Implementation Notes on SPRFN–A Natural Deduction Theorem Prover

TR87-028

September, 1987

Xumin Nie, David Plaisted

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27514

Implementation Notes on SPRFN
-- a Natural Deduction Theorem Prover

Xumin Nie
David A. Plaisted
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27514

Abstract. The natural deduction theorem prover sprfn has been operational since August 1986. Since then, many experiments have been performed on it to test different strategies and data structures; many refinements have been added in the attempt to make it more efficient. Some of these attempts have been successful and some have not been. In this report, we will describe the evolution of our ideas, discuss the test results, motivate and justify our decisions and draw some conclusions. We hope that we can share our experiences as well as lessons in our research on this theorem prover.

This research was supported in part by the National Science Foundation under grant DCR - 8516243.

1. Introduction

Sprfn is a natural deduction type system that proves theorems in first order logic. It is based on a modification of the theorem proving strategy described in Plaisted[82]. Being a theorem prover in first order logic, this prover may also be viewed as an extension of Prolog to full first order logic, that is, non-Horn clauses. Sprfn possesses some desirable features both as a theorem prover and as an extension to the existing programming language -- Prolog. Some features about its implementation are also worth mentioning. We will briefly mention them in what follows. For a more detailed description of these features and their justifications, see Plaisted[87].

[1] Negation is treated as in first-order logic, i.e., with semantics of first-order logic. This is sound, unlike the treatment of negation in Prolog. Also, the prover performs true unification -- unification with occur-check.

[2] The prover is capable of general term rewriting to replace subexpressions by equivalent ones. The term rewriting capability gives the user one way to provide domain dependent knowledge about the problems he is working on.

[3] The prover can perform both forward chaining and backward chaining.

[4] The input syntax for the prover is much the same as the syntax of Prolog. This is expected since the prover is intended to be an extension of Prolog. The prover also has a convenient interface to Prolog source code. Some tasks can be performed efficiently using this interface.

[5] The prover allows user interaction to provide guidance during the course of proofs for problems with equality involved.

[6] The prover uses Prolog style depth-first search with a gradually increasing depth bound , i.e., depth-first iterative deepening search. This search strategy is complete. It can also be easily implemented in Prolog by taking advantage of Prolog's built-in backtracking mechanism.

[7] Subgoals and the solutions to the subgoals are "cached" so that if a subgoal is seen more than once, work is not repeated.

Sprfn is implemented in C-Prolog. It has been designed for use with as little user guidance as possible, for users who have little background about theorem proving. Thus the default set-up of the prover is carefully tuned to provide reasonable overall performance. However, a set of flags and parameters are also provided. These flags and parameters can be set or adjusted by more sophisticated users to have more control over the prover. The first working version is operational around August, 1986. Since then, a number of optimizations and experiments have been done to improve and test the performance of the prover. In this report, we will describe the important efforts invested during the process of implementing this theorem prover and the ideas behind these efforts. We will describe the evolution of those ideas and their impact on the current status of the prover, and justify the decisions and conclusions we have made. We will also share our experiences and lessons in implementing, tuning and using this prover.

2. The Modified Problem Reduction Format

We will introduce the deduction system underlying sprfn -- the modified problem reduction format. The following is a concise description of the deduction system. This description is taken from Plaisted[87], to which interested readers are referred to for a complete description and discussion of this strategy.

A **clause** is a disjunction of literals. A **Horn-like clause** is of the form L :- $L_1, L_2, ..., L_n$ where L and L_i's are literals. L is called the head literal. L_i's constitute the clause body. A clause C is converted into a Horn-like clause **HC** as follows. One of the positive literal in C is chosen as the head literal of **HC** and all other literals in C are negated and put in the clause body of **HC**. For example, $\bar{P} \vee \bar{Q} \vee R \vee W$ can be expressed either as R :- P, Q, not(W) or W :- P, Q, not(R). Only one such representation is chosen for each such non-Horn clause. If C contains only negative literals, we use **false** as the head literal of HC. Thus $\bar{P} \vee \bar{Q}$ will be represented as false :- P, Q.

Assume S is a set of Horn-like clauses. In the modified problem reduction format, a set of inference rules for S is obtained as follows. For each Horn-like clause L :- $L_1, L_2, \ldots, L_n$ in S, we have the following clause rule. We call the $\Gamma's$ on the left of the arrow $\rightarrow$ *assumption list.*

Clause Rules

$$\frac{\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1,\ \Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2,\ \ldots,\ \Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

This rule corresponds to the following idea: if the initial subgoal is $\Gamma_0 \rightarrow L$, then make $L_1, \ldots, L_n$ subgoals in succession; add to assumption list Γ_0 successively the literals that are needed to make each one of L_i provable; finally, return $\Gamma_n \rightarrow L$ where assumption list Γ_n contains all the literals needed to make $L_1, \ldots, L_n$ provable. We also have assumption axioms and case analysis (splitting) rule. The L in the assumption axioms is a positive literal.

Assumption Axioms

$$\Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L \quad \text{if } L \in \Gamma$$

$$\Gamma \rightarrow \bar{L} \Rightarrow \Gamma, \bar{L} \rightarrow \bar{L}$$

Case Analysis (splitting) Rule

$$\frac{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \bar{M} \rightarrow L,\ \Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}$$

The following is a theorem in group theory. We use this example to illustrate the input format to sprfn and the proof generated by sprfn. This theorem states that, if the square of any element in a group equals to the identity element of the group, the multiplication operation of the group is commutative. In the input clauses below, *e* is the identity element, *p(X, Y, Z)* means that the product of *X* and *Y* is *Z*. The labels c1, c2, ..., c7 are given to the clauses for easy reference and do not constitute the input.

```
c1:   p(X, e, X).
c2:   p(e, X, X).
c3:   p(X, X, e).
c4:   p(a, b, c).
c5:   p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
c6:   p(X, V, W) :- p(X, Y, U), p(Y, Z, V), p(U, Z, W).
c7:   false :- p(b,a,c).
```

In the input clauses above, clauses c1, c2 and c3 are the axioms about the identity element of the group. C5 and C6 are the associativity axioms. C4 and C7 are the negation of the theorem. The following is the proof generated by sprfn for this theorem.

```
false:-
        p(b,a,c) :-
                p(b,b,e)
                p(b,c,a) :-
                        p(a,c,b) :-
                                p(a,a,e)
                                p(a,b,c)
                                p(e,b,b)
                        p(c,c,e)
                        p(a,e,a)
                p(e,c,c)
```

In the proof above, *false* is the top-level goal. The subgoals belonging to one goal are aligned on the left. For example, *p(b,b,e), p(b,c,a), p(e,c,c)* are subgoals of *p(b,a,c)*. This corresponds to

$$\frac{[]\rightarrow p(b,b,e),\ []\rightarrow p(b,c,a),\ []\rightarrow p(e,c,c)}{[]\rightarrow p(b,a,c)}$$

which is an instance of the clause rule derived from the input clause c6. We omit the assumption lists for subgoals in the proof since they are all empty.

3. Discrimination Net

3.1. Caching Subgoals

In sprfn, the subgoals which have been worked on are cached. The solutions generated during the course of the proof are also cached for later use. The idea of caching is to avoid repeated work involved in trying to solve the same subgoal more than once. After the prover has generated all the solutions for a subgoal G, a clause *done(D, G)* will be asserted, where D is the measurement of the effort invested in trying to solve the subgoal G. We will call *done(D, G)* a *done clause*.

Each subgoal will be attempted with some effort bound. This effort bound indicates the maximal amount of effort allowed at present to confirm a particular subgoal. If it requires more effort than that the effort bound allows to confirm a subgoal, this subgoal will be attempted later with a larger effort bound. Suppose a subgoal G is to be attempted with the effort bound D. The prover first checks to see if any solution which has been generated could be used for solving G. After all the old solutions have been used, the

prover is about to generate all the possible new solutions for G with effort bound D. However, before it proceeds with trying to generate the new solutions for G, sprfn checks whether there is a done clause *done(D1, G1)* such that D1 is greater or equal to D and G is an instance or a variant of G1. If there is such a done clause, sprfn does not need to work on subgoal G at all since all the possible solutions for G have been generated already. If there is no such done clause, the prover will proceed to generate all the possible new solutions for G. When the prover finishes, a done clause *done(D, G)* will be asserted to indicate that subgoal G has been worked on with effort bound D.

3.2. Basic Structure of Discrimination Net

Because of the possible large number of subgoals, the caching described above may involve a great deal of housekeeping. We used the built-in prolog database to implement the caching by asserting the done clauses. Consequently, checking whether a subgoal is an instance or a variant of another subgoal which has been worked on requires a linear search among all done clauses. In the attempt to speed up this search process, a new data structure called a discrimination net was added to the prover.

Conceptually, a discrimination net organizes a set of strings of symbols as a tree. This tree is organized in the manner that, each node in the tree contains one or more branches which are labelled by the symbols in the strings; and the branches diverge at the places where the strings have different symbols. To search a string in a net, we follow the following discrimination process. Start with the root node of the net as the current node and the first symbol of the string as the current symbol. At any moment, if the current node contains a branch labelled with the current symbol, follow the branch to obtain a new node; if no such branch exists, the string is not in the net; if such a branch exists, repeat the above process with the new node as the current node and the next symbol in the string as the current symbol. When the current node is a leaf node and there is symbol in the remaining part of the string, the string is in the net. Regarding the subgoals as being strings of symbols, we implemented this data structure in sprfn. All the done clauses are stored in a discrimination net.

3.3. Implementations of Discrimination Net

There are three implementations of the discrimination net. The last two implementations are the results of the unsatisfactory performance of its predecessors. We will describe each implementation in this section.

The first implementation uses individual symbol (function symbols, predicate symbols and variables) as label. Each node in the net is represented as a list of <symbol, branch> pairs. Each pair in the list represents a branch labelled by the symbol in the pair. The discrimination net is implemented using the prolog database utilities. A node will be represented by a clause *node(pair_list)* in the database. Note each such clause can be referenced using a reference number assigned by the system so linear search in the database will not be necessary. In this implementation, discrimination net is designed to be a generator. User can specify four kinds of subgoals to be generated: subgoals which are no less general than the current subgoal, subgoal which are no more general than the current subgoal, subgoals which are unifiable with the current subgoal and subgoals which are variants of the current subgoal.

First implementation does not give satisfactory results. We spot several things which are likely to cause inefficiency: verifying the relationship between the subgoals in the database and the current subgoal; the linear search at each node when looking at a branch labelled with a symbol and unfolding a subgoal into a list of symbols before searching. In the second implementation, discrimination net is used as a filter to generate the subgoals which may be unifiable with the current subgoal. In order to avoid the linear search at each node, the input clauses are preprocessed so that each node will have a fixed number of pairs and each symbol has a fixed slot in the nodes. This can be done since the symbol set in the input clauses is fixed for each theorem if we treat all the variables as if they were one symbol. In this implementation, some prolog clauses will be generated at preprocessing time for some frequent operations. For example, suppose there are four symbols in the input clauses, the following clauses will be generated:

```
replaceRef(1, N, pos(X1, X2, X3, X4), pos(N, X2, X3, X4)).
replaceRef(2, N, pos(X1, X2, X3, X4), pos(X1, N, X3, X4)).
replaceRef(3, N, pos(X1, X2, X3, X4), pos(X1, X2, N, X4)).
replaceRef(4, N, pos(X1, X2, X3, X4), pos(X1, X2, X3, N)).

search(P, Type, S, E) :- nonvar(S), % for atomic constants.
```

```
        clause(struct(_, L), _, S), arg(1, L, _:E))).

search(X, Type, S, E) :- nonvar(S), % for variables.
        var(X),!, clause(struct(_, L), _, S),
        functor(L,_,N), skip(L, E, 1, N)).

search(X, Type, S, E) :- nonvar(S), % for constants.
        atomic(X), clause(struct(_, L), _, S),
        index(X/0, I), arg(I,L,_:E).


search(f(X1, X2), Type, S, E) :-    % for function f. I is constant.
        nonvar(S), clause(struct(_, L),_ S),
        arg(I, L, _:A), nonvar(A),
        search(X1, Type, Ref, Ref1), search(X2, Type, Ref1, E).
```

The structure *pos(X1, X2, ..., Xn)* has one slot for each symbol in the input. Each *Xi* represents a <symbol, branch> pair. The structure *struct(Dname, pos(X1, X2, ..., Xn))* represents one node in the discrimination net, where *Dname* is the name of the net. Each symbol will have one *replaceRef* clause generated for it. The *replaceRef* clauses will be used to update individual node. Note that, by utilizing the prolog built-in unification capability, we have avoided the explicit linear search at each node when modifying it. For each predicate symbol and function symbol, a *search* clause is generated. These clauses are used to search the discrimination net. By giving each symbol an unique index which is the slot number for this symbol in the *pos* structure, we also avoid the linear search at each node when searching the net by utilizing the prolog built-in capability. In the first implementation, a searched subgoal is unfolded into a list of symbols before searching. We avoid this too by generating the *search* clauses.

Second implementation does not provide the improvements we have hoped. In both implementations, the insertion and deletion operations result in a large number of *struct* structures being asserted into and retracted from the database. By using individual symbol as the labels, the search operation is also expensive since each symbol in the subgoal will result in a prolog procedure call. In the third implementation, we use the literals in the subgoal rather than the symbols as labels. In this implementation, node is not represented as a list of pairs. Rather, each node is given a unique id when it is generated at runtime as the new done clauses are inserted. Each node is represented by a prolog procedure the code for which is generated as the new done clauses are inserted. The code for this implementation is given in Appendix A. Only insertion and search functions are implemented here. We are still using the discrimination net as a filter to

generate the possible unifiable subgoals with the current subgoal.

3.4. Performance of Discrimination Net

Table 1 gives the performance data of sprfn using the different implementations of the discrimination net. The degeneration of performance is obvious from these data. There are several reasons for the degeneration. One possible reason is that the number of cached subgoals is generally small. Thus, the overhead for maintaining a discrimination net is too much to be compensated for by the saving in search time. Note that, for the examples whose proofs take large amount of cputime to obtain, the performance does not vary very much when the discrimination net is added. This suggests that maintaining the discrimination net can only be effective when there are a large number of subgoals. Another reason seems to be the language we are using. Note that we represent the data structure using the prolog database facility. To modify the data structure, we have to assert into or retract from the prolog database. This may contribute a great deal to the large overhead. The third implementation performs better than the first and second. The reason is twofold. On one hand, it asserts less than the other two do since it is indexing on a larger structure. On the other hand, it does not retract at all when the data structure is modified as other two implementation do. We do not see any better facility provided by Prolog for implementing the discrimination net. We think that Prolog is not suitable for implementing such "lower-level" data structure due to its highly abstract execution mechanism. Other languages such as C may prove to be suitable.

4. Caching most Dominating subgoals

We introduced *done clause* in last section. A done clause *done(D, G)* states that the subgoal G has been worked with effort D. Note that a subgoal or its instances may appear several times during the course of the proof, with different amount of effort involved. Suppose we have two done clauses, *done(D1, G1)* and *done(D2, G2)*, where G1 is an instance or an variant of G2 and D2 is greater or equal to D1. These state that G2 has been worked on with larger amount of effort invested and all solutions that have been obtained for G1 with amount of effort D1 have been obtained for G2 with the amount of effort D2. We only need to maintain *done(D2, G2)* to have complete information. The done clause *done(D1, G1)* can be

deleted. This suggests that for any subgoal and its possible instances which appear as subgoals, some done clauses are not needed. We say *done(D1, G1)* is dominated by *done(D2, G2)* if G1 is an instance or a variant of G2 and D2 is greater or equal to D1. We only need to maintain the most dominating done clause and delete the dominated ones. By only keeping the most dominant done clause, the space taken to cache subgoals will never be greater than the number of essentially different subgoals generated.

Sprfn was modified to keep only the most dominating done clause and delete the dominated ones. We call this **clause elimination**. Table 2 shows the execution time and the number of done clauses maintained by sprfn in the data base at the end of proof. Two sets of data are listed. The set on the left is for sprfn without clause elimination implemented. The set on the right is for sprfn with clause elimination implemented. One can easily see that the performance of the prover does not vary very much. This is due to the fact that the number of done clauses is generally small. The 5.68% loss in cputime is the result of the overhead of detecting dominating relationship and maintaining the database.

Let us go back to the discussion of discrimination net. We conjectured that one of the reasons for the inefficiency of discrimination net is that the number of cached subgoals is generally small. The data in this section have enforced this conjecture. The overhead of maintaining the discrimination net is just too big to be compensated for by the speedup in the search time since the number of subgoals cached is too small.

5. Parameters and Flags

Sprfn is designed to be used with as little user guidance as possible. Thus it may appeal to a larger community. People with little or no background in theorem proving will be able to use it. However, a set of parameters and flags are identified and provided which more sophisticated users can use to enforce more control on the prover. This section explains the motivation for the provision of these parameters and flags and their effect on the behavior of the prover.

Before we go into the explanations of these flags and parameters, we first explain how the search bound is increased. As we indicated before, sprfn uses depth-first iterative deepening search. We use the following formula to calculate the new search bound based on the old search bound. The starting search

bound is 5.

$$NewBound = \begin{cases} Oldbound + 2 & \text{if } Oldbound \le 9 \\ OldBound \times \frac{4}{3} & otherwise \end{cases}$$

Sprfn can alternate between backward chaining and forward chaining. The search bound for forward chaining is approximately two thirds of that of backward chaining.

The remaining part of this section will be devoted to motivating and explaining the important flags and parameters. Some experimental results will be given along the way. Most of these data are obtained using the most recent version of sprfn so the results for some problems are not the best results we have obtained. We especially note that, when we give results by setting some flags or parameters, all the other flags and parameters are at their default states. These parameters and flags affect the following aspects of the prover, which will be discussed separately in what follows:

[1] How backward chaining and forward chaining are combined.

[2] How the old solutions, lemma as they are sometimes called, are used.

[3] How the size of the subgoals is limited by the search bounds.

5.1. Perform only Backward Chaining or Forward Chaining

The prover performs backward chaining and forward chaining alternatively. During the backward chaining phases, the prover tries to derive the top-level goal, generating subgoals as needed. During the forward chaining phases, on the other hand, it only derives the goals that can be decomposed into subgoals already solved, without regard to the relevance of the derived goals to the top-level goal. Users can direct the prover to perform only backward chaining or forward chaining by setting flags which are available in the prover.

Forward chaining is helpful in generating simple facts from the known facts. But for problems with a large number of input clauses, forward chaining tends to clutter up the prover with too many facts, many of which may be redundant and irrelevant. Those redundant and irrelevant facts can easily lead the prover astray, resulting in much useless work. This has been confirmed in many occasions where, by permitting forward chaining, the prover ran out of space before it could obtain the proof. Of course, it might well be that the forward chaining is not well controlled in the prover. As a matter of fact, it is still an interesting research problem how to control forward chaining to make best use of it. Backward chaining, on the other hand, has some advantages. Backward chaining always starts with the top-level goal and generates subgoals as needed. Together with the goal-subgoal structure of the deduction system underlying the prover, backward chaining tends not to generate many useless facts. Thus space will be utilized more efficiently. In some of our experiments, the prover obtained the proofs by performing only backward chaining for some problems whose proofs could not be obtained if both forward chaining and backward chaining were performed. What happens was that space ran out before proofs were obtained These indicate the relative space efficiency of backward chaining. Better space efficiency is a big advantage in theorem proving since it is usually not time, but space, that is run out more quickly. Moreover, backward chaining is able to use only those clauses necessary for achieving the top-level goal, effectively discarding the useless input clauses, if any.

Table 3 gives the statistics for the performance of the prover when only backward chaining is performed. The statistics when both forward chaining and backward chaining are performed are also listed for easy comparison. The overall improvement in performance is obvious from the statistics. This set of data is very representative of our experiments on backward chaining. It gives better performance for most of the problems. By performing only backward chaining, the prover performs better on 31 of 59 problems, with the average speedup of 53.8% relative to the default prover in terms of inferences. But for the 10 problems out of 59, the prover suffers efficiency loss with the average slowdown of 135.6% relative to the default prover, also in terms of inferences. We like to point out that the performance of the prover is better than it may be suggested by the three average ratios. The performance of the prover when performing only backward chaining degenerates so much on one example (wos10) that the corresponding ratio for this problem contributes too much to the overall average, thus overweighs the gains in performance for most of the

problems.

Space efficiency is better. Ls65 is an example. This problem is to prove the transitivity of the less-than relation, given the axioms for integer addition, multiplication , equality and less-or-equal relation. For this problem, the prover runs out of space before obtaining the proof when performing both forward chaining and backward chaining. Forward chaining results in an explosion of search space due to the relatively large number of input clauses of this problem (20). When performing backward chaining only, the prover gets the proof after generating 67 solutions. When performing both forward chaining and forward chaining, the prover still fails to obtain the proof after generating over 600 solutions.

The prover tends to use the clauses relevant to the proof when performing only backward chaining. Cases in question are wos13, wos14 and ls26. These three problems all state some closure property of a subgroup of a group. The proofs for these theorems do not need the associativity axioms and equality axioms, as exhibited below by the proofs for wos13 and wos14.

```
        wos13 proof

false:-
        o(e) :-
                o(a)
                o(a)
                p(a,g(a),e)
        p(e,j(e),j(e))
        p(j(e),e,j(e))


        wos14 proof

false:-
        o(g(a))
                o(e);-
                        o(a)
                        o(a)
                        p(a,g(a),e)
                o(a)
                p(e,g(a),g(a))
```

In the above proofs, literal *o(a)* states that *a* belongs to the subgroup. Literal *p(X, Y, Z)* states that the product of *X* and *Y* is *Z*. *e* is the identity element. *j* is a skolem function symbol and *g* is the inverse function.

By performing backward chaining only, the prover just generates the required subgoals only involving the predicates describing the sub-group and some simple facts directly provable from the basic axioms for a group, thus disregarding the associativity axioms and equality axioms. When the prover performs both forward chaining and backward chaining, it derives a large number of facts using the equality axioms and associativity axioms. These irrelevant facts completely distort the search space, resulting in a great amount of fruitless search.

It is interesting to note that, for the problems whose proofs are long, only performing backward chaining makes prover's performance degenerate. The two noteworthy examples are wos10 and ls103. The proof the prover generates for wos10 has 7 levels of recursion with maximal branching factor 3. For ls103, the proof has 8 levels of recursion with maximal branching factor 3. For these two problems, the prover finds proofs at a larger search bound when only performing backward chaining. When the prover is performing both forward chaining and backward chaining, the search bound for wos10 is 7, and for ls103, 14. When the prover performs only backward chaining, the search bound for wos10 and ls103 are 9 and 18 respectively. These two examples show the usefulness of forward chaining in generating useful facts relevant for the proof, thus reducing the depth of the search space. This is one of the reasons why we include both forward chaining and backward chaining in sprfn.

5.2. To limit the size of subgoals at certain search bound.

For some examples whose input clauses contains large numbers of variables and function symbols, the prover tends to generate many subgoals having large complex structures. Large subgoals are not desirable for two reasons. First, larger subgoals take more space. Second, if the subgoals contain variables, the size of subgoals will "snowball", resulting in more and bigger subgoals. This size propagation will be more serious if the subgoals contain many variables. We somehow would like the prover to focus its effort on the subgoals with reasonable size relative to the current search bound, since, if the proof for one problem can be obtained with a small search bound, the subgoals in the proof are not likely to get very large. We use the search bound since the search bound represents the maximal amount of effort to invest on any subgoal.

Based on the reasoning above, a flag called *maxsize* is introduced. When this flag is set, sprfn checks the size of a subgoal before attempts to prove it. Only if the size of the subgoal does not exceed the current search bound, is this subgoal attempted. Since the search bound is increased after each round, the size allowed for the subgoals will also be increased. Thus it does not remove the completeness of the prover to set the *maxsize* flag. When the *maxsize* flag is set, the syntactic complexity of the subgoals will be limited and the syntactically simplier subgoals will be favored. However, to set *maxsize* may make the prover search deeper into the search space than it would when *maxsize* is not set. This is because the number of subgoals in the search space with a certain search bound will be smaller when *maxsize* is set. Experiments with *maxsize* show dramatic improvements in the overall performance.

Table 4 shows the performance data of the prover when the *maxsize* flag is set. As always, performance data of prover in its default setup are listed for easy comparison. In table 4, *proof size* is the search bound at which the prover obtains the proofs. The prover performs better on 27 of the 58 problems with the average speedup of 51.1% in terms of inferences, when *maxsize* is set. The average slowdown of 1753.1% for the 15 problems is a strong indicator of the unsteady performance of the prover when *maxsize* is set. Again, the performance of the prover when *maxsize* is set is not as bad as it may be suggested by the average ratios of cputime and inference. The two ratios are large because that the prover with *maxsize* set performs so poorly on several problems that the ratios for these problems overweigh the performance gains on other problems.

We can immediately notice the larger proof sizes. The proof sizes increase 25% on the average when *maxsize* is set. This is because the size of the subgoals are limited by the search bound, so the number of subgoals will be reduced. As a consequence, the prover has to go deeper into the search space to find the proofs, thus resulting in the larger proof size. In spite of the increased search bound, however, it often provides dramatic improvements to set *maxsize*, as demonstrated by the data in Table 4, because the subgoals to be attempted are simplier and the number of subgoals will be smaller. This is the primary reason for the improvements. For example, the prover finds a rather large proof for wos10 involving the equality axioms (see table 3) when *maxsize* is not set at search bound 7. When *maxsize* is set, the prover finds a simple proof which only uses the associativity axioms at search bound 9. The subgoals in this proof are much

smaller. Although the search bound is increased, the smaller number of subgoals makes the prover find the proof more quickly. Other examples are wos13, wos14, wos8, wos6, ls28 and ls29. These examples all have proofs which are not long and have small subgoals. All these examples indicate the advantage of the global control over the complexity of subgoals, at least for the problems with short proofs whose subgoals are also small.

It often helps for problems having short proofs with small subgoals to set *maxsize*. What about those problems whose proofs are long or whose proofs are short but contain some large subgoals? One could not help to notice the dramatic degeneration in the performance of the prover when *maxsize* is set on some examples, such as group1, ls23, ls55 and wos1. What is common about these problems is that the proofs for them are short and contain some subgoals whose sizes are large relative to the length of the proofs. The input clauses for group1 and its proof is given below to illustrate what we mean. The proof has only 3 levels of recursion. But the subgoals in the proof are large.

```
p(g(X,Y), X, Y).
p(X, h(X,Y), Y).
p(X, Y, f(X,Y)).
p(U, Z, W) :- p(X, Y, U), p(Y, Z, V), p(X, V, W).
p(X, V, W) :- p(X, Y, U), p(Y,Z, V), p(U, Z, W).
false :- p(j(X), X, j(X)).

false:-
        p(j(h(X,X)),h(X,X),j(h(X,X))) :-
                p(g(X,j(h(X,X))),X,j(h(X,X)))
                p(X,h(X,X),X)
                p(g(X,j(h(X,X))),X,j(h(X,X)))
```

When *maxsize* is set, the prover is forced to go deeper into the search space. The larger expansion of the search space, which is the result of the larger search bound, offsets the benefits of a smaller number of subgoals for these examples. This is more serious for the problems with a large number of input clauses such as wos1, for which the prover even fails to obtain any proof when *maxsize* is set. Although it is inevitable for the prover to search deeper into the search space when *maxsize* is set, the degeneration of performance of the prover for these examples seems to have something to do with the fact that we are using the search bound as the size bound for the subgoals. A question comes up naturally concerning this approach. Why are we using the search bound as the subgoal size bound while they represent different dimensions of

the search space? It is up to the nature of each individual problem to decide what is the proper size of the subgoals at each search bound. What is more suitable may perhaps be to determine the goal size limit from the input clauses and the intermediate solutions of a problem.

5.3. To charge for using old solutions

Any solution is derived with a certain amount of "effort" invested. When an old solution is used, how much effort we charge for it will affect the future expansion of the search space significantly. Before we can elaborate on this, we first briefly explain how the "cost" of the input clauses and the solutions affect the expansion of the search space.

The use of each input clause and solution has a certain cost associated with it. As the prover searches down along any branch of the search tree, the total cost accumulates until it exceeds the search bound. At that point, the search along that branch will be cut off and the prover backtracks to search the remaining branches, if any. In the input, an input clause of the form $L :- L_1, L_2, \ldots, L_n$ has cost n; an input clause of the form $L :- - L_1, L_2, \ldots, L_n$ has cost 0; an input clause of the form $L :-. L_1, L_2, \ldots, L_n$ (note the period at the end) has cost 1. An old solution also has a certain cost associated with it. How to calculate the cost of a solution when it is used will be discussed at length later. During the search process, whenever an input clause is used, its cost is added to the cost accumulated so far, and the total cost must not exceed the search bound. Similarly, whenever an old solution is used during the search, its cost is also added to the cost accumulated so far and the total must not exceed the search bound. When a new solution is derived, the cost accumulated to generate this solution is recorded together with the solution itself. This cost represents roughly the effort invested in deriving this solution. We will call this piece of information **proof-size** since it can be interpreted as the length of the proof to derive this solution.

It is easy to see that how much we charge for using an old solution will greatly affect the expansion of the search space. Not only does it affect the current possibility of search space expansion along the current branch, it also affects the cost of the new solutions that may be generated later on this branch. It is not hard to appreciate the complexity of this matter. The important question to ask is how the cost of an old

solution is calculated. Two important attributes of the solutions seem to be important. The first attribute is how much effort has been invested in deriving a particular solution. This piece of information is recorded with the solution and will be referred to as the **proof-size** of the solution. For those problems whose proofs are long and whose search space has a small branching factor, charging less for the proof-size makes the prover go deep into the search space to find the proof more quickly; it will enable the prover to get a rather long proof within a small search bound. The second attribute is the size of a particular solution, which we will call the **solution-size** of the solution. As we mentioned before, we need some method to guide the prover so that the smaller subgoals are favored. By charging some for the solution size, searching along the branch having large subgoals will be stopped earlier to favor other branches having smaller subgoals.

Based upon these considerations, we provide two parameters, **proof_size_multiplier** and **solution_size_multiplier**. We will refer to them as **psm** and **ssm** respectively in what follows. The user can adjust these two parameters depending on the nature and characteristics of the problems they are working on. The cost of a solution is calculated using the following formula:

$$Cost\_of\_Solution = solution\_size\_multiplier \times solution\_size + proof\_size\_multiplier \times proof\_size$$

Table 5 gives the statistics for the performance of the prover when these two parameters are set to different values. The blank entries indicate the prover's failure to obtain the proof for the corresponding problem under the indicated setting. The default prover uses 0.1 as ssm and 0.4 as psm. Set1 uses 0 as ssm and 1 as psm. Set2 uses 0.125 as ssm and 0 as psm. These sets of values, by the way, work well from our experience.

We can see that the prover performs reasonably well when only considering the proof size of the old solutions while disregarding the solution size. But by only considering the solution size while disregarding the proof size, the prover performs poorly and even fails to obtain some proofs. It is interesting to note that, sprfn with default setup performs poorly on those problems on which it performs well when it uses set1. The examples are ls28, ls29, ls65, ls75, wos13, wos14, wos6, wos7, wos8 and wos9. It is instructive to explain this. The fact that sprfn performs better when only considering the proof size suggests that, for these problems, the solution size does not play important roles in controlling the search space. What it common about these problems is that they all have relatively short proofs and the size of the subgoals in the

proofs are not very large either. For this kind of problem, it is a good idea to be able to search through the search space for the smaller search bound as soon as possible. At smaller search bounds, the proof size is more important than the solution size since the subgoals do not get very large anyway. The default prover seems to have undercharged the proof size, thus leading the search astray. It seems that by only considering the proof size, the prover would be able to find the short proofs more quickly than if it considers the solution size at the same time. The default prover performs better on wos1 and wos11 than it does when using set1. The reason is that, to prove these theorems, some big subgoals are generated during the search. By setting ssm to 0, the prover has no way to stop this quick expansion of search space due to the large number of big subgoals. By setting ssm to non-zero, the quick expansion can be better controlled. Table 5 also shows that the solution size alone is not sufficient for controlling the search space expansion. The prover rarely performs better when using set2, except in the case of wos11 which seems to be an peculiar case. The prover seems to be very sensitive to any control over the size of the subgoals when trying to prove wos11. The proof for wos11 is long and the subgoals tend to get big.

6. Forward Chaining and Backward Chaining

Sprfn performs true Prolog-style backward chaining. By only performing backward chaining, sprfn realizes a complete search procedure. Nevertheless, combination of forward chaining and backward chaining in one theorem prover may help. With due considerations to the fact that forward chaining is a fundamental and widely used problem solving strategy, lack of it may sometimes be a disadvantage. In many cases, we find out that forward chaining is useful. One earlier version of sprfn was implemented to perform only limited forward chaining. The number of inferences was limited; the level of recursion during the forward chaining phases was limited to one, i.e., the prover only derived the goals that could be decomposed into the subgoals already solved; and forward chaining was performed only at the early stage of the proof. We first doubled the allowable inferences for the forward chaining phases, just to get an idea how much impact forward chaining might have. This simple change made a big difference in prover's performance on one problem. The prover took just over 100 seconds to prove some problem for which it used to take over 1,000 seconds. For the later version of the prover where forward chaining and backward chaining are both performed during the whole course of proofs, the prover's performance is greatly affected by the presence

of forward chaining, as shown by some data in Table 3 (wos10, wos11). Although these two examples are not representative of the problems sprfn may be presented to, it does suggest a class of problems in which forward chaining may be helpful; i.e., those problems whose proofs may be long and require the prover to go deep into the search space to obtain them.

The experiments we conducted convince us that it is going to be useful to combine backward chaining and forward chaining in the prover. This combination is certainly going to have great impact on the prover's performance. What we need is some method to combine backward chaining and forward chaining in such a manner so that they will participate in the whole course of proofs and their participation will help the prover's performance. We ask two related questions. First, what is the best way to combine forward chaining and backward chaining in the prover so that both forward chaining and backward chaining will contribute during the whole course of the proofs. Second, how can we control forward chaining and backward chaining so that a kind of balance will be achieved. The meaning of these questions will be better understood from the following discussion.

6.1. Alternation between Forward and Backward Chaining

The deduction system underlying sprfn is a goal-oriented system. To use this deduction system, one starts with the top-level goal. If the confirmation of the top-level goal requires the confirmation of several other goals, these will be attempted one by one, in the same manner as the top-level goal is being attempted. This is the essence of backward chaining. We have mentioned that the prover exploits depth-first iterative deepening search. Whenever a new search bound is established in the search process, it is natural for the prover to start with the top-level goal using backward chaining. If the backward chaining phase fails to confirm the top-level goal, there are two options for the prover. One option is to restart the process of confirming the top-level goal with an increased search bound. Another option is to derive some facts from the known facts before restarting. To adopt the first option, the prover will be performing pure backward chaining. The second option suggests a natural method to combine forward chaining and backward chaining. To adopt the second option, the prover will derive some facts from the known facts before attempting to achieve the top-level goal again. This may sometimes be helpful. Since the failure of the

prover to achieve the top-level goal in the backward chaining phase is due to the fact that the proof for the top-level goal is too long for the current search bound; and forward chaining is not limited by the distance between the subgoals and the top-level goal in the search space, some facts that are generated during forward chaining, useful for achieving the top-level goal , may be too far from the top-level goal in the search space to be reached by backward chaining phase with small search bound.

Sprfn was subsequently changed to incorporate the above idea. During the modification, we introduced two flags, **b_only** and **f_only**, which direct the prover to do either only forward chaining or backward chaining. Here, we are giving user the flexibility to choose the two options mentioned above. If the **b_only** flag is not set, the prover will alternate between backward chaining and forward chaining phases. As before, it derives all the goals that can be derived by only one level of recursion during the forward chaining phases. By restricting the recursion level to one, forward chaining will be controlled so that not many facts would be generated. We direct readers' attention to table 3 to see the effect of forward chaining.

6.2. Balance between Forward and Backward Chaining

Two questions are raised at the beginning of this section. The first question concerns how to combine properly forward chaining and backward chaining. We have discussed this question. Now the remaining question is how to balance between backward chaining and forward chaining.

The prover is in danger of the uncontrolled expansion of the search space. This danger is present both in the forward chaining phases and backward chaining phases. Forward chaining runs out of control more easily since the prover just derives all the facts it can from the known facts. By restricting the level of recursion to one during forward chaining, we are eliminating this danger to some extent. However, experience shows that forward chaining still generates a large number of useless facts, leading the prover astray. To further eliminate this danger, we experimented with other ideas. A lot of experiments were conducted on this topic. We have to admit that better ideas are to be conceived. The following is a summary of what we have done and the results we have obtained.

The basic objective of our research is to balance and control forward chaining and backward chaining so that neither of them get out of control. Our basic idea is summerized as follows. Sprfn alternates between forward chaining and backward chaining. During each phase, as backward chaining or forward chaining proceeds, the prover monitors the expansion of the search space. When the search space gets larger, the prover will gradually decrease the size limit for the subgoals; thus the expansion of the search space will be gradually stopped. The goal size limit function is a linear function of the "size" of the search space. It also uses the current search bound. The important issue is how to measure the "size" of the search space.

Use inference count. This experiment uses the inference count to measure the size of the search space. It is based on the following considerations. First, it may help the prover's performance to set a larger goal size limit relative to search bound when the search bound is small. The subgoals are not likely to be very large when the search bound is small; and it provides a better chance of finding a proof within a small search bound to allow larger subgoals. Second, it seems to be reasonable to maintain a larger and constant goal size limit for backward chaining. Third, we should enforce more control on forward chaining since forward chaining can get out of control more easily. To control forward chaining, we set the goal size limit to be large at the early stage of forward chaining and, as forward chaining proceeds, gradually reduce the size limit to a constant fraction of the search bound. We formulated the following formula to calculate the goal size limit.

$$SizeLimit = \begin{cases} 2 \times SIZE & \text{if } backward\ chaining \\ 2 \times SIZE & \text{if } f\_count < 50 \\ 0.5 \times SIZE & \text{if } f\_count > 150 \\ \frac{11}{4} - f\_count \times \frac{3}{200} & otherwise \end{cases}$$

where **SIZE** is the current search bound; **f_count** is inference count for forward chaining. 50 and 150 are chosen since the prover does not require very many forward chaining inferences to obtain the proofs for easy problems. This function has the effect of favoring backward chaining when the prover proves hard problems.

Use inference count revisited. In the previous experiment, we are not considering the expansion of the search space based on the current search bound. This is because we are using the total inferences performed in the forward chaining phases to determine the goal size limit. In this experiment, we try to remedy this. Still using the number of inferences to measure the size of the search space, we control the expansion of the current search space based on the size of the previous one. We use two similar formulae to calculate the goal size limit for backward chaining and forward chaining. For forward chaining, we use:

$$SizeLimit = \begin{cases} SIZE \times F_1 & \text{if } F < F_a \\ SIZE \times F_2 & \text{if } F > F_b \\ SIZE \times \left(\frac{F \times (F_1 - F_2)}{F_a - F_b} + \frac{F_2 \times F_a - F_1 \times F_b}{F_a - F_b} \right) & \text{otherwise} \end{cases}$$

where F_a, F_b, F_1, F_2 are constants; F is the ratio between the number of inferences performed since this forward chaining phase starts and the number of inferences performed in the previous forward chaining phase. For backward chaining, we use:

$$SizeLimit = \begin{cases} SIZE \times B_1 & \text{if } B < B_a \\ SIZE \times B_2 & \text{if } B > B_b \\ SIZE \times \left(\frac{B \times (B_1 - B_2)}{B_a - B_b} + \frac{B_2 \times B_a - B_1 \times B_b}{B_a - B_b} \right) & \text{otherwise} \end{cases}$$

where B_a, B_b, B_1, B_2 are constants; B is the ratio between the number of inferences performed since this backward chaining phase starts and the number of inferences performed in the previous backward chaining phase. By setting the four parameters to different values, we can have different functions.

Use number of solutions. It seems that the number of the solutions is a better indicator of the expansion of the search space. One reason is that a larger number of solutions indicates wider expansion of the search space. Large expansion of the search space is signified if there are a large number of solutions generated either during the forward chaining or backward chaining phase. Some control should be enforced to limit it. Another reason is that the old solutions will be used later during the course of the proof. A larger number of solutions indicates a greater possibility to have larger expansion at the later phases. Hence, it seems to be reasonable to control the search space based on the number of solutions generated. In this experiment, we control the goal size depending on the number of new solutions generated in this phase.

$$Sizelimit = \begin{cases} SIZE & \text{if } S > P \\ SIZE \times (2 - \frac{S}{P}) & otherwise \end{cases}$$

where S is the number of solutions generated so far since the current forward chaining phase or backward chaining phase starts; P is $maximum(20, 0.5 \times N)$ where N is the number of solutions already generated when the current forward chaining phase or backward chaining phase starts; SIZE is the current search bound.

6.3. Overall Comments

We modified the prover to implement the above ideas. Although the prover does extremely well for a few problems, more often than not, it fails to get the proofs for some problems. Closer examinations reveal that all the methods above seem to discriminate some parts of the search space. By examining the trace of the proofs, we discover that the prover proceeds normally at the beginning of each phase. After some point, a large number of subgoals are rejected because the goal size limit has been decreased. This is the scenario for all the methods above. Because of the Prolog-style depth-first search sprfn performs and the predetermined order of the input clauses, some parts of the search space will be explored by the prover with smaller goal size limits. This explains the unsteady performance of the prover. For some problems, the prover can find proofs without searching those parts of the search space that would have been searched with smaller goal size limits. For some problems, the prover always search some parts of the search space with smaller goal size limits and these parts happen to be relevant for the proof. The prover is forced to search deeper into the relevant parts of search space, while wasting a lot of effort search the irrelevant parts with large goal size limits. We think a carefully chosen constant goal size limit will help the most in achieving reasonably good and constant performance.

7. Conclusion

Sprfn turns out to be a quite respectable theorem prover. The compactness of the prover and the ease of understanding it enable us to modify the prover easily to test different ideas. The prover can get many

relatively non-trivial theorems using the default setup. For harder problems, however, the user may need to adjust the parameters and set the flags based on the characteristics of the problem to obtain the proofs. Some important characteristics of problems are the possible length of the proof, the possible branching factor of the search space, the relative importance of the input clauses and subgoals, the distribution of the subgoals in the search space, etc. To nobody's surprise, the prover still fails to obtain proofs for some hard problems.

During our research on the prover, we have appreciated the importance of "syntactic intelligence" in theorem proving, i.e., taking into consideration the syntactic properties of the problems such as the size of subgoals, the number of variables and the number of function symbols in the subgoals. We have achieved the most dramatic improvements by adding some simple syntactic refinements like *maxsize* flag into the prover. The parameter, **solution_size_multiplier**, is another example. In another extensive research, we investigated the effect of dynamically reordering the subgoals during the proof process based on these syntactic characteristics of subgoals on the performance of the prover. We have obtained some interesting results. However, we will not have room to detail this research in this report.

More complex refinements, on the other hand, have failed to offer the improvements we have hoped. The research described in last section is an example. The lack of success in that research is due to the fact that the nature of the problems varies. Some of them can be solved more efficiently by backward chaining; some, by forward chaining. There are probably as many patterns of search space expansion as there are problems. Any scheme to control and balance forward chaining and backward chaining based on anything less than the properties of individual problem is not likely to offer any improvements on a large scale. That is the lesson we have learned.

We will briefly mention our current and future work on sprfn. At present, we are working on including semantics into the prover. Although some preliminary results have been obtained, more substantial work needs to be done in this area. We feel that we need a better priority scheme for the subgoals and the solutions in the prover. At present, the prover uses the old solutions in the order they are stored. What we need is some methods to identify the more promising solutions and prefer those solutions to others. Better

methods to use forward chaining are also one of the research topics. We are also looking for possible applications for the prover.

References

[1] Plaisted, D.A. "A Simplified Problem Reduction Format" Artificial Intelligence, 18 (1982) pp. 227-261

[2] Plaisted, D.A. "Non-Horn Clause Logic Programming without Contrapositives"

[3] Stickel, M.E <<A PROLOG Technology Theorem Prover: Implementation by an Extended PROLOG Compiler>> Proceedings of the Eight International Conference in Automated Deduction, Oxford, England, July 1986, pp. 573-587

[4] Stickel, M.E & Tyson, W.M. <<An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction>> IJCAI 1983, 1073-1075

[5] Loveland, D.W. <<Automated Theorem Proving: A Logical Base>> Chapter 6. North-Holland Publishing Company, 1978

[6] Plaisted, D.A <<Another Extension of Horn Clause Logic Programming to Non-horn Clauses>> Lecture Notes, 1987

[7] Chang, C & Lee, R << Symbolic Logic and Mechanical Theorem Proving>> Academic Press, New York, 1973

Appendix A -- Prolog code for the third implementation of discrimination Net

```
% top-level call to do insertion
%
dn_insert((Term, Tag)) :-
        copy(Term, Term1), numbervars(Term1,1, _),
        struct(Term, List), struct(Term1, List1), in1(List1, List, Tag).

% top-level call to search a term.
%
dn_search((Term, Tag)) :-
        struct(Term, List), se1(List, Tag).

% top-level call to initialize the discrimination net.
%
dn_init :-
        clean_up(1),
        (retract(nodeid(_)); true), !, assert(nodeid(2)),
        assert((in1([GL|GB], [L|B], T) :-
                        retract(nodeid(M)),
                        assembly(se, 1, [L|A], T1, S1),     % search clause
                        assembly(se, M, A, T1, S2), assert((S1 :- S2)),
                        assembly(in, M, GC, C, T1, S),      % insert clause
                        asserta((in1([GL1|GC], [L1|C], T1) :-
                                        GL1 == GL, !, S
                        )),
                        insert(GB, B, T, M)
                )
        ).

set_up(M) :-
        assembly(in, M, [GL|GB], [L|B], T, S1),
        assembly(se, M, [L|A], T1, S2),
        assert((S1 :- retract(nodeid(M1)),
                        assembly(se, M1, A, T1, S3),
                        assert((S2 :- S3)),
                        assembly(in, M, [GL1|GC], [L1|C], T2, S5),
                        assembly(in, M1, GC, C, T2, S4),
                        asserta((S5 :- GL1 == GL, !, S4)),
                        insert(GB, B, T, M1)
                )).

insert([GL|GB], [L|B], T, M) :-
        set_up(M), M1 is M+1,
        assembly(se, M, [L|A], T1, S1),
        assembly(se, M1, A, T1, S2), assert((S1 :- S2)),
        assembly(in, M, [GL1|GB1], [L1|B1], T2, S3),
        assembly(in, M1, GB1, B1, T2, S4),
        asserta(( S3 :- GL1 == GL, !, S4)),
        insert(GB, B, T, M1).

insert([], [], T, M) :-
        set_up(M), M1 is M+1, assert(nodeid(M1)),
        assembly(in, M, [], [], T1, S1),
        assembly(se, M, [], T1, S2),
```

```
        asserta((S1 :- assert(S2))),
        assembly(se, M, [], T, S3), assert(S3).

clean_up(M) :-
        nodeid(N), M < N,
        name(se, L1),name(M, L2),append(L1, L2, L3), name(S, L3), abolish(S,2),
        name(in, J1),name(M, J2),append(J1, J2, J3), name(I, J3), abolish(I,3),
        M1 is M + 1, clean_up(M1).
clean_up(_).

struct((L :- B), [L|B]).

% utility functions.
%
% yield a ground instance of a term.
%
numbervars('$VAR'(L), L, M) :- !,
        M is L+1.
numbervars(Term, K, M) :-
        functor(Term, _, N),
        numbervars(0, N, Term, K, M).

numbervars(N, N, Term, M, M) :- !.
numbervars(I, N, Term, K, M) :-
        J is I+1, arg(J, Term, Arg),
        numbervars(Arg, K, L), !,
        numbervars(J, N, Term, L, M).

% make a copy of a term
%
copy(Old, New) :-
        asserta(copy(Old)),
        retract(copy(Mid)), !,
        New = Mid.

assembly(Pre, Id, A1, A2, A3, T) :-
        name(Pre, L1), name(Id, L2), append(L1, L2, L3), name(F, L3),
        functor(T, F, 3),
        arg(1, T, A1), arg(2, T, A2), arg(3, T, A3).

assembly(Pre, Id, A1, A2, T) :-
        name(Pre, L1), name(Id, L2), append(L1, L2, L3), name(F, L3),
        functor(T, F, 2),
        arg(1, T, A1), arg(2, T, A2).
```

Appendix B -- The Performance Statistics

Five tables are given in this section. These tables contains the performance data of the prover using the different parameters, flags and data structures. Most of the problems are from [3]. Included are also the 9 problems form [8]. Three different versions of the prover are used to obtain these data. Although the most recent version does not give the best performance for all the problems, it does obtain proofs for a larger number of problems. The two earlier versions have failed to do so. Data in Table 1 are obtained using the earliest version of sprfn where only a small amount of forward chaining was performed. Data in Table 2 are obtained using a more recent version, with the *maxsize* flag set. It should be pointed out that the *maxsize* flag contributed the most to the beautiful performance of this version of the prover. Data in other tables are obtained using the most recent version of the prover. The machines on which tests are run are indicated in the tables.

Table 1. Performance for Different Discrimination Net (Cputime on VAX-780)

Theorem	No disc. net	Version I	Version II	Version III
ances1	8.99	16.20	14.88	13.22
burstall	15.98	31.02	29.52	24.45
dbabhp	1300.72	1796.95	1688.37	3755.85
dm	1.10	1.25	1.30	1.22
ew1	2.40	3.53	3.37	3.02
ew2	1.61	2.40	2.35	2.07
ew3	4.20	8.43	7.55	5.68
example	7776.20	9368.38	8922.40	3877.95
group1	2.55	4.28	3.98	3.00
group2	10.22	16.07	14.78	12.17
hasparts1	4.64	8.91	8.35	6.33
hasparts2	10.18	16.65	16.80	13.13
ls100	1.03	1.28	1.23	1.27
ls103	89.66	248.16	207.28	138.85
ls105	1.72	2.52	2.40	2.13
ls106	1.72	2.50	2.40	2.18
ls111	1.72	2.40	2.40	2.13
ls17	19.65	35.28	33.45	25.25
ls23	56.52	99.18	89.73	61.75
ls26	23.50	28.00	26.27	25.45
ls28	1103.82	1123.63	1115.90	1101.85
ls35	20.24	30.01	26.08	23.00
ls41	14.78	26.97	24.75	21.53
ls5	2.12	3.15	3.00	2.72
ls55	42.18	68.42	60.55	54.57
ls68	38.42	63.25	56.43	50.25
mqw	3.37	6.10	5.83	4.12
num1	4.39	6.70	6.23	5.50
prim	6.57	10.56	10.32	8.48
qw	2.77	4.35	4.40	3.82
rob1	1.12	1.35	1.35	1.43
rob2	10.55	17.57	16.07	13.60
schubert.abst	677.10	1165.22	1067.50	897.60
shortburst	7.37	14.55	13.12	10.52
wos10	1645.55	2319.13	1930.50	1801.30
wos12	4.75	6.82	6.85	5.92
wos13	49.73	98.93	87.55	81.25
wos14	42.22	68.67	60.97	50.30
wos3	1.63	1.87	1.90	1.83
wos6	2909.57	3662.00	3233.61	2802.83
wos7	468.07	577.67	528.83	500.47
wos8	82.38	97.28	96.26	90.35

Average slowdown with respect to prover without discrimination net:

Version 1: 56.17%. Version 2: 45.98%. Version 3: 27.88%.

Table 2. Performance data with clause elimination (maxsize is set).

Theorem	Without Clause El.		With Clause El.	
	cpu time(VAX)	Cached Goals	cpu time(VAX)	Cached Goals
ances1	16.50	53	17.30	24
burstall	8.35	27	9.07	6
dbabhp	8513.74	144	8591.78	107
dm	8.33	8	8.77	4
ew1	2.77	9	2.95	5
ew2	3.02	10	3.47	5
ew3	6.40	19	7.33	3
cl1	11.73	8	13.00	4
cl2	11.50	12	12.45	3
cl3	6.20	10	6.62	4
cl4	6.12	9	6.60	4
cl5	0.67	0	0.72	0
cl6	6.93	6	7.97	3
cl7	4.20	11	4.57	5
cl8	13.62	27	14.65	6
cl9	10.83	26	11.88	12
example	55.03	108	55.93	44
group1	64.12	9	70.50	5
group2	11.23	12	11.95	3
hasparts1	4.67	10	4.45	4
hasparts2	15.75	32	16.02	7
ls100	1.12	2	1.18	2
ls103	33.38	63	33.52	36
ls105	1.90	5	1.90	5
ls106	1.75	4	1.77	4
ls111	1.90	5	1.95	5
ls17	18.02	29	18.73	10
ls23	161.59	16	169.27	9
ls26	8.05	6	8.34	3
ls28	86.28	13	86.88	6
ls35	78.05	7	83.45	3
ls41	4.12	5	4.27	3
ls5	2.12	6	2.15	3
ls55	70.22	25	71.72	11
ls68	45.12	21	49.08	11
mqw	2.53	8	2.72	3
num1	4.17	11	4.53	5
prim	11.88	18	11.85	3
qw	2.83	7	2.90	4
rob1	3.67	9	4.07	3
rob2	12.23	12	13.33	3
schubert.abst	237.56	285	238.47	109
shortburst	3.92	16	4.20	13
wos1	32.52	14	34.53	8
wos10	101.13	390	105.45	7
wos12	3.88	102	4.15	10
wos13	5.78	7	6.18	3
wos14	40.03	23	42.00	8
wos2	16.07	24	16.67	6
wos3	1.57	3	1.85	3
wos6	450.07	33	465.48	12
wos7	168.97	52	172.10	15
wos8	10.72	15	10.80	5

Average slowdown with clause elimination: 5.68%.

Average ratio between the numbers of cached goals: 0.46.

Table 3. Performance data with b_only flag set

Theorem	default prover			prover with b_only set		
	cputime(SUN3)	inference	storage(Kb)	cputime(SUN3)	inference	storage(Kb)
ancestl	12.47	27	87.0	8.57	21	87.6
burstall	7.08	63	91.7	9.92	36	94.4
dbabhp	26.45	190	99.7	118.05	373	144.9
dm	1.13	6	85.3	1.15	6	85.3
ew1	1.92	6	85.6	1.43	6	85.5
ew2	1.28	5	84.3	.95	4	84.2
ew3	4.62	15	86.3	2.60	7	86.5
cl1	1.17	6	85.7	1.17	6	85.7
cl2	19.30	259	86.2	27.10	288	88.4
cl3	3.38	29	85.2	2.97	22	86.2
cl4	3.47	30	85.4	3.03	23	86.3
cl5	.52	4	85.6	.52	4	85.6
cl6	14.20	157	88.0	1.53	8	86.9
cl7	2.57	16	86.6	1.60	5	86.5
cl8	9.60	64	89.3	5.93	29	90.2
cl9	12.80	40	89.1	6.28	16	88.3
example	45.84	236	99.3	46.08	235	100.3
fex4t2	850.62	1002	129.9	2930.18	1478	266.8
group1	1.17	6	86.1	1.15	6	86.1
group2	19.30	259	86.1	27.12	288	88.3
hasparts1	4.87	28	87.3	1.32	7	86.9
hasparts2	9.40	54	88.9	3.87	15	89.2
ls100	.72	4	86.1	.70	4	86.1
ls103	30.50	131	97.7	121.85	371	114.2
ls105	1.03	5	89.7	1.10	5	89.7
ls106	1.10	5	89.7	1.15	5	89.8
ls111	1.10	5	89.8	1.12	5	89.9
ls115	53.53	207	102.2	40.23	110	104.2
ls17	16.60	76	91.9	6.90	26	90.5
ls23	41.58	236	91.8	52.57	253	93.9
ls26	34.67	199	95.6	1.38	7	87.7
ls28	715.22	1117	214.0	33.95	155	100.7
ls29	577.93	941	209.4	168.73	453	125.3
ls35	41.88	336	88.1	20.80	112	89.0
ls41	12.98	80	91.5	12.92	80	91.5
ls5	1.95	5	85.7	1.57	5	85.6
ls55	50.72	369	102.8	50.42	369	102.9
ls68	48.05	291	101.7	47.80	291	101.7
ls75	1076.95	3048	170.2	2124.10	5082	234.0
mqw	2.18	5	86.2	1.67	5	86.1
num1	2.52	17	86.0	1.53	6	85.9
ptm	5.20	32	86.5	5.53	24	89.3
qw	3.25	11	85.2	2.17	11	84.8
rob1	1.08	3	84.1	1.85	3	85.6
rob2	19.68	259	86.6	24.33	253	88.4
schubert.abst	203.02	953	121.5	169.52	474	132.3
shortburst	3.55	24	86.5	1.98	7	86.6
wos1	169.17	683	117.3	165.09	571	116.3
wos10	821.92	2878	165.0	16776.80	20898	974.6
wos11	1502.87	4901	198.9	1850.63	7386	181.8
wos12	3.18	36	91.9	3.20	36	92.0
wos13	1765.88	4084	337.6	55.37	362	116.7
wos14	1698.02	4104	333.6	164.17	687	117.0
wos2	72.68	443	105.6	30.72	231	94.8
wos3	1.22	9	90.3	1.23	9	90.3
wos6	5546.93	7896	551.8	75.70	671	98.3
wos7	333.65	1431	117.3	1273.50	5086	166.7
wos8	1696.70	4120	332.0	13.57	115	94.8
wos9	388.42	1676	149.5	296.57	1671	121.4
ls65	not found			132.85	702	111.7

Three average ratios between the performance data of the default prover and the performance data of the prover with b_only set: for cputime, 1.33; for inference, 0.95; and for memory, 1.06. When b_only is set, the prover performs better on 31 out of 59 problems with the average speedup of 53.8%; it performs worse on 10 out of 59 problems with the average slowdown of 135.6%; it performs the same as the default prover does on the remaining 18 problems.

Table 4. Performance data with maxsize flag set

Theorem	default prover			prover with maxsize set		
	cputime(SUN3)	inference	proof size	cputime(SUN3)	inference	proof size
ances1	12.47	27	18	14.63	27	18
bunstall	7.08	63	7	7.22	50	7
dbabhp	26.45	190	9	59.57	317	14
dm	1.13	6	5	88.22	225	14
ew1	1.92	6	7	2.15	6	7
ew2	1.28	5	7	1.45	5	7
ew3	4.62	15	11	5.23	15	11
cl1	1.17	6	5	55.98	250	11
cl2	19.30	259	9	12.87	127	9
cl3	3.38	29	7	5.55	28	9
cl4	3.47	30	7	5.63	29	9
cl5	.52	4	5	.65	4	5
cl6	14.20	157	7	5.00	32	9
cl7	2.57	16	7	7.07	28	11
cl8	9.60	64	11	11.72	64	11
cl9	12.80	40	9	14.48	31	9
example	45.84	236	14	55.40	236	14
fex412	850.62	1002	18	349.53	436	14
group1	1.17	6	5	199.12	558	11
group2	19.30	259	9	12.98	127	9
hasparts1	4.87	28	9	3.65	20	9
hasparts2	9.40	54	11	13.92	72	18
ls100	.72	4	5	.82	4	5
ls103	30.50	131	14	30.96	97	14
ls105	1.03	5	5	1.33	5	5
ls106	1.10	5	5	1.22	5	5
ls111	1.10	5	5	1.32	5	5
ls115	53.53	207	11	47.72	134	11
ls17	16.60	76	9	17.75	60	9
ls23	41.58	236	7	884.67	1608	9
ls26	34.67	199	7	4.97	31	9
ls28	715.22	1117	7	77.25	252	14
ls29	577.93	941	7	69.00	237	14
ls35	41.88	336	9	95.42	786	11
ls41	12.98	80	5	3.92	31	5
ls5	1.95	5	7	2.17	5	7
ls55	50.72	369	5	5944.98	22884	9
ls68	48.05	291	5	52.67	339	7
ls75	1076.95	3048	7	399.93	1856	9
mqw	2.18	5	7	2.35	5	7
num1	2.52	17	7	6.65	30	11
prim	5.20	32	9	6.13	32	9
qw	3.25	11	9	3.73	11	9
rob1	1.08	3	7	3.72	8	14
rob2	19.68	259	9	13.68	132	9
schubert.abst	203.02	953	24	242.30	947	24
shortburst	3.55	24	7	3.65	19	7
wos1	169.17	683	7	1801.57	3032	9
wos10	821.92	2878	7	66.82	438	9
wos12	3.18	36	5	2.58	26	5
wos13	1765.88	4084	7	9.27	68	7
wos14	1698.02	4104	7	34.12	252	11
wos2	72.68	443	7	13.63	67	7
wos3	1.22	9	5	1.58	9	5
wos6	5546.93	7896	7	212.00	1188	9
wos7	333.65	1431	7	9354.96	21075	11
wos8	1696.70	4120	7	10.62	66	7
wos9	388.42	1676	7	2739.53	8571	9
ls65	not found			113.32	603	9
wos11	1502.87	4901	7	not found		

Three average ratios between the performance data of the default prover and the performance data of the prover with *maxsize* set: for cputime, 1.33; for inference, 5.30; and for proof size, 1.25. When *maxsize* is set, the prover performs better on 27 out of 58 problems with the average speedup of 51.1%; it performs worse on 15 out of 58 problems with the average slowdown of 1753.1%; it performs the same as the default prover does on the remaining 16 problems.

Table 5. Performance data for different parameters.

Theorem	Default(0.1,0.4)		Set1 (0, 1)		Set2 (0.125, 0)	
	Cputime(SUN3)	Inference	Cputime(SUN3)	Inference	Cputime(SUN3)	Inference
ances1	12.47	27	14.73	37	9.43	14
burstall	7.08	63	13.53	129	7.52	65
dbabhp	26.45	190	97.02	583	28.33	199
dm	1.13	6	1.15	6	2.72	14
ew1	1.92	6	1.90	6	1.92	6
ew2	1.28	5	1.28	5	1.28	5
ew3	4.62	15	4.58	15	4.58	15
cl1	1.17	6	1.12	6	5.47	35
cl2	19.30	259	38.85	508	8.72	123
cl3	3.38	29	3.30	29	3.27	29
cl4	3.47	30	3.45	30	3.45	30
cl5	.52	4	.52	4	.52	4
cl6	14.20	157	14.12	157	50.25	711
cl7	2.57	16	2.53	16	2.52	16
cl8	9.60	64	17.22	121	9.48	64
cl9	12.80	40	11.88	36	10.18	32
example	45.84	236	250.07	923	64.47	323
fex4t2	850.62	1002				
group1	1.17	6	1.15	6	19.53	90
group2	19.30	259	38.82	508	8.68	123
hasparts1	4.87	28	4.87	30	4.83	28
hasparts2	9.40	54	29.75	156	9.30	54
ls100	.72	4	.68	4	.70	4
ls103	30.50	131	23.82	94	31.82	147
ls105	1.03	5	1.10	5	1.13	5
ls106	1.10	5	1.10	5	1.12	5
ls111	1.10	5	1.03	5	1.10	5
ls115	53.53	207	48.72	157	57.85	239
ls17	16.60	76	29.08	137	16.83	78
ls23	41.58	236	37.35	202	56.10	321
ls26	34.67	199	34.38	199	324.40	1742
ls28	715.22	1117	195.75	469		
ls29	577.93	941	182.07	453		
ls35	41.88	336	39.93	332	70.12	876
ls41	12.98	80	13.12	73	9.95	64
ls5	1.95	5	1.90	5	1.90	5
ls55	50.72	369	44.27	272	56.57	465
ls65			927.50	3997		
ls68	48.05	291	44.03	238	41.32	270
ls75	1076.95	3048	552.77	1566	3038.50	5318
mqw	2.18	5	2.18	5	2.18	5
num1	2.52	17	2.48	17	2.47	17
prim	5.20	32	23.38	162	5.27	36
qw	3.25	11	7.63	40	3.20	11
rob1	1.08	3	1.05	3	1.07	3
rob2	19.68	259	39.22	508	9.65	133
schubert.abst	203.02	953	525.22	1710	223.50	977
shorthunt	3.55	24	3.68	25	3.45	26
wos1	169.17	683	17198.50	12192		
wos10	821.92	2878			8673.95	21081
wos11	1502.87	4901	5819.28	8924	642.87	4126
wos12	3.18	36	3.17	36	3.13	36
wos13	1765.88	4084	112.77	509		
wos14	1698.02	4104	117.17	540		
wos2	72.68	443	55.40	275	814.78	2947
wos3	1.22	9	1.18	9	1.18	9
wos6	5546.93	7896	190.23	781		
wos7	333.65	1431	131.70	746	6853.66	9516
wos8	1696.70	4120	122.67	569		
wos9	388.42	1676	243.60	932	1158.70	4392

No average data will be calculated because of the many cases of failure.