Unsolvable Problems for Pushdown Automata

Soon after introducing pushdown automata we proved that their membership problem was solvable. Well, this was not quite true since we did it only for deterministic machines. But, it should not be too difficult to extend this result to the nondeterministic variety. Soon we shall when we examine formal languages. Also at that time we shall show that the emptiness and finiteness problems for pushdown automata are solvable too. Unfortunately most of the other decision problems for these machines are unsolvable. So, it would appear that stacks produce some unsolvability.

As usual we shall use reducibility. We shall map some undecidable Turing machine (or actually r.e. set) problems into pushdown automata problems. To do these reductions we need to demonstrate that pushdown automata can analyze Turing machine computations. Recall from before that a *Turing machine configuration* is a string that describes exactly what the machine is doing at the moment. It includes:

a) what is written on the tape,
b) which symbol the machine is reading, and
c) the instruction the Turing machine is about to execute.

If we dispense with endmarkers and restrict ourselves to the two symbol alphabet {0, 1, *b*} then a configuration is a string of the form x(Ik)y or x where x and y are strings over the alphabet {0, 1, *b*}. (We interpret the configuration x(Ik)y to mean that the string xy is on the tape, the machine is about to execute instruction *Ik*, and is reading the first symbol of the string y. A configuration of the form x with no instruction in it means that the machine has halted with x on its tape.) Let us prove some things as we develop the ability of pushdown machines to examine Turing machine computations.

Lemma. *The set of strings that are valid Turing machine configurations is a regular set.*

Proof. This is simple because a Turing machine configuration is of the form $[0+1+b]^*$ or of the form $[0+1+b]^*(I1[0+1]^*)[0+1+b]^*$. (Note that we used square brackets in our expressions because the parentheses were present in the Turing machine configurations.)

Since these are regular expressions and represent a regular set.

A *computation* is merely a sequence of configurations (C_i) separated by markers. It can be represented by a string of the form:

$$C_1\#C_2\# \ldots \#C_n$$

Since the regular sets are closed under concatenation and Kleene star these kinds of strings form a regular set also.

Now let us examine just what makes a sequence of valid Turing machine configurations *not a valid halting computation* for a particular machine. Either:

1) C_1 is not an initial configuration,
2) C_n is not a halting configuration,
3) one of the C_i contains an improper instruction, or
4) for some $i < n$, some C_i does not yield C_{i+1}.

Parts (1) and (2) are easy to recognize. An initial configuration is merely a string of the form (I1)[0+1+*b*]* and a halting configuration is a string of the form [0+1+*b*]*. These are both regular sets. Part (3) is regular also because Turing machine instructions are required to be members of a sequence beginning at I1 and there can be only a finite number of them for a particular Turing machine. Thus this is regular also because a finite automata can recognize finite sets.

Let us now put this in the form of a lemma that we shall use later when we wish to build a pushdown machine which check Turing machine computations.

Lemma. *For each Turing machine there are finite automata that are able to detect:*

a) initial configurations,
b) halting configurations, and
c) configurations with improper instructions.

The final situation that leads to a string not being a valid Turing machine computation is part (4) of our previous list. This when a configuration does not yield the next in the sequence. We shall show now that a nondeterministic pushdown machine can detect this.

Lemma. *For each Turing machine, there is a pushdown automaton that accepts pairs of configurations (for that Turing machine) such that the first does not yield the second.*

Proof. Let C_1 and C_2 be valid configurations for a particular Turing machine. We shall build a pushdown automaton which accepts the input $C_1\#C_2$ whenever it is not the case that $C_1 \rightarrow C_2$. The automaton will

operate in a nondeterministic manner by first selecting the reason that C_1 does not yield C_2, and then verifying that this happened. Let us analyze the reasons why C_1 might not yield C_2.

a) *No configuration can follow C_1.* There are three possible reasons for this, namely:

1) C_1 is a halting configuration.
2) The instruction in C_1 is not defined for the symbol read.
3) C_1 = (Ik)x and the Turing machine wishes to move left.

Since these three conditions are finite, a finite automaton could detect them, and so can our pushdown machine.

b) *The instruction in C_2 is incorrect.* In other words it does not follow from the situation in C_1. This can also be detected by a finite automaton.

c) *The tape symbols in C_2 are wrong.* Suppose that

C_1 =

x	a	(	I	k	)	c	y

and instruction Ik calls for the Turing machine to write the symbol d and transfer to instruction Im if it reads the symbol c. Then it should be the case that

C_2 =

x	(	I	m	)	a	d	y

if the Turing machine moved left, and

C_2 =

x	a	d	(	I	m	)	y

if the Turing machine moved a square to the right. Suppose that instead of things being as they should, we end up with the following configuration:

C_2 =

u	(	I	m	)	v

The errors that could take place have been tabulated in the following table.

left move error:	**$u \neq x$**	**or**	**$v \neq ady$**
right move error:	**$u \neq xad$**	**or**	**$v \neq y$**

The pushdown machine first nondeterministicly selects which of the four errors has taken place and then verifies it. Here is how this works. Our pushdown automaton first scans xacy (the tape in C_1) and uses its stack as a counter to select the position of where the error will be in C_2. It then remembers (in its finite control) exactly what symbol should be in that location on C_2. Now it counts off squares of C_2 until it reaches the site of the error and verifies that the wrong symbol is in that location.

To recap, our pushdown automaton first selects the error that makes C_1 not yield C_2 and then verifies that it did indeed take place.

Now we possess all of the tools necessary to show that pushdown machines can analyze strings and detect whether or not they are Turing machine computations. This is done by enumerating what could be making the string not a proper computation, and then verifying that something of this sort took place.

Theorem 1. *For every Turing machine, there is a pushdown automaton that accepts all strings that are not valid halting computations for the Turing machine.*

Proof Sketch. The pushdown machine in question merely selects (nondeterministicly) the reason that its input cannot be a valid halting computation for the Turing machine. Then it verifies that this error took place. The preceding lemmas provide the list of errors and the methods needed to detect them.

This theorem is very interesting in its on right. as well as being a nice example of nondeterministic computation. In fact, the *guess and verify* strategy cannot be carried out on a deterministic device unless the choices remain constant no matter how long the input gets.

It also indicates that pushdown automata can in some sense analyze the computations of Turing machines. (By the way, npda *cannot* detect *valid* halting computations for Turing machines. This should become clear after observing linear bounded automata that is a more powerful device that is able to recognize valid computations.) The major use of this theorem, however, is to

prove that several decision problems involving pushdown machines are unsolvable. This will be done by reducing unsolvable problems for the r.e. sets to problems for pushdown automata.

Theorem 2. *Whether or not a pushdown automaton accepts every string over its alphabet is unsolvable.*

Proof. We shall reduce the emptiness problem for Turing machines (is $W_i = \varnothing$?) to this problem (known as the Σ^* problem since Σ is the input alphabet) for pushdown machines.

Let M_i be a Turing machine and let $P_{g(i)}$ be the pushdown automaton that accepts the set of strings that are not valid halting computations for Turing machine M_i. Then note that:

$$\begin{aligned} W_i = \varnothing \quad & \text{iff } \forall x[M_i(x) \text{ never halts}] \\ & \text{iff } \forall ax[M_i(x) \text{ has no halting computations}] \\ & \text{iff } P_{g(i)} \text{ accepts every input} \end{aligned}$$

That was the reduction from the emptiness problem for Turing machines to the Σ^* problem for pushdown machines. Recalling our results on reducibilities indicates that both must be unsolvable. (Also they are not r.e.)

Acceptance of strings that are not valid halting computations of Turing machines leads to several other unsolvability results concerning pushdown automata. Two which appear in the exercises are the *equivalence problem* (whether two machines accept the same set) and the *cofiniteness problem* (whether the complement of the set accepted by a pushdown machine is finite). They are proven in the same manner.