ALGORITHMS FOR EFFECTIVE QUERYING OF GRAPH-BASED PATHWAY DATABASES

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ahmet Çetintaş

July, 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Doğrusöz(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. İsmail Hakkı Toroslu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Ali Aydın Selçuk

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

ALGORITHMS FOR EFFECTIVE QUERYING OF GRAPH-BASED PATHWAY DATABASES

Ahmet Çetintaş
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
July, 2007

As the scientific curiosity shifts toward system-level investigation of genomic-scale information, data produced about cellular processes at molecular level has been accumulating with an accelerating rate. Graph-based pathway ontologies and databases have been in wide use for such data. This representation has made it possible to programmatically integrate cellular networks as well as investigating them using the well-understood concepts of graph theory to predict their structural and dynamic properties. In this regard, it is essential to effectively query such integrated large networks to extract the sub-networks of interest with the help of efficient algorithms and software tools.

Towards this goal, we have developed a querying framework along with a number of graph-theoretic algorithms from simple neighborhood queries to shortest paths to feedback loops, applicable to all sorts of graph-based pathway databases from PPIs to metabolic pathways to signaling pathways. These algorithms can also account for compound or nested structures present in the pathway data, and have been implemented within the querying components of Patika (Pathway Analysis Tools for Integration and Knowledge Acquisition) tools and have proven to be useful for answering a number of biologically significant queries for a large graph-based pathway database.

Keywords: Graph Algorithms, Graph Querying, Biological Pathways, Pathway Databases.

ÖZET

ÇİZGE TABANLI YOLAK VERİ TABANLARININ ETKİN SORGULANMASI İÇİN ALGORİTMALAR

Ahmet Çetintaş
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Uğur Doğrusöz
Temmuz, 2007

Bilimsel merak, kalıtım-ölçekli bilginin sistem seviyesindeki araştırmalarına yönelirken, molekül düzeyindeki hücresel süreçler hakkında üretilen veriler hızlanan bir oranla artmaktadır. Çizge tabanlı yolak ontolojiler ve veri tabanları bu tarz bilgiler için geniş bir kullanım alanına sahiptir. Bu gösterim, hücresel ağların programlı bir şekilde bütünleştirilmesinin yanı sıra yapısal ve dinamik özelliklerini tahmin etmeye yönelik olarak çizge teorinin iyi anlaşılmış kavramları kullanılarak araştırılmasını mümkün kılmaktadır. Bu kapsamda, böyle bütünleşik geniş ağların, ilgili alt-ağların etken algoritmalar ve yazılım araçlarının yardımıyla seçilerek çıkarılması amacıyla, etkili olarak sorgulanması zaruridir.

Bu amaçla, protein-protein etkileşiminden, metabolik yolaklara hatta sinyal yolaklarına her türlü çizge tabanlı yolak veri tabanlarına uygulanabilir olmak üzere, basit komşuluk sorgularından en kısa yol yolaklarına ve geri besleme döngülerine pek çok çizge teorik algoritmalar beraberinde bir sorgulama çerçevesi geliştirdik. Bu algoritmalar ayrıca yolak veritabanı içinde mevcut bileşik veya birbirinin içine yerleştirilmiş yapıları da oluşturulabilir ve Patika (Entegrasyon ve Bilgi Kazanma için Yolak Analiz Araçları) yazılımlarının sorgulama unsurları içerisinde uygulanmıştır. Ayrıca, sözkonusu algoritmaların geniş bir çizge tabanlı yolak veritabanı için biyolojik olarak önem arz eden pek çok sorgunun cevaplandırılması için kullanışlı olduğu görülmüştür.

Anahtar sözcükler: Çizge Algoritmaları, Çizge Sorgulama, Biyolojik Yolaklar, Yolak Veri Tabanları.

Acknowledgement

I would like to express my deepest gratitude to my supervisor Assoc. Prof. Uğur Doğrusöz for his invaluable support, continuous encouragement and incredible effort in the supervision of the thesis. It was a wonderful opportunity to work with him.

I would like to thank Prof. Dr. İsmail Hakkı Toroslu and Asst. Prof. Dr. Ali Aydın Selçuk for accepting to read and review the thesis.

I wish to thank Emek Demir and Özgün Babur for their constructive and critical comments. I also wish to thank Çağrı Aksay, Gözde Çözen, Çağatay Bilgin, Hilmi Yıldırım, Hande Küçük and Serhat Tekin for their contributions. I wish to thank all other members of PATIKA team. It was a great experience to be a member of such a friendly team.

Finally, a special thank goes to my wife Öznur for her encouragement, support and patience in every step of my study.

This thesis was supported by TUBITAK (Scientific and Technical Research Council of Turkey) with National Scholarship Programme for MSc Students.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Human genome is expected to create an extremely complex network of information, composed of hundred thousands of different molecules and factors [3, 18]. Knowing the exact map of this network is very important since it will potentially explain the mechanisms of life processes as well as disease conditions. Such knowledge will also serve as a key for further biomedical applications such as development of new drugs and diagnostic approaches. In this regard, a cell can be considered as an inherently complex multi-body system. In order to make useful deductions about such a system, one needs to consider cellular pathways as an interconnected network rather than separate pathways.

Our knowledge about cellular processes is increasing at a rapidly growing pace. Novel large scale analysis methods are already applied to yeast to provide data about the yeast proteome [16, 27]. However, most of the time these data are in fragmented and incomplete forms. One of the most important challenges in bioinformatics is to represent and integrate this type of knowledge. Efficient construction and use of such a knowledge base depends highly on a strong ontology (i.e., a structured semantic encoding of knowledge). This knowledge base then can act as a blueprint for simulations and other analysis methods, enabling

us to understand and predict the behavior of a cell much better.

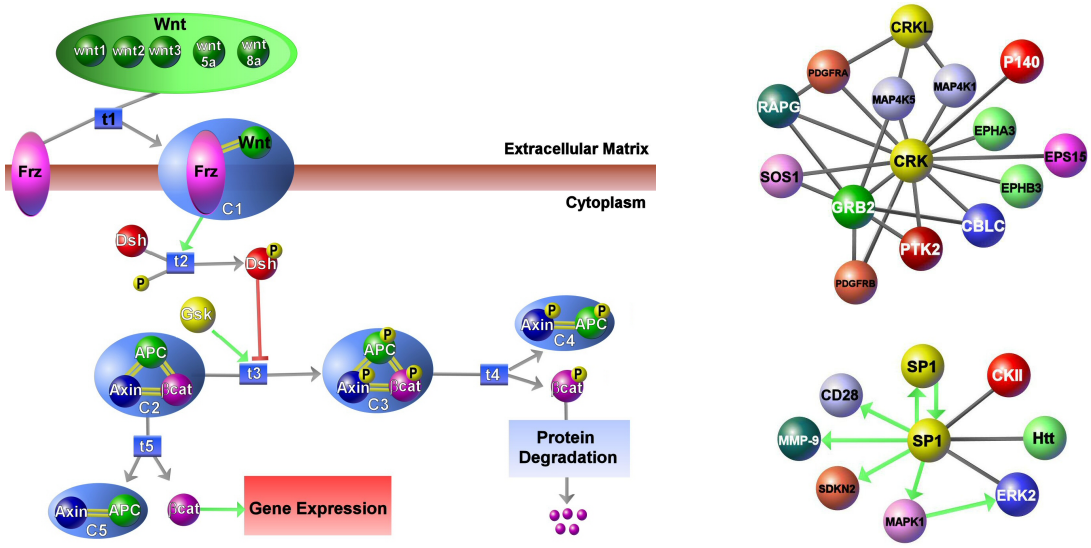

Figure 1.1: Sample pathways represented by PATIKA ontology. **(left)** Canonical wnt pathway containing examples of compound structures such as regular abstractions (e.g. "protein degradation"), homology abstractions (e.g. 5 wnt genes), and molecular complexes (e.g. APC:Axin). **(right)** Partial human interaction networks containing PPIs and transcriptional regulation interactions of proteins CRK and SP1, respectively.

Conventional approaches for representation of cellular pathways are based on pathway drawings composed of still images. Although easy to create, such drawings are often not reusable and the ontologies used are far from being uniform and consistent, highly depending on implicit conventions rather than explicit, formal rules. Recently this has resulted in a major shift in the use of more formal ontologies and dynamic representation of pathways that support programmatic integration and manipulation of pathways regardless of the underlying ontology. Among these, graphs, one of the most common discrete mathematical structures [6], have been most popular for "in-silico" modeling of biological pathways, from metabolic pathways to gene regulatory networks, from protein-protein interaction networks to signaling pathways [2, 10, 20, 15]. Such modeling is crucial for the field of systems biology, which deals with a systems-level understanding of biological networks.

1.2 Contribution

In this thesis, we present a framework for querying a compound graph-based pathway database as well as a number of graph algorithms needed to implement these graph-based queries. We assume a database in which pathways are stored in an integrated manner, as opposed to a list of independent pathways. Thus, a query to the database is performed over this integrated, higher level (Figure 1.2) network of pathways, and aims to find a sub-network of interest, requiring a rich set of graph algorithms.

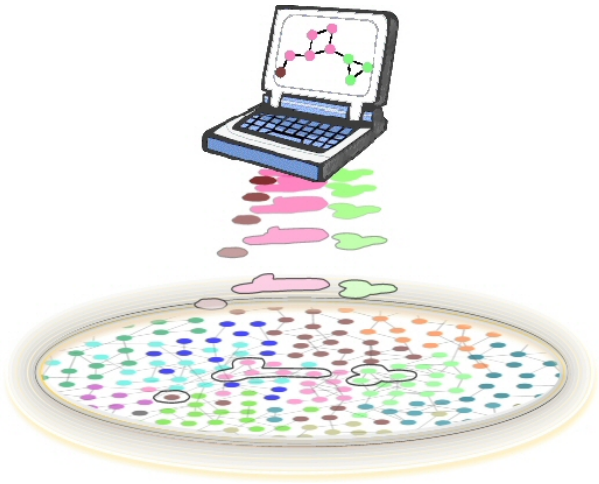

Figure 1.2: Conceptual illustration of how pathways are assumed to be already integrated in the knowledgebase (bottom part, where each pathway is color-coded uniquely), which is typically on disk, and how the sub-network of interest (parts of three different original pathways) may be extracted and presented to the user as a result of a query.

This framework and associated graph algorithms have been implemented as part of a new version of the bioinformatics tool Patika*web* [11]. The range of graph-theoretic queries described as part of our framework is among the most comprehensive ones built so far, and to the best of our knowledge, it is the first querying framework taking into account *compound* structures (i.e. grouping of or abstractions of biological objects to an arbitrary level of depth) in a graph-based knowledgebase.

1.3 Organization of the Thesis

The thesis is organized as follows: Chapter 2 introduces background information on graphs, use of graphs representing pathways and an overlook to the Patika system. Chapter 3 summarizes the studies in the literature on pathway querying and similar graph algorithms on pathway databases. In Chapter 4, the approach and the algorithms that are developed for pathway analysis are described. Chapter 5 includes the software architecture that is implemented in the Patika system. In Chapter 6, execution times of all algorithms implemented are discussed and results are given. Finally, Chapter 7 concludes the thesis.

Chapter 2

Theory and Background

2.1 Preliminary Definitions

Let $G = (V, E)$ be a graph with a non-empty node set V and an edge set E. An edge $e = \{x, y\}$ or simply xy joining nodes x and y is said to be *incident with* both x and y. Node x is called a *neighbor* of y and vice versa.

A path between two nodes n_0 and n_k is a non-empty graph $P = (V, E)$ with $V = \{n_0, n_1, \ldots, n_k\}$ and $E = \{n_0n_1, n_1n_2, \ldots, n_{k-1}n_k\}$, where n_i are all distinct. n_0 and n_k are called the end points of path P. Therefore, we can write P as an ordered set of nodes $P = n_0n_1 \ldots n_k$. The *length* of a path P denoted by $|P|$ is the number of edges on the path.

A path is said to be *directed* if all its ordered edges are directed in the same direction. A directed path P is called an *incoming* (*outgoing*) path of node n if P ends at *target* (starts at *source*) node n.

A directed path is called *positive* (*negative*) if it contains an *even* (*odd*) number of inhibitors (i.e. inhibition edges).

An *incoming* (*outgoing*) path of a node is a directed path ending at (starting with) that node.

Let A and B be set of nodes. Then an A-B path is a path with its ends in A and B, respectively, and no node of P other than its ends is from either set A or B. An A-path is a path where one of the end nodes is from vertex set A, and no other nodes and interactions are from set A.

A path C with identical end nodes is called a cycle.

A directed cycle is called *positive feedback* (*negative feedback*) if it contains an *even* (*odd*) number of inhibitors.

The *distance* between $d_G(x, y)$ between two nodes x and y in graph G is the length of a shortest x-y path in G.

If $G' = (V', E')$ is a subgraph of $G = (V, E)$, and G' contains all the edges $xy \in E$ with $x, y \in V'$, then G' is an *induced subgraph* of G; we say that V' induces G' in G and write $G' = G[V']$.

If node x is the starting node of a directed path that ends up at node y, then node y is said to be a *target* of node x; similarly, node x is said to be a regulator of node y.

2.2 Graph Representation of Pathways

Querying framework and the algorithms described in this thesis have all been designed and implemented assuming the PATIKA ontology [10], which shows utmost similarity to standard representations such as BioPAX [5] and SBGN [22]. However, the results should be able to be applied to most other graph-based pathway representations without difficulty.

PATIKA ontology is based on a qualitative two-level model. At the entity level, interactions and relations can be addressed in an abstract manner, where the exact state of the related parties is unknown, such as protein-protein interactions, inferred relations and literature-derived information. At the state/transition or mechanistic level, each entity is associated with a set of states interacting with

each other via transitions. This level can capture more detailed information such as compartments, molecular complexes and different types of biological events (e.g. covalent modification, transportation and association). This two-level ontology elegantly covers most biological pathway-related phenomena and is capable of integrating information present in the literature and molecular biology databases. Additionally, the PATIKA ontology uses the notion of compound graphs to represent abstractions, which are logical groupings that may be used to handle the complex and incomplete nature of the data. Figure 1.1 shows example pathways drawn at biological entity and state/transition levels.

More formally, a *compound pathway graph* $CG = (G, I)$ [10] is a 2-tuple of a *pathway graph* G and a directed acyclic *inclusion graph* I, where

- $V(G) = V$ is the union of nodes denoting bioentities, states, transitions, molecular complexes, and abstractions of five distinct types: regular, incomplete state, incomplete transition, homology state, and homology transition;
- $E(G)$ is the union of interaction edges of various types (such as PPI edges at bioentity level and activator edges between a state and a transition), some of which are directed;
- $V(I) = V(G)$;
- $E(I)$ is the union of inclusion edges for defining compound structures (molecular complexes and abstractions).

In order for a compound pathway graph $CG = (G, I)$ to comply with the PATIKA ontology, it needs to satisfy certain additional invariants such as regular abstractions cannot have a direct interaction (edge).

2.3 PATIKA Architecture

PATIKA is a project of a multi-component environment for cellular pathway data analysis and collaborative construction, which is in progress since 2000. Different

PATIKA tools for varying purposes with different user interface have been developed. While PATIKA 1.0 and PATIKA*pro* are standalone applications, PATIKA*web* serves as a thin client web application. PATIKA*web* is actually a limited version of PATIKA*pro*. Beta version PATIKA 1.0 beta and PATIKA*web* 2.1 are released softwares and a professional version PATIKA*pro* is currently under development.

PATIKA has a client-server type architecture. Client side component is a visual editor for pathway construction and analysis. Server side maintains the database where *the big picture*(i.e., the main repository for pathway data storage) is stored, and coordinates submission process and user communication.

Pathway editor component of PATIKA is based on Tom Sawyer Software's[1] graph visualization libraries, and implemented using Java technology.

The incorporated graph model is based on PATIKA ontology. The pathway graph has a two layer representation in the architecture, subject graph and view graphs. Each working instance of the editor has exactly one subject graph, which is a related sub-graph of the database. Multiple number of view graphs may be generated as desired based on a single subject graph.

[1] http://www.tomsawyer.com

Chapter 3

Related Work

Graphs have been used in a variety of ways in analyzing cellular networks. In [2] they list three levels of increasing complexity for such analysis, where network topology (global structural properties), interaction patterns (local structural connectivity) and network decomposition (hierarchical functional organization) are addressed at each level, respectively. The representation of such complex networks as graphs has made it possible to investigate the topology and function of these networks using the well-understood concepts of graph theory to predict their structural and dynamic properties or detect special structures or properties in them. In addition, this representation has made the systematic (i.e. programmatic) integration of these complex networks feasible. A comprehensive survey of such prediction, detection and reconstruction methods can be found in [2].

Lately, an extension of this graph representation, namely *hierarchically structured graphs* or simply *compound graphs*, where a node of a biological network may recursively contain or include a sub-network of somehow a logically similar group of biological objects [12, 10]. This extension brings in many benefits in modeling, integration, querying and visualization of biological pathways, most important one being reduction of complexity of large networks through decomposition of the network into distinct components or modules.

There is also a great deal of work on querying of occurrences of sub-structures

(from specified subgraphs to special sub-structures) such as graphlets or motifs in graph-based data, including pathways [2]. Most approaches employ some kind of a graph matching algorithm to find one or all (exact or inexact) instances of the specified subgraph [25, 26, 24]. Yet some others take a comparative approach toward interpreting molecular networks, contrasting and aligning networks of different species and molecular types, and under varying conditions [23].

There exists a heuristic approach in which a merged representation of the two networks being compared is created and then a greedy algorithm is applied for identifying the conserved subnetworks embedded in the merged representation, called a network alignment graph. When there exists a one-to-one correspondence between molecules across the two networks, identifying subnetworks in the alignment is simple, however, in general there may be a complex many-to-many correspondence [23, 19].

Use of graph algorithms such as shortest paths between a specified pair of objects in a graph database has been in use for quite a while [14], and their use in graph-based pathway databases have caught attention in recent years [17, 8, 4]. In [14], they define a query language including an operation for finding shortest paths between two simple objects, which are used as the start and target nodes of the search, respectively. Also query has parameter of *path type* defining a precise structure for the resulting sequence, according to the types of nodes and edges on the path. However, the query language was defined only in fragments, no implementation has been mentioned. Also different path finding and feedback queries are defined for pathway databases in query languages such as PQL [17] and GraphDB [14]. For example, *'Retrieval of a connected graph that includes a set of specified interactions'* and *'Retrieval of cycles that include a set of specified interactions'* queries can be expressed, but cannot be computed in PQL, stated as future work.

Path finding in biological networks has some special problems such as highly connected nodes called ubiques. These ubiquitous nodes lead path finding algorithms to the irrelevant paths. Therefore two approaches can be applied: first approach is filtering out the selection of these highly connected nodes, and second

approach is computing the shortest paths on the weighted metabolic graph where each node is assigned a weight equal to its connectivity in the network [8].

Chapter 4

Method

After a careful requirements analysis, we have come up with the following initial set of graph algorithms that might be of use to people querying cellular pathway databases. This set is, by no means, exhaustive, and may easily be extended.

4.1 Neighborhood

One very simple yet quite powerful operation on graphs is finding the neighbors of a specified source node within a certain distance. k-neighborhood of a node set S can be defined as

$$\mathrm{NB}(S,k) = S \cup \{x \mid x \text{ is on an } S\text{-path } P \wedge |P| \leq k\}$$

Figure 4.1 explains this operation with an example.

k-neighborhood of a node set can be found by running a regular BFS with the seed taken as the specified source node set. All nodes reached within k iterations are in the resulting neighborhood. This operation takes time linear in the number of neighbors plus the total number of edges incident upon these neighbors.

Biological Significance: Neighborhood query relies on the graph-theoretic argument above, but takes a different point of view. It finds out objects that are

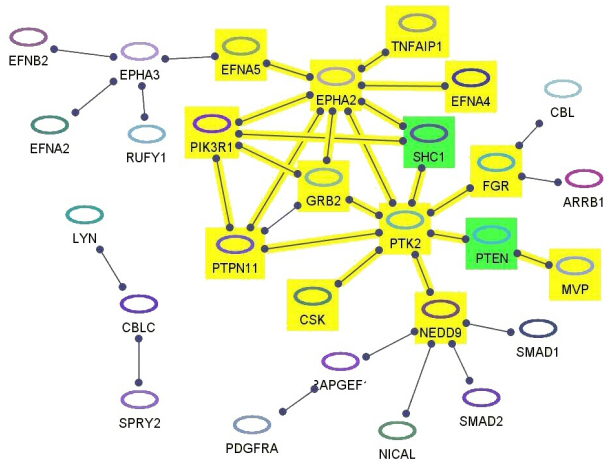

Figure 4.1: 2-neighborhood (yellow) of proteins SCH1 and PTEN (green) in a partial PPI network. Notice that only the edges leading to these neighbors (i.e. visited during traversal) are highlighted.

closest to the given target(s), thus returns a functional neighborhood(as stated in [9]). This query answers questions like:

- In which pathways does my protein take part?
- With which states does this molecule interact?
- What are the other actors taking part in this process?
- Which proteins catalyze this reaction?

4.2 Graph of Interest

Given a graph G and a set of nodes of interest S (e.g. genes of interest), this operation is to find in G all paths of length at most k between any two nodes of the specified node set. The subgraph of G induced by the nodes of the resulting set gives the graph of interest. More formally,

$$\text{GoI}(S,k) = G[B], \text{ where } B = \{x \mid x \text{ is on an } S\text{-}S \text{ path } P \ \wedge \ |P| \leq k\}.$$

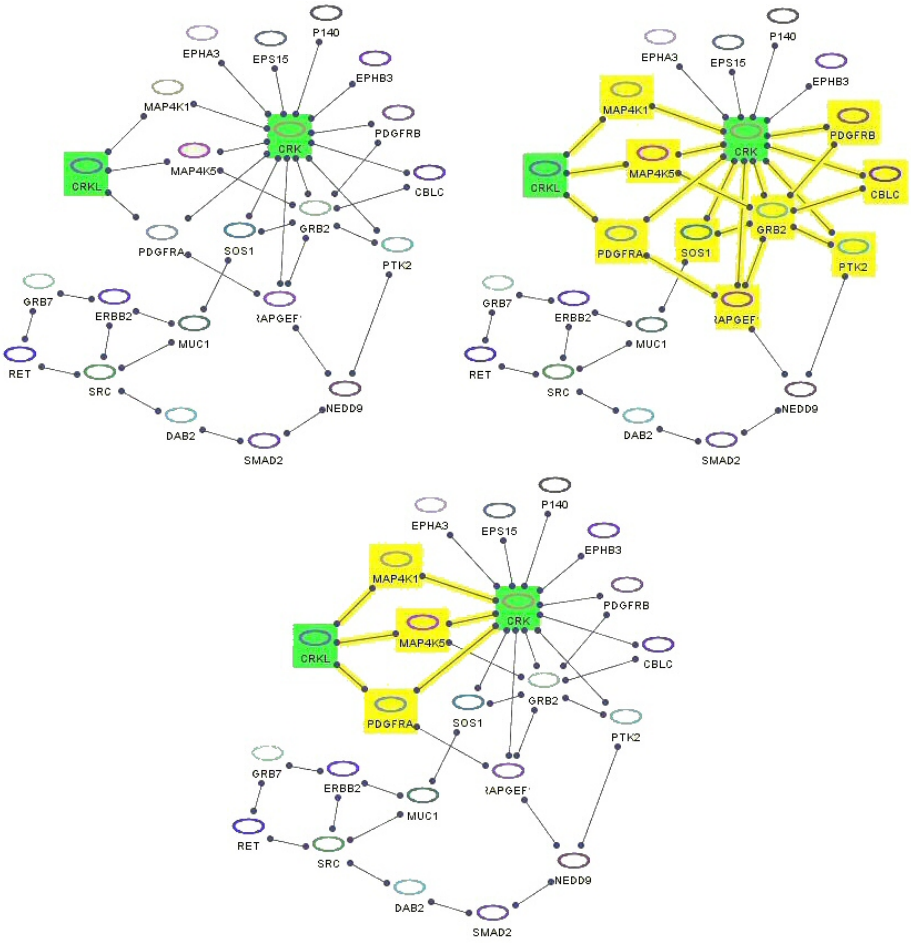

Figure 4.2: **(upper-left)** A PPI network with proteins of interest CRK and CRKL (green); **(upper-right)** Graph of interest (yellow) formed by using paths of length up to 3 ($k = 3$) between nodes of interest (green); **(bottom)** Graph of interest with $k = 2$ ($k = 1$ returns no results).

As the name suggests, this operation is aimed at finding a "minimal" subgraph comprising all the nodes of interest complemented by the "missing links" among these nodes. The parameter k defines how long the paths linking nodes of interest to form a graph of interest is allowed to be. Figure 4.2 explains this operation with an example. Below is the pseudo code for this operation. Here two separate BFS are to be run in forward and reverse directions and combined to form a candidate set. The nodes in this candidate set satisfying the maximum path length constraint are put in a result set, which is to be "purified" by a post-processing phase (algorithm PURIFY) during which degree 1 nodes that do not

lie on paths between source set nodes (effectively subgraphs that are trees in the result set) are pruned.

algorithm GRAPHOFINTEREST(S, k)
1 $C :=$ GOI-BFS(S, k, fwd) $\cup$ GOI-BFS(S, k, rev)
2 **for** $q \in C$ **do**
3 **if** q.LABEL(fwd) + q.LABEL(rev) $\leq k$ **then**
4 $R := R \cup \{q\}$
5 $R :=$ PURIFY(S, R)
6 **return** R

algorithm GOI-BFS(S, k, dir)
1 Add all nodes in set S to queue Q
2 Initialize dir labels of all nodes in S to zero
3 $T := \emptyset$
4 **while** $Q \neq \emptyset$ **do**
5 $n_1 := Q$.DEQUEUE()
6 **for** $e \in n_1$.INCIDENTEDGES() **do**
7 **if** $dir = fwd$ **then**
8 e.LABEL(dir):= n_1.LABEL(dir) +1
9 **else**
10 e.LABEL(dir):= n_1.LABEL(dir)
11 $n_2 := e$. OTHEREND(n_2)
12 $T := T \cup \{e, n_2\}$
13 **if** n_2.LABEL(dir) $> n_1$.LABEL(dir) +1 **then**
14 n_2.LABEL(dir) := n_1.LABEL(dir) +1
15 **if** n_2.LABEL(dir) $< k$ **and** $n_2 \notin S$ **then**
16 Q.ENQUEUE(n_2)
17 **return** T

algorithm PURIFY(S, T, R)

```
1  Remove all orphan edges from R
2  for n ∈ R do
3     while n not in S or T do
4        N := Number of edges of n which are in R
5        if N = 1 then
6           Say e is that only edge
7           Remove n and e from R
8           n := e.getOtherEnd(n)
9        else if N = 0 then
10          Remove n from R
11          break
12       else
13          break
```

The complexity of this operation is clearly linear in the number of nodes in the k-neighborhood of nodes of interest.

Paths-of-Interest (PoI) query, on the other hand, does the same thing *from* a specified set of source molecules *to* a specified set of target molecules. More formally,

$$\text{PoI}(S, T, k) = G[B], \text{ where } B = \{x \mid x \text{ is on an } S\text{-}T \text{ path } P \ \wedge \ |P| \leq k\}.$$

```
algorithm PathsOfInterest(S, T, k)
1  C := GoI-BFS(S, k, fwd) ∪ GoI-BFS(T, k, rev)
2  for q ∈ C do
3     if q.label(fwd) + q.label(rev) ≤ k then
4        R := R ∪ {q}
5  R := purify(S, T, R)
6  return R
```

Biological Importance: Although this query does not attempt to answer a specific question, it allows a quick and easy way for the user the fetch subgraph, that is potentially most interesting for them based on a set of initial nodes. Compared to a neighborhood query, GoI has the specific advantage of filtering out dangling subgraphs that is connected to only one "interest node". GoI is also useful in analyzing microarray data as when given a set of correlated genes, it brings in paths between those genes(as stated in [9]).

4.3 Common Regulation

Common target (regulator) of a source node set S is the set of nodes that are targets (regulators) of *all* nodes in S. More formally, common targets $\mathrm{CT}(S)$ of a source node set S with path length limit k is defined as

$$\mathrm{CT}(S,k) = \{x \mid \forall a \in S \ \ (\exists P \ P \text{ is from } a \text{ to } x \wedge \ |P| \leq k)\}$$

Common regulators $\mathrm{CR}(S,k)$ of a set S can be defined similarly. Figure 4.3 shows an example of this operation. Below is a pseudo code of this algorithm. Input parameter *dir* specifies whether we are asking for targets or regulators, requiring a forward or reverse BFS, respectively. Algorithm CR-BFS simply increases the *reached* count of nodes in the k-neighborhood of seed node n_1, and the nodes reached during such searches are combined in a candidate set. Only the nodes in the candidate set reached from all source nodes are selected to form a result set.

algorithm CommonRegulation(S, k, *dir*)
1 $C := R := \emptyset$ // candidate and results sets, respectively
2 **for** $n_1 \in S$ **do**
3 $C := C \cup$ CR-BFS(n_1, k, *dir*)
4 **for** $n_2 \in C$ **do**
5 **if** $n2$.label(*reached*) $= |S|$ **then**
6 $R := R \cup \{n_2\}$
7 **return** R

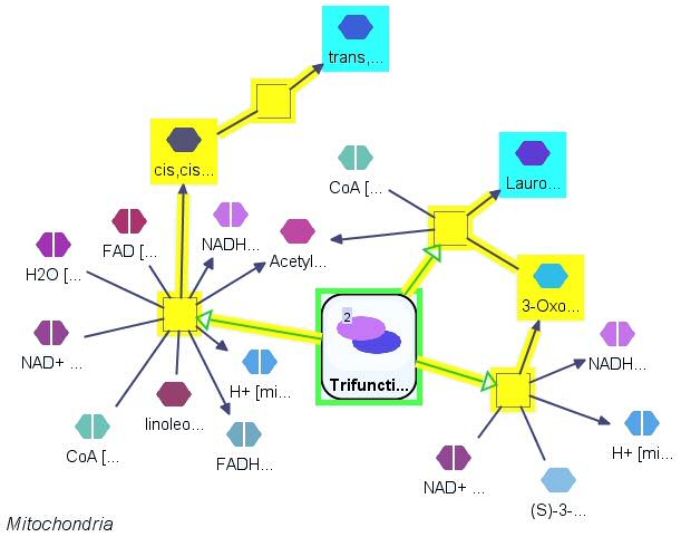

Figure 4.3: Common regulators, with path limit 2, of small molecules containing the word "lauro" in their name (cyan) in this partial mechanistic pathway are found to be a single node representing a molecular complex (green). The paths from the common regulator to the target nodes are highlighted (yellow).

This operation takes $O(|S| \cdot |\mathrm{NB}(S,k)|)$ time as we perform a BFS for each node in S.

In addition, one might require such paths leading to targets or originating from regulators to be positive or negative. For instance, common targets of source nodes S reached by positive paths of length up to k only, denoted by $\mathrm{CT}^{+}(S,k)$, might be of interest. However, we conjecture that complexity of such an operation is asymptotically higher.

Biological Importance: This query becomes important when analyzing correlated events, like microarray expression levels. It finds common regulators/targets, that can possibly explain observed correlation(as stated in [9]). This query answers questions like:

- Why are the expression levels of these two genes correlated?
- Why are the final phenotypes of these two different signals the same?

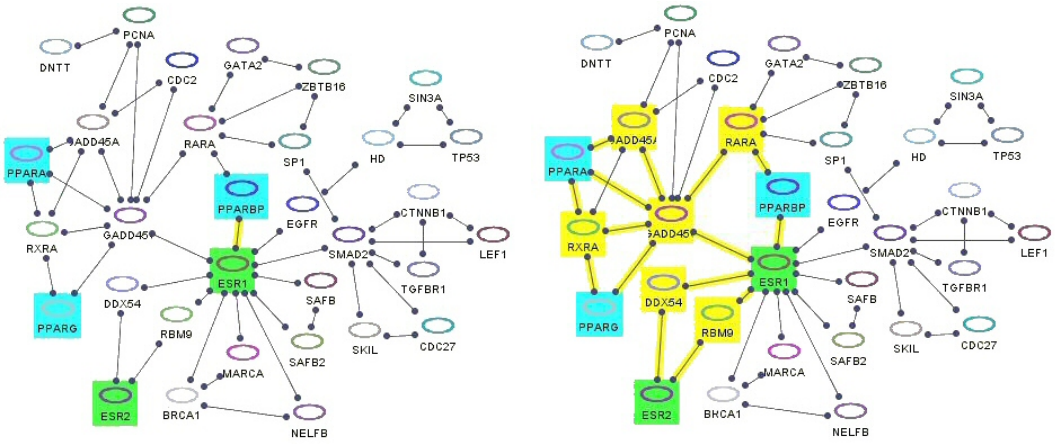

Figure 4.4: Shortest paths (yellow) between bioentities whose names start with "PPA" (cyan) and those whose names contain "ESR" (green) with **(left)** $d = 0$ and **(right)** $d = 2$. Notice that the length of a shortest path between these two node sets is $k = 2$.

4.4 Shortest Path

Given source and target sets S and T, finding shortest S-T paths is a commonly used graph operation [7]. This operation might be constrained by a parameter denoting the maximum length of such paths. In addition, a parameter for "relaxing" the shortest requirement might also be useful. Thus, for instance, the shortest paths between two node sets S and T with maximum length k and further distance d can be defined formally as

$$\mathrm{SP}(S, T, k, d) = \{P \mid P \text{ is an } S - T \text{ path} \wedge$$
$$|P| \leq \min(l + d, k) \ \wedge \ l \text{ is the length of a shortest path}\}$$

Figure 4.4 illustrates this with an example.

Below is the pseudo-code for finding $\mathrm{SP}(S, T, k, d)$, where *mod* specifies whether edges are to be treated directed or undirected.

algorithm ShortestPaths(S, T, k, d, *mod*)
1 **if** $S \cap T \neq \emptyset$ **then**

```
2     return S ∩ T
3  else
4     R := SP-BFS(S, T, k, d, mod)
5     return BuildUpAndEnumPaths(S, T, R)
```

```
algorithm BuildUpAndEnumPaths(S, T, R)
1  for n ∈ R do
2     Construct a new path p for n
3  Add all nodes in set R to queue Q
4  W := ∅
5  while Q ≠ ∅ do
6     n1 := Q.dequeue()
7     for n2 ∈ n1.neighbors(mod) do
8        if n2.label() = n1.label()−1 then
9           W := W ∪ { (n1, n2) }
10          if n2 ∉ W and n2.label()≠ 0 then
11             W := W ∪ {n2}
12             Q.enqueue(n2)
13             if n2 is first neighbor then
14                concatenate n2 to paths of n1
15             else
16                clone all paths of n1 and add to paths of n2
17             Update path list of n2
18             Update sign of paths of n2 with edge (n1, n2)
19 return W, paths
```

Here, SP-BFS runs a breadth-first search starting with nodes in set S in provided *mod*, up to maximum depth k and returns the reached nodes in T. The complexity of the provided version of the algorithm is $O(l + |\mathrm{NB}(S, k)|)$, where l is the total length of the paths enumerated. Notice that the above algorithm *enumerates* all shortest paths. If it suffices to find all the nodes and edges on such paths, rather than listing individual paths, BuildUpAndEnumPaths may be simplified.

In the context of pathways, one might be interested in positive ($\mathrm{SP}^+(S,T,k,d)$) or negative shortest A-B paths in a given pathway graph.

Another type of operation that might be useful is finding first k shortest paths between specified node sets. More formally

$$\begin{gathered}\text{k-SP}^+(A,B,k) = \{P_1, P_2, \cdots, P_k\} \text{ where} \\ P_i, i = 1, \cdots, k, \text{ is a positive } A-B \text{ path and} \\ \textstyle\sum_{i=1}^{k} |P_i| \text{ is minimum over all path sets of size } k\end{gathered}$$

Notice that this path set is not necessarily unique.

Biological Significance: It is commonly accepted that graph theoretic distance of two nodes is correlated with their functional distance. This argument is a long one and is beyond the scope of this document. But to put it simply it has three basis(as stated in [9]):

- In a small-world graph evolved with node duplication events (most biological networks, including reaction networks fall into this category), graph theoretic distance correlates with evolutionary distance.
- Shorter the graph theoretic distance between two nodes, more likely that they are co-regulated, because there are less (control) reactions between them.
- Evolutionarily a very long path with many redundant intermediates should be suboptimal. Intermediates that do not perform control and amplification of the signal, are simply unnecessary vulnerable spots reducing the robustness of the system.

Assuming the above statement is true, and then this query answers the following questions:

- What are the possible route(s) that this protein governs this process?
- How are pathway A and pathway B linked?

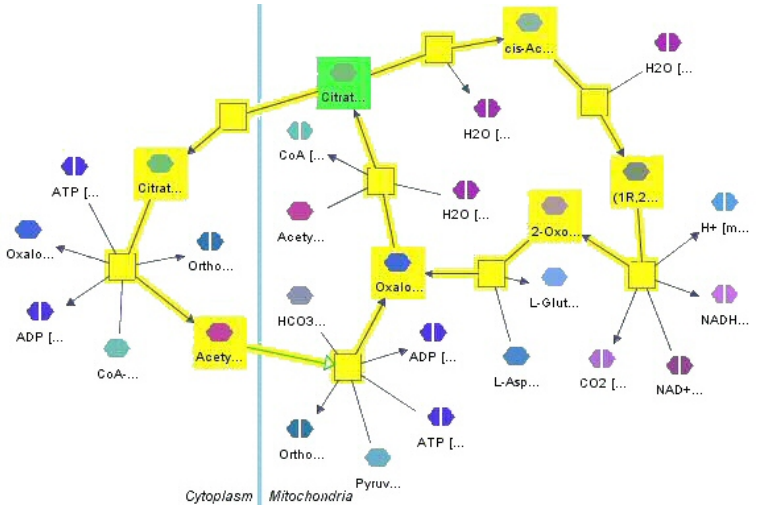

Figure 4.5: Positive feedback (yellow) of a specified Citrate state in mitochondria (green) with up to length 10; the result contains two feedback cycles, one in mitochondria (of length 10) and one through cytoplasm (of length 8).

- What is the most possible route for this signal to be transmitted to the nucleus?

4.5 Feedback

This operation results in a list of positive or negative cycles that contain a specified node. For instance, positive feedback of a node a with maximum length k is defined as

$$\mathrm{FB}^{+}(a,k) = \{C \mid C \text{ is a positive cycle } \wedge\ a \text{ is on } C\ \wedge\ |C| \leq k\}$$

Figure 4.5 illustrates this with an example.

Our algorithm is based on generating all cycles starting from a given set of source nodes in a directed graph as described in [21]. These cycles can be generated by a depth-first search, in which edges are added to a path until a cycle is produced. If a cycle is found, maximum length is exceeded or a dead-end is reached the algorithm backtracks and continues with the next possible edge.

The algorithm picks a node from the source set as the source (s) and builds

a directed path $sn_1n_2n_3 \cdots n_k$. A cycle is found if the next vertex n_{k+1} equals s. After generating this cycle the next edge going out of n_k is explored. If all edges going out from n_k have been explored, the algorithm backs up to the previous vertex n_{k-1} and continues. This process continues until we try to back up past the source node s. At that point all cycles involving s have been discovered, so s can be removed from the graph and the process can be repeated until the source set becomes empty.

To prevent traversing cycles originating at a vertex n_i during the search rooted at s, all vertices on the current path are marked as "unavailable" as extensions of that path. For this, we maintain a flag, which is set to false as soon as n is appended to the current path. That node will remain unavailable until we have backed up past n to its previous vertex on the graph. If the current path up to n did not lead to a cycle, it will remain unavailable even if we back up past it. This prevents redundant dead-end searches. Vertex n will, however, be marked available if a cycle could not be found due to cycle length limit since it is possible for a shorter path to form a cycle by going through n.

```
algorithm FEEDBACK(S, k, desiredSign)
1  path := Empty Stack
2  cycleList := Empty List
3  for n_1 ∈ S do
4     Reset all node flags except REMOVED flag
5     sign := +1 // initially zero inhibitors (even)
6     n_1.INPROCESS := true // finds all cycles s is involved with
7     CYCLE(n_1, 1, sign)
8     n_1.INPROCESS := false // process finished
9     n_1.REMOVED := true
```

```
algorithm CYCLE(n_1, currLength, sign)
1  flag := false
```

```
2  path.push(n1)
3  n1.AVAILABLE := false
4  if currlength < k then
5     for n2 ∈ n1.NEIGHBORS with n2.REMOVED = false do
6        if n2 is an inhibitor then
7           sign := -sign
8        if n2.INPROCESS then // found a cycle
9           if sign = desiredSign then // check if its sign is as desired
10             currCycle := Create list using path
11             currCycle.ADD(w)
12             cycleList.ADD(currCycle)
13          flag := true // found a cycle containing v
14       else if w.AVAILABLE then
15          g := CYCLE(n2, currlength + 1, sign)
16          flag := flag or g
17 else if There are edges going out of n1 then
18    flag := true
19 if flag then
20    UNMARK(n1)
21 else
22    for n2 ∈ n1.NEIGHBORS with n2.REMOVED = false do
23       n2.UnavailablePredecessors.ADD(v)
24 path.POP() // backtrack
25 return flag
```

```
algorithm UNMARK(n1)
1  n1.AVAILABLE := true
2  for n2 ∈ n1.UnavailablePredecessors
3     n1.UnavailablePredecessors.REMOVE(n2)
4     if !n2.AVAILABLE
5        UNMARK(n2)
```

Similar to the basis operation given in [21], this algorithm is of $O(|\mathrm{NB}(S,k)| \cdot (c+1))$ time complexity, where c is the total number of positive or negative cycles generated.

Biological Significance: Feedback loops are an important apparatus used by cellular networks. They can have signal amplifying or stabilizing roles. This query answers questions like(as stated in [9]):

- How is the concentration of this molecule stabilized?
- How does this signal gets amplified?

4.6 Stream

k *upstream* (*downstream*) of a node a is composed of the nodes on the incoming (outgoing) paths to a with length at most k. Positive (negative) upstream of a node a is composed of the nodes on the incoming path that activates (inhibits) (in case of a mechanistic pathway, the preceding transition of) node a. Thus, for instance, k positive upstream of a node a can be formally described as

$$\text{ST-up}^{+}(a,k) = \{x \mid x \text{ is on a positive incoming path } P \text{ of } a \wedge\ |P| \leq k\}$$

A node b might be in both the positive and negative up or downstream of another node a, making those streams (or associated positive and negative paths) ambiguous. Those nodes in the upstream (downstream) of a node a that lead to (reached from) node a with *only* positive paths form the *unambiguous* positive upstream (downstream) of node a. Figure 4.6 illustrates this with examples.

The algorithm performs a brute-force traversing of all the nodes in the k-neighborhood of the source node. It is based on a depth-first search, however, after the recursive processing of a node finishes, that node is marked as unvisited

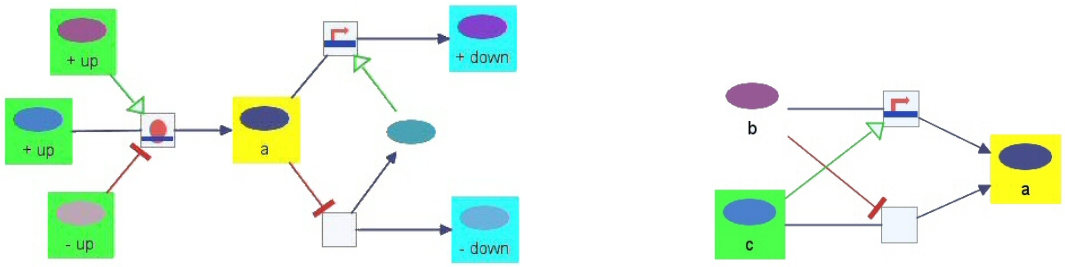

Figure 4.6: **(left)** Up (green) and down (cyan) stream of protein "a" (yellow) in a partial mechanistic pathway. **(right)** Unambiguous positive upstream of node "a" (yellow) contains node "b" (green) only, as node "c" is on both positive and negative paths leading to node "a".

again, potentially leading to multiple visits of nodes and edges. More specifically, every node and edge is processed as many times as the number of different ways they can be reached from source node. In other words, every possible path with length limit from the source node is examined to determine whether it makes a suitable stream or not. Below is the pseudo code for finding positive or negative up or downstream of a desired node with a limited specified distance.

algorithm STREAM(v, $currLength$, $maxLength$, $currSign$, $desiredSign$, dir)

```
1  v.AVAILABLE := false
2  if currLength < maxLength then
3     for w ∈ v.NEIGHBORS(dir) do
4        if w is an inhibitor then
5           currSign := −currSign
6        if currSign = desiredSign then
7           R := R ∪ {w}
8        else
9           A := A ∪ {w}
10       if w.AVAILABLE = true then // prevents infinite loop
11          STREAM(v, currLength + 1, maxLength,
                currSign, desiredSign, dir)
12 v.AVAILABLE := true
13 return R, A
```

Naturally, the time complexity of this algorithm is exponential in the number of nodes and edges in the k-neighborhood of the source node in the worst case. Our experiments show, however, execution time should be acceptable for most interactive applications for small values of k (say up to 10).

Biological Significance: Analyzing upstream and downstream nodes of a molecule is important to be able to retrieve cause/effect relationships, which are critical in diagnosis or drug design. This query answers questions like(as stated in [9]):

- What activated this protein?
- Which processes are affected if this gene is knocked down?
- What are the downstream effects of this drug?

Also in the unambiguous stream case, we require that there are no routes with conflicting effects between the source and target, i.e. all paths are of the same sign. This way we can say unambiguously that to our best knowledge source affects the target in this manner.

All query operations described earlier, make use of traversals over a set of pathway objects of interest. Traversal over pathway objects represented with some sort of a compound structure (e.g. regular abstractions or molecular complexes) calls for a special mechanism as there might be some kind of an equivalence relation between the compound and its members.

4.7 Subgraph Matching

Roughly, a graph $G_1 = (V_1, E_1)$ is said to be isomorphic to another graph $G_2 = (V_2, E_2)$ if one can define a mapping between V_1 and V_2 such that neighborhood relations of each node in V_1 is exactly the same as those of the corresponding node in V_2. *Exact graph matching* problem is given a data graph G (e.g. pathway

knowledgebase) and a model graph H (e.g. pathway to be searched), finding a subgraph or subgraphs of G that are isomorphic to H.

In cases no isomorphism is expected between a data graph G and a model graph H, one might be interested in finding the best matching between them (or H and a subgraph of G), leading to a class of problems known as inexact graph matching. In that case, the matching aims at finding a non-bijective correspondence between H and $G' \subset G$.

Even though most flavors of the graph matching problems are NP-complete [13, 1], there has been a long history of research, mostly focusing on exact subgraph matching. Approximations and restricted versions of the problem have been attacked using various alignment techniques allowing node mismatches and gaps [25, 26, 24]. However, all such methods have suffered from efficiency, which was partially compensated by various indexing techniques.

4.8 Compound Structures & Ubiquitous Entities

We handle this using some traversal options and decide how the traversal should continue when it reaches a compound structure or a member of the compound structure. The list of options can be characterized into two:

- *Link a compound structure and its members:* For instance, should reaching a homology state be interpreted as reaching its members as well (the genes that are homologous) and vice versa.
- *Link members of a compound structure:* For instance, should reaching a member of a molecular complex be interpreted as reaching all members of this complex (thus the traversal should be able to continue from other members as in Figure 4.7).

Remember that we assume six distinct types of compound structures: five kinds of abstractions and molecular complexes as defined earlier. For some of these structures, a user-customized option might not be necessary. For instance, reaching a member of a regular abstraction should rarely be interpreted as reaching all the other members of that abstraction.

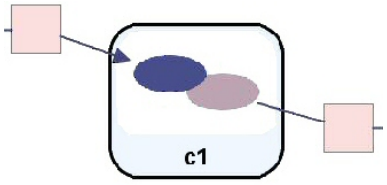

Figure 4.7: Traversal reaching complex "c1" from the transition on the left will continue to transition on the right if and only if "Link Members of Complex" option is true.

Another type of a biological entity that requires special attention is *ubiquitous* or simply *ubique* molecules; that is states with very high degree. For instance a ubiquitous molecule such as ATP might be involved in potentially hundreds if not thousands of reactions at mechanistic level. So, one might prefer not to link two reactions whose only common actors are these kinds of molecules. Therefore traversal over ubiquitous molecules can also be controlled by user-customizable options. Figure 4.8 explains this with an example. Modification to earlier algorithms for handling ubique molecules is similar to the mechanism described for compound structures.

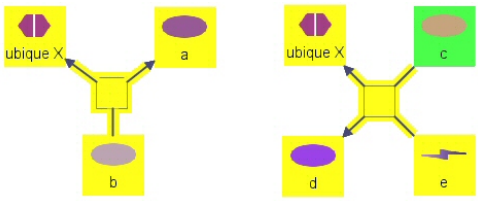

Figure 4.8: Whether or not state "a" is in the 4-neighborhood (green) of state "c" (yellow) depends on whether traversal over ubiques ("ubique x" in this case) is allowed. Obviously, in this case it is allowed.

Chapter 5

Implementation

5.1 Patika

This section briefly describes the implementation of Patika ontology.

5.1.1 Model Layer

Model layer defines first class objects as interfaces, allowing a greater flexibility for its implementors. We assume that the reader already has an acquaintance with the ontology so we do not further explain its concepts, unless an implementation specific explanation is required.

An overview graph of first class objects are given in Figure 5.1. Since abstractions are cross-cutting concerns they were implemented with multiple inheritance.

5.1.2 Concrete Implementations

There are three concrete model implementations, DB (Database) level, S (Subject) level and V (View) level.

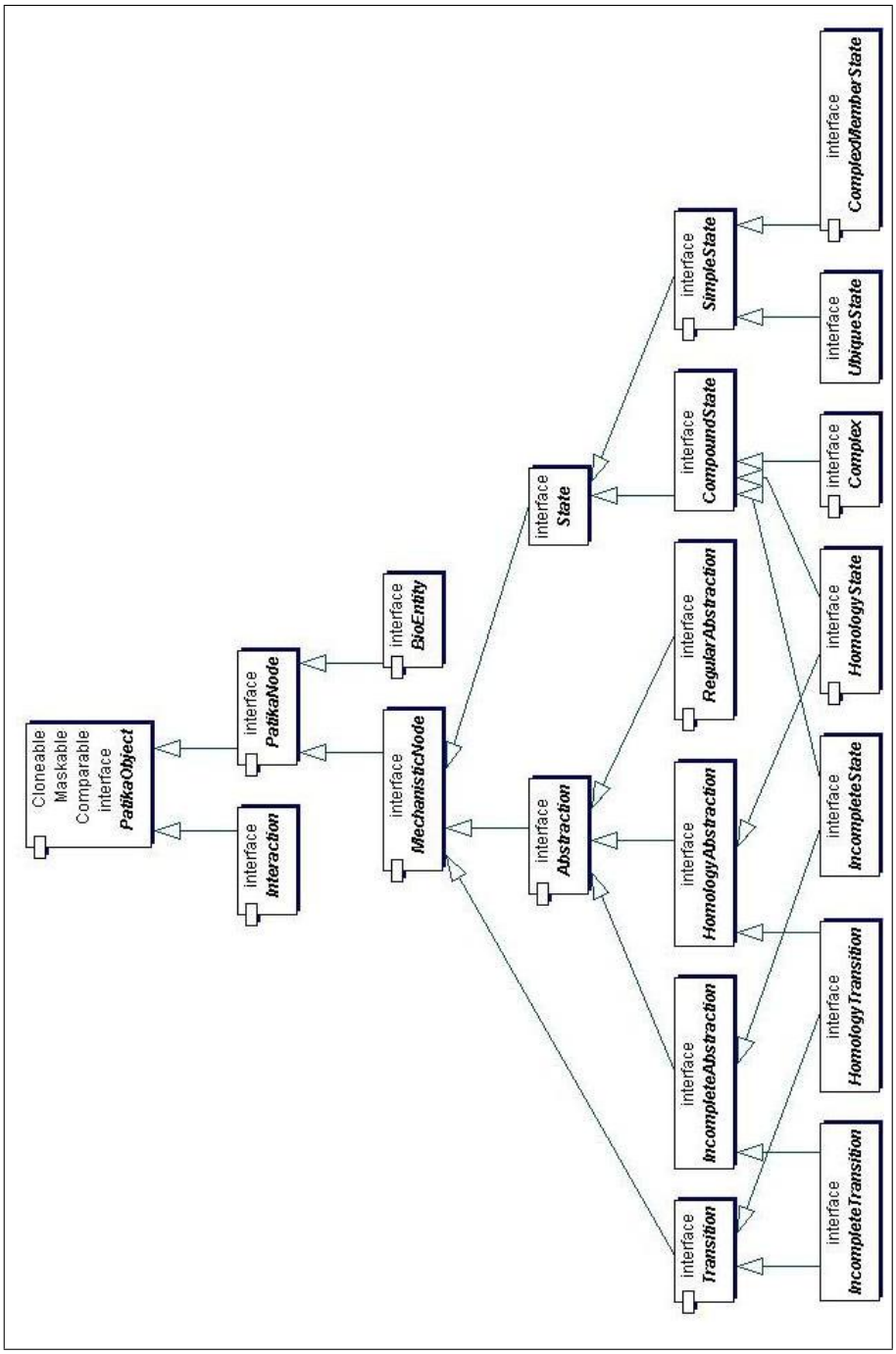

Figure 5.1: Class hierarchy of the primary PATIKA objects.

DB-Level: The server side employs an MVC framework and DB-level acts as the model layer, providing the PATIKA Model interface which is used for manipulating data. DB level is also a DAO(Data Access Object) layer hiding persistence related details from the user, and provides the same consistent PATIKA Model interface. The DB level relies on an in house graph implementation and provides persistence and querying logic.

S-Level: The S-level relies on Tom Sawyer Software's[1] graph libraries for defining an abstract PATIKA graph. S-level is a model layer and contains only topology of the graph and the related data. In a sense, S level acts as a *cached* subgraph of the database and a temporary storage for user created objects and user modifications.

V-Level: V-level defines a compound graph which again relies on Tom Sawyer Software's graph libraries. V-level is a view layer and contains all the drawing information. However each manipulation that is made to the model is delegated to the S layer, which in turn updates views accordingly. V-level provides extra facilities for managing the visualization such as collapsing compound nodes, fetching and merging more objects from the subject graph and laying out external data on them such as expression levels from a microarray experiment.

5.1.3 PATIKA Graphs and Excision Support

A `PatikaGraph` is a set of `PatikaNode`'s and `Interaction`'s. The interface PatikaGraph is implemented in all model layers as DBPatikaGraph, SPatikaGraph and VPatikaGraph. At each layer the PatikaGraph contains same layered PatikaNodes and Interactions. In our implementation, field queries are direct methods of PatikaGraph, and all the queries and query results uses PatikaGraph effectively in all phases of execution and visualization.

Not all subgraphs of a PATIKA graph is valid. Some PATIKA objects depend on other objects for being valid, the latter being called a prerequisite of the former.

[1] http://www.tomsawyer.com

Some example dependency relations are:

- All interactions must have their sources and targets in the view, and if both its source and target is in the view, so must the interaction.
- Each transition must have all of its substrates and products in the view. Although effectors are optional. A transition with missing substrates and products is wrong, in the sense that it clearly violated chemical paradigm. On the other hand leaving out effectors makes it simply *partial*.
- All states must have their bioentity in the graph.
- All complexes must have their complexes member states in the graph and *vice versa*.
- All abstractions must have their members in the graph. The reverse does not hold.

All Patika objects know and can provide a list of their prerequisites.

5.1.4 Patika Server Architecture

Patika software contains a server side component which provides Web services for persistence, querying and integration and two clients for serving different use cases. All clients talk with the server using the same XML based interface over HTTP. Patika*pro* is the heavy client, which is a Java application aimed at users whose primary use case is to edit or extensively analyze the data. On the other hand, Patika*web* is targeted for users who are more interested in read-only access to the database for rapid knowledge-acquisition.

Postgresql is used for main database. It is the most advanced and second most popular open source relational DBMS. It is considered as a slow database, but is ACID compliant and quite stable. It is quite isolated from the system by hibernate layer, so we could switch it by simply changing configuration files anytime.

Underlying container is Tomcat[2], although server can also be configured to run on a J2EE server and a JTA data source, if a clustered environment is needed.

Hibernate[3] is the current ORM (Object to Relational Mapping) tool. Hibernate is a powerful, high performance object/relational persistence and query service for Java. Hibernate allows developing persistent objects using plain old Java classes and relations - including association, inheritance, polymorphism, composition and the Java collections framework.

Spring[4] is a layered Java/J2EE framework, providing several commonly occurring structures in J2EE servers. Spring framework is used for three things: Implementing the IoC pattern for a modular server design, a flexible MVC and managing and isolating Hibernate.

5.1.5 Clients

PATIKA has two different clients, PATIKA*web* and PATIKA*pro*. Overall architecture of these clients are fairly similar, the only difference in their *modus operandi* is that PATIKA*web* has a three-tier architecture, where the most editor operations are done on the so-called *bridge*, whereas PATIKA*pro* sports a plain two-tier architecture, where all editor operations are done a heavy Java application. Mostly due to keep the client thin and high performance, PATIKA*web* provides only read-access to the database and does not allow users to modify queried models.

5.2 PATIKA Query Framework

This section explains the implementation details of PATIKA Query Framework. Firstly the structures used to hold the query data are mentioned, then the execution scenario is described over the query components. Lastly, the Query User

[2]http://www.jakarta.org/tomcat
[3]http://www.hibernate.org
[4]http://www.springframework.org

Interface is illustrated.

5.2.1 Query Structures

There is a separate query object for each type of query, which collects the necessary information and executes the required query. The general structure of query objects is stored in `Query` interface and all query objects implement this interface. There are also `QueryResult` objects which store the results of executions. Execution of the queries depends on the type of the query. Algorithmic queries simply run the corresponding algorithm. Field queries use the associated method directly, implemented in `PatikaGraph` (See Field Queries Document). The networking between client and server for these queries is done via XML files.

5.2.1.1 Query Types

Field Query

These queries are the simplest queries that ask only the object with given field information. Field queries are composed of clauses and conditions. Clauses are the structures in which conditions and clauses are conjunct with ORs and ANDs, using a composite pattern. There are several kinds of conditions.

- *String condition* in which it is checked whether a field is equal to the specified string
- *Integer condition* in which it is checked whether a field is equal to the specified integer
- *Object condition* in which it is checked whether a field is equal to the specified object. These conditions are not directly created directly by the user, but it is required to check an object is equal to the result of another query, like joins in database queries.

```
from ComplexMemberState where BioEntity =
  { from BioEntity where Name = ? }
```

The above query written in `PatikaFieldQuery` language is an example to an object condition in which a string condition is used as an object. This query should get `ComplexMemberStates` of which have BioEntity's naming smt.

- *List condition* in which it is checked whether a field which is a list of something (integer, string or object) has any specified query. List conditions are as object conditions have at least one condition inside.

  ```
  from BioEntity where Names has any ?
  ```

  The above query is an example to a list condition which consists a string condition inside. Bioentities has a an array of Names which are simple strings, a bioentity is chosen if it has any value equal to the specified string in its Names string.

  ```
  from Complex where MemberStates has any
  { from ComplexMemberState where BioEntity =
  { from BioEntity where Names has any ? } }
  ```

  The above query is a list condition query and if one of its `MemberStates` is equal to the inner condition (an object condition consist of a list condition of a string condition), it is chosen. Figure 5.2.1.1 summarizes the aforementioned classes.

These strings are then parsed and transformed into several field query objects. A `FieldQueryParser` object takes a string and then parses it producing an `AbstractSyntaxNode` instance (or rather an instance of one of its subclass) which further may include more clauses and conditions by composition. State diagrams in Figures 5.3 and 5.4 depicts the details of field query parsing.

Field queries are interpreted differently at server and client side. This is achieved by making polymorphing calls to `PatikaGraph` interface. At the client

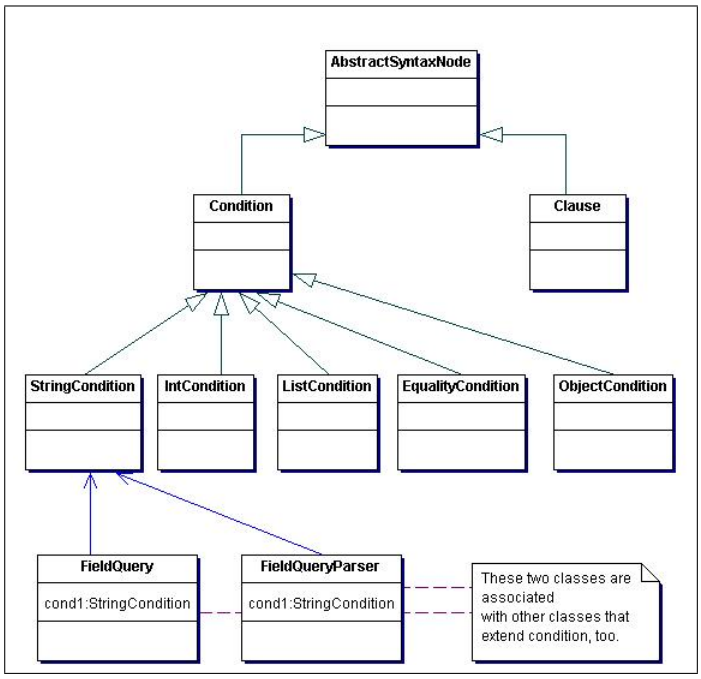

Figure 5.2: The class diagram of field query nodes. A composite pattern was used for arbitrary nesting of query objects.

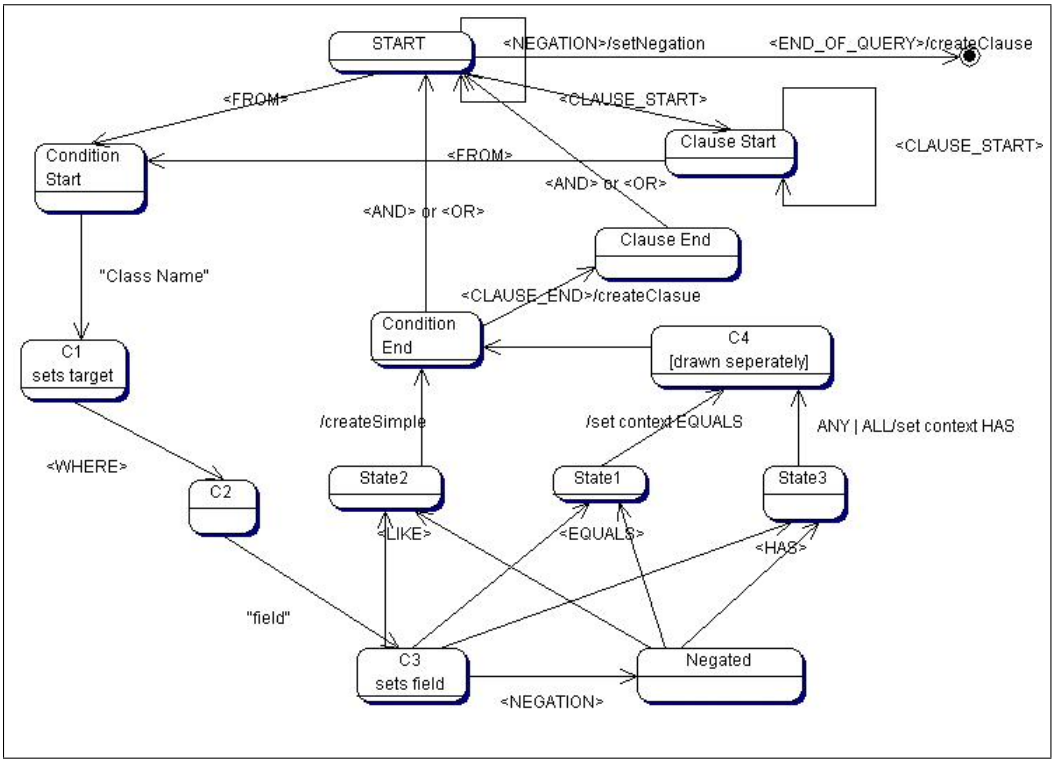

Figure 5.3: General state diagram of `FieldQueryParser`, for parsing the PATIKA query languages field queries.

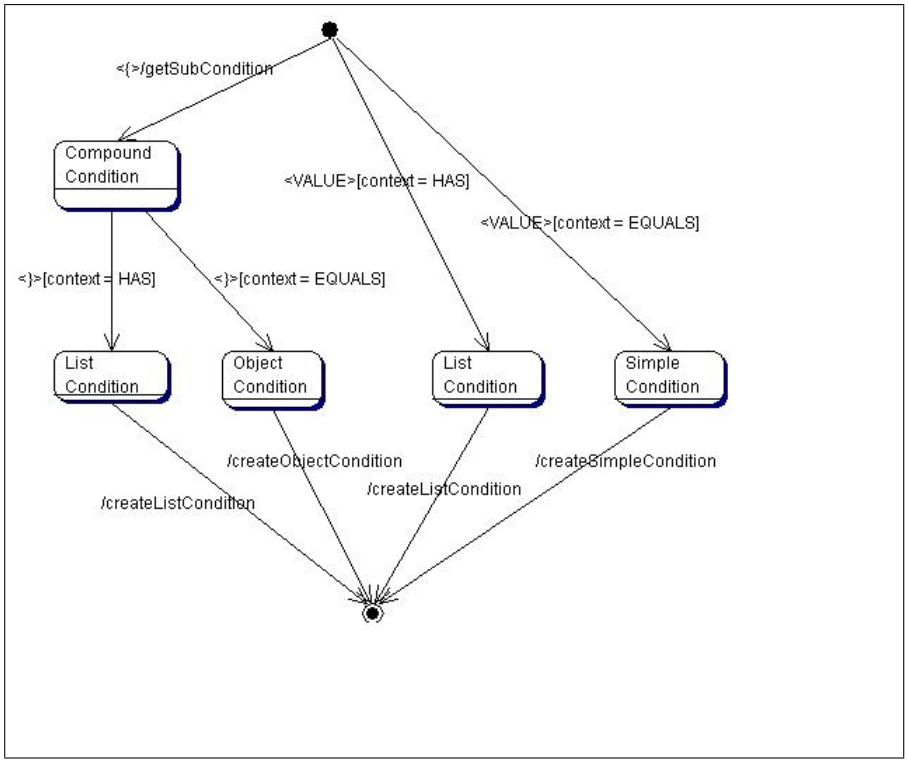

Figure 5.4: State diagram of the `FieldQueryParser`, for deciding on which condition to create. Through composite conditions it is possible to specify arbitrarily nested object relations.

side iteratively all of the PATIKA objects should be sent to the evaluate method of this `AbstractSyntaxNode` one by one, and the list of the ones which return true, should be returned as the result of the query. A visitor pattern was implemented to achieve polymorphism between different S-level objects. Client side does not do anything for the performance, all queries have O(n) time complexity and there is no query optimizer. On the server side interpretation is even more simple. Since our PATIKA Field Query Language is similar but not equal to the Hibernate Query Language (HQL), conversion from our language to HQL can be done with little effort. An `AbstractSyntaxNode` object has an `SynthesizeHibernateQuery` method, which can do this conversion, then only remains running of this query via hibernate query, with its full performance benefits.

Algorithmic (Pathway) queries

These types of queries include mostly the graph theoretic queries and the queries that ask about the pathway information. Examples include shortest path, neighborhood, and common regulation. PATIKA query system defines different graph theoretic queries for different biological problems.

Logical queries

These queries allow performing negation, union and intersection operations on other query results. `AND` query operates as a intersection while `OR` query has a union meaning.

5.2.1.2 Query Tree

Each query has a tree structure. The set parameters of a query are given by another query, and therefore these subqueries become the child queries of root query. The leaves of the query tree is always Field queries. Since the result of all queries are a set of `PatikaObject`s every query can be a child query. For example, a ShortestPathQuery has a source set and a target set as input parameters. These sets are given as different Field queries. During execution, ShortestPathQuery has to wait for the results of these child Field queries.

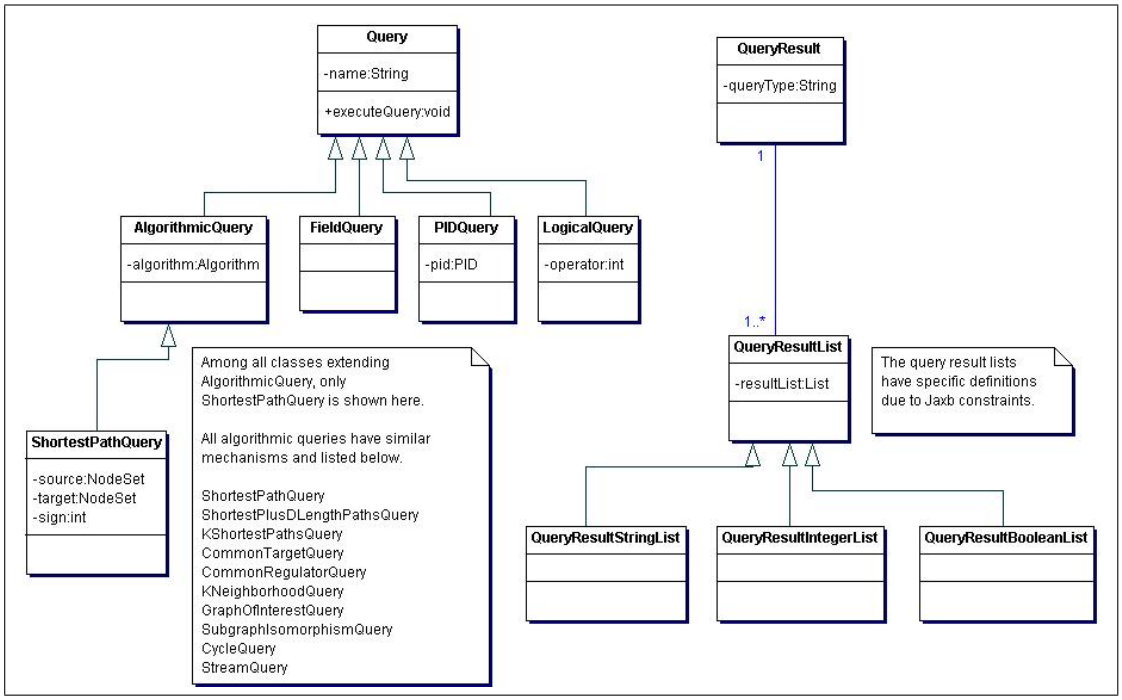

Figure 5.5: Query and Query Result Hierarchy

5.2.1.3 Classes

Query Classes

Each query class implements the `Query` interface. The XML file that is received from the client is converted to a concrete Java object at the server side. This part is handled by JAXB. JAXB converts the XML representation of a query to an object that is a respective subclass of `Query` instance. This query object either instantiates a new respective `Algorithm` for that query and executes the query by the run method of the `Algorithm` or the `Query` object can directly execute the query possibly by calling another method of a class without using any `Algorithm` object. The class hierarchy is given in Figure 5.5 and class composition is given in Figure 5.6.

The methods for the Query interface are:

- `executeQuery():QueryResult`

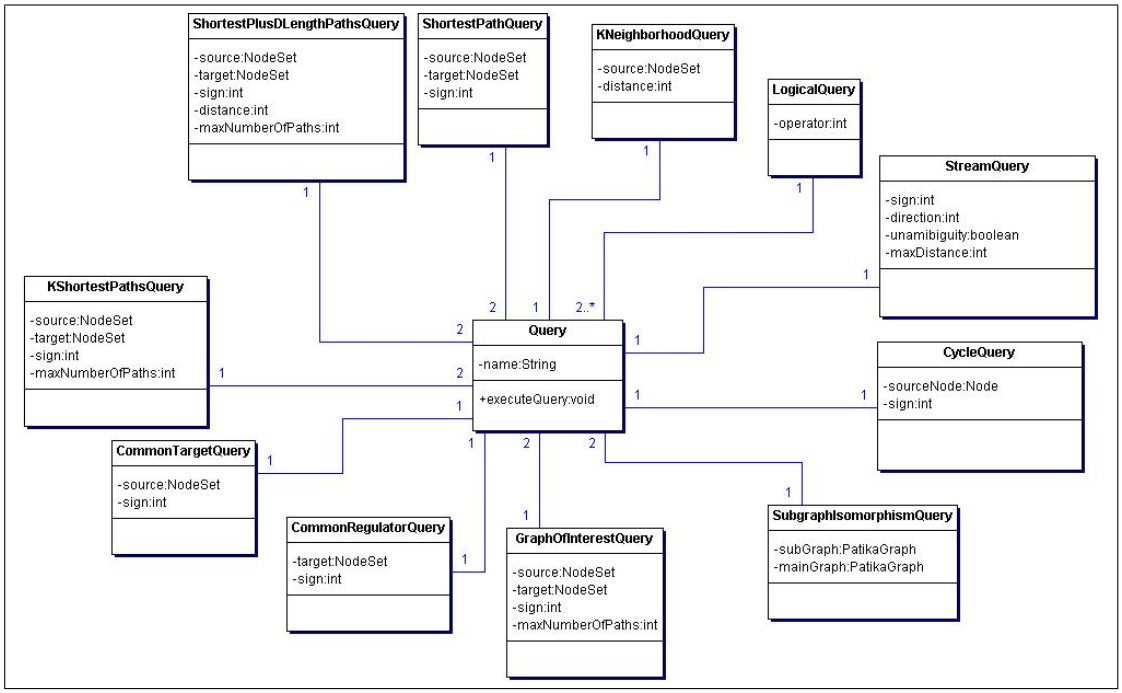

Figure 5.6: Query Composition

- `getList():List`

`executeQuery` method executes the query and builds the `QueryResult` for future uses. `getList` method returns the result of the query as a `List`. If the query is executed, the result list is returned. If the query is not executed, first the query is executed and then the resulting list is returned. This list is obtained from the `QueryResult` object. This method reduces the overhead of re-executing the same query to get the same result. In the query tree, the execution of the query is in the post order. That is for a shortest path, first the source node is retrieved as a field query then the target node is retrieved as a field query; and finally an instance of a `ShortestPathAlgorithm` class is created and executed with the source and the target sets. However, if the set of nodes are known they do not need to be queried. So there are two constructors; one, which takes a `FieldQuery` instance and the other which takes a set of nodes. This situation is similar in other `Query` objects.

Query Result Classes

There is a QueryResult class to hold the result of the queries. `QueryResult` class holds source, target, result sets and a model (i.e. `PatikaGraph`) of the result. The model of the result is the excised graph of the result set from database. In general, `QueryResult` class is sufficient to store the results of the queries. However, if more information should be stored, there are extended `QueryResult` classes like `NeighborhoodQueryResult` class. Examples of such information are list of paths, cycles and neighborhood classes. For handling these cases, there exists a list of lists structure in behind. This structure is not in Java classes but in XML forms, however it directly affects the `QueryResult` class design and worth to mention in this section.

List of lists: There is not a common output structure for the results of the queries. Some of the queries can return only one node (like `GetByPIDQuery`) whereas the others can return 'list of nodes' or 'list of list of nodes'. As an example if the `getList` method returns a list that is composed of `PatikaObject`s to get the result of the `ShortestPathQuery` there occurs a problem. The `ShortestPathAlgorithm` gets a list of target nodes and a list of source nodes as inputs to determine the shortest path from these source nodes to targets. The problem is that there can be more than one path in the result. So there is a problem of how to return these paths. If only one list is used to store the result then the paths are merged together and the client cannot unmerge this structure and the result of the query cannot be viewed correctly. In order to construct a standard structure for all types of query results, list of list structure is used. List of lists structure can hold all three types of query results: only one object, a list of objects and a list of lists of objects.

If there exists a different type of list, then a special XML conversion is needed and a new derived class should be created. By this design, `QueryResult` structure is extremely generic in the ability of storing different types of results such as paths and cycles.

Model: There will also be the model of the resulting set which is a concrete `PatikaGraph` object. The model part is actually optional, since when the user makes query on local machine then there is no need to store the model in the

query result. However, if the user runs a query from database, the excised model should be stored in the `QueryResult` so that the user can reach the resulting objects.

5.2.1.4 XML Formats

XML Representation of a query simply holds all the information to define that query. The definition includes the subqueries, query parameters, and traversal options. Query parameters vary according to the query type. Parameter examples are distance limit, directed or not and positive or negative paths. Traversal options describe how the graph algorithms will traverse over the compound and ubique structures.

XML representation of the result of a query is composed of three parts: the object ID lists, list of lists and the model. In object ID lists are ID lists of source, target and result set objects. The list of lists structure could be just a single list of objects or multiple lists of objects such as multiple paths or cycles. In the model part all the information about these objects are listed. The objects that are not int the result of the query but required because of the restrictions of Patika graph structure are also listed in the model part (i.e. the existence of a transition lead all of its products and substrates to be included in the model).

5.2.1.5 Conversion between XML and Java classes

Jaxb (Java APIs for XML Binding) will be used to convert both XML definitions to Java objects and Java objects to XML definitions. A schema that defines the valid XML files will be formed to check if the given XML file is valid or not. After defining the schema, given a valid XML file, Jaxb can convert this XML file to either Java classes or interfaces. However, if the schema should have complicated types (such as a whole graph model in query results) there should be converter classes that uses the XML Schema Definition to create the corresponding concreter Java class. In our implementation, query classes are

converted via using the easy JAXB facility, while the query result classes has special converter classes for this conversion task.

5.2.2 Query Execution and Data Flow

PATIKA Query Framework aims to provide users with a strong yet easy to use tool for retrieving and analyzing pathway graphs. Queries in PATIKA can be very complicated, due to the nesting structure. Therefore, it is worth to mention about the execution scenario.

In the query scenario, server receives an HTTP request to the query URL. Servlet reads the query in the XML format, dispatches correct controller which in turn unmarshals `Query` type objects from the XML and runs them against the database. All queries return a set of PATIKA objects and some query specific result data e.g. in the case of shortest paths a set of lists describing paths found. However a set of PATIKA objects does not necessarily reflect a consistent view. The minimal consistent subgraph of the database containing the set is determined and copied into a `DBPatikaGraph`, forming a model. This model is marshaled into XML and finally the XML is sent back to the client. On the client side the graph is unmarshaled from XML. Already existing objects are detected, their version is checked and if outdated they are updated, new objects are merged to the model. The resulting view is incrementally laid out.

PATIKA query component has several layers and Figure 5.7 shows order of the data flow of query execution through these layers. The components are explained in flow order as follows.

5.2.2.1 Query User Interface

As the queries in PATIKA will be constructed according to user needs, a user interface layer works in client side to get the inputs of queries from user. This layer allows the user to select the type, inputs and other query-specific attributes from a dialog. When the query result is found (client side query) or taken from

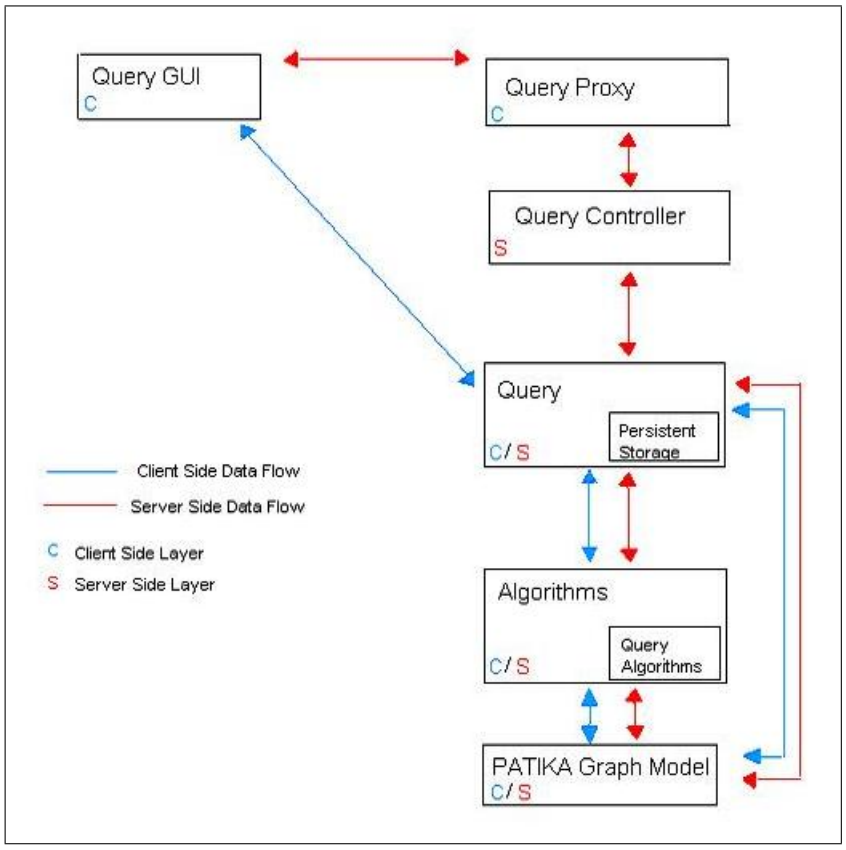

Figure 5.7: Data Flow of Query Framework in Patika

Query Proxy (server side query), this layer is responsible of retrieval of the query result to the user. Details of Query User Interface will be explained later.

5.2.2.2 Query Proxy

As some queries run in server side, the constructed queries with the help of Query Dialog should be sent to the main server to be evaluated. At those times, the queries are converted to XML format (i.e. marshaled) and sent to the main server from client computers. Query Proxy component is responsible from this operation. When the query is evaluated in server side and the result of the query is received from the server, Query Proxy makes the reverse operation and converts the XML data to its original form. When the operation is completed, it sends the query result to Query Interface.

5.2.2.3 Query Controller

Query Controller works in server side and is responsible for evaluating the queries that come from client side. To achieve this, it first converts the XML formatted query to its original form (i.e. unmarshalls it) and then creates a Query object using the parameters. When the query result is found, it converts the result to XML format and sends to Query Proxy.

5.2.2.4 Query

Query is the object constructed by client or server side that contains all the information (i.e. type, running environment, predicates, etc.) about the query created by the user. Also the Query Results are included in this component. Query component sends a Query Result object to the Query Controller. During the evaluation the Queries and Query Results are concrete objects of Java classes as mentioned above, however they are in XML format in traveling between components.

5.2.2.5 Query Algorithms

Query Algorithms are needed for evaluation of queries. As the query types, input parameters and evaluation criteria change from query to query, query algorithm design and implementations differ between them. So, there exist different algorithms for query evaluation. One exception is that field queries do not need any Query Algorithm objects; they are straightforward and evaluated directly using the graph model in client side or server side.

Every query can also receive result of another query as input. In this context, it is possible to create nested queries, which provides querying also according to non primitive fields. For example, by using nested field queries, users can inquire `ComplexMemberState`s whose `BioEntities` have name something. In that example, the inner primitive field query, returns bioentities according to their names.

5.2.2.6 PATIKA Graph Model

The queries of PATIKA run on the graph model in both client and server sides. If the query will run on view or subject graph, graph model on client side is used as the environment to run the query. However, if the database will be used as the environment; the server side graph model is used.

Queries can be grouped into three class: field queries, algorithmic queries and logical queries. PATIKA system allows composing and combining them, allowing a very powerful querying system.

5.2.2.7 Server Side Query Sequence

Figure 5.8 is the sequence diagram for query execution in the server side. Using the parameters taken from the user, through a GUI, the query is constructed, converted to a custom XML format and sent to server side by Query Proxy. In the server side, Query Manager unmarshals the XML and creates a Query object.

Once the query object is created, it is run on database either after the creation of an appropriate algorithm or directly (if it is a Field Query).

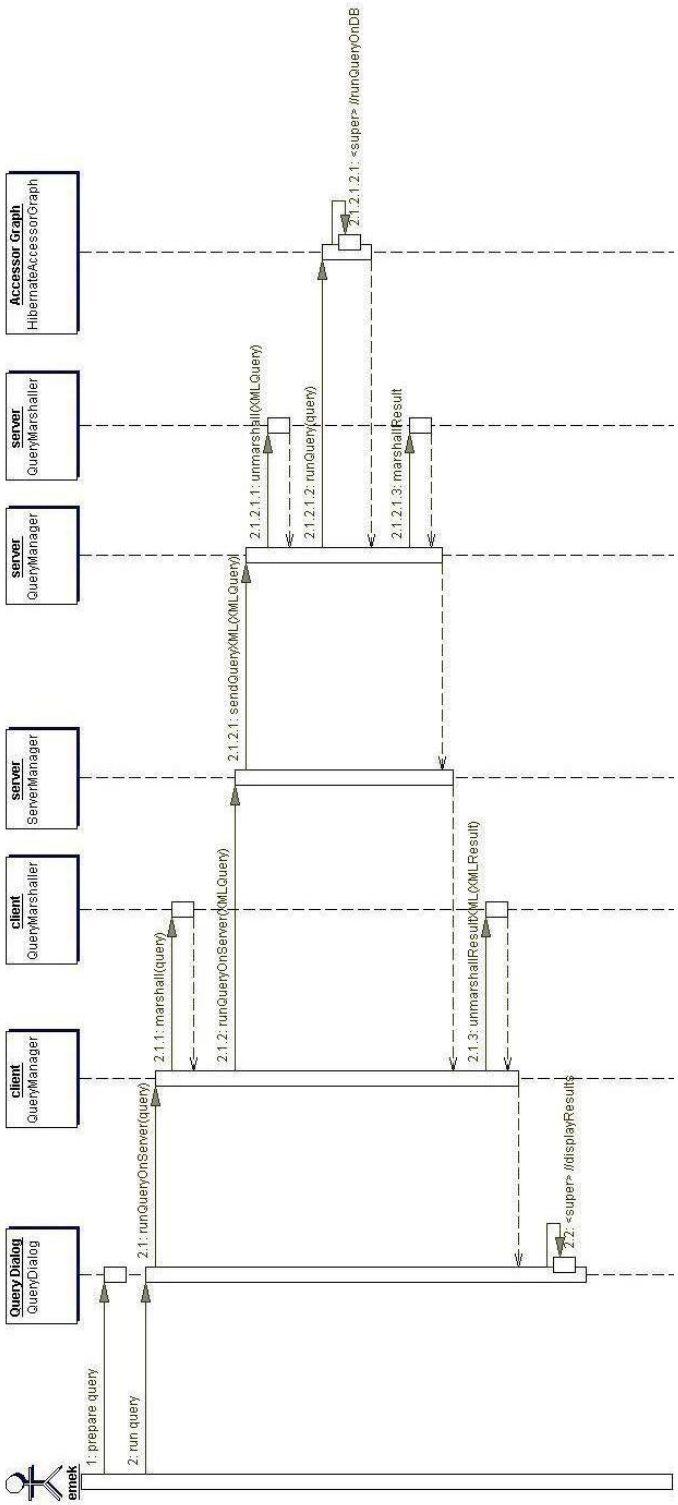

Figure 5.8: Sequence diagram showing that agent emek is running a server side query

The result of the query needs to be sent back to client side and same steps are followed in the reverse order. Query Manager marshals the result of the query into XML format and sends to Query Proxy. Query Proxy unmarshalls the query result and passes the query result to the Query Interface.

5.2.3 Query User Interface

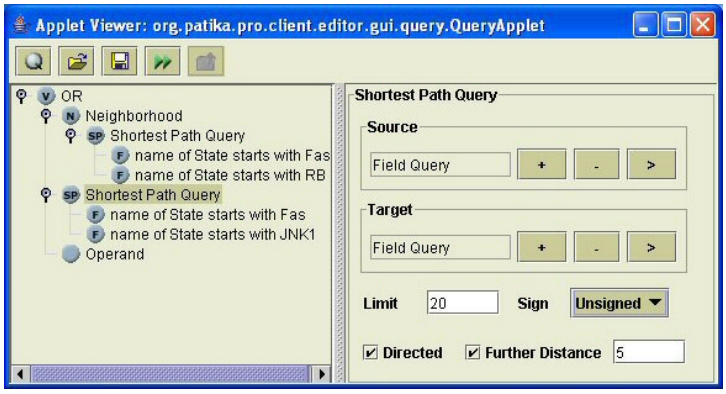

Figure 5.9: Sample query tree to find the **union** of 1-neighborhood of the objects on the shortest path from states whose name starts with "Fas" to states whose name starts with "RB" **with** the shortest path from states whose name starts with "Fas" to states whose name starts with "JNK1"

Querying component of PATIKA supports both SQL-like queries and an array of graph-theoretic queries for finding shortest paths, feedback loops, positive/negative paths, common regulations, or graph of interest based on user's genes of interest. For utmost flexibility, the queries are allowed to be recursively organized into a tree, where result of one query might be source and/or target to another. In addition, queries may be related to each other through "AND" and "OR" operators. Figure 5.9 shows a sample query prepared in the Query Dialog. A query may be executed not only on the database but also on the currently loaded pathway model. After the execution of a query initiated from the query dialog finishes, the returning result (i.e. pathway model) is summarized by the Query Result Dialog.

Once retrieved from the database, the query results may be merged to the

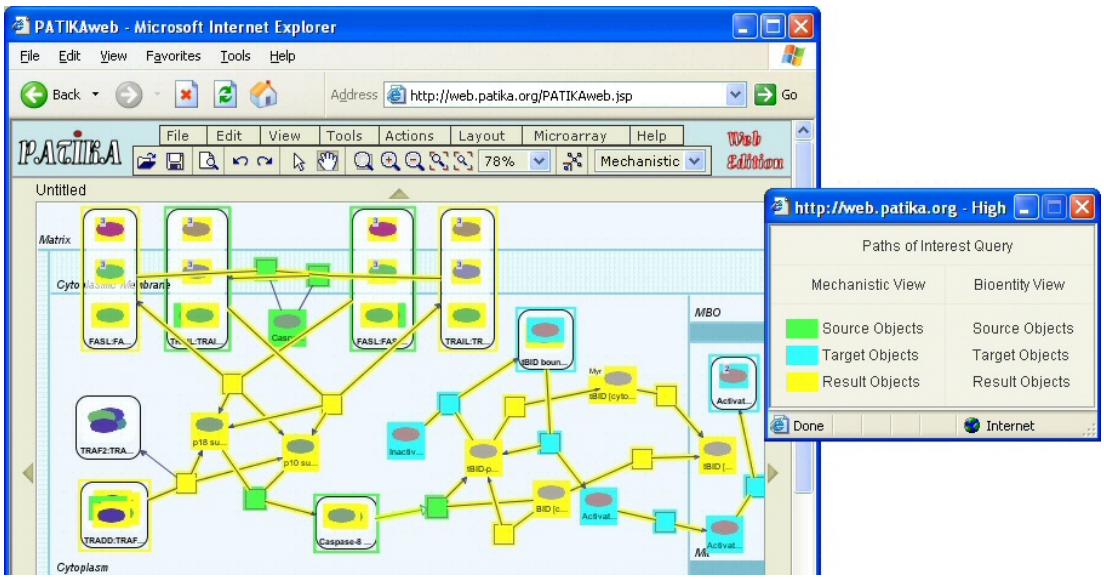

Figure 5.10: Mechanistic view of the result of the following sample query: paths-of-interest (yellow) with source of all mechanistic nodes whose names contain "caspase-8" (green) and target for those whose names contain "bax" (cyan) with limit 8; Highlight Legend Dialog for this query shown on right.

user's current view and highlighted to provide an incremental user-friendly retrieval and analysis interface. Notice that the results can be viewed in either or both of bioentity and mechanistic levels. Figure 5.10 shows a sample query result; interpretation of highlight colors can be made using the Query Highlight Legend Dialog. Alternatively query results may form a new pathway model from scratch. Constructed models can be saved in XML, exported to standard formats such as BioPAX and SBML or converted to static images. The query interface of PATIKA*web*has been implemented as an applet (Figure 5.11 and Figure 5.12). In PATIKA*pro*, the interface is again a windows application. The interface details will be explained over the PATIKA*web*version.

The initial query node of the query forest must be constructed by using the 'New Query' tool in the toolbar. Its sub-queries or parameters may be constructed by the pop-up menu of a query node (Figure 5.13).

The user may choose whether the query is to run on the database or the view (i.e. local/current model) (Figure 5.14).

Figure 5.11: The user is asked to agree to run the PATIKA Query Applet.

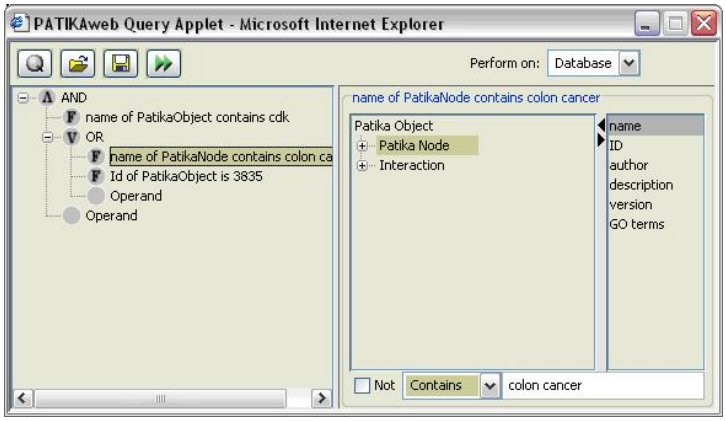

Figure 5.12: The Query Dialog consists of a toolbar (top) and panels for query forest (left) and parameters (right).

When the user clicks on the 'Execute Query' button, the query tree rooted at the currently selected query node of the query forest is executed (Figure 5.15).

5.2.3.1 Query Results

After the execution of a query initiated from the Query Dialog finishes, the returning result (i.e. pathway model) is summarized by the Query Result Dialog (Figure 5.16).

A number of statistics about the result is displayed in this dialog:

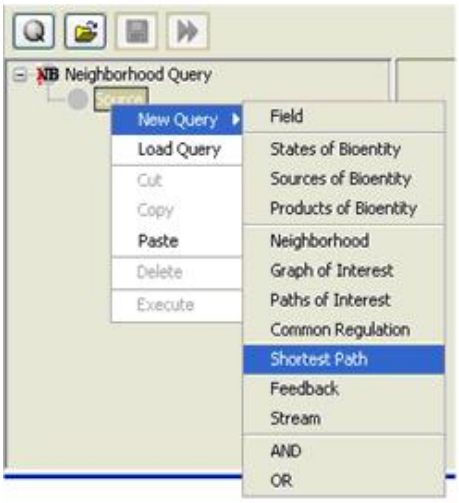

Figure 5.13: The source of a Neighborhood query is set through the pop-up menu of its 'source' query node.

Figure 5.14: The user can specify whether the query is to execute on the database or the view.

- Mechanistic nodes/edges (number of nodes/edges at the mechanistic level)
- Bioentity nodes/edges (number of bioentities/interactions at the bioentity level)

If the user does checks the 'Replace Current Model with Results' option, then the previous pathway model and its views are discarded and the pathway

Figure 5.15: The query rooted at the OR query is executed when the user presses associated button.

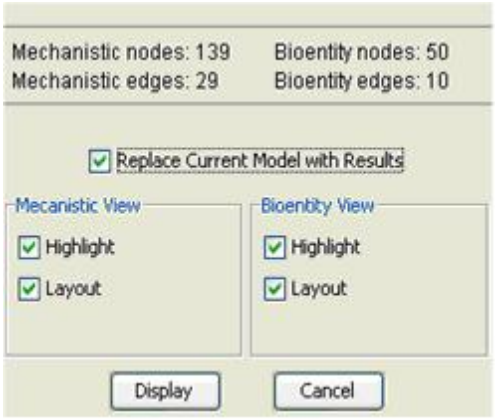

Figure 5.16: Sample Query Result Dialog

Figure 5.17: Sample Query Highlight Legend Dialog, where the last query is a shortest path query, and source, target and result (shortest paths) objects are highlighted with distinct colors (green, cyan and yellow, respectively).

model of the query results is displayed. If this option is unchecked, resulting model is merged into the existing one, possibly modifying both views. The user may opt to highlight the new objects (pathway objects that were not part of the previous model but belongs to the model of the query result) in the views displayed. Normally a separate color is used for different roles in the resulting model. For instance in a neighborhood query, the source nodes are highlighted with a distinct color, whereas neighbors are highlighted with a different common color. The colors are pulled out of a fix color set sequentially. The legend for the highlight colors of the last query may be reached through 'View | Query Highlight Legend'. A sample highlight legend is shown in Figure 5.17.

If the user checks 'Layout', then the associated view is also laid out.

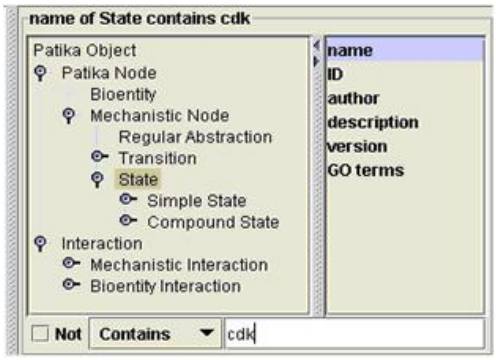

Figure 5.18: You may constrain your query to a PATIKA object of a specific type.

5.2.3.2 Persisting Queries

You may persist the constructed queries locally on disk for later use clicking on the save tool in the Query Dialog toolbar. The default extension for XML based PATIKA queries is '.pql'. Note that it's the query tree rooted at the selected query object that is saved on disk, not the entire query forest or the entire associated query tree.

To reload a previously saved query, click on the load tool in the Query Dialog toolbar, and point to the proper '.pql' file. This will not destroy the current query forest but add a new query tree at the very end.

5.2.3.3 Basic Queries

Field Query

The simplest query type that can be performed in PATIKA through the query framework is the field query. Following fields of objects can be queried: name, PATIKA ID, Author ID, description, version and GO terms. In most queries including the field query, you can specify and constrain the search to a PATIKA object type using the PATIKA object tree (Figure 5.18). Please refer to Chapter 5.1 on PATIKA Model Implementation for a better understanding of the PATIKA object tree.

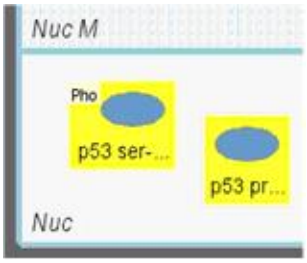

Figure 5.19: All states of protein 'tp53' (yellow) as obtained by a 'States of a Bioentity' query

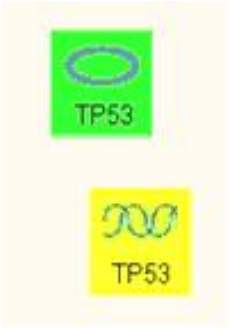

Figure 5.20: Sources (yellow) of protein 'tp53' (green) as obtained by a 'Sources of a Bioentity' query

States of a Bioentity

This query is used to find all states of a specified set of bioentities in the current model or the database. For instance, all available states of protein 'tp53' can be obtained through this query as shown in Figure 5.19.

Sources of a Bioentity

This query may be used to find all source bioentities of a specified set of bioentities in the current model or the database. As shown in Figure 5.20, DNA 'tp53' is found as the only source bioentity of protein 'tp53' in PATIKA database.

Products of a Bioentity

This query may be used to find all product bioentities of a specified set of bioentities in the current model or the database.

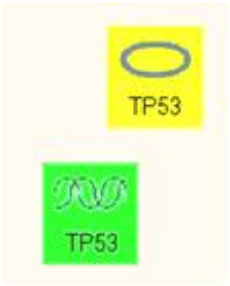

Figure 5.21: Products (yellow) of DNA 'tp53' (green) as obtained by a 'Products of a Bioentity' query

Figure 5.22: Queries can be combined through logical operators: Search for all `PatikaNode`'s whose description contains 'colon cancer' or ID equals 3835 authored by Joe Smith.

5.2.3.4 Logical Queries

Queries can be combined using logical operators. In addition, most queries are defined recursively taking the result of another query as input (Figure 5.22).

5.2.3.5 Advanced Queries

More advanced graph algorithms that involve traversal starting with a set of source and/or target objects are explained in this section.

Advanced Query Options

Below are some common traversal options valid for all advanced queries. These options can be accessed and set through the button 'Options' on the right hand side of each query panel (Figure 5.23).

Traversal Over Ubiques Traversal over ubiquitous molecules may be

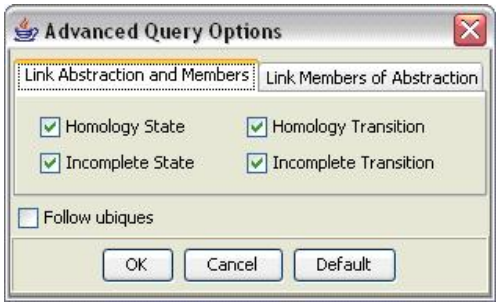

Figure 5.23: Advanced Query Options Dialog is used to set traversal options for ubiques and abstractions.

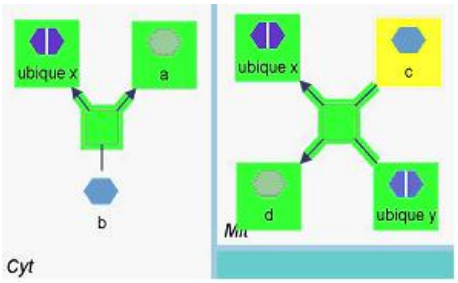

Figure 5.24: Whether or not 'a' is in the 4-neighborhood (green) of 'c' (yellow) depends on whether traversal over ubiques ('ubique x' in this case) is allowed. Obviously, in this case it is allowed.

avoided by un-checking 'Follow Ubiques' flag in Options Dialog. Since such molecules are involved in potentially hundreds if not thousands of reactions at mechanistic level, one might prefer not to link two reactions whose only common actors are these kinds of molecules.

Traversal Over Abstractions These options decide how the traversal should continue, when it reaches an abstraction or a member of an abstraction. For instance, reaching a member of a complex molecule should often be interpreted as reaching all members of this complex; thus the traversal should be able to continue from another member.

Neighborhood Query

Neighborhood query (NB) results in a set of molecules that are at most a specified distance from the source set (Figure 5.25).

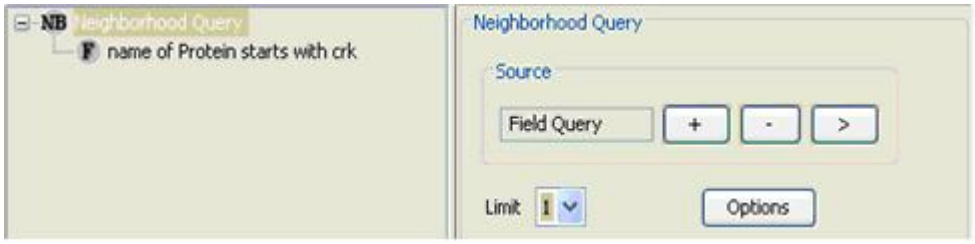

Figure 5.25: Sample Neighborhood Query Dialog for 1-neighborhood of protein bioentities whose names start with 'crk'

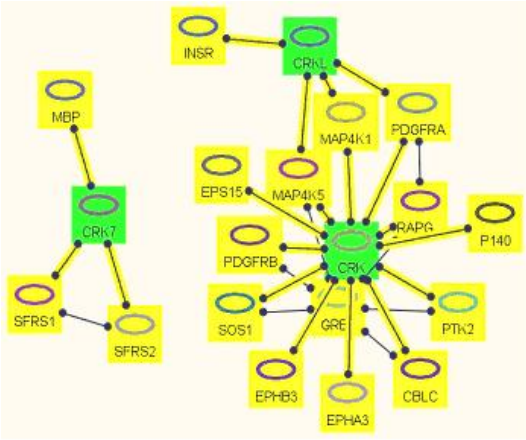

Figure 5.26: 1-neighborhood of protein bioentities whose names start with 'crk' (result of the query in Figure 5.25)

- limit specifies the maximum distance of a molecule from a source molecule to be considered as part of the result of this query.

There are shortcuts for the neighborhood query in the pop up menu of nodes in the currently displayed view. When you select 'Find Neighbors in View' or 'Find Neighbors in Database' item in the pop up, a neighborhood query, respectively on the current view or the database, is performed where the source is the field query with the ID of the hit object, and limit is 1.

Graph/Paths of Interest

Graph-of-Interest (GoI) query aims at completing the 'missing links' (and molecules on these links) among a set of molecules of interest that is no longer than a specified limit (Figure 5.28).

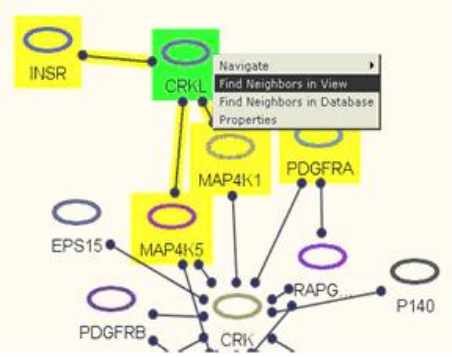

Figure 5.27: Immediate neighbors (yellow) of protein 'CRKL' (green) may be queried using its popup menu

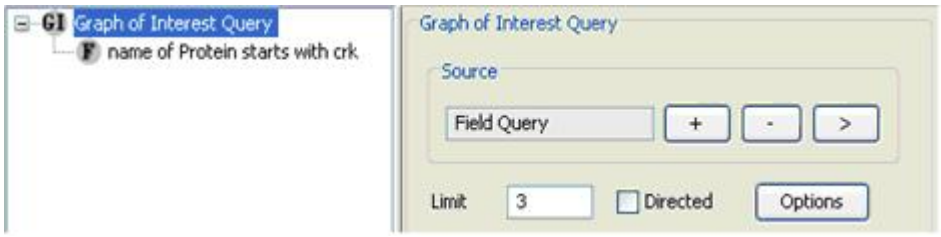

Figure 5.28: Sample GoI Query Dialog

- limit specifies the maximum length of a missing link from a molecule in the specified source set to a molecule in the same set to be considered as part of the result of this query. Paths-of-Interest (PoI) query, on the other hand, does the same thing from a specified set of source molecules to a specified set of target molecules (Figure 5.30).

- limit specifies the maximum length of a missing link between a molecule in the source set and a molecule in target set to be considered as part of the result of this query.

Common Target / Regulator

If node A is the starting node of a directed path that ends up in node B, then node B is said to be a target of node A; similarly, node A is said to be a regulator of node B. In this context, common target of a source molecule set S is the set of molecules that are targets of all molecules in S. Similarly, the common regulator of a target molecule set S is the set of molecules that are regulators

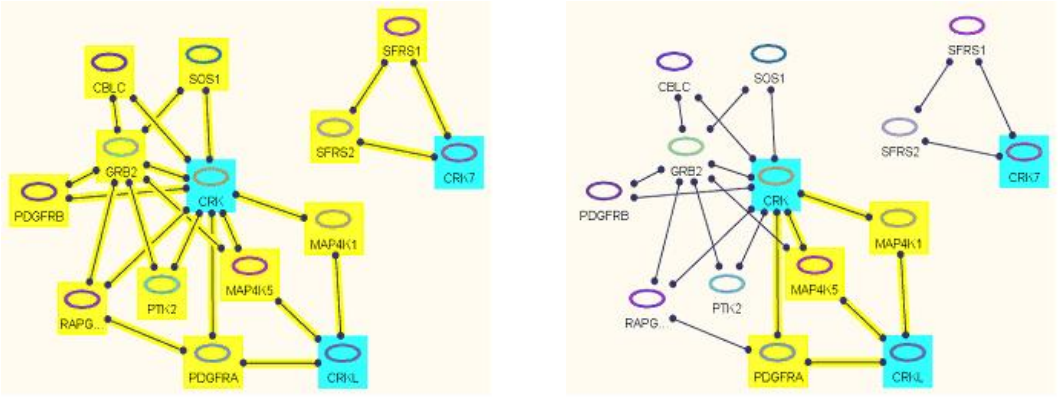

Figure 5.29: Result of the query in Figure 5.28: GoI (yellow) of protein bioentities whose names start with 'crk' (cyan) where limit is 3 (left) and where limit is 2 (right). Notice that each protein on the GoI (yellow) is on at least one path between two source nodes (cyan) whose length is at most limit.

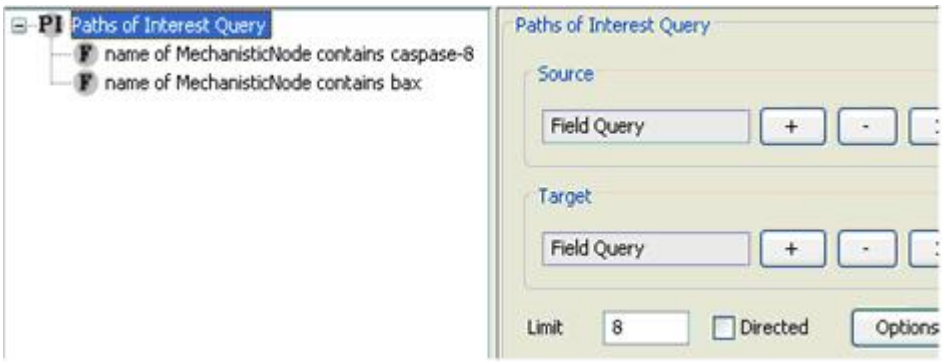

Figure 5.30: Sample PoI Query Dialog

of all molecules in S. This query results in a set of molecules that are common targets (regulators) of all the molecules in the source (target) set (Figure 5.32).

- limit specifies the maximum distance of a target (regulator) molecule from a source (target) molecule to be considered as part of the result of this query.
- direction specifies whether we are interested in finding targets or regulators.
- include regulation paths specifies whether the actual paths from the given molecule set to the targets (or from the regulators to the given molecules) are to be included as part of the result.

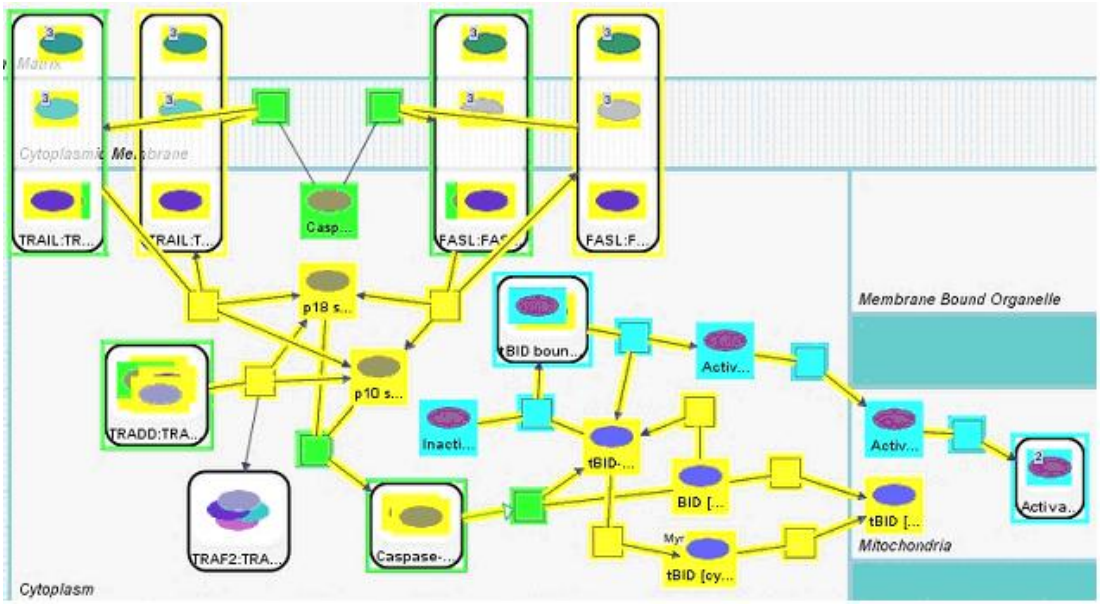

Figure 5.31: Result of the query in Figure 5.30: PoI (yellow) between mechanistic nodes whose names contain 'caspase-8' (green) and those whose names contain 'bax' (cyan)

Shortest Paths

This query may be used to find the shortest paths (that is, pathways of shortest length) between two sets of pathway objects (i.e., source and target sets) (Figure 5.34). The user may specify source and target object sets by other PATIKA queries (e.g., field queries).

- limit specifies the maximum length of a path to be considered as a result of this query.

Figure 5.32: Sample Common Regulator Query Dialog

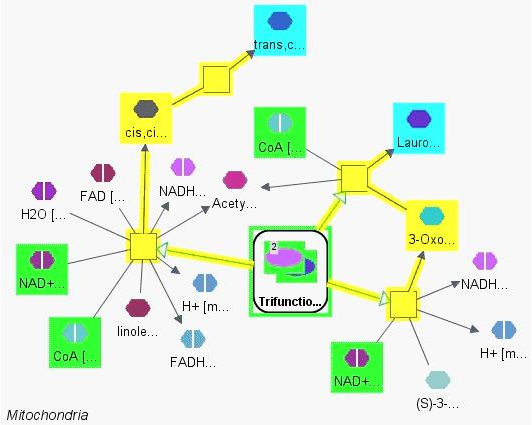

Figure 5.33: Result of the sample common regulator query in Figure 5.32, where we find common regulators (green) of the simple states whose names contain 'lauro' (cyan); paths leading from common regulators to the targets are shown in yellow.

- directed is used to specify whether the edge (interaction) direction (e.g., product edge of a mechanistic view, going from the associated transition to the product state) is to be taken into account or not.
- further distance allows selection of paths that are longer than the shortest path(s) up to a certain length.

Feedback Query

This query results in a list of positive or negative cycles that contain a specified node and can be used in studying cellular networks (Figure 5.36). These queries can have signal amplifying and stabilizing roles and may help answer questions such as 'How is the concentration of a molecule stabilized?' and 'How did a signal get amplified?'

- limit specifies the maximum length of a cycle to be considered as a result of this query.
- sign tells whether we should filter the resulting cycles to be restricted to the positive or negative ones only.

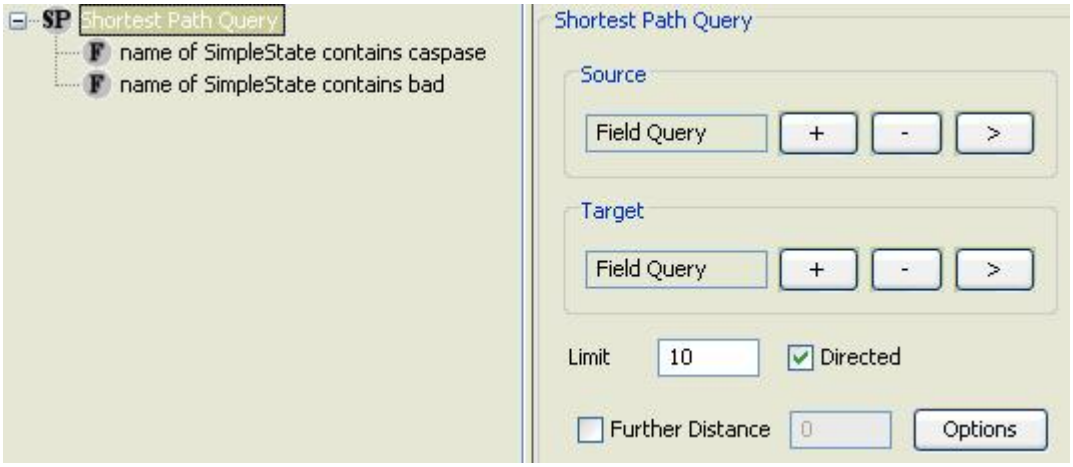

Figure 5.34: Sample Shortest Paths Query Dialog

Stream Query

This query may be used for analyzing upstream and downstream of a molecule (Figure 5.38). It has a significant role in retrieving cause/effect relationships which are essential for diagnosis, following up the development of certain events and drug design. In addition, these queries can provide answers to questions such as:

- Which molecule activates this protein/molecule?
- Which processes are affected if this molecule/gene is knocked down?
- What are the downstream/upstream effects of certain drugs/molecules?

The query is constructed by defining a source set as a sub-query, typically a field query, from the query dialog. The user also specifies the sign (i.e., positive or negative) of the path, and a threshold for the path length from the source or to the target. All nodes that have an outgoing/incoming path (for upstream and downstream, respectively) satisfying the constraints are returned, and highlighted with a different color.

- direction is used to specify whether we are to find the upstream of target objects or the downstream of source objects.

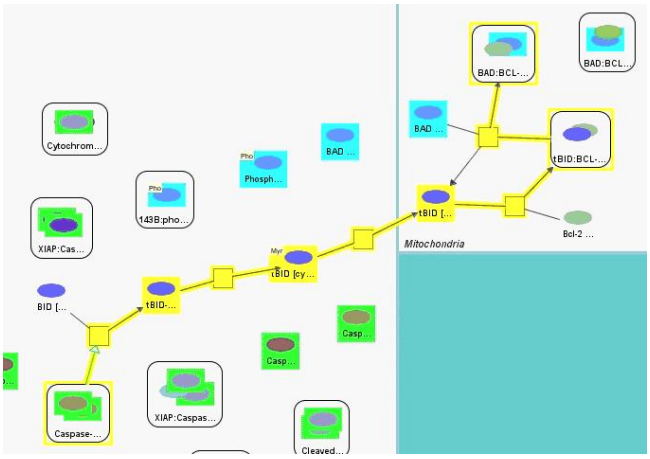

Figure 5.35: Result of the sample shortest paths query in Figure 5.34, where we find directed shortest paths (yellow) from simple states whose names contain 'caspase' (green) to simple states whose names contain 'bad' (cyan)

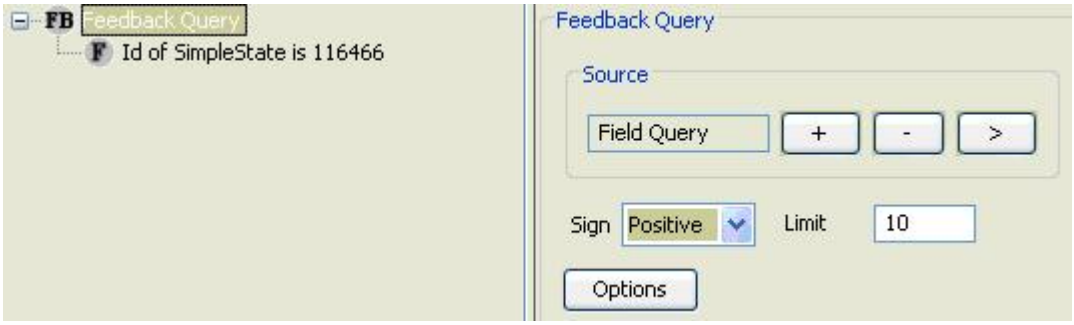

Figure 5.36: Sample Feedback Query Dialog

- limit specifies the maximum length of a path to be considered as part of the result of this query.
- sign tells whether we should filter the resulting paths to be restricted to the positive or negative ones only.
- ambiguity specifies whether or not ambiguous streams (multiple paths of conflicting sign with same source and destination) should be part of the result.

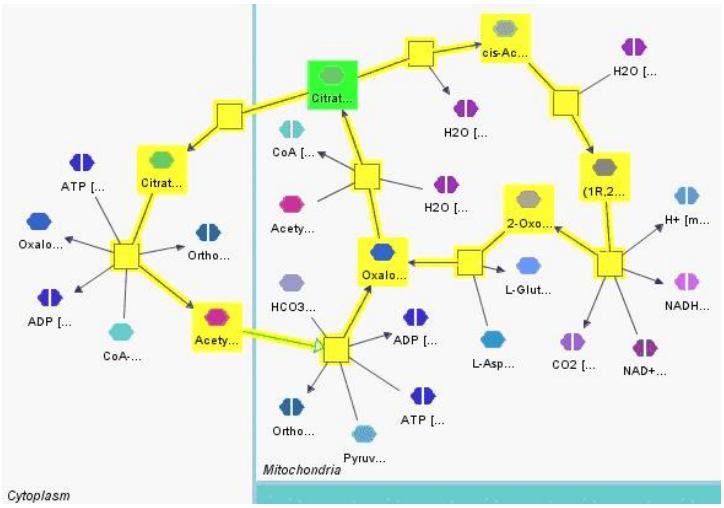

Figure 5.37: Result of the sample feedback query in Figure 5.36, where we look for positive feedback (yellow) of a given Citrate state (with specified ID) in mitochondria (green) with up to length 10; the result contains two feedback cycles, one in mitochondria (of length 10) and one through cytoplasm (of length 8).

5.2.3.6 Sample Session

Following is a sample session in which subsequent queries and complexity management operations are performed to form a model that might be of use to a PATIKA*web* user. Suppose the user is studying the effects of FAS Ligand on apoptosis. One good way to start is by searching for the relations between FAS Ligand and the Caspase complexes in the cell. In order to find out the states of FAS Ligand in the cell, we perform the query in Figure 5.40, where we ask for simple

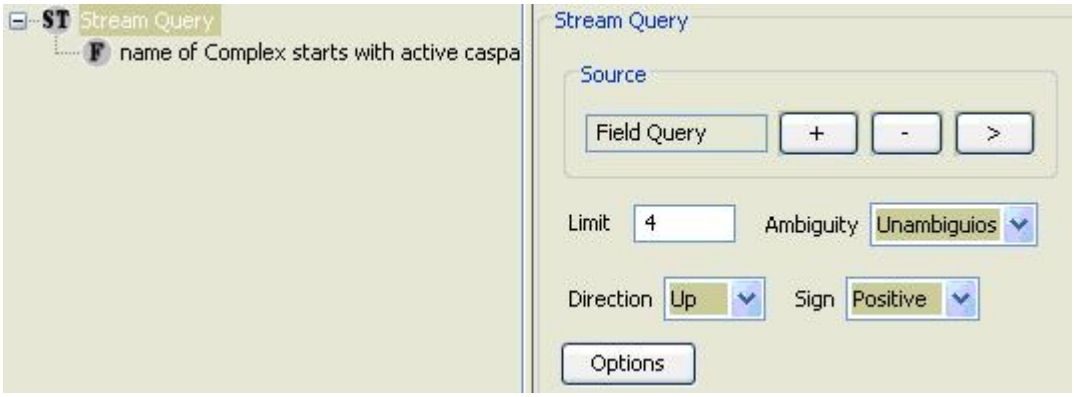

Figure 5.38: Sample Stream Query Dialog

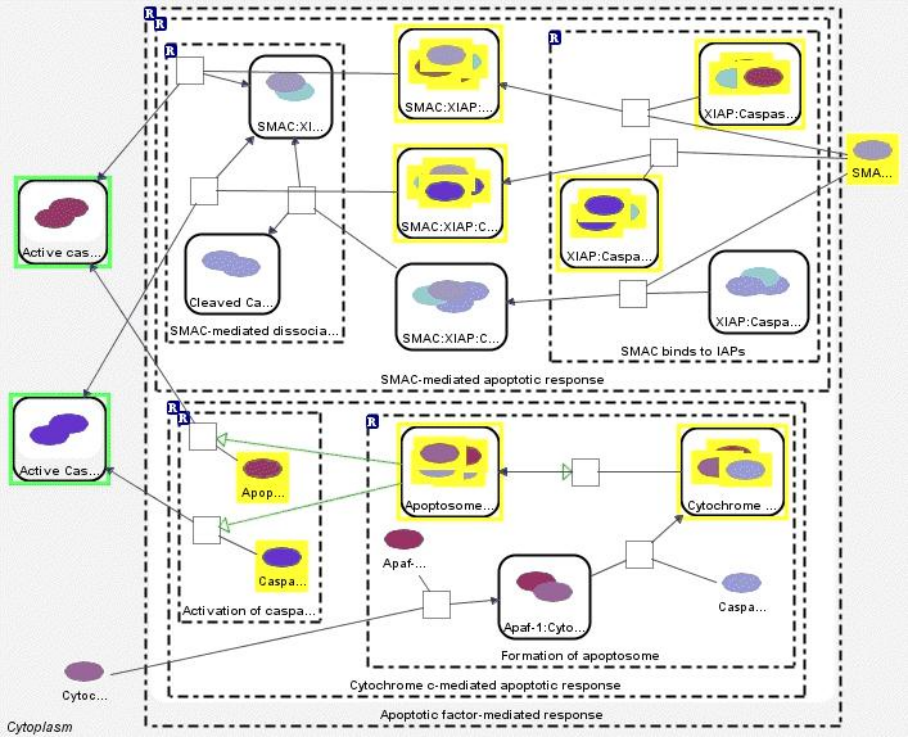

Figure 5.39: Result of the sample stream query in Figure 5.38, where we look for unambiguous positive upstream, with limit 4, (yellow) of complexes whose names start with 'active caspase' (green)

states whose names start with 'FASL'.

We see six states highlighted in the result of the query (Figure 5.41). One is the free extracellular FAS Ligand, and remaining ones are members of several complexes spanning the cytoplasmic membrane. And we may check how many Caspase complexes we have in the database, which are not a precursor or a pro-caspase (Figure 5.42).

Caspase query returns a total of 11 complex molecules, which are all in cytoplasm (Figure 5.43). Now we know that the database contents that we want to 'start from' and we want to 'reach to'. The most popular query for finding relatively short paths between source and target molecules is the 'Shortest Path Query' described earlier. We may use the previous FAS Ligand and Caspase field queries as the source and target fields of the shortest path query (Figure 5.44).

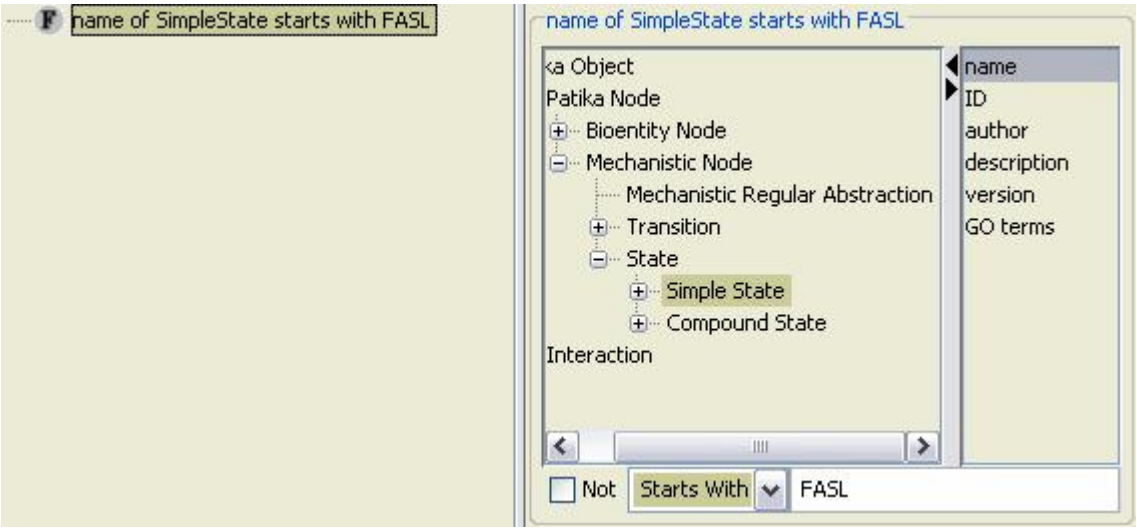

Figure 5.40: Query for simple states whose name starts with 'FASL'

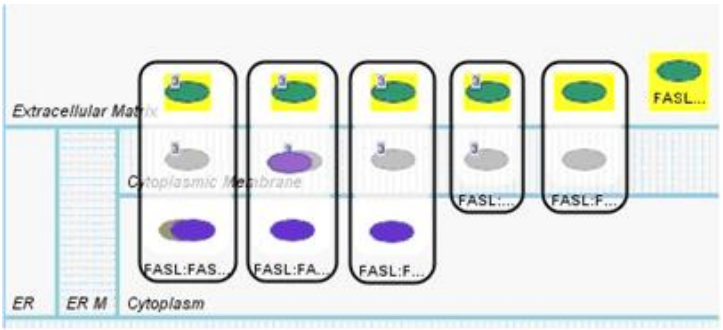

Figure 5.41: Result (yellow) of the FAS Ligand query in Figure 5.40

Result of the shortest path query retrieves paths of length 4 (Figure 5.45). These are paths involving the FAS Ligand complex on the cytoplasmic membrane and the Caspase-8 dimer in cytoplasm. This picture might be very helpful but it still has many missing relations. There are several ways to obtain a more complete picture. First alternative is to use the shortest path query with the "Further Distance" parameter. Figure 5.46 shows the same query with further distance set to 8. Since the shortest path length is 4, this query would bring us the paths from source to target nodes of length at most 12. Figure 5.47 shows the resulting model.

Another way of doing the same query is to use a 'Paths-of-Interest' (PoI) query with limit 12. Since this query will bring all paths of length at most 12, between

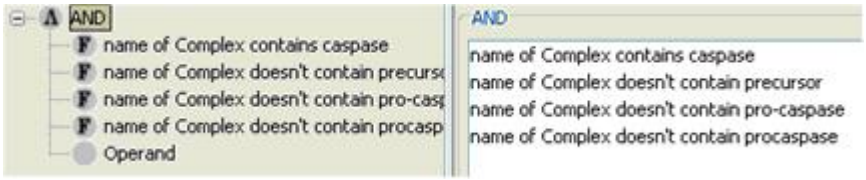

Figure 5.42: Query for Caspase complexes, which does not contain words 'precursor', 'pro-caspase' or 'procaspase'

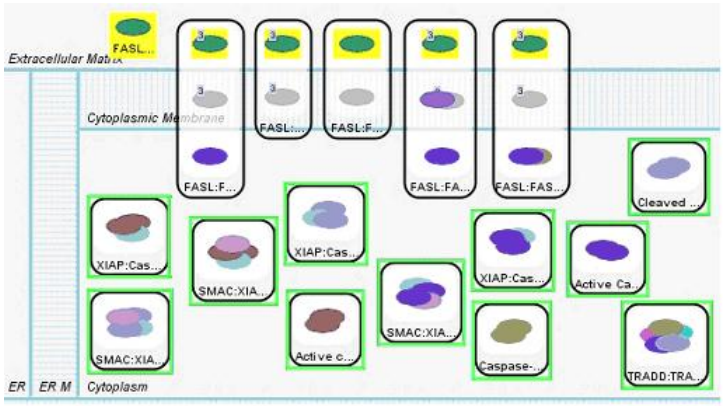

Figure 5.43: Result (green) of Caspase query in Figure 5.42 added to the existing model

source and target sets, the result will be identical to the previous shortest path query with further distance 8. Thus PoI query is simply a more convenient way of querying paths when we have a good estimation of the length of the shortest path. When finding paths between source and target sets is not sufficient, the user has the option to do a Graph-of-Interest (GoI) query. GoI query aims at completing the 'missing links' (and molecules on these links) among a set of molecules of interest that is no longer than a specified limit. So a "minimal" graph including the specified objects of interest can be constructed through this query. Figure 5.48 shows a directed GoI query with limit 5, where the previous source and target sets are joined into an OR query as molecules of our interest.

Since the GoI query finds all paths between a number of seed nodes (not from a specified source to specified target), the result contains more paths, not necessarily depicting a direction in the information flow. In the resulting model (Figure 5.49) we see that there are two isolated components. First one contains

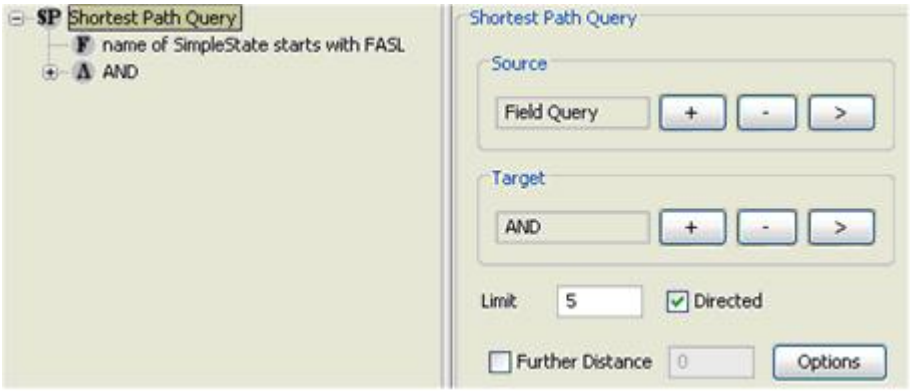

Figure 5.44: Shortest path query using the previous queries as source and target; the query limits the distance to 5 and considers directions.

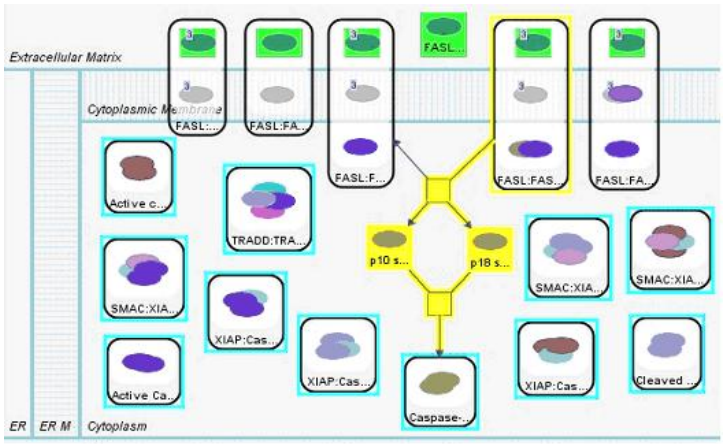

Figure 5.45: Result of the shortest path query in Figure 5.44; we find that the two shortest paths (yellow) from FAS Ligand (green) to Caspase complexes (cyan) goes to Caspase-8 dimer in cytoplasm, each with 2 transitions (4 steps).

the previous FAS Ligand path we have found. We see that an additional Caspase complex is connected; however, the graph does not imply that this new Caspase complex has been involved in the FAS Ligand signaling process. Second component contains all other Caspase complexes. Notice that only two Caspase complexes have a relation with FAS Ligand signaling process in the database (at least within the distance we have specified); the user may choose to concentrate on these for further analysis.

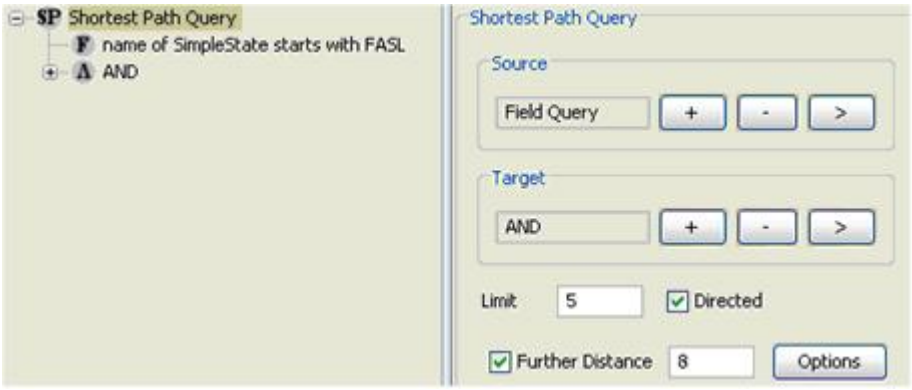

Figure 5.46: Shortest path query with further distance set to 8

5.2.4 Query Framework Deployment

Figure 5.50 depicts how the system will be deployed and especially how the interaction will be provided during server side query execution. Thin Bridge has an HTTP communication with Web Browser. Both Heavy and Thin Clients have many-to-one HTTP communication with server, Patika Server. Patika Server is deployed upon Tomcat Servlet which instantiates Query Controller according to the type of HTTP request coming from clients. In client sides if the query is to be run on server side, the relevant query data is converted into Patika XML and then Query Jaxb which is deployed under Query Controller processes that data by reconverting.

In Patika Server, Spring Controller is deployed which includes Query Controller and Submission components. Under Query Controller, after Query Jaxb component instantiates Query, that query is executed by using Query Algorithms which access Patika Model and the result is held in QueryResult. Persistence component of Patika Server is deployed with the collaboration of Spring and Hibernate. In Persistence component a PostgreSql Relational Database Management System is installed. Hibernate Object Relational Mapping access database via JDBC and configures Hibernate XML map during run-time. In DB Model component HibernateAccessorGraph access to ORM via Spring ORM Support. Patika Model, which is accessible from Query Controller, is an instance of that DB Model. Spring View component is instantiated by Spring Controller when it

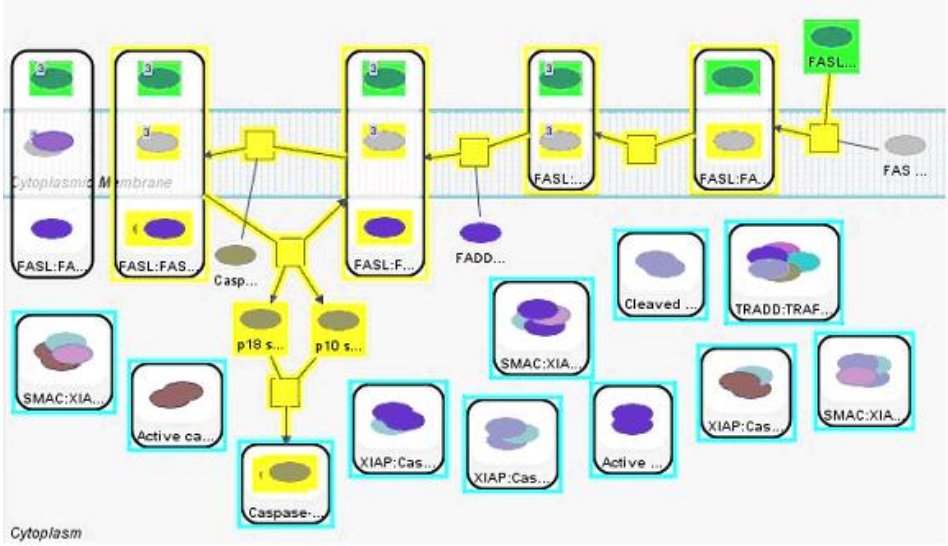

Figure 5.47: Result of the shortest path query in Figure 5.46. Paths of length up to 12 (yellow) are found between source (green) and target (cyan) sets, since the shortest path length is 4.

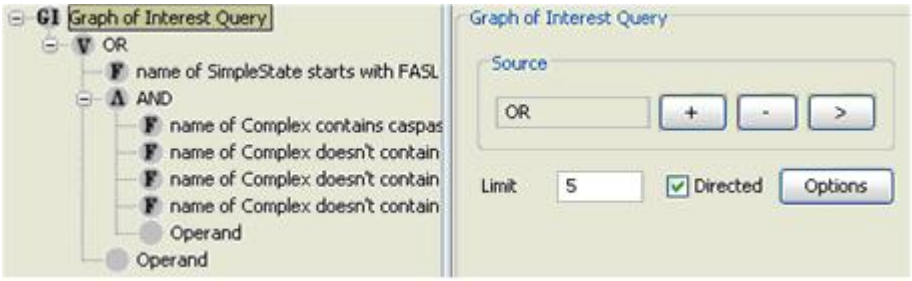

Figure 5.48: A GoI query where the previous FAS Ligand and Caspase complex queries are gathered into an OR query and used as seed (molecules of interest)

is needed to send the query result to the clients. In Spring View, Patika Jaxb accesses the Query Result structure and converts it into Patika XML. Then the client gets that Patika XML to display in its view.

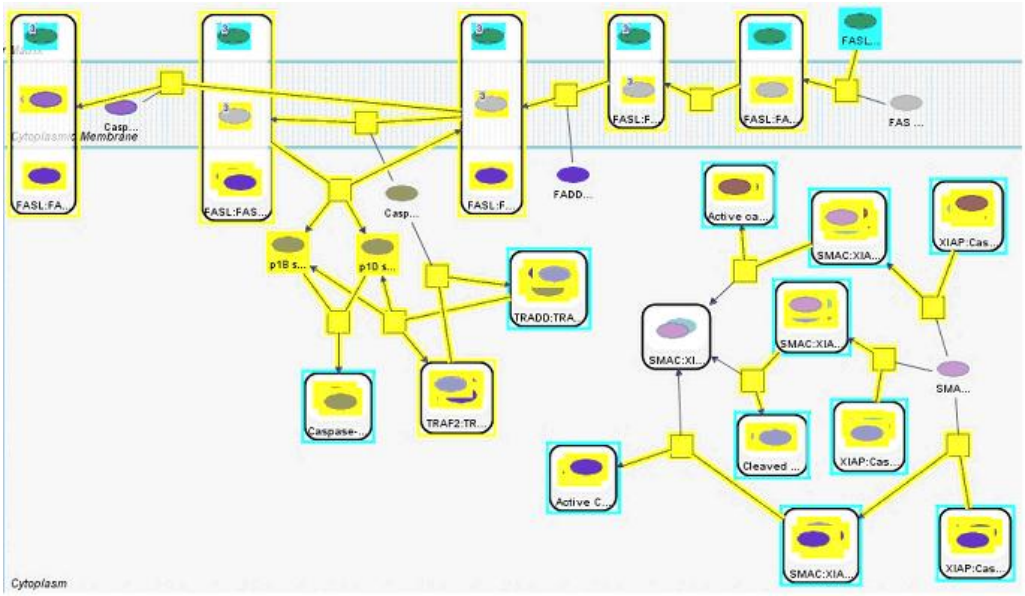

Figure 5.49: Result of the GoI query in Figure 5.48

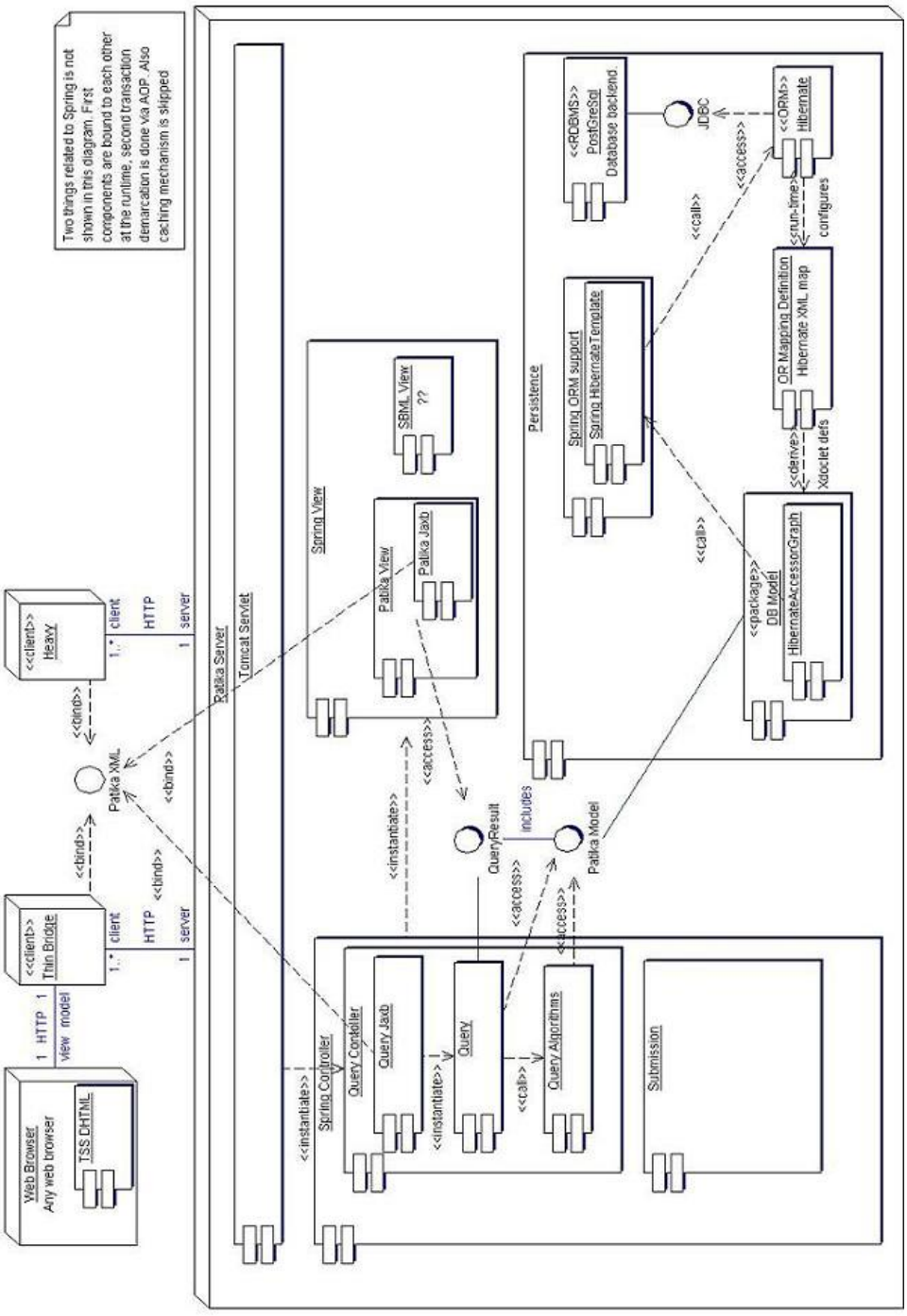

Figure 5.50: Deployment Diagram of Query Framework in PATIKA

Chapter 6

Test Results and Performance

We have performed a number of experiments to test the performance of our algorithms using the implementation within PATIKA*web*. The tests were performed on an ordinary personal computer using a randomly created integrated pathway knowledgebase, consisting of about 20,000 pathway nodes and 30,000 edges. The knowledgebase was assumed to be held in memory, however. So it should be expected that these algorithms will execute slower on pathway databases on disk. However, with the success of current high performance object/relational persistence and query services such as Hibernate, the slowdown should not be dramatic.

Our experiments have revealed that our theoretical analysis is in line with the results of our implementation. Below we detail out some of them.

One set of experiments we have performed was for computing the graph of interest $\text{GoI}(S, k)$. Remember that the time complexity of this algorithm is linear in the number of nodes in the k-neighborhood of nodes of interest or the size of the result set as Figure 6.1 verifies. We have also measured k versus execution time with random source sets of various sizes. As Figure 6.2 illustrates, independent of the source set size, execution time increases rapidly (but not exponentially) up until a certain distance (between 12 and 15) after which, it remains constant. One might expect an exponential increase in the number of nodes reached (and thus

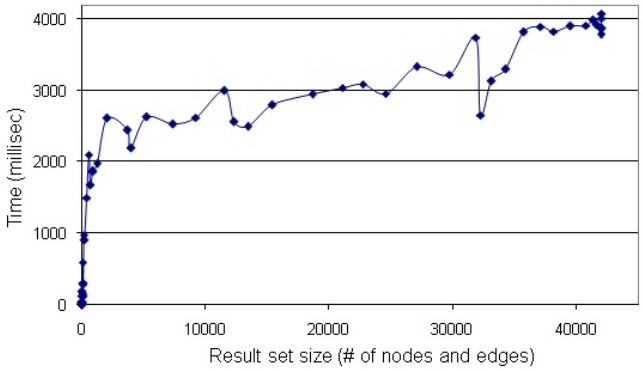

Figure 6.1: Result set size vs. execution time for GoI algorithm

the execution time) as distance increases; however, in practice, potentially all of the neighbors of a newly visited node might already have been visited, avoiding a combinatorial increase. And after a certain number of steps all nodes would have been already visited.

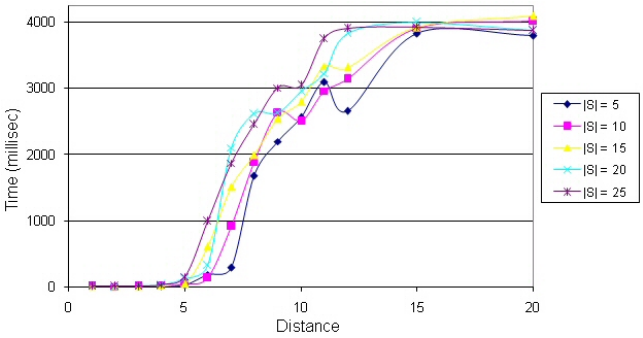

Figure 6.2: Distance (k) vs. execution time for GoI algorithm for various source set sizes ($|S|$)

Another set of experiments we have performed was for computing shortest paths $\mathrm{SP}(S, T, k, d)$. Remember that time complexity of this algorithm is $O(l + |\mathrm{NB}(S, k)|)$, where l is the total length of the paths enumerated. Here l can be exponential in the size of the graph in the worst case. However, our experiments show that on the average that is not the case and l is dominated by the second term. The reason for this situation is that, roughly speaking, if the shortest length between sets is "too long" then the number of paths found is small, and similarly if number of paths found is large then shortest length between sets is short. Obviously, l equals number of paths found times shortest path length between

sets. Therefore there is a trade off between number of paths and shortest length between sets, and this hides the exponential behavior expected in the worst case. Figure 6.3 shows the plot of the shortest path length versus execution time for the shortest path algorithm. However, if we do not enumerate individual paths,

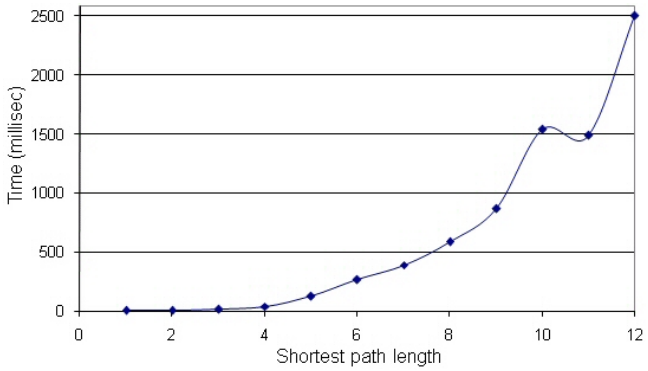

Figure 6.3: Shortest length vs. total execution time of the shortest-path algorithm

then the algorithm should behave linear in the number of k-neighborhood of the source set, where k is the shortest length between sets.

Time complexity of computing common regulation $\mathrm{CR}(S,k)$ is $O(|S| \cdot |\mathrm{NB}(S,k)|)$. Experiments also support this complexity as shown in Figure 6.4. While the distance limit k is fixed and the source set size increases slightly, the algorithms behaves linear as the $|\mathrm{NB}(S,k)|$ term does not change much for different values of $|S|$. With different fixed k limits, linear behavior of the computing can also be seen. Also the aggregate execution time is fairly sufficient for interactive usage.

For the Stream algorithm, it is very important to have a reasonable execution time, since its time complexity is non-polynomial. Our experiments show that this algorithm is sufficiently fast for interactive usage. In Figure 6.5, the execution times of Stream algorithm are illustrated. The experiments are done with fixed k. As it can be seen, even if the source set size and distance limit k are both large, the execution time reaches up to no more than 10 seconds.

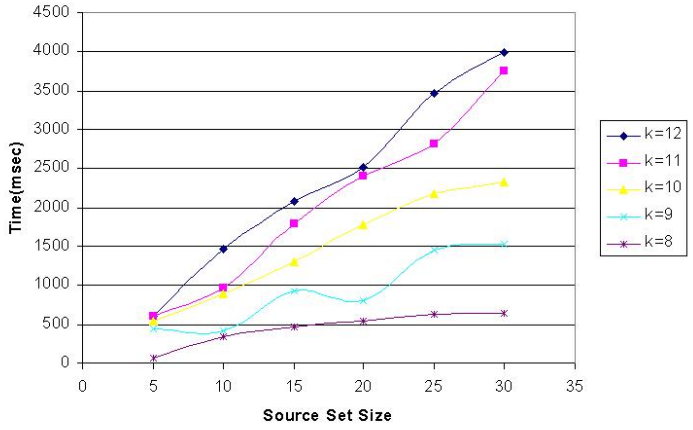

Figure 6.4: Distance (k) vs. execution time for CR algorithm for various source set sizes ($|S|$)

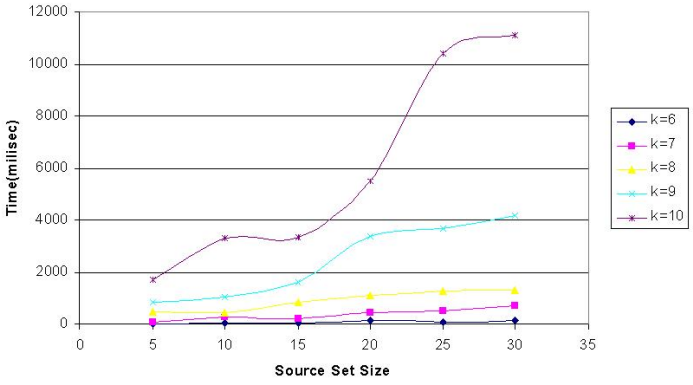

Figure 6.5: Distance (k) vs. execution time for Stream algorithm for various source set sizes ($|S|$)

Chapter 7

Conclusion and Future Work

In this thesis, we have presented a querying framework along with a number of graph theoretic algorithms applicable to all sorts of graph-based pathway databases. The framework provides both querying and query result visualization through a user friendly interface.

The queries are designed for answering a number of biologically significant queries for large graph-based pathway databases. Towards this goal, the queries of Neighborhood, Graph of Interest, Shortest Paths, Common Regulation, Feedback and Stream are described with formal graph definitions. Beyond these algorithmic queries, basic field and logical queries are also provided for utmost utility.

Neighborhood, Graph of Interest, Shortest Path and Common Regulation algorithms are based on breadth first graph traversal with novel node labeling techniques for handling sign of paths and filtering out irrelevant paths. While Common Regulation has super-linear complexity, other BFS-based algorithms are linear in the number of nodes and edges in the k-neighborhood of given source sets.

Feedback algorithm is a modified version of the algorithm of generating all cycles mentioned in [21] which is based on depth first graph traversal. The contribution of our algorithm is to be able to put a constraint on the length and

the sign of cycles. Stream algorithm is optimal despite its exponential complexity. The experimental results also show these two algorithms are fairly satisfactory in execution time.

The main contribution of our work to the field is a *whole* framework providing advanced query creation, execution and query result visualization with use of graph-theoretic algorithms for biological analysis over an *integrated* pathway database with support for *compound* structures.

However, there is space for improvement. As future work, we may look at ways to speed up our algorithms as they are destined to be part of interactive systems. Also visualization and query interface may be improved for more user friendly software. Most importantly, there may be defined new and powerful queries such as subgraph matching with efficient algorithms.

Bibliography

[1] A. M. Abdulkader. *Parallel Algorithms for Labelled Graph Matching*. PhD thesis, School of Mines, Colorado, 1998. Understanding and Analysis, MIUA 2001.

[2] T. Aittokallio and B. Schwikowski. Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics*, 7(3):243–255, 2006.

[3] M. Arnone and E. Davidson. The hardwiring of development: organization and function of genomic regulatory systems. *Development*, 124(10):18511864, 1997.

[4] M. Baitaluk, M. Sedova, A. Ray, and A. Gupta. BiologicalNetworks: visualization and analysis tool for systems biology. *Nucleic Acids Research*, 34(Web-Server-Issue):466–471, 2006.

[5] BioPAX. Biological pathways exchange, 2007. http://www.biopax.org.

[6] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. McMillan, 1976.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1985.

[8] D. Croes, F. Couche, S. J. Wodak, and J. van Helden. Metabolic PathFinding: inferring relevant pathways in biochemical networks. *Nucleic Acids Research*, 33:W326, 2005.

[9] E. Demir. *An Ontology for Computer-Aided Modeling of Cellular Processes*. PhD thesis, Department of Computer Engineering, Bilkent University, 2005.

[10] E. Demir, O. Babur, U. Dogrusoz, A. Gursoy, A. Ayaz, G. Gulesir, G. Nisanci, and R. Cetin-Atalay. An ontology for collaborative construction and analysis of cellular pathways. *Bioinformatics*, 20(3):349–356, 2004.

[11] U. Dogrusoz, E. Erson, E. Giral, E. Demir, O. Babur, A. Cetintas, and R. Colak. PATIKAweb: a Web interface for analyzing biological pathways through advanced querying and visualization. *Bioinformatics*, 22(3):374–375, 2006.

[12] K. Fukuda and T. Takagi. Knowledge representation of signal transduction pathways. *Bioinformatics*, 17(9):829–837, 2001.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[14] R. Gting. GraphDB: Modeling and querying graphs in databases. In *Proc 20th Int. Conf. on Very Large Databases*, pages 297–308, Santiago, Chile, Sept. 1994.

[15] R. Hofestädt and S. Thelen. Qualitative modeling of biochemical networks. *In Silico Biology*, 1:39–53, 1998.

[16] T. Ito, T. Chiba, R. Ozawa, M. Yoshida, M. Hattori, and Y. Sakaki. A comprehensive two-hybrid analysis to explore the yeast protein interactome. *Proceedings of the National Academy of Sciences*, 98:4569–4574, 2001.

[17] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, 2005.

[18] G. Miklos and G. Rubin. The role of the genome project in determining gene function: insights from model organisms. *Cell*, 86(4):521–529, 1996.

[19] H. Ogata, W. Fujibuchi, S. Goto, and M. Kanehisa. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucleic Acids Res.*, 28(20):40214028, 2000.

[20] V. N. Reddy, M. L. Mavrovouniotis, and M. N. Liebman. Petri net representations in metabolic pathways. In *1st International Conference on Intelligent Systems for Molecular Biology*, pages 328–336, 1993.

[21] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms : Theory and Practice*. Prentice-Hall, 1977.

[22] SBGN. Systems biology graphical notation, 2007. http://www.sbgn.org.

[23] R. Sharan and T. Ideker. Modeling cellular machinery through biological network comparison. *Nature Biotechnology*, 24(4):427–433, April 2006.

[24] T. Shlomi, D. Segal, E. Ruppin, and R. Sharan. QPath: a method for querying pathways in a protein-protein interaction network. *BMC Bioinformatics*, 7:199, 2006.

[25] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[26] K. Y. Yip, H. Yu, P. M. Kim, M. Schultz, and M. Gerstein. The tYNA platform for comparative interactomics: a web tool for managing, comparing and mining multiple networks. *Bioinformatics*, 22(23):2968–2970, 2006.

[27] H. Zhu, M. Bilgin, R. Bangham, D. Hall, A. Casamayor, P. Bertone, N. Lan, R. Jansen, S. Bidlingmaier, T. Houfek, T. Mitchell, P. Miller, R. Dean, M. Gerstein, and M. Snyder. Global analysis of protein activities using proteome chips. *Science*, 293:2101–2105, 2001.