CSE6390 3.0 Special Topics in AI & Interactive Systems II
Introduction to Computational Linguistics
Instructor: Nick Cercone - 3050 CSEB - nick@cse.yorku.ca
Winter Semester, 2010

Markov (or normal) algorithms

In the early 1950's the Russian mathematician A. A. Markov proposed what he termed "normal algorithms" to attack the problem of transforming strings of symbols from one string to another in a mechanical way. We call them *Markov algorithms*. This transformation is an abstraction from many of our common problems. For example adding two numbers x and y can be considered the problem of transforming the string "x + y" into the string "z" where "z" represents the sum of x and y. As another example the problem of information storage and retrieval can be thought of as the problem of transforming strings representing queries for documents into strings representing the documents satisfying the query.

A couple of basic guidelines to keep in mind when working with Markov algorithms are:

1. *When transforming a string, we shall generally not want to operate on the entire string (which may be arbitrarily long) at once, but rather only a small contiguous part of it.*
2. *Assume that the device which is to utilize these algorithms is capable of recognizing the occurrences of a given substring within a given string. The occurrences may be several in number and may overlap.*
3. *A particular occurrence of a substring may be distinguished by marking it with an asterisk (*).*

Example:

The word *ratatattat* contains three successive occurrences of the string *tat*, namely

*ra*tat*attat*   *rata*tat*tat*   *ratatat*tat**

The first two of these three occurrences overlap by one letter.

We shall consider these occurrences as numbered, and shall refer to the left-most occurrence of a string *A* in a string *B* as the first occurrence of *A* in *B*. One special string, the empty string, contains no symbols. It plays a role analogous to the empty set. If a given string *A* contains *n* symbols, the empty string *W* is considered to have *n+1* occurrences in *A*: before the first symbol (the first occurrence of *W*), after the last symbol, and between every two adjacent symbols.

The transformations of which a Markov algorithm is composed are those that replace the first occurrence of a specified string *A* in the given string by another specified string *B*. Markov algorithms consist of a sequence of such transformations.

Definition - Let us consider strings of symbols from a given finite symbol set, called the alphabet. We suppose that the alphabet does not contain the symbols "→" and "•".

A *simple* (Markov) production is a string *A* →*B*, where *A* and *B* are strings in the alphabet. A *conclusive* (Markov) production is a string *A* →•*B*, where *A* and *B* are strings in the alphabet. In the production *A* →*B* (*A* →•*B*) the *antecedent* is *A* and the *consequence* is *B*.

Definition - Let *A* →*B* (or *A* →•*B*) be a Markov production where *A* and *B* are strings in the alphabet. Let *S* be a string of symbols in the alphabet. The production is *applicable* to *S* when there is at least one occurrence of *A* in *S*. Otherwise the production is *not applicable* to *S*.

Examples:

Let the alphabet be the English alphabet a,b,...,z. Let the string S be "abactababrstc"

- applying the production act →bbb to S, we obtain "abbbbababrstc"
- applying the production ba →•one to S we obtain "aonectababrstc"
- applying the production tab →*W* to *S* we obtain "abacabrstc"

The production abc →rst is not applicable to *S*.

Definition - A *Markov Algorithm* is a finite sequence $P_1, P_2, \ldots, P_n$ of *Markov productions* to be applied to strings in a given alphabet according to the following rules. Let *S* be a given string. The sequence is searched to find the first production P_i whose antecedent occurs in *S*. If no such production exists, the operation of the algorithm halts without change in *S*. If there is a production in the algorithm whose antecedent occurs in *S*, the first such production is applied to *S*. If this is a conclusive production, the operation of the algorithm halts without further change in *S*. If this is a simple production, a new search is conducted using the string *S'* into which *S* has been transformed. If the operation of the algorithm ultimately ceases with a string *S**, we say that *S** is the *result* of applying the algorithm to *S*.

Example:

Take the alphabet to be {a, b, c, d}. The algorithm is given below.

Algorithm M1

M11: [conclusive]	a d →	•d c
M12: [simple]	b a →	W
M13: [simple]	a →	b c
M14: [simple]	b c →	b b a
M15: [simple]	W →	a

Taking S = "dcb" we apply the algorithm
by M15 dcb becomes adcb
by M11 adcb becomes dccb and halts.

Taking S = "dbc" we apply the algorithm
by M14 dbc becomes abba
by M12 abba becomes db
by M15 db becomes adb
by M11 adb becomes dcb and halts.

Taking S = "bdc" we apply the algorithm
by M15 - abdc, by M13 - bcbdc,
by M14 - bbabdc, by M12 - bbdc,
by M15 - abbdc, by M13 - bcbbdc,
by M14 - bbabbdc, by M12 - bbbdc,
by M15 - abbbdc, ...

The operation of the algorithm has not ceased at this point, and it is rather evident that the algorithm when applied to bdc will operate without ceasing, producing longer and longer strings of the form b...bdc.

The algorithm in the example above has no purpose. But if the concept of a Markov algorithm is to be useful, we must show that we can accomplish meaningful tasks with these algorithms.

Example:

Let the alphabet be unspecified, and let *A* be a fixed string in this alphabet. We wish to transform the arbitrary string *S* into the string *AS*. This is easily accomplished with the following.

Algorithm M2

M21: [conclusive] W → •A

Not all tasks are as easy to accomplish. Suppose, for example, that we wished to transform *S* not into *AS*, but rather into *SA*. We cannot use algorithm M2, for successive applications of this will produce the strings *AS, AAS, AAAS*, ... Nor can we write the algorithm as *S* →•*SA*, for there would need to be infinitely many productions with no first production. In fact because the productions are always applied to the first occurrence of *A* in *B*, there is difficulty any time we wish to operate with the second, third, or last occurrence. We overcome this difficulty by introducing the use of special marker symbols which are not a part of the given alphabet. By use of these markers we can mark a particular point with a string and operate on them at that point.

Example:
Let β be a *marker* not in the alphabet. If *S* is a string in the alphabet, the result of applying algorithm M3 to *S* is the string *SA*.

Algorithm M3

M31: [interchange]	$\beta\delta \rightarrow \delta\beta$		$\delta, A \in$ member of alphabet
M32: [conclusive]	$\beta \rightarrow \bullet A$		
M33:	$W \rightarrow \beta$		

Since *S* initially does not contain β, the third production is then used to move β past the symbols in *S*. If *S* contains n occurrences of symbols, then after *n* steps we obtain the string *S*β. At this point the first production no longer applies, and the second production produces *SA*. Since this production is conclusive, the string *SA* is then the result.

In the preceding example, we have introduced a new notation. Namely, in the first production we have used the variable δ which ranges over the symbols in the alphabet. Thus the first line is not really a production, but rather a *production schema*, denoting all the productions which can be obtained by substituting symbols of the alphabet for δ.

Because of the manner in which the Markov algorithms are used, the order in which the productions are written is vital. If the first two lines of algorithm M3 were interchanged, the result would be to transform *S* into *AS*, rather than into *SA*, and the productions represented by $\beta\delta \rightarrow \delta\beta$ would never be used. Within production schema the order is not critical. A little thought should convince you that the production schema represents sections of the algorithm in which the order of the individual productions applies and they will all do the same thing, in different contexts.

We conclude this section with several examples of tasks which can be accomplished by Markov algorithms. The development of this subject goes far beyond this introduction. In particular, *any task which can be accomplished by the use of algorithms in a liberal sense can be accomplished by a Markov algorithm*.

In the following examples the alphabet will be left unspecified except that it contains none of the markers or special symbols which are explicitly stated.

Example:

This algorithm transforms every string into the empty string.

Algorithm M4

M41: [production schema]	$\delta \rightarrow W\delta$	$\delta \in$ member of the alphabet

In operation this algorithm successively picks off the first letters of a string, as long as any letters remain. When the string becomes the empty string, the process halts since there is not transformation whose antecedent is the empty string.

Example:

This algorithm leaves the empty string unchanged but deletes the first letter of any non-empty string and halts. Marker: β.

Algorithm M5

M51: [prod. schema]	$\beta\delta \rightarrow \bullet W$	$\delta \in$ member of the alphabet
M52:	$\beta \rightarrow \bullet W$	
M53:	$W \rightarrow \beta$	

Example:

Often it is useful to know the number of symbols in a string. This is easily accomplished in tally notation by replacing every symbol by a tally marker. Special symbol: 1.

Algorithm M6

M61: [prod. schema]	$\delta \rightarrow 1$	$\delta \in$ member of the alphabet

Since “1” is not an element of the alphabet, the operation ceases when every symbol has been transformed.

Example:

At times one wishes to discard a portion of the symbol string, as one would discard data after computing the answer. The following algorithm discards everything to the left of the special symbol β.

Algorithm M7

M71: [prod. schema]	δβ	→	β	δ ∈ member of the alphabet
M72:	β	→	•*W*	

Example:

In almost every problem there is some point of which a decision must be made, dependent on the results of a calculation up to that point. We now present a Markov algorithm for making such a decision. An arbitrary string in the given alphabet is examined to determine whether it is a specified string *A*. If it is, the entire string is replaced by the string *B*; otherwise the entire string is replaced by the string *C*. Marker: ß.

Algorithm M8

M81:	δβ	→	βδ	δ ∈ member of the alphabet
M82:	βδ	→	β	
M83:	β	→	•*C*	
M84:	*A*δ	→	β	
M85:	*A*	→	β	
M86:	*A*	→	•*B*	
M87:	*W*	→	β	

If the given string, *P*, does not contain an occurrence of the string *A*, the last production introduces a β, and then the second and third production schema erase *P* and replace it with *C*. If *P* contains an occurrence of *A*, but is not *A*, either the fourth or fifth production schema is used to introduce the β; the first schema moves the βto the left end of *P* and then the second and third operate as before. Finally, if the string *P* is actually *A*, the sixth production applies and *P* is transformed into *B*. Notice that the productions in M8 refer directly to the string *A*, which might be quite long. Since *A* is known *a priori*, this is permissible: we could always replace such a reference by a letter for letter search for *A*.

Example:

Another procedure which is quite common is that of doubling or duplicating a string. Often we wish to perform transformations which destroy a string, but which are only tentative in nature: at some point we may decide that the transformations are wrong and we wish to begin anew. Thus we must be able to save a copy of the original string to which we can return. Given a string *P*, the following algorithm produces the string *PP* and halts. Markers: §, β, σ

Algorithm M9

M91: [prod. schema]	*f*δβ	→	δβ*f*	δ, *f* ∈members of the alphabet
M92: [prod. schema]	§*f*	→	*f*β*f*§	
M93:	β	→	σ	
M94:	σ	→	*W*	
M95:	§	→	•*W*	
M96:	*W*	→	§	

Example:

Another procedure which is quite common is that of reversing a string of characters. We do this by moving the first character to the end as before, then moving the next character down to the position just preceding the first character, and so on. Markers: §, β

Algorithm M10

M101:	§β	→	•*W*	δ, *f* ∈ members of the alphabet
M102:	§δβ	→	βδ	
M103:	§δ*f*	→	*f*§δ	
M104:	§δ	→	βδ	
M105:	*W*	→	§	

Illustrating this algorithm on the string “ABCD” we have

by M105 =>	§ A B C D
by M103 =>	B § A C D
by M103 =>	B C § A D
by M103 =>	B C D § A
by M104 =>	B C D ß A
by M105 =>	§ B C D ß A
by M103 =>	C § B D ß A
by M103 =>	C D § B ß A
by M102 =>	C D ß B A
by M105 =>	§ C D ß B A
by M103 =>	D § C ß B A
by M102 =>	D ß C B A
by M105 =>	§ D ß C B A
by M102 =>	ß D C B A
by M105 =>	§ßD C B A
by M101 =>	D C B A