Chapter-6

NP hard and NP Complete problems

Basic Concepts

The computing times of algorithms fall into two groups.

Group1– consists of problems whose solutions are bounded by the polynomial of small degree.

Example – Binary search o (log n) , sorting o(n log n), matrix

multiplication 0(n 2.81). NP –HARD AND NP – COMPLETE

PROBLEMS

Group2 – contains problems whose best known algorithms are non polynomial.

Example –Traveling salesperson problem 0(n22n), knapsack problem 0(2n/2) etc. There are two classes of non polynomial time problems

1. NP- hard
2. NP-complete

A problem which is NP complete will have the property that it can be solved in polynomial time iff all other NP – complete problems can also be solved in polynomial time.
The class NP (meaning non-deterministic polynomial time) is the set of problems that might appear in a puzzle magazine: ``Nice puzzle."

What makes these problems special is that they might be hard to solve, but a short answer can always be printed in the back, and it is easy to see that the answer is correct once you see it.

Example... Does matrix A have LU
decomposition? No guarantee if answer is
``no".

Another way of thinking of NP is it is the set of problems that can solved efficiently by a really good guesser.
The guesser essentially picks the accepting certificate out of the air (Non-deterministic Polynomial time). It can then convince itself that it is correct using a polynomial time algorithm. (Like a right-brain, left-brain sort of thing.)
Clearly this isn't a practically useful characterization: how could we build

such a machine? Exponential Upper bound

Another useful property of the class NP is that all NP problems can be solved in exponential time (EXP).
This is because we can always list out all short certificates in exponential time and check all O (2nk) of them.
Thus, P is in NP, and NP is in EXP. Although we know that P is not equal to EXP, it is possible that NP = P, or EXP, or neither. Frustrating!

NP-hardness

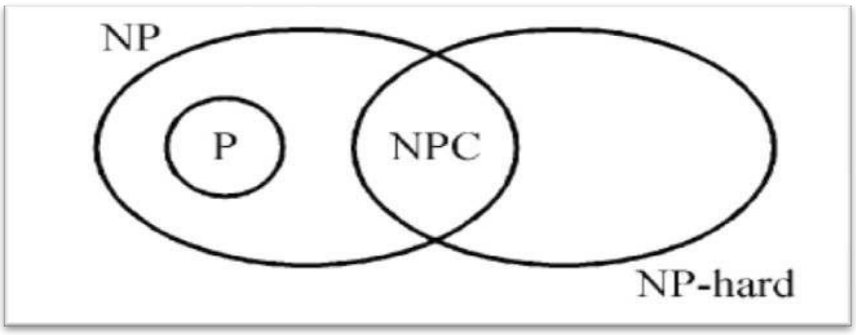
As we will see, some problems are at least as hard to solve as any problem in NP. We call such problems NP-hard.
How might we argue that problem X is at least as hard (to within a polynomial factor) as problem Y?
If X is at least as hard as Y, how would we expect an algorithm that is able to solve X to behave?

NP –HARD and NP – Complete Problems Basic Concepts

If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.

All NP-complete problems are NP-hard, but all NP- hard problems are not NP-complete.

The class of NP-hard problems is very rich in the sense that it contains many problems from a wide variety of disciplines.



P: The class of problems which can be solved by a deterministic polynomial algorithm.

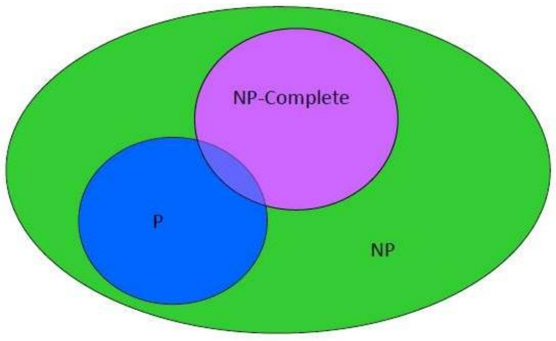
NP: The class of decision problem which can be solved by a non-deterministic polynomial algorithm.

NP-hard: The class of problems to which every NP problem reduces

NP-complete (NPC): the class of problems which are NP-hard and

belong to NP. NP-Competence

- How we would you define NP-Complete
- They are the "hardest" problems in NP



Deterministic and Nondeterministic Algorithms

- Algorithmswiththepropertythattheresultofeveryoperationisuniquelydefine daretermedd eterministic
- Such algorithms agree with the way programs are executed on a computer.
- In a theoretical framework, we can allow algorithms to contain operations whose outcome are not uniquely defined but are limited to a specified set of possibilities.
- Themachineexecutingsuchoperationsareallowedtochooseanyoneoftheseou tcomessubje cttoaterminationcondition.
- This leads to the concept of non deterministic algorithms.
- To specify such algorithms in SPARKS, we introduce three statements Choice (s) ……… arbitrarily chooses one of the elements of the set S. Failure …. Signals an unsuccessful completion.
  Success: Signals a successful completion.
- Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates.
- A non deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a successful signal.
- A machine capable of executing an on deterministic algorithm is called an un deterministic machine.
- While non deterministic machines do not exist in practice they will provide strong intuitive reason to conclude that certain problems cannot be solved by fast deterministic algorithms.

Nondeterministic algorithms

A non deterministic algorithm
consists of Phase 1: guessing
Phase 2: checking

- If the checking stage of a non deterministic algorithm is of polynomial time- complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems : (must be decision problems)
  –e.g. searching,
  MST Sorting
  Satisfy ability problem (SAT)
  travelling salesperson problem
  (TSP)
  Example of a non deterministic algorithm
  // The problem is to search for an element x //
  // Output j such that A(j) =x; or j=0 if x is
  not in A // j choice (1 :n)
  if A(j) =x then print(j) ; success
  endif print ('0') ; failure
  complexity 0(1);

Non-deterministic decision algorithms generate a
zero or one as their output.

Deterministic search algorithm complexity. (n)

- Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time iff the corresponding optimization problem can.
- The decision is to determine if there is a 0/1 assignment of values to xi $1 \leq i \leq n$ such that $\sum pi\ xi \geq R$, and $\sum wi\ xi \leq M$, R, M are given numbers pi, wi ≥ 0, $1 \leq i \leq n$.
- It is easy to obtain polynomial time nondeterministic algorithms for many problems that can be deterministically solved by a systematic search of a solutions pace of exponential size.

Satisfiability

- Letx1, x2, x3…. xn denotes Boolean variables.
- Let xi denotes the relation of xi.
- A literal is either a variable or its negation.
- A formula in the prepositional calculus is an expression that can be constructed using literals and the operators and ^ or v.
- A clause is a formula with at least one positive literal.
- The satisfy ability problem is to determine if a formula is true for some assignment of truth values to the variables.
- It is easy to obtain a polynomial time non determination algorithm that terminates s successfully if and only if a given prepositional formula E(x1, x2……xn) is satiable.
- Such an algorithm could proceed by simply choosing (non deterministically) one of the 2n possible assignment so f truth values to

(x1, x2…xn) and verify that E(x1,x2…xn) is true for that assignment.

Some NP-hard Graph Problems

ThestrategytoshowthataproblemL2isNP-hardis

3. Pick a problem L1 already known to be NP-hard.
4. Show how to obtain an instance I1 of L2m from any instance I of L1 such that from the solution of I1 We can determine (in polynomial deterministic time) thesolutiontoinstanceIofL1
5. Conclude from (ii) that L1L2.
6. Conclude from (1),(2), and the transitivity
   of that Satisfiability L1 L1L2
   Satisfiability
   L2 L2is NP-
   hard

1. Chromatic Number Decision Problem (CNP)
   a. A coloring of a graph G = (V,E) is a function f : V □ { 1,2, …, k} i V
   b. If (U, V) E then f(u) f(v).
   c. The CNP is to determine if G has a coloring for a given K.
   d. Satisfiability with at most three literals per clause chromatic number problem. CNP is NP-hard.

2. Directed Hamiltonian Cycle(DHC)
   Let G=(V,E) be a directed graph and length n=1V1
   TheDHCisacyclethatgoesthrougheveryvertexexactlyonceandthenreturnstot hestartingv ertex.
   The DHC problem is to determine if G has a directed Hamiltonian Cycle.
   Theorem: CNF (Conjunctive Normal Form)
   satisfiability DHC DHC is NP-hard.

3. Travelling Salesperson Decision Problem (TSP) :
   1. The problem is to determine if a complete directed graph G = (V,E) with edge costs C(u,v) has a tour of cost at most

Theorem: Directed Hamiltonian Cycle (DHC) TSP

2. But from problem (2) satisfiability DHC
   Satisfiability TSP TSP is NP-hard.

Sum of subsets

TheproblemistodetermineifA={a1,a2,……,an}(a1,a2,………,anarepositiveintegers) has a subset S that sum s to a given integer M.

Scheduling identical processors

Let Pi1 ≤ i ≤m be identical processors or
machines Pi. Let Ji 1≤ i ≤ n be n jobs.
Jobs Ji requires ti processing time

A schedule S is an assignment of jobs to processors.

For each job Ji, S specifies the time interval s and the processors on which this job i is to be processed.
A job cannot be processed by more than one process or at any given time.

The problem is to find a minimum finish time non-preemptive schedule. The finish time of S is FT(S) = max {Ti}1≤i≤m. Where Ti is the time at which processor Pi finishes processing all jobs (or job segments) assigned to it

An NP-hard problem L cannot be solved in deterministic polynomial time.

By placing enough restrictions on any NP hard problem, we can arrive at a polynomials solvable problem.

Examples

CNF-Satisfy ability with at most three literals per clause is NP-hard. If each clause is restricted to have at most two literals then CNF-satisfy ability is polynomial solvable. Generating optimal code for a parallel assignment statement is NP-hard, However if the expressions are restricted to be simple variables, then optimal code can be generated in polynomial time.

Generating optimal code for level one directed a-cyclic graphs is NP-hard but optimal code for trees can be generated in polynomial time.

Determining if a planner graph is three color able is NP-Hard

Todetermineifitistwocolorableisapolynomialcomplexityproblem.
(Weonlyhavetoseeifitisbipartite)

General definitions - P, NP, NP-hard, NP-easy, and NP-complete... - Polynomial-time reduction • Examples of NP-complete problems

P - Decision problems (decision problems) that can be solved in polynomial time - can be solved “efficiently”

NP - Decision problems whose “YES” answer can be verified in polynomial time, if we already have the proof (or witness)

Co-NP - Decision problems whose “NO” answer can be verified in polynomial time, if we already have the proof (or witness)
E.g. the satisfy ability problem (SAT) - Given a Boolean formula is it possible to assign the input x1...x9, so that the formula evaluates to TRUE?

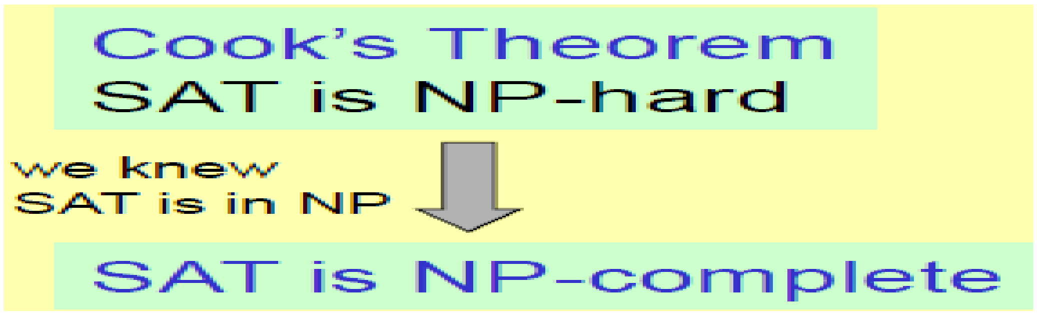
If the answer is YES with a proof (i.e. an assignment of input value), then we can check the proof in polynomial time (SAT is in NP). We may not be able to check the NO answer in polynomial time. (Nobody really knows.)

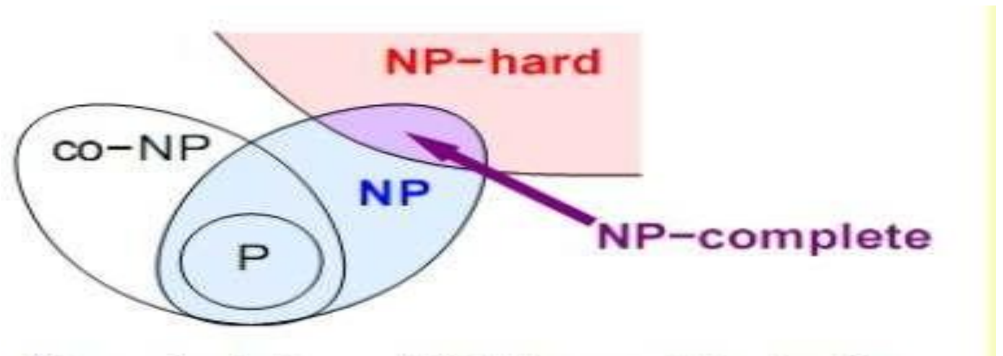
- NP-hard

A problem is NP-hard iff an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NPNP-hard problems are at least *as hard as* NP problems

- NP-complete

A problem is NP-complete if it is NP-hard, and is an element of NP (NP-easy)





More of what we *think* the world looks like.

- The classes N P-hard and N P-complete
  - Polynomial complexity
    - An algorithm A is of polynomial complexity if there exists a polynomial $p()$ such that the computation time of A is $O(p(n))$ for every input of size n

  Definition 4 P N

  is the set of all decision problems solvable by deterministic algorithms in polynomial time.

  is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

  - Since deterministic algorithms are a special case of nondeterministic algorithms, P ⊆ N P
  - An unsolved problem in computer science is: Is P = N P or is P N P?
  - Cook formulated the following question: Is there any single problem in N P such that if we showed it to be in P, then that would imply that P = N P? This led to Cook's theorem as follows:

  Theorem 1 *Satisfiability is in* P *if and only if* P = N P.

- Reducibility
  - Show that one problem is no harder or no easier than another, even when both problems are decision problems

  Definition 5 *Let A and B be problems. Problem A* ***reduces*** *to B (written as $A \propto B$) if and only if there is a way to solve A by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.*
    - If we have a polynomial time algorithm for B, then we can solve A in polynomial time
    - Reducibility is transitive
      - $A \propto B \wedge B \propto C \Rightarrow A \propto C$

  Definition 6 *Given two sets A and $B \in$* ***N*** *and a set of functions* ***F*** *:* ***N*** *→* ***N*** *, closed under composition, A is called* ***reducible*** *to B $(A \propto B)$ if and only if*

$$\exists f \in \boldsymbol{F} \;\mid\; \forall x \in \boldsymbol{N}\,,\; x \in A \Leftrightarrow f(x) \in B$$

- Procedure is called polynomial-time *reduction algorithm* and it provides us with a way to solve problem A in polynomial time
  - Also known as *Turing reduction*
  - Given an instance α of A, use a polynomial-time reduction algorithm to transform it to an instance β of B
  - Run the polynomial-time decision algorithm on instance β of B
  - Use the answer of β as the answer for α
  - Reduction from squaring to multiplication
    - All we know is to add, subtract, and take squares
    - Product of two numbers is computed by

$$2 \times a \times b = (a + b)^2 - a^2 - b^2$$

    - Reduction in the other direction: if we can multiply two numbers, we can square a number
  - Computing $(x + 1)^2$ from x^2
    - For efficiency sake, we want to avoid multiplication
  - Turing reductions *compute* the solution to one problem, assuming the other problem is easy to solve
- Polynomial-time many-one reduction
  - Converts instances of a decision problem A into instances of a decision problem B
  - Written as $A \leq_m B$; A is many-one reducible to B
  - If we have an algorithm N which solves instances of B, we can use it to solve instances of A in
    - Time needed for N plus the time needed for reduction
    - Maximum of space needed for N and the space needed for reduction
  - Formally, suppose A and B are formal languages over the alphabets Σ and Γ
    - A many-one reduction from A to B is a total computable function $f : \Sigma^* \rightarrow \Gamma^*$ with the property

$$\omega \in A \Leftrightarrow f(\omega) \in B, \ \forall \omega \in \Sigma^*$$

    - If such an f exists, A is many-one reducible to B
  - A class of languages $\boldsymbol{C}$ is *closed* under many-one reducibility if there exists no reduction from a language in
    $\boldsymbol{C}$ to a language outside $\boldsymbol{C}$
    - If a class is closed under many-one reducibility, then many-one reduction can be used to show that a problem is in $\boldsymbol{C}$ by reducing a problem in $\boldsymbol{C}$ to it
    - Let $S \subset P(\boldsymbol{N})$ (power set of natural numbers), and $\leq$ be a reduction, then S is called closed under $\leq$ if

$$\forall s \in S \ \forall A \in \boldsymbol{N} \ \ A \leq S \ \Leftrightarrow A \in S$$

    - Most well-studied complexity classes are closed under some type of many-one reducibility, including P
      and NP
  - Square to multiplication reduction, again
    - Add the restriction that we can only use square function one time, and only at

the end

  - Even if we are allowed to use all the basic arithmetic operations, including multiplication, no reduction exists in general, because we may have to compute an irrational number like $\sqrt{2}$ from rational numbers
  - Going in the other direction, however, we can certainly square a number with just one multiplication, only at the end
  - Using this limited form of reduction, we have shown the unsurprising result that multiplication is harder in general than squaring
* Many-one reductions map *instances* of one problem to *instances* of another
  - Many-one reduction is weaker than Turing reduction
  - Weaker reductions are more effective at separating problems, but they have less power, making reductions harder to design

– Use polynomial-time reductions in opposite way to show that a problem is NP-complete

        * Use polynomial-time reduction to show that no polynomial-time algorithm can exist for problem B
        * $A \subset \boldsymbol{N}$ is called *hard* for S if
$$\forall s \in S \;\; s \leq A$$
          $A \subset \boldsymbol{N}$ is called *complete* for S if A is hard for S and A is in S
        * Proof by contradiction
            · Assume that a known problem A is hard to solve
            · Given a new problem B, similar to A
            · Assume that B is solvable in polynomial time
            · Show that every instance of problem A can be solved in polynomial time by reducing it to problem B
            · Contradiction
    – Cannot assume that there is absolutely no polynomial-time algorithm for A

Definition *A problem A is NP-hard if and only if satisfiability reduces to A (satisfiability $\propto$ A). A problem A is NP-complete if and only if A is NP-hard and $A \in \mathsf{NP}$.*

- There are NP-hard problems that are not NP-complete
- Only a decision problem can be NP-complete
- An optimization problem may be NP-hard; cannot be NP-complete
- If A is a decision problem and B is an optimization problem, it is quite possible that $A \propto B$
    * Knapsack decision problem can be reduced to the knapsack optimization problem
    * Clique decision problem reduces to clique optimization problem
- There are some NP-hard decision problems that are not NP-complete
- Example: Halting problem for deterministic algorithms
    * NP-hard decision problem, but not NP-complete
    * Determine for an arbitrary deterministic algorithm A and input I, whether A with input I ever terminates
    * Well known that halting problem is undecidable; there exists no algorithm of any complexity to solve halting problem
        · It clearly cannot be in NP
    * To show that "satisfiability $\propto$ halting problem", construct an algorithm A whose input is a propositional formula X
        · If X has n variables, A tries out all the 2^n possible truth assignments and verifies whether X is satisfiable
        · If X is satisfiable, it stops; otherwise, A enters an infinite loop
        · Hence, A halts on input X iff X is satisfiable
    * If we had a polynomial time algorithm for halting problem, then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for halting problem
    * Hence, halting problem is an NP-hard problem that is not in NP

Definition *Two problems A and B are said to be* ***polynomially equivalent*** *if and only if $A \propto B$ and $B \propto A$.*

- To show that a problem B is NP-hard, it is adequate to show that $A \propto B$, where A is some problem already known to be NP-hard
- Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto A$ and $A \propto B$, then satisfiability $\propto B$
- To show that an NP N -hard decision problem is -complete, we have just to exhibit a polynomial time nondeter- ministic algorithm for it

Polynomial time

- Problems that can be solved in polynomial time are regarded as tractable problems

1. Consider a problem that is solved in time $O(n^{100})$
   - It is polynomial time but sounds intractable
   - In practice, there are few problems that require such a high degree polynomial
2. For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another
3. The class of polynomial-time solvable problems has nice closure properties
   - Polynomials are closed under addition, multiplication, and composition
   - If the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial