CS 208 : Automata Theory and Logic

Spring 2024

Instructor : Prof. Supratik Chakraborty

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 $\varphi_1 \leftrightarrow \varphi_2$.

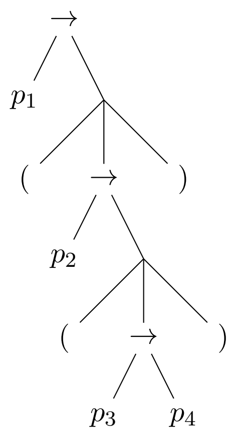
Propositional formulae as strings and trees:

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.

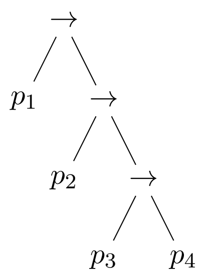
1.2 Semantics

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

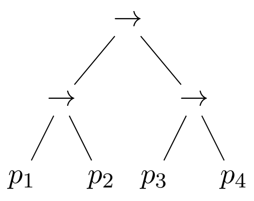
$$[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$$



(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	$\varphi_1 \land \varphi_2$
0	0	0
0	1	0
1	0	0
1	1	1

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

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

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

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 $[\![\varphi]\!] = 0$.
- $[\![\varphi_1 \land \varphi_2]\!] = 1$ iff $[\![\varphi_1]\!] = [\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \lor \varphi_2]\!] = 1$ 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	$p \lor q$
UNSAT	$p \land \neg p$
valid	$p \lor \neg p$
semantically entails	$\neg p \models p \to q$
semantically equivalent	$p \to q$, $\neg p \lor q$
equisatisfiable	$p \land q$, $r \lor s$

Table 1.4: Some examples for the definitions.

Example. Consider the formulas $\varphi_1 : p \to (q \to r)$, $\varphi_2 : (p \land q) \to r$ and $\varphi_3 : (q \land \neg r) \to \neg p$. 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**
$\wedge$	$\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$ $\quad \dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e_2}$
$\vee$	$\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i_1}$ $\quad \dfrac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i_2}$	$\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \to \varphi_3 \quad \varphi_2 \to \varphi_3}{\varphi_3} \vee_e$
$\to$	$\dfrac{\boxed{\begin{matrix}\varphi_1 \\ \vdots \\ \varphi_2\end{matrix}}}{\varphi_1 \to \varphi_2} \to_i$	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2} \to_e$
$\neg$	$\dfrac{\boxed{\begin{matrix}\varphi \\ \vdots \\ \bot\end{matrix}}}{\neg\varphi} \neg_i$	$\dfrac{\varphi \quad \neg\varphi}{\bot} \neg_e$
$\bot$		$\dfrac{\bot}{\varphi} \bot_e$
$\neg\neg$		$\dfrac{\neg\neg\varphi}{\varphi} \neg\neg_e$

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.	$\perp$	$\neg_e$ 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.	$\neg p$	$\wedge_{e_1}$ 1
5.	$\perp$	$\neg_e$ 3,4
6.	$p \to \perp$	$\to_i$ 3–5
7.	q	assumption
8.	$\neg q$	$\wedge_{e_2}$ 1
9.	$\perp$	$\neg_e$ 7,8
10.	$q \to \perp$	$\to_i$ 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 $\Sigma \vdash \varphi$

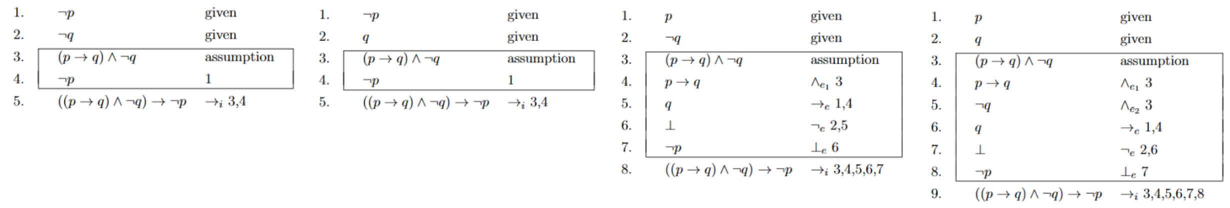
Let's take some example of semantic entailment. $\Sigma = \{p \to q, \neg q\} \models \neg p$.

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

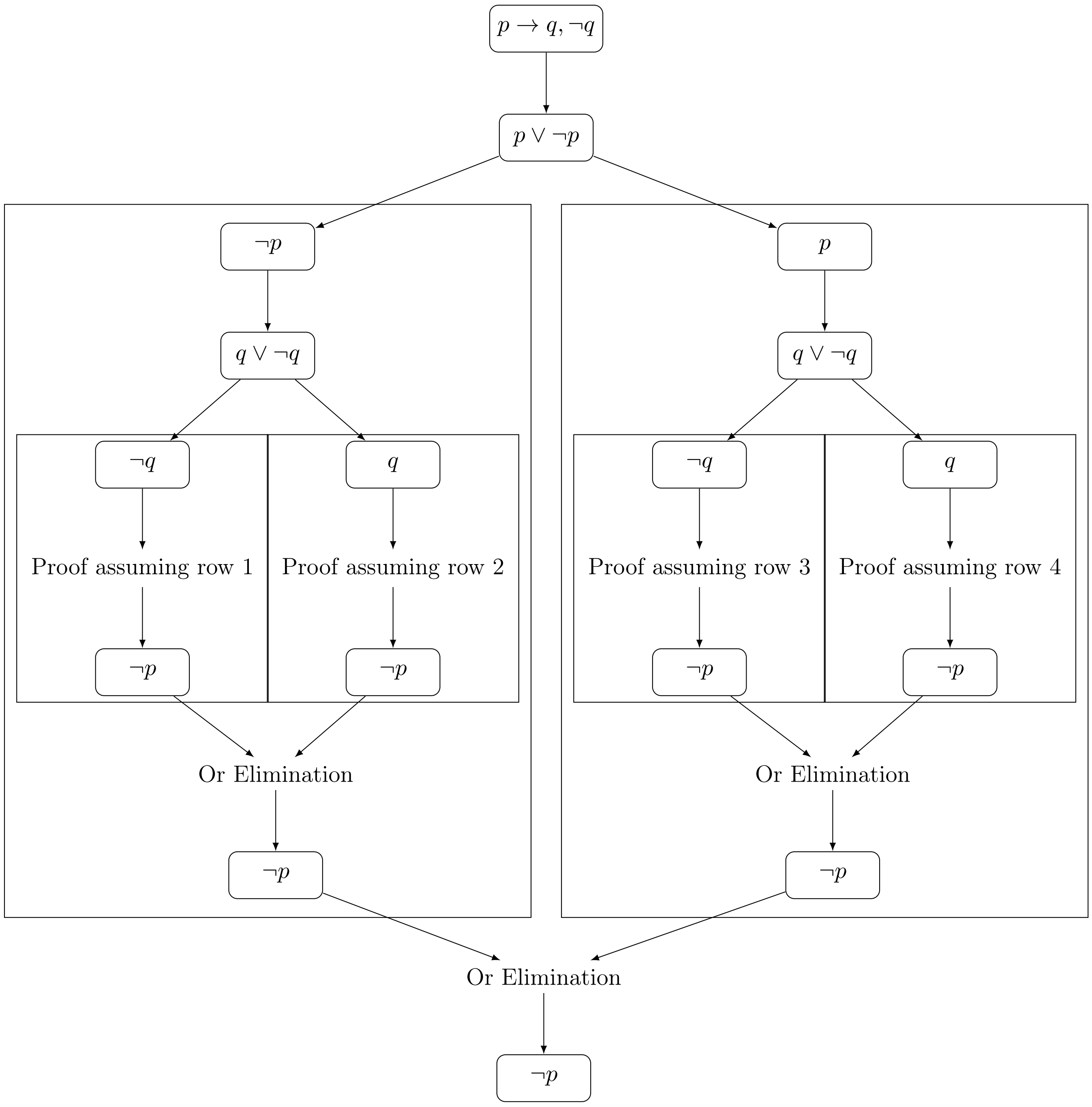
This looks promising, but we aren't done, we have only proven our formula under all possible assumptions, but we haven't exactly proven our formula from nothing given. But note that the reasoning we are doing looks a lot like case work, and we can think of the $\vee_e$ rule. In words, this rule states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

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

[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.

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

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

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.

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

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

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).

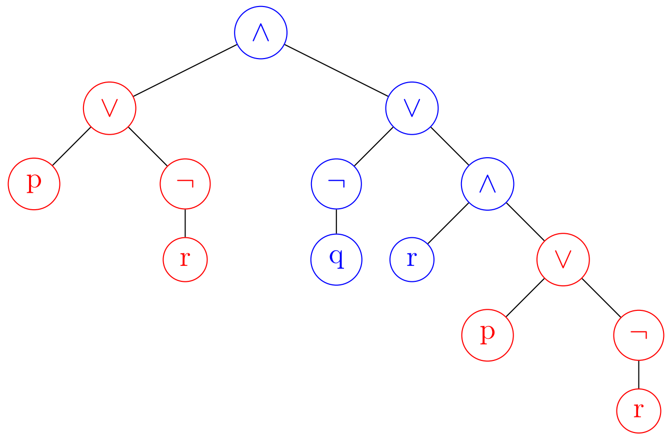
$$\neg(\phi_1 \wedge \phi_2) \dashv\models (\neg\phi_1 \vee \neg\phi_2)$$

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

1.8 Negation Normal Forms

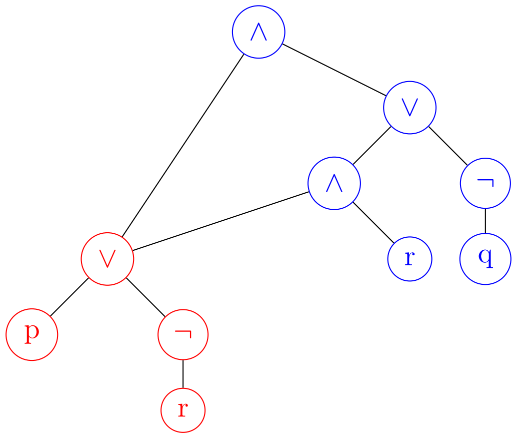
In mathematical logic, a formula is in negation normal form (NNF) if the negation operator ($\neg$) is only applied to variables and the only other allowed Boolean operators are conjunction ($\wedge$) and disjunction($\vee$) One can convert any formula to a NNF by repeatedly applying DeMorgan's Laws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

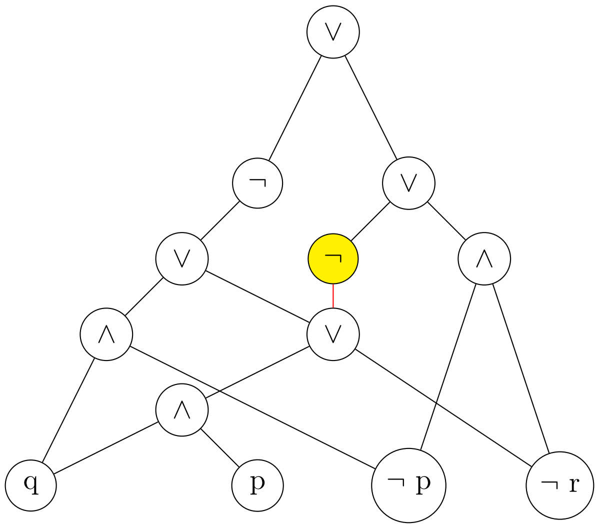
DAG Representation



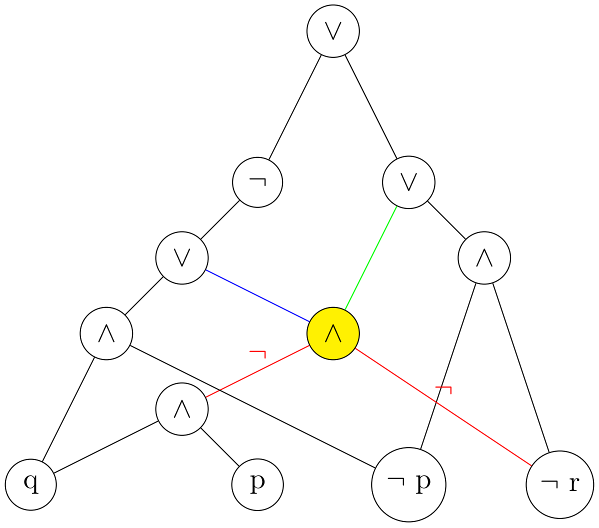
1.9 From DAG to NNF-DAG

Given a DAG of propositional logic formula with only ∨ , ∧ and ¬ nodes , can we efficiently get a DAG representing a semantically equivalent NNF formula ?

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



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.

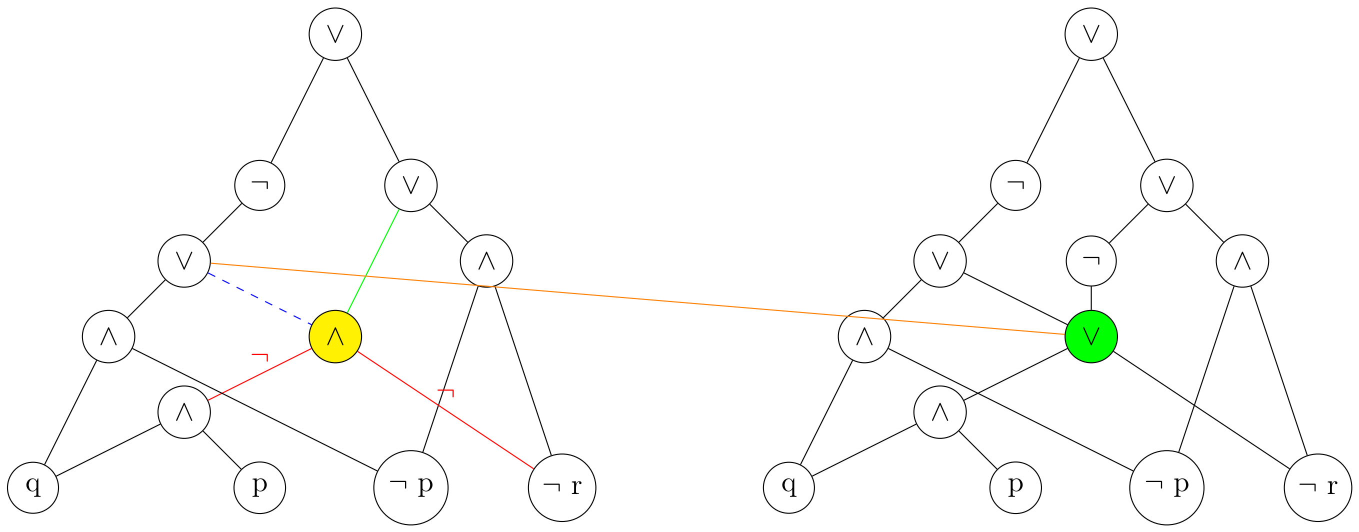


Figure 1.3: Step 1

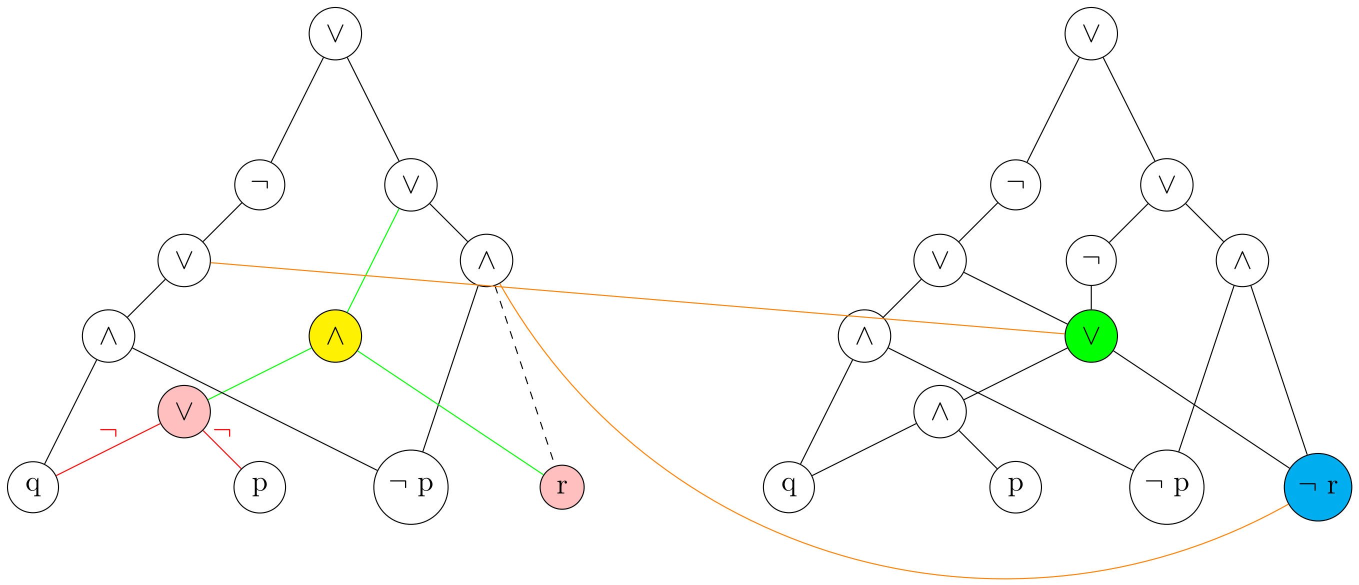


Figure 1.4: Step 2

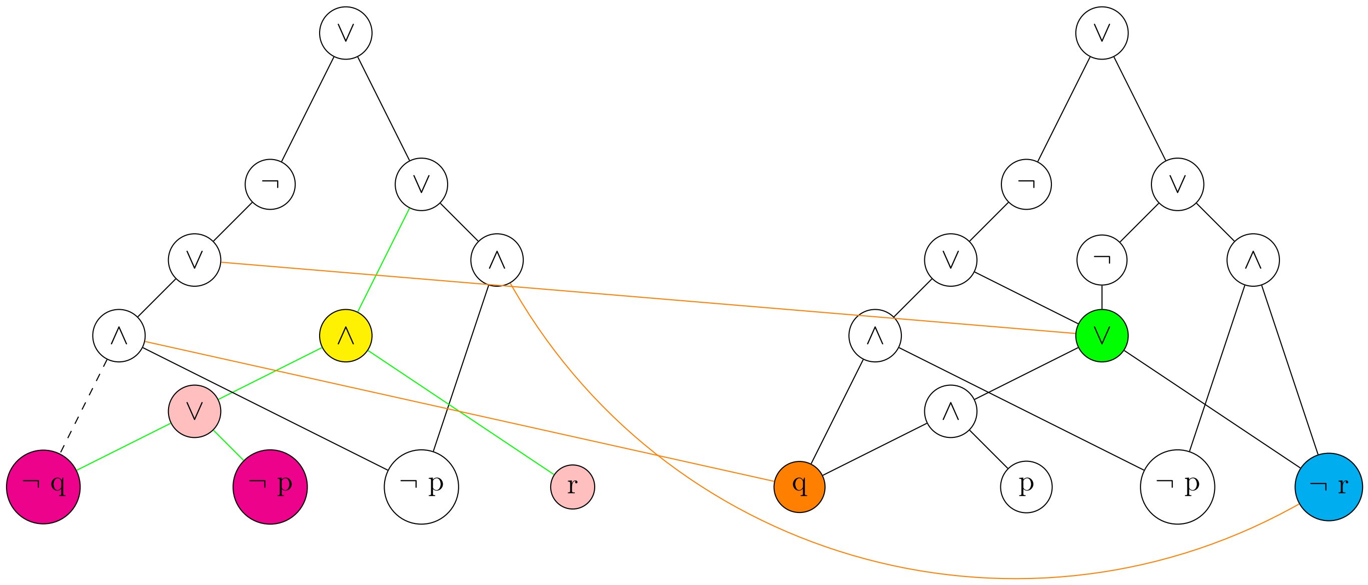


Figure 1.5: Step 3

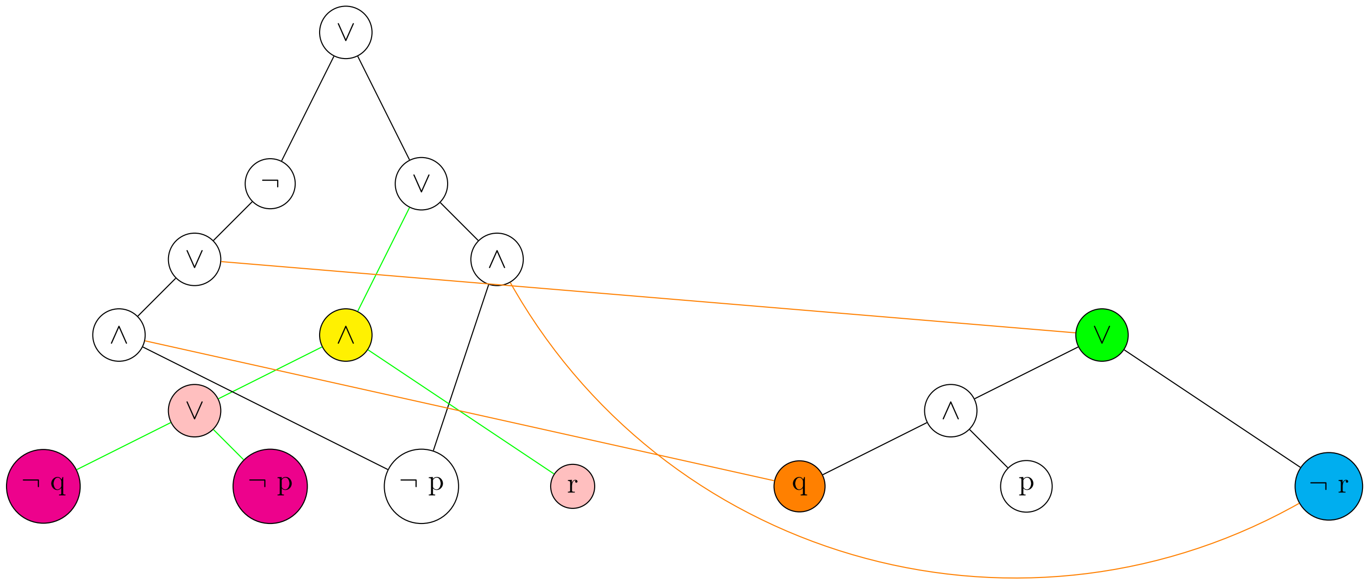


Figure 1.6: Step 4 : Remove all redundant nodes

1.10 An Efficient Algorithm to convert DAG to NNF-DAG

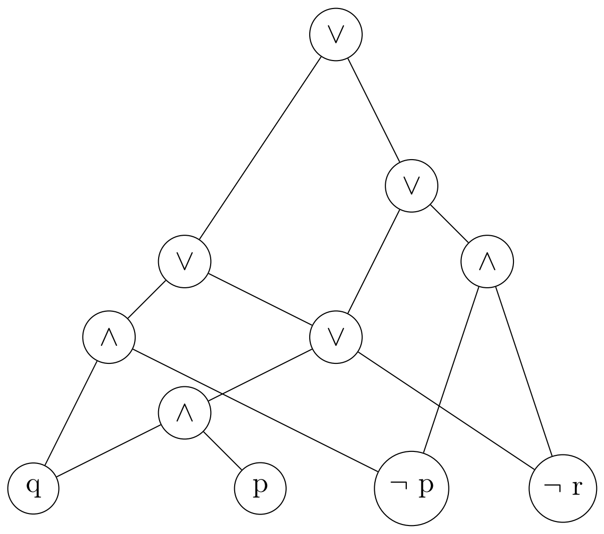


Figure 1.7: Step 1:Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

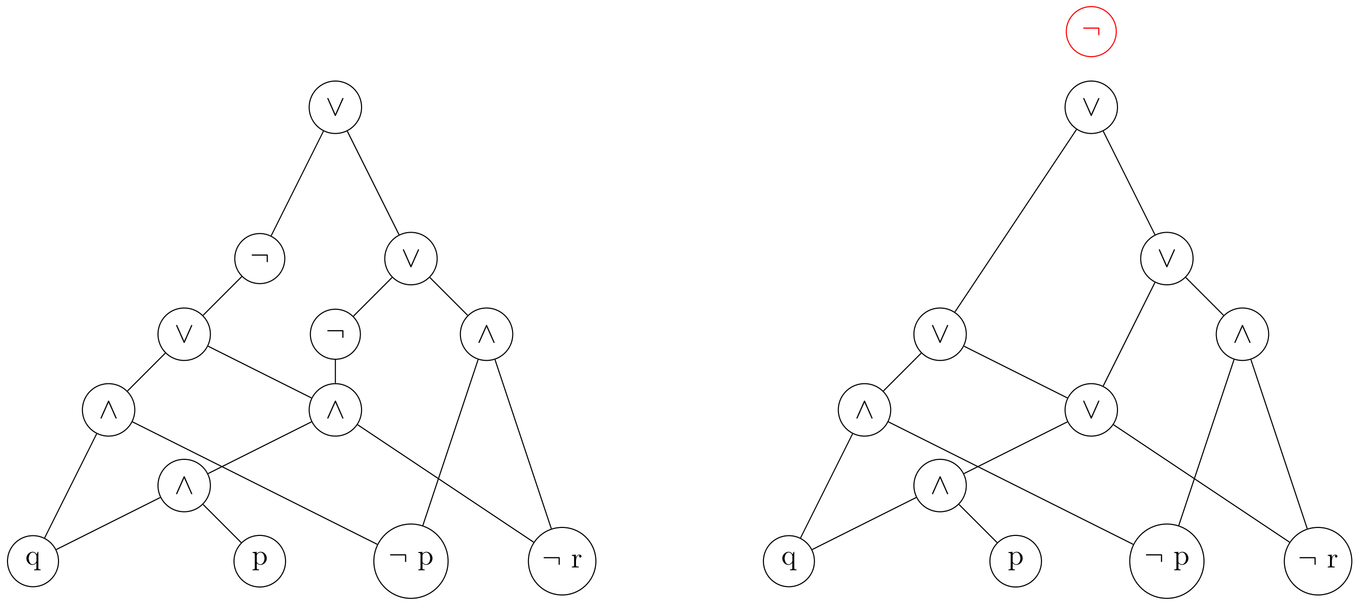


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

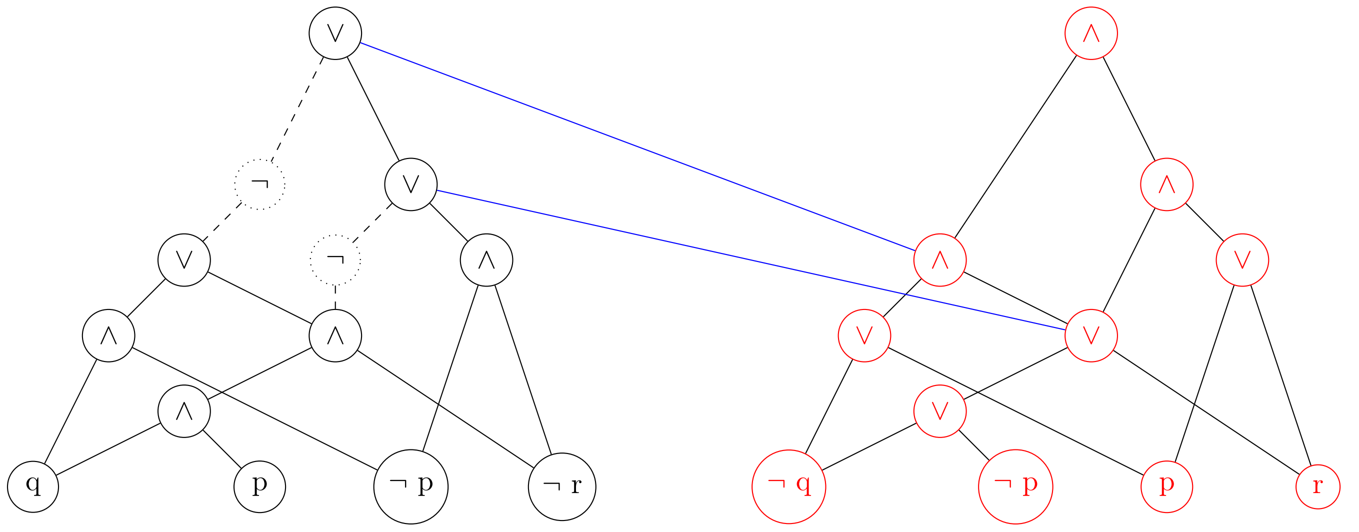


Figure 1.9: Step 3: Remove the "¬" nodes from the first DAG by connecting them to the corresponding node in the negated DAG.

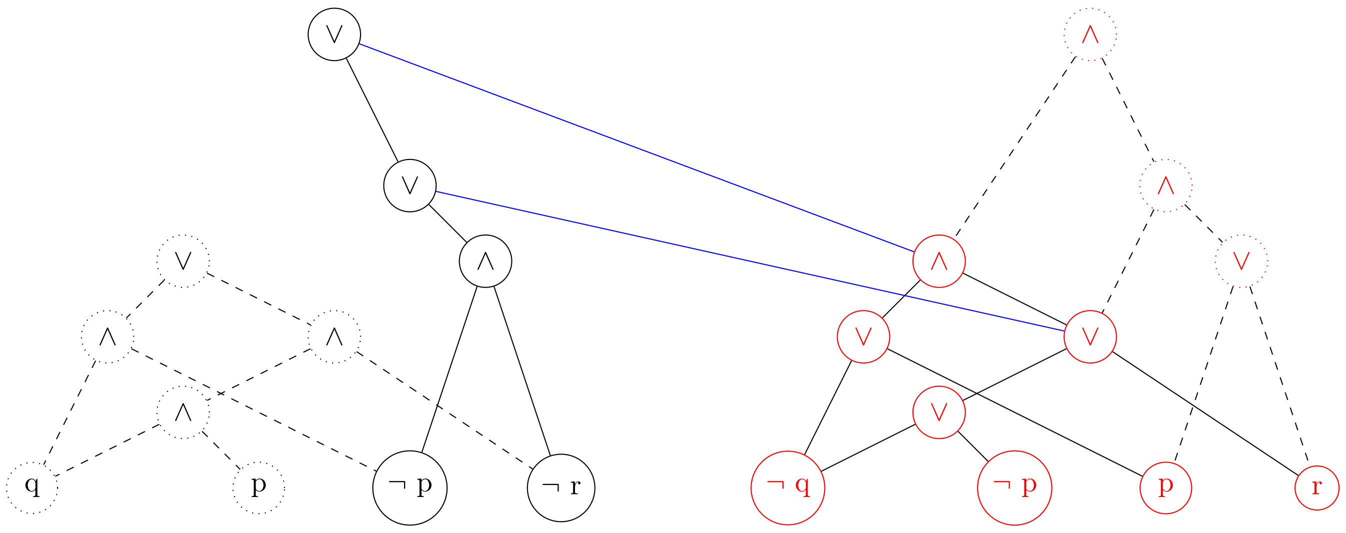


Figure 1.10: Step 4: Remove all the redundant nodes.

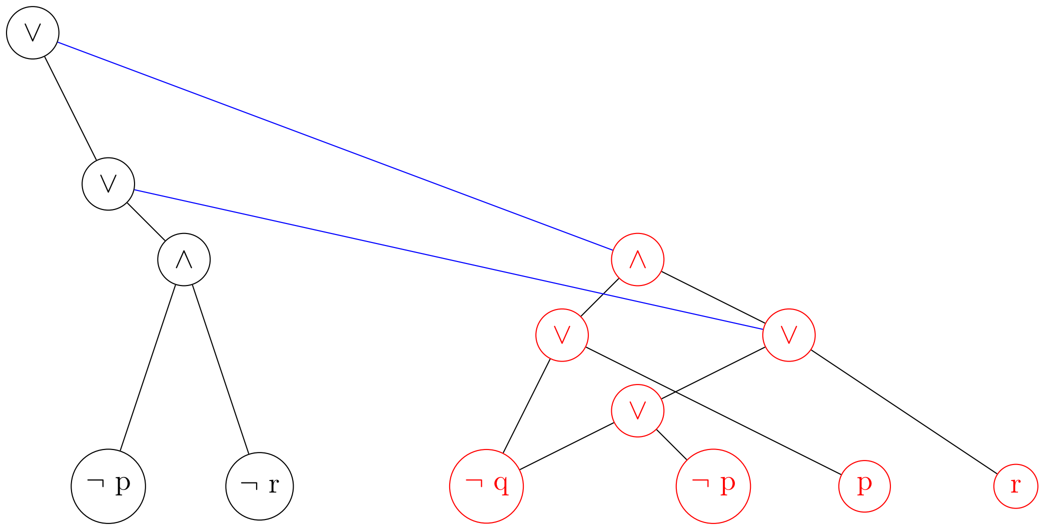


Figure 1.11: NNF - DAG

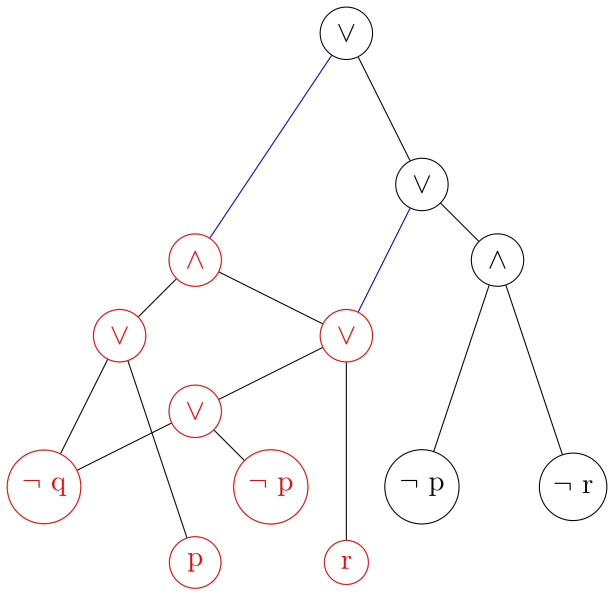


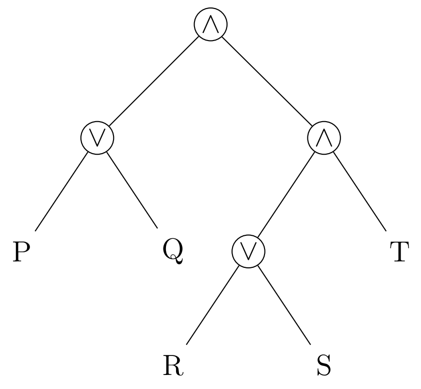
Figure 1.12: The NNF DAG (rearranged)

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.

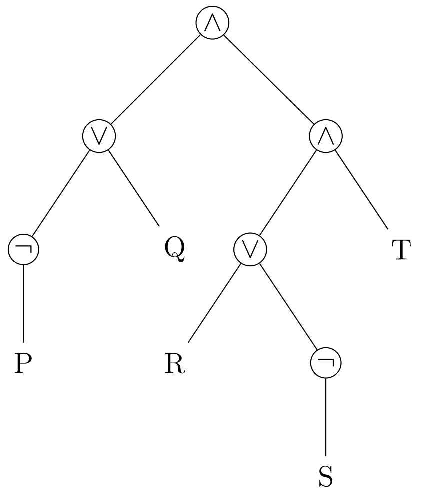
1.11 Conjunctive Normal Forms

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



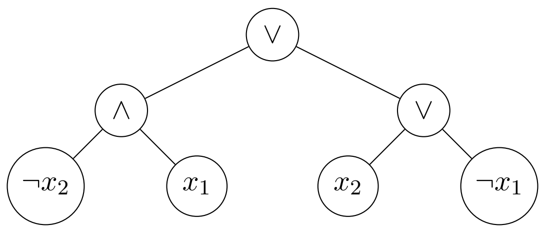
Some Important Terms

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

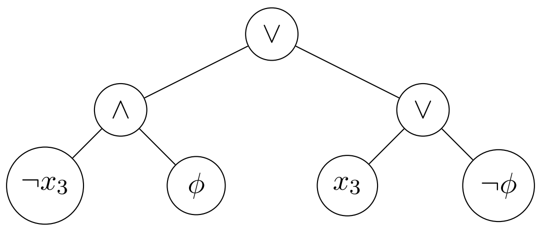
- **CLAUSE :**
  - A clause is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal
  - Example : p ∨ q ∨ (¬ r) , p ∨ (¬ p) ∨ (¬ r) -> Not Allowed
- **CUBE :**
  - a Cube is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal
  - Example p ∧ q ∧ (¬ r) , p ∧ (¬ p) ∧ (¬ 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 ∨ q ∨ (¬ r)) ∧ (q ∨ r ∨ (¬ s) ∨ 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 ∧ q ∧ (¬ r)) ∨ (q ∧ r ∧ (¬ s) ∧ t)

Given a DAG of propositional logic formula with only ∨ , ∧ and ¬ nodes , can we efficiently get a DAG representing a semantically equivalent CNF/DNF formula ?

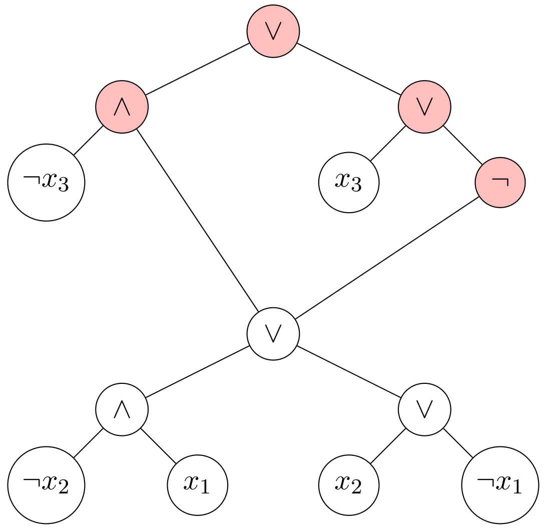
Tutorial 2 Question 2:
The Parity Function can be expressed as (((....($x_1 \oplus x_2$) $\oplus x_3$) .... $\oplus x_n$)
The Parse Tree for $x_1 \oplus x_2$ is



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:

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

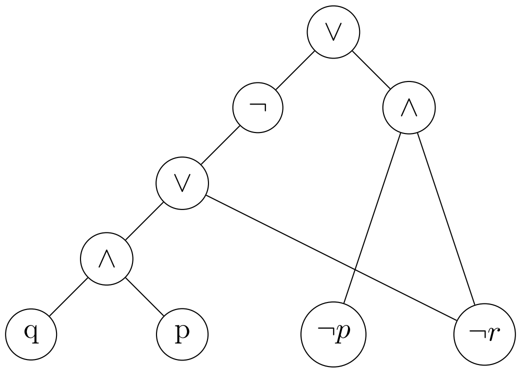
4. For checking the equisatisfiabiliy of ϕ and ϕ': Think about an assignment which makes ϕ true then apply that assignment to ϕ'.

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 $(\neg((q \land p) \lor \neg r)) \lor (\neg p \land \neg r)$



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

$$(t_1 \iff p \land q) \land (t_2 \iff t_1 \lor \neg r) \land (t_3 \iff \neg t_2) \land (t_4 \iff \neg p \land \neg r) \land (t_5 \iff t_3 \lor t_4) \land (t_5)$$

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:

$$(\neg x_1 \lor \neg x_2 \lor x_3) \land (\neg x_4 \lor x_5 \lor \neg x_3) \land (\neg x_1 \lor \neg x_5) \land (x_5) \land (\neg x_5 \lor x_3) \land (\neg x_5 \lor \neg x_1)$$

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.

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

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

```
Function HORN($\phi$):
    foreach occurrence of $\top$ in $\phi$ do
        mark the occurrence
    while there is a conjunct $P_1 \wedge P_2 \wedge \ldots \wedge P_{k_i} \to P'$ in $\phi$ do
        // such that all $P_j$ are marked but $P'$ isn't
        if all $P_j$ are marked and $P'$ isn't then
            mark $P'$
    if $\perp$ is marked then
        return'unsatisfiable'
    else
        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

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

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($\varphi$)
  $v \leftarrow$ 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

$$(a \lor \neg b) \land (\neg a \lor b)$$

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-

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

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

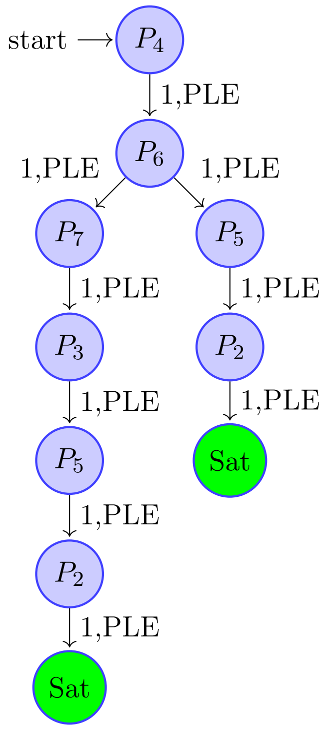


Figure 1.13: DPLL

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

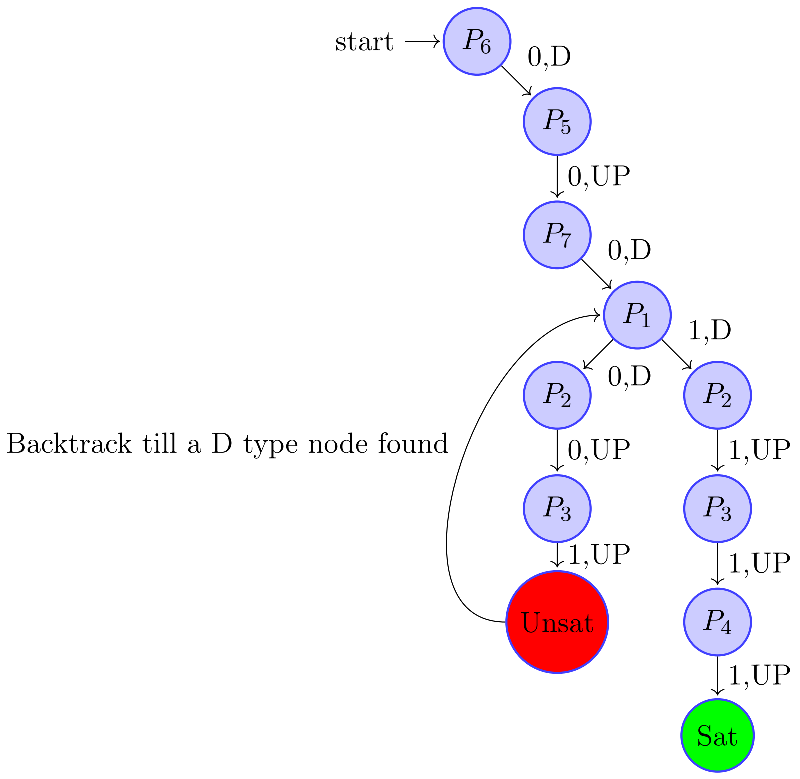


Figure 1.14: DPLL without point 2

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.

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:

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

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:

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

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),

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

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

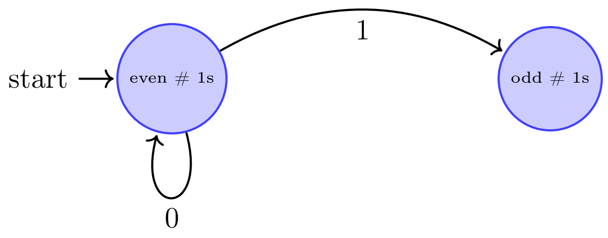
In our case we want to know whether there were even or odd number of 1s.

even # 1s
odd # 1s

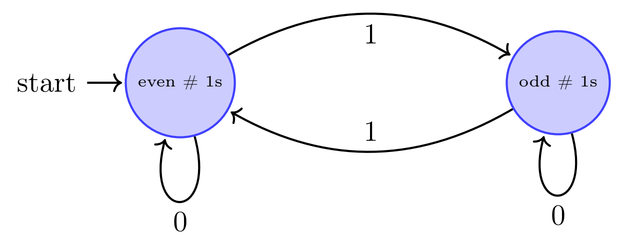
Where do we start from ? When I have seen nothing there are even number of 1s.

start
even # 1s
odd # 1s

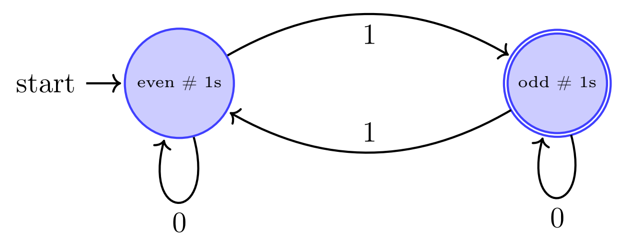
Now, suppose I receive a 0, I would remain in the same state, but if I get a 1 , the parity changes.



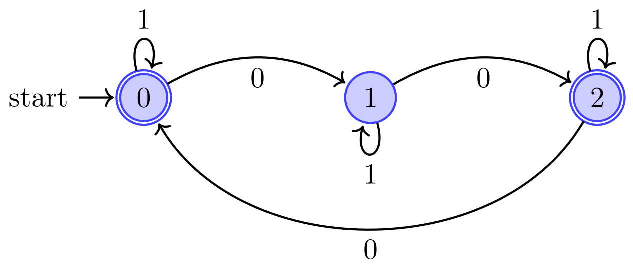
Now, if I am in the second state, if I get a 1 I will change states, and if I get a 0, parity is unchanged to I remain in the same state:



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$\}$



2.3 DFA Design - Example 1

Draw a Deterministic Finite Automaton (DFA) for $L := \{w \in \{a, b\}^*\} : 2$ divides $n_a(w)$ and 3 divides $n_b(w)\}$. Here $n_a(w)$ stands for the number of a's in w, and $n_b(w)$ stands for the number of b's in w. For example, $n_a(abbaab) = n_b(abbaab) = 3$. Therefore, $ababbaa \in L$ but $aabababa \notin L$.

In certain scenarios, expressing a language solely through propositional logic becomes impractical, particularly when the length of the strings is unknown or variable. For instance, consider above example. In this case, the length n of the string is not explicitly provided, making it challenging to construct a propositional logic expression directly. Propositional logic typically operates on fixed, predetermined conditions or patterns within strings, which cannot accommodate variable lengths.
However, deterministic finite automata (DFAs) offer a suitable alternative for such situations. DFAs

are well-suited for languages where the structure and properties depend on the characters within the string rather than on fixed string lengths. By employing states and transitions based on input characters, DFAs can effectively recognize languages with variable-length strings and complex patterns, making them a more appropriate choice when string length is not predetermined.
We will denote Σ as the set of alphabets.
$\sum = \{a, b\}$
$L = \{\omega \in \sum^* : n_a(\omega)\, is\, divisible\, by\, 2\, and\, n_b(\omega)\, is\, divisible\, by\, 3\}$

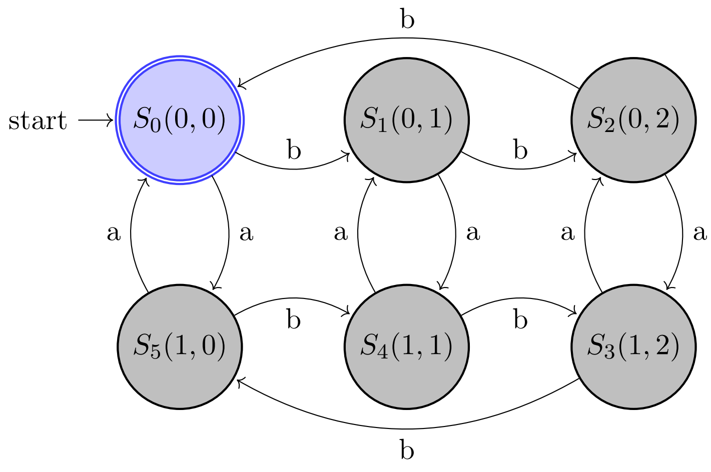


Figure 2.1: DFA for above task

S_0 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 0$
S_1 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 1$
S_2 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 2$
S_3 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 2$
S_4 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 1$
S_5 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 0$

2.4 DFA Design - Example 2

We will look at another example now.

$$\Sigma = \{a, b\}$$

$$L = \{\mathbf{w} \in \Sigma^* \mid \mathbf{n}_{ab}(\mathbf{w}) \;=\; \mathbf{n}_{ba}(\mathbf{w})\}$$

$$w \;= \underline{a\overline{b}}\overline{a}\underline{a\overline{ba}b} \quad (\mathbf{n}_{ab}(\mathbf{w}) = 3 \;\&\; \mathbf{n}_{ba}(\mathbf{w}) = 2)$$

If we try to cleverly convert the problem into a simpler one, we will observe that $\mathbf{n}_{ab}(\mathbf{w}) = \mathbf{n}_{ba}(\mathbf{w})$ will always be true if the start and end alphabets are same (be it a or b).

So the above problem simplifies to:

$$L = \{\mathbf{w} \in \Sigma^* \mid \text{First and Last alphabet are same}\}$$

Forming an automation for this task can be done as:

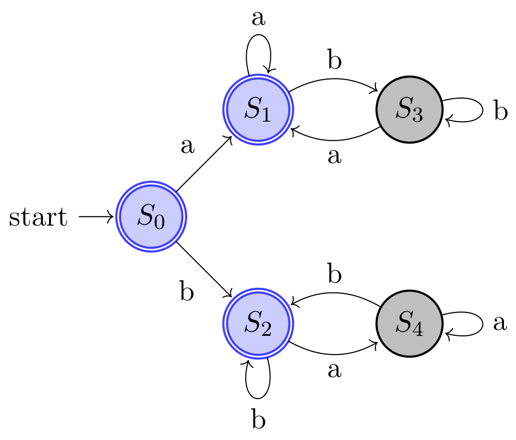


Figure 2.2: DFA: Last and First alphabets are same

Chapter 3

Non-Deterministic Finite Automata (NFA)

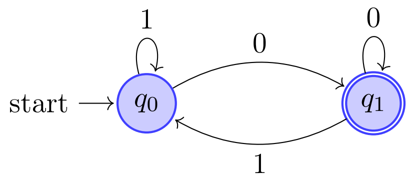
In the last few lectures, we covered the formalization of deterministic finite automata (DFA) where the transition function outputs a single state for a given input and current state. In this lecture, we will discuss non-deterministic finite automata (NDFA) where the transition function can output multiple states (or a set of states) instead.

3.1 NFA Representation

As discussed earlier, a DFA is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a finite set of all states
- Σ is the alphabet, a finite set of input symbols
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $F \subseteq Q$ is the set of final/accepting states

For example, consider the following automaton:



It can be represented as:

$$\text{DFA}\ (\ \{q_0, q_1\},\ \{0, 1\},\ q_0,\ \delta,\ \{q_1\}\ )$$

where δ is the transition function:

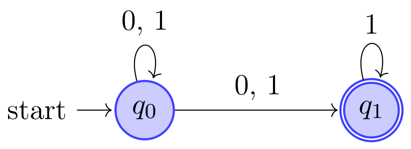
Q	Σ	Q'
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_0

3.1.1 Formalization of Non-Deterministic Finite Automata

However, in NDFA,

- $q_0 \subseteq Q$ is the set of initial states
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function

Hence, consider the following non-deterministic finite automaton (A):



It can be represented as:

$$\text{NDFA}\left(\ \{q_0, q_1\},\ \{0,1\},\ \{q_0\},\ \delta',\ \{q_1\}\ \right)$$

where δ' is the transition function: As we can see, δ' is a partial function whose output is a set of

Q	Σ	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	1	$\{q_0, q_1\}$
q_1	1	$\{q_1\}$

states instead of a single state.

3.1.2 NFA into action

Due to its transition function, an NDFA gives **choices for path taken** at some states for a given input string. Any string for which there exists a path from the initial state to a final state is considered to be accepted by the NDFA.

Hence, the NDFA shown above has the language $L(A) = \Sigma^* \setminus \{\epsilon\}$, i.e., the set of all strings over the alphabet Σ except the empty string.
For example, the string 011 is accepted by A as it has the following path $q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_1$.

To determine if a string is accepted by an NDFA, we can check if the set of states reachable from the initial state by reading the string contains any final state.

Here we have: Here we have:

Initial State	Σ^*	Reachable States
q_0	0	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$
q_0	011	$\{q_0, q_1\}$

As $\{q_0, q_1\}$ contains $q_1 \in F$, the string 011 is accepted by A. By construction, there will always be a set of choices which reach q_1 from q_0 for the input 011.

3.2 Equal expressiveness of DFA or NFA

We claim that regular languages, ie, the set of languages which can be defined by DFAs, is exactly the set of languages which can be defined by NFAs. In other words, for every NFA, there exists a DFA that accepts the same language, and vice versa.Even though NDFA gives choices at some states, it can still be represented as some equivalent DFA. Even though NDFA gives choices at some states, it can still be represented as some equivalent DFA. This implies that **NDFAs have no more expressive power than DFAs** in terms of string acceptance. However, representation in form of NDFA is much more succinct than DFA.

We can convert a NDFA to a DFA by considering the set of reachable states as states of the equivalent DFA.

3.2.1 Construction of DFA from NFA

For an NFA $(Q, \sum, Q_0, \delta, F)$, construct a DFA $(\mathcal{P}(Q), \sum, Q_0, \delta', F')$ with:

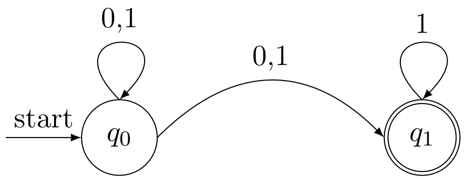
$$\delta'(G, \sigma) = \bigcup_{q \in G} \delta(q, \sigma)$$

$$F' = \{q : q \in \mathcal{P}(Q), q \cap F \neq \phi\}$$

Notice that in our new DFA, the states are labelled as subsets of the states of the NFA. This means that we have 2^n states in our DFA if the NFA had n. It is left to the reader to verify that the DFA we have defined satisfies all the requirements of a DFA.
We claim that after the same characters are inputted into both the NFA and DFA, the state of the DFA is labelled the same as the set of current states of the NFA. This claim is easy to check using the definition of the δ' function.
Now, in an NFA, a string is accepted if any one of the active states at the end of the string is in the set of accepting states. Clearly, with our interpretation of the DFA, this is equivalent to being in a state that belongs to the F' we have defined.

3.2.2 Step Wise Conversion from NFA to DFA

Now we'll see how to convert a NFA to a DFA through an example.
We call the following NFA 'A'



The language depicted by this NFA is all the strings formed using {0,1} excluding the empty string(ϵ). We represent this as:

$$L(A) = \{w \in \{0,1\}^* | w \text{ is accepted by A}\}$$

that is,

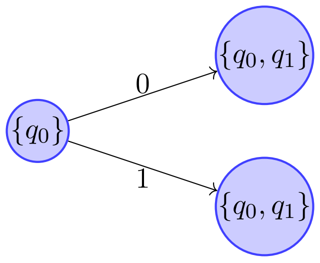
$$L(A) = \Sigma^* \backslash \{\epsilon\}$$

The transition function(δ) table looks as follows:

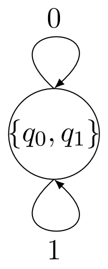
Q	Σ	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	00	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$

To convert this NFA to a DFA, we need to track the states that can be reached after n choices which will be a subset of 2^Q.

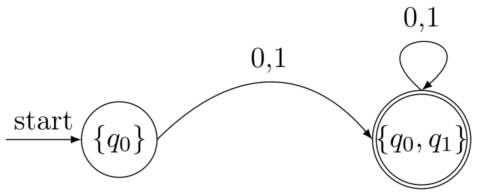
Step 1



Step 2



Final DFA



Call this DFA 'A''.
One property we've extensively used in the conversion is:

$$\textbf{If } S \subseteq Q\textbf{, then } \delta(S,0) = \bigcup_{q \in S} \delta(q,0)$$

Now, to show that the languages represented by the NFA(A) and the DFA(A') are the same, i.e, $L(A) = L(A')$, we need to show the following:

1. $L(A) \subseteq L(A')$
2. $L(A') \subseteq L(A)$

3.2.3 Proof of Equivalence

To show that the languages represented by the NFA(A) and the DFA(A') are the same, i.e, $L(A) = L(A')$, we need to show the following:

- $L(D) \subseteq L(N)$
- $L(N) \subseteq L(D)$

We can prove that the equivalent DFA D accepts exactly the same language as the original NDFA N, i.e, $L(D) = L(N)$.

3.3 Reflective Insights

Even with all the extra "powers" and behaviours NFAs can have on top of those of DFAs, they are equally expressive. This shows us that the limitation in expressiveness is not in the behaviour of state transitions but in the finiteness of states. However, due to the exponential blowup in the number of states, it is often more human-readable to express certain automata/languages through NFAs, making it a convenient representation tool.

3.4 Proof of Equivalence – Correctness

In the last lecture, we discussed the conversion of Nondeterministic Finite Automata (NFA) to its equivalent Deterministic Finite Automata (DFA)

3.4.1 Claim

We aim to demonstrate the equivalence of the languages accepted by NFA A and DFA A', denoted as $L(A)$ and $L(A')$ respectively. In other words, we want to show that $L(A) = L(A')$, where A represents the original NFA and A' represents the DFA obtained through subset construction from NFA A. Alternatively, we can establish that $L(A) \subseteq L(A')$ and $L(A') \subseteq L(A)$, which implies $L(A) = L(A')$.

3.4.2 Proof

We'll prove this claim by showing that for all $n \geq 0$ and for every word $w \in \Sigma^*$ with $|w| = n$, NFA A can reach state $q \in Q$ [1] upon reading w if and only if DFA A' reaches state $S \subseteq Q$ such that $q \in S$.
Given the condition that for all $n \geq 0$, and for every word w, where $w \in \Sigma^*$ such that $|w| = n$, we aim to demonstrate the equivalence between the NFA A's ability to reach state $q \in Q$ upon reading word w and the DFA A''s capability to reach state $S \subseteq Q$, where $q \in S$. In other words, we want to show that words in the language recognized by NFA A reach a certain state (say, a final state q), **if and only if** words in the language recognized by DFA A' reach state S (where $S \subseteq Q$, $q \in S$, and S is a final state in the equivalent DFA A').
Thus, we can establish that the set of words accepted by NFA A is equivalent to the set of words accepted by DFA A', implying $L(A) = L(A')$, thereby validating our claim.

We will demonstrate this by induction on n.

1. Base case: When $n = 0$, i.e., $|w| = 0$, it essentially means that we are at the initial state of the automaton. The initial state of the DFA A' is represented by the singleton set containing the initial states of the NFA A. So, definition of initial state of DFA A' satisfies the claim.

2. Induction hypothesis: Assume the claim holds for all $0 \leq n < k$ for some $k > 0$.

3. Inductive step: We'll show that the claim holds for $n = k$.
   Since $|w| = k$, let $w = w'.a$ (w' concatenate a) , where $w \in \Sigma^*$, $w' \in \Sigma^*$, $a \in \Sigma$, and $|w'| = k-1$.

   By hypothesis , there exists the following path in both automata :

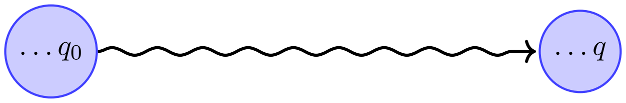
   $$\xrightarrow{\qquad\qquad w' \qquad\qquad}$$

   This basically means that there exists some path in NFA and equivalent DFA which is as follows :

   In NFA A :

   $$\begin{tikzcd} q_0 \arrow[r, squiggly] & q \end{tikzcd}$$

   In DFA A' [2]:

   Now, if a symbol a, $a \in \Sigma$ comes, word w is formed :

   $$\xrightarrow{\qquad\qquad w' \qquad\qquad}\xrightarrow{\ a\ }$$

   $$\xrightarrow{\qquad\qquad\quad w \qquad\qquad\quad}$$

[1]Q is the set of states in our original NFA A.
[2]$\ldots q_0 \Rightarrow$ set containing initial states of the original NFA A
$\ldots q \Rightarrow \subseteq Q$ containing the state q, where Q is the set of states of the original NFA A
$\ldots \hat{q} \Rightarrow \subseteq Q$ containing the state $\hat{q}$, where Q is the set of states of the original NFA A

NFA A will reach some state $\hat{q}$

$$\begin{tikzcd} q_0 \arrow[r, squiggly] & q \arrow[r, "a"] & \hat{q} \end{tikzcd}$$

DFA A' will reach some state $\ldots\hat{q}$ ($\ldots\hat{q} \subseteq Q$ and contains $\hat{q}$)

$$\begin{tikzcd} \ldots q_0 \arrow[r, squiggly] & \ldots q \arrow[r, "a"] & \ldots\hat{q} \end{tikzcd}$$

This is because state $\ldots q$ of DFA A' contains q, which transitions DFA A' to state $\ldots\hat{q}$ (a set containing $\hat{q}$) when symbol a is encountered.
Thus, we have shown that for all $n \geq 0$, and for every word w where $w \in \Sigma^*$ such that $|w| = n$, NFA A can reach state $q \in Q$ on reading word w **AND** DFA A' reaches state $S \subseteq Q$ such that $q \in S$.

Hence, proved the condition and the claim.

Remarks:

(a) As the length of the word increases, the number of choices for state transitions in the NFA grows exponentially.

(b) If an NFA has N states, the equivalent DFA can have an exponential number of states in the worst case.

3.5 Compiler Special Case – Lexical Analyzer

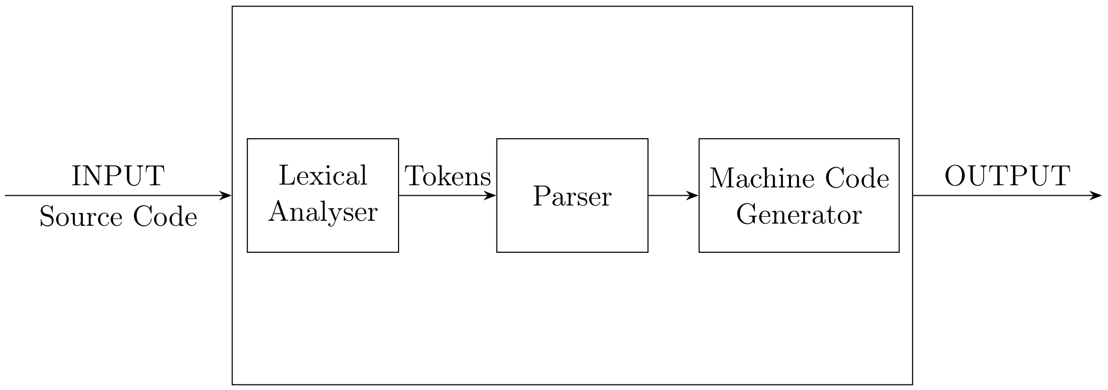


Figure 3.1: A Simple Compiler

In simpler terms, a compiler is composed of three main components: a lexical analyzer, a parser, and a code generator. Its primary function is to process a sequence of characters as input. The lexical analyzer breaks down a sequence of characters into tokens, which are then analyzed by a parser. To accomplish this, the lexical analyzer employs a NFA to determine whether a particular state can be reached in the corresponding DFA and traces a path accordingly. When faced with a long string, finding the final state can be challenging. At each step, there are multiple choices to

explore. However, as the length of the input increases, exhaustively exploring each choice becomes increasingly difficult. Converting an NFA to a DFA may result in an exponential increase in states. However, itâs important to note that not all states are necessary to reach the final state. This excessive expansion of states may lead to unnecessary complexity, creating the entire DFA when itâs not actually required.
So, the lexical analyzer determines whether there exists a path in the automaton that leads to an accepting state for a given word without constructing the equivalent DFA.
Suppose the Lexical Analyser of Compiler have to check whether a GIVEN WORD is accepted by the following NFA.

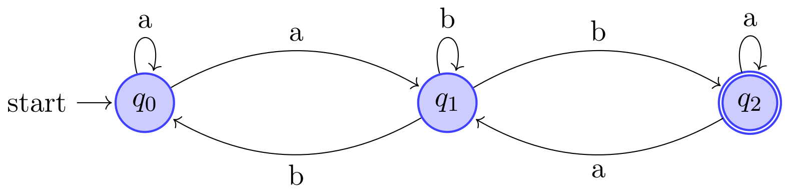


Figure 3.2: Lexical analyser Automaton

Question: How can a lexical analyzer determine whether a given word leads to an accepting state without constructing the equivalent DFA?
Answer:

$$\begin{tikzcd}
\{q_0\} \arrow[r, "a"] & \{q_0, q_1\} \arrow[r, "b"] & \{q_0, q_1, q_2\} \arrow[r, "a"] & \{q_0, q_1, q_2\} \arrow[d, "b"] \\
 & \{q_0, q_1, q_2\} & \{q_0, q_1, q_2\} \arrow[l, "b"'] & \{q_0, q_1, q_2\} \arrow[l, "a"']
\end{tikzcd}$$

Lexical Analyser can track the set of states the NFA could be in after reading each symbol of the GIVEN WORD, updating the state based on the transitions specified by the GIVEN NFA.

- If the final set of states contains at least one accepting state of the original NFA, then there exists a successful path for the given word to reach an accepting state, indicating acceptance by the given automaton.
- The time complexity for this process is $O(n \cdot k)$, where n represents the size of the automaton and k denotes the length of the word. This complexity is significantly lower than the usual exponential time complexity observed in similar processes.[3]
- Here the GIVEN WORD is accepted by the GIVEN NFA[4]

[3]Size of automaton = No. of states in Automaton + No. of Transition Arrows in Automaton
[4]q_2 is contained in the last set

3.6 NFA with $\epsilon-$edges

A variation of NFA is NFA with ϵ edges, it may expand the language by making words accepted that were otherwise unaccepted.

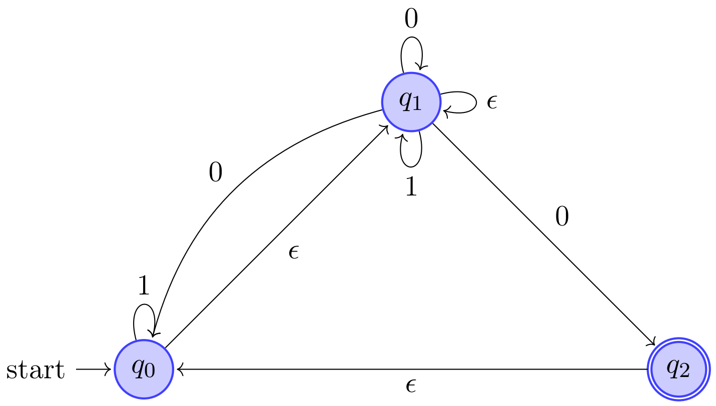


Figure 3.3: Automaton with Epsilon Edges

$\epsilon-$edges bring non-determinism to the NFA, as you can sit on a node and take one of the $\epsilon-$edges possible from that node to jump to another node without consuming any letter of the input.
Figure 3.3 shows how ϵ edges are used to connect states of an automaton for free(without consuming any letter from input), we can see that $10 \notin L$ without the ϵ edge between q_0 and q_1, but with the presence of this ϵ edge $10 \in L$.
$\epsilon-$edge also allows us to connect two automatons. In Figure 3.4, the accepting states of L_1 are connected to the start states of L_2 automaton, which generates an automaton for accepting $L_1 \cdot L_2$. Here $\cdot$ is the concatenation operator. So if $\omega_1 \in L_1$ and $\omega_2 \in L_2$, then $\omega_1 \cdot \omega_2 \in L_1 \cdot L_2$ will be accepted by this new automaton.

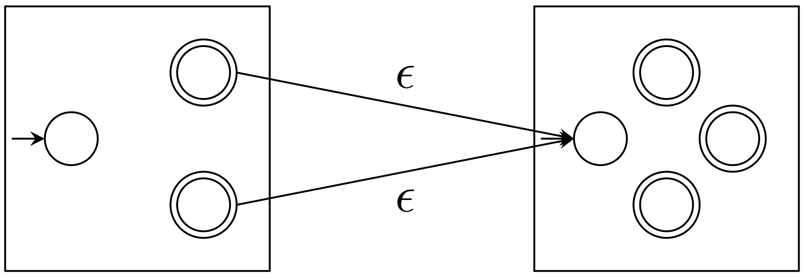


Figure 3.4: L_1 accepting automaton and L_2 accepting automaton connected

Now, we will try to find an equivalent DFA of this NFA having ϵ edges. For that, we will first find an NFA without ϵ edges which will preserve the original NFA, and this obtained NFA can then be constructed into a DFA.
Initially, just look at the $\epsilon-$edges only and find for each state its $\epsilon-$closure, which is the set of the states that we can reach from it by taking only $\epsilon-$edges.
From each node, we can go to every node that is present in the $\epsilon-$closure of that node for free. So wherever non-ϵ edges take us from these nodes in $\epsilon-$closure, we can reach from the node itself whose

closure it was. So all these states will be connected to this node, in the new NFA.
The starting states and the final states of this new NFA will be the $\epsilon-$closures of the start and final states respectively of the original NFA.
So NFA without $\epsilon-$edges for the NFA in Figure 3.3 will look like as shown in Figure 3.5

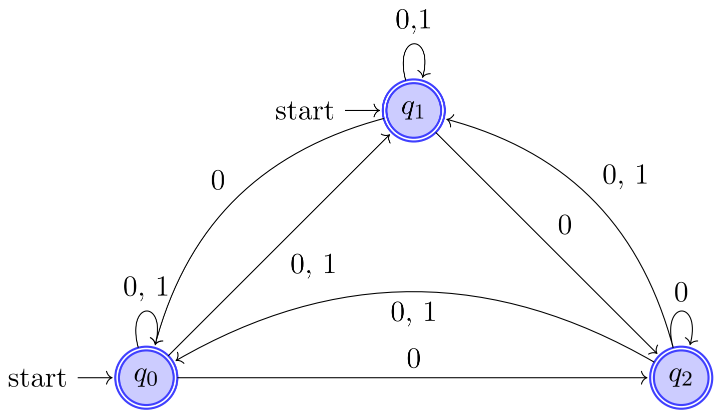


Figure 3.5: Automaton without Epsilon Edges

3.6.1 Equivalence with DFA

- If we can show that ϵ-NFA have an equivalent NFA, then the equivalence of ϵ-NFA and DFA is proved because every NFA has an equivalent DFA[5].

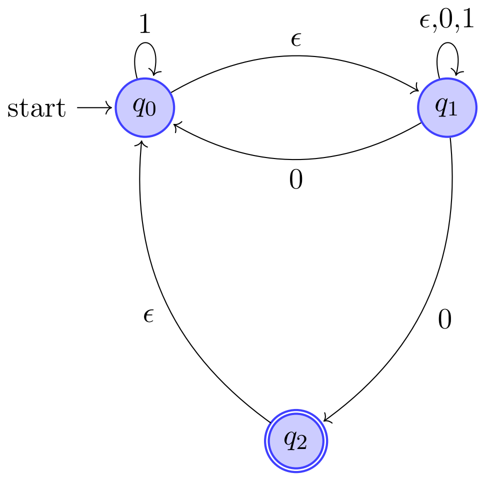


Figure 3.6: A Simple ϵ-NFA

Focus on ϵ-Edges of the Automaton to get the Epsilon Closure of the states in an Automaton

For Figure-3.6 Automaton:

[5]equivalence of NFA and DFA is already proved

$$q_0 \rightsquigarrow \{q_0,\, q_1\} \qquad \epsilon\text{-closure}_{q_0}$$

$$q_1 \rightsquigarrow \{q_1\} \qquad \epsilon\text{-closure}_{q_1}$$

$$q_2 \rightsquigarrow \{q_0\ ,\ q_1\ ,\ q_2\} \qquad \epsilon\text{-closure}_{q_2}$$

Rules for converting ϵ-NFA to equivalent NFA[6]:

1. Non-epsilon transitions for a symbol, denoted as $a \in \Sigma$, originating from any state q' within the epsilon closure of a state, say q, will also be present as non-epsilon transitions for state q in the equivalent NFA. These transitions leads to the same destination states for q in new NFA as they were for the state q' in the original epsilon-NFA.
2. The states in the epsilon closure of the initial state in the epsilon-NFA can serve as the initial state in the new NFA. However, this is not necessary, we can make an equivalent NFA where ϵ-NFA and the new NFA have same initial state(s).
3. All states within the epsilon-NFA that include the accepting state in their epsilon closure will also act as final states in the new/equivalent NFA.

For the Figure-3.6 epsilon-NFA, after applying the aforementioned rules, we obtain the following equivalent NFA:

[6] These rules will become more clearer in the next class, when we will learn about leading epsilon transitions and trailing epsilon transitions within the states of ϵ-NFA.

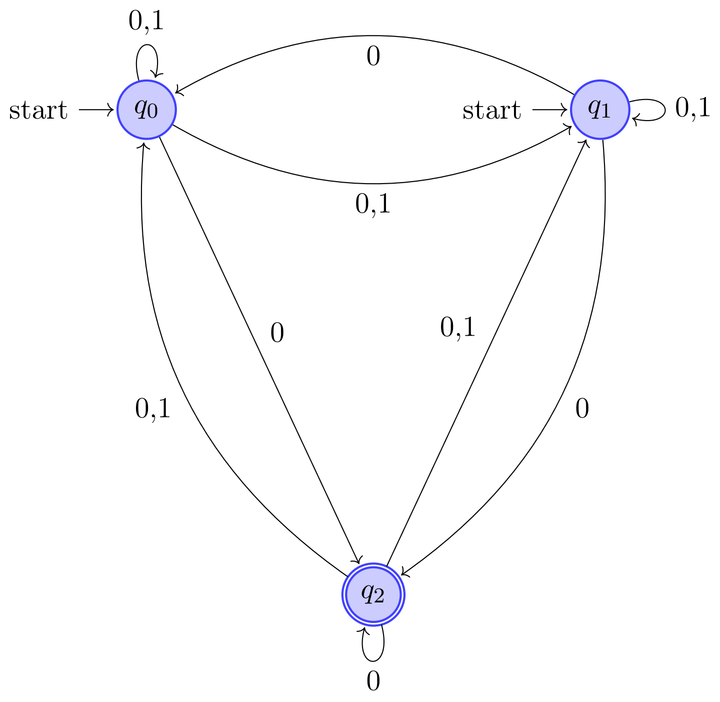


Figure 3.7: Equivalent NFA for Figure-3.6

3.7 Recap

We were trying to convert an NFA with ε edges to an equivalent NFA without ε edges in the previous lecture. We'll do that in more detail in this lecture.

3.7.1 Converting ε-NFA to non-ε-NFA

- **ε-closure** of a node is defined as set of those nodes which can be reached from that node by traversing over ε-edges, i.e. without consuming any character from the alphabet Σ. Also, the node itself is trivially a part of its ε-closure.
- **ε-edges** are those edges which can be traversed without consuming any character from the alphabet Σ, i.e. by consuming an empty string . Observe that the string "10" $\in L$ with ε-edges but without ε-edges, string "10"$\notin L$ where L is the Language of the NFA.

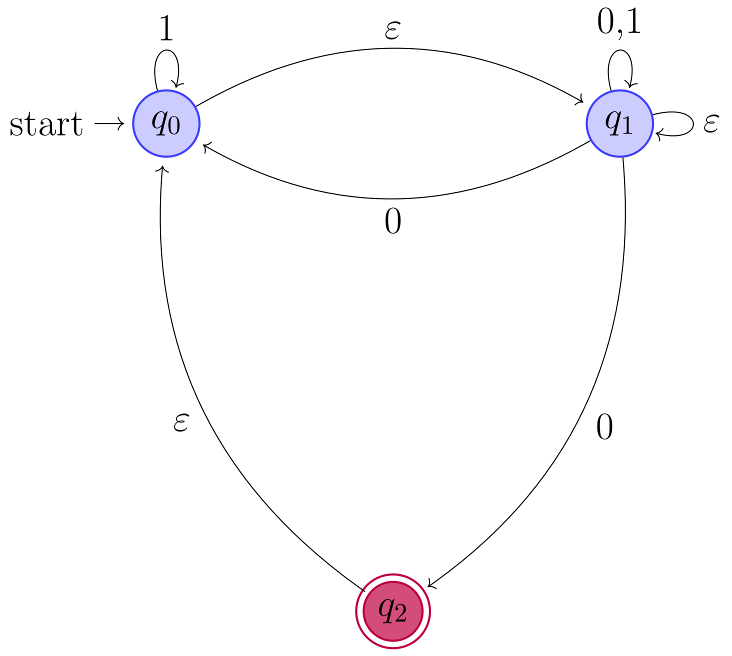


Figure 3.8: A Simple ε-NFA with alphabet $\Sigma = \{0, 1\}$

For example in this NFA,

$$\varepsilon\text{-closure}(q_0) = \{q_0, q_1\}$$

$$\varepsilon\text{-closure}(q_1) = \{q_1\}$$

$$\varepsilon\text{-closure}(q_2) = \{q_0, q_1, q_2\}$$

Now for the algorithm to convert an NFA with ε-edges to an equivalent NFA without ε-edges with the same alphabet $\Sigma = \{0, 1\}$ and with the same states, apply the following three steps:-

1. For each node q in NFA, find its ε-closure (say S) and mark all its non-ε-edges as they are. Now, for each node $q' \neq q$ in S, mark all non-ε-edges starting from q' going to q'' as extra edges starting from q going to q''. For eg., for the node q_0, the only distinct node in its ε-closure is q_1 so mark the edges $\{0, 1\}$ from q_1 to q_1, $\{0\}$ from q_1 to q_0 and $\{0\}$ from q_1 to q_2 as extra edges $\{0, 1\}$ going from q_0 to q_1, $\{0\}$ from q_0 to q_0 and $\{0\}$ from q_0 to q_2 respectively (marked in red).

2. Mark all those states as accepting whose ε-closures contain atleast one of the accepting states of the ε-NFA. For eg., here only q_2 has the accepting state q_2 in its ε-closure, so only q_2 is marked as accepting.

3. Starting states in the new non-ε-automaton will be the same as the starting states in the original ε-automaton.

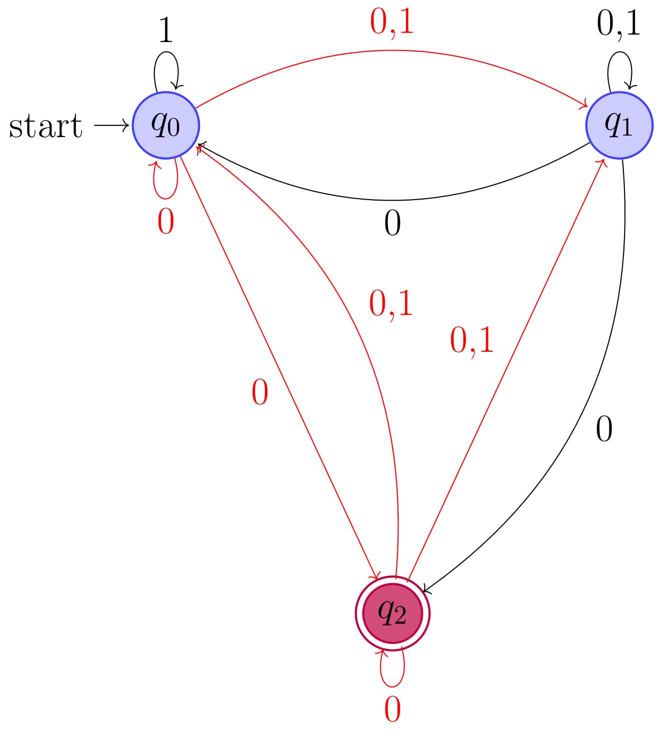


Figure 3.9: Equivalent non-ε-NFA

Subsequently, we'll use "original/ori" for the ε-automaton and "new" for the non-ε-automaton.

- One may ask why the states which are in the ε-closures of the accepting states of the original automaton are not marked as accepting in the non-ε-NFA. The answer is :- every string which reaches an accepting state a can also reach any of the states in ε-closure(a) by taking ε-edges and those strings are indeed in the language of the ε-NFA. But there are many such strings also which can reach at the states in the ε-closure(a) which are not in the language of the ε-NFA. So, if we mark those states as accepting, we are changing the language which is not our intention. For eg., $q_0 \in \varepsilon$-closure(q_2) but if we mark q_0 as accepting in the non-ε-NFA, then the string "1" will also be included in the language of the non-ε-NFA but "1" $\notin L(\varepsilon$-NFA).

3.7.2 Correctness of Algorithm

- If we say $w \in L(ori)$ it means that either w as it is $\in L(ori)$ or w with some ε's inserted $\in L(ori)$. On the other hand, if we say $w \in L(new)$ then it means that w as it is $\in L(new)$.

Now, what we mean by correctness of the algorithm is that the language of the new non-ε-automaton should exactly be equal to the language of the original ε-automaton. So we need to prove that

$$L(new) = L(ori)$$

1. Let's prove $L(new) \subseteq L(ori)$
   Let's take any arbitrary string $w = c_1c_2 \dots c_m \in L(new)$
   (a) Case 1: w is empty
   It means atleast one of the starting states s in the new automaton is an accepting state. Now, according to our algorithm (step 2), all accepting states in the new automaton are either accepting states in the original automaton as well or are those states which have atleast one accepting state in the original automaton in their ε-closures. So, ε-closure(s) definitely contains an accepting state in the original automaton. So, we have a path ε^k for some $k \geq 0$ from s to an accepting state in the original automaton and thus $w \in L(ori)$.
   (b) Case 2: w is non-empty
   It means that we have a sequence of jumps from a starting state s_0 in the new automaton to s_1 upon reading c_1 and then from s_1 to s_2 upon reading c_2 and so on till the state s_m upon reading c_m such that s_m is an accepting state. Now, according to our algorithm (step 1), for s_0 in the original automaton either we have a direct edge $\{c_1\}$ from s_0 to s_1 or we have some $s_0' \in \varepsilon$-closure(s_0) which has the edge $\{c_1\}$ from s_0' to s_1. So, effectively we can reach s_1 from s_0 in the original automaton as well by following a path $\varepsilon^k c_1$ for some $k \geq 0$. Similarly, we can have a path $\varepsilon^{k_1} c_2 \varepsilon^{k_2} c_3 \dots$ for $k_i \geq 0$ till s_m in the original automaton. Now, according to our algorithm (step 2), either s_m is an accepting state in the original automaton as well or there lies some accepting state in the ε-closure(s_m). So, finally, we can have a sequence of $\varepsilon^{k'}$ for some $k' \geq 0$ to reach to an accepting state from s_m in the original automaton and thus, $w \in L(ori)$.
   Hence, proved.

2. Let's prove $L(ori) \subseteq L(new)$
   Let's take any arbitrary string $w = c_1c_2 \dots c_m \in L(ori)$
   (a) Case 1: w is empty
   It means that there exists atleast one starting state s in the original automaton such

that ε-closure(s) contains an accepting state s'. Now, according to our algorithm (step 2), all accepting states in the new automaton are either accepting states in the original automaton as well or are those states which have atleast one accepting state in the original automaton in their ε-closures. So, the starting state s is an accepting state in the new automaton and thus $w \in L(new)$.

(b) Case 2: w is non-empty
It means that we have a path $\varepsilon^{k_0} c_1 \varepsilon^{k_1} c_2 \ldots \varepsilon^{k_{m-1}} c_m \varepsilon^{k_m}$ for all $k_i \geq 0 \; \forall i \in \{0, 1, 2 \ldots, m\}$ from a starting state s_0 in the original automaton to an accepting state s_{m+1} where taking ε^{k_i} from any state x means going to some state $y \in \varepsilon$-closure(x). Here, s_1 is the state reached after reading c_1, s_2 after reading c_2… s_m after reading c_m and s_{m+1} after taking ε^{k_m}. Now, according to our algorithm (step 1), for s_0 in the new automaton, we have a direct edge $\{c_1\}$ from s_0 to s_1. Similarly, we'll have direct edges from s_1 to s_2 and so on. So, finally, we can reach s_m from s_0 in the new automaton. Now, according to our algorithm (step 2), s_m is an accepting state in the new automaton because ε-closure(s_m) contains accepting state s_{m+1} in the original automaton. Thus, $w \in L(new)$.
Hence, proved.

Thus, we've shown that $L(new) = L(ori)$

3.7.3 Intuition of the algorithm

- It should be clear from the correctness proof itself where we constructed paths in the new and original automatons using the step 1 of the algorithm. Also, if the path in the original automaton had trailing ε's we weren't able to reach the same state in the new automaton but had to make the languages of the two automatons exactly same so we concluded step 2 of the algorithm.

3.7.4 Extras

- **Language of a node** q is defined as the set of all those strings $w \in \Sigma^*$ such that upon reading w character by character we can reach q from any of the starting states of the automaton.

- Note that in ε-automaton we could also take ε-edges in between the characters, before the first character as well as after the last character of w and reach q so such strings would also be considered in the language of q.

- And the language of an automaton is defined as the union of the languages of all its accepting nodes. So, we have

$$L(new) = \bigcup_i L(q_i) \, \forall \text{ accepting states } q_i \text{ of the new non-}\varepsilon\text{-automaton}$$
$$L(ori) = \bigcup_j L(q_j) \, \forall \text{ accepting states } q_j \text{ of the original } \varepsilon\text{-automaton}$$

- A lemma: $L(q_i)_{original} \supseteq L(q_i)_{new}$ holds true where q_i is any node of the NFA and $L(q_i)$ is the language of that node.

- Proof: Going by the same idea as we did in Correctness proof part 1, we can prove the lemma. Like if $w \in L(q_i)_{new}$ and w is empty then q_i must be one of the starting states in the new automaton and also in the original automaton so $w \in L(q_i)_{original}$ since starting states are the same in both the automatons. If $w \in L(q_i)_{new}$ and w is non-empty, then we can have a path with some ε's in between the characters of w from some starting state s_0 to q_i in the original automaton and thus $L(q_i)_{original} \supseteq L(q_i)_{new}$ is indeed true.
  Where the $\supset$ sign comes in is the case when the path of w has some trailing ε's. For eg. consider this ε-NFA and its equivalent non-ε-NFA,

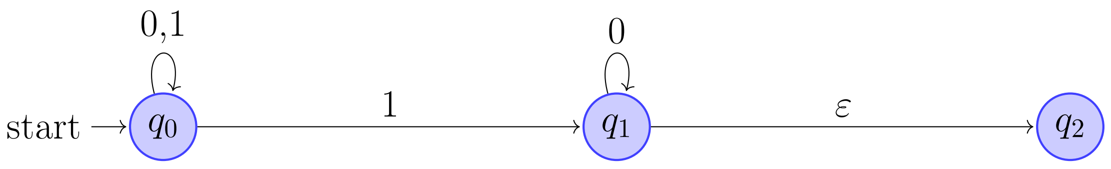


Figure 3.10: ε-NFA

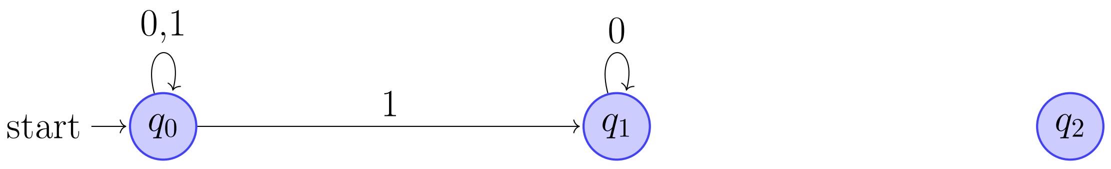


Figure 3.11: Equivalent Non-ε-NFA

$L(q_2)_{original}$ is non-empty and for instance contains the string "1" with the path "1ε" from q_0 to q_1 but $L(q_2)_{new}$ is clearly empty,i.e., the null set ϕ. This happened because the path contained a trailing ε. So, $L(q_2)_{new} \subset L(q_2)_{original}$.

3.7.5 Significance of ε-edges

ε-edges allow us to jump from one part of the NFA to another part of the NFA without consuming any additional character. Thus, if we have to do some operations sequentially we can make our life simple by using ε-edges. For eg., suppose we have to check $w = u.v$ such that u contains an even number of 1's and v contains $1\,mod\,3$ number of 1's. We can accomplish the above task by constructing two NFA's wherein the first NFA will accept all the satisfying u's and the second NFA will accept all the satisfying v's. Now, we'll just join the accepting states of first NFA to the starting states of the second NFA using ε-edges and convert the accepting states of first NFA & the starting states of the second NFA into normal intermediate states. Thus, we will get a single NFA with starting states same as the starting states of the first NFA and accepting states same as the accepting states of the second NFA. Here, ε-edges allowed us to capture the non-determinism of the breaking point between u & v.

* Any automaton containing ε-edges cannot be a DFA because ε-edges bring in uncertainty as we could choose to stay in that state or take the ε-edge.

3.7.6 Examples

- $L = \{x \in \Sigma^* | x = u.v.w, v \in \Sigma^*, u, w \in \Sigma^{+,|u| \leq 2, u=w}\}$ where $\Sigma = \{0, 1, 2 \ldots 9\}$
  For instance, "0000" $\in L$ because either take $u = w = 0, v = 00$ or take $u = w = 00, v = \varepsilon$
  Also, "1234" $\notin L$ because we cannot have any satisfiable u, v, w.
  It's NFA will be a combination of 110 NFA's of the following form :-

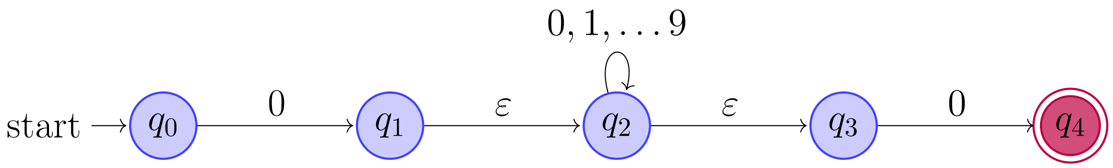


Figure 3.12: ε-NFA for $u = w = 0$

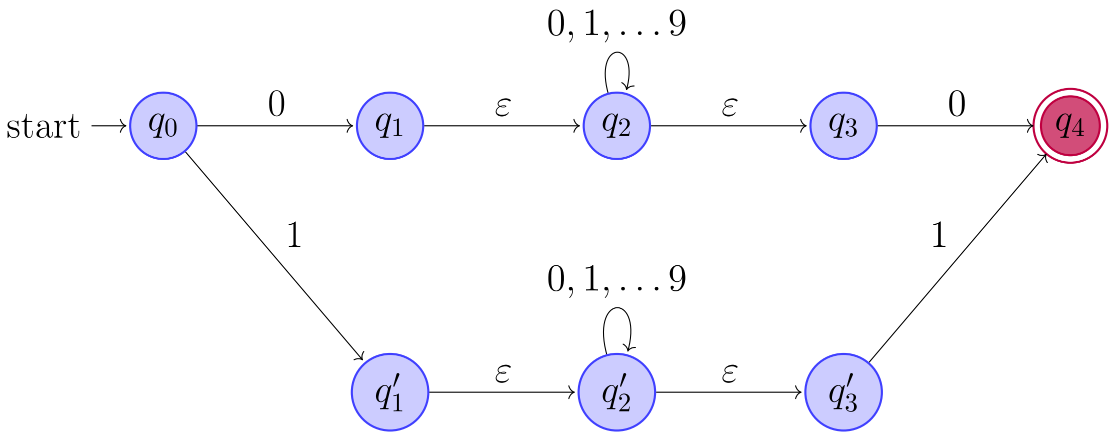


Figure 3.13: ε-NFA for $u = w = 0, 1$

One for each $u = w = 0, 1, 2 \ldots 9$ so 10 here and 100 more for $u = w = 00$ to 99. We can have separate NFA's with different possible q_1', q_3' states. Like instead of 0 put 1 to 9 and 00 to 99 there on the edge from q_0 to q_1' and q_3' to q_4 and in this way we can form the whole NFA just like in figure 6.

- $L = .^*xxx.^*$ where $\Sigma = \{a, b, \ldots z\}$
  Here, we want to locate "xxx" in the text so the following NFA captures this language:-

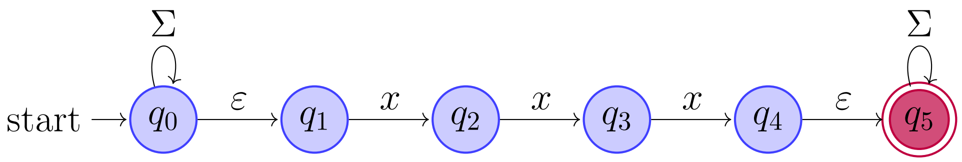


Figure 3.14: ε-NFA for searching "xxx"

- **Takeaway task :** Think about how KMP implicitly constructs NFA for pattern matching.

3.8 Equivalence in Finite Automata

In our exploration of finite automata, we've encountered deterministic finite automata (DFA), non-deterministic finite automata without epsilon transitions (NFA), and non-deterministic finite automata with epsilon transitions (ε-NFA). Surprisingly, despite their apparent differences, these three models are fundamentally equivalent in terms of computational power.
DFA, characterized by their deterministic nature, are particularly useful in scenarios where determinism is crucial, such as in hardware design where predictability is paramount.
ε-NFA, on the other hand, introduce a high level of abstraction by allowing transitions without consuming input symbols, which can simplify the representation of certain languages and aid in conceptual clarity.
NFA without epsilon transitions also provide a similarly high level of abstraction, allowing for flexibility in modeling complex systems and languages.

Chapter 4

Regular Expression

4.1 Introduction

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as $(5 + 3) \times 4$. Similarly, we can use the regular operations to build up expressions describing languages, which are called regular expressions. An example is: $(0 \cup 1)0^*$. The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part is the language $\{0, 1\}$. The part 0^* means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the $\times$ symbol in algebra, the concatenation symbol $\circ$ often is implicit in regular expressions. Thus $(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression. Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as `awk` and `grep` in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions.

4.2 Formal Definition of a Regular Expression

Let R be a regular expression if:

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \cdot R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the Kleene star of the language R_1, respectively.

4.3 Semantics of Regular Language

The semantics of regular language involves understanding the structure and meaning of expressions within the language.

4.3.1 Atomic Expressions

Consider an atomic expression $[a]$, where a represents a single letter. In this context, $[a]$ denotes the language consisting of only the string a. However, it's important to note that the a within $[a]$ is not considered a letter of the alphabet; rather, it signifies a language comprising a single letter string.

4.3.2 Union Operation

For expressions e_1 and e_2, $[e_1 + e_2]$ represents the union of the languages denoted by $[e_1]$ and $[e_2]$. In simpler terms, $[e_1 + e_2]$ encompasses all strings that belong to either $[e_1]$ or $[e_2]$.

4.3.3 Concatenation Operation

When considering e_1 concatenated with e_2, denoted as $[e_1.e_2]$, it signifies the concatenation of languages represented by $[e_1]$ and $[e_2]$. This operation results in a language consisting of all possible combinations of strings where the first part belongs to $[e_1]$ and the second part belongs to $[e_2]$.

4.3.4 Example

Let's illustrate with an example: $(ab) + a$. This expression represents the union of the language containing the string ab and the language containing the string a, resulting in $\{ab, a\}$.

4.3.5 Order of Precedence

In the semantics of regular language, the order of precedence for operations is as follows: $*$ (Kleene star) $>$ $\cdot$ (concatenation) $>$ $+$ (union).

4.3.6 Kleene Star

The Kleene star operation $[e_1^*]$ denotes the union of zero or more concatenations of $[e_1]$. In other words, $[e_1^*]$ encompasses all possible strings that can be formed by concatenating any number of strings from $[e_1]$.

4.3.7 Example

For an expression $e_1 = a + b$, the language denoted by $[e_1]$ is $\{a, b\}$. Therefore, $[(a + b)^*]$ represents the set of all possible strings comprising as and bs, including the empty string ε, a, b, ab, ba, aa, bb, and so on.

4.3.8 Further Examples

- $[a^* + b^*] = \{u \in \Sigma^* \mid u = a^n \text{ or } u = b^n \text{ for some } n \geq 0\}$
- $[a^* \cdot b^*] = \{u \in \Sigma^* \mid u = a^n b^m \text{ and } n \geq 0 \text{ and } m \geq 0\}$
- $[(a^* \cdot b^*)^*]$ represents the set of all strings containing any number of occurrences of strings composed of as followed by bs.
- $0^*10^* = \{w \mid w \text{ contains a single } 1\}$
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$
- $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string 001 as a substring}\}$
- $1^*(01^+)^* = \{w \mid \text{every 0 in w is followed by at least one } 1\}$
- $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$
- $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of } 3\}$
- $01 \cup 10 = \{01, 10\}$
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup \{0, 1\} = \{w \mid w \text{ starts and ends with the same symbol}\}$
- $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$
- $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$
- $1^*\emptyset = \emptyset$
- $\emptyset^* = \{\varepsilon\}$ The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Chapter 5

DFA Minimisation

5.1 Minimum States in a DFA

So, far we have dealt with DFA, NFA without ϵ, NFA with ϵ and Regular Expressions. We have also seen that all of them are equivalent and inter-convertible and represent regular languages.
Consider the language which consists of all strings which are terminated by one. The regular expression for this will be: $(0+1)^*1$. Here is a 2-state DFA for the same language:

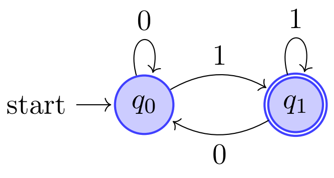


Figure 5.1: 2-state DFA for the language

The above DFA has two states q_0 and q_1. q_0 is reached when the last seen letter was 0 (also at the start). q_1 is reached when the last seen letter was 1 (also, it is an accepting state).
We can even construct a 4-state DFA for the same language. Here is an example:

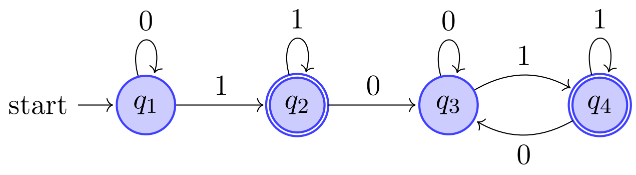


Figure 5.2: 4-state DFA for the language

The above DFA has four states q_1, q_2, q_3, q_4.

- q_1: Last letter 0 and no 1s so far
- q_2: Last letter 1 and no 101 seen so far
- q_3: Last letter 0 and more than one 1s seen so far

- q_4: Last letter 1

One can construct many more 4-state DFAs for the same language. Above is another example.

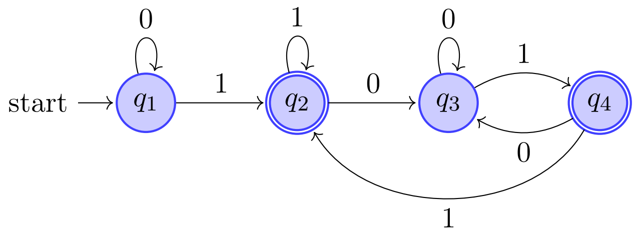


Figure 5.3: Another 4-state DFA for the same language

A natural question which comes to our mind is that can we construct another 2-state DFA for this language. One can find through trial and error, that this is not possible.
Is one state DFA possible for this language? Let us assume that this is possible. Two cases arise. If that state is accepting, then it will also accept ϵ, which is not possible. If that state is not accepting, then the language will be empty. We have arrived at a contradiction. Hence, one state DFA is not possible for this language.
Therefore, for this language, the minimum states in any DFA can be 2. We also observe that number of such 2-state DFAs is 1. So, we will now claim that for every language, the number of minimal DFAs is 1 and try to prove this. Before we prove this, we will introduce the notion of indistinguishability.

5.2 Indistinguishability

Two states of a DFA q_i and q_j are considered indistinguishable, if $\forall w \in \Sigma^*$, we start with q_i, process w and reach q_i' and we start with q_j, process w and reach q_j', then either $q_i' \in F$ and $q_j' \in F$ or $q_i' \notin F$ and $q_j' \notin F$, where F is the set of all final states of the DFA. So, we are basically changing the start states and checking whether we reach the same type of states or not through the same string.
This relation is denoted by $\equiv$. It has the following properties:

- It is **reflexive**. It is clear to see that every state is indistinguishable to itself, as it will reach a particular state on seeing w. (Due to the nature of a DFA)

- Also, it is clear to see that this relation is **symmetric**.

- This relation is also **transitive**. We can prove this by contradiction. Let us assume that $(q_i \equiv q_j) \wedge (q_j \equiv q_k)$ but $q_i \not\equiv q_k$. Then $\exists w$ such that $q_i' \in F$ and $q_k' \notin F$, where q_i' and q_k' are the states we reach from q_i and q_k respectively on seeing w. From the equivalence of q_i and q_j, we have $q_j' \in F$ but from the equivalence of q_j and q_k, we have $q_j' \notin F$, where q_j' is the state that we reach from q_j on seeing w. Hence, we have arrived at a contradiction. Therefore, this relation is transitive.

- A relation which is reflexive, symmetric and transitive, is **equivalent**. Thus the states of the DFA, on which this relation is defined can be partitioned into equivalence classes.

In the above example (Figure: 5.2), q_1 and q_3 belong to the same equivalence class, and q_2 and q_4 also belong to another equivalence class. Let us now try to construct a 2-state DFA from the above 4-state DFA example (Figure: 5.2). We will choose, one element each from both of the equivalence classes. Let us take q_1 from the first class and q_4 from the second class.

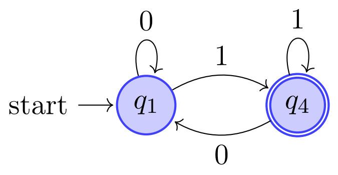


Figure 5.4: 2-state DFA constructed from the 4-state DFA

From q_1, if we see a 0, we land at q_1 itself. If we see a 1, we would have landed at q_2, but since q_2 and q_4 are equivalent, we replace q_2 by q_4. Similarly, from q_4, if we see a 1, we remain at q_4, but if we see a 0, we would have landed at q_3, but since q_1 and q_3 are equivalent, we replace q_3 by q_1. Also, q_2 and q_4 both were acceptable earlier, now q_4 is acceptable.

So, through these equivalence classes, we have minimized our 4-state DFA into a 2-state DFA, and also this 2-state DFA is structurally the same as the previous one (Figure: 5.1), thus again making us think that the claim that there is a single minimal DFA for every language might be correct.

Another interesting thing we observe is that an accepting state cannot be indistinguishable from a non-accepting state. We can take $w = \epsilon$, and observe that the states we reach from this pair of states do not satisfy the definition of indistinguishability relation. However, an initial state and a non-initial state can belong to the same equivalence class. (For example, above q_1 and q_3 belonged to the same class.)

Now, several important questions arise. How can we find the equivalence classes of this relation? When we will come to know that we cannot compress our DFA further (by compress, we mean reducing the number of states of the DFA)? How can we prove our claim that the minimal DFA will be unique?

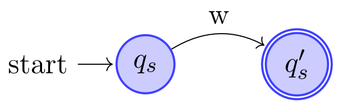
We will try to answer all these questions subsequently. Let us start with the easiest one.

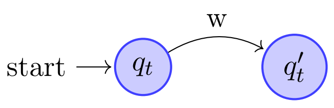
5.3 Equivalence classes of Indistinguishability relation

Now, we will develop an algorithm to find the equivalence classes of this relation.

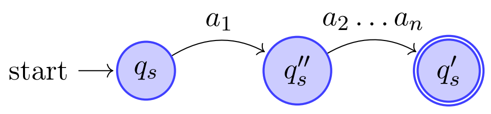
Firstly, we should keep in mind our previous observation that an accepting state cannot be indistinguishable from a non-accepting state. That is $q_i \not\equiv q_j \ \forall q_i \in F$ and $q_j \in (Q \backslash F)$, where Q is the set of all states of the DFA.

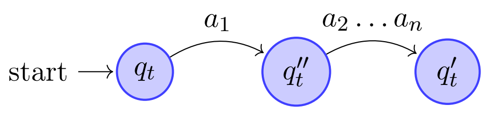
Suppose, we find two states q_s and q_t which are distinguishable. Thus there exists a string w such that q_s leads to an accepting state but q_t leads to a non-accepting state.



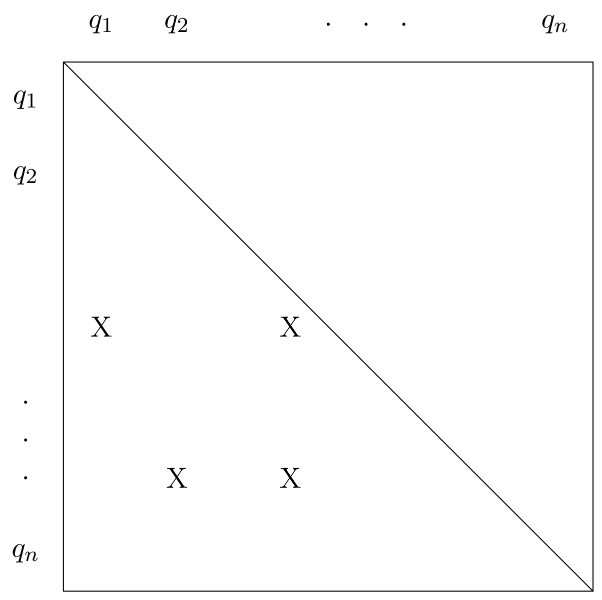


Now, w can be decomposed into $a_1a_2 \ldots a_n$, where $|w| = n$.

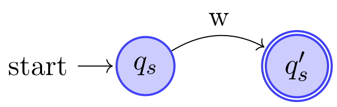


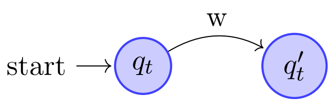


From, the above figures, one can observe that q_s'' and q_t'' are also distinguishable, where $w' = a_2 \dots a_n$ is the string which is making them distinguishable.
Thus, we can observe that for all states q_i and q_j such that $q_i \not\equiv q_j$ and for all $a \in \Sigma$, such that q_s on seeing a lands at q_i and q_t on seeing a lands at q_j then q_s will be distinguishable with q_t, where q_s and q_t are two states of the DFA.(We are basically extending $w' = a + w$.)
Therefore using a distinguishable pair, we have found another one. This is the basis of our algorithm. We initialize our set with all the pairs, where one state belongs to the set of accepting states and other state does not belong to the set of non accepting states. And then through the above step, we keep on increasing the size of this set. (Note that this algorithm is not exponential, because there are only $\binom{n}{2}$ pairs possible, and we do need to check an already visited pair.)

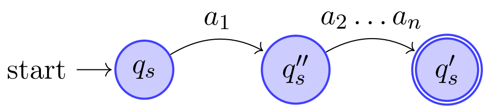


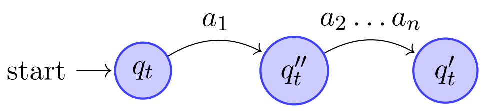
We stop the process, when no more crosses can be inserted.
Now, it is quite natural for us to ask the question that whether our algorithm is correct or not i.e. can we still find a pair of distinguishable states, which are not detected even after our algorithm finishes?
Proof: Let us assume that there are two states q_s and q_t which are distinguishable but not recognized by our algorithm. By definition, there exists a string w such that q_s leads to an accepting state on seeing w and q_t reaches a non-accepting state.



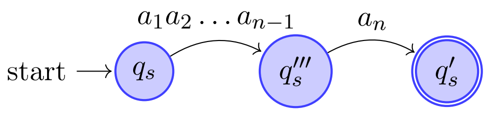


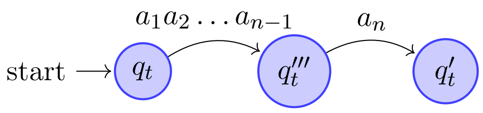
Now w can be written as $a_1a_2\dots a_n$, where $|w| = n$. And also assume that we reach q''_s and q''_t on seeing a_1 from q_s and q_t respectively. It is clear that, if our algorithm has not detected q_s and q_t as distinguishable, then it would not even have detected q''_s and q''_t as a distinguishable pair. (Because, if it would have done, then the next step would have been to make q_s and q_t as distinguishable.)





Now, we will inductively move forward our algorithm. Thus q'''_s and q'''_t would not also be detected as distinguishable after our algorithm finishes. But clearly, this is not possible, because our algorithm initializes the set of pairs of {accepting, non-accepting} states and then it's first step is to move backwards. So, it would have marked q'''_s and q'''_t as distinguishable in the first step itself.





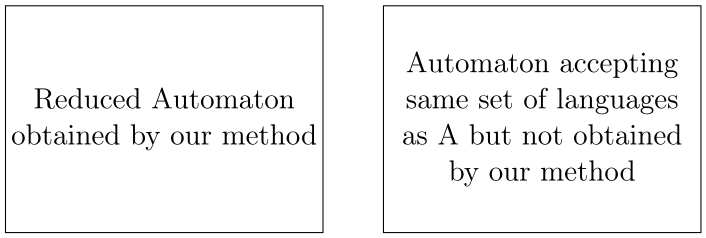
We have achieved contradiction. Therefore, we can safely conclude that our algorithm terminates and is also correct.
Now, once our algorithm detects all pairs of distinguishable states, we can choose equivalence classes of the indistinguishability relation. (Then, we can choose one representative from each class and move forward with our proof of existence of a unique minimal DFA.)
It is worth noting that distinguishability is not an equivalent relation. It is not even reflexive. It is also not transitive. But, it is indeed symmetric. And also, it proved out to be very useful for finding these equivalence classes.

5.4 Further Analysis

We have developed an algorithm which can shrink the size of a given DFA. But is that the best we can do? Can the size of the DFA be further reduced? Are there many DFAs which are minimal or just one unique? The following sections will try to answer these questions.
Note: We are assuming that there is no redundant nodes in final DFA obtained, i.e. those nodes which can not be reached from starting state by any string. If there is one such, we can always remove it to get an equivalent DFA.

5.4.1 Optimality of Acquired DFA

Let's say we have an automaton X, which is the reduced automaton obtained by picking one state from each of the equivalence classes of the set of states, and Y is an automaton obtained by some other method for the same language.


Reduced Automaton obtained by our method
Automaton accepting same set of languages as A but not obtained by our method

Automaton X Automaton Y

We wish to show that the number of states of Y is at least as many as X.
Let's define a new term "**Language of State**". Given a DFA A and a state s, we define L_s^A as the set of all finite strings which will end in any of the accepting state if we start from s. Readers should be able to see that when we say that two states are distinguishable, we mean that there Language is different. Similarly, when we say that the two states are indistinguishable, we are actually asserting that the language of the two states is same.

Let say we started with the automaton D. After termination the resultant automaton was A. Let S_A represent the set of distinct such languages of nodes of A. Since A has all states indistinguishable from every other state, the $|S_A|$ = no of states.
Claim: Given any DFA B, equivalent to A,

$$\forall L \in S_A(\exists S(L_S^B \equiv L)), \text{ where S represent a state of B}$$

Proof: If L $\in$S$_A$, then by definetion it must be the language of some node in A, say s. Since all the nodes of A are ir-redundant, so let's say string w is one such string through which we can reach this node starting from the starting state of A. Now run the same string w on the automaton B. Say we reach the state S. Then $L \equiv L_s^A \equiv L_S^B$, because if say string x is present in former and not in latter then, string $w.x$ will be an accepting string in A and not in B, and vice-versa. Thus $L_S^B \equiv L$.
For every language in S_A, we must have at least one node in B. Since one node has a specific language, this gives us a lower bound to number of nodes, $|S_A|$.
A achives the lower bound of states, hence it is the(?) minimum state DFA which represent the same regular language represented by D.

5.4.2 Uniqueness of Acquired DFA

So far we have discussed about proving the optimality of the automaton A, which is the output of our algorithm. We defined an important notion of "Language of State" and proved a very important result which can be summarized as follow **Given two DFAs A and B where both represent some particular regular language, then for any string w, the states reached in A and B by reading it will have same language,i.e.** $L_{s_A}^A \equiv L_{s_B}^B$. Now we will try to answer that if there are mutiple DFAs which are optimal or just one.
Let's consider two automatons, one is our DFA A which has been proved to be optimal and another DFA C which is a equivalent DFA and is also optimal(given). We will show that C has to be isomorphic to A.

Since C is an optimal DFA, it can not be further shrunk. That implies two things: No redundant nodes, no pair of nodes is indistinguishable. That means that the language of the nodes of C is distinct and unique. Since A and C are both optimal, both must have same number of states, and all the nodes of both of them are reachable, and all the nodes of both of them have unique languages(unique over the domain of single DFA not collectively).
Consider any node s of A, take a string w which take us to it, run it over B, say we reach the node S, then s and S have same language. Since the nodes of B are distinguishable, so S does not depend on w.

This can be used to define an injection from the nodes of A to the nodes of B. It is injective function because of distinguishability of the nodes of A as well as of B. The function is also **bijective** because the number of nodes of A and B are both finite and equal, making it both injective and surjective, thus bijective.

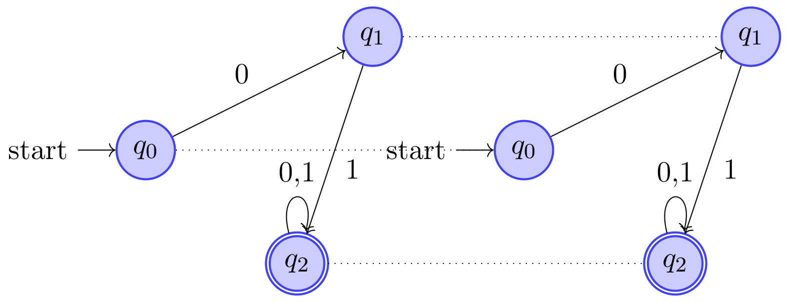


Figure 5.9: Two DFAs with corresponding states connected by dotted lines.

Claim:
As per this function, starting state of A will be mapped to starting stata of B. . This is because the languages of these two states is essentially the language of there respective DFA which is same(given).
Claim:
An accepting state will be mapped to an accepting state of B. . This is because any state which has ϵ in it's language is an accepting state by the definition of "language of state", similarly an accepting state will have ϵ in it's language. So an accepting state of A mapped to whatever state of B, that state must have ϵ in it's language making it an acceptable state of B.
Claim:
Consider Two states of A, S_1^A and S_2^A. Suppose that they are mapped to S_1^B and S_2^B respectively. Then if there is an α labelled edge going from S_1^A to S_2^A, there must be an α labelled edge going from S_1^B to S_2^B also. Consider any string w that takes us to S_1^A, we can see that $w.\alpha$ will take us to S_2^A. Now since S_1^A is mapped to S_2^A and S_1^B is mapped to S_2^B we can say that "any" string that takes us to one in A will take us to the image of it when run on B(Why? proved above:)). Thus w takes us to S_1^B and $w.\alpha$ takes us to S_2^B. Because a DFA is deterministic, there should be α labelled edge going from S_1^B to S_2^B.(Food for thought: Is mentioning determinism important here?)
Isomorphism of labelled graphs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving.
Isomorphism of DFAs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving, where im-

age of starting state is starting state and image of a accepting state is an accepting state.
All these three claims shows that A and B are same automatons.

This tool can be used to check wether two different regular expressions represent the same language or different languages. Just convert them into DFAs, and then to the minimal DFAs. Check for the isomorphity of the two DFAs if they are isomorphic, then the regular expressions are equivalent otherwise not. (Isomorphism $\Longleftrightarrow$ equivalence)

5.5 From states to words

Till now we were talking about languages of states and equivalence of two states. Now we will extend this notion for words.

5.5.1 Language of word

Consider a language L. we define Language of word w, $[w]$ as set of all strings x such that $w.x \in L$. Now we will define a relation " $\tilde{}_L$ ". If languages of two words *w1* and *w2* is same for L, then we say that w1 is related to w2. This is a reflexive, symmetric and transitive relation. (Basically, if set 1 and set 2 are same and then it is also true for the other way round that is set 2 and set 1 are same. If set 1 and set 2 are same and if set 2 and set 3 are same then set 1 and set 3 will also be same.) Thus we have defined an equivalence relation over words, where the equivalence classes depends on language L.

5.5.2 Relation between states of minimal DFA and equivalence classes for $\tilde{}_L$

If a string w ends up in a state of DFA, say s, then the language of the state is equivalent to $[w]_L$. This can be easily proved by using definition of the "language of state" and "language of word". If a string $w.x \in$ L, then if you run the $w.x$ on DFA, you first reach that state using w then x takes to some accepting state. Hence x $\in L_s^{DFA}$. Also if x $\in L_s^{DFA}$, then essentially $w.x$ will end in a accepting state because w ends in state s.
A particular equivalence class represent a particular language and a state represent a particular language. Since for a minimal DFA, all the states represent distinct language, we can define a bijection from equivalence classes to the state of minimal DFA, where a equivalence class is mapped to that state which represent the same language as that of any string in that equivalence class.

5.6 Setting up the Parallel

We defined the Nerode equivalence relation on a language L over the alphabet Σ. This relation states that $\forall w_1, w_2 \in \Sigma^*$, $w_1 \sim_L w_2$ iff $\forall x \in \Sigma^*$, $(w_1 \cdot x \in L \iff w_2 \cdot x \in L)$

If the language L is regular, then a minimized DFA $A = (Q, \Sigma, \delta, q0, F)$ can also be defined for the language L.

For this language L, w_1 and w_2 are two words in Σ^* such that $w_1 \sim_L w_2$. Let q_i and q_j be two states $\in Q$ such that

$$q_i \rightarrow \text{state reached in A after reading } w_1$$
$$q_j \rightarrow \text{state reached in A after reading } w_2$$

Since $w_1 \sim_L w_2$, it follows that for all $x \in \Sigma^*$, the state reached in DFA A after reading $w_1 \cdot x$ and the state reached in DFA A after reading $w_2 \cdot x$ will either both belong to F or both belong to $Q \setminus F$. Otherwise, one word would be accepted by L while the other would not, leading to a contradiction in the definition of the equivalence relation.

Therefore, by the definition of indistinguishability, states q_i and q_j can be concluded to be indistinguishable.However, in a minimized DFA, all distinct pairs of states are distinguishable. Consequently, the only states in A that can be indistinguishable are those where the state is compared with itself.

This demonstrates that if two words/strings belong to the same equivalence class with respect to L, then both strings will end up in the same state $q \in Q$ in the minimized DFA.

Till now, we have defined two equivalence relations $\sim_L$ over the language L and $\equiv$ over the states of a DFA characterizing a language L. We aim to define a relation between the number of equivalence classes of both these relations.

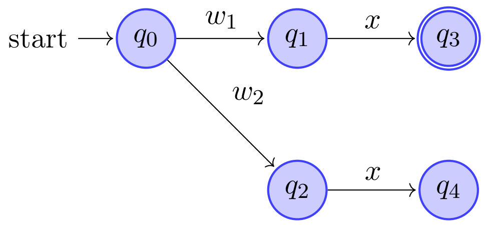
5.6.1 Can $| \sim_L | > | \equiv |$?

NO. We will in fact prove below that $| \sim_L | \leq | \equiv |$

Let there be strings w_1,w_2 s.t.

$$w_1 \in i^{th} \text{ equivalence class of } \sim_L$$
$$w_2 \in j^{th} \text{ equivalence class of } \sim_L$$
$$\text{where } i \neq j$$

Since they belong to different equivalence classes, we can say $\exists$ x $\in \Sigma^*$ s.t. $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ (or can be vice-versa, does not matter).



- Here, q_1 represents the equivalence class of w1 while q_2 represents the equivalence class of w2.
- By the definition of indistinguishability, q1 and q2 are distinguishable states as there exists a letter in the alphabet which leads them to another pair of distinguishable states.

- Doing this for every pair of equivalence classes, we find that the states representing the distinct equivalence classes are all distinguishable from each other leading us to the following relation.

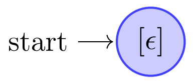
$$\mid \sim_L \mid \leq \mid \equiv \mid$$

.i.e. the number of indistinguishability equivalence classes is atleast the the number of Nerode equivalence classes

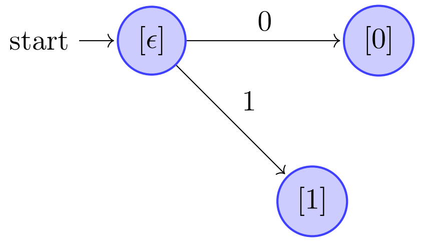
5.6.2 Can$\mid \sim_L \mid < \mid \equiv \mid$?

In order to inspect this , we are going to construct a DFA using $\sim_L \subseteq \Sigma^* \times \Sigma^*$ relation which will then accepts the language L. Let here, $\Sigma = 0, 1$

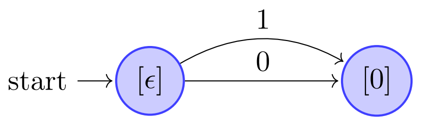
- A state denoting $[\epsilon]$ is made and is also denoted as the start state.



- Then we pick a letter from the alphabet and look the transitions from each of the existing states. If the next word already lies in the equivalence class of one of the existing states, we draw the arrow representing the particular transition otherwise a new state is made representing the equivalence class of the newly made word.

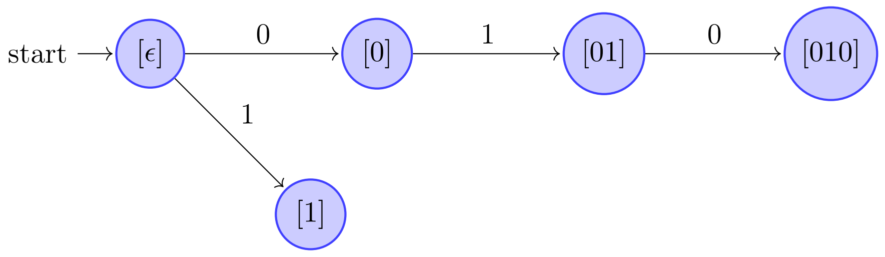


Here, 1 does not lie in the equivalence class of 0, i.e., [0].



If 1 lies in [0], then the automata will look like this.

- By repeatedly applying point 2, the automata can keep on expanding like as below



- However, this construction is guaranteed to converge because we have already proved that $\mid \sim_L \mid \leq \mid \equiv \mid$. It is known that the cardinality of equivalence classes of the indistinguishability relation is equal to the number of states in the minimal DFA representing the language L.Since the number of such states is finite, it follows that $\mid \equiv \mid$ is also finite.Given that $\mid \sim_L \mid \leq \mid \equiv \mid$, it can be deduced that $\mid \sim_L \mid$ is finite. Consequently, this guarantees the convergence of the algorithm in question.[If the above algorithm does not converge, that means new states will keep on forming contradicting the $\mid \sim_L \mid \leq \mid \equiv \mid$ identity.]

- Finally, all those states whose equivalence classes have words that are accepted by the language L are marked as accepting states.

Hence, the aforementioned algorithm successfully allows us to construct a finite state automata(DFA) which accepts only the words that are accepted by the language L. Note that here the number of states in the new automata is equal to $\sim_L$.
Now, let us assume that $| \sim_L | < | \equiv |$ holds. This implies that our newly constructed automata is the minimized DFA for the language L.
However, in the last lecture we had proved that the DFA constructed by $\equiv$ relation is minimized one and is unique. Hence, it leads to a contradiction,making our assumption wrong.

Thus proved that

$$| \sim_L |= | \equiv |$$

Note: The Nerode equivalence relation,unlike the indistinguishability relation,can be defined for any language $L \subseteq \Sigma^*$, irrespective of the fact that whether the language is regular or not.

5.7 Myhill-Nerode Theorem

L is regular if and only if $\sim_L$ has a finite number of equivalence classes.

This theorem provides an exact characterization of a regular language, unlike the Pumping Lemma. While the Pumping Lemma does not guarantee that L is regular if it holds, here, if $\sim_L$ has a finite number of equivalence classes, the language L must be regular. The proof of this theorem can be found here.

Chapter 6

Pushdown Automata

6.1 Pushdown Automata for non-regular Languages

Thus far, the finite automata we have studied cannot be used to represent non-regular languages. One such example of a non-regular language is the language of balanced parentheses.Let it be L.

> **Proving why L is not a regular language** *It is known that intersection a language L with any regular language preserves the regularity of the original language.i.e. the new language formed after intersection will hav ethe same regularity as the original language L.*
> *Therefore, considering* $L \cap (^*)^*$ *results in* $(^n)^n$ *which by applying the Pumping Lemma can be shown that that it is not a regular language, it can be thus concluded that the language of balanced parenthesis is also not a regular language.*

So if we can equip the finite state automata with more structures like a stack, it will be abl to accept the non-regular languages too.

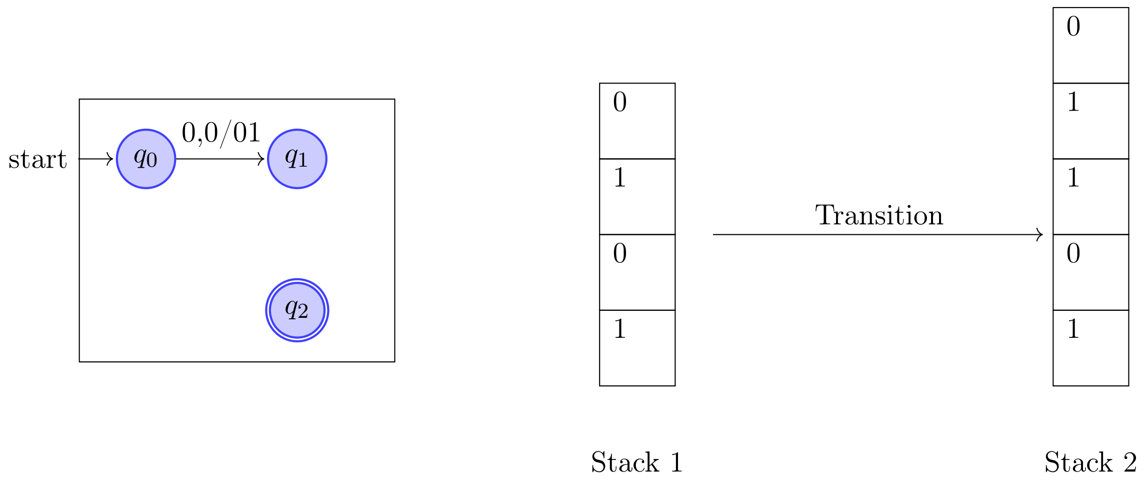


Figure 6.1: Automaton and Stacks

6.1.1 Formal Definition of Pushdown Automata

The formal notation P of PDA(Push Down Automaton) is a follows:

$$P \ = \ (Q, \ \sum, \ \Gamma, \ \delta, \ q_0, \ Z_0, \ , \ F)$$

Here the symbols in the R.H.S represent the following:

Q: A finite set of states, like the states of a finite automaton.

$\sum$: A finite set of input symbols, also analogous to the corresponding component of finite automaton.

Γ: A finite *stack alphabet*, which has not finite-automaton analog, is the set of symbols that we are allowed to push onto the stack.

δ: The transition function. Its structure is described above. Its formal definition is as follows

$$\delta(q, a, X) = (p, \gamma)$$

Here q is a state in Q, a is the input symbol(can beϵ) synonymous to the a in structure of transition, X is the symbol at the top of the stack synonymous to b. And output is the new state p, to which the automaton went to and γ is the symbol string that replaces X at the top of stack, synonymous to z.

q_0: The start state. The PDA is in this state before making any transition.

Z_0: The start symbol. Initially, the stack consists of one instance of this symbol only and nothing else.

F: The set of accepting states, or final states.

6.1.2 Constructing an Example

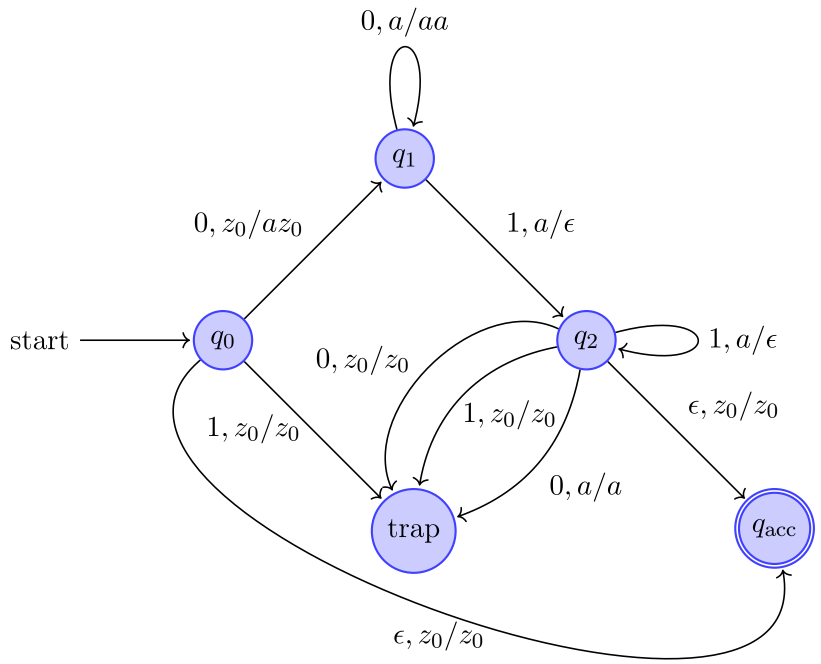
How to construct this automata?

- Same as finite automata, we first construct the start state and start reading the different letters from the alphabet Σ
- Now,Whenever a move is made on the automata by reading a letter, we also inspect the top of the stack.Based on the transition function which depends on both the alphabet and the state of the stack, the neccessary transition is made on the automata and on the stack.Note that there is always a special stack letter which is pushed onto the stack in the initial state, which on being read signifies an empty stack.
- For example, earlier in figure 1, on reading the input 0 and the top of the stack also being 0, a transition to q1 state is made and 01 is pushed onto the stack after popping off the earlier 0.If we do not want to push any symbols onto the stack during the transition, then we can replace the top of the stack with ϵ .
- In one transition, only one stack letter can be popped off the stack.

The Pushdown Automaton can be used to recognize regular languages as well. However, in doing so, transitions on the automaton may involve pushing and popping the same symbol off the stack, rendering the stack somewhat redundant.
Now, let's draw a PDA for a non regular Language. Let's take for example, a language L such that

$$L \equiv \{0^n1^n |\ n \geq 0\}$$

It can also be written as $(^n)^n$, a special case of balanced parentheses problem, where (stands for 0, and) stands for 1. It can be proved through Pumping Lemma that this language is not regular. Also it can be said that since a normal DFA(Finite State Automaton) cannot store the number of (brackets appearing before the) brackets in finite number of states, it cannot be represented by DFA.
So to make its PDA, here is our intuition:- Let the set of stack symbols, Γ be a, z_0. We start the stack with z_0 in it, push an extra a on top of it whenever it sees a 0, and pops a, pushes nothing if it sees 1. If the number of 1's is equal to the number of 0's, it should see z_0 on an input of ϵ, just like at the start state, and should go to an accepting state. All other edge cases have been handled in the PDA itself. The PDA is given below:


$0, a/aa$
q_1
$0, z_0/az_0$
$1, a/\epsilon$
start
q_0
q_2
$1, a/\epsilon$
$0, z_0/z_0$
$1, z_0/z_0$
$1, z_0/z_0$
$\epsilon, z_0/z_0$
$0, a/a$
trap
q_{acc}
$\epsilon, z_0/z_0$