pp. 176-176-182. Variants of Turing Machines (Sec. 3.2)

- Remember: a language is Turing recognizable if some TM accepts it.
- Adding “features” may simplify programmability but DO NOT affect what a TM can compute.
  - Anything a “fancy” TM can compute, can be computed with a basic TM (perhaps with more complex set of δs.)
- **Option to “stay still”** (p. 176) (not move head)
  - δ:QxΓ → QxΓx {L, R, S} – S means stay still
  - δ(q,a) -> (r,b,S) can be replaced by 2 transitions of standard TM
    - δ(q,a) -> (r_1,b,R)
    - δ(r_1,x) -> (q,x,L) for all x in Γ
  - Thus no TM with “S” option can compute anything not computable by basic TM
  - But may be “faster” or easier to program
- **MultiTape TM** (p. 176)
  - Assume M has **k tapes**: all use same \( \Gamma \)
    - 1\(^{st} \) one as in basic machine (i.e. holds initial input)
    - Rest are initially all blank
  - Separate read/write head under each tape
    - That can be moved individually
  - \( \delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R,S\}^k \)
    - \( \delta(q,a_1, \ldots, a_k) = (r, b_1, \ldots, b_k, d_1, \ldots, d_k) \) means
      - If in state \( q \), and for all \( 1 \leq i \leq k \), tape i has \( a_i \) under its head
      - Then for all \( I \), change \( a_i \) to \( b_i \) on tape i
      - And for all \( I \), move tape i in direction \( d_i \)
  - **Proof**: assume M is a \( k \) tape TM \((Q,\Sigma,\Gamma,\delta,q_{\text{start}},q_{\text{accept}},q_{\text{reject}})\). Construct equivalent 1-tape TM \( S \)
    \((Q',\Sigma,\Gamma',\delta',q'_{\text{start}},q'_{\text{accept}},q'_{\text{reject}})\) as follows:
    - Assume starting tape is \( w_1 \ldots w_n \)
    - Add new characters to \( \Gamma' \)
      - For each \( x \) in \( \Gamma \), add a new symbol \( x' \) to \( \Gamma' \)
        - ‘ indicates a tape head is on that cell
        - Include a □’
      - Add a special symbol \# to \( \Gamma' \)
        - To mark start of a new simulated “tape”
• Add new initial states with transitions that do following
  • Insert a # onto left of tape, moving w right one place
  • Replace w₁ by w₁’
  • Write k-1 copies of #□’ to end of w
  • Write a final # at end
  • Resulting tape looks like #w₁’...wₙ#□’#□’ ... #□’#
• The ith “#” indicate the start of the ith tape
• The ith ‘ed symbol indicates the current position of
  the ith tape head
• (p. 177) Fig. 3.14 diagrams 3-tape example
• To simulate with S a single transition of M from state q
  • Sequentially try each rule from M that starts with q:
    • Move to the ith ‘ed symbol and compare to aᵢ
    • If we find a mismatch, quit and try next rule
    • If we have match on all aᵢs, go back to start of tape
      and go back to each ‘ed symbol in sequence
      • Replace by bᵢ
      • Move simulated tape head i by moving L, R, or S,
        and replace that symbol by its ‘ed version
    • On a move R where we hit a #
      • 1ˢᵗ move entire rest of string right one position
      • Then write a blank
• **B: Bidirectional Infinite Tape**
  - Tape goes on forever in both directions, not just right
  - First emulate on a 2-tape TM
    - Tape 1 is the right hand side of the double sided tape
    - Tape 2 is the left handed side of the double sided tape
    - Have a special # on start of both sides of tape
  - Two sets of states from B:
    - one where we are on right hand side of B’s tape
    - other where we are on left hand side of B’s tape
  - If in a right-side of tape state and move L, add additional states to check if new cell is cell 0
    - This is case where B has crossed the center of its tape, moving left
    - If so, switch to correct state on 2\textsuperscript{nd} tape
      - And whenever original state says move left, new transition says more right, and vice versa
  - If on 2\textsuperscript{nd} tape, and move right into a cell with a #
    - I.E. have crossed the center of the original tape and moving right
    - Move left to cell 0, switch to equivalent state that uses 1\textsuperscript{st} tape
  - Then emulate 2-tape machine on a basic TM
**S: TM with a Stack**

- \( \delta : Q \times \Gamma_1 \times \Gamma_2 \rightarrow Q \times \Gamma_1' \times \Gamma_2' \times \{L, R\} \)
  - \( \Gamma_2 \): tape characters
  - \( \Gamma_1 \): stack characters

- Having a stack is useful to simplify programming by supporting subroutines and recursive operations

- **Solution:** Simulate on a 2-tape machine
  - One tape is original tape
  - 2\(^{nd}\) tape is stack
  - \( \Gamma_1' \) and \( \Gamma_2' \) include duplicates of \( \Gamma_1 \) and \( \Gamma_2 \) i.e. \( a \) and \( a' \)
    where ‘ed symbols represent “top of stack”
  - Any push or pop to stack causes switch to states that modify just stack

- Then emulate 2-tape on single tape
• (p. 178) **NTM: NonDeterministic TMs**
  • \( \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R,\}) \)
    • Each \((q, a)\) can lead to one of a set of transitions
    • There are multiple choices for each state & tape symbol
    • If *any* of these choices lead to an accept state, then TM accepts its input
• (p. 179) **Theorem 3.16: Every nondeterministic TM N has equivalent deterministic TM D**
  • Solution: have D work thru each possible variation in N’s transitions sequentially
    • In a breadth-first exploration of *tree* of choices
      • Each node in tree is a configuration of N
      • Root node is initial configuration
      • Explore all possible set of choices at level \( k \) before trying any choices at level \( k+1 \)
        • If any choice leads to \( q_{\text{accept}} \), accept
        • If all choices lead to \( q_{\text{reject}} \), reject
      • Looping is still possible
• D has 3 tapes (see Fig. 3.17 on page 179)
  • Tape 1: Input tape – never changed
  • Tape 2: Simulation tape: copy of N’s tape having made one set of choices
  • Tape 3: Keeps track of which node in tree Tape 2 represents
    • Let $b = \text{size of largest set of possible choices from one transition}$
    • $\Gamma_3 = \{1, \ldots, b\}$
    • Eg. 431 on tape 3 means tape 2 represents
      • Having made 4$^{\text{nd}}$ choice at root,
      • Having made 3$^{\text{rd}}$ choice from above
      • Having made 1$^{\text{st}}$ choice from above
  • Computation as follows:
    • Copy tape 1 to 2
    • Initialize tape 3 to $\epsilon$
    • Use Tape 2 to simulate one branch of N’s tree
      • Before each step of N, consult next symbol on tape 3 to determine which choice to make
        • If accepting configuration found, enter accept state
• Replace string on tape 3 with next string in tree ordering and restart if any of following
  • No more symbols on tape 3
  • Simulation ended up “invalid”
  • Choice on tape is invalid
• D clearly computes anything N does but with 3 tapes
  • But a 3-tape TM can be simulated by a 1 tape TM
    • SLOWLY!!!
  • Thus N can be simulated by a basic 1-Tape TM!
• (p. 180) Corollary 3.18. A language is Turing-recognizable if some NTM recognizes it
  • Proof: all NTMs can be converted into a TM
• A NTM is a Decider if all branches halt
  • In proof of Theorem 3.16 we can modify simulation of N so that if N always halts then so does D.
  • Thus Corollary 3.19: L is decidable iff some NTM decides it
• (p. 180) An **Enumerator** of a language L is a TM with
  • A “printer” where each rule can also output a symbol
  • An initial blank “work tape”
  • A set of rules that uses work tape to generate all possible strings from a language
    • And write each string to the printer
• (p. 181) **Theorem 3.21** A language L is Turing-recognizable iff some enumerator can enumerate it.
  • If: assume TM E enumerates L, following TM M accepts it
    • Given a string w, M runs E from start
    • For each string that is output, compare it to w
    • If ever a match, accept it
    • All (and only) w’s from L will be accepted!
  • Only if: Assume TM M accepts L, construct E as follows:
    • Build an enumerator E’ for all strings in ∑*
    • Do the following for i=1, 2, ....
      • Run E’ to generate next string
      • For each output from E’ run M for exactly i steps
        • Guarantees we will stop
      • If accepted, print out string from E’
    • Equivalent logically to running parallel set of Ms, each running on a different string from ∑*
• **Summary of all this**
  
  • No computer can compute anything that basic TM cannot
  • With caveat of enough memory
  • Thus all computers compute *exactly* the same class of algorithms
  
  • Any reasonable programming language can be used to write a TM emulator
  • Thus any reasonable programming language can be compiled into any other reasonable language
  • Thus all programming languages describe *exactly* the same class of algorithms