Inferring Lexical and Grammatical Structure from Sequences

Craig G. Nevill Manning
Biochemistry Department
Stanford University
Stanford, CA 94305-5307
cnevill@stanford.edu

Ian H. Witten
Department of Computer Science
University of Waikato
Hamilton, New Zealand
ihw@cs.waikato.ac.nz

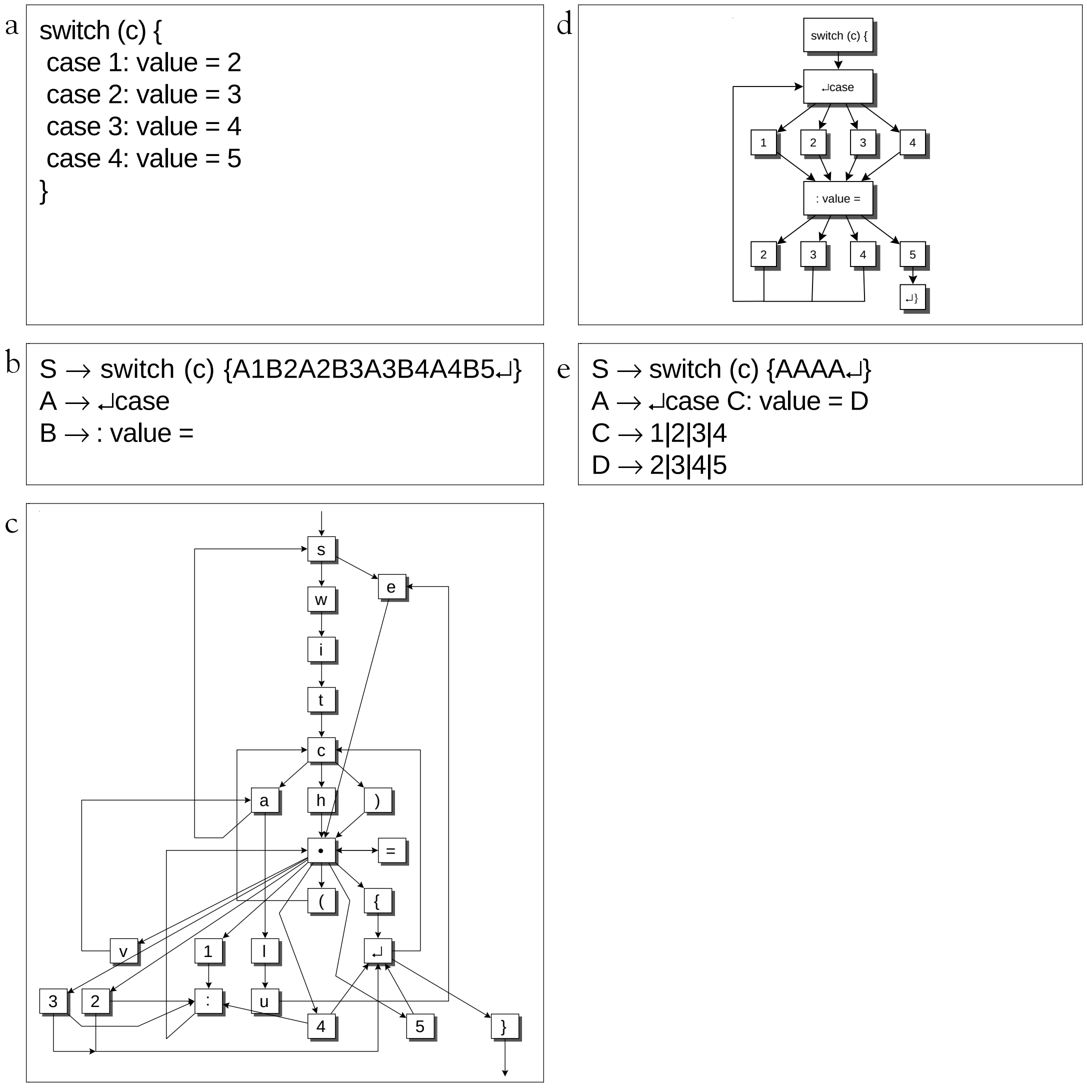
1 Introduction

In a wide variety of sequences from various sources, from music and text to DNA and computer programs, two different but related kinds of structure can be discerned. First, some segments tend to be repeated exactly, such as motifs in music, words or phrases in text, identifiers and syntactic idioms in computer programs. Second, these segments interact with each other in variable but constrained ways. For example, in English text only certain syntactic word classes can appear after the word 'the'—many parts of speech (such as verbs) are necessarily excluded.

This paper shows how these kinds of structure can be inferred automatically from sequences. Let us make clear at the outset what aspects of sequence structure we are *not* concerned with. We take no account of numerical frequencies other than the 'more than once' that defines repetition. We do not consider any similarity metrics between the individual symbols that make up the sequence, nor between 'similar' subsequences such as transposed or transformed motifs in music. Finally, although we are certainly interested in *nested* repetitions, we do not analyze *recursive* structure in sequences—such as self-similarity in fractal sequences. All of these regularities are interesting ones that would be well worth taking into account, but lie beyond the scope of this paper.

We begin with an example that both illustrates the utility of inferring the kinds of structure we seek and shows what our techniques can do. Next we present an efficient and non-obvious algorithm for identifying exact repetitions—including nested repetitions—in time which is linear with the length of the sequence. Then we describe a very simple algorithm for identifying interactions between sequence elements. The focus of this paper is on how these two algorithms can work together, for their combination is far more powerful than either alone. In Section 5 we show how they combine to generate the kind of structure sought in the original motivating example.

Although the two methods work well together on many simple examples, the results frequently conflict with intuition in the inference of branching structure. Section 6 discusses this difficulty and possible workarounds, none of which seem to provide a completely satisfactory general solution. It is noted that for small-scale problems intuition can be misleading because of the syntactic and semantic knowledge that we bring to most sequence inference situations. The minimum description length principle seems to provide the only satisfactory general approach, and Section 7



a
```
switch (c) {
 case 1: value = 2
 case 2: value = 3
 case 3: value = 4
 case 4: value = 5
}
```

b
```
S → switch (c) {A1B2A2B3A3B4A4B5↵}
A → ↵case
B → : value =
```

e
```
S → switch (c) {AAAA↵}
A → ↵case C: value = D
C → 1|2|3|4
D → 2|3|4|5
```

Figure 1 Detecting the structure of a *switch* statement (a) the original statement; (b) grammar inferred for (a); (c) automaton formed from the individual characters in (a); (d) automaton formed from the contents of rule S; (e) a grammatical representation of (d).

shows how it can be used to solve challenging problems concerning the inference of branching structure in large-scale situations where sufficient raw data is available. A satisfactory account of inference of branching structure in short sequences, however, remains an open problem.

2 A motivating example

Figure 1 shows part of a program resembling a C switch statement.[1] Visually its structure is clear: the keyword *switch* and a pair of braces enclosing a list of *case* labels with colons and line breaks in well-defined places. Much of this clarity stems from layout—the special symbols that represent space and line breaks—along with (no doubt!) the reader's familiarity with the C programming language. Our aim in

[1] The absence of semicolons and a *break* statement at the ends of the lines will be explained later.

this paper is to detect this structure automatically, without any specialist knowledge.

Grammar and automaton inference techniques will be described in Sections 3 and 4 respectively. Given a particular sequence, the grammar inferencer finds a hierarchical structure that describes it. Although it detects and eliminates repetition at the lexical level, it is blind to looping and branching. The automaton inferencer re-expresses the sequence in a way that makes looping and branching explicit, although in doing so it loses the original sequence and obscures lexical repetition.

Applying grammatical inference to the sequence yields the representation in Figure 1b. The starting rule *S* regenerates the original sequence exactly when the non-terminals *A* and *B* are expanded. In general (although not in this extremely simple example) rules will be nested, and the nesting reveals potentially interesting regularities in the sequence. Although exact repetitions in the sequence have been incorporated into the grammar—the words *case* and *value* are each represented by a single non-terminal—repetitions that involve variation remain undetected—the grammar is oblivious to the repetitive *case* … *value* … substructure.

The fact that the sequence comprises a fixed skeleton interspersed with variable parts suggests that a different modeling technique may abstract the branching information. Automaton inference at a character level results in the structure of Figure 1c. The messy graph reveals nothing about any regularities in the sequence. The problem is that the branching and looping does not apply to individual symbols but to groups of symbols. For example, the automaton contains a single state for the symbol *s* regardless of whether it is used in the word *switch* or *case*.

Neither technique on its own successfully captures the structure of the sequence. However, their combination—grammar inference for identifying significant segments like *case* and *value*, and automaton inference for capturing branching and looping—is quite successful. Presenting rule *S* of Figure 1b as a sequence to the automaton inferencer yields the structure of Figure 1d, which describes the original statement well. The various possibilities 1, 2, 3, and 4 that separate the *case* node from the *value* node naturally suggest a kind of branching structure. An alternative to this representation appears in Figure 1e, and we return to this later.

3 Lexical inference

SEQUITUR infers a hierarchical description of recurring segments of a sequence. Space does not permit a thorough exposition of SEQUITUR's implementation and application, but a comprehensive description can be found in Nevill-Manning (1996). Also, the SEQUITUR web site, `http://www.cs.waikato.ac.nz/sequitur`, includes an on-line system that infers hierarchies from user-supplied sequences.

a

abcabdabcabd	S → AA
	A → BcBd
	B → ab

c

S → AA	S → AA
A → Cd	A → BcBd
B → bc	B → ab
C → BcB	

b

S → abcab	S → AcA
	B → ab

Figure 2 Forming a grammar from a sequence.

Figure 2a shows a short sequence, *abcabdabcabd*, and the grammar inferred from it. The grammar provides both a proposed structure and a compressed representation for the sequence. The SEQUITUR algorithm requires time linear in the length of the input sequence, despite the fact that the grammar that it builds can be as deep as $O(\sqrt{n})$ (Nevill-Manning, 1996). It operates by enforcing of two constraints on the grammar. First, in the left hand sides of the grammar rules, no digram (pair of symbols) can appear more than once. If such a digram is present, a new rule is formed to factor out the repetition. Figure 2b shows the formation of rule *B* to remove the repetition of *ab*. Second, every rule must be referenced more than once. If a rule is used only once, it is expanded and removed from the grammar. Figure 2c shows how rule C, which is used only once, is deleted from the grammar. The two constraints eliminate redundancy from the grammar, and—almost as a by-product—create structure. Applying these rules incrementally as symbols are appended to rule *S* of the grammar results in an algorithm that, with appropriate data structures, runs efficiently, and typically processes sequences at the rate of two million symbols per minute on a workstation. This is vastly in excess of earlier schemes such as Wolff's (1980).

The four million symbol sequence of letters in the French Bible produces a grammar with 98,000 rules, 652,000 symbols in the right hand sides of rules, and 399,000 symbols in rule *S*. A small excerpt is reproduced in Figure 3a as a collection of trees, where each branch corresponds to a rule expansion. Most of the word boundaries have been correctly inferred, despite SEQUITUR's ignorance of the special significance of spaces. Furthermore, the word *commencement* has been subdivided into *commence*, the root verb, and *ment*, the suffix that makes the word a noun. The inference that this is an appropriate subdivision of the word is based on the observation that *commence* appears elsewhere with a different suffix, and *ment* appears elsewhere following a different root. Similarly, the phrase *cieux et la terre* is identified because it appears in 19 different places in the text of the Bible. A similar structure can be inferred from English, German, and Italian versions of this sentence—the last appears in Figure 3b.

SEQUITUR excels at identifying exact lexical structure such as words, word parts, and multi-word phrases. However, it does not generalize, and the inferred grammar is capable of reproducing only the input sequence—SEQUITUR cannot characterize the various ways in which words can interact. The clearest symptom of this is the size of rule *S*, which typically accounts for one third of the entire grammar. Rule *S* contains the original sequence with exact repetitions factored out. As far as SEQUITUR is concerned, it is the unstructured part. And this is where a more subtle layer of structure can be found.

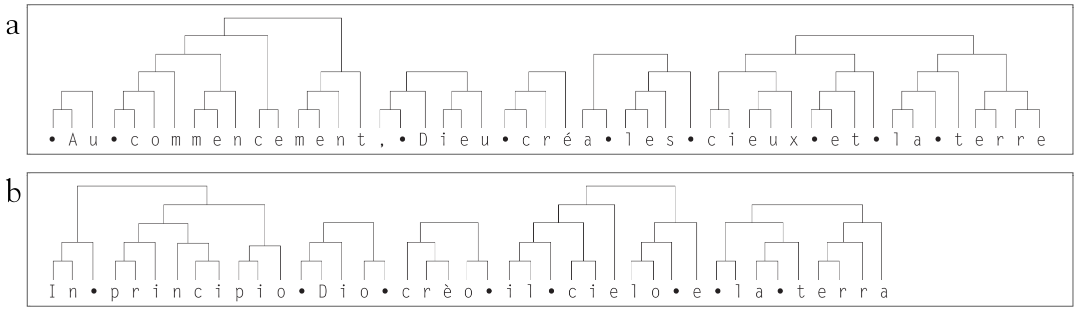

a
• A u • c o m m e n c e m e n t , • D i e u • c r é a • l e s • c i e u x • e t • l a • t e r r e
b
I n • p r i n c i p i o • D i o • c r è o • i l • c i e l o • e • l a • t e r r a

Figure 3 Excerpts from hierarchies inferred from two versions of the Bible.

4 Inference of branching structure

A simple way of inferring branching and looping structure was proposed by Gaines (1976) as a way to learn a procedure from an example of its execution. The idea is to take a sequence of symbols and form an automaton with one state for every unique symbol and transitions between every pair of nodes whose symbols are adjacent in the sequence.

For example, forming an automaton from the string *abcabdabcabd* yields the automaton in Figure 4, representing an alternative structure to the grammar of Figure 2a. The automaton suggests possible branching and looping structure in the sequence, and is capable of generating, in addition to the input, many strings of different length and composition. But generalization has a price: additional information is required to recreate the original string, and the fact that the string consists of two exact repetitions has been lost.

On a short abstract sequence, it is difficult to argue that one model is a better description of the sequence than the other. A longer sample would clarify whether the sequence consists of repeated *abcabd* segments, or whether the repetition is coincidental and the underlying model is a less restrictive branching structure.

Figure 1a is long enough to allow subjective evaluation of potential structures, and the automaton of Figure 1c fails dramatically to capture the structure that we expect. It allows almost any symbol to precede or follow a space, and can generate words such as *caswitcaswitch* from the promiscuous transitions between symbols. The problem is that the sequence is expressed at an inappropriate level of granularity: the symbol *s*, on its own, does not have a well-defined lexical role—it is significant only within the context of the tokens *switch* or *case*. The graph represents an over-generalization to an enormous space of strings, and the original sequence could only be recreated using a correspondingly large amount of disambiguation information.

5 Putting them together

The techniques of SEQUITUR and the automaton inference are complementary, and their combination has much greater inferential power than either alone. To integrate them, SEQUITUR processes the entire sequence and factors out repetition by forming a hierarchy of deterministic rules. Rule *S*, now composed partly of non-terminal symbols, is presented as input to the automaton inference mechanism, which produces the automaton shown in Figure 1d. Each node corresponds to a lexical element identified by SEQUITUR, and they participate in branching and looping structure. The two techniques are analogous to the lexical analysis and parsing phases of a compiler, except that the lexicon and grammar are inferred rather than prescribed.

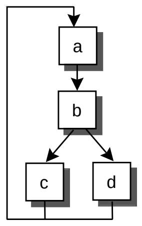


Figure 4 Automaton for *abcabdabcabd*.

6 Problems with inferring non-deterministic structure

We have reached a point where we can deal successfully with examples like that of Figure 1. However, the scheme nevertheless has serious shortcomings, which are exemplified by the fact that the Figure 1 sequence is idealized.

The problem

Figure 5a shows a more plausible version of a *switch* statement that includes a *break* at the end of each *case*. In creating a grammar for this sequence, the rule $A \rightarrow$ ↵ *case* is formed as before, as shown in Figure 5b, but after the third line a longer rule $C \rightarrow DA$ is created, which prepends the *break* statement of the previous line to each of the last three *case* lines. In rule S, then, the first line starts with A, whereas the others start with C. This disrupts the repetitive structure and produces the automaton of Figure 5c. The problem is that in the last three *case* clauses, rule A has submerged beneath the level of rule S, hidden within a higher-level rule.

Potential solutions

It is possible to recover the desired structure in this particular example using a process of 'retrospective reparsing' (Nevill-Manning, 1996). Although it comes at substantial computational cost—its complexity is quadratic rather than linear in the length of the original sequence—this is nevertheless quite feasible for some applications. However, it does not provide a general solution, and we do not pursue it here.

Another approach is to abandon the clear separation between the grammar and automaton inference processes and look for branches incrementally while the

a

```
switch (c) {
 case 1: value = 2; break;
 case 2: value = 3; break;
 case 3: value = 4; break;
 case 4: value = 5; break;
}
```

b

S → switch (c) {A1B2C2B3C3B4C4B5D↵}
A→ ↵ case
B → : value =
C → DA
D →; break;

c

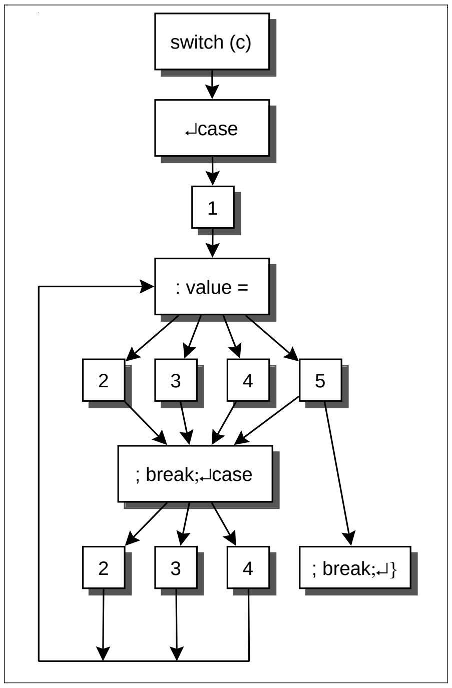


Figure 5 A *switch* statement with *break*: (a) the text of the statement; (b) grammar inferred for (a); (c) automaton formed from rule S of (b).

grammar is being formed. Consider how this might work for Figure 5. The first *case* label encountered is not preceded by a *break*, and when the second one appears, rule *S* contains the sequence '*switch* (*c*) {*A*1*B*2*break*;*A*2*B*'. The *A*…*B*…*A*…*B* structure signals a branch, which, once formed, will match the third and fourth *case* statements and prevent them merging with the preceding *break* into a longer rule. This incremental processing offers two advantages over a *post hoc* traversal of the graph: first, in an on-line learning situation, predictions and inferences are available immediately for feedback to the user; second, branches like the *case*…*value* one can be captured early and forestall inappropriate rule creation.

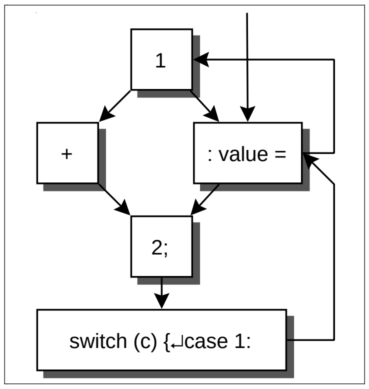
Again, problems arise. First, the approach relies on the order in which examples appear—a deficiency which could perhaps be fixed by further development of the retrospective reparsing technique alluded to above. Second and more fundamental is the question of justifying such inferences. Many apparent relationships arise purely by chance rather than from actual structure in the source, and spurious branches inevitably occur. For example, Figure 6 shows the *case* statement preceded by the assignment *value* = 1 + 2. The pattern 1…*B*…1…*B* in the resulting grammar implies a branch. In this case, however, the parts between 1 and *B* have little in common: they are + and '*: value* = '. If the branch were accepted they would be combined into a new rule, confounding intuition about the sequence's structure. Furthermore, it is now impossible for the first *case* line to take part in a branch.

Justifying inferences by the MDL *principle*

One way to justify one structure over another is to judge the quality of an inference by the length of the inferred structure plus the amount of extra information required to regenerate the original sequence from it. This is called the minimum description length (MDL) principle: short descriptions are preferred to longer ones.

The result is disconcerting. Without going into details of coding methods, it turns out that it is hard to argue that Figure 1d is a more economical representation of the original *switch* statement than Figure 1b. Both involve the same three literal

a
```
value = 1 + 2;
switch (c) {
 case 1: value = 2;
```

c


b
```
S → A1 + Bswitch (c) {
case 1 : AB
A → value =
B → 2;
```

d
```
S → ABswitch (c) { case B
A → value =
B → 1C2;
C → + | : A
```

Figure 6 A spurious branch: (a) (beginning of) the original statement; (b) grammar inferred for (a); (c) automaton formed from the contents of rule *S*.

strings '*switch (c)*', '*case*', and '*: value =* '. Rule *S* of Figure 1b has eleven other constants and eight non-terminals, while the graph of Figure 1d has twelve nodes labeled with constants, each with between one and four exit pointers. To make matters worse, about sixteen bits of additional disambiguation information are required to regenerate the original sequence from Figure 1d.

Moreover, to deal correctly with the dilemma raised by Figures 1 and 6 a test is required for choosing the *case…value* branch over the 1…2 branch. It is hard to imagine how this might be done using description lengths alone.

This situation is counter-intuitive: the case statement *looks* structured, but the branching structure cannot be justified purely on structural grounds. At least part of the problem can be attributed to the rule-formation process. The correspondence between *case* and *value* suggests a useful prediction: all nine characters '*: value =* ' can be predicted by conditioning on the previous *case*. However, these characters are reduced to a single non-terminal symbol because once the symbol '*:*' appears, the rest of this subsequence is entirely predictable. The apparent utility of predictive branches is greatly reduced when simple repetitive structure is recognized.

The fundamental problem is that since an inference can never be completely justified, there is always potential for generalizing the grammar incorrectly by acting on one. Our intuitive reading of Figure 1a is conditioned by prior knowledge of the language concerned. One solution is to use much more data, to ensure that true regularities are more clearly differentiated from coincidental ones, and that is what we do next.

7 A solution for large samples

In this section we adopt the MDL justification, defer generalization until the entire grammar has been formed, and order inferences according to how much information they save. In this way, the most justified inferences are made first. We evaluate the overall success of inference by the amount of compression obtained over the original sample.

The sequence used for testing is an excerpt from a large genealogical database which, for flexibility, is stored textually. The current edition of the database is stored on many dozens of CD-ROMs, providing a strong motivation for an effective compression scheme. The inference and compression scheme is described in full by Nevill-Manning *et al.* (1996); we can only summarize here.

Consider each word to be a distinct symbol. This shortens the sequence, allowing us to consider a very large sample. SEQUITUR infers a grammar from this sequence of words, and rule *S* is transformed into a finite state automaton. The automaton is formed after the entire grammar has been formed, so there are ample statistics on which to base decisions about branch utility. Fortunately, the data is such that SEQUITUR does not concatenate tokens inappropriately, so we can concentrate on how to identify appropriate branches.

Above we expressed branches as a finite state automaton. However, here it is more convenient to implement them by introducing a new non-terminal into the grammar. Each alternative in the branch gives rise to an alternative expansion for the non-terminal, similar to rules C and D in Figure 1e. A new rule can then be formed that concatenates the symbol before the branch, the new non-terminal, and the symbol after the branch, as in rule A of Figure 1e. It is possible to calculate the

```
S → AS
A → 0 B indi↵1afn B↵1 name C↵1sex D↵1husb B↵1wife B↵1chil B ...
B → @26ds-kx@ | @21b7-wr@ | @93gb-dd@...      individual identifiers
C → dan reed olsen | mary ann bernard | ...    names
D → m | f                                     genders
...
```

Figure 7 Structure inferred for the genealogical database.

potential savings of such a transformation, which embodies the fact that the symbol before the branch predicts the symbol after the branch. The branches are ranked, and implemented in rank order, taking care to choose the better branch when two overlap.

Further savings result from merging some of the newly introduced non-terminals when their alternative expansions have a significant intersection. This is the case for certain identification codes in the original data that are used as keys to link records, so each key occurs in at least two different contexts. Merging these symbols requires slightly more disambiguation information to recreate the sequence, but much less storage for the expansions.

The final grammar, appropriately encoded, is nine times smaller than the original sequence, while PPM (Cleary and Witten, 1984) achieves a factor of only six. Part of the inferred structure is shown in Figure 7, and corresponds closely to the formal database format.

8 Conclusions

We have presented two techniques that infer structure from sequences. The first infers deterministic lexical structure, but ignores non-deterministic branching and looping behavior. The second infers non-deterministic structure in the form of an automaton, but overgeneralizes when sequences are expressed at too fine a granularity. The techniques work well together, and can infer plausible structure from sequences that involves both exact, lexical, repetition, and branching structure in the way that tokens are combined.

Unfortunately, the synergy disintegrates in the face of more complex sequences. *Post hoc* application of automaton inference often causes lexical elements to be inferred that cross loop boundaries. Coincidental juxtapositions of unrelated symbols can give rise to rules that stymie later generalization. One possible solution, the integration of branch inference into SEQUITUR, proves unworkable for similar reasons: premature, mistaken inference of a branch can prevent recognition of correct branches or exact repetition later on.

The problem stems from two sources. First, sequences often appear to be more structured than they really are from a purely syntactic point of view. The structure that we prefer intuitively often lacks quantitative justification, and the most concise representation turns out to be the ungeneralized sequence. Second, recognizing repetition and branching greedily, although efficient, inevitably leads to incorrect inferences from which it is hard to recover. We have nevertheless shown that in cases where inference of repetition and branching are separable, and sufficient data exists for statistical justification, appropriate structure can be correctly inferred.

Acknowledgments

Thanks to Dan Olsen for providing the genealogical database excerpt.

References

Cleary, J.G., and Witten, I.H. (1984) "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, COM-32(4), 396–402.

Gaines, B.R. (1976) "Behaviour/structure transformations under uncertainty," *International Journal of Man-Machine Studies*, 8, 337–365.

Nevill-Manning (1996) "*Inferring Sequential Structure*," Doctoral thesis, University of Waikato, Hamilton, New Zealand.

Nevill-Manning, C.G., Witten, I.G., and Olsen, D.R. (1996) "Compressing semi-structured text using hierarchical phrase identification," *Proc. Data Compression Conference*, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press, 53–72.

Wolff, J.G. (1980) "Language acquisition and the discovery of phrase structure," *Language and Speech, 23*(3), 255–269.