Modeling Sequences using Grammars and Automata

Craig G. Nevill-Manning
Department of Computer Science, University of Waikato, Hamilton, New Zealand
Email: cgn@waikato.ac.nz

David Maulsby
Department of Computer Science, University of Calgary, Calgary T2N 1N4, Canada
Email: maulsby@cpsc.ucalgary.ca

An inferred structure can be used to explain a sequence, to predict it, or to state it more concisely. The structure might explain a sequence by providing an insight into the grammar of a natural language or the regularities of a strand of DNA. It might be used to extrapolate a sequence by predicting the actions of a user performing a repetitive task, in order to expedite the process. Finally, since the inferred structure is often smaller than the original sequence, it can might used as a more economical representation of the data.

Laird (1992) suggests that sequence prediction is a fundamental machine learning problem that has been all but overlooked outside the data compression community. He lists several applications: *dynamic program optimization,* which re-orders the text of a program to improve its execution speed, *dynamic buffering algorithms,* where the next most likely cache request is predicted based on previous requests, *human-machine interfaces,* which adapt to specific tasks to speed their completion, *anomaly detection systems,* where unpredictable events are flagged as possible anomalies, and *information theoretic applications* such as data compression, where a stream of data is reduced in size when it contains symbols that are predictable.

This paper presents two sequence modeling techniques. The first induces a context-free deterministic grammar from a sequence. It was motivated by a specific machine learning problem, that of modeling a sequence of actions performed by a computer user, but it can also be applied to other machine learning problems, and its performance as a data compression technique is the best in its class. The second technique induces a push-down finite-state automaton from a sequence. It was designed to derive an executable program from a program execution trace expressed in high-level language statements, and is capable of recognizing branches and loops, as well as recursive and non-recursive procedure calls. The inductive capabilities of these two techniques are complementary, and following their description we examine how they can be combined into a more powerful system.

Modeling sequences using grammars

In the standard grammatical inference problem, several sequences belonging to a language are provided, possibly accompanied by sequences that are not in the language. These are treated as positive and negative examples respectively, and the inference algorithm attempts to determine the unique grammar that generates the language (Angluin and Smith, 1983).

The structural inference problem addressed here is less rigorously defined. The algorithms attempt to find structure in a sequence, but there is only one sequence supplied, and there may be many equally acceptable inferred structures. Furthermore, the sequence of symbols may not have been generated by a context-free grammar, and so even the best induced structure may be an approximation to the source process.

The idea of the grammatical inference technique is simple. Given a sequence of

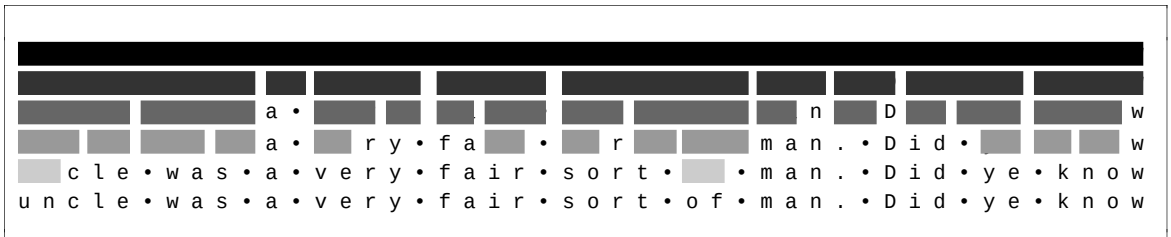

Figure 2: Structure derived from English text

symbols, a grammar is constructed by replacing any repeated sequence of two or more symbols by a non-terminal symbol. For example, given a sequence S ::= a b c d e b c d f, the repeated ‘b c d’ is condensed into a new non-terminal, say A:

S ::= a A e A f
A ::= b c d.

Sequences that are repeated may themselves include non-terminals. For example, suppose S was augmented by appending ‘b c d e b c d f g’. First, the two occurrences of ‘b c d’ would be replaced by A, yielding S ::= a A e A f A e A f g, and then the repeated sequence ‘A e A f’ would be replaced by a new non-terminal, say B:

S ::= a B B g
A ::= b c d
B ::= A e A f.

To implement this method efficiently, processing proceeds from left to right, and greedy parsing is used to select the longest repetition at each stage.

This technique can be used as the basis of a compression scheme, where it is the best in its class, and is only 8% worse than the best lossless compression scheme.

STRUCTURE DISCOVERY

Figure 2 shows the decomposition of part of the grammar induced from Thomas Hardy’s novel *Far from the Madding Crowd*. The darkest bar at the top represents one non-terminal that covers the whole phrase: ‘uncle•was•a•very•fair•sort•of•man.•Did•ye•know’. The existence of this rule indicates that the whole phrase occurs at least twice in the text. Bullets are used to make the spaces explicit.

The top-level rule comprises nine non-terminals, represented by the nine lighter bars on the second line. These correspond to the fragments ‘uncle•was•’, ‘a•’, ‘very•’, ‘fair•’, ‘sort•of•’, ‘man’, ‘.•D’, ‘id•ye’ and ‘•know’. These fragments include two phrases (‘uncle•was•’ and ‘sort•of•’), five words, and two fragments ‘.•D’ and ‘id•ye’. It is interesting that the letter at the start of the phrase ‘Did•ye•know’ is grouped with the preceding period. Although this is different from normal word boundaries in English, without knowing the relationship between lower- and upper-case letters the association between the capital letter and the period is stronger than with the rest of the word.

At the next level, the phrase ‘uncle•was•’ is split into its two constituent words, and ‘id•ye’ is also split on the space. The other words are split into less interesting digrams and trigrams. In other parts of the text prefixes and suffixes can be observed being split from the root, for example *play* and *ing*, *re* and *view*. The technique successfully recognizes structure in a variety of other sequences, including program source code, and graphical descriptions of plants.

Modeling using automata

The second modeling technique attempts to reconstruct a computer program from a high-level trace of the program’s execution. The sequences in this study were provided by C programs, which were modified to print each statement as it was executed. The aim of this system is to infer flow of control, rather than properties of the data involved, so the

a)

```
main()
{
  for (i = N ; i >= 0; i --)
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
          t = a[j - 1];
          a[j - 1] = a[j];
          a[j] = t;
  }
}
```

b)

```
i = N
if (i >= 0)
i >= 0  (t)
j = 1
if (j < i)
j < i  (t)
if (a[j - 1] > a[j])
a[j - 1] > a[j]  (f)
j ++
...
```

c)

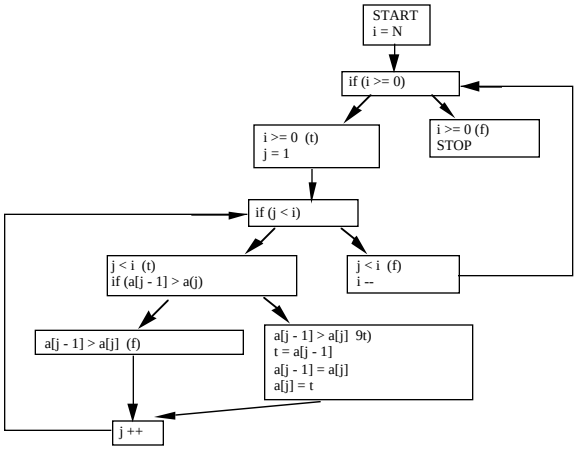

Figure 3 (a) Bubblesort program, (b) part of the trace produced by (a)
(c) finite state automaton produced from the bubblesort trace

trace contains variables rather than values. The only control statements that are included in the action sequences are tests; no explicit branching information is included. As actions are transmitted, each one becomes a state in a finite state automaton. The branches are implicit in the sequence; whenever an action appears that is identical to a previous one, a branch to the matching state is created. When the process has finished, there is one state for each unique action in the trace. This technique is based on Gaines (1976).

For example, when the bubble sort program in Figure 3(a) was executed and traced, it produced a variablized action sequence of which part is shown in Figure 3(b). The sequence was reconstructed into the finite state automaton in Figure 3(c). This automaton can in fact be 'executed' to sort an array.

When this method is applied to a program which includes procedure calls, the resulting automaton is non-deterministic. However, an algorithm has been developed which performs graph transformations to reconstruct the original program, even when it contains recursion.

Current work

Each of these two techniques has particular strengths and weaknesses. The grammar induction technique excels at demonstrating how terminal symbols without internal structure combine to produce higher-level symbols with complex internal structures. However, the slightest variation between two subsequences causes the algorithm to ignore their similarities. Attempts at modifying the technique to tolerate non-determinism in sequences and to induce a non-deterministic grammar have been unsuccessful.

The automaton induction technique, on the other hand, excels at recognizing branching, looping and recursive structures, but is particularly sensitive to the level of abstraction of the symbols that it takes as input. Using the technique to model the sequence of characters in English text would result in a small automaton (with one state for each unique character used) with edges between most pairs of nodes.

Consider the program fragment in Figure 6(a). Is it possible to use these induction techniques to extract a generalized grammar for a C *switch* statement from this string? Individually, the techniques fail to capture the structure of the fragment. However, applying the automaton modeler to the first rule in the grammar results in the automaton of 6(b), which can be generalized to the regular expression below.

(a)

```
{
  switch (getchar()) {
  case 'f':
    value = 1;
  case 'h':
    value = 2;
  case 'e':
    value = 8;
  case 'i':
    value = 7;
  }
switch (getchar()) {
  case '3':
    value = a;
  case '4':
    value = b;
  case '5':
    value = c;
  case '6':
    value = d;
  }
}
```

(b)

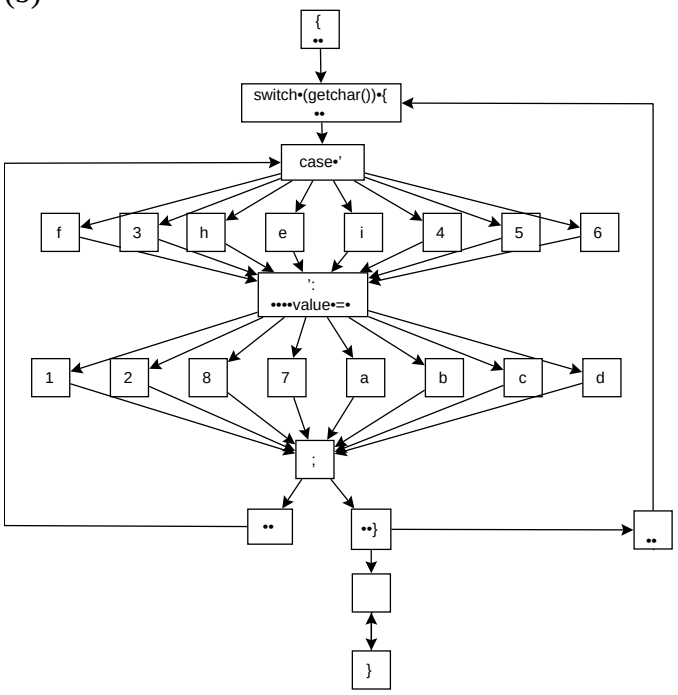

Figure 6 (a) C program fragment
(b) The result of applying both modeling techniques to (a)

```
{
  ( switch (getchar()) {
    ( case '?':
      value = ?; )*
  } )*
},
```

Work is continuing on more difficult sequences. Once the algorithms are mature, we hope to apply them to some of the tasks described in the introduction.

Conclusions

Sequence modeling is an important machine learning problem which has several interesting applications. The grammar induction technique described here provides insights into the ways in which symbols combine to form higher-level symbols, while providing a compression scheme which is the best in its class. The automaton induction scheme provides ways of recognizing branching, looping and procedure calls in a linear sequence of actions. Together, these techniques form a promising method for determining a range of structures in sequences presented at different levels of abstraction.

Acknowledgments

We gratefully acknowledge Tim Bell for helping us to develop our ideas. This work is supported by the New Zealand Foundation for Research, Science and Technology.

References

Angluin, D., Smith, C.H. (1983), "Inductive inference: theory and methods", *Computing Surveys* 15(3), 237-269.

Gaines, B. R. (1976): Behaviour/structure transformations under uncertainty. *International Journal of Man-Machine Studies.* 8(3): 337-365, 1976

Nevill-Manning, C.G. (1993), "Programming by Demonstration", *New Zealand Journal of Computing* 4(2), 15-24.