Lecture 5: Computational Complexity

(3 units)

Outline

- Computational complexity
- Decision problem, Classes $\mathcal{NP}$ and $\mathcal{P}$.
- Polynomial reduction and Class $\mathcal{NPC}$
- $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$?

The Goal of Computational Complexity

- Provide a method of quantifying problem difficulty;
- Compare the relative difficulty of two different problems;
- Rigorously define the meaning of an efficient algorithm;
- State that one algorithm for a problem is “better” than another.

Computational Complexity

- The ingredients that we need to build a theory of computational complexity for problem classification are the following:
  - A class $\mathcal{C}$ of problems to which the theory applies;
  - A (nonempty) subclass $\mathcal{C}_{\mathcal{E}} \subseteq \mathcal{C}$ of "easy" problems;
  - A (nonempty) subclass $\mathcal{C}_{\mathcal{H}} \subseteq \mathcal{C}$ of "hard" problems;
  - A relation $\lhd$ "not more difficult than" between pairs of problems.
- Our goal is just to put some definitions around this machinery:
  - $Q \in \mathcal{C}_{\mathcal{E}}$, $P \lhd Q \Rightarrow P \in \mathcal{C}_{\mathcal{E}}$;
  - $P \in \mathcal{C}_{\mathcal{H}}$, $P \lhd Q \Rightarrow Q \in \mathcal{C}_{\mathcal{H}}$.

Decision problem

- The theory we develop applies only to decision problems
- Problems that have a “yes-no” answer.
  - Optimization: $\max\{c^T x \mid x \in S\}$;
  - Decision: $\exists x \in S$ such that $c^T x \geq k$?
- Example: The Bin Packing Problem (BPP): We are given a set S of items, each with a specified integral size, and a specified constant C, the size of a bin.
  - Optimization: Determine the smallest number of subsets into which one can partition S such that the total size of the items in each subset is at most C
  - Decision: For a given constant K determine whether S can be partitioned into K subsets such that that the total size of the items in each subset is at most C

- Turning Opt to Decision:
  Suppose you know that $l \leq z^* \leq u$, where $l, z^*, u \in \mathbb{Z}$. How can you solve Opt by solving a series of Decision problems?
- Binary search:

```
1.  if ($u - l <= 1$), $z^* = l$; exit();
2.  $k = (l + u)/2$; if (dec($k$)==false) $l = k$; else
$u = k$; goto 1;
```

  Requires at most $\log_2(u - l) + 1$ calls to `dec(k)`

Measuring the Difficulty of a Problem

- We are interested in knowing the difficulty of a problem, not an instance.
- A problem (or model) is an infinite family of instances whose objective function and constraints have a specific structure.
- Possible methods of measuring:
  - Empirical:
    Does not given us any real guarantee about the difficulty of a given instance;
  - Average case running time: Difficult to analyze and depends on specifying a probability distribution on the instances.
  - Worst case running time:
  - The complexity theory is based on a worst-case approach.

Running Time of an Algorithm

- The running time of an algorithm depends on size of the input.
- A time complexity function specifies, as a function of the problem size, the largest amount of time needed by an algorithm to solve any problem instance.
- How do we measure problem size?
  - The length of the amount of information necessary to represent the problem in a reasonable encoding scheme.
  - For a problem instance X, the length of the input $L(X)$ is the length of the binary representation of a “standard” representation of the instance.

- An integer $2^n \leq x < 2^{n+1}$ can be represented by a vector $(v_0, v_1, \ldots, v_n)$, where

$$x = \sum_{i=0}^{n} v_i 2^i, \;\; v_i \in \{0, 1\}.$$

  Note that $n \leq \log_2 x < n + 1$. So, it requires a logarithmic number of bits to represent $x \in \mathbb{Z}$, $\log_2 x + 1$.

- Example: TSP on n cities with costs $c_{ij} \in \mathbb{Z}$, $\max_{i,j} c_{ij} = \theta$, it requires $L(X) = \log(n) + n^2 \log(\theta)$ bits to represent an instance.
- Example: Knapsack: n, a_j, c_j, b, where a_j, c_j, b are rational, $a_j \leq b$.

$$L(X) = \log(n) + (2n + 2)\log(b) + 2n\log(\theta),$$

  where $\theta = \max c_j$.

- How to measure computation time?
  - We want the measure to be independent of particular computers;
  - Count the number of elementary operations;
  - Assume that each elementary operation is done in unit time. *This is reasonable as long as the size of numbers does not grows too rapidly.*
- Given a problem P, and algorithm A that solves P, and an instance X of problem P.
  - $L(X)$=The length (in a reasonable encoding) of the instance;
  - $g_A(X)$=the number of elementary calculations required to run algorithm A on instance X;
  - $f_A(l) = \sup\{g_A(X) \mid L(X) = l\}$ is the running time of algorithm A.

- $f_A(l)$ is $O(g(l))$ if there exists $c > 0$ such that $f_A(l) \leq cg(l)$ for all l. Asymptotic behavior of the function as $k \to \infty$ is considered.
- A is said to be polynomial time algorithm if $f_A(l) = O(l^p)$ for some fixed p.
  - A is strongly polynomial if $f_A(l)$ is bounded by a polynomial function that does not involve the data size (magnitude of numbers).
  - A is weakly polynomial if it is polynomial and not strongly polynomial. The l in $O(l^p)$ contains terms involving $\log \theta$, where θ is largest number in a given instance.
- A is said to be exponential time algorithm if $f_A(l) \neq O(l^p)$, for any p, i.e., there exist $c_1, c_2 > 0$, $d_1, d_2 > 1$ and positive integer k such that

$$c_1 d_1^l \leq f_A(l) \leq c_2 d_2^l, \quad \forall \text{integer } l \geq l'.$$

- A pseudo polynomial algorithm A is one that is polynomial in the length of the data when encoded in unary.
  - Unary means that we are using a one-symbol alphabet. (not binary)
- Practically, it means that A is polynomial in the parameters and the magnitude of the instance data θ not $\log(\theta)$.
- Example: The Integer Knapsack Problem. There is an $O(Nb)$ algorithm for this problem, where N is the number of items and b is the size of the knapsack.
- This is not a polynomial-time algorithm. If b is bounded by a polynomial function of n, then it is.
- Let θ be the largest number in a given instance. The length of the input data is $l = O(\log(\theta))$. If an algorithm requires θ steps, then it is an exponential algorithm! since $f_A(l) \geq O(\theta) = O(2^l)$.

The class $\mathcal{NP}$

- The problem class $\mathcal{NP} \neq$ “Non-polynomial”.
- $\mathcal{NP}=$ the class of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. .....
  # % * & ?!
- $\mathcal{NP} \approx$ the class of decision problems with the property that for every instance for which the answer is “yes”, there is a short (polynomial) certificate. The certificate is your “proof” that what you are telling me is the truth.
- If a decision problem associated with an optimization problem is in $\mathcal{NP}$, then the optimization problem can be solved by answering the decision problem a polynomial number of times.
- Let $\mathcal{P}$ be the class of problems in $\mathcal{NP}$ that can be solved in polynomial time.

- Examples: For an 0-1 knapsack problem instance X.
  - Decision problem: Is there exists $x \in S = \{x \in \{0,1\}^n \mid a^T x \leq b\}$ such that $c^T x \geq k$?
  - The length of input: $L(X) = \log(n) + 2n\log(\theta) + \log b + \log k$.
    For an instance for which the answer is "Yes", it suffices:
    (a) read a solution $x \in \{0,1\}^n$;
    (b) check $a^T x \leq b$ and $c^T x \geq k$.
    Both (a) and (b) can be done in time polynomial in L.
- How about integer knapsack problem?

- Problems in $\mathcal{P}$:
  - Shortest path problem with nonnegative weights: $O(m^2)$. Note that the number of operations is independent of the magnitude of the edge weights, so it is strongly polynomial.
  - Solving systems of equations $Ax = b$.
    (a) The solution $x = A^{-1}b$ can be found by Guassian elimination, n pivots and each pivot requires $O(n^2)$ calculations. So total calculation is $O(n^3)$.
    (b) The magnitude of the numbers that occur is bounded by the largest determinant of any square submatrix of (A, b). Since $det(A)$ involves $n! < n^n$ terms, this largest number is bounded by $(n\theta)^n$, where θ is the largest entry of (A, b). This means that the size of their representation is bounded by a polynomial function of n and $\log\theta$: $\log((n\theta)^n) = n\log(n\theta)$:
    (c) Polynomial in the size of the input.
  - Assignment Problem: $O(n^4)$.

- Linear programming: $\min\{c^T x \mid Ax = b,\ x \geq 0\}$.
- Algorithms for linear programming:
  - George Dantzig developed Simplex Method in 1947.
  - However, the simplex algorithm has poor worst-case behavior: Klee and Minty constructed a family of linear programming problems for which the simplex method takes a number of steps exponential in the problem size. In fact, for some time it was not known whether the linear programming problem was solvable in polynomial time, i.e. of complexity class P.
  - Leonid Khachiyan in 1979 introduced the ellipsoid method for LP in 1979, which is the first worst-case polynomial-time algorithm for linear programming (with complexity $O(n^4 L)$).
  - N. Karmarkar proposed a projective interior-point method for LP in 1984 (with complexity ($O(n^{3.5} L)$).
  - Path-following algorithms since 1990s.
- Open problem in LP: Does LP admit a strongly polynomial-time algorithm?

- The Class co-$\mathcal{NP}$. The class of problems for which the "complement" problem to is $\mathcal{NP}$.
- co-$\mathcal{NP}$ $\approx$ the class of decision problems with the property that for every instance for which the answer is "no", there is a short certificate.

The class $\mathcal{NPC}$

- If problems $P, Q \in \mathcal{NP}$, and if an instance of P can be converted in polynomial time to an instance of Q, then P is polynomially reducible to Q. We will write this as $P \lhd Q$.
- We now want to ask the question: What are the hardest problems in $\mathcal{NP}$?
- If $P \in \mathcal{NPC}$, then $Q \in \mathcal{NP} \Rightarrow Q \lhd P$.
- If $P \in \mathcal{NP}$ and we can convert in polynomial time every other problem $Q \in \mathcal{NP}$ to P, then P is in this sense the hardest problem in $\mathcal{NP}$.
- An optimization problem is called NP-hard if its decision problem is in $\mathcal{NPC}$.

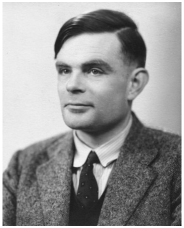
- Stephen Cook proved in 1970 Satisfiability problem is in $\mathcal{NPC}$. –First known $\mathcal{NPC}$ problem!

Stephen Cook (1939–)

- Satisfiability problem: Given $N = \{1, \ldots, n\}$ and $2m$ subsets $\{C_i\}_{i=1}^m$ and $\{D_i\}_{i=1}^m$. Does there exist $x \in \{0,1\}^n$ such that

$$\sum_{j \in C_i} x_j + \sum_{j \in D_i} (1 - x_j) \geq 1?$$

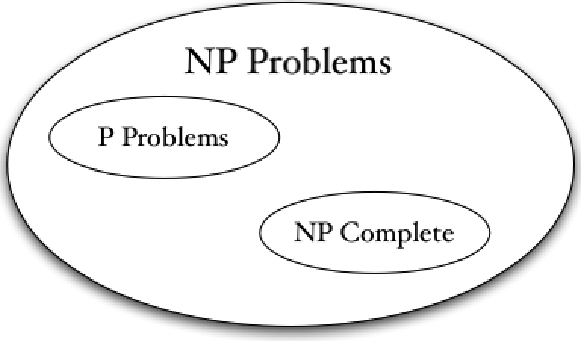
- Stephen A. Cook won 1982 Turing Award



Alan Turing (1912–1954)

- Turing Award Citation: "For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, "The Complexity of Theorem Proving Procedures," presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, Laid the foundations for the theory of NP-Completeness. The ensuing exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade."

- $Q \in \mathcal{P},\ P \lhd Q \Rightarrow P \in \mathcal{P}$.
- $P \in \mathcal{NPC},\ P \lhd Q \Rightarrow Q \in \mathcal{NPC}$.



P vs NP Problem – Clay Mathematics Institute

- A million dollar open question:
$$\mathcal{P} = \mathcal{NP}?$$
- http://www.claymath.org/Millennium Prize Problems/P vs NP/
- Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice.
- This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical.

- Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students.
- Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer.
- How to win it? $\mathcal{P} \cap \mathcal{NPC} \neq \emptyset \Rightarrow \mathcal{P} = \mathcal{NP}$.