CSCE 222
Discrete Structures

Turing Machines, Computability, and NP-Completeness

Dr. Tim McGuire

Grateful acknowledgement to Dr. Richard Anderson, University of Washington, for some of the material upon which these notes are adapted.

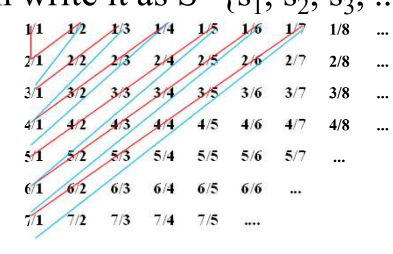
Based on Chapter 13 of Rosen
Discrete Mathematics and its Applications

Modeling Computation

- We learned earlier the concept of an algorithm.
  - A description of a computational procedure.
- Now, how can we model the computer itself, and what it is doing when it carries out an algorithm?
  - For this, we want to model the abstract process of *computation* itself.

Remember Cardinality??

- Cardinality
- A set S is *countable* iff we can write it as S={s_1, s_2, s_3, ...} indexed by ℕ
- Set of rationals is countable
  - "dovetailing"



- Σ* is countable
  - {0,1}* = {λ,0,1,00,01,10,11,000,001,010,011,100,101,...}
- Set of all (C++, Java, etc.) programs is countable

4

However, …

- The set of real numbers is not countable
  - “diagonalization”

		1	2	3	4	5	6	7	8	9	...
r_1 D=	0. 0.	5^{1}	0	0	0	0	0	0	0	...	...
r_2	0.	3	3^{5}	3	3	3	3	3	3	...	...
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	......	

Therefore,

- There exist functions that cannot be computed by any program
  - The set of all functions $f : \mathbb{N} \to \{0,1,...,9\}$ is not countable
  - The set of all (Java/C/C++) programs is countable
  - So there are simply more functions than programs

Do we care?

- Are any of these functions ones that we would actually want to compute?
  - The argument does not even give any example of something that can't be done, it just says that such an example exists
- We haven't used much of anything about what computers (programs or people) can do
  - Once we figure that out, we'll be able to show that some of these functions are really important

7

Turing Machines

Church-Turing Thesis

Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

- Evidence
  - Intuitive justification
  - Huge numbers of equivalent models to TM's based on radically different ideas

8

Components of Turing's Intuitive Model of Computers

- Finite Control
  - Brain/CPU that has only a finite # of possible "states of mind"
- Recording medium
  - An unlimited supply of blank "scratch paper" on which to write & read symbols, each chosen from a finite set of possibilities
  - Input also supplied on the scratch paper
- Focus of attention
  - Finite control can only focus on a small portion of the recording medium at once
  - Focus of attention can only shift a small amount at a time

9

What is a Turing Machine?

Steam-powered Turing Machine
Artist: Sieg Hall, 1987

10

What is a Turing Machine?

- Recording Medium
  - An infinite read/write "tape" marked off into cells
  - Each cell can store one symbol or be "blank"
  - Tape is initially all blank except a few cells of the tape containing the input string
  - Read/write head can scan one cell of the tape - starts on input
- In each step, a Turing Machine
  - Reads the currently scanned symbol
  - Based on state of mind and scanned symbol
    - Overwrites symbol in scanned cell
    - Moves read/write head left or right one cell
    - Changes to a new state
- Each Turing Machine is specified by its finite set of rules

Sample Turing Machine

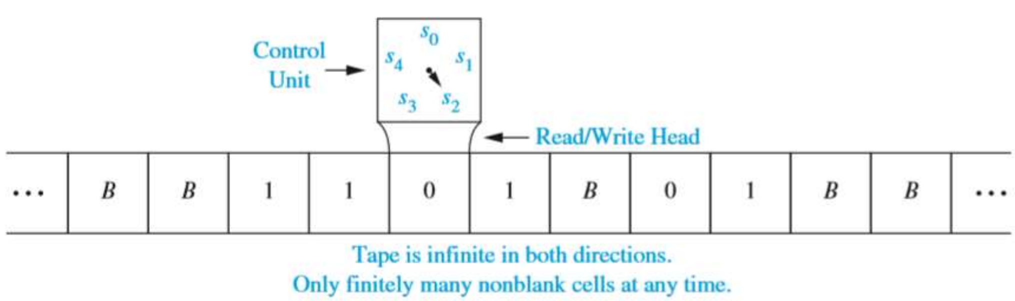


FIGURE 1 **A Representation of a Turing Machine.**

What is a Turing Machine?

13

Turing Machine ≡ Ideal Java/C Program

- Ideal C/C++/Java programs
  - Just like the C/C++/Java you're used to programming with, except you never run out of memory
    - constructor methods always succeed
    - **malloc** never fails
- Equivalent to Turing machines except a lot easier to program !
  - Turing machine definition is useful for breaking computation down into simplest steps
  - We only care about high level so we use programs

14

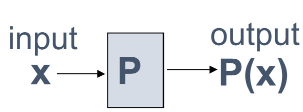
Turing's idea: Machines as data

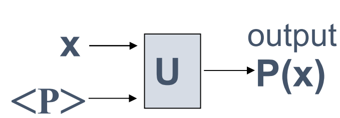
- Original Turing machine definition
  - A different "machine" **M** for each task
  - Each machine **M** is defined by a finite set of possible operations on finite set of symbols
    - **M** has a finite description as a sequence of symbols, its "code"
- You already are used to this idea:
  - We'll write <**P**> for the code of program **P**
  - i.e. <**P**> is the program text as a sequence of ASCII symbols and **P** is what actually executes

15

Turing's Idea: A Universal Turing Machine

- A Turing machine interpreter **U**
  - On input <**P**> and its input **x**, **U** outputs the same thing as **P** does on input **x**
  - At each step it decodes which operation **P** would have performed and simulates it.
- One Turing machine is enough
  - Basis for modern stored-program computer
    - Von Neumann studied Turing's UTM design





16

Halting Problem

See also Module 7, Algorithms for another look at the Halting problem.

- Given: the code of a program **P** and an input **x** for **P**, i.e. given **(<P>,x)**
- Output: **1** if **P** halts on input **x**
  0 if **P** does not halt on input **x**

Theorem (Turing): There is no program that solves the halting problem "The halting problem is undecidable"

17

Proof by contradiction

- Suppose that **H** is a Turing machine that solves the Halting problem

  Function **D(x)**:
  - if **H(x,x)=1** then
    - **while** (true); /* loop forever */
  - else
    - **no-op**; /* do nothing and halt */
  - endif

- What does **D** do on input **<D>**?
  - Does it halt?

18

Does D halt on input <D>?

Function D(x):

- if H(x,x)=1 then
  - **while** (true); /* loop forever */
- else
  - **no-op**; /* do nothing and halt */
- endif

D halts on input <**D**>

⇔ **H** outputs **1** on input (<**D**>,<**D**>)

[since **H** solves the halting problem and so **H**(<**D**>,**x**) outputs **1** iff **D** halts on input **x**]

⇔ **D** runs forever on input <**D**>

[since **D** goes into an infinite loop on **x** iff **H**(**x**,**x**)=**1**]

19

That's it!

- This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have

20

SCOOPING THE LOOP SNOOPER

A proof that the Halting Problem is undecidable

by Geoffrey K. Pullum (U. Edinburgh)

No general procedure for bug checks succeeds.
Now, I won't just assert that, I'll show where it leads:
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called *P*
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and *P* gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs...

21

SCOOPING THE LOOP SNOOPER

...
Here's the trick that I'll use -- and it's simple to do.
I'll define a procedure, which I will call *Q*,
that will use *P*'s predictions of halting success
to stir up a terrible logical mess.
...
And this program called *Q* wouldn't stay on the shelf;
I would ask it to forecast its run on *itself*.
When it reads its own source code, just what will it do?
What's the looping behavior of *Q* run on *Q*?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

22

SCOOPING THE LOOP SNOOPER

I've created a paradox, neat as can be —
and simply by using your putative *P*.
When you posited *P* you stepped into a snare;
Your assumption has led you right into my lair...

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A *reductio*: There cannot possibly be
a procedure that acts like the mythical *P*.
...

Full poem at:

www.lel.ed.ac.uk/~gpullum/loopsnoop.html

23

Finally,

THE P VS. NP QUESTION

The $1M question

- The Clay Mathematics Institute
- Millennium Prize Problems

- Is P = NP?
  The most important open problem in computer science

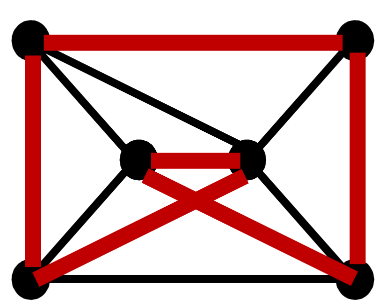
The P versus NP problem (informally)

Is proving a theorem much more difficult than checking the proof of a theorem?

Let's start at the beginning…

Hamilton Cycle

Given a graph G = (V,E), a cycle that visits all the nodes exactly once



The Problem “HAM”

Input: Graph G = (V,E)

Output: YES if G has a Hamilton cycle

NO if G has no Hamilton cycle

Satisfiability problem SAT

- **Input:** conjunctive normal form with *n* variables, $x_1, x_2, \ldots, x_n$.
- **Problem:** find an assignment of $x_1, x_2, \ldots, x_n$ (setting each x_i to be 0 or 1) such that the formula is true (satisfied).
- **Example:** conjunctive normal form is

  (x1 OR NOT x2) AND (NOT x1 OR x3).
- The formula is true for *assignment*

  x1=1, x2=0, x3=1.

Note: for *n* Boolean variables, there are 2^n assignments.

•Testing if formula=1 can be done in polynomial time for any given assignment.

•Given an assignment that satisfies formula=1 is hard.

30

Decision Versus Search Problems

Decision Problem

YES/NO

Does G have a Hamilton cycle?

Search Problem

Find a Hamilton cycle in G if one exists, else return NO

Decision/Search Problems

We'll look at decision problems because they have almost the same (asymptotically) complexity as their search counterparts

Polynomial Time and The Class "P" of Decision Problems

What is an efficient algorithm?

Is an O(n) algorithm efficient?

How about O(n log n)?

$O(n^2)$?

$O(n^{10})$?

$O(n^{\log n})$?

$O(2^n)$?

$O(n!)$?

polynomial time

$O(n^c)$ for some constant c

non-polynomial time

We consider **non-polynomial** time algorithms to be inefficient.

And hence a **necessary** condition for an algorithm to be efficient is that it should run in polynomial-time.

Asking for a polynomial-time algorithm for a problem sets a (very) low bar when asking for efficient algorithms.

The question is: can we achieve even this?

Class P and Class NP

- Class P contains those problems that are solvable in polynomial time.
  - They are problems that can be solved in $O(n^k)$ time, where n is the input size and k is a constant.
- Class NP consists of those problem that are *verifiable* in polynomial time.
- What we mean here is that if we were somehow given a solution, then we can verify that the solution is correct in time polynomial in the input size to the problem.
- Example: Hamilton Circuit: given an order of the n distinct vertices $(v_1, v_2, \ldots, v_n)$, we can test if (v_i, v_{i+1}) is an edge in G for $i=1, 2, \ldots, n-1$ and (v_n, v_1) is an edge in G in time $O(n)$ (polynomial in the input size).

37

Class P and Class NP

- Based on definitions, P⊆NP.
- If we can design a polynomial time algorithm for problem A, then problem A is in P.
- However, if we have not been able to design a polynomial time algorithm for problem A, then there are two possibilities:

1. polynomial time algorithm does not exist for problem A **or**
2. we are not smart.

Open problem: P≠NP?

Clay $1 million prize.

38

Why Care?

NP Contains Lots of Problems We Don’t Know to be in P

- Classroom Scheduling
- Packing objects into bins
- Scheduling jobs on machines
- Finding cheap tours visiting a subset of cities
- Allocating variables to registers
- Finding good packet routings in networks
- Decryption
- …

OK, OK, I care.
But Where Do I Begin?

We know P $\subseteq$ NP. How could we prove that NP $\subseteq$ P?

I would have to show that every set in NP has a polynomial time algorithm…

How do I do that?
It may take forever!
Also, what if I forgot one of the sets in NP?

We can describe one problem L in NP, such that if this problem L is in P, then NP $\subseteq$ P.

It is a problem that can capture all other problems in NP.

Polynomial-Time Reductions

Suppose we have a black box (an algorithm) that could solve instances of a problem X; If we give the input of an instance of X, then in a single step, the black box will return the correct answer.

Question:

Can arbitrary instances of problem Y be solved using polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?

If yes, then Y is **polynomial-time reducible** to X.

The "Hardest" Set in NP

NP-Complete

- A problem X is NP-complete if it is in NP and any problem Y in NP has a polynomial time reduction to X.
  - it is the hardest problem in NP
  - If an NP-complete problem can be solved in polynomial time, then any problem in class NP can be solved in polynomial time.
- The first NPC problem is *Satisfiability* probelm
  - Proved by Cook in 1971 and obtains the Turing Award for this work

46

Satisfiability problem

- **Input:** conjunctive normal form with *n* variables, $x_1, x_2, \ldots, x_n$.
- **Problem:** find an assignment of $x_1, x_2, \ldots, x_n$ (setting each x_i to be 0 or 1) such that the formula is true (satisfied).
- **Example:** conjunctive normal form is
- (x1 OR NOT x2) AND (NOT x1 OR x3).
- The formula is true for *assignment*
- x1=1, x2=0, x3=1.
- **Note:** for *n* Boolean variables, there are 2^n assignments.
- Testing if formula=1 can be done in polynomial time for any given assignment.
- Given an assignment that satisfies formula=1 is hard.

47

The First NP-complete Problem

- Theorem: Satisfiability problem is NP-complete.
  - It is the first NP-complete problem.
  - S. A. Cook in 1971 http://en.wikipedia.org/wiki/Stephen_Cook
  - Won Turing prize for his work.
- Significance:
  - If Satisfiability problem can be solved in polynomial time, then ALL problems in class NP can be solved in polynomial time.
  - If you want to solve P≠NP, then you should work on NP-Complete problems such as satisfiability problem.
  - We can use the first NPC problem, Satisfiability problem, to show that other problems are also NP-complete.

48

How to show that a problem is NP-Complete?

- To show that problem A is NP-complete, we can
  - First find a problem B that has been proved to be NP-complete.
  - Show that if Problem A can be solved in polynomial time, then problem B can also be solved in polynomial time.
- That is, to give a polynomial time reduction from B to A.
- ***Remark:*** *Since a NP-Complete problem,* problem B*, is the hardest in class NP, problem A is also the hardest*

Hamilton circuit and Longest Simple Path

- **Hamilton circuit :** a circuit uses every vertex of the graph exactly once except for the last vertex, which duplicates the first vertex.
- It was shown to be NP-complete.
- **Longest Simple Path:**
- Input: $V=\{v_1, v_2, ..., v_n\}$ be a set of nodes in a graph and $d(v_i, v_j)$ the distance between v_i and v_j, find a longest simple path from u to v .
- **Theorem: The longest simple path problem is NP-complete.**

Theorem: The longest simple path (LSP) problem is NP-complete.

Proof:
Hamilton Circuit Problem (HC): Given a graph G=(V, E), find a Hamilton Circuit.
We want to show that if we can solve the longest simple path problem in polynomial time, then we can also solve the Hamilton circuit problem in polynomial time.
Design a polynomial time algorithm to solve HC by using an algorithm for LSP.
Step 0: Set the length of each edge in G to be 1
Step 1: for each edge $(u, v) \in E$ **do**
find the longest simple path P from u to v in G.
Step 2: **if** the length of P is n-1 **then** by adding edge (u, v) we obtain an Hamilton circuit in G.
Step 3: **if** no Hamilton circuit is found for every (u, v) **then**
print "no Hamilton circuit exists"

Conclusion:
- if LSP can be solved in polynomial time, then HC can also be solved in polynomial.
- Since HC was proved to be NP-complete, LSP is also NP-complete.

Some basic NP-complete problems

- **3-Satisfiability :** Each clause contains at most three variavles or their negations.
- **Vertex Cover:** Given a graph G=(V, E), find a subset V' of V such that for each edge (u, v) in E, at least one of u and v is in V' and the size of V' is minimized.
- **Hamilton Circuit:** (definition was given before)
- History: Satisfiability→3-Satisfiability→vertex cover→Hamilton circuit.
- Those proofs are very hard.


That's all Folks!