Mode Directed Path Finding

Irene M. Ong[1], Inês de Castro Dutra[2], David Page[1], Vítor Santos Costa[2]

[1] Department of Biostatistics and Medical Informatics,
University of Wisconsin – Madison, WI 53706 USA
[2] COPPE/Sistemas, UFRJ Centro de Tecnologia, Bloco H-319, Cx. Postal 68511
Rio de Janeiro, Brasil

Abstract. Learning from multi-relational domains has gained increasing attention over the past few years. Inductive logic programming (ILP) systems, which often rely on hill-climbing heuristics in learning first-order concepts, have been a dominating force in the area of multi-relational concept learning. However, hill-climbing heuristics are susceptible to local maxima and plateaus. In this paper, we show how we can exploit the links between objects in multi-relational data to help a first-order rule learning system direct the search by explicitly traversing these links to find paths between variables of interest. Our contributions are twofold: (i) we extend the *pathfinding* algorithm by Richards and Mooney [12] to make use of *mode declarations*, which specify the mode of call (input or output) for predicate variables, and (ii) we apply our extended path finding algorithm to *saturated bottom clauses*, which anchor one end of the search space, allowing us to make use of background knowledge used to build the saturated clause to further direct search. Experimental results on a medium-sized dataset show that path finding allows one to consider interesting clauses that would not easily be found by Aleph.

1 Introduction

Over the past few years there has been a surge of interest in learning from multi-relational domains. Applications have ranged from bioinformatics [9], to web mining [2], and security [7]. Typically, learning from multi-relational domains has involved learning rules about distinct entities so that they can be classified into one category or another. However, there are also interesting applications that are concerned with the problem of learning whether a number of entities are connected. Examples of these include determining whether two proteins interact in a cell, whether two identifiers are aliases, or whether a web page refers to another web page; these are examples of link mining [6]. A number of approaches for exploiting link structure have been proposed; most of these approaches are graph based, including SUBDUE [3], and ANF [10].

Our focus is on first-order learning systems such as ILP. Most of the approaches in ILP rely on hill-climbing heuristics in order to avoid the combinatorial explosion of hypotheses that can be generated in learning first-order concepts. However, hill-climbing is susceptible to local maxima and local plateaus, which is an important factor for large datasets where the branching factor per

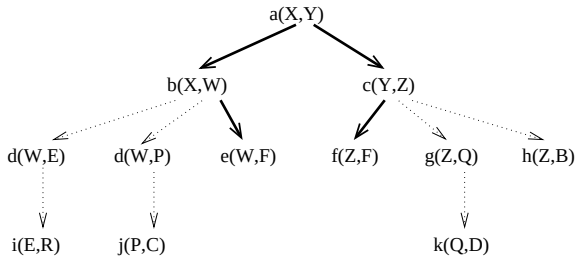

Fig. 1. Search space induced by a saturated clause. The literal at the root of the graph represents the head of the saturated clause. All the other literals in the graph are literals from the body of the saturated clause. Bold arcs indicate a possibly interesting path linking X to Y. Dotted arcs indicate parts of the search space that will not lead to determining connectivity of X and Y.

node can be very large [5, 4]. Ideally, saturation-based search and a good scoring method should eventually lead us to interesting clauses, however, the search space can grow so quickly that we risk never reaching an interesting path in a reasonable amount of time (see Figure 1). This prompted us to consider alternative ways, such as *pathfinding* [12], to constrain the search space.

Richards and Mooney [12] realized that the problem of learning first-order concepts could be represented using graphs. Thus, using the intuition that if two nodes interact there must exist an explanation of the interaction, they proposed that the explanation should be a connected path linking the two nodes. However, pathfinding was originally proposed in the context of the FOIL ILP system, which does not rely on creating a *saturated clause*. A seminal work in directing the search in ILP systems was the use of *saturation* [14], which generalizes literals in the seed example to build a *bottom clause* [8], which anchors one end of the search space. Hence, we propose to find paths in the saturated clause.

The original pathfinding algorithm assumes the background knowledge forms an undirected graph. In contrast, the saturated clause is obtained by using *mode declarations*: in a nutshell, a literal can only be added to a clause if the literal's *input* variables are known to be bound. Mode declarations thus embed directionality in the graph formed by literals. Our major insight is that a saturated clause for a moded program can be described as a directed *hypergraph*, which consists of nodes and hyperarcs that connect a nonempty set of nodes to one target node. Given this, we show that path finding can be reduced to reachability in the hypergraph, whereby each *hyperpath* will correspond to a hypothesis. However, we may be interested in non-minimal paths and in the composition of paths. We thus propose and evaluate an algorithm that can enumerate all such hyperpaths according to some heuristic.

2 The Saturated Clause and Hypergraph

The similarities between the properties of a saturated clause and a hypergraph provide a natural mapping from one to the other. A directed hypergraph $\mathcal{H}$ is

defined by a set of nodes N and a set of hyperarcs H. A hyperarc has a nonempty set of source nodes $S \subseteq N$ linked to a single target node $i \in N$ [1].

A saturated clause can be mapped to a hypergraph in the following way. First, as a saturated clause is formed by a set of literals, it is thus natural to say that each literal L_i is a node in the hypergraph. Second, we observe that each literal or node L_i may need several input arguments, and that each input argument may be provided from a number of other literals or nodes. Thus, if a node L_i has a set of input variables, I_i, each hyperarc is given by a set of literals generating I_i and L_i. Specifically, a mapping can be generated as follows:

1. Each node corresponds to a literal, L_i.
2. Each hyperarc with $N' \subseteq N$ nodes is generated by a set $\mathcal{V}$ of $i-1$ variables $V_1, \ldots, V_{i-1}$ appearing in literals $L_1, \ldots, L_{i-1}$. The mapping is such that
   (i) every variable $V_k \in I_i$ appears as an `output` variable of node L_k
   (ii) every variable V_k appears as argument k in the input variables, I_i, of L_i.

Intuitively, the definition says that nodes in $L_1, \ldots, L_{N'-1}$ with output variables that generates input variables for node $L_{N'}$, will be connected by hyperarcs. Note that if node $L_{N'}$ has a single input argument, the hyperarc will reduce to a single arc. Note also that the same variable may appear as different input arguments, or that the same literal may provide different `output` variables.

Figure 2a shows an example saturated clause and resulting hypergraph. The literal at the root of the graph, $a(X, Y)$, is the head of the saturated clause. Other literals in the graph appear in the body of the saturated clause. All literals of arity 2 have mode $+, -$, that is, `input` and `output`. Literal $e(F, B, D)$, has arity 3 and mode $+, +, -$. Arcs in the figure correspond to dependencies induced by variables, thus there is an arc between $c(X, W)$ and $d(W, F)$ because $d(W, F)$ requires `input` variable W. On the other hand, there is no arc between $c(X, B)$ and $f(C, B)$ since variable B is an `output` variable in both cases. All of the literals except $e(F, B, D)$ has a single input variable, hence those hyperarcs consists of a single arc. However, there are four hyperarcs for node $e(F, B, D)$; they are $d(W, F), c(X, B), g(A, F)$ and $f(C, B)$.

Before we present the path finding algorithm, we need to perform a simple transformation. The graph for a saturated clause is generated from the seed example, L_0. If the seed example has M arguments, it generates M variables, which we transform as follows:

1. Generate M new nodes L'_j, where $j = 1, ..., M$, such that each node will have one of the variables in L_0. Each such node will have an `output` variable V_j.
2. Replace the edge between L_0 and some other node induced by the variable V_j by an edge from the new node L'_j.

Figure 2b shows this transformation for the hypergraph in Figure 2a. Path generation thus reduces to finding all hyperpaths that start from nodes $L'_1, \ldots, L'_M$.

3 Algorithm

In directed graphs, a path π is a sequence of edges $e_1, e_2, \ldots, e_k$ and nodes $n_1, n_2, \ldots, n_k$, such that $e_i = (n_{i-1}, n_i), 1 \leq i \leq k$. The shortest hyperpath

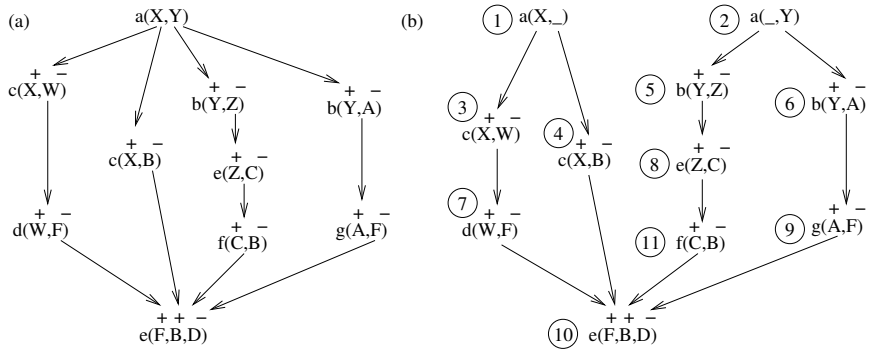

Fig. 2. (a) Hypergraph of our example saturated clause where $a(X,Y)$ is the head of the clause and '+' indicates `input` variable, '−' indicates `output` variable: $a(X,Y) \leftarrow c(X,W), c(X,B), b(Y,Z), b(Y,A), d(W,F), e(Z,C), g(A,F), e(F,B,D), f(C,B)$. (b) Transformation of hypergraph (a) splits the head literal into its component arguments, which then serve as different sources for the path finding algorithm. The number preceeding each literal indicates the label that will be used for that literal in Figure 3.

problem is the extension of the classic shortest path problem to hypergraphs. The problem of finding shortest hyperpaths is well known [1, 11]. We do not require optimal hyperpaths; rather, we want to be able to enumerate all possible paths, and we want to do it in the most flexible way, so that we can experiment with different search strategies and heuristics.

We present our path finding algorithm through the transformed hypergraph shown in Figure 2b. First, we want to emphasize that our goal is to generate paths in the 'path finding' sense, which is slightly different from the graph theoretical sense. More precisely, a hyperpath will lead from a node to a set of other nodes in the hypergraph (i.e. a *path* is a set of hyperpaths), each starting from different source nodes, such that the two hyperpaths have a variable in common. For example, in Figure 2b, nodes $\{1, 4\}$ form a hyperpath, and nodes $\{2, 5, 8, 11\}$ form another hyperpath. Since nodes 4 and 11 share variable B, $\{1, 2, 4, 5, 8, 11\}$ form a *path*. Our algorithm generates paths as *combinations* of hyperpaths.

Given hypergraph $\mathcal{H}$, which includes information for `input` and `output` variables for each node or literal, *source* nodes and desired *maxdepth* (a function of clause length), we describe an algorithm that returns a list of paths connecting all input variables. Figure 3 illustrates our path finding algorithm on the example hypergraph in Figure 2b. The numbered nodes in Figure 3 correspond to the labels of literals in Figure 2b. The depth of the graph is indicated on the left hand side. Current paths are expanded at each depth if the node to be expanded has its `input` variables bound. Otherwise, they are crossed out (e.g., $P(2, 2)$ and $P(3, 3)$). $VariableSet(s, d)$ represents the set of variables reachable from s at depth d. They are used to find common variables between variable sets of different sources at a particular depth. Hyperpaths found so far are indicated by $P(d, n)$, where d indicates the depth and n is the node index at that depth. Paths found are denoted $P(d, n_1) - P(d, n_2)$.

For each source and each depth, starting at depth 1, the algorithm proceeds:

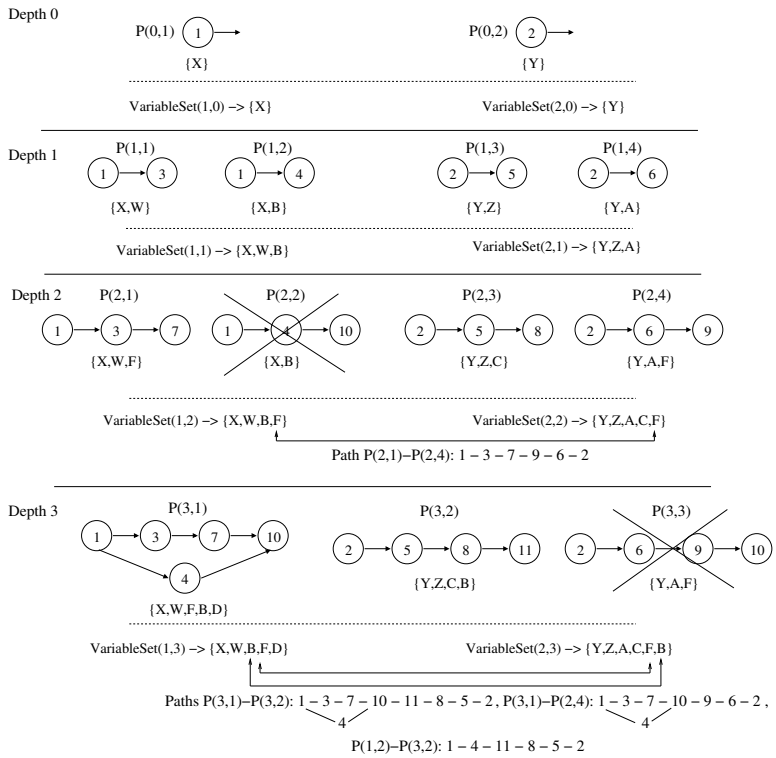

Fig. 3. Illustration of path finding algorithm using mode declarations. The numbered nodes in the graphs are the labels of literals in Figure 3.

1. Expand paths from previous depth, $d-1$, only if `input` variables for the newly expanded node, n, exist in $VariableSet(s, d-1)$ of the previous depth and are bound by parents reachable at the current depth (i.e., all n's `input` variables are bound by parent nodes which contain n's `input` variables).
2. Place all variables reachable from s at current depth d in $VariableSet(s, d)$.
3. Check $VariableSet$ of each source at the current depth for an intersection of variables; for each variable in the intersection create paths from permutations of all nodes containing the variable, including those from previous depths.

Depth 0 corresponds to the initial configuration with our 2 source nodes, 1 and 2, which we represent as hyperpaths $P(0,1)$ and $P(0,2)$ respectively. The variables of source nodes 1 and 2, X and Y are placed into their respective variable sets ($VariableSet(1,0)$ and $VariableSet(2,0)$). We begin at Depth 1. Node 1 has two hyperpaths of size 2; one to node 3 (shown by hyperpath $P(1,1)$) and the other to node 4 ($P(1,2)$). Node 2 can reach nodes 5 and 6 giving hyperpaths $P(1,3)$ and $P(1,4)$. At this depth we can reach variables W and B from X, and variables Z and A from Y, indicated by $VariableSet(1,1)$ and $VariableSet(2,1)$. Since we do not have an intersection of the variable sets, we cannot build a path.

Depth 2 corresponds to expanding the hyperpaths of nodes $3, 4, 5$ and 6. Hyperpath $P(1,1)$ can reach node 7 allowing X to reach variable F. Hyperpath

$P(1,2)$ tries to expand to node 10, but this hyperpath only contains variable B, whereas node 10 requires both F and B as input, hence $P(2,2)$ is not expanded (crossed out). Hyperpath $P(1,3)$ can be expanded with 8, and hyperpath $P(1,4)$ can be extended with node 9. At this point we have hyperpaths from the first argument reaching variables W, B, F, and from the second argument reaching Z, A, C, F. Variable F can be reached with hyperpaths starting at the nodes that define variables X and Y, so we have a path. We thus find that if we combine hyperpaths $P(2,1)$ and $P(2,4)$ we have our first path: $1,3,7,9,6,2$ $(P(d,n_1) - P(d,n_2))$.

At Depth 3, hyperpath $P(2,1)$ can reach node 10 by merging with hyperpath $P(1,2)$. This creates the hyperpath $P(3,1)$ which reaches variables X, W, F, B, D. Hyperpath $P(2,3)$ is expanded to include node 11 but hyperpath $P(2,4)$ cannot be expanded to include node 10 as it does not contain variable B required as an input variable for node 10. Hence $P(3,3)$ is omitted. Now we have two new hyperpaths that can be combined between themselves and with older hyperpaths to generate new paths. Hyperpath $P(3,1)$ reaches variables X, W, F, B, D. We can build a new path by connecting $P(3,1)$ with hyperpath $P(3,2)$, as they both share variable B. $P(3,2)$ can also be connected to $P(2,4)$, as they both share F. Hyperpaths $P(3,2)$ and $P(1,2)$ share variable B, so we can generate the new path $P(3,2) - P(1,2)$. For hyperpaths that are already a path, as they touch X and Y, we can further extend them by merging them with other hyperpaths, obtaining non-minimal paths.

4 Experimental Evaluation

Paths found can be used in a number of ways. One way is to use Richards and Mooney's method to perform search by generating a number of paths, and then refining them [12, 15]. Alternatively, one can consider the paths found as a source of extra information that can be used to extend the background knowledge (i.e., add paths as background knowledge). In this case, paths can be seen as intensional definitions for new predicates in the background knowledge.

We used the UW-CSE dataset by Richardson and Domingos [13] for a first study of path finding on a heavily relational dataset. The dataset concerns learning whether one entity is advised by other entity based on real data from the University of Washington CS Department. The example distribution are skewed as we have 113 positive examples versus 2711 negative examples. Following the original authors, we divided the data into 5 folds, each one corresponding to a different group in the CS Department. We perform learning in two ways for our control and experiment. In the first approach, we used Aleph to generate a set of clauses. In the second approach, we used path finding to find paths, which are treated as clauses. Further, we allow Aleph to decorate paths with attributes by trying to refine each literal on each path.

We were interested in maximizing performance in the precision recall space. Thus, we extended Aleph to support scoring using the f-measure. The search space for this experiment is relatively small, so we would expect standard Aleph

Folds	Aleph		Path Finding	
	# Clauses	Avg Clause Length	# Clauses	Avg Clause Length
Theory	2	4.5	2	5
AI	3	3.7	2	5.5
Graphics	3	4	2	6
Languages	1	3	3	7
Systems	1	4	3	5.7

Table 1. Theory Comparison

Folds	Aleph			Path Finding		
	Recall	Precision	F1 measure	Recall	Precision	F1 measure
Theory	38	27	32	81	11	19
AI	63	9	16	75	12	21
Graphics	85	46	60	95	20	33
Languages	22	100	36	33	100	50
Systems	87	8	15	82	9	16

Table 2. Test Set Performance (results given as percentage)

search to find most paths. The two systems do find different best clauses, as shown in Table 1 which show both number of clauses and average clause length, including the head literal. Although most of the clauses found by Aleph are paths, the path finding implementation does find longer paths that are not considered by Aleph and also performs better on the training set.

Table 2 summarizes performance on the test set. This dataset is particularly hard as each fold is very different from the other [13], thus performance on the training set may not carry to the testing set. Both approaches perform similarly on the Systems fold. The AI fold is an example where both approaches learn a common rule, but path finding further finds an extra clause which performs very well on the training set, but badly on the test data. On the other hand, for the Languages fold both approaches initially found the same clause, but path finding goes on to find two other clauses, which in this case resulted in better performance. We were surprised that for both Graphics and Systems, path finding found good relatively small clauses that were not found by Aleph.

5 Conclusions and Future Work

We have presented a novel algorithm for path finding in moded programs. Our approach takes advantage of mode information to reduce the number of possible paths and generate only legal combinations of literals. Our algorithm is based on the idea that the saturated clause can be represented as a hypergraph, and the use of hyperpaths within the hypergraph to compose the final paths. Muggleton used a similar intuition in *seed* based search, using a heuristic to classify clauses: a clause's score depends on coverage and the distance the literals have to each entity [8]. In contrast, our approach is independent of scoring function.

Preliminary results on a medium sized dataset showed that path finding allows one to consider a number of interesting clauses that would not easily be

considered by Aleph. On the other hand, path finding does seem to generate longer clauses, which might be more vulnerable to overfitting. In future work we plan to combine paths using the approach in Davis *et al.* [4] as well as apply this algorithm to larger datasets, where path finding is necessary to direct search.

Acknowledgments

Support for this work was partially funded by U.S. Air Force grant F30602-01-2-0571. The first author acknowledges support from NLM training grant 5T15LM005359. This work was done while Inês and Vítor were visiting UW-Madison. Finally, we thank Ian Alderman for helpful discussions and Rich Maclin for comments on the paper.

References

1. G. Ausiello, R. Giaccio, G. Italiano, and U. Nanni. Optimal traversal of directed hypergraphs. Technical report, Dipartimento di Informatica e Sistemistica, Universit di Roma La Sapienza, 1997.
2. S. Chakrabarti. *Mining the Web*. Morgan Kaufman, 2002.
3. D. Cook and L. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15:32–41, 2000.
4. J. Davis, I. Dutra, D. Page, and V. S. Costa. Establishing identity equivalence in multi-relational domains. In *Proc. of Intelligence Analysis*, 2005.
5. J. Davis, V. Santos Costa, I. Ong, D. Page, and I. Dutra. Using bayesian classifiers to combine rules. In *3rd Workshop on MRDM at KDD*, 2004.
6. L. Getoor. Link mining: A new data mining challenge. *SIGKDD Explorations*, 5:84–89, 2003.
7. D. Jensen. Prospective assessment of AI technologies for fraud detection: A case study. In *Working Papers of the AAAI-97 Workshop on Artificial Intelligence Approaches to Fraud Detection and Risk Management*, 1997.
8. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
9. D. Page and M. Craven. Biological applications of multi-relational data mining. *SIGKDD Explorations*, 5:69–79, 2003.
10. C. Palmer, P. Gibbons, and C. Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *Proc. of ACM SIGKDD on Knowledge Discovery and Data Mining*, 2002.
11. G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.
12. B. Richards and R. Mooney. Learning relations by pathfinding. In *National Conference on AI*, pages 50–55, 1992.
13. M. Richardson and P. Domingos. Markov logic networks. Technical report, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, 2004.
14. C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.
15. S. Slattery and M. Craven. Combining statistical and relational methods for learning in hypertext domains. In *Proc. of ILP*, pages 38–52, 1998.