CS 361 Meeting 21 — 4/17/20

The Church-Turing Thesis

(Click for video)

1. An important part of our agenda at this point is to explore the case that the Turing Machine is a model that captures everything a computer can do. This assumption is known as the Church-Turing Thesis. We cannot prove this, but we can reassure ourselves that it seems reasonable to make this assumption in several ways.

   - We have already been building a case that we can implement algorithms on a Turing machine in ways that mimic familiar ways you have already learned to implement algorithms in traditional programming languages and using conventional architectures.
     - We have seen it is possible to copy information like an assignment statement, to do simple arithmetic, to mimic control structures like if statements and loops and to divide a Turing machine's tape into subsections corresponding to distinct program variables and even arrays.
     - Taking this approach to the limit, we could potentially describe a Turing machine that could interpret its input as a machine language program for some available architecture and then simulate the machines execution of that program. It would be possible but very tedious!
     - If we believe that traditional programming languages and architectures are "computationally complete", then this would support the claim that TMs are too. Alas, it would not rule out the possibility that there is some more powerful notion of computation that we are missing.
   - To convince ourselves that Turing Machines are as powerful as we can get, we will explore models of computation based on adding features to the TM and to other models.
     - If adding features does not increase the computational power of the model, maybe the model is as powerful as it could be!
     - For example, we have already done a bit of this with DFAs. We saw (surprisingly?) that adding nondeterminism to our model for DFAs did not extend its computational power. On the other hand, if we changed the DFA model so that a DFA could move back and forth and write on its tape as long as it did not go beyond the space originally filled with its input, this would increase the power of the model.

2. As a starting point, rather than adding a feature to the TM model, let's consider the impact of adding a feature to one of our less powerful models, the PDA. What happens to the computational power of a PDA if we give it two stacks?

   - We have seen that the intersection of two context-free languages can be a language that is not context-free.
   - For homework, I asked you to prove that the intersection of a context-free language and a regular language is context-free.
   - The answer to this homework question involved starting with a PDA for a CFL and a DFA for a regular language and building a new PDA that simulated the simultaneous execution of the original PDA and DFA to make sure each input string belonged to both machine's language.
   - It isn't obvious how to do a similar construction for two PDA's because in order to simulate two PDA's with one PDA, you would have to fit two stacks into one.
   - If, however, you merged two 1-stack PDAs into a 2-stack PDA, it would be easy to simulate the stacks of the original 1-stack PDAs.
   - This suggests that a 2-stack PDA could recognize languages that are not context-free. We will show that this is the case shortly.

2-PDAs

(Click for video)

1. Recall the formal definition of a pushdown automaton:

   Definition: A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

Click here to view the slides for this class

Q is a finite set of states,
Σ is a finite input alphabet,
Γ is a finite stack alphabet,
$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function, and
$F \subset Q$ is the set of final or accepting states.

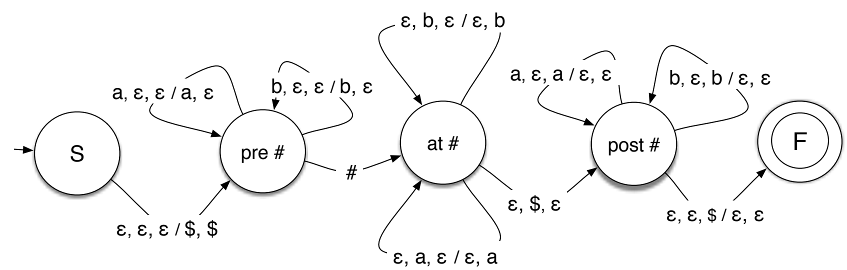
2. A PDA is constructed by adding a stack to a DFA. Adding a stack clearly increases the power of the machine! What if we did it again? That is, would a PDA with two stacks be more powerful that a PDA with just one stack?

    - To better motivate this idea, recall the homework problem where you were asked to show that the intersection of a context-free language and a regular language had to be a context-free language.
    - The correct approach to this problem was to take a PDA for the context-free language and a DFA for the regular language and show how to build a single PDA whose state set could keep track of both the state of the original PDA and the original DFA while its stack held the same contents as the original PDA's stack.
    - Suppose we tried the same approach to show that the intersection of two context-free languages had to be context free. We would start with two PDAs. We could build a new PDA that kept track of the states of both of the original PDA's but the new PDA would not have the resources to keep track of two separate stacks!
    - We saw that in fact, the intersection of two context-free languages may be a language that is not context-free (i.e., recognizable by a PDA).
    - If we had a PDA with two stack, we could simulate two, 1-stack PDAs! So, such a machine should be able to recognize the intersection of the languages of any 1-stack PDAs (i.e. the intersection of any pair of context-free languages).
    - By similar logic, one might imagine that a 2-stack PDA would not be able to simulate 2 2-stack PDAs so that recognizing the intersection of the languages of 2-stack PDAs might require 4-stack PDAs. Fortunately, this logic is not correct.

3. To explore these ideas, we can define a 2-tape PDA:

    Definition: A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:
    Q is a finite set of states,
    Σ is a finite input alphabet,
    Γ is a finite stack alphabet,
    $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon \times \Gamma_\epsilon)$ is the transition function, and
    $F \subset Q$ is the set of final or accepting states.

4. The diagram below is an example of the specification of a 2-stack PDA.



    - If a transition arrow is labeled "a, b, c / d, e" it means that if the machine is in the state where the arrow originates, it can transition to the target of the arrow if a is the next input character, b and c are the characters at the tops of its two states. If the transition is used, then the symbols b and c should be popped from the stacks and d and e should be pushed. Any of a, b, c, d, and e can be empty.
    - While in state pre #, the machine shown scans until it finds a # while pushing all of the as and bs it encounters onto its first stack so that at the end the first symbol from the input is at the bottom of the first stack and the last is at the top of the first stack.
    - When it hits the #, it pops each symbol from its first stack and pushes it on the second stack until the first stack is empty. When this is complete, the second stack contains a copy of the first half

of the input with the first symbol at the top of the stack and the last symbol at the bottom.

- Next, it scans to the right popping symbols off the second stack as long as each symbol on the stack matches the next input symbol.
- If the second half of the input matches the first, it will eventually empty the second stack and then transition to its only accepting state. Thus, the machine is designed to accept $w\#w$.

5. Remember, $w\#w$ is not a context-free language. The machine we just described therefore verifies our suspicion that 2-tape PDAs are more powerful than single stack PDAs.

The Power of 2 Stacks

(Click for video)

1. It may be a bit surprising, but adding a second stack to a PDA gives us a formalism for describing languages that is just as powerful as a Turing Machine!

2. To appreciate how we can justify this claim, recall the notion of a Turing machine configuration:

    Definition: A **configuration** of a Turing machine is a triple (u, q, v) where $q \in Q$ is the current state, uv is the contents of the non-blank portion of the tape with u being the portion to the left of the current head position and u being the portion from the symbol currently under the head to the end of the non-blank tape.

    - Given a TM configuration, we could store all the symbols that are before the tape head (u) in one stack and all of the symbols after the tape head (including the symbol under the tape head) in the second stack.
    - We could arrange for this initially using a few states equivalent to the pre# and at# states of the sample machine for $w\#w$ shown above.
    - We could then define the remaining states and transitions of the 2-tape PDA in a way that mimicked any TM. In particular, if δ_{TM} is the TM's transition function and δ_{PDA} is the transition function for our PDA then
        - If $\delta_{TM}(q, a) = (q', b, Right)$ then $\delta_{PDA}(q', \epsilon, \epsilon, a) = \{(q', b, \epsilon)\}$.
        - If $\delta_{TM}(q, a) = (q', b, Left)$ then for all c,
            * $\delta_{PDA}(q', \epsilon, \epsilon, a) = \{(q'\text{-push}, \epsilon, b)\}$, and
            * $\delta_{PDA}(q'\text{-push}, \epsilon, c, \epsilon) = \{(q', \epsilon, c)\}$

          (the intermediate state q'-push-c is only required because our definition of PDA's limits us to pushing one symbol on the stack at a time).

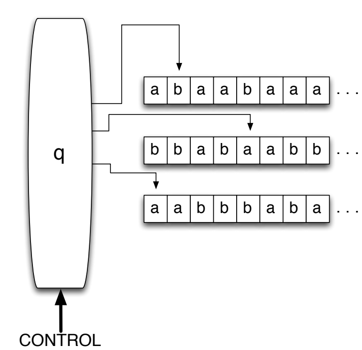
Thus, a 2-tape PDA is at least as powerful as a Turing Machine!

n-tape Turing Machines

(Click for video)

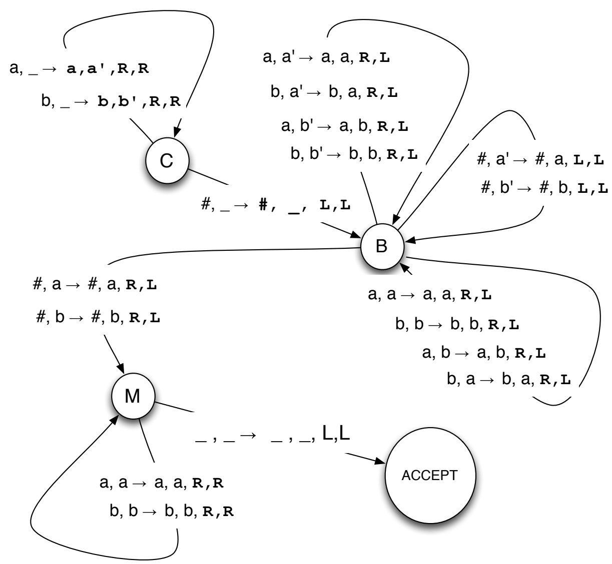
1. Given that adding multiple stacks to a PDA increases the model's power in a fundamental way, we might wonder what happens to the computational power of a TM if we give it more than one tape?

2. That is, we stick with one finite control that will be in a single state at any point, but we give the machine n-tapes and one read head that can be independently positioned for each tape as suggested by the figure below:

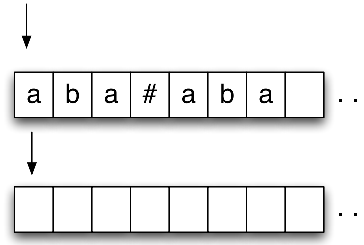


3. Such a machine might or might not be able to compute things that a Turing machine cannot compute, but it is much easier to program. To see this consider the following example.

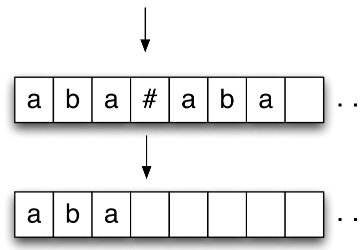
    - At the risk of making you think it is the only language any TM can decide, I will again use $w\#w$.
    - This language does not require a machine with many tapes, but it is definitely easier to recognize $w\#w$ on a 2-tape TM than on a single-tape TM.
    - On a single-tape TM, recognizing $w\#w$ requires making $|w|$ passes back and forth from one copy of w to the other w marking matching symbols to verify that each symbol has a match.
    - On a 2-tape TM, we can complete the entire process in one and a half passes:



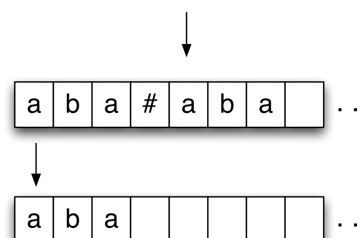
      – The machine would start with the input on its first tape and nothing on the other.



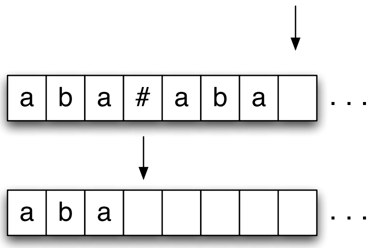
      – It would first scan the input from left to right copying symbols from the input tape to the machine's second tape until reaching a #. This is the role of state C = COPY.



      – Next, it moves the head on the second tape back to the left end of the tape (We would like to keep the head on the first tape right where it is, but since my version of n-tape TMs require each head to move left or right at each step, we have to make it wiggle back and forth). This is the role of state B = BACKUP.



      – Finally, it scans right on both tapes at the same time making sure that the contents of the second tape matches the contents of the second half of the input/first tape. This is the role of state M = MATCH.



- Hopefully, you could imagine how having multiple tapes could simplify recognizing other languages:
  - For example, the other day we considered

$$\{i\#x\#w_1\#w_2\#...\#w_k \mid i, x, w_i \in \{0,1\}^*, i \leq k,\ \&\ x = w_i\}$$

  - A 3-tape TM could recognize this language easily by first copying i from the input tape to a second tape and next copying w to its third tape. Then, as it moved to the right on its input tape it could decrement the value of n on the second tape each time it hit a marker. When the counter became 0, it could match the w on the third tape with w_i on the input tape. It would never have to back up on its input tape!