Week 3

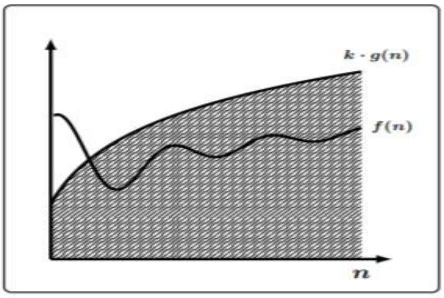
Asymptotic Notation – Big Oh

- **Big Oh Notation (upper bounds)**
  - Let $f(n)$ and $g(n)$ be any real-valued function. We say that g **eventually dominates** f if there is some constant $k > 0$ such that

$$\boxed{f(n) \leq k \cdot g(n) \qquad \text{for all `large' } n}$$

  - We say that $f(n)$ belongs to the class $O(g(n))$, read **'big oh of g'**, if $f(n)$ is eventually dominated by $g(n)$.

$$\boxed{\begin{array}{c} f(n) = O(g(n)) \\ \text{or} \\ f(n) \in O(g(n)) \end{array}}$$



<u>Complexity Classes P and NP</u>

<u>EVERYTHING THAT IS EXPONENTIAL CAN NOT BE SOLVED IN POLYNOMIAL TIME</u>

- ***Polynomial Time Problems***
  - A decision problem **X** is said to be *decidable/solvable* in **polynomial time** if there is a **deterministic Turing Machine** *M* such that:
    - ***M*** accepts ***X***
    - ***T(n)*** $\in O(n \wedge k)$ is dominated by a **polynomial function**, where

    T(n) = number of steps required to terminate on input of length **n**

  - The **complexity class P** is the class of all problems that are decidable in polynomial time
    - **P** - all problems decidable in polynomial time
    - ***NP*** – all problems decidable in non-deterministic polynomial time.
    -
- **Non-deterministic Polynomial Time Problems**
  - The class of **non-deterministic polynomial time** problems is defined similarly but replacing ***M*** with a non-deterministic *TM*, for which

    T(n) - number of steps required to terminate on input of length **n** for some possible computation

    - **P** - all problems decidable in polynomial time.
    - ***NP*** – all problems decidable in non-deterministic polynomial time.

  - Problems that belong to **NP** are those for which we can **verify** solution in polynomial time – you only need to show a single computation that accepts the input. However, to find the solution may require an **exhaustive search** of all possible computations

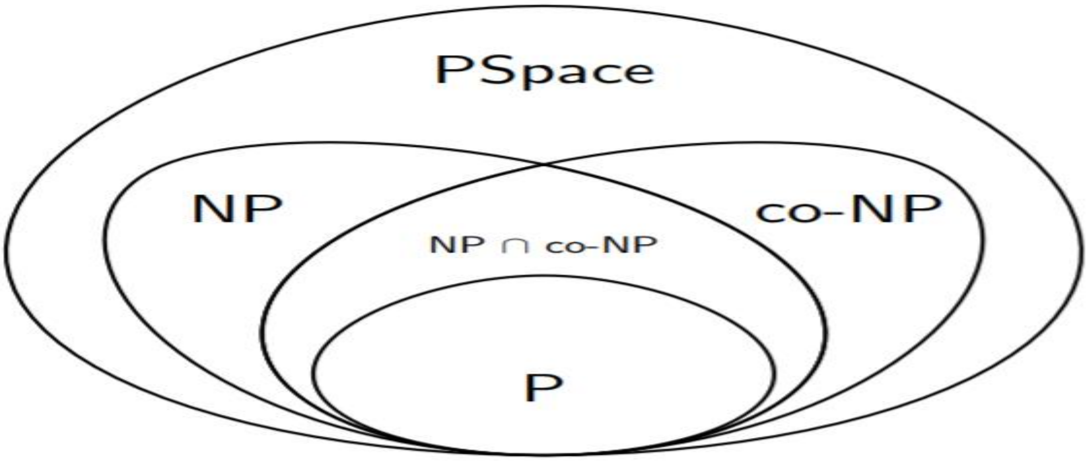
Complexity Class PSpace

- **Polynomial Space Problems**
  - A decision problem **X** is said to be ***decidable/solvable*** *in* **polynomial space** if there is a **deterministic Turing machine *M*** such that:
    - **M** *accepts* **X**
    - ***S(n)*** $\in O(n \wedge k)$ is dominated by a **polynomial function** where

  S(n) = amount of tape used for an input of length ***n***

- The **complexity class P** is the class of all problems that are decidable in polynomial time
  - ***PSpace*** *= all problems decidable in polynomial space*

Complexity hierarchy



The Boolean Satisfiability Problem

The Boolean Satisfiability Problem SAT

Input) A propositional formula F

Output) **True** if and only if F is *satisfiable*

Satisfiable so output True

P	Q	R	$(P \vee \neg R) \rightarrow \neg(\neg Q \vee R)$
True	True	True	False
True	True	False	True
True	False	True	False
True	False	False	False
False	True	True	True
False	True	False	True
False	False	True	True
False	False	False	False

This problem can be solved using power of parallel computation on non-deterministic Turing Machine,

Theorem:

- *The Boolean Satisfiability Problem* ***SAT*** *belongs to the class* ***NP***
  (there is a non-deterministic algorithm for SAT that runs in polynomial time)

Proof:

- **Step 1)** Given a propositional formula **F,** we can decide whether **F** is **satisfiable** by computing the **truth table.**
  - *However, the truth table content contains 2 ^ n rows –* ***NOT polynomial!***
- **Step 2)** However a non-deterministic algorithm can evaluate each row in a separate **parallel processor,** each of which takes at most **polynomial time.**

Week 3

Q.E.D

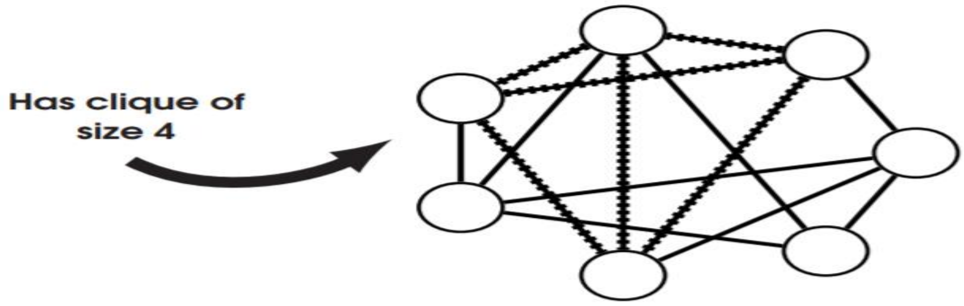
The Clique Problem

Checking if all ***K*** nodes are connected together

The Clique Problem CLIQUE

Input) An undirected graph $G = (V, E)$ and integer $k > 2$

Output) **True** if and only if G contains a clique of size k



Theorem:

- The Clique Problem **CLIQUE** belongs to the class **NP.**
  (there is a non-deterministic algorithm for CLIQUE that runs in polynomial time)

Proof:

- **Step 1)** Given an *undirected graph* **G = (V, E)** and integer **k > 2,** we can decide whether **G** contains a clique of size **k** by checking every subset of vertices of size **k**. (BASICALLY BRUTE-FORCING the answer)
  - *However, there are **n ^ n** possible subsets – **NOT** polynomial!*
- **Step 2)** However, a non-deterministic algorithm can check every possible subset of vertices in **parallel,** each of which takes at most **polynomial time**.

-

Polynomial Reduction

- **Polynomial Reduction**
  - A **polynomial reduction** from a problem ***A*** to a problem ***B*** is a function $f : \Sigma^* \rightarrow \Sigma^*$ **Computable in polynomial time**, that maps instances of ***A*** to instances of ***B*** such that

$$w \in A \quad \Longleftrightarrow \quad f(w) \in B$$

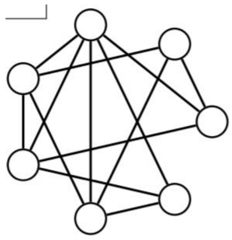
  - We say that ***A*** is reducible to ***B*** and write ***A*** <= ***B***

Nota Bene:

- For **mapping reduction,** we did not care about the time taken to compute the function ***f*** since we were not concerned about **efficiency,** since we were only interested in whether a problem was **decidable.**

Polynomial Reduction

- **The Graph Colouring Problem COLOURING**
    - **Input)** An *undirected graph* ***G = (V, E)*** and set of colours **C**
    - **Output) True** if and only if **V** can be coloured so that adjacent vertices are different colours



G = { Blue B, Green G, Red R }

Theorem The Graph Colouring problem is polynomially reducible to the Boolean Satisfiability Problem.
i.e. **COLOURING $\leq_p$ SAT**.

Proof:

Step 1) Let $G = (V, E)$ be an **undirected graph** and $C = \{\mathrm{B}, \mathrm{G}, \mathrm{R}\}$ be any **set of colours** (we are using three here for illustration)

Step 2) For each vertex $v \in V$ and each colour $i \in C$ designate a propositional variable $P_{v,i}$ that says

$$P_{v,i} \;=\; \text{vertex } v \text{ can be coloured with } i.$$

Step 3) We can write down a **set of formulas** F_G that say that the graph can be coloured with only colours from C,

- Every vertex must be coloured with **some colour**

$$(P_{v,\mathrm{B}} \vee P_{v,\mathrm{G}} \vee P_{v,\mathrm{R}}) \qquad \text{for all } v \in V$$

- No vertex can be coloured with **more than one colour**

$$\neg(P_{v,\mathrm{B}} \wedge P_{v,\mathrm{G}}) \wedge \neg(P_{v,\mathrm{B}} \wedge P_{v,\mathrm{R}}) \wedge \neg(P_{v,\mathrm{G}} \wedge P_{v,\mathrm{R}}) \qquad \text{for all } v \in V$$

- **Adjacent vertices** should be different colours

$$\neg(P_{v,\mathrm{B}} \wedge P_{u,\mathrm{B}}) \wedge \neg(P_{v,\mathrm{G}} \wedge P_{u,\mathrm{G}}) \wedge \neg(P_{v,\mathrm{R}} \wedge P_{v,\mathrm{R}}) \qquad \text{for all } (u, v) \in E$$

Step 3) This set of formulas F_G is **satisfiable** if and only if the graph G can be coloured with k colours

$$G \in \text{COLOURING} \iff F_G \in \text{SAT}$$

(this is a polynomial reduction from COLOURING to SAT)

Q.E.D.

Polynomial Reductions

Theorem The Boolean Satisfiability problem is polynomially reducible to the Clique finding problem. *i.e.* **SAT** $\leq_p$ **CLIQUE**

Proof: Given a formula F with k clauses, we want to construct a graph G_F such that F is satisfiable if and only if G_F has a k-clique.

Step 1) Let $G_F = (V, E)$ where

$$V = \{L^i : L \text{ is a literal appearing in the } i\text{th clause of } F\}$$

Step 2) Connect each vertex to all literals appearing in **different** clauses **UNLESS** they are the negation of the literal

$$(L_1^i, L_2^j) \in E \quad \Longleftrightarrow \quad i \neq j \text{ and } L_1 \not\equiv \neg L_2$$

Step 3) Note the following two observations:

Obv 1) Any clique of size k must contain a **literal from each clause**

(since literals in the same clause are not connected with an edge)

Obv 2) A clique does not contain a **literal** and its **negation**.

(since literals and their negations are not connected with an edge)

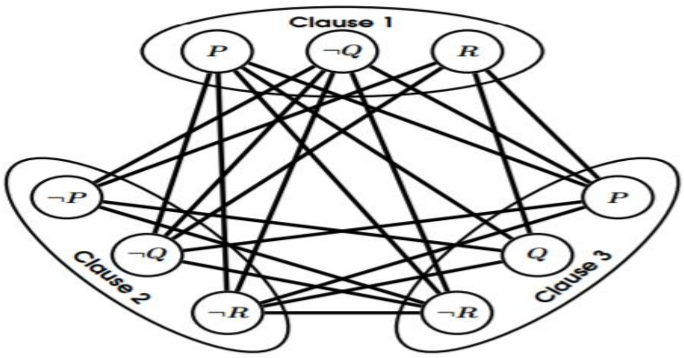
Step 5) Hence, it follows that

$$G_F \text{ contains a } k\text{-clique} \quad \Longleftrightarrow \quad F \text{ is satisfiable}$$

(just make all the literals in the clique 'true')

Q.E.D.

Polynomial Reductions



$$F = (P \vee \neg Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (P \vee Q \vee \neg R)$$

Week 3

NP-Completeness

+b

- **NP-hardness**
  - A problem X is said to be **NP-hard** if *every* problem in **NP** can be polynomially reduced to it.

$$\boxed{Y \leq_p X \qquad \text{for all } Y \in X}$$

(X is at least as hard as *every* NP-problem)

- **NP-completeness**
  - A problem X is said to be **NP-complete** if
    (i) X is **NP-hard** (lower bound),
    (ii) X also **belongs** to the class **NP** (upper bound).

NP-completeness

Theorem If Y is **NP**-hard and $Y \leq_p X$, then X is **NP**-hard.

Proof:

Step 1) If Y is **NP**-hard, then by definition $\boxed{Z \leq_p Y \qquad \text{for all } Z \in \textbf{NP}}$

Step 2) But we also have that $Y \leq_p X$, so that

$$\boxed{Z \leq_p Y \leq_p X \qquad \text{for all } Z \in \textbf{NP}}$$

Q.E.D.

A typical approach to demonstrating that a problem is **NP**-hard is to show that **SAT** is reducible to it. *i.e.* that **SAT** $\leq_p X$.

List of NP-complete Problems

- (Incomplete) List of NP-complete Problems
  - The Boolean Satisfiability Problem SAT
  - The Graph Colouring Problem COLOURING
  - The Clique Problem CLIQUE
  - The Hamilton Cycle Problem HAMILTON CYCLE
  - The Travelling Salesman Problem TSP
  - The Knapsack Problem KNAPSACK

NP-Complete

NP-Complete is a complexity class which represents the set of all problems `X` in NP for which it is possible to reduce any other NP problem `Y` to `X` in polynomial time.

Intuitively this means that we can solve `Y` quickly if we know how to solve `X` quickly. Precisely, `Y` is reducible to `X`, if there is a polynomial time algorithm `f` to transform instances `y` of `Y` to instances `x = f(y)` of `X` in polynomial time, with the property that the answer to `y` is yes, if and only if the answer to `f(y)` is yes.

Example

`3-SAT`. This is the problem wherein we are given a conjunction (ANDs) of 3-clause disjunctions (ORs), statements of the form

```
(x_v11 OR x_v21 OR x_v31) AND
(x_v12 OR x_v22 OR x_v32) AND
...                       AND
(x_v1n OR x_v2n OR x_v3n)
```

where each `x_vij` is a Boolean variable or the negation of a variable from a finite predefined list `(x_1, x_2, ... x_n)`.

It can be shown that *every NP problem can be reduced to 3-SAT*. The proof of this is technical and requires use of the technical definition of NP (*based on non-deterministic Turing machines*). This is known as *Cook's theorem*.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).

NP-hard

Intuitively, these are the problems that are *at least as hard as the NP-complete problems*. Note that NP-hard problems *do not have to be in NP*, and *they do not have to be decision problems*.

The precise definition here is that *a problem `X` is NP-hard, if there is an NP-complete problem `Y`, such that `Y` is reducible to `X` in polynomial time*.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example

The *halting problem* is an NP-hard problem. This is the problem that given a program `P` and input `I`, will it halt? This is a decision problem, but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

Week 3

To show a problem is NP complete, you need to:

Show it is in NP

In other words, given some information `C`, you can create a polynomial time algorithm `V` that will verify for every possible input `X` whether `X` is in your domain or not.

Example

Prove that the *problem of vertex covers* (that is, for some graph `G`, *does it have a vertex cover set of size* `k` *such that every edge in* `G` *has at least one vertex in the cover set*?) is in NP:

- our input `X` is some graph `G` and some number `k` (this is from the problem definition)
- Take our information `C` to be "any possible subset of vertices in graph `G` of size `k`"
- Then we can write an algorithm `V` that, given `G`, `k` and `C`, will return whether that set of vertices is a vertex cover for `G` or not, in **polynomial time**.

Then for every graph `G`, if there exists some "possible subset of vertices in `G` of size `k`" which is a vertex cover, then `G` is in `NP`.

Note that we do **not** need to find `C` in polynomial time. If we could, the problem would be in `P.

Note that algorithm `V` should work for **every** `G`, for some `C`. For every input there should **exist** information that could help us verify whether the input is in the problem domain or not. That is, there should not be an input where the information doesn't exist.

Prove it is NP Hard

This involves getting a known NP-complete problem like SAT, the set of boolean expressions in the form:

(A or B or C) and (D or E or F) and ...

where the expression is *satisfiable*, that is there exists some setting for these booleans, which makes the expression *true*.

Then **reduce the NP-complete problem to your problem in polynomial time**.

That is, given some input `X` for `SAT` (or whatever NP-complete problem you are using), create some input `Y` for your problem, such that `X` is in SAT if and only if `Y` is in your problem. The function `f : X -> Y` must run in **polynomial time**.

In the example above, the input `Y` would be the graph `G` and the size of the vertex cover `k`.

For a *full proof*, you'd have to prove both:

- that `X` is in `SAT` => `Y` in your problem
- and `Y` in your problem => `X` in `SAT`.

marcog's answer has a link with several other NP-complete problems you could reduce to your problem.

Footnote: In step 2 (**Prove it is NP-hard**), reducing another NP-hard (not necessarily NP-complete) problem to the current problem will do, since NP-complete problems are a subset of NP-hard problems (that are also in NP).