ECS122A Handout on NP-Completeness

Contents:

I. Introduction

II. P and NP

III. NP-complete

IV. How to prove a problem is NP-complete

V. How to solve a NP-complete problem: approximate algorithms

I. Introduction

1. Tractable (easy, "not-so-hard") and intractable ("hard") problems

   Generally, we think of problems that are solvable by polynomial-time algorithms are being *tractable*, and problems that require superpolynomial time as being *intractable*.

   Almost all the algorithms we have studied thus far have been polynomial-time algorithms: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.

2. We now turn to a class of problems, called the NP-complete problems, which is a class of very diverse problems, that share the following properties: we *only know* how to solve those problems in time much larger than polynomial, namely exponential time, and if we could solve *one* NP-complete problem in polynomial time, then there is a way to solve *every* NP-complete porblem in polynomial time.

3. Two reasons to study NP-complete porblems:

   (a) The practical one is that if you recognize that a problem is NP-complete, then you have three choices:

   - you can use a known algorithm for it, and accept that it will take a long long time to solve;
   - you can settle for approximating the solution, e.g., finding a nearly best solution rather than the optimum; or
   - you can change your problem formulation so that it is solvable in polynomial time.

   (b) One of the most famous open problem in computer science concerns the NP-complete problem.

   We stated above that "*we only know*" how to solve those problems in time much larger than polynomial, *not that we have proven* that NP-complete problems require exponential time. Indeed, this is a million-dollar question[1], one of the most famous problem in computer science, the question is whether P = NP?, or whether the class of NP-complete problems have polynomial solutions (first posed in 1971).

4. A particular tantalizing aspect of the NP-complete problems is that several of them seem on the surface to be similar problems that have polynomial-time algorithms.

   In each of the following pairs of problems, one is solvable in polynomial time, and the other is NP-complete, but the difference between problems appears to be slight:

[1] http://www.claymath.org/millennium-problems

- Shortest vs. Longest simple paths:
  Shortest path: finding the shortest path from a single source in a directed graph.
  Longest path: finding the longest simple path between two vertices in a directed graph.
- Euler cycle vs Hamiltonian cycle
  - Euler cycle: given a connected, directed graph G, is there a cycle that visits each *edge* exactly once (although it is allowed to visit each vertex more than once)?
  - Hamiltonian cycle: given a directed graph G, is there a simple cycle that visits each *vertex* exactly once?
- Minimum Spanning Tree (MST) vs. Traveling Salesperson Problem (TSP)
  - MST: given a weighted graph and an integer k, is there a spanning tree whose total weight is k or less?
  - TSP: given a weighted graph and an integer k, is there a cycle that visits all vertices exactly once of the graph whose total weight is k or less?
- Circuit value vs Circuit satisfiability
  - Circuit value: given a Boolean formula and its input, is the output True?
  - Circuit satisfiability: given a Boolean formula, is there a way to set the inputs so that the output is True?

5. Optimization problems vs. Decision problems

   Most of problems occur naturally as optimization problems, but they can also be formulated as decision problems, that is, problems for which the output is a simple *Yes* or *No* answer for each input.

   Examples:

   - Graph coloring: a coloring of a graph $G = (V, E)$ is a mapping $C : V \to S$, where S is a finite set of "colors", such that if $(u, v) \in E$, then $C(u) \neq C(v)$.
     - Optimization problem: given G, determine the smallest number of colors needed.
     - Decision problem: given G and a positive integer k, is there a coloring of G using at most k colors? If so, G is said to be *k-colorable*.
   - Hamiltonian cycle:
     - Decision problem: Does a given graph have a Hamiltonian cycle?
     - Optimization problem: Give a list of vertices of a Hamiltonian cycle.
   - TSP (Traveling Salesperson Problem):
     - Optimization problem: given a weighted graph, find a minimum Hamiltonian cycle.
     - Decision problem: given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight at most k?

   To simplify the discussion, we can consider only the decision problems with Yes-No answers, rather than the optimization problems. The optimization problems are at least as hard to solve as the related decision problems, we have not lost anything essential by doing so.

II. P and NP

1. **P** is the class of decision problems that can be solved in polynomial time, i.e., they are polynomial bounded.

   An algorithm is said to *polynomial bounded* if its worst-case complexity is bounded by a polynomial function of the input size, i.e. $T(n) = O(n^k)$.

   Examples: Shortest path, MST, Euler tour, Circuit value.

2. **NP** is the class of decision problems that are "*verifiable*" in polynomial time. What we mean here is that if we were somehow given a "certificate" (= a solution), then we could verify that the certificate is correct in time polynomial in the size of input to the problem.

   NP stands for "Nondeterministic Polynomial time"

   Examples: Longest path, TSP, Hamiltonian cycle, Circuit satisfiability, graph coloring.

3. P $\subseteq$ NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate.

4. Open question: Does P = NP or P $\subset$ NP ?

5. The size of the input can also change the classification of P or NP. Knowing the effect on complexity of restricting the set of inputs for a problem is important.

   Examples:

   - Prime-testing problem
   - Knapsack problem

   Remark: Unfortunately, even with quite strong restrictions on the inputs, many NP-complete problems are still NP-complete. For example, 3-Conjuntive Normal Form (3-CNF) satisfiability problem.

III. NP-complete

1. **NP**-*complete* is the term used to describe decision problems that are the hardest ones in **NP** in the sense that, if there were a polynomial-bounded algorithm for an NP-complete problem, then there would be a polynomial-bounded time for each problem in NP.

2. Polynomial reduction

   Let A and B be two decision problems, we say that A is *polynomially reducible* to B, denoted as $A \leq_T B$, if there is a poly-time computable function T such that

   $$\text{Yes-instance of } A \iff \text{Yes-instance of } B$$

3. Formal definition of **NP**-complete

   A decision problem A is *NP-complete* (NPC) if

   (1) $A \in$ NP and

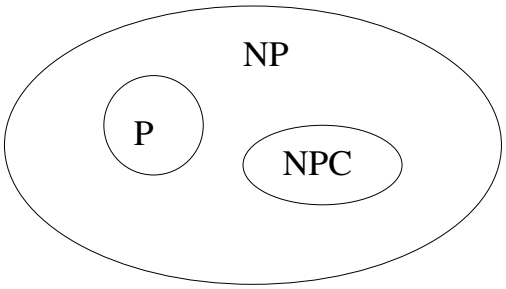
   (2) every (other) problems A' in NP is reducible to A.

4. A first NP-complete problem: The circuit satisfiability (circuit-SAT) problem is NP-complete

   This is the famous Cook's theorem (1971), which is the first major theorem deomonstrating that a specific problem is NP-complete.

5. Theorem: Graph coloring, Hamiltonian cycle, TSP, ... are all NP-complete.

6. How most theoretical computer scientists view the relationships among P, NP and NPC. Both P and NPC are wholely contained within NP, and $\mathrm{P} \bigcap \mathrm{NPC} = \emptyset$.



7. If a problem satisfies the property (2), but not necessarily the property (1), we say the problem is *NP-hard*

   It is important to realize that "NP-hard" does not mean "in NP and hard". It means "at least as hard as any problem in NP". Thus a problem can be NP-hard and not be in NP.

IV. How to prove a problem is NP-complete?

1. Since the reducibility relation is transitive, to prove that a problem A in NP is NP-complete, it suffices to prove that some other NP-complete problem B is polynomially reducible to it. Specifically, choose some known NP-complete problem B, and reduce B to A (*note: not the other way around!*), i.e., show that

   $$B \leq_T A$$

   The logic is as follows:

   Since B is NP-complete, all problems in NP is reducible to B.
   Show B is reducible to A.
   Then all problems in NP is reducible to A.
   Therefore, A is NP-complete

2. Example 1. The directed Hamiltonian cycle (HC) problem is known to be NP-complete. In the following, we show that the directed HC problem is reducible to the undirected HC problem. Therefore, we conclude that the undirected HC problem is also NP-complete.

   PROOF: Let $G = (V, E)$ be a directed graph with n vertices. G is transformed into the undirected graph $G' = (V', E')$ by the transform function T defined as the following:

   $$v \in V \quad \longrightarrow \quad v^1, v^2, v^3 \in V' \text{ and } (v^1, v^2), (v^2, v^3) \in E'.$$

   and

   $$(u, v) \in E \quad \longrightarrow \quad (u^3, v^1) \in E'.$$

   Clearly, T is polynomial-time computable.

   We now show that

   $$G \text{ has a HC} \iff G' \text{ has a HC.}$$

   - "$\Longrightarrow$": Suppose that G has a directed HC, $v_1, v_2, \ldots, v_n, v_1$. Then

     $$v_1^1, v_1^2, v_1^3, v_2^1, v_2^2, v_2^3, \ldots, v_n^1, v_n^2, v_n^3, v_1^1$$

     is an undirected HC for G'.

- "$\Longleftarrow$": Suppose that G' has an undirected HC, the three vertices, say v^1, v^2, v^3 that correspond to one vertex from G must be traversed consecutively in the order v^1, v^2, v^3 or v^3, v^2, v^1, since v^2 cannot be reached from any other vertex in G'. Since the other edges in G' connect vertices with superscripts 1 or 3, if for any one triple the order of the superscripts is 1, 2, 3, then the order is 1, 2, 3 for all triples. Otherwise, it is 3, 2, 1 for all triples. Since G' is undirected, we may assume that its HC is

$$v^1_{i_1}, v^2_{i_1}, v^3_{i_1}, v^1_{i_2}, v^2_{i_2}, v^3_{i_2}, \ldots, v^1_{i_n}, v^2_{i_n}, v^3_{i_n}, v^1_{i_1}.$$

Then $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$ is a directed HC for G. □

3. Example 2: The subset sum problem is known to be NP-complete, In the following, we show that the subset sum problem is reducible to the job scheduling (with penalties) problem. Therefore, we conclude that the problem of job scheduling with penalties is also NP-complete.

- Subset sum decision problem: Given a positive integer C, and the set $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers s_i for $i = 1, 2, \ldots, n$. Assume that $\sum_{i=1}^n s_i \geq C$. Is there a $J \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in J} s_i = C$?
- Job-Scheduling decision problem: Suppose there are n jobs $J_1, J_2, \ldots, J_n$ to be executed one at a time, each takes t_i time, deadline d_i and penalties p_i for missing the deadline. A schedule for the jobs is to find a permutation π of $\{1, 2, \ldots, n\}$, such that the jobs are executed in the order $J_{\pi(1)}, J_{\pi(2)}, \ldots, J_{\pi(n)}$. The corresponding total penalty is

$$P_\pi = \sum_{j=1}^{n} P_{\pi(j)}$$

where

$$P_{\pi(j)} = \begin{cases} p_{\pi(j)} & \text{if } t_{\pi(1)} + \cdots + t_{\pi(j)} > d_{\pi(j)} \\ 0 & \text{otherwise} \end{cases}$$

The decision problem: given k, is there a schedule such that $P_{\pi(j)} \leq k$?

- We show that the subset sum problem is reducible to the job scheduling problem.

PROOF: Let $s_1, s_2, \ldots, s_n$ and C be an input for the subset sum problem. Let the input be transformed into the following input for the job scheduling problem

$$\begin{aligned} t_i &= p_i = s_i \quad \text{for } 1 \leq i \leq n, \\ d_i &= C \quad \text{for } 1 \leq i \leq n, \\ k &= \sum_{i=1}^{n} s_i - C. \end{aligned}$$

Clearly the transformation takes polynomial time.

Now we shows that

Yes-instance of the subset sum $\Longleftrightarrow$ Yes-instance of the job scheduling.

– "$\Longrightarrow$": suppose that the subset sum input produces a YES answer; i.e., there is a subset J of $N = \{1, 2, \ldots, n\}$ such that $\sum_{i \in J} s_i = C$. Then let π be any permutation of N that causes all jobs with indices in J to be done before any job with indices in $N - J$. The first $|J|$ jobs are completed by their deadline since $\sum_{i \in J} t_i = \sum_{i \in J} s_i = C$ and C is the deadline for all jobs. Then the total penalty

$$P_\pi = \sum_{j=1}^{|J|} p_{\pi(j)} + \sum_{j=|J|+1}^{n} p_{\pi(j)} = 0 + \sum_{j=|J|+1}^{n} p_{\pi(j)} = \sum_{j=|J|+1}^{n} s_{\pi(j)} = \sum_{j=1}^{n} s_j - C = k.$$

Thus the jobs can be done with total penalty $\leq k$.

– "$\Longleftarrow$" Let π be any schedule for the jobs with total penalty $\leq k$. Let m be largest such that

$$\sum_{i=1}^{m} t_{\pi(i)} \leq C; \tag{1}$$

i.e., m is the number of jobs completed by the deadline C. The penalty, then, is

$$\sum_{i=m+1}^{n} p_{\pi(i)} \leq k = \sum_{i=1}^{n} s_i - C. \tag{2}$$

Since $t_i = p_i = s_i$ for all $1 \leq i \leq n$, we must have

$$\sum_{i=1}^{m} t_{\pi(i)} + \sum_{i=m+1}^{n} p_{\pi(i)} = \sum_{i=1}^{m} s_{\pi(i)} + \sum_{i=m+1}^{n} s_{\pi(i)} = \sum_{i=1}^{n} s_i,$$

and this can happen only if the inequalities in (1) and (2) are equalities. Thus

$$\sum_{i=1}^{m} t_{\pi(i)} = C,$$

so the objects with indices $\pi(1), \pi(2), \ldots, \pi(m)$ are a solution to the subset sum problem. □

V. Approximation Algorithms

1. What can we do when we encounters an NP-complete problem? We have three choices:

   (a) You can use a known algorithm for it, and accept that it will take a long time to solve if n is large.

   (b) You can settle for approximating the solution, e.g., finding a nearly best solution rather than the optimum.

   (c) You can change your problem formulation so that it is in P, rather than being NP-complete, for example, by restricting it to work only on a subset of simpler input problems.

   We will show how to get near-optimal solutions in polynomial time. In practice, near-optimality is often good enough.

2. Apprixmate algorithms for Bin Packing Problem

   - Problem statement: suppose we have an unlimited number of bins, each of capacity 1, and n objects with sizes $s_1, s_2, \ldots, s_n$, where $0 < s_i \leq 1$.
   - Decision problem: Do the objects fit in k bins?
   - Optimization problem: Determine the smallest number of bins into which the objects can be packed and find an optimal packing.
   - Theorem: The bin-packing problem is NP-complete.
   - Approximate algorithm:
     First-fit strategy (greedy): places an object in the first bin into which it fits.

- Example: Objects = {0.8, 0.5, 0.4, 0.4, 0.3, 0.2, 0.2, 0.2}
  Approximate Algorithm solution:

B_1	B_2	B_3	B_4
		0.2	
0.2	0.4	0.3	
0.8	0.5	0.4	0.2

  Optimal packing:

B_1	B_2	B_3
	0.2	0.2
0.2	0.3	0.4
0.8	0.5	0.4

- Theorem: Let $S = \sum_{i=1}^{n} s_i$.
  (a) The optimal number of bins required is at least $\lceil S \rceil$
  (b) The number of bins used by the first-fit strategy is never more than $\lceil 2S \rceil$.

3. Apprixmate algorithms for Vertex-Cover Problem

- Problem statement: a vertex-cover of an undirected graph $G = (V, E)$ is a subset of $V' \subseteq V$ such that if the edge $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). In other words, a vertex cover for G is a set of vertices that covers all edges in E.
  The size of a vertex cover is the number of vertices in it.
- Optimization problem: find a vertex cover of minimum size.
- Decision problem: determine whether a graph has a vertex cover of a given size k.
- Theorem: The vertex-cover problem is NP-complete.
- Approximate algorithm
  $C = \emptyset$
  $E' = E$
  while $E' \neq \emptyset$
  let (u, v) be an arbitrary edge of E'
  $C = C \cup \{u, v\}$
  remove from E' every edge incident on either u or v.
  endwhile
  return C
- Theorem. The size of the vertex-cover is no more than twice the size of an optimal vertex cover.