A New Method to Index and Query Sets

Jörg Hoffmann and **Jana Koehler**
Institute for Computer Science
Albert Ludwigs University
Am Flughafen 17
79110 Freiburg, Germany
hoffmann|koehler@informatik.uni-freiburg.de

Abstract

Let us consider the following problem: Given a (probably huge) set of sets S and a query set q, is there some set $s \in S$ such that $s \subseteq q$? This problem occurs in at least four application areas: the matching of a large number (usually several 100,000s) of production rules, the processing of queries in data bases supporting set-valued attributes, the identification of inconsistent subgoals during artificial intelligence planning and the detection of potential periodic chains in labeled tableau systems for modal logics. In this paper, we introduce a data structure and algorithm that allow a compact representation of such a huge set of sets and an efficient answering of *subset* and *superset* queries. The algorithm has been used successfully in the IPP system and enabled this planner to win the ADL track of the first planning competition.

1 Introduction

The problem of how to effectively index and query sets occurs in various computer applications:

Researchers in **object oriented databases** have among others stressed the need for richer data types, in particular set-valued attributes such as for example the set of keywords in a document, the set of classes a student is enrolled in, etc. Typical queries to such an enriched database require to determine all *supersets* or *subsets* of a given query. For example, given a set of classes C, to find all students taking at least these classes the *supersets* of C need to be determined. Or as another example, given that a student has passed some basic courses C, which advanced courses become possible? The answer is found by retrieving all advanced courses whose prerequisite course set is a *subset* of C.

Machine learning is highly concerned with the *utility problem*, i.e., the problem of learning new knowledge in such a way that the learning costs do not exceed the savings in the system's performance that are achieved through learning. A key factor in a good learning algorithm is to effectively match sets of attributes against each other, for example in the form of *preconditions* of large sets of *production rules*. In order to decide which rules apply to a particular situation, all precondition sets have to be determined that are *subsets* of the query set describing the situation.

Artificial intelligence planning is concerned with the problem of constructing a sequence of actions that achieves a set of goals given a particular initial state in which the plan is scheduled for execution. Even in its simplest form, the problem is known to be PSPACE-complete [Bylander, 1991], i.e., planning algorithms are worst-case exponential and techniques to effectively prune the search space are mandatory. Since millions of goal sets are constructed when searching larger state spaces for a plan, it is important to know in advance when a goal set can never be satisfied. This is the case when at least one *subset* of the current goal set has previously been shown to be unsatisfiable.

Modal logics are often formalized using labeled tableau methods, where one is sometimes confronted with infinite branches in the tableau. To guarantee termination, one needs to identify potential periodic chains of labels [Gore, 1997].

In its abstract form, these and other applications have to deal with the following problem: Given a set of sets S stored in some data structure and a query set q, does there exist a set $s \in S$ such that $q \supseteq s$ (or sometimes $q \subseteq s$ depending on the application). Though this seems to be a trivial problem to handle, its difficulty comes from the dimensions in which it occurs: S will quite often contain millions of sets and some applications require to handle a huge sequence of queries q_i over a dynamically changing S.

2 The UBTree Algorithm

The key to a fast set query answering algorithm lies in an appropriate data structure to index a large number of sets of varying cardinality.

2.1 The UBTree Data structure

A node (see Figure 1) N in UBTree consists of three components:

- $N.e$: the element it represents
- $N.T$: the sons, a set of other tree nodes
- $N.EoP$: the End-of-Path marker.

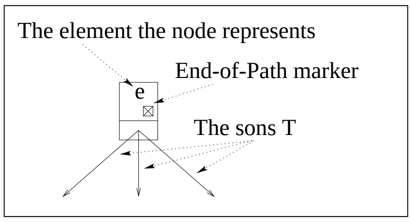

Figure 1: Representation of a single node.

The UBTree is now simply represented by a set T of such nodes and can be seen as a forest. Note that we are speaking about *sets* of tree nodes, i.e., neither the number of trees nor the number of sons any node can have is limited. That is where the UBTree got its name from: Unlimited Branching Tree.
Based on this data structure, the following functions will be defined below:

- insert(set s, tree T) inserts a set s into the tree T.
- lookup_first(set q, tree T) tells us whether *any subset* $s \subseteq q$ is stored in T.
- lookup_subs(set q, tree T) determines *all subsets* for a given query set q from T.
- lookup_sups(set q, tree T) determines *all supersets* for a given query set q from T.

The only assumption on which our algorithm relies is the existence of a total ordering over all elements that can possibly occur in the sets.

2.2 The insert Function

To insert a set s into a forest T, the insert function (Figure 2) creates a *path* starting in T corresponding to that set. In principle, this path is just a series of connected tree nodes corresponding to the ordered elements of s.

Definition 1 *Let $s = \{e_0, \ldots, e_{n-1}\}$ be an ordered set, T be a forest.*
A path $\Phi(s)$ in T corresponding to s is a non-empty tuple of tree nodes $N_0, \ldots, N_{n-1}$ such that

1. $N_0 \in T \wedge N_0.e = e_0$
2. $\forall i \in 1 \ldots n-1 : (N_i \in N_{i-1}.T \wedge N_i.e = e_i)$
3. $N_{n-1}.EoP =$ TRUE

When the insert function is called, T is initialized with the root nodes of the forest. The function then tries to look up the elements of s one after the other (remember that the sets get ordered before they are inserted) and, doing this, follows paths that have been created by other, previously inserted sets. If there is no corresponding node for some element e_i in the set T of nodes (implying that no set starting with $\{e_0, \ldots, e_i\}$ has been inserted before), it creates a new path at that point, i.e., a new son is added to the current node N. Finally, the last node (the one that represents e_{n-1}) is marked as the end of a path. Note that we assume s to be non-empty, i.e., $n > 0$, otherwise there would be no element e_{n-1} and, consequently, no node N to represent it.[1]

```
sort($s$) /* now $s = \{e_0, \ldots, e_{n-1}\}$ */
for all elements $e_i \in s$ do
    $N$ := the node in $T$ corresponding to $e_i$
    if there is no such $N$ then
        insert a new node $N$ corresponding to $e_i$ into $T$
    endif
    $T := N.T$
endfor
mark $N$ as the end of a path
```

Figure 2: insert(s,T)

Figure 3 shows how a forest T with two trees evolves when the 4 (already ordered) sets $s_0 = \{e_0, e_1, e_2, e_3\}$, $s_1 = \{e_0, e_1, e_3\}$, $s_2 = \{e_0, e_1, e_2\}$, and $s_3 = \{e_2, e_3\}$ are inserted one after the other.

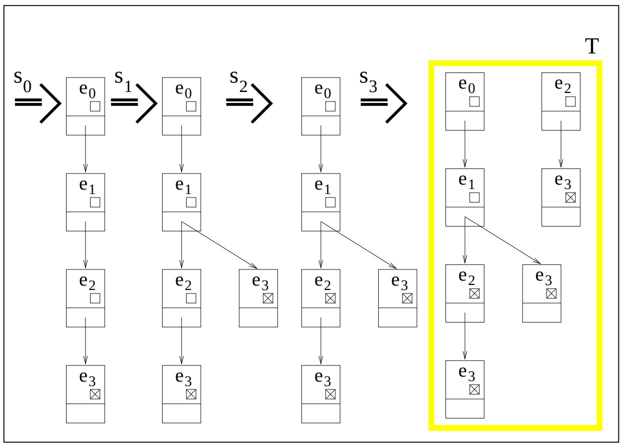

Figure 3: Iterative insertion of 4 sets

2.3 The lookup Functions

We now show how the set of sets S stored in a forest T can be used to answer questions about the query set q.

Let us begin with the lookup_first function, which decides if there is *any* set $s \in S$ in T with $s \subseteq q$. The function simply tries to reach a node N in the

[1]We make this assumption just for simplification, the UBTree algorithm can easily be extended to deal with possibly empty sets.

tree that is marked as the end of a path, using the elements in q as "money" to pay its travel costs. This is to say, given a set of tree nodes T and a set $q = \{e_0, \ldots, e_{n-1}\}$, it finds all nodes $N \in T$ corresponding to an element $e_i \in q$. If any such N is marked as the end of a path, lookup_first succeeds in finding a subset to the query set. Otherwise, the ongoing search is a recursive instance with the set of sons, $N.T$, and the remaining elements $\{e_{i+1}, \ldots, e_{n-1}\}$ as parameters.[2] See Figure 4 for a formal description of the lookup_first function.

```
$M$ := all nodes $N \in T$ that match an element $e_i \in q$
while $M$ is non-empty do
        choose a node $N \in M$
        if $N$ is the end of a path then succeed endif
        /* else:  call search on next tree
        and remaining elements */
        lookup_first($N.T$, $\{e_{i+1}, \ldots, e_{n-1}\}$)
endwhile
/* all elements have failed */
fail
```

Figure 4: lookup_first(q,T)

The function is initially called on the (previously ordered) query set q and the forest T. It is important to notice and crucial to the performance of UBTree that, in using the "money" method, large fractions of the search space can be excluded: for a query set $q = \{q_0, \ldots, q_{n-1}\}$ of length n, 2^n possible subsets need to be considered. Half of them (those that contain q_0) lie in the tree rooted in the node corresponding to q_0. If there is no such node, the search space is immediately reduced to 2^{n-1}.

Note that the function is non-deterministic with respect to which matching node $N \in M$ is chosen first. A possible heuristic could store in each node the distance to the next end-of-path node. The node with the least distance could then be tried first. This distance information can also be used, and in fact is used in our implementation to cut unnecessary branches out of the search tree. If the distance stored in a search node N is greater than the number of elements we have left at that stage of search, we can back up (i.e., **fail**) right away. In this case the query set does not contain enough elements to reach an end-of-path marker.

The lookup_subs function, as given in Figure 5, works in a very similar way. Instead of terminating after identifying one end-of-path marker, i.e., the first matching subset, it has to work its way through *all* nodes it can reach. Again, the function uses the "money" method, i.e., passes only those nodes for which it has matching elements in the query. Every time it finds a node that is the end of a path, it adds the path ending in this node to a global answer set Q. The function is initially called with the ordered query set q and the forest T as parameters, the answer set is initialized with the empty set $Q := \emptyset$. Upon termination of lookup_subs, Q contains all sets $s \in S$ with $s \subseteq q$.

[2] No nodes representing any of the elements in $\{e_0, \ldots, e_i\}$ can exist in a subtree starting in e_i due to the total ordering of the elements.

```
$M$ := all nodes $N \in T$ that match an element $e_i \in q$
for all nodes $N \in M$ do
    if $N$ is the end of a path then
        $Q := Q \cup$ { the set on the path to $N$ }
    endif
    lookup_subs($N.T$, $\{e_{i+1}, \ldots, e_{n-1}\}$)
endfor
```

Figure 5: lookup_subs(q,T)

Finally, we show how to retrieve all supersets $s \in S$, $s \supseteq q$ from T. This can be done by finding all paths in T that comprise the path corresponding to q as a (possibly disconnected) subpath. The lookup_sups algorithm is shown in Figure 6. Again, the answer set is initialized with $Q := \emptyset$ and the function is called with the ordered query set q and the forest T as input parameters.

```
$M$ := all nodes $N \in T$ matching $e$ with $e < e_0$
call lookup_sups($q$, $N.T$) on all nodes $N \in M$
if a node $N \in T$ corresponds to $e_0$ then
  if $\{e_1, \ldots, e_{n-1}\} \neq \emptyset$ then
    lookup_sups($\{e_1, \ldots, e_{n-1}\}$, $N.T$)
  else
    $P$ := all end-of-path nodes $N'$ in the tree under $N$
    $Q := Q \cup \{s \mid s$ is represented by a node $N' \in P\}$
  endif
endif
```

Figure 6: lookup_sups(q,T)

At each stage of the search, lookup_sups does the following: First, it searches all trees that start with an element *preceding* the first element in the query set; these are trees that can possibly contain nodes for the whole set. If there is a node $N \in T$ that directly corresponds to the first element in the (possibly already reduced) query set, the query set gets further reduced by this element. Now the function only needs to find matching nodes for the remaining elements. If there are no such elements left, the search has succeeded, i.e., every end-of-path node that can be reached from N will yield a superset to the query. Otherwise, search needs to be continued with the reduced query set and the appropriate set of nodes.

3 Theoretical Properties of UBTree

We state the soundness and completeness of the lookup_first function in the following theorem.

Theorem 1 *Let T be a forest* UBTree *that has been constructed by iteratively inserting all sets $s \in S$. The* lookup_first *function, as defined in Section 2, succeeds on a query set q and T if and only if there exists a set $s \in S$ with $s \subseteq q$.*

A similar theorem stating the soundness and completeness of the lookup_subs and lookup_sups functions can be proven.

To analyze the runtime behavior of the lookup_subs and lookup_first functions, let us reexamine the algorithm in Figure 5 from a different perspective. At each stage, the function tries to find a node N in T that matches the *first* element e_0 of q. If this node is found, there is a new recursive instance, with $N.T$ and $q \setminus \{e_0\}$ as parameters; afterwards, the function works on T and $q \setminus \{e_0\}$. If on the other hand, no node in T matches e_0, the function simply skips e_0 and continues with $q \setminus \{e_0\}$. From these observations, we get the following recursive formula for the number of search nodes that are visited by lookup_subs (which is an upper limit to the number of nodes visited by lookup_first):

$$\begin{aligned} E(i) &= P^N * (1 + 2 * E(i-1)) + (1 - P^N) * E(i-1) \\ &= P^N + (1 + P^N) * E(i-1) \end{aligned}$$

Here, P^N denotes the *probability* that the node N matching e_0 is in T, and $E(i)$, consequently, denotes the *expected* number of visited search nodes with i elements to go. Obviously, $E(1) = P^N$. It is easily proven that the recursion results in:

$$\begin{aligned} E(n) &= P^N * \sum_{i=0}^{n-1} (1 + P^N)^i \\ &= P^N * \frac{(1 + P^N)^n - 1}{(1 + P^N) - 1} \\ &= (1 + P^N)^n - 1 \qquad (1) \end{aligned}$$

The probability P^N at each stage of the search is equal to the probability that there is a set $s \in S$ which *starts* with the elements that would be represented on a path to N. It is an open question, how an upper limit for P^N can be determined. In the worst-case, when all nodes N are present in the tree, we get $P^N = 1$ and $E(n) = 2^n - 1$ search nodes for a query set of size n.[3]

[3]It should be noticed, that, when matching rules or determining unsatisfiable goals in planning, the query sets are small while $|S|$ is very large. Thus, searching 2^n nodes is still much better than checking $|S|$ sets for inclusion.

The number of nodes visited by the lookup_sups function is only dominated by the total number of nodes that are in the forest. Let e_{max} denote the maximal element with respect to the total ordering, i.e., $e < e_{max}$ for all $e \neq e_{max}$. Trying to find all supersets of the query set $\{e_{max}\}$, lookup_sups has to search the whole structure. Consequently, we determine an upper bound for the total number of nodes in the tree.

Theorem 2 *Let T be a forest in which exactly the sets $s \in S$ have been iteratively inserted. If the total number of distinct elements in all of the sets $s \in S$ is P, then the total number of nodes in T is at most $2^P - 1$.*

The worst case occurs if and only if *all* sets containing e_{max} are contained in S.

4 Empirical Evaluation

To demonstrate the effectiveness of the approach, we discuss examples taken from the use of UBTree in the IPP planning system [Koehler *et al.*, 1997]. Following [Hellerstein *et al.*, 1997], the *workload* for UBTree is determined by the following factors:

- the *domain*, which is the set of all possible sets, i.e., given P logical atoms to characterize states, the domain comprises 2^P sets,
- an *instance* of the domain is the finite subset $S \subseteq 2^P$ that is currently stored in T,
- the *set of queries*, which is the set of goals that are constructed during planning.

Note that the workload is *dynamic*. Starting with an empty UBTree structure T, a generated goal set is added if no plan was found by the planning system, i.e., T is monotonically growing containing $|S|$ sets at the end.

Since it is extremely difficult to make distribution assumptions for instances and query sets in a planning system, we use the following parameters to characterize the size of a UBTree:

- P : the total number of distinct logical atoms in S.
- $|S| = |\{s_0, \ldots, s_{k-1}\}|$: the number of all stored sets.
- $|T|$: the total number of nodes in the forest T.
- $C = \frac{|T|}{\Sigma_{i=0}^{k-1} |s_i|}$: the storage cost, which would be equal to 1 in a trivial data structure simply representing all sets separately.
- $|Q|$: the total number of queries that have been answered during the process.

Figure 7 shows the parameters for forests of increasing size in two different planning domains (the *blocksworld* shown in the upper part and the *briefcase world* shown in the lower part of the table).

The larger UBTree grows (reflected by increasing $|S|$ and $|T|$) and the smaller P is, the better values are obtained for the storage cost.

P	$\|S\|$	$\|T\|$	C	$\|Q\|$
71	4846	55035	0.69	8778
96	56398	584015	0.61	112337
115	324628	3042203	0.55	669127
33	1618	4370	0.36	5291
46	22044	59743	0.29	61909
46	92971	210666	0.23	243175
61	1058930	2007326	0.16	24369798

Figure 7: Typical sizes of UBTree.

Figures 8, 9, 10, and 11 illustrate the runtime behavior of the query functions reflected in the number of searched nodes for a given query-set size $|q|$ and $|S|$. Note that $|S|$ is shown on a logarithmic scale in all figures.

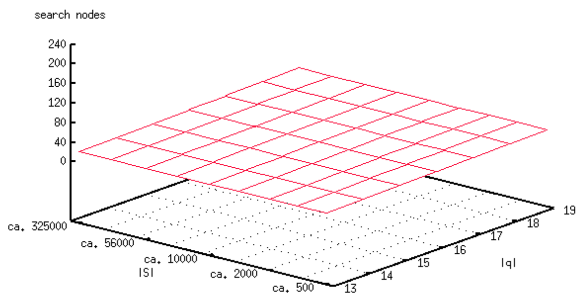

Figure 8: The average number of visited search nodes as a function of query-set size $|q|$ and instance size $|S|$ (shown on a logarithmic scale) for the lookup_first function in the positive case where the query succeeds.

Figure 8 indicates that the lookup_first function needs significantly less search time on queries where it can retrieve a subset than on those where it must fail. In fact, the average runtime behavior in the positive case was **linear** in the size of the query set and completely **independent** of the instance size throughout all our experiments. The function never visited more than an average of $|q| * 2.5$ search nodes on positive queries. In the general case, the behavior tends to be logarithmic in the instance size. The behavior with respect to the query-set size varied in different domains: sometimes clearly polynomial, even sub-linear; sometimes, as in the case of Figure 9, possibly exponential, but to a small base (as indicated by equation 1).

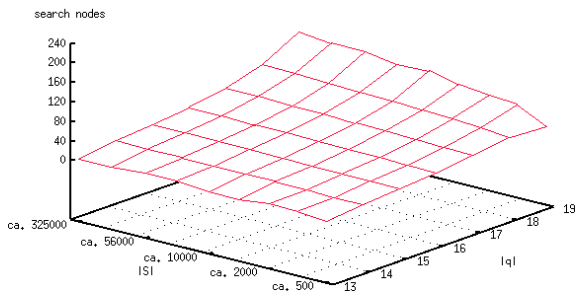

Figure 9: The average number of visited search nodes as a function of query-set size $|q|$ and instance size $|S|$ (shown on a logarithmic scale) for the lookup_first function averaged over all, i.e., succeeding and failing, queries.

A behavior as shown in Figure 10 is typical for the lookup_subs function: clearly logarithmic in $|S|$ and probably exponential, to a small base, in the query set size.

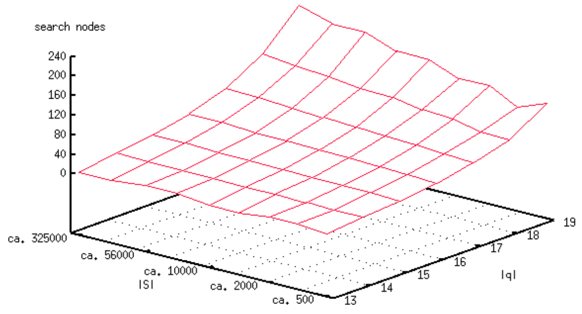

Figure 10: Number of search nodes for lookup_subs, averaged over all queries.

Figure 11 shows a typical picture for lookup_sups that we found in all investigated domains. It turns out that the performance of this function decreases for small query sets. This happens because small sets are likely to have more supersets than big ones. The behavior with respect to the instance size tends to be linear. Note that again, S is shown on a *logarithmic* scale and that the z scale is different here.

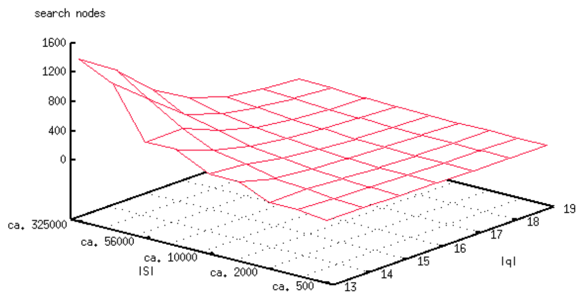

Figure 11: Number of search nodes for lookup_sups, averaged over all queries.

5 Related Work

In order to deal with set queries in *planning*, a partial subset test that only considers sets of size $|q| - 1$ to a given query set of size $|q|$ has been developed in [Blum and Furst, 1997]. Obviously, this can be done in linear time, but such a test must be inherently incomplete. With UBTree, a complete test is available that runs in almost linear time in practice despite its exponential worst-case behavior.

For *databases*, RD trees [Hellerstein and Pfeffer, 1994] have been proposed as an effective means in answering superset queries, but they are very limited in handling subset queries. In contrast to UBTree where a set is spread over several nodes, an RD tree is organized such that the leaf nodes contain the sets and non-leaf nodes contain supersets (of different size) of their children nodes to effectively guide search. To handle subset queries, inverted RD trees are used which are equivalent to RD trees on the complements of the base sets. Two serious problems occur in this approach. First, a non-leaf node needs to be recomputed if a new set is inserted into one of its leaf-children. Second, even if only small sets are stored in the leaves, their complements and their supersets in the corresponding parent nodes grow impractically large. UBTree avoids both problems.

An optimized implementation of the RETE *pattern matching* algorithm [Forgy, 1982] to handle large numbers of production rules is described in [Doorenbos, 1994]. The indexing structure for preconditions of rules is similar to UBTree, but the way how the elements of the query set match the stored sets in the indexing structure is quite different because the preconditions can contain variables, while UBTree deals with sets of ground atoms only. Thus, the problem of *null activations*, where rule nodes are activated though not all of their preconditions are satisfied by the current input, can occur in RETE, but not in UBTree, which would index all ground instances of pattern matching rules instead.

6 Conclusion

Depending on the particular requirements of an application, UBTree can be further optimized. For example, if one is only interested in keeping *minimal* sets, all non-minimal sets can be pruned from the tree. Furthermore, if all sets $s \in S$ are constructed from a finite and fixed domain of elements, an implicit bitmap representation of the sets can further reduce memory consumption and query times.

As we saw in Section 4, the worst-case behavior of the algorithms, especially of the lookup_sups function, depends on the ordering of the elements. In application areas where one has information about the likelihood of appearance of the elements in any set s, it should be possible to generate a total ordering that minimizes the number of tree nodes. One simply orders elements with high likelihood of appearance *before* those with low values. Thereby, both storage and search costs can be reduced.

References

[Blum and Furst, 1997] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298, 1997.

[Bylander, 1991] T. Bylander. Complexity results for planning. In *IJCAI-91*, pages 274–279. Morgan Kaufmann, San Francisco, CA, 1991.

[Doorenbos, 1994] R. Doorenbos. Combining left and right unlinking for matching a large number of learned rules. In *AAAI-94*, pages 451–458. Morgan Kaufmann, San Francisco, CA, 1994.

[Forgy, 1982] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Gore, 1997] R. Gore. Tableau methods for modal and temporal logics. Technical report, Australian National University, 1997.

[Hellerstein and Pfeffer, 1994] J. Hellerstein and A. Pfeffer. The RD-Tree: An index structure for sets. Technical Report 1252, University of Wisconsin Computer Sciences Department, 1994.

[Hellerstein *et al.*, 1997] J. Hellerstein, C. Papadimitriou, and E. Koutsoupias. On the analysis of indexing schemes. In *PODS-97*, pages 249–256. ACM, 1997.

[Koehler *et al.*, 1997] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *ECP-97*, volume 1348 of *LNAI*, pages 273–285. Springer, 1997.