Intractable Problems

- We have shonw that there are problems which cannot be decided by a computer. Now we look at the problems which can be decided and find that for many of them the only algorithms we have are not practical because they require *too much time*.

- Your customer asks you to design an algorithm for a particular problem. You work really hard, but the best you can come up with is to try all possible solutions, and there are many of them.
  - Do you try harder, or do you decide that perhaps you should settle for an approximate solution?
  - *Goal*: develop a method you can use for persuading yourself and your customer that the problem is hard (intractable), so hard that it is unlikely that anyone could find a fast algorithm for it.

Polynomial Time

Running time must depend on the input – but how? Clearly, just to e.g., read or copy an input string depends on the *length*

We measure the running time as a *function* of the *input length* (i.e. length as a string of symbols).

Types of analysis:

- *Worst case analysis* uses the maximu running time over all inputs of a particular length.
- *Average case analysis* uses the average of all running times over all inputs (or over some distribution of inputs) of a particular length.

Time Complexity

A deterministic TM M has a *(worst-case) running time* (or *time complexity*) $t(n)$ if whenever M is given an input w of length n (i.e. $|w| = n$,) M halts after making at most $t(n)$ moves, regardless of whether M accepts.

- $t(n)$ is a function, such as $6n^2 + n$, $3n \log n$, etc.

- We say M runs in time $t(n)$ and that M is an $t(n)$-time Turing machine.

Big-O Notation

Review from csc225.

- We estimate the running time of an algorithm using *asymptotic notation*. E.g. use the highest order term of the expression for the running time.
  - Let f and g be functions $f, g : N \to R^+$. We say $f(n) = O(g(n))$ if there are positive integers c, n_0 s.t. for every integer $n > n_0$,

$$f(n) \leq cg(n)$$

.

  - Bounds of the form $O(\log n)$ are *logarithmic bounds*
  - Bounds of the form $O(n^c)$ are *polynomial bounds*.
  - Bounds of the form $O(2^{n^\delta})$ for δ a positive real number are called *exponential bounds.*

Analyzing Algorithms for TM

1. A $O(n^2)$ algorithm for $\{0^k 1^k \mid k \geq 0\}$.

2. A $O(n \log n)$ algorithm for the same problem.

3. *Linear time* $O(n)$ algorithm on a two-tape machine.

The *time complexity class* TIME$(t(n))$ is the collection of all languages that are decidable by an $O(t(n))$ time TM.

Complexity Relationship among Models

- A $t(n)$ time multitape TM can be simulated by a $O((t(n))^2)$ single tape TM. Recall that in the simulation, the tapes are stored consecutively.

- Let N be a nondeterministic TM all of whose branches halt (*decider*). The *running time* of N, $f(n)$ is the maximum number of steps that N uses on any branches of its computation on any input of length n.

- A $t(n)$ time nondeterministic TM can be simulated by a $O(2^{O(t(n))})$ time deterministic TM. Recall the simulation: Breadth first search of the tree of possible computations, using three tapes.

The Class P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. I.e., $\mathsf{P} = \cup_k \mathsf{TIME}(n^k)$.

A TM is *poly-time* if it's running time is $O(n^k)$ for some k.

Thesis: Deterministic poly-time TM's and the class P adequately capture the intuitive notions of practically feasible algorithms, and realistically solvable problems, respectively.

Examples of Problems in P

1. The importance of coding: unary vs. binary; adjacency matrix, adjacency list.

2. $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a path from s to $t\}$.

3. $RELPRIME = \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$.

   Algorithm R uses the Euclidian algorithm as a subroutine:

   INPUT: $\langle x, y\rangle$, natural numbers represented in binary.

   (a) Repeat until $y = 0$:
   (b) $\quad x \leftarrow x \mod y$
   (c) $\quad$ exchange x and y
   (d) Output x

Algorithm R for $RELPRIME$

Run *Euclidean Algorithm* on $\langle x, y \rangle$. If the result is 1, accept, else reject.

- Analysis: values of x drops every other round by at least 1/2.
- Cost is $(O(min\{2\log_2 x, 2\log_2 y\}) = O(n)$.

Every CFL is in P

Recall *CYK algorithm* for testing if a string w is in a CFG which is in Chomsky normal form. Keep an array $table(i, j)$ of the variables which can generate the substring $w_i., w_{i+1}, ..., w_j$ Build this table from bottom up – filling in entry (i, j) requires looking at entries (i, k) and (k, j) for $i \leq k \leq j$. So total cost is $O(n^3)$ where n is the length of the input string w (Fill in at most n^2 entries, each one requires considering at most $2n$ other entries.)

The Class NP

Example: HAMPATH

- A *Hamiltonian path* in a directed graph is a directed path that goes through each node exactly once.
- **PROBLEM:** HAMPATH
  INPUT: An directed graph G
  OUTPUT: "YES" if and only if there is a Hamiltonian path in G
- Encoding scheme: list of nodes and edges
- A solution is *polynomially verifiable*.
- How to solve this problem deterministically?
- How to solve this problem nondeterministically?
- Note that $\overline{HAMPATH}$ is not polynomially verifiable.

Verifiers

- A *verifier* for a language A is an algorithm V where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A language A is *polynomially verifiable* is it has a verifier whose running time is polynomial in $|w|$
- In the definition above c is called a *certificate* or *proof*.
- Note that if A has a poly-time verifier, then we may assume there is a polynomial that bounds $|c|$ in terms of $|w|$ (Why?)

Sudoku is Polynomially Verifiable

Consider Sudoku for $n \times n$ input grids, with entries $1 \ldots n$

$$SUDOKU = \{\langle \Pi \rangle \mid \Pi \text{ is a solvable Sudoku puzzle}\}$$

A certificate for Π is just the list of values for all empty cells of Π – at most n^2 entries, each can be given by a $\log n$-bit string. Size $O(n^2 \log n) = O(n^3)$

To verify a certificate – must check uniqueness for every row, column, and subgrid. Can be done in $O(n^3)$ time (Why? Think of CNF encoding...)

Definition of NP

NP is the class of languages that have polynomial time verifiers.

NP $\equiv$ "Non-deterministic polynomial time"

Theorem: A language L is in NP iff it is decided by a nondeterministic poly-time TM.

Proof $(\Longrightarrow)$ Let V be a verifier for L. We define the NDTM M as follows.

On input w:

1. Use nondeterminism to "guess" a certificate c
2. Call $V(\langle w, c \rangle)$
3. Answer YES iff V answers YES

$(\Longleftarrow)$ An accepting computation is a poly-sized certificate that can be verified in polynomial time.

P vs NP

NTIME$(t(n))$ is the class of languages decided by $O(t(n)$-time NDTMs.

So we have shown NP $= \bigcup_k$ NTIME(n^k)

It is then easy to see that P $\subseteq$ NP (every deterministic machine is also a nondeterministic machine

Is P $=$ NP$?$

In other words, for every problem, is it this case that if we can *verify* solutions in polynomial time, then we can also *decide* whether the problem has a solution in polynomial time?

Most fundamental question in (theoretical) Computer Science.

CLIQUE

INPUT: A graph G and an integer k (no greater than the number of nodes in G)

QUESTION: Does G contain a clique of size k, i.e, a subset of k nodes such that every two nodes in the subset are joined by an edge of G?

Clique is in NP

Proof: There is a polytime verifier V which takes as input $\langle\langle G, k\rangle, c\rangle$ and does the following

1. Test whether c is a set of k nodes
2. Test whether G contains all edges between every pair nodes in c
3. If both pass, answer YES, else answer NO

Step (1) takes time $O(k)$, and step (2) takes time $O(k^2)$. So V runs in time $O(n^2)$ ($n = |\langle G, k\rangle|$. Note that $n \geq |\langle G\rangle| \geq k$, so we may assume k is in binary.)

ALTERNATIVE PROOF: There is a non deterministic polytime TM which decides CLIQUE (guess a subset of nodes and verify that it is a clique).

SUBSET-SUM (KNAPSACK) is in NP

INPUT: A set S of numbers and a number t.

QUESTION: Is there a subset S' of S whose elements sum to t?

How to code inputs? List of binary numbers coding S and t

Theorem: SUBSET-SUM is in NP.

Proof idea: A certificate c for SUBSET SUM is a subset of S. To verifiy, check that each integer in the certificate is in S and add them up to see if the sum is t. On a 2-tape TM, this takes $O(\sum_{s \in S'} |s|))$ time, which is polynomial in the length of the instance; hence it's polynomial in the length of the instance.

Questions

- How to win \$1,000,000: Is $\mathsf{P} = \mathsf{NP}$? (Clay Prize)
- Note that we can solve any problem in NP in exponential time, i.e., $\mathsf{NP} \in \bigcup_k \mathsf{TIME}(2^{n^k})$
- What about parallel machines?
- What about quantum machines?
- Doesn't the encoding scheme matter?
- Worst case v. average case time complexity?
- Does $\mathsf{NP} \cap co\text{-}\mathsf{NP} = \mathsf{P}$?