THEORY OF INTRACTABLE

Swarup Kr Ghosh

In P or Not in P, That's The Question!

- We find a lot of interesting problems on algorithms which can be solved in polynomial time with respect to the input size.
- For instance,
  - sorting n numbers takes $O(n \log n)$.
  - computing the MST of a graph $G = (V, E)$ takes $O(|V| \log |V| + |E|)$.
- All of these problems have naturally originated from realistic engineering areas such as networking, databases etc.

In P or Not in P, That's The Question!

- The obvious question is:

  "can all natural problems be solved in polynomial time?"
- We still do not know the answer to this question.
- Many natural problems seem to be difficult.
  - No one knows if there exist any poly-time algorithm to solve any of these 10,000 + natural problems.
  - Worse yet, no one has a proof that no such algorithm exists either.

In P or Not in P, That's The Question!

- So, when you're asked to write a program for solving a problem for which you can't find an efficient solution, you could ...

1. Email - ask the Prof.

In P or Not in P, That’s The Question!

- So, when you’re asked to write a program for solving a problem for which you can’t find an efficient solution, you could ...

2. Give up.

In P or Not in P, That’s The Question!

- So, when you’re asked to write a program for solving a problem for which you can’t find an efficient solution, you could ...

3. Spend the next 6 months working on the problem.

In P or Not in P, That's The Question!

- So, when you're asked to write a program for solving a problem for which you can't find an efficient solution, you could …

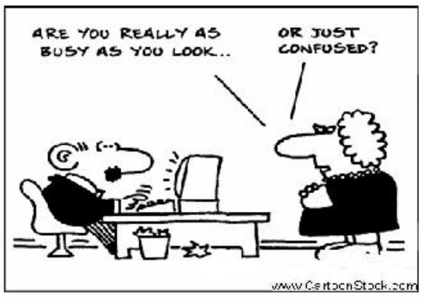
4. Give the boss a brute-force algorithm which takes a century to finish.

In P or Not in P, That's The Question!

- So, when you're asked to write a program for solving a problem for which you can't find an efficient solution, you could …

5. Mathematically show that this problem does not have a poly-time solution.

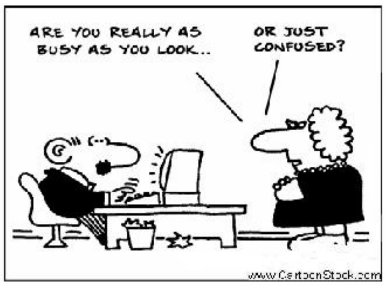
This approach is highly unlikely, it is very hard to show such a result.



In P or Not in P, That's The Question!

- So, when you're asked to write a program for solving a problem for which you can't find an efficient solution, you could ...

For the hard problems,
the best lower bound found is
$\Omega(n)$ which is totally useless!



Or, try a sixth approach...

In P or Not in P, That's The Question!

6. Mathematically show that your problem is "equivalent" to some problem which nobody knows how to solve!

- This approach makes the most sense.
- However,
  - what exactly do we mean by a "hard" problem, and
  - how do we show that two problems are "equivalently" hard?

ARE ALL PROBLEMS SOLVABLE?

Efficiency and Solvability

- A generally-accepted minimum requirement for an "efficient" algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c.
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$ $O(n^{100000})$
  - Non polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$
- Not all problems can be solved this quickly.
- There are many problems which are non-computable, i.e. no algorithm exists for solving such a problem!

Non-computability

- For example:
  - we write programs making mistakes which prevents those from terminating.
- Normally, we estimate execution time for the program and forcibly abort if it uses more time that that allocated to it.
- Obvious problems are-
  1. Non-terminating programs will waste entire time allocated.
  2. Wrong estimate of the execution time - the program may be aborted a short time before it would otherwise halt.

Non-computability

- Once it was a Million Dollar Question:
- Could we write an algorithm to determine whether an arbitrary program halts or not?
- Thanks to Alan Turing: the answer is NO!

A classical example of a problem that CANNOT BE SOLVED by ANY computer, no matter how much time is provided.

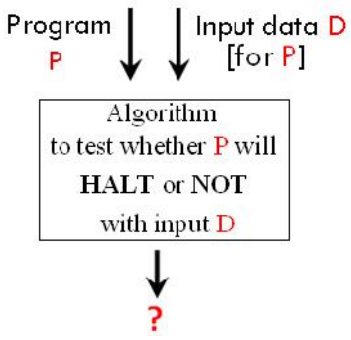
Turing's Halting Problem

- The halting problem is:
- Can we get an algorithm which, given
  1. an arbitrary program P and
  2. its input data D,

  can tell us whether or not P will eventually halt when executed with input data D ?
- Alan Turing proved that no program can be written that can solve the halting problem for all possible inputs.

Turing's Halting Problem

- A schematic of the problem

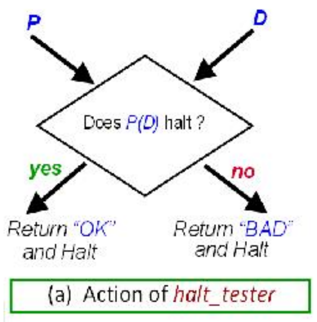


- We now provide a sketch of the proof.

The Halting Problem (HP): Proof

- Assume that there is an algorithm halt_tester for solving HP.
  - Inputs P (Program) and D (input data to P)

```
function halt_tester(P,D)
{
if P halts when executed
   with D
   then
      { return "OK",
          halt }
   else
      { return "BAD",
         halt    }
}
```

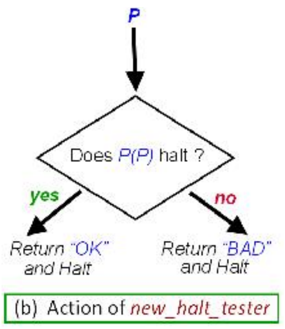


(a) Action of halt_tester

The Halting Problem (HP): Proof

- Since halt_tester tests the termination of a program P for any data D, we can write A more limited algorithm new_halt_tester which tests:
- the termination of P when the data is P itself.

```
function new_halt_tester (P)
{
/* checks whether program P
    halts if executed with P
    itself as data  */
    halt_tester (P,P)
}
```

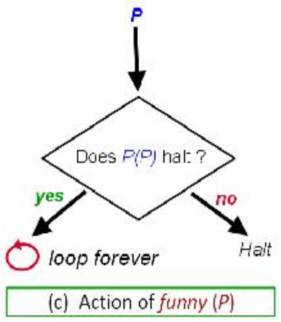


(b) Action of new_halt_tester

The Halting Problem (HP): Proof

- As algorithm halt_tester exists, and so new_halt_tester exists, we may write the following algorithm, called funny, which has just one input P.

```
function funny (P)
{
/* In order to write this module
we are assuming that halt_tester
exists */
if new_halt_tester (P)
        returns "BAD"
then  halt
else  loop forever
```



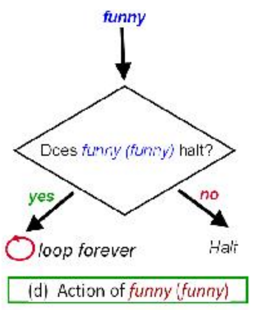
(c) Action of funny (P)

The Halting Problem (HP): Proof

- Now what happens when funny is executed with itself as data?
- It is same as (c) except that funny substituted for P.

```
function funny (funny)
{

if new_halt_tester (funny)
        returns “BAD”
then  halt
else  loop forever
```

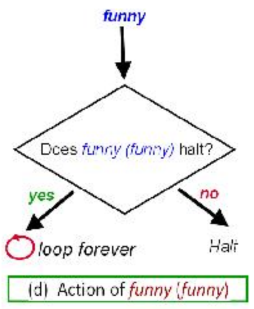


The Halting Problem (HP): Proof

- It demonstrates a contradiction:
  - if funny (funny) halts then it loops forever.
  - if funny (funny) loops forever then it halts.
  - In other words the execution of funny (funny) can neither halt nor loop for ever.

```
function funny (funny)
{

if new_halt_tester (funny)
        returns “BAD”
then  halt
else  loop forever
```



The Halting Problem (HP): Proof

- This contradiction can be resolved only by admitting that the algorithm funny cannot exist.

ĝ

- The only assumption made in deriving funny: halt_tester exists.

ĝ

- If funny cannot exist then neither can halt_ tester.

ĝ

- So, we have shown that no algorithm halt_tester for solving the halting problem can exist.

NON-COMPUTABLE

DECISION PROBLEMS

Decision Problems

- Problems where the answer is either *YES* or *NO*.
- Most of the time questions about other kinds of computational problems can be reduced to similar questions about decision problems.
    - Moreover decision problems are very simple.
- We can identify a decision problem with the subset of inputs that have answer *YES*.
- Simplified notation:
    - we write
        - $x \in Q$ in place of $Q(x) = YES$ and
        - $x \notin Q$ in place of $Q(x) = NO$.

Decision Problems

- Another perspective is that we are talking about membership queries in a set.
- Example:
    - Input: a natural number x,
        - Decision Problem:
            - Question: is x an even number?
        - Membership Problem:
            - Question: is x in $Even = \{ 0,2,4,6,\cdots \}$?
- *YES* answer on an input is accepting the input.
- *NO* answer on an input as rejecting the input.

Decision Problems

- We will look at algorithms for decision problems and discuss how efficient those algorithms are in their usage of computable resources.
- Remarks:

1. If we want to do everything formally and precisely we would need to fix a model of computation like the standard Turing machine model –
   - to precisely define what we mean by an algorithm and its usage of computational resources.
2. If we talk about computation over objects that the model cannot directly handle, we need to encode them as objects that the machine model can handle,
   - e.g. if we are using Turing machines we need to encode objects like natural numbers and graphs as binary strings.

EFFICIENT, INTRACTABLE, DECIDABLE

Efficient, Intractable, Decidable

- Efficient:
- A generally-accepted minimum requirement for an efficient algorithm is that its running time is polynomial, $O(n^c)$ for some constant c.
- Corresponding problems are called tractable problems.

Efficient, Intractable, Decidable

- Intractable:
- A problem is called intractable if it is impossible to solve it with a poly-time algorithm.
- Intractability is a property of the problem, not just of any algorithm to solve the problem.

Efficient, Intractable, Decidable

- Decidable:
- An algorithm that will take any instance of the problem as input and produce the correct answer (yes/no) as output.
- Some problems are decidable but intractable:
  - As they grow large, we are unable to solve them in reasonable time.

THREE CATEGORIES OF PROBLEMS

Three Categories of Problems Cat 1

1. Deterministic Problems: for which poly-time algorithms have been found:
   - One that always computes the correct answer.
   - Solved by conventional computers.
   - Have polynomial upper and lower bounds.

- Why polynomial?
  - if not, very inefficient.
  - nice closure properties.
  - the sum and composition of two polynomials are always polynomials too.

Three Categories of Problems Cat 2&3

2. Problems that have been proven to be intractable.
   - i.e. no polynomial time algorithm exists

     (e.g. Halting problem).

3. Problems that have not been proven to be intractable, but for which poly- time algorithms have never been found.
   - No one has found a polynomial time algorithm, but
   - No one has proven that one doesn't exist either.

- The interesting thing is that tons of problems fall into the *3rd* category and almost none into the second.

COMPLEXITY CLASS P

Complexity Class P

- P = Problems with Efficient Algorithms for finding Solutions.
- The main resource we care about is the worst-case running time of algorithms with respect to the input size,
  - i.e. the number of basic steps an algorithm takes on an input of size n.
  - The size of an input x is n if it takes n - bits of computer memory to store x, in which case we write $|x|=n$.
- *So, by efficient algorithms* we mean *algorithms that have polynomial worst-case running time*.

Nondeterministic Algorithms

- A nondeterminstic algorithm consists of
  phase 1: guessing
  phase 2: checking
- If the checking stage of a nondeterministic algorithm of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems : (must be decision problems)
  - e.g. Satisfiability Problem (SAT)
    3-SAT, Clique, Vertex Cover
    Traveling Salesperson Problem (TSP)

Complexity Class NP

- So, NP is the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a non-deterministic algorithm).
  - A deterministic computer is what we know.
  - A nondeterministic computer is one that can "guess" the right answer or solution.
- Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes.

Subset Sum Problem: Again

- Consider the subset sum problem, a problem that is easy to verify, but whose answer may be difficult to compute.
  - Given a set of integers, does some nonempty subset of them sum to 0 ?
  - The answer is "yes", because {−2, −3, −10, 15} add up to 0 can be quickly verified with 3 additions.
- However, finding such a subset in the first place could take more time.
  - Hence this problem is in NP (quickly checkable).
  - But not necessarily in P (quickly solvable).

Nondeterministic Operations & Functions

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nonderministic searching algorithm:

```
j ← choice(1 : n)   /* guessing */
if A(j) = x then success   /* checking */
             else failure
```

Nondeterministic Operations & Functions

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal.
- The time required for *choice(1 : n)* is O(1).
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation.

General Definitions

Problems
- Decision problems (yes/no)
- Optimization problems (solution with best score)

P - Decision problems that can be solved by deterministic algorithm in polynomial time
- can be solved "efficiently"

NP - Decision problems which can be solved by nondeterministic algorithm in polynomial time. Whose "YES" answer can be verified in polynomial time.

co-NP - Decision problems whose "NO" answer can be verified in polynomial time.

Formal definition of NP

- A decision problem is a parameterized problem with a yes/no answer.
- A verifier for a decision problem is an algorithm V that takes two inputs:

  (1) an instance I of the decision problem and (2) a certificate C.

  - V returns "true" if C verifies that the answer to I is "yes" and "false" otherwise.
  - Furthermore, for each "yes" instance I, there must be at least one such certificate, and for each "no" instance there must be no certificates at all.
  - If V runs in time bounded by a polynomial in the size of I, then it is called a polynomial-time verifier.
- A decision problem is in NP if it has a polynomial-time verifier. Note that every problem in P is in NP in a trivial sense.

Certificate

- In computational complexity theory, a certificate (or a witness) is a string that certifies the answer to a computation, or certifies the membership of some string in a language.
  - A certificate is often thought of as a solution path within a verification process, which is used to check whether a problem gives the answer "Yes" or "No".
- In the decision tree model of computation, certificate complexity is the minimum number of the input variables of a decision tree that need to be assigned a value in order to definitely establish the value of the Boolean function.

Example: Verifier-based Definition of NP

- Let us consider the subset sum problem:
- Assume that we are given some integers, such as

  {−7, −3, −2, 5, 8},

  and we wish to know whether some of these integers sum up to zero.
  - In this example, the answer is "yes", since the subset of integers {−3, −2, 5} corresponds to the sum (−3) + (−2) + 5 = 0.
- The task of deciding whether such a subset with sum zero exists is called the subset sum problem.

Example: Verifier-based Definition of NP

- To answer if some of the integers add to zero we can create an algorithm which obtains all the possible subsets.
  - As the number of integers that we feed into the algorithm becomes larger, the number of subsets grows exponentially and so does the computation time.

Verifier-based Definition of NP

- However, notice that, if we are given a particular subset (often called a certificate), we can easily check or verify whether the subset sum is zero, by just summing up the integers of the subset.
    - So if the sum is indeed zero, that particular subset is the proof or witness for the fact that the answer is "yes".
- An algorithm that verifies whether a given subset has sum zero is called verifier.

Verifier-based Definition of NP

- A problem is said to be in NP if:

  there exists a verifier for the problem that executes in polynomial time.

    - In case of the subset sum problem, the verifier needs only polynomial time, for which reason the subset sum problem is in NP.
- The "no"-answer version of this problem is stated as:

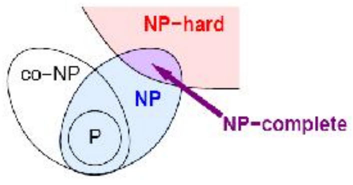
  "given a finite set of integers, does every non-empty subset have a nonzero sum?".

Verifier-based Definition of NP

- Note that the verifier-based definition of NP does not require an easy-to-verify certificate for the "no" answers.
- The class of problems with such certificates for the "no" answers is called co-NP.
- In fact, it is an open question whether all problems in NP also have certificates for the "no"-answers and thus are in co-NP.

NP Class

- P is the set of decision problems that can be solved in polynomial time.
- NP is the set of decision problems with the following property:
  - Intuitively, we can verify a YES answer quickly if we have the solution in front of us.



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

General Definitions

e.g. The satisfiability 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.)

General Definitions

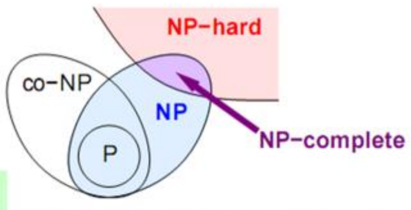
- NP-hard
  - A problem is NP-hard iff an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least *as hard as* NP problems
- NP-complete
  - A problem is NP-complete if it is NP-hard, and is an element of NP (NP-easy)

Cook's Theorem
SAT is NP-hard

we knew
SAT is in NP

$\Downarrow$

SAT is NP-complete



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

Some Concepts of NPC

- Definition of reduction: Problem A reduces to problem B ($A \propto B$) iff A can be solved by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.
- Up to now, none of the NPC problems can be solved by a deterministic polynomial time algorithm in the worst case.
- It does not seem to have any polynomial time algorithm to solve the NPC problems.

Some Concepts of NPC

- The theory of NP-completeness always considers the worst case.
- The lower bound of any NPC problem seems to be in the order of an exponential function.
- Not all NP problems are difficult. (e.g. the MST problem is an NP problem.)
- If $A, B \in$ NPC, then $A \propto B$ and $B \propto A$.
- Theory of NP-completeness

  If any NPC problem can be solved in polynomial time, then all NP problems can be solved in polynomial time. This implies NP = P

Decision Problems

- The solution is simply “Yes” or “No”.
- Optimization problems are more difficult.
- e.g. the traveling salesperson problem
  - Optimization version:

    Find the shortest tour
  - Decision version:

    Is there a tour whose total length is less than or equal to a constant c ?

Solving Optimization Problem using Decision Algorithm

- Solving TSP optimization problem by decision algorithm :
  - Give c_1 and test (decision algorithm)

    Give c_2 and test (decision algorithm)

    ⋮

    Give c_n and test (decision algorithm)
  - We can easily find the smallest c_i

Polynomial Time Reduction

- How to know another problem, A, is NP-complete?
  - To prove that A is NP-complete, reduce a known NP-complete problem to A

$$\begin{tikzcd}\boxed{\text{SAT}} \arrow[r, "\text{reduction in P(n)}"] & \boxed{\text{A}} \arrow[r, "\text{solve A}"] & \boxed{\text{YES/NO}} \arrow[ll, bend left]\end{tikzcd}$$

- Requirement for Reduction
  - Polynomial time
  - YES to A also implies YES to SAT, while NO to A also implies No to SAT (Note that A must also have short proof for YES answer)

Polynomial Time Reduction

- An example of reduction

  3-CNF: A Boolean formula in Conjunctive Normal Form having exactly 3 literals per clause.

$$\overbrace{(a\vee \underset{\text{literal}}{b}\vee c)}^{\text{clause}}\wedge(b\vee\bar{c}\vee d)\wedge(\bar{a}\vee c\vee\bar{d})$$

  3SAT: is a Boolean formula in *3-CNF* has a feasible assignment of inputs so that it evaluates to TRUE?

  - reduction from SAT to 3SAT (3SAT is NP-complete)

$$\begin{tikzcd}\boxed{\text{SAT}} \arrow[r, "\text{change into 3CNF}", "\text{(in polynomial time, without changing function value)}"'] & \boxed{\text{3SAT}} \arrow[r, "\text{solve 3SAT}"] & \boxed{\text{YES/NO}} \arrow[ll, bend left]\end{tikzcd}$$

Cook-Levin Theorem

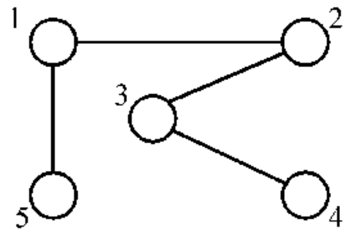
Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.

Example of NPC Problem: Vertex Cover

- Def: Given a graph G=(V, E), S is the Vertex/Node Cover if $S \subseteq V$ and for every edge $(u, v) \in E$, either $u \in S$ or $v \in S$ or both.

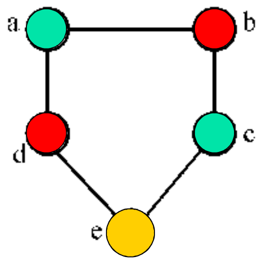


node cover :
{1, 3}
{5, 2, 4}

- Decision problem : $\exists$ S $|S| \leq K$?

Example of NPC Problem: Chromatic Number (CN)

- Def: A coloring of a graph $G=(V, E)$ is a function $f : V \rightarrow \{ 1, 2, 3,\ldots, k \}$ such that if $(u, v) \in E$, then $f(u)\neq f(v)$. The CN problem is to determine if G has a coloring for k.
- E.g.



3-colorable
$f(a)=1$, $f(b)=2$, $f(c)=1$
$f(d)=2$, $f(e)=3$

<Theorem> Satisfiability with at most 3 literals per clause (3SAT) ó CN.

Example of NPC Problem: Set Cover

- Def: $F = \{S_i\} = \{ S_1, S_2, \ldots, S_k \}$

  $\bigcup_{S_i \in F} S_i = \{ u_1, u_2, \ldots, u_n \}$

  T is a set cover of F if $T \subseteq F$ and $\bigcup_{S_i \in T} S_i = \bigcup_{S_i \in F} S_i$

The set cover decision problem is to determine if F has a cover T containing no more than c sets.

- example

  $F = \{\underset{s_1}{(a_1, a_3)}, \underset{s_2}{(a_2, a_4)}, \underset{s_3}{(a_2, a_3)}, \underset{s_4}{(a_4)}, \underset{s_5}{(a_1, a_3, a_4)}\}$

  $T = \{ s_1, s_3, s_4 \}$ set cover

  $T = \{ s_1, s_2 \}$ set cover, exact cover

Example of NPC Problem: Sum of Subset

- Def: A set of positive numbers $A = \{ a_1, a_2, \ldots, a_n \}$
  a constant C
  Determine if $\exists\ A' \subseteq A \quad \sum_{a_i \in A'} a_i = C$
- e.g. A = { 7, 5, 19, 1, 12, 8, 14 }
  - C = 21, A′ = { 7, 14 }
  - C = 11, no solution

<Theorem> Exact Cover ó Sum of Subsets.

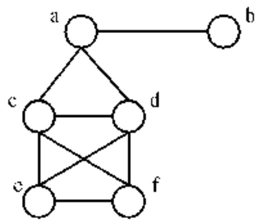
Example of NPC Problem: Partition Problem

- Def: Given a set of positive numbers $A = \{ a_1, a_2, \ldots, a_n \}$,
  determine if $\exists$ a partition P, $\sum_{i \in p} a_i = \sum_{i \notin p} a_i$
- e. g. A = {3, 6, 1, 9, 4, 11}
  partition : {3, 1, 9, 4} and {6, 11}

<Theorem> Sum of Subsets ó Partition

Example of NPC Problem: Max Clique

- Def: A complete subgraph of a graph G=(V,E) is a clique. The max (maximum) clique problem is to determine the size of a largest clique in G.
- e. g.

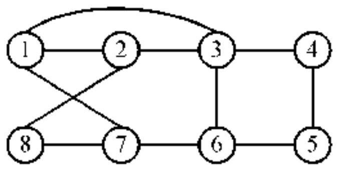


Different cliques :
{a, b}, {a, c, d}
{c, d, e, f}
Maximum clique :
(largest)
{c, d, e, f}

<Theorem> SAT ó Clique decision problem.

Example of NPC Problem: Hamiltonian Cycle

- Def: A Hamiltonian cycle is a round trip path along n edges of G which visits every vertex once and returns to its starting vertex.
- e.g.



Hamiltonian cycle : 1, 2, 8, 7, 6, 5, 4, 3, 1.

<Theorem> SAT ó Hamiltonian cycle (in a directed graph)

Traveling Salesperson Problem

- Def: A tour of a directed graph G=(V, E) is a directed cycle that includes every vertex in V. The problem is to find a tour of minimum cost.

<Theorem> Directed Hamiltonian cycle ó Traveling Salesperson decision problem.

Toward NP-Completeness

- Cook's Theorem
  The SAT problem is NP-complete.
- Once we have found an NP-complete problem, proving that other problems are also NP-complete becomes easier.
- Given a new problem Y, it is sufficient to prove that Cook's problem, or any other NP-complete problems, is polynomially reducible to Y.
- Known problem -> unknown problem

NP-Completeness Proof

The following problems are NP-complete:

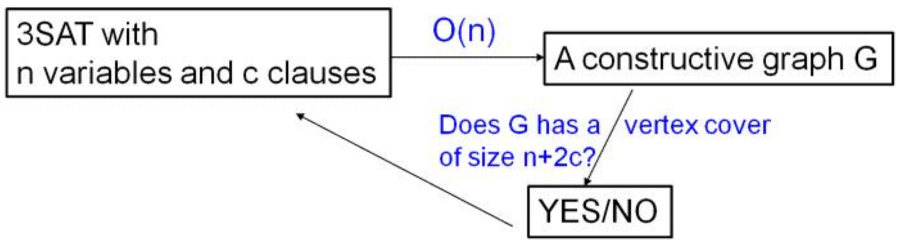
Vertex Cover(VC) and Clique.

Definition:

- A vertex cover of G=(V, E) is V'⊆V such that every edge in E is incident to some v∈V'.
- Vertex Cover(VC): Given undirected G=(V, E) and integer k, does G have a vertex cover with ≤k vertices?
- CLIQUE: Does G contain a clique of size ≥k?

Examples of NPC Problems: Vertex Cover

Reduction

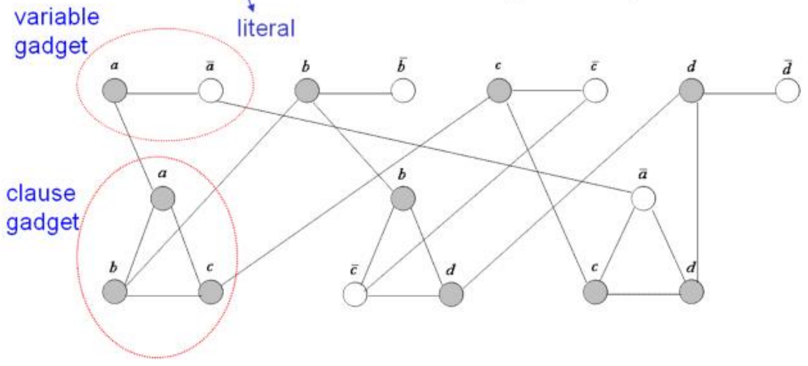


3-SAT to Vertex Cover

Vertex cover - an example of the constructive graph

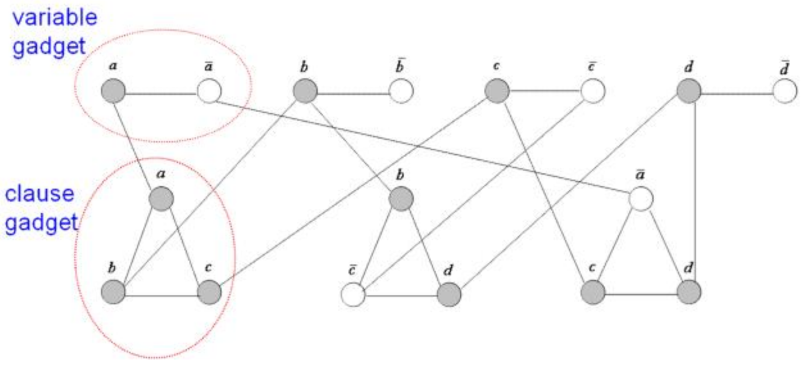
$$\overbrace{(a \vee b \vee c)}^{\text{clause}} \wedge (b \vee \bar{c} \vee d) \wedge (\bar{a} \vee c \vee \bar{d})$$

literal



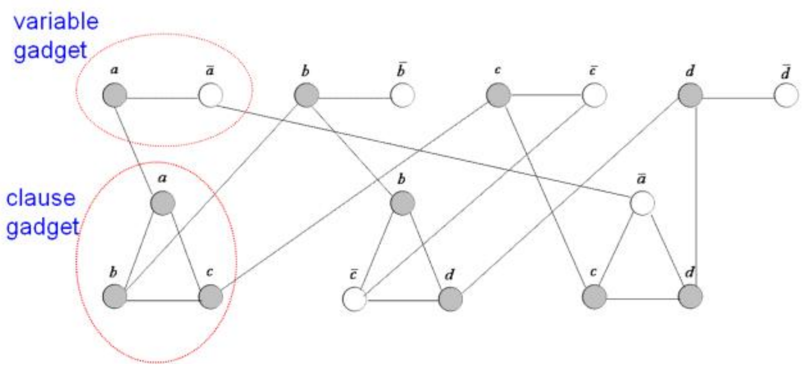
3-SAT to Vertex Cover

- Vertex cover
  - we must prove:
  the graph has a n+2c vertex cover, if and only if the 3SAT is satisfiable (to make the two problem has the same YES/NO answer)



3-SAT to Vertex Cover

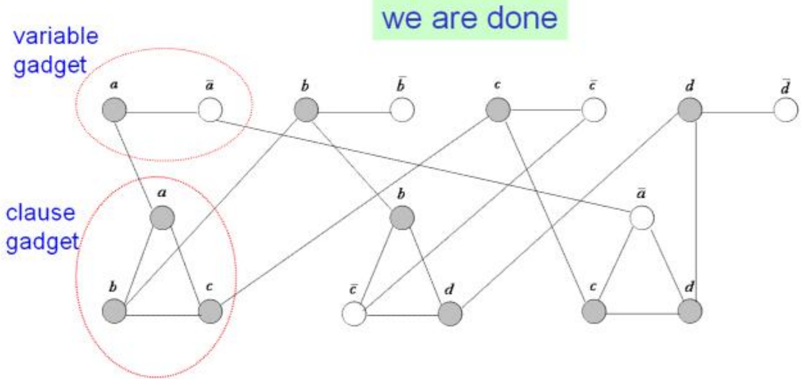
- Vertex cover
  - if the graph has a n+2c vertex cover
    1) there must be 1 vertex per variable gadget, and 2 per clause gadget
    2) in each clause gadget, set the remaining one literal to be true



3-SAT to Vertex Cover

- Vertex cover
  - if the 3SAT is satisfiable
    1) choose the TURE literal in each variable gadget
    2) choose the remaining two literal in each clause gadget

we are done



Proof: Clique is NP-Complete

Proof: To show CLIQUE is NP-complete.

Two things to be done:

- Show CLIQUE is in NP.
- Show that every language in NP is polynomial time reducible to CLIQUE.

Sufficient to give polynomial time reduction from some NP-complete language to CLIQUE.

Proof: Clique is NP-Complete

Let us try to reduce SAT to CLIQUE:

Let F be a CNF-formula.
Let $C_1,C_2,\ldots,C_k$ be the clauses in F.
Let $x_{j,1}$, $x_{j,2}$, $x_{j,m}$ be the literals of C_j.

Hint: Construct a graph G such that
F is satisfiable $\Leftrightarrow$ G has a k-clique

Proof: Clique is NP-Complete

We construct a graph G as follows:

1. For each literal $x_{j,q}$, we create a distinct vertex in G representing it.

2. G contains all edges, except those

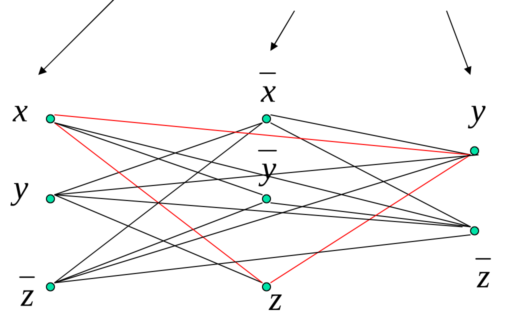
(i) joining two vertices in same clause.
(ii) joining two vertices whose literals is the negation of the others.

SAT to Clique

$$E = (x + y + \overline{z}) \bullet (\overline{x} + \overline{y} + z) \bullet (y + \overline{z})$$

- Example:

G =



- Make "column" for each of k clauses.
  - No edge within a column.
  - All other edges present except between x and x'.

- G has k-clique (k is the number of clauses in E), iff E is satisfiable. (Assign value 1 to all literals in clique)

Proof: Clique is NP-Complete

We now show that

G has a k-clique ⇔ F is satisfiable

(=>) If G has a k-clique,

1. The k-clique must have a vertex from each clause.

2. Also, no vertex will be the negation of the others in the clique.
Thus, by setting the corresponding literal to TRUE, F will be satisfied.

Proof: Clique is NP-Complete

(<=) If F is satisfiable, at least a literal in each clause is set to TRUE in the satisfying assignment.
So, the corresponding vertices forms a clique
Thus, G has a k-clique.

Finally, since G can be constructed from F in polynomial time, so we have a polynomial time reduction from 3SAT to CLIQUE.

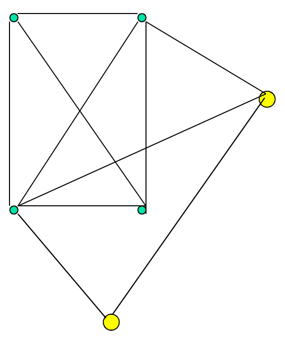
Thus, CLIQUE is NP-complete

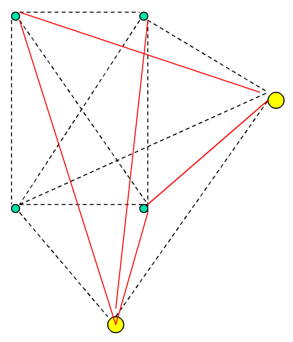
Clique to Vertex Cover

- Problem: Given undirected G=(V, E) and integer k, does G have a vertex cover with ≤k vertices?
- Theorem: The VC problem is NP-complete.
- Proof: (Reduction from CLIQUE, i.e., given CLIQUE is NP-complete)
  - VC is in NP. This is trivial since we can check it easily in polynomial time.
  - Goal: Transform arbitrary CLIQUE instance into VC instance such that CLIQUE answer is "yes" iff VC answer is "yes".

Clique to Vertex Cover

- Claim: CLIQUE(G, k) has same answer as VC ($\overline{G}$, n-k), where n = |V|.
- Observe: There is a clique of size k in G iff there is a VC of size n-k in $\overline{G}$.





Clique to Vertex Cover

- Observe: If D is a VC in $\overline{G}$, then $\overline{G}$ has no edge between vertices in V-D.

  So, we have k-clique in G $\Leftrightarrow$ n-k VC in $\overline{G}$
- Can transform in polynomial time.

End of Lecture