Announcements

1. Homework 7 due Friday.

2. Midterms will be graded this year!

n-tape Turing Machines

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 $C = \text{COPY}$.

- 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 $B = \text{BACKUP}$.

- 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 $M = \text{MATCH}$.

- Hopefully, you could imagine how having multiple tapes could simplify recognizing other languages:
  - For example, the other day we considered
    \[
    \{i\#x\#w_1\#w_2\#\ldots\#w_k \mid i, x, w_i \in \{0, 1\}^*, i \leq k, \ & x = w_i\}
    \]
  - 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!

Extra Tapes are Handy But...

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 $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where
- $Q$ is a finite set of states,
- $\Sigma$ is a finite input alphabet (not containing the blank symbol),
- $\Gamma$ is a finite tape alphabet which is a superset of $\Sigma$ including the blank symbol,
- $\delta : Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{\text{Left}, \text{Right}\}^n$ is the transition function,
2. Now, I would like to argue that a multi-tape TM is no more powerful than a single-tape TM. To do this, I 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

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

we can construct a 1-tape TM

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

such that

\[ L(M^1) = L(M^N) \]

and \( M^1 \) halts on \( w \) iff \( M^N \) halts on \( w \).

7. The basic idea is to 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 \( i \)th 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 symbols in \( \Sigma \). The initial steps performed by the simulator will be to scan its input from left to right replacing each \( x \) in the input with \( (x, \ldots, \cdot, \ldots, \cdot) \) with the spaces representing the contents of the other \( N-1 \) simulated tapes.

As a result, to simulate an \( N \)-tape machine with alphabet \( \Gamma \), we will use a single-tape machine with alphabet \( \Gamma' = \Sigma \cup (\Gamma \times \{', \epsilon \})^N \).
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/transition 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
\[
\delta(q, b, a, a) = (q_k, a, b, L, R, R)
\]
- 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 \( M^3 \)'s tape head is irrelevant since the primed symbols on the tape encode the positions of \( M^3 \)'s heads): 

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:

```java
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 tran-
position given as parameters the elements of a tuple of the form 
\((q_k, a, a, b, L, R, R)\) that describes the transition.

```java
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 = tape1[p-1];
                tape1[p-1] = marked copy of tape1[p-1];
            }
        }
        if ( tape2[p] is marked ) {
            ...
        }
    }
}
```

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:
  
  \[ Q' = Q \times \Gamma^N \times \{L, R\}^N \times (\Gamma \cup \{?\})^N \times \ldots \]

- The intent here is that if \( q' \in Q' \), then 
  
  \[ q' = ( q_k, W, M, U, \ldots ) \] where 
  - \( q_k \in Q \) is the state the simulated machine is entering,
  - \( W = (w_1, w_2, \ldots w_n) \) & \( w_i \in \Gamma \) is the symbol the \( i \)th 
    tape head should write 
  - \( M = (m_1, m_2, \ldots m_n) \) & \( m_i \in \{L, R\} \) is the direction the 
    \( i \)th tape head should move 
  - \( U = (u_1, u_2, \ldots u_n) \) & either \( u_i \in \Gamma \) is the new symbol 
    under the \( i \)th tape head or 
    \( u_i = ? \) if this symbol is not 
    yet known

- We left the \( \ldots \) 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 write 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.

- 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 under-

Head1, underHead2, and underHead3, since it will need to know 
their values to determine the simulated machine’s next transition.
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**

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 $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where
   
   - $Q$ is a finite set of states,
   - $\Sigma$ is a finite input alphabet (not containing the blank symbol),
   - $\Gamma$ is a finite tape alphabet which is a superset of $\Sigma$ 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,
   - $q_{\text{accept}}$ is the accept state, and
   - $q_{\text{reject}} \neq q_{\text{accept}}$ 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 $(q_0, \epsilon, w)$ and ending with a configuration in $q_{\text{accept}}$.

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 $abab$:

```
(ε, S, 0101)
(0, S, 101)
(#, H0, 101)
(01, S, 01)
(0#, H1, 01)
(#0, H1, 01)
(#01, H0, 1)
(0#1, H0, 1)
(010, S, 1)
(01#, H0, 1)
(0#10, H1, )
(01#0, H1, )
(0101, S, )
(010#, H1, )
(0#101, R, )
(01#01, R, )
(010#1, R, )
(0#101, R, )
```

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 $w \in L$. 