HW3CSDS343

1 Multi-core TM

Let's try to relate out multi-core turing machine to a k-tape turing machine. Our k-tape turing maachine has a theorem which any language L can be decided on a k-tape turing machine can also be decided on a normal turing machine.

Unlike the multi-core turing machine, our k-tape turing machine only processes one state at a time as opposed to one state for each head.

We can fix this issue by adding a new tape which will be a string of each state the tapes are in. That is the ith character will be the state for the ith tape. As such, we create states $q_1 read, q_2 read, \ldots, q_k read$ to read each of these states. Each of these read states move our machine to the corresponding $q_i s$ where i is the tape to be affected and s is the state for that tape. Given a $q_i s$ we apply the corresponding transition to our ith tape. On our state tape, we change the current symbol to match the state that our ith tape should be in now, and move to the state tape head to the right and go into the $q_{i+1} read$ state to read the state of the next tape. When $i = k$ we apply the correct transition to the k-tape then move our state head back to the start, checking if any state is q_{accept}, if any are we move our machine to q_{accept}.

Essentially, for each tape we assign it a state on our state tape and read off the tape moving left to right and applying the transitions accordingly.

This in all is $O(kn)$ where n is the number of steps the multi-core TM would take and k is the number of tapes we have.

Thus, we know that a regular TM can model a multi-core TM because a regular TM can do anything a k-tape turing machine can do and we just proved a k-tape turing machine can do anything a multi-core TM can do.

2 Broken TM (left move will move head to start)

Theorem: A broken tape machine can model a normal tape machine.

Sub-theorem: A broken tape machine can model a 2-tape broken machine.

Sub-proof:
First we will prove that a single broken tape TM can model a dual tape broken TM.

First we place a # at the end of our tape. Our single broken tape is modeled $T_1\#T_2$ where T_1, T_2 represents the contents of tape 1 and 2 respectively. Additionally, we are able to mark where the head is on each tape in order to keep track of the head of each tape. Additionally with the head marker we can mark the state to remember which state to run. A right move for each tape is trivial, as is a left move with tape 1. A left move on tape 2 is unmark the current character, reset to the left, ignore everything until we find the #, then we mark the next character as tape 2 head and mark the corresponding tape.

Now what if we wish to use the blanks to the right of either tape? This is trivial for tape 2 (keep moving right). For tape 1 however, we enter a copy state where we copy our tape 2 to after a new #. That is our tape now reads $T_1\#T_2\#T_2$. We can then replace everything from our first # to the second # as blanks to allow our tape 1 to use blanks.

Thus our machine works to read tape 1, follow any subprocesses needed (left move, extend blanks etc) and apply the correct state for tape 1. Then we follow the same process for tape 2, complete subprocesses and apply the correct state transformation.

As such, we are able to run 2 broken tapes on a single broken tape machine.

Sub-theorem: A 2-tape broken machine can model a regular TM.

Sub-proof:
For our 2-boken tape machine, we will have tape 1 be the input while tape 2 will be used to store our count. Every time we move one to the right on our input tape, we would incriment our count by one (adding a 0 to the end of tape 2). Once we go left on our tape (and go back to start) we remove the first 0 on our tape 2 (replace with a #). Then for each 0 on tape 2 we move one space to the right on tape 1 and stop once we reach a blank on tape 2. Since we have one less 0 then right moves we stop one space to the left of where we were on tape 1. This process will simulate one left move. Thus we would continue our correct state and run whichever state we need to.

More formally for each state that moves left qL_i, we create a substate qL_{iR} which counts the 0s to return back to its original state. Specifically, our qL_{iR} will ignore all # and move the head on tape 1 to the right when there are 0's on tape 2, once we reach a blank we run qL_i for tape one and adjust accordingly. Everytime we need to move left we reset both heads to the left, count the 0's on tape 2 (after removing the first 0) and for each 0 on tape 2 we move the head on tape 1 once to the right.

Thus since we can move left normally (while preserving state) we are able to mimic a normal turing machine on our 2 tape turing machine.

Proof:
Since we have proved both of our sub-theorems, we know that a broken turing machine will still work as a normal turing machine because we proved a broken turing machine can mimic a 2 broken tape TM which we proved is able to mimic a regular TM.

3 Prove the following language is recognizable and not decidable: $H_{TM} = \{< M, w > | M$ halts on inut $w\}$

Recognizability

Theorem

H_{TM} is Turing-recognizable.

Proof:

Create a "Universal Turing Machine" U

If we already agree that simulating a TM is trivial, then we create U which simply simulats the run of M on w and if M accepts or rejects then U accepts.

Otherwise we have the full formal proof as follows:

U takes as input the description of a TM M and a string w and it runs M on w.

U will have 3 tapes
Tape 1: $< M, w >$
Tape 2: Simulate M on w
Tape 3: Store current state of M

U on input x

1. Verify x is of the form $< M, w >$ with M a valid TM description and w a string of M's Σ
2. Copies w on tape 2, places the tape 2 head on left
3. Write q_0 to tape 3

At each step, U looks at state on tape 3 + symbol head on tape 2. Searches δ on tape 1 to find the correct transition step.

Once found, write the new state to tape 3 and change the symbol at tape 2 head and move the tape 2 head according to the δ

If tape 3 ever goes to q_{accept} or q_{reject} then U accepts.
Otherwise U runs forever.

Therefore U will recognize $H_{TM} = \{< M, w > |M$ halts on input $w\}$.

H_{TM} is not decidable

Theorem

H_{TM} is not Turing-decidable

Proof:

Proof by contradiction. Assume H_{TM} is decidable. Then $\exists M_{H_{TM}}$ that decides H_{TM}

Create a TM D

D on input x will either accept or run forever.

Within D we place $M_{H_{TM}}$.
D has a transformer which transforms out input x into the form $< x, x >$
D also has a function which maps the output of $M_{H_{TM}}$ such that if $M_{H_{TM}}$ accepts then we run forever (loop back into $M_{H_{TM}}$. If $M_{H_{TM}}$ rejects then D accepts.

What happens when we run D on input D.
D will convert out input into the form $< D, D >$ which we then run $M_{H_{TM}}$ on.

However we run into the following contradiction:
If $M_{H_{TM}}$ accepts $< D, D >$, then D must halt (accept) on input D.

- However if $M_{H_{TM}}$ accepts then D will run forever on input D, meaning that $M_{H_{TM}}$ should reject.
  This creates a contradiction because D cannot both accept and run forever on input D.

If $M_{H_{TM}}$ rejects $< D, D >$, then D will run forever on input D.

- However if $M_{H_{TM}}$ rejects then D will accept on input D, meaning that $M_{H_{TM}}$ should accept.
  This creates a contradiction because D cannot both accepts and run forever on input D.

Therefore $M_{H_{TM}}$ cannot exist by contradiction therefore H_{TM} is not decidable.