Patterns and Logic for Reasoning with Networks

Angelika Kimmig[1], Esther Galbrun[2], Hannu Toivonen[2], and Luc De Raedt[1]

[1] Departement Computerwetenschappen, K.U. Leuven
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
{angelika.kimmig,luc.deraedt}@cs.kuleuven.be
[2] Department of Computer Science and
Helsinki Institute for Information Technology HIIT
P.O. Box 68, FI-00014 University of Helsinki, Finland
{esther.galbrun,hannu.toivonen}@cs.helsinki.fi

Abstract. Biomine and ProbLog are two frameworks to implement bisociative information networks (BisoNets). They combine structured data representations with probabilities expressing uncertainty. While Biomine is based on graphs, ProbLog's core language is that of the logic programming language Prolog. This chapter provides an overview of important concepts, terminology, and reasoning tasks addressed in the two systems. It does so in an informal way, focusing on intuition rather than on mathematical definitions. It aims at bridging the gap between network representations and logical ones.

1 Introduction

Nowadays, large, heterogeneous collections of uncertain data exist in many domains, calling for reasoning tools that support such data. Networks and logical theories are two common representations used in this context. In the setting of bisociative knowledge discovery, such networks are called BisoNets [1]. The Biomine project has constructed a large network (or BisoNet) of biological knowledge and provided several reasoning mechanisms to explore this network [2]. ProbLog [3], on the other hand, provides a logic-based representation language and corresponding inference methods that have been used in the context of the same network. Both Biomine and ProbLog allow one to associate probabilities to network edges and thereby to reason about uncertainty. For ProbLog, this idea has recently also been extended to other types of labels, such as for instance costs, connection strengths, or revenues [4]. In this chapter, we highlight the common underlying ideas of these two frameworks, focusing on illustrative examples rather than formal detail. We provide an overview of network-related inference techniques from a logical perspective. These techniques can potentially be used to support bisociative reasoning and knowledge discovery. The aim is to bridge the gap between the two views and to point out similarities and opportunities for cross-fertilization.

The chapter is organized as follows: We first introduce the Biomine and ProbLog frameworks and their underlying concepts in Section 2. Section 3 then

M.R. Berthold (Ed.): Bisociative Knowledge Discovery, LNAI 7250, pp. 122–143, 2012.

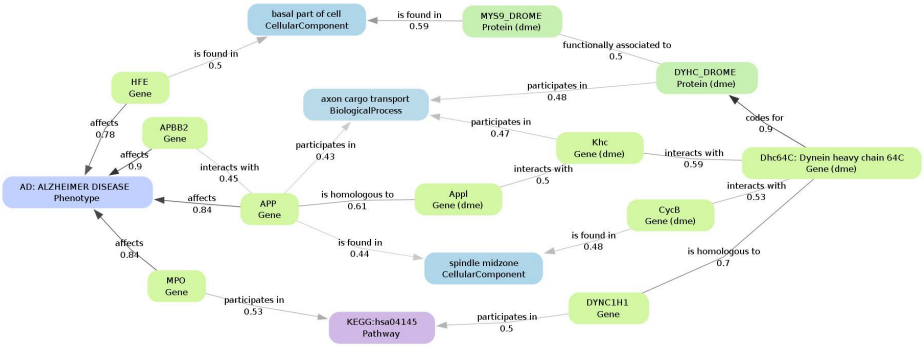

Fig. 1. An example of a subgraph extracted from `Biomine`

gives an overview of various inference and reasoning tasks, focusing on the structural aspect, before Section 4 discusses their extension towards the use of probabilities and other types of labels.

2 The `Biomine` and ProbLog Frameworks

The `Biomine` project has contributed a large network of biological entities and relationships between them, represented as typed nodes and edges, respectively [2]. The `Biomine` network is probabilistic; to each edge is associated a value that represents the probability that the link between the entities exists. A subnetwork extracted from this database is shown in Figure 1. Inspired on the `Biomine` network, ProbLog [3] extends the logic programming language Prolog with independent random variables in the form of probabilistic facts, corresponding to `Biomine`'s probabilistic edges. In the remainder of this section, we will introduce the basic terminology used in the context of these frameworks for reasoning about networks.

2.1 Using Graphs: `Biomine`

Figure 2 gives a simplified representation of the `Biomine` subnetwork of Figure 1. We will use this representation for illustration throughout the chapter. Nodes have numbers as identifiers. There are five node types (`tn1` to `tn5`). The number of edge types has been reduced to three (`te1`, `te2` and `te3`) and their directions have been removed. We use colors and border styles to represent the node types, and line styles to represent the edge types; see Figure 5 for the exact mapping.

In general, nodes and edges could have several types simultaneously. Also, edges can be directed and there may exist multiple edges between a given pair of nodes. For ease of explanation, we will only consider the simpler case where

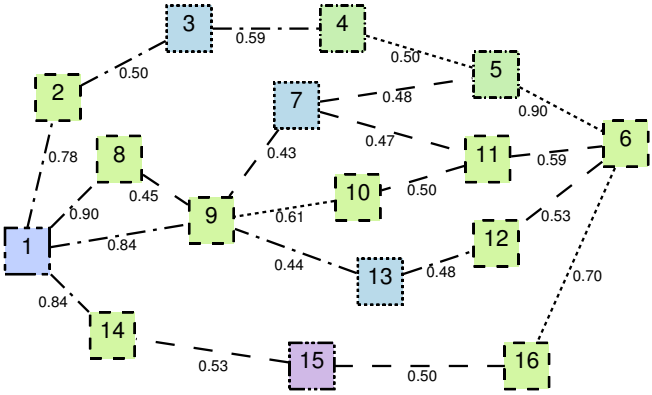

Fig. 2. A simplified representation of the `Biomine` subgraph

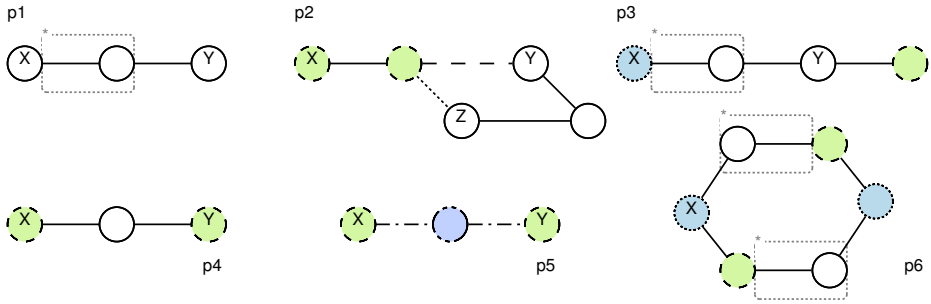

Fig. 3. Examples of graph patterns

edges and nodes have a single type and there is at most one edge between any pair of nodes.[1]

A *graph pattern* is an expression over node and edge types. It is an abstract graph that defines a subgraph by means of a set of constraints over the connection structure and edge and node types. Six example patterns, `p1` to `p6`, are presented in Figure 3.

The pattern nodes are represented using circles to distinguish them from network nodes, which are represented as squares. Pattern nodes and edges are either required to be of a given type, or can be of arbitrary type. The latter is denoted using white nodes and solid edges. *Query nodes* are labeled with capital letters, these are the main points of interest when querying the network. As in regular expressions, the star denotes unlimited repetitions of substructures.

For instance, pattern `p1` corresponds to a path of length at least one between the query nodes `X` and `Y`, using arbitrary node and edge labels, whereas `p5` specifies the exact number of edges and all edge and node types.

[1] Allowing multiple edges between the same pair of nodes can be done by introducing explicit edge identifiers, both in the network and, where needed, also in the patterns.

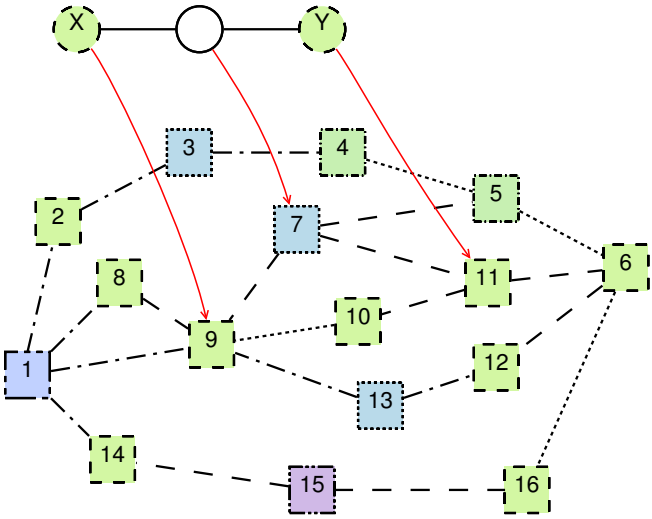

Fig. 4. Example of instantiation of pattern `p4`

A *substitution* assigns network nodes to nodes in a pattern. An *instantiation* maps a pattern onto the network using subgraph isomorphism. Thus, it is a substitution of all nodes in the pattern in such a way that a corresponding edge mapping exists as well. An *answer substitution* is a restriction of an instantiation to the query nodes.

An example instantiation of pattern `p4` is shown in Figure 4. There might be several possible instantiations of a pattern with the same answer substitution. For instance, `p4{X/9, Y/11}` can be instantiated in two ways, either by mapping the middle node to `7`, as in the illustration, or by mapping it to `10`.

While we here consider a flat type system, where types are either given or completely undefined, it is also possible to use type hierarchies. When instantiating patterns, a node (respectively an edge) can then be mapped to a node (edge) of same type or one of its descendant types. The hierarchies used in our example are shown in Figure 5, where the undefined type is the root node of the hierarchy.

2.2 Using Logic: ProbLog

As ProbLog is based on the logic programming language Prolog, we first illustrate the key concepts of Prolog by means of an example; for a more detailed introduction, we refer to [5]. We defer discussion of the probabilistic aspects of ProbLog to Section 4.

In Prolog, the network of Figure 2 (ignoring the probability labels) can be represented as a set of *facts*:

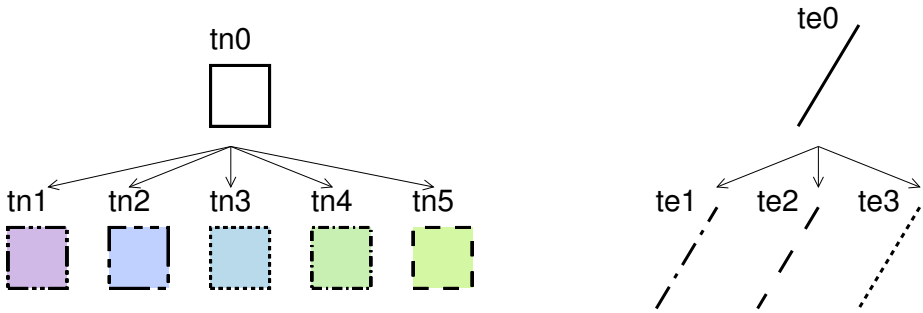

Fig. 5. Node (left) and edge (right) types hierarchies

$$\begin{array}{l} \texttt{arc}(1,2,\texttt{te1}).\quad \texttt{arc}(2,3,\texttt{te1}).\quad \texttt{arc}(1,8,\texttt{te1}). \\ \texttt{arc}(8,9,\texttt{te2}).\quad \texttt{arc}(9,10,\texttt{te3}).\quad \texttt{arc}(1,9,\texttt{te1}).\quad \ldots \\ \texttt{node}(1,\texttt{tn2}).\quad \texttt{node}(2,\texttt{tn5}).\quad \texttt{node}(3,\texttt{tn3}).\quad \ldots \end{array} \tag{1}$$

Here, `arc(1,2,te1)` states that there is a directed edge from node `1` to node `2` of type `te1`; `node(1,tn1)` specifies that node `1` is of type `tn1`, and so forth.[2] `arc/3` is a *predicate* of *arity* 3, that is, with 3 *arguments*. To obtain undirected edges, a Prolog program would define an additional predicate `edge/3` as follows:

$$\texttt{edge}(\texttt{X},\texttt{Y},\texttt{T}) \;:-\; \texttt{arc}(\texttt{X},\texttt{Y},\texttt{T}). \tag{2}$$

$$\texttt{edge}(\texttt{X},\texttt{Y},\texttt{T}) \;:-\; \texttt{arc}(\texttt{Y},\texttt{X},\texttt{T}). \tag{3}$$

Here, uppercase letters indicate *logical variables* that can be instantiated to constants such as `1` or `tn3`. The definition of `edge/3` above consists of two *clauses* or rules. The first clause states that `edge(X,Y,T)` is true for some nodes `X` and `Y` and node type `T` if `arc(X,Y,T)` is true. The second clause gives an alternative precondition for the same conclusion. Together, they provide a disjunctive definition, that is, `edge(X,Y,T)` is true if at least one of the rules is true.

For instance, `edge(2,1,te1)` is true due to the second clause and the fact `arc(1,2,te1)`, where we use the substitution {`X/2, Y/1, T/te1`} to map rule variables to constants. `edge(2,1,te1)` is said to *follow* from or to be *entailed* by the Prolog program.

More formally, Prolog answers a given query by trying to *prove* the query using the facts and clauses in the program. The answer will be `yes` (possibly together with a substitution for the query variables, which are considered to be existentially quantified), if the query follows from the program (for that substitution). Query `?- edge(2,1,te1)` results in the answer `yes` due to clause (3) and fact `arc(1,2,te1)`. For `?- edge(2,1,te2)`, the answer is `no`, as Prolog terminates without finding a corresponding fact to complete the proof. For `?- edge(A,B,C)`,

[2] Alternatively, one could also use facts such as `te1(1,2)` and `tn1(1)`.

Prolog will return the substitution {A/1, B/2, C/te1}, and will allow the user to keep asking for alternative answers, such as {A/2, B/3, C/te1}, {A/1, B/8, C/te1}, and so forth, until no more substitutions can be generated from the program.

For convenience, we also define edges of arbitrary type:

$$\texttt{edge}(\texttt{X},\texttt{Y}) \ :- \ \texttt{edge}(\texttt{X},\texttt{Y},\texttt{T}). \tag{4}$$

Alternatively, one could encode the type hierarchy in Figure 5:

$$\begin{aligned}
\texttt{edge}(\texttt{X},\texttt{Y},\texttt{te0}) \ &:- \ \texttt{edge}(\texttt{X},\texttt{Y},\texttt{te1}).\\
\texttt{edge}(\texttt{X},\texttt{Y},\texttt{te0}) \ &:- \ \texttt{edge}(\texttt{X},\texttt{Y},\texttt{te2}).\\
\texttt{edge}(\texttt{X},\texttt{Y},\texttt{te0}) \ &:- \ \texttt{edge}(\texttt{X},\texttt{Y},\texttt{te3}).
\end{aligned}$$

Prolog also allows for more complex predicate definitions, such as a path between two nodes:

$$\begin{aligned}
\texttt{path}(\texttt{X},\texttt{Y}) \ &:- \ \texttt{edge}(\texttt{X},\texttt{Y}).\\
\texttt{path}(\texttt{X},\texttt{Y}) \ &:- \ \texttt{edge}(\texttt{X},\texttt{Z}),\texttt{path}(\texttt{Z},\texttt{Y}).
\end{aligned} \tag{5}$$

The set of facts in a Prolog program are also called the *database*, and the set of clauses the *background knowledge*.

To simplify notation and to closely follow the network view, in the remainder of this chapter we assume that different logical variables are mapped onto different constants; this could be enforced in Prolog by adding atoms of the form $\texttt{X} \neq \texttt{Y}$ to predicate definitions.

So far, we have focused on encoding information about a specific network. However, Prolog allows one to encode both data and algorithms within the same logical language, and thus makes it easy to implement predicates that reason about the program itself, for instance, by simulating proofs of a query in order to generate additional information. As we will see in Section 3, this provides a powerful means to cast reasoning tasks in terms of queries; we refer to [5] for a detailed discussion.

In the logical setting, a pattern corresponds to a *predicate*. As in the graph setting, its definition imposes constraints on the types and connection structure. For example, the predicate `path(X,Y)` defined above directly corresponds to pattern `p1` in Figure 3. The *query variables* `X` and `Y` correspond to the query nodes in graph patterns. Query variables are mapped to constants using *answer substitutions* as in the network setting. Building on the definitions above, the full set of patterns in Figure 3 can be encoded as follows:

$$\texttt{p1}(\texttt{X},\texttt{Y}) \ :- \ \texttt{path}(\texttt{X},\texttt{Y}). \tag{6}$$

$$\begin{aligned}
\texttt{p2}(\texttt{X},\texttt{Y},\texttt{Z}) \ :- \ &\texttt{node}(\texttt{X},\texttt{tn5}),\texttt{edge}(\texttt{X},\texttt{A}),\texttt{node}(\texttt{A},\texttt{tn5}),\texttt{edge}(\texttt{A},\texttt{Y},\texttt{te2}),\\
&\texttt{edge}(\texttt{A},\texttt{Z},\texttt{te3}),\texttt{edge}(\texttt{Y},\texttt{B}),\texttt{edge}(\texttt{Z},\texttt{B}).
\end{aligned} \tag{7}$$

$$\texttt{p3}(\texttt{X},\texttt{Y}) \ :- \ \texttt{node}(\texttt{X},\texttt{tn3}),\texttt{path}(\texttt{X},\texttt{Y}),\texttt{edge}(\texttt{Y},\texttt{A}),\texttt{node}(\texttt{A},\texttt{tn5}). \tag{8}$$

$$\texttt{p4}(\texttt{X},\texttt{Y}) \ :- \ \texttt{node}(\texttt{X},\texttt{tn5}),\texttt{edge}(\texttt{X},\texttt{A}),\texttt{edge}(\texttt{A},\texttt{Y}),\texttt{node}(\texttt{Y},\texttt{tn5}). \tag{9}$$

$$\texttt{p5(X,Y)} :- \texttt{node(X,tn5), edge(X,A,te1), node(A,tn2),}$$
$$\texttt{edge(A,Y,te1), node(Y,tn5).} \quad (10)$$
$$\texttt{p6(X)} :- \texttt{node(X,tn3), edge(X,A), node(A,tn5), path(A,B),}$$
$$\texttt{node(B,tn3), edge(B,C), node(C,tn5), path(C,X).} \quad (11)$$

Furthermore, using logic also allows one to easily express additional constraints on patterns. For instance, pattern `p7` states that node `X` has a neighboring node whose type is not `tn5`, while pattern `p8` states that it has two outgoing edges of the same type:

$$\texttt{p7(X)} :- \texttt{edge(X,Y), not(node(Y,tn5)).}$$
$$\texttt{p8(X)} :- \texttt{edge(X,Y,T), edge(X,Z,T).}$$

For ease of presentation, we assume that patterns are always defined by a single clause. Note that this does not preclude disjunctive patterns, as can be seen for `p1` in (6) above.

A substitution θ is an *answer substitution* for a pattern p if the query $p\theta$ follows from the Prolog program. For instance, {`X/9, Y/11`} is an answer substitution for pattern `p4(X,Y)`, as `p4(9, 11)` follows from our example program.

An *explanation* for a pattern is a minimal set S of database facts such that the pattern follows from S and the background knowledge. For instance, {`node(9,tn5), arc(9,7,te2), arc(7,11,te2), node(11,tn5)`} is an explanation for `p4(9, 11)`.

2.3 Summary

Table 1 summarizes the key terms introduced in this section.

Table 1. Correspondence of the different terminologies

logic view	graphical view
background knowledge	set of patterns
set of facts, database	graph
predicate	pattern
query variables	query nodes
explanation	instantiation

3 Inference and Reasoning Techniques

This section provides an overview of a broad range of reasoning techniques. We start with the classical tasks of deduction and abduction, that are both concerned with matching given patterns against the graph or database. Next,

we discuss various settings for induction, that is, for inferring patterns under different conditions. We then in turn consider techniques that combine pattern creation and pattern matching, that identify nodes in the graph, and that modify the database or the background knowledge in a number of different settings. Throughout the discussion, we assume a Prolog program encoding the graph and possible background knowledge as discussed in Section 2.2. This allows us to view the different reasoning tasks as queries asked to a Prolog system.

3.1 Deduction: Reasoning about Node Tuples

The question answered by deduction is whether there exists an instantiation of a pattern in a graph, or, equivalently, whether the pattern follows from the Prolog program encoding the graph. It thus directly corresponds to answering Prolog queries as discussed in Section 2.2.

In our example, given the ground query `?- p2(8,7,10)`, deduction will produce an affirmative answer, as there is an instantiation of the pattern using the real nodes `9` and `11`. Similarly, `p4(8,10)` is true but `p2(10,7,8)`, `p2(9,6,11)` and `p4(14,10)` are false.

To summarize, *given a Prolog program, a pattern* `p` *and a substitution* θ *that grounds* `p`*, deduction corresponds to answering the query* `?-` `p`θ *from the program.*

The decision problem of deduction as described here forms the basis for many other reasoning tasks on the level of node tuples; we discuss some examples next.

Answer Enumeration. For non-ground patterns, the enumeration problem associated to deduction corresponds to finding *all* answer substitutions for the pattern. Alternatively, one can ask for *some* answer substitution chosen from the set of all possible ones. For instance, one possible answer substitution for `?- p2(X,Y,Z)` would be {`X/8, Y/7, Z/10`}, whereas `?- p4(15,Y)` does not produce an answer substitution, as there is no proof of this query.

Thus, *given a Prolog program and a pattern* `p`*, the answer substitution and enumeration problem of deduction correspond to finding one or all answer substitutions for the query* `?- p` *from the program, respectively.*

Representative Nodes. A binary pattern `p(X,Y)` can be used to find a set of *representative nodes*, that is, nodes `r` that, when substituted for `X`, lead to a set of patterns `p(r,Y)` such that all other nodes appear in an answer substitution for at least one such pattern. For instance, using `p(X,Y) :- edge(X,Z),edge(Z,Y)`, one set of representative nodes is $\{1, 4, 16\}$. Note that here, some nodes are associated to several representative nodes, in this example node `7` is associated to both the representative nodes `1` and `4`. A harder variant of the problem would be to require that there is exactly one such representative for each node.

In a nutshell, *given a constant* k*, a Prolog program encoding a network with nodes* N*, and a pattern* `p(X,Y)`*, the task of finding representative nodes is to find a subset* $S \subseteq N$ *of size* k *such that for each node* $y \notin S$ *there is a node* $s \in S$ *for which the query* `?- p(s,y)` *is answered affirmatively.*

Spread of Influence. Recursive patterns such as `path(X,Y)` can be used to measure distances from a given node in a network to all other nodes in terms of the minimal number of edges needed to reach the other node. This principle can be regarded as the basis of techniques measuring spread of influence, and can be used to enumerate nodes by increasing distance.

For instance, using pattern `path(1,X)`, thus measuring the distance from node `1`, the closest set of nodes is $\{2, 8, 9, 14\}$, the next one $\{3, 7, 10, 13, 15\}$, and so forth.

In Prolog, this could easily be realized by extending the path predicate with a third argument that counts the number of edges traversed:

$$\texttt{path}(\texttt{X},\texttt{Y},1) \ :- \ \texttt{edge}(\texttt{X},\texttt{Y}).$$
$$\texttt{path}(\texttt{X},\texttt{Y},\texttt{L}) \ :- \ \texttt{edge}(\texttt{X},\texttt{Z}),\texttt{path}(\texttt{Z},\texttt{Y},\texttt{L}),\texttt{L is N}+1.$$

One would then ask a sequence of queries `?- path(1,X,i)` with $\texttt{i} = 1, \ldots, n$ up to a maximum length n, though some extra book-keeping would be required to filter out nodes that have been returned as an answer on previous levels already.

Thus, *given a Prolog program, a maximum distance n, and a recursive pattern* $\texttt{p}(\texttt{x},\texttt{Y},\texttt{D})$ *with source node* `x`*, spread of inference corresponds to answer enumeration for the sequence of queries* `?- p(x,Y,i)` *for* $\texttt{i} = 1, \ldots, n$.

3.2 Abduction: Reasoning about Subgraphs

The task of abduction is closely related to that of generating an explanation for a query as discussed in Section 2.2. In terms of graphs, it directly corresponds to finding a minimal instantiation of a pattern. In the logical setting, abduction is not restricted to database predicates, but can use all predicates marked *abducible*. In the context of patterns and networks, one could simply assume all predicates used in pattern definitions to be abducible and implement a predicate `abduce`; see [5] for a general definition of this predicate and more details.

For instance, when calling the query `?- abduce(p6(7),E)` (cf. Equation (11)) and assumming that all predicates are abducible, the answer `E` would be the conjunction of `node(7,tn3)`, `edge(7,9)`, `node(9,tn5)`, `path(9,13)`, `node(13,tn3)`, `edge(13,12)`, `node(12,tn5)`, and `path(12,7)`.

To summarize, *given a Prolog program and a pattern* `p`*, abduction corresponds to answering the query* `?- abduce(p,E)`.

Again, one can also consider the corresponding enumeration problem, where the task is to find all explanations or instantiations.

3.3 Induction: Finding Patterns

Frequent Patterns. The usual frequent subgraph mining problem corresponds to the problem of finding all patterns from a given pattern language with more than a chosen number of instantiations. The pattern language will specify both allowed structures of patterns and which nodes in patterns can be query nodes.

For instance, for a frequency threshold of 3, all patterns in Figure 3 would be frequent.

Similarly to `abduce/2` above, one could implement a Prolog predicate

$$\texttt{frequent}(\texttt{P},\texttt{T}) \ :- \ \texttt{pattern}(\texttt{P}),\texttt{count}(\texttt{P},\texttt{N}),\texttt{N} >= \texttt{T}.$$

that returns patterns with a frequency greater or equal to a user-defined threshold `T`. It relies on suitable definitions of `pattern/1` (defining elements of the pattern language) and `count/2` (counting instantiations of a given pattern). Then, finding frequent patterns for a given frequency threshold `t` corresponds to answering the query `?- frequent(P,t)`. While this simple approach illustrates the basic idea, an efficient implementation would clearly be more involved.

To summarize, *given a frequency threshold* `t` *and a Prolog program including definitions of a pattern language and a counting function, finding frequent patterns corresponds to answering the query* `?- frequent(P,t)`.

Concept Learning. The aim of concept learning is to construct a definition of a new predicate that covers all positive examples, but none of the negative ones. In our context, examples are node tuples, but for convenience we represent them as ground instances of the pattern to be found. Again, this could be realized in Prolog based on a suitable definition of a predicate

$$\begin{aligned}\texttt{concept}(\texttt{C}) \ :- \ & \texttt{hypothesis}(\texttt{C}),\\ & \texttt{findall}(\texttt{P},(\texttt{pos}(\texttt{P}),\texttt{not}(\texttt{covers}(\texttt{C},\texttt{P}))),[]),\\ & \texttt{findall}(\texttt{N},(\texttt{neg}(\texttt{N}),\texttt{covers}(\texttt{C},\texttt{N})),[]).\end{aligned}$$

Here, `hypothesis/1` enumerates possible concepts, `covers/2` checks whether the concept covers an example, and `pos/1` and `neg/1` define examples. The Prolog builtin `findall/3` is used here to verify that there is no positive example that is not covered by the concept, and no negative one that is covered. In general, its third argument is a list of all instantiations of the variable in the first argument for which the query in the second argument holds, and `[]` denotes the empty list.

For instance, assume we are given examples `pos(q(3,1))`, `pos(q(7,15))` and `neg(q(7,4))`. Then, querying `?- concept(C)` could return `C = (q(X,Y) :- p3(X,Y))` as a possible solution.

Thus, *given a Prolog program including definitions of a hypothesis language and positive and negative examples, concept learning corresponds to answering the query* `?- concept(C)`.

Generalisation. Comparing patterns based on a *generality* relation provides a means to choose between alternative solutions. Given a Prolog program including two patterns

$$\begin{aligned}\texttt{p}_\texttt{a} \ :- \ & \texttt{a}_1,\ldots,\texttt{a}_\texttt{n}.\\ \texttt{p}_\texttt{b} \ :- \ & \texttt{b}_1,\ldots,\texttt{b}_\texttt{m}.\end{aligned}$$

$\mathtt{p_a}$ is *more general* than $\mathtt{p_b}$ if the query `?-` $\mathtt{p_a}$ follows from the program that is obtained by adding the facts $\mathtt{b_1}$ to $\mathtt{b_m}$ to the original program, where each variable is replaced by a new constant symbol. For instance, `p4(X,Y)` (Eq. (9)) is more general than `p5(X,Y)` (Eq. (10)), as `?- p4(x,y)` can be proven from our example program extended with the facts `node(x,tn5)`, `edge(x,a,te1)`, `node(a,tn2)`, `edge(a,y,te1)`, and `node(y,tn5)`.

From the perspective of graphs, generality can again be seen as a form of subgraph isomorphism, this time between patterns where the nodes and edges of the more general pattern are mapped to those of the more specific one of same type or children type. Notice that in the literature on logical and relational learning there are multiple notions of generality that can be employed [6].

The notion of generality can also be used to find a maximally specific common generalisation of two given patterns, that is, a pattern that is more general than each of the input patterns, but for which there is no more specific pattern that also fulfills this criterion. A corresponding Prolog predicate `generalize/3`, queried as `?- generalize(p2(X,Y),p4(X,Y),G)` would provide the answer `C = (node(X,tn5), edge(X,A), edge(A,Y))`.

Thus, *given a Prolog program and two patterns* $\mathtt{p_a}$ *and* $\mathtt{p_b}$*, the task of generalization corresponds to answering the query* `?- generalize(`$\mathtt{p_a}$`,`$\mathtt{p_b}$`,P)`.

Clustering. Patterns can also be used to *cluster* node tuples: all node tuples that satisfy a given pattern fall into the same cluster. The task of clustering a given set of node tuples then corresponds to that of finding k patterns that cluster the node tuples into k disjoint (or possibly overlapping) subsets based on characteristics of their local connection.

A very simple set of clustering patterns in our example would be the set containing `node(X,tni)` for $i = 1, \ldots, 5$ that would simply cluster single nodes by their types.

In a nutshell, *given a Prolog program, a constant k and a set of node tuples T, clustering is the task of finding a set P of k patterns such that for each $t \in T$, there is exactly one pattern $p \in P$ for which t is an answer substitution for p.*

3.4 Combining Induction and Deduction

As deduction matches patterns against the database, while induction constructs new patterns, the two approaches can be naturally combined to find both patterns and corresponding substitutions simultaneously.

Analogy. Node tuples can be considered analogous if they are answer substitutions for the same pattern. Given a substitution, the problem of finding analogous tuples can be defined as finding a pattern for which this substitution is an answer substitution along with all other answer substitutions for it. The more specific the pattern, the stronger the analogy.

For example, the pairs of nodes `(2,8)`, `(12,16)`, `(14,9)` and `(9,11)` are analogous with respect to pattern `p4`, i.e., in the sense that they are all pairs

of nodes of type `tn5` separated by an intermediate node. However, pattern `p5` defines a stronger analogy that only relates the pairs `(2,8)` and `(14,9)`. In the case of an asymmetric pattern, it can be interesting to consider the sets of real nodes assigned to one particular query node in the pattern, in other words, nodes that take the same role in the analogy.

That is, *given a Prolog program and a substitution θ, the task of reasoning by analogy is to find a pattern* `p` *for which θ is an answer substitution as well as the set S of all answer substitutions for* `p`.

Synonyms. Two structurally distinct patterns are *synonyms* of one another if they have the same answer substitutions. Synonyms are also known as *re-descriptions* or *syntactic variants*. Finding synonym patterns and their answer substitutions can be one way of finding sets of objects of special interest. Furthermore, given two networks with node and edge types from different domains, finding synonyms can help to establish mappings between these domains.

For instance, the following two patterns are synonyms (albeit only covering a single node due to the simplicity of the example graph):

$$\texttt{s1(X)} :- \texttt{edge(X,Y,te1),edge(X,Z,te1),edge(Y,Z)}.$$
$$\texttt{s2(X)} :- \texttt{node(X,tn2)}.$$

Thus, *given a Prolog program, finding synonyms means finding a set of patterns P such that each pattern in P has the same set of answer substitutions.*

3.5 Modifying the Knowledge Base

We now turn to a set of techniques that modify the graph, database, or background knowledge. The key difference to the techniques discussed so far is that we now allow for loosing information.

Graph Simplification. The goal of graph simplification is to remove *redundant* edges from a graph. Here, redundancy is defined with respect to paths: an edge is considered redundant if all pairs of nodes connected in the original graph are also connected in the graph after removing the edge. In the purely structural case, graph simplification thus corresponds to finding a spanning tree; we will come back to the use of additional quality measures in Section 4.6.

That is, *given a Prolog program including a set E of facts representing edges and a predicate* `path/2`*, graph simplification finds a minimal set $S \subseteq E$ such that the set of answer substitutions for* `path(X,Y)` *remains the same when reducing E to S in the program.*

Subgraph Extraction. The aim of subgraph extraction is to find a subgraph of a given maximal size while retaining as much information as possible with respect to a given set of examples, that is, answer substitutions for a pattern.

For instance, given examples `path(3,7)` and `path(3,13)` and an upper limit of 5 edges, our network could be compressed to contain `edge(2,3)`, `edge(1,2)`, `edge(1,9)`, `edge(9,7)` and `edge(9,13)` only.

Thus, *given a Prolog program including a set E of facts representing edges, a constant k and a set T of answer substitutions for pattern* `p`*, subgraph extraction finds a set $S \subseteq E$ of size at most k such that all $\theta \in T$ are also answer substitutions for* `p` *when reducing E to S in the program.*

Abstraction. The task of abstraction is to rewrite the database using new predicates that abstract away some of the information present in the initial database. While techniques such as graph simplification and subgraph extraction also loose information, abstraction differs in that it replaces database predicates by a new predicate, obtained by computing answers for the pattern defining the new predicate. For instance, one could replace the predicate `arc/3`, that is, the directed, typed edges, using

$$\mathtt{p(X,Y)} \ :- \ \mathtt{node(X,T), node(Y,T), T \neq tn5, path(X,Y).}$$

that is, edges that correspond to paths between pairs of nodes of the same type (different from `tn5`) in the original network. This would result in the new database

$$\begin{array}{llll} \mathtt{p(3,7).} & \mathtt{p(3,13).} & \mathtt{p(7,13).} & \mathtt{p(4,5).} \\ \mathtt{p(7,3).} & \mathtt{p(13,3).} & \mathtt{p(13,7).} & \mathtt{p(5,4).} \end{array}$$

Abstractions can be created using any technique that identifies patterns and thus predicate definitions. Instead of adding the definitions of these predicates to the database, it computes all groundings of the new predicate, adds these to the database, and deletes the old facts.

Thus, *given a Prolog program, a database predicate* `d` *and a pattern* `p`*, abstraction adds* $\mathtt{p}\theta$ *for all answer substitutions θ for* `p` *to the program and deletes the definition of* `d`*.*

Predicate Invention. The key idea of predicate invention is to introduce new patterns that can be used to represent the background knowledge more compactly. For instance, the DUCE system [7] measures compactness using the minimum description length principle. As an example, consider the following set of rules:

$$\begin{array}{l} \mathtt{q1(Z)} \ :- \ \mathtt{edge(Z,Y), edge(Y,X), edge(X,W), edge(W,V), node(V,tn1).} \\ \mathtt{q2(Z)} \ :- \ \mathtt{edge(Z,Y), edge(Y,X), edge(X,W), edge(W,V), node(V,tn2).} \\ \mathtt{q3(Z)} \ :- \ \mathtt{edge(Z,Y), edge(Y,X), edge(X,W), edge(W,V), node(V,tn3).} \end{array}$$

Inventing a predicate `dist4(X,Y)` allows one to rewrite these definitions more compactly as

```
dist4(Z,V) :- edge(Z,Y),edge(Y,X),edge(X,W),edge(W,V).
q1(Z) :- dist4(Z,V),node(V,tn1).
q2(Z) :- dist4(Z,V),node(V,tn2).
q3(Z) :- dist4(Z,V),node(V,tn3).
```

While this transformation has preserved the meaning of the original fragment, this need not be the case in general. Similar principles can also be used to compress graphs by replacing instantiations of a pattern by new nodes [8].

In general, *given a Prolog program, the task of predicate invention is to introduce new pattern definitions which are then used to rewrite the program more compactly.*

3.6 Summary

Table 2 summarizes the different reasoning techniques presented in previous sections. It recapitulates the information provided to and the problem solved by each of them.

4 Using Probabilistic or Algebraic Labels

So far, we have restricted our discussion to crisply defined networks and logical theories. However, in both `Biomine` and ProbLog, the information provided is uncertain. This uncertainty is expressed by attaching probabilities to edges or facts, and can be exploited in various ways for reasoning. Furthermore, ProbLog has recently been generalized to aProbLog [4], where probabilities can be replaced by other types of labels, such as costs or distances. In this section, we first briefly review the probabilistic model underlying `Biomine` and ProbLog, and then illustrate how the techniques from Section 3 can benefit from the probabilistic setting. While some of these techniques have already been realized in `Biomine`, ProbLog, or other probabilistic frameworks, for others, the details of such a transfer are still open. Finally, we touch upon the perspectives opened by aProbLog.

4.1 The Probabilistic Model of Biomine and ProbLog

In the probabilistic graph model underlying `Biomine`, a value is associated to each edge, indicating the probability that the relationship exists. In `Biomine`, these values are obtained as the product of three factors, indicating the *reliability*, the *relevance*, and the *rarity* (or specificity) of the information, cf. [2], but they can be obtained in a different way as well. Existences of the edges are considered independent from each other. This actually defines a probability distribution over

Table 2. Summary of the different reasoning methods

Sec.	Methods	Information	Problem
3.1	Deduction	pattern, substitution	is it an answer substitution?
3.1	Answer Enumeration	pattern	list all answer substitutions
3.1	Representative Nodes	pattern, integer k	find k representative nodes
3.1	Spread of Influence	recursive pattern, node	enumerate nodes by distance
3.2	Abduction	pattern	find an/all instantiation(s)
3.3	Frequent Patterns	frequency threshold	list all frequent patterns
3.3	Concept Learning	pos./neg. examples	find a discriminative pattern
3.3	Generalisation	two patterns	find a generalized pattern
3.3	Clustering	substitutions, integer k	find k clustering patterns
3.4	Analogy	substitution	find a pattern and answer substitutions
3.4	Synonyms		find a set of patterns with same answer substitutions
3.5	Graph Simplification		maximally reduce graph keeping answer substitutions for path
3.5	Subgraph Extraction	examples, integer k	reduce graph to size $\leq k$ respecting examples
3.5	Abstraction	database predicate, pattern	replace predicate definition by pattern instances
3.5	Predicate invention		reduce program size via new predicates

possible subnetworks, i.e., deterministic instances of the probabilistic network. Each subnetwork E_i has probability

$$P(E_i) = \prod_{x \in E \setminus E_i} (1 - p_x) \prod_{x \in E_i} p_x \tag{12}$$

where E is the set of edges in the probabilistic network, E_i is the set of edges realised in the deterministic instance and p_x the existence probability of edge x. For instance, the network in Figure 6 has probability (starting with the edges involving node 1) $0.78 \cdot (1 - 0.9) \cdot 0.84 \cdot 0.84 \cdot \ldots = 1.237e - 06$.

In ProbLog, probabilities are associated to ground facts instead of edges, and again, these facts are considered to correspond to independent random variables. The directed edges of (1) are now represented as follows:

$$\texttt{0.78} :: \texttt{arc}(1,2,\texttt{te1}). \quad \texttt{0.50} :: \texttt{arc}(2,3,\texttt{te1}). \quad \texttt{0.90} :: \texttt{arc}(1,8,\texttt{te1}).$$
$$\texttt{0.45} :: \texttt{arc}(8,9,\texttt{te2}). \quad \texttt{0.61} :: \texttt{arc}(9,10,\texttt{te3}). \quad \texttt{0.84} :: \texttt{arc}(1,9,\texttt{te1}).$$

In analogy to Equation (12), ProbLog thus defines a probability distribution over instances E_i of a probabilistic database with facts E.

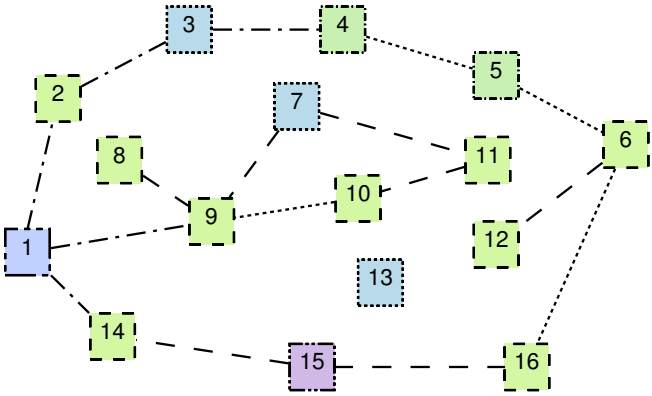

Fig. 6. A network sampled from the probabilistic graph in Figure 2

One can now ask for the probability of a specific pattern instantiation, which corresponds to the probability that this subgraph is present in a randomly sampled network. Due to the independence assumption, this probability is obtained by simply multiplying the probabilities of the instance's edges. Put differently, it corresponds to the sum of probabilities of all subnetworks of the probabilistic network that contain the instance. For example, the probability of the instantiation of pattern `p4(9,11)` presented in Figure 4 is $0.43 \cdot 0.47 = 0.2021$. The probability of its instantiation using node `10` as the middle node instead is $0.61 \cdot 0.50 = 0.305$.

The same principle of summing over all relevant subnetworks is also used to define the probability of a pattern q, called *success probability* in ProbLog:

$$P_s(q) = \sum_{E_i \subseteq E : q \text{ follows from } E_i} P(E_i). \tag{13}$$

Clearly, directly following this definition to calculate probabilities is infeasible in any network of realistic size. However, several alternative approaches have been developed, either based on sampling large numbers of networks or on enumerating pattern instantiations instead of full subnetworks. The latter approach, followed by ProbLog, requires to address the *disjoint-sum-problem*, that is, the fact that more than one instantiation of the same pattern can exist in the same subnetwork. It is therefore not possible to simply sum the probabilities of all instantiations, as this would count such subnetworks multiple times. Consider again the two instantiations of pattern `p4(9,11)` above. There are many subnetworks that allow for both instantiations (including the one in Figure 6), and we thus cannot simply sum these probabilities. Instead, we could split the relevant set of subnetworks into three disjoint parts based on the edges occurring in the instantiations: (1) all networks including the edges between `9` and `7` and between `7` and `11`, with probability 0.2021, (2) all networks that do not contain the edge between `7` and `9`, but the edges between `9` and `10` and between `10` and

11, with probability $(1 - 0.43) \cdot 0.61 \cdot 0.50 = 0.17385$, and (3) all networks including edges between 9 and 10, 10 and 11, 7 and 9, but not the one between 7 and 11, with probability $0.61 \cdot 0.50 \cdot 0.43 \cdot (1 - 0.47) = 0.0695095$. (1) includes all networks that allow for the first instantiation (regardless of the second), (2) and (3) those that allow for the second, but not the first. Thus, the total probability is $0.2021 + 0.17385 + 0.0695095 = 0.4454595$.

In practice, ProbLog represents all instantiations of the pattern as a propositional formula, and then uses advanced data structures to calculate the probability of this formula; we refer to [9] for the technical details.

While the success probability takes into account all instantiations of a pattern, it is also possible to approximate it using its most probable instantiation only.

4.2 Probabilistic Deduction

While deduction in the classical sense is concerned with deciding whether a substitution is an answer substitution for a given pattern in the network, in a probabilistic setting, it asks for the *probability* that this is the case, and thus solves Equation (13).

Answer Enumeration. When considering the set of all answer substitutions for a pattern, probabilities provide a natural means of ranking these. For instance, each answer substitution for `p5(X,Y)` corresponds to a single instantiation, that is, two edges linking node `1` to two of its neighbors. The most likely answer substitutions (omitting symmetric cases for brevity) thus are {`X/8,Y/9`} and {`X/8,Y/14`}, each with probability $0.9 \cdot 0.84 = 0.756$, followed by {`X/9,Y/14`} (probability 0.7056), {`X/2,Y/8`} (0.702), and finally {`X/2,Y/9`} and {`X/2,Y/14`} (0.6552 each).

Non-redundant Set of Representatives. Finding a non-redundant set of representatives as proposed in [10] consists in solving the representative nodes problem (cf. Section 3.1) in a probabilistic setting.

Using the `path` predicate, the aim is to find a set X of k representative nodes such that the probability that {`Y/y`} is an answer substitution for `path(x,Y)` for some `x` in X is maximum for each original node `y`. More formally, the set of representative nodes is defined as

$$argmax_{X \subset N, |X|=k} \sum_{y \in N} max_{x \in X} P_{path(x,y)}$$

where $P_{path(x,y)}$ is the probability of the best instance of `path(x,y)`, namely the most probable path between nodes `x` and `y`.

Spread of Influence. Instead of using the number of recursive steps or the size of the instantiation as a measure for the distance, in a probabilistic context, spread of inference can use the probability that a substitution is an answer substitution for a pattern. It would thus prefer more distant nodes (in terms of path length) if their probability of being connected to the source node is higher.

4.3 Probabilistic Abduction and Top-k Instantiations

In the presence of probabilities, one might not be interested in finding an explanation for a query but rather in finding the most probable one.

In that setting an interesting alternative to the enumeration problem of abduction is the task of finding the k most probable instantiations of a given pattern.

Note that identifying the k most probable instantiations of a pattern might return rather uninteresting results if they are all about the same node tuple. In order to obtain a more diverse set of answers one might look for the k tuples with most probable instantiations instead (corresponding to deductive answer enumeration approximating probabilities by those of the most likely instantiations), or even require the tuples to not overlap.

4.4 Patterns and Probabilities

When looking for patterns, probabilities can again provide a natural way to select between various alternative solutions.

Pattern Mining. Probabilistic local pattern mining in ProbLog [11] extends pattern mining in multi-relational databases to the probabilistic setting. Instead of a counting function, it uses a scoring function based on the probabilities of candidate patterns on given node tuples. It thus basically replaces the 0/1-membership function of frequent pattern mining with a gradual one based on probabilities. Probabilities of individual instances are combined using sum (resulting in a kind of probabilistic frequency) or product (resulting in a kind of likelihood function).

Concept Learning. Concept learning in the context of ProbLog has been studied in [12], where the relational rule learner FOIL is lifted to work with probabilistic data and examples.

Generalisation. In a probabilistic setting, generalisation can be used in different contexts and ways. For instance, probabilistic explanation based learning in ProbLog [13] generalizes an explanation of an example query in terms of database predicates by replacing constants by variables, thus obtaining a new pattern definition. Stochastic logic programs, a probabilistic logic language inspired on probabilistic grammars, can be learned from examples in the form of proofs by generalizing pairs of clauses extracted from these examples [14]. In the latter case, probabilities are associated to clauses, and need to be adapted during generalisation as well.

Clustering. In the context of clustering, probabilities can express the degree to which an example belongs to a cluster. One would then no longer require that node tuples are assigned to single clusters.

4.5 Combining Induction and Deduction

Analogy. While the generality of a pattern provides a first means to assess the strength of an analogy, the probabilities of the different groundings additionally provide a means to rank all node tuples that are analogous with respect to a certain pattern. For instance, while `(2,8)`, `(12,16)`, `(14,9)` and `(9,11)` are all analogous with respect to pattern `p4`, the probabilities are much higher for `(2,8)` and `(14,9)` than for the other two pairs. In the context of ProbLog, both local pattern mining [11] and probabilistic explanation based learning [13] have been used for reasoning by analogy.

Synonyms. In the context of finding synonyms, probabilities allow for choosing a subset of candidate synonyms based on the probabilities of the corresponding answer substitutions, and to thus restrict a possibly large set of synonyms to a set that is more suitable for manual inspection.

4.6 Modifying the Probabilistic Knowledge Base

Simplification of a Probabilistic Graph. The problem introduced by Toivonen *et al* [15] consists in simplifying probabilistic networks while maintaining the connectivity. It refines the task of graph simplification as defined in Section 3.5 by using the probabilities as an additional quality measure.

The aim is to find a minimal database by dropping edges while keeping the probability of the `path` predicate for each pair of nodes constant. With the probability of `path(x,y)` for a pair of nodes `x` and `y` defined as the probability of the best instantiation, this corresponds to maintaining the best paths between all pairs of nodes.

This definition might be too strict, as it might not allow for significant reductions of database size. In a later work [16], the condition is relaxed to maintaining the overall best path quality as close to the original as possible.

Subgraph Extraction. Various approaches to extract subgraphs with strong connections among given nodes have been developed in the context of `Biomine` and ProbLog [17,18,19,20]. These works all aim at maintaining high probabilities for connections between selected nodes. In `Biomine`, connections are typically defined as paths between pairs of nodes from a given set, while ProbLog theory compression [18] provides them as positive examples in the form of ground patterns whose definitions are included in the background knowledge. The latter also takes into account corresponding negative examples by using a score that encourages high probabilities for positive and low probabilities for negative examples.

Abstraction. In a probabilistic database or network, abstraction would need to take into account the probability labels as well. However, simply labeling the new facts with their probabilities as deduced from the old program may introduce hidden dependencies between facts that might be undesirable.

Predicate Invention. When applying predicate invention to a probabilistic database, the probabilities provide a means to measure the information loss and balance it against the compactness of the representation obtained. While the underlying probability distributions could be maintained for transformations that maintain the meaning of the program, how to adapt probabilities for transformations that generalize the program is an open question.

4.7 Beyond Probabilities

While probability labels provide one way of defining a quality measure on different subnetworks or databases, in certain situations, it can be more convenient to use different types of labels, such as for instance costs, capacities, or numbers of co-occurrences. For instance, in the context of a transportation network where edges are labeled with travel times, prices, or the number of available seats, one could be interested in shortest or cheapest routes, or in routes allowing for the largest group of passengers traveling together, or even in some criterion balancing these requirements. In a co-authorship graph where edges are labeled with the number of joint papers, one could be interested in patterns suggesting strong collaboration networks.

aProbLog [4] generalizes ProbLog to labels from arbitrary commutative semirings, that is, sets of labels together with two binary operators with certain characteristics.[3] Multiplication is used to define labels of subsets of the database sets (as done for the semiring of probabilities in Equation (12)), while addition is used to define labels of queries in terms of these (as done in Equation (13)). In the case of probabilities, negative literals are naturally labeled with $1-p$, where p is the label of the database facts; in the general case considered in aProbLog, these labels need to be given explicitly. By replacing summation with maximization, one obtains another probabilistic semiring that can be used to obtain most likely database instances. The examples given above can be formalized in this framework.

Inference in aProbLog generalizes that in ProbLog, and the framework thus allows one to explore the tasks discussed in this chapter in the context of different types of labels on basic relations without the need to redefine the underlying machinery.

5 Conclusions

We have given an overview of network inference tasks from the perspective of the `Biomine` and ProbLog frameworks. These tasks provide information at the node, subgraph, or pattern level, and they differ in the types of input they assume in addition to the basic graph, such as training examples or background knowledge.

[3] More formally, a commutative semiring is a tuple $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ where *addition* $\oplus$ and *multiplication* $\otimes$ are associative and commutative binary operations over the set $\mathcal{A}$, $\otimes$ distributes over $\oplus$, $e^{\oplus} \in \mathcal{A}$ is the neutral element with respect to $\oplus$, $e^{\otimes} \in \mathcal{A}$ that of $\otimes$, and for all $a \in \mathcal{A}$, $e^{\oplus} \otimes a = a \otimes e^{\oplus} = e^{\oplus}$.

They all have been or can be extended to exploit the probabilistic information present in both frameworks, or other types of labels as supported in aProbLog, a recent generalization of ProbLog to algebraic labels.

Acknowledgments. A. Kimmig is supported by the Research Foundation Flanders (FWO Vlaanderen). This work is partially supported by the GOA project 2008/08 Probabilistic Logic Learning and by the European Commission under the 7th Framework Programme, contract no. BISON-211898. This work has been supported by the Algorithmic Data Analysis (Algodan) Centre of Excellence of the Academy of Finland (Grant 118653).

References

1. Kötter, T., Berthold, M.R.: From Information Networks to Bisociative Information Networks. In: Berthold, M.R. (ed.) Bisociative Knowledge Discovery. LNCS (LNAI), vol. 7250, pp. 33–50. Springer, Heidelberg (2012)
2. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link Discovery in Graphs Derived from Biological Databases. In: Leser, U., Naumann, F., Eckman, B. (eds.) DILS 2006. LNCS (LNBI), vol. 4075, pp. 35–49. Springer, Heidelberg (2006)
3. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2462–2467 (2007)
4. Kimmig, A., Van den Broeck, G., De Raedt, L.: An algebraic Prolog for reasoning about possible worlds. In: Burgard, W., Roth, D. (eds.) Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011), pp. 209–214. AAAI Press (2011)
5. Flach, P.: Simply logical - Intelligent Reasoning by Example. John Wiley (1994), `http://www.cs.bris.ac.uk/~flach/SimplyLogical.html`
6. De Raedt, L.: Logical and Relational Learning. Springer (2008)
7. Muggleton, S.: Duce, an oracle-based approach to constructive induction. In: McDermott, J.P. (ed.) Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI 1987), pp. 287–292 (1987)
8. Cook, D.J., Holder, L.B.: Substructure discovery using minimum description length and background knowledge. J. Artif. Intell. Res. (JAIR) 1, 231–255 (1994)
9. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming (TPLP) 11(2-3), 235–262 (2011)
10. Langohr, L., Toivonen, H.: Finding representative nodes in probabilistic graphs. In: Proceedings of the Workshop on Explorative Analytics of Information Networks at ECML PKDD, WEAIN 2009 (2009)

11. Kimmig, A., De Raedt, L.: Local query mining in a probabilistic Prolog. In: Boutilier, C. (ed.) Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 1095–1100 (2009)
12. De Raedt, L., Thon, I.: Probabilistic Rule Learning. In: Frasconi, P., Lisi, F.A. (eds.) ILP 2010. LNCS, vol. 6489, pp. 47–58. Springer, Heidelberg (2011)
13. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic Explanation Based Learning. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 176–187. Springer, Heidelberg (2007)
14. De Raedt, L., Kersting, K., Torge, S.: Towards learning stochastic logic programs from proof-banks. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 752–757. AAAI Press/The MIT Press (2005)
15. Toivonen, H., Mahler, S., Zhou, F.: A Framework for Path-Oriented Network Simplification. In: Cohen, P.R., Adams, N.M., Berthold, M.R. (eds.) IDA 2010. LNCS, vol. 6065, pp. 220–231. Springer, Heidelberg (2010)
16. Zhou, F., Mahler, S., Toivonen, H.: Network simplification with minimal loss of connectivity. In: Webb, G.I., Liu, B., Zhang, C., Gunopulos, D., Wu, X. (eds.): Proceedings of the 10th IEEE International Conference on Data Mining (ICDM 2010), pp. 659–668 (2010)
17. Hintsanen, P.: The Most Reliable Subgraph Problem. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS (LNAI), vol. 4702, pp. 471–478. Springer, Heidelberg (2007)
18. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic Prolog programs. Machine Learning 70(2-3), 151–168 (2008)
19. Hintsanen, P., Toivonen, H.: Finding reliable subgraphs from large probabilistic graphs. Data Mining and Knowledge Discovery 17(1), 3–23 (2008)
20. Kasari, M., Toivonen, H., Hintsanen, P.: Fast Discovery of Reliable k-terminal Subgraphs. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part II. LNCS, vol. 6119, pp. 168–177. Springer, Heidelberg (2010)