CS 361 Meeting 22 — 4/20/20

Equivalence of 1 and Multi-tape Turing Machines

(Click for video)

1. We can give a formal definition of how a multitape TM differs from a single tape TM..

    Definition: A n-tape Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

    Q is a finite set of states,

    Σ is a finite input alphabet (not containing the blank symbol),

    Γ is a finite tape alphabet which is a superset of Σ including the blank symbol,

    $\delta : Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{Left, Right\}^n$ is the transition function,

    q_0 is the start state,

    q_{accept} is the accept state, and

    $q_{reject} \neq q_{accept}$ is the reject state.

2. We can show that a multi-tape TM is no more powerful than a single-tape TM. To do this, we must show that single-tape TM can simulate the computation of any multi-tape TM. In particular, since we are interested in TMs as deciders, what I really want to show is that the sets of languages recognized and decided by multi-tape TMs and by single-tape TMs are identical.

3. The approach usually taken to establish such a theorem is to show how each of the two models of computation could simulate the other.

    - When we showed that NFAs were of equivalent power to DFAs, we showed that an NFA could recognize any language recognized by a DFA (trivially, since the DFA is an NFA), and that any language recognized by an NFA could be recognized by a DFA (using the subset construction).
    - The proof that a 2-stack PDA could simulate a TM by keeping the two pieces of the tape in each TM configuration in its two stacks was another, similar example.

4. In the examples of such simulations we have seen before (i.e., the NFA-DFA and the 2PDA-PDA equivalences), the machine that did the simulating was close enough to the machine being simulated that the simulator took one step (or maybe two!) for each step the simulated machine took.

5. Our simulation of a multi-tape TM on a single-tape TM will be different. The single-tape TM will have to take many steps to simulate each step of the multi-tape machine.

6. So, our general task is to describe a general procedure by which given an n-tape TM

    $$M^N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

    we can construct a 1-tape TM

    $$M^1 = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject})$$

    such that

    $$L(M^1) = L(M^N)$$

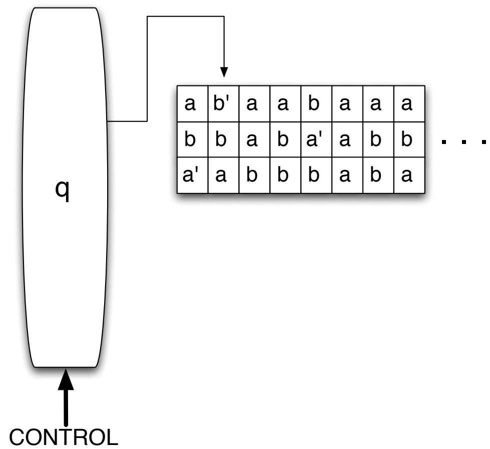
    and M^1 halts on w iff M^N halts on w .

7. I will instead use a single-tape TM with a tape alphabet that is much bigger than the alphabet of the machine it is simulating. The alphabet of the simulator will include n-tuples of characters from the alphabet of the simulated machine so that the symbol on the ith square of the single-tape simulator can represent all of the symbols at position i on the N tapes of the simulated machine. In addition, the alphabet will allow the simulator to mark any of the symbols in these tuples to indicate the symbol currently under each of the N tape heads of the simulated machine.

8. The simulating machine's alphabet gets a bit more complicated than this because, initially, its tape will contain the input written using

Click here to view the slides for this class symbols in Σ. The initial steps performed by the simulator will be to scan its input from left to right replacing each x in the input with $(x, \_, \_, \ldots, \_)$ with the spaces representing the contents of the other $N-1$ simulated tapes.

As a result, to simulate an N-tape machine with alphabet Γ, we will use a single-tape machine with alphabet $\Gamma' = \Sigma \cup (\Gamma \times \{', \epsilon\})^N$.



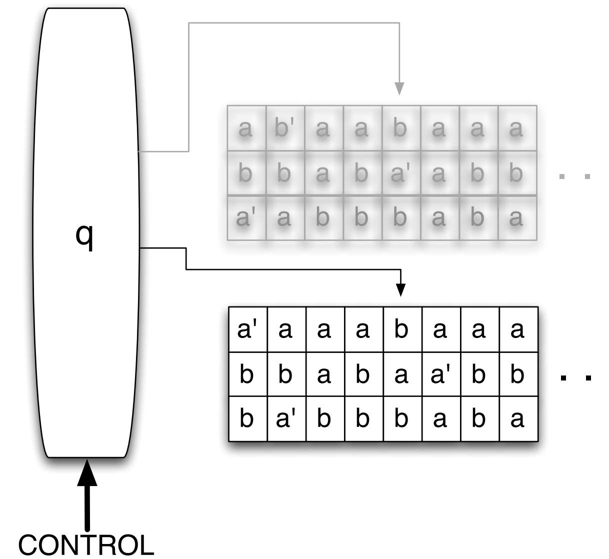
- Remember that the real goal here is to fit the contents of N tapes onto one. There are many ways of doing this.
- In the book, Sipser gives a construction in which the contents of N tapes is kept on one tape by first writing the contents of the first tape with a special mark on the symbol that is under the first tape head. This is followed by a special marker and the contents of the second tape. This is followed by another maker and the third tape, and so on.

9. Given our approach to encoding N tapes on 1, we still have our work cut out for us. We need to design M^1 in such a way that it can simulate each state transition made by M^N. M^1 will normally require many steps/transitions to simulate a single transition made by M^N.

   - To appreciate this, suppose that M^1 was simulating a 3-tape TM M^3, was in the configuration shown above, and that the transition function for M^3 included

$$\delta(q, b, a, a) = (q_k, a, a, b, L, R, R)$$

   - In this case, M^1 would need to update its tape as shown below to update its encoding of M^3's configuration appropriately (Note that the actual position of M^1's tape head is irrelevant since the primed symbols on the tape encode the positions of M^3's heads.):



   - This involves changing the symbols written on positions 1, 2, 5, and 6 on M^1's tape. This cannot be done in one step. Instead, M^1 will go through many steps and possibly many states to make the necessary changes on its tape.

10. To get a sense for what the set of states M^1 will need to accomplish this, suppose instead that you had to write a Java program to do this simulation.

    - To keep things simple, think about a 3-tape machine rather than an N-tape machine.

- Assume that the Java program will use three "infinite" arrays of characters to represent the tapes:

```
    private char [] tape1 = new char[∞];
    private char [] tape2 = new char[∞];
    private char [] tape3 = new char[∞];
```

- Assume that the Java program will define a method like the one shown below to perform all the steps needed to make one transition given as parameters the elements of a tuple of the form (q_k, a, a, b, L, R, R) that describes the transition.

```
public void applyDelta(
         int newState,
         char write1, char write2, char write3,
         char move1, char move2, char move3 ) {

   char underHead1, underHead2, underHead3;

   for( int p = 0; char[p] != '\ '; p++ ) {
      if ( tape1[p] is marked ) {
         tape1[p] = write1;
         if ( move1 = 'R' ) {
            underHead1 = tape1[p+1];
            tape1[p+1] = marked copy of tape1[p+1];
         }else{
            underHead1 = tapei[p-1];
            tape1[p-1] = marked copy of tape1[p-1];
         }
      }
      if ( tape2[p] is marked ) {
            ...
      }
   }
}
```

- This code is meant to be illustrative rather than correct or complete.

- The idea is that the method makes a pass through the elements of the arrays that represent the 3-tapes of the simulated machine looking for the positions of the tape heads. When it finds one, it updates the contents appropriately and, depending on whether the transition function says the head should move left or right, put a mark on the appropriate adjacent array element. It also remembers the newly marked letters using the variables underHead1, underHead2, and underHead3, since it will need to know their values to determine the simulated machine's next transition.

11. The technique (trick?) we will use to build a TM that can implement the same algorithm as this Java code is to use for our states tuples that can encode the values of the parameters and local variables used in the program.

    - For the N-tape version, we need for our state tuple to include a new state from the simulated machine's state set, a sequence of N symbols to be written by the three tape heads, a sequence of N directions the tape heads should move, and a sequence used to remember the N symbols under the new tape head positions.
    - With this in mind, we might use a state set of the form:

    $$Q' = Q \times \Gamma^N \times \{L, R\}^N \times (\Gamma \cup \{?\})^N \times ...$$

    - The intent here is that if $q' \in Q'$, then

    $q' = (\ q_k, W, M, U, ...)$ where
    $q_k \in Q$ is the state the simulated machine is entering,
    $W = (w_1, w_2, ...w_n)$ & $w_i \in \Gamma$ is the symbol the ith tape head should write
    $M = (m_1, m_2, ...m_n)$ & $m_i \in \{L, R\}$ is the direction the ith tape head should move
    $U = (u_1, u_2, ...u_n)$ & either $u_i \in \Gamma$ is the new symbol under the ith tape head or $u_i =$? if this symbol is not yet known

    - We left the ... in our description of this set of states to suggest that we could easily add more information. In particular, our machine will need a way to know whether it is just sweeping left to right looking for head positions or whether it has found one and is currently wiggling back a step to simulate one head moving left. We can easily keep track of such things by adding components to our state tuple.

12. Using these ideas, we can design a single tape TM to simulate any n-tape TM. Thus, adding extra tapes does not add any extra power.

Nondeterministic Turing Machines

(Click for video)

1. The next extension we will explore is the addition of nondeterminism to the Turing machine model.

   Definition: A nondeterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

   Q is a finite set of states,

   Σ is a finite input alphabet (not containing the blank symbol),

   Γ is a finite tape alphabet which is a superset of Σ including the blank symbol,

   $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{Left, Right\})$ is the transition function,

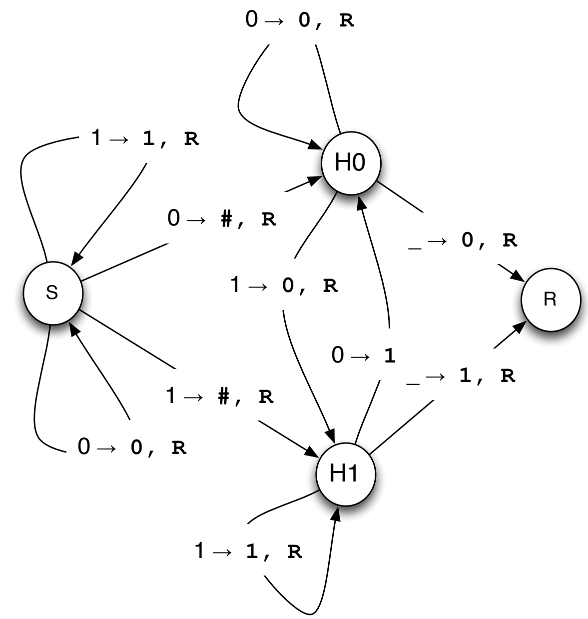
   q_0 is the start state,

   q_{accept} is the accept state, and

   $q_{reject} \neq q_{accept}$ is the reject state.

   We say that a nondeterministic TM accepts an input w if and only if there is some sequence of configurations in which each configuration yields the following configuration that starts with the initial configuration (q_0, ϵ, w) and ending with a configuration in q_{accept}.

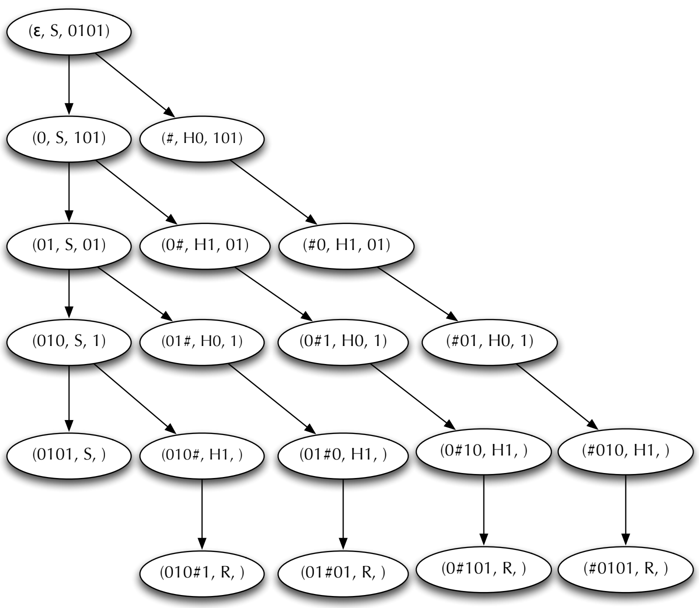
2. The diagram below shows how non-determinism could be used to simplify the task of converting an input of the form ww' into the form $w\#w'$ (for further processing by a machine like the one Sipser presents that recognizes $w\#w$).



- This machine sits in the state S moving right through the as and bs on the tape for as long as it feels like..
- It nondeterministically guesses that it is at the midpoint, writes a # in place of the character it was scanning and moved to state A or state B to remember the character that was replaced.
- It then bounces back and forth between states A and B, replacing each symbol on the tape by the preceding symbol while always ending up in the state that will remember the replaced symbol.
- When it finds the end of the tape, it writes the last preceding symbol over the space and moves to state R (for ready).
- Assuming it correctly guessed when to transition out of state S, the contents of the tape will now match the input with a # inserted in the middle. In particular, if state R of this machine is connected to a machine that moves the input head to the left and then behaves like Sipser's $w\#w$ machine, it will be possible to reach the accept state iff the original input was of the form ww.

3. We can visualize the possible executions of a nondeterministic TM as a possibly infinite tree in which each node is a configuration such that:
   - The root is the initial configuration,
   - Each configuration in the tree yields exactly the set of configurations that correspond to its children.

   For example, The machine shown above would have the computation tree shown below when applied to the input *abab*:



4. A nondeterministic TM recognizes a language L if the computation tree for a string w contains an accepting configuration as one of its leaves iff $w \in L$. A nondeterministic TM decides a language L if all of its computation trees are finite and the computation tree for a string w contains an accepting configuration as one of its leaves iff $w \in L$.

Deterministic TMs are as powerful as Nondeterministic TMs

(Click for video)

1. To show that nondeterminism does not increase the power of the Turing machine model we need to describe a general procedure by which given a nondeterministic TM

$$N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

   we can construct a deterministic TM

$$D = (Q', \Sigma, \Gamma', \delta', q_0', q_{accept}', q_{reject}')$$

   such that

$$L(N) = L(D)$$

   and D halts on w iff all of N's possible computations on w are finite (i.e., its computation tree is finite).

2. To do this, we will construct a deterministic machine, D, that will enumerate all possible configurations that would appear in the computation tree of the nondeterministic machine N in breadth first order. Eventually, our deterministic machine's tape will be filled with a long sequence of configurations. Configurations corresponding to nodes at the top of the computation tree will appear at the beginning of the tape. As the machine is allowed to compute longer and longer, configurations corresponding to deeper and deeper levels in the tree will be written on the end of the tape.

   The algorithm to follow is simple to express if we assume D has two tapes:
   - Convert input w into an initial configuration for N, (q_0, ϵ, w). This will all happen on tape 1. Tape 2 will still be empty.
   - Repeat (until you see an accept or reject configuration or there are no unexpanded configurations on tape 1):
     - Copy first unexpanded configuration from tape 1 to tape 2 (overwrite any previous configuration on tape 2).

- Expand the configuration on tape 2 by writing all configurations derivable from this configuration at the end of tape 1. This will require building knowledge of N's transition function into the transition function for D.

3. The tape alphabet Γ' that we use for D will have to contain all the symbols in Γ together with:

   - Some delimiter to serve in the role of the commas that separate the components of a configuration.
   - Symbols that can be used to encode all the states of N.
   - Two delimiters we can use to separate configurations from one another on our tape. One will be used in the section of tape containing configurations that have been expanded ($), the other will be used between configurations that still need to be expanded (#)..

4. The machine D, will clearly recognize the same language as N since it will find an accepting configuration if and only if one occurs in the computation tree.

5. If the computation tree is finite, D will eventually halt. Therefore, if N decides a language, D will decide the same language.