Design and Implementation of Jump Tables for Fast Indexing of Logic Programs

Steven Dawson sdawson@cs.sunysb.edu

C.R. Ramakrishnan cram@cs.sunysb.edu

I.V Ramakrishnan ram@cs.sunysb.edu

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

The principal technique for enhancing the speed of clause resolution in logic programming languages such as Prolog is indexing. Given a goal, the primary objective of indexing is to quickly eliminate clauses whose heads do not unify with the goal. Efforts at maximizing the performance of indexing automata have focused almost exclusively on constructing them with small depth, which in turn translate into making fewer transitions. Performance of an automata also critically depends on its ability to make each transition efficiently. This is a problem that has largely been ignored and constitutes the topic of this paper.

Although using jump tables ensure that each transition can be done in constant time they are usually very sparse and hence are seldom used in implementations. We describe a novel method to construct dense jump tables for indexing automata. We present implementation results that show that our construction indeed yields jump tables that are dense enough to be practical. We also show that indexing automata that use jump tables based on our method, improve the overall performance of Prolog programs. We also provide experimental evidence that our method is a general technique for compressing transition tables of other finite state automata such as those used in scanners.

Contact author: Steven Dawson
E-mail: sdawson@cs.sunysb.edu
Tel: (516) 632-8470
Fax: (516) 632-8334

1 Introduction

Optimizing clause resolution is a problem of considerable importance for efficient evaluation of resolution-based logic programs. The principal technique used for enhancing the speed of clause resolution steps is indexing. When resolving a goal, a clause becomes applicable if its head unifies with the goal. The sole objective of indexing is to quickly eliminate many clauses that are not applicable. Indexing can yield substantial gains in speed, since it reduces the number of clauses on which unification will be performed and can also avoid the pushing of a choice point. These benefits of indexing have long been recognized, and considerable effort has been made to develop fast and effective indexing techniques for logic programs (see [CRR92, Car87, HM89, PN91, RRW90] for example). The typical approach underlying high performance techniques is to preprocess the clause heads of a logic program into an indexing trie. The essential idea is to partition terms based upon their structure. A tree is formed, and at each point where two terms have different symbols, a separate branch for each term is added. (See Figure 1 below.)

```
p(a,a).
p(a,b).
p(a,d).
p(b,d).
p(b,b).
p(b,e).
p(c,e).
```

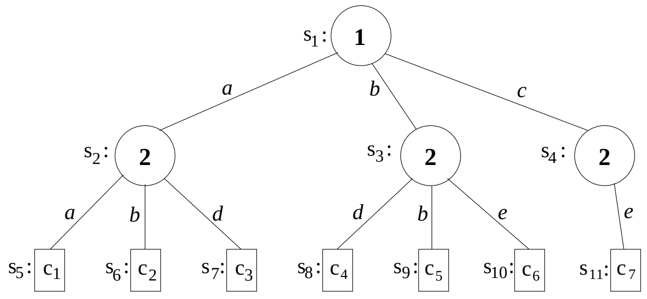

Figure 1: Predicate and indexing trie

Observe that the trie is organized as a tree-structured finite-state automaton with the root as the start state, leaves being the final states, and the edges, denoting transitions, representing elementary comparison operations. Each state specifies a position in the goal to be inspected upon reaching that state. The edge labels on the outgoing transitions specify the function symbols expected at that position. A transition is taken if the symbol in the goal at that position matches the label on an outgoing edge. We associate with each final state a set of clauses called its *index* set. On reaching a leaf state, the clauses in its index set are selected for resolution with the goal term. Note that the index set is a superset of the clauses that are applicable. Although our simplistic description above does not discuss variables, several variants to the indexing trie have been proposed for handling them (see [RRW90] for example). Such automata are routinely used to index terms in functional programming [SRR92], automated deduction [McC92] and term rewriting systems [Chr89].

Advances in indexing techniques have resulted in substantial improvement in the execution time of Prolog programs [DRR$^+$95], so much so that indexing time constitutes a significant part of the overall execution time. Thus, devising faster indexing techniques without compromising their effectiveness is a problem of considerable importance for implementing high-performance logic programming systems. Research on fast indexing methods have traditionally focused on techniques for constructing tries of small depth. Indexing based on such tries translates into making fewer state transitions for selecting an index set, thereby improving indexing time. Indexing times can be further improved by executing each transition efficiently. Techniques that facilitate efficient state transitions for indexing have not been well researched so far and constitutes the topic of this paper.

Recall that, in order to make a transition from the current state, the corresponding goal symbol in the position specified by the state must match the edge label on one of its outgoing transitions. A simple sequential search of the edge labels for a match is clearly very time consuming. Therefore, Prolog systems, such as Sicstus, ALS, and XSB, typically resort to hash-based state transitions. Although hashing reduces the time needed to make a state transition, the possibility of collisions precludes guaranteeing uniform transition time. To reduce collisions substantially, one must resort to complex hashing schemes such as perfect hashing, which guarantee no collisions, and hence, constant transition time [FKS84]. The main problem with such hash functions is that they involve several time-consuming operations, typically including modulus with respect to prime numbers. Furthermore even determining such hash functions is computationally expensive [CHK85]. Therefore, implementations often use simple hashing methods, such as bit masking. Although such hash functions can be computed quickly, they lead to frequent collisions, and hence, non-uniform transition times.

Executing any state transition in constant time can also be guaranteed by directly accessing the next state through jump tables. In this method each function symbol is assigned a unique positive integer as its *id*. Every state is associated with a jump table whose entries either point to next states or are void (denoted by $\perp$). The indexing automaton uses the id assigned to the input symbol inspected in the current state, to directly index into its jump table to make a transition to the next state. Note that it is necessary to store only the segment of the jump table containing valid transitions. If the id of an input symbol indexes into this segment, and the corresponding transition is valid, this transition is taken. Otherwise, indexing failure occurs. For example, in Figure 1 suppose we assign integers $1, 2, 3, 4$, and 5 to the symbols a, b, c, d, and e respectively. The ranges of indices for the jump tables in states s_1, s_2, s_3 and s_4 are $[1..3], [1..4], [2..5]$ and $[5..5]$, respectively, and the jump tables are $[s_2, s_3, s_4]$,$[s_5, s_6, \perp, s_7]$,$[s_9, \perp, s_8, s_{10}]$, and $[s_{11}]$.

Although jump tables facilitate fast transitions, they are seldom used in implementations because they tend to be sparse, *i.e.,* contain many void entries. In the example above the jump tables for states s_2 and s_3 have void entries. For jump tables to become practical, it is critical that they be densely populated. Suppose in the example above we had instead assigned $1, 2, 3, 4$, and 5 to e, d, b, a, and c, respectively. The ranges of jump table indices for states s_1, s_2, s_3, and s_4 would be $[3..5]$, $[2..4]$, $[1..3]$, and $[1..1]$; and the jump tables would be $[s_3, s_2, s_4]$, $[s_7, s_6, s_5]$, $[s_{10}, s_8, s_9]$, and $[s_{11}]$. Note that these jump tables contain no void entries. The problem now is to devise a numbering scheme for symbols that minimizes the total number of void entries in all jump tables. This has remained open, and is addressed in this paper.

Jump tables can be used in conjunction with hashing. This is particularly useful when, for example, the best numbering scheme still yields sparse jump tables for some states, or a state makes transitions on symbols for which numbering is inappropriate (*e.g.*, integers). Thus jump tables with numbering can be viewed as a complementary scheme to hash tables, and avoids exclusive reliance on hash tables.

Summary of results

1. We show that the problem of assigning numbers to symbols, such that the total number of void entries in jump tables is minimized, is NP-complete (Section 2).

2. We present an efficient heuristic, called the Symbol Numbering Method (SNM), that reduces the number of void entries (Section 3).

3. We provide strong experimental evidence for the practical use of jump tables. The results show that jump tables improve the overall execution time of Prolog programs. Furthermore, the space efficiency of jump tables obtained using SNM is often better (and no worse) than that of hash tables (Section 4).

4. We discuss reasons for the improved space efficiency of jump tables over hash tables and offer ways of further improving the performance of jump tables (Section 5).

5. We argue that SNM is a useful scheme for compressing transition tables of general finite state automata, such as those used in scanners and parsers. In particular, we present experimental results that demonstrates its effectiveness in compressing transition tables (Section 6).

2 Space Minimization of Jump Tables

In this section we formalize the problem of determining the numbering scheme that minimizes the space of jump tables and study its computational complexity.

Notation An indexing trie is a finite state automaton, with S as its set of states, and whose symbols are drawn from an enumerable set $\mathcal{F}$. A transition from a state s on input symbol α to a destination state d is denoted by (α, d). There is a special state $\perp \in S$ with no outgoing transitions, called the *fail state* of the automaton. The set of symbols on which successful transitions can be made from the state $s \in S$, called the *label set* of s, is denoted by λ_s. Let $\tau_s = \{(\alpha_1, s_1), (\alpha_2, s_2), \ldots (\alpha_n, s_n)\}$, where $\alpha_i \in \mathcal{F}$ and $s_i \in S$ denote the set of outgoing transitions in the state $s \in S$. The set of outgoing transitions is realized as *jump tables*, formalized as follows.

Jump Tables Let $\nu : \mathcal{F} \rightarrow N$ be a one-to-one function, called the *numbering* function, that maps each symbol in the alphabet to a distinct natural number. Since $\mathcal{F}$ is enumerable, clearly such a function exists. Let $min_s = \min(\{\nu(x) \mid x \in \lambda_s\})$, and $max_s = \max(\{\nu(x) \mid x \in \lambda_s\})$. The function $jt_s : \{i \in N \mid min_s \leq i \leq max_s\} \rightarrow S$, that represents the jump table associated with state s, is defined as:

$$jt_s(\nu(x)) = \begin{cases} \tau_s(x) & \text{whenever } x \in \lambda_s \\ \perp & \text{otherwise} \end{cases}$$

Note that $max_s - min_s + 1$ is the size of the jump table of state s. The number of void elements in the table is $(max_s - min_s + 1) - |\lambda_s|$.

Let s denote the current state of the automaton and y denote the input symbol inspected in the current state, *i.e.*, the input symbol. Let *next_state*(s, y) be the destination state. Using jump tables, the function *next_state* is defined as:

$$next\_state(s, y) = \begin{cases} jt_s(\nu(y)) & \text{if } min_s \leq \nu(y) \leq max_s \\ \perp & \text{otherwise} \end{cases}$$

Note from the above definition that transition to the destination state can be made in constant time. Given a numbering function ν and an indexing automaton with states S, the total space occupied by jump tables, *Space*(ν, S) is given by

$$Space(\nu, S) = \sum_{s \in S} (max_s - min_s + 1)$$

The total size of jump tables, and hence the total number of void entries in the tables, varies with the numbering function. We now show that the problem of minimizing the total size of jump tables is NP-complete.

Complexity of Jump Table Space Minimization

The corresponding decision problem of jump table space minimization can be stated as

> Given an automaton with states S and a positive integer K, is there a numbering function ν such that $Space(\nu, S) \leq K$?

Membership of the above problem in NP is obvious. Using transformation from Optimal Linear Arrangement [GJ79], we show:

Theorem 1 *Jump Table Space Minimization is NP-complete.*

Proof: The NP-complete problem of Optimal Linear Arrangement is stated as follows: Given a graph $G = (V, E)$ and a positive integer M, is there a one-to-one function $f : V \rightarrow \{1, 2, \ldots, |V|\}$ such that $\sum_{(u,v)\in E} |f(u) - f(v)| \leq M$?

Given any instance of the above problem, it can be easily transformed into an instance of Jump Table Space Minimization as follows: construct an automaton with states S such that for each edge $(u, v) \in E$ there is a unique state s in S with $\lambda_s = \{u, v\}$. Let $K = |E| + M$. Clearly, the transformation can be done in polynomial time. We now show that a function f exists iff there is a numbering function ν for the automaton with states S satisfying the constraints of the jump table minimization problem.

if: Define $f(u) = |\{x \in V \mid \nu(x) \leq \nu(u)\}|$. Clearly, $f : V \rightarrow \{1, 2, \ldots |V|\}$ is one-to-one, and $|f(u) - f(v)| \leq |\nu(u) - \nu(v)|$. Since ν is a solution to the jump table space minimization problem, ν is such that $\sum_{s\in S}(\nu(u) - \nu(v) + 1) \leq K$ where $\{u, v\} = \lambda_s$ and $u < v$. Hence $\sum_{(u,v)\in E} |\nu(u) - \nu(v)| + 1 \leq K \Rightarrow \sum_{(u,v)\in E} |f(u) - f(v)| \leq \sum_{(u,v)\in E} |\nu(u) - \nu(v)| \leq K - |E| = M$.

only if: Let $g(x) : \mathcal{F} \rightarrow \{|V| + 1, |V| + 2, ...\}$ be a one-to-one function. Clearly, such a function exists since $\mathcal{F}$ is enumerable. Define $\nu(x) = f(x)$ if $x \in V$ and $\nu(x) = g(x)$ otherwise. Now, $\nu(x) - \nu(y) = f(x) - f(y)$ for all $x, y \in V$. Since f is a solution to the optimal linear arrangement problem, $\sum_{(u,v)\in E} |f(u) - f(v)| \leq M \Rightarrow \sum_{s\in S}(max_s - min_s) \leq M$ from transformation. Hence, $Space(\nu, S) = \sum_{s\in S}(max_s - min_s + 1) \leq M + |E| = K$. ▮

3 Symbol Numbering Method (SNM)

As shown in the previous section, it is unlikely that there is an efficient algorithm for finding minimum space jump tables. In this section we describe the Symbol Numbering Method, an efficient heuristic for reducing jump table size that, in practice, yields small jump tables. The heuristic is based on two observations. First of all, the space taken by jump tables depends more on the numbers assigned to frequently occurring symbols than on the numbers assigned to other symbols. Secondly, the space tends to depend more on tables having fewer entries than on tables having more entries. This is because larger tables will necessarily have a wider range of numbers, and hence, more flexibility in the numbering of its component symbols. These attributes of the symbols and tables are captured naturally by a bipartite graph representation, called the *index* *symbol graph* of the program, where the nodes represent symbols and tables, and the edges denote membership of the symbols in the tables.

More formally, let $C = \{\lambda_1, \ldots, \lambda_m\}$ be the collection of label sets of all the states in the indexing automata of all the predicates in the program. Let $A = \{\alpha_1, \ldots, \alpha_n\} = \bigcup_{1 \leq i \leq m} \lambda_i$ be the set of all symbols in the automata. Let $G = (V, E)$ be an undirected graph, where $V = \{v_1, \ldots, v_n, v_{n+1}, \ldots, v_{n+m}\}$ and $E = \{(v_i, v_{n+j}) \mid \alpha_i \in \lambda_j\}$. We refer to vertices $v_1, \ldots, v_n$ as *symbol* vertices, and vertices $v_{n+1}, \ldots, v_{n+m}$ as *table* vertices. Note that the jump table for a state has valid entries only in positions corresponding to the symbols in the state's label set. The degree of symbol vertex v_i is equal to the number of label sets in which symbol α_i appears, and the degree of table vertex v_{n+j} is equal to the number of distinct symbols appearing in the label set λ_j.

For example, consider the seven simple predicates in Figure 2a. The indexing automaton for each predicate has one state. The index symbol graph of the program is given in Figure 2b. The symbol vertices are labeled above by the symbols occurring in the predicates. The table vertices are labeled below by the names of the associated predicates. The edges connect each table vertex with the symbol vertices of every symbol that is a member of the corresponding label set. For example, the table vertex labeled "*p6/1*" has three incident edges, connecting it to symbol vertices a, e, and f.

The index symbol graph provides a means of extracting certain "binding" relationships among the symbols. Informally, we say that two symbols are tightly bound to one another if a small difference between their assigned numbers leads to void entries in the jump tables. The larger this difference can become without leading to void entries, the less tightly bound the symbols are considered to be. For example, symbols e and f in Figure 2 are tightly bound, since any difference greater than 1 in their numbers would lead to void entries in the jump table for *p3/1*. On the other hand, symbols h and i are not as tightly bound, since their numbers could potentially differ by as much as 4 without resulting in any void entries. The heuristic, informally described below, is designed to account for these binding relationships, based on the two observations mentioned at the beginning of this section. The heuristic works in two stages.

In the first stage the index symbol graph is used to determine a rough ordering of the symbols, based on their frequency of occurrence and tightness of binding to other symbols. The result is one or more tree structures, which are traversed in the second stage to perform the actual numbering.

Stage 1 The first stage of the heuristic consists of a depth-first search of the index symbol graph, resulting in a spanning tree of each connected component in the graph. Note that, since index symbol graph is bipartite, in a depth first search we will alternately visit table and symbol vertices. At each step in the depth-first search, the choice of which vertex to visit next is guided by the two observations outlined at the beginning of this section. If the next vertex to be visited is a symbol vertex, then the *maximum degree* symbol vertex is chosen; if the next vertex is a table vertex, then the *minimum degree* table vertex is chosen. The depth-first search begins at the symbol vertex of highest degree, say v_i. Thus, the next vertex to be visited is the table vertex of lowest degree connected to v_i, and we proceed by alternately visiting symbol vertices of highest degree and table vertices of lowest degree. When a symbol vertex is visited, the degree of each table vertex to which it is connected is decreased by one. This reflects the increasing binding among the symbols in the corresponding label sets that remain to be visited.

The index symbol graph in Figure 2b illustrates the first stage of the heuristic. Each vertex is

```
p1(a).                                  p5(a).
p1(b).  p2(g).                          p5(b).
p1(c).  p2(h).  p3(f).  p4(g).          p5(c).  p6(a).  p7(g).
p1(d).  p2(i).  p3(e).  p4(k).          p5(d).  p6(f).  p7(k).
p1(e).  p2(j).                          p5(e).  p6(e).
p1(f).  p2(k).                          p5(f).
```

(a)

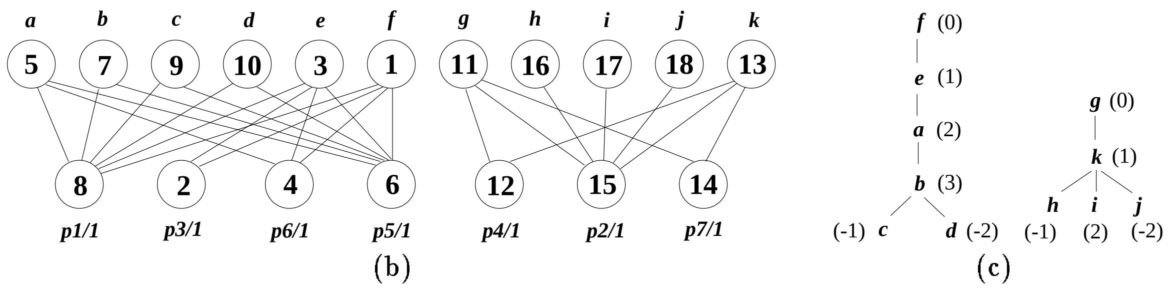

Figure 2: Seven simple predicates (a), the index symbol graph (b), collapsed spanning trees (c)

labeled according to the order in which it was visited in the depth-first search. The table vertices are not used in the second stage and are not recorded in the spanning tree. For example, the spanning trees obtained after stage 1 for the index symbol graph in Figure 2b are shown in Figure 2c.

Stage 2 The numbering of each tree (connected component) can be done independently, since the differences between numbers of symbols from distinct connected components have no effect on the size of the jump tables. We begin by assigning the number 0 to the symbol represented by the root. We then perform a preorder traversal of the tree, numbering each symbol as we visit its vertex. Note, however, that the actual numbers assigned may not mirror the preorder traversal. Observe that each chain of vertices between two branch points in the tree is such that a parent symbol occurs in all label sets that contain the child symbol. We number such groups sequentially, and "position" the group so that the distance of the group from the first symbol on its parent chain is minimized. This means that the group will be numbered increasing from the greatest number yet assigned, or decreasing from the least number yet assigned.

Consider the numbering of the left tree in Figure 2c. There is a chain of symbols *f, e, a, b,* numbered sequentially starting from 0. Next, symbol *c* is numbered -1, minimizing its difference with f. Finally symbol d is numbered -2, also minimizing its difference with f[1]. The other tree is numbered similarly. It is easily verified that this numbering yields jump tables for this example with no void entries, and hence, no wasted space.

Complexity Let n denote the number of symbols, m the number of tables, and $s = \Sigma_{j=1}^{m} |\lambda_j|$ the total size of all label sets. The number of vertices in G is $n + m$ and the number of edges is s. Note that $n + m \leq 2s$. Thus, the space required for the bipartite graph is $O(s)$. Even if the spanning trees are constructed separately, the space is increased by at most a constant factor.

The time required to perform the heuristic is given by the time needed to construct the index

[1]Note that this gives a relative numbering among symbols in one tree. The final, global numbering among all symbols is easily obtained from the relative numbering simply by adding an appropriate constant to the number of each symbol in a tree.

symbol graph, plus the time needed to perform the depth-first search, plus the time to traverse each connected component (for numbering). The time required to build the index symbol graph is proportional to its size, $O(s)$. The time to select a vertex to visit is bounded by the maximum degree of any vertex. A symbol may appear in at most m label sets, and any label set may contain at most n symbols. Thus, the maximum degree of any vertex is $O(\max(m,n))$, and the time required for the depth-first search is $O(s \cdot \max(m,n))$. The time required to traverse the trees for numbering is clearly bounded by the depth-first search time. Thus, the overall time for the heuristic is $O(s \cdot \max(m,n))$.

4 Performance

The Symbol Numbering Method has been implemented, and the performance of jump tables obtained using SNM was compared to two hash-based indexing schemes in the XSB system on several benchmark programs[2]. The programs include a sample of the border predicate from the CHAT-80 system, three queries on the Dutch national flag program (dnf), three simple parsers (ll1, ll2, and ll3), a theorem prover (dboyer) and map coloring program (map) from a set of Andorra-I benchmarks, and three queries on a 5000-fact sample of a chemical database (synchem). Each of the programs was compiled using unification factoring [DRR$^+$95], which, with the exception of dboyer, substantially improves the performance of the programs, due in large part to improved indexing.

The two hashing methods tested were modulus with respect to a prime number (mod hash) and bit masking (bit mask hash). Table 1 compares the overall query evaluation times for jump tables to those of the two hashing schemes. Two times are reported for each hashing method. The first is the lowest observed time across a large number of runs, and the second is the highest observed time for which the time increase could be attributed to hash collisions[3]. The jump table figures show the execution time (which is constant across all runs) as well as the speedups compared to hashing. Observe that jump table indexing is always as fast as the fastest hashing results and is often substantially faster than the slower hashing results. Mod-based hashing consistently gives the slowest performance (but more consistent performance than bit mask hashing), due to a particularly expensive mod operation on Sparc machines. This is, of course, architecture dependent, but it does indicate that perfect hashing schemes, which rely even more on mod operations, are unlikely to perform well in practice. Bit mask hashing, while offering performance comparable to jump tables in the best case, gives varying results due to (unpredictable) hash collisions.

Further light can be shed on these numbers by the considering the execution time of the different indexing instructions. On average, a single mod-based hashing required 2.20 μs, while one bit mask hashing instruction took 0.50 μs. The jump table indexing instruction, which involves two bounds checks followed by a table lookup operation, took 0.62 μs. However, note that any hashing scheme requires some means of verifying the value of the hashed symbol. In the absence of any collision, this verification takes an additional 0.32 μs. Jump table indexing needs no such verification, and hence is faster than either hashing instruction, even in the absence of collisions. The speed of jump table indexing is more apparent in programs where a relatively large portion of the time is spent in indexing (*e.g.,* map).

[2]All benchmarks were executed on a Sun SPARCstation 20 running SunOS 5.3, using XSB version 1.4.2. The benchmarks programs are available by ftp from `ftp.cs.sunysb.edu` in `pub/XSB/benchmarks`

[3]Hashing in XSB is performed on symbol pointers whose values can vary from one run to another.

Program	Mod hash Time Low/High	Bit mask hash Time Low/High	Jump tables Time	 Speedup (mod) Low/High	 Speedup (bit) Low/High
border	4.29/4.29	3.70/3.70	3.63	1.18/1.18	1.02/1.02
dnf(s1)	4.43/4.43	3.75/5.88	3.68	1.20/1.20	1.02/1.60
dnf(s2)	4.31/4.31	3.66/5.63	3.59	1.20/1.20	1.02/1.57
dnf(s3)	5.05/5.05	4.25/5.00	4.18	1.21/1.21	1.02/1.20
ll1	4.93/6.44	4.24/5.70	4.23	1.17/1.52	1.00/1.35
ll2	5.06/5.06	4.15/5.20	4.13	1.23/1.23	1.00/1.26
ll3	4.24/4.24	3.57/4.40	3.55	1.19/1.19	1.01/1.24
dboyer	4.20/4.20	4.10/4.10	4.03	1.04/1.04	1.02/1.02
map	4.44/5.48	2.58/3.62	2.33	1.91/2.35	1.11/1.55
synchem(alc)	4.49/4.49	3.99/3.99	3.88	1.16/1.16	1.03/1.03
synchem(eth)	4.32/4.32	3.26/3.26	3.21	1.35/1.35	1.02/1.02
synchem(CC)	5.05/5.05	4.65/4.65	4.40	1.15/1.15	1.06/1.06

Table 1: Comparing CPU times for jump table indexing with two hashing schemes

It might be expected that the fast, guaranteed time performance of jump tables must come at some expense in space, and with a naive numbering of the symbols, this might be true. However, our experiments show that, using SNM, even space utilization can be improved. Table 2 compares the space usage and efficiency of jump tables with that of the two hashing methods. Efficiency was computed as the ratio of the minimum space required by any indexing table to the space actually used. For the above benchmarks, jump tables were always at least as space efficient as either hashing method and often gave 100% efficiency. To achieve comparable space performance in the hashing schemes wherever possible, more collisions would necessarily result, giving worse time performance. On the other hand, for hashing to approach the consistent time performance of jump tables would require an even greater sacrifice in space.

It should be pointed out, however, that examples can be constructed for which the space required by even the smallest possible jump table may be too great. Furthermore, due to the presence of symbols that cannot be numbered, such as integers, it may not be possible to compress the jump tables further. In such cases it is reasonable to use a fast hashing method (such as bit masks) as an alternative. For example, in the synchem benchmark, nearly half the tables contained integers and hash-based indexing was used in these cases. The jump tables obtained by SNM for the remaining cases were more than 90% space efficient. The use of jump tables for these cases improved query evaluation times, while increasing the overall space efficiency to over 60%.

Finally, it should be noted that SNM itself, which is done at predicate load time, adds little overhead. In the worst case (synchem), SNM and jump table creation added less than 20% to the (small) load time. In summary, the performance results show that

- jump tables offer guaranteed time performance, and this performance is as good as or better than even the simplest hashing methods;
- using SNM, jump tables need not incur a large space penalty, and may even improve space efficiency;

Program	Mod hash		Bit mask hash		Jump tables	
	Bytes Low/High	Efficiency Low/High	Bytes Low/High	Efficiency Low/High	Bytes	Efficiency
border	68/68	0.47/0.47	64/64	0.50/0.50	32	1.00
dnf	28/28	0.43/0.43	32/32	0.50/0.50	12	1.00
ll1	56/88	0.27/0.43	32/80	0.30/0.75	24	1.00
ll2	60/108	0.22/0.40	48/64	0.37/0.50	24	1.00
ll3	60/108	0.22/0.40	48/64	0.37/0.50	24	1.00
dboyer	912/1000	0.41/0.45	864/904	0.46/0.48	764	0.54
map	188/188	0.34/0.34	96/144	0.44/0.67	72	0.89
synchem	72792/85640	0.35/0.41	59080/71640	0.42/0.51	48436	0.62

Table 2: Comparing space usage for jump tables with two hashing schemes

- jump tables are more practical than perfect hashing schemes – perfect hashing functions are expensive to derive at compile/load time and, since they depend on mod operations, expensive to use at run time.

5 Space Utilization of Jump Tables

Traditional Prolog implementations do not use jump tables, primarily because of possible space degradation. However the experimental results presented in Section 4 show that indexing using jump tables usually outperforms hash-based schemes even in terms of space efficiency. In this section, we characterize the space utilization of the hashing methods used in well-known Prolog systems, and compare them with the space behavior of jump tables. We identify situations where jump table based schemes perform very well and where degradation is possible. We also suggest ways to prevent such degradation.

5.1 Hash Tables vs. Jump Tables

In a hash-based indexing scheme, the size of the hash table and the number of collisions determine the space needed to represent all the outgoing transitions. In Prolog implementations, the hash function (and hence the size of the hash table) is chosen based on the number of transitions. Note that in order to reduce the possibility of collisions, the size of the table must be considerably larger than the number of transitions. Thus the space efficiency, denoted by e, which is the ratio of number of transitions to the total space needed to represent all the transitions, is bounded for hash tables. The bounds themselves may vary, depending on the hash function used and how collisions are handled.

In XSB and ALS Prolog, which use mod hashing, the hash table size is a prime number greater than the number of transitions. In Sicstus Prolog, which uses bit mask hashing, the table size is a power of two, usually larger than three times the number of transitions[4]. ALS and Sicstus explicitly maintain set of buckets for each hash entry. While ALS stores the buckets in a binary

[4]We use Sicstus v2.1 #9 (compact code) for all these comparisons. Larger table sizes are needed to offset the number of collisions incurred by the simpler masking hashing function.

Method		Space Efficiency (e)
Jump Table		$0 < e \leq 1$
Sequential Search		$e = 0.5$
Hashing	– XSB	$0.167 \leq e < 1$
	– ALS	$0.20 \leq e < 0.25$
	– Sicstus	$0.136 < e \leq 0.214$

Figure 3: Bounds on Space Efficiencies.

tree, Sicstus uses a list. In XSB, however, the hash table points directly to code, instead of pointing to a list of buckets. When there is no collision on a hash entry, the table points to the code for the corresponding clause, which contains unification instructions for all positions in the head of the clause. This code is a part of the try-retry-trust chain that is invoked when the indexed position in the goal is a variable, and hence does not contribute to any additional space overhead. When there is a collision on some entry, a new try-retry-trust chain is created for all the colliding symbols.

Based on the hash functions and the collision management strategy used, we can readily compute the bounds on the space efficiencies of the various hashing schemes. These bounds, as well as those on sequential search and jump table strategies, are given in Figure 3. Observe, from the table, that hash tables guarantee a definite lower bound on space efficiency whereas jump tables offer no such lower limit. On the other hand, the upper bound efficiencies of hash tables are low, while jump tables can achieve 100% efficiency. More interestingly, the performance results in section 4 indicate that, using SNM, jump tables often achieve 100% efficiency. We now discuss the reasons underlying their good space performance, and suggest ways to further improve their space as well as time performance.

5.2 Impact of Type Discipline on Performance of Jump Tables

In the case of well-typed programs, the set of symbols on which transitions are made from any state belong to the same type. Hence, symbols of a given type can be numbered independently of symbols of a different type. This degree of freedom enables SNM to yield jump tables with nearly 100% space efficiency on well-typed programs. It turns out that, although Prolog is an untyped language, programs generally follow an implicit typing discipline. This is the main reason that the jump tables of indexing automata for Prolog programs attain high space efficiencies using SNM.

We can exploit knowledge of types in a program for improving the access time of jump tables even further. For instance, consider a state with the labels a, b and c on its outgoing transitions. If we have type information that whenever this state is reached, the symbol inspected in the input goal belongs to the set $\{a, b, c\}$ (another characteristic of well-typed programs), we can omit the bounds check, and directly index into the jump table. Note that transitions thus made cost no more than a table lookup.

5.3 Inter-module Application of SNM

Our development of SNM was based on complete knowledge of the label sets of all the states in the indexing automata for every predicate in the program. Hence SNM cannot be used at compile time in systems that support separate compilation. However, if all modules are loaded statically, *i.e.*, before the program is executed, SNM can be used at load time. We now naturally extend SNM to systems where modules are loaded dynamically as follows.

When a new module is being loaded, symbols numbered in previously loaded modules cannot easily be renumbered. Hence SNM assigns numbers, distinct from numbers previously used, only to the new symbols in the current module. Due to a loss of type discipline across modules, incremental application of SNM to each load module may yield sparse jump tables. In practice, such loss occurs primarily due to "accidental" synonyms – when the same symbol is used to represent different objects in different modules. Note that accidental synonyms arise only in module systems that are not atom-based, *i.e.*, where all function symbols are drawn from a global space. An atom-based module system that allows the user to intuitively specify the scope of symbols greatly reduces the frequency of this phenomenon, and improves the space efficiency of jump tables with SNM.

6 Generality of SNM

In the previous sections, we have shown the effectiveness of SNM in making jump tables practical for indexing in Prolog programs. We now demonstrate the utility of this technique for compressing transition tables of finite state automata that arise in applications such as parsers and scanners. In these applications, transitions can be stored in a two-dimensional table, indexed by the pair *(state, input symbol)* and transitions can be made with one table lookup operation. However, since transition tables are large in general, a suite of table compression techniques with various time-space tradeoffs have been developed (see [DDH84] for a good introduction). In the following, we first argue why these techniques are not applicable for minimizing jump tables in indexing automata.

6.1 Table Compression Methods

We start with a brief description of compression techniques, currently in widespread use, that guarantee fast constant-access times. These techniques can be broadly classified into *content-based* methods and *structure-based* methods.

Content-based Compression Methods Content based methods exploit the similarity between transitions in different states or on different input symbols. An important compression technique used in scanners, called the *Equivalent Symbols* method (ESM), is based on the observation that many sets of symbols exhibit identical lexical properties. For instance, in many scanners the digits '0', '1', ..., '9' are indistinguishable; *i.e.*, from any state, the destination states on the transitions on '0', ..., '9' are the same. Such symbols are put in one equivalence class, and the transitions on these symbols are replaced by *one transition* labeled by the corresponding equivalence class. The compressed table is represented as a two-dimensional array, indexed by state and equivalence class of symbols.

Two effective content-based techniques for compressing parser tables are *Line Elimination* [Bel74] and *Graph Coloring* [DDH84]. The Line Elimination method is based on the observation that there are many states in an LR parser in which there is only one valid action, regardless of the input symbol. Similarly, there are many input symbols such that the same valid action is performed, regardless of the current state. The rows and columns representing such states and input symbols are removed from the transition table, and the valid actions themselves are stored with the corresponding state or symbol. This process is repeated until the table is irredundant. The Graph Coloring method is based on the following observation: if two states do not perform contradictory valid actions on any input symbol, then their outgoing transitions can be represented by a single row. Note that this method compresses the transition table by reducing the number of states.

Structure-based Compression Methods Structure-based methods compress the tables based only on which positions in the table that represent valid transitions, and not on the transitions themselves. In a commonly used structure-based method, called the *Row Displacement* method (RDM) [AU77], the transition table is mapped to a linear array. Each row in the table is represented by a sequence of consecutive elements in the array such that valid entries of one row do not overlap with the valid entries of another row. The basic idea is to minimize the number of invalid entries in the linear array. All entries in the array are tagged with the corresponding row number, and each access now involves checking the tag to ensure the validity of the transition. The tags are usually maintained in a separate array. Methods such as the RDM are typically used to further compress tables resulting from content-based methods such as ESM.

Applicability to Indexing Automata It appears that the above table compression methods are not applicable for minimizing the jump table space of indexing automata. Firstly, the content-based compression methods exploit the sharing inherent in the automata used by scanners and parsers. But traditional trie-based indexing automata (e.g., path automata and first string automata [RRW90]) are tree structured. This means that no two transitions lead to the same destination state, and hence none of the content based methods apply. Secondly, since the number of possible input symbols is unbounded for indexing automata, fast access schemes for tables compressed using RDM are infeasible.

Even when the indexing automata are structured as DAGs [KS91], the above compression methods cannot be readily applied. Note that the graph coloring method is essentially a state minimization technique and hence yields no further compression on a DAG in which the number of states is already minimized. The line elimination method is applicable only when there is some symbol on which all states make a transition to the same destination state. Clearly, such transitions are not representable without cycles[5] and hence line elimination method is inapplicable. It is not clear if the indexing DAGs exhibit the symmetry needed for ESM to be effective. In any case, we show (in section 6.2 below) that SNM naturally generalizes ESM.

6.2 SNM as a Compression Technique

We now provide experimental evidence of the effectiveness of SNM for compressing transition tables used in scanners. It should be noted that SNM can be applied to tables already compressed using other methods, and that these methods can also be used to compress the tables produced by SNM. This interoperability makes SNM a valuable tool in the suite of compression strategies used in scanners and parsers.

[5]Transition tables may exhibit this property after factoring out error entries. However, note that error factoring involves maintaining an independent array that records which positions in the transition table contain valid transitions, and is not suited for making fast transitions.

Program	Full Table	ESM		ESM + RDM		SNM		Min. Space	
`Cdecl`	109568	29140	(73.40%)	23872	(78.21%)	14208	(87.03%)	10724	(90.21%)
`Coral`	79872	32644	(59.13%)	25944	(67.52%)	8940	(88.81%)	8460	(89.41%)
`Course`	92160	43268	(53.05%)	36380	(60.53%)	12628	(86.30%)	11676	(87.33%)
`Detex`	143360	55708	(61.14%)	40628	(71.66%)	19236	(86.58%)	15132	(89.42%)
`Equals`	154112	57424	(62.74%)	59684	(61.27%)	23288	(84.89%)	18884	(87.75%)
`Flex`	161280	56288	(65.10%)	21076	(86.93%)	14140	(91.23%)	9168	(94.32%)
`Postgres`	95232	38024	(60.07%)	38076	(60.02%)	17588	(81.53%)	14016	(85.28%)
`Storm`	66048	24064	(63.57%)	23220	(64.84%)	12172	(81.57%)	10296	(84.41%)
`Web2C`	134656	56568	(57.99%)	52728	(60.84%)	29092	(78.47%)	19324	(85.65%)

Table 3: Space usage (in bytes) of various scanner table compression methods.

SNM, as presented thus far in the paper, is a structure-based compression technique. However, it can be easily generalized to become a content-based method as follows. The numbering function was defined as a one-to-one function from the set of symbols to a set of integers. We now generalize the numbering function to be a many-to-one function with the constraint that two symbols are mapped to the same integer if and only if the they make the same transitions from every state. Two symbols are mapped to the same integer if and only if they belong to the same equivalence class (as defined by ESM), thus generalizing ESM.

Table 3 compares the space compression obtained by SNM to those obtained by traditional methods on different scanners. All scanners were automatically generated using `Flex`. The table lists the total space used to represent the transition tables and includes support structures, such as the tag table for RDM and the bounds tables for SNM. The scanners listed in the table were taken from various systems: `Cdecl`, a system for encoding C and C++ type-declarations; `Coral`, a deductive database system; `Course`, a compiler for a Pascal-like language (used in a compilers course); `Detex`, an utility to remove TeX commands from a text file; `Equals`, a functional language compiler; `Flex`, a scanner generator; `Postgres`, a database management system; `Storm`, an equational theorem prover; and `Web2C`, a Web to C translator.

In the table, the space requirements of the uncompressed transition tables are listed under Full Table. Columns ESM and ESM+RDM list the space used to represent the transition tables after applying ESM and ESM followed by RDM respectively. Column SNM lists the space used after applying SNM. The last column, Min Space, lists the minimum space needed to represent the valid transitions, and indicates the sparseness of the transition tables. Each column also lists the percentage reduction in space achieved by the compression technique (compared to the space occupied by the full table).

Observe from Table 3 the effectiveness of SNM in reducing space usage. In particular, SNM results in compression factors of over 78% in all the examples, and obtains better compression than any other compression scheme. Furthermore, the compression achieved by SNM is close to the maximum possible.

Recall that compression techniques trade table space for access time. Accessing tables compressed using ESM takes typically 20% longer than accessing full (uncompressed) tables, while compression using ESM+RDM incurs an overhead of 50%. Tables compressed using SNM typically take 65% longer time to access than uncompressed tables. Scanner generators usually compress

ESM tables using methods that do not guarantee constant access times. Such methods can achieve compression factors of more than 80% but result in access time overheads of more than 120%. Note that SNM offers a good compromise with 80% space compression and 65% access time overhead. Thus, SNM is a useful compression technique that can be added to the suite of methods currently used to compress scanner tables.

7 Conclusion

Efficient indexing is crucial for high performance logic programming systems. Advanced techniques such as *unification factoring* have resulted in substantial reduction in program execution times. With the increase in proportion of time spent in indexing, it is even more important that the elementary indexing operations themselves are as fast as possible. Jump tables are an effective means of achieving fast, constant-time transitions in indexing automata. In fact, as our experimental results show, indexing based on jump tables is significantly faster than hash-based indexing. Thus in programs that rely heavily on indexing, the overall execution times can be substantially improved. However, jump tables are seldom used in implementations since they are usually sparse. In this paper, we have addressed the problem of improving the space efficiency of jump tables, thus making the use of jump tables practical.

We devised a scheme that makes jump tables dense, by suitably numbering the symbols. We showed that the problem of determining the numbering scheme that minimizes the total space of jump tables is NP-complete, and presented an efficient heuristic, called the Symbol Numbering Method, to reduce the total space. Experimental results indicate that jump tables improve the overall execution time of Prolog programs, even compared to the fastest hashing method. Furthermore, the space efficiency of jump tables obtained using SNM is often better (and no worse) than that of hash tables. Moreover, SNM is general in the sense that it provides an effective scheme for compressing transition tables of finite state automata, such as scanners and parsers.

References

[AU77] A.H. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

[Bel74] J.R. Bell. A compression method for compiler precedence tables. In *Proc. of the IFIP Congress 74*, pages 359–362, Stockholm, August 1974.

[Car87] M. Carlsson. Freeze, indexing and other implementation issues in the WAM. In *International Conference on Logic Programming*, pages 40–58, 1987.

[CHK85] G.V. Cormack, R.N.S. Horspool, and M. Kaiserswerth. Practical perfect hashing. *The Computer Journal*, 28(1):54–58, 1985.

[Chr89] J. Christian. Fast Knuth-Bendix completion : Summary. In *RTA'89*, pages 551–555. Springer-Verlag LNCS 355, 1989.

[CRR92] T. Chen, I. V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In *Joint International Conference/Symposium on Logic Programming*, pages 639–653, 1992.

[DDH84] P. Denker, K. Dürre, and J. Heuft. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems*, 6(4):546–572, October 1984.

[DRR$^+$95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *ACM Symposium on Principles of Programming Languages*, pages 247–258, January 1995.

[FKS84] L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, July 84.

[GJ79] M. Garey and D. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completness.* W.H. Freeman and Company, 1979.

[HM89] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[KS91] S. Kliger and E. Shapiro. From decision trees to decision graphs. In *North American Conference on Logic Programming*, pages 97–116, 1991.

[McC92] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:147–167, 1992.

[PN91] D. Palmer and L. Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In *International Conference on Logic Programming*, pages 429–442, 1991.

[RRW90] R. Ramesh, I. V. Ramakrishnan, and D. S. Warren. Automata-driven indexing of Prolog clauses. In *ACM Symposium on Principles of Programming Languages*, pages 281–290. ACM Press, 1990.

[SRR92] R. C. Sekar, I. V. Ramakrishnan, and R. Ramesh. Adaptive pattern matching. In *International Conference on Automata, Languages, and Programming*, number 623 in LNCS, pages 247–260. Springer Verlag, 1992. To appear in *SIAM J. Comp.*