数理解析研究所講究録
第482巻 1983年 145-155

An Algorithm for Intelligent Backtracking

Taisuke Sato

Electrotechnical Laboratory

1. INTRODUCTION

The purpose of this paper is to propose a method to improve the backtracking behavior of Prolog. This method was derived from the one invented for OL(ordered linear) refutation. Therefore it takes into consideration failures by 'occur check' as well as failures by 'clash(different constant symbols)' unlike other intelligent backtracking methods [1][3][4]. Moreover we can be sure that it never destroys the completeness of proof search.

When we apply our backtracking method not to a prover like OL but to PROLOG, we have to treat the failures containning 'not' or 'cut'. Modifications needed for backtracking in case of those failures are given in the last section.

Currently, Prolog searches for an answer in a top-down and serial manner. If every choice has completely failed, Prolog starts backtracking to the most recent step where untried choices are left and selects an alternative to restart a proof search process.

The backtracking process described above simply goes back over the path in reverse order without analysing the cause of failure. Hence, it is called Naive Backtracking (NB for short).

However, NB is rather inefficient. Let us take a simple example(see fig. 1). For AND-goal A(x,y)B(x)C(y), A(x,y) is called first. A(z,a)<- unifies with A(x,y) and the resulting mgu(substitution, variable bindings) is {x\z, y\a}. Then the second goal B(x) is called. The actual goal is B(z) because the value of x is z. This goal immediately succeeds by unification with B(a)<-. mgu {z\a} is produced. The total substitution obtained up to this step is {x\z, y\a}*{z\a}(* denotes substitution composition). Thirdly C(y) is called. The actual goal is C(a). But since there is no input clause whose head is unifiable with C(a), backtrack occurs. According to NB, we return to the most recent step where B(a)<- was selected and retry the input clause B(b)<- as an alternative.

```
<-A(x,y),B(x),C(y) ...AND-goal

    A(z,a)<-
    A(z,b)<-
    B(a)<-
    B(b)<-    x,y,z are variables.
    C(b)<-    a,b are constants.
```

fig. 1 A simple example of a program

This selection is destined to fail. The reason is that since the cause of the backtrack -- the value <a> of the variable <y> -- is not eliminated, we will again reach the step of unifying C(a) with C(b)<- sooner or later. If we want to avoid the double occurences of the same failure such as unifying C(a) whith C(b)<-, we must retry the step where the vaiable <y> is bound to the constant <a>, i.e., where the goal A(x,y) is unfied with A(z,b)<-.

Based on these observations, several intelligent backtracking methods (IB for short) have been proposed which analyze the cause of failures and decide what to do at the next step to avoid repeating the failure.

IB generally does not return step by step but skips over several steps at a time. In other words it cuts off OR branches of the proof search tree. Therefore the problem of IB is to insure that proof paths skipped over actually do not lead to a solution. If proof paths are cut off carelessly, we may lose the chance to find a solution.

Let us call such a backtracking method safe that cuts off only those paths that never lead to a solution. There are several literatures on IB[1][3][4], but they concentrate on the efficiency of IB, and have little consideration for safeness of their backtraking methods.

We pursue a backtracking method which is both intelligent and safe. We first define the proof search tree (search space) associated with the given Prolog program and goal clause.

2. PROOF SEARCH SPACE

In order to define IB without ambiguity, we need a description of the behavior of Prolog foru the given program (Horn set, without 'cut' and 'not' for the time) and goal clause.

The computation process of Prolog can be seen as a transition of a AND-goal. Every AND-goal in the process is represented by a pair, the template(skelton) of the goal and the substitution.

In what follows, L,M,N... represent goals(literals). α, β, γ ... Represent sequences of several goals, i.e., AND-goals. λ, θ, μ ... Represent substitutions. E λ Represents the application of λ To a expression E.

If the initial AND-goal is α, Then it is represented as $\langle \alpha, e \rangle$(e denotes the null substitution). Suppose that the AND-goal becomes $\langle L\alpha, \theta_1 * \ldots * \theta_{i-1} \rangle$ after i-1 resolution steps. The actual AND-goal is $L\alpha\theta_1 * \ldots * \theta_{i-1}$ and the next goal(conjunct) to solve is $L\theta_1 * \ldots * \theta_{i-1}$ because goals are solved in left-to-right order. Also suppose that an input clause $M \leftarrow \beta$ is selected and renamed whose head is unifiable with $L\theta_1 * \ldots * \theta_{i-1}$. Let the mgu(most general unifier) of M and $L\theta_1 * \ldots * \theta_{i-1}$ be θ_i. After the i-th resolution step, the AND-goal becomes $\langle \beta;\alpha, \theta_1 * \ldots * \theta_i \rangle$. $\theta_1, \ldots, \theta_I$ are called substitution factors. If the template part of the AND-goal becomes empty, i.e., $\langle \cdot, \theta \rangle$, the computation halts and the proof successes. Then the answer substitution is θ .

The proof process described so far corresponds to tracing some branch of a proof search tree(search space, see fig. 2). Each OR node si represents an AND-goal and is labeled by the template of the goal(conjunct) at si. The directed edge ei indicates the literal pair, the goal and head of an input clause to resolve upon. The resulting mgu is represented by θ_i. The number of edges show the possibilities of resolution. A proof search tree like fig. 2 is easily obtained from the connection graph of the Prolog program and initial AND-goal.

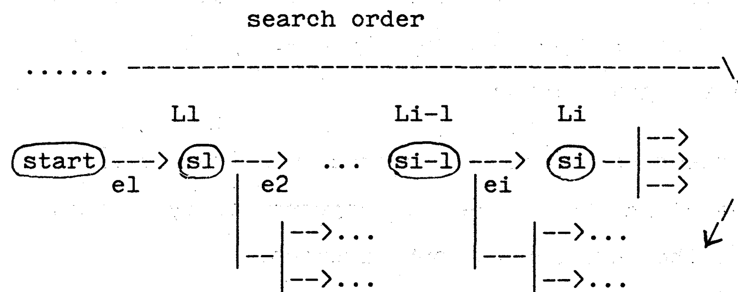


fig. 2 Proof search tree

We trace such a proof search tree in a left-to-right, up-down order. Resolutions(unifications) are performed when we go though edges. If the unification failed, we label the edge by an asterisk. A proof terminates when we reach the node where the AND-goal is empty. It is labeled by □ (null clause) and called an end node. The total path from the starting node to the end node is called a solution path or simply a solution.

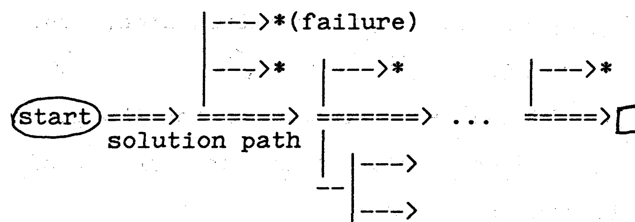


fig. 3 Solution path

If every path through a node s has failed, backtrack starts. we need some definitions before going into the details of the intelligent backtracking method.

3. DEFINITIONS

Def. 1 : < introducing point of a goal template >

For a goal template L to be resolved upon at some node s, there is an ancestor node s' of s where L is introduced to the goal template for the first time. We call s' the introducing point of L and represtnt s' by intro(L).

We notice two facts.

(F1) For every solution path through the introducing point of L, there

must be a node labeled by L. This is obvious because any goal(template) of the AND-goal must be resolved upon until it succeeds.

(F2) Let t,t'... be descendent nodes of intro(L) labeled by L. The partial proof search trees with top node t,t'... are the same since a search tree depends on only the template of the AND-goal. Suppose the AND-goal at intro(L) is $\langle \beta ;L; \alpha , \theta 1* \ldots * \theta i\rangle$. Then AND-goals have the form $\langle L \alpha , \theta 1* \ldots * \theta i* \mu\rangle$ at t, t'... .(μ is the composition of several substitution factors and depends on t, t'...)

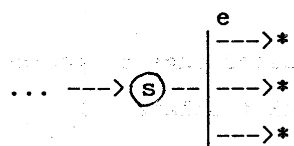
Def. 2 : < failure of edge >

If the resolution(unification) indicated by an edge e does not success, we say that the edge e failed directly. If the resolution succeeded temporarily and a subsuquent search found no solution through e, then we say that the edge e failed indirectly.

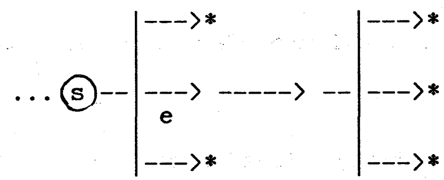
Def. 3 : < failure of node >

If all edges from a node s failed directly, we say that s failed directly. If there is at least one edge that indirectly failed and the other edges failed directly or indirectly, we say that s failed indirectly.

Note that the first backtraking is caused by the direct failure of some node. Fig 4 depicts the failures of a node.



(direct failure of a node s and an edge e)



(indirect failure of a node s and an edge e)

fig. 4 Failed node

---

4. INTELLIGENT BACKTRACKING AT THE DIRECT FAILURE

Since the first backtracking is due to the direct failure of a node, we treat the case of direct failure first.

When an edge e fails directly, the failure often depends on the subset of the previous unifications. To put it differently, the edge e was destined to fail at some previous step.

Def. 4 : < agent(e) of direct failure of an edge >

Suppose that an edge e incident with a node s failed directly. Let { $\lambda_1, \ldots, \lambda_k$ (k >=0)} be all of the substitution factors referred to at s by UA(unification algorithm, we assume Robinson's unification algorithm[2]) to perform the unification indicated by the edge e. { $\lambda_1, \ldots, \lambda_k$} contains the substitution factors referred to for 'occur check'. We call this set of substitution factors the agent of the direct failure of the edge e and represent it by agent(e).

Def. 5 : < occurence point of a prefix for a set of substitution factors >

For a set of substitution factors S = { $\lambda_1, \ldots, \lambda_k$ (K >= 0)} included in the substitution part of AND-goal at a node s, there is the farthest ancestor node s' where S is included in the substitution part <ν> of the AND-goal at s'. We call <ν> the prefix for S and s' the occurence point of S. If k = 0 then s' is the top node of the proof search tree. Or else the AND-goal of s' has the form <• , ... λ_k>.

Def. 6 : < det(e) of the directly failed edge e incident with a node s labeled by a goal template L >

det(e) is defined as the most recent node of s in the occurence point of agent(e) and intro(L). det(e) has the form <...L..., $\nu * \nu'$> where is the prefix for the agent(e).

At det(e) we can foresee the direct failure ofan edge e. The reason is as follows:

Suppose that the edge e indicates the unification of L with the head M of some input clause. Let agent(e) be { $\lambda_1, \ldots, \lambda_k$} and be the prefix for agent(e). The direct failure of e means that L $\lambda_1 * \ldots * \lambda_k$ is not unifiable with M. Accordingly Lν is not unifiable with M either.

If we follow a path from det(e) or its descendants, we inevitably pass through e again by (F1) and (F2). At that time

AND-goal must have the formm $\langle L\alpha, \nu * \nu' \rangle$. Although the edge e indicates the unification of $L\nu * \nu'$ with M, it necessarily fails because $L\nu$ is not unifiable with M.

Following form this, we can define det(s) for a directly failed node s where the failure of s is already inevitable.

Def. 7 : < det(s) for a failed node s >

Let {el,...en} be all nodes incident with s and suppose that every ei failed. det(s) is defined as the most recent node of {det(el) ... det(en)}.

There is no solution path through det(s) or its descendants. For any solution path must go through one of failed edges { el,...,en } by (Fl) and (F2). Therefore our intelligent backtracking method should skip over det(s) or its descendants.

If we return back to the parent node of det(s), at least one of the bindings that caused the failure of s is eliminated. Therefore,

Def. 8 : < btk(s) for a failed node s >

When a node s failed, the backtracking destination btk(s) is the parent node of det(s).

Def. 9 : < agent of a failed node s and det(s) >

Suppose that edges {el,...,en} incident with a node s failed. We define agent(s) and agent(det(s)), by

agent(s) = agent(det(s))
= {agent(el)... agent(en)}

Obviously det(s) is the most recent node of the occurence step of agent(s) and intro(L) of the goal template L at s. The next facts hold with respect to a failed node s and det(s).

(F3) Let { λ1,..., λk (k >= 0)} be the agent of a failed node s = $\langle \alpha, \nu \rangle$. If a node s' can be represented as $\langle \alpha, \nu' \rangle$ and ν' includes { λ1,..., λk}, there is no solution path through s'. Therefore no solution path contains det(s) or its descendants.

(F4) Suppose that a node s failed and det(s), btk(s) are defined respectively. If the substitution factor λ obtained by the edge from det(s) to btk(s) is included in agent(det(s)), λ the is most

recently obtained factor in agent(det(s)). The substitution part ν at det(s) is represented as $\nu = \nu' * \lambda$ for some substitution ν'.

```
 ... ------------------------------------\
                                          \
                                           |
                     search order         /
               <------------------------/
                          |===>*
                       |===|                  el
              |==>*    |  |===>*            |===>.
              |                       L     |    .
              |btk(s)  |_det(s)             |    .
(start) ... ==>(s'') ==>(s') ========>(s)====|    .
              |    λk  :     <Lα , ν'*ν''>  |    .
              |        :                    |en  .
              |        :                    |===>
              | |-->   :              L
              |-|      :
                |-->   :..........> t
                             <Lα , ν'*ν'''>
```

is a prefix for agent(s) = agent(s')
Dotted lines stand for OR branches skipped over by intelligent backtracking.

fig. 5 The relationship among directly failed node s, det(s), btk(s)

---

5. INTELLIGENT BACKTRACKING AT THE INDIRECT FAILURE

In this section we discuss indirect failures. We assume that F3 and F4 are true of every failed node previous to an indirectly failed node s. Note that F3 and F4 hold for the first failed node(direct failure node).

Def. 10 : < agent(e) and det(e) for a indirectly failed edge e >
Suppose that an edge e incident with a node s labeled by a goal template L failed indirectly. e connects s with its son s'. If we need to start backtracking at s, s' already failed and must be det(s'') for some previously failed node s'' and s is btk(s''). Since we are interested in the backtracking at s, we consider s and s' as such. Therefore agent(s') is already defined. F3, F4 are true of s'. Let θ be the substitution factor obtained by the unification indicated by the edge e.

(1) agent(s') = {} : In this case the cause of failure of s' is the existence of the AND-goal template of s'. We define,

agent(e) = {}

det(e) = intro(L).

(2) agent(s') =\= {} and θ is not included in agent(s') : θ Is not reponsible for the failure of s'. Therefore,

agent(e) = agent(s'),
det(e) = the most recent node in the occurence node of det(s') and intro(L).

(3) agent(s') =\= {} and θ is included in agent(s') : Let agent(s') be { λ 1,..., λ k} and λ k be the most recently produced substitution factor in agent(s'). θ is λ k since F4 holds for s'. Let {μ1,...,μ l (l >= 0)} be the substitution factors referred to by UA to produce λ k. {μ1,...,μl} excludes the substitution factors referred to by UA only for 'occur check'. This is because in order to produce λ k successfully, only {μ1,...,μl} is needed(details omitted).

agent(e) = { λ 1,..., λ k-1, μ1,...,μ l }
det(e) = the most recent node in the occurence node for agent(e) and intro(L).

We complete the definition of agent(e) and det(e) for an indirectly failed edge e. Every proof search process starting at det(e) or its decendant nodes must pass through the edge e if it succeeds. But the unification indicated by e always fails.(proof omitted. It is based on the assumption that F3, F4 is valid for every node that already failed directly or indirectly)

As for agent(e) and det(e) of the directly failed edge e, they are already given by def. 4 and def. 6. Thus agent(e) and det(e) are defined for every failed edge incident with an indirectly failed node s. det(s) and btk(s) for an indirectly failed node is given by def. 7 and def. 8 respectively. agent(s) and agent(det(s)) for an indirectly failed node s is given by def. 9. Note that F3 holds for an indirectly failed node s and det(s) and F4 for det(s) and btk(s) again. Thus based on the induction on the number of failures, we can be sure of the safeness of our backtracking method given by from def. 1 to def. 9.

6. MISCELLANEOUS MATTERS

In order to apply our intelligent bactracking method to a real situation, some modifications are needed.

a) false failures :

Our backtracking method is not applicable to failures other than the ones caused entirely by variable bindings. Consider a backtrack containning 'cut'. When such backtrack occurs , our intelligent backtracking method is not applicable since the information about variable bindings expected to be obtained from the skipped paths is lost. Similarly when a user forces a failure to get another answer, this failure is not ascribed to variable bindings. We call such failure a false failure. When we start backtracking at a failed node s and at least one of the failures of s is a false failure, we are compelled to adopt NB(details omitted).

b) 'not' : Safeness of our backtracking method is not insured with respect to failures containning a goal 'not(P)' because even if a goal not(P) occurred and P succeeded, the further instantiated P may fail. But if P in 'not(P)' is ground, this is not the case. Therefore, when 'not(P)' failed and P is ground, our method is applicable to this case.

c) prevention of the same failure :

Our backtracking method redoes the only one variable binding. Therefore if a failure of an edge e is due to multiple variable bindings, we are in danger of failing again at e. To avert such danger we have only to avoid passing through the edge e unless we return to the ancestor node of det(e).

d) implementation :

Records needed for intelligent backtraking are,

(1) intro(L) for a goal template L

(2) agent(e) for a failed edge

(3) the substitution factor(mgu) for a successful edge and the list of the substitution factors referred to by UA.

In order to implement our method, we add a step identification number to a variable cell when variable binding occurs. For example, if a variable <x> gets its value <a> at N-th step, the record is <x, N, a>. But since recording a step number in a variable cell consumes extra bits and the number of total steps of a proof is unpredictable, we will records the step number block by block. This means that we assign 1 to the first 5 steps and assign 2 to the next 5 steps and so on. Thus we can save bits for recording a step number to avoid "step number over flow".

Our backtrack method could be more intelligent if we built our backtracking theory based on 'substitution conpoment' instead of

'substitution factor'. MGU produced by Robinson's unification algorithm has the form {x1\t1}*...*{xn\tn}. Each {xi\ti} is called substitution component. Since failures depend not on a mgu as a whole but on some substitution components of the mgu, we can develop intelligent backtracking theory based on the dependencies of substitution components which is completely in parallel with our method presented here. The resulting backtracking method, however, will be impractical because of the expense of book keeping for dependencies of substitution components. Therefore we did not discuss such backtracking methods.

7. CONCLUSION

We have proposed an intelligent backtracking method based on a search proof tree obtained from the connection graph of a Prolog program. Although our method is assured to be safe, i.e., never overlooks a solution path, it only points out the step from which the retrial of an altenative search may succeed. It can not indicate a promising step to return. Therefore further refinement of our intelligent backtracking method should enable it to consider possiblities of success as well as safe.

ACKOWLEDGEMENT : The auther is grateful to Dr. Tanaka, Chief of Machine Inference Section of Electrotechnical Laboratory and other members of the section for helpful discussion.

REFERENCES :

[1] Bruynooghe, M. :"Analysis of Dependencies to Improve the Behavior Logic Prolog", 5th conf. on Automated Deduction, Lec. Note in Comp. Sci. Springer, 1980.

[2] Chang,C.L. and Lee,R.C.T. :"Symbolic Logic and Mechanical Theorem Proving", Academic Press, 1973.

[3] Pereira,L.M. and Porto,A. : "Selective Backtracking for Logic programs", 5th conf. on Automated Deduction, Lec. Note in Comp. Sci. Springer, 1980.

[4] Lasserre,C. and Gallaire,H. : "Controlling Backtrack in Horn Clauses Programming", ACM Logic Programming Workshop, Budapest, 1980.