Optimization of Parser Tables for Portable Compilers.

Article *in* ACM Transactions on Programming Languages and Systems · October 1984

DOI: 10.1145/1780.1802 · Source: DBLP

3 authors, including:

Peter Dencker

9 PUBLICATIONS **61** CITATIONS

SEE PROFILE

Optimization of Parser Tables for Portable Compilers

PETER DENCKER, KARL DÜRRE, and JOHANNES HEUFT
Universität Karlsruhe

Six methods for parser table compression are compared. The investigations are focused on four methods that allow the access of table entries with a constant number of index operations. The advantage of these methods is that the access to the compressed tables can be programmed efficiently in portable high-level languages like Pascal or FORTRAN. The results are related to two simple methods based on list searching. Experimental results on eleven different grammars show that, on the average, a method based on graph coloring turns out best.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*portability*; D.3.4 [**Programming Languages**]: Processors—*compilers; parsing; translator writing systems and compiler generators*; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Experimentation, Languages, Performance

Additional Key Words and Phrases: Graph coloring, sparse matrices, table compression

1. INTRODUCTION

LR-parsers are currently the standard choice of compiler designers for the syntactical analysis of programs. These parsers are controlled by tables, which in their original form are too large for practical use in compilers. We survey and compare six general compression methods for sparse tables. They have been implemented and applied to parser tables generated from grammars of eleven different languages. We have focused on methods that particularly allow the efficient usage of the compressed parser tables in **portable** compilers. Here, portability means that the compiler is completely written in a portable high-level language like Pascal or FORTRAN [36]. The methods are compared for their algorithmic complexity, their compression potential, and their induced table-access overhead.

Most of the grammars were taken from real compiler construction projects without change. These grammars include Ada[1] [1], AL [23, 33], BALG [19], LEX

[1] Ada is a registered trademark of the U.S. Department of Defense.

Authors' present addresses: P. Dencker, Systeam KG Dr. Winterstein, Am Entenfang 10, 7500 Karlsruhe 21, West Germany; K. Dürre, Institut für Informatik I, Universität Karlsruhe, Kaiserstr. 12, D-7500 Karlsruhe 1, West Germany; J. Heuft, Gesellschaft für Mathematik und Datenverarbeitung mbH, Schloss Birlinghoven, D-5105 St. Augustin 1, West Germany.

[18], LIS [22], MINILEX [30], and Pascal [24]. The other grammars ALGOLW [40], ALGOL60 [34], Euler [41], and XPL [32] have been chosen for reference purposes. All grammars are published in [10].

Because many entries in LR-Parser tables are identical (i.e., error entries) or even insignificant (i.e., "don't cares"), the tables can easily be compressed to about 20 percent of their original size or less ([4], 6.8) by storing them in lists. The well-known YACC (yet another compiler-compiler) [26], for instance, uses such a technique (see Section 2.5) where the information retrieval requires a list search.

The disadvantage of list storing techniques is that a list search is required for information retrieval. This search may become more expensive than the access by index operations if (machine-dependent) list search instructions are unavailable. Binary search would be useful only if the lists are long enough to gain an appropriate trade-off. Fels' results [16] for a SIEMENS 7.755 show that a binary search beats a linear search only for more than 16 elements per list. Our experimental results show that, on the average, this boundary is exceeded only in two cases. To substantiate this, investigations of the dynamic access frequency would be necessary. Even when such instructions like IBM's "translate and test" are exploited, the search criterion is restricted in its size to 8 bits. For example, the technique stated by Pager [35] restricts the number of symbols and pseudo-symbols to 256, so that for languages like Ada no parser could be generated. For these reasons we have been looking for compression methods without such limitations and with fast constant-access time.

Since 1977 we have gained practical experience with two methods based on graph coloring [13] and line elimination [6]. They are part of our LALR(1)-Parser-Generator [9], which has proved its practicability and portability in various academic and industrial compiler projects running on different machines.

These two methods combine the properties of fast table access and machine independence for the parsers generated. They borrow from Joliat [27, 28] the idea of error matrix factoring. Joliat's method is based on automata theory. After factoring out a Boolean error matrix, he interprets the rest of the parser tables in the same way as those for an incompletely specified finite-state automaton, where the original error entries represent "don't care" conditions. His method keeps the parser tables in matrix form. The information retrieval requires some simple index operations. These index operations are basic to many common programming and machine languages.

The methods described in Section 2 ignore the semantics of the parser table entries. Therefore optimization methods based on the semantics of the entries may be applied independently. However it should be noted that it is just this kind of optimization that gives our methods their excellent space performance. A method that exploits the special structure of LALR(1) parser tables and some tuning possibilities of the different methods are discussed in Section 3 in detail. In Section 3.7 we give an overview of sophisticated extensions with which we have not experimented. The theoretical and experimental results are presented and interpreted in Section 4.

In Section 1.1 we give the notation and an example of the kind of parser table we have used. In Section 1.2 we introduce the error matrix, which is essential for the use of sparse matrix optimization methods to parser tables.

Figure 1

	T-table						N-table			
	⊥	+	*	(	)	*id*	*Z*	*E*	*T*	*F*
	1	2	3	4	5	6	1	2	3	4
1	.	.	.	2	.	*7	1	4	3	*5
2	.	.	.	2	.	*7		5	3	*5
3	−3	−3	6	.	−3	.				
4	−1	7			.					
5	.	7			*6					
6	.	.	.	2	.	*7				*4
7	.	.	.	2	.	*7			8	*5
8	−2	−2	6	.	−2	.				

1.1 Notation and Example

We take the notation from Anderson, Eve, and Horning [3] with the following convenient additions:

$V_{T'}$ denotes the union of the terminal vocabulary V_T with the endmarker ⊥,
V' denotes the union of $V_{T'}$ with the nonterminal vocabulary V_N,
Q denotes the finite set of states of a given parser.

The *T-table* and *N-table* of [2, 3 (4.2)] can now be defined as two partial mappings:

$$\begin{aligned} \text{T-table: } Q \times V_{T'} \rightarrow & \{\text{error}\} \cup \{\text{shift}\} \times Q \cup \\ & \{\text{reduce, shift-reduce}\} \times P \\ \text{N-table: } Q \times V_N \rightarrow & \{\text{shift}\} \times Q \cup \\ & \{\text{shift-reduce}\} \times P. \end{aligned}$$

Figure 1 shows the LALR(1)-parser tables for the augmented example grammar G with productions

$$\begin{array}{ll} 1\text{: } Z \rightarrow E & \\ 2\text{: } E \rightarrow E + T & 3\text{: } E \rightarrow T \\ 4\text{: } T \rightarrow T * F & 5\text{: } T \rightarrow F \\ 6\text{: } F \rightarrow (E) & 7\text{: } F \rightarrow id \end{array}$$

Termination of the parser is achieved in the example by detecting a shift-action into the first state, that is, the start state of the parser.

Unsigned integers q denote a shift action into state q,
"$*p$" denotes a shift-reduce with production p,
"$-p$" denotes a reduce with production p,
"." denotes an error entry,
" " denotes an insignificant entry.

Contrary to [2, 3] we employ the "scan-production" number entry for both T-table and N-table and refer to it as "shift-reduce" because it is a concatenation of a shift- and a reduce-action. **Insignificant** entries are those that can never be accessed during a parse. A definition for insignificant entries in the T-table may be found in [9 (3.4.2)]. In the following, however, we treat them as error entries because we have not as yet found an efficient algorithm to compute them.

	⊥	+	*	(	)	*id*
	1	2	3	4	5	6
1	1	1	1	0	1	0
2	1	1	1	0	1	0
3	0	0	0	1	0	1
4	0	0	1	1	1	1
5	1	0	1	1	0	1
6	1	1	1	0	1	0
7	1	1	1	0	1	0
8	0	0	0	1	0	1

Fig. 2. The error matrix to the T-table of Figure 1

1.2 Error Matrix Factoring

In contrast to an N-table, a T-table is not sparse by nature. It contains many error entries (see Figure 1). To make optimization methods for sparse matrices applicable, we have to make it look sparse. To this end we may factor out of the T-table the most frequent entry, as was done by Joliat [27, 28]. Obviously this is the error entry. The resulting binary matrix is called the *error matrix*. Figure 2 shows it for the T-table of Figure 1. Now the error entries in the T-table become insignificant if the T-table is only accessed when the error matrix indicates no error entry. Hence the T-table has become sparse. Note that in the following sections we refer to the error matrix as the negated **sigmap**.

2. TABLE COMPRESSION SCHEMES

Typical parser tables are either sparse, that is, they have only a few significant entries, or they may become sparse when factoring out the error entries as described above. Thus, all methods for compressing sparse tables are applicable. As entries of parser tables are not modified during parsing, we do not consider methods allowing insertion and deletion of entries. In the next sections we consider for compression a table T defined as

T:ARRAY [1 . . m, 1 . . n] OF data.

We give short descriptions of the six methods compared in the following sections. The first four methods are called **index access** methods because the access is done by index operations. The other two methods are called **list search** methods because the access is performed by searching in lists.

2.1 Graph Coloring Scheme (GCS)

In this scheme [13, 39] we use the fact that one row can be merged with another if both do not have different significant values in any column position. Thus we are looking for a partition of all rows into classes such that the rows in each class do not collide and can be merged. Such a row partition with a minimal number of classes represents an optimal compression of all table rows. This problem, however, is equivalent to the problem of looking for the minimal coloring of a graph by the construction below.

We consider a graph where each vertex uniquely represents a table row. Vertices are adjacent if and only if the respective rows collide, that is, have different significant values in at least one column position. Two vertices of this graph

having the same color in a vertex coloring represent two noncolliding rows and thus can be merged to a single row. For the coloring approximate algorithms may be used (see Section 3). Reducing T by merging all rows of each color class yields a $(g_r * n)$-table T' where $g_r \leq m$ is the number of colors used for coloring the graph representing the rows of T.

Usually the compressed table will also be sparse, and therefore the columns may be merged in the same manner giving a table

value: ARRAY [1 . . g_r, 1 . . g_c] OF data

(where $g_c \leq n$ is the number of colors used for coloring the graph representing the columns of table T').

If the environment guarantees that only significant entries are accessed, the above structure is sufficient for information retrieval. Otherwise, we need

sigmap: ARRAY [1 . . m, 1 . . n] OF boolean.

For parser tables this **sigmap** corresponds to the *negated* error matrix described in Section 1.2.

In any case, two vectors

rowmap: ARRAY [1 . . m] OF 1 . . g_r

and

columnmap: ARRAY [1 . . n] OF 1 . . g_c

are necessary for representing the mapping from the rows and columns of table T to the rows and columns of the table **value**. The value of a table entry $T[i, j]$ is obtained by

IF **sigmap**[i, j] THEN **value**[**rowmap**[i], **columnmap**[j]]
ELSE **insignificant**.

If only significant values are accessed there is no need for **sigmap** and the access is done by

value[**rowmap**[i], **columnmap**[j]].

The row-merging method reported in [27, 28] is similar to this scheme; however, the connection with graph coloring, which opens the field for application of different heuristic graph coloring algorithms, is not mentioned there.

2.2 Line Elimination Scheme (LES)

LR-parser tables (as many precedence tables) usually contain many rows and columns in which all significant entries have the same value. Therefore we can apply a method which Bell [6] originally suggested for precedence tables and in which this characteristic is exploited. The compression is done by scanning all rows of T, therein eliminating each row i where all significant entries have the same value v_i; then the same is done for all columns by eliminating each column j with single significant value v_j. This value v_i (respectively, v_j) is entered in a vector

r: ARRAY [1 . . m] OF data (respectively, **c**: ARRAY [1 . . n] OF data)

by

$\mathbf{r}[i] = v_i$ (respectively, $\mathbf{c}[j] := v_j$).

In case a column has been eliminated, some other rows may also be eliminated in a second scan of the remaining rows, and a second column scan may also bring new column elimination. This procedure is applied alternatively to rows and columns until no further lines can be eliminated. If such an elimination occurs in the **s**th scan of rows (respectively, columns) the number **s** of scans is entered in a decision vector **dr**(respectively, **dc**) by

$\mathbf{dr}[i] := \mathbf{s}$ (respectively, $\mathbf{dc}[j] := \mathbf{s}$).

For each row i (respectively, each column j) that could not be eliminated, the number of the very last scan s_{max} is entered in the decision vector **dr** (respectively, **dc**), and the mapping to row i' (respectively, column j') of the compressed $l_r * l_c$ table **value** is entered in vector **r** (respectively, **c**). The four vectors **dr**, **dc**, **r**, **c** represent the mappings

dr: $[1 \,..\, m] \rightarrow [1 \,..\, s_{max}]$,
dc: $[1 \,..\, n] \rightarrow [1 \,..\, s_{max}]$,
r: $[1 \,..\, m] \rightarrow [1 \,..\, l_r] \cup$ data,
c: $[1 \,..\, n] \rightarrow [1 \,..\, l_c] \cup$ data.

If arbitrary entries are accessed, the same binary matrix **sigmap** as in the scheme above is needed. A table entry $T[i, j]$ is obtained by

```
IF sigmap [i, j]
THEN
  IF dr[i] < dc[j]
  THEN r[i]
  ELSE
    IF dr[i] > dc[j]
    THEN c[j]
    ELSE value[r[i], c[j]]
ELSE insignificant
```

As in the Graph Coloring Scheme, **sigmap** and the first IF statement are not needed if we know that only significant entries are accessed.

2.3 Row Displacement Scheme (RDS)

This method was suggested by several authors [4, 42]. The essential idea is to merge all rows of table T into a single vector

value: ARRAY $[1 \,..\, h]$ OF data

in such a way that one row at a time is embedded into **value**. This is done by shifting the row from index 1 to higher indices until all its significant elements are overlapping only insignificant entries in **value**. For retrieval of information, vectors

rowindex: ARRAY $[1 \,..\, h]$ OF $0 \,..\, m$

and

rowpointer: ARRAY $[1 \,..\, m]$ OF $0 \,..\, h-n$

are needed, where **rowindex**[k] contains a row number i, if **value**[k] contains a significant element of row i. **rowpointer**[i] points to the position in **value**, which the zeroth element $T[i, 0]$ of row i would have. Thus the access to an arbitrary element $T[i, j]$ may be realized by

IF **rowindex**[**rowpointer**[i] + j] ⟨ ⟩ i
THEN **insignificant**
ELSE **value**[**rowpointer**[i] + j].

If only significant entries are accessed, this procedure is simplified to

value[**rowpointer**[i] + j]

and the vector **rowindex** is not necessary.

2.4 Significant Distance Scheme (SDS)

In this method [5] the heading and trailing insignificant elements of each row in T are ignored, and the remaining intervals from the first to the last significant elements are placed in a vector

value: ARRAY [1 . . h] OF data

in a linear sequence row after row. For retrieval of an entry $T[i, j]$ three additional vectors

rowpointer : ARRAY [1 . . m] OF $-n$ . . $h-1$,
first, last : ARRAY [1 . . m] OF 0 . . n

are introduced. In **first**[i] the columnindex of the first and in **last**[i] the columnindex of the last significant element of row i is entered. **rowpointer**[i] points to that position in **value** that the zeroth element $T[i, 0]$ of row i would have. Thus the access is done by

IF ($j <$ **first**[i]) or ($j >$ **last**[i])
THEN **insignificant**
ELSE **value**[**rowpointer**[i] + j].

In general, this basic version of the method does not save very much space. Therefore Beach suggests an additional heuristic for gaining better compression by producing a special column permutation. In this case an additional vector

columnperm: ARRAY [1 . . n] OF 1 . . n

is needed and in the access j has to be replaced by **columnperm**[j]. If only significant entries are accessed the vectors **first** and **last** are superfluous and $T[i, j]$ is accessed by

value [**rowpointer**[i] + j].

2.5 Row Column Scheme (RCS)

A commonly employed compression method using list structure is the Row Column Scheme [37]. In this scheme, the storage of insignificant elements is completely avoided by the help of three vectors

value : ARRAY [1 . . h] OF data.
columnindex : ARRAY [1 . . h] OF 1 . . n.
rowpointer : ARRAY [1 . . m + 1] OF 1 . . h + 1.

Scanning the table T row by row, the vector **value** is filled with significant elements $T[i, j]$ only. The column index j is entered in the respective position of **columnindex**. When a new row i is started, the index of the next free element of **value** is entered in **rowpointer**[i]. When a table element $T[i, j]$ is looked up, the index j is searched in **columnindex** from position **rowpointer**[i] to **rowpointer**[$i + 1$] $-$ 1. In the case in which j is found in position k, **value**[k] is the value of $T[i, j]$; otherwise $T[i, j]$ is insignificant. One should note that this method cannot be simplified when accessing only significant elements. Sophisticated extensions of this basic method are discussed in Section 3.7.

2.6 Suppressed Zero Scheme (SZS)

This method is often used to compress text files. The compression is done by scanning to the table T row by row and sequentially entering the significant values in a vector

tablevector: ARRAY [1 . . $2 * h + 1$] OF integer.

The number of consecutive insignificant elements before the first or after the last significant element, or between two significant elements, of a row are entered in **tablevector** before, after, or between these elements, together with a label marking the insignificance. For faster access it is advisable to use a vector

rowpointer: ARRAY [1 . . $m + 1$] OF 1 . . $2 * h + 1$

similar to the Row Column Scheme.

3. HEURISTICS AND TUNING

In this Section we discuss the heuristics that have been applied additionally in order to improve the methods. In Section 3.7 an overview of some possible extensions is presented.

3.1 Code Reduction Scheme (CRS)

Whereas the methods described in Section 2 aim at reducing the size of the tables, the Code Reduction Scheme aims at reducing the size of entries.

It is a divide-and-conquer method for data. The basic idea is to divide the information of a table entry into a large base information that is common to all entries of a single column and a small displacement information that is just enough to distinguish different entries in that column. The displacement information is left in the table, whereas the base information is placed in a separate vector (one for each column). As a highly desirable side effect, the rows and columns of the tables show more uniformity (see Figure 3). Our experimental results show that including this scheme yields better preconditions for the Graph Coloring Scheme and the Line Elimination Scheme than leaving it out. It is only in this method that we use the information in the contents of the parser tables.

The tables generated by an LALR(1)-parser-generator have the following properties:

(1) For the set of states Q, a partition Q/V' may be defined by

$$Q_a = \{q' \mid \text{T-table}(q, a) = (\text{shift}, q'), q \in Q\} \; \forall a \in V_{T'}$$

		Q_T						Q_N			
		0	5	6	7	0	0	0	1	3	0
		P_T						P_N			
		2	2	2	2	2	3	0	0	0	0
PLP		1	2	3	4	5	6	1	2	3	4
0	1				1		*1	1	2	2	*1
0	2		1			*1					
0	3	−1	1								
1	4	−1	−1	1		−1					
2	5	−1	−1	1		−1					
3	6				1		*1			1	*1
3	7				1		*1				*2
3	8				1		*1		1	2	*1
PL											
5 6 7											

Figure 3

and

$$Q_a = \{q' \mid \text{N-table}(q, a) = (\text{shift}, q'), q \in Q\} \; \forall a \in V_N$$

where

$$Q_a \cap Q_b = \emptyset, \qquad a \neq b, \qquad a, b \in V'.$$

(2) For the set of productions P, a similar partition P/V' may be defined by

$$P_a = \{p \mid \text{T-table}(q, a) = (\text{shift-reduce}, p), q \in Q\} \; \forall a \in V_{T'}$$

and

$$P_a = \{p \mid \text{N-table}(q, a) = (\text{shift-reduce}, p), q \in Q\} \; \forall a \in V_N$$

where

$$P_a \cap P_b = \emptyset, \qquad a \neq b, \qquad a, b \in V'.$$

(a) First, because of Property (1) we may renumber the states consistently such that for all $a \in V'$ the states of Q_a have consecutive numbers. For retrieval of information we need two mappings

$$Q_T: \begin{cases} V_{T'} \to Q \cup \{0\} \\ a \quad \to \min Q_a - 1 \end{cases}$$

$$Q_N: \begin{cases} V_N \to Q \cup \{0\} \\ a \quad \to \min Q_a - 1 \end{cases}$$

where

$$\min Q_a = \begin{cases} \min\{q \mid q \in Q_a\} & \text{if } Q_a \neq \emptyset, \\ 1 & \text{otherwise.} \end{cases}$$

Now we can replace state number q in a table column a by a **displacement**

$$d_q = \begin{cases} q - Q_T(a), & a \in V_{T'}, \\ q - Q_N(a), & a \in V_N. \end{cases}$$

Thus, all displacements in a column $a \in V'$ are in the range $1 \cdots |Q_a|$.

This normalization of displacement entries increases the probability of identical entries in different columns and decreases the number of bits necessary to store such entries.

(b) The same technique may be applied to production numbers using the partition P/V' instead of partition Q/V'. Here we have the two mappings

$$P_T: \begin{cases} V_{T'} \to P \cup \{0\} \\ a \quad \to \min P_a - 1 \end{cases}$$

$$P_N: \begin{cases} V_N \to P \cup \{0\} \\ a \quad \to \min P_a - 1 \end{cases}$$

where

$$\min P_a = \begin{cases} \min\{p \mid p \in P_a\} & \text{if } P_a \neq \emptyset \\ 1 & \text{otherwise} \end{cases}$$

and the displacement of a production number p is given by

$$d_p = \begin{cases} p - P_T(a), & a \in V_{T'}, \\ p - P_N(a), & a \in V_N. \end{cases}$$

(c) The reduce entries in the T-table do not fit into such a classification. We treat them as follows: For each row q of the T-table we construct a list PL_q (production list) consisting of those production numbers that have reduction entries in the row and replace the production numbers in the row by respective indices j_p in PL_q. Now we concatenate all lists into one vector PL. For retrieval of information a vector PLP (production list pointer) contains the start position (minus one) of each list PL_q in PL, such that

$$PL\,[PLP[q] + j_p]$$

yields p.

The following example may be helpful in understanding the essential ideas of the Code Reduction Scheme.

Let G again be our example grammar with the generated LALR(1)-parser tables as in Figure 1. Then Figure 4 shows the tables after state (QR) and production (PR) renumbering before the mapping onto the displacements is performed. Note that the vectors PR and QR do not add to the storage requirement if all production and state information is renumbered consistently. Figure 3 shows the tables after application of the Code Reduction Scheme.

Part (a) of the Code Reduction Scheme was first mentioned by Anderson [2]. We added the steps (b) and (c) to get smaller numbers all over the table. When applying this scheme, we are able to store the entries of the tables in 1 byte even

PR:		1	2	3	4	5	6	7				
		5	6	7	2	1	3	4				
QR		1	2	3	4	5	6		1	2	3	4
1	1				8		*4		1	3	5	*1
8	2		6			*3						
5	3	−5	6									
3	4	−6	−6	7		−6						
2	5	−7	−7	7		−7						
7	6				8		*4				4	*1
6	7				8		*4					*2
4	8				8		*4			2	5	*1

Figure 4

in the case of the larger grammars. Note that there could be considerable savings with bit storage techniques in the case of the smaller grammars in a trade-off of portability and fast access (compare with [8]). Anderson, Eve, and Horning [3] rejected method (a) in favor of another state permutation allowing the deletion of empty rows in the N-table. They argue that for a large number of states there is no entry in the N-table. But we have observed that this number decreases considerably if LR(0)-reduce states are removed and replaced by shift-reduce transitions.

3.2 Graph Coloring Heuristics

The principles of the Graph Coloring Scheme are independent of the actual coloring of the respective graph. Usually, a minimal coloring cannot be achieved because of the high algorithmic complexity of the problem, which is NP-complete [29, 17]. Therefore we have used the following approximate algorithm, which can be obtained from a backtrack algorithm by suppressing the backtracking and which proved slightly better in experimental results than other approximate algorithms [11]:

while there is a noncolored vertex in G
loop Choose a noncolored vertex v that is blocked for a maximum number of colors (blockings caused by already colored adjacent vertices); Color vertex v with the least possible color;
repeat

Unfortunately the approximation may be very bad in special cases. This is common to all approximate coloring schemes and hence cannot be improved by switching to another method [25]. For each number g there are g-chromatic graphs (i.e., graphs colorable with g but not with $g - 1$ colors), for which the algorithm may use a number g_a of colors for coloring that is linearly bounded on the vertex number n, that is, $g_a = O(n)$ [14].

For instance, let us consider a graph constructed from the complete 3-partite graph with $3p$ vertices and the three independent vertex sets $\{a_1, \ldots, a_p\}$, $\{b_1, \ldots, b_p\}$, and $\{c_1, \ldots, c_p\}$ by deleting all edges (b_i, c_i) joining b_i and c_i for $1 \leq i \leq p$ and all edges $\{a_i, b_j\}$ and $\{a_i, c_j\}$ for $1 \leq i \leq p$ and $1 < j \leq p$. This graph may be colored by our algorithm in the vertex sequence $a_1 b_1 c_1 a_2 \cdots a_p b_p c_p$, and then it

	sigmap							eq		et						sigmap'				
	1	2	3	4	5	6				1	2	3	4	5	6					
1	0	0	0	1	0	1		1		1	2	3	4	5	4	0	0	0	1	0
2	0	0	0	1	0	1		1								1	1	1	0	1
3	1	1	1	0	1	0		2								1	1	0	0	0
4	1	1	0	0	0	0	⇒	3								0	1	0	0	1
5	0	1	0	0	1	0		4												
6	0	0	0	1	0	1		1												
7	0	0	0	1	0	1		1												
8	1	1	1	0	1	0		2												

Fig. 5. The compression of sigmap corresponding to the error matrix of Fig. 2 with the help of two vectors **et** and **eq**.

will use $p + 1$ colors by assigning the colors $1, 2, 2, 1, 3, 3, \ldots, 1, p + 1, p + 1$ to the vertices in the said sequence.

Preordering the vertices by decreasing degree, as proposed in [7], will not change the behavior [21], as each vertex can be joined with an appropriate number of 1-degree vertices producing the above order $a_1 b_1 c_1 \cdots a_p b_p c_p$ when applying the preordering rule. Also, experimental results using random graphs with up to 1000 vertices [11] showed no significant change of the approximation behavior when these additional heuristics were applied.

In the same way the conjecture stated in [7]—that the algorithm with decreasing degree ordering will "color a clique with an almost maximum dimension first"—should be treated cautiously, because an infinite number of counterexamples exists. For instance, for any number $p \geq 3$ a graph may be constructed with $2p + 1$ vertices containing a vertex with p adjacent vertices of degree 1 and joined to one vertex of a complete graph with p vertices; obviously, a 2-clique will be colored first, whereas there is also a p-clique.

Nevertheless, despite this bad behavior in some cases, the average behavior for random graphs has been shown to be quite good for such approximate coloring schemes [20]. An even better result is achieved when coloring the collision graphs of the investigated parser tables. We know this from the fact that these graphs have complete subgraphs with nearly as many vertices as the number of colors used in the produced colorings. Therefore, the algorithm can be recommended for use in the Graph Coloring Scheme.

3.3 Binary Table Compression

In the Graph Coloring Scheme and the Line Elimination Scheme a binary table **sigmap** is used for accessing the T-table. Since the compression may often save more than 90 percent of the original storage requirement, this binary table **sigmap** makes up an essential part of the total storage need. Fortunately **sigmap** has a quite uniform structure, which we can compress by merging identical rows and columns as shown in Figure 5. Joliat proposed this technique in [27]. Table I gives the improvement for the different grammars.

Table I. Binary Matrix Optimization

language	sigmap table size	storage need	opt table size	storage need
ADA	381 * 96	4572	137 * 72	1710
AL	365 * 136	6205	121 * 69	1590
ALGOLW	128 * 67	1152	63 * 42	573
ALGOL60	129 * 58	1032	63 * 45	565
BALG	486 * 136	8262	151 * 73	2132
EULER	62 * 75	620	34 * 32	273
LEX	134 * 73	1340	62 * 46	579
LIS	330 * 104	4290	141 * 75	1844
MINILEX	89 * 41	534	35 * 30	270
PASCAL	209 * 62	1672	75 * 41	721
XPL	90 * 46	540	44 * 28	312

For **sigmap** and **sigmap'** the equation

$$\textbf{sigmap}[q, a] = \textbf{sigmap}'[\textbf{eq}[q], \textbf{et}[a]]$$

holds for all $q \in Q$ and $a \in V_{T'}$.

Note. If we knew the insignificant entries in the original T-table (see Section 1.1) we could incorporate them in **sigmap** and apply one of the table compression methods to it as well.

3.4 Row Displacement Heuristics

Parser tables have 2–20 percent significant entries for the grammars considered (see Table III). A little calculation shows that in the Row Displacement Scheme it is advisable to replace the mapping vector **rowindex** used for the T-table by

the binary table **sigmap** as needed in the Graph Coloring Scheme and the Line Elimination Scheme (see Table V). But by this trade-off the better access-time behavior of the Row Displacement Scheme compared with the Graph Coloring Scheme is lost (see Figure 6). In our implementation of the Row Displacement Scheme we incorporated the "first fit decreasing" heuristic, which was already suggested in [42] and analyzed in [38].

3.5 Combination of Compression Methods

As the resulting tables **value** from the Graph Coloring Scheme and the Line Elimination Scheme may still have insignificant entries, a further application of one of the table compression schemes may be considered. Evidently, the combinations of the Line Elimination Scheme with itself and the Graph Coloring Scheme with itself will cause no improvement. It can be shown that a combination of the Graph Coloring Scheme with the Line Elimination Scheme will never give better results than the combination of the Line Elimination Scheme with the Graph Coloring Scheme if we have a minimal coloring of the respective collision graphs. Since the value table of the Graph Coloring Scheme is not sparse enough to allow a compression by the Row Displacement Scheme or the Significant Distance Scheme, only the combinations of the Line Elimination Scheme with the Graph Coloring Scheme and the Line Elimination Scheme with the Row Displacement Scheme are left for experimental considerations.

Note that in an application of one of the combinations to the T-table, the binary table **sigmap** used in the Line Elimination Scheme is sufficient for retrieval, and therefore the binary table **sigmap** in the Graph Coloring Scheme and the vector **rowindex** in the Row Displacement Scheme are superfluous. Additionally, we can combine the two mappings **rowmap** and **columnmap** used in the Graph Coloring Scheme with the mappings **r** and **c** in the Line Elimination Scheme. A similar concatenation can be applied to the mappings **rowpointer** in the Row Displacement Scheme and **r** in the Line Elimination Scheme.

These savings are not possible in a combination of the Line Elimination Scheme with the Significant Distance Scheme, and experiments on this combination do not seem worthwhile. In practice the combination of the Line Elimination Scheme with the Graph Coloring Scheme has another advantage. It is much faster than the Graph Coloring Scheme alone (see Section 4.1) because the Graph Coloring Scheme has only to be applied to the already compressed tables.

3.6 Table Transposition

Since all compression methods considered give the parser tables an arbitrary orientation in rows and columns, we can apply the methods to the transposed tables as well. Evidently we get no different results for the Line Elimination Scheme. For the Row Column Scheme and the Suppressed Zero Scheme experiments with the transposed tables never showed better results.

3.7 Sophisticated Extensions

The methods mentioned in this section are usually found in the literature as improvements over the basic Row Column Scheme. We have not implemented these methods since their efficient usage is limited to machine-specific instructions, for example, bit-field and character-search instructions. The trade-off

between saved storage and efficient access needs to be considered anew for different machines. We think that this conflicts with the idea of portability.

One of these methods is to extract default values from the tables as suggested in some publications [3, 4, 35]. For the nonterminal part of the tables, this is easily done by computing for each nonterminal the most frequent entry and eliminating this entry in the column of this nonterminal. Since there can be no error when accessing the nonterminal table, the eliminated entry is used as default whenever the searched entry was not found.

The same method may be applied to the terminal table whenever there is a possibility of detecting an error correctly (see [4]). Obviously, the methods using the binary table **sigmap** have this property.

Another possibility is the extraction of default values of each state. Note that in the terminal table the default action for the states is almost always the error entry, if the error matrix is not factored out before extracting the defaults.

To save superfluous bits, some authors propose bit storage techniques (e.g., [8]): For every vector and table their largest entry is computed and the vector (respectively, table) is restored in a smaller data structure where each entry needs as many bits as the largest entry does. However, when looking for a fast table access it is advisable to use 8-bit or 16-bit units on a machine where these units are directly addressable.

When the table structures are examined, it is observed that there are groups of rows whose members have almost the same structure. Some of these rows have identical structure and may be merged easily when doing list compression: The starting value of the identical row, which is already stored in the vector **value**, is assigned to the corresponding field in the vector **rowpointer**. If there are rows with only a few different significant entries saving additional space may be achieved by merging the identical tails of the rows. Such tails are merged by compressing one row completely and replacing the tail of the others by a link to the former. If, in connection with default values, difficulties arise the default treatment may be incorporated into the link [35].

Joliat [27] proposes a strength reduction by linearizing the table and precomputing the start indices of each row, thus replacing an index calculation by an additional vector and a vector access. If only significant entries are accessed, the leading and trailing insignificant entries of each row may be omitted. The precomputed indices then point to the virtual start address of the corresponding row. This is just the idea of the simple Significant Distance Scheme that is applied here to the compressed tables.

4. RESULTS

In this Section we give a short review on the behavior of the methods with respect to the complexity of the compression procedures, the amount of time used for access, and the storage needed for the compressed data structure, and then provide the experimental results on the storage requirements.

4.1 Complexity of Compression Procedures

Since the graph coloring problem is NP-complete, in practice only an approximate coloring algorithm is possible. When using the algorithms with time complexity

$O(n^2)$ given in Section 3.2, the generation of the graph from the table will be the most complex part of the Graph Coloring Scheme. Therefore, we have a worst case complexity $O(n * m^2, m * n^2)$ and a best case complexity $O(m^2, n^2)$.

We obtain the same worst case complexity for the Line Elimination Scheme if exactly one row (respectively, one column), is eliminated in each scan. In the best case we have $O(m * n)$.

In the Row Displacement Scheme, the worst case occurs if none of the rows can be embedded into the prefix of the vector **value** constructed from the previous rows. For each position i in this prefix it may be necessary to compare a section of the prefix bounded by i and $n + i - 1$ with the actual row. Thus we have, $O(m^2 * n^2)$ comparisons in the worst case. The best case complexity is $O(m * n)$.

The simplest version of the Significant Distance Scheme has complexity $O(m * n)$. The additional heuristic reordering of the columns suggested in [5] has worst and best case complexity $O(m^2 * n^2)$.

When the different schemes are applied to different grammars, our experimental results show that the Line Elimination Scheme is by far the fastest, since the best case $O(m * n)$ will usually apply. In spite of having a higher worst case complexity, the Row Displacement Scheme turns out faster than the Graph Coloring Scheme, since the Row Displacement Scheme approaches its best case $O(m * n)$ better than the Graph Coloring Scheme approaches its best case $O(m^2, n^2)$.

In fact, the Row Displacement Scheme is 3 times, and the Graph Coloring Scheme about 10 times slower than the Line Elimination Scheme measured in CPU time on a SIEMENS 7.760 using Pascal. Because the best case complexity of the Significant Distance Scheme with the additional heuristics is $O(m^2 * n^2)$, the execution time of the generation cannot be put into linear relation to the other schemes, taking into account the size of the considered grammars. For the smallest grammar, the execution time was 50 times slower than the Line Elimination Scheme. In the case of the five largest grammars, the execution time exceeded our resources (more than 2 hours of CPU time).

4.2 Efficiency of Access Procedures

The execution time needed to access an entry in a compressed table mainly depends on the number of vector and matrix accesses. Additional time is needed for looking up a single bit instead of an addressable unit.

Figure 6 gives an overview of the retrieval time measured in terms of accesses. For each of the four index access methods, the first line shows the number of these accesses when looking up arbitrary entries, and the second line contains the number of accesses if only significant entries are examined. The column "Bit access necessary" indicates whether an access to the compressed binary table **sigmap'** is necessary or not (see Section 3.3). Column "Number of matrix accesses" includes one access to the **sigmap'** in all cases where a bit access is necessary. The last column summarizes the number of conditions that have to be evaluated for the access to an entry. For the Significant Distance Scheme the access costs are given for the version with column permutation.

For the Code Reduction Scheme, one add operation and one- (respectively, two-) vector accesses have to be added in all of the four schemes. Obviously the

	Number of vector accesses	Number of matrix accesses	Bit access necessary	Number of conditions
GCS				
Arbitrary	2 or 4	1 or 2	Yes	1
Significant	2	1	No	0
LES				
Arbitrary	2–8	1 or 2	Yes	1–3
Significant	3–6	0 or 1	No	1 or 2
RDS				
Arbitrary	2 or 4	0	No	1
Significant	2	0	No	0
SDS				
Arbitrary	2 or 5	0	No	2
Significant	3	0	No	0

Fig. 6. Table access costs.

Row Displacement Scheme guarantees the fastest access followed closely by the Significant Distance Scheme.

Detailed considerations [12] show that the Graph Coloring Scheme is 3 to 4 times slower when accessing an arbitrary element calculated for a SIEMENS 7.755 using FORTRAN and a UNIVAC 1108 using FORTRAN and Assembler. This result is mainly due to the bad bit instruction performance of the machines used.

For the list methods, the Row Column Scheme and the Significant Distance Scheme, the access time is dependent on the number of significant entries per row. In the worst case, the Suppressed Zero Scheme needs twice as many comparisons as the Row Column Scheme (see Section 4.4 and Tables IV and V). In the best case the same number of comparisons is needed. The average search list length in a running compiler depends heavily on the language and on the manner in which the tables are constructed. Therefore, benchmarking (see [27] for XPL) seems useful only for each specific environment.

4.3 Theoretical Storage Requirements

In general, the minimal storage requirements for the tables will differ for the different compression methods. This minimum will be reached for the Line Elimination Scheme and for the two list search methods Row Column Scheme and Suppressed Zero Scheme in each application, whereas the other three methods will produce results not necessarily equal to their minima, depending on the order in which the rows and columns are treated. The generation of the respective minimum of each of these methods is an NP-complete problem: Determinating a minimal graph coloring is proved to be NP-complete [29], and gaining the minimum compression of the Row Displacement Scheme is also NP-complete [15]. A method similar to that used in [15] will yield the NP-completeness of the Significant Distance Scheme.

Evidently, the methods described are approximation methods, and there is ample room for additional heuristics, as described in Section 3. In certain cases

these heuristics may generate results arbitrarily far from the minimum. Experimental results on very sparse random tables (densities from 0.1 to 1 percent) under the stipulation that arbitrary elements are accessed [12] showed that the Row Displacement Scheme is superior to the Graph Coloring Scheme with a factor decreasing with increasing density. In this interval of density, the Row Column Scheme had results equal to the Row Displacement Scheme. In fact the storage need for the Row Column Scheme is a lower bound for the storage need of the Row Displacement Scheme in the version using **rowindex**. It is possible that the Graph Coloring Scheme and the Row Displacement Scheme yield similar results for the higher densities that parser tables have. The Suppressed Zero Scheme has in the worst case the same storage need as the Row Column Scheme, but less than the Row Column Scheme when there are many consecutive significant entries in T.

4.4 Experimental Results

To compare the schemes presented and the combination of the Line Elimination Scheme with the Graph Coloring Scheme and the Line Elimination Scheme with the Row Displacement Scheme discussed in the preceding Section, we chose 11 available LALR(1)-grammars and applied the different methods to their parser tables. In Table II we give the characteristics of the grammars and in Table III the characteristics of the corresponding unoptimized parser tables.

The experimental optimization results are shown in Tables IV and V and the comparison of the best results for each method in Tables VI and VII. The columns "storage need" in Tables IV and V include all access vectors, and Table V includes in addition the optimized **sigmap** where applicable.

The results are expressed in bytes assuming a byte machine, that is, integer values in the range of 0 to 255 are stored in 1 byte, and values in the range of 0 to $2^{16} - 1$ are stored in a 2-byte unit. Furthermore the number of bytes used for a row of a binary table with b bits will be $(b + 7)$ **div** 8. In the presentation of the algorithms we have used ranges of integers with lower bound 1 for simpler notation; however, in our implementation we have used lower bound 0.

The Code Reduction Scheme was also applied to all results shown in Tables IV–VII. On the one hand it adds to the access time of the tables (see Section 4.2), but on the other hand it appears to cut the space requirements of the tables by half.

From Tables VI and VII the Graph Coloring Scheme emerges as the best compression method on the average. Even when we look at the results that lie 20 percent above the minimum, for the T-table only the combination of the Line Elimination Scheme with the Graph Coloring Scheme is competitive with the Graph Coloring Scheme, whereas for the N-table we get a slightly scattered field led by the Graph Coloring Scheme.

Those cases where the compressed tables could be generated by using the Significant Distance Scheme (see Section 4.1) behave comparable to the Row Displacement Scheme for the N-table and slightly better for the T-table. The Line Elimination Scheme alone trails the other methods in most cases. In combination with the Graph Coloring Scheme it does not effect an improvement over the Graph Coloring Scheme alone, whereas in combination with the Row Displacement Scheme there generally is a slight improvement. This improvement

Table II. Grammar Characteristics

language	productions	terminals	nonterminals	righthand side length
ADA	370	96	170	782
AL	306	136	115	702
ALGOLW	146	67	69	272
ALGOL60	158	58	76	287
BALG	406	136	172	1045
EULER	125	75	50	196
LEX	202	73	109	330
LIS	328	104	129	706
MINILEX	81	41	37	175
PASCAL	274	62	166	470
XPL	108	46	51	212

for the combination of the Line Elimination Scheme with the Row Displacement Scheme is due to the deletion of significant entries from the tables by the Line Elimination Scheme (see Tables IV and V).

The list-searching methods Row Column Scheme and Suppressed Zero Scheme are not competitive with the above methods (even in combination with the Code Reduction Scheme). Comparison with published results on list searching methods (see Figure 7) leads us to the conjecture that the Row Column Scheme and the Suppressed Zero Scheme are handicapped by the particular factorization of the parser tables that we have used. In the direct comparison of both methods, the Suppressed Zero Scheme requires less storage. The reason for this is that there are some consecutive significant entries in the parser tables. On the other hand, the Suppressed Zero Scheme requires more comparisons of keys for access in the

Table III. Characteristics of the Nonoptimized Parser Tables

language	states	T-table storage need	T-table signif. entries	T-table % of sig entries	N-table storage need	N-table empty rows	N-table signif. entries	N-table % of sig entries
ADA	381	73152	2325	6.36 %	129540	154	1925	2.97 %
AL	365	99280	4193	8,45 %	83950	191	1716	4.09 %
ALGOLW	128	17152	1123	13.09 %	17664	60	831	9.41 %
ALOGOL60	129	14964	928	12.40 %	19608	52	768	7.83 %
BALG	486	132192	4326	6.55 %	167184	221	2098	2.51 %
EULER	62	9300	1377	29.61 %	3100	22	726	23.42 %
LEX	134	19564	1797	18.37 %	29212	51	1432	9.80 %
LIS	330	68640	2408	7.02 %	85140	153	1527	3.59 %
MINILEX	89	7298	752	20.61 %	3293	44	262	7.96 %
PASCAL	209	25916	1156	8,92 %	69388	77	794	2.29 %
XPL	90	8280	472	11.40 %	4590	42	398	8.67 %

worst case, as can be seen from the lengths of both search lists (the last two columns of Tables IV and V).

The methods Line Elimination Scheme and Row Displacement Scheme are sensitive to the different structure of the T-table and the N-table, whereas the Graph Coloring Scheme and the Significant Distance Scheme show the same optimization behavior for both tables. For methods involving the Graph Coloring Scheme, the heuristics for coloring columns before rows is generally ahead of the opposite order heuristics. Contrary to this the Row Displacement Scheme shows better results when merging rows rather than columns. The only exception appears in the N-table of XPL.

Replacing vector **rowindex** by matrix **sigmap** in the Row Displacement Scheme results in saving 25–60 percent of storage. This saving is especially high in those cases where the elements of the vector **rowindex** have to be stored in more than 1 byte.

The fact that the Graph Coloring Scheme saves more space than the Row Displacement Scheme is contrary to the results for very sparse random tables

Table IV. Optimization Results for the N-table .

language	GCS table size	GCS storage need	LES table size	LES sig. entries	LES storage need	RDS nonsig entries	RDS storage need	SDS storage need	LES & GCS table size	LES & GCS storage need	LES & RDS nonsig entries	LES & RDS storage need	RCS storage need	SZS storage need
ADA	110*27 135*16	3861 3051	135*52	902	8462	1017 1671	4044 4276	/	110*20 122*15	3642 3122	310	3035	(8.5) 4954 (32)	3685 (40)
AL	132*16 134*16	2822 2854	112*27	1029	4214	969 2550	3645 4726	/	110*16 111*16	2950 2966	746	3330	(9.9) 4394 (54)	3192 (62)
ALGOL W	42*23 46*20	1301 1255	48*24	485	1684	288 1212	1513 2319	1733	42*20 45*18	1372 1342	103	1248	(12.2) 2058 (35)	1451 (43)
ALGOL60	41*21 45*17	1218 1122	48*32	490	2098	352 657	1530 1729	1717	41*17 41*17	1259 1259	168	1349	(10.0) 1950 (31)	1439 (36)
BALG	72*38 150*14	3738 3102	167*64	939	12348	1178 3502	4592 6288	/	61*64 138*11	5564 3178	160	3245	(7.9) 5514 (40)	4425 (56)
EULER	29*17 29*17	705 705	23*12	174	600	106 244	1056 1170	1125	23*12 23*12	600 600	9	507	(18.2) 1678 (42)	1050 (47)
LEX	42*22 52*18	1385 1397	56*30	524	2394	912 1359	2830 3227	2675	41*16 45*14	1360 1334	147	1509	(17.3) 3352 (55)	2362 (72)
LIS	68*19 90*12	2009 1797	100*38	577	4976	1138 1812	3583 3885	/	65*14 80*10	2086 1976	115	2198	(8.6) 3974 (29)	3314 (46)
MINILEX	27*12 29*11	524 519	32*15	181	806	48 244	562 654	677	27*10 30*10	596 626	64	571	(6.2) 778 (20)	590 (25)
PASCAL	42*13 66* 9	1253 1301	55*25	289	2457	279 429	1823 1887	/	20*21 55* 9	1502 1577	65	1645	(6.0) 2340 (16)	1799 (22)
XPL	30*15 33*11	693 606	28*16	187	832	269 334	949 936	822	28*16 29* 8	832 616	31	608	(8.3) 1080 (21)	828 (27)

Note:

For GCS, and LES and GCS, the two lines for each grammar show the different results for the row–column and the column–row coloring heuristic. For RDS the second line in column "storage need" shows the results for the transposed tables.

For RCS and SZS the lower line indicates the maximum search list length.

For RCS the average search list length of the nonempty rows is indicated in the upper line.

Table V. Optimization Results for the T-table

language	GCS table size	GCS storage need	LES table size	LES sig. entries	LES storage need	RDS nonsig. entries	RDS storage need	SDS storage need	LES & GCS table size	LES & GCS storage need	LES & RDS nonsig. entries	LES & RDS storage need	RCS storage need	SZS storage need
ADA	41*39	4917	256*72	1890	22229	798	11262	/	26*39	4809	466	6532	(6.1) 6543	5324
	60*20	4518					B 6726		39*19	4536			(35)	(47)
AL	33*38	4350	244*108	3401	29949	3713	25453	/	25*33	4422	1944	9307	(11.4) 10121	7511
	53*18	4050					B 11231		44*18	4389			(62)	(78)
ALGOLW	19*24	1518	81*49	940	5226	691	4178	1691	15*18	1527	533	2858	(8.8) 2796	2150
	23*12	1338					B 2937		15*12	1437			(32)	(42)
ALGOL60	20*28	1602	89*41	706	4878	443	3290	1979	20*23	1685	288	2352	(7.2) 2404	1924
	29*12	1390					B 2484		26*12	1537			(29)	(38)
BALG	55*43	7409	284*89	3636	31428	3893	27919	/	43*41	7429	4233	14021	(8.9) 11914	10530
	66*24	6628					B 13613		50*24	6866			(59)	(83)
EULER	15*17	894	37*27	340	1775	615	4337	1389	11*15	941	98	1276	(22.2) 3107	2048
	18*15	909					B 2618		12*14	944			(45)	(56)
LEX	18*27	1600	89*42	942	5056	1259	6708	3237	17*22	1705	307	2704	(13.4) 4190	2937
	24*19	1570					B 4231		22*18	1727			(49)	(55)
LIS	59*31	5023	194*84	2149	19924	1138	12214	/	32*26	4460	1774	7881	(7.3) 6392	5646
	67*15	4199					B 6966		43*14	4230			(32)	(54)
MINILEX	18*20	958	57*28	430	2324	315	2510	1492	14*18	927	113	1360	(8.4) 1880	1356
	25*14	948					B 1713		17*14	913			(25)	(29)
PASCAL	26*25	2147	108*37	670	5764	464	4163	/	41* 9	2137	137	2784	(5.5) 3235	2663
	35*11	1882					B 3264		28*11	2076			(33)	(42)
XPL	22*16	1005	54*22	268	1977	129	1587	1288	10*11	899	33	1180	(5.2) 1329	1044
	26* 9	887					B 1298		12* 9	897			(22)	(27)

Note:

For GCS, and LES and GCS, the two lines for each grammar show the different results for the row–column and the column–row coloring heuristic. For RDS the second line in column "storage need" shows the results when using sigmap instead of rowindex.

For RCS and SZS the lower line indicates the maximum search list length.

For RCS the average search list length row is indicated in the upper line.

Table VI. Comparison of Results Concerning N-tables

	GCS	LES	RDS	SDS	LES&GCS	LES&RDS	RCS	SZS
ADA	1.01 2.4 %	2.79 6.5 %	1.33 3.1 %	/	1.03 2.4 %	1.00 2.3 %	1.63 3,8 %	1.21 2,8 %
AL	1.00 3.4 %	1.49 5.0 %	1.29 4.3 %	/	1.04 3.5 %	1.18 4.0 %	1.56 5.2 %	1.13 3.8 %
ALGOLW	1.01 7.1 %	1.32 9.3 %	1.21 8.6 %	1.39 9.8 %	1.08 7.6 %	1.00 7.1 %	1.64 11.6 %	1.16 8.2 %
ALGOL60	1.00 5.7 %	1.87 10.7 %	1.36 7.8 %	1.53 8.7 %	1.12 6.4 %	1.20 6.9 %	1.74 9.9 %	1.28 7.3 %
BALG	1.00 1.9 %	3.98 7.4 %	1.48 2.7 %	/	1.02 1.9 %	1.05 1.9 %	1.78 3.3 %	1.43 2.6 %
EULER	1.39 22.8 %	1.18 19.4 %	2.08 34.0 %	2.21 36.2 %	1.18 19.4 %	1.00 16.4 %	3.31 54.0 %	2.07 33.8 %
LEX	1.04 4.8 %	1.79 8.2 %	2.12 9.7 %	2.01 9.2 %	1.00 4.6 %	1.13 5.2 %	2.22 11.5 %	1.56 8.1 %
LIS	1.00 2.1 %	2.77 5.8 %	1.99 4.2 %	/	1.10 2.3 %	1.22 2.6 %	2.21 4.7 %	1.84 3.9 %
MINILEX	1.00 15.8 %	1.55 24.4 %	1.08 17.0 %	1.30 20.4 %	1.14 18.0 %	1.10 17.4 %	1.50 23.6 %	1.13 17.8 %
PASCAL	1.00 1.8 %	1.96 3.5 %	1.45 2.6 %	/	1.26 2.2 %	1.31 2.4 %	1.87 3.4 %	1.43 2.6 %
XPL	1.00 13.2 %	1.37 18.2 %	1.54 20.4 %	1.36 18.0 %	1.02 13.4 %	1.00 13.2 %	1.78 23.4 %	1.37 18.0 %
Average factor	1.04	2.01	1.54	1.63	1.09	1.11	1.93	1.42

Note:
For each method the best result is drawn from Table IV.
For each grammar the results of the different methods are given by the factor

$$\frac{\text{storage need for the actual method}}{\text{storage need for the best method}}$$

and in the second line by the percentage of the remaining storage need.

(see Section 4.3). Reasons for this behavior are the higher densities of the parser tables and the special table structure which supports the ability of the Graph Coloring Scheme to overlap coincident significant entries.

4.5 Comparison with Published Results

In Figure 7 we oppose the results on five grammars found in [3, 8, 27, 31, 37] with our results using the Graph Coloring Scheme. From this figure we can see that the Graph Coloring Scheme is competitive in its storage requirement with the published list-searching methods. It is superior in that it allows the access to the compressed tables to be programmed efficiently in portable high-level languages like Pascal or FORTRAN.

Table VII. Comparison of Results Concerning T-tables

	GCS	LES	RDS	SDS	LES&GCS	LES&RDS	RCS	SZS
ADA	1.00 6.1 %	4.90 30.4 %	1.49 9.2 %	/	1.00 6.2 %	1.45 8.9 %	1.45 8.9 %	1.18 7.3 %
AL	1.00 4.1 %	7.39 30.2 %	2.77 11.3 %	/	1.08 4.4 %	2.30 9.4 %	2.80 10.2 %	1.85 7.6 %
ALGOLW	1.00 7.8 %	3.90 30.4 %	2.20 17.1 %	1.26 9.9 %	1.07 8.4 %	2.13 16.7 %	2.09 16.3 %	1.61 12.5 %
ALGOL60	1.00 9.3 %	3.51 32.6 %	1.79 16.6 %	1.42 13.2 %	1.11 10.3 %	1.69 15.7 %	1.73 16.1 %	1.38 12.9 %
BALG	1.00 5.0 %	4.74 23.8 %	2.05 10.3 %	/	1.04 5.2 %	2.16 10.6 %	1.80 9.6 %	1.59 8.0 %
EULER	1.00 9.6 %	1.99 19.1 %	2.37 22.8 %	1.55 14.9 %	1.05 10.7 %	1.43 13.7 %	3.48 33.4 %	2.29 22.0 %
LEX	1.00 8.0 %	3.22 25.8 %	2.69 21.6 %	2.06 16.5 %	1.09 8.7 %	1.72 13.8 %	2.67 21.4 %	1.87 15.0 %
LIS	1.00 6.1 %	4.74 29.0 %	1.66 10.1 %	/	1.01 6.2 %	1.88 11.5 %	1.52 9.3 %	1.34 8.2 %
MINILEX	1.04 13.0 %	2.57 32.1 %	1.88 23.5 %	1.63 20.4 %	1.00 12.5 %	1.49 18.6 %	2.06 25.8 %	1.49 18.6
PASCAL	1.00 7.3 %	3.06 22.2 %	1.73 12.6 %	/	1.10 8.0 %	1.48 10.7 %	1.72 12.5 %	1.41 10.3 %
XPL	1.00 10.7 %	2.23 23.9 %	1.46 15.7 %	1.45 15.5 %	1.01 10.8 %	1.33 14.3 %	1.50 16.1 %	1.18 12.4 %
Average factor	1.00	3.84	2.01	1.56	1.05	1.73	2.07	1.56

Note:
For each method the best result is drawn from Table V.
For each grammar the results of the different methods are given by the factor

$$\frac{\text{storage need for the actual method}}{\text{storage need for the best method}}$$

and in the second line by the percentage of the remaining storage need.

Reference	ALGOLW	ALGOL60	Euler	XPL	Pascal
[30]	—	2821	1606	1250	—
[8]	—	—	—	—	1741
[3]	2434	—	—	1182	—
[35]	1689	—	—	—	—
[26]	—	—	—	1836	—
GCS	2593	2512	1401	1493	3135

Fig. 7. Storage comparison with published results (in bytes). For the N-table of Euler LES & RDS was counted.

5. CONCLUSION

We have presented six compression methods for sparse tables and investigated their applicability to parser tables. Four of them are index access methods, which are suitable for portable compilers. The other two are list-searching methods, which have been included for their good reputation in storage optimization. In our experimental work they served as a reference for the comparison of the index access methods.

From our experimental results we can conclude that the Graph Coloring Scheme turned out best. Therefore we recommend this method for use in parser generating systems. During the development phase of a compiler where the parser generator is used frequently, it is advisable to choose the combination of the Line Elimination Scheme with the Graph Coloring Scheme because it is an order of magnitude faster than the Graph Coloring Scheme alone. For example, the generation of the Ada tables with the Graph Coloring Scheme and column–row coloring took 500 CPU seconds on a SIEMENS 7.760 in contrast to 55 CPU seconds for the same tables using the Line Elimination Scheme with the Graph Coloring Scheme and column–row coloring. The compressed tables need 7569 bytes of storage using the Graph Coloring Scheme and 7658 bytes using the Line Elimination Scheme with the Graph Coloring Scheme. For XPL the corresponding time and space requirements were 17 CPU seconds versus 1.9 CPU seconds and 1493 bytes versus 1513 bytes.

Contrary to the development phase of a compiler, in a production compiler the generation time for the tables is negligible compared to the access time to the table entries, which is twice as fast for the Graph Coloring Scheme than for the Line Elimination Scheme in combination with the Graph Coloring Scheme.

Whether it is useful to add one of the sophisticated extensions to the schemes depends on the weight of three different aims: portability, fast access, and saved storage. It is our opinion that the Graph Coloring Scheme alone or in combination with the Line Elimination Scheme best meets these requirements.

ACKNOWLEDGMENTS

We wish to thank the referees for their helpful criticism and G. Goos, H. Müller, J. Röhrich, F.-P. Schmidt-Lademann, Y. Shimamoto, and G. Wrightson for valuable discussions and comments.

REFERENCES

1. Reference Manual for The Ada Programming Language. Proposed Standard Document. U.S. Department of Defense, July 1980.
2. ANDERSON, T. Syntactic analysis of LR(K) languages. Ph.D. dissertation, Univ. of Newcastle upon Tyne, England, 1972.
3. ANDERSON, T., EVE, J., AND HORNING, J.J. Efficient LR(1) parsers. *Act. Inf. 2* (1973), 12–39.
4. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
5. BEACH, B. Storage organization for a type of sparse matrix. In *Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory and Computing*, (Feb. 25–Mar. 1, Winnipeg). Florida Atlantic Univ., Boca Raton, 1974, pp. 245–252.
6. BELL, J.R. A compression method for compiler precedence tables. In *Proceedings of the IFIP Congress 74*, Booklet 2, (Aug., Stockholm). Elsevier North-Holland, New York, 1974, pp. 359–362.

7. BRÉLAZ, D. New methods to color the vertices of a graph. *Commun. ACM 22*, 4 (Apr. 1979), 251–256.
8. BURGER, W.F. A parser generation tool for micro computers. *COMPSAC 3* (Nov. 1979), 631–634.
9. DENCKER, P. Ein neues LALR-System. Diploma thesis, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1977.
10. DENCKER, P., DÜRRE, K., AND HEUFT, J. Optimization of parser tables for portable compilers. Int. Ber. 22/81, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1977.
11. DÜRRE, K. Approximate Algorithms for Coloring Large Graphs. In *Datenstrukturen, Graphen, Algorithmen*, J. Mühlbacher, Ed. Hanser Verlag, Munich, 1978, pp. 192–203.
12. DÜRRE, K., AND FELS, G. Efficiency of Sparse Matrix Storage Techniques. In *Discrete Structures and Algorithms*, U. Pape, Ed. Hanser Verlag, Munich, 1980, pp. 209–221.
13. DÜRRE, K., GOOS, G., AND SCHMITT, A. Minimal representation of *LR*-parsers. Unpublished Manuscript, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1976.
14. DÜRRE, K., HEUFT, J., AND MÜLLER, H. Worst and best case behaviour of an approximate graph coloring algorithm. In *Proceedings of the 7th Conference on Graphtheoretic Concepts (WG 81)*, (Linz, Austria, June 15–17). Hanser Verlag, Munich, 1982, pp. 339–348.
15. EVEN, S., LICHTENSTEIN, D.I., AND SHILOAH, Y. Remarks on Zeigler's method for matrix compression. Unpublished manuscript, 1977.
16. FELS, G. Effizienzuntersuchungen von Speicherungsmethoden für dünnbesetzte Matrizen. Diploma thesis, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1979.
17. GAREY, M.R., AND JOHNSON, D.S. *Computers and Intractibility.* Pitman, San Francisco, 1979.
18. GOOS, G. Die Programmiersprache LEX. Int. Ber. 1/75, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1975.
19. GOOS, G. Die Programmiersprache BALG, vorläufige Fassung. Int. Ber. 6/75, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1975.
20. GRIMMET, G.R., AND MCDIARMID, G.J.H. On colouring random graphs. *Math. Proc. Cambridge Phil. Soc. 77* (1975), 313–324.
21. HEUFT, J. Untersuchungen über das Approximationsverhalten von Graphenfärbungsalgorithmen. Diploma thesis, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1981.
22. ICHBIAH, J.D. LIS Reference Manual. UB D Dv WS SP31, Siemens AG München 70, 1978.
23. JAKOB, W. Entwurf und Implementierung eines Compilers für eine höhere Manipulatorsprache zur Programmierung eines Roboters. Diploma thesis, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1980.
24. JENSEN, K., AND WIRTH, N. PASCAL User Manual and Report. Springer Verlag, New York, 1978.
25. JOHNSON, D.S. Worst case behavior of graph coloring algorithms. In *Proceedings of the 5th S. E. Conference on Combinatorics, Graph Theory, and Computing*, (Feb. 25–Mar. 1, Winnipeg). Florida Atlantic Univ., Boca Raton, 1974, pp. 513–527.
26. JOHNSON, S.C. YACC—Yet another compiler compiler. CSTR32, Bell Laboratories, Murray Hill, N.J., 1975.
27. JOLIAT, M.L. On the reduced matrix representation of LR(K) parser tables. Tech. Rep. CSRG-28, Univ. of Toronto, Canada, 1973.
28. JOLIAT, M.L. Practical minimisation of LR(k) parser tables. In *Proceedings of the IFIP Congress 1974*, (Aug. Stockholm). Elsevier North-Holland, New York, pp. 376–380.
29. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, Miller, R.E., and Thatcher, J.W. Eds. Plenum Press, New York, 1972, pp. 85–103.
30. KASTEN, U. Die Beispielprogrammiersprache MINILEX. Internal working paper, Fak. f. Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany, 1978.
31. LALONDE, W.R. An efficient LALR parser generator. Tech. Rep. CSRG-2, Univ. of Toronto, Canada, 1971.
32. MCKEEMAN, W.M., HORNING, J.J., NELSON, E.C., AND WORTMAN, D.B. The XPL compiler generator system. In *Proceedings of AFIPS Fall Joint Computer Conference*, (San Francisco, Calif., Dec. 9–11), vol. 33. AFIPS Press, Reston, Va., pp. 617–636.
33. MUJTABE, S., AND GOLDMAN, R. AL user's manual. Tech. Rep. Stan-Cf-79-718, Stanford Artificial Intelligence Laboratory, Stanford, Calif., 1979.
34. NAUR, P. (EDITOR) Revised report on the algorithmic language ALGOL60. *Numer. Math. 4* (1963), 420–452.

35. PAGER, D. Eliminating unit productions from LR parsers. *Act. Inf. 9* (1977), 31–59.
36. POOLE, P.C. *Portable and Adaptable Compilers.* Lecture Notes in Computer Science, vol. 21. Springer Verlag, New York, 1974.
37. TEWARSON, R. Row column permutation of sparse matrices. *Comp. J. 10* (1967/68), 300–305.
38. TARJAN, R.E., AND YAO, A.C. Storing a sparse table. *Commun. ACM 22*, 11 (Nov. 1979), 606–611.
39. SCHMITT, A. Minimizing storage space of sparse matrices by graph coloring algorithms. In *Graphs, Data Structures, Algorithms.* Nagl, M., and Scheider, H.-J., Eds. Hanser Verlag, Munich, 1979, pp. 157–168.
40. WIRTH, N., AND HOARE, C.A.R. A contribution to the development of ALGOL. *Commun. ACM 9*, 6 (June 1966), 413–431.
41. WIRTH, N., AND WEBER, H. Euler: A generalization of Algol, and its formal definition: Part I. *Commun. ACM 9*, 1 (Jan. 1966), 13–23.
42. ZEIGLER, S.F. Smaller faster table driven parser. Unpublished manuscript, Madison Academic Computing Center, Univ. of Wisconsin, Madison, 1977.

Received April 1982; revised February and October 1983; accepted December 1983