oof Rules

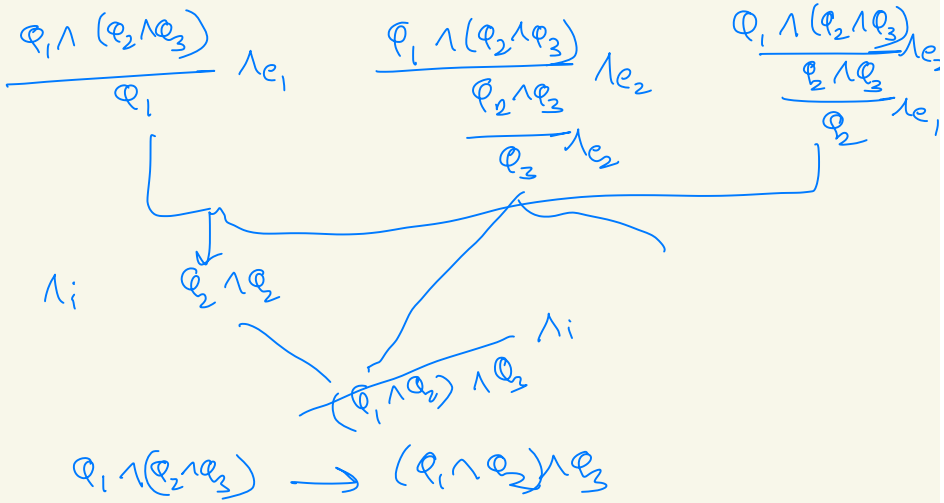
$\wedge_i$ $\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$ $\qquad$ $\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1}$ $\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2}$ $\wedge_e$

$\vee_i$ $\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2}$ $\dfrac{\varphi_2}{\varphi_1 \vee \varphi_2}$ $\qquad$ $\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \to \varphi_3 \quad \varphi_2 \to \varphi_3}{\varphi_3}$ $\vee_e$

$\to_i$ $\dfrac{\begin{array}{|c|}\hline \varphi_1 \\ \vdots \\ \varphi_2 \\ \hline \end{array}}{\varphi_1 \to \varphi_2}$ $\qquad$ $\dfrac{\varphi_1 \to \varphi_2 \quad \varphi_1}{\varphi_2}$ $\to_e$ Modus Ponens

$\neg_i$ $\dfrac{\begin{array}{|c|}\hline \varphi \\ \vdots \\ \bot \\ \hline \end{array}}{\neg\varphi}$ $\qquad$ $\dfrac{\varphi \quad \neg\varphi}{\bot}$ $\neg_e$ $\quad$ $\bot_i$

$\neg\neg_i$ $\dfrac{\varphi}{\neg\neg\varphi}$ $\qquad$ $\dfrac{\neg\neg\varphi}{\varphi}$ $\neg\neg_e$

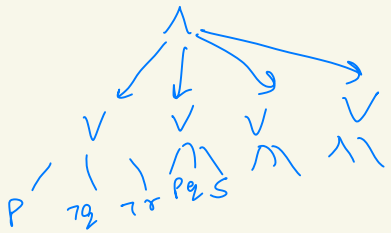


What if a formula is neither valid nor contradiction?
Can we reason directly about satisfiability

φ is valid iff $\neg\varphi$ is not-sat.

Negation Normal form (NNF) → negation is only with literals (not with formulas)
(Use Demorgan's law)

Conjunctive Normal form (CNF)

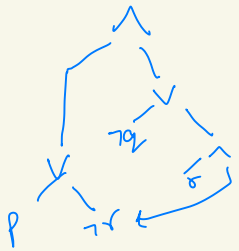


Normal forms

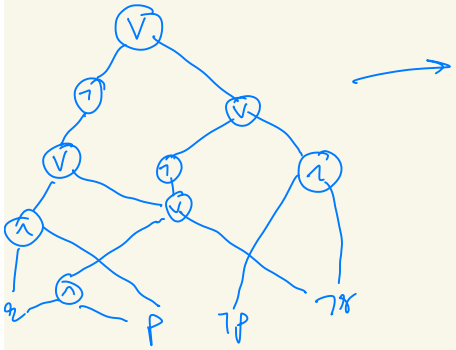
Negation Normal form (NNF) : $(p \vee \neg r) \wedge (\neg q \vee (r \wedge (p \vee \neg r)))$

True representation

DAG representation:



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



→ Create a copy with inverted operators and literals (¬ removed). Then start joining

Literal : variable or its complement

Clause : Disjunction of literals

Cube : Conjunction of literals

Conjunctive Normal form (CNF) : Conjunction of clauses
Product of sums

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

Disjunctive Normal form (DNF) : Disjunction of clauses
(Sum of products)

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

$p, \neg p, q, \neg r$

$(p \vee q \vee \neg r)$, ~~$(q \vee \neg q \vee r)$~~ ✓

~~$(p \wedge q \wedge \neg r)$~~ ~~$(q \wedge \neg q \wedge \neg r)$~~ ✗

Validity checking → DNF

Satisfiability checking → CNF

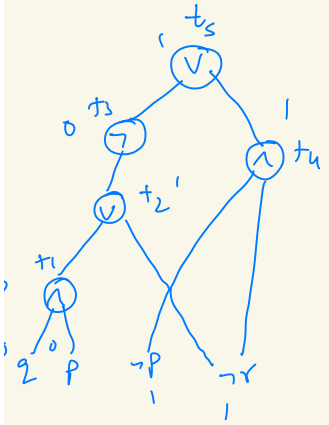
$\varphi(p,q,r)$ → Tseitin Encoding → $\varphi'(p,q,r,t_1,\ldots,t_n)$

Equisatisfiable

not necessarily semantically equivalent

Size of φ' linear in size of φ.

Tseitin Encoding



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

$p=q=r=0$

Unique mapping → $p=q=r=0$
$t_1=0,\ t_2=1,\ t_3=0, t_4=1$
$t_5=1$

Bijection between satisfiability assignments of φ and φ'

(Stronger than equisatisfiability)

towards checking satisfiability of CNF

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

Horn Clause : At most one unnegated var. per clause

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

Horn formula
$x_1 = x_5 = x_3 = 1$
$\rightarrow x_2 = x_4 = 0$

n vars, k clauses

avis Putnam Logemann Loveland (DPLL)

$c_1 \wedge c_2 \wedge \ldots \wedge c_k$

SAT (φ, PA) returns (status, assignment)
status: sat / unsat
assignment: $x_1 = 0, x_2 = 1, x_3 = 0$

0. If $\varphi = T$ return (sat, PA)
   If $\varphi = \bot$ return (unsat, PA)

1. If c_i is a unit clause (clause with single literal l)
   return SAT ($\varphi[l=1]$, PA $\cup \{l=1\}$)

unit propagation — simplify φ after setting $l=1$

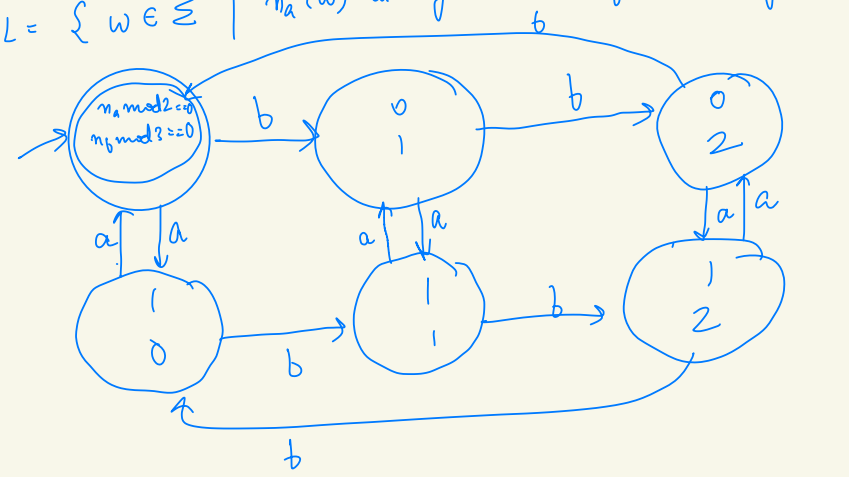
If a literal l does not appear negated in any clause
return SAT ($\varphi[l=1]$, PA $\cup \{l=1\}$)

pure literal elimination

$x :=$ choose_variable (φ) — decision
$v :=$ choose_value () $\ldots \in \{0,1\}$
if SAT ($\varphi[x=v]$, PA $\cup \{x=v\}$).status = sat
return (sat, PA $\cup \{x=v\}$)
else if SAT ($\varphi[x=1-v]$, PA $\cup \{x=1-v\}$).status = sat
return (sat, PA $\cup \{x=1-v\}$)
else return (unsat, PA) — backtrack

Finite Automata

$\Sigma = \{a,b\}$
$L = \{ w \in \Sigma^* \mid n_a(w)$ div by 2 and $n_b(w)$ div. by 3 $\}$



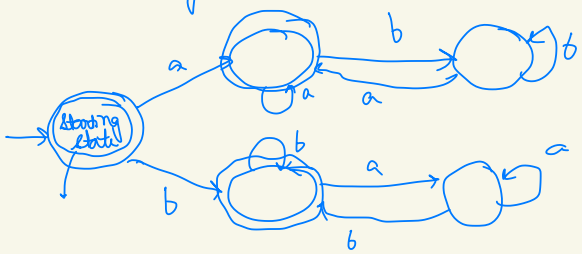
$\Sigma = \{a,b\}$
$L = \{ w \in \Sigma^* \mid n_{ab}(w) = n_{ba}(w) \}$

$w = abaabab$
$n_{ab}(w) = 3$
$n_{ba}(w) = 2$
$\} \; w \notin L$

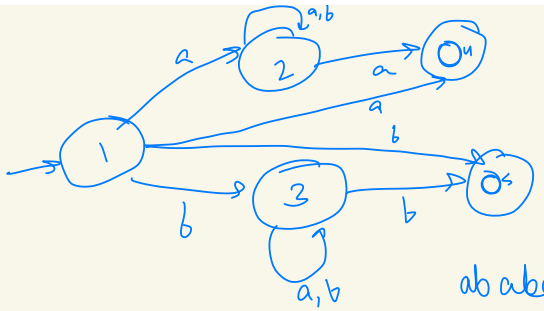
change from ab to ab will have ba in between ab's and ba's will always alternate

Three states +1, -1, 0 y accepting state

$\{ L' = \{ w \in \Sigma^* \mid n_a(w) = n_b(w) \}$
→ There is no finite automaton



Two accepting states.



Non-deterministic FA

ababa

1 22222

1 22224

1 4 → stuck

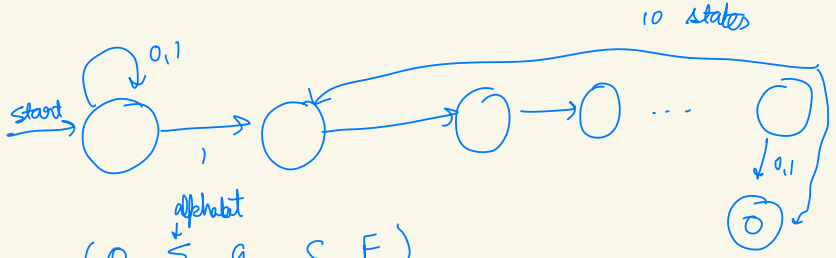
1 224 → stuck

If there is atleast one path which completes the string and ends up in an accepting state, we are done!

$\Sigma = \{0,1\}$

$L = \{ w \in \Sigma^* \mid |w| > 10,\ 10^{th} \text{ letter from end is a} \}$

2^{10} states remember 10 length strings



DFA : $(Q, \Sigma, q_0, \delta, F)$

Σ ← alphabet

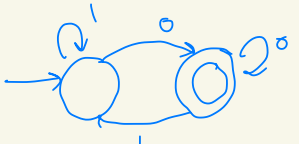
Q ← set of states

$q_0 \in Q$ ← initial state

δ ← Transition fn

$F \subseteq Q$ ← set of final states

$\delta : Q \times \Sigma \to Q$

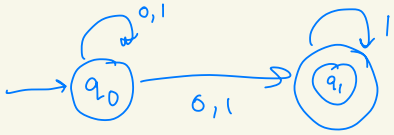


$\{ \{q_0, q_1\}, \{0,1\}, q_0, \delta, \{q_1\} \}$

δ	Q	Σ	Q
	q_0	1	q_0
	q_0	0	q_1
	q_1	0	q_1
	q_1	1	q_0

NFA : $(Q, \Sigma, Q_0 \subseteq Q, \delta, F)$

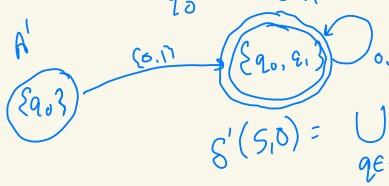
$\delta : Q \times \Sigma \to 2^Q$



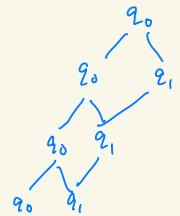
$L(A) = \{ w \in \{0,1\}^* \mid w \text{ is accepted by } A \}$

$L(A) = \Sigma^* \setminus \{\varepsilon\}$

Q	Σ^*	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	00	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$
q_0	011	



$\delta'(S, 0) = \bigcup_{q \in S}^{NFA} \delta(q, 0)$



New graph - DFA on the power set of the earlier states.

$L(A') = L(A)$ → we need to argue for this.

To show $L(A) = L(A')$

(i) $L(A') \subseteq L(A)$] Use induction

(ii) $L(A) \subseteq L(A')$

A: NFA

A': DFA

We will show

for every $n \geq 0$, for every $w \in \Sigma^*$ s.t $|w| = n$

NFA can reach state $q \in Q$ on reading w iff (on reading w)

DFA A' reaches state $S \subseteq Q$, s.t $q \in S$

Induction on n

Base: $n = 0$ from definition of initial state of A'

Hypothesis: Claim holds for $0 \leq n < k$

Induction step: To show that claim holds for $n = k$

$|w| = k$ write $w = w'a$, $a \in \Sigma$, $w' \in \Sigma^*$, $|w'| = k-1$

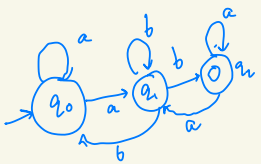
NFA $q_0 \rightsquigarrow q \rightarrow \hat{q}$ iff

DFA $q_0 \rightsquigarrow q \xrightarrow{a} \hat{q}$ ($\hat{q}$ will be here)

many states one of them is q

we will take paths from all of them and take union.

$q \rightarrow \hat{q}$ is there must be a part of the next subset.

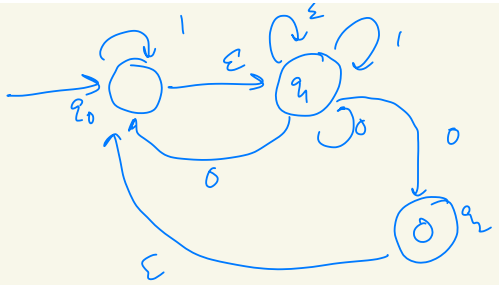


ababab

$\{q_0\} \xrightarrow{a} \{q_0, q_1\} \xrightarrow{b} \{q_0, q_1, q_2\} \xrightarrow{a} \{q_0, q_1, q_2\}$ (a, b)

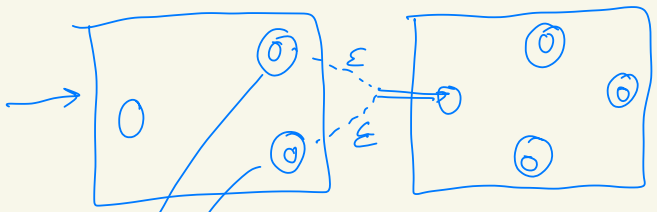
$\{q_0, q_1, q_2\} \xleftarrow{b} \{q_0, q_1, q_2\} \xleftarrow{a} \{q_0, q_1, q_2\}$

$O(nk)$ — n: length, k: size of the automaton



$10 \notin L$ without ε-edges

$10 \in L$ with ''



ε-transitions can happen both in DFAs and NFAs

If we join DFAs with ε then we will get NFAs

$L_1 \cdot L_2$ using these ε strings

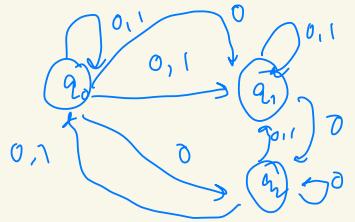
If we do so then these won't remain final.

ε-closure

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

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

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



final states = ε closure of final states

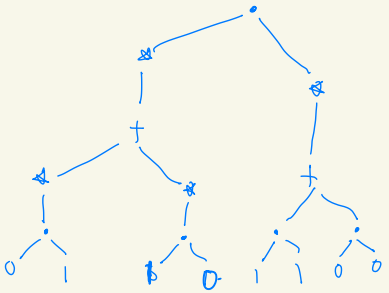
Start state = ε-closure of q_0 (or all start states)

* : Kleene closure
Kleene star

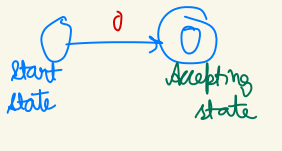
$L(\text{reg exp}) \subseteq L(\text{NFA with } \varepsilon) = L(\text{NFA w/o } \varepsilon) = L(\text{DFA})$

$\Sigma = \{0, 1\}$

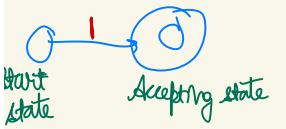
$((01)^* + (10)^*)^* \cdot (11+00)^*$

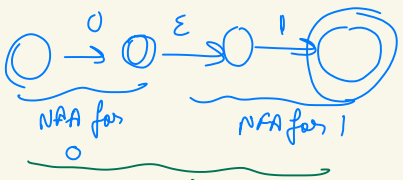


NFA for 0

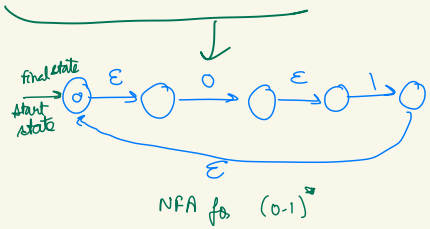


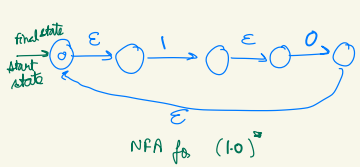
NFA for 1





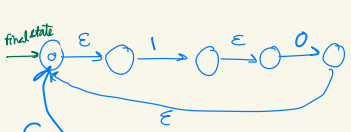
NFA for 0.1

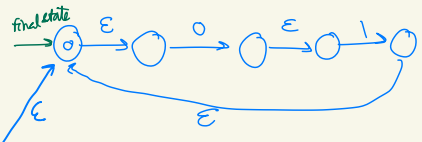


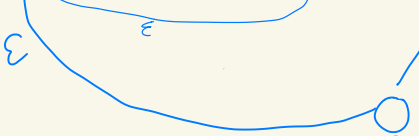


NFA for $(1\cdot0)^*$

NFA for $(0\cdot1)^*$

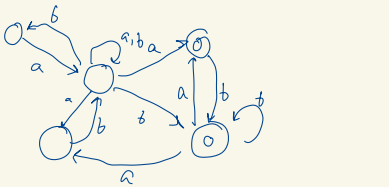


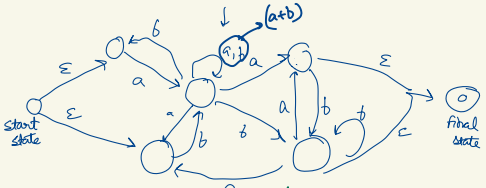


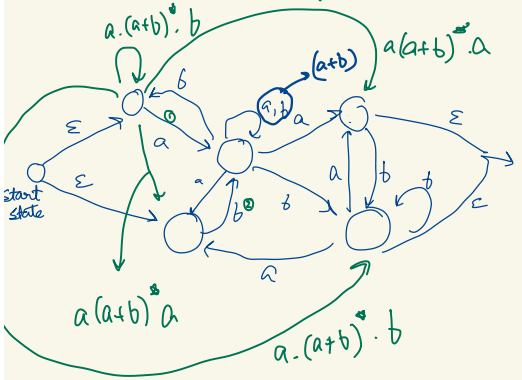


Since we have shown that
$L(\text{reg exp}) \subseteq L(\text{NFA with } \varepsilon)$

NFA for $(0.1)^* + (1.0)^*$



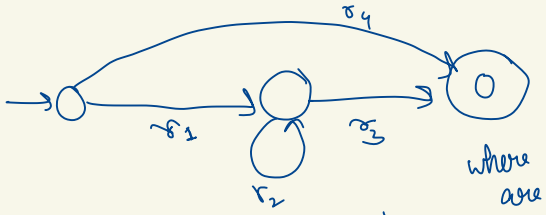




We can now delete edge ①
Similarly ② can be deleted
Now the middle state can't be reached, so that state itself can be removed along with its outgoing edges

One by one, we can remove all of the states

Pre-final state :



where r_1, r_2, r_3, r_4 are regular expressions

Each step is Language preserving

$((r_1 r_2^* r_3) + r_4)$

Reg Exp ≡ NFA with ε ≡ NFA w/o ε ≡ DFA : Kleene's Theorem

Regular Languages

$(a^* b^*)^*$ DFA L_1 $=$ $\subseteq$ $\supseteq$ $(a+b)^*$ DFA L_2 — we ask this question

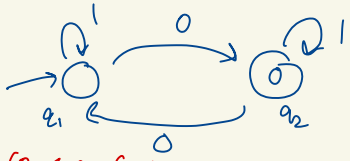
$L_1 \cap (\Sigma^* \setminus L_2) = \phi \equiv L_1 \subseteq L_2$

L_2 is reg $\Leftrightarrow$ L_2^* or $\overline{L_2}$ is reg

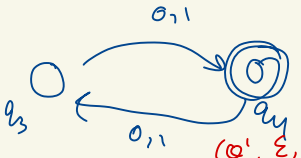
(Use from DFA, final → not final, not final → final)

check if a path from start to final

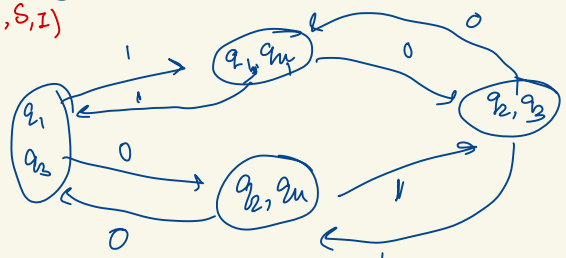
now we need to take intersection



$(Q, \Sigma, q_0, \delta, I)$



$(Q', \Sigma, q_0', \delta', F')$



$(Q \times Q', \Sigma, (q_0, q_0'), \hat{\delta}, \hat{F})$

Intersection DFA

Two automata running in parallel

Starting state = starting state of both product

Number of states = cartesian product of earlier states

One construction can serve the purpose of all boolean operations of the earlier DFAs.

$\hat{\delta}((q_i, q_j'), a) = (\delta(q_i, a), \delta'(q_j', a))$ Product construction for Automata

$\downarrow$

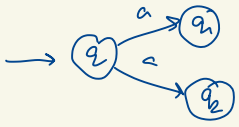
$\in \Sigma$

$\hat{\delta} : (Q \times Q') \times \Sigma \to (Q \times Q')$

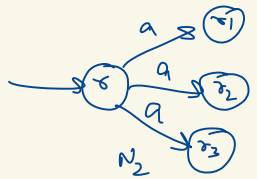
$L_1 = L_2 \Leftrightarrow L_1 \subseteq L_2$ and $L_2 \subseteq L_1$

$L_1 \cap (\Sigma^* \setminus L_2) = \phi$ $\quad$ $L_2 \cap (\Sigma^* \setminus L_1) = \phi$

N_1 and N_2 are NFAs



N_1



N_2

$\hat{\delta}((q,a), a) = \{q_1, q_2\} \times \{r_1, r_2, r_3\}$

$= \delta(q,a) \times \delta'(r,a)$

DFA complement = flip the status of accepting and non-accepting states

for NFA, convert to DFA and then take complement

Closure properties of Regular Languages:

L_1, L_2 = Reg. lang. over $\Sigma_1 = \{a, b\}$

$\overline{L_1}$ $\quad$ $L_1 \cup L_2$, $L_1 \cap L_2$

Reg. Language $L_1 \subseteq \Sigma_1^*$ $a^* b^*$

substitute $(L_1, L_a, L_b) = \{ w \in \Sigma_2^* \mid \exists\ v \in L_1$ s.t. $w \in L_{a_1} L_{a_2} \cdots L_{a_k} \}$

$\Sigma_2 = \{0, 1, 2\}$

$L_a \in \Sigma_2^*$ Reg. Lang.

$L_b \in \Sigma_2^*$ Reg. Lang.

$L_a = 0^* (1.2)^* 1^*$

$L_b = 1^* (0+2)^*$

$L_1 = a^* b^*$ $\quad L_a = 0^* (1+2)^* 1^* \quad L_b = 1^* (0+2)^*$

Eg: $aabbb \in L_1$

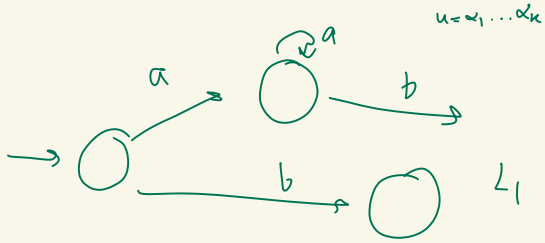
$= u$

a	a	b	b	b
⋮	⋮	⋮	⋮	⋮
α_1	α_2	α_3	α_4	α_5

Replace a by L_a

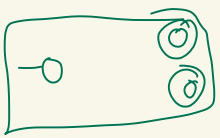
$w \in L_a \cdot L_a \cdot L_b \cdot L_b L_b$?

subst $(L_1, L_a, L_b) = \bigcup_{u=\alpha_1 \dots \alpha_k} L_{\alpha_1} \dots L_{\alpha_k}$

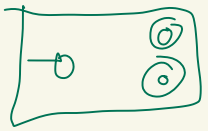


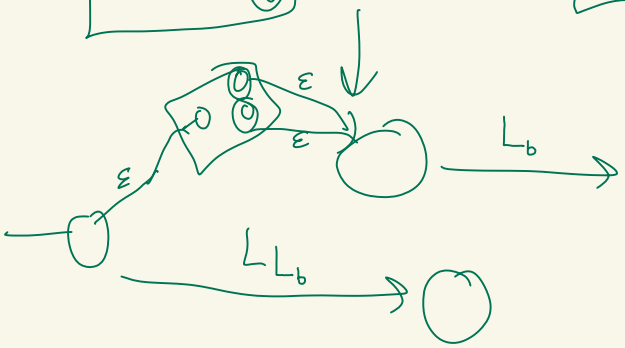
L_1

L_a



L_b





$L_a \cdot L_b$

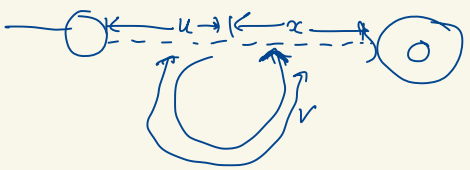
$\Sigma_1 = \{a, b\}$ $\quad L_1 = \{a \cdot b\}$

L_a even number of times
L_b odd number of times

$L' = \{a^n b^m \mid n = 0 \bmod 2,\ m = 1 \bmod 2\}$

Use substitution.

Consider $|A| = 20$ states, A = DFA
$w \in L(A)$ $\quad |w| = 100$ → Loop must exist



$w = u.v.x$

$u.x \in L$

$u.v.v.x \in L$

$u.v^i.x \in L$

for $u \; i \geq 0$

If a DFA of n states accepts a string of length $n+1$, then the language space of that DFA = infinite.

Check between length n and $2n$

Shortest string with this behaviour = $2n$

Else no.

The pumping lemma for regular languages

Let L be a regular language. Then there exists a constant n (which depends on L) such that every string w in L such that $|w| \geq n$, we can break w into three strings $w = xyz$ such that:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. for all $k \geq 0$, the string $xy^k z$ is also in L.

Proof: $L = L(A)$ for some DFA A

$A \rightarrow n$ states

$w = q_1 q_2 \dots q_m \quad m \geq n$

for $i = 0 \dots n$ define $P_i = \hat{\delta}(q_0, q_1, q_2, \dots, q_i)$

$\hat{\delta}$: transition function

$P_0 = q_0$ $\quad q_0$ = start state of A

By pigeonhole principle $P_i = P_j$ for some $i < j$

$w = xyz$

$x = q_1 \dots q_i$

$y = q_{i+1} \dots q_j \rightarrow \neq \varepsilon$

$z = q_{j+1} \dots q_m$

* Regular languages are closed under union, intersection, complement, difference, reversal, closure (star) and concatenation.
* Running time of NFA to DFA conversion, including the case where the NFA has ε-transitions is $O(n^3 2^n)$.
* Automaton → Regex $O(n^3 4^n)$

* (i) $R = R_1 + R_2$ $L(R)$ is empty if and only if $L(R_1)$ and $L(R_2)$ are empty.
* (ii) $R = R_1 R_2$ $L(R)$ is empty if and only if $L(R_1)$ or $L(R_2)$ is empty
* (iii) $R = R_1^*$ $L(R)$ is not empty, it always includes at least ε.
* (iv) $R = (R_1)$ $L(R)$ is empty if and only if $L(R_1)$ is empty since they are the same language.

CS 208 Tutorial 4

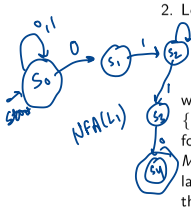
1. Let $\Sigma = \{0,1\}$. In each of the following problems, you have to prove the regularity of a desired language.

   1. For every word w in Σ^*, we say $w' \in \Sigma^*$ is a subword of w iff w' is obtained by replacing some letters in w with ε. Thus, 011 is a subword of 0010111, but is not a subword of 0001. Furthermore, for $w \in \Sigma^*$, define w^R as the string w read in reverse. For example, 011^R is 110. Now, let L be a regular language over Σ. Define

      $$\text{Lossy}(L) := \{w' : w' \text{ is a subword of some word in } L\}$$

      The need for such languages arise in real life: Imagine a channel on some network, on which we are transmitting bit streams. However, the channel is lossy, and some bits are lost in transmission. Therefore, if we transmit a word w, we may receive a subword of w. In particular, if we want to transmit words from a regular language L [1], we'll actually end up receiving subwords of words in L [2].
      Show that if L is regular, so is $\{w^R : w \in \text{Lossy}(L)\}$. by constructing an **NFA** for this language.

   2. Let $L_1 := L((0+1)^*01(0+1)^*10)$. Define

      $$L_2 := \bigcup_{a_1 a_2 \cdots a_k = w \in L_1} S(a_0) \cdot S(a_1) \cdots S(a_k)$$

      where $a_i \in \{0,1\}$ are the alphabets of the word $w \in L_1$, and $S(0) := \{0101\}, S(1) := \{1010\}$ are functions mapping each letter of Σ to a singleton language. Construct a **DFA** for L_2.
      Motivation: Such transformations are used in networks, where we encode individual bits into larger chunks for redundancy in our transformation, so that even if a few bits are corrupted, the transmitted word can still be recovered. Also look up string homomorphisms (and inverse homomorphisms) from Hopcroft, Motwani and Ullmann.

   3. Let L_1, L_2 be regular languages over Σ. Define

      $$\text{interleave}(L_1, L_2) := \{\alpha_1\beta_1\alpha_2\beta_2 \ldots \alpha_k\beta_k : \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k \in \Sigma^*, \alpha_1\alpha_2 \ldots \alpha_k \in L_1, \beta_1\beta_2 \ldots \beta_k \in L_2\}$$

      In the above definition, $\alpha_1\alpha_2 \ldots \alpha_k$ represents the concatenation of strings $\alpha_1, \alpha_2, \ldots, \alpha_k$, and similarly for $\beta_1\beta_2 \ldots \beta_k$.
      Prove that if L_1, L_2 are regular, then interleave(L_1, L_2) is regular.
      Motivation: Imagine a network where packets are arriving from two different channels (look up TDM channels). Then what is received on the other side is an interleaving of the packets received from both the channels.

[1] Why might we want to do so? Recall that regular expressions and hence regular languages can be used to describe sets of words containing specific patterns.
[2] If you are more interested about such things, you can look up *erasure codes* for further reading.

3. $M_1 = (P, \Sigma, \delta_1, p_0, F_1)$, $M_2 = (Q, \Sigma, \delta_2, q_0, f_2)$ } DFAs accepting L_1 and L_2 respectively

$M = (P \times Q, \Sigma, \delta, \{(p_0, q_0)\}, F_1 \times F_2)$

NFA

$\delta((p,q), a) = \{(\delta(p,a), \varepsilon), (p, \delta_2(q,a))\}$

claim: $L(M) = \text{interleave}(L_1, L_2)$

Proof: $\hat{\delta}(\{(p_0, q_0)\}, x) = \{(\hat{\delta}, (p_0, x), \hat{\delta_2}(q_0, f_2))\}$

2. Recall the Pumping Lemma described in class (without mentioning the name as such). To recall: Let L be a regular language. Then there is an integer $p \geq 1$ (depending only on L) such that for any $w \in L$ such that $|w| \geq p$, we can write $w = xyz$, such that:

   1. $|y| \geq 1$.
   2. $|xy| \leq p$. Note that x *may* be ε.
   3. $xy^n z \in L$ for all $n \geq 0$. In particular, $xz \in L$.

   More informally, for any regular language, and for any long enough word in it, after removing some small enough prefix and some suffix, the rest of the word is just some small enough word repeated over and over again, i.e. 'pumped' (that's where the lemma gets its name from).

   The Pumping Lemma is extremely useful to show that a given language is **not regular**. Basically, if we show that a given language L doesn't satisfy the Pumping Lemma, then it can't be regular (since every regular language satisfies the Pumping Lemma).

   The strategy to show a language L non-regular using Pumping Lemma is best viewed as a turn-based game between an adversary (who wants to show that the language is not regular) and a believer (who believes that the language is regular). The game goes as follows:

   - The believer chooses an integer $p > 0$. She claims this is the count of states in the DFA that she believes recognizes the language.
   - The adversary chooses a (often carefully constructed) string $w \in L$ such that $|w| > p$.
   - The believer then splits w into three parts $w = x \cdot y \cdot z$, where $|xy| \leq p$. The believer thinks this is the shortest prefix of w that ends up looping back to a state of the p-state DFA.
   - The adversary now chooses an (often carefully constructed) integer $n \geq 0$ such that $xy^n z \notin L$, thereby winning the game

   If the **adversary can win** the above game (i.e. can choose w in step 2 and n in step 4) *for every choice of p and for every decomposition of w and $x \cdot y \cdot z$ chosen by the believer*, then the language L must be non-regular. Why so? Because if L was indeed regular, there must exist a DFA with a certain number of states accepting L. If the believer chooses p to be this count of states in step 1, then we know from the Pumping Lemma that for every $w \in L$ such that $|w| > p$ which the adversary can choose in step 2, there must exist a decomposition $x \cdot y \cdot z$ with $|xy| \leq p$ such that $xy^n z \in L$ for all $n \geq 0$.

   Take a few moments to understand the above sequence of steps in the game.

   Using the above idea, show that the following languages are **not** regular:

   1. $\{0^n 1^m : n \geq m \geq 0\}$. Consider $n=p$ $m=p$
   2. $\{0^{n^2} : n \geq 0\}$. Consider $n=p$ $n^2 - n = (n-1)^2$ → Hence $\alpha z \notin L$

   It is important to note that the pumping lemma is not and "if and only if" statement: If a language is regular, it satisfies the lemma, but not necessarily the other way arond. Indeed, there are languages that are not regular, yet satisfy the conditions of the pumping lemma.

   Try to prove that the following language is not regular (with knowledge of the fact that $\{a^n b^n \mid n \geq 0\}$ is not regular – a fact that can be proved again using the Pumpling Lemma), yet can be pumped.

$$L := \{c^m a^n b^n : m \geq 1, n \geq 0\} \cup a^* b^*$$

$L_1 = \{c a^n b^m\}$ $L_2 = c a^* b^*$

L_1 ✗ regular L_2 = regular $L_2 \cap L = L_1$

If L were regular then L_1 = regular

3. **Takeaway**: In this question, we'll see an almost templatized way of proving that certain transformations of regular languages are regular.
   Let $\mathcal{L}$ be a regular language, and let $\mathcal{A} := (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a DFA recognizing $\mathcal{L}$. Note that Q is the set of states of $\mathcal{A}$, $q_0 \in Q$ is the start state, $\mathcal{F} \subseteq Q$ is the set of final states, Σ is our alphabet, and $\delta : Q \times \Sigma \mapsto Q$ is the transition function.
   For any $\alpha \in Q, B \subseteq Q$, define $\mathcal{L}(\alpha, B)$ to be the regular language recognized by the DFA $(Q, \Sigma, \delta, \alpha, B)$, i.e. we take $\mathcal{A}$ and change the starting state to α and the end state(s) to B.
   Let $\mathcal{L}$ be a regular language. Prove that the following languages are regular:

   1. $\text{init}(\mathcal{L}) := \{w : wx \in \mathcal{L} \text{ for some } x \in \Sigma^*\}$ $\alpha = q_0$ B → set of all states which can reach F will work.
   2. $\text{CubeRoot}(\mathcal{L}) := \{w : w^3 \in \mathcal{L}\}$

   [Hint: Try to express these languages as union/intersections/concatenations of $L(\alpha, B)$'s for various α, B.]
   See section 4.2 of Hopcroft, Motwani, and Ullmann for more problems of this type.

   These questions can also be solved by converting the DFA of $\mathcal{L}$ to NFAs accepting the given languages. However, many of those conversions are clumsy, and it requires some care to show that the transformed automaton accepts precisely the desired language. The above method, however, is short and much more transparent.

→ Consider two states in between a, b

$q_0 \to a$
$a \to b$
$b \to F$ → Intersection of these three languages — union over all a, b

5. **Takeaway**: A student claims that every regular language over a unary alphabet (say, $\Sigma = \{0\}$) is a finite union of languages of the form $L_{c,d} = \{0^{cn+d} \mid n \geq 0\}$ for constant integers $c, d \geq 0$. Either prove the student wrong by providing a regular language over $\Sigma = \{0\}$ that can't be expressed in the above form (you must give a proof of this), or prove that the student's claim is correct.

$L = \{ 0^{a_1} \; 0^{a_2} \; \dots \; 0^{a_k} \; \dots \}$ $\quad a_i > 0$

↳ regular language over unary alphabet

Now we need to find $(c_1, d_1)\ (c_2, d_2) \dots (c_r, d_r)$

such that $\forall i$, $a_i = c_j n + d_j$ for some j

Case-1 : Language has finite strings then :

$a_1, a_2, \dots, a_n$

$d_1 \; d_2 \; \dots \; d_n$ → finite union.

$c_i = 0 \quad \forall i$

Case-2 : Language has infinitely many strings.

Pumping Lemma → $xy^n z$ → $a + b = d$

$x = a \quad z = b$

$y = c$

Steps for converting NFA with ε to NFA without ε :

(1) Write ε-closure for all states q. (transitive closure)

(2) For every state q, find the union of all of the states that can be reached from ε-closure (q) states on each bit x and add edges from q to those states in the union set, if they are not present.

(3) mark all the states whose ε-closure contains accepting states as accepting.

(4) Remove all the ε-edges.

CS 208 : Automata Theory and Logic

Spring 2024

Instructor : Prof. Supratik Chakraborty

Disclaimer

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

Contents

Chapter 1

Propositional Logic

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

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

	D_1	D_2	D_3	D_4	D_5
S_1					
S_2					
S_3					
S_4					

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

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

1.1 Syntax

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

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

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

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

Table 1.1: Logical connectives.

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

Rules for formulating a formula:

- Every variable constitutes a formula.
- The constants $\top$ and $\bot$ are formulae.
- If φ is a formula, so are $\neg\varphi$ and (φ).
- If φ_1 and φ_2 are formulas, so are $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$.

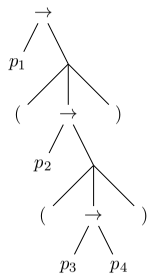
Propositional formulae as strings and trees:

Formulae can be expressed as a strings over the alphabet $\mathbf{Vars} \cup \{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$. **Vars** is the set of symbols for variables. Not all words formed using the alphabet qualify as propositional formulae. A string constitutes a well-formed formula (wff) if it was constructed while following the rules. Examples: $(p_1 \vee \neg q_2) \wedge (\neg p_2 \rightarrow (q_1 \leftrightarrow \neg p_1))$ and $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$.
Well-formed formulas can be represented using trees. Consider the formula $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$. This can be represented using the parse tree in figure Figure 1.1a. Notice that while strings require parentheses for disambiguation, trees don't, as can be seen in Figure 1.1b and Figure 1.1c.

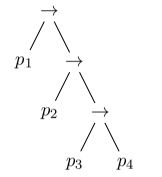
1.2 Semantics

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

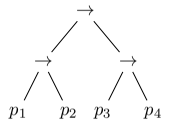
$$[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$$



(a)



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



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

Figure 1.1: Parse trees obviate the need for parentheses.

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

φ	$\neg\varphi$
0	1
1	0

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

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

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

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

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

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

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

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

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

Table 1.2: Truth tables of operators.

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

Rules of semantics:

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

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

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

Example. Let us construct a truth table for $[\![(p \vee s) \to (\neg q \leftrightarrow r)]\!]$ (see Table 1.3).

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

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

1.2.1 Important Terminology

A formula φ is said to (be)

- **satisfiable** or **consistent** or SAT iff $[\![\varphi]\!] = 1$ for some assignment of variables. That is, there is at least one way to assign truth values to the variables that makes the entire formula true. Both a formula and its negation may be SAT at the same time (φ and $\neg\varphi$ may both be SAT).
- **unsatisfiable** or **contradiction** or UNSAT iff $[\![\varphi]\!] = 0$ for all assignments of variables. That is, there is no way to assign truth values to the variables that makes the formula true. If a formula φ is UNSAT then $\neg\varphi$ must be SAT (it is in fact valid).
- **valid** or **tautology**: $[\![\varphi]\!] = 1$ for all assignments of variables. That is, the formula is always true, no matter how the variables are assigned. If a formula φ is valid then $\neg\varphi$ is UNSAT.
- **semantically entail** φ_1 iff $[\![\varphi]\!] \preceq [\![\varphi_1]\!]$ for all assignments of variables, where 0 (false) $\preceq$ 1 (true). This is denoted by $\varphi \models \varphi_1$. If $\varphi \models \varphi_1$, then for every assignment, if φ evaluates to 1 then φ_1 will evaluate to 1. Equivalently $\varphi \to \varphi_1$ is valid.

- **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$. Basically φ and φ_1 have identical truth tables. Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid.
- **equisatisfiable** to φ_1 iff either both are SAT or both are UNSAT. Also note that, semantic equivalence implies equisatisfiability but **not** vice-versa.

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

Table 1.4: Some examples for the definitions.

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

Table 1.5: Proof rules.

In the $\to_i$ rule, the box indicates that we can *temporarily* assume φ_1 and conclude φ_2 using no extra non-trivial information. The $\to_e$ is referred to by its Latin name, *modus ponens*.

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

$$\dfrac{\dfrac{\dfrac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3}\wedge_{e_2}}{\varphi_2}\wedge_{e_1} \quad \dfrac{\dfrac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3}\wedge_{e_2}}{\varphi_3}\wedge_{e_2}}{\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}\wedge_i}\wedge_i$$

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

1.4 Natural Deduction

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

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

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

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

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

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

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

1.	p	premise
2.	$\neg p$	assumption
3.	$\bot$	$\neg_e$ 1, 2
4.	$\neg\neg p$	$\neg_i$ 2–3

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

1.	$p \to q$	premise
2.	$\neg q$	premise
3.	p	assumption
4.	q	$\to_e$ 3,1
5.	$\bot$	$\neg_e$ 4,2
6.	$\neg p$	$\neg_i$ 3–5

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

1.	$\neg p \wedge \neg q$	premise
2.	$p \vee q$	assumption
3.	p	assumption
4.	$\neg p$	$\wedge_{e_1}$ 1
5.	$\bot$	$\neg_e$ 3,4
6.	$p \to \bot$	$\to_i$ 3–5
7.	q	assumption
8.	$\neg q$	$\wedge_{e_2}$ 1
9.	$\bot$	$\neg_e$ 7,8
10.	$q \to \bot$	$\to_i$ 7–9
11.	$\bot$	$\vee_e$ 2,6,10
12.	$\neg(p \vee q)$	$\neg_i$ 2–11

1.5 Soundness and Completeness of our proof system

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

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

The rules that we have chosen are indeed individually sound since they ensure that if for some assignment the premises evaluate to 1, so does the inference. Otherwise they rely on the notion of contradiction and assumption. Hence, soundness for any proof can be shown by inducting on the length of the proof.
A complete proof system is one which allows the inference of *every* valid semantic entailment:

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

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

p	q	$p \to q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

As we can see, whenever both $p \to q$ and $\neg q$ are true, $\neg p$ is true. The question now is how do we derive this using proof rules? The idea is to 'mimic' each row of the truth table. This means that we assume the values for p, q and try to prove that the formulae in Σ imply φ[1]. And to prove an implication, we can use the $\to_i$ rule. Here's an example of how we can prove our claim for the first row:

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

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

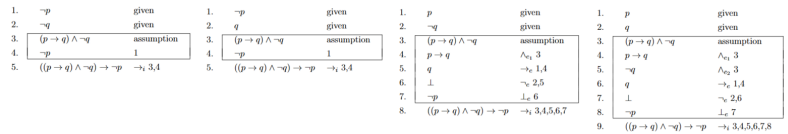


Figure 1.2: Mimicking all 4 rows of the truth table

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

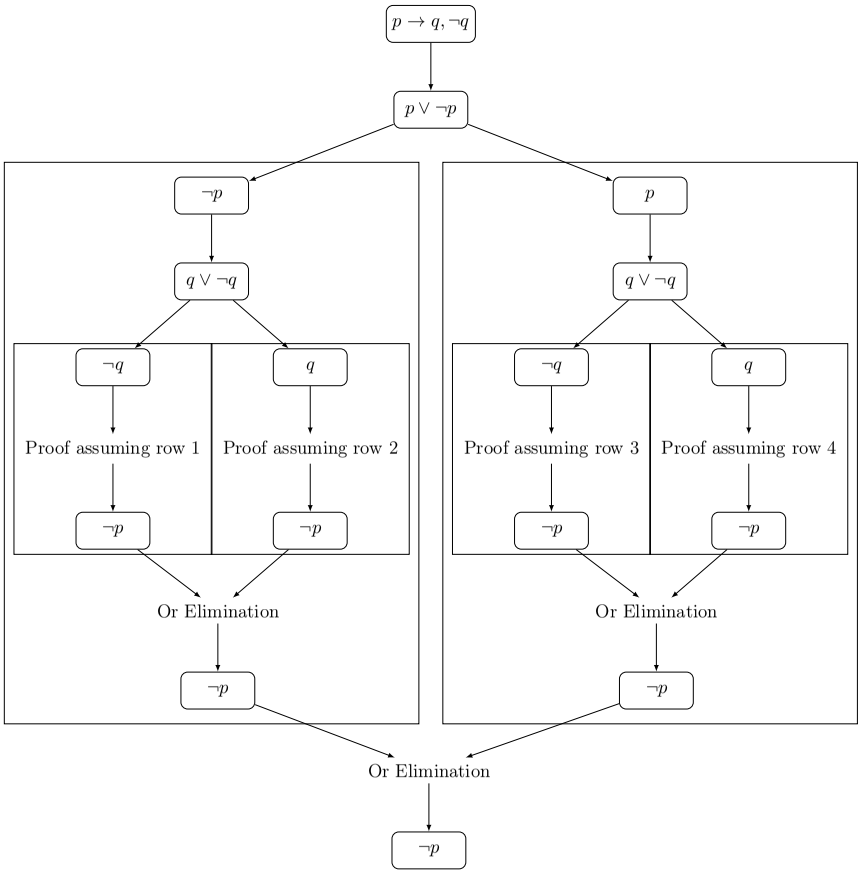
But as seen above, we were able to show a proof for the sequent $\vdash \varphi \vee \neg\varphi$. If we just recursively apply this property for all the variables we have, we should be able to capture every row of truth table. So combining this result, our proofs for each row of the truth table, and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

This can be done using **structural induction** to prove the following:
Let φ be a formula using the propositional variables $p_1, p_2, \ldots, p_n$. For any assignment to these variables define $\hat{p}_i = p_i$ if p_i is set to 1 and $\hat{p}_i = \neg p_i$ otherwise, then:

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

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

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



1.6 What about Satisfiability?

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

not all. Example, for some p and q,

$$\nvdash p \wedge q$$

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

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

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

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.

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

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

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.

$$\phi_1 \longleftrightarrow \phi_2 \dashv\vDash (\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$$

$$(\phi_1 \to \phi_2) \dashv\vDash ((\neg\phi_1) \vee \phi_2)$$

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).

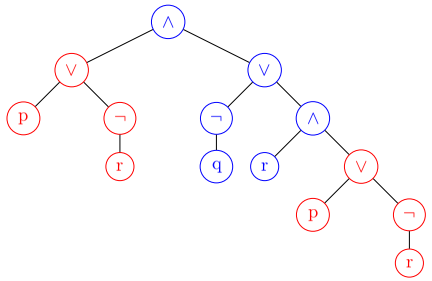
$$\neg(\phi_1 \wedge \phi_2) \dashv\vDash (\neg\phi_1 \vee \neg\phi_2)$$

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

1.8 Negation Normal Forms

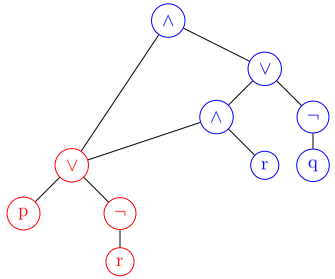
In mathematical logic, a formula is in negation normal form (NNF) if the negation operator ($\neg$) is only applied to variables and the only other allowed Boolean operators are conjunction ($\wedge$) and disjunction($\vee$) One can convert any formula to a NNF by repeatedly applying DeMorgan's Laws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

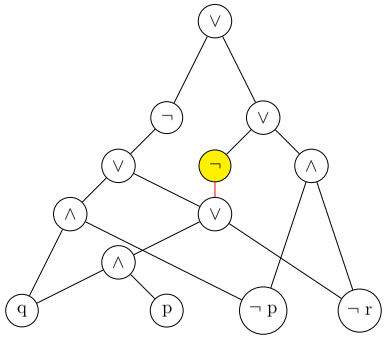
DAG Representation



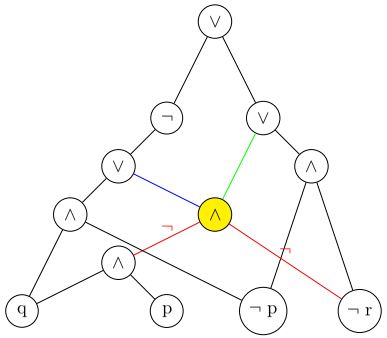
1.9 From DAG to NNF-DAG

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.

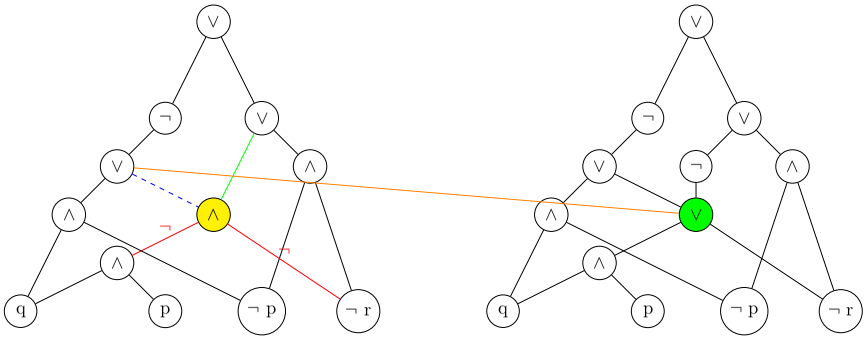


Figure 1.3: Step 1

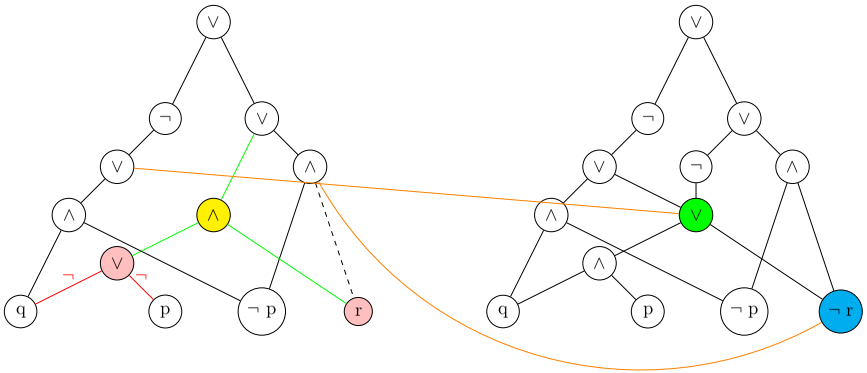


Figure 1.4: Step 2

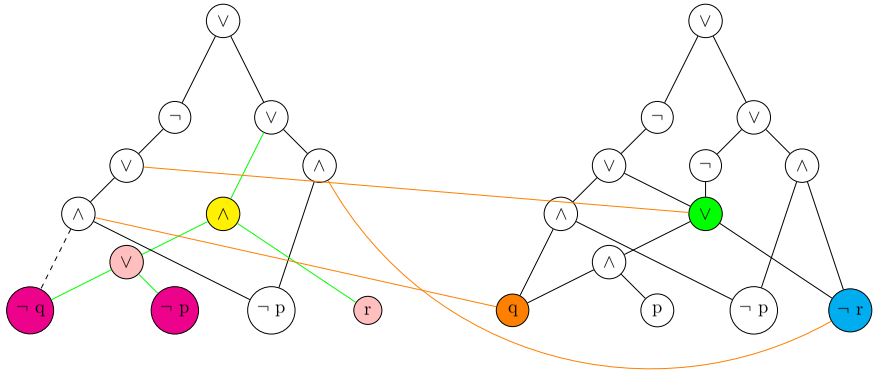


Figure 1.5: Step 3

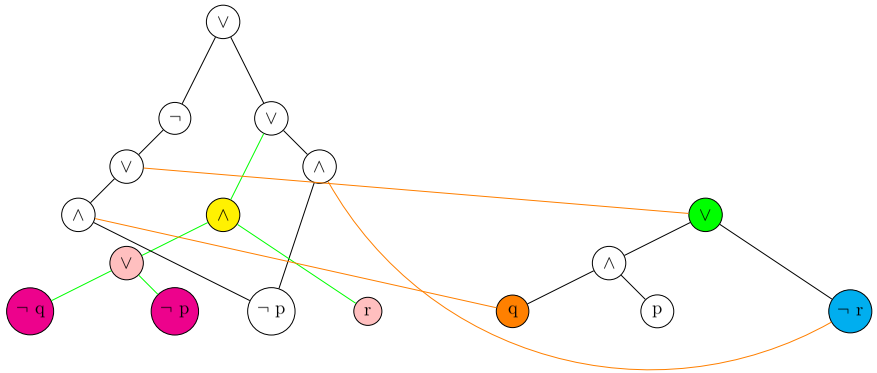


Figure 1.6: Step 4 : Remove all redundant nodes

1.10 An Efficient Algorithm to convert DAG to NNF-DAG

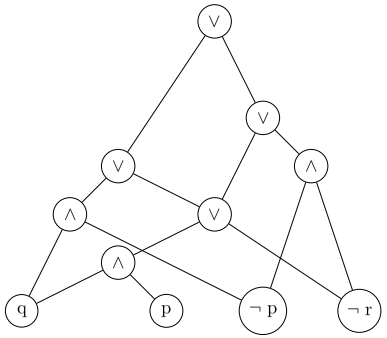


Figure 1.7: Step 1:Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

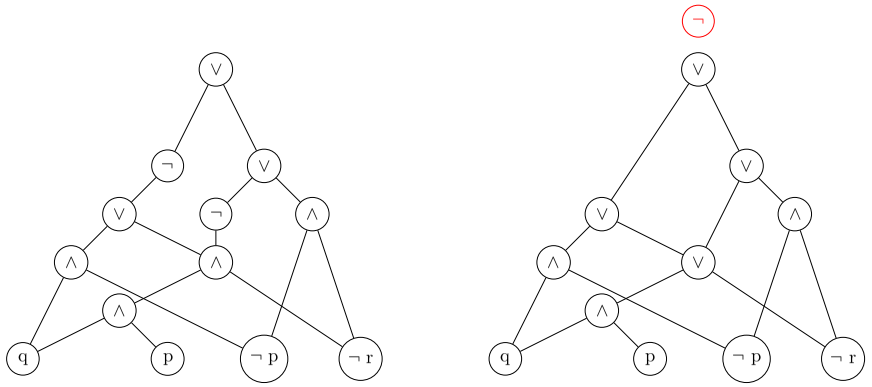


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

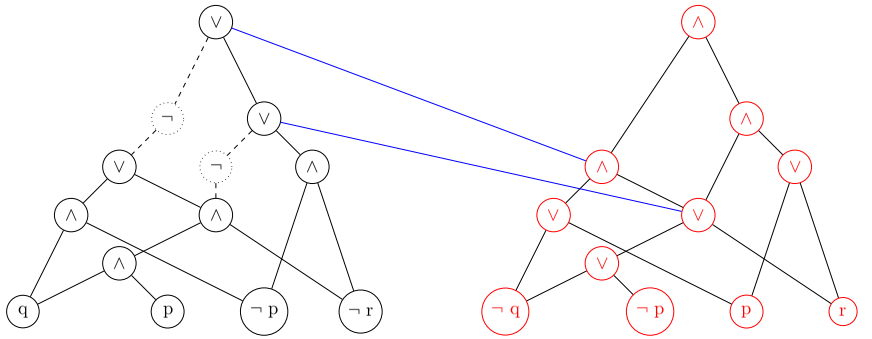


Figure 1.9: Step 3: Remove the "¬" nodes from the first DAG by connecting them to the corresponding node in the negated DAG.

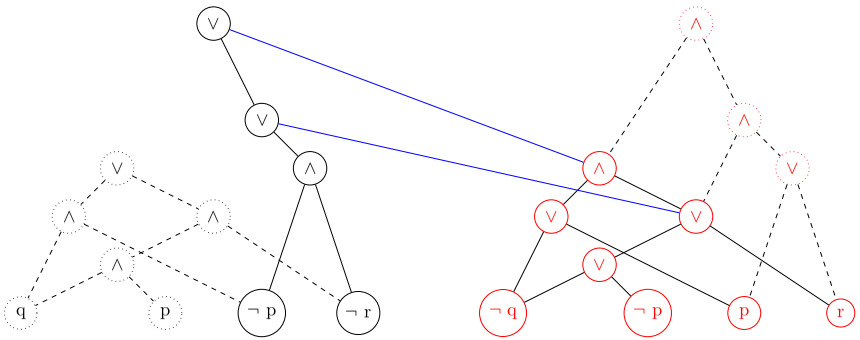


Figure 1.10: Step 4: Remove all the redundant nodes.

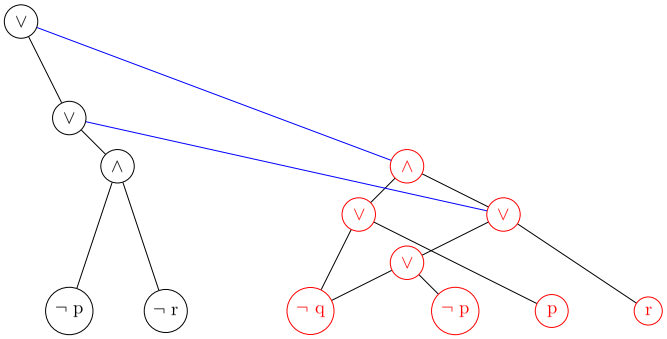


Figure 1.11: NNF - DAG

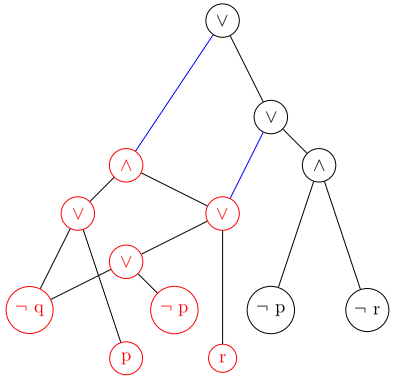


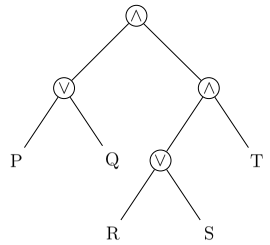
Figure 1.12: The NNF DAG (rearranged)

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.

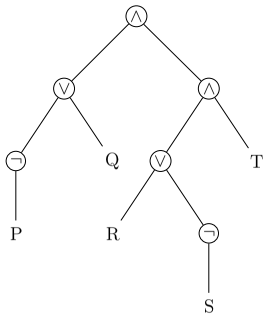
1.11 Conjunctive Normal Forms

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



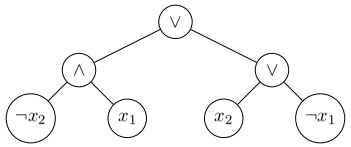
Some Important Terms

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

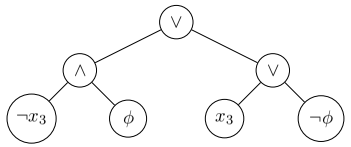
- **CLAUSE :**
  - A clause is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal
  - Example : p ∨ q ∨ (¬ r) , p ∨ (¬ p) ∨ (¬ r) -> Not Allowed
- **CUBE :**
  - a Cube is a conjunction of literals such that a literal and its negation are not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

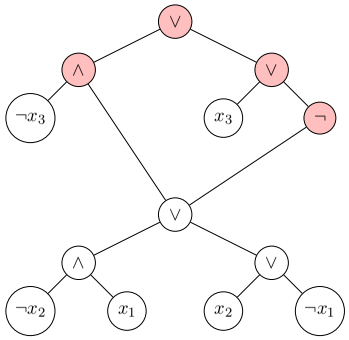
Tutorial 2 Question 2:
The Parity Function can be expressed as (((....$(x_1 \oplus x_2)$) $\oplus x_3$) ..... $\oplus x_n$)
The Parse Tree for $x_1 \oplus x_2$ is



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:

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

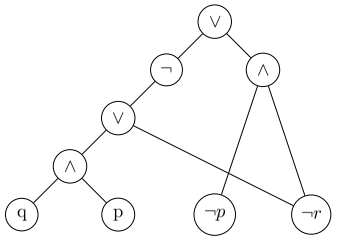
4. For checking the equisatisfiabiliy of ϕ and ϕ': Think about an assignment which makes ϕ true then apply that assignment to ϕ'.

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 $(\neg((q \wedge p) \vee \neg r)) \vee (\neg p \wedge \neg r)$



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

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

1.15 Towards Checking Satisfiability of CNF and Horn Clauses

A Horn clause is a disjunctive clause (a disjunction of literals) with at most one positive literal.
A Horn Formula is a conjuction of horn clauses, for example:

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

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

$$\begin{aligned} x_1 \wedge x_2 &\implies x_3 \\ x_4 \wedge x_3 &\implies x_5 \\ x_1 \wedge x_5 &\implies \bot \\ x_5 &\implies x_3 \\ \top &\implies x_5 \end{aligned}$$

Now we try to find a satisfying assignment for the above formula.
From the last clause we get $x_5 = 1$, now the fourth clause is $\top \implies x_3$.
Then from the fourth clause we get that $x_3 = 1$.
Now in the remaining clauses none of the left hand sides are reduced to $\top$.
Hence, we set all remaining variables to 0 to get a satisfying assignment.

Algorithm 1: HORN Algorithm

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

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

1.16 Counter example for Horn Formula

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

1.16.1 Example

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

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

Let us write down these implications

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

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 φ = ⊤ then
  return SAT(sat, PA)
else if φ = ⊥ then
  return SAT(unsat, PA)
else if C_i is a unit clause (literal l) and C_i ∈ φ then
  {//This step is called Unit Propagation}
  return SAT(φ[l = 1], PA ∪ {l = 1}) {// Here recursively call the algorithm on the simplified}
                                                                  //formula φ[l = 1]
else if l is a pure literal and l ∈ φ then
  {//This step is called Pure Literal Elimination}
  return SAT(φ[l = 1], PA ∪ {l = 1})
else
  {//This step is called Decision Step}
  x ← choose_a_var(φ)
  v ← choose_a_value({0, 1})
  if SAT(φ[x = v], PA ∪ {x = v}).status = sat then
    return SAT(sat, PA ∪ {x = v})
  else if SAT(φ[x = 1 − v], PA ∪ {x = 1 − v}).status = sat then
    return SAT(sat, PA ∪ {x = 1 − v})
  else
    return SAT(unsat, PA)
  end if
end if
```

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

ANS. Yes. Here is an example

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

1.18 DPLL in action

1.18.1 Example

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

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

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

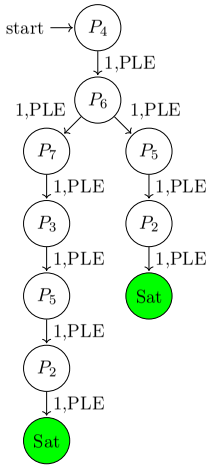


Figure 1.13: DPLL

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

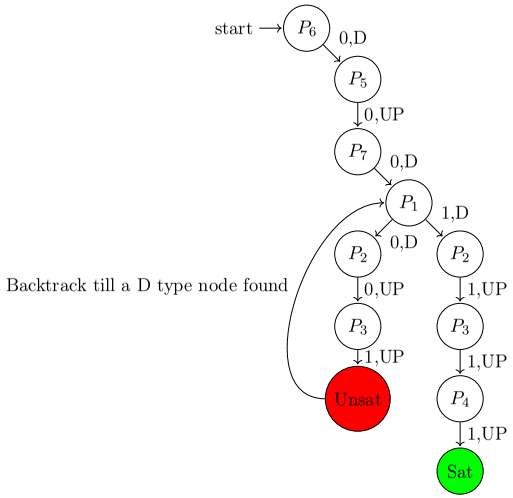


Figure 1.14: DPLL without point 2

1.19 Applying DPLL Algorithm to Horn Formulas

Let us apply DPLL algorithm to Horn Formula
If there are no variables on LHS, it becomes a Unit Clause *i.e.* $\top \to x_i$ and is equivalent to (x_i).
Horn Method can be viewed in terms of DPLL algorithm as following:

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

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

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

1.20 DPLL on Horn Clauses

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

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

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

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

1.21 Rule of Resolution

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

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

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

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

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

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

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

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

1. Since both p_1 and p_2 appear in negated and un-negated form, we apply resolution on C_1 and C_2, which generates C_4, as follows:
$$\frac{(\neg p_1 \vee p_2) \quad (\ p_1 \vee \neg p_2)}{(\neg p_1 \vee p_1)} \text{ resolution}$$
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 $p_1, p_2 \ldots p_m$ be our propositional variables.
Base: $n = 1$ An unsatisfiable CNF in a single literal **must** contain the clauses (p_1) and $(\neg p_1)$ which upon resolution gives us **( )** the empty clause hence we raise **UNSAT**.
Inductive Hypothesis: Assume that our claim holds $\forall m \leq n-1$ we now show that it holds from $m = n$ as follows,

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

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

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

Chapter 2

DFAs and Regular Languages

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

2.1 Definitions

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

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

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

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

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

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

2.2 Deterministic Finite Automata

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

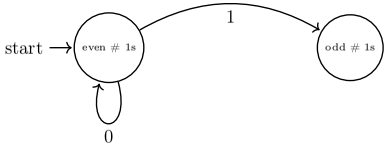
The solution to this lies in our discussion during the first lecture. I will record just one bit of information: whether I have received an even or odd number of 1s till now. Every time I receive a new bit, I will update this information: if it's a 0, I won't do anything, and if it's a 1, I will change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

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

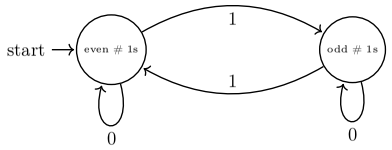
In our case we want to know whether there were even or odd number of 1s.

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

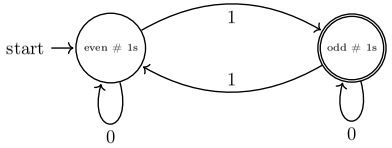
Now, suppose I receive a 0, I would remain in the same state, but if I get a 1 , the parity changes.



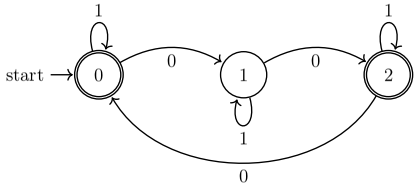
Now, if I am in the second state, if I get a 1 I will change states, and if I get a 0, parity is unchanged to I remain in the same state:



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



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



CS 208 : Automata Theory and Logic Spring 2024

Practice Problems 1

0. A Bit of Warm-Up

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

$$x \to \big((y \to \bot) \vee (\top \to z)\big)$$

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

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

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

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

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

Solution:

1. Indeed, it's possible. Here are suggested steps.
   - Write every implication $\alpha \to \beta$ as $(\neg\alpha \vee \beta)$. This gives us the formula
     $$\neg x \vee \big((\neg y \vee \bot) \vee (\neg\top \vee z)\big)$$
   - Simplify every $(\alpha \vee \bot)$ to α. Similarly, $\neg\top$ can be replaced by $\bot$. This gives us
     $$\neg x \vee \big((\neg y) \vee (z)\big)$$
   - Finally, apply DeMorgan's laws to replace every $\alpha \vee \beta$ with $\neg(\neg\alpha \wedge \neg\beta)$. This gives
     $$\neg\big(x \wedge (y \wedge \neg z)\big)$$
2. Notice that φ can be written as
   $$(p \vee q \vee r) \wedge (p \to q) \wedge (q \to r) \wedge (r \to s) \wedge (s \to t) \wedge (t \to p)$$
   The five implications above force p, q, r, s, t to have the same value in any satisfying assignment of φ (try to convince yourself why this is the case). Therefore, for φ to be satisfied, we must have at least one of p, q, r to be true (because of the first clause of φ), and all of p, q, r, s, t must have the same value (because of the remaining clauses of φ). This implies that the only satisfying assignment of φ is $p = q = r = s = t = 1$. Hence, the semantically equivalent DNF formula is $p \wedge q \wedge r \wedge s \wedge t$.

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

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

1. Core of unsatisfiability

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

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

Solution:

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

   Every proper subset of $\mathcal{S}_n$ either has p_1 missing, $\neg p_3$ missing or some implication $p_i \to p_{i+1}$ missing for $1 \leq i \leq n-2$. We show that in each of these cases, the remaining subset of formulas is satisfiable.

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

2. Logical interpolation

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

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

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

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

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

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

3. DPLL with horns

The Horn-Sat problem entails checking the satisfiability of Horn formulas. We say a formula is a Horn formula if it is a conjunction ($\wedge$) of Horn clauses. A Horn clause has the form $\phi_1 \to \phi_2$ where ϕ_1 is either $\top$ or a conjunction ($\wedge$) of one or more propositional variables. ϕ_2 is either $\bot$ or a **single** propositional variable. In this context, answer the following questions

1. Let's try to solve the Horn-Sat problem using DPLL. We can convert each Horn clause into a CNF clause by simply rewriting $(a \to b)$ as $(\neg a \vee b)$, which preserves the semantics of the formula. The resultant formula is in CNF.
   We have seen in class that if DPLL always chooses to assign 0 to a decision variable before assigning 1 (if needed) to the variable, then DPLL will never need to backtrack when given a Horn formula encoded in CNF as input.
   Suppose our version of DPLL does just the opposite, i.e. it always assigns 1 to a decision variable before assigning 0 (if needed). How many backtracks are needed if we run this version of DPLL on the (CNF-ised version of) following Horn formulas, assuming DPLL always chooses the unassigned variable with the smallest subscript when choosing a decision variable?
   (a)
   $$\Big(\big(\bigwedge_{i=0}^{n-1} x_i\big) \to x_n\Big) \wedge \bigwedge_{i=0}^{n-1} (x_n \to x_i) \wedge \Big(\big(\bigwedge_{i=0}^{n} x_i\big) \to \bot\Big)$$
   (b)
   $$\bigwedge_{i=0}^{n-1} \big((x_i \to x_{n+i}) \wedge (x_{n+i} \to x_i) \wedge (x_i \wedge x_{n+i} \to \bot)\big)$$
2. Since we have a polynomial time formula for Horn-Sat, the next question is if it will allow us to solve the Boolean-SAT problem in polynomial time. Sadly, this is not the case. Find a boolean function that cannot be expressed by a Horn formula. Prove that no Horn formula can represent the given boolean function.

Solution:

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

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

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

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

4. A Game of Sudoku

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

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

- Every cell contains at least one number:
  $$\phi_1 = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigvee_{k=1}^{9} p(i, j, k)$$
- Every cell contains at most one number:
  $$\phi_2 = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{y=x+1}^{9} (\neg p(i, j, x) \vee \neg p(i, j, y))$$
- Every row contains every number:
  $$\phi_3 = \bigwedge_{i=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{j=1}^{9} p(i, j, k)$$
- Every column contains every number:
  $$\phi_4 = \bigwedge_{j=1}^{9} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{9} p(i, j, k)$$
- Every 3×3 box contains every number:
  $$\phi_5 = \bigwedge_{r=0}^{2} \bigwedge_{s=0}^{2} \bigwedge_{n=1}^{9} \bigvee_{i=1}^{3} \bigvee_{j=1}^{3} p(3r + i, 3s + j, n)$$

The final formula is as follows

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

5. Poring over proofs

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

```
1. top
     ---------------------------------------------------------
2.   | x                 assumption                          |
     |  ------------------------------------------------------ |
3.   |  | neg x          assumption                        |  |
4.   |  | bot            bot introduction rule on 2 and 3 |  |
     |  ------------------------------------------------------ |
5.   | neg x             bot elimination rule on 4           |
6.   | bot               bot intro rule on 2 and 5           |
     ---------------------------------------------------------
7. neg x                 neg intro rule on 2 -- 6
     ------------------------------------------------
8.   | x                 assumption                |
9.   | bot               bot intro rule on 7 and 8 |
10.  | neg x             bot elim rule on 9        |
     ------------------------------------------------
11.x -> neg x            impl intro rule on 8 to 10
```

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

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

6. The Resolution Proof System

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

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

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

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

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

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

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

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

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

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

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

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

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

and similarly *or introduction* is useless as well. Therefore, if $\psi \vdash c$, then c must have been a clause of ψ. Therefore, every clause in φ must also be present in ψ.

Hence the clauses of ψ and φ must be the same, ie ψ and φ must be the same, meaning that the CNF obtained is unique.

7. A flavour of DFAs

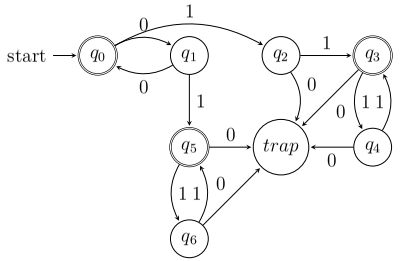
Consider a DFA M with the set of states Q, alphabet $\Sigma = \{0, 1\}$, transition function δ, and accepting state F.

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

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

Solution:

The state diagram for the DFA M is as follows:



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

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

Explanation for the states:

- q0: Even no of 0's
- q1: Odd no of 0's
- q2: Odd no of 1's after even 0's
- q3: Even no of 1's
- q4: Odd no of 1's
- q5: Odd no of 1's after odd 0's
- q6: Even no of 1's
- trap: If 0's seen after 1's

8. More solved problems on DFA

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

CS208 Practice Problem Set 2

1. Let $\Sigma = \{0, 1\}$.

   (a) Construct DFAs for each of the following languages. Try to use as few states as you can in your construction. In the following $n_i(w)$ denotes the count of i's in the string w, for $i \in \Sigma$. Similarly, $|w|$ denotes the length of the string w.

      i. $L_1 = \{0^m 1^n \mid m + n = 0 \mod 2 \text{ and } m = 0 \mod 3\}$. Think of L_1 as the set of all even length strings of 0s followed by 1s, where the count of 0s is a multiple of 3.

      ii. $L_2 = \{0^m 1^n 0^k \mid m, n, k > 0 \text{ and } m + n + k = 0 \mod 2\}$. Think of L_2 as the set of all strings obtained by inserting a block of 1s inside a block of 0s such that the length of the overall string becomes even.

      iii. $L_3 = \{w \in \Sigma^* \mid w = u \cdot v, \text{ where } u, v \in \Sigma^* \text{ and } n_0(u) = 0 \mod 2, n_1(v) = 0 \mod 2\}$. Think of L_3 as the set of all strings that can be cut into two parts and given to two applications, one of which expects an even count of 0s while the other expects an even count of 1s.

      iv. $L_4 = \{w \in \Sigma^* \mid |w| + 2.n_1(w) = 0 \mod 3\}$. Think of L_4 as the set of all strings whose length would be a multiple of 3 if we simply repeated each 1 in the string thrice (without caring about the 0s).

   (b) Construct NFAs for each of the following languages. Feel free to use ϵ-transitions. The purpose of constructing NFAs is really to capture the intuitive structure of strings in the language in as direct a way as possible. Feel free to re-use DFAs constructed in the previous part of this question as building blocks of your NFAs.

      i. $L_5 = \{w \in \Sigma^* \mid w = u \cdot v \cdot x, \text{ where } u, v, w \in \Sigma^* \text{ and } |u| + 2.|v| + 3.|x| = 0 \mod 4\}$. You can think of a string being

broken into three parts and fed to three applications, such that the first (resep. second and third) application charges Re. 1 (resp. Rs. 2 and Rs. 3) to process each letter in the string. Then L_5 is the set of strings that can be processed by spending a muliple of 4 Rupees.

ii. $L_6 = \{w \in \Sigma^* \mid w = u \cdot v$, and either $u \in L_1, v \in L_2$ or $u \in L_2, v \in L_1\}$. Thus, L_6 is the set of all strings that can be broken into two parts, such that the first part belongs to L_1 and the second part belongs to L_2, or vice versa.

iii. $L_7 = \{w \in \Sigma^* \mid u_1 \cdot u_2 \cdot \ldots \cdot u_k$, where $k > 0, k = 0 \mod 2$, and $u_i \in L_2$ for all $i\}$. Thus, L_7 is the language of all strings that can be broken up into an even number of strings from L_2.

iv. $L_8 = \{w \in \Sigma^* \mid n_1(w') = 0 \mod 2$, where w' is obtained from w by replacing every alternate letter starting from the second letter by $\varepsilon\}$. Thus if $w = 0101001$, then $w' = 0001$. You can think of the string w as a sequence of bits coming in so fast that a machine can only read every alternate bit. L_8 is then the sequence of strings that can be accepted by such a lossy automaton for L_2.

2. Suppose r_1 and r_2 are regular expressions over the same alphabet Σ. We say $r_1 = r_2$ to denote equality of the languages represented by r_1 and r_2. In other words, every string in the language represented by r_1 is also included in the language represented by r_2 and vice versa. For each of the following pairs of regular expressions over $\Sigma = \{0, 1\}$, either prove that they represent the same language, or give a string that is present in the language of one but not in the language of the other. In the latter case, you must also describe why your solution string is in the language of one regular expression, but not in that of the other.

   (a) $r_1 = \mathbf{1}^*(\mathbf{1} + \mathbf{0})^*\mathbf{0}^*$ and $r_2 = (\mathbf{0}^*\mathbf{1}^*)^*$

   (b) $r_1 = ((\mathbf{0} + \mathbf{1})^*\mathbf{0})^*\mathbf{0}$ and $r_2 = (\mathbf{0} + \mathbf{1})^*\mathbf{0}^*\mathbf{0}$

   (c) $r_1 = (\mathbf{0} + \mathbf{1})^*\mathbf{01}(\mathbf{0} + \mathbf{1})^*$ and $r_2 = \mathbf{1}^*(\mathbf{0} + \mathbf{1})^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*\mathbf{1}$

3. Give as small an upper bound as you can of the number of *distinct* languages over $(\mathbf{0} + \mathbf{1})^*$ that can be recognized by DFAs using at most n states. Your answer must be a function of n. You must also clearly explain your reasoning.

4. The solution to this problem is specific to your roll number. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, B\}$. We will say a string $w \in \Sigma^*$ is *embedded* in a string $u \in \Sigma^*$ iff w can be obtained from u by replacing some letters in u with ε. Thus, 014 is embedded in 203421049. To see why this is so, notice that 014 can be obtained as $\varepsilon 0 \varepsilon\varepsilon 1 \varepsilon 4 \varepsilon$. However, 014 is not embedded in 23421049.

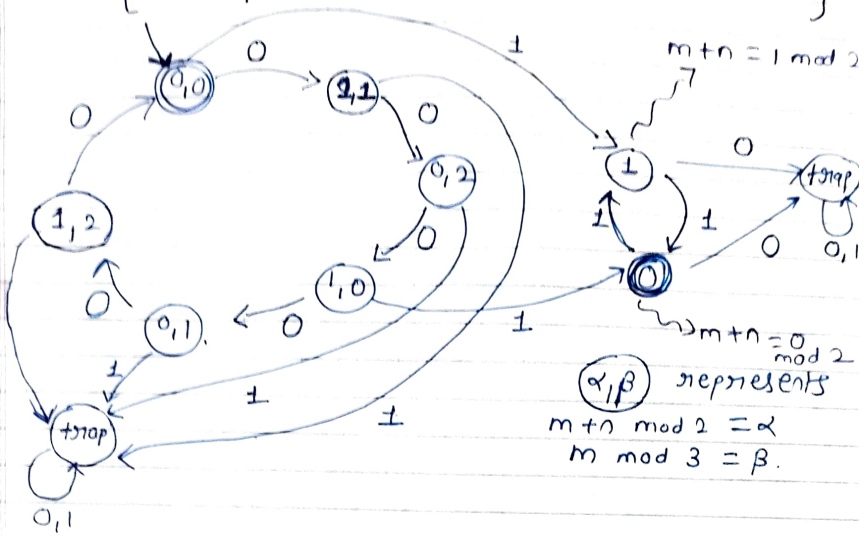
   (a) State your roll no. as a string in Σ^*. Let us call this string ρ.

   (b) Give a DFA with no more than $|\rho| + 1$ states that accepts the language $L_\rho = \{w \mid w \in \Sigma^*, \rho$ is embedded in $w\}$. You must explain what each state in your DFA represents (use a couple of lines of explanation per state), as evidence that you understood the construction of the DFA. Your answer will fetch 0 marks without proper explanation per state.

   Note: You can of course start with a NFA, and then use the subset construction to convert it to a DFA. But this is a long and tedious route, and will not give you much intuition to understand what each state in the resulting DFA represents. So you are strongly advised not to follow this route. There is enough structure in the problem to permit a much simpler construction of a DFA, and you are encouraged to think about it.

1 (i)

$\{0^m 1^n \mid m+n = 0 \bmod 2 \quad m = 0 \bmod 3\}$



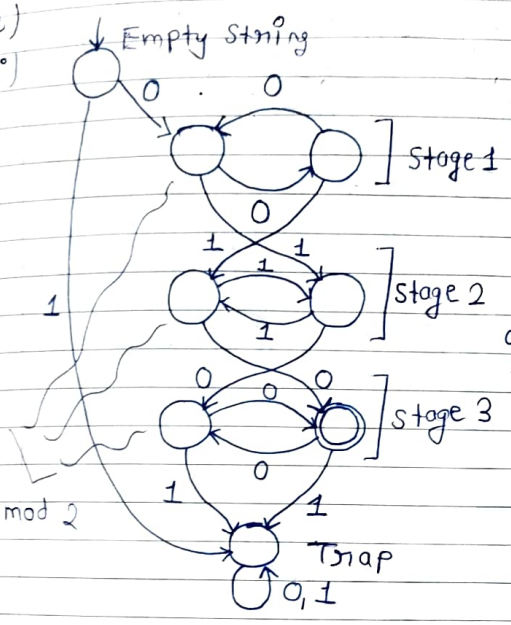
idea

when first '1' comes, check that remainder of m w.r.t is 3 is 0. If after '1' we get '0', go to trap.

problem set - 2

(a)

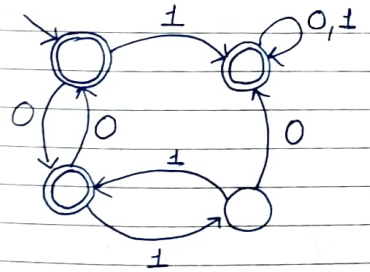
(ii)



Intuition:-

We keep record of the stage in which the string is. Stages are ① only zeros as of now ② zeros followed by ones ③ zeros followed by ones, followed by zeros. At the same time we also store the modulo w.r.t 2.

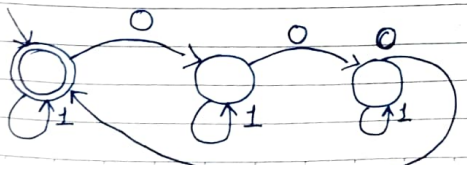
(iii)



Idea:-

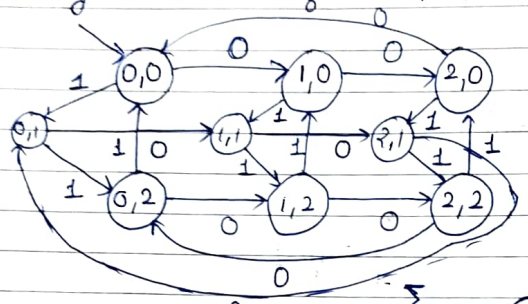
If the number of 1's is 0 mod 2 in the whole string, it's accepted. for odd i.e 1's are 1 mod 2, we have to ensure a prefix with even 0's and odd 1's.

(iv)



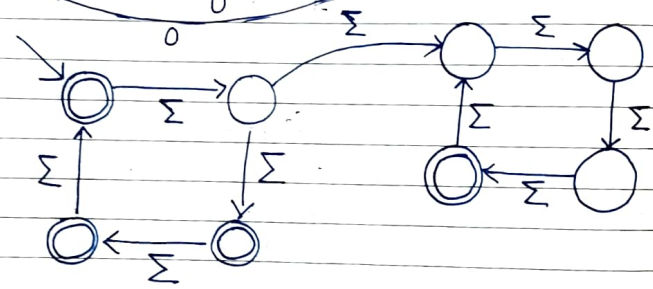
Idea: Just keep record of $n_0(w) \bmod 3$.

for general $\alpha n_0(w) + \beta n_1(w) = k \bmod 3$

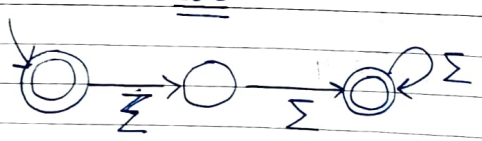


choose those accepting states for which the above is true.

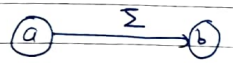
(b)
(i)



Or



Here



means

for all letters there is outgoing edge from a to b.

Idea:-

Only string "a" where a is any letter from Σ are not acceptable. or simply strings of length 1.

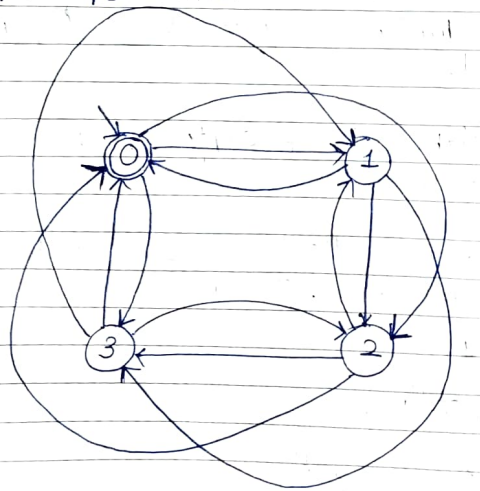
Alternative to b(i)

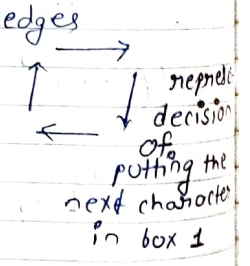
8. we can think of this as a problem where we want to find if there is a to divide total number of characters into three parts such that

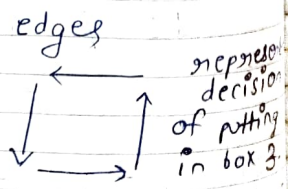
$$|u| + 2|v| + 3|x| = 0 \bmod 4.$$

cardinality of respective parts

for every character we have a choice to keep it in any of the three parts. Here is a NFA for this.

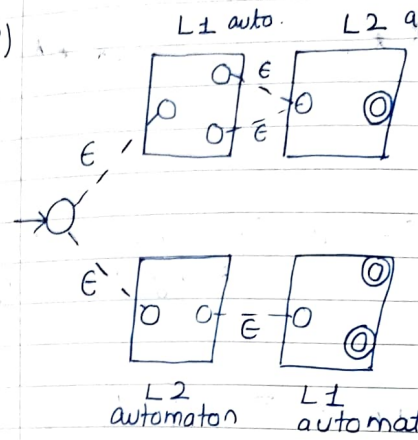




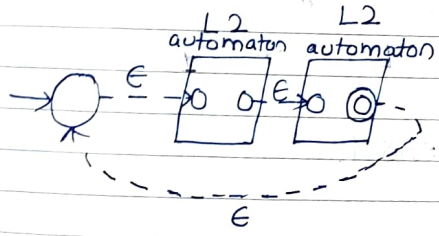


represents decision of putting in box 2.

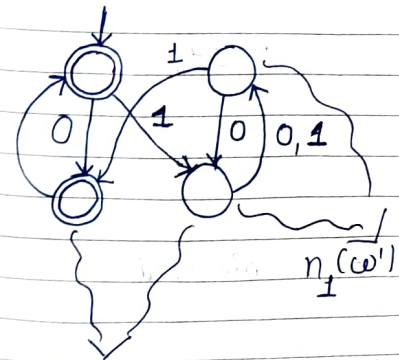
(ii)



(iii)



(iv)



idea :- we have to ignore every alternate character. We also need to keep record of $n_1(\omega')$ mod 2.

this state mean next character has to be avoided.

2(a)

$r_1 = 1^*(1+0)^*0^*$ $\quad r_2 = (0^*1^*)^*$

r_1

$(1+0)^*$ is universal (i.e. any finite string belongs to its langu

why?

$1+0 \Rightarrow \{0,1\}$

So

$(1+0)^* \Rightarrow \Sigma^*$ where $\Sigma = \{0,1\}$

r_1 is $1^*(1+0)^*0^*$

consider any string(finite) α

So

$\epsilon \cdot \alpha \cdot \epsilon$ ϵ Lang of r_1

Hence

r_1 is universal.

r_2 $\quad L(0^*) = \{\epsilon, 0, 00, \ldots\}$

$L(1^*) = \{\epsilon, 1, 11, \ldots\}$

$0 \in L(0^*1^*)$

$1 \in L(0^*1^*)$

So

$L((0^*1^*)^*)$ is also universal.

$L(r_1) = L(r_2)$

(b) $r_1 = ((0+1)^*0)^*0$

$r_2 = (0+1)^*0^*0$

$10 = 1 \cdot \epsilon \cdot 0 \in r_2$

r_1 $\underbrace{((0+1)^*0)}^{*}$

explicitly added/concatenated a zero. so all strings which belongs to $L((0+1)^*0)$ has zero at the end.

$\alpha \in L((0+1)^*0)^*$ either is empty string or ends with 0.

So $\alpha \in L(r_1)$ either is "0" or ends with atleast two zeros. [mean .....00]

$10 \notin L(r_1)$

$L(r_1) \neq (r_2)$

(c) $r_1 = (0+1)^*01(0+1)^*$

$r_2 = 1^*(0+1)^*0(0+1)^*1$

Every string which belongs to $L(r_2)$ ends with '1'.

$010 = \epsilon \cdot 0 \cdot 1 \cdot 0 \in L(r_1)$

$0 \cdot 1 \cdot 0 \notin L(r_2)$

$L(r_1) \neq (r_2)$

We have to know a good upper bound on the number of distinct regular languages that can be represented by automaton of size(states) $\leq n$. (DFA)

$\Sigma = \{0,1\}$

We will use a scheme to generate such DFAs. Consider a n States which are labelled. for now assume $n \geq 3$.

$S_1, S_2, \ldots, S_n$.

Now, for every state we have to decide the state for each letter. arriving

$\Sigma = \{0,1\}$

~~So~~ for each letter, n options. So

$n \times n = n^2$ ways of making outgoing arrows, from state S_i. Hence we will get

$\underbrace{n^2 \cdot n^2 \cdot n^2 \ldots}_{n} = n^{2n}$ different DFAS which could be possibly equivalent

But DFA is complete when we mention the start and accepting states. Consider the Scheme below:

for every graph ~~DFA~~ formed in this way keep S_1 as start state. Now for each such graph make $(n-1 + n-1)$ copies.

for first n-1 copies mark

for next $n-1$ copies.

mark

$\{S_1\}, \{S_1, S_n\}, \{S_1, S_n, S_{n-1}\}, \ldots \{S_1, S_n, \ldots S_2\}$

as accepting.

thus we have

$n^{2n}(2)(n-1)$ DFAs, need not be language wise different, but each DFA represent some language, giving us multi set of languages.

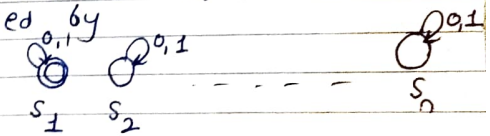
To show that valid $2(n-1)(n^{2n})$ is an upper bound, it is sufficient to show that language represented by any DFA with states $\leq n$ is included in our collection.

~> if $<n$ then add $0,1$ to graph.

Take any DFA, states $\leq n$. If there are x starting & accepting states, label them $S_n, S_{n-1}, \ldots, S_{n-(x-1)}$ in any order. Always label the starting state as S_1. Rest all states can be labelled in any way. It is not difficult to see that one such DFA can be formed as per our scheme.

x are after excluding the accepting state which is also start state.

If $x == n-1$, then the DFA represent $(1+0)^*$ which is covered by

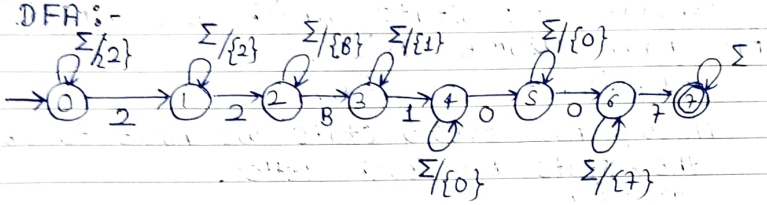


Thus $2(n-1)n^{2n}$ is valid upper bound for $n \geq 3$ for $n = 1, 2$, reader can get the exact

(a) Rollno. - 22B1007 (As an example)

(b) Basically, we have to come up with DFA which accepts any string in which "22B1007" is present as subsequence.

DFA :-



This DFA is basically the implementation of the greedy algorithm for checking if a string is present as a subsequence in another string.

Keep looking for first character, if found then start looking for next character. Repeat this till you find the last character.

Suppose it is present, then we can show that the string will reach the accepting state.

$\ldots 2 \ldots 2 \ldots B \ldots 1 \ldots 0 \ldots 0 \ldots 7 \ldots$

by this moment, the DFA must be in state ≥ 3

After this it must be in ≥ 4

Inverse :

Suppose it could reach the accepting state which means we have used the edges (that's only way to go to it). Hence 22B1007 in this sequence has appeared in the string w.

Roll No.: _____

CS 208 : Automata Theory and Logic Spring 2024

Quiz 1

Time: 21:00 - 23:30 *Max Marks:* 60

Instructions:

- ***Please write your roll number on all pages in the space provided at the top.***
- *Be brief, complete, and stick to what has been asked.*
- ***You must write your answer for every question only in the space allocated for answering the question. Answers written outside the allocated space risk not being graded.***
- ***You can use an extra answer book for rough calculations.***
- ***You must submit this question+answer book in its entirey along with any extra answer book for rough calculations (if you used one).***
- *Untidy presentation of answers, and random ramblings will be penalized by negative marks.*
- *Unless asked for explicitly, you may cite results/proofs covered in class without reproducing them.*
- ***If you need to make any assumptions, state them clearly.***
- ***Do not copy solutions from others. All detected cases of copying will be reported to DADAC with names and roll nos. of all involved. The stakes are high if you get reported to DADAC, so you are strongly advised not to risk this.***

1. **We love NNFs!** [10]

   Let P, Q, R be propositional variables.

   (a) Convert the formula $\big((P \to (Q \to R)) \to \neg(P \to (R \to Q))\big)$ to a semantically equivalent formula in Disjunctive Normal Form (DNF). Do not include cubes that contain both a literal and its negation in your DNF formula. [5]
   You must show all intermediate steps. Answers without steps will fetch no marks.

   Solution:

   $$\begin{aligned}
   & \big((P \to (Q \to R)) \to \neg(P \to (R \to Q))\big) \\
   \Leftrightarrow\ & \neg(P \to (Q \to R)) \vee \neg(P \to (R \to Q)) && \text{... Semantics of } \to \\
   \Leftrightarrow\ & \neg(\neg P \vee (\neg Q \vee R)) \vee \neg(\neg P \vee (\neg R \vee Q)) && \text{... Semantics of } \to \\
   \Leftrightarrow\ & (P \wedge \neg(\neg Q \vee R)) \vee (P \wedge \neg(\neg R \vee Q)) && \text{... DeMorgan's Law and } \neg\neg \text{ elim} \\
   \Leftrightarrow\ & (P \wedge (Q \wedge \neg R)) \vee (P \wedge (R \wedge \neg Q)) && \text{... DeMorgan's Law and } \neg\neg \text{ elim} \\
   \Leftrightarrow\ & (P \wedge Q \wedge \neg R) \vee (P \wedge R \wedge \neg Q) && \text{... Simplify (using associativity)}
   \end{aligned}$$

   (b) Convert the formula $\big(\neg(P \vee (\neg Q \wedge R)) \to (\neg P \wedge (Q \vee \neg R))\big)$ to an equisatisfiable Conjunctive Normal Form (CNF) formula using Tseitin encoding. Do not include clauses that contain both a literal and its negation in your CNF formula. [5]
   You must NOT simplify the given formula or check its satisfiability before applying Tseitin encoding. You must show all intermediate steps. Answers without steps or obtained after simplifying the given formula or after checking its satisfiability will fetch no marks. Answers that give an equisatisfiable formula without using Tseitin encoding will also fetch no marks.

Solution: We first introduce a fresh variable t_i for each sub-formula that is neither a variable nor its negation.

$$\begin{aligned}
& (t_1 \leftrightarrow (\neg Q \wedge R)) \wedge \\
& (t_2 \leftrightarrow (P \vee t_1)) \wedge \\
& (t_3 \leftrightarrow \neg t_2) \wedge \\
& (t_4 \leftrightarrow (Q \vee \neg R)) \wedge \\
& (t_5 \leftrightarrow (\neg P \wedge t_4)) \wedge \\
& (t_6 \leftrightarrow (t_3 \to t_5)) \wedge \\
& t_6
\end{aligned}$$

Next, we expand each bi-implication into a conjunction of two implications.

$$\begin{aligned}
& (t_1 \to (\neg Q \wedge R)) \wedge (t_1 \leftarrow (\neg Q \wedge R)) \wedge \\
& (t_2 \to (P \vee t_1)) \wedge (t_2 \leftarrow (P \vee t_1)) \wedge \\
& (t_3 \to \neg t_2) \wedge (t_3 \leftarrow \neg t_2) \wedge \\
& (t_4 \to (Q \vee \neg R)) \wedge (t_4 \leftarrow (Q \vee \neg R)) \wedge \\
& (t_5 \to (\neg P \wedge t_4)) \wedge (t_5 \leftarrow (\neg P \wedge t_4)) \wedge \\
& (t_6 \to (t_3 \to t_5)) \wedge (t_6 \leftarrow (t_3 \to t_5)) \wedge \\
& t_6
\end{aligned}$$

Next, we use the semantics of $\to$ (or $\leftarrow$) to get

$$\begin{aligned}
& (\neg t_1 \vee (\neg Q \wedge R)) \wedge (t_1 \vee \neg(\neg Q \wedge R)) \wedge \\
& (\neg t_2 \vee (P \vee t_1)) \wedge (t_2 \vee \neg(P \vee t_1)) \wedge \\
& (\neg t_3 \vee \neg t_2) \wedge (t_3 \vee \neg\neg t_2) \wedge \\
& (\neg t_4 \vee (Q \vee \neg R)) \wedge (t_4 \vee \neg(Q \vee \neg R)) \wedge \\
& (\neg t_5 \vee (\neg P \wedge t_4)) \wedge (t_5 \vee \neg(\neg P \wedge t_4)) \wedge \\
& (\neg t_6 \vee (\neg t_3 \vee t_5)) \wedge (t_6 \vee \neg(\neg t_3 \vee t_5)) \wedge \\
& t_6
\end{aligned}$$

Next, we use DeMorgan's laws and $\neg\neg$ elimination to get

$$\begin{aligned}
& (\neg t_1 \vee (\neg Q \wedge R)) \wedge (t_1 \vee (Q \vee \neg R)) \wedge \\
& (\neg t_2 \vee (P \vee t_1)) \wedge (t_2 \vee (\neg P \wedge \neg t_1)) \wedge \\
& (\neg t_3 \vee \neg t_2) \wedge (t_3 \vee t_2) \wedge \\
& (\neg t_4 \vee (Q \vee \neg R)) \wedge (t_4 \vee (\neg Q \wedge R)) \wedge \\
& (\neg t_5 \vee (\neg P \wedge t_4)) \wedge (t_5 \vee (P \vee \neg t_4)) \wedge \\
& (\neg t_6 \vee (\neg t_3 \vee t_5)) \wedge (t_6 \vee (t_3 \wedge \neg t_5)) \wedge \\
& t_6
\end{aligned}$$

Finally, using distributivity of $\wedge$ over $\vee$ and vice versa, we get the desired Tseitin encoding

$$\begin{aligned}
& (\neg t_1 \vee \neg Q) \wedge (\neg t_1 \vee R) \wedge (t_1 \vee Q \vee \neg R) \wedge \\
& (\neg t_2 \vee P \vee t_1) \wedge (t_2 \vee \neg P) \wedge (t_2 \vee \neg t_1) \wedge \\
& (\neg t_3 \vee \neg t_2) \wedge (t_3 \vee t_2) \wedge \\
& (\neg t_4 \vee Q \vee \neg R) \wedge (t_4 \vee \neg Q) \wedge (t_4 \vee R) \wedge \\
& (\neg t_5 \vee \neg P) \wedge (\neg t_5 \vee t_4) \wedge (t_5 \vee P \vee \neg t_4) \wedge \\
& (\neg t_6 \vee \neg t_3 \vee t_5)) \wedge (t_6 \vee t_3) \wedge (t_6 \vee \neg t_5) \wedge \\
& t_6
\end{aligned}$$

Roll No.: _____

2. **How expressive are you?** [15]

We know that propositional logic formulas are constructed using symbols in the set $\{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ in addition to variables, parentheses and commas (if needed). It turns out that such a large set of symbols may not be needed. For example, the set $S' = \{\wedge, \neg\}$ suffices to construct a formula that is semantically equivalent to any propositional logic formula. Indeed, from DeMorgan's laws we know that $p_1 \vee p_2$ is semantically equivalent to $\neg(\neg p_1 \wedge \neg p_2)$ for every propositional variable (or sub-formula) p_1 and p_2.

Let S be a set of symbols that are used in addition to variables, parentheses and commas (if needed) to construct formulas. We say that S is ***propositionally expressive*** if for every propositional logic formula φ over variables $x_1, \ldots x_n$, there is a semantically equivalent formula over $x_1, \ldots x_n$ that uses only symbols from S in addition to variables, parentheses and commas (if needed). From what we have studied in class, it should be easy for you to see that $\{\wedge, \neg\}$ is propositionally expressive.

For purposes of this question, we define new ternary logic connectives α, β, γ and δ with the following semantics.

- $\alpha(p, q, r)$ evaluates to 1 (true) iff either both p and q are 1 (true) or p is 0 (false) and r is 1 (true). For example, $\alpha(1, 1, 0) = \alpha(0, 0, 1) = 1$ but $\alpha(1, 0, 1) = \alpha(0, 1, 0) = 0$. You can think of $\alpha(p, q, r)$ as intuitively implementing "if p then q else r".
- $\beta(p, q, r)$ evaluates to 1 (true) iff p, q and r all have the same truth value. Thus, $\beta(1, 1, 1) = \beta(0, 0, 0) = 1$, but $\beta(1, 0, 1) = \beta(1, 1, 0) = 0$. You can think of $\beta(p, q, r)$ as intuitively implementing "all of p, q, r are in consensus".
- $\gamma(p, q, r)$ evaluates to 1 (true) iff exactly one of p, q and r has the value 1 (true). Thus, $\gamma(1, 0, 0) = 1$ and $\gamma(1, 1, 0) = \gamma(0, 0, 0) = 0$.
- $\delta(p, q, r)$ evaluates to the same truth value as the majority of p, q and r. Thus, $\delta(1, 1, 0) = 1$ and $\delta(0, 1, 0) = 0$.

Given below are three sets S of symbols used to construct formulas. In each case you must indicate whether S is propositionally expressive, with justification.

You may use the fact that $\{\wedge, \neg\}$ is known to be propositionally expressive. Hence, for "Yes" answers, you need to show that $\neg p$ and $p \wedge q$ can be equivalently expressed using the symbols in S, for any propositional variables (or sub-formulas) p and q. For "No" answers, you must show that there is at least one formula that can be written using $\{\wedge, \neg\}$ but a semantically equivalent formula cannot be written using S.

(a) $S = \{\top, \bot, \alpha\}$ [4]

Solution: To show that S is propositionally expressive, one has to find a semantically equivalent formula constructed using S for every formula constructed using $\{\neg, \wedge\}$. But ***if we can find formulas constructed using S that are equivalent to $\neg p$ and $p \wedge q$***, we can find a formula equivalent to any formula constructed using $\{\neg, \wedge\}$! Why? Simple structural induction : take any formula F constructed using $\{\neg, \wedge\}$. F is either a variable or $\neg G$ or $G_1 \wedge G_2$, where G, G_1, G_2 are formulas constructed using $\{\neg, \wedge\}$.

- Base case: p, $\neg p$ and $p \wedge q$ have semantically equivalent formulas constructed using S.
- Assume: Every sub-formula of F has a semantically equivalent formula constructed using S.

- Induction: Suppose F is $\neg G$ (resp. $G_1 \wedge G_2$). Take the formula for $\neg p$ (resp. $q \wedge r$) constructed using S, and replace p (resp. q, r) with G (resp. G_1, G_2). We have a formula for F constructed using S!

Using the above argument (**this is not necessary to be shown as part of your solution**), we can easily conclude that $S = \{\top, \bot, \alpha\}$ is propositionally expressive. Indeed, $\neg p \Leftrightarrow \alpha(p, \bot, \top)$ and $(p \wedge q) \Leftrightarrow \alpha(p, q, \bot)$.

(b) $S = \{\top, \gamma\}$ [4]

Solution: $\neg p \Leftrightarrow \gamma(p, p, \top)$ and $(\neg p \wedge q) \Leftrightarrow \gamma(p, p, q)$. Hence $(p \wedge q) \Leftrightarrow \gamma(\neg p, \neg p, q)$. Using the equivalence for $\neg p$ already obtained above, we get

$$(p \wedge q) \Leftrightarrow \gamma(\gamma(p, p, \top), \gamma(p, p, \top), q)$$

(c) $S = \{\beta, \delta\}$ [7]

Solution: S is not propositionally expressive. There is a very simple formula that cannot be expressed using S : $\varphi(p) = p \wedge \neg p = \bot$

Consider a formula B constructed using $S = \{\beta, \delta\}$ that has only one variable p. Notice that every internal node of the parse tree of B has 3 children and every leaf node of the parse tree must be p. Furthermore, every leaf node has a parent labeled either β or δ. Given that the parse tree of a formula constructed using S necessarily has finite height, there must be some leaf node with both its siblings also as leaf nodes. For such a leaf node, if its parent is β, then we have $\beta(p, p, p) \equiv \top$. On the other hand, if its parent is δ, we have $\delta(p, p, p) \equiv p$. In order to understand what propositional formula the parse tree represents, We can prune such a leaf node along with its parent and replace it with p or $\top$ accordingly. Notice that this introduces $\top$ as a symbol at a leaf of the modified parse tree, but this doesn't change the semantics of the formula represented by the parse tree.

After the above pruning step, again consider the leaf nodes whose siblings are also leaf nodes. Unlike in the previous case, a leaf node can now be either p or $\top$. However, its parent must still be labeled by either β or γ. Furthermore, we know that $\beta(p, p, \top) \equiv \beta(p, \top, p) \equiv \beta(\top, p, p) \equiv \beta(p, \top, \top) \equiv \beta(\top, p, \top) \equiv \beta(\top, \top, p) \equiv \delta(p, p, \top) \equiv \delta(p, \top, p) \equiv \delta(\top, p, p) \equiv p$ and $\delta(p, \top, \top) \equiv \delta(\top, p, \top) \equiv \delta(\top, \top, p) \equiv \delta(\top, \top, \top) \equiv \top$. So if we again prune the leaf, its parent and siblings, we will either replace it with p or $\top$. We can now inductively argue that by continuing this process, the entire parse tree will finally be replaced by p or $\top$. Therefore, the parse tree cannot represent a formula that is semantically equivalent to $\varphi(p) = p \wedge \neg p = \bot$.

Notice that by the same argument, we can't construct a formula semantically equivalent to $\neg p$ either using S.

However, if we expand S to be $\{\bot, \beta, \gamma\}$, we can easily express $\neg p$ as $\beta(p, p, \bot)$ and $p \wedge q$ as $\beta(p, q, \beta(p, p, p))$, where $\beta(p, p, p)$ is equivalent to $\top$. In fact, we don't need γ to be in S at all if $\bot$ and β are in S!

Roll No.: _____

3. **Shipping with SAT** 10

A shipping company has n cargo containers that must be transported via ships. Let $C = \{c_1, \ldots, c_n\}$ denote the containers. The company has m ships; let $S = \{s_1, \ldots s_m\}$ denote these ships. It turns out that not every ship can transport every container. Let $A_i \subseteq C$ be the set of containers that ***are allowed*** to be transported on ship i. Furthermore, each ship s_i has a ***maximum limit*** of l_i containers that it can transport. All l_i's are assumed to be non-negative integers.

The shipping company wants to find a set X of at most k ($0 < k \leq m$) ships that can be used to transport all n containers, while loading each ship only with containers that are allowed on the ship, and without overloading each ship beyond the maximum number of containers it can transport.

As an example, consider $n = 4$, $m = 5$ and $l_1 = l_2 = l_4 = 1, l_3 = l_5 = 2$. Furthermore, suppose $A_1 = \{c_1, c_2\}$, $A_2 = \{c_2, c_3, c_4\}$, $A_3 = \{c_1, c_2, c_4\}$, $A_4 = \{c_2\}$ and $A_5 = \{c_2, c_4\}$. In this example, it is impossible to transport all 4 containers on only 2 ships. However, it is possible to transport all of them on 3 ships. For example, s_1 can be used to transport c_1, s_2 can be used to transport c_3 and s_5 can be used to transport c_2 and c_4. Hence $X = \{s_1, s_2, s_5\}$ is one possible solution the shipping company seeks.

We wish to use a satisfiability checker for NNF formulas to help the shipping company. Specifically, you must construct a propositional NNF formula φ such that

- Given n, m, k, $l_1, l_2, \ldots l_m$ where $0 \leq l_j \leq n$ for each $j \in \{1, \ldots m\}$, and the sets $A_1, A_2, \ldots A_m$, the formula φ can be constructed in time polynomial in m, n and k.
- There is a bijection between satisfying assignments of φ and distinct choices X of at most k ships that can transport all n containers, while respecting each ship's constraints. Note that this means φ must be unsatisfiable if it is impossible to transport all containers in at most k ships.

To construct the above formula, we will use propositional variables x_i for each $i \in \{1, \ldots m\}$, such that that x_i is true iff ship i is included in the set X of chosen ships.

You are free to use auxiliary propositional variables as you consider necessary. However, you must indicate the interpretation (what does the variable represent) for each such auxiliary variables. You are also free to use the cardinality constraints of the form $\sum_{p=u}^{v} b_p \leq w$ for propositional variables $b_u, \ldots b_v$ where $u \leq v$ and $0 \leq w \leq (v - u + 1)$. We have already discussed in Tutorial 1 how such a cardinality constraint can be encoded as a NNF formula in time polynomial in $(v - u)$ and w, possibly with the use of auxiliary propositional variables. Hence, if you are using such cardinality constraints, you don't need to explicitly write the NNF formula corresponding to $\sum_{p=u}^{v} b_p \leq w$, but can simply use $\left(\sum_{p=u}^{v} b_p \leq w\right)$ as a proxy for the NNF formula.

Solution: Let the propositional variable x_i ($1 \leq i \leq m$) encode "ship $s_i \in X$", as required by the question. Furthermore, let propositional variable $t_{i,j}$ ($1 \leq i \leq m, 1 \leq j \leq n$) encode "container c_j is loaded in ship s_i".

The overall NNF formula is obtained as $\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, where

- φ_1 is $\left(\sum_{i=1}^{m} x_i \leq k\right)$. This encodes that at most k ships are chosen to be in the solution set X.
- φ_2 is $\bigwedge_{j=1}^{n} \bigwedge_{i\,:\,c_j \notin A_i} \neg t_{ij}$. This encodes that for each container c_j and for each ship s_i such that c_j is not allowed on s_i, the variable t_{ij} must be false.

- φ_3 is $\bigwedge_{j=1}^{n} \left(\bigvee_{i\,:\,c_j \in A_i} \left(x_i \wedge t_{ij} \wedge \bigwedge_{k\,:\,c_j \in A_k, k \neq i} \neg t_{kj}\right)\right)$. This encodes that for each container c_j, there is exactly one ship s_i such that $s_i \in X$ and $c_j \in A_i$ for which t_{ij} is true. All other ships s_k such that $c_j \in A_k$ must have t_{kj} false. In other words, every container must be in exactly one ship in X that is allowed to transport the container.
- φ_4 is $\bigwedge_{i=1}^{m} \left(x_i \rightarrow \left(\sum_{j=1}^{n} t_{ij} \leq l_i\right)\right)$. This encodes that for each ship s_i, if it is chosen to be in X, the total count of containers loaded on s_i can be no more than l_i.

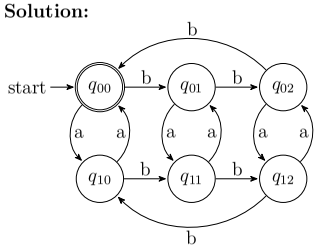
There are some variations of the above formulation that also serve the purpose of the question. Think about if a satisfying assignment of $\varphi_1 \wedge \varphi_3 \wedge \varphi_4$ would also serve the purpose of the shipping company. What would be the interpretation of t_{ij} in this case?

4. **I think I saw you in Tuesday's lecture** [10]

Draw a Deterministic Finite Automaton (DFA) for each of the following languages.

(a) $\mathcal{L} := \{w \in \{a,b\}^* : 2 \text{ divides } n_a(w) \text{ and } 3 \text{ divides } n_b(w)\}$. 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, $ababbaa \in \mathcal{L}$ but $aabababa \notin \mathcal{L}$. [5]

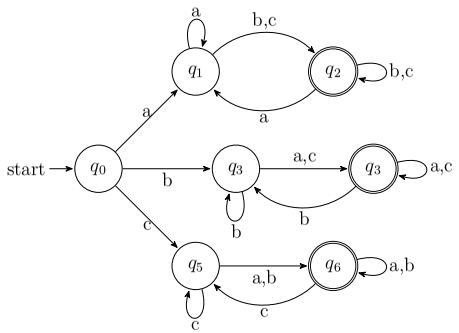
Solution:



State q_{ij} represents "Word w seen so far has $n_a(w) \mod 2 = i$ and $n_b(w) \mod 3 = j$.

(b) $\mathcal{L} := \{w \in \{a,b,c\}^* :$ first and last letters of w are different$\}$. For example, $abbacb \in \mathcal{L}$ but $cbbabac \notin \mathcal{L}$. [5]

Solution:



State	Interpretation
q_0	Haven't seen any letter
q_1	Start and end letters a
q_2	Start letter a, end letter b or c
q_3	Start and end letters b
q_4	Start letter b, end letter a or c
q_5	Start and end letters c
q_5	Start letter c, end letter a or b

5. **To CNF or to DNF** Define the *length* of a CNF (or DNF) formula as the total number of all literals over all clauses (or all cubes, respectively) in the formula. For example, consider the CNF formula $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4)$. This formula has length $2 + 2 + 3 = 7$. Siimilarly, the length of the DNF formula $\psi = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3 \wedge \neg x_4) \vee (\neg x_2 \wedge x_3 \wedge x_4)$ is $2 + 3 + 3 = 8$. [15]

Show that there is a family of formulas $\mathcal{F} = \{\varphi_n \mid n \in \mathbb{N}\}$ such that

- Every φ_n is a DNF formula over $\mathcal{O}(n)$ propositional variables.
- Every φ_n has length in $\mathcal{O}(n)$.
- For every φ_n, there exists **no semantically equivalent** CNF formula ψ_n (over the same variables as φ_n) such that the length of ψ_n grows polynomially with n.

In order to answer this question, you must (a) clearly write the DNF formula φ_n, (b) show that its length is in $\mathcal{O}(n)$, and (c) prove that there exists **no semantically equivalent** CNF formula of length polynomial in n.

In order to score marks in this question, all three parts must be answered correctly.

Note: A formula has polynomial length if and only if $length \in \mathcal{O}(f(n))$ where f is some polynomial in n. You **MUST** prove why φ_n can't be equivalently represented by any polynomial length CNF formula.

Solution:

Note: This is not the only possible solution. There are alternative solutions as well.

Consider the family $\mathcal{F} = \{\varphi_n \mid n \in \mathbb{N}\}$, where

$$\varphi_n = (X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee \cdots \vee (X_n \wedge Y_n)$$

Clearly φ_n is in DNF and has $2n$ variables. Furthermore, its length is $2n$. We now show below that there exists no semantically equivalent CNF formula ψ_n such that the length of ψ_n grows polynomially in n.

Claim 1: Each clause of the CNF formula ψ_n must contain either X_i or Y_i as literals, for every $i \in \{1, \ldots n\}$.

Suppose, if possible, there is a clause C_j that does contains neither X_i nor Y_i as literal. Consider the assignment where X_i and Y_i have value 1, and all literals in C_j have value 0. Under this assignment, φ_n evaluates to 1 (recall φ_n has a cube $X_i \wedge Y_i$), but ψ_n evaluates to 0 as C_j evaluates to 0. Therefore, φ_n is not semantically equivalent to ψ_n – a contradiction! Hence, every clause C_j must have either X_i or Y_i as literal, for every $i \in \{1, \ldots n\}$.

Given Claim 1, the clauses in ψ_n can be divided into two categories: (a) those that have both X_i and Y_i as literals for some $i \in \{1, \ldots n\}$, and (b) those that have either X_i or Y_i, but not both, as literals for every $i \in \{1, \ldots n\}$.

Let us focus on clauses of type (b).

Claim 2: For every tuple of literals in $\{X_1, Y_1\} \times \{X_2, Y_2\} \times \cdots \times \{X_n, Y_n\}$, there is a clause of type (b) in ψ_n that contains the literals in the tuple.

Suppose, if possible, there is a tuple of literals in the Cartesian product such that there is no clause of type (b) in ψ_n that contains the literals in this tuple. Now consider the

assignment that sets all variables in this specific tuple to 1 and all other variables to 0. Thus, this assignment sets exactly one of $\{X_i, Y_i\}$ to 0 and exactly one of them to 1, for every $i \in \{1, \dots n\}$. Since by Claim 1, every clause in ψ_n has either X_i or Y_i as a literal, for every $i \in \{1, \dots n\}$, it follows that every clause in ψ_n, and hence ψ_n itself, evaluates to 1 under this assignment. However, clearly φ_n evaluates to 0 under this assignment, since for every $i \in \{1, \dots n\}$, at least one of X_i and Y_i is 0. Hence, φ_n is not semantically equivalent to ψ_n – a contradiction! It follows that for every tuple of literals in the Cartesian product, the corresponding literals must be present in a type (b) clause of ψ_n.

Since there are 2^n distinct tuples in $\{X_1, Y_1\} \times \cdots \times \{X_n, Y_n\}$, Claim 2 shows that there are at least 2^n clauses of type (b) in ψ_n. Hence the length of ψ_n is at least 2^n.

CS208 Tutorial Solutions

2024

Contents

Tutorial 1

1. Determine if the following formulae are tautology.
   a) $(p \to q) \wedge (q \to r) \to (p \to r)$
   b) $(p \to q) \wedge (q \to r) \to (r \to p)$
   c) $(p_1 \wedge p_2 \dots p_n) \to (p_1 \vee p_2 \dots p_n)$

Solution:

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

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

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

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

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

2. Consider a set S of size n. Recall that for a relation over S to be a partial order we require the following to hold:
   i Reflexive: $x \preceq x$ for all $x \in S$
   ii Transitive: $x \preceq y$ and $y \preceq z$ implies $x \preceq z$ for all $x, y, z \in S$
   iii Antisymmetric: $x \preceq y$ and $y \preceq x$ implies $x = y$ for all $x, y \in S$.

   Moreover, $x \in S$ is called a *maximal* element in $\preceq$ if $x \preceq y$ holds only for $y = x$.
   Consider n^2 propositional variables $\{p_{ij}\}_{1 \le i,j \le n}$ for an enumeration of $S = (x_1, x_2, \dots, x_n)$ where p_{ij} is set to 1 iff $x_i \preceq x_j$.
   Give a formula over these variables which evaluates to $\top$ iff $\preceq$ is a partial order. Also give a formula which evaluates to $\top$ only when $\preceq$ has a maximal element.

Solution: For reflexivity the formula is

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

For transitivity, the formula is

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

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

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

Antisymmetric nature is captured by

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

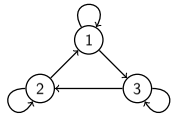
The final formula to check for a partial order then is: $R \wedge T \wedge A$.
If say x_i is a maximal element, we have $\bigwedge_{1 \le j \le n, j \ne i} \neg p_{ij}$.
Hence, "has a maximal element" is encoded as

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

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

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

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



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

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

$$\sum_{i=1}^{n} x_i \leq k \tag{1}$$

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

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

Solution:

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

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

$$\text{[fillable holes]} \quad \bigwedge_{i \in [k]} \bigwedge_{j \in [n]} (p_{ij} \rightarrow x_j)$$

$$\text{[only one pigeon per hole]} \quad \bigwedge_{\substack{i_1, i_2 \in [k] \\ i_1 \neq i_2}} \bigwedge_{j \in [n]} \neg (p_{i_1 j} \wedge p_{i_2 j})$$

$$\text{[each pigeon assigned to } \textit{at least } 1 \text{ hole]} \quad \bigwedge_{i \in [k]} \bigvee_{j \in [n]} p_{ij}$$

Let f be the conjunction of these clauses, then:

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

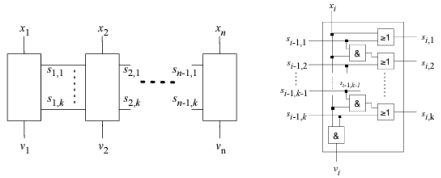


Figure 1: Logic Gate

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

Solution: Let us use the following three propositional variables

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

Let us encode whatever we know

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

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

- at least one of the three statements is true:

$$r \vee \neg b$$

- at least one of the three statements is false:

$$\neg r \vee b$$

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

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

Tutorial 2

---

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

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

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

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

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

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

   Transformation steps

   1. Eliminate Implications:

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

   2. Apply De Morgan's Laws:

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

   Transformation into DNF
   Distribute Conjunction over Disjunction:

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

   Transformation into CNF
   Distribute Disjunction over Conjunction:

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

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

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

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

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

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

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

   Solution: Let

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

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

   1. Suppose $n_i < n$ for some i. Negate the entire formula to convert the CNF into a DNF. Now, choose an assignment of literals in the i^{th} cube (of the DNF) such that the cube, and hence the whole DNF formula, evaluates to 1. Now, consider a variable v, which doesn't appear in the i^{th} cube, and flip its value. The LHS then becomes 0. However, the i^{th} clause stays 1, leading to a contradiction.

   2. Once again, consider the DNF. Note that a conjunction of n literals is satisfied by a unique assignment of variables, and thus, the clauses in the DNF actually encode the assignments that satisfy the formula. Since PARITY is satisfied by 2^{n-1} clauses, we're done.

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

   - For any assignment to the f_is which makes F evaluate to true there exists an assignment to the g_is such that G evaluates to true under this assignment with the same assignment to the f_is.

     $$AND$$

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

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

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

   1. Let's try using 'selector variables' to construct another formula which is strongly-equisatisfiable with C_k. The idea is to use these selector variables to 'select' which literals in our clause are allowed to be false. Use k selector variables $s_1, s_2, s_3 \cdots s_k$ and write a CNF formula that is **strongly-equisatisfiable with** C_k, such that each clause has size $O(k)$.
      [Hint: If you consider the disjunction of $\tilde{s}_1 \vee \tilde{s}_2 \vee \tilde{s}_3 \cdots \tilde{s}_k$, where $\tilde{s}_i$ denotes either s_i or $\neg s_i$, it is true in all but one of the 2^k possible assignments to the s_i variables]
   2. What is the number of clauses in the CNF constructed in the above sub-question? How small can you make the clauses by repeatedly applying this technique?
   3. Consider a k-CNF formula with n clauses. When minimizing the clause size using the above technique repeatedly, what is the (asymptotic) blowup factor in the number of clauses? Observe that the factor is almost linear, upto some logarithmic factors.
      [Hint: Apply the above reduction repeatedly and see how much net blowup occurs]
   4. Can you think of a smarter way to do the reduction that only requires linear blowup?
      [Hint: Consider using different and more auxiliary variables (like the 'selector variables' above) to reduce the size of clauses.]

Solution:

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

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

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

2. The size of clauses is now $k + 1$. The number of clauses is 2^k. Thus, we have obtained an exponential reduction in the size of a clause for a linear factor (the initial size was 2^k blowup in number of clauses. We can reduce the size as long as $\lceil \log_2(\text{size}) \rceil + 1 < \text{size}$, which is true as small as size 3. The size CANNOT be reduced below 3 with this method.

3. We obtain logarithmic reduction for linear blowup. Hence, the factor of blowup will look like $k \cdot \log(k) \cdot \log\log(k) \cdot \log\log\log(k) \cdots \log^{\log^*(k)}(k)$. The factors after the k are all logarithmic factors and can be bounded by a polynomial of any degree > 0.

4. Indeed, we do not need to introduce so many selectors for a clause! We can simply use an indicator s_i to indicate that a clause after p_i is satisfied. Thus, the equisatisfiable formula:

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

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

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

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

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

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

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

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

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

Solution:

a) Let $\mathcal{S}$ denote the family of sets of propositional formulae such that $\mathcal{L} = \{L(\sigma) : \sigma \in \mathcal{S}\}$. Consider $\Sigma = \bigcup_{\sigma \in \mathcal{S}} \sigma$. Since each $\sigma \in \mathcal{S}$ is countable, and a countable union of countable sets is countable, Σ is countable as well. Now, for any infinite bitstring w, $w \in L(\Sigma)$ if and only if $w \models F$ for each $F \in \sigma$, for each $\sigma \in \mathcal{S}$ (we abuse notation and denote the assignment corresponding to w by w). This happens if and only if $w \in L(\sigma)$ for each $\sigma \in \mathcal{S}$, ie $w \in \bigcap_{L \in \mathcal{L}} L$.

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

CS208 Tutorial 3: Finite Automata Theory

1. Often, we are required to find ways to accept sets of strings with various properties. We've seen in class that DFAs and NFAs are ways to achieve this, with an equivalence between NFAs and DFAs. In this question and the next, we'll specify some properties of sets of strings and you will be required to construct (small) DFAs/NFAs to accept these sets (or languages). We aren't insisting that you should find the smallest (in terms of number of states) automata, but try to use as few states as you can.

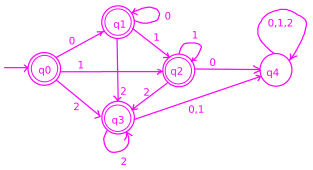
   Let $\Sigma = \{0,1,2\}$. Construct DFAs for recognizing each of the following languages.

   1. $L_1 = \{w \in \Sigma^* \mid w$ doesn't have any pair of consecutive decreasing letters (numbers)$\}$. For example $010 \notin L$ but $00012 \in L$ and $222 \in L$.
   2. $L_2 = \{w \in \Sigma^* \mid w = u.v,\ u \in \Sigma^+,\ v \in \Sigma^*,\ n_0(u) > n_1(u) + 2$ or $n_1(u) > n_0(u) + 2\}$, where $n_i(u)$ denotes the count of i's in u, for $i \in \{0,1\}$. For example, $0010221020012 \in L$ but $012012012 \notin L$.
   3. $L_3 = \{0^n \mid n > 0, n^3 + n^2 + n + 1 = 0 \mod 3\}$

Solution:

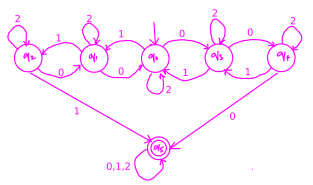
Possible DFAs are given below:

1.



q_1, q_2, q_3 remember that we've not seen any decreasing sequence of letters so far, and the last letter seen was 0, 1, 2, respectively. All of these strings are accepted. q_4 remembers that we have seen a decreasing sequence, and hence anything we see subsequently cannot make us accept the string. So q_4 is a sink or trap state – once you reach there, you can't escape it.

2.

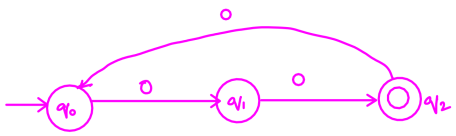


For a prefix u of w seen so far, we need to remember if $n_0(u) - n_1(u)$ is $0, 1, -1, 2, -2$ or greater than 2 or less than -2. Once the prefix u satisfies $n_0(u) - n_1(u) > 2$ or $n_0(u) - n_1(u) < -2$, we can immediately accept the word, regardless of what subsequent letters are seen. States q_0, q_1, q_2, q_3, q_4 remember if $n_0(u) - n_1(u) = 0, -1, -2, 1, 2$ respectively, where u is the (prefix of the) input word seen so far. State q_5 simply remembers whether $|n_0(u) - n_1(u)| > 2$.

3. Notice that $n^3 + n^2 + n + 1 = 0 \mod 3$ is equivalent to $(n^2+1) \cdot (n+1) = 0 \mod 3$. Therefore, $((n^2+1) \mod 3) \cdot ((n+1) \mod 3)$ has to be equal to 0.

   Given that $n \mod 3 \in \{0,1,2\}$, it follows that we require $n^2 = 2 \mod 3$ or $n = 2 \mod 3$. However, for no n is $n^2 = 2 \mod 3$. Why so? Because this would require $((n$

$\bmod 3) \times (n \mod 3)) = 2 \mod 3$. Since $n \mod 3 \in \{0,1,2\}$, $(n \mod 3) \times (n \mod 3)$ can only be in $\{0,1\} \mod 3$. Therefore, we must have $n = 2 \mod 3$. The DFA is now easily obtained as follows:



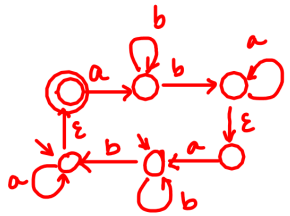
2. Let $\Sigma = \{a, b\}$. Construct NFAs, possibly with ε-transitions for each of the following languages.

   1. $L_4 = \{w \in \Sigma^* \mid n_{ab}(w) \text{ is even}\}$, where $n_{ab}(w)$ denotes the count of times ab appears in w as consecutive letters.
   2. $L_5 = \{w \in \Sigma^* \mid w$ contains the "pattern" abba (as consecutive letters) followed by the "pattern" baba, possibly in an overlapping manner$\}$. For example, $ababbabaabb, bababbabb \notin L$ but $abababbaba, ababbaabbaba \in L$.

   Now try constructing a DFA that recognizes L_5 using the subset construction and ε-edge removal on the NFA constructed above. Do you see an exponential blow-up in the count of states as you do this conversion?

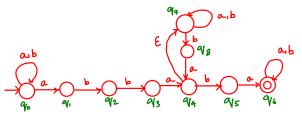
Solution: Possible NFAs are given below. These are not the only solutions. In fact, the first subquestion has a fairly simple DFA that can be directly constructed as well.

1.



The solution given here illustrates the convenience provided by NFAs with ε-transitions. We have two copies of an NFA that reads a sequence of as and bs with a leading a, a single ab change and possibly a trailing sequence of 0 or more as. These two NFAs can be seen in the top row (read left to right) and in the bottom row (read right to left). We simply connect them using ε-transitions, so that we read words with an even count of ab changes. The acceptance state of the overall NFA is therefore the starting state of the top copy of the NFA. The start states of the overall NFA are chosen so that we can accept a leading sequence of 0 or more as, or even such a sequence of bs followed by as (i.e. strings that don't contain a single ab change).

2.



The NFA is quite self-explanatory. This clearly shows the convenience of NFAs with ε-transitions. You can capture the intent of the problem so clearly that a look at the NFA tells you what it's designed to accept. This is not (so easily) the case if you determinize this NFA.

The NFA to DFA construction is left to you as a standard exercise.

3. **Take-away question: Propositional formulas and NFAs**

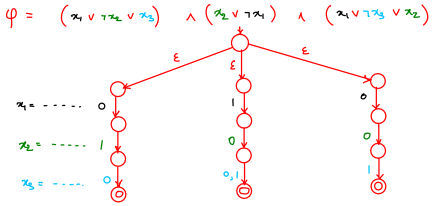
   For a CNF formula φ with m variables and c clauses, show that you can construct in polynomial time an NFA with $O(cm)$ states that accepts all **falsifying or non-satisfying assignments**, represented as boolean strings of length m. You can assume that the formula φ is over variables $x_1 x_2 \ldots x_m$, and you can assume that the NFA is fed as input the word $v_1 \cdot v_2 \cdot \cdots \cdot v_m$, where $v_i \in \{0,1\}$ and v_i is interpreted as the value of propositional variables x_i, for all $i \in \{1, \ldots m\}$.

   Can you construct an NFA in time polynomial in c and m that accepts all and only **satisfying assignments** of the CNF formula φ?

Solution:

1. On input ϕ, construct an NFA that non-deterministically picks one of the c clauses (via ε-transitions), reads the input of length m, and accepts if it does not satisfy the clause. For each clause, we need exactly m states, so the NFA has $O(cm)$ states. It is clear that we can construct the NFA in polynomial time, hence it is of polynomial size.

   Now consider any falsifying assignment of φ. At least one clause must be falsified by this assignment. Hence all literals of this clause must evaluate to 0. The NFA can non-deterministically choose this clause and accept the assignment string. Conversely, if the NFA accepts an assignment string, then there is a clause of φ that is falsified by the assignment. So, the assignment is a falsifying assignment of φ. Hence, the NFA accepts all and only the nonsatisfying assignments of φ. For example: let $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_1) \wedge (x_1 \vee \neg x_3 \vee x_2)$, the corresponding NFA is as shown below.



2. Unless we have a polynomial time algorithm for checking satisfiability of CNF formulas (aka **P vs NP** problem), such a construction is not possible. Indeed, if we could do this, then there would be a way to check in time polynomial in c and m whether φ is (un)satisfiable. Why? Simply construct the NFA for φ and check in the graph corresponding to the NFA whether any final state can be reached from any initial state

by any path. If so, there is a satisfying assignment, else the formula is unsatisfiable. Note that searching for a path in the graph can be done using any graph search algorithm like BFS or DFS in time that is polynomial in the size of the NFA (viewed as a graph). If the NFA's size is polynomial in c and m, this search (BFS/DFS) will also take time polynomial in c and m.

CS 208 : Automata Theory and Logic Spring 2024

Homework 1 Solutions

Question 1 : Implication "Equations" 15 points

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

Let φ_1 and φ_2 be unknown propositional formulas over $x_1, x_2, \ldots x_n$. Consider the following implications labelled (1a), (1b) through (na), (nb). A solution to this set of simultaneous implications is a pair of specific propositional formulas (φ_1, φ_2) such that all implications are **valid**, i.e. evaluate to true for all assignments of variables.

$$\begin{array}{ll|ll} (1a) & (x_1 \wedge \varphi_1) \rightarrow \varphi_2 & (1b) & (x_1 \wedge \varphi_2) \rightarrow \varphi_1 \\ (2a) & (x_2 \wedge \varphi_1) \rightarrow \varphi_2 & (2b) & (x_2 \wedge \varphi_2) \rightarrow \varphi_1 \\ & \vdots & & \vdots \\ (na) & (x_n \wedge \varphi_1) \rightarrow \varphi_2 & (nb) & (x_n \wedge \varphi_2) \rightarrow \varphi_1 \end{array}$$

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

1. *[5 marks]* Suppose we are told that φ_1 is $\bot$. How many semantically distinct formulas φ_2 exist such that implications (1a) through (nb) are valid?
2. *[2 marks]* Answer the above question, assuming now that φ_1 is $\top$.
3. *[5 marks]* If we do not assume anything about φ_1, how many semantically distinct pairs of formulas (φ_1, φ_2) exist such that all the implications are valid?
4. *[3 marks]* Does there exist a formula φ_1 such that there is exactly one formula φ_2 (modulo semantic equivalence) that can be paired with it to make (φ_1, φ_2) a solution of the above implications?

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

1. All implications in the left column reduce to $\bot \rightarrow \varphi_2$. Every such implication is trivially valid (since $\neg\bot \vee \varphi_2$ is true for all φ_2 for all assignments of variables). All implications in the right column reduce to $(x_i \wedge \varphi_2) \rightarrow \bot$. This implication is valid iff $(x_i \wedge \varphi_2)$ evaluates to false for all assignments of the variables. Using the partitioning of variable assignments mentioned above, φ_2 must evaluate to false whenever any of the variables $x_1, \ldots x_n$ evaluates to true. Hence, the semantics of φ_2 is determined for all assignments other than $x_1 = \cdots x_n = 0$. Since φ_2 can have two different truth values (0 or 1) for this assignment, only 2 distinct formulas φ_2 are possible.

2. The right column of implications are trivially valid if $\phi_1 \leftrightarrow \top$ is valid. Using reasoning similar to that above for the left column, once again only 2 distinct formulas φ_2 are possible.

3. Note that $(x_i \wedge \phi_j) \rightarrow \phi_k$ is semantically equivalent to $x_i \rightarrow (\phi_j \rightarrow \phi_k)$. This can be easily checked through truth tables. Therefore, if implications (1a) and (1b) are both valid, then $x_1 \rightarrow (\varphi_1 \leftrightarrow \varphi_2)$ must also be valid. Similarly, for implications (2a) and

(2b), and so on until (na) and (nb). Thus, if all the implications are to be valid, then $\varphi_1 \leftrightarrow \varphi_2$ must evaluate to true for all assignments of variables where at least one of $x_1, \ldots x_n$ is true. In other words, for every φ_1, the semantics of φ_2 can potentially differ from that of φ_1 only for the assignment $x_1 = \cdots x_n = 0$. Therefore, there are only two possible φ_2 for every φ_1. Since the total number of semantically distinct formulas φ_1 on n variables is 2^{2^n} (why?), the total count of semantically distinct pairs (φ_1, φ_2) is $2 \times 2^{2^n} = 2^{2^n+1}$.

4. The above implications allow φ_2 to evaluate to any value in $\{0, 1\}$ when $x_1 = x_2 = \cdots x_n = 0$, regardless of what φ_1 is. Therefore, there are always at least two solutions to the given implications.

Question 2 : Balanced Parentheses 25 points

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

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

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

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

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

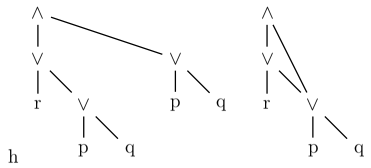


Figure 1: A parse tree and corresponding DAG

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

(a) *[10 marks]* Encode the problem of checking balanced parentheses in a binary string of length n as a propositional logic formula whose DAG size is at most $\mathcal{O}(n^3)$. You must use only n propositional variables corresponding to the n bits in the input string, and the only connectives allowed in your formula are two-input $\vee, \wedge$ and $\neg$. Express the DAG size of your formula as a function of n using big-$\mathcal{O}$ notation, with a clear justification of how you obtained the size expression.

(b) *[5 marks]* Prove that your formula is unsatisfiable for all odd values of n.

(c) *[10 (+ bonus 5) marks]* Suppose your formula is represented as a DAG, as discussed above. Given an input string of 0s and 1s, there is a simple way to evaluate the semantics of the formula. Specifically, we evaluate DAG nodes bottom-up, starting from the leaves (these represent propositional variables whose values are given by the input string) and moving upwards until we reach the root. The value of the root gives the semantics of the formula. Normally, a DAG node (labelled $\wedge, \vee$ or $\neg$) can be evaluated only after all its children have been evaluated. However, if a $\vee$ (resp. $\wedge$) labelled node has a child that has evaluated to 1 (resp. 0), then the node itself can be evaluated to 1 (resp. 0) without evaluating its other child. This can allow us to find the value of the root without evaluating all nodes in the DAG.

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

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

Solution: Let $T_{i,j}$ represent the subformula, which evaluates to $\top$ if the number of open brackets exceeds the number of closed brackets by j in the first i characters of the string. Our formula for balanced parenthesis becomes $T_{n,0}$.
To make this formula well-defined, we must define the complete set of $T_{i,j}$ for all values of i, j. For all values of $i > 0, j < 0$, we have $T_{i,j} = \bot$ and for all values of $j \neq 0$, we have $T_{0,j}$ as $\bot$ and $T_{0,0} = \top$ For all other values of i, j, we have $T_{i,j} = (x_i \wedge T_{i-1,j+1}) \vee (\neg x_i \wedge T_{i-1,j-1})$. This makes the formula well-defined.
Lemma (well-defined): The complete set of propositional variables in the above formula is contained in $\{x_i\}$. We prove this by induction. (Base Cases are $T_{0,j}$ and induction over i)
Lemma (Size): The number of distinct subformulae is in $\mathcal{O}(n^3)$. We prove this by explicitly upper bounding the number of distinct subformulae.
Lemma (Correctness): If $T_{n,0}$ evaluates to $\top$ if and only if the string is balanced. We prove this using induction over even values of n.
(b) In the same way, for part b, we do induction over odd values of n and show that the formula is unsat.
(c) Claim: This formula can be evaluated in $O(n)$ time.
Evaluation Algorithm: We evaluate x_n and, based on whether it is $\top$ or $\bot$, evaluate the appropriate branch in the formula.

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

CS 208 : Automata Theory and Logic Spring 2024

Homework 2 Solutions

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

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

Let us call two sets of formulae Σ_1 and Σ_2 *equisemantic* if for every formula φ, we have $\Sigma_1 \models \varphi$ if and only if $\Sigma_2 \models \varphi$. Furthermore, let us call a non-empty set of formulae Σ *irredundant* if no formula σ in Σ is semantically entailed by $\Sigma \setminus \{\sigma\}$.

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

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

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

(a) *[5 points]* Show that a non-empty satisfiable set Γ with $|\Gamma| \geq 2$ is irredundant if and only if $(\Gamma \setminus \{\gamma\}) \cup \{\neg\gamma\}$ is satisfiable for every $\gamma \in \Gamma$.

(b) *[5 points]* Use the above result to show that Σ'' is irredundant and equisemantic to Σ.

Solution:

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

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

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

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

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

   (b) The fact that Σ'' is equisemantic to Σ is easy to see. We will first show that an assignment α satisfies all formulas in Σ iff it satisfies all formulas in Σ". Clearly, if each of the formulas in Σ evaluates to true for α, then both sides of each implication in Σ' evaluate to true for α. Hence, each of the formulas in Σ' is also true for the same assignment. Conversely, if each of the formulas in Σ' is true for assignment α, then σ_1 is true, and since $(\sigma_1) \to \sigma_2$ is true, this implies σ_2 is true. Using the same reasoning, it follows by induction that σ_n is true for all $n \geq 1$. Thus, each formula in Σ is true for the assignment α.

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

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

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

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

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

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

Example:

Let $\Sigma = \{p_0 \to p_1, p_1 \to p_2, p_2 \to p_3, \ldots\}$.

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

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

Lemma:

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

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

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

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

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

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

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

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

Theorem:

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

Solution:

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

Lemma:

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

Proof:

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

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

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

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

(c) This follows directly from the lemma that was proven earlier. If a PL-definable language does not contain an infinite bitstring x, then there exists a finite prefix y of x such that for every infinite bitstring w, yw is not in the language. Since there are uncountably many infinite bitstrings w, there are uncountably many infinite bitstrings not in the language

(d) Firstly, it is easy to see that every PL-definable is also EPL-definable: PL is a special case of EPL where no dummy variables are used.

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

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

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

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

Some Notation:

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

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

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

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

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