IJCTA, 9(10), 2016, pp. 4563-4574
© International Science Press

Graphical Representation of Prolog Traces

***Indulekha T S *Akhil G Nair *Harikesh P**

Abstract : While designing programming environments, one should consider a very easily understandable way to explain it to the user how their program works. For a programmer who migrates from a procedure oriented language to a logic programming language like Prolog, it is difficult to understand the search strategy used while it derives new information from the existing knowledge base. But for a Prolog programmer it is essential to understand how the unification and searching process are done in order to write and debug programs. Here, a tool which can graphically show the order in which Prolog searches and backtracks for the answer of a particular query is presented. To understand the program execution, a model of the programming language is essential. For that, the model should include some information about the search space, the flow of control through the search space and the Prolog clauses. The proposed system will be helpful for the teaching purpose. The graphical representation of these Prolog trace is very much helpful for the learner to understand how the query has been traced from the whole knowledge base. Instead of showing only the correct path, the system could show failed paths also. It is mainly intended to make the students understand, how the Prolog trace works.

Keywords : logic programming, unification, backtracking, knowledge base, traces, graph.

1. INTRODUCTION

Prolog stands for Programming in Logic which is associated with both artificial intelligence and computational linguistics [1]. It is a logic programming language and generally, we use predicates to describe the relationships or certain properties between individuals or objects. The Predicate calculus gives a good foundation to the logic programming languages, such as Prolog. It is a form of symbolic logic. It also includes quantifiers (such as, for all:" and there exists:").

Prolog works with a knowledge base, which contains stored information. The information represented in the knowledge base are Facts, Rules and Query. They are treated as relations. Fact is just an information in the knowledge base.

Eg. mother(liza,david). says liza is the mother of david.

Rules are conditional statements about our facts.

Eg. parent (X,Y) :-father(X,Y).

defines a rule, X is the parent of Y if X is the father of Y. Then we can query or question the contents in the Knowledge base and Prolog finds out the answer for it.

All Prolog data structures are called terms. A term is either: a constant(which can be either an atom or a number) or a variable. Prolog uses Backtracking and Unification for finding answer for the posed query. In Prolog more than one rule can match a goal. In such case Depth-first search is used with backtracking. Backtracking helps to find alternate solutions for a given query. Unification is the two-way matching of goal with clause head. It is needed for constructing answers to goals. The trace allows the programmer to see all the facts and rules that are executed as part of the query in sequence, along with whether the goal is succeeded or not.

* Department of Computer Science and Applications, Amrita Vishwa Vidyapeetham, Kollam, indulekhats@gmail.com, nanduakhilg @gmail.com, harikeshmp@gmail.com,

To make the deduction process simpler Prolog statements are expressed in a form called horn clauses. Pure Prolog programming defines relations in Horn-clauses. Prolog syntax is largely based on variant of Horn clause. Horn clause consist of a head h which is a predicate and a body containing list of predicates,

$$p1\ ,\ p2\ ,\ ..pn$$

This can be written as, $h\ \leftarrow p1\ ,\ p2\ ,\ ..pn$

It means that h is true if all the *p*'*s* are true [7].

Horn-clause is a logical formula in a particular rule like form, which contains at most one positive literal. A Horn- clause belongs to one of four categories:

a **A rule :** 1 positive literal, at least 1 negative literal. b. A fact or unit: 1 positive literal, 0 negative literals.

***c*.** **A negated goal :** 0 positive literals, at least 1 negative literal.

***d*.** **The null clause :** 0 positive and 0 negative literals.

A query can be run on the relations for computation. A query or question is the goal clause.

Prolog interpreter uses logical methods to resolve queries[8]. Prolog interpreter automatically chooses the fact and rule needed for solving a query. It starts by solving each goal in a query, left to right. For each goal, a corresponding fact or head of a rule is matched. The matching of fact or rule is known as unification. Prolog uses backtracking, when a goal can't be matched further. At this point, Prolog backtrack to the point, where a choice was made of matching a particular fact or rule. From here Prolog will try to match a different fact or rule. Backtracking is the process that works after failing a sub goal, the Prolog system automatically checks for the previous goal and tries to re satisfy it [4].

Prolog debugging is made easy with tracing. We can trace the execution of a Prolog query which will help us to view all the facts and rules that are executed as part of the query, in sequence. It also shows whether the goal is succeeded or not[12].

Tracing in Prolog can be enabled using 'trace' command. Understanding how a Prolog query is carried out is as important as understanding the Prolog language itself. The learner of a language needs a good understanding of what the computer do with their query and how it is carried out [1]. The objective is to represent the trace graphically. This will help a learner, to understand how Prolog finds out answer for a posed query. The representation will help to understand

2. **Variable :** Denoted by string of letters or numbers and starts with either an upper-case letter or underscore.
3. **Complex terms :** (*e.g*: functor(term1, ...term *n*). We say two terms match if they are equal, or the variables in the terms results equal terms.

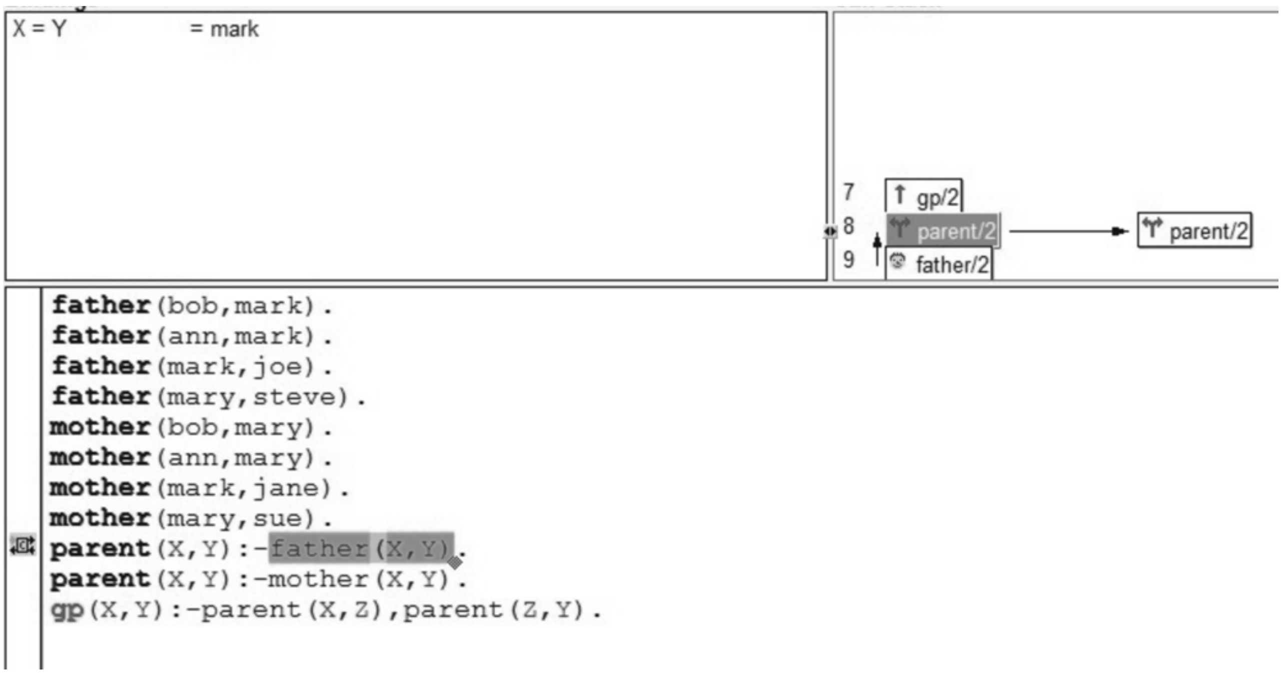
   The algorithm used by Prolog to unify two terms (say Tm1 and Tm2) is as follows;how backtracking and unification happens as Prolog searches for the answer. A graphing API called JUNG is used to draw graphs. The query trace will be obtained and represented as a graph.

2. RELATED WORKS

SWI-Prolog is a free implementation of Prolog program- ming language, which provides an efficient and fast Prolog environment[7]. SWI-Prolog provides an environment with advanced debugging capabilities.

Using traces while learning Prolog is a way to understand Prolog programming language. In trace mode, Prolog will show in step by step how it is finding the answer for a given query. In SWI-Prolog, trace of a Prolog query can be obtained using trace command. SWI-Prolog also have a graphical tracing representation. Graphical trace can be obtained using the 'gtrace' command [5].

The trace obtained using 'gtrace' command is shown in [fig. 1]. The graphical trace obtained like this is limited to showing only the successful flow of the Prolog query trace. It doesn't show the failed paths.



```
father(bob,mark).
father(ann,mark).
father(mark,joe).
father(mary,steve).
mother(bob,mary).
mother(ann,mary).
mother(mark,jane).
mother(mary,sue).
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
gp(X,Y):-parent(X,Z),parent(Z,Y).
```

Fig. 1. gtrace in SWI-Prolog.

3. UNIFICATION AND PROOF SEARCH

A. Unification

Unification is the process by which Prolog matches two terms. Three types of terms are there in Prolog.

1) Constants: It can be either atoms (e.g: kiran) or number
(*e.g*: 94).

```
if both terms are constants then
if T m1 =Tm2 then return success;
else
return fail;
end
else if one term is a variable then
instantiate that variable with the other
end
else if both terms are complex terms and they have the same no. of arguments(arity) then
→ f ind the principal f unctor Fr1 of Tm1
→ f ind the principal f unctor Fr2 of Tm2
if F r1=F r2 then
if each argument ai , (where i =
1..n)of Tm1 unif y with each argument bi (where i =
1..n)of Tm2 then
return success;
end else
return fail;
```

end

e.g: If the query is = (kiran, kiran).

the response will be 'yes' and if the query is = (kiran, bhadra). the response will be 'no'.

(The operator = is used to unify two terms).

Unifying friends(kiran, X) with friends(Y, Bhadra)

- Both are complex terms with arity 2.
- friends is the principal factor of both terms.
- Here a_1 = kiran and a_2 = X and b_1 =Y and b_2 = bhadra.
- So unify a_1 with b_1 and a_2 with b_2.
  Y is a variable and is instantiated to kiran.
  X is a variable and is instantiated to bhadra.
- The term after unification is friends(kiran, bhadra).

B. Proof search

When a query is given, Prolog searches the knowledge base to see if the query is satisfied.

Consider the following knowledge base.

intelligent (*a*). intelligent (*b*). wellmannered (*a*). wellmannered (*b*). hardworking(*b*).

good(X):-intelligent(X), wellmannered(X), hardworking(X).

Suppose we pose the query

good(X)

There is only one answer to the query, good (*b*). Here, X is implied to *b*.

Prolog searches the knowledge base from top to bottom. *i.e*., it tries to find out a match in the first place possible (if it can). When Prolog reads the query good(X) it looks for a match either in a fact or the head of a rule.

In this case, there is only one possibility.

good(X):-intelligent(X), wellmannered(X), hardworking(X).

When Prolog finds a match, it generates a new variable. Let's say the variable name is _G1713.

So, the query becomes

good(_G1713)

from this knowledge base it is known that,

good(_G1713): intelligent(_G1713),wellmannered(_G1713), hardworking(_G1713).

Now the query says : 'find a person that has property good'.

The rule says ' a person has property good, if it has properties intelligent, hardworking and wellmannered.

The graphical representation of this is shown in [fig. 2].

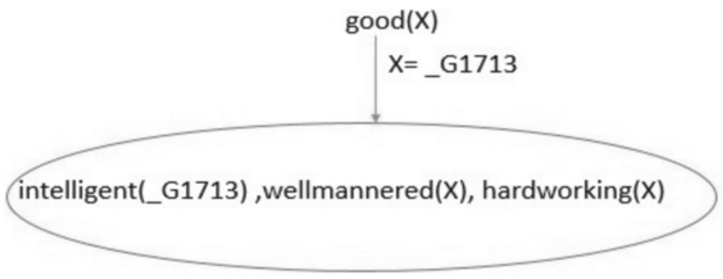


Fig. 2. Elaborated graph of the sample query

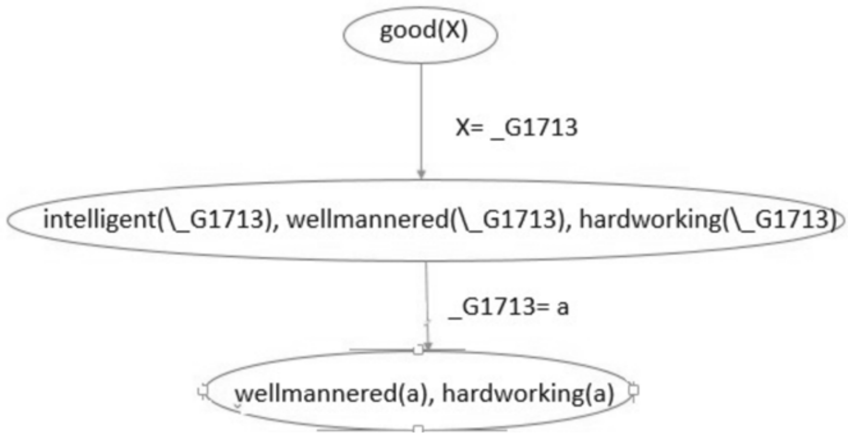

good(X)
X= _G1713
intelligent(_G1713), wellmannered(_G1713), hardworking(_G1713)
_G1713= a
wellmannered(a), hardworking(a)

Fig. 3. Graph of the sample query.

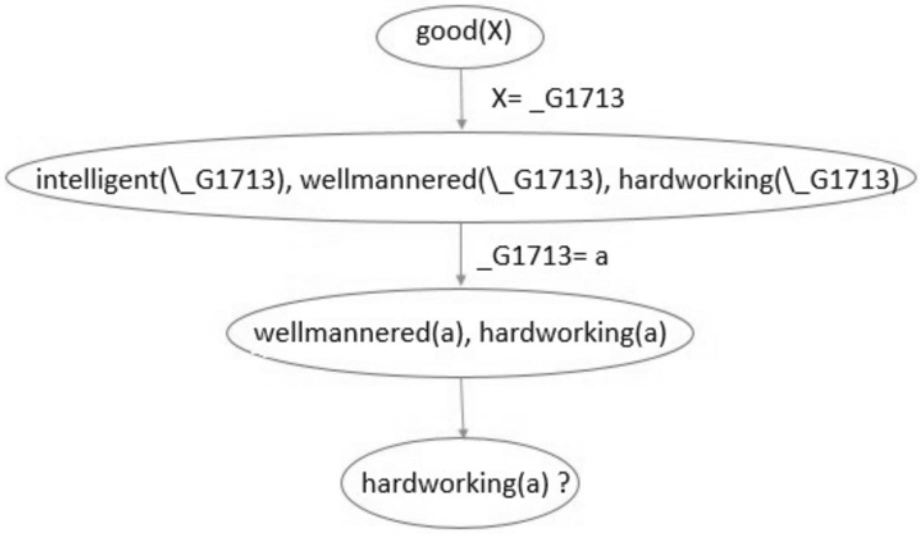

good(X)
X= _G1713
intelligent(_G1713), wellmannered(_G1713), hardworking(_G1713)
_G1713= a
wellmannered(a), hardworking(a)
hardworking(a) ?

Fig. 4. Graph of the sample query.

Now Prolog tries to find another match for the first goal, intelligent(_G1713). It is satisfied in the fact, intelligent(b). Now the value for the variable, _G1713 becomes ‘b’. The goal list is now reduced to wellmannered(b), hardworking(b). The knowledge base contains fact, wellmannered(b) and thus it is matched. The goal list is now left with hardworking(b). This goal too is satisfied, as the fact is contained in knowledge base, leaving the goal list empty. This means the original query is satisfied and the variable, X is instantiated to ‘b’. Now the search graph looks as shown in [fig.5].

Now it has got a list of goals. Prolog tries to satisfy each goals starting from left to right by searching the knowledge- base in a top down fashion.

When it satisfies the first goal intelligent(_G1713) with the fact intelligent(a), the variable X is set ‘a’. And this instantiation will be applied to all occurrences of X in the goal list. Now, the remaining goals are : wellmannered(a), hardworking(a)? Now the proof search graph looks as given in [fig.3].

Now Prolog will try to check if wellmannered(a) is in the knowledgebase, which is the next goal after intelligent(a). wellmannered(a) is in the knowledgebase.

The goal, wellmannered(a) can’t be matched with any facts in the knowledge base. So it backtracks to the last point where it had multiple choice points. This point is where it has the goals, intelligent(_G1713), wellmannered(_G1713), hardworking(_G1713). Now the proof search graph looks as given in [fig.4].

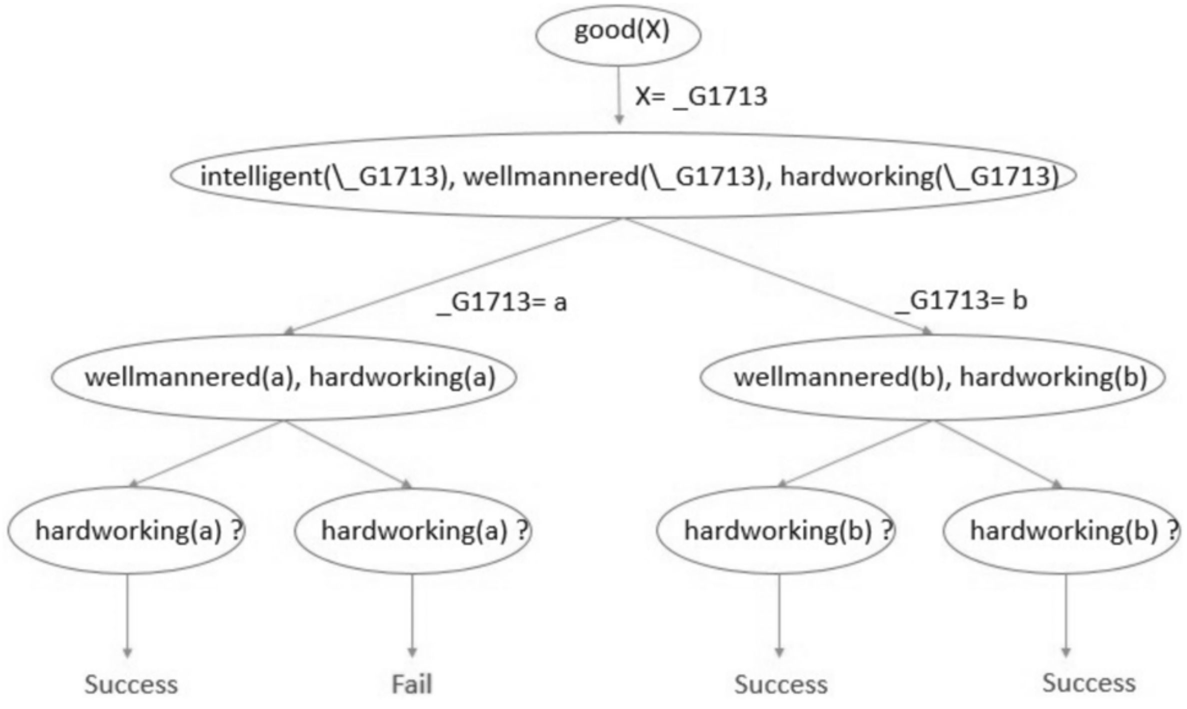

good(X)
X= _G1713
intelligent(_G1713), wellmannered(_G1713), hardworking(_G1713)
_G1713= a
_G1713= b
wellmannered(a), hardworking(a)
wellmannered(b), hardworking(b)
hardworking(a) ?
hardworking(a) ?
hardworking(b) ?
hardworking(b) ?
Success
Fail
Success
Success

Fig. 5. Graph of the sample query.

The graphical representation depicted has a tree structure. The vertices represent the goals that have to be satisfied, and the edges show the variable instantiation done when a match occurs. The leaf nodes contain either the label Success or Failed depending on whether the goal was satisfied or not. The goal is to construct such a tree when a query is posed.

4. METHODOLOGY

A. Obtaining the trace

Trace shows the step by step process of unification. Each line in the trace starts with one of the four following commands: Call, Exit, Fail and Redo.

The step which starts with 'Call' represents the goal that it is attempting to prove at the moment. The command 'Fail' means that the goal failed completely and no more solutions for that goal can be obtained.

The Command 'Exit' means it successfully found solution for the goal clause.

The command 'Redo' means that it is trying to find an alternate way to prove the goal which is already attempted to prove before.

B. Plotting the graph

The Graph Plotting algorithm takes the file, containing trace of the posed query, as input. Each line in the file is processed for plotting the graph. Each line contains three parameters separated by a space.

1. **Current command :** This is the command currently executed. It can be one among the four commands in the trace, which are 'Call, Redo, Exit and Fail'.
2. **Current Level :** This is the current level of proof searching in the depth first searching method.
3. **Current Predicate :** This is the predicate that is being executed according to the command.

The algorithm will plot vertices, edges and rename vertices considering these three parameters. The original query will be plotted as the root vertex, which have the least number for indicating the level of proof search and it starts with the command 'Call'. After processing one line of the trace, all the three parameters are stored as previousCommand, previousLevel and previousPredicate. Then the execution will be moved to the next line of trace. If the line starts with the command ‘Call’, then a vertex is added with a name as the currently processed predicate and an edge is plotted between this vertex and previously added vertex (which is previousPredicate). After this, the last executed predicate will be pushed into a stack. This helps to store the level of search associated with each predicate. It also helps to traverse back to the top levels of proof search tree.

A ‘Redo’ command in the trace indicates that the previous predicate failed and it requires a backtracking. If the current command being processed is ‘Redo’, then the top most item is popped from the stack and an edge is drawn between this popped item and currently executing predicate. After this, the popped item is again pushed into the stack. An item (a predicate) will remain in the stack until it is proved successfully or failed to prove.

If the line starts with the command ‘Exit’, it means the currently executing predicate is successfully proved. So a vertex with the name ‘Success’ is plotted and an edge is added between this vertex and the previously added vertex. After every line with ‘Exit’ command, one item is popped from the stack.

If the line starts with the command ‘Fail’, it means the currently executing predicate can’t be proved correctly. So a vertex with the name ‘Failed’ is plotted and an edge is added between this vertex and the previously added vertex. After every line with ‘Fail’ command, one item is popped from the stack.

Algorithm 1: Plotting graph from trace of the query

Input : file : text file containing trace of query execution

Output : Graph G : Graphical representation of trace

```
G ← DirectedGraph f ile ← trace f ile
previousP redicate ← null previousC ommand ← null previousLevel ← null
Stack s ← null poppedI tem ← null
foreach line l in the file do
String ArrayC [] ← set of column values in the line currentC ommand ← C [0]
currentLevel ← C [1]
currentP redicate ← C [2]
if currentC ommand = “C all” then if previousC ommand
! = “C all”&previousC ommand
! = “Exit”&previousC ommand
! = “Redo”&previousC ommand! = “F ail”
then
Add vertex currentP redicate in Graph G else if previousC ommand = “C all” OR previous Command = “Redo” then
Add Edge
(previousP redicate, currentP redicate) in
Graph G
else if previousC ommand = “Exit” OR
previousC ommand = “F ail” then
poppedI tem ← popped value froms
Add Edge (poppedI tem, currentP redicate)
in Graph G
Push poppedI tem into s push currentP redicate into s SetPreviousValues()
end
if currentC ommand = “Exit” OR
currentC ommand = “F ail” then
```

```
if previousC ommand ! = “Exit” then
Add Edge (previous Predicate, “Success”)
in Graph G
if previous Command ! = “Fail” then
Add Edge (previous Predicate, “F ailed”) in
Graph G
Pop item from Stacks
Set PreviousValues()
end
if current C ommand = “Redo” then
if curret Predicate contains a variable then
Add Edge (popped Item, current Predicate)
in Graph G
Push poppedI tem into s
else
SetPreviousValues()
end end
Function Set PreviousValues ()
previousP redicate ← current Predicate previous Command ! current Command
previous Level ← current Level
```

C. Unification of variables

Unification of variables is an essential step in solving Prolog queries. If there is a variable in the posed query, Prolog will generate another unique variable and replace the variable in the query with the new variable. This new variable name starts with an underscore. Variable unification algorithm is called when there is a variable name in the posed query. A variable name can be found in either ‘Call’ or ‘Redo’ commands.

Algorithm 2: Unification of variable

```
Stack varStack ← null previousP redicate ← null previousC ommand ← null foreach line l in the trace file do
String ArrayC [] ← set of column values in the line currentC ommand ← C [0]
currentLevel ← C [1]
currentP redicate ← C [2]
if currentCommand = “C all” OR
currentCommand = “Redo” then
if there is a variable, name starts with underscore,
in currently processing predicate then Add currentP redicate with its previousP redicate into the stack, varStack
end
if currentCommand = “Exit” OR
currentCommand = “F ail” then
if The value for a variable name is resolved
then
```

```
→ Pop an item from varStack
→ Change the variable name in the
popped item with the resolved value
→ Add an edge between
previousP redicate and
currentP redicate which are extracted from the popped item
end end
end end
```

5. EXPERIM ENTS AND RESULTS

The graph plotting tool is developed in JAVA Programming platform. SWI-Prolog is used in the background for execution of queries. The tool let the user to select a knowledge base. Knowledge base is a file with '*.pl' extension, and it contains Prolog clauses. User can view the content in the selected knowledge base and pose a query. The query trace and the generated graph will be displayed to the user.

A process of SWI-Prolog is invoked using java and the selected knowledge base is loaded into the process. The Prolog command to load the file is sent through the input stream of the process. Output and Error streams of the process are redirected to a file. When the user enters a query, the query is modified to obtain the trace. This modified query is passed into the input stream of the process. The query is modified as follows, leash(-all),trace,userquery.

The command, leash is a predicate used in SWIPL to obtain ports which allows for user interaction. The special shorthand -al refers to all ports. This helps to obtain all trace lines in one go.

JUNG API is used for plotting the graph. JUNG stands for Java Universal Network/Graph framework. Jung is written in java and it is a rich set of open source library that provide common language for visualization and modeling. The tool is tested with different knowledge base and queries, which includes simple Prolog clauses, recursive clauses and Lists. Some of the tested programs are given below.

A. Program with simple clauses

The following knowledge base has simple clauses.

```
wealthy(ram). wealthy(krishna). healthy(ram).
happy(X) : - wealthy(X),healthy(X).  /*X is happy if it is wealthy and healthy*/
```

Consider the following query,

happy(krishna).

Prolog will try to prove the query using the definition clauses of happy. The new goal will be wealthy(krishna),healthy(krishna). The first part of the goal is succeeded as there exist a fact wealthy(krishna). But the second part is failed, as there is no facts, 'healthy(krishna). As a result the original query is also failed. Obtained trace of the posed query:

```
Call: (8) happy(krishna) ? creep
Call: (9) wealthy(krishna) ? creep
Exit: (9) wealthy(krishna) ? creep
Call: (9) healthy(krishna) ? creep
Fail:  (9) healthy(krishna) ? creep
Fail:  (8) happy(krishna) ? creep
```

The search graph generated will be as shown in [fig.6].

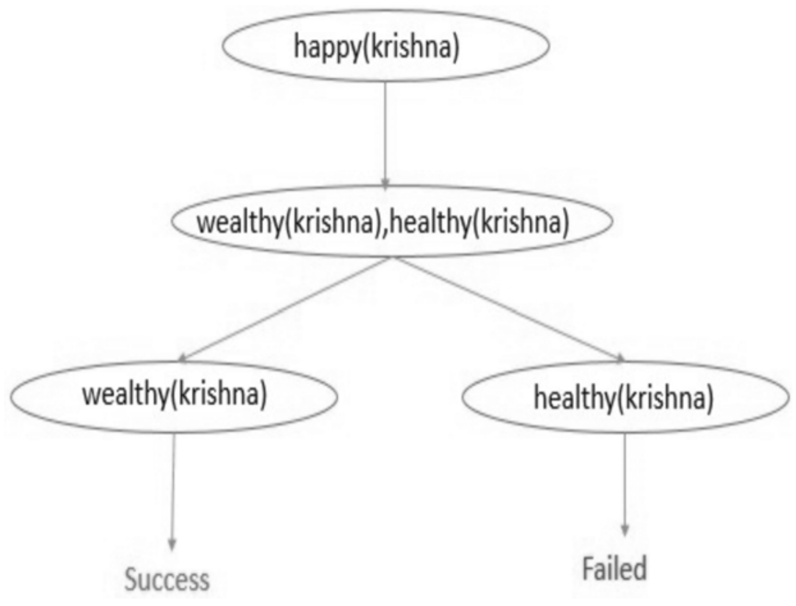


Fig. 6. Search graph of the query, ’happy(krishna)’

B. Program with recursion

Consider the following knowledge base which contains recursive rules,

parent(ram,lal). /* lal is ram’s parent */ parent(lal,krishna). /* krishna is lal’s parent */ parent(krishna,seetha). /* seetha is krishna’s parent */ ancestor(X,Y):- parent(X,Y).

ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).

In this program, the second rule of ancestor is recursively defined. Suppose the following query is posed, ancestor(ram,krishna).

Prolog will try to prove the query using first clause of ancestor, which result in the new goal, parent(ram,krishna). It is failed as there is no such fact exist in knowledge base saying ’krishna’ is rams parent.

Then the second clause of ancestor will be considered for proving. This result in a new goal, parent(ram,Z), ancestor(Z,krishna)

Then it will try to prove first part of the new goal which is, parent(ram,Z). This result in instantiating Z with lal, as first fact in the knowledge base says so. Now Prolog will try to unify second part of the goal which is, ancestor(lal,krishna). For proving this, it will go to the first defined clause of ancestor, which is ’ancestor(X,Y) :- parent(X,Y)’. Then the goal clause will be, ’parent(lal,krishna)’. It is also successfully proved, as the fact parent(lal,krishna) exists in the knowledge base. Both part of the goal clause is succeeded and this results in success of original query. Trace obtained for the posed query is,

Call: (7) ancestor(ram, krishna) ? creep Call: (8) parent(ram, krishna) ? creep

Fail: (8) parent(ram, krishna) ? creep Redo: (7) ancestor(ram, krishna) ? creep

Call: (8) parent(ram, _ G1665) ? creep Exit: (8) parent(ram, lal) ? creep

Call: (8) ancestor(lal, krishna) ? creep Call: (9) parent(lal, krishna) ? creep

Exit: (9) parent(lal, krishna) ? creep Exit: (8) ancestor(lal, krishna) ? creep

Exit: (7) ancestor(ram, krishna) ? creep true.

The obtained proof search graph for the posed query is shown in the figure[fig.7].

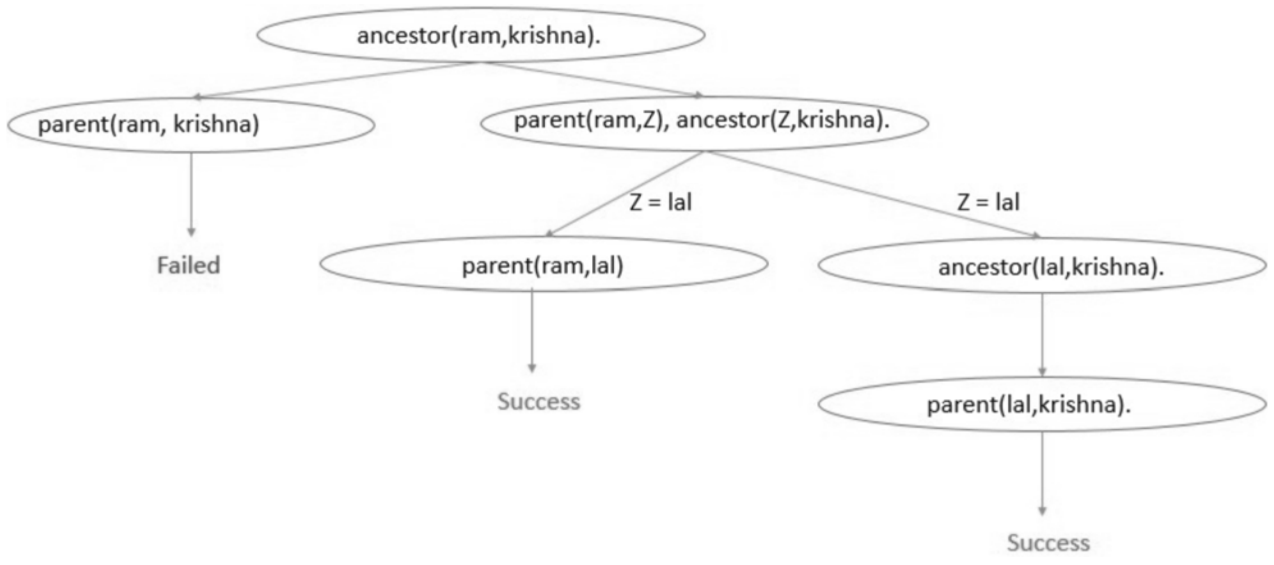


Fig. 7. Search graph of the query, 'ancestor(ram, krishna)'

6. CONCLUSION

Learning Prolog programming language needs better understanding of how the Prolog interpreter finds answer for a given query. Understanding how Prolog unifies and backtracks while searching for answer is as important as learning Prolog language itself. Tracing the execution of a query gives an idea about how the answer has been obtained. Prolog gives trace of a query in textual form, which is hard to understand in a learner's view. Representing the trace graphically gives a much easier way to understand unification and backtracking, that is carried out while resolving a query.

The graphical representation generated by the tool has a tree structure. The vertices represent the goals that have to besatisfied, and the edges show the variable instantiation done when a match occurs. The leaf nodes contain either the label Success or Failed depending on whether the goal was satisfied or not. This helps a learner to understand how Prolog find out answer for the posed query. In future the tool could be enhanced with a module that generates a graphical model for the first-order logic (which the user used to make his knowledge base), and compare it with generated graph of the whole knowledge base. This helps to make sure if the first-order logic is correctly converted to Prolog statements.

7. ACKNOWLEDGEM ENT

We thank department of CSA, Amrita School of Engineering, Amritapuri for infrastructure support. We thank Mrs. Manjusha Nair, Mrs. Thushara M.G, Mrs. Geethalekshmi V and Mr. Jayakumar P for the valuable comments and support they have given.

8. REFERENCES

1. Helen Pain, Alan Bundy, What Stories Should We Tell novive PROLOG progarammers?, Working Paper 156, Department of Artificial Intelli- gence, University of Edinburgh 1984.

2. E. Fogel, Teaching Prolog using intelligent computer-assisted instruction and a graphical trace, T, University of British Columbia, 2010.

3. Torbjrn Lager and Jan Wielemaker, Pengines: Web Logic Programming Made Easy, ICLP, 2014.

4. Jan Wielemaker&Nicos Angelopoulos, Syntactic integration of external languages in Prolog, Budapest, Hungary, 2012.

5. Jan Wielemaker&Vitor Santos Costa, Portability of Prolog programs: theory and case studies,, CICLOPS, 2010.

6. Michael A. Covington, Rob-erto Bagnara, Richard A. O'Keefe, Jan Wielemaker, Si-mon Price, Coding Guidelines for Prolog, TPLP, 2012.

7. Jan Wielemaker, An Overview of the SWIProlog Programming Environ- ment, Katholieke Universiteit Leuven, 2003

8. Ulle Endriss, An Introduction to Prolog Programming, Institute for Logic language and computation, University of Amsterdam, 2015

9. Borger E, Rosenzwerg D, Prolog Tree Algebras A formal Specification of Prolog, Proceedings of the Third International Conference on Information Technology Interfaces, SRCE, Zagreb 1991, pp.513-518

10. Sehrish A qeel, SPSS:An Effective Tool to Compute Learning Outcomes in Academics, International Journal of Computing Algorithm, Vol.5, Issue:01 June 2016

11. Wafa Chama, Raida Elmansouri, and Allaoua Chaoui, Model Checking and Code Geberation for UML diagrams using graph transformation, International Journal of Software Engineering& Applications (IJSEA), Vol.3, N0.6, November 2012