Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving

Charles Elkan*
Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

Abstract: This paper applies the idea of conspiracy numbers to derive two heuristic algorithms for searching and/or trees. The first algorithm is an AO* best-first algorithm but the standard guarantees do not apply usefully to it because it conforms to the economic principle of sunk costs. The second algorithm works depth-first and guides the search done by an iterative deepening SLD-resolution theorem prover that we have implemented. To avoid repeated effort, the prover caches successes and failures. It exploits the fact that a new goal matches a cached goal if it is a substitution instance of the latter, not just if the two are identical. Experimental results indicate that conspiracy numbers and especially the new caching scheme are effective in practice.

1 Introduction

This paper applies the idea of conspiracy numbers [McAllester, 1988] to derive two heuristic algorithms for searching and/or trees. The first algorithm turns out to be a member of the class of AO* best-first algorithms [Nilsson, 1980], but it conforms to the principle of sunk costs (a rule of economic rationality not respected by traditional and/or tree search algorithms) and hence the standard guarantees of termination and admissibility do not apply usefully to it.

The second algorithm works depth-first. It guides the search done by an iterative deepening SLD-resolution theorem prover that we have implemented. In addition to caching successes, the prover caches failures and uses the latter to avoid repeated effort also. The prover exploits the fact that a new goal matches a cached success or failure if it is a substitution instance of the cached goal, not just if the two are identical. Unlike many heuristics for guiding search and saving effort, the conspiracy number and caching ideas introduced in this paper can be implemented efficiently. Our experimental results indicate that conspiracy numbers and especially the new caching scheme are effective in practice.

Section 2 develops the best-first conspiracy numbers algorithm, relates it to traditional and/or tree search algorithms, and then presents the depth-first conspiracy numbers algorithm. Section 3 describes our theorem prover and compares it to a similar PROLOG-technology theorem prover [Stickel, 1986]. The caching done by our prover is discussed in Section 4. Finally, our experimental results appear in Section 5, and Section 6 contains our conclusions.

'This research was supported in part by the United States Office of Naval Research through grant N0014-88-K-0123.

2 Best-first and depth-first conspiracy number algorithms

And/or trees are well-known [Nilsson, 1980; Pearl, 1984], and we shall describe them here only to the extent necessary to make the terminology of this paper understandable. Briefly, an *and/or tree* is a tree where each node is an *and-node* or an *or-node*. The children of and-nodes are required to be or-nodes, and vice versa. An and/or tree may be completely or partially explored. Each node of a completely explored and/or tree is *solved* or *failed*. An internal and-node is solved if each of its children is solved and it is failed if at least one of its children is failed. Conversely, an internal or-node is solved if at least one of its children is solved, and failed if all its children are failed. A partially explored and/or tree also contains leaf nodes called *unexpanded* leaves whose children have not yet been discovered. The solved/failed status of an unexpanded leaf is unknown, as is the status of any internal node of a partially explored tree whose status is not fixed by the status of its children.

A *solution* of an and/or tree is a subtree that demonstrates that the root of the tree is solved. The aim of searching an and/or tree is to find a solution. Concretely, a solution is a subtree such that all its nodes are solved, all the children of each of its and-nodes belong to the subtree, and at least one of the children of each of its or-nodes also belongs to the subtree. The basic searching operation on a partially explored and/or tree is to expand an unexpanded leaf of the tree. When such a leaf is expanded there are three possible outcomes: it can be discovered to be solved, it can be discovered to be failed, or it can be discovered to have children, which are new unexpanded leaves. Different searching algorithms choose in different ways which unexpanded leaf to expand next.

2.1 Best-first conspiracy number search

The notion of conspiracy was invented in the context of min/max tree searching [McAllester, 1988]. Informally, a conspiracy for a min/max tree is a set of leaf nodes of the tree such that if their values were to change, then the value obtained for the root of the tree by min/max propagation might also change. The following definition for and/or trees is analogous.

Definition 2.1: A *conspiracy* for a partially explored and/or tree is an inclusion-minimal set of unexpanded leaves of the tree such that if each node in the set is found to be solved then the root of the tree is also solved. ■

Our aim in searching an and/or tree is to find a solution as quickly as possible. Intuitively, the bigger a conspiracy is, the less likely all of its members can be solved, and the more time will be required to attempt to solve them. Thus an appealing heuristic for guiding search is always to expand next a member of the currently smallest conspiracy. This heuristic characterizes best-first conspiratorial search.

A best-first conspiratorial search algorithm can quickly identify a member of the smallest conspiracy of a partially explored and/or tree by keeping track of conspiracy numbers.

Definition 2.2: The *conspiracy number* $cn(\alpha)$ of a node α is the cardinality of the smallest conspiracy of the partially explored and/or subtree rooted at α. ■

The next lemma provides a constructive definition of conspiracy numbers.

Lemma 2.3:

- If α is a solved leaf then $cn(\alpha) = 0$.
- If α is a failed leaf then $cn(\alpha) = \infty$.
- If α is an unexpanded leaf then $cn(\alpha) = 1$.
- If α is an and-node with children $\alpha_1, \ldots, \alpha_k$ then $cn(\alpha) = \sum_1^k cn(\alpha_i)$.
- If α is an or-node with children $\alpha_1, \ldots, \alpha_k$ then $cn(\alpha) = \min\{cn(\alpha_i)\}$. ■

Lemma 2.3 implies that whenever a leaf is expanded, only the conspiracy numbers of nodes on the path from that leaf to the root need to be updated. The next lemma says how an unexpanded leaf in the currently smallest conspiracy can be found quickly.

Lemma 2.4: An unexpanded leaf β_m that is a member of the smallest conspiracy of a partially explored and/or tree can be found by following a path $\beta_1, \ldots, \beta_m$ from the root β_1 of the tree such that

(i) for each and-node β_j on the path, β_{j+1} is a child or-node of β_j and $cn(\beta_{j+1}) > 0$, and

(ii) for each or-node β_j on the path, β_{j+1} is a child and-node of β_j and $cn(\beta_{j+1}) = cn(\beta_j)$. ■

2.2 Conspiracy numbers, AO* algorithms, and the principle of sunk costs

Traditional and/or tree search algorithms are members of the class of AO* algorithms [Nilsson, 1980; Pearl, 1984]. AO* algorithms are interesting because under certain conditions, they are guaranteed to terminate and to find optimal solutions. Lemma 2.3 shows that the function *en* is of the form used by AO* algorithms. However the standard guarantees do not apply usefully to best-first conspiratorial search, because it respects the principle of sunk costs.

Principle of Sunk Costs: Amounts of resources expended in the past on different activities are irrelevant to the decision of which activity to pursue now. ■

Although people often act contrary to the principle of sunk costs, it is a guideline for rational agents that commands universal assent among economists [McCloskey, 1985; p. 267]. The principle says that, for example, the costs incurred to build a nuclear power plant should have no bearing on the decision whether or not to operate the plant.

For guaranteed termination at a minimal-cost solution, in deciding which node to expand next and when to halt, one must take into account the cost so far of incomplete solutions. The principle of sunk costs disallows this.

Termination. The nodes on an indefinitely long path from the root of an and/or tree can all have the same minimal conspiracy number. Hence best-first conspiratorial search is not guaranteed to terminate.

Admissibility. AO* algorithms find optimal solutions because of their termination criterion, when they use conservative heuristic functions.

Let *Q* be a function assigning measures to solutions. An AO* heuristic function *h* is a *conservative estimator* of *Q* if for all nodes a, *h(α)* is less than the lowest *Q* measure of any solution of the and/or subtree rooted at *α*. The termination criterion for an AO* algorithm is to stop only when no incomplete solution induces an *h* value for the root lower than the *h* value induced by the best complete solution found so far. Then the fact that *h* is a conservative estimator of *Q* guarantees that all solutions not yet found are worse under *Q* than this best solution, given that the *h* values induced by complete solutions are their *Q* measures.

The *Q* that assigns the measure 0 to every solution is the only *Q* according to which the *en* values induced by complete solutions are their *Q* measures. Best-first conspiratorial search does find optimal solutions according to this trivial measure.

The *en* heuristic function is a conservative estimator of some non-trivial solution measures. For example *en* is a conservative estimator of the number of nodes that a solution contains. However if *Q* is a non-trivial measure, then best-first conspiratorial search does not in general find optimal solutions according to Q, because the *en* values induced by complete solutions are less than their *Q* measures.

2.3 Depth-first conspiracy number search

The intuition behind depth-first conspiracy number search is to explore an and/or tree in depth-first fashion, but to backtrack when the smallest conspiracy involving a leaf is too large. Conspiratorial depth-first search has

two advantages: the and/or tree being searched need never be stored explicitly, and the decision whether to expand a leaf can be made using only information available at that leaf. A disadvantage is that leaves are not expanded in best-first order.

Suppose an and/or tree is searched in depth-first fashion. Then at any time one can identify a stack of or-nodes whose subtree is currently being explored. The deepest node on this stack is the one whose subtree will be explored next. Each of the or-nodes in the stack has some sibling or-nodes. If the deepest or-node can be solved, and the siblings of all the less deep or-nodes can also be solved, then a solution has been found for the whole tree. Thus the deepest or-node on the stack and the siblings of the less deep or-nodes constitute a conspiracy. In fact, they constitute the only conspiracy of the current partial and/or tree, since the nodes on the stack and their siblings are the only nodes of the tree whose subtrees have not already been fully explored, and discovered to be failed.

The information needed to decide during bounded depth-first search whether to expand a leaf is its conspiracy depth according to the following definition.

Definition 2.5: The *conspiracy depth* of a node in a partial and/or tree is the size of the smallest conspiracy of the whole tree involving that node. ■

The next lemma implies that conspiracy depths can be computed with only a constant amount of extra work per node, during depth-first exploration of an and/or tree.

Lemma 2.6: Consider the function *cd* defined on the nodes of an and/or tree as follows.

- Let *α* be the root of the tree. Then *cd(α)* — 1.
- Let *α* be an and-node with *cd(α)* — *c.* Let a have it children $\alpha_1,\ldots,ar^*$ which are explored in order. Then *cd(αj)* = c + *k* — j.
- Let *α* be an or-node with *cd(α)* = c. Let the children of α be α 1 , . . . , αk. Then *cd(αj)* = c.

At the time the decision is made whether or not to expand a leaf α during depth-first search of an and/or tree, the conspiracy depth of a is *cd(α).*

Conspiratorially bounded depth-first search can be implemented efficiently.

3 A conspiratorial SLD-resolution theorem prover

Conspiratorially bounded depth-first search guides the search done by a theorem prover that we have implemented. The only inference rule of our prover is SLD-resolution [Lloyd, 1984]. The space that must be searched to find a proof of a given goal (an atomic first-order predicate calculus formula) by SLD-resolution is an and/or tree, where or-nodes correspond to subgoals that must be unified with the head of some matching clause, and and-nodes correspond to bodies of clauses. For such an and/or tree, a conspiracy consists of a set of subgoals that must all be proved together. In order to do so, one must find an an answer substitution for each subgoal such that none of the substitutions conflict. Thus as a conspiracy gets larger, even if whether an answer substitution exists is a statistically independent event for each of its members, heuristically the chance that all the members can be compatibly proved decreases as if the events were negatively correlated.

At the highest level, our prover uses the idea of iterative deepening [Korf, 1985; Stickel and Tyson, 1985]: it repeatedly does depth-first search with increasing conspiracy depth bounds, until a solution is found. Our prover is thus similar to the PROLOG-technology theorem prover PTTP [Stickel, 1986]. PTTP does iteratively deepened depth-first search using a depth function *sd* which is the same as the conspiracy depth function *cd* except on and-nodes. If *α* is an and-node with *k* children α 1 , . . .,αk then *sd(αj)* = *c* + *k* for each child $_{\alpha j;}$, whereas *cd(αj)* = *c* + *k* — *j.* The function *sd* is not consistent with the principle of sunk costs. In depth-first search, the child $_{\alpha j;}$ of an and-node is only expanded if its siblings α 1 , . . . , α j _ i have been solved. The effort required to do this is in the past at the moment the decision is made whether or not to expand *αj.* At that moment it is only relevant what subgoals remain to be solved, and there are *c* + *k* — *j* of them.

A second difference between PTTP and our prover is that PTTP attains first-order completeness, because it uses a model elimination inference rule in addition to SLD-resolution. In future work we plan to extend our prover similarly.

Using conspiracy depths to limit search leads to one difficulty: the conspiracy depth of an indefinitely large partial solution can be finite. If an and-node *α* has a single child *α'* then *cd(α')* = *cd(α),* so in a partial and/or tree, it is possible for the nodes on an infinite path each to have the same conspiracy depth. This problem is circumvented in our prover by using a modified conspiracy depth function *cd'* such that if *αj* is any child of the and-node a, then *cd'(αj)* > cd'(α). Specifically, the definition of *cd'* is identical to that of cd, except that if a is an and-node with *k* children α 1 , . . . αk then *cd'(αj)* — *c* + *k* — *j* + e. For want of any principled way to choose e_1 our prover takes c = 1.

4 Caching successes and failures

This section explains the caching scheme implemented in our SLD-resolution prover. The idea of caching is to store the results of past computations in order not to waste time repeating them in the future [Keller and Sleep, 1986; Pugh, 1988]. Caching is especially useful combined with iterative deepening, because results that have been cached during search to one depth bound can be reused during later searches. Our caching scheme is distinctive in two respects.

First, if we fail to prove a subgoal within a specified conspiracy depth bound, we cache the fact of the failure. If we attempt to prove the same subgoal later within the same or a lower conspiracy depth bound, the attempt is recognized as doomed to failure and it is aborted. Some systems cache subgoals known to be false, which are often called contradictions or nogoods [de Kleer, 1986]. Our prover also caches resource-bounded failures.

Second, we take advantage of the fact that with SLD-resolution, an attempt to prove a subgoal is an attempt to prove any substitution instance of it. If our prover succeeds or fails in proving a subgoal, then later it takes any instance of the subgoal as proven or impossible to prove, respectively.

The new prover maintains two separate caches, called the *success* and *failure* caches. Both the success and failure caches grow monotonically across iterative deepening cycles. The success cache Π is where proven subgoals are remembered. Whenever a subgoal β is proven with answer substitution σ, then β' is added to Π, where $\beta' = \beta\sigma$. The next lemma says that a subgoal can be taken as proven if it is an instance of an entry in Π.

Lemma 4.1: If γ is in Π and γ' is a substitution instance of γ, then $\vdash \gamma'$.
Proof: Suppose γ is in Π, so $\vdash \gamma$. That signifies $\vdash \forall\overline{x}\ \gamma$ where $\overline{x}$ is the vector of free variables of γ. So for any vector of terms $\overline{t}$, it is the case that $\vdash \gamma[\overline{x} \mapsto \overline{t}]$. Let γ' be a substitution instance of γ. That means $\gamma' = \gamma[\overline{x} \mapsto \overline{t}]$ for some $\overline{t}$. ■

Whenever our prover attempts to prove a subgoal β within the conspiracy depth bound m and fails, it adds the pair $\langle\beta, m\rangle$ to the failure cache Ω. The next lemma justifies the way entries in Ω are exploited. The notation $\vdash_m \beta$ means β is provable within the conspiracy depth bound m.

Lemma 4.2: If $\langle\gamma, n\rangle$ is in Ω and γ' is a substitution instance of γ and $m \leq n$, then $\not\vdash_m \gamma'$.
Proof: An attempt to prove γ with resource bound n succeeds with answer substitution σ if for some $m \leq n$ and some σ, it is the case that $\vdash_m \gamma\sigma$. Thus if $\langle\gamma, n\rangle$ is in Ω then for all σ and $m \leq n$, $\not\vdash_m \gamma\sigma$. But $\gamma' = \gamma\sigma$ for some σ. ■

Suppose the attempted proof of the subgoal γ failed absolutely: the space of potential proofs of γ was explored completely and no proof was found. Then for all n, $\not\vdash_n \gamma'$ if γ' is an instance of γ. Our prover takes this into account.

Conversely, the prover only takes a goal γ' as proven because it is an instance of an entry γ in the success cache Π if the depth bound on the current attempt to prove γ' is not greater than the depth of the proof of γ that caused γ to be added to Π. The effect of this restriction is that the and/or tree of potential proofs explored with caching is exactly the same as the tree explored without caching.

The caching scheme just described was independently discovered and implemented in unpublished work by Stickel. He found that the overhead of caching outweighed its benefits, for the class of theorems he investigated. Our experimental results, reported in the next section, are different. The remainder of this section discusses how to implement caches efficiently and proves that caching is asymptotically beneficial under reasonable assumptions.

4.1 Implementing instance-lookup caches

An implementation of a success or failure cache Σ must support two abstract access operations:

- *install*(Σ, γ, m): place γ in Σ with associated depth m, and
- *query*(Σ, γ, m): decide whether a β with associated depth n exists in Σ such that γ is an instance of β and $n \leq m$.

The entries in a cache are atomic logical formulas. Accessing a cache involves substitution instance testing, so it is not possible to use standard hashing techniques to implement caches. For our prover we choose to represent each cache by a binary tree with a member of the cache stored at each node of the tree.

Let $\alpha(n)$ be the value stored at node n. According to a total order $\prec$ on terms, the tree representing each cache is *symmetric-ordered*: for all nodes n,

- if the node l belongs to the left subtree of n then $\alpha(l) \prec \alpha(n)$, and
- if the node r belongs to the right subtree of n then $\alpha(r) \succ \alpha(n)$.

The relation $\prec$ is defined on formulas and their subterms as follows.

- $\prec$ is an alphabetic order on pairs of atomic symbols and on pairs of variables.
- $f(\overline{x}) \prec g(\overline{y})$ if $f \prec g$ or $f = g$ and $\overline{x} \prec \overline{y}$ lexicographically.
- If x is a variable then $x \prec t$ for every non-variable term t.

The operation *query*(Σ, γ, m) is performed by walking down from the root of Σ, at each node n answering *yes* if γ is an instance of $\alpha(n)$, otherwise moving left if $\gamma \prec \alpha(n)$, moving right if $\gamma \succ \alpha(n)$, and answering *no* if n is a leaf. The operation *install*(Σ, γ, m) is performed similarly.

If the tree representing a cache is balanced, then *install*(Σ, γ, m) and *query*(Σ, γ, m) operations both use $O(\log|\Sigma|)$ time. There are standard techniques for keeping symmetric-ordered trees balanced [Tarjan, 1983], which will not be discussed here.

Unfortunately the order $\prec$ on formulas is not *instance-respecting*: it may be the case that γ is a substitution instance of α_1 and $\gamma \prec \alpha_2$ yet $\alpha_2 \prec \alpha_1$. Thus it may happen that *query*(Σ, γ, m) is answered *no* when the correct answer is *yes*. In our experience this happens infrequently. The next lemma says that there is no alternative order to $\prec$ that avoids this problem.

Lemma 4.3: No instance-respecting total order exists.
Proof: Suppose $<$ is an instance-respecting total order. Let x, y, and z be three distinct variables. Without loss of generality assume $x < y < z$. Now x is an instance of z and $x < y$ so $z < y$, a contradiction. ■

The order $\prec$ can be made into an instance-respecting partial order by declaring that if x is a variable, then for any term t, $x \not\prec t$ and $t \not\prec x$. We believe that there are efficient ways to implement caches supporting substitution instance queries based on this partial order, or on discrimination trees [Forgy, 1982; Charniak *et al.*, 1987].

Figure 1: The effect of different depth functions.

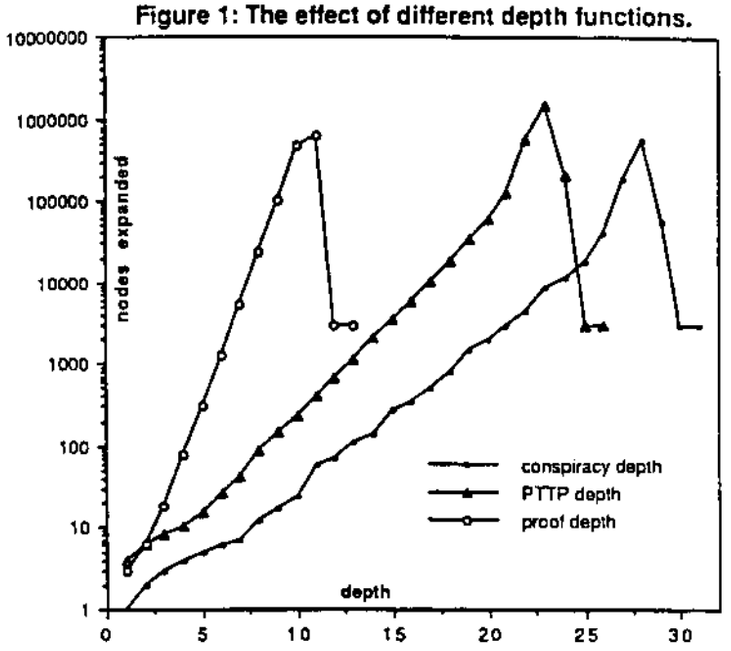
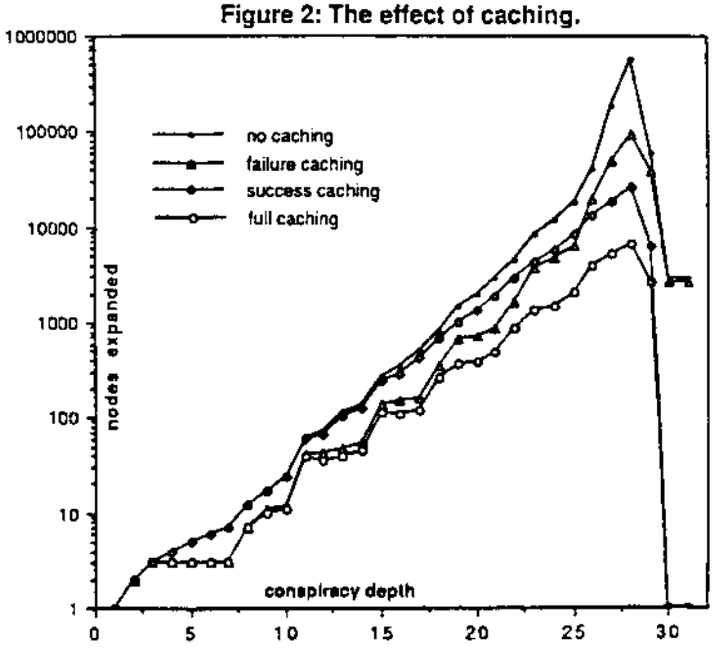


Figure 2: The effect of caching.



4.2 Analyzing the benefits of caching

When a subgoal is found to be an instance of a cache entry, then that subgoal gives rise to no child subgoals. Thus one expects caching to reduce the effective branching factor of the and/or tree representing the space to be searched to find an SLD-resolution proof. The following theorem says how much the effective branching factor must be reduced for caching to be beneficial.

Theorem 4.4: Let the effective branching factor be b_c with caching, and b_o without caching. Let the time to execute an *install*(Σ, γ, m) or *query*(Σ, γ, m) operation be $c(|\Sigma|) = o(|\Sigma|^p)$. Then caching saves time if $1 + p < \log b_0 / \log b_c$.

Proof: Let the time required to find a solution be T_o without caching, and T_c with caching. We wish to know when $T_c < T_0$. Let t_0 be the time required to expand one subgoal, without caching. It is reasonable to assume that t_0 is a constant and $T_0 = m_0 t_0$, where m_0 is the number of subgoals expanded before finding a solution. If a solution is first found at depth d, then a lower bound on m_0 is $m(d-1, b_0)$, where

$$m(d, b) = \sum_{0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}.$$

The solution found first with caching is the same, at the same depth d. The time required to expand one subgoal with caching is $t_c = t_0 + c(|\Sigma|)$, where Σ is the cache at the time that subgoal is expanded. The cache grows monotonically, but cannot have more than one entry for each different subgoal expanded, so $|\Sigma| < b_c^d$ always. Thus

$$T_c \leq m(d, b_c)\left(t_0 + c(b_c^d)\right) = m(d, b_c)\left(t_0 + o(b_c^{dp})\right).$$

Caching is advantageous if $\log T_c < \log T_0$. We have $\log T_0 > d \log b_0 + O(1)$ and

$$\log T_c \leq d \log b_c + dp \log b_c + O(1).$$

Thus for some d_0, for all $d > d_0$, if $1 + p < \log b_0 / \log b_c$ then $T_c < T_0$. ■

Note that if $c(|\Sigma|) = O(\log |\Sigma|)$ then $c(|\Sigma|) = o(|\Sigma|^p)$ for all $p > 0$, so for caching to be beneficial it is only necessary that $b_c < b_0$.

In our experience caches do not grow fast. For the problem described in the next section, where m_0 is greater than 1000 000, both the success and failure caches ended with under 150 entries. It is definitely advantageous to treat any instance of an explicit cache entry as an implicit entry.

5 Experimental results

We have performed two sets of experiments to evaluate the usefulness of the ideas developed in this paper. Each experiment involved using a different search algorithm to search the same and/or tree. The tree used in all our experiments is the tree representing the SLD-resolution search space for a monkey-and-bananas situation-calculus planning problem [Plaisted, 1986].

The first set of experiments compared iterative deepening with three different depth functions: actual proof depth, the PTTP depth function, and the modified conspiracy depth function *cd!*. The results of the experiments are shown in Figure 1. Measured by the total number of nodes expanded, the conspiracy depth function has a slight advantage. (The first solution to the monkey-and-bananas problem has actual proof depth 11, PTTP depth 24, and modified conspiracy depth 29. Search to any depth is cut off when a solution is found, so the maximum number of nodes expanded occurs with PTTP depth bound 23 or conspiracy depth bound 28.)

Using iterative deepening with the conspiracy depth function cd', the second set of experiments compared four caching regimes: no caching, caching of successes only, caching of failures only, and full caching. In each case the same search space was explored. Note that with

caching of successes or with full caching, the first solution found is cached, so the problem can then be solved with just one node expansion when searching to depth 30, 31, and so on.

The results of the second set of experiments are shown in Figure 2. With no caching the effective branching factor is approximately 1.61. Caching successes reduces it to 1.46, caching failures to 1.55, and caching both to 1.40. The improvement is clear. It is especially encouraging that the benefits of caching successes and failures are cumulative. For our sample problem, the branching factor reduction achieved with full caching translates into a 99% reduction in the search space size.

6 Conclusion

In this paper we have described heuristic best-first and depth-first algorithms for searching and/or trees based on the intriguing idea of conspiracy numbers. We have also described an SLD-resolution theorem prover that uses the depth-first conspiracy numbers algorithm and a caching scheme that is sophisticated yet amenable to theoretical analysis. The theorem prover relies only on heuristics that can be implemented efficiently: it does not sacrifice speed for intelligence.

Naturally many issues remain unresolved. We conjecture that an algorithm conforming to the principle of sunk costs provably dominates in some way an algorithm that violates the principle, but we do not know in what way. We also do not know how to characterize the class of theorem-proving problems for which the caching ideas described here are beneficial.

Acknowledgements. Section 2 reports on work done jointly with David McAllester. Discussions with James Altucher, Wilfred Chen, David McAllester, and Alberto Segre were helpful in developing the ideas reported in Section 4.

References

[Charniak *et al.*, 1987] Eugene Chamiak, Christopher K. Riesbeck, Drew V. McDermott, and James R. Meehan. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987. Second edition.

[de Kleer, 1986] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence,* 28:127-162, 1986.

[Forgy, 1982] Charles L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence,* 19:17-37, 1982.

[Keller and Sleep, 1986] Robert M. Keller and M. Ronan Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems,* 8(1):88-108, January 1986.

[Korf, 1985] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence,* 27(1):97-109, September 1985.

[Lloyd, 1984] John W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer Verlag, 1984. Second edition, 1987.

[McAllester, 1988] David A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence,* 1988.

[McCloskey, 1985] Donald N. McCloskey. *The Applied Theory of Price.* Macmillan Publishing Company, New York, second edition, 1985.

[Nilsson, 1980] Nils Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, California, 1980.

[Pearl, 1984] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, 1984.

[Plaisted, 1986] David A. Plaisted. Simplified problem reduction format theorem prover examples, 1986. Distributed over the Arpanet.

[Pugh, 1988] William W. Pugh. *Incremental Computation and the Incremental Evaluation of Functional Programs.* PhD thesis, Cornell University, 1988.

[Stickel and Tyson, 1985] Mark E. Stickel and W. M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* pages 1073 1075, August 1985.

[Stickel, 1986] Mark E. Stickel. A PROLOG technology theorem prover: Implementation by an extended PROLOG compiler. In Jorg H. Siekmann, editor, *Eighth International Conference on Automated Deduction,* number 230 in Lecture Notes in Computer Science, pages 573-587. Springer Verlag, 1986.

[Tarjan, 1983] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.