Our next topic: finite automata and Turing machines

- abstract machines used to study theory of computation
- Combination of slides from textbook (Ch 11) and others
- Assignment 3: create, test automata using JFLAP

JFLAP

- The software we will use for Assignment 3 is called JFLAP
  - FLAP = Formal Language and Automata Package
  - Java program developed at Duke University
  - Download a jar file from jflap.org
  - Launch by double-clicking on the jar file icon, or from terminal:
    `% java -jar JFLAP.jar`

NOTE: The latest version (JFLAP 7) requires Java 6.

If you have Java 5 download an earlier version
(I downloaded JFLAP 6.4)

Motivation

- Why are we studying abstract models of computation this week?
  - textbook has it in Ch 11, at the end of the book
- Answer #1: natural transition from studying scalability and very hard problems
  - are there even harder problems than the $O(2^n)$ TSP?
  - are there problems that can’t be solved using computation?
  - for that matter, what is computation? what does it mean to “compute” something?
- Answer #2: introduces some topics related to machines
  - logic circuits, registers and memory, CPU organization

Chapter 11: Models of Computation

Invitation to Computer Science, Java Version, Third Edition

The text has sections on abstract machines to study theory of computation and Turing machines

I’ve added extra slides here to describe “finite automata"
What Is a Model?

- A simplified representation of a phenomenon
  - Captures the essence (important properties) of the real thing
  - Probably differs in scale from the real thing
  - Suppresses some of the details of the real thing
  - Lacks the full functionality of the real thing

What Is a Model? (continued)

- Models are an important way of studying physical and social phenomena, such as
  - Weather systems
  - Spread of epidemics
  - Chemical molecules
- Models can be used to
  - Predict the behavior of an existing system
  - Test a proposed design

Properties of a Computing Agent

- A computing agent must be able to
  - Accept input
  - Store information and retrieve it from memory
  - Take actions according to algorithm instructions
    - Choice of action depends on the present state of the computing agent and input item
  - Produce output

A Model of a Computing Agent

- A good model for the “computing agent” entity must
  - Capture the fundamental properties of a computing agent
  - Enable the exploration of the capabilities and limitations of computation in the most general sense

NOTE: models of computation are not abstractions of actual computers...
Finite Automata

- The Chomsky hierarchy is fundamental concept in theoretical computer science
- Abstract models of computation can be ranked according to their ability to compute answers to problems
  - finite automata
  - push-down automata
  - linear-bounded automata
  - Turing machines
- We will start with finite automata

States

- As an introduction to finite automata, consider how a vending machine works
- The machine remembers how much money you have inserted
- The state of the machine depends on the number and type of coins you feed it
  - “state” is an abstraction that, in this case, means “memory of past events”
  - initial state: no money inserted
  - final state: enough money inserted
  - intermediate states keep track of how much money has been inserted so far

States (cont’d)

- Other examples of systems with states:
  - light bulb (on/off)
  - doors or gates (open/closed)
  - elevator (moving/stationary)
- States are discrete
  - real doors may be half open, but in our model they are either open or closed
- States can be represented by symbols
  - 0 or 1 for on/off or open/closed
  - s₁, s₂, s₃ ... (stationary on floor i), u₁, u₂, ... (moving up from floor i), d₁, d₂, ... (moving down from floor i)

Controllers

- A practical application: control systems
- Goal is to design circuits that allow or prevent certain state changes
- Example: air lock on a space ship
  - can’t have both doors open at once
  - from a starting state (both doors closed, air in chamber) a valid sequence of state changes is:
    - open inner door
    - close inner door
    - evacuate chamber
    - open outer door

Note: same logic applies to gates on canals, doors in a supermarket, ...
State Diagrams

- We can describe the operation of a system with states by using a state diagram.
  - create a label for each of the $n$ states
  - also define labels for every action that can cause a state change
  - draw a directed graph with $n$ nodes, each representing a single state
  - add a transition from node $i$ to node $j$ if there is an action that can change the system from state $i$ to state $j$

- Example: vending machine (each item costs 25¢)

```
0¢ 5¢ 10¢ 15¢ 20¢ 25¢
Q   D   N   N   N   N
D   D   N   N   N   N
N = nickel, D = dime, Q = quarter
```

State Sequences

- If we’re using the state diagram to design a controller we need to consider messy real-world details.
  - e.g. what happens if a user deposits 3 dimes?
  - how do we connect inputs and outputs?
  - is there a reset button?

- For our model of computation we will just focus on sequences of states.

- Some valid sequences:
  - $Q$
  - $NNNNN$
  - $NDD$

  In this case “valid” means “sequence of coins that lead to a total of 25¢”

Finite Automata

- Our simple state machine can be given a precise mathematical description.

- A finite automaton (sometimes known as a “finite state machine”) is a system defined by:
  - a set of states $S$
  - a finite set of input symbols $\Sigma$
  - a state transition function $\delta$

- One of the states is designated the start state.

- One or more states are designated as final states.

Vending Machine Example

- Here is the mathematical description of the vending machine FA:

  $S = \{ 0¢, 5¢, 10¢, 15¢, 20¢, 25¢ \}$
  $\Sigma = \{ N, D, Q \}$
  $\delta(0¢, N) = 5¢$
  $\delta(0¢, D) = 10¢$
  $\delta(0¢, Q) = 25¢$
  $\delta(5¢, N) = 10¢$
  $\delta(5¢, D) = 15¢$
  $\delta(10¢, N) = 15¢$
  $\delta(10¢, D) = 20¢$
  $\delta(15¢, N) = 20¢$
  $\delta(15¢, D) = 25¢/0¢$
  $\delta(20¢, N) = 25¢$

Note that the function is represented by a table.

$0¢ 5¢ 10¢ 15¢ 20¢ 25¢$
$N   D   N   N   N   N$
$D   D   N   N   N   N$
$N = nickel, D = dime, Q = quarter$
Formal Language

✦ The mathematical description provides a way of defining what the automaton computes
✦ A sentence is a string of symbols from the input alphabet
  ✷ examples: NN, ND, DDD, QDQQQDNQN
✦ A machine accepts a sentence if the symbols in the sentence are labels on a path from the start state to one of the final states
✦ A language is a set of sentences
  ✷ L(M), the language defined by machine M, is the set of sentences accepted by M
✦ The function computed by an FA is a boolean function
  \[ f(S) = \begin{cases} 
  \text{true} & \text{if } S \in L(M) \\
  \text{false} & \text{if } S \notin L(M) 
\end{cases} \]

Simulating FAs with JFLAP

✦ We can use JFLAP to experiment with finite automata
✦ When the program starts you will see a menu with the types of machines you can create

For more information read the tutorial at jflap.org
(stop when you get to the section called “nondeterministic automata”)

Enter the States

✦ After you select “finite automaton” you will see a blank canvas
✦ Just select the state tool and click to place nodes on the canvas

Connect the Nodes

✦ Select the transition tool to add connections
✦ Drag from one state to the next
✦ When you release the mouse, a menu pops up where you can type in the label (the symbol that initiates the transition)

Unfortunately transitions are straight arrows
Use the select tool to drag nodes around so you can see all your transitions and their labels
If you make a mistake use the delete tool (click on a node or transition to erase it)
Designate Start and Accept States

- Right-click* a node to pop up an options menu
- You can
  - change the name of a node
  - designate a node as a start state
  - designate a node as a final state

* or control-click if you use a Mac

Run the Machine

- When your drawing is complete select “multiple run” from the Input menu
- A new panel is added to the display
- Enter a set of test strings in the Input column
- Click the “Run Inputs” button to run the FA on each of the test strings

You can also load test strings from a file -- make unit tests for your FA...

Results

JFLAP tells you which strings were accepted by your automaton

Select “Dismiss Tab” from the File menu to go back to editing...

Save Your Machine

- Use the Save command in the File menu to save your work
  - the file name will end with “.jff”
  - plan on uploading JFLAP files along with your writeup when you submit your homework for this week’s project
- You can also open a saved file to resume working on it later

Machine descriptions are saved in XML files (structured plain text)
Cycles

FAs commonly have cycles
- paths that lead back to a previous state
- A transition can lead back to the same state
- Example: an FA that recognizes strings of the form \(ab^*a\)

\[ a^5 = 5 \text{a's} \]
\[ a^n = n \text{a's (usually means } n > 1) \]
\[ a^* = 0 \text{ or more a's} \]

Multiple Labels

- We can also have two or more transitions leading between any two states
- This FA accepts strings that start with \(a\) or \(b\), contain any number of \(c\)'s, \(d\)'s, or \(e\)'s, and end with \(a\) or \(b\):

\[ \text{This looks like one transition but it is actually two -- they can be edited separately} \]

\[ \text{Unfortunately JFLAP doesn't have a notation for "not a"} \]

Finite Automata and Computation

- Our vending machine FA computes the same thing as the Java program shown at right
- This brings up an interesting question:
  - how “powerful” is an FA?
- Can we develop an FA for any boolean function written in Java?

Limits to Finite Automata

- An example of a function that cannot be computed by any FA:
  - the set of all strings of the form \(a^n b^n\)
  - strings that starts with \(n\ a\)'s and end with the same number of \(b\)'s
- It’s easy* to create machines if we have an upper limit for \(n\)
  - e.g. if \(n = 3\) create a machine for \(ab\), \(aabb\), or \(aaabbb\)
- But this specification is for input strings with an arbitrary value of \(n\)

*One of the projects on the homework...
Memory

✦ An automaton has no way to “count” the number of $a$’s it has seen
   ✷ if it had a counter it could increment for each $a$ and decrement for each $b$
✦ What it needs is a simple memory
✦ The next level up in the Chomsky hierarchy has such a memory
✦ A push-down automaton is basically an FA with a stack
✦ As it moves between states it is allowed to push or pop symbols from its stack

Extra credit: read about PDAs, implement the PDA for $a^n b^n$ in JFLAP, explain how it works...

Limits for Push Down Automata

✦ PDAs are more powerful than FAs
   ✷ i.e. there are functions that can be computed by a PDA that cannot be computed by any FA
✦ But there are many functions that PDAs cannot compute
   ✷ well-known (to computer scientists, at least): $a^n b^n c^n$
✦ To build a machine to compute this function we need to go up to a higher level in the Chomsky hierarchy

FAs and PDAs in Practice

✦ In spite of their limits these simple machines provide a valuable abstraction for a wide variety of useful tasks
✦ A common example: regular expressions
   ✷ a fundamental theorem: for any finite automaton $m$ we can write a regular expression that matches every string in $L(m)$
   ✷ but note that regular expressions in Perl, Ruby, and other modern programming languages have practical extensions that push them up a level in the Chomsky hierarchy
✦ Another example: sed (stream editor)
   ✷ standard tool in Unix/Linux
   ✷ apply simple editing operations to an input text, e.g. extract e-mail addresses, HTML links, ...
   ✷ based on matching patterns defined by a regular expression

The Turing Machine

✦ A Turing machine includes
   ❑ A state machine
   ❑ A (conceptual) tape that extends infinitely in both directions
     ❑ Holds the input to the Turing machine
     ❑ Serves as memory
     ❑ Is divided into cells
The Turing Machine (continued)

- Each cell contains one symbol
  - Symbols must come from a finite set of symbols called the alphabet
- Alphabet for a given Turing machine
  - Contains a special symbol \( b \) (for “blank”)
  - Usually contains the symbols 0 and 1
  - Sometimes contains additional symbols

---

The Turing Machine (continued)

- Input to the Turing machine
  - Expressed as a finite string of nonblank symbols from the alphabet
- Output from the Turing machine
  - Written on tape using the alphabet

---

The Turing Machine (continued)

- Each operation involves
  - Writing a symbol in the cell (replacing the symbol already there)
  - Going into a new state (could be same state)
  - Moving one cell left or right

---

Figure 11.2
A Turing Machine Configuration
The Turing Machine

- A shorthand notation for instructions
  - Five components
    - Current state
    - Current symbol
    - Next symbol
    - Next state
    - Direction of move
  - Form
    (current state, current symbol, next symbol, next state, direction of move)

The Turing Machine (continued)

- Conventions regarding the initial configuration when the machine begins
  - The start-up state will always be state 1
  - The machine will always be reading the leftmost nonblank cell on the tape

- The Turing machine has the required features for a computing agent

Turing Machine Example: A Bit Inverter

- A bit inverter Turing machine
  - Begins in state 1 on the leftmost nonblank cell
  - Inverts whatever the current symbol is by printing its opposite
  - Moves right while remaining in state 1

Figure 11.4
State Diagram for the Bit Inverter Machine
A Bit Inverter (continued)

- Program for a bit inverter machine
  - (1,0,1,1,R)
  - (1,1,0,1,R)

  “If you are in state 1 and see a 0 on the tape replace it with a 1, go to state 1, and move one position to the right”

Textbook’s notation: (Si,Ti,To,So,D)

Machines for Unary Incrementing

- Unary representation of numbers
  - Uses only one symbol: 1
  - Any unsigned whole number \( n \) is encoded by a sequence of \( n + 1 \) 1s

- An incrementer
  - A Turing machine that adds 1 to any number

Can you give a short one- or two-sentence explanation of how this program works?

Note: if there is no transition for a state/symbol combination the machine halts

Figure 11.6
State Diagram for Incrementer

Machines for Unary Incrementing (continued)

- A program for incrementer
  - (1,1,1,1,R)
  - (1,b,1,2,R)

- An alternative program for incrementer
  - (1,1,1,1,L)
  - (1,b,1,2,L)
Unary Incrementer in JFLAP

✦ Create states and transitions the same way you do for FAs
✦ Transition labels have 3 fields:
 ❖ input symbol (erased from tape)
 ❖ output symbol (replaces input symbol on tape)
 ❖ direction to move (L, R, or S)

Testing the Machine with JFLAP

✦ Select “step” from the input menu

Enter the initial contents of the tape and click OK

Don’t forget to specify starting and ending states

Testing with JFLAP (cont’d)

The current state of the machine is highlighted

The current location of the tape head is shown with an arrow and the tape symbol is highlighted

Click the ‘step’ button to single step through the program

A Unary Addition Machine

✦ A Turing machine can be written to add two numbers using unary representation
✦ Input tape holds two unary numbers separated by a blank character
A Unary Addition Machine (continued)

- The Turing machine program

(1, 1, b, 2, R)  
(2, 1, b, 3, R)  
(3, 1, 1, 3, R)  
(3, b, 1, 4, R)

A problem on this week’s assignment is to implement this machine in JFLAP

Question: how do we enter □ in the textbox for the input tape?

OK to use ‘b’ for blank and to have your machine recognize it instead of □

Balanced Parentheses

- Another example: decide whether a tape contains balanced parentheses
  - input: sequence of ( and ) symbols
  - every ( should have a matching )

If the expression is balanced write a 1 and halt
If not write a 0 and halt

Balanced Parentheses (cont’d)

- How it works:
  - scan right to look for a )
  - replace it with X
  - scan left to find matching (  
  - if ( found replace it with X and scan right again  
  - at end, scan left to make sure no ( remains

“inside out” replacement of parentheses with X

State R = “scan right for )”  
State L = “scan left for (”  

[ demo ]
The Church–Turing Thesis

- Church–Turing thesis
  - If there exists an algorithm to do a symbol manipulation task, then there exists a Turing machine to do that task

More commonly stated as “if a function is computable we can create a Turing machine to carry out the computation.”

- Two parts to writing a Turing machine for a symbol manipulation task
  - Encoding symbolic information as strings of 0s and 1s
  - Writing the Turing machine instructions to produce the encoded form of the output

Church, Turing, and Others

- 1931: Gödel states his famous “incompleteness theorem”, proves there are functions that are not computable
- 1936: Turing introduces his idea of an abstract computing agent, argues that it is powerful enough to compute any computable function
- 1936: Church introduces λ-calculus

Figure 11.9 Emulating an Algorithm by a Turing Machine
Church-Turing Thesis

• 1936 on: other models of computation are proposed, but
  ◦ no model is capable of computing a function that cannot be computable in λ-calculus or by a Turing machine
  ◦ any computation performed under any new model of computation can be performed in λ-calculus or by a Turing machine

The Church–Turing Thesis (continued)

- Turing machines define the limits of computability
- An uncomputable or unsolvable problem
  - A problem for which we can prove that no Turing machine exists to solve it

Unsolvable Problems

- The halting problem
  - Decide, given any collection of Turing machine instructions together with any initial tape contents, whether that Turing machine will ever halt if started on that tape

Unsolvable Problems (continued)

- To show that no Turing machine exists to solve the halting problem, use a proof by contradiction approach
  - Assume that a Turing machine exists that solves this problem
    - Show that this assumption leads to an impossible situation
The Barber Paradox

- Imagine a game like Sim City where the goal is to define rules that govern the behavior of each resident in a city.
- Suppose we want to build a city where:
  - only one resident can be the barber
  - the barber is a male
  - all males either shave every morning or go to the barber for a shave
  - the barber shaves only men who do not shave themselves
- The paradox: who shaves the barber?

These rules, taken together, define a goal that is not computable.

We cannot implement these rules in the Sim City rule language, or in Java, C++, Python, or any programming language.

Universal Turing Machines

- Turing based his claim that any computable function could be computed by a Turing machine on the idea of a universal Turing machine (UTM).
- A UTM is a machine that simulates the actions of another machine.
- The key idea:
  - the operation of a machine $T$ is completely defined by its state transition table.
  - the table can be written on a tape and read by another machine $U$.
- One can define a universal machine $U$ that can carry out the operations defined by $T$ and $T$'s input:

Universal Turing Machines (cont’d)

- Here is the first part of the input tape we would make if we want $U$ to simulate the actions of the machine that checks for balanced parentheses:

Universal Turing Machines (cont’d)

- Think about the implications of this idea.
- Before Turing’s paper, there was a widespread assumption that one would design a different computing machine for different functions.
  - “use the right tool for the job”
  - build one machine for computing trajectories of rockets, another for computing tables of logarithms, ...
- Turing’s “universal machine” makes this unnecessary:
  - build just one type of hardware: a UTM that simulates the actions of any TM.
  - the “machines” that compute trajectories, tables, etc are “software” that is fed into the UTM.

Turing’s 1936 paper included the state transition table for the UTM.
Instructions = Data

- The idea that a function can be represented as a string of symbols and manipulated like any other data is the very foundation of modern computer science.

Virtual Machines

- The idea that one computer can simulate the actions of another machine is still an important method used in implementing software.
- A virtual machine is a software program organized very much like a CPU chip:
  - it has arrays to represent instructions and memory (modern equivalent of the Turing machine’s tape)
  - the main loop fetches instructions and updates the memory array depending on what the instructions tell it to do.
- A widely known virtual machine: the Java Virtual Machine (JVM)
  - “applets” are small files containing JVM instructions
  - downloaded from the internet, executed by a JVM operating in a web browser.

The Halting Problem

- In his 1936 paper Turing gave detailed specifications for how a universal Turing machine could simulate the operation of any other Turing machine.
- He also took this one step further: he proved there are functions that cannot be computed by any Turing machine.
- One such function considered by Turing is known as the halting problem:
  - given a description of a machine $T$ and the contents of its tape, can we decide whether the machine will halt?
- There may be some special cases where we can analyze a program and say for sure whether it terminates.
- But the halting problem asks for a general procedure that applies to all Turing machines and all inputs.

Proof

- The proof that one cannot write an algorithm to decide if a Turing machine will halt is a proof by contradiction:
  - first assume that such a machine exists
  - then make a simple construction that modifies the machine
  - show the resulting machine leads to a paradox: there will be a statement about the modified machine that cannot be true if the original assumption is true.

- Are these statements true or false?
  - The statement on the other side of this page is false.
  - The statement on the other side of this page is true.
  - Neither: they are undecidable.
Proof (cont’d)

- We start by assuming there is a Turing machine named \( D \) that can decide whether another Turing machine will halt.
  - inputs to \( D \) are descriptions of a machine \( T \) and \( x \), the input tape for \( T \)
  - think of it as a function \( D(T,x) \) that outputs “yes” or “no”.
- However \( D \) is defined, it must have two final states, one that is entered if \( T \) halts and one that is entered if \( T \) does not halt.
  - the state transition diagram for \( D \) would look something like this:

\[ D \]

\[ \text{Yes} \rightarrow \text{No} \]

\( D \) = “decider”

If \( D \) has more than one of each type of halt state we can easily add new final states so the machine always ends up in one of these two states.

Proof (cont’d)

- For the next step of the proof we will design a new machine named \( E \).
  - \( E \) will use \( D \) and a new machine called \( C \) as “subroutines”
  - \( E \) will have only one input, which will be the description of a machine \( T \)
  - \( E \) first uses machine \( C \) to duplicate information on the tape
  - the tape now has two copies of \( T \) to pass to \( D \)
  - replace the halt state of \( D \) so that if \( D \) decides its input halts \( E \) will go into an infinite loop.

\[ C \rightarrow \text{"copier"} \]

\[ D \]

\[ \text{Yes} \rightarrow \text{No} \]

\( D \)

\[ E \rightarrow \text{"enigma"} \]

\[ \text{Copy of } T \]

Noncomputable Functions

- If the Church-Turing conjecture is true, the fact that no Turing machine can decide if another will halt means we cannot write a program, in any language, that will always decide if another program will halt.
- Some implications of this idea:
  - we cannot decide if a program will ever reach any given state, e.g. if it will ever execute any given statement or produce a certain output.
  - we cannot write a program that will tell us if any two functions are equivalent, i.e. that they will produce the same outputs given the same inputs.
- Note again that these are statements about programs in general:
  - we can reason about individual programs, e.g. prove that a specific program will terminate or reach a certain state
  - we cannot write a general purpose “halt-checker” that will work on any program.
Unsolvable Problems (continued)

- Practical consequences of other unsolvable problems related to the halting problem
  
  - No program can be written to decide whether any given program always stops eventually, no matter what the input
  
  - No program can be written to decide whether any two programs are equivalent (will produce the same output for all inputs)