An Incremental Theorem Prover

Murray Shanahan
University of Cambridge
Computer Laboratory,
Cambridge CB2 3QG.
England.

Abstract[1]

When states of affairs are represented by theories, reasoning about them often involves making a small change to a set of axioms, and computing the consequences of that change for the set of theorems derivable from those axioms. A Prolog-like theorem prover is described which, as it explores the space of possible proofs for a set of formulae, records the structure of that space. This information can be used to search efficiently for proofs for the same formulae with a slightly changed set of axioms. Existing Prolog interpreters throw away all such information, so that the entire space of possible proofs for each formula must be explored from scratch every time the set of axioms is changed, no matter how little that change affects the search space's structure.

Introduction

A robot has to reason about actual or possible states of affairs. The problem of propagating the effects of changes in a state of affairs through a representation of that state of affairs, updating only those parts which are affected by the change and leaving the rest alone, is one aspect of the *frame problem* (Hayes 1973, Dennett 1984). If a logic is used as the representation formalism, then information about a state of affairs will consist in a set of axioms and a set of inferred theorems. If a small change occurs in a state of affairs, bringing about a small change in the set of axioms, then it may be more economical to propagate the consequences of this change through the set of inferred theorems than to regenerate them from scratch. This paper is concerned to show how a Prolog-like theorem prover may be built to work in this way, where the set of axioms is a set of definite clauses and the set of inferred theorems is a set of goal clauses, each with a corresponding set of answer substitutions.

A Prolog interpreter (Kowalski 1979) is a top-down resolution Horn clause theorem prover. Presented with a goal clause Co and a set of definite clauses P, it searches for refutations of Go in the form of a sequence of goal clauses $Co, C_1 \ldots C_n$ where C_n is the empty clause and each C_{i+1}.i is obtained by resolving G_i with some clause in *P* whose head unifies with the leftmost literal in C,. Each such refutation generates a corresponding set of variable bindings, called an answer substitution. Since there may be many clauses in *P* whose heads will unify with the leftmost literal of any given C,, the theorem prover has to search a space of possible refutations, and for each refutation discovered it outputs the corresponding answer substitution. Most extant Prolog interpreters use backtracking to effect a depth-first search, choosing clauses from *P* in top to bottom order.

Having fully explored the search space, a non-incremental theorem prover throws away all record of how each substitution was computed - what sequences of resolutions were tried, which were successful and which were not. Additions and deletions of clauses are then straightforward database operations, but every time refutations have to be found for a goal clause, the search space has to be explored from scratch. Now, suppose that the use of the interpreter is characterised by the repeated presentation of the same set of goal clauses for a slightly changed set of definite clauses. The search spaces explored for each slightly modified set of definite clauses are then likely to overlap considerably. Under these circumstances it is economical to employ an *incremental* theorem prover which maintains *dependency structures* showing how each set of answer substitutions is obtained. Then, if a small change takes place in the set of definite clauses, the consequences of this change are propagated through the dependency structures to the set of answer substitutions. It is not necessary to regenerate the answer substitutions from scratch. The burden of computation is then shifted to the incremental modification of these dependency structures when the set of definite clauses is modified, reducing the search for answer substitutions to a simple lookup Clearly, for this to be a viable proposal, the resulting savings must outweigh the overheads of recording the dependency structures and propagating the consequences of change. In this paper I will outline the construction of an incremental top-down resolution Horn clause theorem prover, in which the maintenance of dependency structures incurs acceptable overheads.

The incremental approach is exploited in *reason maintenance systems* (Doyle 1979, deKleer 1986), which maintain a record of the logical dependencies within a set of propositions, and ensure that consistency is restored after each modification to that set However, research in this area has concentrated on forward-reasoning mechanisms rather than backward-reasoning ones like Prolog. Dependency information of the sort described is also used in reason maintenance systems to effect *dependency directed backtracking,* and similar techniques have been used in Prolog interpreters (Bruynooghe and Pereira 1984).

'This work it supported by the Science and Engineering Research Council of Great Britain.

The Incremental Mechanism

The mechanism I will describe performs the following operations, given a goal clause G0, a set of definite clauses P and corresponding dependency structures 5 add a clause p to *P* and update 5, delete a clause p from *P* and update 5, and output corresponding substitutions for all refutations of G_0. Of course, many other sets of operations are possible For instance, facilities might be included for modifying parts of clauses, and this would permit a finer grain of incremental modification of S. The techniques described extend naturally to the incremental modification of the dependency structures for a set of goal clauses. It is also possible to incorporate negation as failure, but I will not discuss this problem here, nor will I discuss the problems of dealing with infinite proof trees

Now, let us consider the exploration of a search space as a sequence of states *S1 s*^{n}, where the transition from s, to s_{1+1} corresponds to one resolution step and possibly some backtracking Assume that the search space for a goal clause G_0 and a set of definite clauses P has been explored by passing through a sequence of states *T* and that this has generated a set of substitutions *B* Suppose that P' is the same set of clauses as *P* but with one addition Then, to explore the search space for GO and *P'* is to pass through a sequence of states *V* where *T'* is *T* with a number of extra subsequences inserted, and the set of answer substitutions generated *B'* will be a superset of *B* Similarly, suppose *P*^{j} is the same set of clauses as. P bur with one deletion. Then, to explore the search space for G_0 and P' is to pass through a sequence of states T' where *V* is *T* with a number of subsequences removed, and the set of substitutions generated *&* will be a subset of *B* This analysis would be more complicated for a non-monotonic logic since the deletion of a clause could then add to the search space and the addition of a clause could subtract from it

In each case, if a record of *B* and *T* is maintained then the search space for Go and *P* can be explored by propagating the consequences of changing P to *P'* through *T* thus obtaining T'. and then propagating the consequences of changing *T* to *V* through *B* thus obtaining B In addition to preserving *B* and *T.* it is necessary to record which substitutions in *B* depend on which subsequences in T, and which subsequences of *T* depend on which clauses in P respectively the *answer dependencies* and the *clause dependencies* Then, the deletion of a clause from P must bring about the remov.al of those subsequences in *T* which depend on it. giving T', and each deletion of a state in *T* must bring about the deletion of those substitutions in *B* which depend on it giving *B'* Also, it is necessary to record the *predicate dependencies* for each predicate in P. the set of points in *T* at which backtracking took place because of the exhaustion of clauses for that predicate Then, the addition of a clause for a predicate must bring about the restoration of each state at which clauses for that predicate were exhausted. For each such restored state search is resumed, thus generating new subsequences to be inserted into T giving 7", and possibly producing new substitutions to be added to *B,* giving *B'* Each such resumed search continues until a state is reached which is not dependent on the newly added clause, and will therefore already be in *T*

Each state corresponds to a sequence of goal clauses Go ,G_n, where G,-i is the result of resolving G, with some clause in P This can be conveniently represented by a list of clauses C_0 .C_n_i. where C, is the clause resolved with G, to obtain G_{1+1}, and in practise this can be simply a list of pointers or indices into the database of clauses (Clocksin and Alshawi 1986). Then, a sequence of states so-.Sn could be represented by a sequence of such lists. But since, in general, a, will be an extension of some s; where *j* < t, the sequence can be better represented as a tree, whose shape will mirror that of the search space Note that any given node in the tree will have one child for each clause in P for a particular predicate

The three dependency structures mentioned above must be maintained with respect to this tree, the answer dependencies, the clause dependencies and the predicate dependencies. For each leaf in the tree, a record is kept of whether the path from the root to that leaf constitutes a refutation, and if so, the corresponding answer substitution in *B* is indicated For each clause in P. a list is kept of those nodes in the tree which point to that clause Finally, for each predicate in *P.* a list is maintained of those nodes in the tree whose childrens' root nodes all point to clauses for that predicate

The cost of building the dependency structures during search is one tree insertion and two list insertions (of the kind that do not require search) for each resolution step performed The time savings are obtained at the expense of a storage overhead which will be directly proportional to the complexity of the search space

The deletion of a clause *C* proceeds as follows For each node *N* in the clause dependencies fur *C* the tree from *N* downwards is removed The removal of a node requires that all references to that node are deleted from the dependency structures So that this does not involve search, the clause and predicate dependencies can be threaded through the tree, and the removal of a node is then preceded by the deletion of its entries in those lists Whenever a leaf is reached, if that leaf is at the end of a refutation then the corresponding answer substitution is deleted from *B* So, the cost of deleting a clause is directly proportional to the total amount of search subspace whose existence depends on it If this is a small proportion of the overall search space then the savings resulting from adopting the incremental approach are correspondingly large. If it is a large proportion of the overall search space then the savings will be negligible and the extra cost will be of the same order as the cost of the original search

The addition of a clause *C* for a predicate *D* proceeds like this. For each node *S* in the predicate dependencies of *C* the state reprsented by .V is restored and the search is resumed until an area of search space is reached which has already been explored *(ie:* until the interpreter backtracks past *S)* The state represented by the node *N* is restored by tracing back from *N* to the root of the tree, forming a list *L* of the nodes on that path (in root to node order), and starting with the goal clause G_0, generating the sequence of goal clauses Gi...G_n by resolving each G, with the (i + \)th member of *L* to obtain G.^i Since all the search subspaces thus explored have to be explored anyway, the only overhead of adopting the incremental

approach is the initial cost of storing the dependency structures plus the cost of retracing the paths from each *N* to the root. As for deletion, the savings obtained will depend on the proportion of newly explored search space to overall search space. Instead of reconstructing the state, further time savings can be made, at the cost of further storage overheads, by recording the whole structure of the search space, including variable bindings.

The incremental mechanism described so far is capable of making savings when the effects of additions to and deletions from the set of definite clauses are confined to the outermost parts of the tree, near the leaves, or when they are confined to only a few branches and the tree is wide. The mechanism will also prove useful for clause *replacements* (the deletion of a clause followed by the addition of a clause for the same predicate) But it is often the case that the effects of a replacement are confined to a region near the root of the tree, leaving the peripheral foliage untouched. A finer grain of incrementality would be obtained if the mechanism avoided duplicating the work done below the affected area.

A complete solution to this problem is very difficult, and a detailed description is far beyond the scope of this paper. The mechanism must remove those parts of the tree that are dependent on the replaced clause whilst saving the branches below. New nodes are grown to replace the removed sections, using the new clause, and the saved branches are grafted back onto each new section that did not lead to a failure. As a result of growing a new section, some variables may change their bindings, and the consequences of these changes must be propagated through the rest of the tree. This can involve a similar pruning, growing and grafting process to that already described, since some clauses that failed to match before will match now, whilst others that did match before will fail to now. Again, sections of the tree will have to be lifted out and replaced, but this time a new section can have more offshoots than the one it replaces, so that when all the available saved branches have been grafted on, new ones have to be grown for any offshoots that are still incomplete.

I conclude with a brief discussion of how an incremental theorem prover might be embedded in a Horn clause planner similar to the one described by Kowalski (Kowalski 1979). The planner has to search for a sequence of actions $A\backslash .A_n$ such that $T_1...T_n*i$ is a sequence of states of affairs where $T\{$ is the initial state and $T_{nr}\backslash$ is the goal state and each T^{1+1} is the result of performing action A, in state T_x. Representing states of affairs as theories, the planner would be built on the meta-level, and would use meta-level predicates such as 'demonstrate (Theory, Goal)", "add-clause (Theory 1, Clause, Theory'2)" (Bowen and Kowalski 1982) which would be built-in and whose implementation would be based on the incremental approach described Note that the single predicate "add-clause'[1], will suffice for both addition and deletion of clauses. Any action performed in state T, can bring about changes to the axioms of the theory representing T, The planner would use "add-clause" to effect this change, and the incremental theorem prover would propagate its consequences through the theory. Then, in order to demonstrate some property of T,-i, such as the nature or position of a given object the only computation required would be a simple lookup, assuming that the property was one that was also demonstrated for T,.

Of course, it is not necessary to use Prolog for the meta-level problem solver as well as for the object-level representation formalism. But if Prolog is used then the following extension to the incremental mechanism may be necessary. Consider the solution of a goal "add-clause **(T1, C, T2)"**. It would be inefficient to keep entirely distinct copies of Tl and T2, since then the expense of copying Tl when a clause is added would obviate the advantages of the incremental approach. The same problem arises for deletion. Rather, what is required is a single structure which represents both theories. This could be obtained by labelling some of the nodes in the tree with the *contexts* (sets of axioms) in which those nodes (and their children) are to be considered part of the tree (deKleer 1986). A detailed investigation of such an extension is a subject for further research

References

1. Bowen K.A. and Kowalski R A , Amalgamating Language and Metalanguage, in Logic Programming, ed Clark K.L. and Tarnlund S.A. Academic Press (1982).

2. Bruynooghe M. and Pereira L.M., Deduction Revision by Intelligent Backtracking, in Implementations of Prolog, ed Campbell J.A., Ellis Horwood (1984).

3. Clocksin W.F. and Alshawi H, A Method for Efficiently Executing Horn Clause Programs using Multiple Processors, Technical Report, University of Cambridge Computer Laboratory (1986).

4. deKleer J., An Assumption-Based TMS, Artificial Intelligence 28 (1986), pl27.

5. Dennett D., Cognitive Wheels: The Frame Problem of Artificial Intelligence, in Minds, Machines and Evolution, ed Hookway C, Cambridge University Press (1984).

6. Doyle J., A Truth Maintenance System, Artificial Intelligence 12 (1979), p231.

7. Hayes P.J., The Frame Problem and Related Problems in Artificial Intelligence, in Artificial and Human Thinking, ed Elithorn A. and Jones D., Elsevier (1973).

8. Kowalski R.A., Logic for Problem Solving, North Holland (1979).