3.30

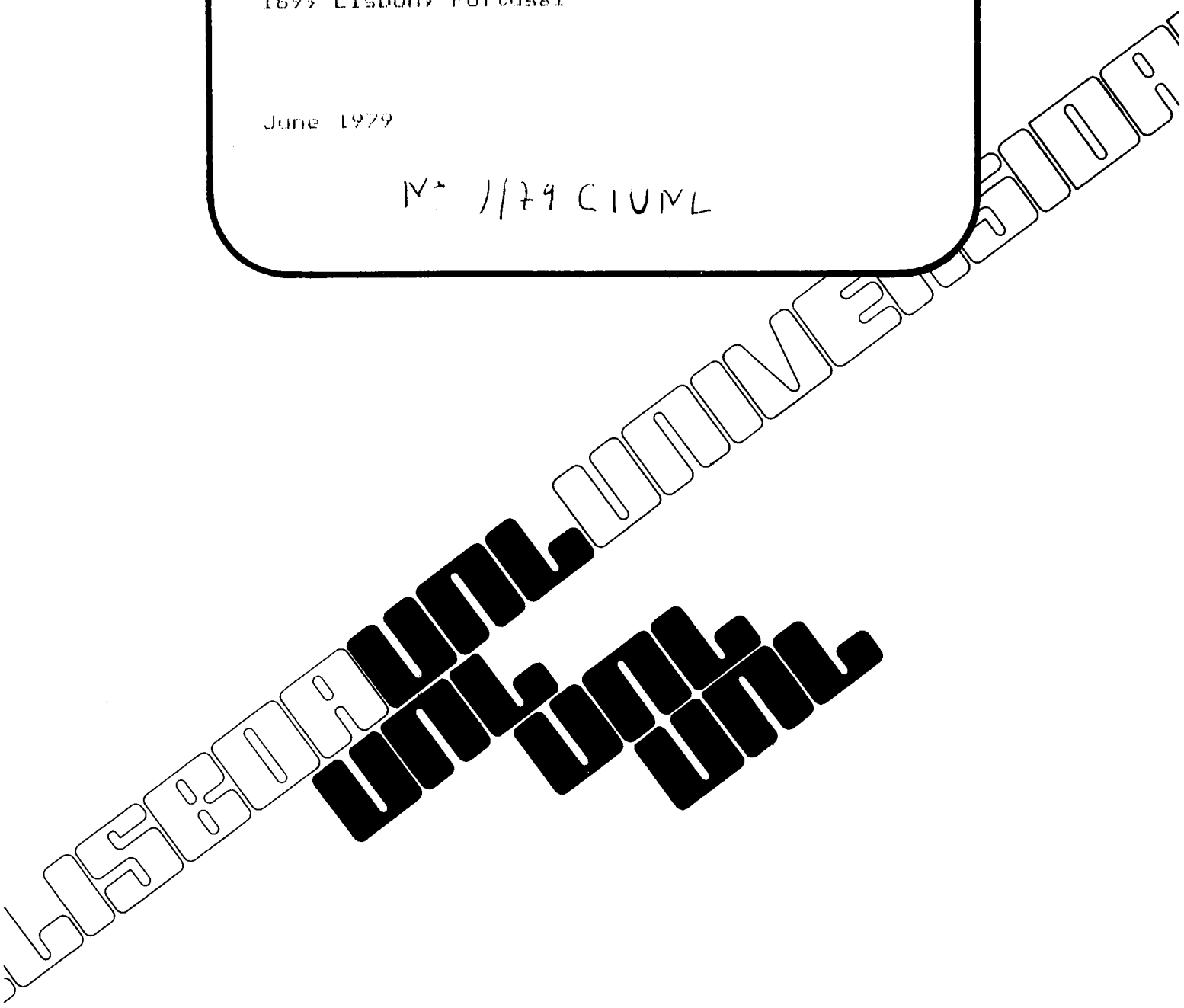
UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE INFORMÁTICA

Luis Moniz Pereira

Centro de Informatica
Universidade Nova de Lisboa
1899 Lisboa, Portugal

June 1979



SEMINÁRIO DOS OLIVAIS
QUINTA DO CABEÇO
OLIVAIS - LISBOA

Backtracking intelligently in AND/OR trees

Luis Moniz Pereira

Centro de Informatica
Universidade Nova de Lisboa
1899 Lisbon, Portugal

June 1979

Nº 1/79 CIUNL

Abstract

A general method is presented for guiding standard backtracking in AND/OR trees only to those nodes where repetition of goal failures may possibly be prevented, thus avoiding much useless search. Each goal that fails originates a stack of admissible backtrack nodes susceptible of solving it, and a simple rule combines individual stacks into a single new stack at AND and OR nodes. The admissible backtrack nodes at each failed goal are readily obtained if each object in a goal is tagged, implicitly or explicitly, with all the nodes on which it depends.

Keywords: artificial intelligence, problem solving, AND/OR trees, backtracking.

Standard backtracking in AND/OR trees is plagued by the inefficiency of looking for remedy where it cannot be found. When a goal fails, for some combination of the objects therein concerned, standard backtracking will hardheadedly return to its previous most recently made choice looking for alternatives, and so on until either some are eventually found or global failure occurs. Let us admit, a frequent case, that the final goal backtracked to is a brother, or a subgoal of a brother, of the failed goal. It may be that, unfortunately, none of the alternatives available there can change the particular combination of objects which had led to failure, for the reason

that the objects present in the two goals share no structure at all. Thus, for each alternative found, search will return to the failed goal with the original combination of objects intact, only to fail again...

The problem we address here is that of avoiding the double wastage involved: the one of, for each alternative considered, generating a possibly large subtree at every brother goal traversed by the backtracking and re-trying the failed goal without hope, and the one of having to undo all such doings.

Our solution to this and related problems consists in admitting backtracking from a failed goal only to those nodes where subsequent failure of the same goal may possibly be avoided.

Actual backtrack nodes are generated only on failure, not as the search moves forward. For a failing terminal goal, its admissible backtrack nodes are precisely those where modifications may occur to the failure originating components of the objects it refers to, plus its parent node, where alternative branches may exist. The goal's admissible backtrack nodes (numbered from the root) are organized in an ordered stack, with the most recent one coming first on the stack.

Let us assume, for the moment, that for each goal the information necessary for finding its admissible backtrack nodes in case of failure is readily available.

When a goal fails, one possible course of action is to pursue any of its brother goals, with the purpose of obtaining better backtracking information if they also fail. When all brother goals have been examined, backtracking is then made, first of all, to the most recent admissible backtrack node susceptible of solving, along with its descendants, the reasons for failure of all failed brother goals. Another course of action, is to return immediately to the failed goal's most recent admissible backtrack node, with no regard to the possible failures of brother goals. This is analogous to the immediate return, in standard backtracking, to the previous most recent choice point.

Both are useful. The first, in particular, is profitably used in pattern-matching. When several objects have to be matched at the same node (a covert AND), failure should not be acted upon at the first mismatch encountered. Continuation of the matching process may yield other mismatches, and admissible backtrack nodes corresponding to them. There is no use in backtracking to a node where only one of the mismatches may be remedied. If, however, the examination of brother goals is too costly because their subtrees may be large, then the second course of action can be preferable.

When backtracking on admissible nodes reaches a parent goal node (which is always an admissible node for its successors), the stacks of admissible backtrack nodes of its failed successors must be combined to form a new single ordered

backtracking stack. This new stack is then merged with the existing stack for that parent node, and the parent of the parent node is included if absent. The existing stack for the parent node may have remained from previous backtrackings or may be empty.

The rule for combining stacks is simple. The stacks coming from ORed branches are just merged to form the new stack, because each contains backtrack nodes susceptible of generating candidate solutions on different branches of the OR. The new stack obtained at ANDed branches coincides with that of the stacks which is less than or equal to any other stack, according to the lexicographic order among stacks. Recall that the stacks are themselves ordered, with the most recent node (that with the highest number) coming first on the stack. Since nodes are numbered from the root, the first admissible backtrack node of the new stack will be the least recent of the most recent nodes of all the stacks. From the point of view of that AND, if there are no alternatives at that node, then the next backtrack node to visit must be the next most recent node on the stack chosen as the new stack, and so on. If there are alternatives at that node, on the other hand, and should there be subsequent failing goals, perhaps at the same places, the stacks of admissible backtrack nodes will be obtained anew, and the process is repeated.

When, finally, there are no more admissible backtrack nodes to try on backtracking, failure of the top goal is reported. This happens only when backtracking reaches the top goal itself, because every parent node is admissible on failure of its successors.

This discussion assumed that at each node information was available concerning its admissible backtrack nodes in the face of failure. Specifically, we need to know, for each object component contributing to the failure, all the nodes on which its presence in the current goal depends.

Because parent nodes are always included as admissible backtrack nodes, node dependencies created by simple transmission of object components up and/or down the tree, through chains of ancestors possibly linked by common variables at brother nodes, need not exist explicitly. However, the original node where any concrete (ie. non-variable) value is acquired by an object component must be retained as a backtrack node by tagging with it the object component.

The only other kind of node which must be tagged to object components, because they are dependent on it, is where two (or more) object components, all previously introduced, are made dependent on each other at the present node. In that case, any previous or subsequent dependencies of those object components are passed among them via the present node. This situation is common in two-way pattern-matching (unification). (Where this kind of dependency does not arise, every object component with a

concrete value is tagged singly with the node where its value originated.)

Failure of terminal goals may occur for many reasons. For example, the goal may be equal to or a particular case of an ancestor; the goal has no potential successors; all potential successors cannot become actual successors on closer examination.

Irrespective of the reasons for failure, the parent goal is always an admissible backtrack node. In case failure does not depend on the objects involved in the goal, it will be the only such node. In case failure depends on the particular objects involved but it is not known which, then all the nodes on which the object components depend are additional admissible backtrack nodes.

In case failure is known to depend precisely on specific object components, then the only extra admissible backtrack nodes are exactly those on which such object components depend on.

Methods of object component tagging are too peculiar to problem representation and system implementation to be considered with any generality. In another publication [Pereira 1979], we expound the application of our theory and techniques to an intelligently backtracking interpreter of programs written in Prolog [Warren 1977] [Pereira 1978], a programming language based on predicate logic where every program execution is

naturally expressed as depth-first search of an AND/OR tree.

For a review of AND/OR trees refer to chapter 6 of [Loveland 1978], especially pages 335-351 and 359-364. Note that we use "brother goal" instead of Loveland's "partner goal".

Conclusion

Standard backtracking in AND/OR trees can be improved if information is available to direct it away from alternatives which are bound to fail. Such information can be gathered as the search moves forward. It suffices to tag each object component in a goal, implicitly or explicitly, with the nodes on which its generation and transmission depend. Upon failure of some goal, the nodes tagged to those of its object components contributing to failure constitute, besides its parent node, the only admissible backtrack nodes for that goal. These are organized in a stack, and the stacks of individual goals are combined, when search backs to an AND or to an OR branching point, into a single stack of admissible backtrack nodes, by means of a simple rule. This rule is general for AND/OR trees, although the methods of object component tagging depend on the problem representation chosen.

An application, described elsewhere, provided a new interpreter of Prolog that promotes intelligent backtracking in every program execution requiring it.

References

[Loveland 1978] Loveland,D.

Automated theorem proving: a logical basis

North-Holland

[Pereira 1978] Pereira,L.M.; Pereira,F.C.N.; Warren,D.H.D.

User's guide to DECsystem-10 Prolog

Laboratorio Nacional de Engenharia Civil, Lisbon.

[Pereira 1979] Pereira,L.M.; Porto,A.

A Prolog interpreter with intelligent backtracking

Universidade Nova de Lisboa, Lisbon

[Warren 1977] Warren,D.H.D.; Pereira,L.M.; Pereira,F.C.N.

Prolog - the language and its implementation

compared with Lisp

ACM Symposium on Artificial Intelligence and

Programming Languages

Sigart Newsletter, August 1977.