PROLOG WITH BEST FIRST SEARCH

Benjie Lu, Zhingqing Liu

Computer Go Research Institute, School of Software Engineering,
Beijing University of Posts and Telecommunications, Beijing 100876, China
lpbjdf@gmail.com, zhiqing.liu@gmail.com

Abstract.

Prolog is a logic programming language with an engine that can search a knowledge base with respect to a query automatically. To satisfy a query of a knowledge base, the Prolog engine always try to use the top-most clause and search the left-most sub query, which is fundamentally a depth first search (DFS). While generally usable, DFS is less effective in many circumstances, and may even fail to discover solutions in some extreme cases. This paper presents an alternative best first search (BFS) for the Prolog engine. Our BFD search strategy employes a heuristics function based on an extended measurement of UCB1, which is a machine learning algorithm for a Markovian Decision Process (MDP). This heuristics function can balance exploitation and exploration during the search process of the Prolog engine, adjust its search path dynamically for the best chance to satisfy its query based on its accumulated knowledge of the knowledge base. Compared with DFS Prolog, BFD Prolog not only improves its search efficiency, but also guarantees theoretically to discover its solutions if they exist. Unlike DFS Prolog, BFS Prolog is completely declarative as knowledge orders are irrelevant.

Keywords:

Best-First Search, heuristic function, algorithm convergence, UCB, and-or tree.

1. INTRODUCTION

Prolog (Programming in Logic) is a logic programming language[1,2,7]. It is based on the foundations of Logics, and was originally designed for natural language processing. Prolog is now widely used in search of artificial intelligence, for construction of expert systems, natural language processing systems, and knowledge systems, etc. Unlike commonly found iterative programming languages, Prolog is declarative. A Prolog program describes internal logical structures of intelligent system declaratively. Declarations in a Prolog program consist of facts and rules, as such a Prolog program is also referred to as a "knowledge base". This helps to describe fundamental properties of an intelligent system precisely. Execution of a Prolog program is conducted by an interpretive engine internal to Prolog. More specifically, the execution is conducted by matching the query with its knowledge base repetitively, to which is commonly referred as unification. The unification process is to prove or disprove the query with the facts and rules in the knowledge base. The Prolog engine employes two strategies in the unification process [2,6]:

(1) Selection of a clause according to the original order of the knowledge base; and

(2) Selection of the left-most goal to be unified.

These two strategies, in effect, make the Prolog execution engine to conduct a depth first search to traverse the knowledge base, so as to provide a capability of intelligent interactions with its user.

Depth first search in Prolog execution is biased with respect to questions. For certain questions, if their solutions are located in a first portion of the search tree, these solutions can be found effectively. However, for certain other questions, if their solutions are located in a last portion of the search tree, these solutions, if ever found, must take a long route on the search tree. More severely, depth first search is not complete, i.e., the Prolog engine may never find a solution even if it exists, caused by searching an endless branch of the search tree. Similarly, the Prolog execution may also be trapped into a local search that it fails to disprove a non-existing solution.

It is therefore very appealing to conduct a best first search in Prolog execution. How ever, it is a great challenge to develop a general heuristic evaluation unction that is suitable for all possible intelligent systems

978-1-4673-5532-2

that can be dramatically different. We propose to use machine learning algorithms to generate a suitable heuristic evaluation function automatically and adaptively. The UCT algorithm is such an algorithm, which has demonstrated important applications in many areas, including computer Go[10,11,12], a grand challenge in artificial intelligence. The UCT algorithm is based on UCB algorithms[8], which is designed for the machine learning multi-armed bandit problem. The UCB algorithms guarantee that their regret, defined as their difference with the optimal algorithm, is no greater than log(n) if each selection is conducted by a heuristic function of $x_i + \sqrt{\log(n)/n_i}$, where n is the total number of selections, ni is the number of selections on each arm, and xi is average reward of each arm. The UCB algorithms help to balance the trade-off between exploitation and exploration in machine learning, such that accumulated experiences are properly used while unknown areas are also explored at the same time. The UCT algorithm applies UCB in each selection in a search[9]. Recently, researches have explored possibilities to replace depth first search in Prolog execution with best first search. In best first search, the traditional two strategies are replaced by a selection procedure based on UCB. In other words, best first search selects a clause with the highest UCB specification, and selects a goal with the highest UCB specification. This leads to significant changes to Prolog execution. The Prolog execution with best first search is no longer deterministic. Instead, it selects the best search path based on experiences and possibilities, finding suitable clauses and goals for unification intelligent, improving efficiency of the Prolog execution. It can also adjust its execution order with respect to specific question and queries. This adaption in Prolog execution is crucial for construction effective expert systems and knowledge bases. This will also provide a wider area of applications to Prolog. The improvement discussed in literature [13] is fundamentally a local best first search strategy. It has limitations as it cannot prevent Prolog execution trapped into local loops so as to fail to complete in tree search. This paper provides a further improvement to this issue. By an extension to UCB, this improvement introduces a global best search strategy, such that best unification can be selected globally. This provides additional flexibility in Prolog execution, making it totally independent from the order in which Prolog facts and rules are specified. As such, Prolog, with best first search execution, becomes a completely descriptive programming language.

The rest of the paper is organized as follows: Section 2 discusses traditional execution model of Prolog based on depth search first, with two examples for which depth first search Prolog fails to solve. Section 3 presents out new execution model of Prolog based on best first search. An example is discussed in Section 4 with respect to the new model. This paper is concluded in Section 5 with summary remarks.

2. PROLOG EXECUTION MODEL

Predicates are the basic unit in Prolog programs. A predicate is also referred to as a term which can be either a primitive term such as a constant or a variable, or a complex term consisting of a functor with 0 or more parameters. A parameter is also a term. The definition of a term is obviously recursive.

A term in Prolog describes a fact. A rule in Prolog describes relationships among terms. A rule consists of a head and a body, in which the head is a term, and the body may consist of one or more terms. A term in the body of a rule is also referred to as a goal. The head of a rule is proved is all terms of its body are proved. Rules can be used to define new logical relationships with existing ones. A fact can be viewed as a rule without its body. As such, a rule and a fact are both referred to as a clause. The set of clauses with the same head name is a procedure. Generally speaking, relationships among goals within the body of a clause are conjunctive. Occasionally they may be alternative, or combinations of the two. The relationships among clauses in a procedure are alternative.

A Prolog program consists of a collection of procedures. A Prolog program is also referred to as a knowledge base, which can find a solution with respect to a query. A query is a term. If the query is a basic term, it is regarded as whether the term is derivable from the knowledge base. If the query is a complex term, it is regarded as whether its variables have instantiations such that instantiated term is derivable from the knowledge base.

The process of Prolog execution is a process if

unification of query against the knowledge base. In this unification process, the selection of clauses is based on the order of the clauses described in the knowledge base. A successful unification may produce new sub-goals, which are further unified based on the order in which the sub-goals are specified. In effect, the Prolog unification process is a depth first search.

We now consider two examples. The first one is a Prolog program to compute factorial recursively, in which its correct execution is very sensitive to the order in which sub-goals are specified in the rule's body:

```
1.factorial(N,F):- N>0,N1:=N-1,
  factorial(N1,F1),F:=N*F1.
2. factorial(0,1).
```

If we change sub-goals in a different order in the following program, it will fail to produce an answer as it is trapped into a left recursion.

```
1.  factorial(N,F):- factorial(N1,F1),
    N>0,N1:=N-1,F:=N*F1.
2. factorial(0,1).
```

The above program fails on a simple query factorial(1; X): We see that Prolog programs are very sensitive to the order in which clauses are specified due to its depth first search. Now we consider our second program

```
1. reverse([],[]).
2. reverse([X|Xs], Zs):- reverse(Xs,Ys),
   append(Ys,[X],Zs).
3. append([ ],Ys,Ys).
4. append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

in which the append procedure appends two lists together, while the reverse procedure reverses a list. Both of the two procedures have a main logical rule and a fact handling border cases, and straightforward and precise. However, the program is very sensitive to the way in which a query is presented, due to the use of depth first search. For example, for query reverse([a]; X):, it can answer correctly with a result X = [a]; and also correctly with a NULL result if requested for additional answers. However, for query reverse(X; [a]), it can answer correctly with a result X = [a], but fails to terminate when requested for additional answers. (The above cases will terminate with stack oveflow in Amzi Prolog and with a handled exception in SCIStus Prolog.[3,4,5])

If we modify the above program as follows:

```
1. reverse([],[]).
2.reverse([X|Xs], Zs):-append(Ys,[X],Zs),reverse(Xs,Ys).
3. append([ ],Ys,Ys).
4. append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

in which we alter the order of the sub-goals in the body of reverse, the execution results will change accordingly: It can answer the query reverse(X; [a]) correctly, but fails on the query reverse([a]; X).

A fundamental goal of Prolog is descriptions of facts and rules, instead of using imperative procedures as in the C programming language. Descriptive specifications are better for expose internal characteristics of intelligent or expert systems. However, descriptiveness of Prolog, as shown in the above examples, is rather limited. Two queries reverse(X; [a]) and reverse([a]; X): are essentially the same.

```
1.  factorial(N,F):- N>0,N1:=N-1,
    factorial(N1,F1),F:=N*F1.
2.  factorial(0,1).
```

and

```
1.  factorial(N,F):- factorial(N1,F1),
    N>0,N1:=N-1,F:=N*F1.
2.  factorial(0,1).
```

are of no difference. But they behave rather completely in depth first search Prolog, because the order in which clauses are specified and the order in which query terms are presented have a tremendous impact of Prolog execution. As such, the Prolog engine may not provide an answer even if it exists; the Prolog engine may not disprove a query if no answer exists. The above problems undermine the descriptiveness of Prolog, as the Prolog engine requires orders to specify its logic structures. This is rather imperative. This combination of imperativeness and descriptiveness is one of the key factors that limit the impact of the language. We aim to provide uniform results between semantics and search. In other words, everything derivable from the knowledge base can be proved; while everything that is not derivable from the knowledge base can be disproved. This is not achievable with depth first search in Prolog execution.

3. BEST FIRST SEARCH PROLOG HEURISTIC FUNCTION AND EXECUTION MODEL

To answer a query in Prolog, its root procedure is called, which is the procedure whose head unify with the query. In execution of the root procedure, it may call

other procedure, sometimes recursively. At each level of procedure invocation, procedures may be called conjunctively or alternatively. For convenience, we shall refer to them as and-call and or-call respectively in following discussions. If the original procedure call is successful, then one of each or-call must be successful and all of each and-call must also be successful. If the original procedure call fails, then one of each and-call must fail and all of each or-call must also fail. The procedure calls as such will form an and-or tree. The structure of the tree is determined by both the structure of the program and the order in which procedure calls are performed. An and-node in the tree succeeds if all of its children succeed; an or-node in the tree succeeds if one of the children succeeds. The root of the and-no tree is an or-node, so are leaf nodes of the tree. The traditional Prolog engine always traverse the and-or tree in a post order, visiting its child nodes from left to right recursively. An alternative approach for the tree traversal was discussed in [13]: For an or-node, its child node with the highest possibility of success is always selected for visiting; for an and-node, its child node with the lowest possibility of success is always selected for visiting. This is the most efficient way for arranging the visiting order of the child nodes, because a success in a child of an or-node eliminates the need to visit the rest of all unvisited children; similarly, a failure in a child of an and-node also eliminates the need to visit the rest of all unvisited children. We call this tree traversal strategy "locally best first search", which always find the most suitable child for subsequent visit. It will improve efficiency of Prolog execution.

However, it does not fully address the completeness problem presented above. In order to provide completeness for Prolog execution, the Prolog engine sometimes must backtrack to an early stage to continue search on a different portion of the search tree. While backtracking on the search tree, unification results must also be backtracked as well. Otherwise the unification results may have changed, resulting in an inconsistency in Prolog execution state. We shall refer to this new strategy "globally best first search", in which subsequent visit shall not be limited to child nodes of the current node, instead all incomplete nodes shall be included as candidates. Some early literatures have discussed Prolog execution with best first search[14]. However these discussions were only at a conceptual level, due to lack of a suitable heuristic function to select the best visiting choice. We now describe such a suitable heuristic function, by extending the UCB measurement as follows:

$$1/L + x_i + \sqrt{\log(n)/n_i}$$

where L is the level of the node, with the level of the root to be 1; n is the number of visiting times for the parent node; ni is the number of visiting times for the current node; xi is an average indicator of its visiting results, with different meanings for and-nodes and or-nodes. For an or-node, xi is its average success rate, while for an and-node, xi is its average failure rate. A higher UCB value indicate a more likely candidate for visiting, either because it is close to the root, with a good historical results (likely to succeed for an or-node or to fail for an and-node), its visiting number is small relative to total visit of its siblings, or a combination of the above three.

With the above UCB measurement, best first search Prolog execution algorithm is as follows:

```
Input: Source Code and Query
Open := {Query}
Closed := {}
Node := Query
while Open != {}    loop
      A := N in Open with the highest UCB
      if exist A's sibling B in Closed then
            backtrack to B
            A := B
      end if
      if A can be proved or disproved then
            update each ancestor of A until Root is reached
      if Root is proved to disproved then
            return true or false accordingly
      end if
else
      Closed := Closed + {Node}
      Node := A
end if
end loop
```

There are two tasks in value update toward the root: 1) Recursively update parent node based on its logic until the root is reached; 2) Recursively update the number of visiting times and the number of successful visiting times.

4. AN EXAMPLE

We have shown that the above factorial example cannot be fully solved by either depth first search or locally best first search. As shown in the figure1, the engine will always visit the first clause that will not end without a call to the second.

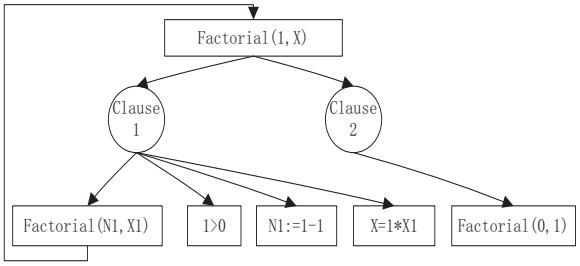

Figure 1. Recursive search in DFS

We now consider the case in global best first search, its tree traversal order is shown in figure 2 and the UCB value of each node in the process of search are listed in table 1.

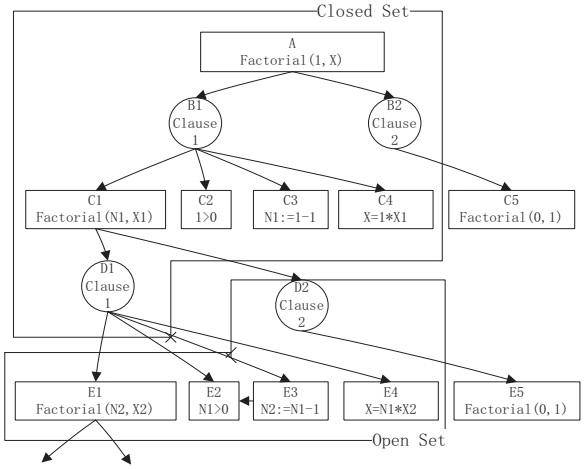

Figure2 Search Process in Global BFS

	1	2	3	4	5	6	7	8	9
A	1								
B1		1/2							
B2		1/2	1/2						
C1			1/3	1/3					
C2			1/3	1/3	1/3				
C3			1/3	1/3	1/3	1/3			
C4			1/3	1/3	1/3	1/3	1/3		
D1					1/4	1/4	1/4	1/4	
D2					1/4	1/4	1/4	1/4	1/4
E1									1/5+sqrt(log7/3)
E2									1/5+sqrt(log4/3)
E3									1/5+sqrt(log4/3)
E4									1/5+sqrt(log4/3)
E5									
Closed set		A	A B1	A B1	A B1 B2	A B1 B2 C1	A B1 B2 C1 C2	A B1 B2 C1 C2 C3	A B1 B2 C1 C2 C3 C4	A B1 B2 C1 C2 C3 C4 D1
Open set	A	B1, B2	B2 C1 C2 C3 C4	C1 C2 C3 C4	C2 C3 C4 D1 D2	C3 C4 D1 D2	C4 D1 D2	D1 D2	D2 E1 E2 E3 E4

Table 1. UCB values of the nodes during search

Experiment results show that this can be solved by global best first search with UCB measurement. It is worthy to note that the level of backtracking is sensitive to the UCB measurement, and our choice is not the only possible one. There may be other suitable functions suitable for this purpose. In global best first search, our UCB measurement can always backtrack to a suitable node for visiting in order to avoid a trap. Therefore our new proposed Prolog engine can always finish execution and be consistent and complete with respect to the semantics of Prolog programs.

5. CONCLUSIONS

We in this paper have improved the locally best first search Prolog execution strategy discussed in [13]. We have extended the traditional UCB measurement such that it is suitable as a heuristic for Prolog execution. This facilitates globally best first search in Prolog execution. Our experiment results indicate that the new strategy helps Prolog for find more suitable and efficient nodes for visiting. Additionally, it can also address the issue that existing Prolog execution may fail to terminate in certain cases. With this strategy, orders in Prolog programs and queries become irrelevant, making it a truly descriptive logic programming language.

REFERENCES

1) Lloyd, J. W. (1984). *Foundations of logic programming*. Berlin: Springer-Verlag. ISBN 3-540-13299-6.

2) Shapiro, Ehud Y.; Sterling, Leon (1994). *The art of Prolog: advanced programming techniques*. Cambridge, Mass: MIT Press. ISBN 0-262-19338-8.

3) http://www.sics.se/isl/sicstuswww/site/index.html

4) http://www.amzi.com/

5) http://www.swi-prolog.org/pldoc/doc/swi/library/edinburgh.pl

6) Kowalski, R. A. (1988). "The early years of logic programming". *Communications of the ACM* **31**: 38.

7) Colmerauer, A.; Roussel, A. (1993). "The birth of Prolog". *ACM SIGPLAN Notices* **28**: 37.

8) http://www.worldlingo.com/ma/enwiki/zh_cn/Fifth_generation_computer

9) http://pauillac.inria.fr/~deransar/prolog/docs.html

10) Gelly, S., Wang, Y., Munos, R., and Teytaud, O.Modification of UCT with patterns in Monte-Carlo Go.Technical Report 6062, INRIA, France, 2006.

11) Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002a). Finite time analysis of the multiarmed bandit problem. Machine Learning, 47, 235-256.

12) L Kocsis and Cs Szepesvari. Bandit Based Monte-Carlo Planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

13) Benjie Lu,Zhingqing Liu ,Hui Gao An Adaptive Prolog Programing Language With Maching Learing Proceedings of IEEE CCIS 2012

14) George F.Luger Artificial Intelligence Structures and Strategies for Complex Problem Solving Forth Edition ISBN 0-201-64866-0 2002 Pearson Education Limited