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

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

	D_1	D_2	D_3	D_4	D_5
S_1					
S_2					
S_3					
S_4					

- 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 $S_1, \ldots, S_4$.
- For every day D_i of the week, have at least one slot free.

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 $p, q, r, x, y, z, \ldots$ etc. Here is a short description of propositional logic symbols:

- **Variables**: They are usually denoted by smalls ($p, q, r, x, y, z, \ldots$ 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 $\bot$. 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	$\bot$	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 $[n] = \{1, \ldots, n\}$) 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 $\bot$ are formulae.
- If φ is a formula, so are $\neg\varphi$ and (φ).
- If φ_1 and φ_2 are formulas, so are $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$.

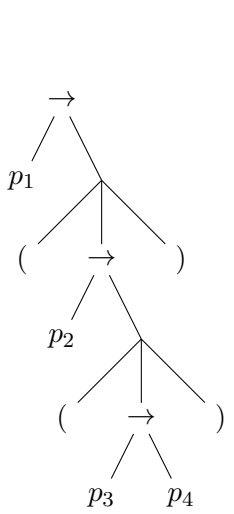
Propositional formulae as strings and trees:

Formulae can be expressed as a strings over the alphabet $\mathbf{Vars} \cup \{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$. **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.

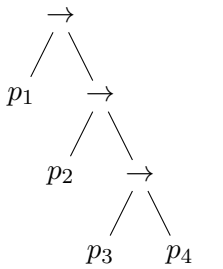
1.2 Semantics

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 φ of n variables is a function

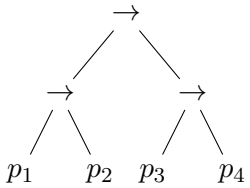
$$[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$$



(a)



(b) Parse tree for $p_1 \to (p_2 \to (p_3 \to p_4))$



(c) Parse tree for $(p_1 \to p_2) \to (p_3 \to p_4)$

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.

φ	$\neg\varphi$
0	1
1	0

(a) Truth table for $\neg\varphi$.

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table for $\varphi_1 \wedge \varphi_2$.

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
0	0	0
0	1	1
1	0	1
1	1	1

(c) Truth table for $\varphi_1 \vee \varphi_2$.

φ_1	φ_2	$\varphi_1 \to \varphi_2$
0	0	1
0	1	1
1	0	0
1	1	1

(d) Truth table for $\varphi_1 \to \varphi_2$.

φ_1	φ_2	$\varphi_1 \leftrightarrow \varphi_2$
0	0	1
0	1	0
1	0	0
1	1	1

(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 $\bot$: 0 (false) and 1 (true) are meanings, while $\top$ and $\bot$ are symbols.

Rules of semantics:

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

- $[\![\varphi_1 \to \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!] = 0$ or $[\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \leftrightarrow \varphi_2]\!] = 1$ iff at both $[\![\varphi_1 \to \varphi_2]\!] = 1$ and $[\![\varphi_2 \to \varphi_1]\!] = 1$.

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 $[\![(p \vee s) \to (\neg q \leftrightarrow r)]\!]$ (see Table 1.3).

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \to (\neg q \leftrightarrow r)$
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 $(p \vee s) \to (\neg q \leftrightarrow r)$.

1.2.1 Important Terminology

A formula φ is said to (be)

- **satisfiable** or **consistent** or SAT iff $[\![\varphi]\!] = 1$ 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 (φ and $\neg\varphi$ may both be SAT).
- **unsatisfiable** or **contradiction** or UNSAT iff $[\![\varphi]\!] = 0$ 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 φ is UNSAT then $\neg\varphi$ must be SAT (it is in fact valid).
- **valid** or **tautology**: $[\![\varphi]\!] = 1$ for all assignments of variables. That is, the formula is always true, no matter how the variables are assigned. If a formula φ is valid then $\neg\varphi$ is UNSAT.
- **semantically entail** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ 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 φ evaluates to 1 then φ_1 will evaluate to 1. Equivalently $\varphi \to \varphi_1$ is valid.

- **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$. Basically φ and φ_1 have identical truth tables. Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid.
- **equisatisfiable** to φ_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 φ_1, φ_2 and φ_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 $\neg\big(\sum_{k=1}^{4}\sum_{j=1}^{5} p_{ijk} \leq 2\big)$.

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

$$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$$

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 φ_1 and φ_2 are true if $\varphi_1 \wedge \varphi_2$ is true, so φ_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 $\wedge_{e_1}$ as φ_1 is kept in the inference even when both φ_1 and φ_2 could be kept in inference. If we use φ_2 in inference then the rule becomes $\wedge_{e_2}$.

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

Connector	**Introduction**	**Elimination**
$\wedge$	$\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$ $\quad$ $\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e_2}$
$\vee$	$\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i_1}$ $\quad$ $\dfrac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i_2}$	$\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \to \varphi_3 \quad \varphi_2 \to \varphi_3}{\varphi_3} \vee_e$
$\to$	$\dfrac{\boxed{\begin{array}{c}\varphi_1 \\ \vdots \\ \varphi_2\end{array}}}{\varphi_1 \to \varphi_2} \to_i$	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2} \to_e$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\varphi \\ \vdots \\ \bot\end{array}}}{\neg\varphi} \neg_i$	$\dfrac{\varphi \quad \neg\varphi}{\bot} \neg_e$
$\bot$		$\dfrac{\bot}{\varphi} \bot_e$
$\neg\neg$		$\dfrac{\neg\neg\varphi}{\varphi} \neg\neg_e$

Table 1.5: Proof rules.

In the $\to_i$ rule, the box indicates that we can *temporarily* assume φ_1 and conclude φ_2 using no extra non-trivial information. The $\to_e$ 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 $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$.

$$\frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3} \wedge_{e_2} \qquad \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1} \wedge_{e_1}$$

$$\frac{\varphi_2 \wedge \varphi_3}{\varphi_2} \wedge_{e_1} \qquad \frac{\varphi_2 \wedge \varphi_3}{\varphi_3} \wedge_{e_2}$$

$$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$$

$$\frac{\varphi_1 \wedge \varphi_2 \quad \varphi_3}{(\varphi_1 \wedge \varphi_2) \wedge \varphi_3} \wedge_i$$

1.4 Natural Deduction

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

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

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 φ at a point if it occurs prior to it in the proof and if **no box enclosing that occurrence of φ 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.	$\bot$	$\neg_e$ 3,1
5.	$\neg p$	$\neg_i$ 2–4
6.	$p \vee \neg p$	$\vee_{i_2}$ 5
7.	$\bot$	$\neg_e$ 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.	$\bot$	$\neg_e$ 1, 2
4.	$\neg\neg p$	$\neg_i$ 2–3

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

1.	$p \to q$	premise
2.	$\neg q$	premise
3.	p	assumption
4.	q	$\to_e$ 3,1
5.	$\bot$	$\neg_e$ 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.	$\neg p$	$\wedge_{e_1}$ 1
5.	$\bot$	$\neg_e$ 3,4
6.	$p \to \bot$	$\to_i$ 3–5
7.	q	assumption
8.	$\neg q$	$\wedge_{e_2}$ 1
9.	$\bot$	$\neg_e$ 7,8
10.	$q \to \bot$	$\to_i$ 7–9
11.	$\bot$	$\vee_e$ 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. $\Sigma = \{p \to q, \neg q\} \models \neg p$.

p	q	$p \to q$	$\neg q$	$\neg p$
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 $p \to q$ 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 Σ imply φ[1]. And to prove an implication, we can use the $\to_i$ 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.	$(p \to q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4

Similarly to mimic the second row, we would like to show $\neg p, q \vdash ((p \to q) \wedge \neg q) \to \neg p$. 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
2.	$\neg q$	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4

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

1.	p	given
2.	$\neg q$	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$p \to q$	$\wedge_{e_1}$ 3
5.	q	$\to_e$ 1,4
6.	$\bot$	$\neg_e$ 2,5
7.	$\neg p$	$\bot_e$ 6
8.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7

1.	p	given
2.	q	given
3.	$(p \to q) \wedge \neg q$	assumption
4.	$p \to q$	$\wedge_{e_1}$ 3
5.	$\neg q$	$\wedge_{e_2}$ 3
6.	q	$\to_e$ 1,4
7.	$\bot$	$\neg_e$ 2,6
8.	$\neg p$	$\bot_e$ 7
9.	$((p \to q) \wedge \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7,8

Figure 1.2: Mimicking all 4 rows of the truth table

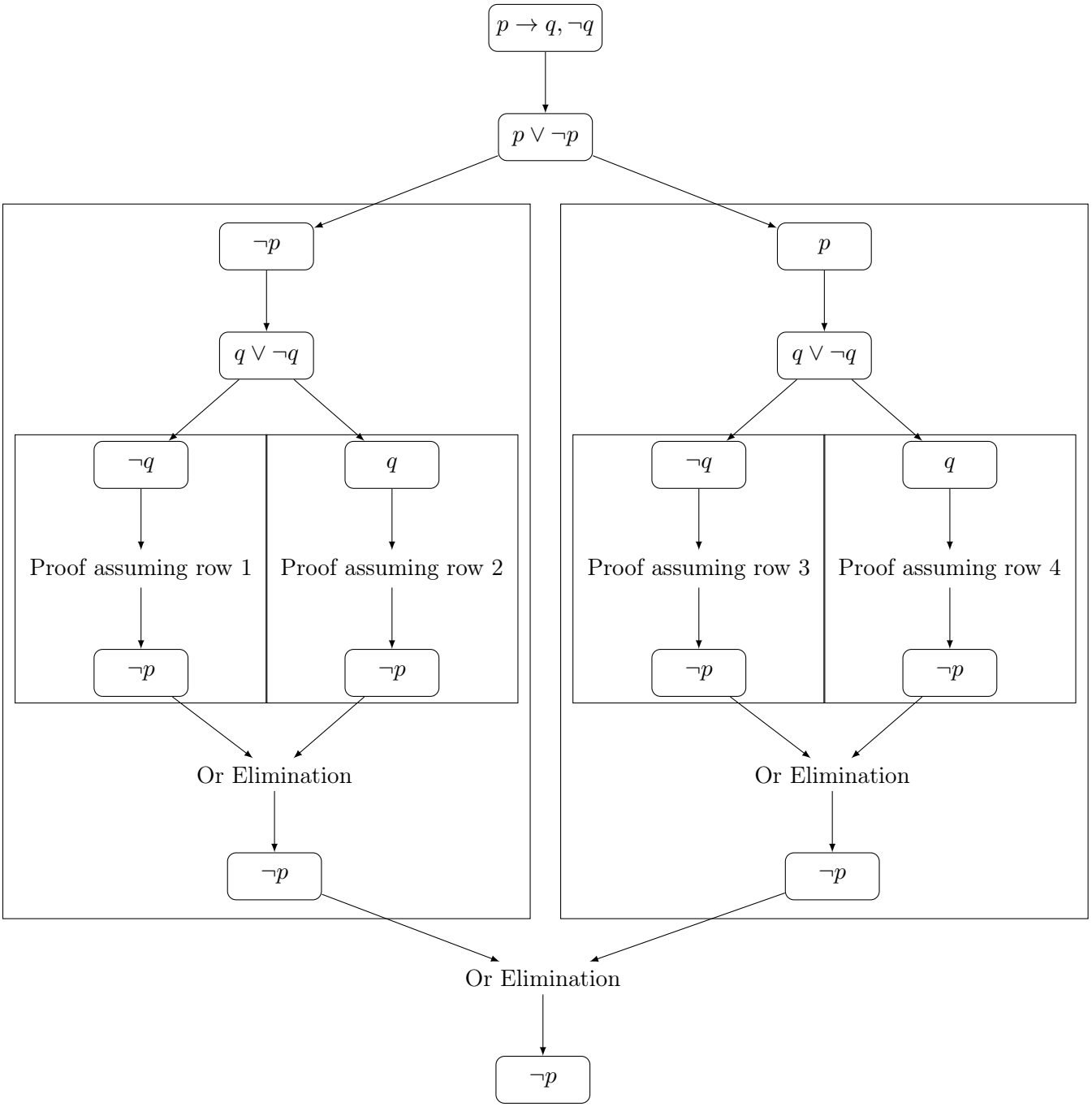
This looks promising, but we aren't done, we have only proven our formula under all possible assumptions, but we haven't exactly proven our formula from nothing given. But note that the reasoning we are doing looks a lot like case work, and we can think of the $\vee_e$ rule. In words, this rule states 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 $\bigwedge_{\phi \in \Sigma} \phi \to \varphi$.

This can be done using **structural induction** to prove the following:
Let φ be a formula using the propositional variables $p_1, p_2, \ldots, p_n$. 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:

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

[1] Σ semantically entails φ is equivalent to saying intersection of formulae in Σ implies φ 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,

$$\nvdash p \wedge q$$

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

$$\vdash \neg p \rightarrow \neg(p \wedge q)$$

$$\vdash (p \rightarrow (q \rightarrow (p \wedge q))$$

An important link between the two situations is that,

A formula ϕ 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.

$$\phi_1 \wedge (\phi_2 \vee \phi_3) \models\!\!\models (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \models\!\!\models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$$

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.

$$\phi_1 \longleftrightarrow \phi_2 \models\!\!\models (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$$

$$(\phi_1 \rightarrow \phi_2) \models\!\!\models ((\neg\phi_1) \vee \phi_2)$$

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(discuissed in the next section).

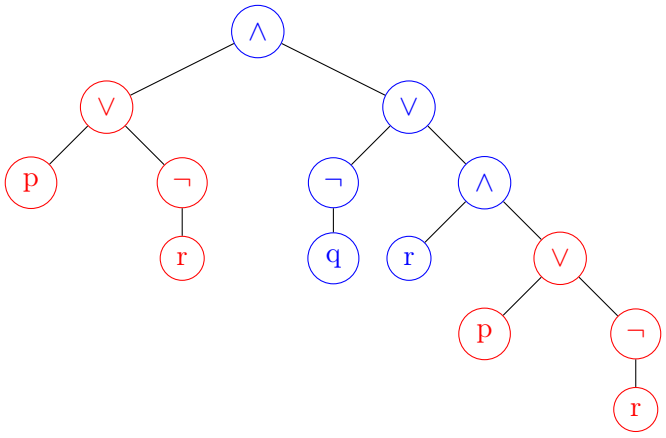
$$\neg(\phi_1 \wedge \phi_2) \models\!\!\models (\neg\phi_1 \vee \neg\phi_2)$$

$$\neg(\phi_1 \vee \phi_2) \models\!\!\models (\neg\phi_1 \wedge \neg\phi_2)$$

1.8 Negation Normal Forms

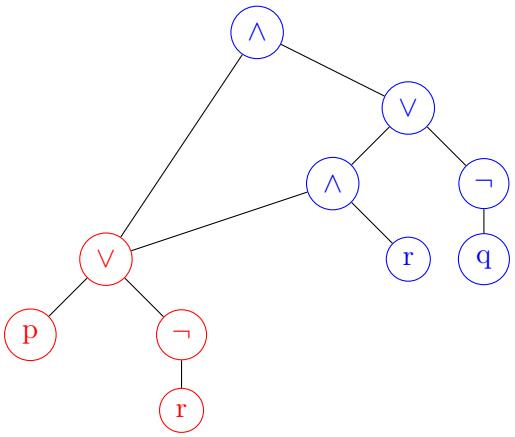
In mathematical logic, a formula is in negation normal form (NNF) if the negation operator ($\neg$) is only applied to variables and the only other allowed Boolean operators are conjunction ($\wedge$) and disjunction($\vee$) One can convert any formula to a NNF by repeatedly applying DeMorgan's Laws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

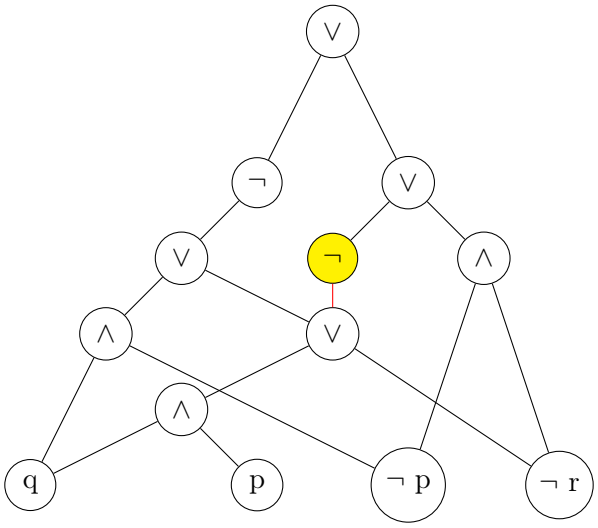
DAG Representation



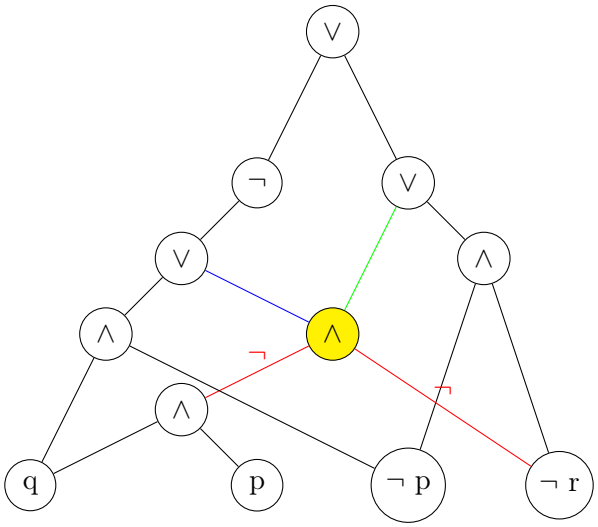
1.9 From DAG to NNF-DAG

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 ¬ 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 , ¬ pushing) only 1 of the copies and if a nodes wants non - negated node then take that node from the copied tree.

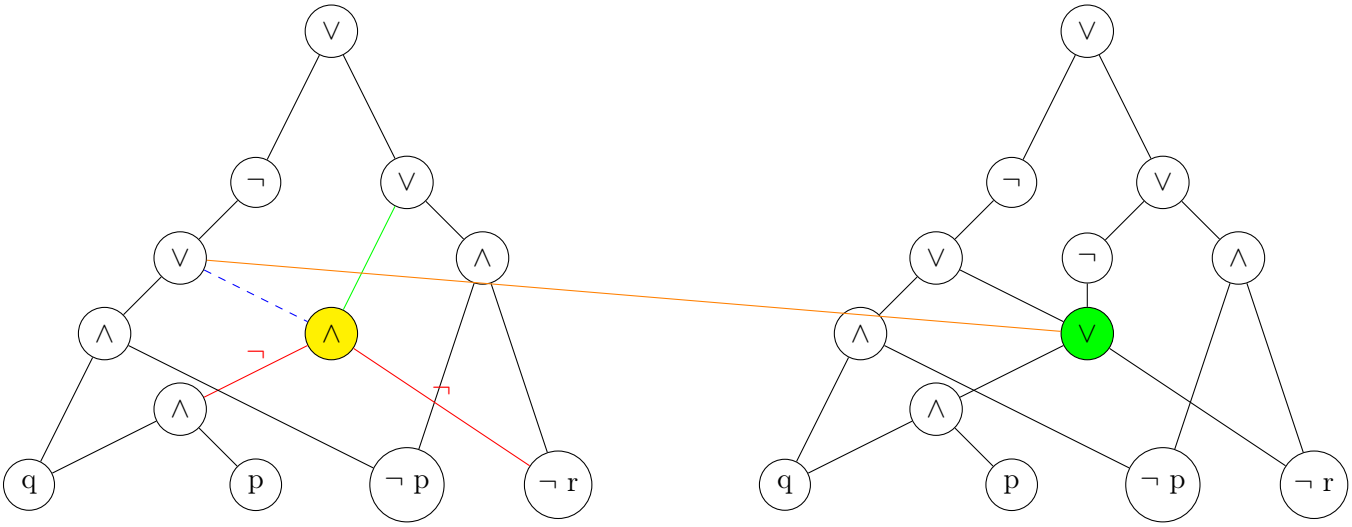


Figure 1.3: Step 1

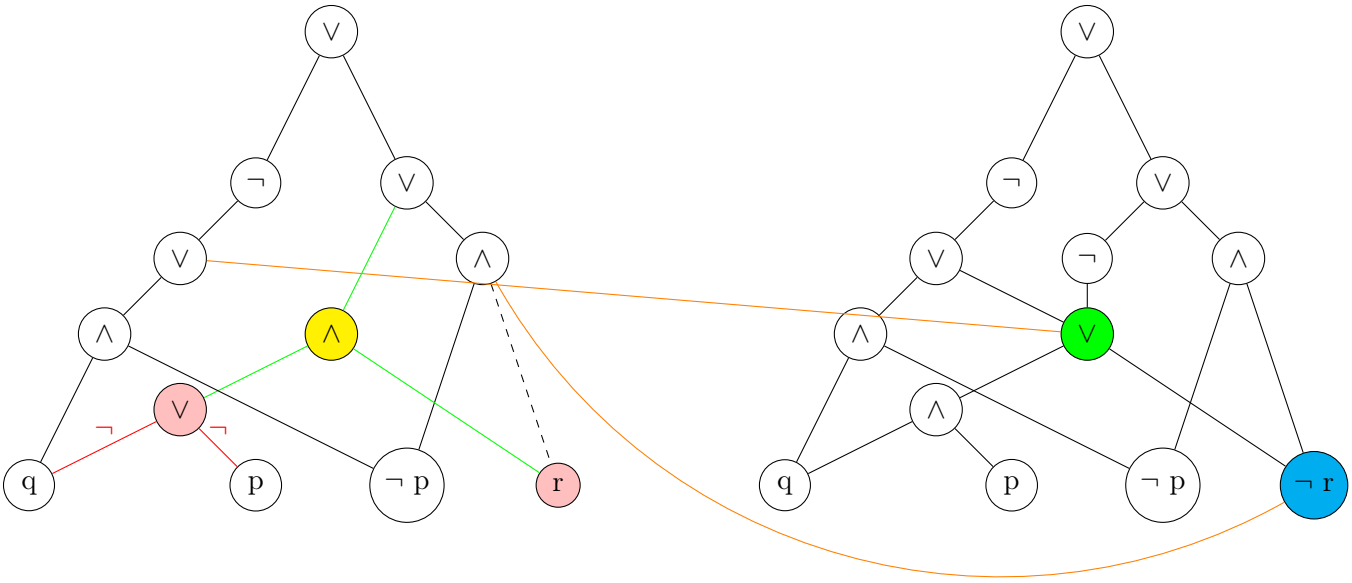


Figure 1.4: Step 2

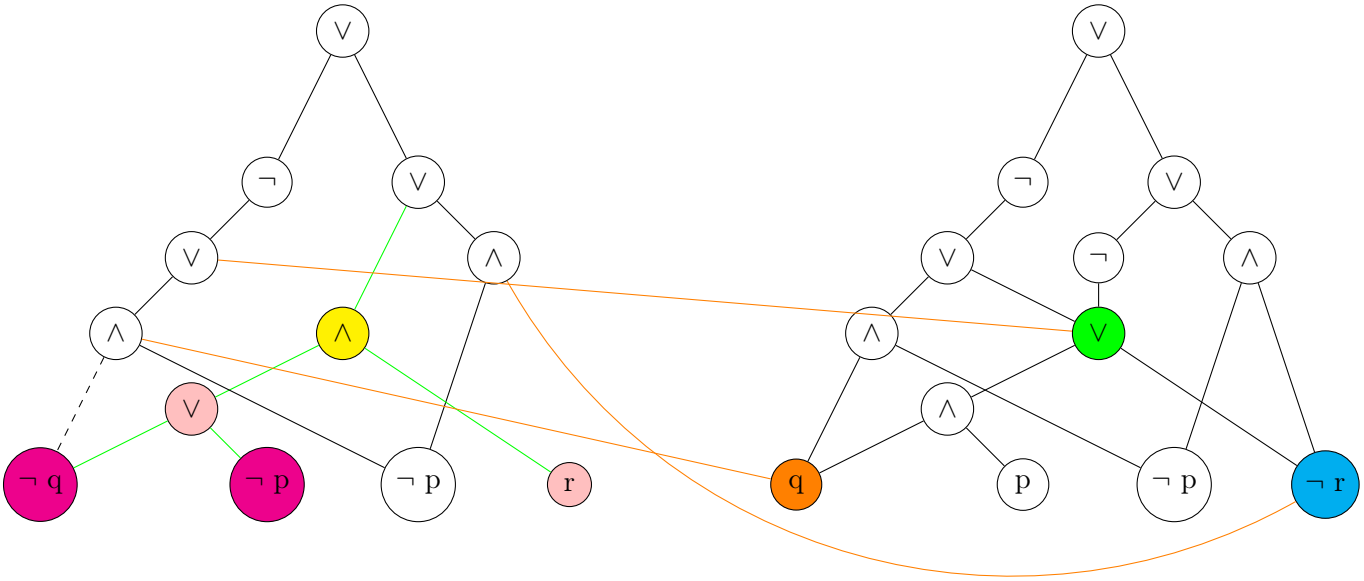


Figure 1.5: Step 3

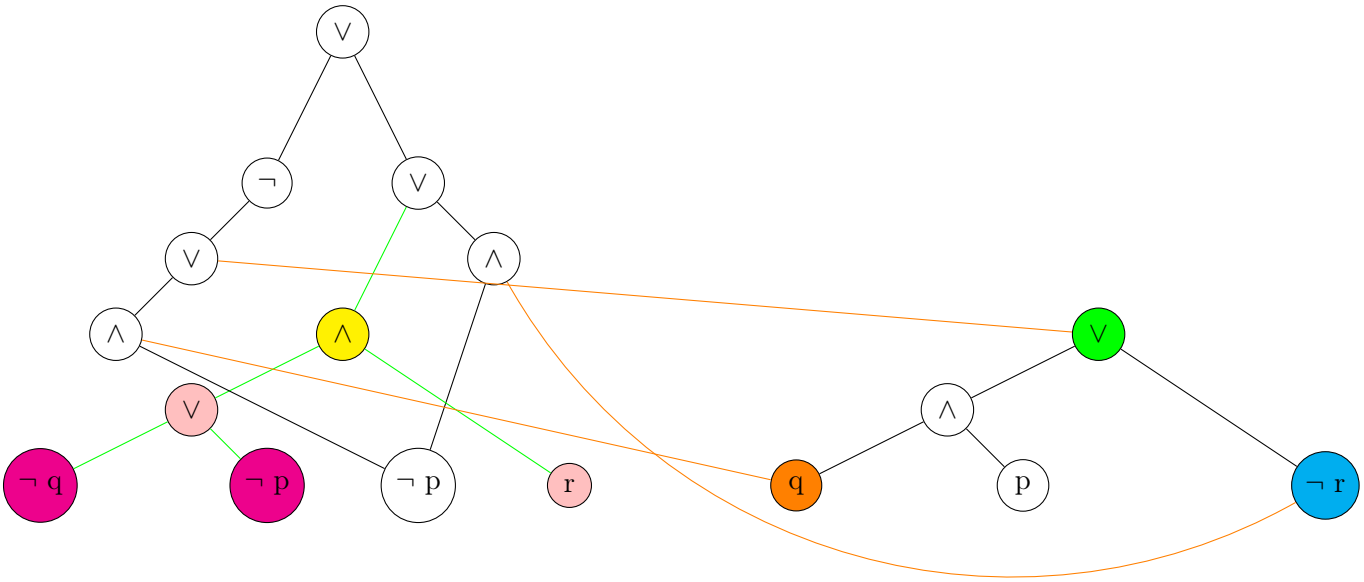


Figure 1.6: Step 4 : Remove all redundant nodes

1.10 An Efficient Algorithm to convert DAG to NNF-DAG

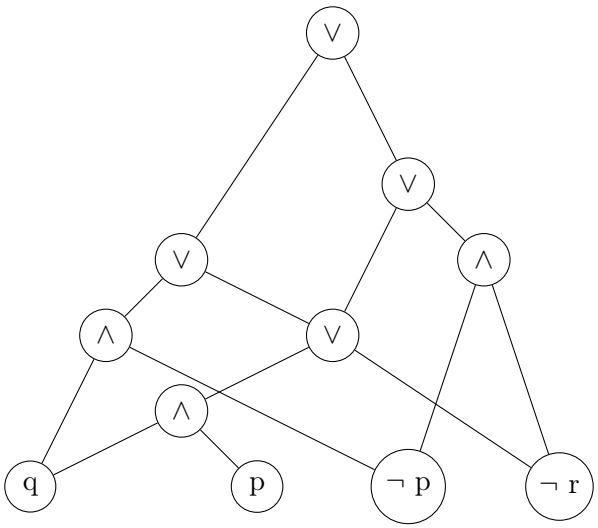


Figure 1.7: Step 1:Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

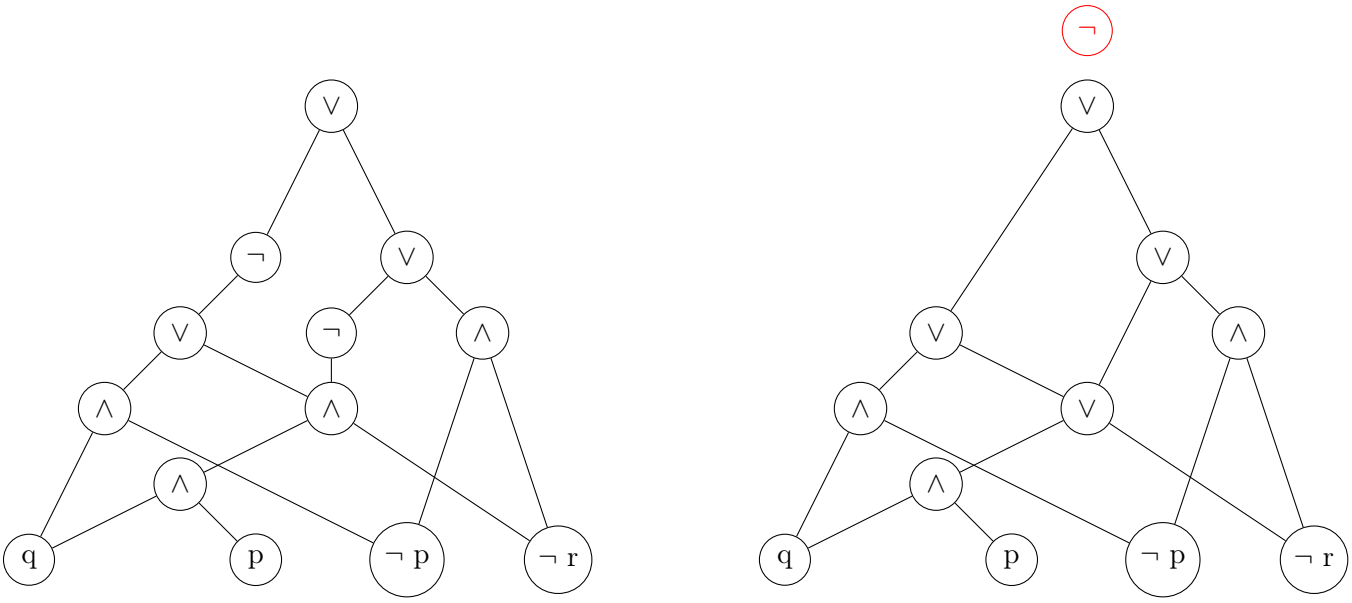


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

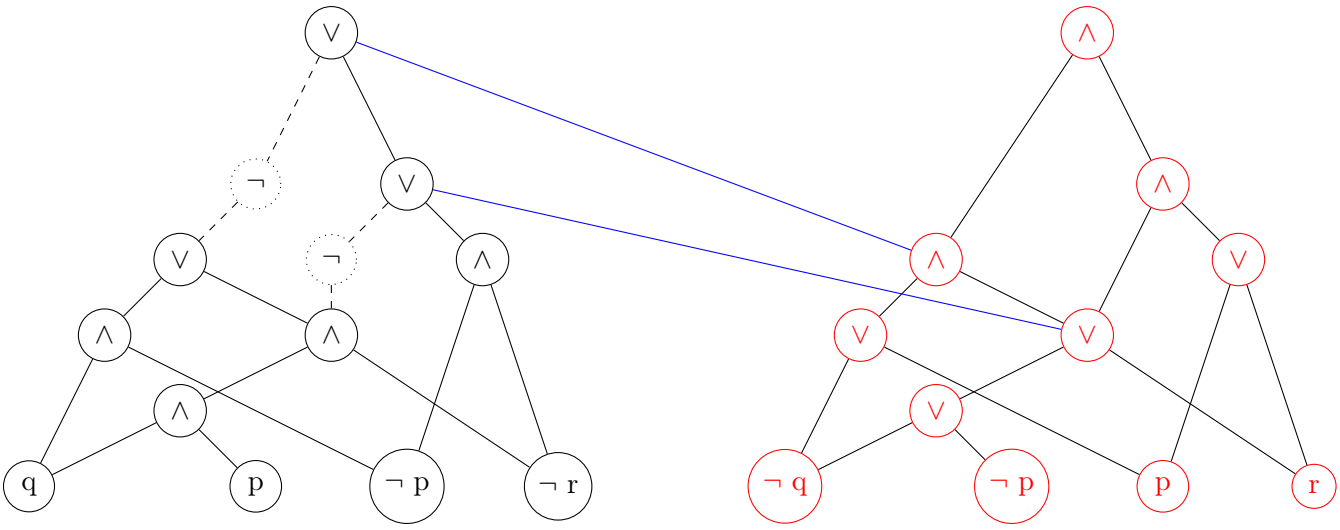


Figure 1.9: Step 3: Remove the "¬" nodes from the first DAG by connecting them to the corresponding node in the negated DAG.

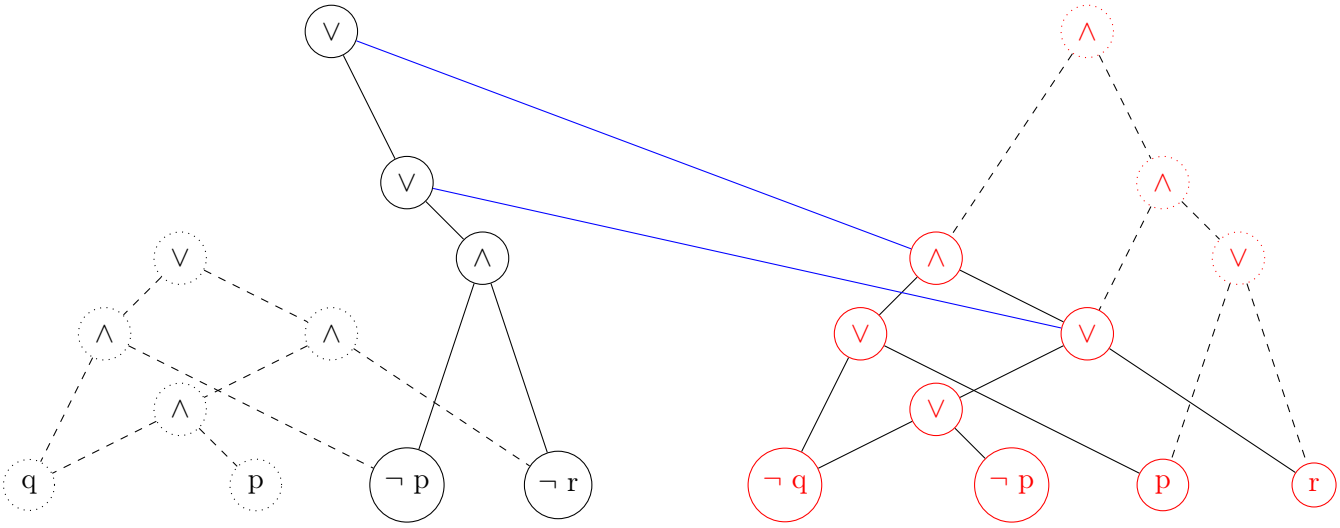


Figure 1.10: Step 4: Remove all the redundant nodes.

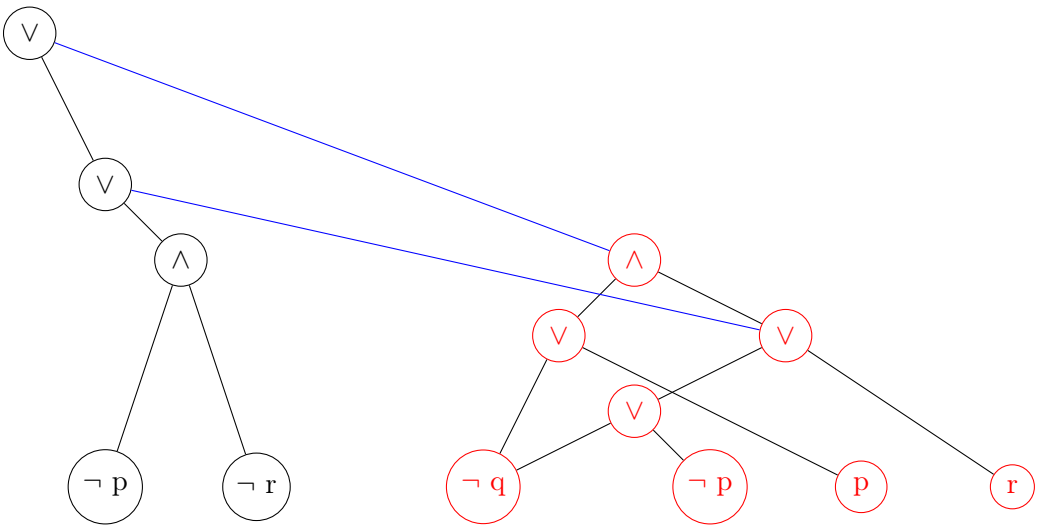


Figure 1.11: NNF - DAG

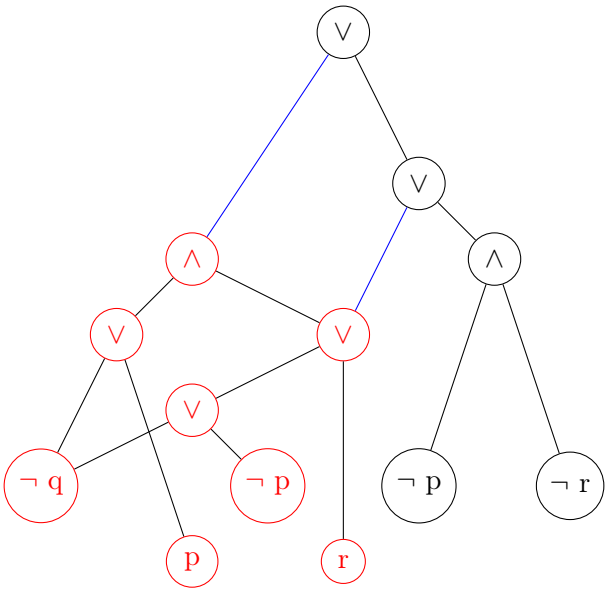


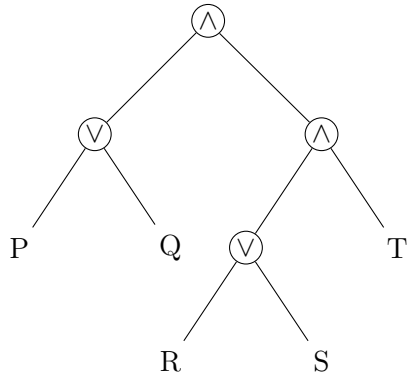
Figure 1.12: The NNF DAG (rearranged)

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.

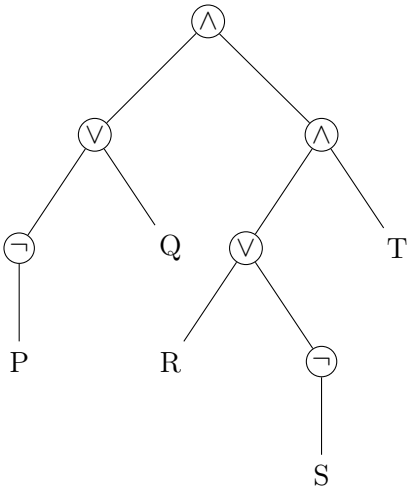
1.11 Conjunctive Normal Forms

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



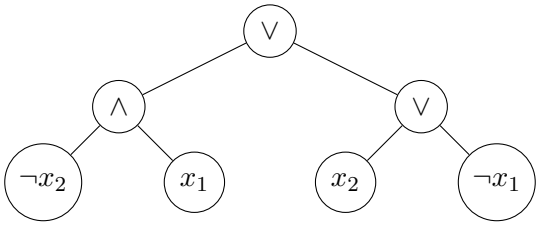
Some Important Terms

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

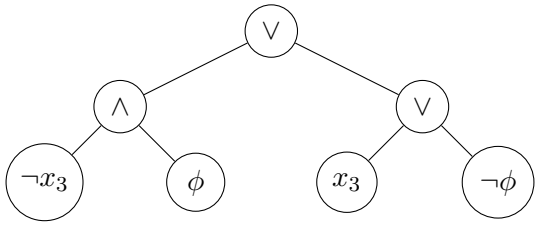
- **CLAUSE :**
  - A clause is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal
  - Example : p $\vee$ q $\vee$ ($\neg$ r) , p $\vee$ ($\neg$ p) $\vee$ ($\neg$ r) -> Not Allowed
- **CUBE :**
  - a Cube is a conjunction of literals such that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

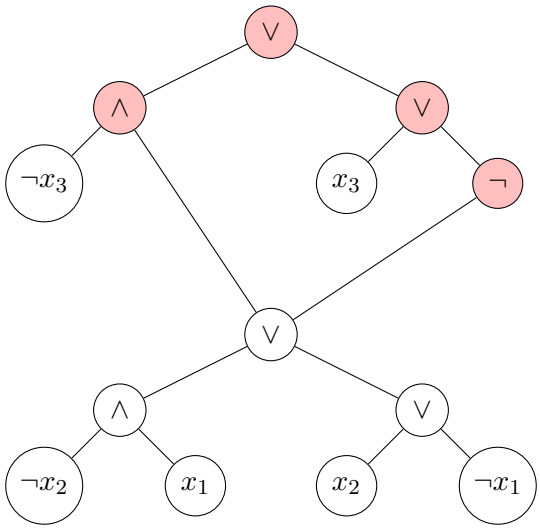
Tutorial 2 Question 2:
The Parity Function can be expressed as (((....($x_1 \oplus x_2$) $\oplus x_3$) ..... $\oplus x_n$)
The Parse Tree for $x_1 \oplus x_2$ is



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 -> 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 , ¬ p. If **there is a clause which does not have both(p , ¬ 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 ∨ q ∨ (¬ 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 ∨ ¬ p is valid and $1 \vee \phi = 1$ for any ϕ.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 , ¬ p. If there is a cube which does not have both(p , ¬ 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 simultaneosly

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, $....,t_n$ for each of the nodes. We will get an equisatisfiable formula $\phi'(p, q, r, t_1, t_2, t_3, ...,t_n)$ which is in CNF.
2. Write the formula for ϕ' 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:

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

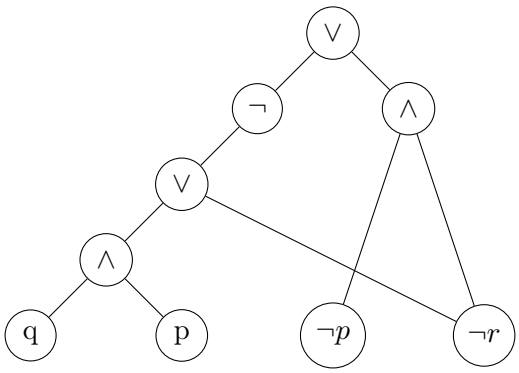
4. For checking the equisatisfiabiliy of ϕ and ϕ': Think about an assignment which makes ϕ true then apply that assignment to ϕ'.

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 sematically 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 conjuction 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.

$$x_1 \wedge x_2 \implies x_3$$
$$x_4 \wedge x_3 \implies x_5$$
$$x_1 \wedge x_5 \implies \bot$$
$$x_5 \implies x_3$$
$$\top \implies x_5$$

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(φ):
2      foreach occurrence of ⊤ in φ do
3          mark the occurrence
4      while there is a conjunct P_1 ∧ P_2 ∧ ... ∧ P_{k_i} → P' in φ do
           // such that all P_j are marked but P' isn't
5          if all P_j are marked and P' isn't then
6              mark P'
7      if ⊥ is marked then
8          return'unsatisfiable'
9      else
10         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

$$b \wedge n \Rightarrow a \quad f \wedge d \Rightarrow a \quad e \wedge d \Rightarrow a$$
$$e \wedge n \Rightarrow a \quad p \wedge n \Rightarrow a \quad d \wedge n \Rightarrow \bot$$
$$b \wedge f \Rightarrow a \quad p \wedge d \Rightarrow \bot$$

Now we want to examine the possible behaviour of this systen 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:

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

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

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 aditional 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. $.. \wedge (\neg x_5) \wedge ..$
  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(φ, PA)

{// These are the base cases for our recursion}
if $\varphi = \top$ **then**
  return SAT(sat, PA)
else if $\varphi = \bot$ **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**}
  $x \leftarrow$ choose_a_var(φ)
  $v \leftarrow$ 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-

$$C_1 : (\neg P_1 \vee P_2) \quad C_2 : (\neg P_1 \vee P_3 \vee P_5) \quad C_3 : (\neg P_2 \vee P_4) \quad C_4 : (\neg P_3 \vee P_4)$$
$$C_5 : (P_1 \vee P_5 \vee \neg P_2) \quad C_6 : (P_2 \vee P_3) \quad C_7 : (P_1 \vee P_3 \vee P_7) \quad C_8 : (P_6 \vee \neg P_5)$$

Lets make two possible decision trees for these clauses.
PLE - Pure literal elimination UP - Unit Propagation D - Decision

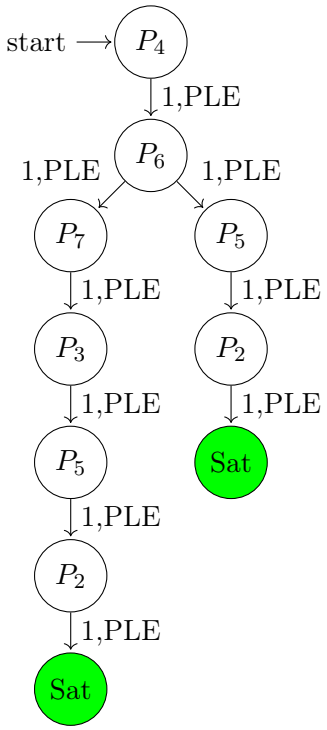


Figure 1.13: DPLL

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

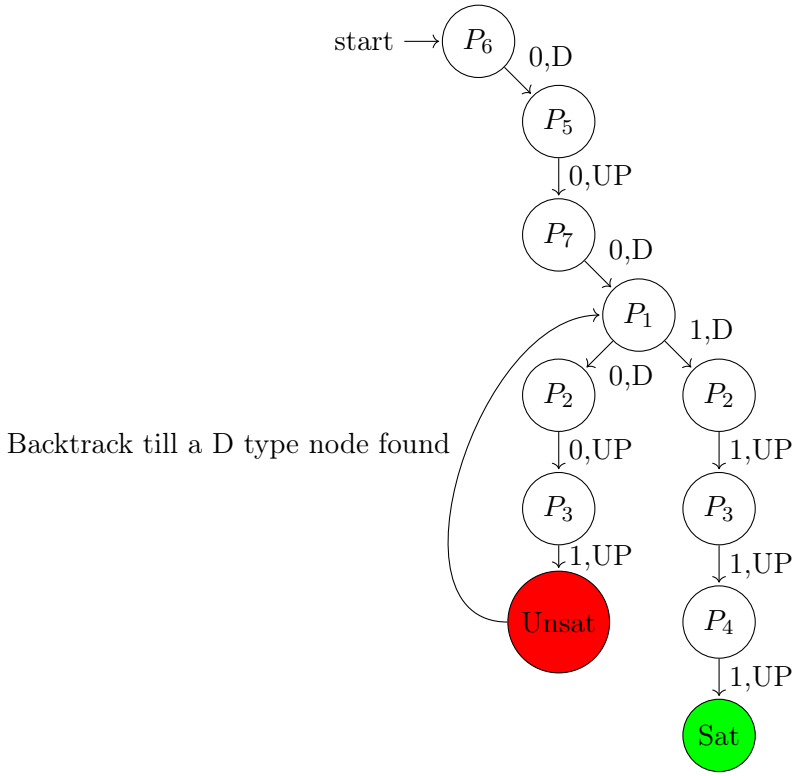


Figure 1.14: DPLL without point 2

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.* $\top \to x_i$ 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.

1.20 DPLL on Horn Clauses

We shall quickly investigate what happens when we feed in **Horn clauses** to the **DPLL** (Davis-Putman-Logemann-Loveland) algorithm.
Consider the following,

- Consider the first step in solving for the satisfiability of a given set of Horn clauses in implication form where,
  - If the LHS of an implication is true we set the literal on the RHS of the implication to be true in all its implications.
  - Repeating the above step sets all essential variable which are to be set to 1, true.

  This step is equivalent to the first two steps of the DPLL algorithm,
  - Satisfy unit literal clauses by assignment.
  - Recomputing the formula for the above bullet.
  - Repeating this procedure until no unit clause is left.

  The above steps in the two different schemes do the same are essentially doing the same thing. Now if the given clauses were Horn, we know that putting all the remaining variables false is a satisfying assignment. This means if our DPLL algorithm preferentially assigns 0 to each decision, the procedure thus converges to the method for checking the satisfiability for Horn formulae.

1.21 Rule of Resolution

This is yet another powerful rule for inference. Let us first jot down the rule here:

$$\frac{(a_1 \vee a_2 \vee a_3 \cdots \vee a_n \vee x) \wedge (b_1 \vee b_2 \vee b_3 \cdots \vee b_m \vee \neg x)}{(a_1 \vee a_2 \vee a_3 \cdots \vee a_n \vee b_1 \vee b_2 \vee b_3 \cdots \vee b_m)} \text{ resolution}$$

However intuitive it may look this rule poses as a powerful tool to check the satisfiability of logical formulae, we can reason out an algorithm to check the satisfiability of a formula (CNF) as follows: Let us first define a formula to be **unresolved** if there exists a literal and its negation in the formula (they cannot be in the same clause by the definition of a clause). If a formula is **resolved** (i.e., not unresolved) then it is satisfiable ('**SAT**'), as the variables which appear in their negated form we assign false, and the other variables to true.

Let $\mathfrak{C}$ be the set of clauses for a give CNF.

1. If $\mathfrak{C}$ contains tautologies we can drop them, if $\mathfrak{C}$ becomes empty upon dropping the tautologies, we mark the given CNF **SAT**. (However by definition, clauses by themselves cannot be tautologies.)
2. As the formula is unresolved, we can apply the resolution rule, this gives us a new clause.
3. If the formula so formed is the empty clause, we deem the formula to be **UNSAT** otherwise check if the formula is resolved, if not from repeat step 1.

Before rationalizing the soundness of the above sequence of steps let us first see an example.
An Example: Consider $\mathfrak{C} = \{C_1, C_2, C_3\}$ as given below:

- $C_1 :=\ \neg p_1 \vee p_2\ (\ p_1 \implies p_2)$
- $C_2 :=\ p_1 \vee \neg p_2\ (\ p_2 \implies p_1)$
- $C_3 :=\ \neg p_1 \vee \neg p_2\ (\ p_1 \wedge p_2 \implies \bot)$

Then, a dry run of the above method would look like:

1. Since both p_1 and p_2 appear in negated and un-negated form, we apply resolution on C_1 and C_2, which generates C_4, as follows:

$$\frac{(\neg p_1 \vee p_2)\ \ (\ p_1 \vee \neg p_2)}{(\neg p_1 \vee p_1)}\ \text{resolution}$$

2. Once again we apply resolution on C_3 and C_4 (this is not really a clause by definition, one can choose to drop the tautologies as soon as encountered),

$$\frac{(\neg p_1 \vee \neg p_2)\ \ (\neg p_1 \vee p_1)}{(\neg p_2 \vee \neg p_1)}\ \text{resolution}$$

3. The clause that we have got is now resolved and thus, our formula is satisfiable.

1.21.1 Completeness of Resolution for Unsatisfiability of CNFs

Here as we claimed above, given any CNF, if it is unsatisfiable the remainder of continuous resolutions is the empty clause which we deem **UNSAT**. We here prove the consistency of the claim.
For this we employ the method of mathematical induction, we induct on the number of propositional variables in our CNF. Let $p_1, p_2 \dots p_m$ be our propositional variables.
Base: $n = 1$ An unsatisfiable CNF in a single literal **must** contain the clauses (p_1) and $(\neg p_1)$ which upon resolution gives us **( )** the empty clause hence we raise **UNSAT**.
Inductive Hypothesis: Assume that our claim holds $\forall m \leq n-1$ we now show that it holds from $m = n$ as follows,

- Remove tautologies from $\mathfrak{C}$, the set of all clauses.
- Choose a literal p_i such that both p_i and $\neg p_i$ both appear in the CNF. (If no such literal exists the formula is resolved as defined earlier and has a satisfying assignment). Apply resolution repeatedly as long as the same p_i satisfies this condition.

- If the CNF contains **( )**, in which case we raise **UNSAT**, otherwise
  - If p_i **vanishes** from the CNF, then calling our hypothesis, we can raise **UNSAT** as the equivalent form that we have got must be unsatisfiable independent of the value of the vanished literal as the initial formula was unsatisfiable.
  - If p_i **exists** in one of negated or un-negated forms. In which case we repeat the procedure. This time the number of available **pairs** has reduced by 1 as p_i cannot be selected again.
- As the selection step can take place at most n times, (as a new **pair**(as in bullet 2) cannot be generated in the CNF by resolution operations), Consider the case with a **pair** available for every literal then the procedure must conclude **UNSAT** in n steps otherwise at the end of n steps we have no **pair**s, which ensures a satisfying assignment for the formula.

Broadly speaking, what we are showing is that upon repeated resolution of an unsatisfiable CNF, if **( )** has not been encountered, the number of propositional variables must decrease.

Chapter 2

DFAs and Regular Languages

Consider a set of formulas made up of propositional variables $\{x_1, x_2 \ldots, x_n\}$, $\phi(x_1, x_2 \ldots, x_n)$. We defined the set $L \subseteq \{0,1\}^n$, the **language** defined by the formula as the set of strings which form a **satisfying assignment** for the formula ϕ.
Basically using **Propositional Logic**, we were able to represent a large set of **finite length** strings having some properties in a **compact form**. This leads us to a question what about string of arbitrary length having some properties. How do we formulate them?
The answer to this is **Automata**: A way to formulate arbitrary length strings in a compact form.

2.1 Definitions

- **Alphabet**: A **finite, non-empty** set of symbols called **characters**. We usually represent an alphabet with Σ. For example $\Sigma = \{a, b, c, d\}$.
- **String**: A **finite** sequence of letters form an alphabet. An important thing to note here is that even though the alphabet may contain just 1 character, it can form **countably infinite** number of strings, each of which are **finite**. In this course we only deal with **finite strings** over a **finite alphabet**.
- **Concatenation Operation ($\cdot$)** We can start from a string , take another string an as the name suggests concatenate them to form another string:

$$\begin{aligned} a \cdot b &\neq b \cdot a && \text{Not Commutative} \\ (a \cdot b) \cdot c &= a \cdot (b \cdot c) && \text{Associative} \end{aligned}$$

- **Identity Element** The algebra of the strings defined over the concatenation operator has the identity element : ε **: empty string**:

$$\sigma \cdot \varepsilon = \varepsilon \cdot \sigma = \sigma$$

  Note the the empty strings remains same for strings of all alphabets.

- **Language** A subset of all finite strings on Σ. This set doesn't have to be finite even though the strings are of finite length.
  Note that a set of all finite strings of Σ is countably infinite (cardinality: $\mathbb{N}$), so the number of languages of Σ is uncountably infinite (cardinality: $2^{\mathbb{N}}$).

- Σ^* is defined to be the set of all finite strings on Σ, including ε. Note that $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$, where Σ^k is the set of all strings on Σ with exactly k letters. Note that we can prove that the number of strings for Σ^* are countably finite by representing each string as a unique number in base $(n+1)$ system , where $|\Sigma| = n$, we can get an injection to natural numbers.

2.2 Deterministic Finite Automata

Generalization of Parity function: Lets go back to the question we asked first. Suppose we are given a string of arbitrary length and don't know the length of the string. This can be done by propositional logic. So we need a new formalism to represent sets of strings with any length. we want to develop a mechanism where we are given the bits of the string one by one, and I don't know when it will stop. So I must be ready with the answer each time a new bit arrives.

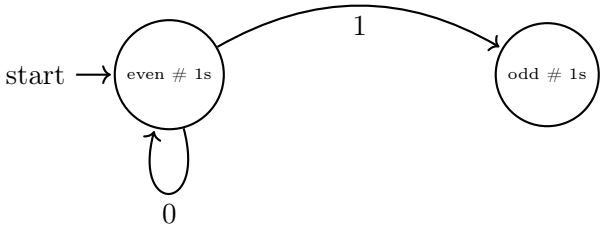
The solution to this lies in our discussion during the first lecture. I will record just one bit of information: whether I have received an even or odd number of 1s till now. Every time I receive a new bit, I will update this information: if it's a 0, I won't do anything, and if it's a 1, I will change my answer from even to odd, or vice-versa.

States: These are nodes which contain relevant summary of what we have seen so far

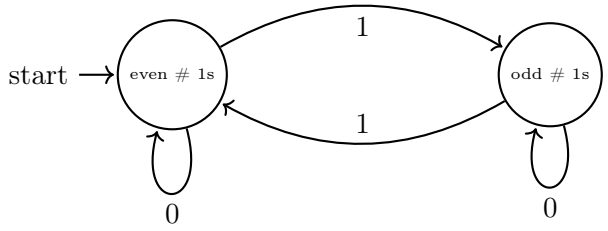
In our case we want to know whether there were even or odd number of 1s.

Where do we start from ? When I have seen nothing there are even number of 1s.

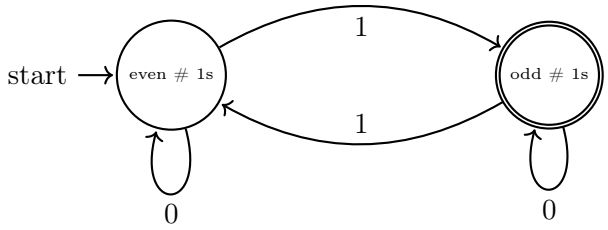
Now, suppose I receive a 0, I would remain in the same state, but if I get a 1 , the parity changes.



Now, if I am in the second state, if I get a 1 I will change states, and if I get a 0, parity is unchanged to I remain in the same state:



So when do we know that our string we have seen till now belongs to some language or not, we know that by marking some states as **accepting states**: usually represented by double circles, if we end up on this state, the string recieved till now belongs to out language: *is accepted*.



Such a formalism with **finite** states is known as **Finite Automata**.
Further, if for every string in Σ^*, there exists a unique path we will follow in the automata, such automata are also known **Deterministic Finite Automata (DFA)**.
Example
$\Sigma = \{0, 1\}$, let $L = \{w | w \in \Sigma^*,$ no. of 0's is a 0 mod 3 or 2 mod 3$\}$