Modeling Sequences using Grammars and Automata

Craig G. Nevill-Manning
Computer Science, University of Waikato, Hamilton, New Zealand
Telephone +64 7 838–4021; email cgn@waikato.ac.NZ

Ian H. Witten
Computer Science, University of Calgary, Calgary T2N 1N4, Canada
Telephone (403) 220-6015; Email ian@cpsc.UCalgary.CA

David L. Maulsby
Computer Science, University of Calgary, Calgary T2N 1N4, Canada
Telephone (403) 220-7299; Email maulsby@cpsc.UCalgary.CA

1 Introduction

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.

2 Discrete Sequence Prediction

Laird (1992) describes a technique for predicting a stream of symbols, and evaluates it using a compression metric. To predict a symbol, his algorithm looks for past occurrences of the last few symbols, and notes which symbols occurred after those contexts. For example, to predict the symbol following the string *predic*, the algorithm looks at the symbols that have followed *predic* in the past. If this string has not occurred often, then the string *redic* is used as a context, and so on, shortening the context until one is found which has occurred often enough to provide reliable statistics. The symbol is encoded according to how many times it occurs in the chosen context, relative to the total number

of times that the context occurs. Fewer bits are used for more likely symbols; more for improbable ones. The contexts are recorded in a 'successor tree', linking each symbol to the symbols that have followed it in the past, and occurrence counts are stored in each node.

This algorithm is conceptually similar to the *prediction by partial match* (PPM) algorithm developed by Cleary and Witten (1984). PPM, however, can be viewed as blending the predictions of several context lengths rather than setting an arbitrary frequency threshold for deciding to use a particular context, and codes symbols in fractional numbers of bits as dictated by the symbol probabilities (Bell et al. 1990). Results for Laird's algorithm are only given for three files, one of them very small, and his method compresses them 5% better than Unix *compress* on average. PPM, however, when measured over the Calgary compression corpus performs 46% better than compress. These figures indicate that the predictive ability of PPM is significantly greater than Laird's algorithm.

While both these techniques concentrate on prediction, their models take the form of graphs containing occurrence counts, which explain little of the structure of the sequence. The structure of a sequence is almost obfuscated by the short-range statistical observations that these modeling systems make. Furthermore, the models are generally much larger than the sequence itself; one would expect a good explanation of a structured sequence to be smaller than the original! The technique described in the next section provides plausible explanations of the structure of a sequence, but at the same time compresses sequence well, implying good predictive ability.

3 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 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.

Because every repeated sequence is replaced by a non-terminal, grammars produced by this technique satisfy the constraint that *every digram in the grammar is unique*. To implement this method efficiently, processing proceeds from left to right, and greedy parsing is used to select the longest repetition at each stage.

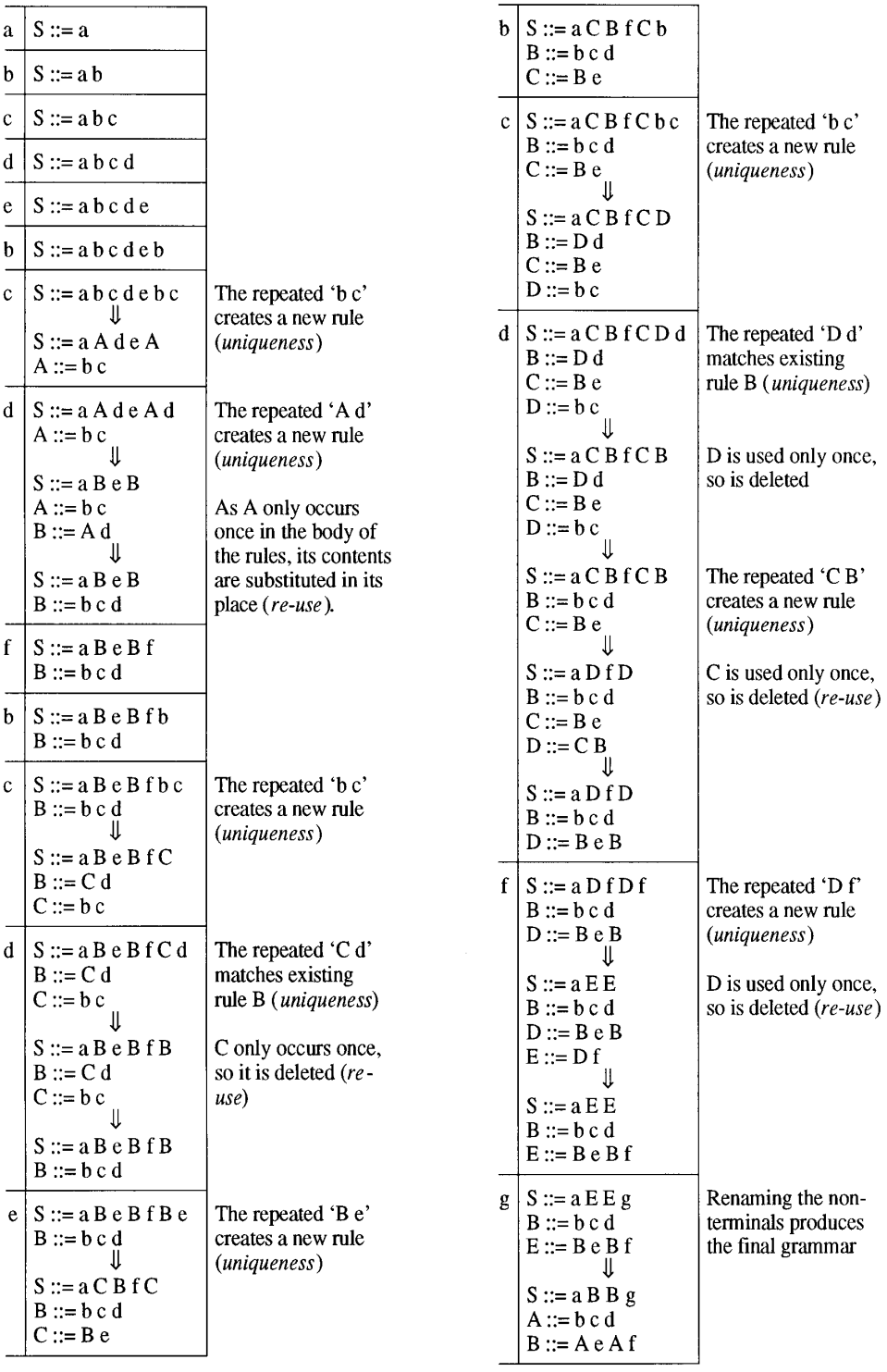


Figure 1: Execution of the induction algorithm

Although this technique for grammar induction is simple and natural in concept, it is—surprisingly—not obvious how to implement it efficiently. The problem is that at any one time there may be several partially matched rules at different levels in the hierarchy. Suppose that the sequence in the example above continues 'b c d e b c'. At this point rules A and B are both partially matched in anticipation that 'd' will occur next. However, if the next symbol is not 'd' both partial matches have to be rolled back and two new rules created, one for the sequence 'b c' and the other for 'A e'. The algorithm must keep track of a potentially unbounded number of partial matches, and when an unexpected symbol occurs it must roll them back and create new rules. To decide whether the next symbol continues the partial matches or terminates them, it is necessary to consider all expansions of the matching rules at all levels. Furthermore, at any particular point in time, the grammar may not satisfy the 'digram uniqueness' constraint, as there are repeated digrams pending the match of a complete rule.

The solution to these problems is to view the modelling process as the enforcement of two constraints: that every digram is unique (*uniqueness*), and that every non-terminal appears multiple times, that is, at least twice (*re-use*). The first constraint is satisfied by creating new rules or matching existing ones, the second by deleting rules. The algorithm works in the following way.

As soon as a new symbol is appended to the first rule (S), it and the preceding symbol constitute a new digram. If the new digram is unique, no further action is necessary: the uniqueness constraint is maintained. If the digram matches the body of an existing rule that contains only two symbols, then the non-terminal that heads the rule is substituted for the digram. Otherwise, if the digram matches another digram anywhere in the grammar, then a new rule is formed. The new rule has the digram as its body, and the non-terminal that heads it replaces the matching digrams.

Any substitution reduces by one the number of times that each symbol of the digram appears in the grammar. If either of the symbols is a non-terminal and now appears only once in the body of the rules, then it is superfluous, and its contents are substituted in its place.

Each substitution creates a digram to the left and right of the substituted symbol, and if these digrams appear elsewhere, the transformations are reapplied, and so on until the constraints are satisfied. To illustrate the algorithm, the earlier example is reworked in Figure 1. Whenever a new digram matches an existing one, the action is explained on the right in terms of the two constraints.

3.1 EXAMPLES OF STRUCTURE DISCOVERY

To illustrate the structures that can be discovered using this technique, Figure 2 shows a sample from two grammars that are inferred from different sequences. In each case the rules are expanded to demonstrate how symbols combine to form higher level rules.

3.1.1 English Text

Figure 2(a) represents 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

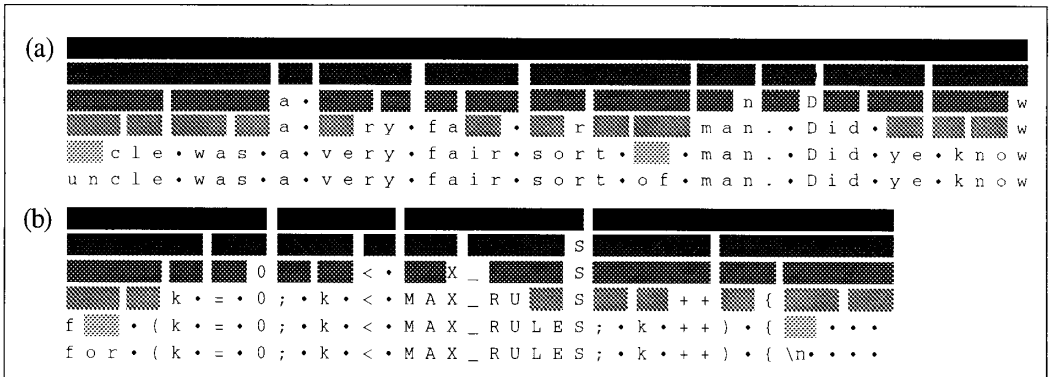


Figure 2: Structure derived from (a) English text and (b) C source code.

between the capital letter and the period is stronger than with the rest of the word. If the representation of the text had included capitalization as a separate 'upper-case' marker that prefixed capital letters then it would be quite correct to associate it with a preceding period rather than with the letters that followed.[1]

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 phrase, 'sort•of•', is split after 'sor', which indicates that the word *sort* has not previously been seen in any other context. 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*.

3.1.2 C Source Code

Figure 2(b) shows the structure of a model formed from a C source file—in fact, in the true spirit of recursive presentation, the file is part of the source code for the actual modeling program. The top level shows four rules which expand to

for•(k•=•0 ;•k•<• MAX_RULES ;•k•++)•{\n••••.

The first and last fragments correspond to the beginning and end of a C 'for' statement with k as the counter variable. The middle two fragments identify a less-than comparison involving k, and a constant MAX_RULES.

At the next level, 'for•(k•=•0' is split into 'for•(k•' and '=•0', indicating that something other than an assignment has followed 'for•(k•' in the past. The phrase ';•k•<•' is split into the variable and the operator ';•k•' and '<•', indicating that other comparisons have been made on k.

The third line separates 'for•(', the standard beginning of a C 'for' statement, from 'k•', the specification of the counter variable. The assignment operator '=•' is separated from the value '0'; the operator '++' is separated from ';•k•' and ')•{' is separated from the following white space '\n•••'.

Overall, the divisions make structural sense in terms of the syntax of C, combining terminal symbols into groups on boundaries that coincide with meaningful blocks within the language.

---

[1] An experiment which involved putting the whole text in lower case and inserting a special character before each capital letter resulted in a 0.5% compression improvement. Of course, the text could still be recreated exactly.

3.2 DATA COMPRESSION PERFORMANCE

The structure that is inferred in this way can be used to compress the original data, performing 35% better than Unix *compress* and only 8.4% worse than PPM on the Calgary corpus, indicating that its predictive power is still high, as well as the extra explanatory power that it offers over the other two schemes. Descriptions of the actual coding algorithms and results are given in Nevill-Manning *et al.* (1994).

4 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 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.[2] 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). This sequence consists of one C expression per line, with a line following each test to indicate whether or not it was successful (*t* and *f* for true and false respectively). The sequence was reconstructed into the finite state automaton in Figure 3(c). Sequences of states with no branches have been combined into one state, and START and STOP states are labeled. 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. Consider the bubble sort program in Figure 4(a), where the array which is being sorted is printed during the *swap* operation. The corresponding automaton in Figure 4(b) has two non-test states (shaded) which have more than one outgoing transition.

When a procedure returns, the next state is determined not by a test on variables, but by the contents of the top of the program stack. It is therefore necessary to add a state push and pop operation to some of the induced branches, corresponding to a procedure call and return respectively.

The automaton can be rearranged to form a correct, deterministic automaton in the following way:

- Find a non-deterministic state and mark it as a procedure return;
- Find the start of the procedure;
- Remove the subgraph beginning at the start of the procedure and ending at the procedure return and label it as a procedure;
- Pair the states before the start of the procedure with the states following the return by examining the original trace;
- Connect each pair by inserting a procedure call between them.

---

[2] The inclusion of the tests is necessary for the program reconstruction technique described below, but are unlikely to be explicit in a trace of, say, a user performing the task. The induction of tests, as with the induction of variables, is a difficult problem which is ignored here. Furthermore, procedure parameters are ignored, as is the scoping of variables, which involve further work after the procedures have been identified.

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 ++
if (j < i)
j < i  (t)
if (a[j - 1] > a[j])
a[j - 1] > a[j]  (t)
t = a[j - 1];
a[j - 1] = a[j];
a[j] = t;
j ++
if (j < i)
j < i  (t)
if (a[j - 1] > a[j])
a[j - 1] > a[j]  (t)
```

c)

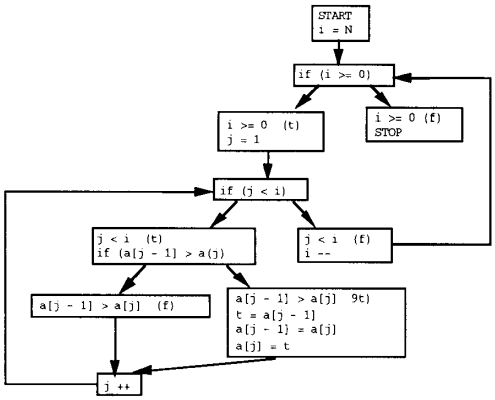


Figure 3: (a) bubblesort program (b) part of the trace produced by (a)
(c) finite state automaton produced from the bubblesort trace

When all procedures have been identified, no non-deterministic states will remain, and the set of graphs forms an executable program. Figure 4(c) shows the final form of the automaton derived from the trace of the program in Figure 4(a).

There are two problems with this process: the difficulty in finding the beginning of a procedure, and the effects of recursion. A discussion of these two problems follows.

4.1 IDENTIFYING THE START OF A PROCEDURE

In the automaton, a procedure appears as a group of states which can all be reached from one 'source' state (the first expression in the procedure), and from which a unique 'sink' state (which is non-deterministic) is reachable. Furthermore, this group of states should only be connected to the rest of the graph through the source and sink sates. That is, a procedure is a set of nodes which have a common ancestor, and a common descendant which is followed by a return. The source node may only be connected to the rest of the graph by incoming edges, and the sink only by outgoing edges.

Unfortunately, the automaton is cyclic, which means that states can be their own 'ancestors', making procedure identification difficult, if not impossible. The solution to this problem is to identify loops when the automaton is created, making use of the order in which states are created. If an edge is added which connects a state to one of its ancestors, this edge is marked as a loop. Ignoring the loops means that the graph is acyclic, and the procedure definition above becomes useful.

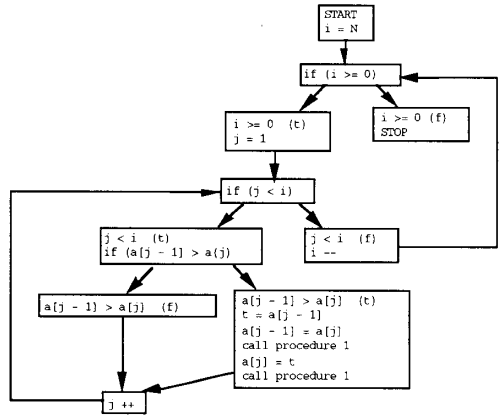
a)

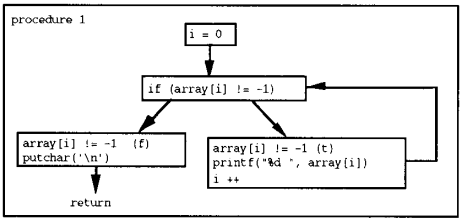
```
main()
{
  for (i = N; i >= 0; i --) {
    for (j = 1; j < i; j ++)
      if (a[j - 1] > a[j]) {
    t = a[j - 1];
    a[j - 1] = a[j];
    print_array(a);
    a[j] = t;
      print_array(a);
      }
  }
}

print_array(array)
int array[];
{
  int i;

  for (i = 0; array[i] != -1; i ++)
    printf("%d ", array[i]);
  putchar('\n');
}
```

c)





b)

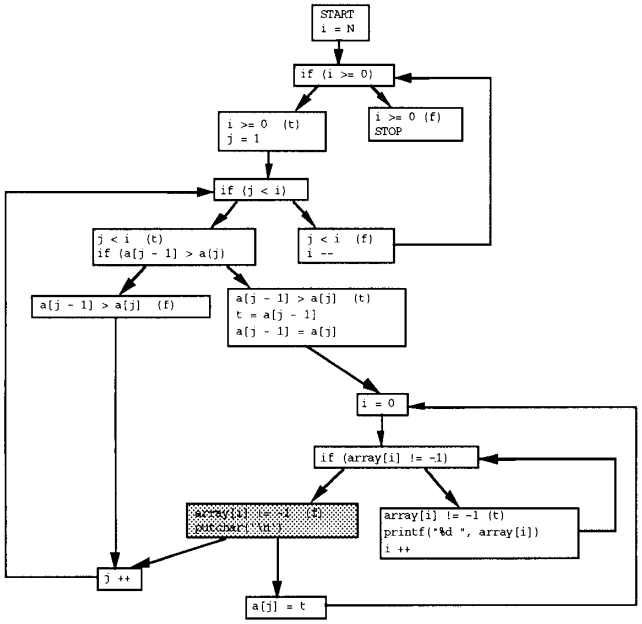


Figure 4: (a) bubblesort program with procedure calls, (b) automaton derived from trace of (a), (c) the effect of extracting a procedure from (b)

a)

```
main()
{
  recurse(10);
}

int recurse(n)
int n;
{
  puts("enter recursion");

  if (n == 0) {
    puts("no more recursion");
    return;
  }
  else {
    puts("before recursion");
    recurse(n - 1);
    puts("between recursion");
    recurse(n - 1);
    puts("after recursion");
    return;
  }
}
```

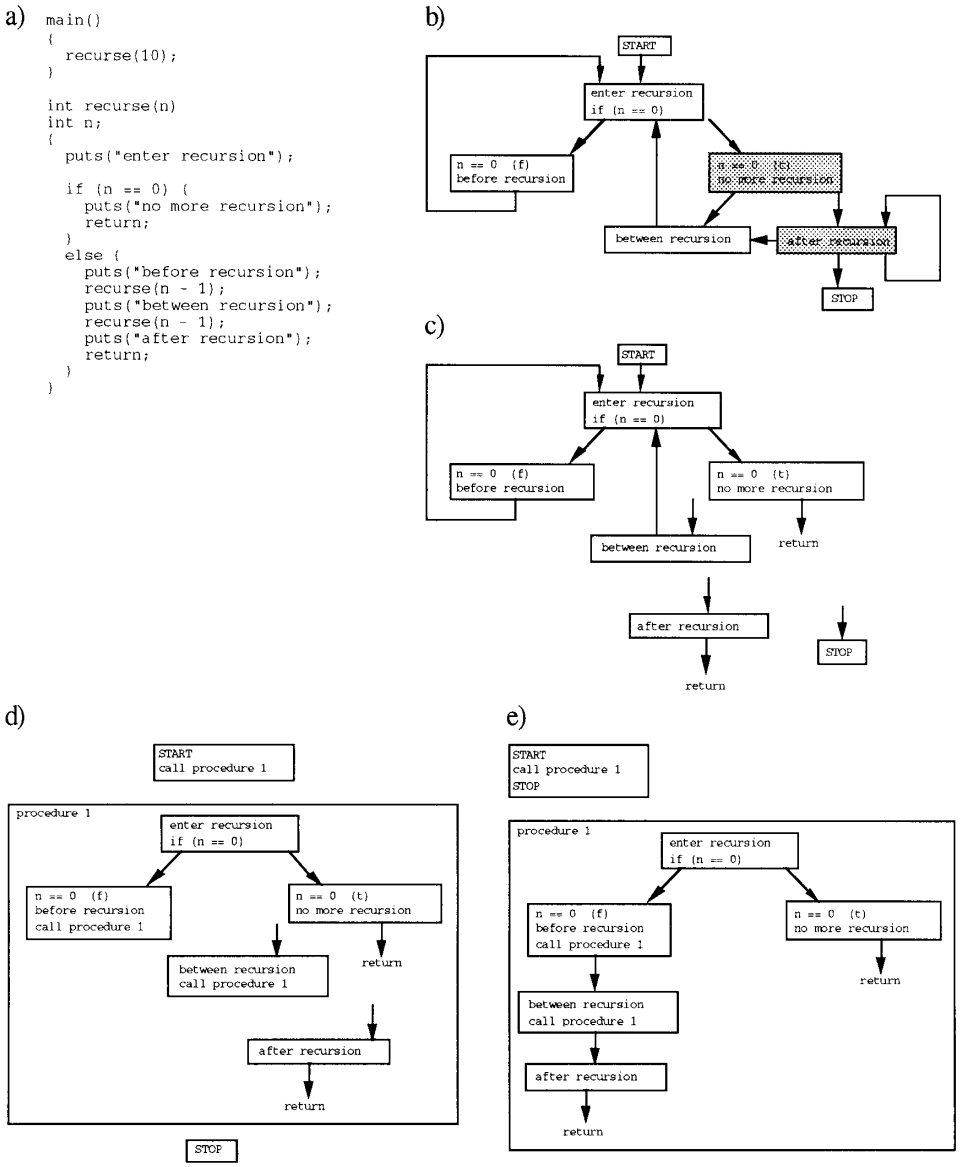


Figure 5: (a) prototypical doubly-recursive program, (b) automaton produced from trace of (a), (c) the effect of treating non-determinism as function returns in (b), (d) part (c) with the procedure identified, (e) part (d) with states before and after calls matched up.

The algorithm to find procedures in the graph is therefore as follows:

- Visit every node in the tree, using a depth-first search. At each node, label the node a candidate source node and:
  - Visit every descendant of the candidate until a return (a sink node) is reached, and record all nodes visited
  - If there is only one sink node, then the group of nodes forms a procedure
  - Remove the procedure from the graph as described above.

This algorithm finds the start of a procedure, provided that the action sequence is complete (see below). This algorithm correctly identifies the procedure in Figure 4(b).

4.2 THE EFFECT OF RECURSION

To study the effect of recursion, consider the prototypical doubly recursive program in Figure 5(a). No more states can be combined in this automaton produced from this program due to the non-deterministic transitions (Figure 5(b)). We begin by identifying the returns, which occur after the gray states. Replacing the transitions with returns leaves three states with no incoming transitions: '`between recursion`', '`after recursion`' and '`STOP`'. These states are therefore identified as states following procedure calls (Figure 5(c)). The beginning of the procedure can then be identified as the source state of a group of states which lead only to other states in the group. The procedure is isolated as above, and procedure calls are inserted in place of transitions to the start state (Figure 5(d)). Finally, by re-examining the trace, the order of the calls and the states following calls can be determined, and the automaton can be reconstructed (Figure 5(e)). This algorithm is discussed more fully in Nevill-Manning (1993).

5 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 overlook 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 connections between most pairs of nodes.

Consider the program fragment in Figure 6(a). Is it possible to use these induction techniques to extract a generalised grammar for a C *switch* statement from this string? Applying the grammatical modeler to this sequence produces the grammar in Figure 6(b). Several useful structural parts have been identified, such as the start of the switch statement in rule J and the case label in rule E. It is not, however, a particularly concise or useful description of a switch statement. The result of applying the automaton modeler is shown in Figure 6(c). It fails to explain much of the structure, apart from the high frequency of the space character (represented by a bullet), and the row of some of the symbols that follow a single quote.

If the body of rule S from the grammar is supplied to the automaton modeler as a sequence, the result is as in Figure 6(d) (non-terminal symbols have been expanded to show their content). The values assigned to the variable 'value' are in one row (1, 2, 8, 7, a, b, c, d), and the case labels are split over two rows (h, e, l, 4, 5, 6 and f, 3). Due to the greedy left-to-right parsing, rule J includes rule E, the *case* keyword. We would prefer rule E to stand on its own, to bring the case labels together. As a post-processing step, we look at all the 2-predecessors and 2-successors of each node. If they share a common suffix or prefix respectively, we remove the affix from the nodes, and form a new node from it. In this case, both 2-predecessors of the node for rule F (':\n fred =) have a suffix of E, so a new node is created with the contents of E. The resulting graph (Figure 6(e)) describes the syntax of the fragment quite well. The same technique is used for adjacent nodes, but this situation does not arise in this example. Expressing the automaton as a regular expression gives:

```
{
  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;
  }
}
```

Figure 6(a): C program fragment

Grammar	Expansion of rules
S ::= B J f F 1 G h F 2 G e F 8 G i F 7 K C C x I C J 3 F a G 4 F b G 5 F c G 6 F d K \n } \n \n \n	The whole text
A ::= t c h	tch
B ::= { C • •	{\n••
C ::= \n D	\n••
D ::= • •	••
E ::= c a s e • '	case•'
F ::= ' : I f r e d • = •	':\n••••fred•=•
G ::= H E	;\n••case '
H ::= ; C	;\n••
I ::= C D	\n••••
J ::= s w i A • (g e A a r ( )) • B E	switch•(getchar())•{\n••case•'
K ::= H }	;\n••}

Figure 6(b): Grammar induced from sequence in 6(a)

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

and generalising the set of case labels and constants to ? gives

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

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

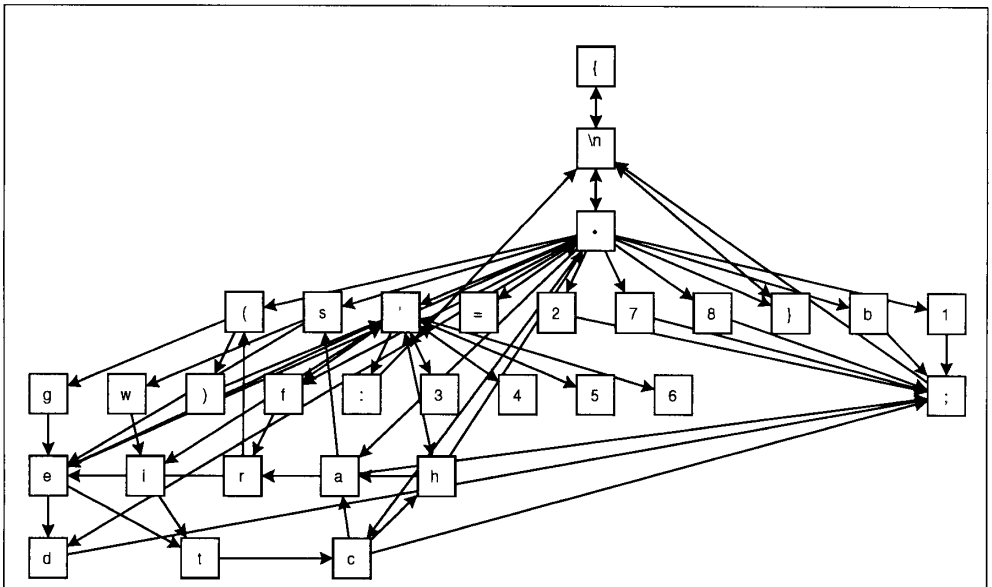


Figure 6(c): Automaton induced from fragment in Figure 6(a).

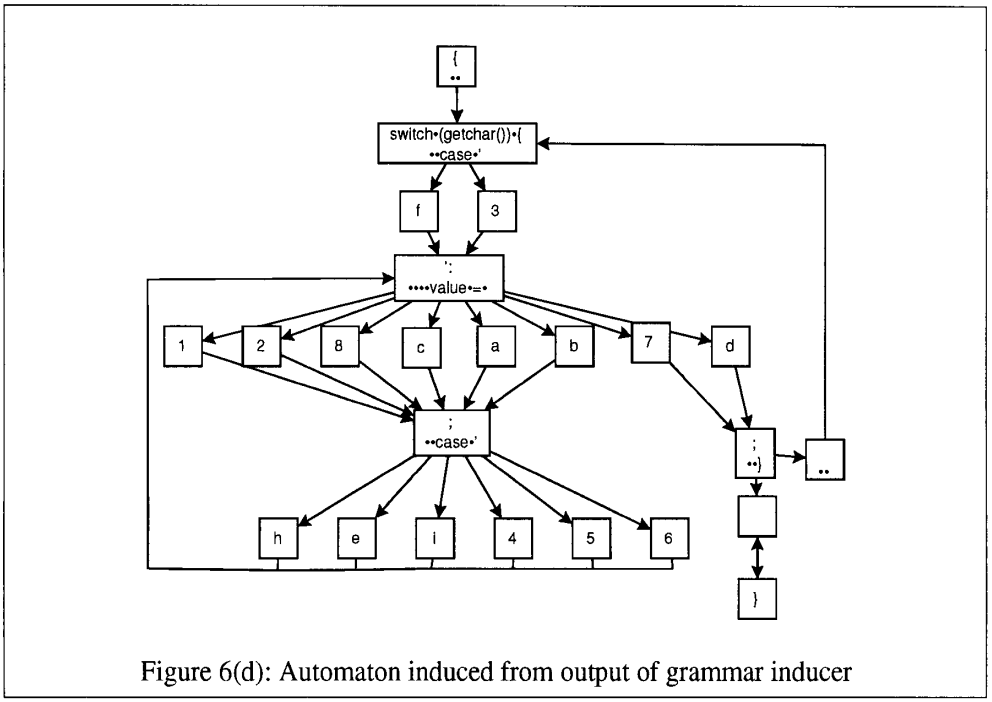


Figure 6(d): Automaton induced from output of grammar inducer

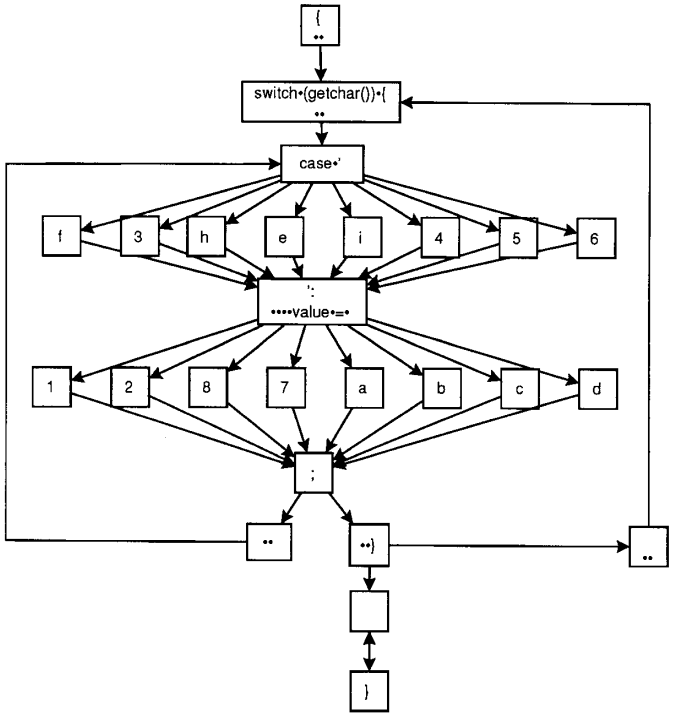


Figure 6(e): Figure 6(d) rearranged to reverse the effects of greedy parsing

6 Conclusion

Sequence modeling is an important machine learning problem which has several interesting applications. Much work has been done on this problem for the purposes of data compression, but these models generally emphasize prediction over explanation. The grammar induction technique described here balances prediction and explanation, providing 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.

7 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.

8 References

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

Bell, T.C., Cleary, J. G., Witten, I.H. (1990) *Text Compression*, Prentice Hall, Englewood Cliffs, NJ.

Cleary, J.G., 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(3): 337-365, 1976

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

Nevill-Manning, C. G., Witten, I.H., Maulsby, D.L., (1994) "Compression by Induction of Hierarchical Grammars" *Proceedings of the Data Compression Conference 1994*, IEEE Computer Society Press.