A Low-Cost High-Accuracy Intelligent Backtracking Algorithm

Alan D. Dewar John G. Cleary

Abstract

We present an intelligent backtracking algorithm with high accuracy and low overhead, especially for certain classes of algorithms. Information is associated with each variable binding and with each proof-tree node. Performance results indicate that our approach can attain a similar accuracy, but at a lower cost, when compared with other similar intelligent backtracking approaches.

1 Introduction

Standard interpreters for Prolog make use of backtracking to explore alternate possible paths to a solution. The straight-forward naive approach always retries the most-recently-succeeded goal whenever a failure occurs. An intelligent backtracking algorithm, on the other hand, is more selective in which goal it will retry. Specifically, only those goals which actually contributed to the failure should be considered as potential backtrack points. Retrying other goals cannot prevent the failure, and so simply results in unnecessary, and sometimes quite expensive, computation.

One of the issues affecting intelligent backtracking is the presence of non-logical predicates in a logic program. These include 'cut,' which alters the control flow and hence the meaning of a program, and built-in predicates which have side-effects, such as input/output and changes to the database. Our approach does not address this issue, but applies only to pure logic programs.

Intelligent backtracking necessarily involves overhead which is not present in naive backtracking. A number of tradeoffs are possible, involving space, time, complexity and accuracy. Generally, greater accuracy requires more space and time overhead and a more complex algorithm. Some previous approaches to intelligent backtracking are described below, ordered from most to least accurate and highest to lowest cost.

The approach of Cox, Matwin and Pietrzykowski [3, 7] separates the unification constraints from the proof-tree structure. The set of constraints, if consistent, defines a solution; otherwise, maximal unifiable subsets of constraints are determined, in order to select backtrack candidates. Although very precise selection of backtrack nodes is possible, a substantial overhead is required.

The approach of Bruynooghe and Pereira [1] is complementary to that of Cox, Matwin and Pietrzykowski, in that it deals with minimal non-unifiable subsets of constraints. This approach associates a deduction tree with each variable binding and with each call. The deduction tree associated with a variable binding specifies which calls affected the value to which the variable is bound. When a unification conflict is encountered, this information is used to determine which nodes should be considered as potential backtrack candidates. The deduction tree associated with a call is used to determine potential backtrack points when all clauses have been exhausted for a particular call.

Our approach is similar to that of Bruynooghe and Pereira, but differs in that we eliminate much of the overhead associated with a call. Whereas Bruynooghe and Pereira associate a deduction tree with each call, we instead set a flag in the individual nodes of the deduction tree itself when backtracking is initiated. The penalty of this simplification is a slightly lower accuracy in some instances. Our algorithm performs best with shallow data structures and combinatorial problems.

Kumar and Lin describe an algorithm which incurs very little overhead as execution proceeds forward [6]. It is only when backtracking commences that their scheme is invoked. Upon unification failure, the variables involved are examined and all proof-tree nodes which contributed in any way to their values are considered as backtrack candidates. The disadvantage of this is that many nodes which did not actually contribute to the failure may become backtrack candidates. This results in lower accuracy than our approach, but incurs less overhead.

Chang and Despain describe a semi-intelligent backtracking method which makes use of static data dependency analysis [2]. Potential backtrack candidates for calls are determined statically at compile time, based on worst-case assumptions. This results in almost no run-time overhead. However, the accuracy of selection of backtrack candidates is not high, due to the worst-case assumptions and to the inability of the approach to deal with information which crosses clause boundaries.

A simple form of our algorithm is described in the next section. Section 3 expands the description to the complete algorithm. In Section 4, some performance results are presented. Section 5 concludes with a discussion of the results and the applicability of the algorithm.

2 Basic Algorithm

Backtracking is initiated by a unification failure. Suitable candidates for a node to which to backtrack thus include all nodes which contributed to the value of the incompatible arguments to unification. For this reason, each variable has associated with it a set of *dependencies*. This set is determined at the time at which the variable is bound (possibly to another variable).

As an example of variable dependency sets, consider the unifications which occur in the following clause:

```
X=Y, X=a.
```

Given the standard left-to-right execution order, and assuming that the variable `X` is bound to `Y` rather than the reverse, the execution of this clause results in `X` being bound to `Y` and `Y` being bound to the constant `a`. This may be denoted by the expression `X←Y←a`. The dependency set of `X` includes `Y`, since `Y` was an argument to the unification which bound `X`. Similarly, the dependency set of `Y` includes `X`, since a unification involving `X` is what caused `Y` to become bound.

In addition to having a dependency set, each bound variable also keeps a reference to an associated deduction-tree node, referred to as its *binding node* and represented by a stack frame, denoting the call which caused the variable to become bound. In the above example, the binding node of `X` is the call `X=Y`, and the binding node of `Y` is the call `X=a`.

The unification algorithm is based on the standard recursive one. It receives as arguments the two expressions to be unified, as well as a *variable-argument set* consisting of all variables encountered as expression arguments at prior levels of recursion. The set received by the top-level invocation is empty. The variable-argument set of a particular invocation, which is passed to any recursive invocations, consists of all elements of the received set plus those expression arguments which are variables. It may be that neither expression is a variable, as in the unification of two structures or of two constants, in which case the variable-argument set is the same as the received set. If an argument consists of a variable, however, the variable is included in the variable-argument set, even if it is bound to some value.

As an example of variable-argument sets, consider the unification `W←X←f(Y) = f(Z←a)`. The recursive unification which results in `Y` becoming bound to `a` has as its variable-argument set `{W,Y,Z}`. `X` is not included, since it is not a direct argument to an invocation of unification.

When unification causes a variable to become bound, it is the variable-argument set which becomes that variable's dependency set. Upon unification failure, the variable-

argument set at the point of the failure (possibly within a recursive invocation of the unification algorithm) is returned, and denotes all variables whose bindings contributed to the failure. This set can be viewed as the dependency set of the failure.

Before backtracking commences after a unification failure, the failure's dependency set is traversed. Each variable in the set has its binding node flagged as a backtrack candidate. (All variables in the set must be bound, else the unification could not have failed.) The same is done for the binding nodes of any bound variables to which these variables are bound, and so on recursively. The current node is also flagged. The set of backtrack candidates thus consists of all nodes at which a binding occurred which contributed to the failure of the unification, plus the current node, which may have later OR-siblings.

Backtracking consists of examining proof-tree nodes, from most- to least-recently succeeded, until a node is found which is flagged. This is the node which is retried. If no such node exists, there are no more solutions.

When retrying a node results in the exhaustion of clauses for the call, the parent node of the retried node is flagged and backtracking recommences as above. The flagging of the parent is necessary, in order that later OR-siblings of the parent may be tried.

A node's flag is cleared only when the node is retried. Consequently, backtracking may encounter a node which was flagged by an earlier unification failure than the one causing the current backtracking. This is necessary, since retrying such nodes could resolve the earlier unification failure and produce a solution.

The following example illustrates the basic algorithm. Consider the following program and query:

```
p(a).                                                    (1)
p(_).

q(b,_).
q(_,a).

r(c).
r(d) :- ...  .

s(b).

?- p(X), q(X,Y), r(Z), s(Y).
```

The query has as its first solution `X` uninstantiated, `Y`$\leftarrow$`b`, and `Z`$\leftarrow$`c`. Execution proceeds as follows (refer to Figure 1). First, `p(X)` succeeds and produces the binding `X`$\leftarrow$`a` at node 1. At node 2, `q(X,Y)` then fails, due to the conflict `X`$\leftarrow$`a` $\neq$ `b`. Nodes

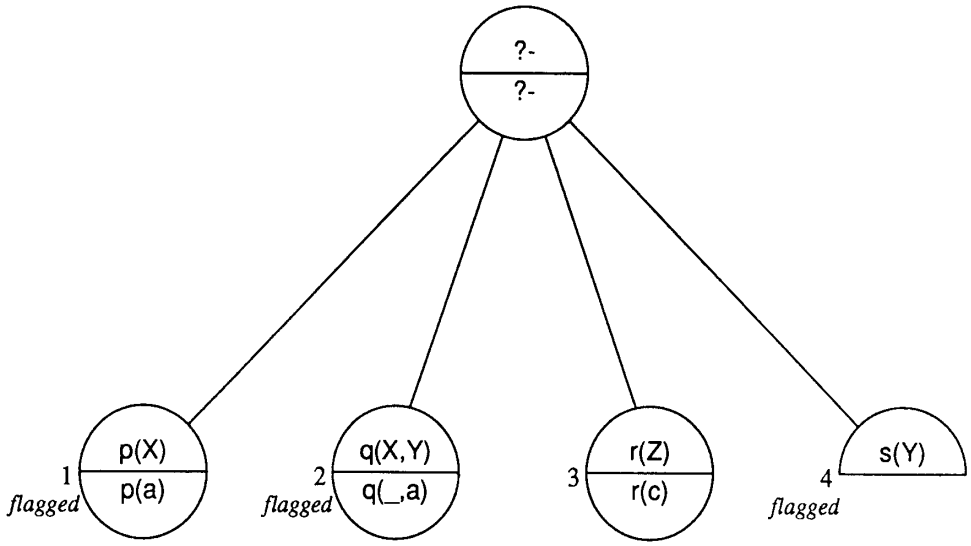

Figure 1: Deduction tree as call to `s(Y)` fails in Example 1

1 and 2 are flagged. Backtracking retries node 2, which succeeds with `Y←a`. The call `r(X)` then succeeds with `Z←c`. Next, as illustrated in Figure 1, the call `s(Y)` fails, due to the conflict `Y←a` $\neq$ `b`. This causes nodes 2 and 4 to be flagged. Backtracking fails to find any alternative clauses for `s(Y)`, and so continues, skipping node 3 and retrying node 2. No further alternatives exist for `q(X,Y)`, and so node 1 is retried. Forward execution then proceeds without further backtracking, producing a solution to the query.

3 Improvements

The basic algorithm above can result in non-optimal backtracking in a number of simple cases. Consider, for example, the following program and query:

```
p(a,Z) :- q(Z).                                    (2)
p(c,c) :- ...  .

q(b).
q(a).

?- X=a, p(Y,Y), X=b.
```

Figure 2 shows the state of the deduction tree when backtracking commences due to the conflict between `X←a` and `b`. Node 3 was involved in earlier backtracking, due to a conflict between `Z←Y←a` and `b`. As a result, node 2 was flagged as a back-

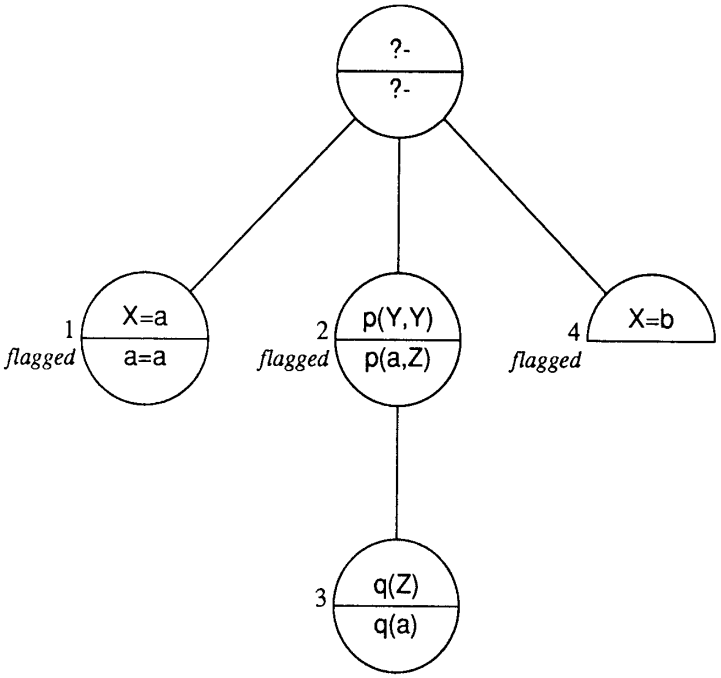

Figure 2: Deduction tree as backtracking commences in Example 2

track candidate. The current conflict has caused node 1 to be flagged. Backtracking could logically skip back to node 1, but does not do so because of node 2's flag. Instead, alternatives to the call `p(Y,Y)` are sought, involving arbitrarily much work and guaranteed to result in the same conflict as encountered at node 4.

In order to avoid such problems, a different kind of flag is introduced. Rather than a simple boolean indicator, a *node index* is used for flagging proof-tree nodes which are backtrack candidates.

Each node of the proof tree is assigned a unique node index, with indices monotonically increasing as nodes are generated by forward execution. This monotonicity applies to nodes generated following backtracking as well; i.e., the first node generated following backtracking will have an index greater than that of the last node generated before backtracking, even if the new node replaces one which had a much lower index.

Upon unification failure, backtrack candidates are flagged with the index of the node for which the unification was being attempted. This index is referred to as the *backtrack index*. Backtracking then involves a search for the most-recently-succeeded node whose flag value is not less than the backtrack index. Thus, nodes which in the simpler algorithm would have been flagged and retried may now be skipped because their flag index is small.

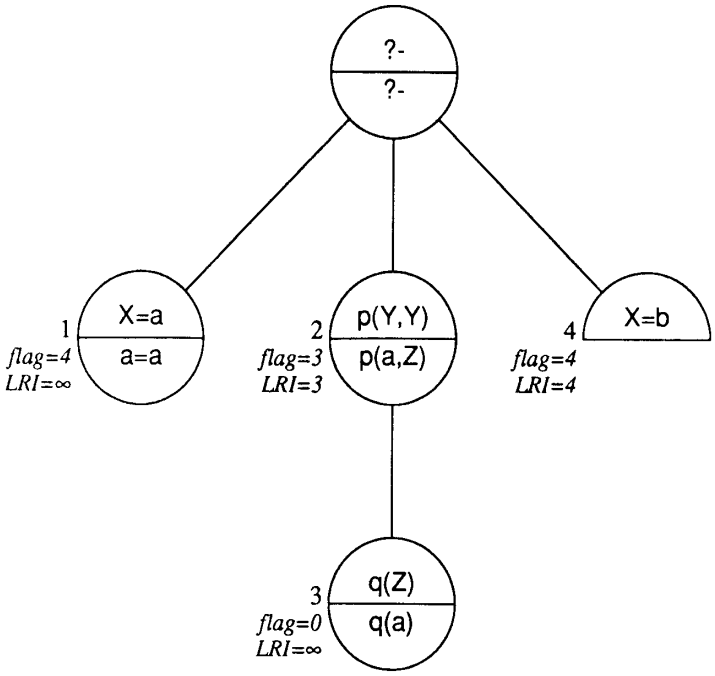

Figure 3: Deduction tree as backtracking commences in Example 2 (index flags)

Each node maintains a *least retry index*. This value is initially infinity, and is updated when the node is retried: if the backtrack index is less than the node's least retry index, the least retry index is set to the value of the backtrack index. Thus, the least retry index denotes the least backtrack index due to which a node has ever been retried.

If all clauses are exhausted for a call which is being retried, backtracking continues, but with the node's least retry index as the new backtrack index. The backtrack index can thus decrease, but can never increase during a single backtrack phase (although forward execution may be followed by new backtracking with a higher backtrack index). As with the basic algorithm, the parent node of the retried node is flagged, with the new backtrack index.

Continuing backtracking with a lower backtrack index constitutes the resumption of some earlier backtracking. However, since the flag values which denote backtrack points include not only the backtrack index but also any greater values, those nodes which should be retried due to later backtracking are also found.

Figure 3 shows how the improved algorithm deals with Example 2. Backtracking skips over node 2 and proceed directly to node 1. This occurs because the flag index of node 2 is 3 (node 2 was flagged due to a conflict in node 3), whereas the backtrack index is 4.

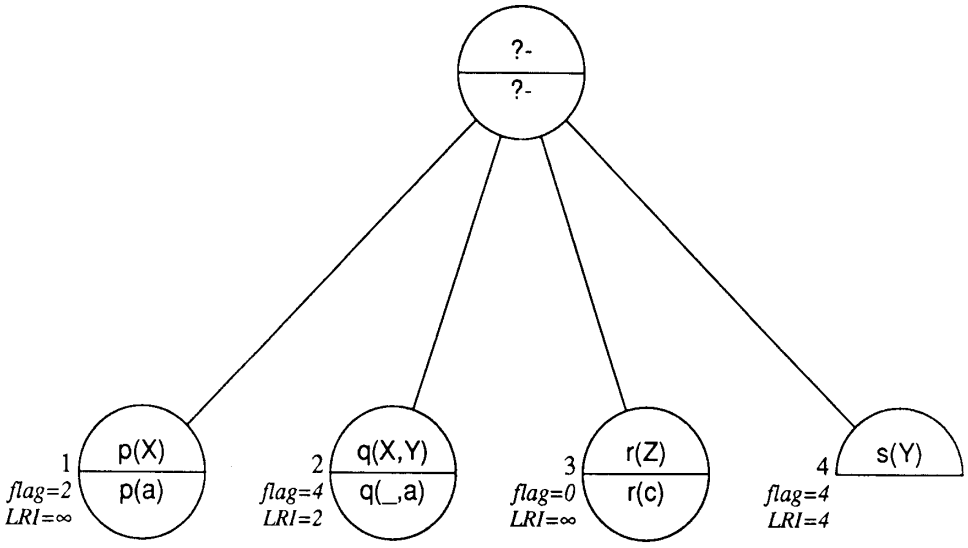

Figure 4: Deduction tree as call to `s(Y)` fails in Example 1 (index flags)

Figure 4 revisits Example 1. Execution proceeds as described earlier. When the call to `s(Y)` fails, the flag index of node 1 is 2 (node 1 was flagged when `q(X←a,Y)` failed to unify with `q(b,_)`), and of nodes 2 and 4 is 4 (`s(Y←a)` fails to unify with `s(b)`). Node 3 is not flagged. Backtracking is able to retry node 1 because the backtrack index changes from 4 to 2 when `q(X,Y)` is retried for the second time. This change takes place because the least retry index of `q(X,Y)` is 2. This example thus illustrates the need for the least retry index.

There are cases in which the selection of backtrack candidates can be inaccurate, as a result of the backtrack index decreasing. The following example illustrates this:

```
p(a).                                                    (3)
p(b).

q(c).

r(b,b).
r(a,d).

s(e).
s(c).

t(b).

?- p(X), q(Y), r(X,Z), s(Y), t(Z).
```

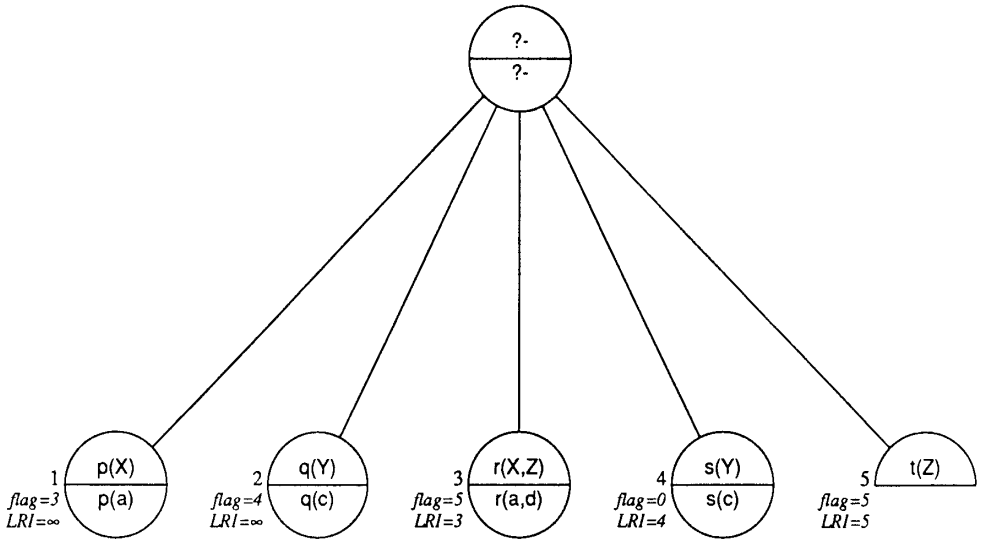

Figure 5: Deduction tree as call to `t(Z)` fails in Example 3 (index flags)

The proof tree for this example, as t(Z) fails, is shown in Figure 5. Backtracking at node 3 has caused node 1 to be flagged with index 3, and node 3's least retry index to be set to 3. Backtracking at node 4 has flagged node 2 with index 4, and has set node 4's least retry index to 4. The failure of `t(Z←d)` to unify with `t(b)` now causes node 3 to be flagged with index 5. Backtracking at node 3 will find no more clauses for `r(X,Z)` and will continue backtracking with new backtrack index 3. Since node 2's flag is 4, node 2 will be retried, even though it did not contribute in any way to the cause of the current backtracking.

4 Performance Results

Relative performance results for naive backtracking and backtracking with the basic and improved algorithms above are presented below. Table 1 shows the number of stack frames created by the interpreter. One stack frame is created for each call, and is used for all clauses which are tried for the call. Table 2 shows the number of variable bindings produced. This includes those bindings which were undone due to backtracking, as well as the bindings in effect for the solution. Table 3 shows the number of backtrack checks made. For the naive interpreter, this is the same as the number of retries done. For the intelligent-backtracking interpreters, this includes not only the number of retries done but also the number of times it was determined that a call need not be retried.

The overhead associated with variables' dependency sets is shown in Table 4. The largest average size of the dependency sets is 2.45, and the overall maximum size is 3.

Problem	Naive	Flag	Index	Flag/ Naive	Index/ Naive
5 queens	338	322	167	0.953	0.494
6 queens	6,619	6,326	2,487	0.956	0.376
7 queens	5,655	5,339	656	0.944	0.116
8 queens	128,606	123,831	20,191	0.963	0.157
map colour (good order)	45	45	45	1.000	1.000
map colour (bad order)	89,251	16,771	134	0.188	0.0015

Table 1: Stack frames

Problem	Naive	Flag	Index	Flag/ Naive	Index/ Naive
5 queens	992	896	420	0.903	0.423
6 queens	19,317	17,559	6,206	0.909	0.321
7 queens	18,239	16,343	1,580	0.896	0.087
8 queens	361,786	333,136	45,548	0.921	0.126
map colour (good order)	35	35	35	1.000	1.000
map colour (bad order)	33,730	9,966	73	0.295	0.0022

Table 2: Bindings

Problem	Naive	Flag	Index	Flag/ Naive	Index/ Naive
5 queens	428	336	130	0.785	0.304
6 queens	9,347	7,542	2,629	0.807	0.281
7 queens	8,713	6,929	594	0.795	0.068
8 queens	172,881	142,993	19,358	0.827	0.112
map colour (good order)	288	268	268	0.931	0.931
map colour (bad order)	1,070,733	111,185	606	0.104	0.00057

Table 3: Backtracking checks

Problem	Average	Maximum
5 queens	2.26	3
6 queens	2.34	3
7 queens	2.35	3
8 queens	2.45	3
map colour (good order)	1.37	2
map colour (bad order)	1.18	2

Table 4: Dependency-set size

Problem	Chang & Despain	Kumar & Lin	Bruynooghe & Pereira	Dewar & Cleary binds	Dewar & Cleary checks
6 queens	1.000	1.1660	0.651	0.321	0.281
7 queens	1.000	—	0.195	0.087	0.068
8 queens	1.000	—	0.229	0.126	0.112
map colour (good order)	1.007	0.9636	1.635	1.000	0.931
map colour (bad order)	0.001	0.0007	0.003	0.002	0.00057

Table 5: Comparison of algorithms: ratio with naive

The problems to which the backtracking algorithms were applied include a simple generate-and-test n-queens implementation, and a map-colouring problem. Both of these examples are taken from Bruynooghe and Pereira [1]. In each case, the statistics are for the first solution generated. The number of queens in the n-queens problem ranges from five to eight. The map-colouring problem involves a map of thirteen regions, with a "good" and a "bad" ordering of goals being tried.

The time overhead incurred by the intelligent backtracking algorithms is associated with unification and backtracking. Typically, these are also the most time-intensive aspects of a naive interpreter. Tables 2 and 3 thus give some indication of how much speedup can be expected with intelligent backtracking. The actual speedup will be somewhat lower than the operation-count ratio might imply, since naive backtracking need not incur as much overhead for each operation.

A comparison of our results to those of some other approaches is presented in Table 5. Note that a direct comparison is not possible, as the figures quoted are for different quantities. Bruynooghe and Pereira compare actual timings, as do Kumar and Lin, whereas Chang and Despain compare PLM instruction counts [4, 5]. (Kumar and Lin also compare PLM instruction counts and machine cycles, though those figures are not reproduced here.) The figures for our algorithm are taken from Tables 2 and 3.

5 Discussion

The information associated with variable bindings in our backtracking algorithm is similar to that of Bruynooghe and Pereira's approach [1]. However, where they maintain a set of proof-tree nodes, we maintain a set of variables. Bruynooghe and Pereira's approach can be more efficient when different variables maintain pointers to the same proof-tree node. We eventually need to compute the same information they already store, when backtracking is initiated and nodes are to be flagged. The overhead of potentially having several variables all refer to the same node can also be eliminated. Our algorithm could easily be modified to use Bruynooghe and Pereira's approach.

Our space overhead associated with proof-tree nodes, on the other hand, is less than that of Bruynooghe and Pereira. Instead of having an entire proof tree (or set of proof-tree nodes) associated with each node, we have only the overhead of three pieces of information: the node's index, its backtrack flag, and its least backtrack index. However, we pay a penalty of decreased accuracy in some cases, as the result of keeping less explicit information.

A variable's dependency set can, in principle, be arbitrarily large. In practice, however, a size of three or less is most typical. Larger sets occur when unification involves deep structures containing variables. Thus, our space overhead is minimal for shallow structures.

Intelligent backtracking yields the greatest improvement for programs in which a large amount of backtracking is done. Problems such as map colouring, which inherently require backtracking and which have very shallow data structures, can benefit the most from our intelligent backtracking algorithm.

A number of alternatives are possible to the approach we have taken to representing information associated with bindings, with proof-tree nodes and with the cause of backtracking. Different representations can result in more or less overhead and in improved or diminished accuracy. These alternatives are the basis for future research.

In conclusion, the intelligent backtracking algorithm we have described provides significant performance improvements for some classes of problems. By flagging proof-tree nodes directly, rather than explicitly associating proof trees with nodes, we achieve a lower overhead than Bruynooghe and Pereira's similar approach, though at some cost in accuracy.

6 Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*, pages 194–215. Ellis Horwood Limited, 1984.

[2] Jung-Herng Chang and Alvin M. Despain. Semi-intelligent backtracking of prolog based on static data dependency analysis. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 10–21, Boston, Massachusetts, 1985. IEEE Computer Society.

[3] P.T. Cox. Finding backtrack points for intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*, pages 216–233. Ellis Horwood Limited, 1984.

[4] A.M. Despain and Y.N. Patt. The berkeley prolog machine. In *Proceedings of the IEEE Spring CompCon Conference*, San Francisco, California, 1985.

[5] T. Dobry, A.M. Despain, and Y.N. Patt. Performance studies of a prolog machine architecture. In *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, Massachusetts, 1985.

[6] Vipin Kumar and Yow-Jian Lin. An intelligent backtracking scheme for prolog. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 406–414, San Francisco, California, 1987. IEEE Computer Society.

[7] Stanislaw Matwin and Tomasz Pietrzykowski. Intelligent backtracking in plan-based deduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(6):682–692, November 1985.