Learning from experience

Craig G. Nevill-Manning,
Department of Computer Science, University of Waikato, New Zealand.

Life is a sequence of events; some of them predictable, some unexpected. We instinctively look for structure in the stream of events we observe, and learning to anticipate these events is an important part of functioning successfully in the world. Machine learning emphasises learning from unordered facts, but for machines to be helpful in our day-to-day activities, they also need to be able to recognise patterns in the sequences they observe.

The sequence of events that constitute our lives is often monotonous; we end up repeating certain sequences of actions again and again. We have built machines to relieve some of this monotony, but they often fail to be flexible enough to free us of all but the most common repetitive tasks.

SEQUITUR is a system which infers structure from a sequence, and uses the structure to explain and extrapolate the sequence. It is capable of recognising structure in a variety of real-world sequences, but is of particular utility in automating repetitive computing tasks.

Introduction

The world is full of sequences which are structured and predictable. We instinctively look for this structure in an attempt to make sense of our environment, and to allow us to plan ahead.

Not only do we observe sequences, but we also *act* in predictable ways and perform tasks which are repetitious. Computers were intended to free us from the tyranny of repetition, yet we often find ourselves performing tedious, repetitive tasks ourselves. Even programmers have difficulty automating small tasks, as the overhead of writing and debugging a program often outweighs its benefits for a one-off job.

Programming by demonstration (PBD) automates repetitive tasks by generalising a demonstration of the task to form a program. An essential function of a PBD system is to recognise structure in a sequence of events and to create instructions to extrapolate the sequence.

SEQUITUR is a system which observes a sequence, recognises its structure, and uses the structure to explain and predict the sequence. SEQUITUR first finds recurring sub-sequences, and forms a hierarchical grammar to describe the 'vocabulary' of the sequence. Next it looks for branches, loops, equivalent symbols and 'procedures'. The model that SEQUITUR induces can be represented as a grammar or a transition network, and is driven by Occam's exhortation to find a simple explanation for observations.

This paper describes SEQUITUR's algorithm and provides examples of its application to several diverse sequences.

Vocabulary

Finding phrases

The simplest kind of sequential structure is the recurrence of a sub-sequence of symbols. In natural language, these sequences are words, phrases, roots and affixes. In programming by demonstration, they are sub-tasks made up of atomic actions. In program code, they are reserved words, variables and function names. The first task that SEQUITUR performs is to identify these phrases, show how small phrases combine to form larger phrases, and restate the sequence in terms of this new vocabulary.

Sequence	Grammar
a **b c** d **b c**	S ← a A d A A ← b c
a b c d b c e **a b c d b c**	S ← B e B A ← b c B ← a A d A

Figure 1: two simple sequences and the phrases extracted from them

The first sequence in Figure 1 shows how a repeated subsequence is replaced by a grammar production, and the second sequence shows how this is performed hierarchically. The algorithm consists of enforcing two constraints: every digram in the grammar must be unique, and every rule must be used more than once. To satisfy the first constraint, a new rule is formed, and to satisfy the second constraint, the redundant rule is removed. The algorithm operates incrementally, so that the grammar obeys both constraints after the processing of each character. An implementation on a SparcStation 10 processes sequences at a megabyte per minute.

Figure 2 shows the hierarchies from portions of some longer sequences. Each of the bars represents a rule; the contents of the rule is expanded beneath it.

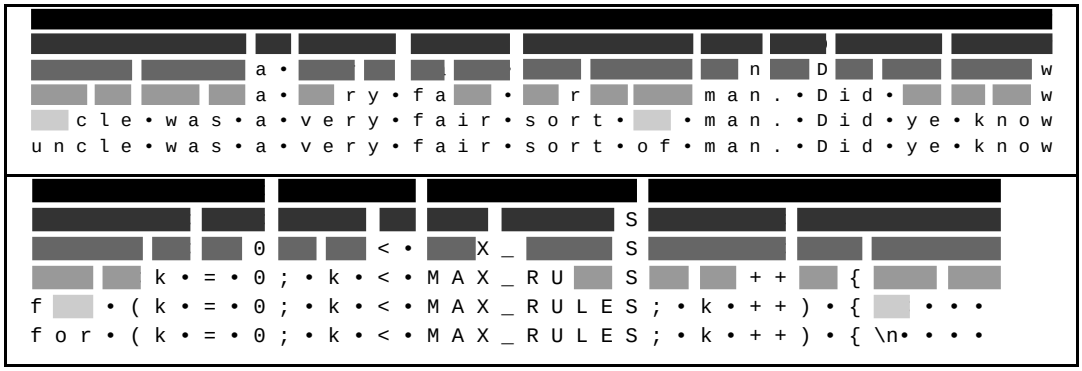


Figure 2: (a) a phrase from ‘Far from the Madding Crowd’ by Thomas Hardy
(b) excerpt from a C program

This algorithm captures much of the structure of the vocabulary a sequence; how the individual symbols combine to make meaningful blocks. In figure 2(a) it captures roots, affixes, words and phrases. In figure 2(b), it captures the reserved word ‘for’, delimiters and the basic structure of the for statement; it breaks the sequence up around the semicolons, and separates the variable I from the rest of the assignment statement.

Generosity

Because the algorithm operates on-line, it must form phrases greedily; it always extends a rule if it will reduce the size of the sequence so far. Some of these decisions are harmful in the long run. Figure 3 shows the same sequence processed left-to-right and right-to-left. The second grammar is smaller than the first, due to the presentation order of the sequence. It has been shown (Storer, 1988) that finding the best set of phrases to describe a sequence is NP-hard, but a simple heuristic can often improve the grammar. For example, in figure 3(b), the symbols that precede *C* both have *c* as a suffix, so *C* can be extended leftwards, which results in the same grammar as figure 3(c) (as B becomes superfluous, and is deleted).

(a) bcd bcd bcef acef	(b) S ← A A B C a c C A ← B d B ← b c C ← e f	(c) S ← B B b A a A A ← c e f B ← b c d

Figure 3: the sequence in (a) processed (b) left to right and (c) right to left

Generous parsing is most effective on well-structured sequences. Consider the grammar in figure 5 (a). When the grammar is expanded to a recursion depth of three, it produces a sequence of which part is shown in figure 5(b). The grammar formed from this sequence is shown in 5(c), and 5(d) shows the grammar that is formed when generous parsing is used. The generous grammar is much more compact, and resembles the original grammar much more than the first grammar (especially when rules A, C, E, F and G are removed, as in 5(e)). This grammar is an example of an l-system, and is discussed further in the section on *recursion.*

(a) S ← F[+F]F[-F]F F ← f	(b) f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f[+f]f[-f]f[+f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f[+f]f[-f]f]f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f[+f]f[-f]f]f[+f]f[-f]f[+f[+f]f[-f]f]f[+f]f[-f]f[-f[+f]f[-f]f]f]f[+f]f[-f]f
(c) S ← C E I K J K D f A ← f [B ← D A C ← H F D ← f] E ← B + F ← B - G ← C D H ← A + I ← G N J ← E F K ← E G L ← F G M ← H I D J K N N ← J L J	(d) S ← B F A G A A ← B] B B ← D F C G C C ← D] D D ← f F E G E E ← f] f F ← [+ G ← [- (e) S ← B [+ B] B [- B] B B ← D [+ D] D [- D] D D ← f [+ f] f [- f] f

Figure 5: (a) a recursive grammar, (b) part of the sequence produced by (a), (c) the grammar induced from (b), (d) the grammar induced with generous parsing, (e): (d) with rules B, C, E, F and G expanded

Compression

Occam's razor suggests that theories shouldn't be more complicated than they need to be, and restating the sequence more concisely is SEQUITUR'S justification for making these transformations. The textual representation of the grammar is somewhat smaller than the original sequence, but the compression achieved is much

less than any purpose-built data compression utilities. The sequence does not need to be sent as text, but can be encoded using a frequency model and an arithmetic coder, which encodes symbols according to their frequency: more frequent symbols are shorter, less frequent symbols are longer. This improves compression to about the same as the UNIX COMPRESS utility. However, there are many more sophisticated compression techniques which perform better than COMPRESS, and a new encoding scheme is required if this scheme is to outperform them.

In the grammar representation, we send the model (all the rules except S) separately from the sequence given the model (rule S). It is possible to improve compression by sending the model and the sequence simultaneously; that is, having the decoder build the model adaptively. The sequence is sent in the following way: the first time a rule is used, its contents are transmitted. The second time it is used, it is sufficient to transmit a reference to the first occurrence. At this point, the decoder forms a new rule, and it can thereafter be referenced with the appropriate non-terminal symbol. This scheme outperforms all other dictionary compression schemes, which account for all commonly used compression utilities, at the expense of longer computation time. (Nevill-Manning, et al., 1994).

The minimum description length (MDL) principle is a formalisation of Occam's razor which states that in learning, we should choose the theory which allows the observations to be encoded in the smallest number of bits. MDL assumes that the language in which the theory and the data are represented is as efficient as possible. This example shows that finding an efficient representation is not a straightforward task. The textual representation implies that the structure that the algorithm finds is much worse than the structure found by the simplest compression schemes. The probability-based encoding shows that it is equal to COMPRESS, but the adaptive scheme shows it to be better than the theories found by any other dictionary compression scheme. The implication is two-fold: finding an efficient representation for the purpose of applying MDL is difficult, but adaptive transmission of the model along with the sequence may be a useful approach in finding efficient representations.

Grammar

The hierarchical decomposition produced by the algorithm described so far has three shortcomings as a description of the structure of the sequence:

- It describes the vocabulary of the sequence, but doesn't capture any non-linear structure, like loops or branches.
- It is expressed as a grammar, but can only produce one sentence; the original sequence.
- It has limited predictive power. While it can predict the completion of a partially matched rule, it cannot predict the sequence of symbols in the first rule.

We would now like to generalise the grammar to make it more descriptive, more productive, and more predictive. For the discussion below, it is helpful to represent the sequence as a transition network by creating a state for each unique symbol in the first rule and inserting transitions between states whose symbols are adjacent in the sequence (Figure 6). This network can be traversed to reproduce the original sequence, but can also produce many other sequences. The transition network is too general, as all context information is forgotten when a transition is made, and considerable extra information is required to reproduce the original sequence. The true structure of the sequence is likely to be a compromise between the grammar and the transition network; a compromise which minimises the size of the structure and the extra information required to recreate the sequence given the structure.

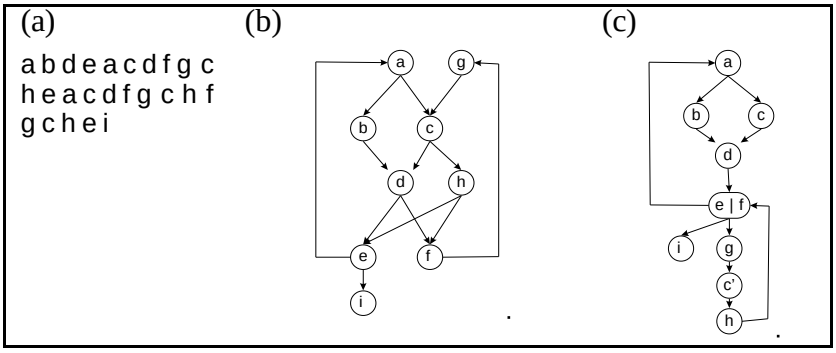


Figure 6: (a) a simple sequence, (b) the transition diagram representing the sequence, (c) the transition diagram after recognition of a branch and equivalent symbols.

The goal of the generalisation is not to capture every conceivable structure that may be present in an arbitrary sequence, but instead to recognise certain structures that are likely to be produced by the particular source process. For the purposes of programming by demonstration, we assume the source to be a program, and the likely structures are loops, conditionals and procedure calls (including recursion). For grammar-based sequences, we also look for subsequences which occur in the same contexts. These five structures are discussed in turn:

Equivalent symbols

If two symbols often occur in the same contexts, the grammar can generalised and simplified by treating them as equivalent symbols. That is, if the set of the

predecessors of one symbol is similar to the set of the predecessors of the other, then the symbols are equivalent. In figure 6(b), nodes e and f are both preceded by nodes d and h This is similar to k-reversibility in grammatical inference (Berwick et al., 1987). We would expect to find such symbols when they were alternative expansions for the same non-terminal in a non-deterministic grammar, e.g. they are both verbs in a sequence of English text.

When such a pair of symbols are found, a new rule is created with two right hand sides— one for each of the equivalent symbols—and the non-terminal that heads the new rule is substituted where the two original symbols occurred. In figure 1, X and Y both occur following A and B, so X and Y are generalised. The size of the grammar has decreased from 16 symbols to 14 symbols.

S ←	S ← ...A...B...A...B...
...A X...B Y...A Y...B X...	A ← a a Z
A ← a a	B ← b b Z
B ← b b	Z ← x x
X ← x x	Z ← y y
Y ← y y	

Figure 7: merging equivalent symbols X and Y

To reproduce the original sequence, it is necessary to supply extra information to select the correct right hand side for Z. In this case, one bit has to be supplied each time A or B is used, so four bits should be added to the size of the grammar.

Branches

Nodes a, b, c and d in figure 6(b) typify a branch structure. This structure could be produced by an *if...then...else* or *case* construct in a program. We cannot say that nodes b and c are equivalent, as there is only one context, but node a can be used as a context to predict node d. If the alternate nodes have any other transitions in or out, the node must be cloned, so that the context information is retained. The transition diagram in figure 6(a) allows a transition from g through c to h, as well as from a through c to d. To rectify this, node c is cloned, resulting in figure 6(b). This means that when we are leaving node c, we have definitely come from node a, so can predict node d.

Loops

Borrowing from structured programming constraints, we require that loops do not overlap. This means that any candidate loop must not include a node with an edge to a node outside the loop (except for the first and last nodes in the loop).

Procedure calls

A procedure is a sub-sequence which is repeated in the sequence, but where the repetitions are not contiguous. If the procedure is repeated verbatim, a rule is formed for it, but if there are branches within it, we must recognise it in some other way. It is possible to recognise procedure calls in a sequence by performing a search for a group of nodes where all paths through the group begin at a particular start node, and end in a particular stop node. That is, there can be no transition to any node in the group from nodes outside the group except via the start node, and no transitions from any node in the group to nodes outside the group except via the

stop node. A group of this form can be found by an $O(n^2)$ search of the network. This is described in more depth in Nevill-Manning (1990).

Recursion

The l-system in figure 5(a) draws a Koch curve if the terminal symbols are interpreted as turtle commands: f = draw a line forward, - = turn left, + = turn right, [= save state,] = restore state (Prusinkiewicz, 1990). The grammar obtained using generous parsing in figures 5 (d) and (e) is a non-recursive version of the original l-system, and the original can be reproduced by performing Prolog unification between each of the rules.

Putting it all together

This paper has so far described techniques for efficiently forming a vocabulary from a sequence, and recognising branches, loops, procedure calls, recursion, and equivalent symbols in a sequence. This section describes how these techniques fit together to make inferences.

The system has been designed with two principles in mind: (a) it should perform transformations on the fly, and allow them to be undone by later transformations with the benefit of hindsight, and (b) it should avoid ad-hoc thresholds for deciding whether transformations will be performed. This provides a simple system which is not optimised for a small set of sequences, but to which domain-dependent heuristics can be added. As each new symbol is observed, the vocabulary is updated, generous parsing is performed, and then the grammar is examined to see whether the it is possible to apply any of the transformations. As only parts related to the last symbol in the sequence will have changed, this check can be performed efficiently.

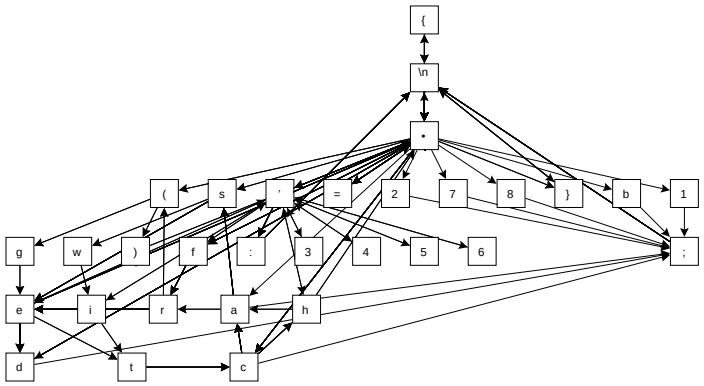
Figure 8(a) shows a portion of a C program. Figure 8(b) shows the structure recognition techniques without the vocabulary formation part. As it must find structure between the individual characters, it is not able to give particularly useful insights. The vocabulary derived from this sequence is shown in figure 8(c). It captures the reserved words, variable names and delimiters, but without the structure recognition techniques, it cannot recognise the overall structure because of the different variable names switch values, case labels and values. However, when the two parts are combined, it produces the structure in figure 8(d), which captures much of the desired structure. Work is continuing on refining and applying these techniques to more complicated sequences.

(a)

```
{
   switch (getchar()) {
   case 'f':value = 1;
   case 'h':mark  = 2;
   case 'e':mark  = 8;
   case 'i':value = 7;
   }

  switch (n) {
   case 4 : mark  = 45;
   case 30: value = 6;
   case 3 : value = 38;
   case 5 : mark  = 3;
   }
}
```

(b)



(c)

```
S  ←   B M g e A G ( ) Q E
f K 1 I h J 2 I e J 8 I i K
7 X N M n Q O 4 W 4 5 T 0 R
S 6 T U S 3 8 V 5 W 3 X Y
A  ←   t c h
B  ←   { D
C  ←   e ~
D  ←   N ~
E  ←   O '
F  ←   ' :
G  ←   a r
H  ←   = ~
I  ←   L E
J  ←   G P
K  ←   G S
L  ←   ; D
M  ←   s w i A ~ (
N  ←   \n ~
O  ←   c a s C
P  ←   m G k ~ ~ H
Q  ←   ) ~ B
R  ←   : ~
S  ←   v a l u C H
T  ←   V 3
U  ←   ~ R
V  ←   L O
W  ←   U P
X  ←   L Y
Y  ←   } \n
```

(d)

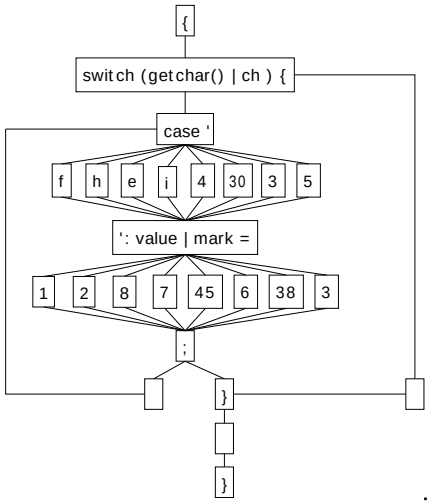


Figure 8: (a) fragment of C code, (b) structure recognition without vocabulary forming (c) the vocabulary, (d) vocabulary and structure recognition

Conclusion

SEQUITUR successfully recognises some of the regularities that occur in real sequences by looking for recurring phrases and recognising common non-linear structure. It does this on-line, so that the structures can be used to explain and predict real-time sequences, but it is also effective in recognising structures in static sequences like English text. SEQUITUR will eventually become part of a PBD system, modelling and predicting the actions we perform, to strike a blow in the battle against the tyranny of repetition.

Acknowledgments

I am grateful to Ian Witten and Dave Maulsby for the ideas and insights that they have contributed to this research, and to Przemyslaw Prusinkiewicz for suggesting the l-system application.

References

Prusinkiewicz, P. & Lindenmayer, A. (1990) The algorithmic beauty of plants, *Springer-Verlag.*

Berwick, R.C., & Pilato, S. (1987) Learning syntax by automata induction. *Machine Learning, 2, 9-38.*

Storer, J.A.. (1988) Data Compression—methods and theory, *Computer Science Press.*

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.