Human Interaction and Emerging Technologies (IHIET 2023), Vol. 70, 2023, 99–107
https://doi.org/10.54941/ahfe1002934

Efficient Inductive Logic Programming Based on Predictive A*-Like Algorithm

Moeko Okawara[1], Junji Fukuhara[1], Munehiro Takimoto[1], Tsutomu Kumazawa[2], and Yasushi Kambayashi[3]

[1]Tokyo University of Science, Noda, Chiba 2780022, Japan
[2]Software Research Associates, Inc., Japan
[3]Nippon Institute of Technology, Japan

ABSTRACT

Recently, research scientists have developed various machine learning techniques. In particular, deep learning contributes to creating structured data such as tables from unstructured data, i.e., images and sounds. On the other hand, most of the machine's decisions and actions are hard to be explained or verified. As another perfectly explainable approach, Inductive Logic Programming (ILP) has been used in data mining. ILP is useful to extract relevant relations from structured data. ILP inductively infers a hypothesis as a result of learning from given examples and background knowledge. In the inference process, ILP explores hypothesis candidates while calculating a cover set that is a set of examples deduced from each candidate. This process costs a lot. We propose a new search algorithm of the hypothesis through efficiently computing the cover set on a relational database management system (RDBMS). Some modern RDBMSs process SQL with multi-worker or GPU in parallel. Our ILP efficiently calculates the cover set through transforming each deduction into SQL on such an RDBMS. However, the overhead for launching SQL processing is very expensive and often decreases effectiveness of the parallel execution. We extend a hypothesis search algorithm A*-like of Progol, which is one of ILP systems, to refine several hypothesis candidates with high evaluation scores simultaneously. We implemented our extension in Progol and evaluated it in practical experiments. Our results show that our method remarkably improves the performance of ILP systems.

Keywords: Inductive logic programming, Cover set, SQL

INTRODUCTION

Various machine learning (ML) techniques have been developed widely over the last decade. In particular, deep learning (DL) contributes to ML for creating a lot of structured data such as tables from unstructured data, i.e., images and sounds. The results have led to much success in engineering, but most of their decisions and actions are hard to be explained or verified. On the other hand, Inductive Logic Programming (ILP) (Martínez-Angeles et al., 2014) i.e., a perfectly explainable ML approach, has been used in data mining. ILP, which is based on first-order predicate logic, is one of the symbolic approaches that is useful to deal with structured data and the relations between them. In a practical sense, we can add the results generated by ILP into given background knowledge and make the knowledge database rich. Thus, ILP becomes more important for data mining than before, and we can extract meaningful relations between the structured data. However, contrary to DL, it is not easy for ILP to perform a learning process efficiently, because we cannot make ILP processes uniformly executable in parallel on GPUs. This learning process is an inductive prediction process that uses positive and negative examples as training samples. In the process, ILP explores hypothesis candidates while calculating a cover set. A cover set is a set of examples deduced from each candidate. Notice that from the finally obtained hypothesis, the positive examples should be deduced, and the negative ones should not be deduced with the background knowledge. The cover set is known to be uniformly calculated in the relational operations on a relational database management system (RDBMS) such as SQL. Since modern RDBMSs can not only manage memory operations safely but also execute SQL in parallel utilizing parallel workers or GPUs. Thus, we can execute ILP in partially parallel. However, we cannot ignore the significant performance overhead of launching the procedure for each cover set calculation. In order to mitigate this problem, we propose an extension of an A*-like algorithm (Muggleton, 1996), which is the algorithm for searching for a hypothesis adopted by in Progol (Muggleton, 1991). Progol is one of the most popular ILP systems. The algorithm incrementally refines each hypothesis candidate through adding a literal to it, and calculating its cover set to check whether it satisfies the condition as a hypothesis. Our algorithm, called Predictive A*-like algorithm, simultaneously performs several refinements with high possibility as a hypothesis. Even though the refinements may include redundant hypotheses due to multiple-time-findings of the same hypothesis, the predictive refinement reduces a lot of overhead cost for launching the procedure of cover set calculation. Thus, our algorithm can generate a hypothesis more efficiently than the traditional search algorithm. We have extended Progol to implement Predictive A*-like algorithm on PostgreSQL. We demonstrate that our extended Progol works significantly well to obtain practical experimental results.

The rest of this paper is organized as follows. Section 2 gives the preliminary of ILP. Section 3 describes the manner of our translation of logical programs to SQL for calculating cover sets on an RDBMS. Section 4 describes our proposed extension of A*-like algorithm. Section 5 presents experimental evaluations of our system. Section 6 discusses related work. Finally, we conclude our discussion in section 7.

PRELIMINARY

Knowledge Representation

ILP is a kind of symbolic artificial intelligences and based on logic programming. In particular, Progol is based on first-order logic and uses the logic programming language Prolog for its uniform representation. The knowledge in Prolog is represented as a set of Horn clauses. For example, Noby's family relationship of Doraemon, which is a Japanese cartoon, is presented as follows:

$$\text{father(nobiru, nobisuke).father(nobisuke, noby).mother(tamako, noby).} \quad (1)$$

$$\text{grandfather(X, Z)} : -\text{father(X, Y), father(Y, Z).} \quad (2)$$

$$\text{grandfather(X, Z)} : -\text{father(X, Y), mother(Y, Z).} \quad (3)$$

The tuples with predicates father, mother and grandfather respectively represent relationships father-child, mother-child and grandfather-grandchild between their elements, called a literal. The literals compose a clause, which has a period at the end of it. The clause with ':-' is called a rule, where the left hand side of it is called a head and the right hand side is called a body. The sequence of literals in the body means logical-and of them, and the rule means that the body implies the head under corresponding variables *X*, *Y* and *Z*. Rules father and mother with no body, which are called facts, mean facts between persons nobiru, nobisuke, tamako and noby. These clauses mean logical-or of them; hence, they represent that *X* is a grandfather of *Z*, if *Y* is a father of *Z* and *X* is a father of *Y* or *Y* is a mother of *Z* and *X* is a father of *Y*, respectively.

Inductive Logic Programming

The outline of the algorithm of Progol (Muggleton, 1996) is as follows:

(1) if $E = \emptyset$, return H
(2) Generate MSH from the first example e of E
(3) Search for the hypothesis space to gain the optimal hypothesis
(4) $B := B \cup H'$, $H := H \cup H'$
(5) $E' := \{e' \mid e' \in E \text{ and } B \models e'\}$
(6) $E := E \setminus E'$
(7) go back to 1

First, given positive examples *E* and background knowledge *B*, Progol picks up an example *e* from *E* (step 2) and generalizes it to a rule *H'* (step 3) one by one. Second, once *H'* is generated, it is added into *B* (step 4), of which the cover set is subtracted from *E* (steps 5 and 6). Finally, when *E* becomes an empty set, a set of all the generated *H'* is the hypothesis *H* (step 1). Notice that since *H'* is also represented as a fact or rule, it can be added into the current background knowledge to extend it. Moreover, Progol requires mode declarations. They correspond to a specification of generated rules as a hypothesis, consisting of ones for heads and bodies of the rules. The declarations are used to derive the most specific hypothesis (MSH) through entailing each example. Since MSH is the most complex hypothesis, it not only gives the lower limit to the hypothesis search, but also enables generating more specific hypothesis candidates through adding literals included in MSH to the current candidates. The process that makes hypothesis candidates more specific is called refinement, and corresponds to searching process of *H'* (step 3).

In the step 3, Progol searches for a hypothesis while refining a hypothesis candidate with a higher score of the predefined the evaluation function *f*. The function *f* returns the higher score when a hypothesis candidate has the large cover set of positive examples, the small cover set of negative examples, and the short clause. This searching manner is called an A*-like algorithm. It is conducted as follows:

```
(1) Open := { [] }, Closed := ∅
(2) s := best(Open), Open := Open - {s}, Closed := Closed ∪{s}
(3) Open := (Open ∪ refinements(s)) \ Closed
(4) if terminated(Closed, Open) return best(Closed)
(5) else if Open = ∅ return e
(6) else goto 2
```

Hypothesis candidates are generated through refinements starting with a clause that has no literal. Once the hypothesis candidates are generated, they are added to the work-list *Open* (step 3). The next hypothesis candidate *s* to refine is selected from *Open* based on the function *best* (step 2), to give a hypothesis candidate with the highest score given by of *f*. Once *s* is refined, s is added to another work-list *Closed* and then removed from *Open* (step 2).

The process is repeated until the predefined terminate condition *terminated* is satisfied or *Open* becomes empty. The terminate condition gives true when neither the remaining candidates nor any of their refinements have higher scores of *f* than the current one, resulting in the current candidate as a clause in the final hypothesis (step 4). If *Open* becomes empty, *e* itself is regarded as a hypothesis clause (step 5).

TRANSLATION INTO SQL

Our ILP system instructs an RDBMS to calculate a cover set of examples in SQL. Modern RDBMSs can not only manage memory operations but also process SQL in parallel utilizing parallel workers or GPUs. Our system, taking the advantages of an RDBMS, calculates the cover set safely and efficiently. In general, deductions required by calculation of the cover set can be executed as the relational operations, which are described in SQL queries. Notice that it is difficult for SQL to represent functions used in Prolog; hence, descriptions in our ILP are similar to Datalog (cBioPortal docs, n.a.), and has no function. Our system performs the deductions on an RDBMS in two steps i.e., a translation step and an execution step. The translation step translates hypothesis candidates and background knowledge into SQL. First, we collect the information of elements with the same variable from each literal in the clause of a hypothesis candidate. Second, we generate SQL command inner-joins of a table corresponding to each literal as join operations. At this time, based on collected variable information, we add the condition of the join, where columns corresponding to the same variable are specified as equal columns, as an on-clause. Finally, we generate a SQL select-clause as projection operations to leave the required columns from the joined table. Fig. 1 shows that the join and projection of tables. The query corresponding to the operations in Fig. 1 is represented in SQL as follows:

select rule0.c0, rule1.c1 from rule0 inner join rule1 on rule0.c1 = rule.c1

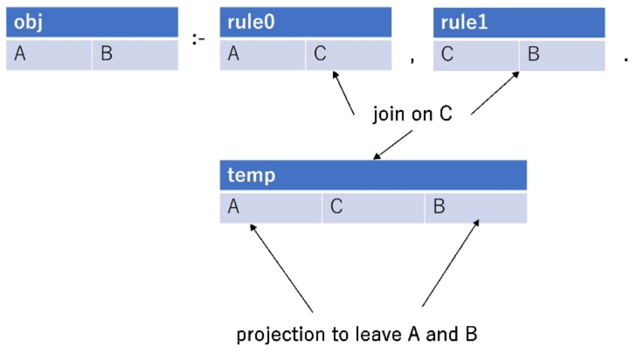

Figure 1: Join and projection of tables.

In our covering process, positive and negative examples have special columns, kind, id and weight as case sets in addition to ones corresponding to elements of literals. The column kind, which shows distinction of a positive or negative example for each row, is set to one for the positive one and zero for the negative one. The distinction enables an RDBMS to handle both of positive and negative examples as a single table. The column id, which represents the correspondence to the original clause, is set to a unique number that is the same as a label attached to the original clause.

The id number enables an RDBMS to send the cover set table as a sequence of numbers without sending the table itself, contributing to suppressing communication cost between the ILP system and an RDBMS. The column weight is used to compute the size of a cover set; hence, it is typically set to one, but if the cover set includes redundant examples, it is set to the number of the same examples, to suppress redundant computation. Once the hypothesis satisfying the terminate condition is found, we remove the positive cover set of it is from E. For example, the removal of k-th positive example is represented by the following SQL query:

delete from positive-examples where id = k.

PREDICTIVE A*-LIKE ALGORITHM FOR SQL DEDUCTION

The cover set computation occurs, each time a new hypothesis candidate is generated through refinement. The refinement process generates more specific hypothesis candidates through adding a new literal to the current candidate from the MSH body. For example, given the k-th literal of the MSH body, a new hypothesis is generated as follows:

1. If the k-th literal of the MSH body is possible, it will be added to the body of the current hypothesis candidate.

2. The current position k of the original candidate and the newly generated candidate is replaced with $k + 1$.

In this refinement, a new literal is added to the current candidate from the MSH body one by one. This refinement manner requests an RDBMS to compute the cover set per generation of a hypothesis candidate. However, this request costs a lot because its overhead in occurs frequently. The overhead rule(A, B) :- t0(A, D), t1(B, E).

Figure 2: Conventional Method for Refining a Hypothesis.

rule(A, B) :- t0(A, D), t1(B, E).

rule(A, B) :- t0(A, D), t2(D, C).

rule(A, B) :- t0(A, D), t3(C, B).

Figure 3: Proposal Method for Refining a Hypothesis.

includes connecting costs between the system and an RDBMS, and preparation costs for processing queries on a machine with the RDBMS. We extend the refinement to a multiple refinement that simultaneously generates several hypothesis candidates through adding a literal to each position in $\{k' \mid k \leq k'$ the number of literals of MSH$\}$. Consider MSH in the following:

rule(A, B) : -t0(A, D), t1(B, E), t2(D, C), t3(C, B).

(2) In addition, consider the current hypothesis candidate in the following:

rule(A, B) : -t0(A, D).

(3) Traditional Progol generates a new hypothesis candidate as shown in Fig. 2, while our system generates the candidates as shown in Fig. 3. Furthermore, our system applies the multiple refinements to several candidates. For example, assume that we specify ":- set(refine, *n*)?" on our ILP system. Multiple refinements *multiple-refinement* is repeatedly applied to several candidates until the number of generated new candidates exceeds the value of the user defined threshold *n* as follows:

```
(1)  $Open := \{ [] \}, Closed := \emptyset$
(2)  $S := \emptyset$
(3)  repeat
(4)  s := $best\ (Open), Open := Open - \{s\}, Closed := Closed \cup \{ s \}$
(5)  $S := S \cup multiple\text{-}refinment\ (s)$
(6)  until $|S| > n$
(7)  $Open := Open \cup sendToRDBMS(trans(S))$
(8)  if $terminated\ (Closed, Open)$ return $best\ (Closed)$
(9)  else if $Open = \emptyset$ return $e$
(10) else goto 2
```

We call the above extension of A*-like algorithm Predictive A*-like algorithm, which predictively refines several hypothesis candidates based on their evaluation scores. In the algorithm, *trans* translates a sequence of clauses in *S* into SQL queries. First, it translates each clause into a joined table as mentioned in Section 3. Second, it combines them with SQL union-all commands. Besides, as shown in Section 3, it generates an SQL select-clause to leave required columns. Once SQL queries are generated, it is sent to an RDBMS by *sendToRDBMS*, which performs the execution step. Finally, after an RDBMS returns id and kind of cover sets, *sendToRDBMS* calculates the evaluation score from the number of each kind based on *f*, sets the result to corresponding a new candidate, and returns *S*.

Thus, Predictive A*-like algorithm suppresses frequency of calling *sendToRDBMS*, so that it contributes to suppressing overhead and making hypothesis search efficient.

EVALUATION

We conducted numerical experiments to evaluate how effective our proposed method is compared with the traditional execution.

Environment and Settings

To show the effectiveness of our method, we implemented our proposed Progol based on SQL, called SProgol in OCaml. We adopted PostgreSQL as a RDBMS, and evaluated conducted some experiments on them. In the experiments of SProgol, we evaluated the following issues.

1. The effectiveness of using PG-Strom. PG-Strom is an extension module for using GPU on PostgreSQL.
2. The effectiveness of multiple refinements as compared to the original refinement. The predefined degree *n* of the multiple refinements is specified by description ":-set(refine, *n*)?".

The former aims to whether computing cover sets on GPU or not. Our experimental environment is presented as follows:

– OS: CentOS Linux release 7.9.2009 (Core)
– CPU: AMD EPYC 7232P 8-Core Processor
– GPU: NVIDIA A100-PCIE-40GB
– CUDA Toolkit: 11.4
– PostgreSQL: 13.4
– OCaml: 4.14.0
– PG-Strom 3.0

In each experiment, the negative examples were automatically generated four times as many as the positive examples, based on stochastic logic programs (Muggleton, 1995), which became active by setting ":-set(posonly)?".

Experimental Results

Fig. 4 shows the results of the experiments for breath cancer datasets (cBioPortal docs, n.a.), sample and patient which has 1124 positive examples respectively. The patient dataset has much more properties than the sample one; hence we extracted two datasets with dozen properties from the first ones in the patient dataset, making them patient1 and patient2. For the all of three datasets, we assume the hypothesis with two arguments in the head. We show the results of CPU and GPU as ratio of ones for multiple refinements with each degree to ones with the original refinement. The result of SProgol with multiple refinement shows 5.47x speedup on CPU and, 4.06x speedup on GPU for the sample dataset, 5.45x speedup on CPU and, 4.32x speedup on GPU for the patient1 dataset, and 6.82x speedup on CPU and, 6.83x speedup on GPU on patient2 dataset at the best for each. Notice that the result on GPU was evaluated for degree 5 to 20 degree of the multiple refinements because generated tables could not be allocated in the memory on GPU. As

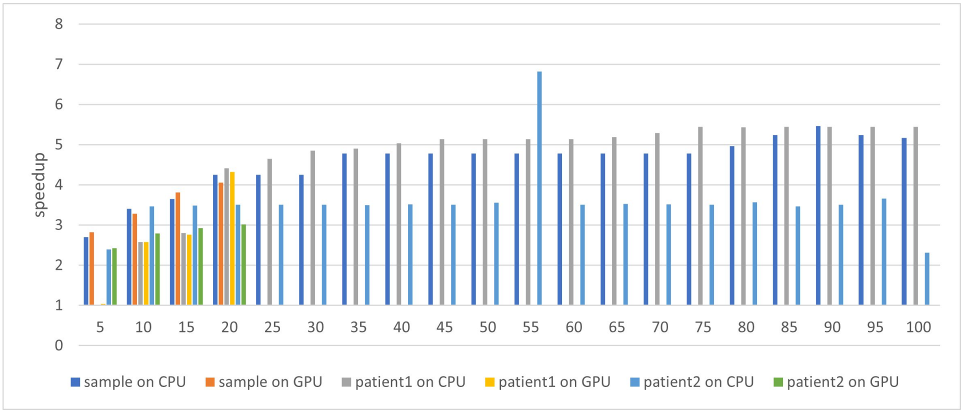

Figure 4: Execution time for breath cancer programs.

shown in the results, the multiple refinements through around 100 degrees contribute to the speedup of calculating cover sets on RDBMS.

RELATED WORKS

Eyad Algahtani and Dimitar Kazakov propose computation of the cover set for a hypothesis on GPUs based on propositional logic (Algahtani et al., 2018). Martínez-Angeles et al. propose the bottom-up evaluation of Datalog programs on GPU (HeteroDB, 2021). They are implemented for specific architectures, hence, once the specifications of each GPU are changed, they must to reimplemented. Actually, GPUs are rapidly progressing and their execution environments are sensitive to the architectures. Our ILP uses SQL as a kind of intermediate representations. The characteristic enables us to take advantage of the latest functionalities of GPUs through RDBMS supporting them. Predictive A*-like algorithm contributes to exploiting the potential of the SQL based ILP.

CONCLUSION AND FUTURE WORKS

We have demonstrated how a modern RDBMS accelerates the computation of the cover sets of hypothesis candidates in ILP parallel workers or a GPU, and have proposed Predictive A*-like algorithm as an extension of the hypothesis search algorithm. The experimental results show that Predictive A*-like algorithm makes the SQL based ILP system significant speedup. However, our current ILP system has not addressed the curse of dimensionality, which occurs in problems requiring a huge amount of clauses, yet, as well as other machine learning systems. In the future work, we are planning to improve our current implementation using meta-heuristics for giving the final hypothesis regardless of the dimensionality.

ACKNOWLEDGMENT

This work is partially supported by Japan Society for Promotion of Science (JSPS), with the basic research programs (C) (No.19K119090 and

22K11988), Grant-in-Aid for Scientific Research (KAKENHI), and JST SPRING, (No.JPMJSP2151), and receives the provision of evaluation environments from HIGHRESO Co., Ltd.

REFERENCES

Algahtani, E., Kazakov, D.: GPU-accelerated hypothesis cover set testing for learning in logic. *In CEUR Proceedings of the 28th International Conference on Inductive Logic Programming CEUR Workshop Proceedings* (01 2018).

Ceri, S., Gottlob, G., Tanca, L.: *Logic Programming and Databases*. Springer Publishing Company, Incorporated, 1st edn. (2012).

HeteroDB: PG-Strom manual (2021), http://heterodb.github.io/pg-strom/.

Martínez-Angeles, C.A., Dutra, I., Costa, V.S., Buenabad-Chávez, J.: A Datalog Engine for GPUs. In: Hanus, M., Rocha, R. (eds.) *Declarative Programming and Knowledge Management*. pp. 152–168. Springer International Publishing, Cham (2014).

Memorial Sloan Kettering Cancer Center: cBioPortal docs: https://www.cbioportal.org/.

Muggleton, S.: Inductive logic programming. *NGCO* 8, 295–318 (1991). https://doi.org/10.1007/BF03037089

Muggleton, S.: Inverse entailment and progol. *New Gen. Comput.* 13(3–4), 245–286 (dec 1995). https://doi.org/10.1007/BF03037227

Muggleton, S.: Stochastic logic programs. *Advances in inductive logic programming*, 32,254–264(1996).

Muggleton, S: Progol (2001) https://www.doc.ic.ac.uk/∼shm/Software/progol4.4/