From PDAs to Turing Machines

November 26, 2012

CS 81: Computability and Logic

Practical Parsing

Recursive Descent

- ✓ Form of code follows the grammar.
- ✓ Efficient and correct for LL grammars.

Another very practical method: Shift-Reduce Parsing

- ✓ Efficient and correct for LR grammars
- ✓ Parser code is usually computer-generated!

But neither of these handle ALL CFGs…

CYK (CKY) Algorithm

- ✓ Works for any CFG.
- ✓ Parses inputs of length n in $O(n^3)$.
- ✓ Requires a grammar in "Chomsky Normal Form"
  - All rules of the form $A \to a$ or $A \to BC$.
  - Any CFG can be put into this form
    - If the empty string should be accepted, handle it separately.
    - You may or may not be happy with the new parse tree
- ✓ Key ideas:
  - For every substring, what nonterminals produce it?
  - Dynamic programming for efficiency

Dynamic Programming

Recursive expressions, such as

$$\begin{array}{lcll} f(n) & := & 1 & \text{if } n < 3 \\ f(n) & := & f(n-1) + f(n-2) & \text{otherwise} \end{array}$$

are very clear, but inefficient if taken literally.

Two roughly-equivalent solutions:

- ✓ Memoization: keep track of what f values have been computed
- ✓ Dynamic Programming: Compute all f values in a good order

CYK Algorithm

Let

$$x = x_1 x_2 \cdots x_n$$

be the string to be parsed.

Define

$$a(i, j) := \{ B \mid B \Rightarrow^* x_i x_{i+1} \cdots x_j \}$$

Then $x \in L(G)$ iff $S \in a(1, n)$

$$\begin{aligned} a(i, i) &= \{ C \mid C \to x_i \} \\ a(i, k) &= \{ C \mid C \to AB \wedge A \in a(i, j) \wedge B \in a(j + 1, k) \} \end{aligned}$$

CYK “Wavefront” Matrix

a(1, 1)	a(1, 2)	a(1, 3)	a(1, 4)		a(1, n-1)	a(1, n)
	a(2, 2)	a(2, 3)	a(2, 4)		a(2, n-1)	a(2, n)
		a(3, 3)	a(3, 4)		a(3, n-1)	a(3, n)
			a(4, 4)			
					a(n-1, n-1)	a(n-1, n)
						a(n, n)

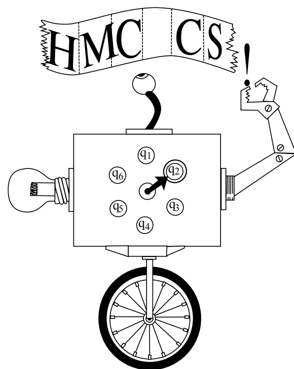
Each entry is computed from entries in its same row and column, e.g.
a(1, 4) from
a(1,1) and a(2, 4),
a(1, 2) and a(3, 4),
a(1, 3) and a(4, 4).

CYK Example

Let's parse `(()())` using the grammar

$$
\begin{aligned}
S &\rightarrow LT \\
T &\rightarrow SR \\
S &\rightarrow LR \\
S &\rightarrow SS \\
L &\rightarrow (\\
R &\rightarrow)
\end{aligned}
$$

Turing Machines



Turing Machines

- ✓ Named after (not by) Alan Turing
- ✓ Perhaps the most important computational model (if not the most practical)
- ✓ Simple, yet apparently universal
- ✓ Church-Turing Thesis (a.k.a. Church's Thesis)
  - Any "intuitively computable" procedure can be performed by a TM

Why Turing Machines?

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes

Why Turing Machines?

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent†. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as

Why Turing Machines?

17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Let us imagine the operations performed by the computer to be split up

Official Definition

A Deterministic TM consists of

- ✓ A finite set K of control states
- ✓ A finite alphabet Σ
- ✓ A finite "tape alphabet" Γ $\quad (\Sigma \subset \Gamma, \; \sqcup \in \Gamma \setminus \Sigma)$
- ✓ A starting state $s \in Q$
- ✓ Halting states $H = \{y, n\} \subseteq K$
- ✓ Transitions taken from

$$((Q \setminus H) \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

Running a Turing Machine

- ✓ Write the finite input in the middle of an infinite blank tape
- ✓ Position the
- ✓ Run the TM

$$K \times \Gamma \quad \rightarrow \quad K \times \Gamma \times \{L, R\}$$

- ✓ TM halts iff we enter a halting state "y" or "n".
  - The TM *accepts* the input if it ends in y.
  - The TM *rejects* the input if it ends in n.
  - TM might never halt!

Beyond Decision Problems

Rather than accepting a language, we can use a TM to compute a function $f(x) = y$:

- ✓ The machine starts with some input x on the tape
- ✓ The machine halts (however) with some string y on the tape.
- ✓ If the machine diverges (does not halt) on some inputs, it computes a *partial function*.

TM Demo

http://ironphoenix.org/tril/tm/

Configurations

An instantanous snapshot of a TM is called a *configuration*

- ✓ The "state" of the "whole machine"
- ✓ Contents?
- ✓ Why are configurations always finite?

Universal Turing Machines

UTMs can be shown to exist by constructing them.

Think about what would be required.

- ✓ The tape has to hold the tape of the machine being simulated.
- ✓ The tape has to hold the program of the machine being simulated.
- ✓ The tape has to hold the current state of the machine being simulated.

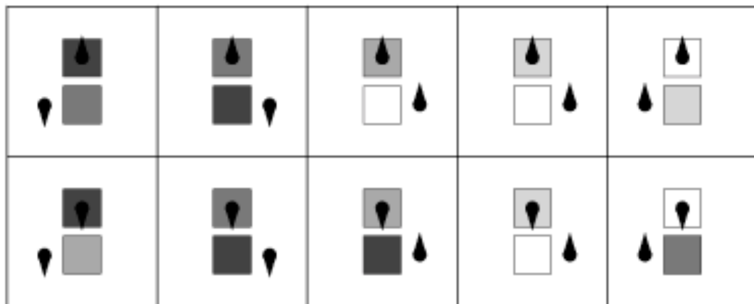
All this is possible, if somewhat laborious to construct.

Specific UTMs

- ✓ The first was constructed by Turing himself.
- ✓ Shannon showed any UTM could be converted either to a 2-symbol machine or to a 2-state machine (with lots more states or symbols, respectively).
- ✓ Minsky (1960) gave a 7-state 6-symbol machine.
- ✓ Watanabe (1961) gave an 8-state 5-symbol machine.
- ✓ Minsky (1962) gave a 7-state 4-symbol machine.
- ✓ Rogozhin (1996) gave a 4-state 6-symbol machine
- ✓ Wolfram and Reed (2002) gave a 2-state 5-symbol machine.
- ✓ Smith and Wolfram (2007) gave a 2-state 3-symbol machine.
- ✓ No 2-state 2-symbol UTM exists.

A Specific UTM

2-state, 5 symbol UTM published by Wolfram in 2002



adapted from Wolfram, S. *A New Kind of Science*. Wolfram Media, p. 707, 2002.

TM Programming Tips

- ✓ Divide the work into different phases/subroutines
- ✓ Controller has an arbitrarily large "finite memory".
- ✓ Squares can be "marked" and "unmarked" (finitely many ways)
- ✓ Take advantage of TM extensions

TM Variations

The following yield no extra power:

- ✓ Adding the option to write *or not* on each step.
- ✓ Adding the option to stay-in-place rather than moving L/R.
- ✓ Making the tape infinite in both directions
- ✓ Adding an extra "Erase Tape" move.
- ✓ Multiple tapes with multiple (independent) read/write heads
- ✓ 2-D infinite tape
- ✓ Nondeterminism (!)

Many attempts to define models of computation; all turn out to be equivalent to Turing Machines.

- ✓ If you can do it in Prolog or Python or C++, a TM can do it (slowly)