QUERY PROOF STRUCTURE CACHING FOR INCREMENTAL EVALUATION OF TABLED PROLOG PROGRAMS

Taher Ali[1], Ziad Najem[2], and Mohd Sapiyan[1]

[1]Department of Computer Science, Gulf University for Science and Technology, Kuwait
ali.t@gust.edu.kw, sapiyan.m@gust.edu.kw
[2]Department of Computer Science, Kuwait University, Kuwait
najem@cs.ku.edu.kw

ABSTRACT

The incremental evaluation of logic programs maintains the tabled answers in a complete and consistent form in response to the changes in the database of facts and rules. The critical challenges for the incremental evaluation are how to detect which table entries need to change, how to compute the changes and how to avoid the re-computation. In this paper we present an approach of maintaining one consolidate system to cache the query answers under the non-monotonic logic. We use the justification-based truth-maintenance system to support the incremental evaluation of tabled Prolog Programs. The approach used in this paper suits the logic based systems that depend on dynamic facts and rules to benefit in their performance from the idea of incremental evaluation of tabled Prolog programs. More precisely, our approach favors the dynamic rules based logic systems.

KEYWORDS

Incremental evaluation of tabled Prolog, Incremental tabulation for Prolog queries, Justification based truth maintenance systems, Tabulation, Memoing.

1. INTRODUCTION

Tabled resolution for logic programs [1] mitigates some of the well-known problems of Prolog, including the tendency to fall into infinite loops, repeating subcomputations, and the unsatisfactory semantics of negation. The implementations of tabling [2, 3, 4,5] have become stable and efficient. The incremental evaluation of logic programs [6] maintains the tabled answers complete and consistent in response to the changes in the database of facts and rules. The basic idea behind incremental tabulation is that when some facts or rules change in a program, the system recomputes only the results affected by the change, instead of re-evaluating and tabling the query answers from scratch. The critical challenges for the incremental evaluation are how to detect which table entries need to change, and how to compute the changes. One of the efficient approaches to achieve these challenges is to use the symbolic support graph [7]. The symbolic

David C. Wyld et al. (Eds) : CST, ITCS, JSE, SIP, ARIA, DMS - 2014
pp. 311–321, 2014. © CS & IT-CSCP 2014 DOI : 10.5121/csit.2014.4129

support graph caches the dependencies between the tabled answers to propagate the changes to the tables when the related facts/rules are added/deleted. This approach requires to cache the answers of the query in a table along with the support graph to maintain the completeness and correctness of tabled answers.

$:-table\ connected/2$
$edge(a,b)$ %F1
$edge(a,c)$ %F2
$edge(b,d)$ %F3
$edge(c,d)$ %F4
$edge(d,e)$ %F5
$edge(f,g)$ %F6
$connected(X,Y):-edge(X,Y).$ %R1
$connected(X,Y):-edge(X,Z),connected(Z,Y).$ %R2

Figure 1: Translative closure program of the directed edge relationship

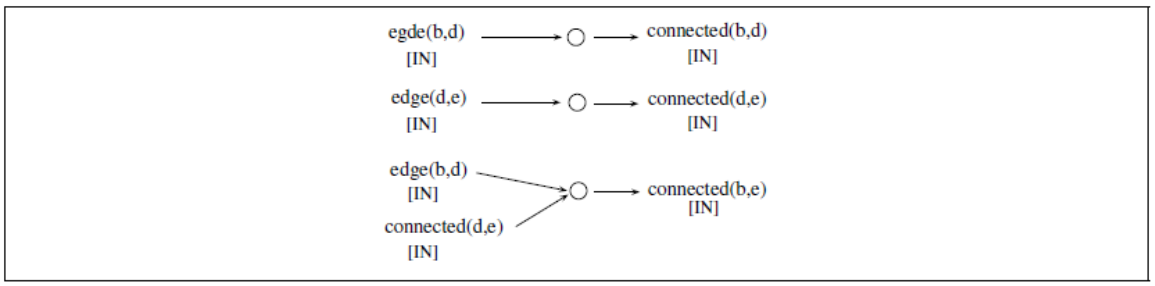

Figure 2: JTMS network installed by THE SYSTEM after proving query ? – *connected* (b, Y) for the first time.

The other challenge for the incremental evaluation is to avoid the re-computation which is required to update the tabled answers due to the changes in the related database of facts and rules. The current technique [8] uses extra data structures (dynamic dependency graph) to interleave the propagation of deletion and insertion operations caused by the updates of facts and rules. The technique tries to minimize the challenging problem of re-computation which is caused by the updates. This paper presents an alternative approach to incremental tabulation that is capable of working in non-monotonic situations. The main idea is to cache the proof generated by the deductive inference engine rather than the end results. In order to be able to efficiently maintain the proof to be updated, the proof structure is converted into a justification-based truth-maintenance (JTMS) network [9, 10].

2. CACHING THE QUERY PROOF AS A JTMS NETWORK

The main idea of our approach is to cache the proof generated by the deductive inference engine rather than caching the end results. The proof structure is converted into a justification-based truth-maintenance (JTMS) network. JTMS saves the dependency between deduced facts and the facts used to make the deduction in order to be able to efficiently cache the proof structure. The system translates every successful branch of a query into a JTMS network that links the facts and

the rule to the answer generated by that branch. Consider the evaluation of the query: ? - *connected*(b,Y) with respect to the PROLOG program of Figure 1. Figure 2 shows the justifications installed by the system when it proves the query ? - *connected* (b,Y) with respect to the PROLOG program of Figure 1. These justifications represent the proof structure of the query ? - *connected*(b,Y). A justification is installed for each complete branch of the SLD-tree. When a query is reevaluated, the system returns the answers of the query by collecting the IN consequences of each query's JTMS justification.

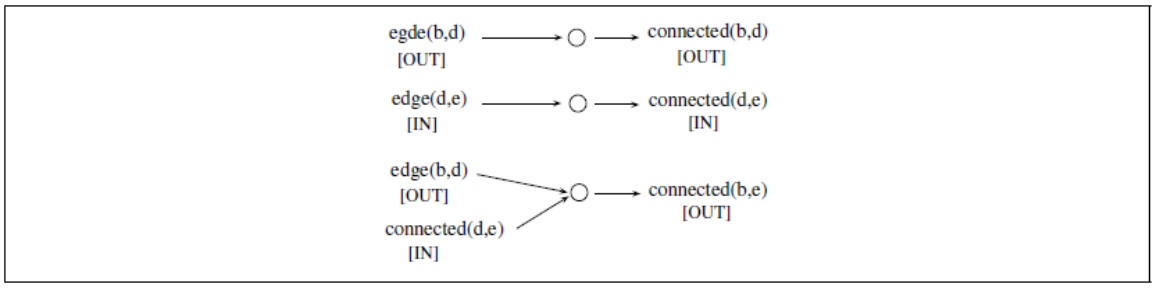

Figure 3: JTMS network of Figure 2 after retracting the fact *edge*(b, d) to the database of Figure 1.

When changes in database take place, we have to ensure that the proof structure is both sound and complete. Consider the following changes to the base facts of Figure 1 after caching the proof structure of the query ?-*connected*(b,Y) for the first time:

1. Retracing the fact *edge*(b,d)

   The system has to ensure that whenever base facts participating as antecedents in any justification are asserted/retracted, the effect of this assertion/retraction should be propagated through the JTMS justifications in order to keep the proof structure sound. Achieving this is not difficult since changing the state of any antecedent that is asserted/retracted to/from the database requires marking the label from IN/OUT or vice versa, and after that, propagating the effect of this change through the whole network. Figure 3 shows the effect of retracting the fact *edge*(b, d) from the database of the Figure 1. The first effect of this retraction is on the first justification since *edge*(b,d) is in the antecedent list of that justification. This results in marking *connected*(b,d) from IN to OUT. Since *edge*(b,d) is in the antecedent list of the 3rd justification, the result of outness propagation marks *connected*(b,e) from IN to OUT. This method of propagating inness/outness ensures that whenever the query is re-evaluated, the returned results by the system are valid answers regardless whether or not the database has been changed. Note that ensuring the soundness of the proof structure does not require any PROLOG inference work.

2. Asserting the fact *edge*(b,f)

   Here the situation is more complicated. the system has to take care about the effect of asserting new data that was not available when a query was evaluated for the first time. This is important since asserting new data to the database may add to the set of results that are already available for the query or even remove some of them. The system handles this problem by monitoring the nodes that may contribute to some new results of the query.

Whenever a new fact that is related to a monitored node is asserted, query resumption takes place to update the query's cached proof structure. Referring back to the example of Figure 1, when the system proves the query ?-*connected* (b,Y) for the first time, it marks the nodes that will participate in resuming this query when new data is asserted. Those nodes come from the right hand side of program rules, i.e. *edge*(X,Y) and *connected* (Z,Y). Whenever new data that is related to the marked nodes is asserted, the query ?-*connected*(b, Y) resumes its work to update the proof structure of the query. Figure 4 shows the effect of asserting the fact *edge*(b, f) to the database of Figure 1 on the JTMS network of Figure 3. Three new justifications have been installed upon resuming the query after the assertion of *edge*(b, f). An important point that should be mentioned here is that, in order to keep the proof structure complete, the system has to use the help of the PROLOG inference engine.

3. Asserting the fact *edge*(b,d)

The retracted fact *edge*(b,d) is asserted back to the base facts of Figure 1. The system is going to change the label of the TMS node attached to this fact from OUT to IN and then propagates the effect of this change in label throughout the JTMS network. Figure 5 shows the effect of asserting back the fact *edge*(b,d) to the database of Figure 1 on the JTMS network of Figure 4.

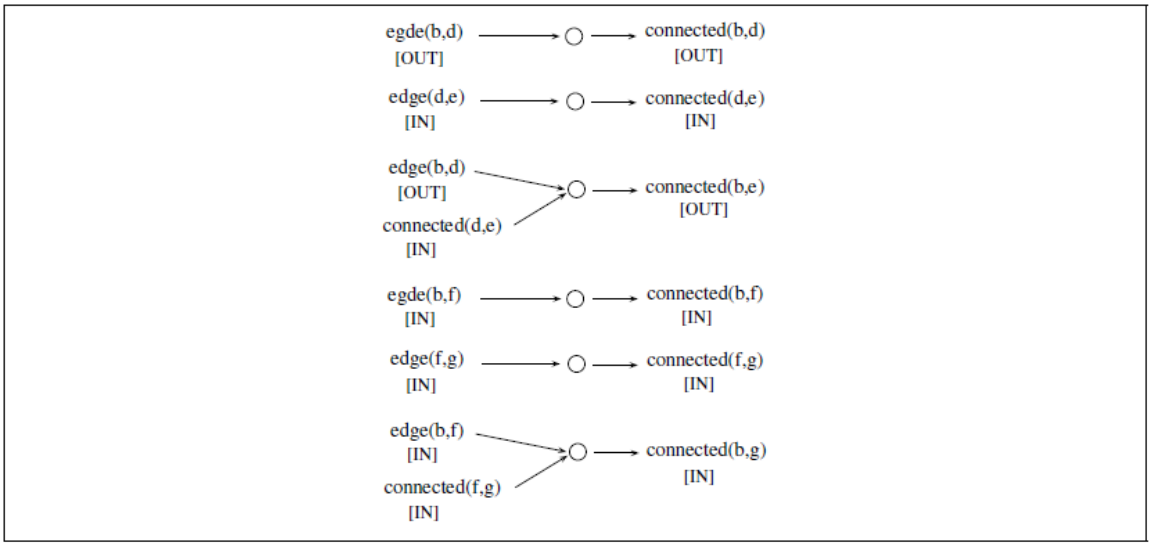

Figure 4: JTMS network of Figure 3 after asserting the fact *edge*(b, f) to the database of Figure 1.

3. IMPLEMENTATION

The main objective of this research is to provide a PROLOG system which supports incremental tabulation by using the justification-based truth-maintenance system, and this is what we achieved. The system evaluates the query only once with maintaining enough information to ensure both consistency and completeness of the collected solutions as the dynamic state changes. When the query is re-evaluated, the system returns the cashed answers which are always up to

date. There are two approaches to integrate tabling support into existing PROLOG systems. The first approach is to modify and extend the low-level engine. The advantage of this approach is the run-time efficiency, however, the drawback is that it is not efficiently portable to other Prolog systems because the engine level modifications are slightly more complex and time consuming. This approach is used by the XSB [2] system. XSB is the only PROLOG implementation so far that supports incremental tabulation. The second approach to incorporate tabled evaluation into existing PROLOG systems is to apply the source level transformations to a tabled program, and then use external tabling primitives to provide direct control over the search strategy. This idea was first explored by Fan and Dietrich [11] and later used by Rocha, Silva and Lopes [12] to implement tabled PROLOG systems. The main advantage of this approach is the portability of applying it on different PROLOG systems. The drawback is of course the efficiency, since the implementation is not at a low level. Our implementation approach is based on applying the source level transformations to a tabled program. We named our approach as JLOG (Justification-based Logic), the idea of this name came from the word PROLOG (Programming in logic).

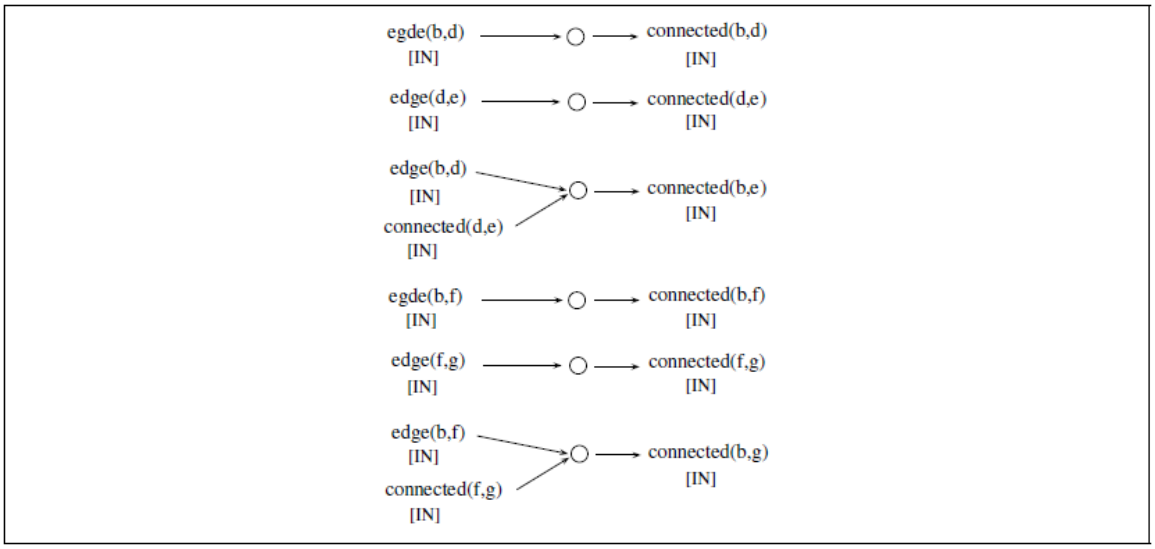

Figure 5: JTMS network of Figure 4 after asserting back the fact *edge*(b,d) to the database of Figure 1.

4. RESULTS AND DISCUSSION

To have a look at the performance of the system, JLOG is to compare it with:

1. Normal PROLOG (NP) implementations [13] that do not support tabulation.

2. Tabled PROLOG (TP) implementations [4, 2] that support monotonic (static facts and rules) logic systems.

3. Incremental tabled PROLOG (ITP) implementations [2] that supports non-monotonic (dynamic facts and rules) logic systems. This is considered to be the main assessment factor since the main objective of this research is to support incremental tabulation. Our

benchmark is the XSB system since it is the only PROLOG implementation that so far supports the incremental tabulation.

The main assessment factors for testing the performance of our approach are categorized into the following:

1. Evaluating the query for the first time

   We execute PROLOG queries on normal, tabled, incremental tabled PROLOG and JLOG. The execution time of these queries is analyzed and compared among the four systems.

2. Re-evaluation of a query

   Once a query is evaluated for the first time, the same query is re-evaluated again on normal, tabled, incremental tabled PROLOG and JLOG. The execution time of re-evaluating this query is analyzed and compared among the four systems.

```
edge(Sem,S1,S2):-    reg(Sem,Course,S1,Section),
                     reg(Sem,Course,S2,Section),
                     S1 < S2.
connected(Sem,X,Y) :-  edge(Sem,X, Y).
connected(Sem,X,Y) :-  edge(Sem,X, M), connected(Sem,M, Y).
```

Figure 6: Translative closure PROLOG program to find the connected students in a certain semester.

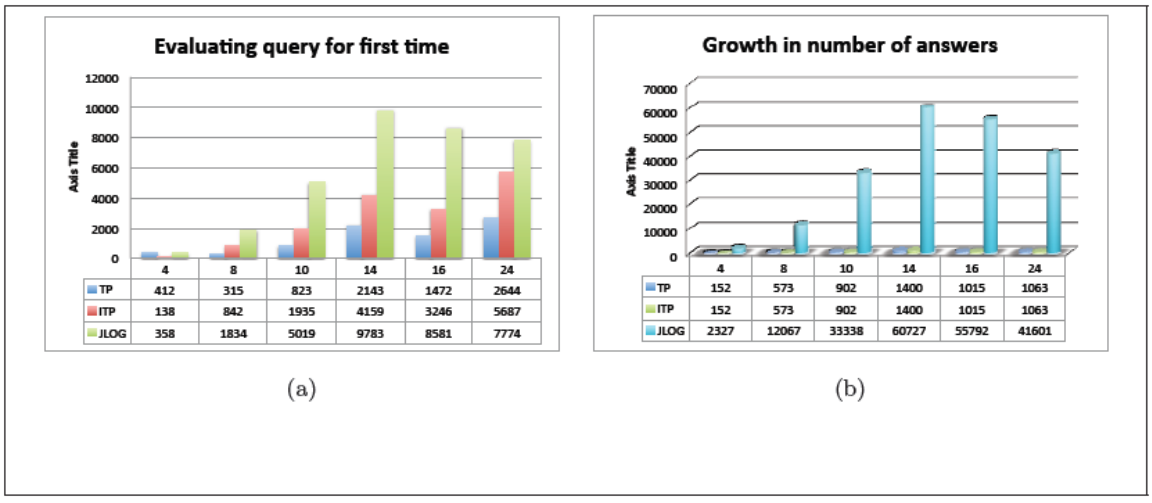

Figure 7: Statistics of evaluating the query *connected*(S*em*, 946, Y).

3. Evaluating a subquery related to a previously proven query

   We compare the time it takes to evaluate subqueries related to some previously proven queries on normal, tabled, incremental tabled PROLOG and JLOG.

4. The cost of maintaining the cashed proof structure up-to-date for a previously proven query

   After the query is proven for the first time, we assert/retract PROLOG facts or rules to/from the PROLOG program that would change the state of the cached answers of the query.

We tested the performance of JLOG by implementing a small business intelligence [14], or a reporting tool for a mid-size University. We have chosen this approach rather than the standard benchmark dataset to test the system on real data. The objective is to observe if the system is able to work under real applications. Graph reachability is a classic problem with many applications in the real-world. The graph reachability problem has been used as a benchmark in any PROLOG tabled implementation. We mapped the graph reachability to the student information database using the following scenarios:

- Picking a certain student in an academic semester, we would like to know the set of students that can be reached from, connected to, this particular student. We used the assumption that all students registered in the same class are connected to each other, i.e. we add an edge between each couple of students registered in the same class (There are so many scenarios in the student information system that can be mapped to the reachability problem, we just picked one example). Then we apply the transitive closure; if student X is connected to Y, Y is connected to Z; we conclude that X is connected to Z.

- In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph. In a student information database, we would like to know how many connected components of students exist in a certain semester. Each student registered in the current semester is represented as a vertex in the graph. Whenever two students are registered in the same class, we add an edge between these two students (vertices) in the graph

Figure 6 shows the translative closure PROLOG program to find the connected students in a certain semester. The first rule in the program connects each couple of students registered in the same class. The second rule uses the transitive relation to connect students indirectly. Given the enrollment data up to a certain academic year, we would like to list all the students connected to a particular student. For example, the query *connected*(S*em*,946,Y) finds the list of students connected to the student number 946 in all the semesters that exist in the database of facts. Figure 7 presents the statistics of evaluating the query *connected*(S*em*,946,Y) for the first time. The graph that is going to be constructed from the relation edge/3 contains cycles which yields that this query suffers from infinite loop in Np while it terminates successfully in all tabled (Tp, ITp, JLOG) runs. The query generates a lot of redundant answers which are neglected by Tp and ITp. These answers are not neglected by JLOG, hence it is suffering from overhead when the query is proved for the first time. To test the correctness (soundness) and completeness of the cached

answers, we picked samples of the data such that the number of edges (facts), coming from the reg/4 predicate, is between 2 to 4kb. The sample takes snapshot of the data before the first day of classes, i.e. start of add/drop period in the university. First we evaluate the general query related to this program which is *connected*(S*em*,X,Y). We pass the semester values for which we are looking the list of connected students. Then, we use the following scenarios to test the soundness and completeness of the cached proof structure:

1. We track all the changes that take place on the predicate reg/4 starting from the 1st day of add drop period until the end of semester. When a student drops (soundness) a class, the related PROLOG fact is retracted. When a student adds a new class, then the fact is asserted. This can be a new fact (completeness) if the student is adding the class for the first time, or it can be an old fact (soundness) because the student dropped the class after registering it for the first time and then decided to reenroll back in the class. The current version of JLOG updates the JTMS network, attached to the cached query, whenever the assert/retract command is executed. This means that the query proof structure is always updated and returns the correct answers. Figure 8 shows the statistics of maintaing the soundness and completeness of the query *connected*(S*em*,X,Y) based on the changes in the predicate reg/4. For the same add/drop events ITp (XSB) is faster than JLOG. The reason behind this difference in the performance is coming from the fact that JLOG is updating the JTMS network after each assert or retract command, while ITp is handling the situation through batch processing since it is implemented at low level. When the tables were updated after each assert or retract command in ITp, the performance of the system was degraded. For example, for the semester 1101, ITp takes 47608 milliseconds to update the query answers through batch processing, see Figure 8. This time jumps to 12,972,825 milliseconds when we tried to update the tables after each assert/retract command. JLOG updates the JTMS network after each assert/retract in 120,842 milliseconds which is significantly lower than the time taken by ITp to handle the events one by one.

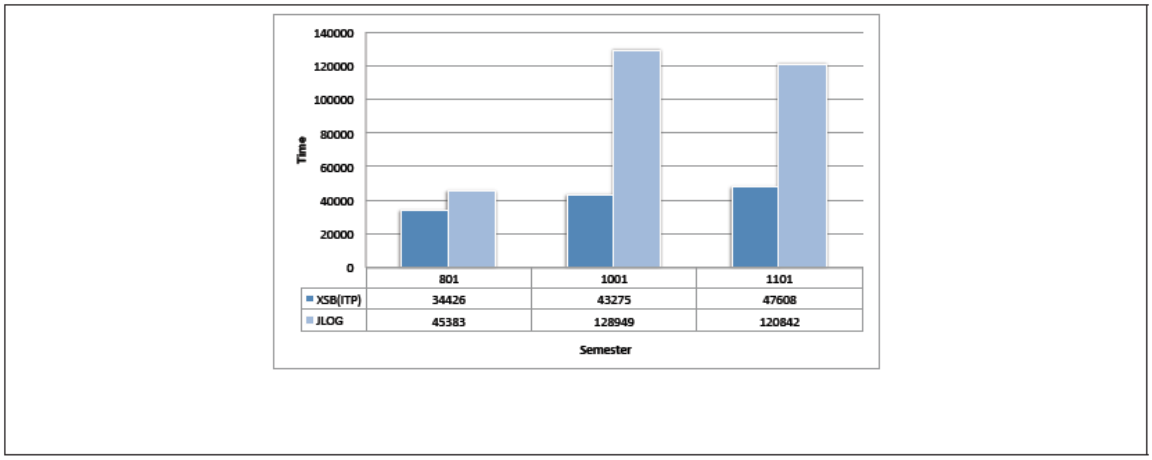

Figure 8: Statistics of maintaing the soundness and completeness of the query *connected*(S*em*,X,Y) based on the changes in the predicate reg/4.

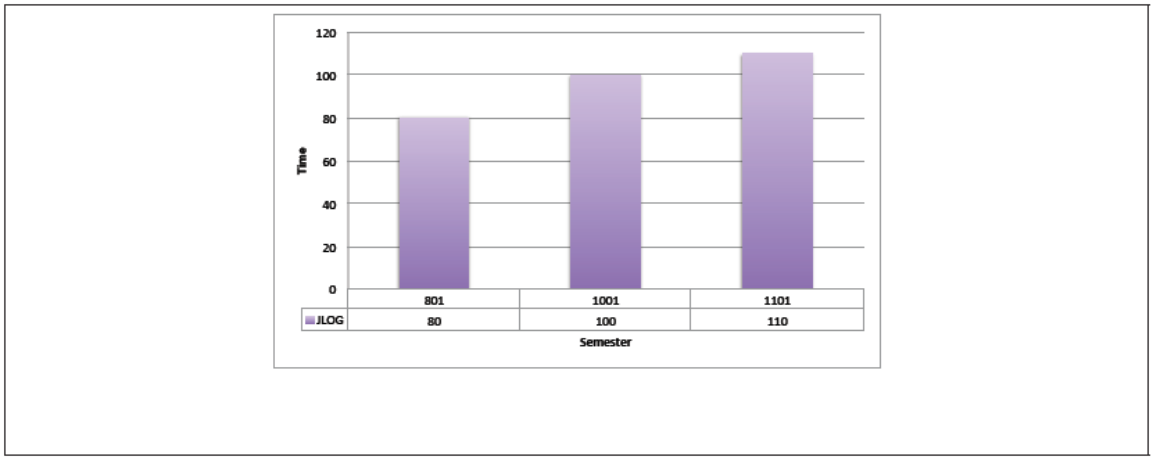

Figure 9: Maintaing the soundness of the query *connected*(S*em*,X,Y) after retracting of the rule *connected*(X,Y) : -*edge*(X,M), *connected*(M,Y) from the PROLOG program of Figure 6.

2. The second scenario is used to test soundness of the query which is related to assertion/retraction of rules in the program of Figure 6. Consider the case where we would like to know the list of students who are connected directly and we want to exclude the tuple of students who are connected indirectly. This can be achieved by retracting the third rule in the program of Figure 6 which connects the students indirectly. In order to be able to retract the rule, the predicate *connected*/2 must be defined as incrementally dynamic. Once *connected*/2 is defined as dynamic predicate, the query *connected*(S*em*,X,Y) does not terminate in ITp. ITp fails to handle the query due to an infinite loop. JLOG handles the situation smoothly. Figure 9 shows the time taken by JLOG for maintaing the soundness of the query *connected*(S*em*,X,Y) after retracting of the rule *connected*(X,Y) : -*edge*(X,M), *connected*(M,Y) from the PROLOG program of Figure 6. JLOG handles the retraction easily because it is a single event used to update the JTMS network and does not require any inference work from the PROLOG side.

5. CONCLUSION

This paper proposed a general framework of a subsystem integrated with PROLOG inference engine (SWIPROLOG, YAP-PROLOG, XSB, ..., etc) that uses justification-based truth maintenance system to support incremental tabulation that can work under nonmonotonic logic. Our system evaluates a query only once, maintaining enough information to ensure both consistency and completeness of the collected solution as the dynamic state changes. The main idea of the system is to cache the proof generated by the PROLOG inference engine as a JTMS network rather than saving the end results as it is the case for most tabling systems. The approach presented in this paper is suitable for a query that depends on dynamic information is to be evaluated repeatedly as the dynamic state changes. The are few advantages of our approach. The first advantage comes from caching the query answers in one consolidate subsystem (JTMS Network). Then the evaluation of subqueries requires no inference work. Another system advantage is related to handling the assertion/retraction of rules. On the other hand, the approach is suffering from few limitations. The first limitation of the system is the inability to handle queries with infinite answers. The other limitation of current approach occurred when the query is evaluated for the first time. JLOG is paying sufficient overhead since it caches the proof structure

of the query rather than the end results. JLOG is not a good choice for the queries generating a large number of answers. The large number of answers for a query requires large JTMS network to be installed for the query in order to cache the proof structure. We need to study carefully the memory usage of JLOG and see how this issue can be resolved by controlling or compacting the memory management for the JTMS network.

REFERENCES

[1] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. J. ACM, 43(1):20–74, January 1996.

[2] Terrance Swift and David Scott Warren. Xsb: Extending prolog with tabled logic programming. TPLP, 12(1-2):157–187, 2012.

[3] Ricardo Rocha, Fernando Silva, Ricardo Rocha Fern, and Vítor Santos Costa. Yaptab: A tabling engine designed to support parallelism. 2000.

[4] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. TPLP, 12(1-2):5–34, 2012.

[5] Neng-Fa Zhou, Isao Nagasawa, Masanobu Umeda, Keiichi Katamine, and Toyohiko Hirota. Bprolog: A high performance prolog compiler. In Takushi Tanaka, Setsuo Ohsuga, and Moonis Ali, editors, IEA/AIE, page 790. Gordon and Breach Science Publishers, 1996.

[6] Diptikalyan Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In ICLP, pages 392–406, 2003.

[7] Diptikalyan Saha and C. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In Maurizio Gabbrielli and Gopal Gupta, editors, Logic Programming, volume 3668 of Lecture Notes in Computer Science, pages 235–249. Springer Berlin / Heidelberg, 2005.

[8] Diptikalyan Saha and C. Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In Sandro Etalle and Miroslaw Truszczynski, editors, Logic Programming, volume 4079 of Lecture Notes in Computer Science, pages 56–71. Springer Berlin / Heidelberg, 2006.

[9] Truong Quoc Dung. A revision of dependency-directed backtracking for jtms. In Günther Görz and Steffen Hölldobler, editors, KI, volume 1137 of Lecture Notes in Computer Science, pages 57–60. Springer, 1996.

[10] Gerhard Brewka, David Makinson, and Karl Schlechta. Jtms and logic programming. In LPNMR, pages 199–210, 1991.

[11] Changguan Fan and Suzanne Wagner Dietrich. Extension table built-ins for prolog. Softw. Pract. Exper., 22(7):573–597, July 1992.

[12] R. Rocha, C. Silva, and R. Lopes. Implementation of Suspension-Based Tabling in Prolog using External Primitives. In J. Neves, M. Santos, and J. Machado, editors, Local Proceedings of the13th Portuguese Conference on Artificial Intelligence, EPIA'2007, pages 11–22, Guimarães, Portugal, December 2007.

[13] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. Theory and Practice of Logic Programming, 12(1-2):67–96, 2012.

[14] Oksana Grabova, Jerome Darmont, Jean-Hugues Chauchat, and Iryna Zolotaryova. Business intelligence for small and middle-sized enterprises. SIGMOD Record, 39(2):39–50, June 2010.

AUTHORS

Tahir M. Ali received his BSc and Ms from Kuwait University and PhD from University of Malaya. He is currently an Assistant Professor of Computer Science in Gulf University for Science and Technology, and also serving as the IT director. His main research interest is in field of Artificial Intelligence (AI), in particular, logic programming and scheduling algorithms.

Ziad H. Najem received his BSc from Kuwait University and Ms and PhD from University of Illinois at Urbana-Champaign. Prior to joining the Department of Computer Science at Kuwait University in 1999, Dr. Najem worked as a Scientific Researcher at Kuwait Institute for Scientific Research.

Mohd Sapiyan Baba is currently a Professor of Computer Science in Gulf University for Science and Technology, Kuwait. He was a lecturer in University of Malaya for more than 30 years, teaching Mathematics and Computer Science courses, and supervised numerous students for their research projects at undergraduate and postgraduate levels. His main research interest is in field of Artificial Intelligence (AI), in particular, the application of AI in Education