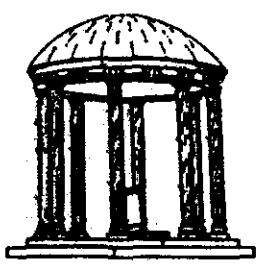
Refinements to Depth-first Iterative Deepening Search

TR89-004

January 1989

Xumin Nie
David A. Plaisted



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Refinements to Depth-first Iterative Deepening Search *

Xumin Nie *and* David A. Plaisted
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
Internet: {nie, plaisted}@cs.unc.edu

Abstract

This paper will discuss two refinements to the depth-first iterative deepening search strategy. The first refinement, the priority system, is an attempt to simulate best-first search using depth-first iterative deepening search. A new data structure, the priority list, is introduced into depth-first iterative deepening search by the refinement. Some complexity results about the priority system are also given. The second refinement is based on a syntactic viewpoint of proof development, which views the process of finding proofs as an incremental process of constructing instances with a certain property. We quantify this process to control the depth-first iterative deepening search. Both refinements have been implemented in a sequent-style back chaining theorem prover and tested on a large number of problems and have been shown to be effective.

Key Words and Phrases: search strategy, depth-first iterative deepening search, automatic deduction.

Length in words. 4500.

1. Introduction

The depth-first iterative deepening search strategy (DFID) has been the subject of some study [5, 11]. DFID involves repeatedly performing exhaustive depth-first search with increasing depth bounds. DFID and its variant, iterative-deepening-A* (IDA*), have some distinctive advantages. They require minimal memory to operate, as opposed to breadth-first search or A* search; they are complete strategies as opposed to pure depth-first search; they always find optimal solutions; and they are usually just as efficient as breadth-first or A* search in spite of the effort spent on repeated search.

DFID has been used recently by several researchers in theorem proving [1, 12]. In particular, it is used to implement Plaisted's modified problem reduction format [10]. In this paper, we will discuss two refinements to the DFID strategy based on Plaisted's implementation. The first refinement is an attempt to simulate best-first search using DFID in order to make the search concentrate on more important goals. A new data structure, a priority list, is introduced. The second refinement is based on the observation that the process of finding a proof is a process of incrementally constructing some ground instances of the input clauses. This incremental process can be quantified and used to control DFID.

* This work was supported in part by the National Science Foundation under grant DCR-8516243 and by the Office of Naval Research under grant N00014-86-K-0680.

2. First Refinement: Priority System

One of the essential features in any automatic theorem proving system is for its search strategy to have an estimation of the importance or relevance of its goals and always to work on the best available goal based on the estimation [9]. The best-first search strategy is the most commonly used search strategy to achieve this. In best-first search, a *priority function* is defined which assigns priority values to the goals. A goal queue, sometimes called the *open list*, is also maintained in best-first search, which contains all the unfinished goals together with their priority values. The search always selects the goal from the goal queue with the "best" priority value to attempt next. It has been demonstrated that it adds substantial power to a theorem prover to use the best-first search strategy [2, 7] if the priority function is suitably chosen. The brute-force DFID search as implemented in [10] does not have a good method of favoring and concentrating on the important goals. The problem we try to solve is to incorporate the use of priority into DFID. In the following discussion, we assume that a smaller priority value indicates a more important goal.

Suppose we have a priority function and we would like to use it in such a way that the final solutions only involves the goals with the smallest priority values possible. If we know that a solution exists and know the smallest bound B which allows the solution to be found, we can just set the priority bound to B and perform the depth-first iterative search until the solution is found, deleting the goals whose priority values are bigger than B. The problem is that we usually do not know the value of B unless we have found a solution. What we would like to do is to find the solution under some bound reasonably small compared to the smallest bound B. We call the resulting strategies the *priority systems*.

Let's consider the L smallest operations[1]. We do not know these operations in advance. However, at any moment of the search, we know the L smallest operations performed so far. Assume $C_1, C_2, \cdots C_L$ are the priority values of the L smallest operations performed so far. If an operation has a priority value P not smaller than the largest of $C_1, C_2, \cdots C_L$, we know that this operation will not be one of the L smallest operations. If an operation has a priority value P smaller than one of $C_1, C_2, \cdots C_L$, this operation may be among the L smallest operations and we should replace the largest value in $C_1, C_2, \cdots C_L$ with C. This is the basic idea behind the priority systems.

We propose the following search procedure. The search procedure will operate as the depth-first iterative deepening search does. We call the depth-first iterative deepening search the *underlying strategy*. The search procedure records the L smallest priority values among the operations performed so far in the *priority list* of length L. A priority list stores a sequence of priority values $C_1, C_2, \ldots, C_L$ in non-increasing order. The first L operations are performed and their priority values are put in the list. For each successive operation, we compute its priority value P. If P is less than C_1, the operation is performed and the priority list is updated by deleting C_1 and inserting P at the appropriate place. If P is greater or equal to C_1, the operation is rejected. Let's consider an example. Suppose that a priority list of length 5 is [10, 7, 6, 6, 3]. An operation with priority value 10 will be rejected. An operation with priority value 5 will be accepted and the priority list is updated to [7, 6, 6, 5, 3]. Note that each operation is either rejected or updates the priority list to a strictly smaller value. When no operation is possible, the priority values $C_1, C_2, \cdots C_L$ in the priority list are the L smallest operations performed so far. But this fact is not

[1] We use the word *operation* to refer an unit of work performed by the search process. It can mean a goal being generated, for example.

important. What is important is that the search procedure favors small operations more and more as the search proceeds.

We would like to point out that, if a solution is found by the priority system with the largest value in the priority list being B, B may not be the smallest possible bound. However, the search procedure has an interesting property. Given a problem P, suppose N operations are performed by the underlying strategy to find a solution S for P. Let B_0 be the smallest bound for the priority values which permits the solution S to be found. Let L_0 be the number of operations among the N operations whose priority values are less than B_0. We have

Theorem A: The above search procedure will find the solution S when the priority list is of length greater than or equal to L_0

when the procedure satisfies the *monotonicity condition*, which states

> for any bound B, if some or all of the operations with priority values ≥ B are deleted, the number of possible operations with priority value less than B will not increase.

Proof: At any given time after L_0 operations have been performed, the largest element in the priority list will be greater than or equal to B_0. Thus the number of operations with priority less than B_0 will be less than or equal to L_0 by the monotonicity condition. Since the priority list is of length ≥ L_0, all these operations will be performed and the proof will be found.

We will analyze this procedure. The question we ask is: Given that the length of the priority list is L, what is the number of operations possible. Suppose the first L operations have complexity $C_1, C_2, \ldots, C_L$ in non-increasing order and let $M_L = C_1$. Note that each operation is either rejected or changes the sequence to a strictly smaller value. With a priority list of length L, we can have a minimum of L operations (the first L operations to fill up the list) and a maximum of $M_L \times (L-1)$ operations after the first L operations, assuming a priority value is a natural number. We invoke successive trials of the procedure with the priority list being of length $L \times C^i$ ($i = 0,1,\cdots$ and $C > 1$) respectively, until a solution is found. Letting w(i) denote the amount of work when the priority list is of length $L \times C^i$, we easily have

$$C^i \times L \le w(i) \le L \times C^i \times M_{C^iL}$$

where M_{C^iL} (i=0,1, . . . ,) is the largest priority of the first $C^i \times L$ operations. We can derive

$$P_1(i) = \frac{w(i+1)}{w(i)} \le C \times M_{C^iL}.$$

Let I be the smallest i such that w(I) is enough to find a solution. We try to estimate the ratio

$$P_2(I) = \frac{\sum_{k=0}^{I} w(k)}{w(I)}$$

Note that $P_2(I)$ bounds the unnecessary work performed by this procedure.

$$\sum_{k=0}^{I} w(k) = \sum_{k=0}^{I} C^k \times L \times M_{C^kL} \le M_{C^IL} \times L \sum_{k=0}^{I} C_k = M_{C^IL} \times L \times \frac{C^{I+1}-1}{C-1} \le M_{C^IL} \times L \times \frac{C^{I+1}}{C-1}$$

Thus we have

$$\frac{C}{C-1} \le P_2(I) = \frac{\sum_{k=0}^{I} w(k)}{w(I)} \le \frac{C}{C-1} \times M_{C^L}$$

The analysis above is a worst-case analysis and the result is admittedly weak. Note that the complexity depends on the bound M_{C^L}, and we do not know what is their expected value. Theoretically it can be arbitrarily large. The fact that the complexity depends on M_{C^L} also presents a practical problem: At the beginning of the search, we do not have any control over the complexity of the operations. It should be interesting to give some probabilistic analysis. In practice, though, we believe that the values for $P_1(i)$ and $P_2(I)$ are almost constant with respect to C. Our experiments on Stickel's problems confirm this. Table 1 shows the average values for $p_1(i)$ and $p_2(I)$.

Table 1. Average Values for $P_1(i)$ and $P_2(I)$

	C = 2	C = 3	C = 4
$P_1(i)$	2.10	3.13	4.24
C/(C-1)	2.00	1.50	1.33
$P_2(I)$	2.88	1.98	1.93

We propose a modification of the previous search procedure. In this procedure, we define the priority of an operation to consist of possibly multiple units of work. The number of units of an operation is called the *weight* of the operation. As before, we assume the priority list is of some length L and is represented by a sequence of priority values $C_1, C_2, \ldots, C_L$ in non-increasing order. An operation with priority value C and weight W will be rejected if one of the first W values in the priority list is less than or equal to C. If the operation is not rejected, the priority list will be updated by inserting W copies of C into the priority list.

Let's determine how many operations can be performed when the priority list is of length L. This is, again, a worst-case analysis. The analysis is straight forward if we realize that it takes L/N operations of priority N to fill an empty priority list with value N. To perform the maximal number of operations, we should fill the list with the largest priority value possible first. So the first candidate is the operation with weight L, the second candidate is the operation with weight L–1, the third with weight L–2, etc. The last operations are those with weight 1. Therefore, the maximal number of operations is

$$\sum_{i=1}^{L} \frac{L}{i} = L \times \sum_{i=1}^{L} \frac{1}{i} \approx O(L \times \ln(L))$$

We invoke successive trials of the procedure with the priority list being of length $L \times C^i$ ($i = 0,1,\cdots$ and $C > 1$) respectively, until a solution is found. Letting v(i) denote the amount of work when the priority list is of length $L \times C^i$, we can show

$$Q_1(i) = \frac{v(i+1)}{v(i)} = \frac{C^{i+1} \times \ln(C^{i+1} \times L)}{C^i \times \ln(C^i \times L)} = C \times \frac{i \times \ln(C) + \ln(C) + \ln(L)}{i \times \ln(C) + \ln(L)} = C \times \left(1 + \frac{\ln(C)}{i \times \ln(C) + \ln(L)}\right) \le C \times \left(1 + \frac{1}{i}\right) \le 2C \quad (i > 0)$$

That is, the amounts of work for successive trials of the procedure increase only by a constant factor. And if a solution is found when the priority list is of length $C^i \times L$, we have

$$Q_2(I) = \frac{\sum_{k=0}^{I} C^k \times L \times \ln(C^k \times L)}{C^I \times L \times \ln(C^I \times L)} = \sum_{k=0}^{I} \frac{1}{C^{I-k}} \times \frac{\ln(C^k \times L)}{\ln(C^I \times L)} \leq \sum_{k=0}^{I} \frac{1}{C^k} \leq \frac{C}{C-1}$$

That is, the amount of work is dominated by the last trial, if the last trial performs the maximal number of operations before the solution is found. Table 2 shows the average values of $Q_1(i)$ and $Q_2(I)$ obtained from our experiment on Stickel's problem set.

Table 2. Average Values for $Q_1(i)$ and $Q_2(I)$

	C = 2	C = 3	C = 4
$Q_1(I)$	2.09	3.12	4.18
C/(C-1)	2.00	1.50	1.33
$Q_2(I)$	2.90	2.19	1.90

To ensure completeness, the implementation of the priority system consists of *stages*. Each stage consists of several round of depth-first search with increasing bounds. To be specific, given two natural numbers n_1 and n_2, we can have a sequence of integers, determined by n_1 and n_2, $n_1 = m_1, m_2, \cdots, m_k =< n_2$, and $m_i < m_{i+1}$ $(i = 1, \cdots, k-1)$. A simple example would be $n_1 = m_1, m_1 + 1, m_1 + 2, \ldots m_1 + k = n_2$. Let's use $\{n_1, n_2\}$ to denote the consecutive rounds of depth-first search with cut-off bound being $m_1, m_2, \cdots, m_k$. The control structure of the priority system can be seen as

stage 1	stage 2	stage 3	stage 4	.....	stage i	.....
$\{S_1, E_1\}$	$\{S_2, E_2\}$	$\{S_3, E_3\}$	$\{S_4, E_4\}$	.....	$\{S_i, E_i\}$	.....

Figure 1

where usually $S_i = S_j$ and $E_i < E_j$ $(i < j)$ and the priority list for stage i is $C^{i-1}L$ where C (> 1) and L are constants. We note that the monotonicity condition may be invalid in our implementations. We also note that the bounds P_1 (Q_1) and P_2 (Q_2) may not work unless $E_i = \infty$ and $S_i = 1$ for all i. Nevertheless they seem to work well in practice.

We have implemented many versions of the priority system. They differ in some technical details which will not be discussed here. Our experience with these implementations are positive. We only show one implementation and compare its performance with that of the underlying strategy on Stickel's problem set [12]. The priority function used measures the number of symbols in a goal. The effect is to favor smaller goals. We summarize the results in Table 3.

Table 3. Summary Data for a Priority System

	underlying strategy	priority system
Average Time Per Theorem	224.30	178.86
Average Inferences Per Theorem	1245	4038

We note that the priority system is faster and more reliable. It not only gives a 20 percent improvement on the average, it also solves 3 more problems. The much larger average number of

inferences for the priority system is due to the fact that inferences performed by the priority systems usually involve smaller solutions and subgoals and can be performed more quickly. We also mention that the priority system has been used to solve several problems from [13] which cannot be solved using the underlying strategy.

3. Second Refinement: Proof Complexity Measure

Many applications in deductive database, logic programming and theorem proving require to find some instances which satisfy a certain property. For example, a query p(X) to a database system directs the database system to find an instantiation x for X that makes p(x) a logic consequence of the facts in the database. In resolution-based theorem proving systems, proving a theorem is equivalent to finding a contradictory set of ground clauses which are instances of the general clauses from the negation of the theorem. Our second refinement is based on the observation that a search for proofs can be viewed as an incremental process of building up the required instances. This viewpoint is especially natural for a back chaining theorem proving system and can be used to control the search process.

Let's consider the problem reduction format in its purest form. One is given a conclusion G to be established and a set of assertions of the form $L :- L_1, L_2, \ldots, L_n$ (implications) or L (premises). An implication $L :- L_1, L_2, \ldots, L_n$ is understood to mean $L_1 \wedge \cdots \wedge L_n \supset L$. The L_i's are *antecedent* and L is the *consequent*. The top-level goal will be the conclusion G. To confirm a goal L, one begins with a search of the premises to see if any premise matches L. If there is such a premise, L is confirmed. Otherwise, the set of implications is searched and one implication whose consequent matches with L will be selected, if one exists. The antecedents in the implication will be considered as new goals to be confirmed, much in the same manner as L has been. Note that if L contains some logical variables, a match with a premise or the consequent of an implication will bind these variables with other structures through unification. In the context of search control, these bindings will increase the complexity of the goal L if a variable is bound to a non-variable term, considering a structure is more complex than a variable. We can quantify the increase in complexity from unifications and use it to control the search.

The modified problem reduction format [10] embodies similar structure. Letting solve(G, S, F, E) be the procedure to solve goal G with the effort bound E, starting bound S and F–S being the cost of solving G in depth-first fashion, the following prolog code for the input clause $L:-L_1,L_2,\ldots,L_n$ illustrates our implementation

```
solve(L0, S, F, E) :-
    match(L0, L, [L1, L2 , . . . , Ln], V, [V1, V2 , . . . , Vn]),
    E0 is S + clause_cost(L :- L1, L2 , . . . , Ln) + match_cost(L0, V),
    E0 ≤ E,
    F1 is E0 + match_cost(L1, V1), solve(L1, F1, E1 ,E),
    F2 is E0 + match_cost(L2, V2), solve(L2, F2, E1 ,E),
    . . .
    Fi is E0 + match_cost(Li, Vi), solve(Li, Fi, Ei ,E),
    . . .
    Fn is E0 + match_cost(Ln, Vn), solve(Ln, Fn, En ,E),
    F is max{E1, E2 , . . . , En}.
```

The procedure *match*(L_0, L, [L_1, L_2 , . . . , L_n], V, [V_1, V_2 , . . . , V_n]), collects the variables in L_0 into the list *V*, performs a unification operation between L_0 and L and collects the variables in L_i into the list V_i ($1 \le i \le n$) after the unification operation. The procedure *clause_cost* determines the

cost of using an input clause. The procedure *match_cost* determines the cost of matching variables in the unification operations. The top-level call is solve(false :- [ ], 0, F, E) where E is the input and F is the output.

We first define the function *complexity* for a term t: (1) complexity(t) = 0 if t is a variable; (2) complexity(t) = $n + \sum_{i=1}^{n}$complexity(t_i) if $t = f(t_1, t_2, \ldots, t_n)$ and $n \geq 0$. For a positive literal $L = p(t_1, t_2, \ldots, t_n)$, we define complexity(L) = max{complexity(t_1) , ... , complexity(t_n)}. For a negative literal $N = \neg L$, we define complexity(N) = complexity(L).

We will give two definitions for *clause_cost*. Let C denote the input clause $L :- L_1, L_2, \ldots, L_n$. We say the clause C satisfies the *variable condition*, denoted by *variable_condition(C)*, if there is a variable v in C such that v occurs more in one of $L_1, L_2, \ldots, L_n$ than it does in L. Let B = max{complexity(L_1), complexity(L_2) , ... , complexity(L_n)} and H = complexity(L), we give two definitions for *clause_cost*, which will be called CC1 and CC2 respectively.

$$\text{CC1:}\quad \text{clause\_cost(C)} = \begin{cases} B-H & \text{if } B>H \\ 1 & \text{if } B=H \\ 1 & \text{if variable\_condition(C)} \wedge B<H \\ 0 & \text{otherwise} \end{cases}$$

and

$$\text{CC2:}\quad \text{clause\_cost(C)} = \begin{cases} B-H & \text{if } B>H \\ 1+\lfloor \log_3 n \rfloor & \text{if } B=H \\ 1 & \text{if variable\_condition(C)} \wedge B<H \\ 0 & \text{otherwise} \end{cases}$$

These two definitions obviously favor the input clauses which reduce the complexity of the subgoals. The variable condition is introduced to take into consideration the fact that more occurrences of a variable in a literal of the clause body will increase the complexity of the subgoals during the proof and this increase of complexity during the proof needs to penalized. The logarithmic term in CC2 is introduced to penalize long clauses since they result in bigger branching factors.

The procedure *match_cost* can also be defined in different ways. Its purpose is to determine the cost of the unification operation. We define *match_cost* only to charge for the binding of the variables. In the call *match_cost(L, V)*, L is the subgoal and V is the list of terms bound to the variables in L. Let $V = [t_1, t_2, \ldots, t_n]$ and $S = [s_1, s_2, \ldots, s_m]$ where S is the set of non-variable subterms of $t_1, t_2, \ldots, t_n$, we also give two definitions for *match_cost*, which will be called MC1 and MC2 respectively.

$$\text{MC1:}\quad \text{match\_cost(L,V)} = \sum_{i=1}^{n} \text{complexity}(t_i)$$

$$\text{MC2:}\quad \text{match\_cost(L,V)} = \sum_{i=1}^{m} \text{complexity}(s_i)$$

That is, MC2 does not charge for repeated subterms. The idea is that we can regard a subterm as a piece of information about the proof. The multiple occurrences of the a subterm should be

encouraged since this may indicate better concentration of the search process.

Consider an example where the subgoal is p(f(g(a),X)) and the clause is p(f(X,Y)):-q(f(X,Y)). The match_cost between p(f(g(a),X)) and p(f(X,Y)) is 0 since no variable in the subgoal is bound to a complex term. If the subgoal is p(X) and the clause is p(f(g(a,X))):-q(X), the match_cost between p(X) and p(f(g(a,X))) is complexity(f(g(a,X))). In general, it is the variable bindings to complex terms in the subgoals that count for match_cost.

We have experimented with different definitions of *match_cost* and *clause_cost* using the depth prover. We use (CC1, MC1) to denote the prover using CC1 for *clause_cost* and MC1 for *match_cost*. The results are summarized in Table 4. All four combinations perform well. The combination (CC2,MC2) appears to be the best in general. No special attention is given to any individual problem in these experiments.

Table 4. Summary Data for Proof Complexity Measures

	underlying strategy	(CC1,MC1)	(CC1,MC2)	(CC2,MC1)	(CC2,MC2)
Average Time Per Theorem	224.30	154.95	158.57	191.95	110.26

We have customized the definitions for *match_cost* and *clause_cost* to solve several problems from [13], including am8, gcd, lcm, exq1 and exq2. The basic idea is to favor certain function symbols, certain clauses or certain terms by charging less for them. The underlying strategy can not solve them as efficiently or can not solve them at all.

4. Related Works

The priority system is built on the DFID search strategy. The motivation is to simulate best-first search using DFID; the implementations of the priority system is based on an implementation of DFID. Let's consider the worst-case of the priority system. The worst-case behavior of the priority system occurs when the priority list has to reach its maximal length in theorem A to find a solution. Consider the search space formalized as a tree. Assume that the minimal solution length is N, the branching factor is B. We also assume that S_i is 1 and E_i is i×S in figure 1. At stage i, the priority list is of length $C^{i-1}\times L$. Suppose a solution is found at stage I. The total number of operations performed by the priority system will be at most

$$PS(I) = \sum_{i=1}^{I} DFID(i\times S) = \sum_{i=1}^{I} E\times B^{i\times S} = E\times\sum_{i=1}^{I} B^{i\times S} \approx \frac{B^S}{B^S-1}\times E\times B^{I\times S} = \frac{B^S}{B^S-1} DFID(I\times S)$$

where $E = (\frac{B}{B-1})^2$ and $I = \max\{\frac{N}{S}, (N+1)\times\log_C B - \log_C(L(B-1))+1\}$. This analysis is similar to that in [11] and uses one result $DFID(d) = E\times B^d$ from it. We can see that the priority system is generally a constant factor $\frac{B^S}{B^S-1}$ times as expensive as DFID. But the priority system can be less efficient because the depth I×S can be much larger than the depth required for DFID. We point out that this comparison is based on worst-case analysis and does not consider the heuristic effect of the priority system. From our experience, the priority system general performs better than pure DFID. Furthermore, the priority list can be implemented using much less space than the goal queue in the breadth-first [8] search or A* search [3, 4], since only the priority values need to be stored, which seems to be a significant advantage.

An elaborate scheme is described in [7] for calculating the complexity of the clauses in a resolution theorem prover. That scheme could probably be used to define our priority function in the priority system or to define the procedure *match_cost*. Our proof complexity method probably has a more intuitive meaning, partially because the term complexity has a more direct impact on the search process. Thus our method could be more easily understood and used by the user. The proof complexity does not charge anything for the matches between constant symbols (function symbols or predicate symbols) and usually does not charge anything for a match between a variable and unary function symbols. As a result, the proof paths which contain less variables are favored. This makes our method similar to the idea of *twin symbols* in [13]. One major difference is that we use DFID and [7, 13] use best-first search.

5. Conclusions

We have discussed two refinements — a priority system and a proof complexity measure — to the depth-first iterative deepening search strategy and have shown that the two refinements provide significant improvements in a theorem prover. The priority system is a complete search strategy that preserves the advantage of requiring small memory to operate and allows the otherwise brute-force DFID search to use certain heuristic information provided by the priority function. The proof complexity measure can be used for the back chaining systems, which are common in intelligent systems. We have used it to solve some problems which can not be solved without it. In general, we feel that it can be a powerful tool to help us solve difficult problems.

References

1. Boae, S., E.M. Clarke, D.E. Long and S. Michaylov, "Parthenon: A Parallel Theorem Prover for Non-Horn Clauses", Technical Report No. cmu-cs-88-137, Computer Science Department, Carnegie Mellon University, 1988.

2. Greenbaum, S., "Input Transformation and Resolution Implementation Techniques for Theorem Proving in First-Order Logic", Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, 1986.

3. Hart, P.E., N.J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. on Sys. Sci. and Cybernetics*, July 1968.

4. Huyn, N., R. Dechter and J. Pearl, "Probabilistic Analysis of the Complexity of A*", *Artificial Intelligence*, Vol. 15, pp. 241-254, 1980.

5. Korf, R.E., "Depth-first Iterative Deepening: an Optimal Admissible Tree Search", *Artificial Intelligence*, Vol. 27, 97-109, 1985.

6. Kowalski, R., "Search Strategies for Theorem-Proving", *Machine Intelligence*, Vol. 5, pp. 181-201, 1970.

7. Overbeek, R, J. McCharen and L. Wos, "Complexity and Related Enhancements for Automated Theorem-Proving Programs", *Comp. & Math. with Appls*, Vol. 2, pp. 1-16, 1976.

8. Pearl, J., and R.E. Korf, "Search Techniques", *Ann. Rev. Comput.Sci.*, pp. 451-467, Vol 2, 1987.

9. Plaisted, D.A. and S. Greenbaum, "Problem Representations for Back Chaining and Equality in Resolution Theorem Proving", First Annual AI Applications Conference, Denver, Colorado, December 1984.

10. Plaisted, D.A., "Non-Horn Clause Logic Programming Without Contrapositives", *Journal of Automated Reasoning*, Vol 4, No. 3, September 1988.

11. Stickel, M.E. and M.W. Tyson, "An Analysis of Consecutively Bounded Depth-first Search with Application Automated Deduction", *Proc. of IJCAI*, pp. 1073-1075, 1985.

12. Stickel, M.E., "A PROLOG Technology Theorem Prover: Implementation by an Extended PROLOG Compiler", *Proc. of IJCAI*, pp. 573-587, Oxford, England, July 1986.

13. Wang, T.C. and W.W. Bledsoe, "Hierarchical Deduction", *Journal of Automated Reasoning*, Vol. 3, No. 1, 1987.