

# CS 208 : Automata Theory and Logic

Spring 2024

**Instructor** : Prof. Supratik Chakraborty

## **Disclaimer**

This is a compiled version of class notes scribed by students registered for CS 208 (Automata Theory and Logic) in Spring 2024. Please note this document has not received the usual scrutiny that formal publications enjoy. This may be distributed outside this class only with the permission of the instructor.

# Contents

<b>1</b>	<b>Propositional Logic</b>	<b>3</b>
1.1	Syntax . . . . .	3
1.2	Semantics . . . . .	4
1.2.1	Important Terminology . . . . .	6
1.3	Proof Rules . . . . .	7
1.4	Natural Deduction . . . . .	9
1.5	Soundness and Completeness of our proof system . . . . .	10
1.6	What about Satisfiability? . . . . .	12
1.7	Algebraic Laws and Some Redundancy . . . . .	13
1.7.1	Distributive Laws . . . . .	13
1.7.2	Reduction of bi-implication and implication . . . . .	13
1.7.3	DeMorgan's Laws . . . . .	13
1.8	Negation Normal Forms . . . . .	13
1.9	From DAG to NNF-DAG . . . . .	14
1.10	An Efficient Algorithm to convert DAG to NNF-DAG . . . . .	18
1.11	Conjunctive Normal Forms . . . . .	22
1.12	Satisfiability and Validity Checking . . . . .	24
1.13	DAG to Equisatisfiable CNF . . . . .	25
1.14	Tseitin Encoding . . . . .	25
1.15	Towards Checking Satisfiability of CNF and Horn Clauses . . . . .	26
1.16	Counter example for Horn Formula . . . . .	27
1.16.1	Example . . . . .	27
1.17	Davis Putnam Logemann Loveland (DPLL) Algorithm . . . . .	28
1.18	DPLL in action . . . . .	30
1.18.1	Example . . . . .	30
1.19	Applying DPLL Algorithm to Horn Formulas . . . . .	31

# Chapter 1

## Propositional Logic

In this course we look at two ways of computation: a state transition view and a logic centric view. In this chapter we begin with logic centered view with the discussion of propositional logic.

**Example.** Suppose there are five courses  $C_1, \dots, C_5$ , four slots  $S_1, \dots, S_4$ , and five days  $D_1, \dots, D_5$ . We plan to schedule these courses in three slots each, but we have also have the following requirements:

- For every course  $C_i$ , the three slots should be on three different days.
- Every course  $C_i$  should be scheduled in at most one of  $S_1, \dots, S_4$ .
- For every day  $D_i$  of the week, have at least one slot free.

	$D_1$	$D_2$	$D_3$	$D_4$	$D_5$
$S_1$					
$S_2$					
$S_3$					
$S_4$					

Propositional logic is used in many real-world problems like timetables scheduling, train scheduling, airline scheduling, and so on. One can capture a problem in a propositional logic formula. This is called as encoding. After encoding the problem, one can use various software tools to systematically reason about the formula and draw some conclusions about the problem.

### 1.1 Syntax

We can think of logic as a language which allows us to very precisely describe problems and then reason about them. In this language, we will write sentences in a specific way. The symbols used in propositional logic are given in Table 1.1. Apart from the symbols in the table we also use variables usually denoted by small letters  $p, q, r, x, y, z, \dots$  etc. Here is a short description of propositional logic symbols:

- **Variables:** They are usually denoted by smalls ( $p, q, r, x, y, z, \dots$  etc). The variables can take up only true or false values. We use them to denote propositions.
- **Constants:** The constants are represented by  $\top$  and  $\perp$ . These represent truth values true and false.

- **Operators:**  $\wedge$  is the conjunction operator (also called AND),  $\vee$  is the disjunction operator (also called OR),  $\neg$  is the negation operator (also called NOT),  $\rightarrow$  is implication, and  $\leftrightarrow$  is bi-implication (equivalence).

Name	Symbol	Read as
true	$\top$	top
false	$\perp$	bot
negation	$\neg$	not
conjunction	$\wedge$	and
disjunction	$\vee$	or
implication	$\rightarrow$	implies
equivalence	$\leftrightarrow$	if and only if
open parenthesis	(	
close parenthesis	)	

Table 1.1: Logical connectives.

For the timetable example, we can have propositional variables of the form  $p_{ijk}$  with  $i \in [5]$ ,  $j \in [5]$  and  $k \in [4]$  (Note that  $[n] = \{1, \dots, n\}$ ) with  $p_{ijk}$  representing the proposition ‘course  $C_i$  is scheduled in slot  $S_k$  of day  $D_j$ ’.

### Rules for formulating a formula:

- Every variable constitutes a formula.
- The constants  $\top$  and  $\perp$  are formulae.
- If  $\varphi$  is a formula, so are  $\neg\varphi$  and  $(\varphi)$ .
- If  $\varphi_1$  and  $\varphi_2$  are formulas, so are  $\varphi_1 \wedge \varphi_2$ ,  $\varphi_1 \vee \varphi_2$ ,  $\varphi_1 \rightarrow \varphi_2$ , and  $\varphi_1 \leftrightarrow \varphi_2$ .

### Propositional formulae as strings and trees:

Formulae can be expressed as strings over the alphabet  $\mathbf{Vars} \cup \{\top, \perp, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (, )\}$ .  $\mathbf{Vars}$  is the set of symbols for variables. Not all words formed using the alphabet qualify as propositional formulae. A string constitutes a well-formed formula (wff) if it was constructed while following the rules. Examples:  $(p_1 \vee \neg q_2) \wedge (\neg p_2 \rightarrow (q_1 \leftrightarrow \neg p_1))$  and  $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$ .

Well-formed formulas can be represented using trees. Consider the formula  $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$ . This can be represented using the parse tree in figure Figure 1.1a. Notice that while strings require parentheses for disambiguation, trees don’t, as can be seen in Figure 1.1b and Figure 1.1c.

## 1.2 Semantics

Semantics give a meaning to a formula in propositional logic. The semantics is a function that takes in the truth values of all the variables that appear in a formula and gives the truth value of the formula. Let 0 represent “false” and 1 represent “true”. The semantics of a formula  $\varphi$  of  $n$  variables is a function

$$\llbracket \varphi \rrbracket : \{0, 1\}^n \rightarrow \{0, 1\}$$

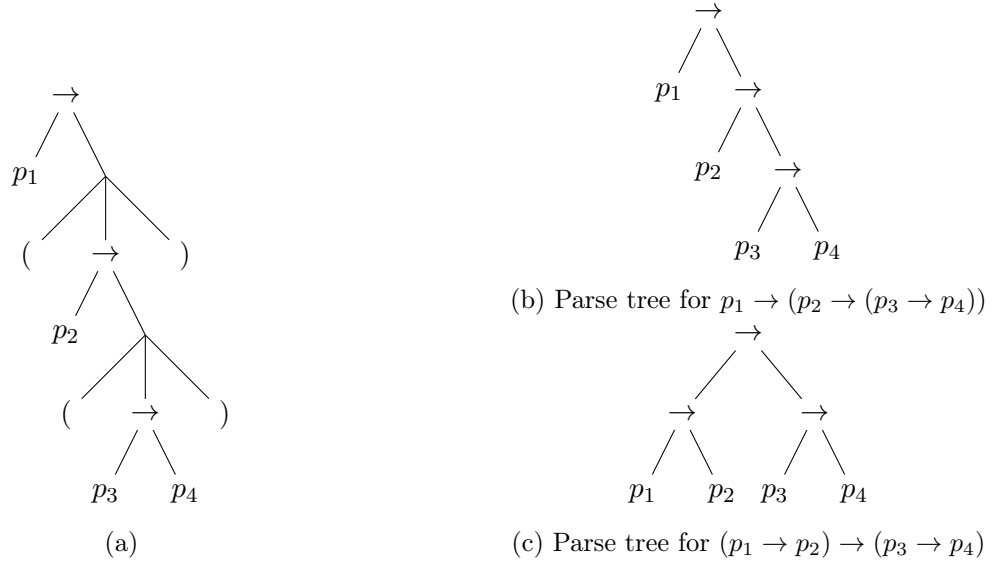


Figure 1.1: Parse trees obviate the need for parentheses.

It is often presented in the form of a truth table. Truth tables of operators can be found in table Table 1.2.

$\varphi$	$\neg\varphi$	$\varphi_1$	$\varphi_2$	$\varphi_1 \wedge \varphi_2$	$\varphi_1$	$\varphi_2$	$\varphi_1 \vee \varphi_2$
0	1	0	0	0	0	0	0
1	0	0	1	0	0	1	1
		1	0	0	1	0	1
		1	1	1	1	1	1

(a) Truth table for  $\neg\varphi$ . (b) Truth table for  $\varphi_1 \wedge \varphi_2$ . (c) Truth table for  $\varphi_1 \vee \varphi_2$ .

$\varphi_1$	$\varphi_2$	$\varphi_1 \rightarrow \varphi_2$	$\varphi_1$	$\varphi_2$	$\varphi_1 \leftrightarrow \varphi_2$
0	0	1	0	0	1
0	1	1	0	1	0
1	0	0	1	0	0
1	1	1	1	1	1

(d) Truth table for  $\varphi_1 \rightarrow \varphi_2$ . (e) Truth table for  $\varphi_1 \leftrightarrow \varphi_2$ .

Table 1.2: Truth tables of operators.

**Remark.** Do not confuse 0 and 1 with  $\top$  and  $\perp$ : 0 (false) and 1 (true) are meanings, while  $\top$  and  $\perp$  are symbols.

#### Rules of semantics:

- $\llbracket \neg\varphi \rrbracket = 1$  iff  $\llbracket \varphi \rrbracket = 0$ .
- $\llbracket \varphi_1 \wedge \varphi_2 \rrbracket = 1$  iff  $\llbracket \varphi_1 \rrbracket = \llbracket \varphi_2 \rrbracket = 1$ .
- $\llbracket \varphi_1 \vee \varphi_2 \rrbracket = 1$  iff at least one of  $\llbracket \varphi_1 \rrbracket$  or  $\llbracket \varphi_2 \rrbracket$  evaluates to 1.

- $\llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket = 1$  iff at least one of  $\llbracket \varphi_1 \rrbracket = 0$  or  $\llbracket \varphi_2 \rrbracket = 1$ .
- $\llbracket \varphi_1 \leftrightarrow \varphi_2 \rrbracket = 1$  iff at both  $\llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket = 1$  and  $\llbracket \varphi_2 \rightarrow \varphi_1 \rrbracket = 1$ .

**Truth Table:** A truth table in propositional logic enumerates all possible truth values of logical expressions. It lists combinations of truths for individual propositions and the compound statement's truth.

**Example.** Let us construct a truth table for  $\llbracket (p \vee s) \rightarrow (\neg q \leftrightarrow r) \rrbracket$  (see Table 1.3).

$p$	$q$	$r$	$s$	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1
0	0	0	1	1	1	0	0
0	0	1	0	0	1	1	1
0	0	1	1	1	1	1	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	0
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Table 1.3: Truth table of  $(p \vee s) \rightarrow (\neg q \leftrightarrow r)$ .

### 1.2.1 Important Terminology

A formula  $\varphi$  is said to (be)

- **satisfiable** or **consistent** or SAT iff  $\llbracket \varphi \rrbracket = 1$  for some assignment of variables. That is, there is at least one way to assign truth values to the variables that makes the entire formula true. Both a formula and its negation may be SAT at the same time ( $\varphi$  and  $\neg\varphi$  may both be SAT).
- **unsatisfiable** or **contradiction** or UNSAT iff  $\llbracket \varphi \rrbracket = 0$  for all assignments of variables. That is, there is no way to assign truth values to the variables that makes the formula true. If a formula  $\varphi$  is UNSAT then  $\neg\varphi$  must be SAT (it is in fact valid).
- **valid** or **tautology**:  $\llbracket \varphi \rrbracket = 1$  for all assignments of variables. That is, the formula is always true, no matter how the variables are assigned. If a formula  $\varphi$  is valid then  $\neg\varphi$  is UNSAT.
- **semantically entail**  $\varphi_1$  iff  $\llbracket \varphi \rrbracket \preceq \llbracket \varphi_1 \rrbracket$  for all assignments of variables, where  $0$  (false)  $\preceq 1$  (true). This is denoted by  $\varphi \models \varphi_1$ . If  $\varphi \models \varphi_1$ , then for every assignment, if  $\varphi$  evaluates to 1 then  $\varphi_1$  will evaluate to 1. Equivalently  $\varphi \rightarrow \varphi_1$  is valid.

- **semantically equivalent** to  $\varphi_1$  iff  $\varphi \models \varphi_1$  and  $\varphi_1 \models \varphi$ . Basically  $\varphi$  and  $\varphi_1$  have identical truth tables. Equivalently,  $\varphi \leftrightarrow \varphi_1$  is valid.
- **equisatisfiable** to  $\varphi_1$  iff either both are SAT or both are UNSAT. Also note that, semantic equivalence implies equisatisfiability but **not** vice-versa.

Term	Example
SAT	$p \vee q$
UNSAT	$p \wedge \neg p$
valid	$p \vee \neg p$
semantically entails	$\neg p \models p \rightarrow q$
semantically equivalent	$p \rightarrow q, \neg p \vee q$
equisatisfiable	$p \wedge q, r \vee s$

Table 1.4: Some examples for the definitions.

**Example.** Consider the formulas  $\varphi_1 : p \rightarrow (q \rightarrow r)$ ,  $\varphi_2 : (p \wedge q) \rightarrow r$  and  $\varphi_3 : (q \wedge \neg r) \rightarrow \neg p$ . The three formulas  $\varphi_1$ ,  $\varphi_2$  and  $\varphi_3$  are semantically equivalent. One way to check this is to construct the truth table.

On drawing the truth table for the above example, one would realise that it is laborious. Indeed, for a formula with  $n$  variables, the truth table has  $2^n$  entries! So truth tables don't work for large formulas. We need a more systematic way to reason about the formulae. That leads us to proof rules...

But before that let us get a closure on the example at the beginning of the chapter. Let  $p_{ijk}$  represent the proposition 'course  $C_i$  is scheduled in slot  $S_k$  of day  $D_j$ '. We can encode the constraints using the encoding strategy used in tutorial 1 - problem 3. That is, by introducing extra variables that bound the sum for first few variables (sum of  $i$  is atmost  $j$ ). Using this we can encode the constraints as :  $\sum_{k=1}^4 p_{ijk} \leq 1$ ,  $\sum_{j=1}^5 p_{ijk} \leq 1$ ,  $\sum_{i=1}^5 p_{ijk} \leq 1$ ,  $\sum_{k=1}^4 \sum_{i=1}^5 p_{ijk} \leq 3$ ,  $\sum_{k=1}^4 \sum_{j=1}^5 p_{ijk} \leq 3$  and  $\neg(\sum_{k=1}^4 \sum_{j=1}^5 p_{ijk} \leq 2)$ .

### 1.3 Proof Rules

After encoding a problem into propositional formula we would like to reason about the formula. Some of the properties of a formula that we are usually interested in are whether it is SAT, UNSAT or valid. We have already seen that truth tables do not scale well for large formulae. It is also not humanly possible to reason about large formulae modelling real-world systems. We need to delegate the task to computers. Hence, we need to make systematic rules that a computer can use to reason about the formulae. These are called as proof rules.

The overall idea is to convert a formula to a normal form (basically a standard form that will make reasoning easier - more about this later in the chapter) and use proof rules to check SAT etc.

Rules are represented as

$$\frac{\text{Premises}}{\text{Inferences}} \text{Connector}_{i/e}$$

- **Premise:** A premise is a formula that is assumed or is known to be true.



- **Inference:** The conclusion that is drawn from the premise(s).
- **Connector:** It is the logical operator over which the rule works. We use the subscript  $i$  (for introduction) if the connector and the premises are combined to get the inference. The subscript  $e$  (for elimination) is used when we eliminate the connector present in the premises to draw inference.

**Example.** Look at the following rule

$$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e1}$$

In the rule above  $\varphi_1 \wedge \varphi_2$  is assumed (is premise). Informally, looking at  $\wedge$ 's truth table, we can infer that both  $\varphi_1$  and  $\varphi_2$  are true if  $\varphi_1 \wedge \varphi_2$  is true, so  $\varphi_1$  is an inference. Also, in this process we eliminate (remove)  $\wedge$  so we call this AND-ELIMINATION or  $\wedge_e$ . For better clarity we call this rule  $\wedge_{e1}$  as  $\varphi_1$  is kept in the inference even when both  $\varphi_1$  and  $\varphi_2$  could be kept in inference. If we use  $\varphi_2$  in inference then the rule becomes  $\wedge_{e2}$ .

Table 1.5 summarises the basic proof rules that we would like to include in our proof system.

Connector	Introduction	Elimination
$\wedge$	$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$	$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e1} \quad \frac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e2}$
$\vee$	$\frac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i1} \quad \frac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i2}$	$\frac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \rightarrow \varphi_3 \quad \varphi_2 \rightarrow \varphi_3}{\varphi_3} \vee_e$
$\rightarrow$	$\frac{\boxed{\begin{array}{c} \varphi_1 \\ \vdots \\ \varphi_2 \end{array}}}{\varphi_1 \rightarrow \varphi_2} \rightarrow_i$	$\frac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2} \rightarrow_e$
$\neg$	$\frac{\boxed{\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}}}{\neg \varphi} \neg_i$	$\frac{\varphi \quad \neg \varphi}{\perp} \neg_e$
$\perp$		$\frac{\perp}{\varphi} \perp_e$
$\neg\neg$		$\frac{\neg\neg\varphi}{\varphi} \neg\neg_e$

Table 1.5: Proof rules.

In the  $\rightarrow_i$  rule, the box indicates that we can *temporarily* assume  $\varphi_1$  and conclude  $\varphi_2$  using no extra non-trivial information. The  $\rightarrow_e$  is referred to by its Latin name, *modus ponens*.

**Example 1.** We can now use these proof rules along with  $\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$  as the premise to conclude  $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$ .

$$\begin{array}{c}
 \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3} \wedge_{e2} \quad \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1} \wedge_{e1} \\
 \frac{\varphi_2 \wedge \varphi_3}{\varphi_2} \wedge_{e1} \quad \frac{\varphi_2 \wedge \varphi_3}{\varphi_3} \wedge_{e2} \\
 \frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i \\
 \frac{\varphi_1 \wedge \varphi_2 \quad \varphi_3}{(\varphi_1 \wedge \varphi_2) \wedge \varphi_3} \wedge_i
 \end{array}$$

## 1.4 Natural Deduction

If we can begin with some formulas  $\phi_1, \phi_2, \dots, \phi_n$  as our premises and then conclude  $\varphi$  by applying the proof rules established we say that  $\phi_1, \phi_2, \dots, \phi_n$  syntactically entail  $\varphi$  which is denoted by the following expression, also called a *sequent*:

$$\phi_1, \phi_2, \dots, \phi_n \vdash \varphi.$$

We can also infer some formula using no premises, in which case the sequent is  $\vdash \varphi$ . Applying these proof rules involves the following general rule:

We can only use a formula  $\varphi$  at a point if it occurs prior to it in the proof and if **no box enclosing that occurrence of  $\varphi$  has been closed already**.

**Example.** Consider the following proof of the sequent  $\vdash p \vee \neg p$ :

1.	$\neg(p \vee \neg p)$	assumption
2.	$p$	assumption
3.	$p \vee \neg p$	$\vee_{i_1} 2$
4.	$\perp$	$\neg_e 3, 1$
5.	$\neg p$	$\neg_i 2-4$
6.	$p \vee \neg p$	$\vee_{i_2} 5$
7.	$\perp$	$\neg_e 6, 1$
8.	$\neg\neg(p \vee \neg p)$	$\neg_i 1-7$
9.	$p \vee \neg p$	$\neg\neg_e 8$

**Example.** Proof for  $p \vdash \neg\neg p$  which is  $\neg\neg_i$ , a derived rule

1.	$p$	premise
2.	$\neg p$	assumption
3.	$\perp$	$\neg_e 1, 2$
4.	$\neg\neg p$	$\neg_i 2-3$

**Example.** A useful derived rule is *modus tollens* which is  $p \rightarrow q, \neg q \vdash \neg p$ :

1.	$p \rightarrow q$	premise
2.	$\neg q$	premise
3.	$p$	assumption
4.	$q$	$\rightarrow_e$ 3,1
5.	$\perp$	$\neg_e$ 4,2
6.	$\neg p$	$\neg_i$ 3-5

**Example.**  $\neg p \wedge \neg q \vdash \neg(p \vee q)$ :

1.	$\neg p \wedge \neg q$	premise
2.	$p \vee q$	assumption
3.	$p$	assumption
4.	$\neg p$	$\wedge_{e1}$ 1
5.	$\perp$	$\neg_e$ 3,4
6.	$p \rightarrow \perp$	$\rightarrow_i$ 3-5
7.	$q$	assumption
8.	$\neg q$	$\wedge_{e2}$ 1
9.	$\perp$	$\neg_e$ 7,8
10.	$q \rightarrow \perp$	$\rightarrow_i$ 7-9
11.	$\perp$	$\vee_e$ 2,6,10
12.	$\neg(p \vee q)$	$\neg_i$ 2-11

## 1.5 Soundness and Completeness of our proof system

A proof system is said to be sound if everything that can be derived using it matches the semantics.

**Soundness:**  $\Sigma \vdash \varphi$  implies  $\Sigma \models \varphi$

The rules that we have chosen are indeed individually sound since they ensure that if for some assignment the premises evaluate to 1, so does the inference. Otherwise they rely on the notion of contradiction and assumption. Hence, soundness for any proof can be shown by inducting on the length of the proof.

A complete proof system is one which allows the inference of *every* valid semantic entailment:

**Completeness:**  $\Sigma \models \varphi$  implies  $\Sigma \vdash \varphi$

Let's take some example of semantic entailment.  $\Sigma = \{p \rightarrow q, \neg q\} \models \neg p$ .

$p$	$q$	$p \rightarrow q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

As we can see, whenever both  $p \rightarrow q$  and  $\neg q$  are true,  $\neg p$  is true. The question now is how do we derive this using proof rules? The idea is to ‘mimic’ each row of the truth table. This means that we assume the values for  $p$ ,  $q$  and try to prove that the formulae in  $\Sigma$  imply  $\varphi$ <sup>1</sup>. And to prove an implication, we can use the  $\rightarrow_i$  rule. Here’s an example of how we can prove our claim for the first row:

1.	$\neg p$	given
2.	$\neg q$	given
3.	$(p \rightarrow q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$	$\rightarrow_i$ 3,4

Similarly to mimic the second row, we would like to show  $\neg p, q \vdash ((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ . Actually for every row, we’d like to start with the assumptions about the values of each variable, and then try to prove the property that we want.

1.	$\neg p$	given	1.	$\neg p$	given	1.	$p$	given	1.	$p$	given
2.	$\neg q$	given	2.	$q$	given	2.	$\neg q$	given	2.	$q$	given
3.	$(p \rightarrow q) \wedge \neg q$	assumption	3.	$(p \rightarrow q) \wedge \neg q$	assumption	3.	$(p \rightarrow q) \wedge \neg q$	assumption	3.	$(p \rightarrow q) \wedge \neg q$	assumption
4.	$\neg p$	1	4.	$\neg p$	1	4.	$p \rightarrow q$	$\wedge_e$ 3	4.	$p \rightarrow q$	$\wedge_e$ 3
5.	$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$	$\rightarrow_i$ 3,4	5.	$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$	$\rightarrow_i$ 3,4	5.	$q$	$\rightarrow_e$ 1,4	5.	$\neg q$	$\wedge_e$ 3
						6.	$\perp$	$\neg_e$ 2,5	6.	$q$	$\rightarrow_e$ 1,4
						7.	$\neg p$	$\perp_e$ 6	7.	$\perp$	$\neg_e$ 2,6
						8.	$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$	$\rightarrow_i$ 3,4,5,6,7	8.	$\neg p$	$\perp_e$ 7
									9.	$((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$	$\rightarrow_i$ 3,4,5,6,7,8

Figure 1.2: Mimicking all 4 rows of the truth table

This looks promising, but we aren’t done, we have only proven our formula under all possible assumptions, but we haven’t exactly proven our formula from nothing given. But note that the reasoning we are doing looks a lot like case work, and we can think of the  $\vee_e$  rule. In words, this rule states that if a formula is true under 2 different assumptions, and one of the assumptions is always true, then our formula is true. So if we just somehow rigorously show at least one of our row assumptions is always true, we will be able to clean up our proof using the  $\vee_e$  rule.

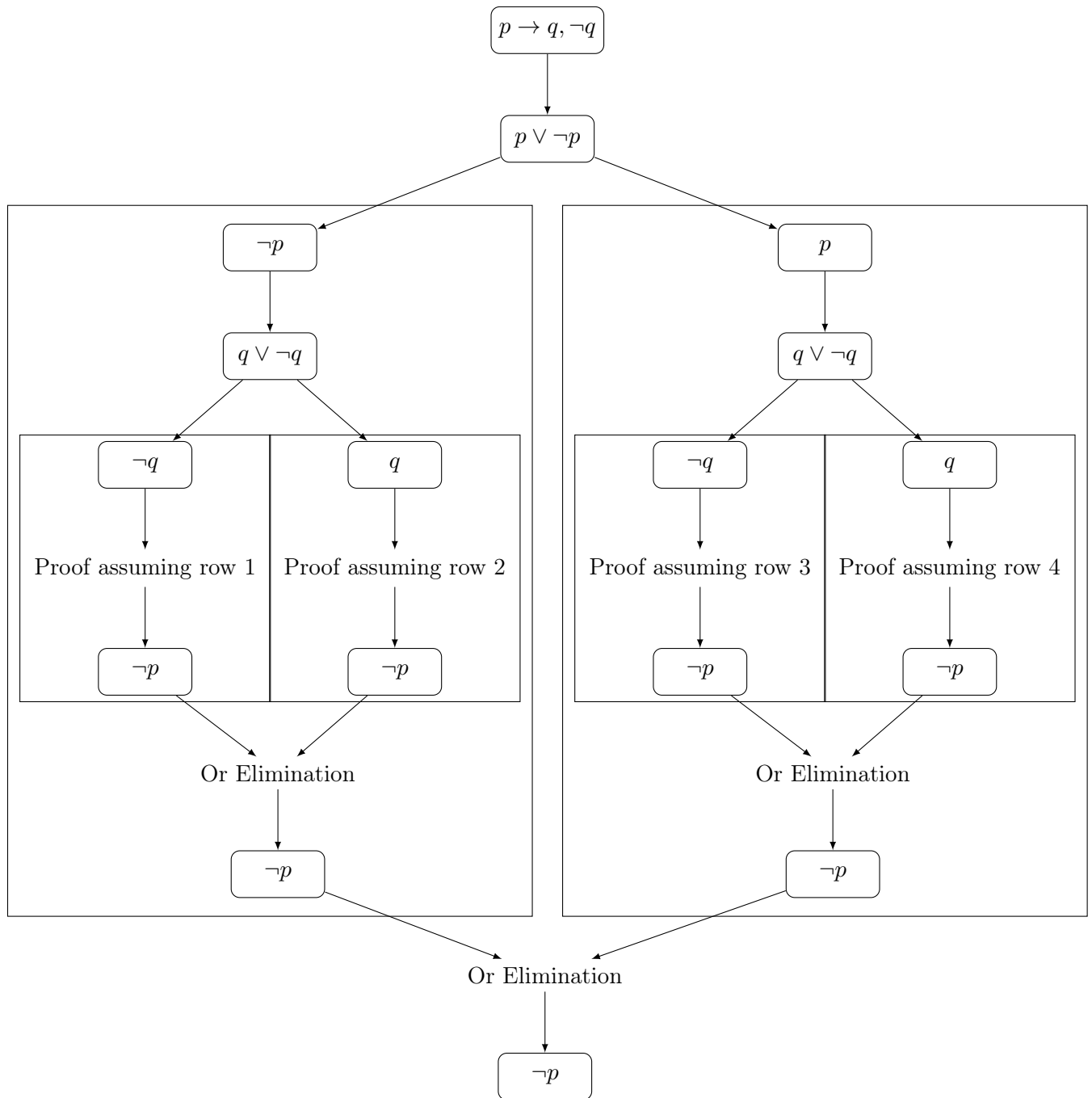
But as seen above, we were able to show a proof for the sequent  $\vdash \varphi \vee \neg \varphi$ . If we just recursively apply this property for all the variables we have, we should be able to capture every row of truth table. So combining this result, our proofs for each row of the truth table, and the  $\vee_e$  rule, the whole proof is constructed as below. The only thing we need now is the ability to construct proofs for each row given the general valid formula  $\bigwedge_{\phi \in \Sigma} \phi \rightarrow \varphi$ .

This can be done using **structural induction** to prove the following:

Let  $\varphi$  be a formula using the propositional variables  $p_1, p_2, \dots, p_n$ . For any assignment to these variables define  $\hat{p}_i = p_i$  if  $p_i$  is set to 1 and  $\hat{p}_i = \neg p_i$  otherwise, then:

- $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \varphi$  is provable if  $\varphi$  evaluates to 1 for the assignment
- $\hat{p}_1, \hat{p}_2, \dots, \hat{p}_n \vdash \neg \varphi$  is provable if  $\varphi$  evaluates to 0 for the assignment.

<sup>1</sup> $\Sigma$  semantically entails  $\varphi$  is equivalent to saying intersection of formulae in  $\Sigma$  implies  $\varphi$  is valid



## 1.6 What about Satisfiability?

Using Natural Deduction, we can only talk about the formulas that are a contradiction or valid. But there are formulas that are neither i.e. they are satisfiable for some assignment of variables but

not all. Example, for some  $p$  and  $q$ ,

$$\not\models p \wedge q$$

But clearly,  $p \wedge q$  is satisfiable when both  $p$  and  $q$  are true. Natural deduction can only claim statements like,

$$\begin{aligned} &\vdash \neg p \rightarrow \neg(p \wedge q) \\ &\vdash (p \rightarrow (q \rightarrow (p \wedge q))) \end{aligned}$$

An important link between the two situations is that,

A formula  $\phi$  is valid **iff**  $\neg\phi$  is not satisfiable

## 1.7 Algebraic Laws and Some Redundancy

### 1.7.1 Distributive Laws

Here are some identities that help complex formulas to some required forms discussed later. These formulas are easily derived using the natural deduction proof rules as discussed above.

$$\begin{aligned} \phi_1 \wedge (\phi_2 \vee \phi_3) &\models (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3) \\ \phi_1 \vee (\phi_2 \wedge \phi_3) &\models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3) \end{aligned}$$

### 1.7.2 Reduction of bi-implication and implication

We also see that bi-implication and implication can be reduced to  $\vee, \wedge$  and  $\neg$  and are therefore redundant in the alphabet of our string.

$$\begin{aligned} \phi_1 \longleftrightarrow \phi_2 &\models (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1) \\ (\phi_1 \rightarrow \phi_2) &\models ((\neg\phi_1) \vee \phi_2) \end{aligned}$$

### 1.7.3 DeMorgan's Laws

Similar to distributive laws, the following laws (again easily provable via natural deduction) help reduce any normal string to a suitable form (discussed in the next section).

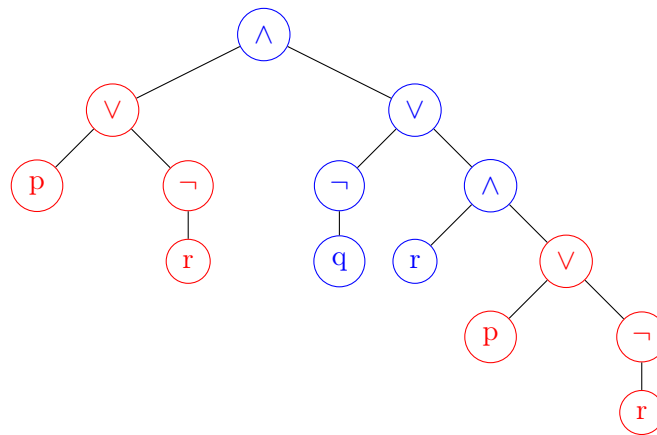
$$\begin{aligned} \neg(\phi_1 \wedge \phi_2) &\models (\neg\phi_1 \vee \neg\phi_2) \\ \neg(\phi_1 \vee \phi_2) &\models (\neg\phi_1 \wedge \neg\phi_2) \end{aligned}$$

## 1.8 Negation Normal Forms

In mathematical logic, a formula is in negation normal form (NNF) if the negation operator ( $\neg$ ) is only applied to variables and the only other allowed Boolean operators are conjunction ( $\wedge$ ) and disjunction ( $\vee$ ). One can convert any formula to a NNF by repeatedly applying DeMorgan's Laws to any clause that may have a  $\neg$ , until only the variables have the  $\neg$  operator. For example

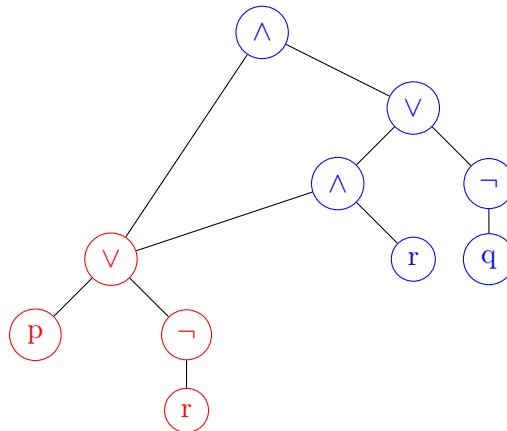
A NNF formula may be represented using its parse tree, which doesn't have any negation nodes except at the leaves, consider  $(p \vee \neg r) \wedge (\neg q \vee (r \wedge (p \vee \neg r)))$

**Parse Tree Representation** of the Above Example:



Since , the red part of the tree is repeating twice , we can make a **DAG(Directed Acyclic Graph)** instead of the parse tree.

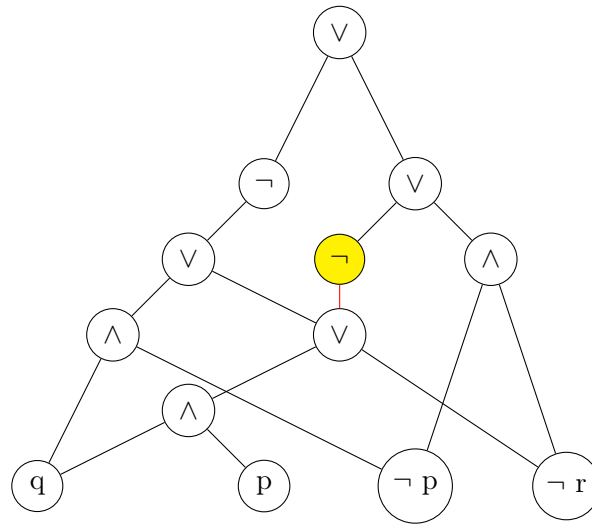
**DAG Representation**



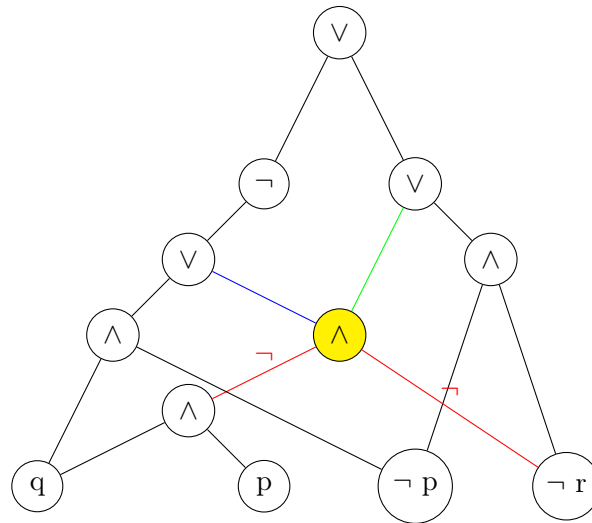
## 1.9 From DAG to NNF-DAG

Given a DAG of propositional logic formula with only  $\vee$  ,  $\wedge$  and  $\neg$  nodes , can we efficiently get a DAG representing a semantically equivalent NNF formula ?

Idea 1: Let's push the " $\neg$ " downwards by applying the De Morgans Law and see what happens.  
 Lets Consider the following example and the highlighted  $\neg$ .



Pushing down the highlighted  $\neg$  across the red edge.



Now, we have an issue in the blue edge. The blue edge wanted the non-negated tree node but due to the above mentioned change, it is getting the negated node. So, this idea won't work. We want to preserve the non-negated nodes as well.

Modification : Make two copies of the DAG and negate (i.e.,  $\neg$  pushing) only 1 of the copies and if a node wants non-negated node then take that node from the copied tree.



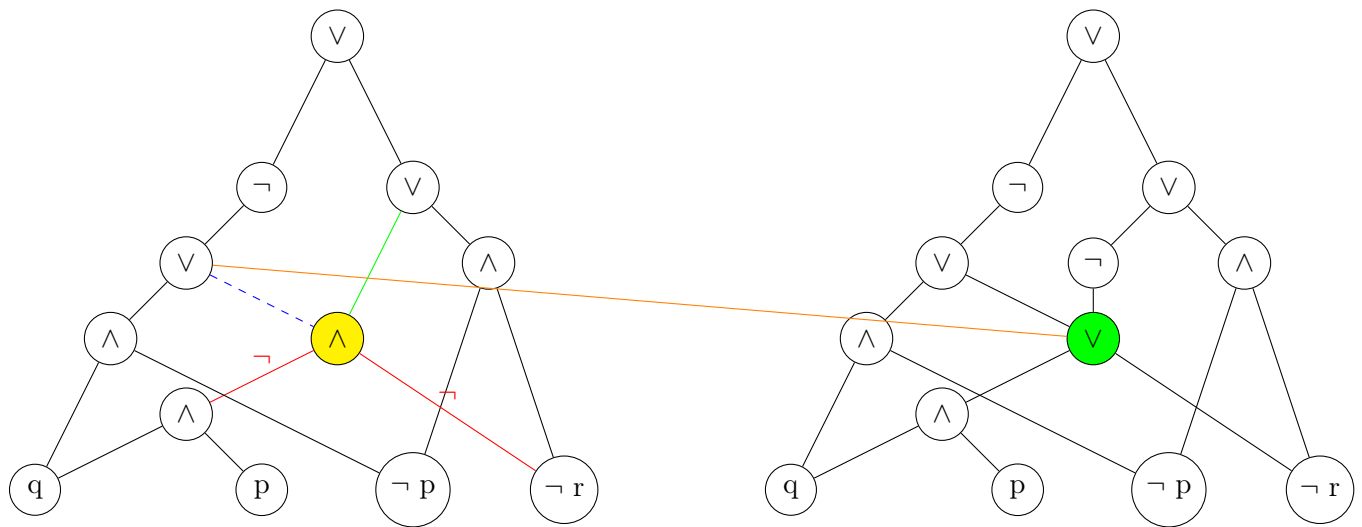


Figure 1.3: Step 1

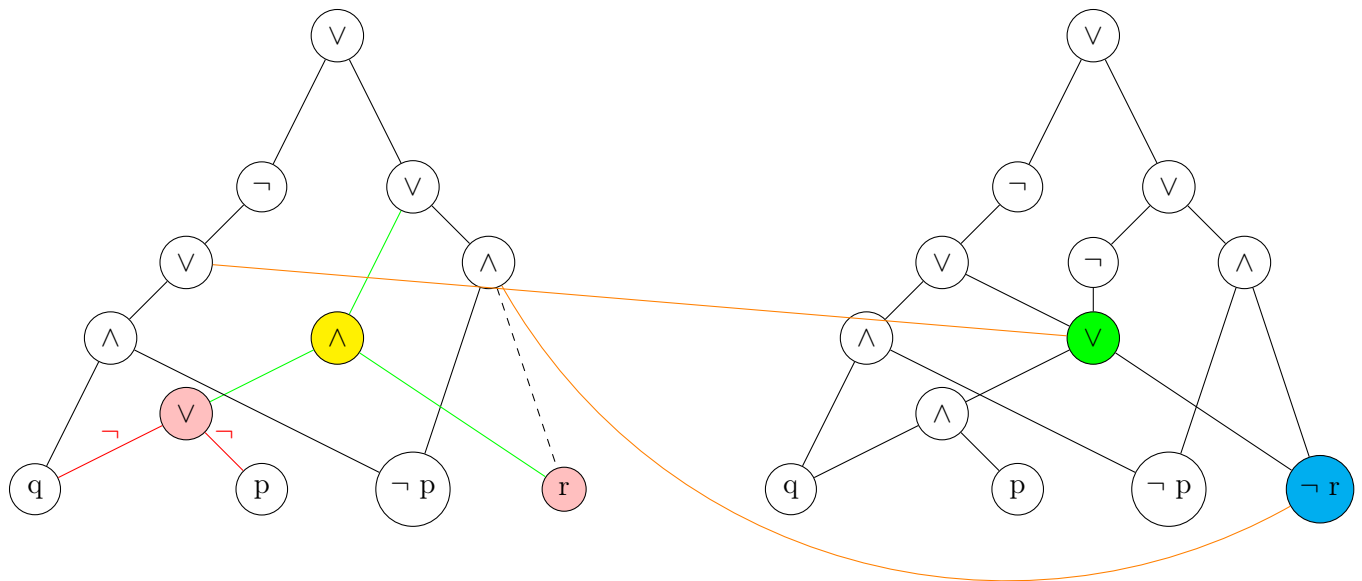


Figure 1.4: Step 2

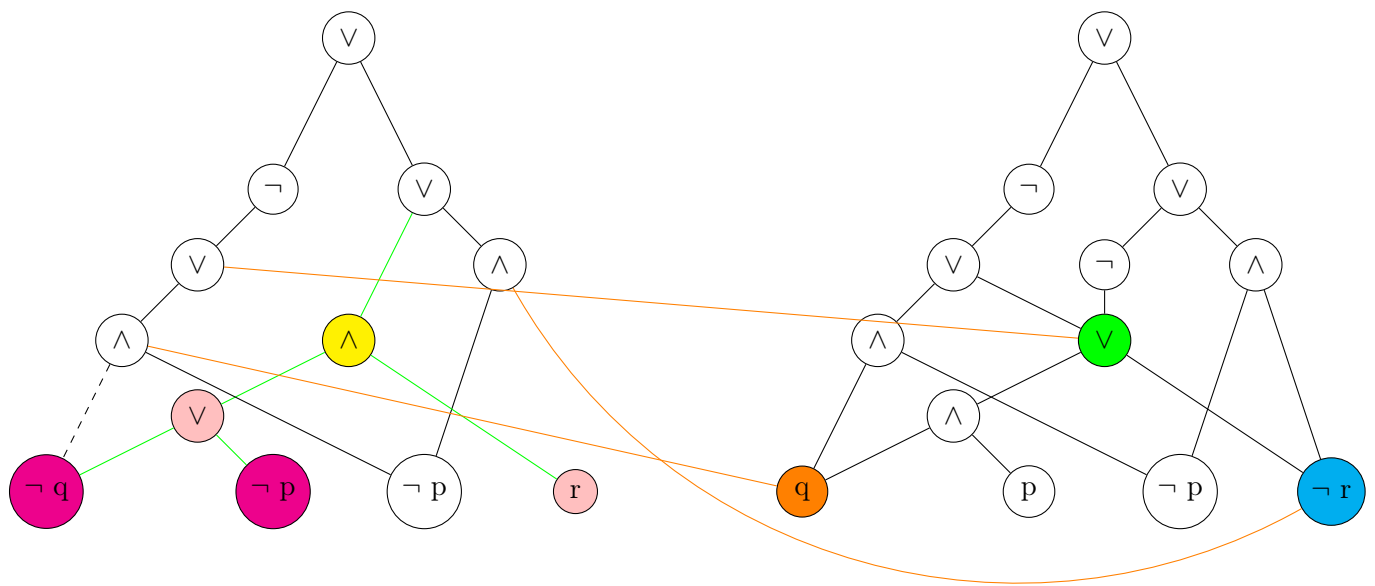


Figure 1.5: Step 3

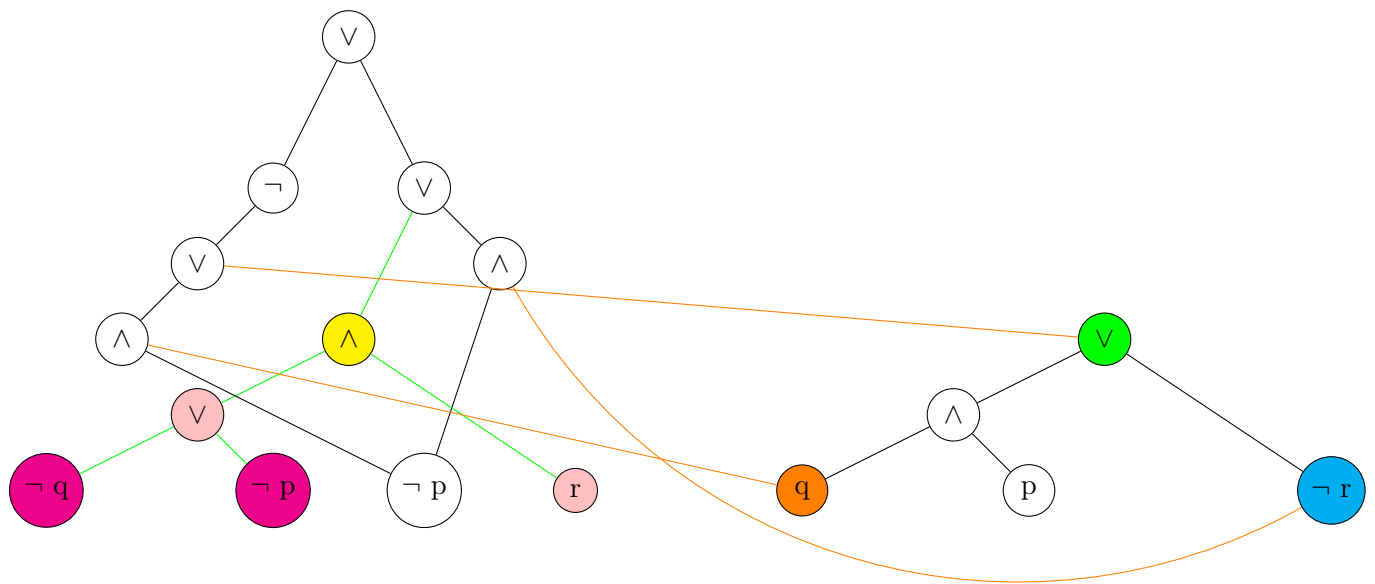


Figure 1.6: Step 4 : Remove all redundant nodes

### 1.10 An Efficient Algorithm to convert DAG to NNF-DAG

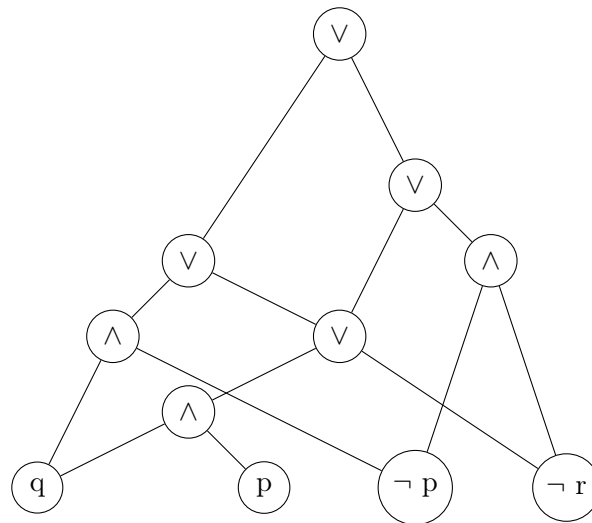


Figure 1.7: Step 1: Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

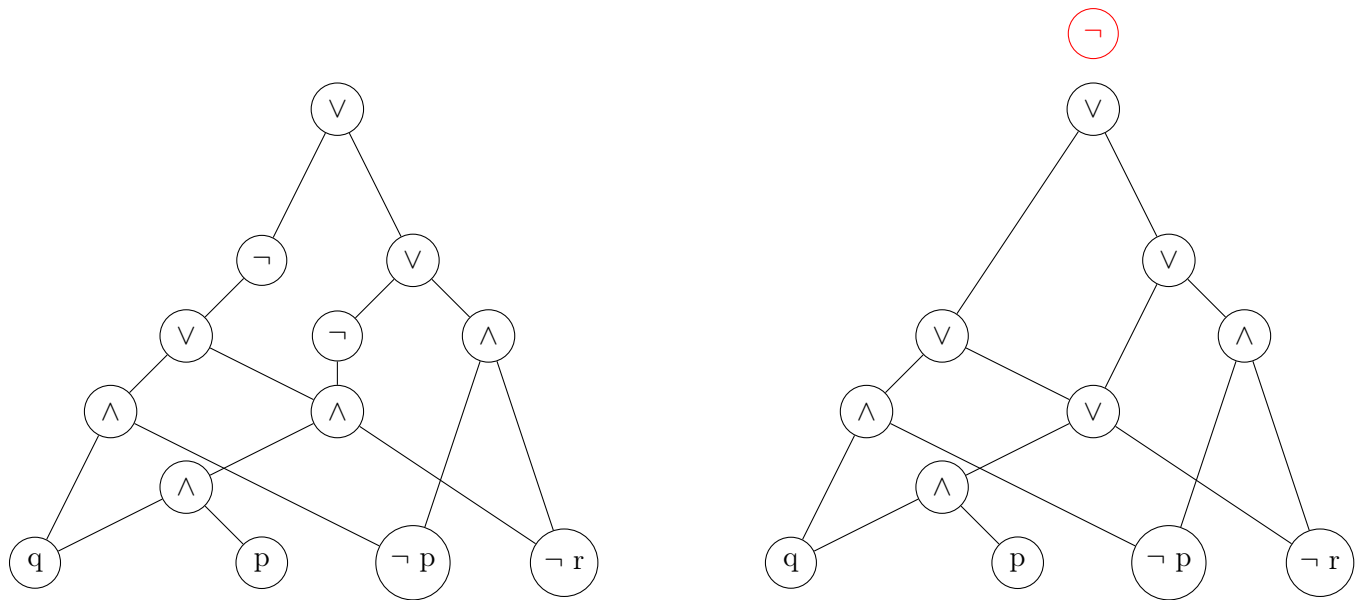


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

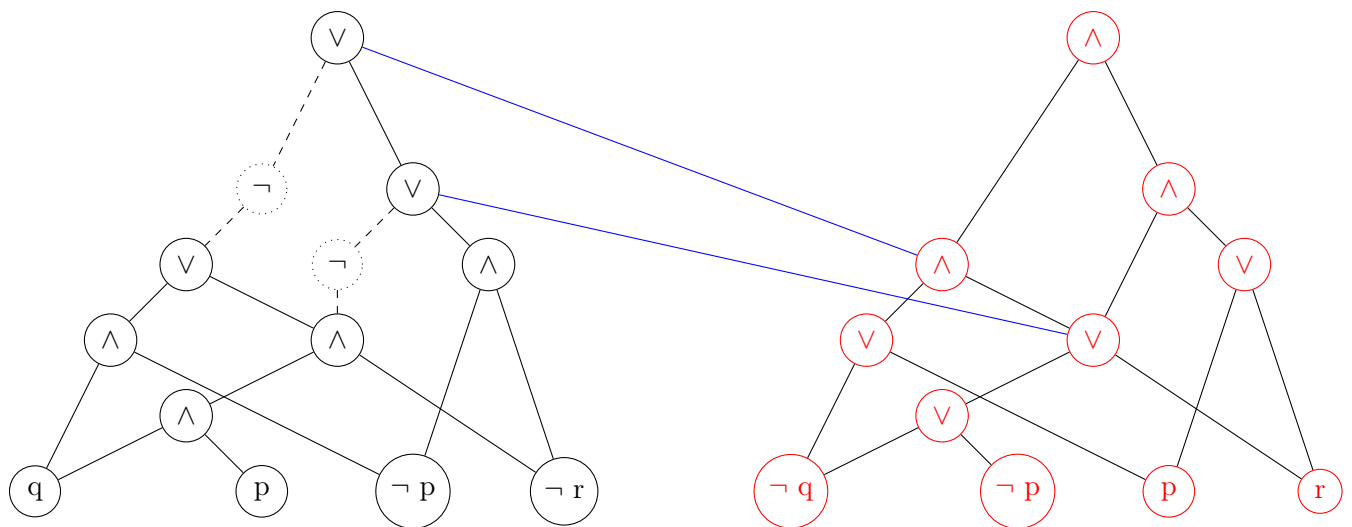
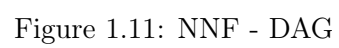


Figure 1.9: Step 3: Remove the " $\neg$ " nodes from the first DAG by connecting them to the corresponding node in the negated DAG.



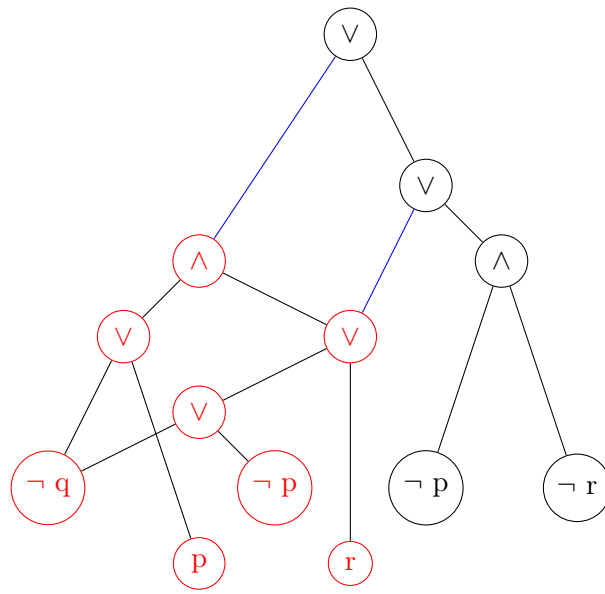


Figure 1.12: The NNF DAG (rearranged)

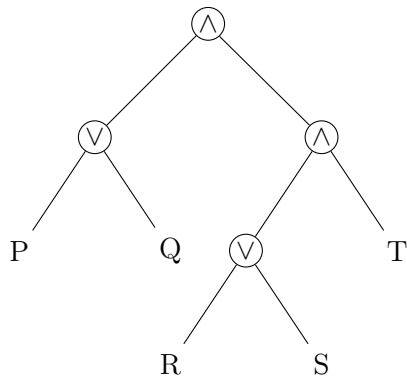
**NOTE :** The size of the NNF - DAG obtained using the above algorithm is atmost two times the size of the given DAG. Hence we have an  $O(N)$  formula for converting any arbitrary DAG to a semantically equivalent NNF - DAG.

## 1.11 Conjunctive Normal Forms

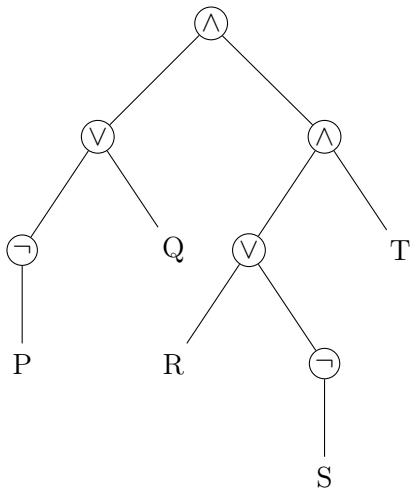
A formula is in conjunctive normal form or clausal normal form if it is a conjunction of one or more clauses, where a clause is a disjunction of variables.

**Examples:**

- Parse Tree for a formula in CNF



- Parse Tree for the formula  $(\neg(p \wedge \neg q) \wedge (\neg(\neg r \wedge s)) \wedge t)$  in NNF



## Some Important Terms

- **LITERAL :**
  - Variable or it's complement.
  - Example :  $p$  ,  $\neg p$  ,  $r$  ,  $\neg q$

- **CLAUSE :**

- A clause is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal
- Example :  $p \vee q \vee (\neg r)$  ,  $p \vee (\neg p) \vee (\neg r)$  -> Not Allowed

- **CUBE :**

- a Cube is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal
- Example  $p \wedge q \wedge (\neg r)$  ,  $p \wedge (\neg p) \wedge (\neg r)$  -> Not Allowed

- **CONJUNCTIVE NORMAL FORM(CNF)**

- A propositional formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of clauses
- Product of Sums
- Example  $(p \vee q \vee (\neg r)) \wedge (q \vee r \vee (\neg s) \vee t)$

- **DISJUNCTIVE NORMAL FORM(DNF)**

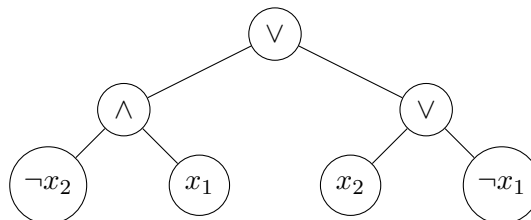
- A formula is said to be in Disjunctive Normal Form (DNF) if it is a disjunction of cubes.
- Sum of Products
- Example  $(p \wedge q \wedge (\neg r)) \vee (q \wedge r \wedge (\neg s) \wedge t)$

**Given a DAG of propositional logic formula with only  $\vee$  ,  $\wedge$  and  $\neg$  nodes , can we efficiently get a DAG representing a semantically equivalent CNF/DNF formula ?**

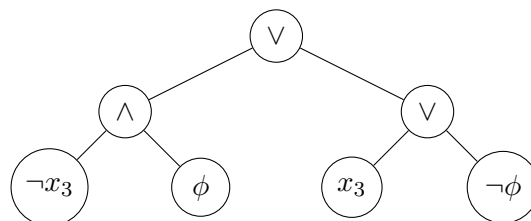
Tutorial 2 Question 2:

The Parity Function can be expressed as  $(( ( \dots (x_1 \oplus x_2 ) \oplus x_3 ) \dots \oplus x_n )$

The Parse Tree for  $x_1 \oplus x_2$  is

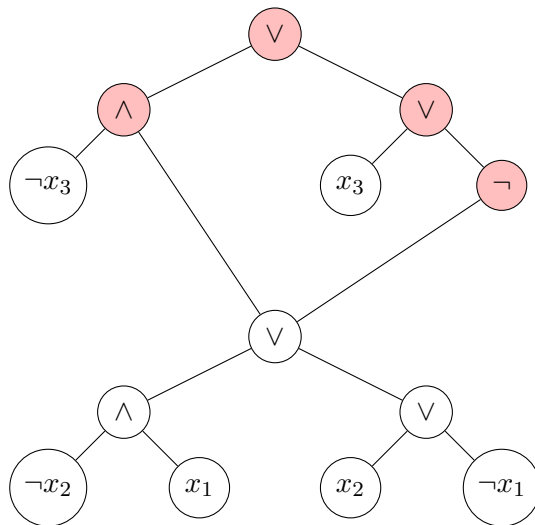


Now consider  $\phi = x_1 \oplus x_2$  then Parse tree for  $(x_1 \oplus x_2) \oplus x_3$  will be





Now putting  $\phi = x_1 \oplus x_2$  in the tree, we get the following DAG representation



We notice that on adding  $x_i$  we are adding 4 nodes. Hence, the **size of DAG of the parity function is atmost  $4n$** . And we have already shown that size of NNF-DAG is atmost 2 times the size of DAG. So, the **size of the semantically equivalent NNF-DAG is atmost  $8n$** .

Also , in the Tutorial Question we have proved that the DAG **size of the semantically equivalent CNF/DNF formula is atleast  $2^{n-1}$** .

**NNF  $\rightarrow$  CNF/DNF (Exponential Growth in size of the DAG)**

## 1.12 Satisfiability and Validity Checking

It is **easy to check validity of CNF**. Check for every clause that it has some  $p$  ,  $\neg p$ . If **there is a clause which does not have both( $p$  ,  $\neg p$ )**, then the **Formula is not valid** because we can always assign variables in such a way that makes this clause false.

Example - If we have a clause  $p \vee q \vee (\neg r)$  , we can make it 0 with assignments  $p = 0$ ,  $q = 0$  and  $r = 1$ .

And if both a variable and its conjugate are present in a clause then since  $p \vee \neg p$  is valid and  $1 \vee \phi = 1$  for any  $\phi$ . So, the clause will be always valid.

Hence , we have an  **$O(N)$  algorithm to check the validity of CNF** but the price we pay is conversion to it(which is exponential).

It is hard to check validity of DNF because we will have to find an assignment which falsifies all the cubes.

What is meant by satisfiability?

Given a Formula , is there an assignment which makes the formula valid.

It is **easy to check satisfiability of DNF**. Check for every cube that it has some  $p$  ,  $\neg p$ . If there is a cube which does not have both( $p$  ,  $\neg p$ ), then the Formula is satisfiable because we can always assign variables in such a way that makes this cube true and hence the entire formula true. Overall, **We just need to find a satisfying assignment for any one cube.**

Hard to check satisfiability using CNF. We will have to find an assignment which simultaneously

satisfy all the cubes.

**NOTE :** A formula is valid if it's negation is not satisfiable. Therefore , we can convert every validity problem to a satisfiability problem. Thus, it suffices to worry only about satisfiability problem.

### 1.13 DAG to Equisatisfiable CNF

**Claim:** Given any formula, we can get an equisatisfiable formula in CNF of linear size efficiently.

*Proof:* Let's consider the DAG  $\phi(p, q, r)$  given below:

1. Introduce new variables  $t_1, t_2, t_3, \dots, t_n$  for each of the nodes. We will get an equisatisfiable formula  $\phi'(p, q, r, t_1, t_2, t_3, \dots, t_n)$  which is in CNF.
2. Write the formula for  $\phi'$  as a conjunction of subformulas for each node of form given below:

$$\begin{aligned}\phi' = & (t_1 \iff (p \wedge q)) \wedge \\ & (t_2 \iff (t_1 \vee \neg r)) \wedge \\ & (t_3 \iff (\neg t_2)) \wedge \\ & (t_4 \iff (\neg p \wedge \neg r)) \wedge \\ & (t_5 \iff (t_3 \vee t_4)) \wedge \\ & t_5\end{aligned}$$

3. Convert each of the subformula to CNF. For the first node it is shown below:

$$\begin{aligned}(t_1 \iff (p \wedge q)) &= (\neg t_1 \wedge (p \wedge q)) \wedge (\neg(p \wedge q) \vee t_1) \\ &= (\neg t_1 \vee p) \wedge (\neg t_1 \vee q) \wedge (\neg p \vee \neg q \vee t_1)\end{aligned}$$

4. For checking the equisatisfiability of  $\phi$  and  $\phi'$ : Think about an assignment which makes  $\phi$  true then apply that assignment to  $\phi'$ .

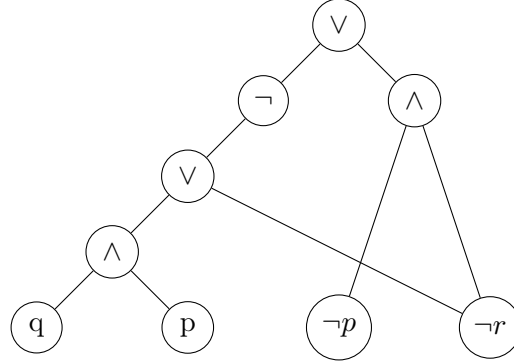
### 1.14 Tseitin Encoding

The Tseitin encoding technique is commonly employed in the context of Boolean satisfiability (SAT) problems. SAT solvers are tools designed to determine the satisfiability of a given logical formula, i.e., whether there exists an assignment of truth values to the variables that makes the entire formula true.

The basic idea behind Tseitin encoding is to introduce additional auxiliary variables to represent complex subformulas or logical connectives within the original formula. By doing this, the formula can be transformed into an equivalent CNF representation.

Say you have a formula  $Q(p, q, r, \dots)$ . Now we can use and introduce auxiliary variables  $t_1, t_2, \dots$  to make a new formula  $Q'(p, q, r, \dots, t_1, t_2, \dots)$  using Tseitin encoding which is equisatisfiable as  $Q$ .  $Q'$  is equisatisfiable as  $Q$  but not semantically equivalent. Size of  $Q'$  is linear in size of  $Q$ .

Lets take an example to understand better. Consider the formula  $(\neg((q \wedge p) \vee \neg r)) \vee (\neg p \wedge \neg r)$



We define a new equisatisfiable formula with auxiliary variables  $t_1, t_2, t_3, t_4$  and  $t_5$  as follows:

$$(t_1 \iff p \wedge q) \wedge (t_2 \iff t_1 \vee \neg r) \wedge (t_3 \iff \neg t_2) \wedge (t_4 \iff \neg p \wedge \neg r) \wedge (t_5 \iff t_3 \vee t_4) \wedge (t_5)$$

## 1.15 Towards Checking Satisfiability of CNF and Horn Clauses

A Horn clause is a disjunctive clause (a disjunction of literals) with at most one positive literal.

A Horn Formula is a conjunction of horn clauses, for example:

$$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_4 \vee x_5 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \wedge (x_5) \wedge (\neg x_5 \vee x_3) \wedge (\neg x_5 \vee \neg x_1)$$

Now we can convert any horn clause to an implication by using disjunction of the literals that were in negation form in the horn clause on left side of the implication and the unnegated variable on the other side of the implication. So all the variables in all the implications will be unnegated.

So the above equation can be translated as follows.

$$x_1 \wedge x_2 \implies x_3$$

$$x_4 \wedge x_3 \implies x_5$$

$$x_1 \wedge x_5 \implies \perp$$

$$x_5 \implies x_3$$

$$\top \implies x_5$$

Now we try to find a satisfying assignment for the above formula.

From the last clause we get  $x_5 = 1$ , now the fourth clause is  $\top \implies x_3$ .

Then from the fourth clause we get that  $x_3 = 1$ .

Now in the remaining clauses none of the left hand sides are reduced to  $\top$ .

Hence, we set all remaining variables to 0 to get a satisfying assignment.

**Algorithm 1:** HORN Algorithm

---

```

1 Function HORN( $\phi$ ):
2   foreach occurrence of  $\top$  in  $\phi$  do
3     | mark the occurrence
4   while there is a conjunct  $P_1 \wedge P_2 \wedge \dots \wedge P_{k_i} \rightarrow P'$  in  $\phi$  do
5     | // such that all  $P_j$  are marked but  $P'$  isn't
6     | if all  $P_j$  are marked and  $P'$  isn't then
7       | | mark  $P'$ 
8     | if  $\perp$  is marked then
9       | | return 'unsatisfiable'
10    | else
11    | | return 'satisfiable'

```

---

**Complexity:**

If we have  $n$  variables and  $k$  clauses then the solving complexity will be  $O(nk)$  as in worst case in each clause you search for each variable.

**1.16 Counter example for Horn Formula**

In our previous lectures, we delved into the Horn Formula, a valuable tool for assessing the satisfiability of logical formulas.

**1.16.1 Example**

We are presented with a example involving conditions that determine when an alarm ( $a$ ) should ring. Let's outline the given conditions:

1. If there is a burglary ( $b$ ) in the night ( $n$ ), then the alarm should ring ( $a$ ).
2. If there is a fire ( $f$ ) in the day ( $d$ ), then the alarm should ring ( $a$ ).
3. If there is an earthquake ( $e$ ), it may occur in the day ( $d$ ) or night ( $n$ ), and in either case, the alarm should ring ( $a$ ).
4. If there is a prank ( $p$ ) in the night ( $n$ ), then the alarm should ring ( $a$ ).
5. Also it is known that prank ( $p$ ) does not happen during day ( $d$ ) and burglary ( $b$ ) does not takes place when there is fire ( $f$ ).

Let us write down these implications

$$\begin{aligned}
 b \wedge n &\Rightarrow a & f \wedge d &\Rightarrow a & e \wedge d &\Rightarrow a \\
 e \wedge n &\Rightarrow a & p \wedge n &\Rightarrow a & d \wedge n &\Rightarrow \perp \\
 b \wedge f &\Rightarrow a & p \wedge d &\Rightarrow \perp
 \end{aligned}$$

Now we want to examine the possible behaviour of this system under the assumption that alarm rings during day. For this we add two more clauses:

$$\top \Rightarrow a \quad \top \Rightarrow d$$

This directly gives us that  $a, d$  have to be true, what about the rest? We can see that setting all the remaining variables to false is a satisfying assignment for this set of formulae.

Hence we have none of prank, earthquake, burglary or fire and hence alarm should not ring.

This means that our formulae system is incomplete.

To achieve this, we try to introduce new variables  $Na$  (no alarm),  $Nf$  (no fire),  $Nb$  (no burglary),  $Ne$  (no earthquake), and  $Np$  (no prank).

We extend the above set of implications in a natural way using these formulae:

$$a \wedge Na \Rightarrow \perp \quad b \wedge Nb \Rightarrow \perp \quad f \wedge Nf \Rightarrow \perp \quad e \wedge Ne \Rightarrow \perp \quad p \wedge Np \Rightarrow \perp$$

$$Nb \wedge Nf \wedge Ne \wedge Np \Rightarrow Na$$

All the implications will hold true for the values  $b = p = e = f = Nb = Ne = Nf = Np = 0$ .

Here, we are getting  $b = Nb$ , which is not possible, hence, we need the additional constraint that  $b \iff Nb$ . But on careful examination we see that this cannot be represented as a horn clause.

Therefore, it becomes necessary to devise an alternative algorithm suited for evaluating satisfiability.

## 1.17 Davis Putnam Logemann Loveland (DPLL) Algorithm

This works for more general cases of CNF formulas where it need not be a Horn formula. Let us first discuss techniques and terms required for our Algorithm.

- **Partial Assignment (PA) :** It is any assignment of some of the propositional variables.  
Ex.  $PA = \{x_1 = 1, x_2 = 0\}$  ;  $PA = \{\}$ , etc.
- **Unit Clause :** It is any clause which only has one literal in it. Ex.  $\dots \wedge (\neg x_5) \wedge \dots$   
Note: If any Formula has a unit clause then the literal in it has to be set to true.
- **Pure Literal :** A literal which doesn't appear negated in any clause. Say a propositional variable  $x$  appear only as  $\neg x$  in every clause it appears in., or say  $y$  appears only as  $y$  in every clause.  
Note: If there is a pure literal in the formula, it does not hurt any clause to set it to true. All the clauses in which this literal is present will become true immediately.

We will now utilize every techniques we learnt to simplify our formula. First we check if our formula has unit clause or not. If yes then we assign the literal in that clause to be 1. Note,  $\varphi[l = 1]$  is the formula obtained after setting  $l = 1$  everywhere in the formula. We also search for pure literals. If we find a pure literal then we can simply assign it 1 (or 0 if it always appears in negated form) and proceed, this cannot harm us (cause future conflicts) due to the definition of Pure Literal. If we do not have any of these then we have only one option left at the moment which is try and error.

We assign any one of the variable in the formula a value which we choose by some way (not described here). Then we go on with the usual algorithm until we either get the whole formula to

be true or false. At this step we might have to backtrack if the formula turns out to be false. If it is true we can terminate the algorithm.

**Note:** Our algorithm can be as worse as a **Truth Table** as we are trying every assignment. But as we are applying additional steps, after making a decision there are high chances that we get a unit clause or a pure literal.

Now as we have done all the prerequisites let us state the algorithm.

---

**Algorithm 2:** SAT( $\varphi$ , PA)

---

```

{ // These are the base cases for our recursion }
if  $\varphi = \top$  then
    return SAT(sat, PA)
else if  $\varphi = \perp$  then
    return SAT(unsat, PA)
else if  $C_i$  is a unit clause (literal  $l$ ) and  $C_i \in \varphi$  then
    { // This step is called Unit Propagation }
    return SAT( $\varphi[l = 1]$ ,  $PA \cup \{l = 1\}$ ) { // Here recursively call the algorithm on the simplified }
                                                    // formula  $\varphi[l = 1]$ 

else if  $l$  is a pure literal and  $l \in \varphi$  then
    { // This step is called Pure Literal Elimination }
    return SAT( $\varphi[l = 1]$ ,  $PA \cup \{l = 1\}$ )
else
    { // This step is called Decision Step }
     $x \leftarrow \text{choose\_a\_var}(\varphi)$ 
     $v \leftarrow \text{choose\_a\_value}(\{0, 1\})$ 
    if SAT( $\varphi[x = v]$ ,  $PA \cup \{x = v\}$ ).status = sat then
        return SAT(sat,  $PA \cup \{x = v\}$ )
    else if SAT( $\varphi[x = 1 - v]$ ,  $PA \cup \{x = 1 - v\}$ ).status = sat then
        return SAT(sat,  $PA \cup \{x = 1 - v\}$ )
    else
        return SAT(unsat, PA)
    end if
end if

```

---

**Question** Can the formula be a horn formula after steps 1 and 2 can't be applied anymore? i.e. If our formula does not have any unit clause or pure literal can it be a horn formula?

**ANS.** Yes. Here is an example

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

## 1.18 DPLL in action

### 1.18.1 Example

Consider the following clauses, for which we have to find whether all can be satisfied for a variable mapping or not using DPLL algorithm-

$$\begin{array}{llll} C_1 : (\neg P_1 \vee P_2) & C_2 : (\neg P_1 \vee P_3 \vee P_5) & C_3 : (\neg P_2 \vee P_4) & C_4 : (\neg P_3 \vee P_4) \\ C_5 : (P_1 \vee P_5 \vee \neg P_2) & C_6 : (P_2 \vee P_3) & C_7 : (P_1 \vee P_3 \vee P_7) & C_8 : (P_6 \vee \neg P_5) \end{array}$$

Lets make two possible decision trees for these clauses.

PLE - Pure literal elimination      UP - Unit Propagation      D - Decision

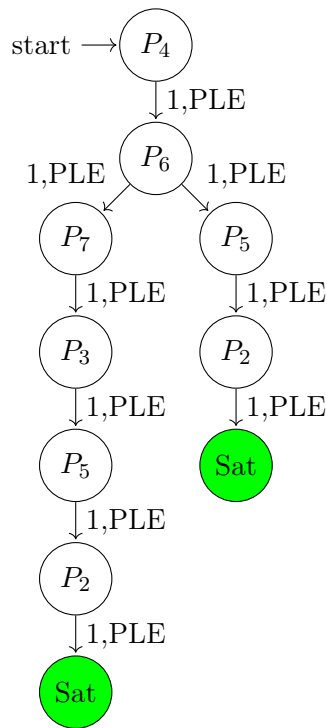


Figure 1.13: DPLL

The following is the decision tree if we remove the point 2 of DPLL. Note the increase in number of operations.

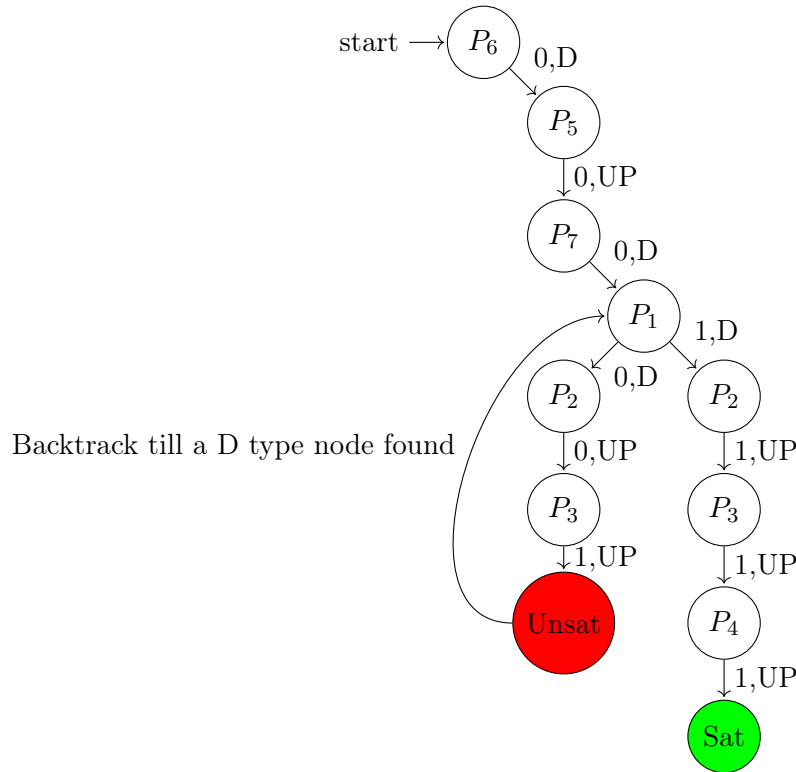


Figure 1.14: DPLL without point 2

## 1.19 Applying DPLL Algorithm to Horn Formulas

Let us apply DPLL algorithm to Horn Formula

If there are no variables on LHS, it becomes a Unit Clause *i.e.*  $\top \rightarrow x_i$  and is equivalent to  $(x_i)$ .

Horn Method can be viewed in terms of DPLL algorithm as following:

- Apply Unit Propagation until you can't apply.
- After that, set all remaining variables to 0.

Advantage of Horn's method is after all possible Unit Propagations are done, it sets all remaining variables to 0, but in DPLL we need to go step by step for each remaining variable.

But Horn's method can only be applied in a special case, moreover, in Horn's method we only figure out which variables to set true as opposed to DPLL which can figure out whether variable needs to be set to true or false via the Pure Literal Elimination.



# Practice Problems 1

## 0. A Bit of Warm-Up

- (a) A student has written the following propositional logic formula over variables  $x, y, z$ .

$$x \rightarrow ((y \rightarrow \perp) \vee (\top \rightarrow z))$$

Is it possible to write a semantically equivalent formula using only the operators  $\neg$  and  $\vee$ ?

- (b) Consider the following CNF formula over propositional variables  $p, q, r, s, t$ .

$$\varphi(p, q, r, s, t) = (p \vee q \vee r) \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s) \wedge (\neg s \vee t) \wedge (\neg t \vee p)$$

Write a semantically equivalent DNF formula with only one cube. Try to avoid application of the distributive laws, as this can lead to a blow-up of intermediate formula sizes.

- (c) We wish to show that  $\varphi \models (\neg p \vee (q \wedge r \wedge s \wedge t))$ . One way to show this is to show that  $\varphi \wedge \neg(\neg p \vee (q \wedge r \wedge s \wedge t))$  is unsatisfiable. Since  $\neg(\neg p \vee (q \wedge r \wedge s \wedge t)) = (p) \wedge (\neg q \vee \neg r \vee \neg s \vee \neg t)$ , our problem reduces to showing unsatisfiability of  $\varphi \wedge (p) \wedge (\neg q \vee \neg r \vee \neg s \vee \neg t)$ . Show using repeated application of resolution rules (and nothing else) that the above formula is indeed unsatisfiable.

### Solution:

1. Indeed, it's possible. Here are suggested steps.

- Write every implication  $\alpha \rightarrow \beta$  as  $(\neg \alpha \vee \beta)$ . This gives us the formula

$$\neg x \vee ((\neg y \vee \perp) \vee (\neg \top \vee z))$$

- Simplify every  $(\alpha \vee \perp)$  to  $\alpha$ . Similarly,  $\neg \top$  can be replaced by  $\perp$ . This gives us

$$\neg x \vee ((\neg y) \vee (z))$$

- Finally, apply DeMorgan's laws to replace every  $\alpha \vee \beta$  with  $\neg(\neg \alpha \wedge \neg \beta)$ . This gives

$$\neg(x \wedge (y \wedge \neg z))$$

2. Notice that  $\varphi$  can be written as

$$(p \vee q \vee r) \wedge (p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \rightarrow s) \wedge (s \rightarrow t) \wedge (t \rightarrow p)$$

The five implications above force  $p, q, r, s, t$  to have the same value in any satisfying assignment of  $\varphi$  (try to convince yourself why this is the case). Therefore, for  $\varphi$  to be satisfied, we must have at least one of  $p, q, r$  to be true (because of the first clause of  $\varphi$ ), and all of  $p, q, r, s, t$  must have the same value (because of the remaining clauses of  $\varphi$ ). This implies that the only satisfying assignment of  $\varphi$  is  $p = q = r = s = t = 1$ . Hence, the semantically equivalent DNF formula is  $p \wedge q \wedge r \wedge s \wedge t$ .

3. We don't even need all clauses of  $\varphi$  to show unsatisfiability by resolution. Here are suggested resolution steps (alternative resolution steps are also possible).

Clause id	Clause	How obtained
1	$\neg p \vee q$	Given
2	$p$	Given
3	$q$	Resolvent of 1, 2
4	$\neg q \vee \neg r \vee \neg s \vee \neg t$	Given
5	$\neg r \vee \neg s \vee \neg t$	Resolvent of 3, 4
6	$\neg q \vee r$	Given
7	$r$	Resolvent of 3, 6
8	$\neg s \vee \neg t$	Resolvent of 7, 5
9	$\neg r \vee s$	Given
10	$s$	Resolvent of 7, 9
11	$\neg t$	Resolvent of 8, 10
12	$\neg s \vee t$	Given
13	$t$	Resolvent of 10, 12
14	Empty clause ()	Resolvent of 11, 13

## 1. Core of unsatisfiability

A set  $\mathcal{S}$  of propositional logic formulae is said to form a *minimal unsatisfiable core* if  $\mathcal{S}$  is unsatisfiable (i.e. there is no assignment of values to variables that satisfies all formulas in  $\mathcal{S}$ ), but every proper subset of  $\mathcal{S}$  is satisfiable.

1. Given an unsatisfiable set  $\mathcal{S}$  of propositional logic formulae, the minimal unsatisfiable core may not be unique. Give an example where  $\mathcal{S}$  has at least two minimal unsatisfiable cores  $\mathcal{C}_1$  and  $\mathcal{C}_2$  such that  $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$ .
2. Show that for every  $n > 0$ , we can find a set of  $n$  propositional logic formulae that form a minimal unsatisfiable core. Thus, we can have minimal unsatisfiable cores of arbitrary finite size.

### Solution:

1. This is fairly straightforward. Consider the set  $\mathcal{S} = \{p, p \rightarrow q, \neg q, p \rightarrow r, \neg r\}$ . The two minimal unsatisfiable cores are  $\mathcal{C}_1 = \{p, p \rightarrow p_2, \neg p_2\}$  and  $\mathcal{C}_2 = \{p, p \rightarrow p_3, \neg p_3\}$
2. The set of formulas  $\mathcal{S}_n = \{p_1, p_1 \rightarrow p_2, p_2 \rightarrow p_3, \dots, p_{n-2} \rightarrow p_{n-1}, \neg p_{n-1}\}$  satisfies the condition of the question. Why is the entire set  $\mathcal{S}_n$  is unsatisfiable? This should be easy to figure out.

Every proper subset of  $\mathcal{S}_n$  either has  $p_1$  missing,  $\neg p_{n-1}$  missing or some implication  $p_i \rightarrow p_{i+1}$  missing for  $1 \leq i \leq n-2$ . We show that in each of these cases, the remaining subset of formulas is satisfiable.

- If  $p_1$  is missing, all the other formulas are satisfied by setting  $p_2 = \dots = p_{n-1} = 0$ .
- If  $\neg p_{n-1}$  is missing, all the other formulas are satisfied by setting  $p_1 = \dots = p_{n-2} = 1$ .
- If  $p_i \rightarrow p_{i+1}$  is missing for  $1 \leq i \leq n-2$ , then all other formulas are satisfied by setting  $p_1 = \dots p_i = 1$  and  $p_{i+1} = \dots p_{n-1} = 0$ .

## 2. Logical interpolation

Let  $\varphi$  and  $\psi$  be propositional logic formulas such that  $\models \varphi \rightarrow \psi$  (i.e.  $\varphi \rightarrow \psi$  is a tautology). Let  $Var(\varphi)$  and  $Var(\psi)$  denote the set of propositional variables in  $\varphi$  and  $\psi$  respectively. Show that there exists a propositional logic formula  $\zeta$  with  $Var(\zeta) \subseteq Var(\varphi) \cap Var(\psi)$  such that  $\models \varphi \rightarrow \zeta$  and  $\models \zeta \rightarrow \psi$ . This result is also known as *Craig's interpolation theorem* as applied to propositional logic. The formula  $\zeta$  is called an *interpolant* of  $\varphi$  and  $\psi$ .

As a specific illustration of the above result, consider the formulas  $\varphi = ((p \rightarrow q) \wedge (q \rightarrow \neg p))$  and  $\psi = (r \rightarrow (p \rightarrow s))$ , where  $p, q, r, s$  are propositional variables. Convince yourself that  $\models \varphi \rightarrow \psi$  holds in this example. Note that  $Var(\varphi) = \{p, q\}$  and  $Var(\psi) = \{p, r, s\}$ . Let  $\zeta = \neg p$ . Then  $Var(\zeta) \subseteq Var(\varphi) \cap Var(\psi)$ . Convince yourself that  $\models \varphi \rightarrow \zeta$  and  $\models \zeta \rightarrow \psi$  hold in this example.

**Solution:** Given a formula  $\varphi$  and a variable  $v \in Var(\varphi)$ , let  $\varphi[v = \top]$  denote the formula obtained by replacing all occurrences of  $v$  in  $\varphi$  with  $\top$ , and then simplifying the resulting formula. By simplification, we mean the obvious ones like  $\alpha \wedge \top = \alpha$ ,  $\alpha \vee \top = \top$ ,  $\alpha \wedge \neg \top = \perp$ ,  $\alpha \vee \neg \top = \alpha$  for all sub-formulas  $\alpha$  of  $\varphi$ . In a similar manner, we define  $\varphi[v = \perp]$ .

Note that  $Var(\varphi[v = \top]) = Var(\varphi) \setminus \{v\}$  and similarly for  $\varphi[v = \perp]$ .

Now define the formula  $\zeta = \bigvee_{v \in Var(\varphi) \setminus Var(\psi)} \bigvee_{a \in \{\perp, \top\}} \varphi[v = a]$ . Notice that  $Vars(\zeta) \subseteq Vars(\varphi) \cap Vars(\psi)$ .

You should now be able to argue that (i)  $\varphi \models \zeta$ , and (ii)  $\zeta \models \psi$ . Proving (i) should be straightforward from the definition of  $\zeta$ . To prove (ii), take any satisfying assignment of  $\zeta$ . By definition of  $\zeta$ , this gives an assignment of truth values to variables in  $Vars(\varphi) \cap Vars(\psi)$  such that this assignment can be augmented with an assignment of truth values to variables in  $Vars(\varphi) \setminus Vars(\psi)$  to satisfy the formula  $\varphi$ . However, since  $\varphi \rightarrow \psi$  is a tautology, this (augmented) assignment also satisfies  $\psi$ . Recalling that satisfaction of  $\psi$  cannot depend on the assignment of truth values to variables not present in  $\psi$ , we conclude that the assignment of truth values to variables in  $Vars(\varphi) \cap Vars(\psi)$  itself satisfies the formula  $\psi$ . Hence  $\zeta \models \psi$ .

### 3. DPLL with horns

The Horn-Sat problem entails checking the satisfiability of Horn formulas. We say a formula is a Horn formula if it is a conjunction ( $\wedge$ ) of Horn clauses. A Horn clause has the form  $\phi_1 \rightarrow \phi_2$  where  $\phi_1$  is either  $\top$  or a conjunction ( $\wedge$ ) of one or more propositional variables.  $\phi_2$  is either  $\perp$  or a **single** propositional variable. In this context, answer the following questions

1. Let's try to solve the Horn-Sat problem using DPLL. We can convert each Horn clause into a CNF clause by simply rewriting  $(a \rightarrow b)$  as  $(\neg a \vee b)$ , which preserves the semantics of the formula. The resultant formula is in CNF.

We have seen in class that if DPLL always chooses to assign 0 to a decision variable before assigning 1 (if needed) to the variable, then DPLL will never need to backtrack when given a Horn formula encoded in CNF as input.

Suppose our version of DPLL does just the opposite, i.e. it always assigns 1 to a decision variable before assigning 0 (if needed). How many backtracks are needed if we run this version of DPLL on the (CNF-ised version of) following Horn formulas, assuming DPLL always chooses the unassigned variable with the smallest subscript when choosing a decision variable?

(a)

$$\left( \left( \bigwedge_{i=0}^{n-1} x_i \right) \rightarrow x_n \right) \wedge \bigwedge_{i=0}^{n-1} (x_n \rightarrow x_i) \wedge \left( \left( \bigwedge_{i=0}^n x_i \right) \rightarrow \perp \right)$$

(b)

$$\bigwedge_{i=0}^{n-1} \left( (x_i \rightarrow x_{n+i}) \wedge (x_{n+i} \rightarrow x_i) \wedge (x_i \wedge x_{n+i} \rightarrow \perp) \right)$$

2. Since we have a polynomial time formula for Horn-Sat, the next question is if it will allow us to solve the Boolean-SAT problem in polynomial time. Sadly, this is not the case. Find a boolean function that cannot be expressed by a Horn formula. Prove that no Horn formula can represent the given boolean function.

#### Solution:

1. It's best to write out the implications as clauses when you are trying to apply DPLL.

In problem (a), no unit clauses or pure literals are obtained until all of  $x_0$  through  $x_{n-1}$  are assigned the value 1, one at a time. Once  $x_{n-1}$  is assigned 1, unit propagation leads to a conflict –  $x_n$  must be assigned both 1 and 0 by unit propagation. This causes a backtrack, which ends up setting  $x_{n-1}$  to 0. Once this happens, unit propagation assigns the value 0 to  $x_n$ , resulting in the partial assignment  $x_{n-1} = x_n = 0$ , which satisfies all clauses. Therefore, there is exactly one backtrack.

In part (b), every time a variable  $x_i$  for  $0 \leq i \leq n-1$  is assigned 1, unit propagation causes  $x_{n+i}$  to be in conflict (must be assigned both 0 and 1). This induces a backtrack that sets  $x_i$  to 0, followed by unit propagation setting  $x_{n+i}$  to 0. DPLL will then choose  $x_{i+1}$  as the next decision variables, assign it 1 and the above process repeats. So, in this case, DPLL will incur  $n$  backtracks, one for each of  $x_0, \dots, x_{n-1}$ .

2.  $F = (a \vee b)$ . Any formula  $\phi$  which is equivalent to a Horn formula has the following property: if  $v_1$  and  $v_2$  are two valuations that make  $\phi$  evaluate to  $\top$ , then the valuation  $v_1 \wedge v_2$  also makes  $\phi$  evaluate to  $\top$ . Now consider  $F = (a \vee b)$ .  $F$  is satisfied by  $(\top, \perp)$  and  $(\perp, \top)$  but not  $(\perp, \perp)$ . Hence, no Horn Formula can represent  $F$ .

## 4. A Game of Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic Sudoku, the objective is to fill a  $9 \times 9$  grid with digits so that each column, each row, and each of the nine  $3 \times 3$  subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9 (and as one can naturally deduce; contain each of the digits exactly once). The puzzle setter provides a partially completed grid, which has a single solution for a well-posed puzzle. Encode the puzzle as a CNF formula. You are free to play around with different encodings; use auxiliary variables and attempt to make succinct (read; "optimised") formulae.

**Solution:** Let  $p(i, j, k)$  be a propositional variable that asserts if the cell in row  $i$  and column  $j$  has the value  $k$ . One can clearly note that  $1 \leq i, j, k \leq 9$ . We encode the following constraints

- Every cell contains at least one number:

$$\phi_1 = \bigwedge_{i=1}^9 \bigwedge_{j=1}^9 \bigvee_{k=1}^9 p(i, j, k)$$

- Every cell contains at most one number:

$$\phi_2 = \bigwedge_{i=1}^9 \bigwedge_{j=1}^9 \bigwedge_{x=1}^8 \bigwedge_{y=x+1}^9 (\neg p(i, j, x) \vee \neg p(i, j, y))$$

- Every row contains every number:

$$\phi_3 = \bigwedge_{i=1}^9 \bigwedge_{n=1}^9 \bigvee_{j=1}^9 p(i, j, n)$$

- Every column contains every number:

$$\phi_4 = \bigwedge_{j=1}^9 \bigwedge_{n=1}^9 \bigvee_{i=1}^9 p(i, j, n)$$

- Every  $3 \times 3$  box contains every number:

$$\phi_5 = \bigwedge_{r=0}^2 \bigwedge_{s=0}^2 \bigwedge_{n=1}^9 \bigvee_{i=1}^3 \bigvee_{j=1}^3 p(3r + i, 3s + j, n)$$

The final formula is as follows

$$\phi = (\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5)$$

## 5. Poring over proofs

A student has given the following proof of  $\top \vdash x \rightarrow \neg x$ . What are the sources of problem in this proof (else we would be in serious trouble with true being equivalent to false).

1. top

2.	x	assumption	
3.	neg x	assumption	
4.	bot	bot introduction rule on 2 and 3	
5.	neg x	bot elimination rule on 4	
6.	bot	bot intro rule on 2 and 5	
7.	neg x	neg intro rule on 2 -- 6	
8.	x	assumption	
9.	bot	bot intro rule on 7 and 8	
10.	neg x	bot elim rule on 9	
11.	x $\rightarrow$ neg x	impl intro rule on 8 to 10	

**Solution:** Thankfully, there is an error in the proof.

Step 5 infers  $\neg x$  outside the inner box by applying  $\perp$ -elimination rule on the result of Step 4. However,  $\perp$  was derived in the scope of the inner box in Step 4. Deriving  $\neg x$  inside the inner box after Step 4 would have been fine, but  $\perp$ -elimination doesn't allow us to infer  $\neg x$  outside the scope of the inner box.

## 6. The Resolution Proof System

Consider the formula  $\bigoplus_{i=1}^n x_i$ , where  $\oplus$  represents xor. It can be shown by induction that this is semantically equivalent to the PARITY function that evaluates to true if and only if an odd number of variables are assigned true.

Show that  $\bigwedge_{\substack{S \subseteq \{1,2,\dots,n\} \\ |S| \equiv 1 \pmod{2}}} \left( \bigvee_{i \in S} x_i \vee \bigvee_{j \notin S} \neg x_j \right)$  is the only CNF equivalent to  $\varphi$ , upto adding tautological clauses and repeating variables.

**Hint:** If two CNFs  $\varphi$  and  $\psi$  are equivalent, then for every clause  $\alpha$  in  $\varphi$ , we have  $\psi \vdash \alpha$  and for every clause  $\beta$  in  $\psi$ , we have  $\varphi \vdash \beta$ , where the proofs used are resolution proofs.

As a refresher, the resolution proof system is a system of proof rules for CNFs that is both sound and complete, ie if  $\varphi$  and  $\psi$  are CNFs, then  $\varphi \models \psi$  if and only if  $\varphi \vdash \psi$ . We say  $\varphi \vdash \psi$  iff for every clause  $c$  in  $\psi$ , we have  $\varphi \vdash c$ . The proof rules that can be applied in resolution proofs are:

1. (*Assumption*) For every clause  $c$  in  $\varphi$ ,  $\varphi \vdash c$
2. (*Resolution*) If we have  $\varphi \vdash p \vee c_1$  and  $\varphi \vdash \neg p \vee c_2$  for any clauses  $c_1$  and  $c_2$  and propositional variable  $p$ , then we can deduce  $\varphi \vdash c_1 \vee c_2$ .
3. (*Or Introduction*) For any clauses  $c_1$  and  $c_2$ , if we have  $\varphi \vdash c_1$ , then we can deduce  $\varphi \vdash c_1 \vee c_2$

Other than these, we are implicitly allowed to reorder any of the literals within a clause and any of the clauses within a CNF, and we can remove tautological clauses from the CNF (these are the clauses that contain a variable as well as it's negation).

**Solution:** An assignment  $\alpha$  satisfies the formula if and only if the number of variables set to 1 by  $\alpha$  is odd. An assignment does not satisfy the CNF given in the problem (call it  $\varphi$ ) if and only if the some clause is falsified. This occurs if and only if an even number of variables are set to true. Therefore, an assignment satisfies  $\varphi$  if and only if an odd number of variables are set to 1. Therefore,  $\varphi$  in the question is semantically equivalent to  $\bigoplus_{i=1}^n x_i$ .

Assume there is some other CNF  $\psi$  also equivalent to  $\bigoplus_{i=1}^n x_i$ , where there are no tautological clauses in  $\psi$  and no repeated variables in a single clause.  $\varphi$  and  $\psi$  will be equivalent CNFs, ie  $\varphi \models \psi$  and  $\psi \models \varphi$ .

From the first condition, we can conclude that for any clause  $c$  in  $\psi$ , we have  $\varphi \vdash c$  by a resolution proof.

The **key idea** here is to note that when two clauses of  $\chi$  are resolved, only tautological clauses result. Say there are clauses  $c_1 = x \cup C_1$  and  $c_2 = \neg x \cup C_2$  being resolved with respect to the variable  $x$ . There must be some other variable  $y$  such that  $y$  is present in  $C_1$  and  $\neg y$  is present in  $C_2$  (or vice versa). This is as the number of non-negated variables in each clause in  $\chi$  must be odd. Therefore, the resolvent  $C_1 \cup C_2$ , will contain both  $y$  and  $\neg y$  and will therefore be a tautology.

Also note that since each clause of  $\varphi$  already contains every variable, applying *or introduction* on a clause either leaves it unchanged, or results in a tautology.

This means that the only proof rule that can be used to get non-tautological clauses is *Assumption*. Hence, if  $\varphi \vdash c$ ,  $c$  must have been a clause of  $\varphi$  in the first place. This means that all the clauses in  $\psi$  must have already been present in  $\varphi$ .

Now, we must also have  $\psi \vdash c$  for every clause  $c$  in  $\varphi$ . Since every clause in  $\psi$  is also a clause in  $\varphi$ , we can again conclude that *resolution* on them also produces only tautologies,



and similarly *or introduction* is useless as well. Therefore, if  $\psi \vdash c$ , then  $c$  must have been a clause of  $\psi$ . Therefore, every clause in  $\varphi$  must also be present in  $\psi$ . Hence the clauses of  $\psi$  and  $\varphi$  must be the same, ie  $\psi$  and  $\varphi$  must be the same, meaning that the CNF obtained is unique.

## 7. A flavour of DFAs

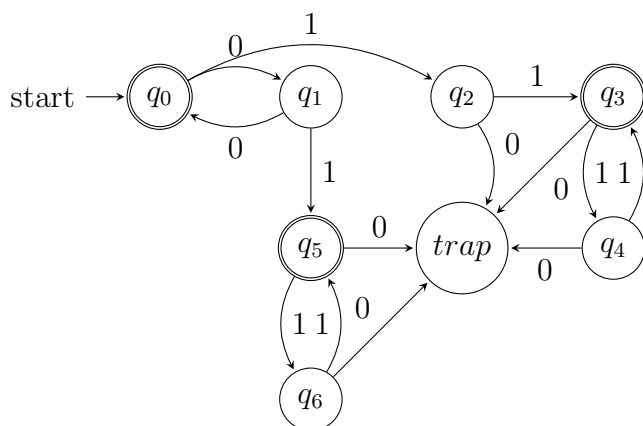
Consider a DFA  $M$  with the set of states  $Q$ , alphabet  $\Sigma = \{0, 1\}$ , transition function  $\delta$ , and accepting state  $F$ .

Draw the state diagram for the DFA  $M$  that accepts the language  $L = \{0^n 1^m \mid m + n \text{ is even}\}$ . The DFA should recognize strings where the total number of '0's and '1's is even. Provide the diagram along with the state transitions.

**Note:**  $L(M)$  represents the language accepted by DFA  $M$ , and  $\sim_L$  is the equivalence relation induced by the language.

### Solution:

The state diagram for the DFA  $M$  is as follows:



where the set of states  $Q = \{q_0, q_1, \dots, q_6, \text{trap}\}$ , alphabet  $\Sigma = \{0, 1\}$ , transition function  $\delta$ , and accepting state  $F = \{q_0, q_5, q_3\}$  and initial state is  $q_0$ .

State	Input 0	Input 1
$q_0$	$q_1$	$q_2$
$q_1$	$q_0$	$q_5$
$q_2$	trap	$q_3$
$q_3$	trap	$q_4$
$q_4$	trap	$q_3$
$q_5$	trap	$q_6$
$q_6$	trap	$q_5$

Explanation for the states:

- $q_0$ : Even no of 0's
- $q_1$ : Odd no of 0's
- $q_2$ : Odd no of 1's after even 0's
- $q_3$ : Even no of 1's
- $q_4$ : Odd no of 1's
- $q_5$ : Odd no of 1's after odd 0's
- $q_6$ : Even no of 1's
- trap: If 0's seen after 1's

## 8. More solved problems on DFA

Please look at some solved examples from the textbook *Introduction to Automata Theory, Languages and Computation* by J.E. Hopcroft, R. Motwani and J.D. Ullman (2nd or later editions). For example, you can look at Examples 2.2, 2.4, 2.5 to get a feel of DFAs for simple languages.

# Homework 1 Solutions

## Question 1 : Implication "Equations"

15 points

In this question, we will consider "implications" in the same spirit as "equations" between unspecified propositional formulas. You can think of these as "equations" where the unknowns are propositional formulas themselves, and the  $=$  symbol has been replaced by  $\rightarrow$ .

Let  $\varphi_1$  and  $\varphi_2$  be unknown propositional formulas over  $x_1, x_2, \dots, x_n$ . Consider the following implications labelled (1a), (1b) through (na), (nb). A solution to this set of simultaneous implications is a pair of specific propositional formulas  $(\varphi_1, \varphi_2)$  such that all implications are **valid**, i.e. evaluate to true for all assignments of variables.

(1a) $(x_1 \wedge \varphi_1) \rightarrow \varphi_2$	(1b) $(x_1 \wedge \varphi_2) \rightarrow \varphi_1$
(2a) $(x_2 \wedge \varphi_1) \rightarrow \varphi_2$	(2b) $(x_2 \wedge \varphi_2) \rightarrow \varphi_1$
$\vdots$	$\vdots$
(na) $(x_n \wedge \varphi_1) \rightarrow \varphi_2$	(nb) $(x_n \wedge \varphi_2) \rightarrow \varphi_1$

In each of the following questions, you must provide complete reasoning behind your answer. Answers without reasoning will fetch 0 marks.

1. [5 marks] Suppose we are told that  $\varphi_1$  is  $\perp$ . How many semantically distinct formulas  $\varphi_2$  exist such that implications (1a) through (nb) are valid?
2. [2 marks] Answer the above question, assuming now that  $\varphi_1$  is  $\top$ .
3. [5 marks] If we do not assume anything about  $\varphi_1$ , how many semantically distinct pairs of formulas  $(\varphi_1, \varphi_2)$  exist such that all the implications are valid?
4. [3 marks] Does there exist a formula  $\varphi_1$  such that there is exactly one formula  $\varphi_2$  (modulo semantic equivalence) that can be paired with it to make  $(\varphi_1, \varphi_2)$  a solution of the above implications?

**Solution:** First note that the entire space of assignments of  $x_1, \dots, x_n$  can be partitioned into sub-spaces where  $x_1$  is true,  $\neg x_1 \wedge x_2$  is true,  $\neg x_1 \wedge \neg x_2 \wedge x_3$  is true, and so on until  $\neg x_1 \wedge \dots \wedge \neg x_n$  is true.

1. All implications in the left column reduce to  $\perp \rightarrow \varphi_2$ . Every such implication is trivially valid (since  $\neg \perp \vee \varphi_2$  is true for all  $\varphi_2$  for all assignments of variables). All implications in the right column reduce to  $(x_i \wedge \varphi_2) \rightarrow \perp$ . This implication is valid iff  $(x_i \wedge \varphi_2)$  evaluates to false for all assignments of the variables. Using the partitioning of variable assignments mentioned above,  $\varphi_2$  must evaluate to false whenever any of the variables  $x_1, \dots, x_n$  evaluates to true. Hence, the semantics of  $\varphi_2$  is determined for all assignments other than  $x_1 = \dots = x_n = 0$ . Since  $\varphi_2$  can have two different truth values (0 or 1) for this assignment, only 2 distinct formulas  $\varphi_2$  are possible.
2. The right column of implications are trivially valid if  $\phi_1 \leftrightarrow \top$  is valid. Using reasoning similar to that above for the left column, once again only 2 distinct formulas  $\varphi_2$  are possible.
3. Note that  $(x_i \wedge \phi_j) \rightarrow \phi_k$  is semantically equivalent to  $x_i \rightarrow (\phi_j \rightarrow \phi_k)$ . This can be easily checked through truth tables. Therefore, if implications (1a) and (1b) are both valid, then  $x_1 \rightarrow (\varphi_1 \leftrightarrow \varphi_2)$  must also be valid. Similarly, for implications (2a) and

(2b), and so on until (na) and (nb). Thus, if all the implications are to be valid, then  $\varphi_1 \leftrightarrow \varphi_2$  must evaluate to true for all assignments of variables where at least one of  $x_1, \dots, x_n$  is true. In other words, for every  $\varphi_1$ , the semantics of  $\varphi_2$  can potentially differ from that of  $\varphi_1$  only for the assignment  $x_1 = \dots = x_n = 0$ . Therefore, there are only two possible  $\varphi_2$  for every  $\varphi_1$ . Since the total number of semantically distinct formulas  $\varphi_1$  on  $n$  variables is  $2^{2^n}$  (why?), the total count of semantically distinct pairs  $(\varphi_1, \varphi_2)$  is  $2 \times 2^{2^n} = 2^{2^n+1}$ .

4. The above implications allow  $\varphi_2$  to evaluate to any value in  $\{0, 1\}$  when  $x_1 = x_2 = \dots = x_n = 0$ , regardless of what  $\varphi_1$  is. Therefore, there are always at least two solutions to the given implications.

## Question 2 : Balanced Parentheses

25 points

A *decision problem* is a computational problem that has a "yes" or "no" answer for every given input. An input to a decision problem is often encoded simply as a string of 0's and 1's. Not surprisingly, we can encode *some* decision problems  $P$  in propositional logic. Specifically, we construct a propositional logic formula  $\varphi_P$  over as many propositional variables as the count of letters (0s and 1s) in the binary string encoding the input, such that if the binary string is interpreted as an assignment to the propositional variables, then the "yes"/"no" answer to  $P$  is obtained from the value given by the semantics of  $\varphi_P$ . If the semantics of  $\varphi_P$  evaluates to 0 (or "false") for the given input, then the output of  $P$  for that input is "no"; else, the output is "yes".

Consider the following decision problem of checking if a given string of parentheses is balanced. Given a binary string of length  $n$ ,  $n \geq 1$ , where 0 encodes '(' and 1 encodes ')', we say that the string has balanced parentheses if and only if:

- The number of open parentheses, represented by 0s, in the entire string equals the number of closing parentheses, represented by 1s.
- In every proper prefix of the string, the number of open parentheses is at least as much as the number of closing parentheses.

We wish to encode the above problem in propositional logic. Recall from your data structures and algorithms course that checking balanced parentheses can be solved in polynomial time (in fact, with linear time complexity). We want this to be reflected in some aspects of your solution to this problem.

In class, we saw that every propositional formula  $\varphi$  can be represented by a parse tree whose internal nodes are labelled by connectives and whose leaves are labelled by propositional variables. Sometimes, two different sub-trees in a parse tree may be identical. In such cases, it makes sense to represent the formula as a directed acyclic graph (DAG), where syntactically common sub-formulas are represented exactly once. As an example, consider the parse tree and corresponding DAG in Fig. 1, both representing the formula  $(r \vee (p \vee q)) \wedge (p \vee q)$ . Note

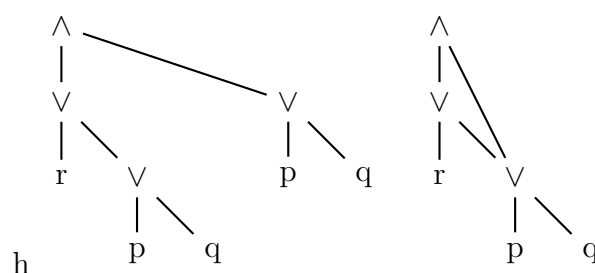


Figure 1: A parse tree and corresponding DAG

that a DAG representation of a formula can be exponentially smaller than a parse tree representation. Furthermore, a DAG representation suffices to evaluate the semantics of a formula, since the semantics of a shared sub-formula needs to be evaluated only once. Thus, if the DAG representation of a formula is small, its semantics can be evaluated efficiently. We define the *DAG size* of a formula to be the number of nodes in its DAG representation. This is also the number of syntactically distinct sub-formulas in the formula.

- (a) [10 marks] Encode the problem of checking balanced parentheses in a binary string of length  $n$  as a propositional logic formula whose DAG size is at most  $\mathcal{O}(n^3)$ . You must use only  $n$  propositional variables corresponding to the  $n$  bits in the input string, and the only connectives allowed in your formula are two-input  $\vee, \wedge$  and  $\neg$ . Express the DAG size of your formula as a function of  $n$  using big- $\mathcal{O}$  notation, with a clear justification of how you obtained the size expression.
- (b) [5 marks] Prove that your formula is unsatisfiable for all odd values of  $n$ .
- (c) [10 (+ bonus 5) marks] Suppose your formula is represented as a DAG, as discussed above. Given an input string of 0s and 1s, there is a simple way to evaluate the semantics of the formula. Specifically, we evaluate DAG nodes bottom-up, starting from the leaves (these represent propositional variables whose values are given by the input string) and moving upwards until we reach the root. The value of the root gives the semantics of the formula. Normally, a DAG node (labelled  $\wedge, \vee$  or  $\neg$ ) can be evaluated only after all its children have been evaluated. However, if a  $\vee$  (resp.  $\wedge$ ) labelled node has a child that has evaluated to 1 (resp. 0), then the node itself can be evaluated to 1 (resp. 0) without evaluating its other child. This can allow us to find the value of the root without evaluating all nodes in the DAG.

What is the worst-case number of DAG nodes (as a function of  $n$  in big- $\mathcal{O}$  notation) that need to be evaluated for your formula in part (a), in order to find the value at the root node for any input string of length  $n$ ? You must give justification for why you really need these many nodes to be evaluated in the worst-case. Answers without justification will fetch 0 marks.

You will be awarded bonus 5 marks if you can show that your formula requires evaluating only  $\mathcal{O}(n)$  DAG nodes.

**Solution:** Let  $T_{i,j}$  represent the subformula, which evaluates to  $\top$  if the number of open brackets exceeds the number of closed brackets by  $j$  in the first  $i$  characters of the string. Our formula for balanced parenthesis becomes  $T_{n,0}$ .

To make this formula well-defined, we must define the complete set of  $T_{i,j}$  for all values of  $i, j$ . For all values of  $i > 0, j < 0$ , we have  $T_{i,j} = \perp$  and for all values of  $j \neq 0$ , we have  $T_{0,j}$  as  $\perp$  and  $T_{0,0} = \top$ . For all other values of  $i, j$ , we have  $T_{i,j} = (x_i \wedge T_{i-1,j+1}) \vee (\neg x_i \wedge T_{i-1,j-1})$ . This makes the formula well-defined.

Lemma (well-defined): The complete set of propositional variables in the above formula is contained in  $\{x_i\}$ . We prove this by induction. (Base Cases are  $T_{0,j}$  and induction over  $i$ )

Lemma (Size): The number of distinct subformulae is in  $\mathcal{O}(n^3)$ . We prove this by explicitly upper bounding the number of distinct subformulae.

Lemma (Correctness): If  $T_{n,0}$  evaluates to  $\top$  if and only if the string is balanced. We prove this using induction over even values of  $n$ .

(b) In the same way, for part b, we do induction over odd values of  $n$  and show that the formula is unsat.

(c) Claim: This formula can be evaluated in  $\mathcal{O}(n)$  time.

Evaluation Algorithm: We evaluate  $x_n$  and, based on whether it is  $\top$  or  $\perp$ , evaluate the appropriate branch in the formula.

We prove the above evaluation is correct using induction and has  $\mathcal{O}(n)$  time complexity.

# Homework 2 Solutions

## 1. The Case of Dr. Equisemantic and Mr. Irredundant 15 points

Assume we have a countably infinite list of propositional variables  $p_1, p_2, \dots$ . For this problem, by “formula”, we always mean a finite string representing “syntactically-correct formula”. Let  $\Sigma$  be a set of formulae. For any formula  $\varphi$ , we say  $\Sigma \models \varphi$  (read as  $\Sigma$  semantically entails  $\varphi$ ) if for any assignment  $\alpha$  of the propositional variables that makes all the formulae contained in  $\Sigma$  true,  $\alpha$  also makes  $\varphi$  true.

Let us call two sets of formulae  $\Sigma_1$  and  $\Sigma_2$  *equisemantic* if for every formula  $\varphi$ , we have  $\Sigma_1 \models \varphi$  if and only if  $\Sigma_2 \models \varphi$ . Furthermore, let us call a non-empty set of formulae  $\Sigma$  *irredundant* if no formula  $\sigma$  in  $\Sigma$  is semantically entailed by  $\Sigma \setminus \{\sigma\}$ .

1. [5 points] Show that any set of formulae  $\Sigma$  must always be countable. This implies that we can enumerate the elements of  $\Sigma$ . Assume from now on that  $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \dots\}$ .
2. Suppose we define  $\Sigma'$  as follows:

$$\begin{aligned} \Sigma' = \{ & \sigma_1, \\ & (\sigma_1) \rightarrow \sigma_2, \\ & (\sigma_1 \wedge \sigma_2) \rightarrow \sigma_3, \\ & (\sigma_1 \wedge \sigma_2 \wedge \sigma_3) \rightarrow \sigma_4, \\ & \vdots \\ & \} \end{aligned}$$

Suppose we remove all the tautologies from  $\Sigma'$  and call this reduced set  $\Sigma''$ . Prove that  $\Sigma''$  is irredundant and equisemantic to  $\Sigma$ . You can proceed as follows:

- (a) [5 points] Show that a non-empty satisfiable set  $\Gamma$  with  $|\Gamma| \geq 2$  is irredundant if and only if  $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$  is satisfiable for every  $\gamma \in \Gamma$ .
- (b) [5 points] Use the above result to show that  $\Sigma''$  is irredundant and equisemantic to  $\Sigma$ .

### Solution:

1. Recall from your course on Discrete Structures that a set is *countable* if either it is finite, or if there is a bijection between the set and the set of natural numbers. Equivalently, a set is countable if there is an injective function from the set to the set of natural numbers. Recall also that the Cartesian product of two countable sets is countable, and the countable union of countable sets is also countable.

Now, consider the set  $S$  of all finite strings. This set must be countable. To see why this is so, take the set  $S_1$  of all strings of length 1. This set is countable because the number of propositional variables is countable and we only have an additional finite set of non-variable symbols (parentheses, logical and symbol, etc.). Now consider the set  $S_2$  of all strings of length 2. Being the Cartesian product of two countable sets ( $S_1 \times S_1$ ), this set must also be countable. Inductively, the set  $S_n$  of all strings of length  $n$ , for any  $n \geq 2$ , is the Cartesian product of two countable sets ( $S_{n-1} \times S_1$ ), and hence must be countable. Clearly  $S = \bigcup_{n=1}^{\infty} S_n$ . Being a countable union of countable sets, the set  $S$  must therefore be countable.

The set of all syntactically-correct formulas is a (strict) subset of the set of all possible finite strings. Being a subset of a countable set, it must be countable.  $\Sigma$ , in turn, is a subset of the set of all syntactically-correct formulas. Being a subset of a countable set,  $\Sigma$  must be countable.

2. (a) Note first that since  $\Gamma$  is satisfiable, so is  $\Gamma \setminus \{\gamma\}$  (every  $\alpha$  that satisfies all formulas in  $\Gamma$  certainly satisfies all formulas in  $\Gamma \setminus \{\gamma\}$ ). Now, if  $\gamma$  is not semantically entailed by  $\Gamma \setminus \{\gamma\}$ , there must be some assignment  $\alpha$  that makes each formula in  $\Gamma \setminus \{\gamma\}$  true but makes  $\gamma$  false. This same assignment  $\alpha$  would thus make  $\neg\gamma$  true, and hence would be a satisfying assignment for  $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$ . If this holds for every  $\gamma \in \Gamma$ , this means that no element of  $\Gamma$  is semantically entailed by the rest of the elements. Hence  $\Gamma$  is irredundant.

To prove the other direction, suppose  $\Gamma$  is irredundant. By definition, no formula  $\gamma \in \Gamma$  is semantically entailed by  $\Gamma \setminus \{\gamma\}$ . In other words, for every  $\gamma \in \Gamma$ , there exists an assignment  $\alpha$  (dependent on  $\gamma$  in general) that satisfies every formula in  $\Gamma \setminus \{\gamma\}$ , but does not satisfy  $\gamma$ . Hence, this  $\alpha$  satisfies all formulas in  $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$ . Since the above argument holds for all  $\gamma \in \Gamma$ , this proves the statement.

- (b) The fact that  $\Sigma''$  is equisemantic to  $\Sigma$  is easy to see. We will first show that an assignment  $\alpha$  satisfies all formulas in  $\Sigma$  iff it satisfies all formulas in  $\Sigma''$ . Clearly, if each of the formulas in  $\Sigma$  evaluates to true for  $\alpha$ , then both sides of each implication in  $\Sigma'$  evaluate to true for  $\alpha$ . Hence, each of the formulas in  $\Sigma'$  is also true for the same assignment. Conversely, if each of the formulas in  $\Sigma'$  is true for assignment  $\alpha$ , then  $\sigma_1$  is true, and since  $(\sigma_1) \rightarrow \sigma_2$  is true, this implies  $\sigma_2$  is true. Using the same reasoning, it follows by induction that  $\sigma_n$  is true for all  $n \geq 1$ . Thus, each formula in  $\Sigma$  is true for the assignment  $\alpha$ .

Let  $S$  denote the set of satisfying assignments of  $\Sigma$ . We have just shown above that  $S$  is also the set of satisfying assignments of  $\Sigma'$ . Now suppose  $\Sigma \models \varphi$ . This is equivalent to saying that every assignment in  $S$  satisfies  $\varphi$ . Since  $S$  is also the set of satisfying assignments of  $\Sigma'$ , this is equivalent to saying that  $\Sigma' \models \varphi$ . Hence, if  $\Sigma \models \varphi$ , then  $\Sigma' \models \varphi$  too. A similar reasoning shows the result the other way round. Hence,  $\Sigma$  and  $\Sigma'$  are equisemantic.

Since  $\Sigma''$  is just  $\Sigma'$  without the tautologies, and tautologies, being always true, do not change equisemanticness,  $\Sigma''$  is equisemantic to  $\Sigma$ .

Let us now look at irredundancy. Let  $\Sigma'' = \{\eta_1, \eta_2, \dots\}$  in order after removing the tautologies from  $\Sigma'$ . Consider any  $\eta_n \in \Sigma''$ . Since  $\eta_n$  is not a tautology, there must be an assignment  $\alpha$  that makes  $\eta_n$  false. By the semantics of  $\rightarrow$ , this assignment must make each of  $\sigma_1, \dots, \sigma_{n-1}$  true and  $\sigma_n$  false. Since  $\alpha$  makes each of  $\sigma_1, \dots, \sigma_{n-1}$  true, it satisfies all implications  $\eta_1, \dots, \eta_{n-1}$  (both sides of implication having formulas in  $\{\sigma_1, \dots, \sigma_{n-1}\}$ , must evaluate to true). Similarly,  $\alpha$  satisfies all implications  $\eta_{n+1}, \dots$ , since the left side of each of these implications has  $\sigma_n$  conjuncted with other formulas, and  $\sigma_n$  evaluates to false under assignment  $\alpha$ . Therefore,  $\alpha$  is a satisfying assignment for  $\Sigma'' \setminus \{\eta_n\}$ . Since  $\alpha$  also falsifies  $\eta_n$ , it immediately follows that  $\alpha$  satisfies all formulas in  $(\Sigma'' \setminus \{\eta_n\}) \cup \{\neg\eta_n\}$ . Since the above argument holds for every  $\eta_n \in \Sigma''$ , we conclude that  $\Sigma''$  has no element that is semantically entailed by the others. Hence  $\Sigma''$  is irredundant.



## 2. A follow-up of the take-away question of Tutorial 2 25 points

To recap from the take-away question of Tutorial 2, we will view the set of assignments satisfying a set of propositional formulae as a language and examine some properties of such languages.

Let  $\mathbf{P}$  denote a countably infinite set of propositional variables  $p_0, p_1, p_2, \dots$ . Let us call these variables positional variables. Let  $\Sigma$  be a countable set of formulae over these positional variables. Every assignment  $\alpha : \mathbf{P} \rightarrow \{0, 1\}$  to the positional variable can be uniquely associated with an infinite bitstring  $w$ , where the  $i^{\text{th}}$  bit  $w_i = \alpha(p_i)$ . The language defined by  $\Sigma$ , also called  $L(\Sigma)$ , is the set of bitstrings  $w$  for which the corresponding assignment  $\alpha$ , that has  $\alpha(p_i) = w_i$  for each  $i$ , satisfies  $\Sigma$ , that is, for each formula  $F \in \Sigma$ ,  $\alpha \models F$ . In this case, we say that  $\alpha \models \Sigma$ . Let us call the languages definable in this manner as PL-definable languages.

### Example:

Let  $\Sigma = \{p_0 \rightarrow p_1, p_1 \rightarrow p_2, p_2 \rightarrow p_3, \dots\}$ .

Then  $L(\Sigma) = \{1111\dots, 0111\dots, 0011\dots, 0001\dots, \dots, 0000\dots\}$ , or, to be precise, if we denote the infinite bitstring containing only 1s by  $1^\omega$  and the infinite bitstring containing only 0s by  $0^\omega$ , and the finite bitstring consisting of  $k$  0s by  $0^k$ , then  $L(\Sigma) = \{0^k 1^\omega : k \in \mathbb{N}\} \cup \{0^\omega\}$ .

- (a) [5 marks] Show that the language  $L$  consisting of all infinite bitstrings except  $000\dots$  (the bitstring consisting only of zeroes) is **not** PL-definable. You may want to prove the following lemma in order to solve this question:

### Lemma:

For every PL-definable language  $L$  and bitstring  $x \notin L$  there exists a finite prefix  $y$  of  $x$  such that for any infinite bitstring  $w$ ,  $yw \notin L$  ( $yw$  refers to the concatenation of  $y$  and  $w$ ).

- (b) [5 + 5 points] Show that PL-definable languages are closed neither under countable union nor under complementation.

**Hint:** Try using the result proven in part (a)

- (c) [10 points] Show that a PL-definable language either contains every bitstring or does not contain uncountably many bitstrings.

**Hint:** Try using the lemma proven in part (a)

- (d) [Bonus 10 points] A student tries to extend the definition of PL-languages by allowing the use of "dummy" variables.

Let  $\mathbf{X} = \{x_0, x_1, \dots\}$  denote a countably infinite set of "dummy" variables and let  $\Sigma$  denote a countable set of formulae over both positional and dummy variables. An infinite bitstring  $w$  is in the language defined by  $\Sigma$  if and only if there exists an assignment  $\alpha : \mathbf{P} \cup \mathbf{X} \rightarrow \{0, 1\}$  such that  $\alpha \models \Sigma$  and  $w_i = \alpha(p_i)$  for each  $i$ . Note that the assignment of "dummy" variables in  $\mathbf{X}$  are not represented in  $w$ . Let us call the languages definable this way extended PL-definable languages, or EPL-definable languages.

Show that EPL and PL are equally expressive, ie every EPL-definable language is a PL definable language and vice versa. This means our attempt to strengthen PL this way has failed. You can use the following theorem without proof:

### Theorem:

Let  $S_0, S_1, S_2, \dots$  denote an infinite sequence of non-empty sets of finite bitstrings such that for every  $i > 0$  and for every bitstring  $x \in S_i$  and every  $j \leq i$ , there exists a prefix  $y$  of  $x$  in  $S_j$ . Then there exists an infinite bitstring  $z$  such that every  $S_i$  contains a prefix of  $z$ .

### Solution:

- (a) Let us first prove the lemma mentioned in the question.

#### Lemma:

For every PL-definable language  $L$  and bitstring  $x \notin L$  there exists a finite prefix  $y$  of  $x$  such that for any infinite bitstring  $w$ ,  $yw \notin L$  ( $yw$  refers to the concatenation of  $y$

and  $w$ ).

**Proof:**

Say  $L = L(\Sigma)$  for some countable set of formulae  $\Sigma$ . If  $x \notin L(\Sigma)$ , then there must be some  $F \in \Sigma$  such that  $x \not\models F$ . Since  $F$  is a formula, and all formulas are finite strings by definition,  $F$  contains only a finite number of positional variables  $p \in \mathbf{P}$ . Let the largest index of any positional variable present in  $F$  be  $n$ . For any infinite bitstring  $z$  with first  $n + 1$  bits being the same as  $x$ , the values of all the positional variables present in  $F$  are the same in both  $x$  and  $z$ , which means that  $z \not\models F$  as well. This means that, if we let  $y$  denote the finite prefix of  $x$  consisting of the first  $n + 1$  bits of  $x$ , then for any infinite bitstring  $w$ ,  $yw \not\models F$ , by the same argument. This means that  $yw \notin L$ , proving the lemma. In fact, by a similar argument, the lemma can be strengthened to stating that for any infinite bitstring  $x \notin L$ , there exists a finite set of positions  $S$ , such that for any infinite bitstring  $y$ , such that bits of  $y$  at the positions in  $S$  match those of  $x$ ,  $y \notin L$  as well.

Now, consider  $L$  as defined in the problem, ie consisting of every infinite bitstring, except  $000\dots$ . Let  $x = 000\dots$ . We will show that  $L$  is not PL-definable, by contradiction. Assume  $L$  is PL-definable. By the lemma we just proved, there exists a finite prefix  $y$  of  $x$  such that for any infinite bitstring  $w$ ,  $yw \notin L$ , ie there are infinitely many bitstrings not in  $L$ . This contradicts the fact that  $000\dots$  is the only bitstring not in  $L$ . Therefore,  $L$  cannot be defined in PL.

- (b) Consider the language  $L = \{000\dots\}$  (this language consists only of the infinite bitstring  $000\dots$ ). This language can be defined in PL as  $L(\Sigma)$  where  $\Sigma = \{\neg p_0, \neg p_1, \neg p_2\dots\}$ . However, its complement is the language consisting of all infinite bitstrings other than  $000\dots$ , which, as shown earlier, is not PL-definable. Therefore, PL-definable languages are not closed under complementation.

Consider the countably infinite family of languages  $L_i = L(\{p_i\})$  for each  $i \in \mathbb{N}$ . Each  $L_i$  consists of strings where the  $i^{th}$  bit is 1, and clearly, each  $L_i$  is PL-definable. Let  $L = \bigcup_{i=0}^{\infty} L_i$ .  $L$  is the language consisting of all infinite bitstrings other than  $000\dots$ , which, as shown earlier, is not PL-definable. Therefore, PL-definable languages are not closed under countable union.

- (c) This follows directly from the lemma that was proven earlier. If a PL-definable language does not contain an infinite bitstring  $x$ , then there exists a finite prefix  $y$  of  $x$  such that for every infinite bitstring  $w$ ,  $yw$  is not in the language. Since there are uncountably many infinite bitstrings  $w$ , there are uncountably many infinite bitstrings not in the language.
- (d) Firstly, it is easy to see that every PL-definable is also EPL-definable: PL is a special case of EPL where no dummy variables are used.

We will now show that every EPL-definable language is PL-definable. Say  $\Sigma = \{F_0, F_1, \dots\}$  is a countable set of formulae over the variables in  $\mathbf{P} \cup \mathbf{X}$ .

Consider  $\Sigma' = \{\bigwedge_{j=0}^i F_j : i \in \mathbb{N}\} = \{F_0, F_0 \wedge F_1, F_0 \wedge F_1 \wedge F_2, \dots\}$ . We will show that  $L(\Sigma) = L(\Sigma')$ .

For any word  $w$ ,  $w \in L(\Sigma)$  if and only if there exists an assignment  $\alpha : \mathbf{P} \cup \mathbf{X} \rightarrow \{0, 1\}$  such that  $w_i = \alpha(p_i)$  for each natural  $i$  and for each  $F \in \Sigma$ ,  $\alpha \models F$ , ie  $\alpha \models F_0$ ,  $\alpha \models F_1$ , and so on. This is equivalent to saying  $\alpha \models F_0$ ,  $\alpha \models F_0 \wedge F_1$ ,  $\alpha \models F_0 \wedge F_1 \wedge F_2$ , so on,

ie  $w \in L(\Sigma')$ . Therefore,  $L(\Sigma) \subseteq L(\Sigma')$ . In a similar manner, if  $w \in L(\Sigma')$ , then there cannot be any  $F_i$  such that the corresponding assignment  $\alpha \not\models F_i$ . Hence,  $w \models \Sigma'$ , and  $L(\Sigma) \subseteq L(\Sigma')$ . From the two inclusions proved above, we have  $L(\Sigma) = L(\Sigma')$ . We will henceforth work with  $\Sigma'$ , and denote  $F_0 \wedge F_1 \cdots \wedge F_i$  as  $F'_i$ .  $F'_i$  satisfy a special property - for any assignment  $\alpha$ , if  $\alpha \models F'_i$ , then for every  $j \leq i$ ,  $\alpha \models F'_j$ .

### Some Notation:

- Let  $Vars_d(F)$  denote the set of dummy variables whose indices are at most the largest index of a dummy variable in the formula  $F$  - for example, if  $F = p_0 \vee x_0 \vee x_2$ , then  $Vars_d(F) = \{x_0, x_1, x_2\}$ . We have  $Vars_d(F'_i) \subseteq Vars_d(F'_{i+1})$  for every natural  $i$ .
- For any EPL formula  $F$ , let  $Ass_d(F)$  denote the set of possible assignments to the set  $Vars_d(F)$  of dummy variables
- For any formula  $F$  and assignment  $\alpha$  to the dummy variables in  $F$ , let  $F(\alpha)$  denote the formula obtained by substituting each dummy variable  $x$  with its value  $\alpha(x)$ . Note that  $F(\alpha)$  no longer contains any dummy variables and only has positional variables, ie it is a formula in PL.

Define  $\Sigma'' = \{ \bigvee_{\alpha \in Ass_d(F'_i)} F'_i(\alpha) : F'_i \in \Sigma' \}$ . This is a countable set of PL-formulae. We will show that  $L(\Sigma'') = L(\Sigma')$ , which will imply that  $L(\Sigma'') = L(\Sigma)$ .

Say some word  $w \in L(\Sigma')$ . This means there exists some assignment  $\alpha : \mathbf{P} \cup \mathbf{X} \rightarrow \{0, 1\}$  such that  $\alpha(p_i) = w_i$  and  $\alpha \models F'_i$  for each natural  $i$ . Now, since  $\alpha \models F'_i$ , we have  $\alpha \models F'_i(\alpha)$ , which implies that  $\alpha \models \bigvee_{\alpha \in Ass_d(F'_i)} F'_i(\alpha)$  for each natural  $i$ , which means  $w \in L(\Sigma'')$ .

On the other hand, say  $w \in L(\Sigma'')$ . This means that  $w \models \bigvee_{\alpha \in Ass(Vars_d(F'_i))} F'_i(\alpha)$  for each  $i$ , ie for each  $i$ , there exists an assignment  $\alpha_i : Vars_d(F'_i) \rightarrow \{0, 1\}$  such that  $w \models F'_i(\alpha_i)$ . Let  $S_i$  denote the set of such assignments, interpreted as finite bitstrings (eg:  $x_0 \rightarrow 0, x_1 \rightarrow 1, x_2 \rightarrow 0$  is interpreted as the bitstring 010). Now, for any  $\alpha \in S_i$ ,  $w \models F'_i(\alpha)$ , which means that for any  $j \leq i$ ,  $w \models F'_j(\alpha)$  as well. This means there is a prefix of the bitstring corresponding to  $\alpha$  in each  $S_j$ , for each  $j \leq i$ . Therefore, the theorem can be applied, and hence there is an infinite bitstring such that every  $S_i$  contains a prefix of it. This infinite bitstring denotes an assignment to the entire set of dummy variables, and hence there exists an assignment  $\alpha : \mathbf{X} \rightarrow \{0, 1\}$  such that  $w \models F'_i(\alpha)$  for each  $i$ , ie there exists an assignment  $\alpha' : \mathbf{P} \cup \mathbf{X} \rightarrow \{0, 1\}$  such that  $\alpha'(p_i) = w_i$  and  $\alpha'(x_i) = \alpha(x_i)$  for each  $i$ , and, as we have seen, such an  $\alpha$  will have  $\alpha \models F'_i$  for each  $i$ . Therefore,  $w \in L(\Sigma')$ . This means that  $w \in L(\Sigma'')$  if and only if  $w \in L(\Sigma')$ , and hence  $L(\Sigma'') = L(\Sigma') = L(\Sigma)$ .

Therefore, every EPL-definable language can also be defined in PL.

## Tutorial 2

1. A *literal* is a propositional variable or its negation. A *clause* is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal. Similarly, a *cube* is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal. A propositional formula is said to be in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses. A formula is said to be in *Disjunctive Normal Form (DNF)* if it is a disjunction of cubes.

Let  $P, Q, R, S, T$  be propositional variables. An example DNF formula is  $(P \wedge \neg Q) \vee (\neg P \wedge Q)$ , and an example CNF formula is  $(P \vee Q) \wedge (\neg P \vee \neg Q)$ . Are they semantically equivalent?

For the propositional formula  $(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$ , find semantically equivalent formulas in CNF and DNF. Show all intermediate steps in arriving at the final result.

*Hint: Use the following semantic equivalences, where we use  $\varphi \Leftrightarrow \psi$  to denote semantic equivalence of  $\varphi$  and  $\psi$ :*

- $\varphi_1 \rightarrow \varphi_2 \Leftrightarrow (\neg \varphi_1 \vee \varphi_2)$
- *Distributive laws:*
  - $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$
  - $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \Leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- *De Morgan's laws*
  - $\neg(\varphi_1 \wedge \varphi_2) \Leftrightarrow (\neg \varphi_1 \vee \neg \varphi_2)$
  - $\neg(\varphi_1 \vee \varphi_2) \Leftrightarrow (\neg \varphi_1 \wedge \neg \varphi_2)$

**Solution:** Propositional formula  $(P \vee T) \rightarrow (Q \vee \neg R) \vee \neg(S \vee T)$ .

**Transformation steps**

1. Eliminate Implications:

$$\neg(P \vee T) \vee ((Q \vee \neg R) \vee \neg(S \vee T))$$

2. Apply De Morgan's Laws:

$$(\neg P \wedge \neg T) \vee ((Q \vee \neg R) \vee (\neg S \wedge \neg T))$$

**Transformation into DNF**

Distribute Conjunction over Disjunction:

$$(\neg P \wedge \neg T) \vee (Q) \vee (\neg R) \vee (\neg S \wedge \neg T)$$

**Transformation into CNF**

Distribute Disjunction over Conjunction:

$$\begin{aligned} &(\neg P \vee Q \vee \neg R \vee \neg S) \wedge (\neg P \vee Q \vee \neg R \vee \neg T) \\ &\wedge (\neg T \vee Q \vee \neg R \vee \neg S) \wedge (\neg T \vee Q \vee \neg R \vee \neg T) \end{aligned}$$

2. Consider the parity function,  $\text{PARITY} : \{0,1\}^n \mapsto \{0,1\}$ , where PARITY evaluates to 1 if and only if an odd number of inputs is 1. In all of the CNFs below, we assume that each clause contains any variable at most once, i.e. no clause contains expressions of the form  $p \wedge \neg p$  or  $p \vee \neg p$ . Furthermore, all clauses are assumed to be distinct.

1. Prove that any CNF representation of PARITY must have  $n$  literals (from distinct variables) in every clause.

*[Hint: Take a clause, and suppose the variable  $v$  is missing from it. What happens when you flip  $v$ ?]*

2. Prove that any CNF representation of PARITY must have  $\geq 2^{n-1}$  clauses.

*[Hint: What is the relation between CNF/DNF and truth tables?]*

PARITY is ubiquitous across Computer Science; for example, in Coding Theory (see Hadamard codes), Cryptography (see the Goldreich-Levin theorem), and discrete Fourier Analysis (yes, that is a thing)! PARITY is ubiquitous across Computer Science; for example, in Coding Theory (see Hadamard codes), Cryptography (see the Goldreich-Levin theorem), and discrete Fourier Analysis (yes, that is a thing)!

**Solution:** Let

$$\text{PARITY} := \bigwedge_{i=1}^m \left( \bigvee_{j=1}^{n_i} \ell_{ij} \right)$$

be the CNF representation of PARITY. We want to prove that  $n_i = n$  for every  $i$ , and  $m \geq 2^{n-1}$ .

1. Suppose  $n_i < n$  for some  $i$ . Negate the entire formula to convert the CNF into a DNF. Now, choose an assignment of literals in the  $i^{\text{th}}$  cube (of the DNF) such that the cube, and hence the whole DNF formula, evaluates to 1. Now, consider a variable  $v$ , which doesn't appear in the  $i^{\text{th}}$  cube, and flip its value. The LHS then becomes 0. However, the  $i^{\text{th}}$  clause stays 1, leading to a contradiction.
2. Once again, consider the DNF. Note that a conjunction of  $n$  literals is satisfied by a unique assignment of variables, and thus, the clauses in the DNF actually encode the assignments that satisfy the formula. Since PARITY is satisfied by  $2^{n-1}$  clauses, we're done.

3. We have already defined CNF and DNF formulas in Question 1. Recall the definition of *equisatisfiable* formulas taught in class. A **stronger notion of equisatisfiability** is as follows: A formula  $F$  with variables  $\{f_1, f_2 \dots f_n\}$  and a formula  $G$  with variables  $\{f_1, f_2 \dots f_n, g_1, g_2 \dots g_m\}$  are **strongly-equisatisfiable** if:

- For any assignment to the  $f_i$ s which makes  $F$  evaluate to true there exists an assignment to the  $g_i$ s such that  $G$  evaluates to true under this assignment with the same assignment to the  $f_i$ s.

AND

- For any assignment setting  $G$  to true, the assignment when restricted to  $f_i$ s makes  $F$  evaluate to true.

Why does this help us? A satisfying assignment for  $G$  tells us a satisfying assignment for  $F$ , and if we somehow discover that  $G$  has no satisfying assignments, then we know that  $F$  also has no satisfying assignments. In other words, the satisfiability of  $F$  can be judged using the satisfiability of  $G$ . Satisfiability is a central problem in computer science (for reasons you will see in CS218 later), and such a reduction is very valuable. In particular, for a formula  $F$  in  $k$ -CNF we look for a formula  $G$  that has a small number of literals in each clause, and the total number of clauses itself is not too large - ideally a linear factor of  $k$  as compared to  $F$ .

Now consider an  $r$ -CNF formula, i.e a CNF formula with  $r$  literals per clause. To keep things simple, suppose  $r = 2^k$ , for some  $k > 0$ . We will start with the simplest case: a  $2^k$ -CNF formula containing a single clause  $C_k = p_0 \vee p_1 \vee p_2 \dots p_{2^k-1}$  with  $2^k$  literals.

1. Let's try using 'selector variables' to construct another formula which is strongly-equisatisfiable with  $C_k$ . The idea is to use these selector variables to 'select' which literals in our clause are allowed to be false. Use  $k$  selector variables  $s_1, s_2, s_3 \dots s_k$  and write a CNF formula that is **strongly-equisatisfiable with**  $C_k$ , such that each clause has size  $O(k)$ .

[Hint: If you consider the disjunction of  $\tilde{s}_1 \vee \tilde{s}_2 \vee \tilde{s}_3 \dots \tilde{s}_k$ , where  $\tilde{s}_i$  denotes either  $s_i$  or  $\neg s_i$ , it is true in all but one of the  $2^k$  possible assignments to the  $s_i$  variables]

2. What is the number of clauses in the CNF constructed in the above sub-question? How small can you make the clauses by repeatedly applying this technique?
3. Consider a  $k$ -CNF formula with  $n$  clauses. When minimizing the clause size using the above technique repeatedly, what is the (asymptotic) blowup factor in the number of clauses? Observe that the factor is almost linear, upto some logarithmic factors.

[Hint: Apply the above reduction repeatedly and see how much net blowup occurs]

4. Can you think of a smarter way to do the reduction that only requires linear blowup?

[Hint: Consider using different and more auxiliary variables (like the 'selector variables' above) to reduce the size of clauses.]

### Solution:

1. The idea is very similar to a MUX (or multiplexer) from digital electronics. We consider the  $2^k$  possible disjunctions:  $s_1 \vee s_2 \vee \dots \vee s_k$ ;  $s_1 \vee s_2 \vee \dots \vee \neg s_k$  etc. We associate a non-negative integer with each such disjunction, where the  $k$ -bit binary encoding of the integer is obtained as  $b_1 b_2 \dots b_k$ , where  $b_k = 1$  if  $s_k$  is in the disjunction and  $b_k = 0$  otherwise. For example, the integer associated with the disjunction  $(s_1 \vee s_2 \vee \dots \vee s_k)$  is  $11 \dots 1$  or  $2^k - 1$ , and the integer associated with  $(\neg s_1 \vee \neg s_2 \vee \dots \vee \neg s_k)$  is  $00 \dots 0$  or 0. Let the disjunction corresponding to the number  $0 \leq q \leq 2^k - 1$  in binary be denoted  $D_q$ . Consider the CNF formula:

$$\bigwedge_{i=0} (D_i \vee p_i)$$

If any of the  $p_i$ s is true, the unique assignment setting  $D_i$  to false satisfies all the other  $2^k - 1$  clauses. On the other hand if all the  $p_i$ s are false, any assignment to the  $s_i$  variables will dissatisfy atleast one  $D_q$ . Thus this formula is strongly equisatisfiable - any assignment setting  $C_k$  to true has a satisfying assignment to the  $s_i$ s, and any assignment setting our formula to true must have one of the  $p_i$ s true, which makes  $C_k$  true.

2. The size of clauses is now  $k + 1$ . The number of clauses is  $2^k$ . Thus, we have obtained an exponential reduction in the size of a clause for a linear factor (the initial size was  $2^k$  blowup in number of clauses. We can reduce the size as long as  $\lceil \log_2(\text{size}) \rceil + 1 < \text{size}$ , which is true as small as size 3. The size CANNOT be reduced below 3 with this method.
3. We obtain logarithmic reduction for linear blowup. Hence, the factor of blowup will look like  $k \cdot \log(k) \cdot \log \log(k) \cdot \log \log \log(k) \cdots \log^{\log^*(k)}(k)$ . The factors after the  $k$  are all logarithmic factors and can be bounded by a polynomial of any degree  $> 0$ .
4. Indeed, we do not need to introduce so many selectors for a clause! We can simply use an indicator  $s_i$  to indicate that a clause after  $p_i$  is satisfied. Thus, the equisatisfiable formula:

$$(p_1 \vee p_2 \vee s_2) \wedge (\neg s_2 \vee p_3 \vee s_3) \wedge (\neg s_3 \vee p_4 \vee s_4) \cdots$$

works and has a linear blowup! Think about Tseitin encoding covered in class.

However this method also fails to lower the size below 3. Indeed we should suspect that it is not possible to reduce the size below 3, because 2-SAT has a linear time solution while 3-SAT is NP-Complete!

#### 4. [Take-away question – solve in your rooms]

In this question we will view the set of assignments satisfying a set of propositional formulae as a language and examine some properties of such languages.

Let  $\mathbf{P}$  denote a countably infinite set of propositional variables  $p_0, p_1, p_2, \dots$ . Let us call these variables positional variables. Let  $\Sigma$  be a countable set of formulae over these positional variables. Every assignment  $\alpha : \mathbf{P} \rightarrow \{0, 1\}$  to the positional variable can be uniquely associated with an infinite bitstring  $w$ , where  $w_i = \alpha(p_i)$ . The language defined by  $\Sigma$  - denoted by  $L(\Sigma)$  - is the set of bitstrings  $w$  for which the corresponding assignment  $\alpha$ , that has  $\alpha(p_i) = w_i$  for each natural  $i$ , satisfies  $\Sigma$ , that is, for each formula  $F \in \Sigma$ ,  $\alpha \models F$ . In this case, we say that  $\alpha \models \Sigma$ . Let us call the languages definable this way PL-definable languages.

- (a) Show that PL-definable languages are closed under countable intersection, ie if  $\mathcal{L}$  is a countable set of PL-languages, then  $\bigcap_{L \in \mathcal{L}} L$  is also PL-definable.
- (b) Show that PL-definable languages are closed under finite union, ie if  $\mathcal{L}$  is a finite set of PL-languages, then  $\bigcup_{L \in \mathcal{L}} L$  is also PL-definable.

[Hint: Try proving that the union of two PL-definable languages is PL-definable. The general case can then be proven via induction. If  $\mathbf{F} = \{F_1, F_2, \dots\}$  and  $\mathbf{G} = \{G_1, G_2, \dots\}$  are two countable sets of formulae, then an infinite bitstring  $w \in L(\mathbf{F}) \cup L(\mathbf{G})$  if and only if either  $w \models$  every  $F_i$  or  $w \models$  every  $G_i$ .]

#### Solution:

- a) Let  $\mathcal{S}$  denote the family of sets of propositional formulae such that  $\mathcal{L} = \{L(\sigma) : \sigma \in \mathcal{S}\}$ . Consider  $\Sigma = \bigcup_{\sigma \in \mathcal{S}} \sigma$ . Since each  $\sigma \in \mathcal{S}$  is countable, and a countable union of countable sets is countable,  $\Sigma$  is countable as well. Now, for any infinite bitstring  $w$ ,  $w \in L(\Sigma)$  if and only if  $w \models F$  for each  $F \in \Sigma$ , for each  $\sigma \in \mathcal{S}$  (we abuse notation and denote the assignment corresponding to  $w$  by  $w$ ). This happens if and only if  $w \in L(\sigma)$  for each  $\sigma \in \mathcal{S}$ , ie  $w \in \bigcap_{L \in \mathcal{L}} L$ .
- b) We show that the union of two PL-definable languages is PL-definable. The general case can then be handled via induction. Let  $\mathbf{F}$  and  $\mathbf{G}$  be two countable sets of formulae. Let  $\mathbf{H} = \{F \vee G : F \in \mathbf{F}, G \in \mathbf{G}\}$  (note that this set is countable since  $\mathbf{F}$  and  $\mathbf{G}$  are countable). An infinite bitstring  $w$  lies in  $L(\mathbf{H})$  if and only if, for every  $F \in \mathbf{F}$  and  $G \in \mathbf{G}$ ,  $w \models F \vee G$ . Now, if  $w \in L(\mathbf{F})$ , then  $w \models F$  for every  $F \in \mathbf{F}$ , hence,  $w \models F \vee G$  for every  $F \in \mathbf{F}$  and every  $G \in \mathbf{G}$  and so  $w \in L(\mathbf{H})$ . Similarly, if  $w \in L(\mathbf{G})$ , then  $w \in L(\mathbf{H})$ , ie for any  $w \in L(\mathbf{F}) \cup L(\mathbf{G})$ , we have  $w \in L(\mathbf{H})$ . Let us show that these are the only bitstrings in  $L(\mathbf{H})$ . Assume some infinite bitstring  $w$  is such that  $w \notin L(\mathbf{F})$  and  $w \notin L(\mathbf{G})$ . This means that there is some  $F \in \mathbf{F}$  and  $G \in \mathbf{G}$  such that  $w \not\models F$  and  $w \not\models G$ . This means that  $w \not\models F \vee G$ , which is a member of  $\mathbf{H}$ . Hence  $w \notin L(\mathbf{H})$ . This means that for any infinite bitstring  $w$ ,  $w \in L(\mathbf{H})$  if and only if  $w \in L(\mathbf{F}) \cup L(\mathbf{G})$ , ie  $L(\mathbf{H}) = L(\mathbf{F}) \cup L(\mathbf{G})$ , showing that  $L(\mathbf{F}) \cup L(\mathbf{G})$  is PL-definable.



# CS208 Tutorial Solutions

2024

## Contents

Tutorial 1      [2](#)

# Tutorial 1

1. Determine if the following formulae are tautology.

- a)  $(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$
- b)  $(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (r \rightarrow p)$
- c)  $(p_1 \wedge p_2 \dots p_n) \rightarrow (p_1 \vee p_2 \dots p_n)$

**Solution:**

a) Yes. We can show this by using truth table as follows.

$p$	$q$	$r$	$p \rightarrow q$	$q \rightarrow r$	$(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$
true	true	true	true	true	true
true	true	false	true	false	true
true	false	true	false	true	true
true	false	false	false	true	true
false	true	true	true	true	true
false	true	false	true	false	true
false	false	true	true	true	true
false	false	false	true	true	true

b) No. Its not tautology as for p as false and q as true and r as true, we can show that  $(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (r \rightarrow p)$  is false

c) Yes. It is a tautology. However, we cannot verify this by drawing a truth table as it will become excessively cumbersome. Let us assume it is not a tautology, which means there exists an assignment for which  $(p_1 \wedge p_2 \dots p_n)$  evaluates to  $\top$  and  $(p_1 \vee p_2 \dots p_n)$  evaluates to  $\perp$ . Any assignment for which  $(p_1 \wedge p_2 \dots p_n)$  evaluates to  $\top$  must assign  $p_1$  as  $\top$ . If  $p_1$  is  $\top$ , then  $(p_1 \vee p_2 \dots p_n)$  must also evaluate to  $\top$ . Hence, we have a contradiction. Hence, the given formula is a tautology

**Key Takeaway:** Truth Tables act as the first principle for proving certain properties of (a) given formula(e). However, using truth tables for verification becomes cumbersome, and hence, we desire a proof system. A proof system is a set of rules for constructing proofs, which will be covered in the next part of the course.

2. Consider a set  $S$  of size  $n$ . Recall that for a relation over  $S$  to be a partial order we require the following to hold:

- i Reflexive:  $x \preceq x$  for all  $x \in S$
- ii Transitive:  $x \preceq y$  and  $y \preceq z$  implies  $x \preceq z$  for all  $x, y, z \in S$
- iii Antisymmetric:  $x \preceq y$  and  $y \preceq x$  implies  $x = y$  for all  $x, y \in S$ .

Moreover,  $x \in S$  is called a *maximal* element in  $\preceq$  if  $x \preceq y$  holds only for  $y = x$ .

Consider  $n^2$  propositional variables  $\{p_{ij}\}_{1 \leq i, j \leq n}$  for an enumeration of  $S = (x_1, x_2, \dots, x_n)$  where  $p_{ij}$  is set to 1 iff  $x_i \preceq x_j$ .

Give a formula over these variables which evaluates to  $\top$  iff  $\preceq$  is a partial order. Also give a formula which evaluates to  $\top$  only when  $\preceq$  has a maximal element.

**Solution:** For reflexivity the formula is

$$R = p_{11} \wedge p_{22} \dots \wedge p_{nn}.$$

For transitivity, the formula is

$$T = \bigwedge_{1 \leq i, j, k \leq n} p_{ij} \wedge p_{jk} \rightarrow p_{ik} = \bigwedge_{1 \leq i, j, k \leq n} \neg p_{ij} \vee \neg p_{jk} \vee p_{ik}.$$

Observe that we don't really need to consider the cases when any 2 of  $i, j, k$  are the same since if the reflexivity clauses are satisfied:

$$\begin{aligned} (i = j) \quad p_{ii} \wedge p_{ik} \rightarrow p_{ik} &\equiv \top \wedge p_{ik} \rightarrow p_{ik} \equiv \top \\ (j = k) \quad p_{ij} \wedge p_{jj} \rightarrow p_{ij} &\equiv p_{ij} \wedge \top \rightarrow p_{ij} \equiv \top \\ (i = k) \quad p_{ij} \wedge p_{ji} \rightarrow p_{ii} &\equiv p_{ij} \wedge p_{ji} \rightarrow \top \equiv \top \end{aligned}$$

Antisymmetric nature is captured by

$$A = \bigwedge_{1 \leq i, j \leq n, i \neq j} \neg(p_{ij} \wedge p_{ji}) = \bigwedge_{1 \leq i, j \leq n, i \neq j} \neg p_{ij} \vee \neg p_{ji}.$$

The final formula to check for a partial order then is:  $R \wedge T \wedge A$ .

If say  $x_i$  is a maximal element, we have  $\bigwedge_{1 \leq j \leq n, j \neq i} \neg p_{ij}$ .

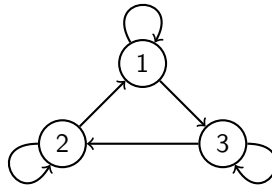
Hence, "has a maximal element" is encoded as

$$M = \bigvee_{1 \leq i \leq n} \left( \bigwedge_{1 \leq j \leq n, j \neq i} \neg p_{ij} \right).$$

Let us look at a concrete example with  $n = 3$ .

$$\begin{aligned} R &= p_{11} \wedge p_{22} \wedge p_{33} \\ T &= (\neg p_{12} \vee \neg p_{23} \vee p_{13}) \\ &\quad \wedge (\neg p_{13} \vee \neg p_{32} \vee p_{12}) \\ &\quad \wedge (\neg p_{21} \vee \neg p_{13} \vee p_{23}) \\ &\quad \wedge (\neg p_{23} \vee \neg p_{31} \vee p_{21}) \\ &\quad \wedge (\neg p_{31} \vee \neg p_{12} \vee p_{32}) \\ &\quad \wedge (\neg p_{32} \vee \neg p_{21} \vee p_{31}) \\ A &= (\neg p_{12} \vee \neg p_{21}) \\ &\quad \wedge (\neg p_{23} \vee \neg p_{32}) \\ &\quad \wedge (\neg p_{13} \vee \neg p_{31}) \\ M &= (\neg p_{12} \wedge \neg p_{13}) \\ &\quad \vee (\neg p_{21} \wedge \neg p_{23}) \\ &\quad \vee (\neg p_{31} \wedge \neg p_{32}) \end{aligned}$$

Consider the following relation which is not a partial order and doesn't have a maximal element.



Here  $(p_{11} = 1, p_{12} = 0, p_{13} = 1, p_{21} = 1, p_{22} = 1, p_{23} = 0, p_{31} = 0, p_{32} = 1, p_{33} = 1)$ . Substituting in values, you can confirm that  $R, A$  evaluate to  $\top$  while  $T, M$  evaluate to  $\perp$ .

3. Let  $n$  and  $k$  be integers so that  $n > 0$ ,  $k \geq 0$  and  $k \leq n$ . You are given  $n$  booleans  $x_1$  through  $x_n$  and your goal is to come up with an efficient propositional encoding of the following constraint:

$$\sum_{i=1}^n x_i \leq k \quad (1)$$

In the above equation, assume the booleans  $x_i$  behaves as 0 when false and 1 when true. We can arrive at the encoding by adding  $O(n \cdot k)$  auxiliary variables and only need  $O(n \cdot k)$  clauses. We try to solve this problem iteratively.

- Let  $s_{i,j}$  denote that at least  $j$  variables among  $x_1, \dots, x_i$  are assigned 1 (true). One can deduce that  $s_{i,j}$  makes sense only when  $i \geq j$ . We now try to represent this constraint in propositional logic.
- Now given that you have represented  $s_{i,j}$  for all appropriate  $1 \leq i \leq n$  and  $0 \leq j \leq k$ ; come up with a propositional formula that represents the condition of the equation (1)

#### Solution:

- We start with simple observations;  $x_i \leftrightarrow s_{i,1} \forall i$  s.t.  $1 \leq i \leq n$  and that  $s_{1,j} = 0 \forall j$  s.t.  $1 < j \leq k$
- We recursively can notice that,  $s_{i-1,1} \rightarrow s_{i,1} \forall i$  s.t.  $1 < n$
- $x_i \wedge s_{i-1,j-1} \rightarrow s_{i,j}$  and  $s_{i-1,j} \rightarrow s_{i,j}$  where  $i, j$  satisfy  $1 < j \leq k$ ,  $1 < i < n$
- Now to represent the final constraint we must make sure the following,  $x_i \rightarrow \neg s_{i-1,k}$  for all  $i$  such that  $1 < i \leq n$
- This could be viewed as a logic circuit similar to an adder circuit taught in your Digital Logic Design course

Note: For the constraint  $\sum_{i=1}^n x_i \geq k$  there is an interesting encoding that uses  $O(kn)$  auxiliary variables and  $O(k^2n)$  clauses using the Pigeon-Hole principle, as follows:  
Let the  $x_i$ 's define *holes* and let  $x_i = 1$  mean that the  $i$ th hole is *fillable*. Our aim is to define  $p_{ij}$  for  $i \in [k], j \in [n]$  such that it is 1 when *pigeon*  $i$  is assigned to hole  $j$ . The constraints we need to capture are:

$$\begin{aligned} \text{[fillable holes]} \quad & \bigwedge_{i \in [k]} \bigwedge_{j \in [n]} (p_{ij} \rightarrow x_j) \\ \text{[only one pigeon per hole]} \quad & \bigwedge_{\substack{i_1, i_2 \in [k] \\ i_1 \neq i_2}} \bigwedge_{j \in [n]} \neg (p_{i_1 j} \wedge p_{i_2 j}) \\ \text{[each pigeon assigned to at least 1 hole]} \quad & \bigwedge_{i \in [k]} \bigvee_{j \in [n]} p_{ij} \end{aligned}$$

Let  $f$  be the conjunction of these clauses, then:

$$\text{Satisfying assignment for } f \iff \text{at least } k \text{ holes fillable} \equiv \sum_{i=1}^n x_i \geq k$$

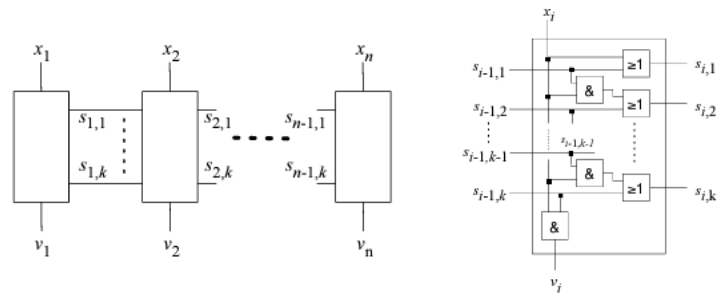


Figure 1: Logic Gate

4. Two friends find themselves trapped in a room. There are three doors coloured - red, blue and green respectively. Behind exactly one of the doors is a path to their home :-). The other two doors lead to horrible places. The inscriptions on the three doors are as follows:  
 Red door: "Your home town is not behind blue door"  
 Blue door: "Your home town is behind red door"  
 Green door: "Your home town is not behind blue door"  
 Given the fact that at LEAST ONE of the inscriptions is true and at LEAST ONE of them is false, which door would lead the boys home?

**Solution:** Let us use the following three propositional variables

- $r$ : home town is behind red door
- $b$ : home town is behind blue door
- $g$ : home town is behind green door

Let us encode whatever we know

- behind one of the doors is a path to home, behind the other two doors is a way to ocean:

$$(r \wedge \neg b \wedge \neg g) \vee (\neg r \wedge b \wedge \neg g) \vee (\neg r \wedge \neg b \wedge g)$$

- at least one of the three statements is true:

$$r \vee \neg b$$

- at least one of the three statements is false:

$$\neg r \vee b$$

$r$	$b$	$g$	2.5	2.6	$2.5 \wedge 2.6$
T	F	F	T	F	F
F	T	F	F	T	F
F	F	T	T	T	T

Thus, the friends should rush to the green door to get back to their home !!