A Grammar Compiler for Connected Speech Recognition

Michael K. Brown and Jay G. Wilpon, *Senior Member, IEEE*

***Abstract*—It is well known that syntactic constraints, when applied to speech recognition, greatly improve accuracy. However, until recently, constructing an efficient grammar specification for use by a connected word speech recognizer was performed by hand and has been a tedious, time-consuming task prone to error. For this reason, very large grammars have not appeared.**

We describe a compiler for constructing optimized syntactic digraphs from easily written grammar specifications. These are written in a language called grammar specification language (GSL). The compiler has a preprocessing (macroexpansion) phase, a parse phase, graph code generation and compilation phases, and three optimization phases. Digraphs can also be linked together by a graph linker to form larger digraphs. Language complexity is analyzed in a statistics phase.

Heretofore, computer generated digraphs were often filled with redundancies. Larger graphs were constructed and optimized by hand in order to achieve the required efficiency. We demonstrate that the optimization phase yields graphs with even greater efficiency than previously achieved by hand. We also discuss some preliminary speech recognition results of applying these techniques to intermediate and large graphs.

With the introduction of these tools it is now possible to provide a speech recognition user with the ability to define new task grammars in the field. GSL has been used by several untutored users with good success. Experience with GSL indicates that it is a viable medium for quickly and accurately defining grammars for use in connected speech recognition systems.

I. Introduction

OVER the past several years there have been many breakthroughs in the area of automatic speech recognition. As the capabilities of connected word automatic speech recognition systems have improved, the tasks to which they have been applied have also become more sophisticated [3]–[6], [8], [9], [16], [20], [21]. Such tasks include a speech controlled robot (SCR) [9], flight information and reservation retrieval (FIRL) [17], naval resource management (DARPA-1000, developed by Bolt, Beranek, and Newman) [20], and recognition of office correspondence text [6]. These systems range in vocabulary size from several tens of words to several thousand words [6], [20], [21]. Because current acoustic pattern matching techniques are not perfect, the above systems have had to augment their algorithms with increased knowledge of the task syntax and semantics. By combining these linguistic principles (which constrain the language) with good acoustic recognition algorithms, the overall accuracy of the speech recognition system has improved dramatically.

One difficulty in building connected speech recognition systems with large vocabularies lies in specifying an efficient representation of the syntax to be used. Without syntactic constraints, connected speech recognition accuracy is often poor. Syntactic constraints are, however, a two-edged sword. While the syntax eliminates unwanted or impossible word combinations it also limits the language that the system can recognize. It is clearly desirable to build very large syntactic specifications.

Efficient syntactic specifications have been constructed in the past by hand-coding digraphs (graphs with directed arcs) that represent the state transitions in a finite state automaton or machine (FSM). To date all commonly known practical connected speech recognition systems use FSM's for grammar representation [14], [16], [20]. Automatic coding of syntactic constraints has been employed before [17], [18], [22] but hand optimization was needed to gain the necessary efficiency. The automata, which are *Moore machines* [19], are used to constrain the effective vocabulary as the speech is being processed, thereby saving considerable computation and eliminating nonsensical phrases. An example of a fully hand-coded system is FIRL [16]. The DARPA-1000 grammar [20] was partly hand processed. The hand-coding of these graphs is a time-consuming, tedious process prone to errors.

In our recent research on the *speech controlled intelligent robot* [8], the robot has an environment with dynamic syntactic and semantic characteristics. That is, as the functionality improves periodically the syntactic and semantic requirements of the robot system change at regular intervals. Our first robot grammar contained about 99 000 sentences with a 51-word vocabulary. Today we are using our eleventh grammar containing about 15 septillion (1.47×10^{24}) sentences with a 125-word vocabulary (perplexity of 12.4). With such rapid development it soon becomes clear that there is a real need for a *grammar compiler* that can quickly and easily build the new graphs.

The grammar compiler has many of the components of any computer language compiler. A preprocessing phase provides file inclusion and macroexpansion. The compile phase parses an easily written input specification and generates "code," i.e., a file containing numbers that indicate states and transitions. The "code" can then be read into a speech recognition program to direct the acoustical processing. A three-stage optimization phase improves the efficiency of the representation. There is also provision for linking several graphs into a larger graph (this relabels the states and branches of the graph, similar to relocating code).

With the grammar compiler we can build a grammar in a few hours that previously would have taken weeks to produce by hand. One such example is the flight information and reservation retrieval (FIRL) grammar, part of the work in syntactic constraints reported by Rabiner and Levinson [21]. This grammar is represented by a graph originally containing 144 states, 497 branches, and 21 terminals. The original grammar was constructed (including debugging) by hand in several days, according to the designer (Levinson [15]). This same grammar, with

Manuscript received March 3, 1989; revised November 30, 1989.
The authors are with AT&T Bell Laboratories, Murray Hill, NJ 07974.
IEEE Log Number 9040375.

additions, was written for the grammar compiler and debugged in a few hours.

Furthermore, the hand-coded grammar had errors that went undetected for years. These residual errors were detected and corrected by the first run of the optimizer described in Section IV-A1. The result of optimizing (deterministic finite automaton conversion and phase I optimization only) the original grammar was a graph containing 125 states, 559 branches, and 10 terminals. The complexity of the graph (language entropy, see Section V) is unchanged, but the number of states needed to represent it is reduced. There are many duplications in the 559 branches and after phase II optimization (Section IV-A2) only 422 real branches remain. The remaining branches are "null" branches, representing state transitions without processing any speech data. These null branches are used for convenience and efficiency to indicate optional or alternative paths in the graphs. Since null branches are essentially cost free, it is the real branches that we care about when determining processing efficiency.

We will describe the input language in the following section. The compiler will be discussed in Section III followed by a section on graph construction and optimization. Section V describes computation of language entropy, some applications of these grammars and the corresponding statistical results. We then conclude in Section VI.

II. Grammar Specification Language

The regular grammar for constraining speech input is specified in the language grammar specification language (GSL) by a set of specification statements that contain representations of patterns that are to be accepted. These patterns are compiled into a FSM having one starting state (state 0) and one or more terminal states. Any single statement having no special forms can simply be written as an ordinary sentence. Each statement must be terminated with a period, which by default, specifies a terminal state unless overridden by a diversion (defined below). In addition, intermediate accepting states can be specified inside a sentence using an asterisk. These states are optional terminal states that may be used as a stopping place or passed through to reach some later terminal state. Comments are specified by a '#'. All text following the '#' to the end of the line is ignored.

It is advantageous to consolidate many spoken sentence possibilities into a single input sentence since this yields a concise specification and reduces the number of states required to represent the grammar, so the use of a *disjunctive* form is provided. This form consists of a parenthetical list of *phrase* elements separated by vertical bars. For example, "find (an|another|the next) object," produces the graph shown in Fig. 1 (in this and all succeeding graphs double circles indicate terminal states). These disjunctions can be nested arbitrarily deeply.

Disjunctions can contain a *null* element indicating that all of the elements of the disjunction are optional. Two forms of the null element are allowed: a) the '[null]' special form causes the digraph generator to "wire" all of the necessary bypassing branches of the graph to the required successor states without using null branches, and b) the NULL-ARC form causes the generation of true null branches (i.e., cost-free transitions to another state). The first digraph form is usable by most past and present connected speech recognizers. The second form is usable only by the newer recognizers and is a more efficient representation.

Disjunctions can obtain digraph *diversions*. Diversion registers are provided that can be used to divert the current subpath of the digrah from its normal flow to any place designated by '@n,' where n is a number from 0 to 999. This is accomplished by the use of a '&n' form ('&n' is like "go to @ n" where '@n' is a label). This mechanism can be used to generate cycles in the grammar as well as allow creation of certain desirable forms for some of our special purpose speech recognition hardware. Note that *statements* in the grammar given below are effectively disjunctive elements (although they are handled differently) and can be diverted to other statements or back into themselves if desired. Diversion of a statement into itself can result in the specification of a path in the digraph that never terminates, however, so care must be exercised.

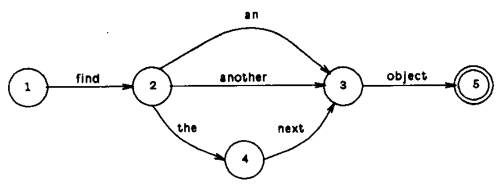

Fig. 1. State diagram for "find (an|another|the next) object."

An explicit specification of the input grammar is given as follows:

```
⟨grammar⟩      =
                 | ⟨grammar⟩ ⟨statement⟩

⟨statement⟩    =  ⟨toplist⟩ '.'

⟨toplist⟩      =  ⟨phrase⟩
                 |  ⟨phrase⟩ '&n'

⟨phrase⟩       = ⟨element⟩
                 |  ⟨phrase⟩ ⟨element⟩
                 |  ⟨phrase⟩ '*'
                 |  '@n'⟨element⟩
                 |  ⟨phrase⟩ '@n'

⟨element⟩      =  ⟨word⟩
                 |  '(' ⟨disjunction⟩ ')'
                 |  '(' ⟨phrase⟩ ')'

⟨disjunction⟩  = ⟨list⟩ '|' ⟨list⟩
                 |  ⟨disjunction⟩ '|' ⟨list⟩

⟨list⟩         = ⟨toplist⟩
                 |  '[null]'

⟨word⟩         = [A–Z a–z–'-]
```

where the basic element is an arbitrary alphabetic word consisting of any number of the indicated symbols. The only restriction on the location of asterisks is that they cannot appear at the beginning of the statement since this would specify the start state as an accepting state. An asterisk at the end of a sentence is accepted, but redundant and thus has no real effect. Diversion registers ('@n') can be set anywhere. Digraph diversions ('&n') can only occur at the end of a subphrase. Note that, unlike some earlier grammar specification methods (e.g., HARPY's BNF specification with "nonterminal intersection ambiguities") [17], GSL specifies completely unambiguous grammars.

The input grammar specifications are initially processed by the UNIX® m4 macro preprocessor (AT&T Bell Laboratories, 1986). Therefore, the grammar description may be augmented with the macrofacilities provided by m4. In the grammars that have recently been developed, we have relied heavily on the m4 macro **define**. The second argument of **define** is installed as the replacement text of the macro whose name is the first argument. If the replacement text contains a macro name it will be recursively expanded. For example, using the statement **define (ABC, Good Morning)** causes the text **Good Morning** to be inserted wherever the variable **ABC** exists. A toy example of GSL without macros is shown in Fig. 7(a). An actual grammar specification currently being used in our research, which uses most of the features of the grammar compiler, appears in Appendix I.

III. Digraph Construction

Digraphs are constructed by top-down parsing of the input grammar. There are two forms of digraph construction. The first form contains only real (nonnull) branches. To illustrate, let us consider the following grammar specification:

define(N, [null])

(a | b | c) ((a | b | c) (a | b | c | N) | N).

The first line in this specification is a macro that causes each instance of 'N' in the grammar specification to be expanded to '[null]'. The corresponding digraph is shown in Fig. 2(a). The multiple branches between two states are shown as one branch with multiple labels. In this case the null disjunction elements result in alternate paths for each possible case. This unfortunately causes the resulting digraph to be large.

The second digraph form is obtained by substituting NULLARC for [null] in the first macrodefinition. The resulting digraph is shown in Fig. 2(b) (unlabeled branches are null). This form is used with hidden Markov model based recognizers currently under research at AT&T Bell Laboratories [15]. A third form, shown in Fig. 2(c), can be generated by conversion of the graph in Fig. 2(a) to deterministic form (more about this in Section IV).

The internal representation of the digraph is constructed in part after each statement is parsed. Digraph branches are collected into "from-state" structures. After all statements are processed these from-state structures are relabeled (if diversions are present), sorted and split into transition rules (state connectivity sets in the graph), which are compiled into the digraph map file. A (generally small) list of terminal states is generated as the parse trees are searched. This list is ultimately sorted uniquely (there often are redundancies) and the results deposited in the digraph map file after the state transition rules. A simple symbol table of vocabulary words is also maintained on a hash table. Digraphs containing several thousand states can be compiled in less than 30 s on a Sun3® workstation.

IV. Digraph Minimization

The graphs constructed by the compile phase of the digraph compiler are nondeterministic finite state automata (NFA) representations and are generally suboptimal in number of states and branches since the digraph is a direct translation of the input specification, which may be poorly written. Here, NFA implies that, given a particular state and symbol, the next state is not uniquely defined. Redundant paths in the digraph cause unnecessary computation during the acoustic matching of input speech since duplicate pattern templates will sometimes be compared simultaneously while searching different parts of the digraph.

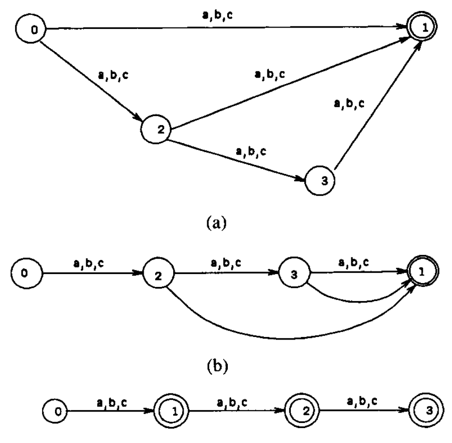

Fig. 2. (a) Digraph without null branches. (b) Digraph with null branches. (c) DFA from Fig. 1(a).

This additional processing can make the difference between achieving real-time performance or not.

For speech recognition, it is important to reduce the number of branches, which represent word tokens, as much as possible. Such reductions are achieved in three ways. First, a reduction in the number of states removes redundant branches associated with the redundant states. Second, redundant branches are examined directly and replaced by single branches using "null" branches for proper connection (and this process generally adds states). The third phase eliminates redundant "null" branches and, thus, removes some states.

Two approaches to minimization are taken. The first attempts to reduce the NFA directly by searching for isomorphic and homeomorphic (isomorphic to a subset) subgraphs that can be removed. The second approach relies on the fact that most of our grammars can be represented as deterministic finite state automata (DFA) that are no larger than the NFA. There is no guarantee that the DFA will be as small as the NFA, for, in general, an NFA with n states may grow to as large as 2^n states when converted to a DFA. An NFA optimizer is needed in these cases. However, parts of speech (which are few in number) tend to cluster around similar sets of successor states in the NFA, i.e., verbs will all generally cause similar state transitions, nouns will cause another set. The number of such sets (which determines the number of DFA states) is generally smaller than the number of NFA states in the union of sets. Thus the DFA is often smaller than the NFA by a significant amount. Once the DFA in obtained, well-known DFA minimization algorithms that guarantee optimality can be applied [12].

A. NFA Reduction

Three digraph optimization phases are available to reduce redundancies. There are, however, some redundancies that are not removed. Much or all of the remaining redundancy can be eliminated by modifying the input specification. In general, the only way to guarantee that no redundant paths remain is to convert the NFA into a DFA and apply a DFA minimization algorithm. Frequently, however, direct application of the NFA optimizer achieves the same results. The advantage of direct application of the NFA optimizer is that a larger number of states are not added to the graph, whereas, taking the alternative approach of DFA conversion and reduction sometimes results in a much larger number of states.

The first optimization phase identifies isomorphic or homeomorphic subgraphs and eliminates them. The second optimizer replaces duplicate subsets of branches with a single subset and null branches connecting the original source states to this subset. The third optimizer searches for strings of null branches and replaces them with single null branches where possible. All optimization phases are polynomial time algorithms, however, phase one may take considerable time if the graph is large. The optimization effort is quadratic in the number of states for phase one and linear in the number of branches for phases two and three. Phase I optimization of large digraphs may take many minutes of processing time. It is the only stage that requires more than one minute of processing time for most grammars (on a Sun 3 workstation, for example).

1) Phase I Optimization: The digraph reduction algorithm identifies five types of digraph redundancy. There are additional redundancy types that are not detected. Common forward and retrograde isomorphic subgraphs, homeomorphic subgraphs, and redundant production paths and branches are found. Retrograde forms are simply those obtained from the graph by reversing the sense of direction of the digraph branches. The five types are illustrated in Fig. 3. In this diagram the start state is labeled 0 and the terminal state is 1. A trivial example of a forward isomorphic redundancy is the subgraphs connected between states 8-1 and 10-1. Redundant branches are connected between states 7 and 8. The subgraph of states 2-9-10-1 is homeomorphic to the subgraph of states 2-7-8-1. A retrograde isomorphic equivalence occurs at states 0-3 and 0-5. Redundant productions occur for states 3-4-2 and 5-6-2. In this case either the 3b4 or the 6d2 branch can be removed without altering the grammar.

The basic framework of the algorithm for identifying these redundancies is an extension of a well-known algorithm for DFA minimization. See, for example [12, p. 70] or [13, p. 302]. Additional capability has been added to identify retrograde and homeomorphic subgraphs as well as other redundant branching patterns. The algorithm identifies state pairs that have common descendants or descendants that cannot be distinguished by language accepted by the automaton. In this case, however, we are working with an NFA that contains the various types of redundancies illustrated in Fig. 3.

The basic DFA algorithm consists of marking state pairs in a table when it is known or can be shown that the two states are distinct, i.e., identical word sequences do not lead to the same states. Each state pair is considered only once so the table is a triangular matrix. Initially all pairs of states consisting of one terminal (accepting) and one nonterminal state are marked. Then, for each unmarked state pair, the production rules are commonly applied at each state and if all of the resulting state pairs are not marked in the table then a pointer back to the current state pair is attached from each resulting state pair. If any of the resulting state pairs are marked in the table then the current state pair is also marked, along with any state pairs pointed to by this state pair and, recursively, any state pairs pointed to by the marked state pairs. After all state pairs have been processed in this manner, the remaining unmarked state pairs are isomorphically indistinguishable. An inductive proof for the DFA algorithm is given by Hopcroft and Ullman [12].

To see how the algorithm works consider first a state pair consisting of terminal state p and nonterminal state q, designated (p, q). Let the set of sentence segments that can be found

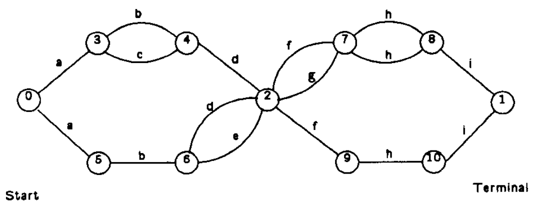

Fig. 3. Example state digram.

from the start state to state p be designated S_p, and from the start state to state q be designated S_q. We can only combine p and q if both are terminal (accepting) states or both are nonterminal states or if $S_p \equiv S_q$ since any other combinations would allow additional sentences to be accepted or eliminate sentences. That is, to merge p and q either S_p and S_q must both be entirely accepted as complete sentences or neither can be entirely accepted unless $S_p \equiv S_q$, in which case they are accepted if either p or q are terminal states. Since the algorithm cannot tell if $S_p \equiv S_q$ all terminal/nonterminal state pairs must be marked as distinguishable without further knowledge. This is equivalent to taking the safest approach; if we cannot prove that p and q are equivalent we must assume that they are not. Later we will show that these states can also be combined in an NFA when retrograde isomorphisms are identified.

Now consider an unmarked state pair (p, q). Let the set of symbols for which a transition exists at any state be defined by the alphabet Σ. Let the descendant of state p be $r = \delta(p, x)$, where δ is a transition rule, when input $x \in \Sigma$ is applied at state p, and let the descendant of q be $s = \delta(q, x)$. Assume state pair (r, s) is marked. Then the set of sentence segments or language from r to a terminal state is distinguishable from the language from s to a terminal state. Call these L_r and L_s, respectively. Then the composition xL_r is a longer sentence segment from p to a terminal state and likewise for xL_s from q to a terminal state. But, since (r, s) is marked, $xL_r \not\equiv xL_s$, the states in (p, q) are distinguishable and must also be marked. If (r, s) is not marked then we assume until proven otherwise that $L_r \equiv L_s$ and $xL_r \equiv xL_s$. If the descendants are unmarked for all $x \in \Sigma$ then a pointer to (p, q) is attached from all $D = \{(r, s) \mid r = \delta(p, x), s = \delta(q, x), x \in \Sigma\}$. The trivially indistinguishable case occurs when $\delta(p, x) = \delta(q, x)$ for all $x \in \Sigma$. If at any future time any one or more members of D should prove to be distinguishable, either by direct inspection or via pointers from other state pairs, then (p, q) will also be distinguishable and will be marked. Thus, any state pairs that remain unmarked after all state pairs have been exhaustively considered are indistinguishable up to an isomorphism.

In our case we have an NFA, i.e., there may be more than one path for a particular symbol from a given state. For an NFA the set of symbols for which a transition exists at any state p is defined by the possibly null alphabet Σ_p. Now we are not only interested in isomorphic redundancies but also homeomorphic redundancies, i.e., a graph G isomorphic to a subgraph of graph H. Identifying such homeomorphisms requires looking both forward and backward (retrograde) from a state in the digraph. The set of symbols accepted in each direction from a subset state must be a subset of the set of symbols accepted in the corresponding direction by the superset state. Such state pairs can be found by expanding the state table to a square matrix of doublets consisting of a forward mark and a retrograde mark. Let the row index indicate the subset state p and the column index indicate the superset state q (i.e., $\Sigma_p \subset \Sigma_q$). Terminal/nonterminal state pairs are marked in the forward table for subset states (but not in the retrograde table). Pairs are not marked for terminal states that are not subset states, i.e., only rows in the mark table are initially marked (see Table I). Then, for each state pair (p, q), with descendants $r = \delta(p, x)$, and $s = \delta(q, x)$, $x \in \Sigma_p$ (note that, in general, r and s may be multiply defined), if $\Sigma_p \subset \Sigma_q$ and all $D = \{(r, s) \mid r = \delta(p, x), s = \delta(q, x), x \in \Sigma_p\}$ are unmarked, then create a pointer to (p, q) from each member of D. If (r, s) is marked or $\Sigma_p \not\subset \Sigma_q$ then mark (p, q) and recursively mark all state pairs pointed to by (p, q).

Similarly, for each state pair (p, q) with ancestors (m, n) the algorithm is applied with the sense of direction in the digraph reversed. The argument is a straightforward extension of the previous argument. The forward and retrograde mark tables can be treated separately since we are looking for subgraphs extending in only one direction at a time. With the additional provision that an improper subset of the alphabet can be accepted at each state, the argument in the forward direction is essentially unchanged. Thus, we are looking for an isomorphism in a subgraph attached to one of the states. Unmarked state pairs (p, q) in the forward part of the table indicate that p is homeomorphically (perhaps isomorphically) equivalent to q in the forward direction. Forward isomorphic equivalence is identified when the forward mark table is unmarked in both symmetric elements of the matrix. That is, $\Sigma_p \subset \Sigma_q$ and $\Sigma_q \subset \Sigma_p$ which can only mean $\Sigma_p = \Sigma_q$ and the states are indistinguishable. No forward isomorphic equivalence will exist between a terminal and nonterminal state since one of the two symmetric elements has been marked initially.

Similarly, retrograde isomorphic equivalence is identified when symmetric elements in the retrograde mark table remain unmarked. However, in the retrograde isomorphic case the terminal states may be indistinguishable from nonterminal states because if p is a terminal state and q is not a terminal state, and if language L_p from the start state is accepted then L_q is also accepted since, by isomorphic equivalence, $L_q \equiv L_p$.

When both the forward and retrograde parts of any state pair element of the mark table matrix are simultaneously unmarked, then a homeomorphic subgraph exists between indistinguishable state pairs and the entire homeomorphic subgraph can be removed without altering the language. Homeomorphic equivalence can occur if the subset state p is nonterminal and the superset state q is terminal (i.e., $\Sigma_p \subset \Sigma_q$) but not if p is terminal and q is nonterminal. To see this consider the language from the start state to p designated L_p and to q designated L_q where $L_p \subset L_q$. If q is terminal then L_p is an accepted language since the superset $L_q = L_p + (L_q - L_p)$ is accepted. However, if p is terminal and q is nonterminal then $L_q - L_p$ is not in the accepted language and p cannot be equivalent to q.

Redundant production paths can be identified from the mark table when the forward element of (p, q) and the retrograde element of (q, p) are both unmarked. In this case, a forward homeomorphism exists from p and a retrograde homeomorphism exists from q. Then branches common to the subset graph from p forward can be removed from the superset graph, or branches common to the subset graph from q backward can be removed but only one of these sets can be removed. For example, in Fig. 3 we can remove the 3b4 branch or the 6d2 branch and not reduce the accepted language because ''abd...'' is represented twice between indistinguishable state pairs. Redundant branches such as 7h8 are, of course, trivially removed by checking uniqueness at each state.

TABLE I
MARK TABLE AND STATISTICS FOR GRAPH IN FIG. 3

```
Optimizing...
   Starting with 16 branches, 12 rules and 11 nodes.
      Node     7 is nonterminal accepting state in digraph
Mark table before search (backward, forward):
00    00    00    00    00    00    00    00    00    00    00
01    00    01    01    01    01    01    00    01    01    01
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
01    00    01    01    01    01    01    00    01    01    01
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
00    00    00    00    00    00    00    00    00    00    00
Mark table after search (backward, forward):
00    01    01    01    01    01    01    01    01    01    01
11    00    11    11    11    11    11    10    11    11    11
11    11    00    11    11    11    11    11    11    11    11
11    11    11    00    11    01    11    11    11    11    11
11    11    11    11    00    11    10    11    11    11    11
11    11    11    01    11    00    11    11    11    11    11
11    11    11    11    01    11    00    11    11    11    11
11    11    11    11    11    11    11    00    11    11    11
11    11    11    11    11    11    11    11    00    11    10
11    11    11    11    11    11    11    00    11    00    11
11    11    11    11    11    11    11    11    00    11    00
      Branch reduction possible for nodes 4 and 6
      Isomorphic reduction: relabeling node 8 to 10
      Homeomorphic reduction: relabeling node 9 to 7
      Retrograde isomorphic reduction: relabeling node 3 to 5
   There were 1 isomorphic, 1 homeomorphic, and 1 retrograde
     isomorphic reductions.
   Optimized digraph contains 11 branches, 8 rules and 8 nodes.
```

The mark table for the grammar of Fig. 3 is shown in Table I along with some statistics output by the optimizer. The rows and columns represent states 0 through 10 from top to bottom and from left to right. In each doublet the first digit is the mark for retrograde and the second digit is the mark for forward productions. Initially only the forward elements of rows 1 and 7 (the terminal states) are marked when paired with nonterminal states. After marking all distinguishable state pairs the mark table contains patterns that indicate the type of reduction possible. Note that state pair (8, 10) could have been reduced on either isomorphic or homeomorphic grounds.

2) Phase II Optimization: The second phase of optimization identifies intersections of sets of branches connected to a particular state from various other states. For example, consider the subgraph of Fig. 4. The subgraph contains 12 branches from states 2, 3, and 4 to state 1. Branch A is present in all three sets. Branch B appears only from states 2 and 4, C only from 2 and 3, D only from 3 and 4. Branches E, F, and G occur only once each. Fig. 5 illustrates the from-state sets, set intersections and the resulting digraph with null (unlabeled) branches. Even though the resulting subgraph contains four more states and four additional branches, only seven of the branches cost any recognition processing time and there is only one instance of each branch type.

3) Phase III Optimization: Null arcs are added to the graph in two phases of the processing by the grammar compiler, the compile phase, and the second optimization phase. Because of this it is possible to have unnecessary sequences of null arcs. Furthemore, the grammar specification written by the program-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.