A new Method to index and query Sets

Jörg Hoffmann Jana Koehler
Institute for Computer Science
Albert Ludwigs University
hoffmann|koehler@informatik.uni-freiburg.de

July 1998

TECHNICAL REPORT No. 108

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 three 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, and the identification of inconsistent subgoals during artificial intelligence planning. 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.

1 Introduction

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

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, 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. Usually, planners are working by backchaining from the goals, i.e., they match the effects of actions to goals and if the goals form a subset of the actions' effects, the actions' preconditions become the new goals for the planner. Even in its simplest form, the problem is known to be PSPACE-complete [2], 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 the current goal set is a *superset* of a goal set that has previously been shown to be unsatisfiable.

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.

The paper is organised as follows: In Section 2, we introduce a tree-based data structure that allows for a compact representation of a huge set of sets. We define an algorithm that indexes sets and matches the query set to the indexed sets either returning the *first subset*, *all subsets*, or *all supersets*. In Section 3, we prove the algorithm sound and complete and discuss its worst-case complexity. Section 4 shows an empirical analysis illustrating the effectiveness of the method on both the subset and superset computation. In Section 5, we compare our method to other existing approaches and in Section 6, we conclude with an outlook on further improvements.

2 The UBTree Algorithm

The key to an effective 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.

The UBTree is now simply represented by a set T of such nodes and can be seen as a forest of tree structures. 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.

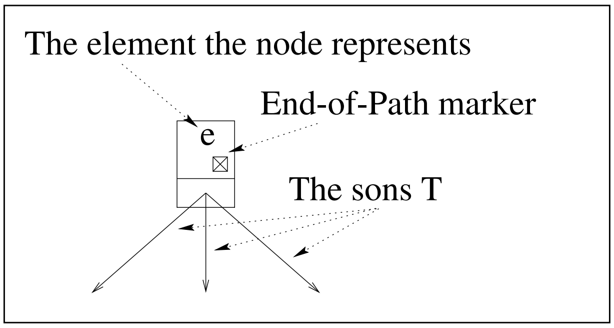


Figure 1: Representation of a single node.

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* to a given query set from T.
- lookup_sups(set q,tree T) determines *all supersets* to a given query set 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 in S, 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. $N_{n-1}.EoP = \mathsf{TRUE}$

```
sort(s) /* now s = {e_0,...,e_{n-1}} */
for all elements e_i ∈ 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)

When the function is called, T is initialised 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]

Figure 3 shows how a forest T with two trees evolves from the null node $\perp$ when the 4 (already ordered) sets $\{e_0, e_1, e_2, e_3\}$, $\{e_0, e_1, e_3\}$, $\{e_0, e_1, e_2\}$, and $\{e_2, e_3\}$ are inserted one after the other.

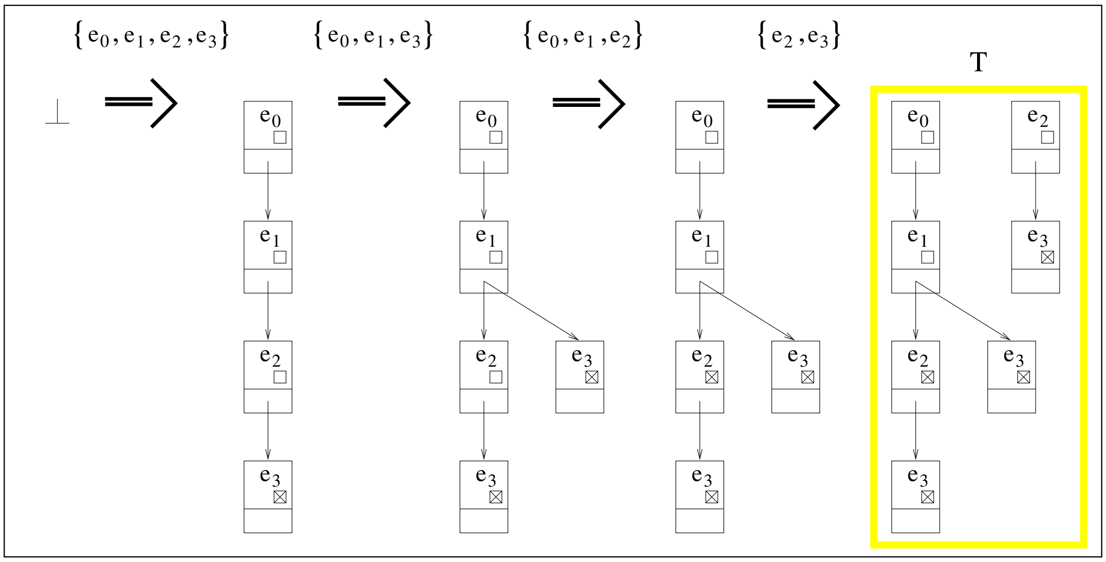


Figure 3: Iterative insertion of 4 sets

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 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] Coming back to the "money" analogy, the search traverses the whole forest, passing only those nodes for which it possesses the right money, i.e., elements.

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 $q = \{q_0, \ldots, q_{n-1}\}$

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

[2] Note that 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.

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 (due to the ordering of the elements). If there is no such node, the search space is immediately reduced to 2^{n-1}. A sketch of the lookup_first function is given in Figure 4.

```
M := all nodes N ∈ T that match an element e_i ∈ q
while M is non-empty do
        choose a node N ∈ M
        /* did we hit a subset s of q?  */
        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}, ..., 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. Note that it 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, and try the node with the least distance 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 because the query set does not contain enough elements to reach an end-of-path marker.

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

Figure 5: lookup_subs(q,T)

The lookup_subs function, as sketched in Figure 5, works in a very similar way, but instead of terminating after identifying one end-of-path marker, i.e., the first matching subset, it has to work through the whole forest, again passing 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 forest T and the ordered query set q as parameters, the answer set is initialised with the empty set $Q := \emptyset$. Upon termination of lookup_subs, Q contains all sets $s \in S$ with $s \subseteq q$.

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

At each stage of the search, lookup_sups does the following: if there are elements left in the query set, it searches all trees that start with an element *preceding* the first element in this 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 search is called on the appropriate tree and the query set gets further reduced by this element. Matching nodes for all elements have been found if at some node N the query set becomes empty. In this case, every end-of-path node that can be reached from N will yield a superset to the query. If q remains non-empty throughout the whole search, no superset to the query exists.

```
if q is empty then
  if N is the end of a path then
    Q := Q ∪ { the set that is represented by the path to N }
  endif
  call lookup_sups(N.T, q) on all nodes N ∈ T
else
  M := all nodes N ∈ T that match an element e with e < e_0
  call lookup_sups(N.T, q) on all nodes N ∈ M
  if a node N ∈ T corresponds to e_0 then
    lookup_sups(N.T, {e_1, ..., e_{n-1}})
  endif
endif
```

Figure 6: lookup_sups(q,T)

3 Theoretical Properties of UBTree

3.1 Soundness and Completeness

We state the soundness and completeness of the lookup_first function in the following theorem. The proof relies on the notion of a *path* as defined in Section 2, Definition 1.

Theorem 1 *Let T be a forest* UBTree *that has been constructed out of the null node $\perp$ 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 $s \subseteq q$.*

Proof:

$\Rightarrow$: Say the lookup_first function succeeds when visiting a node N_{m-1} in some tree out of T. Let $N_0, \ldots, N_{m-2}$ be the "partial path" to N_{m-1}, meaning that $N_0 \in T$ and $N_{i+1} \in N_i.T$ for $0 \leq i \leq m-2$. Obviously, $N_0, \ldots, N_{m-2}, N_{m-1}$ is the path $\Phi(s)$ for some set $s \in S$. Furthermore, lookup_first only visits nodes that correspond to elements of q. Therefore, $N_i.e \in q$ for $0 \leq i \leq m-1$, implying that $s \subseteq q$ holds.

$\Leftarrow$: Each set $s \in S$ has been inserted into T, i.e., there is a path $\Phi(s) = N_0, \ldots, N_{m-1}$ corresponding to s in T. If lookup_first succeeds without ever visiting N_{m-1} (implying that a different subset of q was found first, see above), we have nothing to show. If this is not the case, we prove that the function will eventually visit N_{m-1}. Then we are done: N_{m-1} is marked as the end of a path, and lookup_first will succeed when visiting it.

We prove that lookup_first will eventually visit each node N_i in $\Phi(s)$ for $0 \leq i \leq m-1$ by induction over i . Note that $m \geq 1$ as we assume s to be non-empty.

$i = 0$: N_0 corresponds to some $e_i \in s$, and with $s \subseteq q$, $e_i \in q$ holds. Furthermore, N_0 is the root node of a tree in T. Therefore, N_0 is a member of the set M that is determined by lookup_first in the initial call. As the function does not succeed without visiting N_{m-1}, which it cannot visit without having visited N_0 first, the function will eventually choose N_0.

$i \to i+1$: The argument is basically the same as in the base case: The function will eventually visit the node N_i per induction. As the search does not succeed without visiting N_{m-1}, and as N_{i+1} is a member of the set M determined by lookup_first when visiting N_i ($N_{i+1} \in N_i.T$ and $N_{i+1}.e = e_i \in q$), N_{i+1} will eventually be chosen. ■

We can use almost the same proof for a similar theorem stating the soundness and completeness of the lookup_subs function. The only difference is that we do not have to argue about the function succeeding or not, which makes the proof even easier. A very similar proof can also be constructed for the lookup_sups function.

3.2 Worst-case Complexity

As one would expect knowing that the size of the power set to any set is exponential, the UBTree algorithms must perform exponential in the size of the given sets in the worst case.

Theorem 2 *If called on a query set q of size n, the* lookup_first *(*lookup_subs*) function has to visit 2^n search nodes in the worst case.*

One can easily find an example where both functions have to visit that many nodes, ending up without finding any subsets at all. On the other hand, as both functions only visit nodes that represent elements $q_i \in q$, it should be clear that they never visit more than $2^{|q|}$ nodes.[3]

The worst-case complexity of the lookup_sups function is a little bit more difficult to determine. In principle, the number of search nodes that has to be visited is only dominated by the total number of nodes that are in the forest: Trying to find all supersets of a query set $\{e_{max}\}$ with $e < e_{max}$ for all elements e represented in T, lookup_sups has to search the whole structure. Consequently, we determine an upper bound for the total number of nodes in the tree:

Theorem 3 *Let T be a forest in which exactly the sets $s \in S$ have been iteratively inserted starting from $\perp$. 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.*

Proof: Follows immediately from the construction of worst-case UBTree structures. The worst case occurs if and only if *all* sets out of 2^P containing the "greatest element" with respect to the total ordering have been inserted into the tree. ■

4 Empirical Evaluation

To demonstrate the effectiveness of the approach, we discuss examples taken from the use of UBTree in the IPP planning system [6]. Following [4], 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 characterise states, the domain comprises 2^P sets,

[3]Constructing worst case search trees for $|q|$ elements, we get at most $2^{|q|}$ nodes.

- 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 subgoals that are constructed during planning.

Note that the workload is *dynamic*. Starting with an empty UBTree structure T, a generated subgoal set is added when lookup_first failed on it, 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 characterise 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.

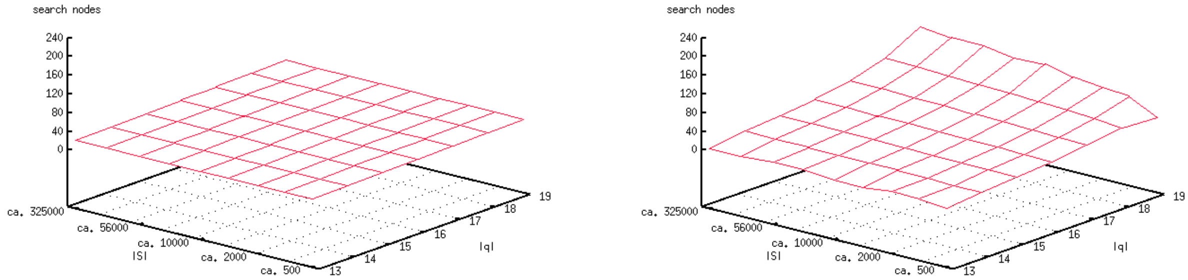
Figure 7 shows the parameters for forests of increasing size in two different planning domains (upper and lower part of the table). The larger UBTree grows (reflected by increasing $|S|$ and $|T|$) and the smaller P becomes, the better values are obtained for the storage cost.

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 and 9 illustrate the runtime behaviour of the query functions reflected in the number of searched nodes for given $|q|$ and $|S|$. Note that $|S|$ is shown on a logarithmic scale in all figures.

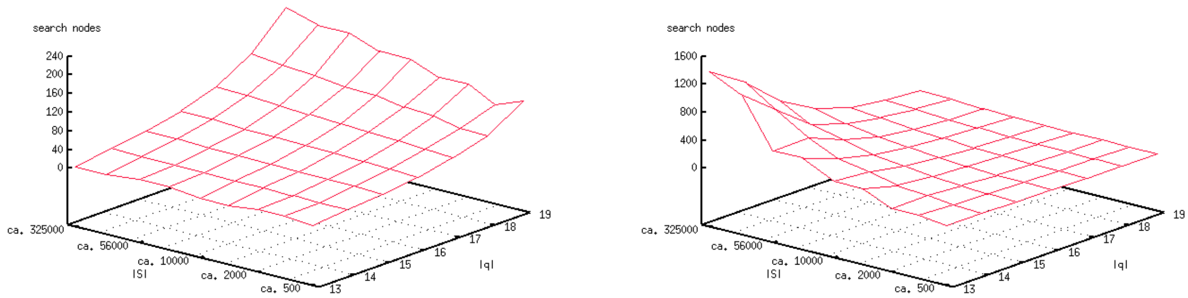
Figure 8(a) indicates that the lookup_first function needs significantly less search time on queries where it can retrieve a subset than on those where it must fail. In fact, the average runtime behaviour in the positive case was **linear** in the size of the query set and completely **independent** 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 behaviour tends to be logarithmic in the instance size. The behaviour with respect to the query-set size varied in different domains: sometimes clearly polynomial, even sub-linear; sometimes, as in the case of Figure 8(b), possibly exponential, but to a small basis.



(a) lookup_first, positive case (b) lookup_first, general case

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. Figure (a) shows the values obtained in the positive case where the query succeeds, Figure (b) shows the values averaged over all, i.e., succeeding and failing queries.

A behaviour as shown in Figure 9(a) is typical for the lookup_subs function: clearly logarithmic in $|S|$ and probably exponential in the query set size, but to a small basis. Figure 9(b) 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 behaviour with respect to the instance size remains polynomial.



(a) lookup_subs (b) lookup_sups

Figure 9: The number of search nodes for the lookup_subs and the lookup_sups functions, averaged over all queries. Note that again, S is shown on a *logarithmic* scale and that z scales are different.

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 [1]. 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 behaviour.

For *databases*, RD trees [5] 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 organised 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 to handle large numbers of production rules is described in [3]. The indexing structure for preconditions of rules is slightly similar to UBTree, but the way how the elements of the query set match the stored sets in the indexing structure is quite different and appears to be more inefficient than the UBTree algorithm, because of the problem of *null activations*, where rule nodes are activated though not all of their preconditions are satisfied by the current input.

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 behaviour of the algorithms, especially of the lookup_sups function, depends on the ordering of the elements: an element that appears frequently in the sets s should not be the maximal element with respect to the total ordering. 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 by simply ordering elements with high likelihood *before* those with low values, and thereby reducing both storage and search costs.

References

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

[2] T. Bylander. Complexity results for planning. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA, 1991.

[3] R. Doorenbos. Combining left and right unlinking for matching a large number of learned rules. Technical Report CMU-CS-94-132, Carnegie Mellon University, 1994.

[4] J. Hellerstein, C. Papadimitriou, and E. Koutsoupias. On the analysis of indexing schemes. In *Proceedings of the Conference on Principles of Database Systems*. ACM, 1997.

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

[6] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *LNAI*, pages 273–285. Springer, 1997.