



# Hands-on Session (pre-requisites)







- **SAT solvers (Zchaff or MiniSAT) installed in the machines.**
  - **Download Zchaff from <http://cse.iitkgp.ac.in/~bdcaa/fs2020/> , Unzip and make.**
- **BDD solvers (CUDD) installed in the machines.**
  - **Download CUDD version 3.0.0 from <http://davidkebo.com/cudd> and Untar.**
  - **Run the following**
    - **./configure**
    - **make**
    - **make check**
    - **sudo make install**
  - **Include CUDD include libraries in C/C++ path**
    - **export CPATH=/`<path-to-cudd>`/:/`<path-to-cudd>`/cudd:/`<path-to-cudd>`/util/**
  - **To build add -lcudd -lutil -lm to the gcc command**
  - **To visualize install graphviz and run command : `dot -Tpng your_filename.dot -o output.png`**

# Conjunctive Normal Form

- In Boolean logic, a formula is in conjunctive normal form (CNF) or clausal normal form if it is a conjunction of one or more clauses, where a clause is a disjunction of literals.
- For example,  
$$(\neg C \vee A) \wedge (\neg C \vee B) \wedge (\neg A \vee \neg B \vee C)$$
- The input of SAT solvers are a set of clauses in CNF.
- We need to model the problem with a set of literals and express the constraints in terms of clauses made of those literals.
- Tseytin transformation takes an input of an arbitrary combinatorial logic circuit and produces a Boolean formula in CNF, which can be solved by a SAT-solver.

# Circuit to CNF Representation

Characteristic function

Type	Operation	CNF Sub-expression
 <b>AND</b>	$C = A \cdot B$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee \bar{C}) \wedge (B \vee \bar{C})$
 <b>NAND</b>	$C = \overline{A \cdot B}$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee C) \wedge (B \vee C)$
 <b>OR</b>	$C = A + B$	$(A \vee B \vee \bar{C}) \wedge (\bar{A} \vee C) \wedge (\bar{B} \vee C)$
 <b>NOR</b>	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\bar{A} \vee \bar{C}) \wedge (\bar{B} \vee \bar{C})$
 <b>NOT</b>	$C = \bar{A}$	$(\bar{A} \vee \bar{C}) \wedge (A \vee C)$
 <b>XOR</b>	$C = A \oplus B$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee B \vee \bar{C}) \wedge (A \vee \bar{B} \vee C) \wedge (\bar{A} \vee B \vee C)$

## AND Gate to CNF:

$$C \Leftrightarrow A \wedge B$$

$$\equiv (C \Rightarrow A \wedge B) \wedge (A \wedge B \Rightarrow C)$$

$$\equiv (\neg C \vee (A \wedge B)) \wedge (\neg(A \wedge B) \vee C)$$

$$\equiv (\neg C \vee A) \wedge (\neg C \vee B) \wedge$$

$$(\neg A \vee \neg B \vee C)$$

# DIMACS Format

- A file format which the SAT-solver takes as its input.
- A file can start with some comment lines. These are just text lines that start with a lower case "c". Example:

```
c This is a comment line
```

- After the initial comments, the next line of the file must tell how many variables ( $V$  a positive integer) and how many clauses ( $N$  a positive integer) in this CNF format:

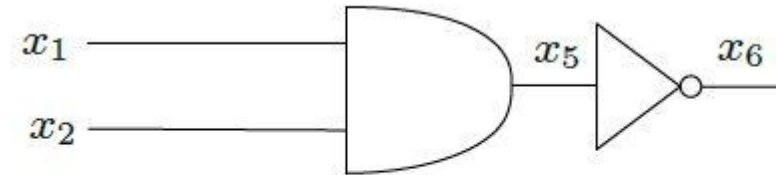
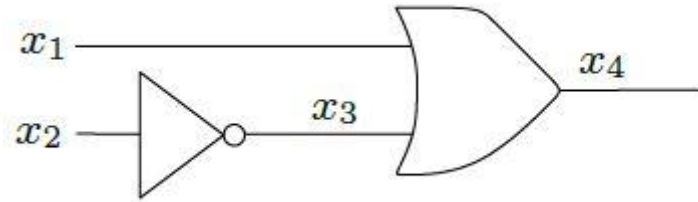
```
p cnf V N
```

- The next  $N$  lines of the file each specify one single clause. DIMACS format assumes your variables are  $x_1, x_2, x_3, \dots, x_n$ . You specify a positive literal (like  $x_2$  or  $x_7$ ) in this clause with a positive integer (in this case, 2 or 7).
- Specify a negative, complemented literal with a negative integer( so  $\neg x_5$  is -5 and  $\neg x_{23}$  is -23).
- End each clause line with a 0.
- $(x_1 + \neg x_3) (x_2 + x_3 + \neg x_1)$  in DIMACS format looks like the following snippet.

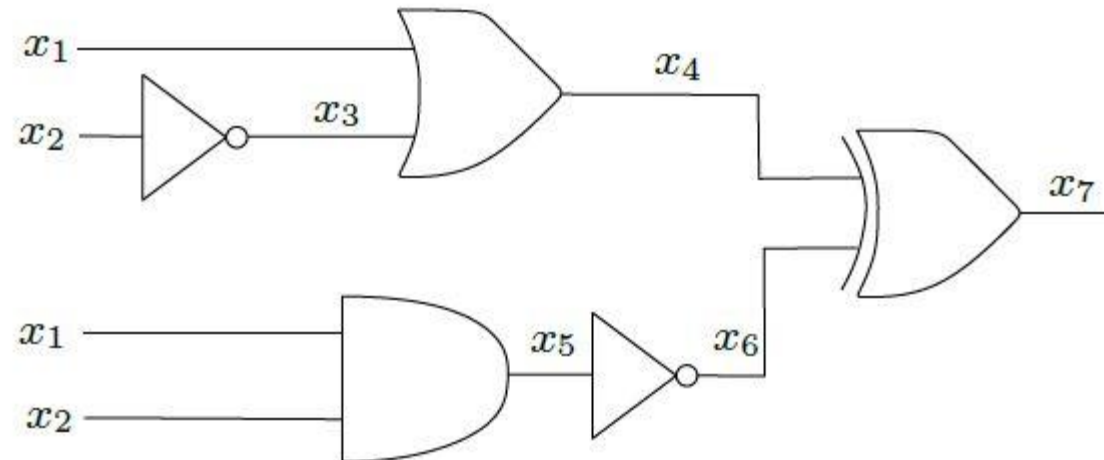
```
c Comment line begins by 'c'  
c This is second comment line  
p cnf 3 2  
1 -3 0  
2 3 -1 0
```

# Equivalence Checking of Two Circuits Using SAT

- Transform the circuits into CNF.



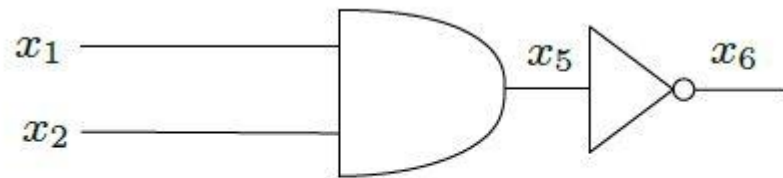
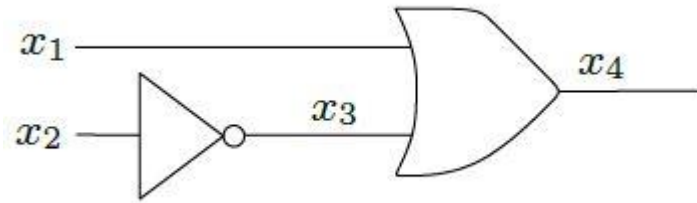
- Are these two circuits equivalent?









$x_4$	$x_6$	$x_7$
0	0	0
0	1	1
1	0	1
1	1	0

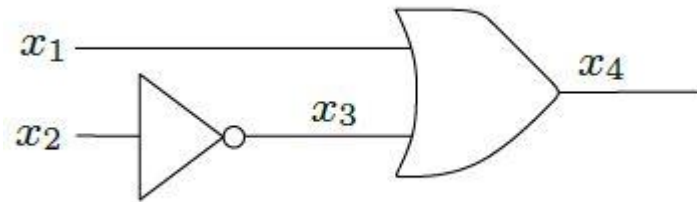


# Convert the Circuits to CNF



Type	Operation	CNF Sub-expression
 <b>AND</b>	$C = A \cdot B$	$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee \bar{C}) \wedge (B \vee \bar{C})$
 <b>NAND</b>	$C = \overline{A \cdot B}$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee C) \wedge (B \vee C)$
 <b>OR</b>	$C = A + B$	$(A \vee B \vee \bar{C}) \wedge (\bar{A} \vee C) \wedge (\bar{B} \vee C)$
 <b>NOR</b>	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\bar{A} \vee \bar{C}) \wedge (\bar{B} \vee \bar{C})$
 <b>NOT</b>	$C = \bar{A}$	$(\bar{A} \vee \bar{C}) \wedge (A \vee C)$
 <b>XOR</b>	$C = A \oplus B$	$(\bar{A} \vee \bar{B} \vee \bar{C}) \wedge (A \vee B \vee \bar{C}) \wedge (A \vee \bar{B} \vee C) \wedge (\bar{A} \vee B \vee C)$

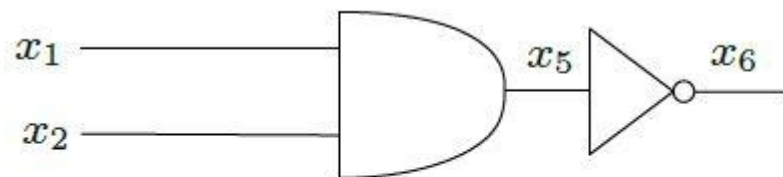
# CNF Formulations of the Circuits



1.  $X_3 \Leftrightarrow \neg X_2$   
 $(X_2 \vee X_3) \wedge (\neg X_2 \vee \neg X_3)$

2.  $X_4 \Leftrightarrow X_1 \vee X_3$   
 $(X_1 \vee X_3 \vee \neg X_4) \wedge (\neg X_1 \vee X_4) \wedge (\neg X_3 \vee X_4)$

3.  $X_5 \Leftrightarrow X_1 \wedge X_2$   
 $(\neg X_1 \vee \neg X_2 \vee X_5) \wedge (X_1 \vee \neg X_5) \wedge (X_2 \vee \neg X_5)$



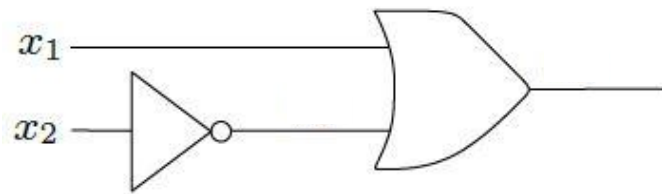
4.  $X_6 \Leftrightarrow \neg X_5$   
 $(X_6 \vee X_5) \wedge (\neg X_6 \vee \neg X_5)$

5.  $X_4 \oplus X_6$   
 $(X_4 \vee X_6) \wedge (\neg X_4 \vee \neg X_6)$

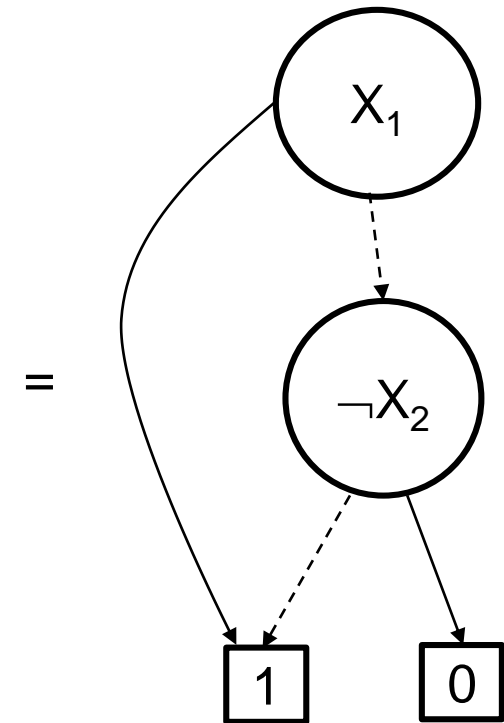
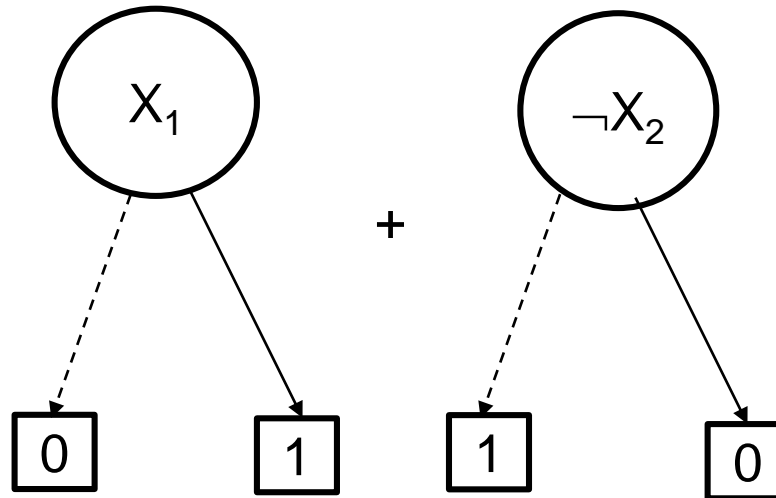


# Gate Level Circuit to BDDs

- Each input of the circuit is a BDD.
- Each gate becomes an operator that produces a new BDD.
- Example:



$$f = X_1 + \neg X_2$$



BDD for f

# Using CUDD

- CUDD is a C/C++ library for creating different types of decision diagrams (BDDs, ZDDs, ADDs).

- In order to use CUDD you must include two header files

```
#include "cudd.h"
```

```
#include "util.h"
```

- You should link `libcudd.a`, `libmtr.a`, `libst.a`, and `libutil.a` to your executable.

```
gcc -o main main.c -lcudd -lutil -lm
```

- To use the functions in the CUDD package, one has first to Initialize a `DdManager` using `Cudd_Init()`

```
DdManager *manager;
```

```
manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
```

- The constant 1 is returned by `Cudd_ReadOne`. The BDD logic 0 is obtained by complementation (`Cudd_Not`) of the constant 1

# Using CUDD

- CUDD has a built-in garbage collection system. When a BDD is not used anymore, its memory can be reclaimed.
- To facilitate the garbage collector, we need to “reference” and “dereference” each node in our BDD:

```
Cudd_Ref(DdNode*)
```

```
Cudd_RecursiveDeref(DdNode*)
```

- The DdNode is the core building block of BDDs. New DdNodes can be created using the Cudd\_bddNewVar function.

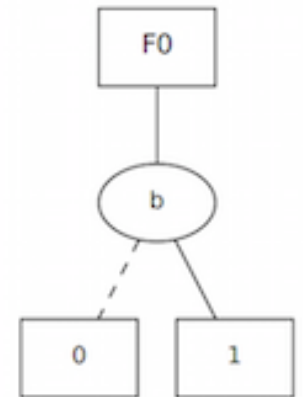
```
DdNode *bdd = Cudd_bddNewVar(DdManager*)
```

- Sample Program

```
// This program creates a single BDD variable
int main (int argc, char *argv[])
{
    DdManager *gbm; /* Global BDD manager. */
    char filename[30];

    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
    DdNode *bdd = Cudd_bddNewVar(gbm); /*Create a new BDD variable*/
    Cudd_Ref(bdd); /*Increases the reference count of a node*/
    bdd = Cudd_BddToAdd(gbm, bdd); /*Convert BDD to ADD for display */

    sprintf(filename, "./bdd/graph.dot"); /*Write .dot filename*/
    write_dd(gbm, bdd, filename); /*Write the dd to a file*/
    Cudd_Quit(gbm);
    return 0;
}
```



# BDD of Boolean functions

- CUDD has inbuilt functions for expressing Boolean operations.

`Cudd_bddXor` (DdManager\*, DdNode\*, DdNode\*)

`Cudd_bddAnd` (DdManager\*, DdNode\*, DdNode\*)

`Cudd_bddOr` (DdManager\*, DdNode\*, DdNode\*)

`Cudd_bddXnor` (DdManager\*, DdNode\*, DdNode\*)

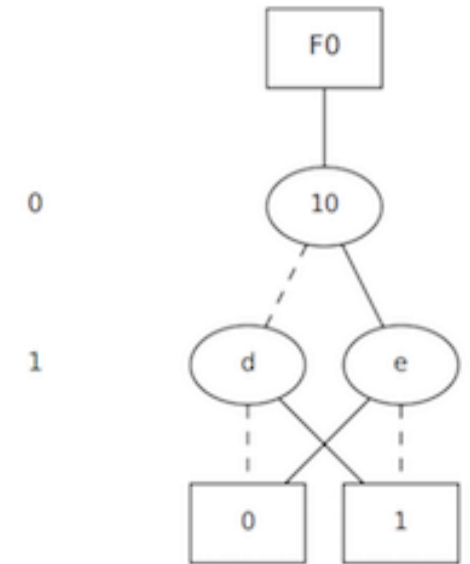
`Cudd_bddNand` (DdManager\*, DdNode\*, DdNode\*)

`Cudd_bddNor` (DdManager\*, DdNode\*, DdNode\*)

`Cudd_Not` (DdNode\*)

# Implementing XOR Using CUDD

```
int main (int argc, char *argv[])  
  
{  
    char filename[30];  
    DdManager *gbm; /* Global BDD manager. */  
    gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);  
    DdNode *bdd, *x1, *x2;  
    x1 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x1*/  
    x2 = Cudd_bddNewVar(gbm); /*Create a new BDD variable x2*/  
    bdd = Cudd_bddXor(gbm, x1, x2); /*Perform XOR*/  
    Cudd_Ref(bdd); /*Update the reference count*/  
    bdd = Cudd_BddToAdd(gbm, bdd);  
    sprintf(filename, "./bdd/graph.dot"); /*Write .dot filename*/  
    write_dd(gbm, bdd, filename);  
    Cudd_Quit(gbm);  
    return 0;  
}
```



# Comparing Logic Implementations

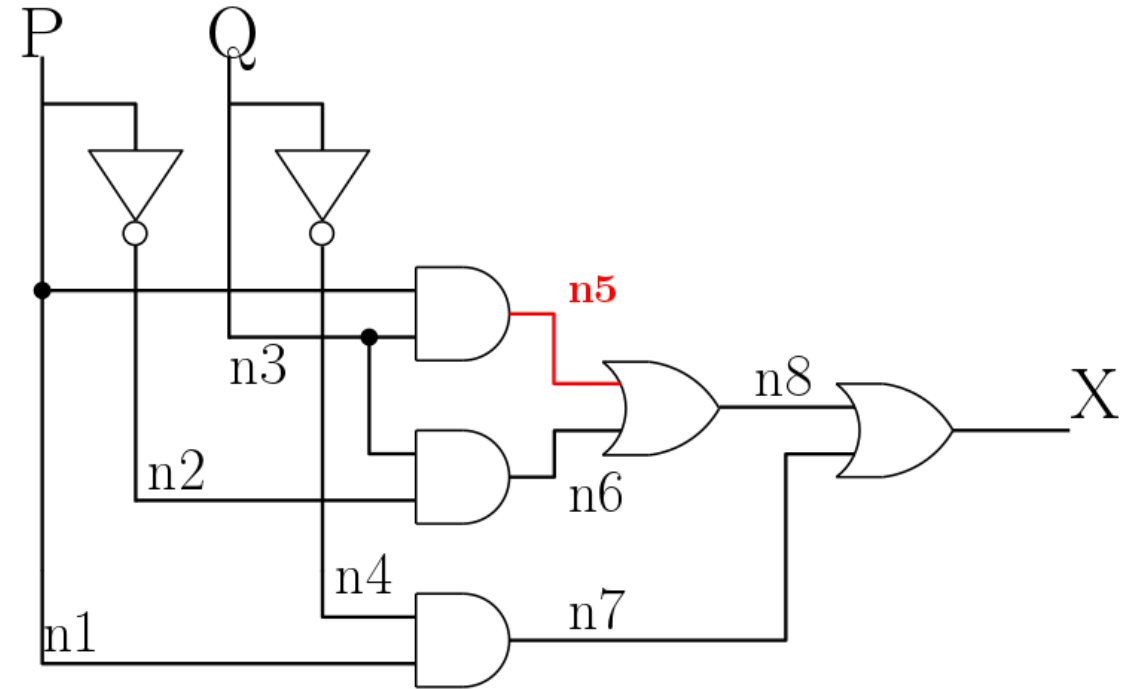
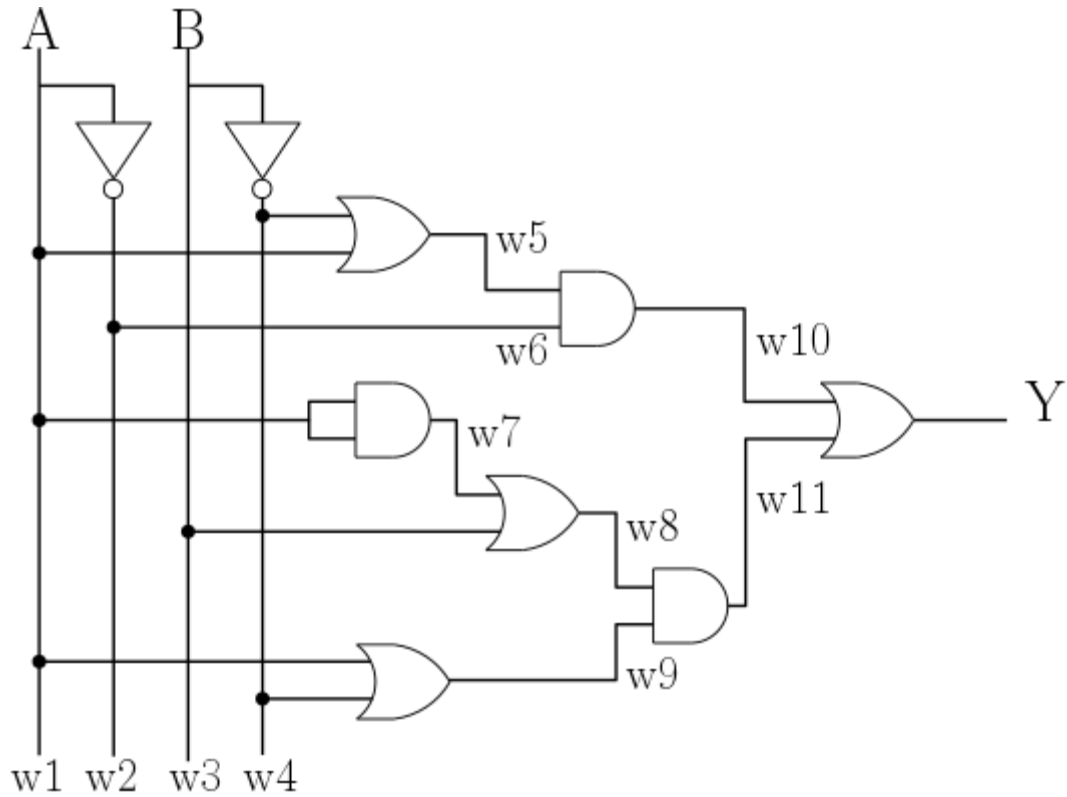
- Are two Boolean logics  $F$  and  $G$  the same?
  - Build BDD for  $F$ .
  - Build BDD for  $G$ .
  - Compare pointers to roots of  $F$ ,  $G$ .
  - If pointers are same,  $F == G$ .
- What inputs make functions  $F$ ,  $G$  give different answers?
  - Build BDD for  $F$ .
  - Build BDD for  $G$ .
  - Build the BDD for  $H = F \text{ xor } G$ .
  - Check if  $H$  is satisfiable or not.



# Tautology Checking and Satisfiability with BDDs

- **Tautology Checking**
  - The function will be reduced to one node pointing to 1.
- **Satisfiability Checking**
  - Any path from root to “1” leaf is solution!

# Observe the two circuits



Inputs A and B are equivalent to inputs P and Q.

# Assignments on SAT

1. Represent Circuit-1 and Circuit-2 in Conjunctive Normal Form (CNF).
2. Represent Circuit-1 and Circuit-2 in DIMACS format.
3. Check whether the two circuits are equivalent or not.
4. If not equivalent find the input condition(s) for which the output is SAT.

# Assignments on BDD

1. Represent Circuit-1 and Circuit-2 in as BDDs.
2. Check whether the two circuits are equivalent or not.
3. Find the input condition(s) for which the output is 1.

# Graph Coloring : SAT Formulation

We are given a graph  $G = (V,E)$

A coloring of the  $n$  vertices of the graph with  $k$  colors is a map;  $f: V \rightarrow \{1, \dots, k\}$

- $f(v)$  denotes the color of vertex  $v$

A coloring is a *proper coloring*, if, adjacent vertices must receive different colors.

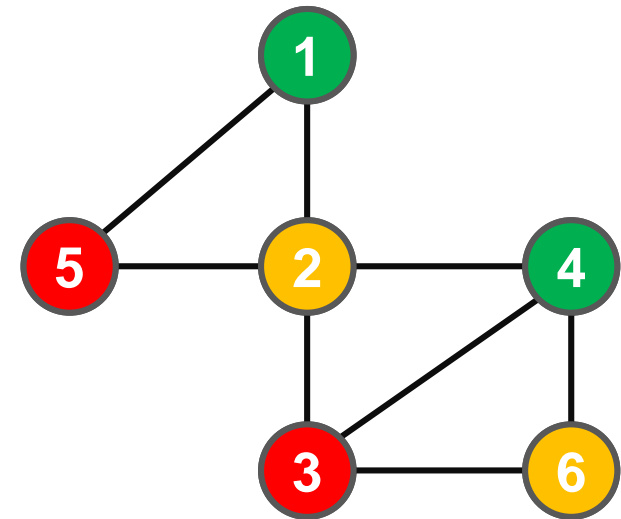
## PROBLEM

- To find the minimum  $k$  such that a proper  $k$ -coloring of  $G$  is possible

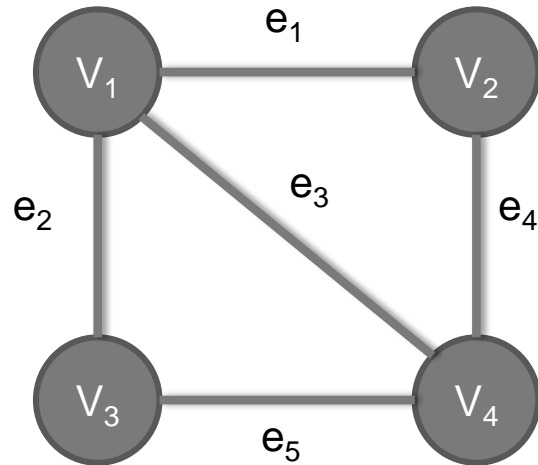
In how many ways can we color the  $n$  vertices with  $k$  colors?

Each vertex may receive one of the  $k$  colors

Number of colorings (not necessarily proper colorings) =  $k^n$



# Graph Colouring



## Types of Constraints:

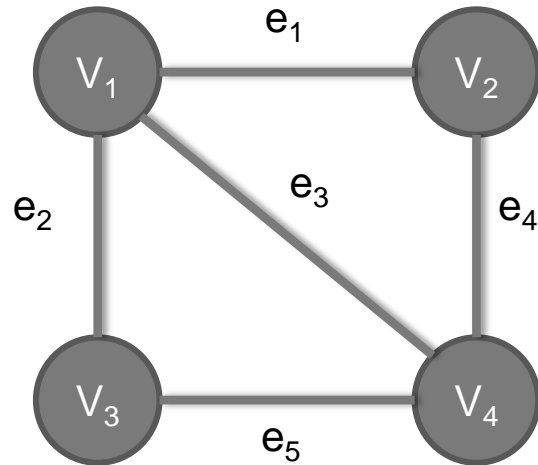
1. Vertex Constraints: A vertex must get exactly one color.
2. Edge Constraints: No two *adjacent* vertices should be colored with the same color

## Boolean State Encoding:

- Each color is given a number “i” – assume N colors
- Each vertex is given a number “j”
- For “k” colors, each vertex has “i” Boolean variables. Vertex “j” has variables numbered as  $[(j-1)*N + i]$ : For  $N = 3$  colors, Vertex  $V_3$  is represented as the three Boolean variables  $x_7$ ,  $x_8$  and  $x_9$  respectively representing that the vertex  $V_3$  is colored by colors “1”, “2” or “3”.



# Graph Colouring



## Vertex Constraints:

For Vertex  $V_1$ :

Assign it a color :  $(x_1 \vee x_2 \vee x_3)$

Exactly one color :  $(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) (\neg x_2 \vee \neg x_3)$

## Edge Constraints:

For Vertex  $V_1$ : edge  $e_1$

Color 1:  $(\neg x_1 \vee \neg x_4)$

Color 2:  $(\neg x_2 \vee \neg x_5)$

Color 3:  $(\neg x_3 \vee \neg x_6)$

What about with two colors?

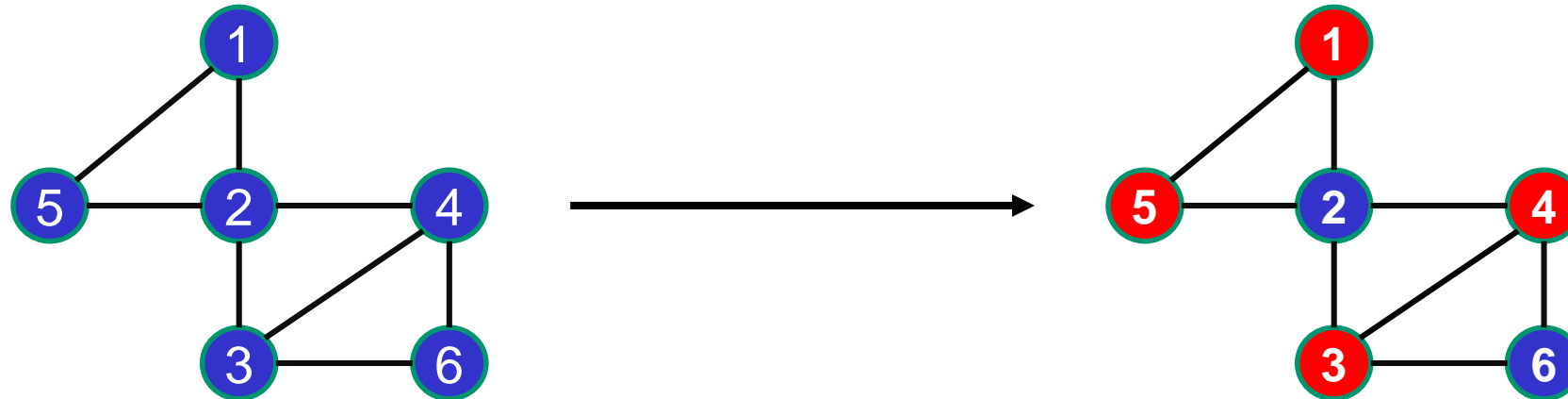
# Frequency Allocation

In mobile telephony, the frequency allocation problem is stated as follows. There are a number of transmitters deployed and each of them can transmit on any of a given set of frequencies. Different transmitters have different frequency sets. Some transmitters are so close that they cannot transmit at the same frequency, because then they would interfere with each other. You are given the frequency range of each transmitter and the pairs of transmitters that can interfere if they use the same frequency. The problem is to determine if there is any possible choice of frequencies so that no transmitter interferes with any other.

# Minimum Vertex Cover

A *vertex cover* of a graph  $G$  is a set  $S$  of vertices such that  $S$  contains at least one endpoint of every edge of  $G$ .

*PROBLEM: To find the minimum size vertex cover*



# Airline Operation

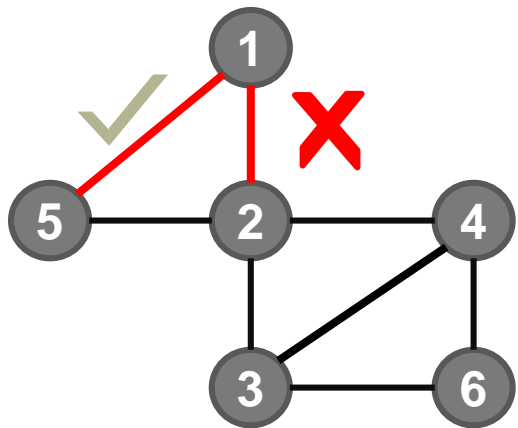
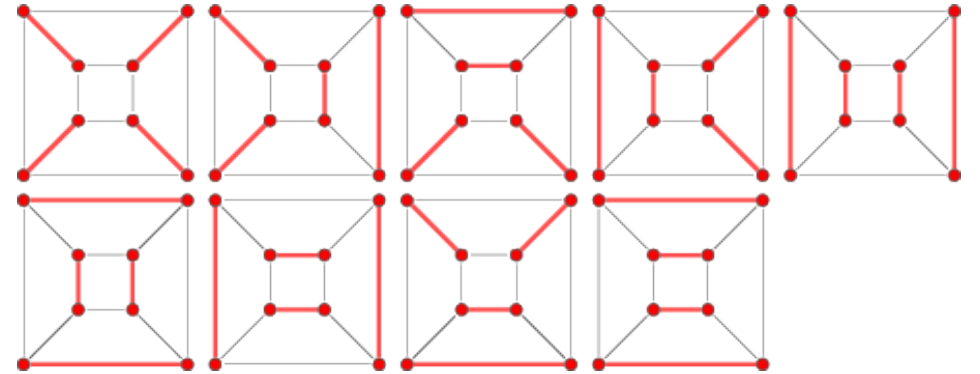
An airline company operates flights between various small (Class C/D/E) and large airports (Class B – like Chicago ORD). It wants to identify the least number of airport hubs from which it needs to operate its large aircrafts like the Boeing 747/777/787 or A-380/A-350. Come up with a SAT formulation that can help them.

- You want the minimum number of airport hubs to operate from, so that all small airports are covered.
- We discriminate between airports (some cannot act as hubs) - Large aircrafts cannot land at all airports.
- By minimizing these hubs, the aircraft saves on operating costs.

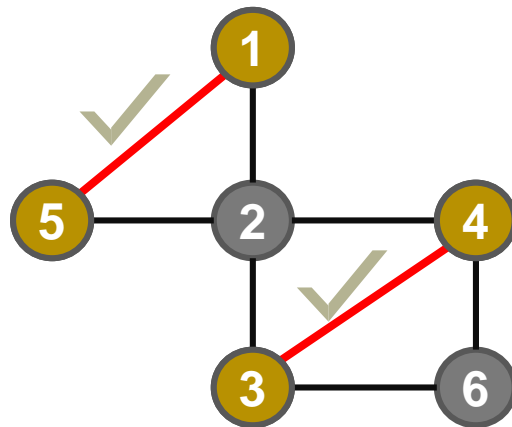
# Perfect Matching

**Matching:** A choice of edges, every vertex has at most one edge of the matching incident on it.

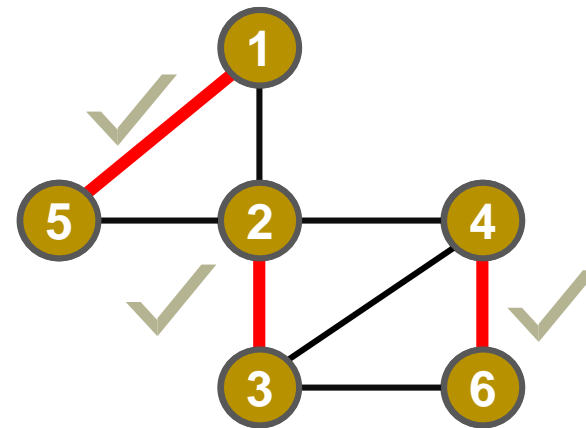
**Perfect Matching:** A matching that covers all vertices



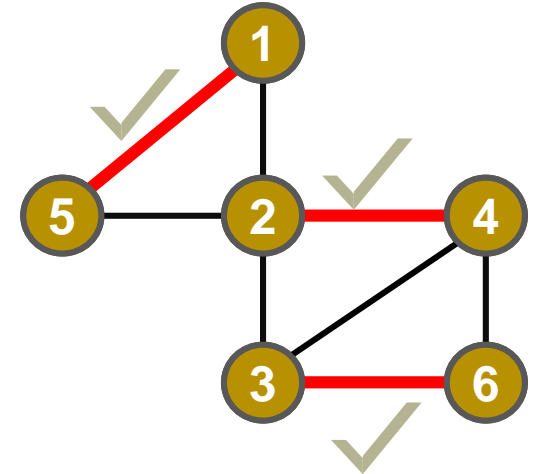
NOT a Matching  
(1)



A Matching  
(2)



Perfect Matching  
(3)



Perfect Matching  
(4)

# Scheduling a Conference

**Scheduling Speakers at a conference. There are  $N$  speakers and  $N$  time slots planned for a conference. Every speaker has a set of time slots in which there are available/unavailable. You wish to check if there is a way to assign a speaker to a preferred time slot, such that every speaker is able to speak at the conference.**