation into the long notation and carries on.

See Figure 3-16 and Figure 3-17.

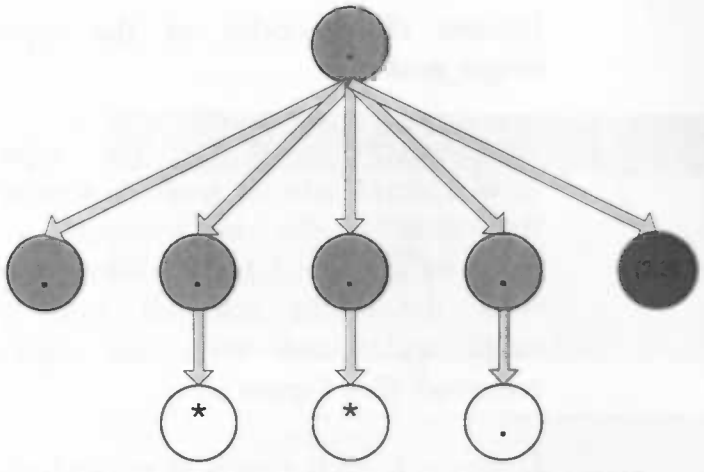


Figure 3-16: qNode-tree, [(2)] node is changed into [.][.].

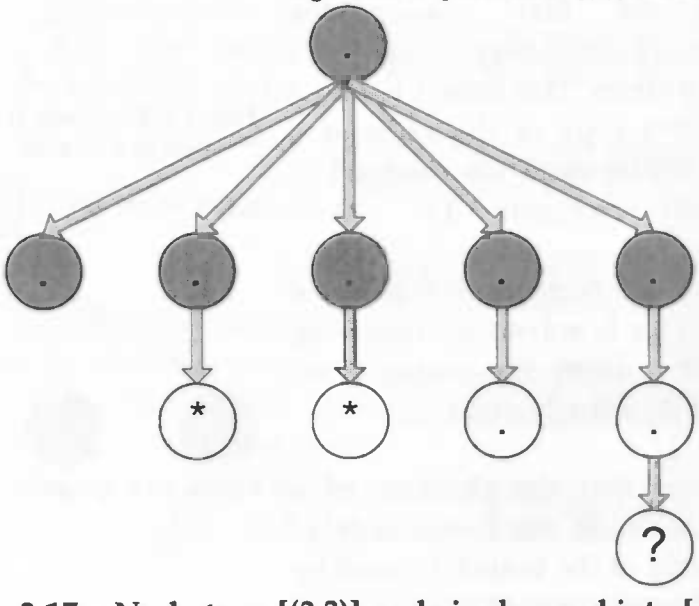


Figure 3-17: qNode-tree, [(2,3)] node is changed into [.][.][?].

Nodes that are collections are treated in basically the same way. Just replace any of the nodes in the examples above by a collection and it will be handled the same way. One exception is that we must check for and handle a case in which a collection is of the type zeroOrMore and the contents of that collection are empty. This will lead to a unending loop in the program.

For example, the E-GLIDE query <[*]>* would become [*]<[*]>* and that could become <[*]>* which is the same as the original query.

To counter this, the findPossibleRoutes algorithm checks for the possibility that a collection can have zero nodes. If so, it eliminates that possibility by changing the offending nodes into other types. Note that the possibility of having zero nodes is still maintained by the zeroOrMore collection. This however cannot result in an unending loop.

Figure 3-18 shows a collection being changed, so the zero nodes problem can not occur. All tree possibilities are checked, so no possible answers to the query are lost.

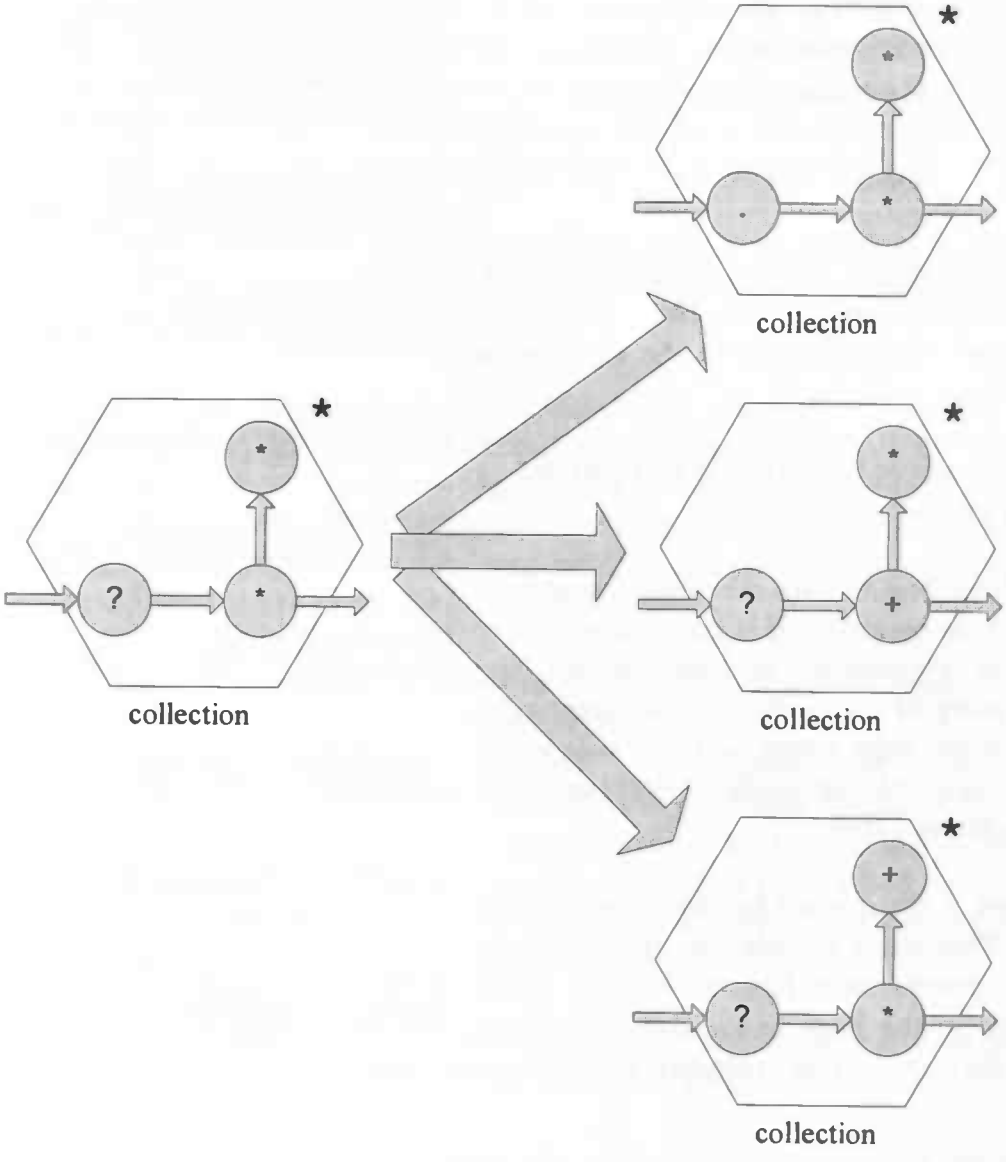


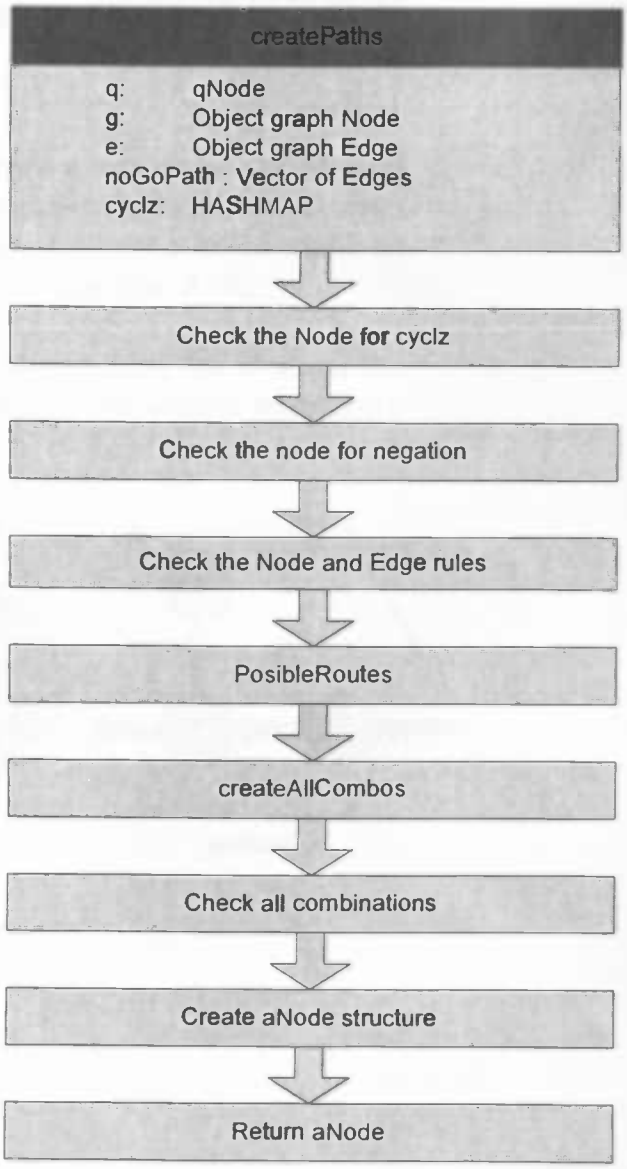
Figure 3-18: A collection that can have zero nodes, split into three parts that cannot have zero nodes.

After iterating over al the node and collections of the possibleRoutes vector the findPossibleRoutes algorithm returns a vector filled with all possible mutations of the original possibleRoutes vector.

3.3.6.5 CreatePaths

Input : **qNode** **q**
Object graph-node **g**
Object graph-edge **e**
Vector of Edges **NoGo**
HashMap of graph-nodes

Output: **aNode**



CreatePaths is the workhorse of the graph query engine and calls the important subroutines that help create the answer to the query.

A aNode answer is created to be returned as result of createPaths.

CreatePaths begins with checking the qNode for the cycle property. If it has this property the cyclz Hashmap[1] is checked for the integer entry of the cycle. If it does not exist a New hash map entry is built using the integer value of the cycle as key and the graph-node as value.

If it exists, a child is added to a copy of the qNode. This child qNode has the property that its graph-node has to be the node contained in the hash map. The cyclz hash map in the copy has its cycle key and value removed.

[1] A hashmap is a Java implementation of a hash table. A **hash table** is a data structure that implements an associative array. Like any associative array a hash table is used to store many key value associations (this is a many to one relationship as the hash table is almost universally smaller than the number of keys). A hash table maintains two arrays, one for keys, one for values (or possibly one array of (key, value) pairs - it doesn't really matter). The elements of these arrays are referred to as buckets. When required to find the associated value for a given key, the key is fed through a hash function to yield an integer (called the hash value). This integer is then the index to the associated value.[c]

The createPaths procedure is then recalled using the copied qNode. It will now search for the node and edge that complete the cycle. The answer aNode will become the result and is returned to the caller of createPaths.

The next part of createPaths handles the negation. If a qNode has the negation property it will check if the negation collection can be found from starting from the graph-node (g). If this is the case, the procedure returns an aNode of the type BAD_NODE. If it cannot be made, it checks if the children of the negation collection can be found starting from the graph-node (g) and the procedure returns the resulting aNode.

Note that the case of the negation collection having no children is already handled by the findPossibleRoutes algorithm described in section 3.3.6.4 on page 47.

After negation is handled the createPaths algorithm invokes the checkNodeAndEdge routine checking the graph-edge e and the graph-node g (see section 3.3.6.2 on page 43). if the resulting Boolean is true, graph-edge e and the graph-node are added to the answer aNode and the procedure continues. If the resulting Boolean is false then the procedure returns a aNode of the type BAD_NODE.

A check is now made if the qNode has children. If not we are done and we can return the answer aNode as type ENDING.
If the qNode does have children we invoke the procedure findPossibleRoutes with the children of the qNode (see section 3.3.6.4 on page 47. The resulting vectors are then processed by the makeAllCombos algorithm (See section 3.3.6.3 on page 44 to create a stack filled with pairs consisting of a vector containing qNodes and a vector containing graph-nodes. The vectors have the same length.

Now for each pair of vectors the elements at the same index of those vectors is used as input for a new call of createPaths.

To prevent multiple calls of createPaths with the same pairs we put each result in a hashmap with as key the graph-node/qNode pair and as value the resulting aNode. Before we call createPaths we check in this map if we do not already know the outcome of the query.

If the result of this check or a call to createPaths is a aNode with a type other than BAD_NODE, the result is stored as a child in a temporary aNode of the type EMPTY_AND.

If one of the pairs of elements returns a BAD_NODE the temporary aNode becomes a BAD_NODE.

After a pair of vectors is completely checked and the temporary aNode is still of the type EMPTY_AND we add it to a second temporary node of the type OR. All pairs of vectors are checked and if of the type EMPTY_AND they are added to the second temporary node.

When we are done checking the pairs of vectors we examine the second temporary node. If it has no children the query has failed and we return a node of the type BAD_NODE. If it has only one child we add the children of that child to the answer aNode, and change its type into AND.

If the second temporary node has more than one child we change the answer aNodes type to OR and add the children of the second temporary node as children to the answer aNode.

Last but not least we return the answer aNode.

Chapter 4: Evaluation

The GQL has become a useful tool in searching for structures within large graphs. We now look at the requirements specification defined in section 2.2.1 to evaluate if our program meets those requirements.

4.1 requirements specification

4.1.1 Searching for graph structures.

E-GLIDE can define any graph-structure (the only 'structures' within a graph are tree's and cycles both can be described and combined by E-GLIDE).

To construct a E-GLIDE query for any graph-structure we can use the following depth first search algorithm which as analogous to the one described for Glide in Appendix C.

1. When visiting a node, a [.] printed.
2. When the next node to be visited has unvisited siblings a '(' is printed. A ')' is printed when all the siblings have been visited.
3. In a cycle, (we encounter a node already visited) the first and last node representations are labelled by inserting '%' followed by a unique (for each cycle) integer.

Note that multiple edges between two nodes can be described as cycle edges. Also note that E-GLIDE has a syntax that allows for a more compact representation of graphs.

The graph query engine can find all occurrences of a structure within a target graph. A small sample of the tests run on structure-finding can be found in Appendix B.

4.1.2 Testing of node and edge attributes.

In our tests we used a number of rules to test this feature, the results of these tests can be found in *Appendix B: Test results* .
Also using Jython we successfully tested the node and edges of the target graph (see section 4.2). It is however only possible to test individual nodes and edges. It is not (jet) possible to ask '*node has the same number of incoming edges as node C*' this problem is being addressed by Dinne Bosman.

4.1.3 Searching for multiple repetitions of structures.

The building of this feature was more difficult then might be expected since it involved a very complicated scheme of copying and testing for the empty collection. It is however now integrated in the E-GLIDE language and the graph query engine (see Test 16 in appendix B and Example 7 in section 3.2.2).

4.1.4 Possibility for negation of structures.

Negation can be used to search for branches that are not part of a other structure. It is however advisable to use the rules to define most negative rules on nodes and edges (such as this node may not be green), because it takes the algorithm less time. A test for the negation feature can be found in appendix B Test 14.

4.1.5 The graph data structure should not be modified.

The graph interaction module provides a interface with the graph that does not change the structure or contents.

4.1.6 Possibility for Jython integration.

The rules we give to the E-GLIDE nodes and edges have been used in the main program using Jython-scripts. The combination of the two has proved to be successful in searching out node and edges with certain attributes. Section 4.2 contains a screenshot of the main program using Jython with the GQL.

4.1.7 Preferred user interaction.

The E-GLIDE language has become a powerful language for describing a graph structure. Wile testing the application we quickly got used to its syntax, and we expect that with help of the E-GLIDE user manual a future user will also learn to use E-GLIDE in a short amount of time.

We set out from the beginning of this project to keep E-GLIDE as close to Glide as possible. While E-GLIDE has different in syntax then Glide it still has the same regular expression-like quality we liked in Glide.

(see section 3.2.2 and appendix C)

The program has al the properties defined in the requirements specification so we can say that our mission has been accomplished.

4.2 Integration

Currently the GQL is being integrated into the main program by Dinne Bosman. Preliminary tests with test-graphs and queries show promising results. The GQL is being used in the module that allows the user of the main program to select nodes and edges. Dinne Bosman has written an user interface for writing E-GLIDE queries and defining Jython scripts for the rules.

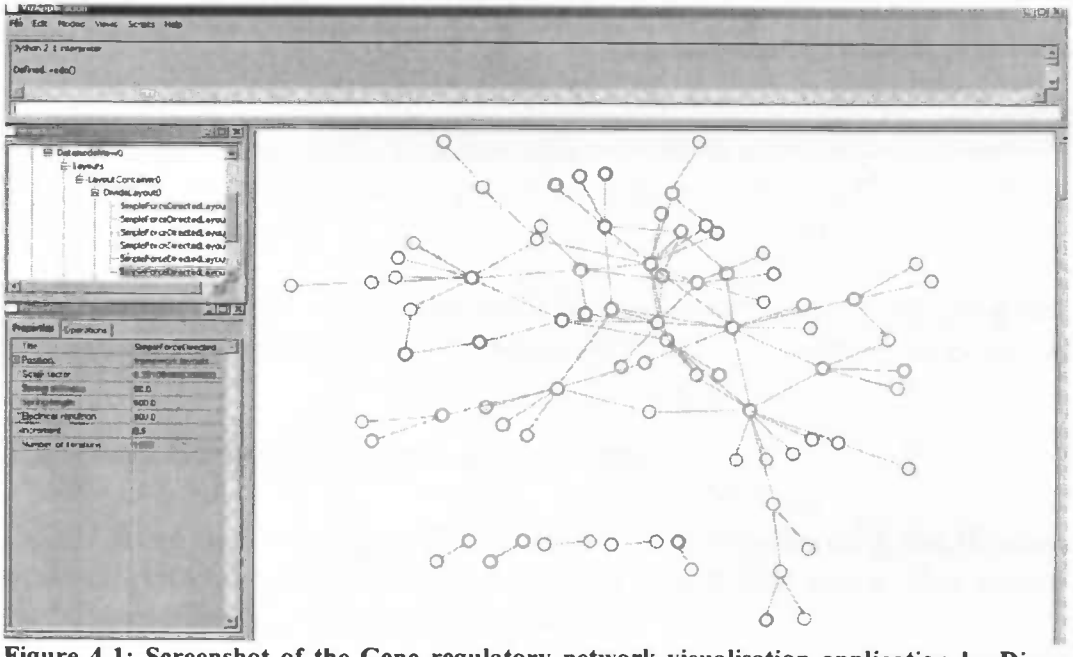


Figure 4-1: Screenshot of the Gene regulatory network visualisation application by Dinne Bosman.

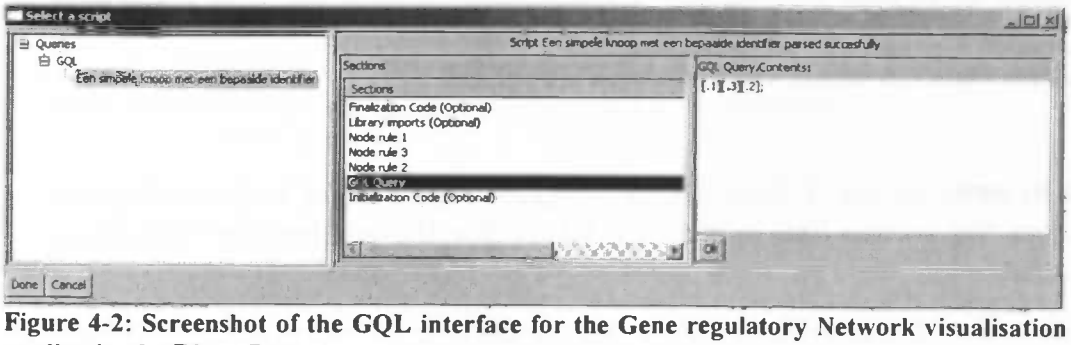


Figure 4-2: Screenshot of the GQL interface for the Gene regulatory Network visualisation application by Dinne Bosman.

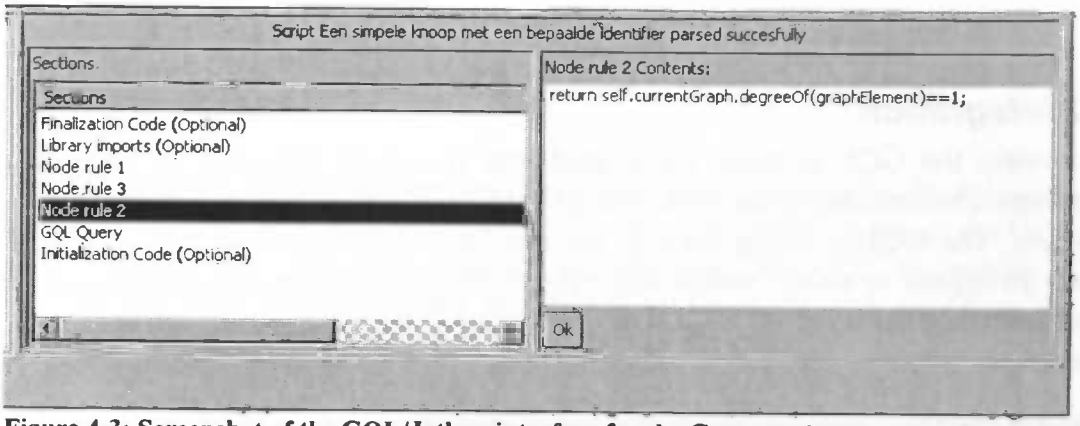


Figure 4-3: Screenshot of the GQL/Jython interface for the Gene regulatory network visualisation application by Dinne Bosman.

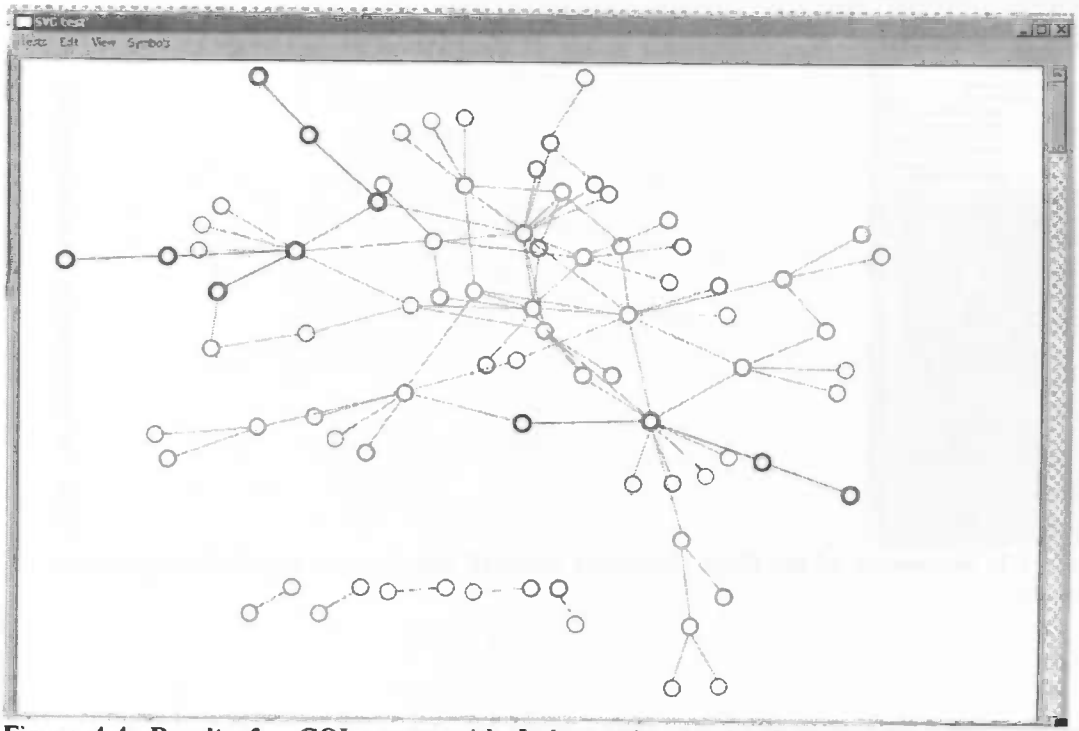


Figure 4-4: Result of a GQL query with Jython rules, in the Gene regulatory network visualisation application by Dinne Bosman.
Nodes and edges represented with darker lines are selected.

It is to early to say if the GQL is completely successful in providing all features we would like to see in a GQL. But the preliminary test show promise.

Chapter 5: Conclusions and future work

5.1 Future work

5.1.1 Split qNodes into separate classes

The qNode has become a very large structure. Some qNode types like the shadow-node use only a few of the data structures that are part of the qNode. This means we haul a lot of data structures and data around for no good reason.

We can build a java class qNode containing the methods and data structures common to all qNode types, and build separate sub-classes of this qNode for each qNode type containing only the data and methods unique to those types.

This would not only reduce the amount of memory used by the program but also make it easer to add new types of qNode.

5.1.2 Add Scripts for graph algorithms

One of these nodes could be the shortest-path-node denoting the shortest possible path between the nodes on either side if that node. This would mean implementing Dijkstra's algorithm[1].

5.1.3 Add possibility for referencing to other nodes

At this point it is not possible for a rule to reference to another node in the query (see section 4.1.2. Some restructuring of the aNode may be required to make this possible. It would make the GQL a more powerful tool and open up the possibility for more database-like functions such as counting of occurrences.

[1] **Dijkstra's algorithm**, named after its inventor, the Dutch computer scientist Edsger Dijkstra, solves the shortest path problem for a directed graph with nonnegative edge weights. For example, if the vertices of the graph represent cities and edge weights represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between two cities.[c]

5.2 Conclusions

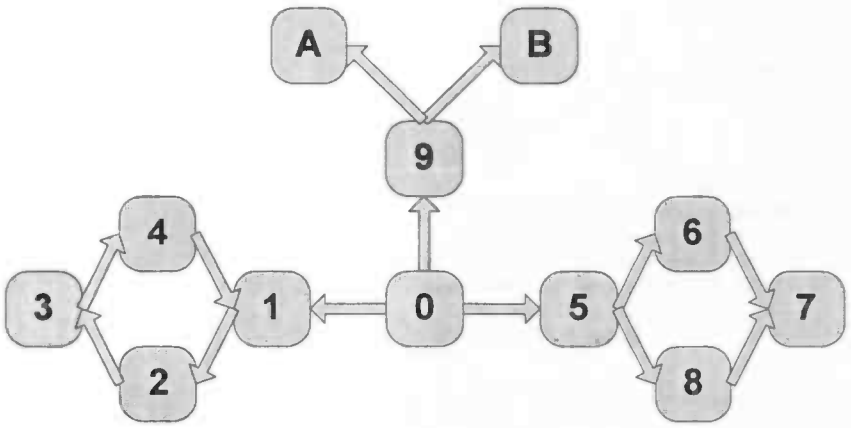
The GQL has become a useful tool and does what is was designed to do. However there are a number of points (listed in the preceding paragraph) on which the GQL could be improved. The main program benefits from the GQL by making it possible for users to select nodes and edges with a query and providing the different modules an easy way for selecting parts of the graph.

Appendix A: A formal language definition for E-GLIDE

```
quer → grap SEMYCOL
                ;
grap → gelm:
                | grap gelm
                ;
gelm → subg
                | path
                ;
subg → LPAREN grap RPAREN
                | LBREAK grap RBREAK sort
                | LBREAK grap RBREAK mmax
                | LBREAK NOSHOW grap RBREAK sort
                | LBREAK NOSHOW grap RBREAK mmax
                | MINUS LBREAK grap RBREAK sort
                | MINUS LBREAK grap RBREAK mmax
                        ;
path → snod:n
                | snod path
                ;
snod → LVEREM node RVEREM
                | edge LVEREM node RVEREM
                | LVEREM NOSHOW node RVEREM
                | edge LVEREM NOSHOW node RVEREM;
                ;
node → sort:s
                | sort ccol
                | sort rule
                | sort ccol rule
                | mmax
                | mmax ccol
                | mmax rule
                | mmax ccol rule
                ;
sort → POINT
                | TIMES
                | QUESTION
                | PLUS
                ;
mmax → LPAREN numb COMMA numb RPAREN
                | LPAREN numb RPAREN
                ;
ccol → cycl
                | cycl COMMA ccol
                ;
cycl → PERCEN numb
                ;
edge → LACCOL numb RACCOL
                ;
rule → numb
                ;
numb → NUMBER
                ;
```

Appendix B: Test results

We used the following directed graph to test the GQL.



0...9 A and B are node-identifiers which the test application prints as answer to queries. A and B stand for numbers 10 and 11 respectively.

The query and their results are presented in tables containing the collection result, which contains all nodes that are part of an answer, and individual results. Individual results are formatted as follows: Each line represents an individual answer to the query, unless it is preceded by '|' this means that we take the previous line and add that which is behind the '|' to create a path.

For example:

```
A
 |12
 |34
```

Means we have found 2 paths `A12` and `A34`.

A number of nodes-identifiers between '(' and ')' construct a branch starting from the node preceding the '(' if a branch is directly followed by another branch the second branch begins at same node the first branch begins. This is basically the same as branches constructed in E-GLIDE.

For node-rules we use the following:

1. Node-identifier is even.
2. Node-identifier is odd.
3. Node has only one edge.

For an edge-rule we use

5. Edge is from left to right.

QUERY	COLLECTION RESULT
[.];	2 4 8 9 6 B 1 3 7 A 5 0
INDIVIDUAL RESULTS	
2 4 9 8 B 6 1 3 A 7 5 0	

Test 1: [.];

QUERY	COLLECTION RESULT
[.1];	2 4 8 6 A 0
INDIVIDUAL RESULTS	
2 4 8 6 A 0	

Test 2: [.1];

QUERY	COLLECTION RESULT
[.1][.2];	4 8 3 7 2 9 6 1 A 5 0 □
INDIVIDUAL RESULTS	
2 |3 |1 4 |1 |3 8 |7 |5 6 |7 |5 A9 0 |9 |5 |1	

Test 3: [.1][.2];

QUERY	COLLECTION RESULT
[.2][.2];	9 B
INDIVIDUAL RESULTS	
9B B9	

Test 4: [.2][.2];

QUERY	COLLECTION RESULT
[.2]{5}[.2];	9 B
INDIVIDUAL RESULTS	
9B	

Test 5: [.2]{5}[.2];

QUERY	COLLECTION RESULT
[.3];	B A
INDIVIDUAL RESULTS	
B A	

Test 6: [.3];

QUERY	COLLECTION RESULT
[(4)]{5}[.3];	4 8 B 2 9 6 1 A 5 0
INDIVIDUAL RESULTS	
2109 |B |A 4109 |B |A 8509 |B |A 6509 |B |A	

Test 7: [(4)]{5}[.3];

QUERY	COLLECTION RESULT
[.%8][*][?%8];	4 8 3 7 2 6 1 5
INDIVIDUAL RESULTS	
2 |3412 |1432 4 |1234 |3214 8 |7658 |5678 6 |7856 |5876 1 |2341 |4321 3 |4123 |2143 7 |8567 |6587 5 |8765 |6785	

Test 8: [.%8][*][?%8];

QUERY	COLLECTION RESULT
[.%8]{5}[*]{5}[?%8];	2 4 1 3
INDIVIDUAL RESULTS	
23412 41234 12341 34123	

Test 9: [.2 %8]{5}[*][?%8];

QUERY	COLLECTION RESULT
[. 2 %8]{5}[*]{5}[?%8];	2 4 1 3
INDIVIDUAL RESULTS	
12341 34123	

Test 10: [.2 %8]{5}[*][? %8];

QUERY	COLLECTION RESULT
[. 2 %8]{5}[*]{5}[? 2 %8];	-
INDIVIDUAL RESULTS	
-	

Test 11: [. 2 %8]{5}[*]{5}[? 2 %8];

QUERY	COLLECTION RESULT
[.]{5}[(2)]([.])[.];	2 4 1 3 0
INDIVIDUAL RESULTS	
341 |(0)2 |(2)0	

Test 12: [.]{5}[(2)]([.])[.];

QUERY	COLLECTION RESULT
[?]{5}[(5,7)];	2 4 1 3 0
INDIVIDUAL RESULTS	
12341. 01234 |1 |.	

Test 13: [?]{5}[(5,7)];

QUERY	COLLECTION RESULT
[.]-<[.2]>.[.];	4 8 3 7 2 9 6 1 A 5 0
INDIVIDUAL RESULTS	
9 |A |0 1 |2 |4 |0 3 |4 |2 7 |8 |6 5 |8 |6 |0	

Test 14: [.]-<[.2]>.[.];

QUERY	COLLECTION RESULT
[.%1][*][.%1][.][.%2][*][.%2];	4 8 3 7 2 6 1 5 0
INDIVIDUAL RESULTS	
2341(2)(05 |8765 |6785) 4321(4)(05 |8765 |6785) 8765(8)(01 |2341 |4321) 6785(6)(01 |2341 |4321)	

Test 15: [.%1][*][.%1][.][.%2][*][.%2];

QUERY	COLLECTION RESULT
<[.]{5}[.]>(3);	4 8 3 7 2 9 6 1 5 0
INDIVIDUAL RESULTS	
23410 |9 |5 4105 |87 |67	

Test 16: <[.]{5}[.]>(3);

Appendix C: Glide Language

This is the glide manual from
http://alpha.dmi.unict.it/~graphgrep/graphgrep1.html

Let us formulate the Glide regular graph expressions using examples. First consider the graph in Fig. 1.

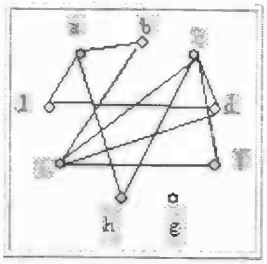


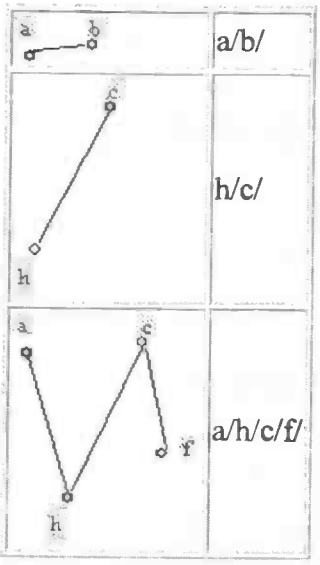
Fig. 1: Generic graph.

a. A *node* is represented by its label followed by the special character '/'

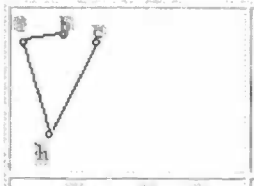
a/

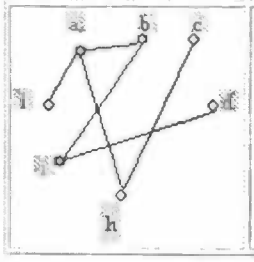
b/

b. An *edge* is represented by two consecutive node representations; a *path* of length n is represented by n+1 consecutive node representations.



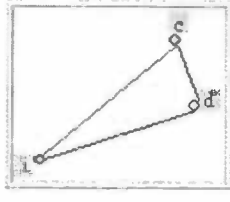
c. *Branches* are grouped using nested parentheses '(' and ')'.



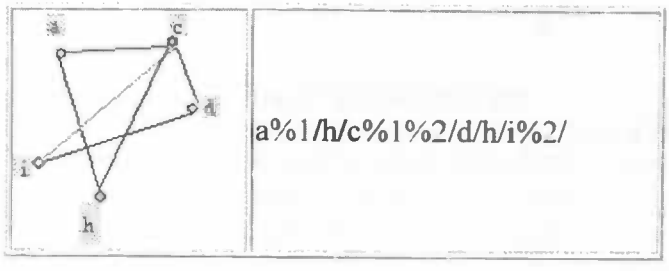


d. *Cycles* are broken by cutting an edge and labeling it with an integer.

1. The labels of the nodes of the cut edge are followed by '%', an integer and '/'.

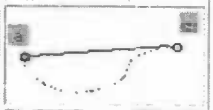
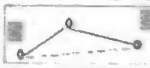


2. The labels of the nodes of the cut edge are followed by a list of '%'and integers. In this case the same node is a vertex of several cut edges.

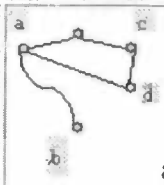


Note: by node label we mean the value used in a matching algorithm to establish the equivalency between two nodes.

Glide uses wildcards for approximate searches. The wildcards are used to match single nodes or paths. Wildcards must be always followed by '/' . The wildcards used by Glide and their semantics are given below.

Wildcard	Semantic (matching with)	Example	
.	any (single) node	a/./c/	
*	zero or more nodes	a/*/c/	
?	zero or one node	a/?/c/	
+	one or more nodes	a/+/c/	

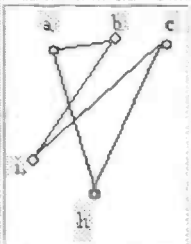
Glide manipulates any combination of wildcards. A recursive semantics is provided (see examples).

Glide expression	Semantic
a/./././b/	any path of length 4 beginning with a node 'a 'and ending with a node 'b'
a/+/./b/	any path of length at least 3 beginning with a node 'a' and ending with a node 'b'
a/?/./	any path of length at least 2 and at most 3 starting with a node 'a'
././a/	any path of length 2 beginning with a node 'a '
a%1/(*/b/)./c/d%1/	 any path where 'a' and 'c' are connected through a node, and a path between a and b exists.

The graph expressions (E's) in Glide are generated by the following grammar, where *digit* is a positive integer and *label* is a string.

E <-- *'label/'* | *'label{%digit}/'* | './' | '*/' | '?/' | '+/'

E <-- EE | '('E')'

Graph	Incorrect graph expression	Correct graph expression
	a/((b/i/c%1)/h/c%1/)	a/(b/i/c%1/)h/c%1/

Rules to construct a well formed Glide graph expression:

- The minimum number of parentheses is used.
- The label of the nodes eventually concatenated with a list of '%' and integers is always followed by '/'.

Example of incorrect graph expressions:

- a/b/((a/b/c/)a/b/)
- a/b
- a/+

In addition, the Glide language can be used to store graphs in a compact way. A depth first search algorithm can be used to generate a Glide representation of a graph. When visiting a node, its label is printed followed by '/' (unless this node is visited during a backtracking step). When the next node to be visited has unvisited siblings a '(' is printed. A ')' is printed when all the siblings have been visited. In a cycle, the first and last node representations are labeled by inserting '%' followed by a unique (for each cycle) integer before '/'.

Acronyms

aNode	Answer Node
BMI	Biomolecular Informatics
CMBI	Centre for Molecular and Bio-molecular Informatics
CUP	Constructor of Useful Parsers
DNA	Deoxyribonucleic acid
E-GLIDE	Extended Graph Linear Description
GIM	Graph Interaction Module
GLIDE	Graph Linear Description
GQE	Graph Query Engine
GQL	Graph Query Language
GRN	Gene Regulatory Network
JFLEX	Jave Free Lexer
LALR	Look-Ahead LR
mRNA	mesenger Ribonucleic acid
NP	non-deterministic polynomial-time
qNode	Question Node
RNA	Ribonucleic acid
RuG	Rijksuniversiteit Groningen (Univerity of Groningen)
YACC	Yet Another Compiler Compiler

References

a Dinne Bosman, Evert-Jan Blom, Patric Ogao; Developing an Automated Gene Network Identification, Modelling, Visualization & Simulation System, Report Institute of Mathematics and Computer Sciences, University of Groningen, 2003.

b http://alpha.dmi.unict.it/~graphgrep/graphgrep1.html

c Wikipedia, the free encyclopedia, http://en.wikipedia.org/

d Aho, Sethi, Ullman; Compilers Principles, Techniques, and Tools, Addison-Wesley, 1986.

e http://www.jflex.de/

f Scott Hudson, Frank Flannery, C. Scott Ananian; http://www.cs.princeton.edu/~appel/modern/java/CUP/

g http://www.cs.nyu.edu/cs/faculty/shasha/papers/graphgrep/

h H.Bakker; Diktaat talen en automaten, Univerity of Groningen, 1999.