CISC 4090
Theory of Computation

Complexity

Professor Daniel Leeds
dleeds@fordham.edu
JMH 332

Computability

Are we guaranteed to get an answer?

Complexity

How long do we have to wait for an answer? (Ch7)

How much resources do we need to find the answer? (Ch8)

Time complexity example: Deciding $\{0^k1^k | k\geq 0\}$

Computation steps:

- Scan left to right and confirm no 0's after 1's **2n steps**
- Repeat if both 0s and 1 left on tape $\frac{n}{2}$ **steps**
  - Scan across tape removing a single 0 and a single 1 **4n (or n) steps**
- If neither 0's or 1's remain on tape, accept; otherwise, reject **(n steps)**

Total steps: $2n + 4n \times n/2 + n = 2n^2+3n \rightarrow O(n^2)$

Characterizing run-time

If TM M halts on all inputs, there exists f: N->N
f(n) = max # steps on any input of length n

- "M runs in time f(n)"
- "M is an f(n) time Turing machine"

Asymptotic analysis – "Big O" and "Small O"

Assess runtime as input grows large

- Only consider highest-order term
- Ignore constant co-efficients

Example: $f(n) = 5n^4+3n^2+10n+5$

$f(n)=O(n^4)$

7

Big-O Defined

$f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every $n \geq n_0$

- $f(n) \leq c\, g(n)$
- $g(n)$ is "asymptotic upper bound" for $f(n)$

Big-O does not require the upper bound to be "tight"

$f(n) = 5n^4+3n^2+10n+5 \rightarrow O(n^4)$, $O(n^5)$, $O(2^n)$

8

Beyond polynomial

Exponential bounds, like $O(2^n)$, much bigger

Logarithmic bounds, like $O(\log n)$, much smaller

$O(\log_2 n) = O(\log_{10} n) = O(\ln n)$ – only constant difference

9

Small-O defined

$f(n) = o(g(n))$ if for any real number $c>0$, n_0 exists such that for every $n \geq n_0$

- $f(n) < c\, g(n)$

In other words:

$f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) = 5n^4+3n^2+10n+5 \rightarrow \cancel{o(n^4)}$, $o(n^5)$, $o(2^n)$

10

Time complexity class

Let t: N->R^+ be a function

TIME(t(n)) is collection of all languages decidable by an O(t(n)) time Turing machine

TIME(t(n)) is a time complexity class

Tighter bound for $\{0^k1^k\}$ on TM

Page 280 of text

Scan across tape, reject if 0 found right of 1

Repeat as long as some 0s and 1s remain

- Scan across tape and confirm total number of 0s and 1s is even
- Scan again, crossing off every other 0 starting with the first 0 and every other 1, starting with first 1

If no 0s and no 1s left, accept; else reject

Computing the tighter bound

Each step cuts input size in half.
Number of times to cut n in half: $\log_2 n$

Scan across tape, reject if 0 found right of 1 2n steps

Repeat as long as some 0s and 1s remain $\log_2$ n loops

- Scan across tape and confirm total number of 0s and 1s is even 2n steps
- Scan again, crossing off every other 0 starting with the first 0 and every other 1, starting with first 1
  2n steps

If no 0s and no 1s left, accept; else reject n steps

$2n + \log_2 n\,(2n+2n) + 2n = 4n + 4\,n \log_2 n$

$= \mathbf{O(n \log n)}$

Complexity on 2-tape Turing machine **computed**

Scan across tape and reject if 0 right of 1 n

Scan across 0s on tape 1, writing each onto tape 2 n

Scan across the 1s on tape 1, removing a 0 from tape 2 for each 1 k

If run out of 0s while still reading 1s, reject n-k

If 0s left after 1s finished, reject

If no 0s left, accept

3n = **O(n)**

Relationship between Single and Multi-tape TM

Theorem: Let t(n) be function, where t(n)≥n. Then every t(n) time multitape TM has equivalent $O(t^2(n))$ time single-tape TM

Convert any multi-tape TM to single-tape TM

Active part of tape t(n); make t(n) traversals through tape: $t^2(n)$

18

Relationship between Single and Multi-tape TM

Theorem: Let t(n) be function, where t(n)≥n. Then every t(n) time multitape TM has equivalent $O(t^2(n))$ time single-tape TM

Convert any multi-tape TM M to single-tape TM S

Across full runtime, M makes visits at most t(n) tape locations

In S simulation, include t(n) locations from each of the *k* tapes

For each S step, need to read/write each tape – k t(n) steps to traverse all tapes in one direction

S takes t(n) computation loops, one for each step of M

O(t(n)) O(t(n)) = $\mathbf{O(t^2(n))}$

19

Relationship between DTM and NDTM

Let N be NDTM that is a decider. Run time of N is max number of steps that N uses on any branch of its computation on input of length n

Does not correspond to real-world computer

- Except maybe a quantum computer!

20

NDTM -> DTM

If NDTM N decides language A in t(n) steps, can construct DTM D to decide A in $O(b^{t(n)})$ steps, where b is the maximum number of possible branches for a state-input pair.

D must simulate each branch of N, using breadth first search

At each step of computation, N chooses one branch down tree of possible branches

If b possible branches taken at each step, and there are t(n) steps, there are a total of $b^{t(n)}$ possible terminal points – exponential!!!

21

Polynomial time

Difference between polynomial times considered small compared to exponential time

Big picture:
- Exponential: brute force trying every solution to see what fits
- Polynomial: more efficient computation

"Reasonable" computational models are polynomial-time equivalent

Class P

P is the class of languages decidable in polynomial time on a deterministic single-tape Turing Machine

$$\mathrm{P} = \cup_k \ \mathrm{TIME}(\mathrm{n}^\mathrm{k})$$

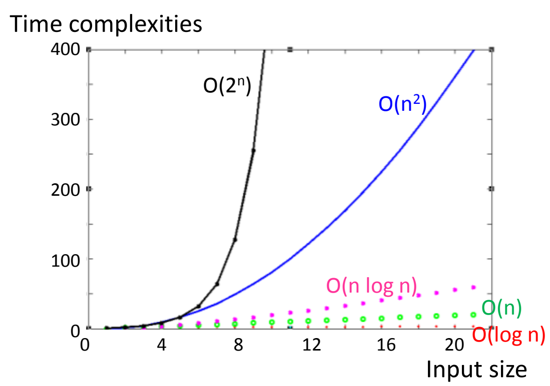
P is class of problems realistically solvable on a computer*

Problems in P

To show an algorithm belongs to P, we need to:
- Provide polynomial upper bound on number of stages
- Examine each stage to ensure it can run in polynomial time

Polynomial composed with polynomial -> polynomial

Address input encoding size:
- Graphs as list of nodes and edges, or adjacency matrix
- Binary encoding of integers



Example: Path problem

Is there a path from s to t in graph G?

- PATH={<G,s,t> | G is directed graph with directed path from s to t}

Brute force

- If G has m nodes, path cannot be more than m
- Upper bound on possible paths m^m
- Try each "path" one by one for legality and for linking s-to-t

26

Example: Path problem

Is there a path from s to t in graph G?

- PATH={<G,s,t> | G is directed graph with directed path from s to t}

Breadth first search

- Place mark on node s
- Repeat until no new nodes marked
  - Scan all edges; If edge (a,b) found from marked node a to unmarked b, mark b
- If t is marked, accept; Otherwise, reject

27

Path – Breadth first search complexity

- Place mark on node s
- Repeat until no new nodes marked
  - Scan all edges; If edge (a,b) found from marked node a to unmarked b, mark b
- If t is marked, accept; Otherwise, reject

In class I mistakenly claimed $O(n^3)$ complexity; $O(n^4)$ is a safer lower bound

Measure complexity based on number of nodes n

Place mark on node s <- n steps (find node s in list of m nodes)

Repeat until no new nodes marked <- n steps (if only mark one new node per loop)

For each edge (a,b) with a already marked, mark b also <- $O(n^3)$

(n^2 max total edges, for each edge,
search for a to see if marked (n steps), then mark b in list if needed (n steps)

In total: $n^2 x(n+n) = 2n^3$

If t marked, accept; else, reject <- n steps (find node t in list of n nodes)

In total: f(n) $n+nx2n^3+n = 2m^4+2n =$ **$O(n^4)$ – POLYNOMIAL!**

28

Example: RELPRIME

Two numbers are relatively prime of 1 is the largest number that evenly divides them both

- 10 and 21 are relatively prime
- 10 and 22 are not relatively prime

Solution: search all divisors from 2 until min(x,y)/2

- min(x,y)/2 numbers tried, min(x,y)/2 steps
- size of input n = length of binary encoding = $\log_2(\max(x,y))$
- 2^n steps – exponential complexity!

29

RELPRIME – faster solution

"Euclidean algorithm"

E = On input <x,y>
- Repeat until y=0
  - Assign x <- x mod y
  - Exchange x and y
- Output x

R = On input <x,y>
- Run E on <x,y>
- If result is 1, accept; Otherwise, reject

30

Simulating the Euclidean algorithm

x=10 y=21

x=10	y=21	MOD
x=21	y=10	SWAP
x=1	y=10	MOD
x=10	y=1	SWAP
x=0	y=1	MOD
x=1	y=0	SWAP

x=1 when y=0, so original numbers relatively prime

31

Euclidean Algorithm – complexity

x = x mod y <- new x always less than y

- If old x is twice y or more, new x will be cut at least in half
- If old x between y and 2y, new x will be cut at least in half
  - new x = x-y

Number of loops: $2\log_2(\max(x,y))$
Length of input (in binary): $\log_2(x)+\log_2(y)=O(\log_2(\max(x,y)))$
Number of loops: O(n)

32

Hamiltonian Path Example

Hamiltonian path in directed graph G: directed path that goes through each node exactly once

HAMPATH = { <G,s,t> | G is directed graph with Hamiltonian path from s to t}

Brute force
- List all possible paths and confirm if it's a valid path
- If m nodes, m! paths – exponential

33

Polynomial Verifiability – e.g., HAMPATH

- If given an answer, can determine if it is correct in polytime
- Path at most m long, graph has at most m^2 edges

Polynomial Verifiability – e.g., HAMPATH

- If given an answer, can determine if it is correct in polytime
- Path at most m long, graph has at most m^2 edges

For each proposed edge in path O(m)
- Search list of edges in G to find/not-find path edge $O(m^2)$

Total complexity: $O(m^3)$

Verifier definition

Verifier for language A is algorithm V where:
- A={w | V accepts <w,c> for some string c}
- Verifiers uses extra information in the "certificate" c to verify string w is in A

A is polytime verifiable if it has a polytime verifier
- Complexity measured in terms of w

NP definition

NP is class of languages that have polytime verifiers

NP <- Nondeterministic Polynomial time

HAMPATH c: proposed m-1 step path along nodes in graph G.

For HAMPATH
- Write list of m nodes, nondeterministically selected
- Check for repeats in list; if repeats, reject
- Check if s=$node_1$ and t=$node_m$; if either fails, reject
- For each i between 1 and m-1, check ($node_i$,$node_{i+1}$) is in G. If not, reject; else except.

NTIME and Class NP

NTIME(t(n)) = {L | L is language decided by O(t(n)) time nondeterministic Turing machine}

$\text{NP} = \cup_k \ \text{NTIME}(\text{n}^\text{k})$

NP Example: CLIQUE

A clique in an undirected graph is a subgraph where every two nodes are connected by an edge

- A k-clique is a clique containing k nodes

CLIQUE: {<G,k> | G is an undirected graph with a k-clique}

- k is a parameter. Determining clique for certain fixed k is in P

Prove CLIQUE∈NP

Just prove clique is certificate

Here is a verifier for V for CLIQUE

- V = On input <<G,k>, c> (m nodes)
  - Test if c contains k nodes, all in G O(m)
  - Test whether G contains all edges connecting all nodes in c
    For each node pair in c, check if matching pair in G
    $O(m^2)xO(m^2) \rightarrow O(m^4)$
  - If both pass, accept; else, reject

This is a polytime algorithm! $O(m^4)$

NTM method:

- Nondeterministically generate c
- Test c with V

Another NP Example: Subset-Sum

- SUBSET-SUM problem: given a list of numbers $S=\{x_1,x_2,\ldots,x_n\}$, determine if a subset of numbers add to the target t
- SUBSET-SUM={<S,t>|$S=\{x_1,\ldots,x_n\}$ and some $\{y_1,\ldots,y_k\}\subseteq\{x_1,\ldots,x_n\}$ and $\sum y_i = t$}

SUBSET-SUM ∈ NP

Polytime verifiable!

Does P = NP?

In other words: Are there polynomial time solutions to all algorithms that are polytime verifiable?

Probably not, but it's an open question!

If P=NP, lots of "hard" problems become doable – including cracking encrypted networks!

Class coNP

- A language is coNP if its complement is NP
- Unknown if all coNP languages are also NP

NP-Complete problems

The <u>hardest</u> problems in NP are **NP complete**

If polynomial time algorithm for these problems, P=NP

- If any NP problem requires more than polytime, then NP-complete problems also require more than polytime

Since no polytime solution has been found for an NP-complete problem, if we determine new problem is NP-complete, reasonable to give up search for general polytime solution to this problem

Satisfiability: An NP-complete problem

Consider Boolean operator AND, OR, NOT

Consider set of Boolean variables

Boolean formula is satisfiable if some assignment of T's and F's makes the total formula True (T)

- Ex: $(x' \vee y) \wedge (x \vee z')$
- Ex: $(x' \vee y) \wedge (x \vee y') \wedge (x \vee y) \wedge (x' \vee y')$

Cook-Levin Theorem

SAT can be used to solve all problems in NP

I.e., SAT∈P iff P=NP

I.e, every problem in NP **efficiently reduces** to SAT

Polynomial time reducibility

When problem A is efficiently reducible to problem B, an efficient solution to B will efficiently solve A

- **"Efficiently reducible"** means in polynomial time
- If language A is polytime reducible to language B, and B has polytime solution, then A has polytime solution!
- A efficiently reduces to B, B efficiently reduces to C
  -> A efficiently reduces to C

3SAT

Special case of satistfiability

Terms:

- Boolean or negated Boolean is "literal"
- Clause is several literals joined by V
- Boolean formula is in "conjunctive normal form" if it has only clauses connected by ⋀s

"3cnf" – each clause has 3 literals

- E.g., (x1Vx2Vx3) ⋀ (x1Vx2'Vx3) ⋀ (x1'Vx2Vx3)
- 3SAT – language of 3cnf formulas that are satisfiable

Polytime reduction from 3SAT to CLIQUE

Given 3SAT formula create a graph G

Nodes in G are organized into k groups of 3 nodes called $t_1,\ldots,t_k$

Each triple corresponds to one of the clauses in the formula

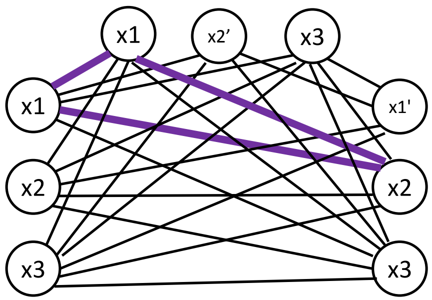
Each node in triple is labeled with literal in the clause

Edges in G connect all but two types of pairs of nodes in G

- No edge is between nodes in same triple
- No edge is present between nodes with contradictory labels

Formula satisfiable iff G has k-clique

(x1Vx2Vx3) Λ (x1Vx2’Vx3) Λ (x1’Vx2Vx3)



Why does this reduction work?

At least one value must be true per clause

Select one node corresponding to true literal in satisfying assignment

Selected nodes must form k-clique

- There will be k nodes
- Every pair of nodes will be connected by an edge because they don’t violate one of the 2 no-edge conditions

Alternatively

- If k-clique, all literals are in different clauses and hence will be satisfiable, since any logically inconsistent assignments are not represented in G

Prove SAT is NP-complete

Full 5-page proof in the textbook

Proof idea:

- Can convert any NP problem to SAT by having Boolean formula represent simulation of NP machine on input
- Note: general SAT relies on Boolean logic – building blocks of computer computations

How do we know SAT is in NP?

If we have a certificate c=True/False values of variables x_1 … x_k, we can verify in polytime that a formula is true.