MARKOV MACHINE CODE

A Markov function performs a single substring replacement operation on some of its arguments and returns the others unchanged. In general, Markov functions operate on strings. Here we restrict those strings to certain finite sequences of bits. This means that we can view our Markov function as either a total computable function $f : N \to N$ or a total computable function $f : S \to S$, where S is the set or type of finite strings on the alphabet $\{0, 1\}$. A Markov function can be used as the update rule in a partial recursive "tape machine" program.

We specify a Markov function as a finite sequence or list of "productions" or "replacement rules." The constraints placed on these productions are justified by the "source code" that such a machine code supports. A "group" is of the form $10, 110, 1110, ...$ In other words, any natural number of 1s followed by a 0 is a group. A replacement rule describes the replacement of one non-empty finite sequence of groups by another. For instance, $1101110 \to 1010$ is a valid replacement or production. So is $10 \to 11111010$. On the other hand, $01 \to 110$ is not a valid production, because 01 is not a group.

We can define a Markov function $f := 110 \to 10; 1110 \to 110$. We can also use a notation like $f := \frac{110}{10}, \frac{1110}{110}$ or $f := 110 > 10 : 1110 > 110$.

For "machine code" Markov functions, it is convenient to restrict the domain to inputs x that are also finite non-empty sequences of groups. While the domain won't include *all* natural numbers, every valid input string is readable as a positive even integer. To calculate fx[1] we apply the first possible production at the leftmost possible position of x and return the result. If no production applies, then x is returned unchanged.

For instance, $f11010 = 1010$. Also $f1110110 = 110110$ and $f10 = 10$.

We can encode the definition of a Markov function by using 0 to represent both $\to$ and ; in our first notation.

For instance, if $f := 110 \to 10$, then $f := 110010$.

If $f := 10 \to 110; 110 \to 10$, then $f := 1001100110010$.

This "machine code" notation is the "official" way to represent such a function, and this notation demonstrates that every Markov function is also just a positive even integer.

Any Markov function f can be used to generate a partial recursive "tape

[1] I just mean $f(x)$ but am aiming at a cleaner notation that suggests the action of the kind of tape machine we are building from such functions.

machine" function f^*. To calculate f^*x, we generate the sequence T_n, with $T_0 = x$ and $T_{n+1} = f(T_n)$. We generate this sequence until $f(T_i) = T_i = T_{i+1}$ and return T_i. This first fixed point in the sequence T_i is our output, but only if and when such a fixed point is found. A *non-immediate* repetition of a value demonstrates an infinite loop or cycle, which at least allows us to understand the function to "return" a symbol like ∞ or $\nearrow$ to indicate that the calculation failed. Of course one can actually treat ∞ as a genuine value *if* one reserves it for the detection of an infinite loop. In complicated cases, the proceeding calculation is simply indeterminate. Further computation may or may not persuade us to decide that $f^*(x) = y$ or $f^*(x) = \infty$, so that x is not in the domain of f^*.

For instance, let $f := 1001100110010$. In a more readable form, this is $f := 10 \to 110, 110 \to 10$.

So $f^*(110) = 110 \to 10 \to 110 = \infty$. The repetition is not immediate, so we don't have a fixed point for f, but it's clear that we have found a cycle of length 2 in the sequence of instantaneous descriptions T_n.

On the other hand, $f^*(11110) = 11110 \to 11110 = 11110$. The repetition is immediate and f has no productions that can be applied to 11110, so this fixed point is returned.

Our description so far is not intuitive as the typical presentation of Markov functions, but it has the advantage of already being in the form for implementation directly on bits of actual computer memory in a language like C. Now that we have the machine code for this language, we can build a high level language on top of it.

Let us consider a "typical" Markov function that operates on strings created from the digits 0 through 9. We simply encode $0 = 10$, $1 = 110$ and so on.

If our high-level description of f is $f := 2 \to 3, 1 \to 5$, then we f to get $f := 1110 \to 11110, 110 \to 1111110$. We might call this the assembly language version of f. Our actual machine code is

$$f := 1110011110011001111110.$$

Because this is just a string of 1s and 0s, we can read it immediately as the base-2 representation of a number. Compiling a typical Markov function is simple. We just need to encode the countable alphabet as $10, 110, 1110, ...$, reserving 0 as a punctuation mark. Of course we could also reserve 10 as a punctuation mark, bumping up all the other encodings, and work strictly with groups. Some might find this preferable,

since it's conceptually simpler, even at the cost of lengthening programs and tape descriptions.

We can encode $0, |, f_0, f_1, f_3, ...$ as $10, 110, 1110, 11110, 111110, ...$ We can define an official "high level" tape machine (in the low level Turing style) by declaring the convention that f_0 is added to the beginning of the input string. So $T_0 = f_0 \frown X$, where $\frown$ represents concatenation. Then $T_1 = f(T_0)$ and so on as before. The symbols f_i are like the "heads" of a Turing machine, though with tape machines we can use as many f_i symbols simultaneously as we want. Declaring an official "start state" is convenient when programming in the "tape machine language." In machine code terms, we just declare $T_0 = 1110 \frown x$ and define $T_{n+1} = f(T_n)$ as before.

Strictly speaking, the groups are just $s_0, s_1, s_2, ...$ We could also understand tape machines to operate on finite sequences of non-negative integers. The tape $||0|0|||0$ is equivalent to $(1, 0, 3)$. In this case, we might express a Markov function f as

$$f := [(0, 2) \to (3)], [(0, 0) \to (4, 1)].$$

So $f(3, 0, 2) = (3, 3)$, and so on.