Abstract. There has recently been a lot of interest in learning from graphs. Most approaches to this problem up to this point have been pragmatic. While there definitely exists a need for such research, theoretically sound approaches that yield comprehensible theories of higher expressive power are also desirable.

These are strong points of ILP, so it seems a good starting point for such an approach. Thus, in order to express hypotheses about graphs, a graph logic has to be chosen. A good candidate is Cardelli et al’s GL, since it has decent expressive power while retaining acceptable computational complexity. The most important operator in this logic is composition, which non-deterministically composes (splits) a graph in two parts. This makes the logic very flexible, since it allows quantification over subgraphs with specified properties, but this power comes at a price.

We argue that a restricted form of composition is much more useful for our purposes, and that little expressive power is sacrificed as long as recursive theories are allowed.

1 Introduction

There has recently been a lot of interest in learning from graphs and structured data. Most approaches to this problem up to this point have been more or less pragmatic, i.e., focused on creating algorithms that achieve decent accuracy on large datasets, using just a limited amount of computational resources (see [CH06] for a recent collection of this kind of work). Such algorithms tend to yield theories about their domain that have low expressive power, and sometimes even take the form of just a set of frequent subpatterns.

While there definitely exists a need for such research, it is also desirable to have approaches that can be shown to be mathematically sound, while yielding comprehensible theories of higher expressive power. Such approaches of course have an intrinsic theoretical interest, but certainly offer the perspective of useful applications as well.

Since soundness and comprehensible theories are strong points of ILP, techniques from this field are an obvious good starting point for such an approach. Thus, in order to express hypotheses about graphs, a graph logic has to be chosen. A good candidate is Cardelli et al’s GL and its variants, since it has decent
expressive power (between FO and MSO) while retaining acceptable computational complexity to be usable as query language. The most interesting operator in this logic is composition, which non-deterministically composes (splits) a graph in two parts. This makes the logic very flexible, since it allows quantification over subgraphs with specified properties, but this power of course comes with a pricetag.

In this paper, we argue that in practice, \( GL \) as programming language requires more computational resources than \( GL \) used as query language. We also argue that composition yields problems when defining an lgg operator for graphs as logical objects, since such an operator could yield an exponential number of lgs. So, a restricted (linear) form of composition is much more useful for our purposes, and that little expressive power is sacrificed as long as recursive theories are allowed. In other words, the computational burden can be shifted from the operator to the more extensive use of recursive predicates. Although learning recursive theories cannot be considered ‘solved’, it is at least a familiar research problem. Some methods have been proposed to tame its complexity, and these could be readily be applied in our framework.

2 The graph logic \( GL \) and its variants

The spatial graph logic \( GL_\mu \) was introduced in [CGG01,CGG02]. Its expressive power was studied in [DGG07], which introduced the notation \( GL \) and \( GL_\mu \) for the variants without and with the least fixed-point operator, respectively, and for their linear variants \( LGL \) and \( LGL_\mu \).

Of course, alternative logics have been known for a long time (first order language of graphs, MSO logic for graphs) as well as alternative query languages (TQL, Strudel and Graphlog). The main reason we are interested in \( GL \) (and its variants) is that it allows graphs to be treated as logical objects and that it expresses properties of graphs in a very direct and concise way. This makes it an interesting candidate for ILP-like applications, since it is expected to introduce a language bias that is easily interpretable in terms of graph properties, enhancing comprehensibility. This is true both of the theories generated by a learning algorithm and the background theory that the user might specify in order to define a language bias.

3 The Query Language

The core of the graph query language consists of first order logic, with quantification restricted to edge label- (\( \alpha \)) and node variables (\( \xi \)), an ‘edge’ predicate written as \( o(\xi_1, \xi_2) \), and the (graph) composition operator \( \cdot \). Graphs are represented as multisets of edges, and the query \( Q_1 \cdot Q_2 \) is true iff the current graph can be split in parts \( G_1 \) and \( G_2 \) such that \( Q_1 \) is true of \( G_1 \). For the purpose of this discussion, details of transducers, abstraction and the least fix-point operator can be ignored.
Composition is sometimes called exponential composition, to distinguish it from the linear variant which is denoted $Q_1|Q_2$. For this variant, the left graph, for which $Q_1$ must be true, consists of just one edge.

We define the useful predicate here$(x)$ with the formula $\text{in\_degree}(x) \geq 1 \lor \text{out\_degree}(x) \geq 1$. The formula $\forall x.\text{here}(x) \rightarrow \phi$ says that $\phi$ holds for all nodes in the graph. This is abbreviated as $\forall x \in G. \phi$ ($\forall x_1, \ldots, x_n \in G. \phi$).

This notation allows the expression of some common properties of graphs. For example, the following formula defines a graph consisting of a single path:

$$\text{path}(x, y) \triangleq \text{in\_degree}(x) = 0 \land \text{out\_degree}(y) = 0 \land \forall z \in G.((z \neq x \rightarrow \text{in\_degree}(z) = 1) \land (z \neq y \rightarrow \text{out\_degree}(y) = 1))$$

A graph consisting of just (nonoverlapping) cycles is defined by $\text{cycles} \triangleq \forall x \in G.\text{degree}(x) = 2$. Transducers can be used to define operations on graphs like transitive closure, for example.

This logic is not able to deal with hyperedges and isolated vertices, and does not allow for labels for vertices. The latter can easily be introduced. The logic does not really need to be extended to deal with hyperedges; there are well-known ways to rewrite a hypergraph to a ‘normal’ graph. However, to improve readability it is preferable to work on hypergraphs directly, all that is needed for this is some syntactic sugaring. Since hyperedges are by default ordered, ordered trees are just a specific case of such graphs, and these are obviously useful in computational linguistics.

The same is true of isolated vertices. The logic $\text{GL}$ and variants do not allow for this, since they define graphs as just multisets of edges. Once all edges that have vertex $v$ as origin or destination are removed from a graph, vertex $v$ itself is no longer accessible either. It is however very easy to encode such graphs, simply by introducing a reserved edge label and using it to label loop edges for all vertices. This obviously does not extend the expressive power of the logics.

4 Complexity and expressive power

In [DGG07] it is demonstrated that without recursion, the linear and exponential versions of the logic are equivalent to first-order (FO) and monadic second-order (MSO) logics on graphs representing strings, respectively. Extended with the fix-point operator for recursion, both are able to express PSPACE-complete problems.

For query languages, it makes sense to distinguish the following:

1. combined complexity $\{(G, \phi) : G \models \phi\}$;

2. data complexity the complexity class that contains all sets $G_\phi$.

The combined complexity takes the size of both query and database into account, the data complexity is defined strictly in terms of size of the database. Since in real-life situations, queries are generally much smaller than the databases

---

1 We do not need to introduce natural numbers into the logic for this; [DGG07] shows how to define the degree predicates for any number $n$. 
they query, the latter is considered to be more realistic. However, in the case of
graph mining, the so-called transactional setting is quite common. This entails
dealing with many graphs of restricted size, and thus the combined complexity,
which is generally higher, may be more relevant in such cases. Whether this is
really the case is hard to determine a priori, and can probably only be decided
in practice.

However, in an ILP context, GL would be used as a description- or program-
ming language more than a query language. In other words, it would not be
used exclusively for model checking. This leaves open the possibility that the
complexity goes up in such a setting, and in the following we argue that this is
actually the case.

5 Learning

Casting graph learning in the ILP framework makes a whole spectrum of ap-
proaches to this problem possible: learning from positive and negative examples,
learning from positive data, intensional clustering, decision trees or -lists, regres-
sion etc. All of these require the definition of a refinement operator for graphs,
like least general generalization.

As the name suggests, the least general generalization operation is meant to
generate, given two clauses, a clause that generalizes both, but is still as specific
as possible. One aspect of this operation is the generalization of literals, which
involves matching and generalizing their arguments. Normally anti-unification
is the appropriate approach to this problem. Given two terms \( t_1 \) and \( t_2 \), anti-
unification yields a (unique) term \( t \) such that there exist substitutions \( \sigma_1, \sigma_2 \)
such that \( \sigma_1[t_1] = t \) and \( \sigma_2[t_2] = t \).

In the case of graph logic, where graphs are represented by multisets, this
approach will not work. One reason is that there generally is no unique term.
The composition operator \( \circ \) is both associative and commutative. Unification of
graphs that are represented using this operator is thus equivalent to a special case
of unification under an equivalence theory (E-unification), namely unification
modulo associativity and commutativity.

When used as a query language, generally one of the terms is ground (since
the logic is only used for model checking) and this greatly simplifies matters.
However, when used as a programming language, where both terms can contain
variables representing subgraphs, nodes and edge labels, the full power of AC-
unification is necessary. This operation is known to be finitary, i.e., it always
yields a finite number of most general unifiers, if the terms are unifiable. This
number can be very high even for simple terms ([BHK+88]), so computing the
complete set of mgus is generally not a feasible approach. Even the problem
of checking whether two terms containing AC-function symbols are unifiable
(AC-matching) is only known to be in NP ([KN92]).

Thus, pure composition seems to be an operation that is too powerful for
our purposes. Note that the problems with composition remain even with a

\(^2\) However, AC linear matching is in P ([BH96]).
restricted class of graphs for which otherwise difficult problems are known to be polynomial-time solvable, such as outerplanar graphs. It is also difficult to see how such problems could be avoided in any formalism with the same type of multi-set semantics. It may be possible to come up with optimizations which compile out composition in certain cases, but this is expected to be very hard.

Thus, linear composition is much more attractive from a computational point of view, especially when combined with recursion, and makes implementation easier as well. Whether GL has the same expressive power as LGL is currently an open question, but it is clear that these systems are very close in expressive power ([KN92]). Thus, it seems we can shift the computational burden from the exponential composition operation to recursive clauses.

6 Recursion

In order to restrict composition to the linear case while preserving expressive power, recursive predicates must be allowed. Learning recursive logic programs is a notoriously difficult problem, however, positive results do exist. For example, in [Sha83] an algorithm is presented that identifies such programs in the limit, in the context of learning from interpretations, from positive and negative data. Generally, positive results are either theoretically sound but only applicable to a restricted class, or theoretically unjustified heuristics are used to prune the search space. We give some more examples:

In [Mal03] the ILP system ATRE is described. Building on work on multiple predicate learning (cf [RLD93]), ATRE is able to learn recursive theories from real world data in reasonable time. ATRE explores just a polynomially bounded part of the search space, although it may still take exponential time to do so. A short overview of work in this direction can also be found in [Mal03].

In the setting of nonmonotonic inductive logic programming, introduced in [Hel89], the focus is on finding interesting properties of the examples. In the formalization from [RD94], concepts are represented as clausal theories, and examples as interpretations that are models of the theory. It has been shown there that first order range-restricted clausal theories consisting of clauses made up of up to \( k \) literals of size at most \( j \) are PAC-learnable from positive data in polynomial time. This is an expressive class which includes recursive concepts. This framework has been implemented as the CLAUDIEN system, which does not generate all clauses in \( jk - CT \), but uses an optimal refinement operator and a bias specification mechanism.

A more theoretical perspective is offered by work done on the subject of elementary formal systems (EFS), basically logic programs consisting of definite clauses whose arguments are patterns instead of terms. A definite clause of an EFS is hereditary (H-EFS) if every every pattern in the body is a subword of a pattern in the head. It has been shown ([MSS00]) that \( H - EFS(m, k, t, r) \) is polynomial-time (PAC)learnable, where theories consist of at most \( m \) hereditary definite clauses with predicate symbols of arity at most \( r \), where \( k \) and \( t \) bound the number of occurrences of variables in the head and the number of atoms in the
body, respectively. This is a strong theoretical result, and offers the perspective of PAC-learnability results for GL-based programs. For this we need a notion of pattern that has a multi-set semantics, which could easily be done.

7 Conclusions

The exponential composition operator is known to be more or less tractable in the context of a graph query language. However, for an ILP setting it seems all but unusable, and the linear variant seems a much better candidate. In order not to lose expressive power, theories expressed with this operator need to be recursive, which although problematic is not infeasible for restricted subclasses.

References


