2019-2020 COMP21111 Revision Notes
Author: weilue luo
Contact: weilue222@gmail.com
*This note is not completely exhaustive and contains personal understanding.

Glossary

Entails
: A entails B if A's models is subset of B's models.

Equivalent
: A and B are ***equivalent*** if they have the same models.

Rewrite Rule
: Equivalences to simplify formulas.

Monotonicity
: $a_x \leq a_y$ implies $f(a_x) \leq f(a_y)$.

Position & Subformula
: Empty position is ε
: Position π in a formula A, subformula at a position, denoted $A|_\pi$.
: Variable is considered a subformula.

Polarity
: $pol(A, \epsilon) = 1$.
: polarity *invert* whenever $\neg$ is encountered.
: polarity is `0` whenever $\leftrightarrow$ is encountered.

Clauses
: A disjunction of literals.

Clausal Form
: A set of clauses which satisfiable iff the formula is so.

K-SAT
: The problem of checking satisfiability for sets of clauses containing k variables

Crossover Point
: The value of r which probability of unsatisfiability crosses 0.5.

ε-window
: The interval values of r where probability is between ϵ and $1 - \epsilon$.

Witness of Instance
: Any data *D* such that given *D* one can check if a decision problem has a `yes` answer.

Short Notes

- A formula with n propositional variables has 2^n interpretations.
- Propositional ***satisfiability*** is `NP-complete`.
- **SAT** and **3-SAT** are `NP-complete`.
- **2-SAT** is decidable in `linear` time.
- Propositional ***validity*** is `coNP-complete`.
- ***Equivalent*** checking can be *reduce* to unsatisfiability checking.
- If all occurrences of an atom `p` is *positive* then for the purpose of checking satisfiability we can replace `p` by `true`, (`false` respectively).
- Ratio of clauses per variable is $r = m/n$ and $\pi(r, n)$ is probability of unsatisfiability which is a *monotonic* function.

- For **3-SAT**, as we increase the number of variables, the transition become *steeper* but the *crossover points* do not vary much $(\textasciitilde 4.25)$. It is hard to determine satisfiability around *crossover point* , but if it is far from *crossover point* then the problem is relatively easy to solve.
- Every propositional formula can be transformed into an equivalent ***if-else-then*** normal form.
- In ***QBF*** formula, only *free* variable matters.
- A *closed* formula is a ***QBF*** formula with no *free* variable.
- For a ***closed QBF*** formula F, result of $I \models F$, F satisfiability, F validity are the same.
- *Satisfiability/validity* of formulas with ***free*** variables can be reduced to checking ***truth/falsity*** of ***closed*** formulas.
- For every **transition system** $\mathbb{S}$ and state s there exists a unique **computation tree** for $\mathbb{S}$ starting at s, up to the order of children.

Lemmas

Monotonicity Replacement
: Polarity act as sign here, it will invert the interpretation if it is negative
: If $pol(A, \pi) = 1$, then $I(B) \leq I(B')$ implies $I(A[B]\pi) \leq I(A[B']\pi)$
: If $pol(A, \pi) = -1$, then $I(B') \leq I(B)$ implies $I(A[B]\pi) \leq I(A[B']\pi)$

Satisfiability Checking

We often want to check if there exists a solution for a particular problem. These problems can often express as propositional formula. Thus looking for a solution becomes checking satisfiability of the corresponding propositional formula.

Truth Table

Easiest way but grows exponentially w.r.t size of variables, which is unacceptable, problem encoded in boolean propositional formula is often huge.

Compact Truth Table (lec 2)

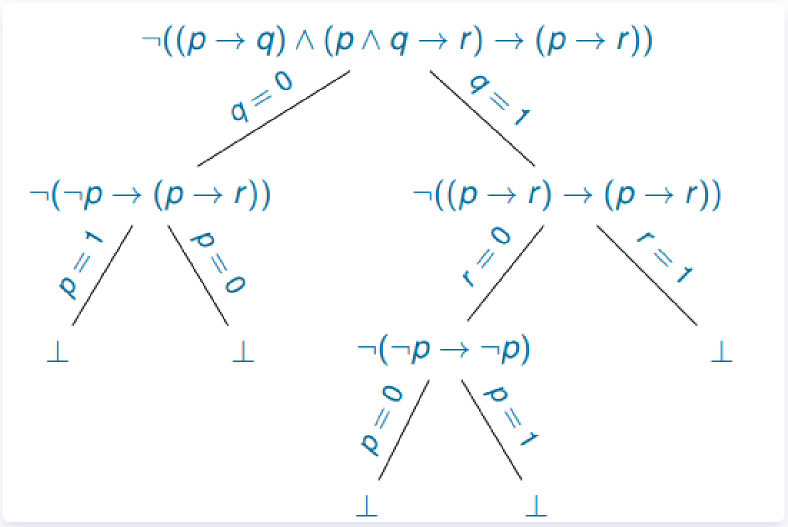
Evaluate formula based on assigning values to only subset of variables. A way to reduce size of truth table, but not enough.

subformlas	interpretations
...	...
variables	assignments

Splitting algorithm (lec 2)

Simplify the formula, return accordingly if it is `tautology` or `contradiction`.
Assign true/false to one formula's atom, repeat from step 1, if either return `satisfiable` then return `satisfiable` else return `unsatisfiable`.

```
procedure split(F)
parameter function select
input formula F
output satisfiable or unsatisfiable

begin
    F = simplify(F)
    if F is True: return satisifiable
    if F is False: return unsatisifiable
    p, b = select(F)
    if b is True:
        if split(F^True_p) is satisfiable:
            return satisfiable
        else:
            return split(F^False_p)
    else:
        if split(F^False_p) is satisfiable:
            return satisfiable
        else:
            return split(F^True_p)
end
```



$\neg((p \to q) \land (p \land q \to r) \to (p \to r))$
$q = 0$
$q = 1$
$\neg(\neg p \to (p \to r))$
$\neg((p \to r) \to (p \to r))$
$p = 1$
$p = 0$
$r = 0$
$r = 1$
$\perp$
$\perp$
$\neg(\neg p \to \neg p)$
$\perp$
$p = 0$
$p = 1$
$\perp$
$\perp$

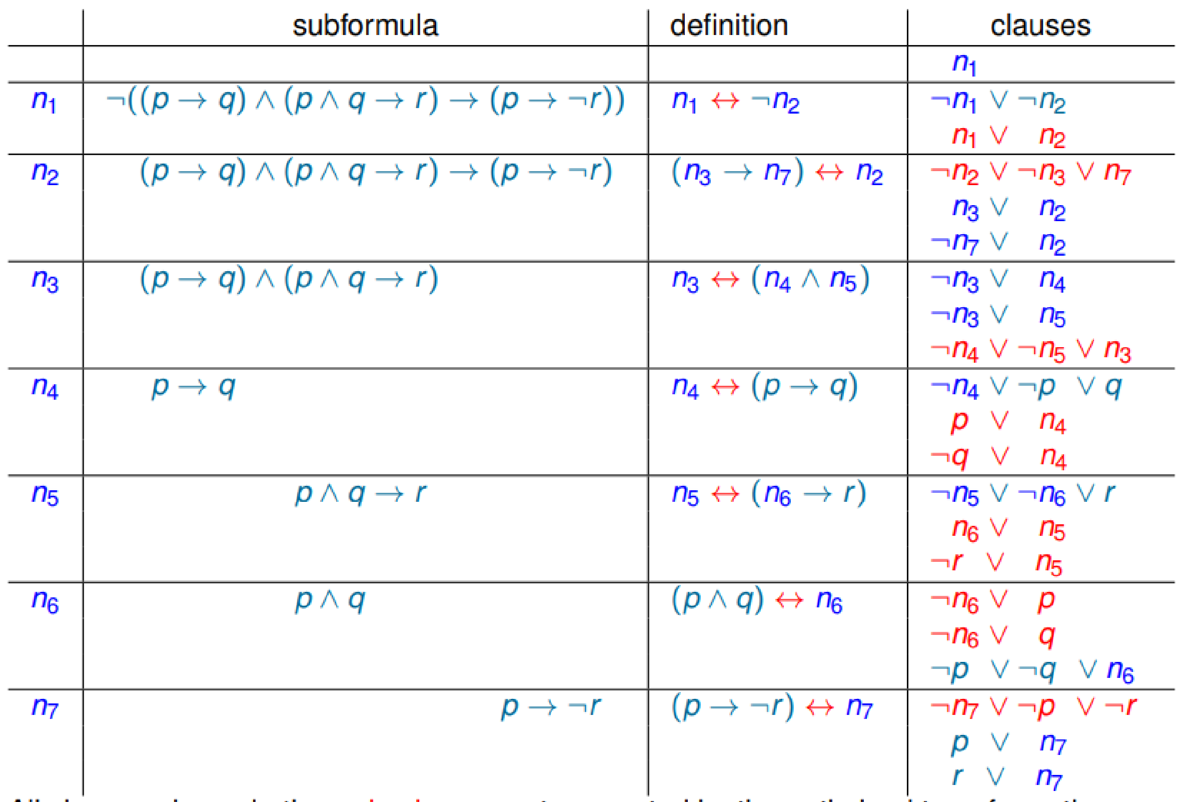
CNF (lec 3)

Transform satisfiability problem of formula to set of clauses. However transformation can grow exponentially. (variables connected by ↔).

[Optimised] Clausal Form (lec 3)

Introduce names for every **non-literal** sub formula. This ensure at most linear increase in size. Furthermore, if we apply two lemmas we can reduce the number of generated clauses.

Monotonic Replacement Lemma
: Let A[B]π where pol(A, π) is positive [/negative]. Then A[B]π is satisfiable if and only if A[n] ∧ (n → B)[/B → n] is satisfiable, (where n is a new variable that does not occur A[B]π).



	subformula	definition	clauses
			n_1
n_1	$\neg((p \to q) \wedge (p \wedge q \to r) \to (p \to \neg r))$	$n_1 \leftrightarrow \neg n_2$	$\neg n_1 \vee \neg n_2$ $n_1 \vee n_2$
n_2	$(p \to q) \wedge (p \wedge q \to r) \to (p \to \neg r)$	$(n_3 \to n_7) \leftrightarrow n_2$	$\neg n_2 \vee \neg n_3 \vee n_7$ $n_3 \vee n_2$ $\neg n_7 \vee n_2$
n_3	$(p \to q) \wedge (p \wedge q \to r)$	$n_3 \leftrightarrow (n_4 \wedge n_5)$	$\neg n_3 \vee n_4$ $\neg n_3 \vee n_5$ $\neg n_4 \vee \neg n_5 \vee n_3$
n_4	$p \to q$	$n_4 \leftrightarrow (p \to q)$	$\neg n_4 \vee \neg p \vee q$ $p \vee n_4$ $\neg q \vee n_4$
n_5	$p \wedge q \to r$	$n_5 \leftrightarrow (n_6 \to r)$	$\neg n_5 \vee \neg n_6 \vee r$ $n_6 \vee n_5$ $\neg r \vee n_5$
n_6	$p \wedge q$	$(p \wedge q) \leftrightarrow n_6$	$\neg n_6 \vee p$ $\neg n_6 \vee q$ $\neg p \vee \neg q \vee n_6$
n_7	$p \to \neg r$	$(p \to \neg r) \leftrightarrow n_7$	$\neg n_7 \vee \neg p \vee \neg r$ $p \vee n_7$ $r \vee n_7$

All clauses shown in the red color are not generated by the optimised transformation.

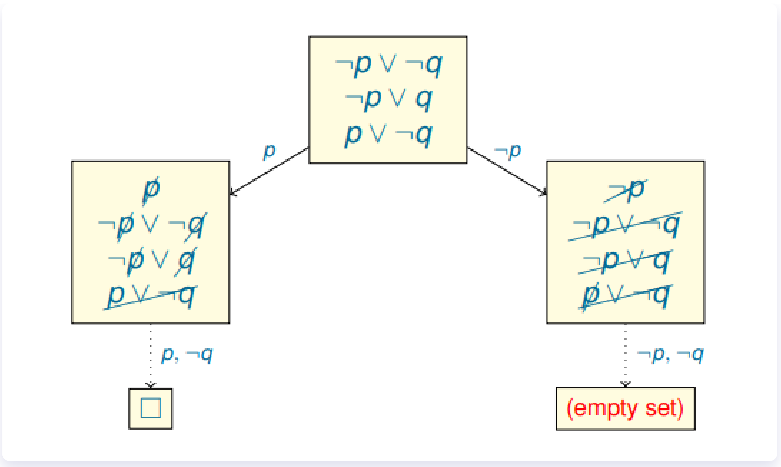
Unit Propagation (lec 4)

We can further reduce the size of set of clauses if there exists **unit clause** in the set as the corresponding atom must be true in order for the set of clauses to be true. This can result in **more unit clauses** thus lead to **more propagations.

DPLL (lec 4)

We can combine unit propagation and splitting to form a more efficient version of splitting algorithm called DPLL.

```
procedure DPLL
input set of clauses S
output satisfiable or unsatisfiable
parameter function select

begin
    S = unit_propogation(S)
    if S is empty: return satisfiable
    if S contains empty clause: return unsatisfiable
    L = select(S)
    return DPLL({S U L}) is satisfiable ? satisfiable : DPLL({S U ¬L})
end
```



Note
: Use $\rightarrow$ denote `select` and $\dashrightarrow$ denote `unit propogation`.

Optimizations

1. **Pure Literal**
   We can remove clauses containing **pure literal**.
   This may result in **more pure literals** and result in **removing more clauses**.
2. **Tautology**
   We can remove tautology clauses.

Horn Clauses (lec 4)

If we have **a set of horn clauses**. By apply **unit propagation**:

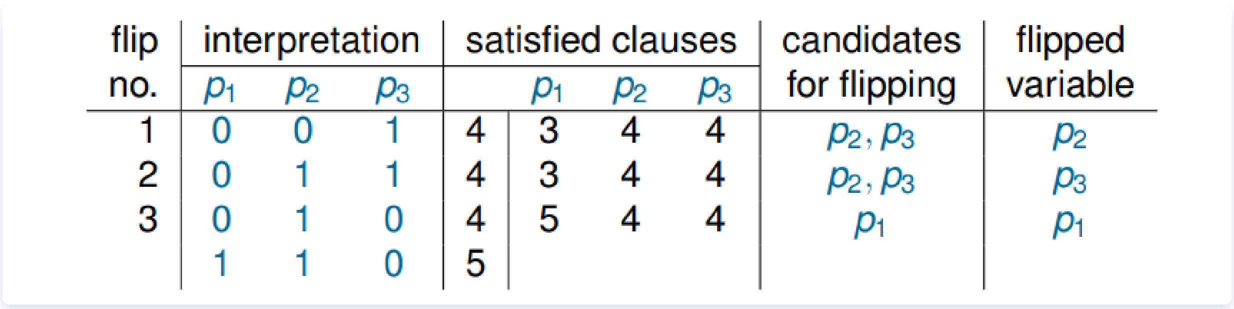
- Result contains **empty clause** then it is **unsatisfiable**.
- Else it is **satisfiable**.
  Because every clause has **more than two literals**. That means **at least one negative literal in every clause**. Thus we can **set all other literals** to **false** to obtain a true interpretation and declare the set of horn clauses to be **satisfiable**.

> Does the opposite version of horn clauses work as well?

GSAT (lec 5)

Sometimes for large problem we might interested in checking satisfiability within a short amount of time, thus we introduce a local search satisfiability algorithm which cannot establish unsatisfiability.
The idea is to starts with some random interpretations, flip variable optimally and try to find a satisfiability.

```
procedure GSAT
input set of clauses S
output interpretation I satisfy S or I don't know
parameter MAX_TRIES, MAX_FLIPS

begin
    for MAX_TRIES times:
        I = random interpretation
        if I satisfy S: return I
        for MAX_FLIPS times:
            p = best flip candidate (satisfy most clauses if flipped p)
            I = I.flip(p)
            if I satisfy S: return I
    return I don't know
end
```

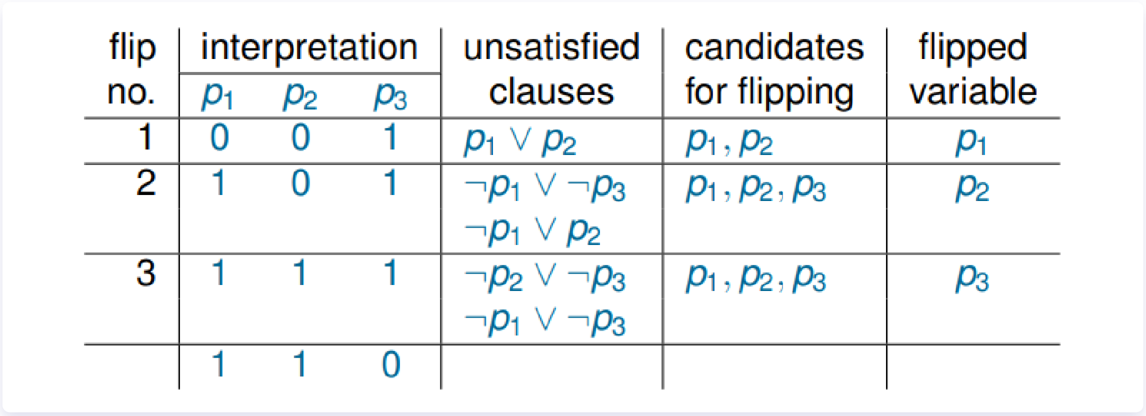


flip no.	interpretation p_1	p_2	p_3	satisfied clauses	p_1	p_2	p_3	candidates for flipping	flipped variable
1	0	0	1	4	3	4	4	p_2, p_3	p_2
2	0	1	1	4	3	4	4	p_2, p_3	p_3
3	0	1	0	4	5	4	4	p_1	p_1
	1	1	0	5					

Walk SAT (WSAT) (lec 5)

Although GSAT enable us to find a solution in a large problem quickly but we may stuck in a *plateau* point during max_flips loop where further flips does not change the number of satisfying clauses. Thus an alternative arise. The idea of this approach is to randomly flip a variable in the false clause instead.

```
procedure Walk SAT
input set of clauses S
output an I satisfy S or I don't know
parameters MAX_TRIES, MAXFLIPS

begin
    for MAX_TRIES times:
        I = random interpretation
        if I satisfy S: return I
        for MAX_FLIPS times:
            p = random variable from random false clause
            I = I.flip(p)
            if I satisfy S: return I
    return I don't know
end
```



flip no.	interpretation p_1	p_2	p_3	unsatisfied clauses	candidates for flipping	flipped variable
1	0	0	1	$p_1 \lor p_2$	p_1, p_2	p_1
2	1	0	1	$\neg p_1 \lor \neg p_3$ $\neg p_1 \lor p_2$	p_1, p_2, p_3	p_2
3	1	1	1	$\neg p_2 \lor \neg p_3$ $\neg p_1 \lor \neg p_3$	p_1, p_2, p_3	p_3
	1	1	0			

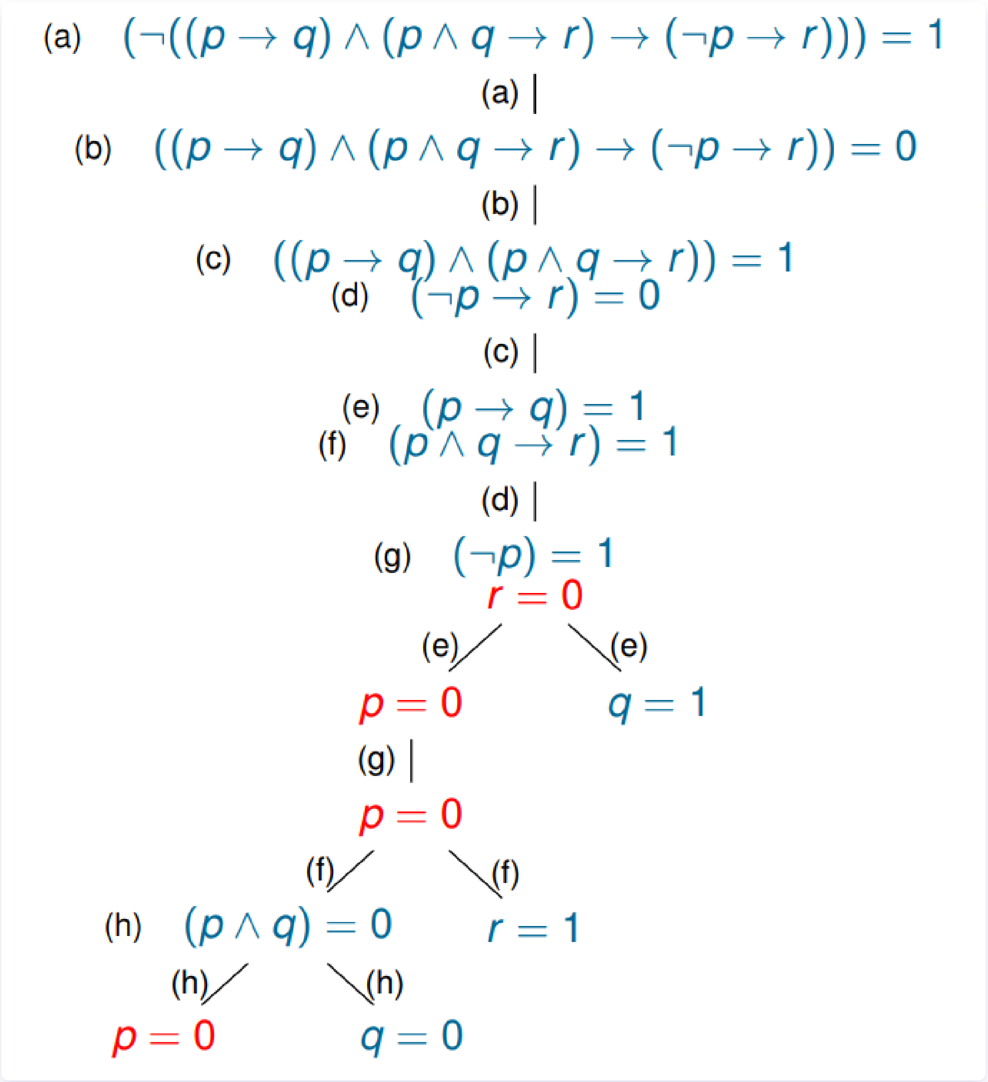
GSAT with random walks (lec 5)

We can also combine the two approaches above with an extra probability parameter to decide which approach to use during the loop.

```
procedure GSAT with random walk
input set of clauses S
output I which satisfy S or I don't know
parameters MAX_TRIES, MAX_FLIPS, probability 0 ≤ k ≤ 1

begin
    for MAX_TRIES times:
        I = random interpretation
        if I satisfy S: return I
        for MAX_FLIPS times:
            with k probability do:
                p = best flip candidate
            else:
                p = random variable from random false clause
            I = I.flip(p)
            if I satisfy S: return I
    return I don't know
end
```

Tableau (lec 5)

Satisfiability can be represent using sign formula = 1. We can build a tableaux for signed formula and check for satisfiability by looking for open branch.



(a) $(\neg((p \to q) \land (p \land q \to r) \to (\neg p \to r))) = 1$
(a)
(b) $((p \to q) \land (p \land q \to r) \to (\neg p \to r)) = 0$
(b)
(c) $((p \to q) \land (p \land q \to r)) = 1$
(d) $(\neg p \to r) = 0$
(c)
(e) $(p \to q) = 1$
(f) $(p \land q \to r) = 1$
(d)
(g) $(\neg p) = 1$
$r = 0$
(e)
(e)
$p = 0$
$q = 1$
(g)
$p = 0$
(f)
(f)
(h) $(p \land q) = 0$
$r = 1$
(h)
(h)
$p = 0$
$q = 0$

Assume a formula denote as F. We can check for *validity* if there is a *close tableau* for $F = 0$ (i.e. $F = 0$ is not possible). Similarly we can check equivalence for any two formulas A and B if there is a *close tableau* for $A \leftrightarrow B = 0$ (i.e. $A \leftrightarrow B$ is always 1) . A *fully expanded tableau* for $F = 1$ give all models of F.

Summary

General Formula

: **Splitting**

    Recursively try true and false for every variables.

Semantic Tableau

    Apply logical reduction on signed formula.

Set of Clauses

: **DPLL**

    Recursively apply unit propagation on true/false version of every variables

: **Randomized**

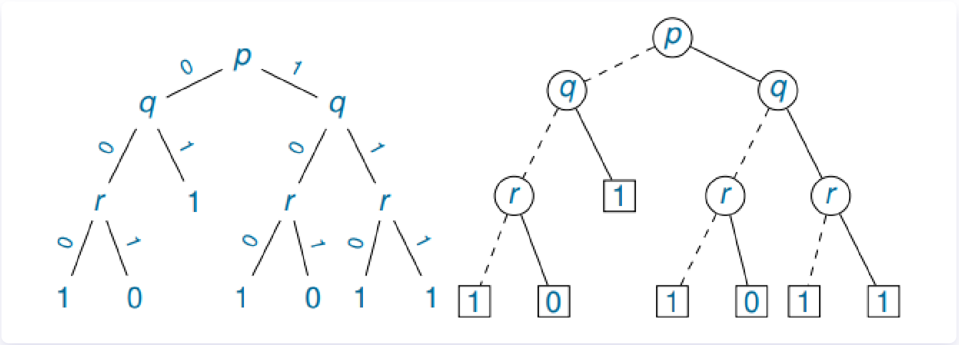
Try random interpretation with max_tries, max_flips and probability k.
Update with flip variable with best flip possible (may stuck at `plateau` point)
and/or random variable from random false clause.

Data Structures

We may want to reuse large propositional formula over and over again, check for properties, negation and equivalence. We need to parse the formula into some kind of compact data structure such that these operations can be easily carry out.

Binary Decision Tree (lec 6)

By removing redundant information in the *splitting tree* one can obtain the *binary decision tree* .



```
procedure binary decision tree
input formula F
output the binary decision tree of F
parameter select

begin
    F = simplify(F)
    if F is true: return true
    if F is false: return false
    p = select(F)
    return tree(BDT(F_p^false), p, BDT(F_p^true))
end
```

The original formula is *forgotten* --- syntax is lost but semantic is preserved. Two formulas with the same binary decision tree is guaranteed to be equivalent.

Properties

- Worst case exponential size.
- Interpretation checking can be done in *linear* in number of variables.
- Satisfiability/Validity checking can be done in *linear* in tree size.
- Equivalent/Some operations(e.g. conjunction) are hard to check/implement.

If-Else-Then Normal Form (lec 6)

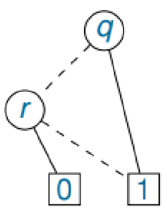
We can easily convert *binary decision tree* to *if-else-then* normal form.

$$\begin{array}{llllllllllll}
\textit{if} & p & \textit{then} & \textit{if} & q & \textit{then} & \textit{if} & r & \textit{then} & \top \\
 & & & & & & & & \textit{else} & \top \\
 & & & & & \textit{else} & \textit{if} & r & \textit{then} & \bot \\
 & & & & & & & & \textit{else} & \top \\
 & & \textit{else} & \textit{if} & q & \textit{then} & \top & & & \\
 & & & & & \textit{else} & \textit{if} & r & \textit{then} & \bot \\
 & & & & & & & & \textit{else} & \top
\end{array}$$

$$\textit{if } A \textit{ then } B \textit{ else } C \equiv (A \to B) \wedge (\neg A \to C).$$

Binary Decision Diagram (lec 6)

Binary decision tree are not good enough, many operations are hard to implement and it is not compact.
It contains redundant tests and isomorphic subtree. Here we first make it compact by transforming BDT to Binary Decision Diagram (BDD).



Now the representation is compact, **satisfiability** and **validity** checking can be done in **constant** time.
Note: when transforming, use a *bottom up* approach.

Order Binary Decision Diagram (lec 6)

Although we successfully solved the compact problem, but **equivalent** checking and some boolean operations such as **conjunction** are still hard to implement. We solve these two problems by converting BDD to Order Binary Decision Diagram (OBDD)
The main problem of BDD is that different branch can do test on different variable. Thus we introduce an *order* into the diagram (e.g. $P > Q > R$).

Now the diagram has more properties:
: *Equivalence* / *satisfiability* / *validity* can be check in **constant** time.
: Many boolean operations are *easy to implement* .

Now we have an OBDD which has nice properties, but often we have many formulas and we would like to combine them and check its properties, to do that we need to define a few more operations.

Integrating node into a dag

```
procedure integrate
input node n1, variable p, node n2
output a node in modified D representing if p then n1 else n2
parameter global dag D

begin
    if n1 == n2: return n1  // value of p does not matter
    node = tree(n1, p, n2)
    D.add_if_absent(node)
    return node
end
```

Integrating formula into a dag

We make node representing the formula and use `integerate()` function above

```
procedure OBDD
input formula F
output a node in modified D representing F
parameter global dag D

begin
    // base case: tautology or contradiction
    F = simplify(F)
    if F is true: return 1
    if F is false: return 0
    // recursive case:
    // build a tree representing formula F
    // then integrate into global dag D bottom up
    p = max_variable(F)
    left = OBDD(F_p^false)
    right = OBDD(F_p^true)
    return integrate(left, p, right, D)
end
```

Integrate nodes as disjunction/conjunction into a dag

Sometimes we want to integrate formulas as disjunction/conjunction into a global dag, we can make nodes representing these formula then integrate them as follows.

```
procedure disjunction_conjunction
input k nodes representing k formulas
output a node in modified D representing disjunction_conjunction of input nodes
parameter global dag D

begin
    // a tautology in disjunction make disjunction tautology
    // a contradiction in conjunction make conjunction contradiction
    if nodes.contains(1_0): return 1_0
    // skip contradiction/tautology, irrevelent
    if nodes.contains(0_1): return disjunction_conjunction(nodes.remove_all(0_1))
    p = nodes.max_variable()
    lefts, rights = [], []
    // split node containing current order variable
    for node in nodes:
        if node.val is p:
            lefts.add(node.left)
            rights.add(node.right)
        else:
            lefts.add(node)
            rights.add(node)
    // lefts representing obdd's path when p (max var) is false
    // rights now representing obdd's path when p (max var) is true
    left = disjunction_conjunction(lefts)
    right = disjunction_conjunction(rights)
    return integrate(left, p, right, D)
end
```

Quantified Boolean Formula (lec 7 & 8)

Sometimes we would like some of the variable having some kind of properties, sometimes we do not care what value is eventually assigned to the formula, as long as we have an assignment that satisfy the formula. Thus we add some quantification to our variable in the formula, here we include two operators: $\forall$ and $\exists$ (e.g. $p \to \forall q \exists p \, (q \leftrightarrow p) \vee r$).

Rectified Formula

Sometimes we would like to substitute parts of the formula to check for properties without changing other parts of the formula. But we cannot directly substitute variables because we have given some special properties to part of them (*bounded*). So we need to give different names to distinguish *bound* and *free* variables, so that when we substitute we only make changes to variables with same properties. Formulas which we can freely substitute formulas are called **rectified**.

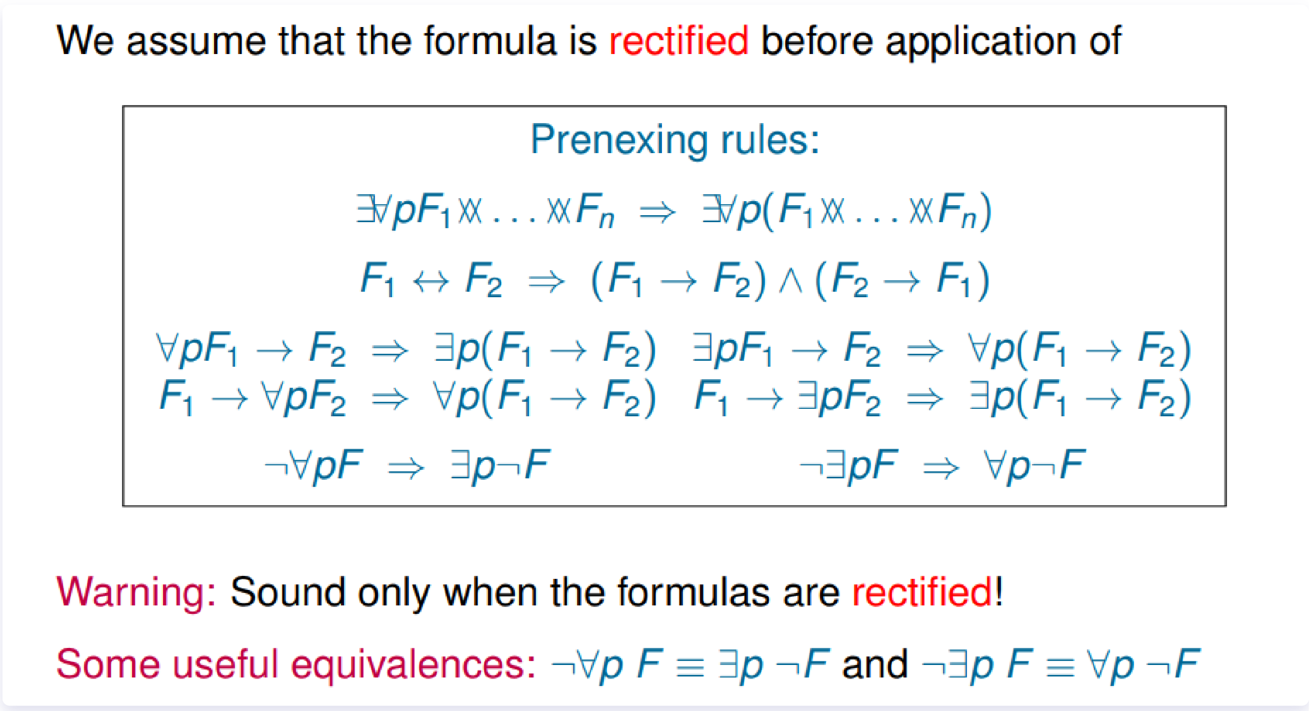
Jointly Rectified Formulas

A set of formulas S is *jointly rectified* if

- A variable is *free* then so it is in all formulas.
- There is only one occurrence of any quantifiers in formulas.

Prenex Form

A formula is in *prenex form* if all *quantifiers* are in front of the formula.
We can convert any *QBF* to its *prenex* form.



We assume that the formula is rectified before application of

Prenexing rules:

$$\exists\!\!\!\forall p F_1 \wedge\!\!\!\!\vee \ldots \wedge\!\!\!\!\vee F_n \;\Rightarrow\; \exists\!\!\!\forall p(F_1 \wedge\!\!\!\!\vee \ldots \wedge\!\!\!\!\vee F_n)$$

$$F_1 \leftrightarrow F_2 \;\Rightarrow\; (F_1 \to F_2) \wedge (F_2 \to F_1)$$

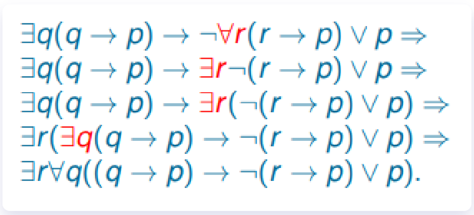
$$\forall p F_1 \to F_2 \;\Rightarrow\; \exists p(F_1 \to F_2) \qquad \exists p F_1 \to F_2 \;\Rightarrow\; \forall p(F_1 \to F_2)$$

$$F_1 \to \forall p F_2 \;\Rightarrow\; \forall p(F_1 \to F_2) \qquad F_1 \to \exists p F_2 \;\Rightarrow\; \exists p(F_1 \to F_2)$$

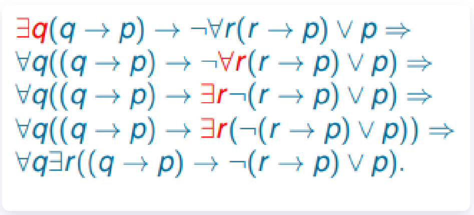
$$\neg\forall p F \;\Rightarrow\; \exists p \neg F \qquad \neg\exists p F \;\Rightarrow\; \forall p \neg F$$

Warning: Sound only when the formulas are rectified!

Some useful equivalences: $\neg\forall p\, F \equiv \exists p\, \neg F$ and $\neg\exists p\, F \equiv \forall p\, \neg F$



$$\begin{aligned}
&\exists q(q \to p) \to \neg\forall r(r \to p) \vee p \Rightarrow\\
&\exists q(q \to p) \to \exists r\neg(r \to p) \vee p \Rightarrow\\
&\exists q(q \to p) \to \exists r(\neg(r \to p) \vee p) \Rightarrow\\
&\exists r(\exists q(q \to p) \to \neg(r \to p) \vee p) \Rightarrow\\
&\exists r\forall q((q \to p) \to \neg(r \to p) \vee p).
\end{aligned}$$



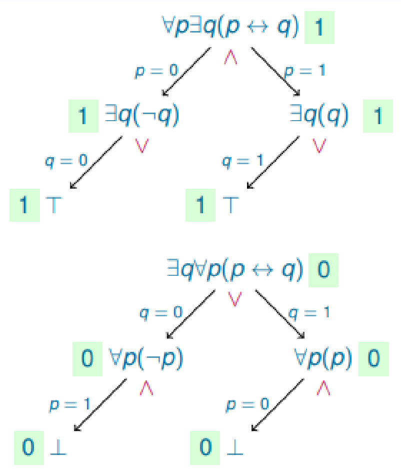
$$\begin{aligned}
&\exists q(q \to p) \to \neg\forall r(r \to p) \vee p \Rightarrow\\
&\forall q((q \to p) \to \neg\forall r(r \to p) \vee p) \Rightarrow\\
&\forall q((q \to p) \to \exists r\neg(r \to p) \vee p) \Rightarrow\\
&\forall q((q \to p) \to \exists r(\neg(r \to p) \vee p)) \Rightarrow\\
&\forall q\exists r((q \to p) \to \neg(r \to p) \vee p).
\end{aligned}$$

Apply Algorithms on QBF

- **Pure Atom Elimination**
  If an atom p is pure then,
  - There exists an assignment for p so that p will be true so if it is existentially quantified then all clauses containing p can be removed.
  - There exists an assignment for p so that p will be false so if it is universally quantified then p can be removed from all clauses.
- **Universal Literal Deletion**
  For any non-tautology clause. If there exists a universal quantified literal l such that all existential literals is quantified before l, then l can be removed without changing the result. This is because
  - If any *existential literal* is ***true*** then the clause is ***true*** regardless of l.
  - If all *existential literals* is ***false***, then the leftover l will be false as it is universally quantified.
- **Unit Propagation**
  Given a variable, if it is ***universally quantified***, then the set of clauses is ***false***, else just carry out normal unit propagation.
- **Implementations**

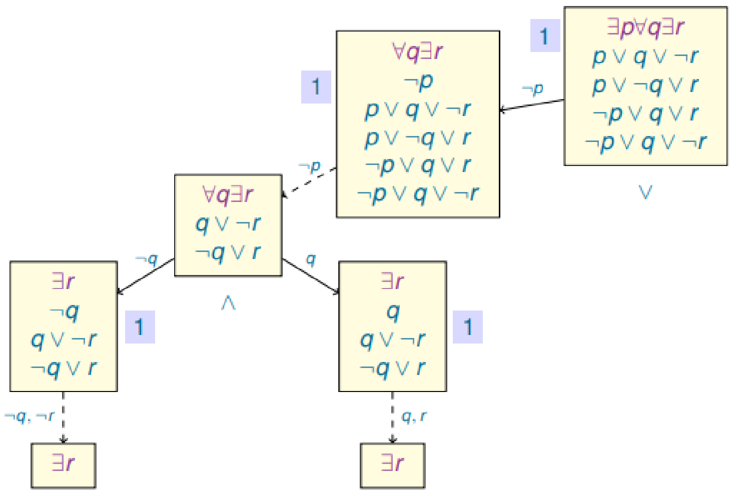
```
// for splitting and DPLL
// after select
switch(alg(F_b^false), q):
    case (true, E):
        return true
    case (false, E):
        return alg(F_b^true)
    case (true, A):
        return alg(F_b^true)
    case (false, A):
        return false

// for OBDD
// after select and split nodes into left (l) and right (r)
if EA_quants.contains(p):
    // existential quantified means either negative or positive will work,
    // universal quantified means both negative and positive must work,
    // so we add both side either case
    return EA_quant(E_quants.remove(p), set(l + r))
// normal variable, so carry out usual task
left = EA_quant(E_quant, l)
right = EA_quant(E_quant, r)
return integrate(left, p, right, D)
```

- **Splitting**



Note: the order of variables is important!
Two-player game: by selecting a value for $\exists q$ one is trying to make the formula true, by selecting a value for $\forall p$ one is trying to make it false.

- **DPLL**



Note
$\forall$-quantification output node representing conjunctions
$\exists$-quantification output node representing disjunctions
The only thing we need to change is ***base case*** and function ***name*** .

States Changing System

Finite Domains (lec 9)

> Propositional Logic for Finite Domain (PLFD)

Often in real world, ***true*** and ***false*** are often not enough to representing states and we may want to assign different range of values to different variables. However, we can always transform logic with more states into boolean propositional logic.
Here we introduce some notations. The set of values we can assign to a variable is called ***finite domain*** , it must be ***non-empty*** and ***finite*** .
If we have a variable ***x*** then $dom(x)$ represents the ***finite domain*** of ***x*** , this implies $x \in dom(x)$.

PLFD $\rightarrow$ Propositional Logic

Let say we have a PLFD formula F.

1. Replace old variables, let $x = v$ be x_v, denote replaced formula as F_{prop}.
2. Limit variables to their domains, for example, limiting variable x to its domain $\{v_1 \dots v_n\}$, denote as D_x.
$$\underbrace{\overbrace{(x_{v_1} \lor \cdots \lor x_{v_n})}^{at\ least\ one\ within\ domain\ values} \quad \land \quad \overbrace{\bigwedge_{i<j} (\neg x_{v_i} \lor \neg x_{v_j})}^{can't\ have\ two\ at\ the\ same\ time}}_{exactly\ one\ from\ the\ domain\ for\ variable\ x}$$
3. We need to do this for ***every*** variable, denote as D_{all}.
$D_{all} = D_{x_1} \land \cdots \land D_{x_n}$
4. The original PLFD formula F is *satisfiable* ***if and only if*** the propositional formula $D_{all} \land F_{prop}$ is *satisfiable* .

For example, let x has domain range $\{a, b, c\}$ and we interested in checking satisfiability of
$\neg(x = b \lor x = c)$
Then this is ***equivalent*** to check satisfiability of following boolean propositional formula
$(x_a \lor x_b \lor x_c) \land (\neg x_a \lor \neg x_b) \land (\neg x_a \lor \neg x_c) \land (\neg x_c \lor \neg x_b) \land$
$\neg(x_b \lor x_c)$

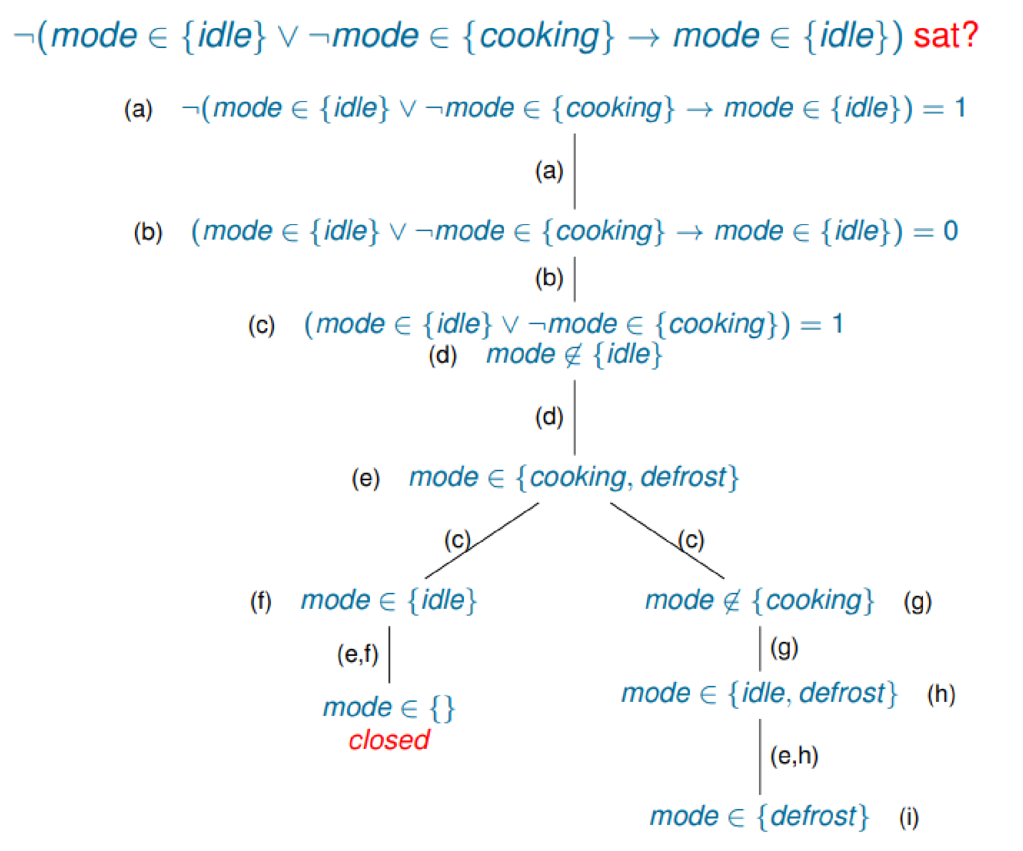
Examples

$$
\begin{aligned}
\mathit{Recharge} &\overset{\text{def}}{=} \text{customer} = \mathit{none} \wedge \\
&\quad \text{st\_coffee}' \wedge \text{st\_beer}' \wedge \\
&\quad \mathit{only}(\text{st\_coffee}, \text{st\_beer}). \\
\mathit{Customer\_arrives} &\overset{\text{def}}{=} \text{customer} = \mathit{none} \wedge \text{customer}' \neq \mathit{none} \wedge \\
&\quad \mathit{only}(\text{customer}) \\
\mathit{Customer\_leaves} &\overset{\text{def}}{=} \text{customer} \neq \mathit{none} \wedge \text{customer}' = \mathit{none} \wedge \\
&\quad \mathit{only}(\text{customer}). \\
\mathit{Coin\_insert} &\overset{\text{def}}{=} \text{customer} \neq \mathit{none} \wedge \text{coins} \neq \mathit{3} \wedge \\
&\quad (\text{coins} = \mathit{0} \rightarrow \text{coins}' = \mathit{1}) \wedge \\
&\quad (\text{coins} = \mathit{1} \rightarrow \text{coins}' = \mathit{2}) \wedge \\
&\quad (\text{coins} = \mathit{2} \rightarrow \text{coins}' = \mathit{3}) \wedge \\
&\quad \mathit{only}(\text{coins}).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Dispense\_beer} &\overset{\text{def}}{=} \text{customer} = \mathit{student} \wedge \text{st\_beer} \wedge \\
&\quad \text{disp} = \mathit{none} \wedge (\text{coins} = \mathit{2} \vee \text{coins} = \mathit{3}) \wedge \\
&\quad \text{disp}' = \mathit{beer} \wedge \\
&\quad (\text{coins} = \mathit{2} \rightarrow \text{coins}' = \mathit{0}) \wedge \\
&\quad (\text{coins} = \mathit{3} \rightarrow \text{coins}' = \mathit{1}) \wedge \\
&\quad \mathit{only}(\text{st\_beer}, \text{disp}, \text{coins}). \\
\mathit{Dispense\_coffee} &\overset{\text{def}}{=} \text{customer} = \mathit{prof} \wedge \text{st\_coffee} \wedge \\
&\quad \text{disp} = \mathit{none} \wedge \text{coins} \neq \mathit{0} \wedge \\
&\quad \text{disp}' = \mathit{coffee} \wedge \\
&\quad (\text{coins} = \mathit{1} \rightarrow \text{coins}' = \mathit{0}) \wedge \\
&\quad (\text{coins} = \mathit{2} \rightarrow \text{coins}' = \mathit{1}) \wedge \\
&\quad (\text{coins} = \mathit{3} \rightarrow \text{coins}' = \mathit{2}) \wedge \\
&\quad \mathit{only}(\text{st\_coffee}, \text{disp}, \text{coins}). \\
\mathit{Take\_drink} &\overset{\text{def}}{=} \text{customer} \neq \mathit{none} \wedge \text{disp} \neq \mathit{none} \wedge \\
&\quad \text{disp}' = \mathit{none} \wedge \\
&\quad \mathit{only}(\text{disp}).
\end{aligned}
$$

Algorithm for PLFD

Tableau


$\neg(mode \in \{idle\} \vee \neg mode \in \{cooking\} \rightarrow mode \in \{idle\})$ sat?
(a) $\neg(mode \in \{idle\} \vee \neg mode \in \{cooking\} \rightarrow mode \in \{idle\}) = 1$
(a)
(b) $(mode \in \{idle\} \vee \neg mode \in \{cooking\} \rightarrow mode \in \{idle\}) = 0$
(b)
(c) $(mode \in \{idle\} \vee \neg mode \in \{cooking\}) = 1$
(d) $mode \notin \{idle\}$
(d)
(e) $mode \in \{cooking, defrost\}$
(c)
(c)
(f) $mode \in \{idle\}$
$mode \notin \{cooking\}$ (g)
(e,f)
$mode \in \{\}$
closed
(g)
$mode \in \{idle, defrost\}$ (h)
(e,h)
$mode \in \{defrost\}$ (i)

The formula is satisfiable and the model is $I = \{mode \mapsto defrost\}$

Finite State Transition System

We want a way to model system which change state from one and another so that we can check properties of the system. Thus we build a **transition system**. We define transition system as follows.

$\mathbb{S} = \{S, ln, T, X, dom\}$

X and dom define a **PLFD** where X is a set of variables and dom is the domain of the variables. Let's denote all possible interpretations of X as $\mathbb{I}$.

$S \subseteq \mathbb{I}$ is a *non-empty* set of ***states*** .

$ln \subseteq S$ is a *non-empty* set of ***initial states*** .

$T \subseteq S \times S$ is a set of pairs of ***transition relations*** .

We call this transition system finite because for every variable $x \in X$, $dom(x)$ is finite.

Computation Tree and Paths

Now we have a transition system, we may want to investigate properties such as ensure that there is no path from a starting state s such that it leads to some unwanted states s'. Thus we introduce **computation tree** to find all possible paths of our system.

Let $\mathbb{S}$ be a finite state transition system, we introduce ***nodes*** as the states in S and a **computation tree** starting at a state s called ***root*** . Then we expand the root by stating for every node n in the tree, its children are nodes n'' from $(n', n'') \in T$.

A ***computation path*** is any branch of the computation tree.

> If a transition system have multiple initial states does it has more than one root?

Linear Temporal Logic

We have all possible paths of a **transition system**, we can use ***linear temporal logic*** to investigate some properties of a given **computation path**.

- $\square$: always in the future.

- $\Diamond$: eventually in the future.
- $\bigcirc$: next.
- $F\ \mathbf{U}\ G$: F has to hold before G is ***true***, which must happen at current or future.
- $F\ \mathbf{R}\ G$: G has to hold before & includes F is ***true***. If F never hold then G is always ***true***.

Equivalences

- $\neg \bigcirc F \equiv \bigcirc \neg F$
- $\neg \Diamond F \equiv \Box \neg F$
- $\neg \Box F \equiv \Diamond \neg F$
- $\neg (A\ \mathbf{U}\ B) \equiv \neg A\ \mathbf{R}\ \neg B$
  - RHS $\equiv \left.\begin{matrix} A & A & \dots & A & \neg A \\ \neg B & \neg B & \dots & \neg B & \neg B \end{matrix}\right\}$ ensured B is false when first $\neg A$ appeared, this break $\mathbf{U}$'s until property.
  - If $\neg A$ never appear then does not satisfy $\mathbf{U}$'s must happen property either.

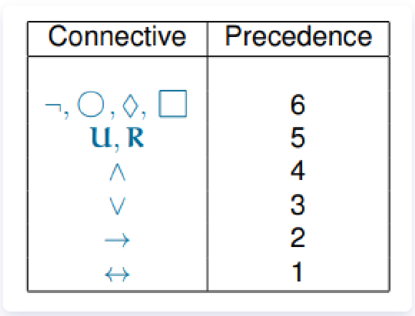
LTL with $\mathbf{U}$ and $\bigcirc$ has the same expressive power as LTL

- $\Diamond F \equiv \top\ \mathbf{U}\ F$
- $\Box A \equiv \neg(\top\ \mathbf{U}\ \neg A)$
- $A\ \mathbf{R}\ B \equiv \neg(\neg A\ \mathbf{U}\ \neg B)$

Unwinding Properties

- $\Diamond F \equiv F \lor \bigcirc \Diamond F$
- $\Box F \equiv F \land \bigcirc \Box F$
- $A\ \mathbf{U}\ B \equiv B \lor (A \land \bigcirc (A\ \mathbf{U}\ B))$
- $A\ \mathbf{R}\ B \equiv B \land (A \lor \bigcirc (A\ \mathbf{R}\ B))$

Precedence



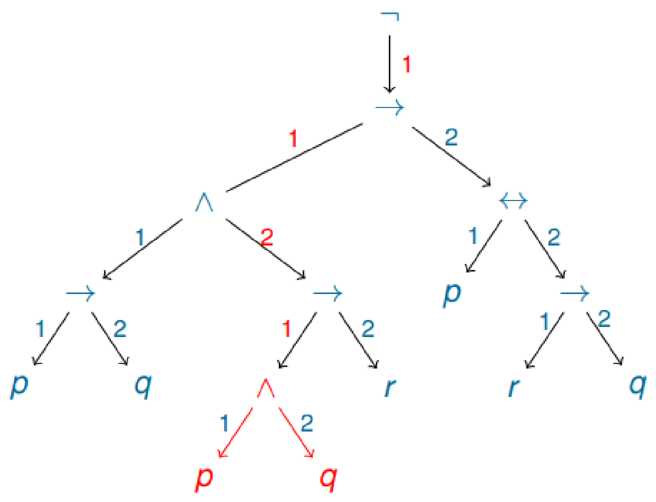
Connective	Precedence
$\neg, \bigcirc, \Diamond, \Box$	6
$\mathbf{U}, \mathbf{R}$	5
$\land$	4
$\lor$	3
$\to$	2
$\leftrightarrow$	1

Note
: If we want to show two **LTL** formulas are not *equivalent* then we present a *path* which satisfy one but not the other.
: We may also want to consider the problem: does a **LTL** formula holds for all paths or only some of it?

Miscellaneous

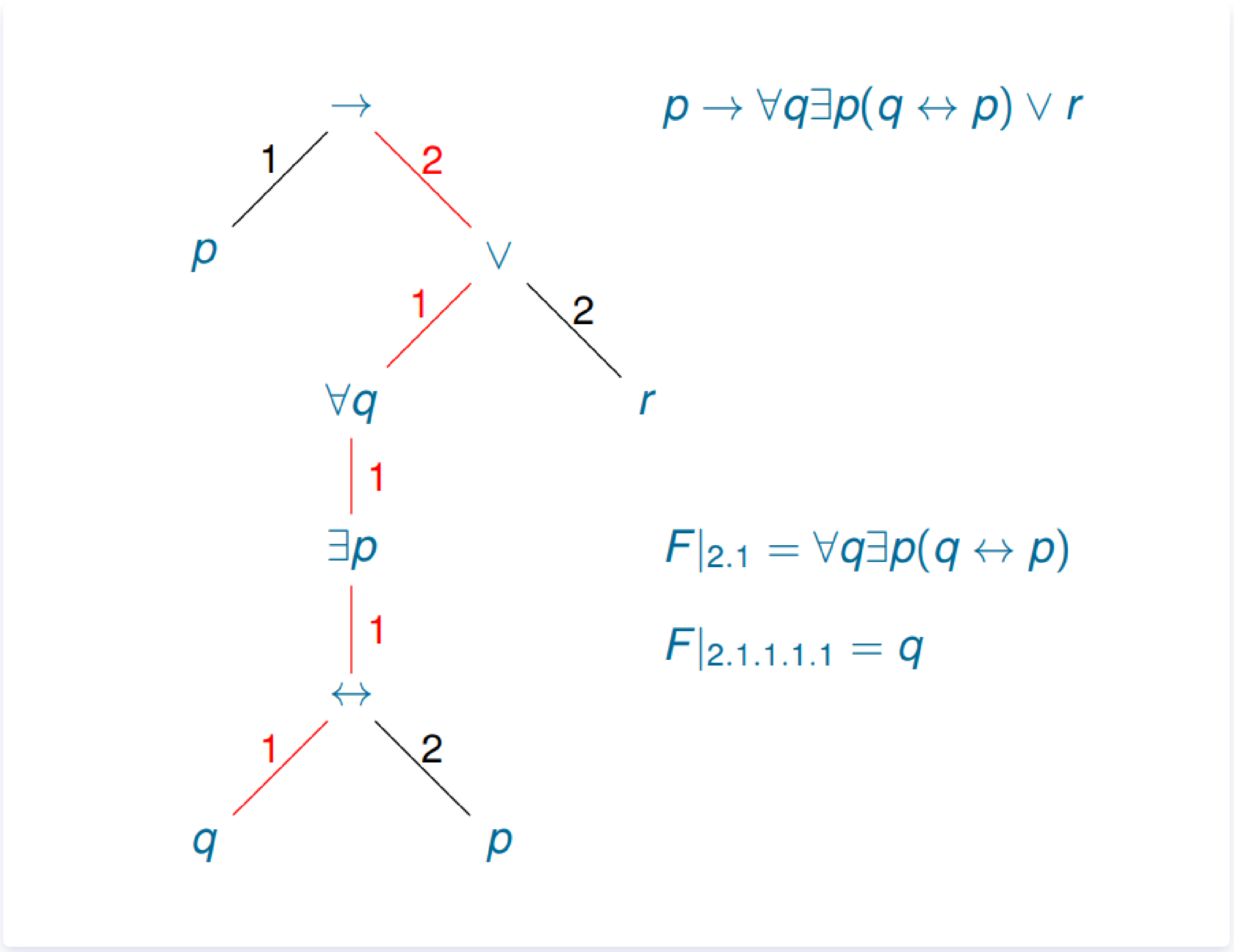
Parse Tree (lec 2)

$A \overset{\text{def}}{=} \neg((p \to q) \land (p \land q \to r) \to (p \leftrightarrow (r \to q)))$.



- Position in the formula: 1.1.2.1;
- Subformula at this position: $p \land q$; denoted $A|_{1.1.2.1} = p \land q$.
- Position of A is ϵ.

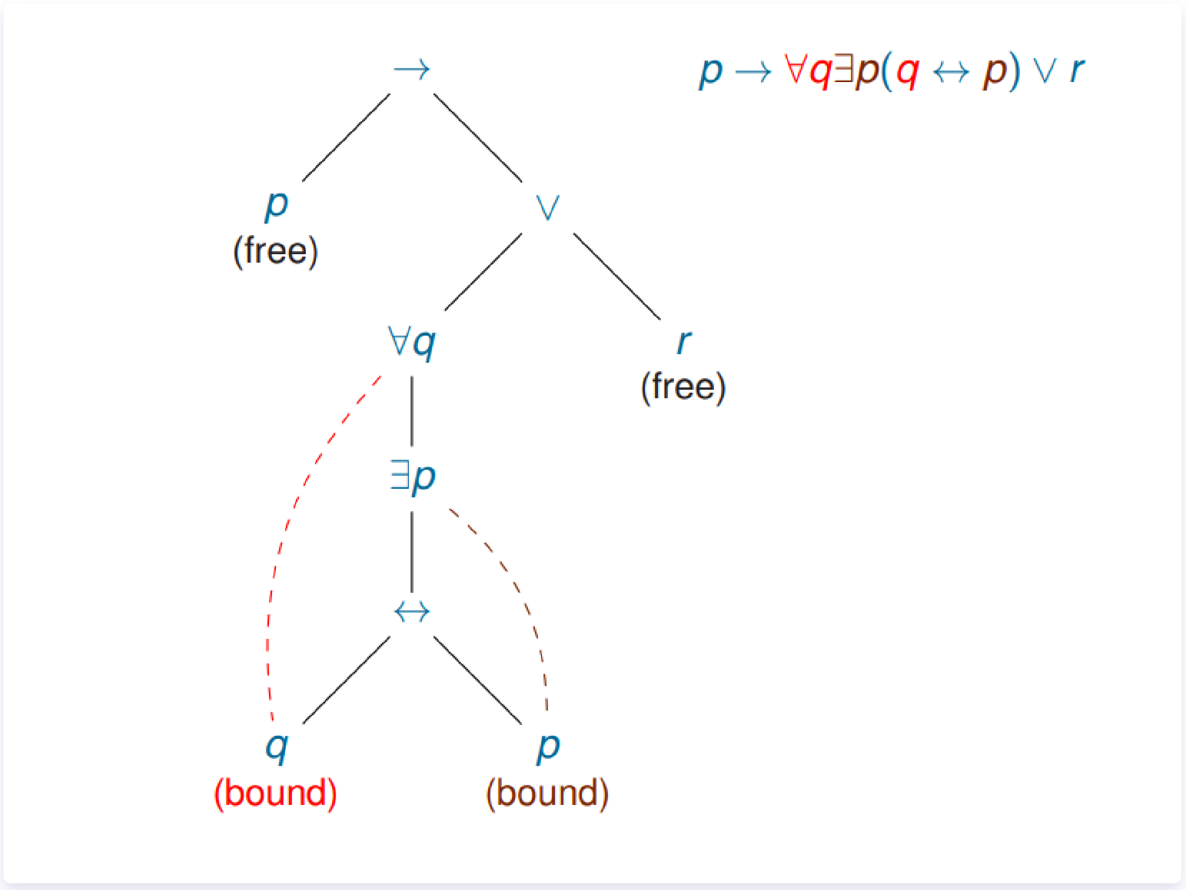
QBF Parse Tree



$$F|_{2.1} = \forall q \exists p (q \leftrightarrow p)$$

$$F|_{2.1.1.1.1} = q$$

Demonstrate free/bound variable using parse tree.



Pure Atom (lec 2)

Replace atom with consistent polarity with corresponding tautology or contradiction.

$$\begin{array}{cc}
\neg((p \to q) \wedge (p \wedge q \to r) \to (\neg p \to r)) & \Rightarrow \\
\neg((\perp \to q) \wedge (\perp \wedge q \to r) \to (\neg\perp \to r)) & \Rightarrow \\
\neg(\top \wedge (\perp \wedge q \to r) \to (\neg\perp \to r)) & \Rightarrow \\
\neg((\perp \wedge q \to r) \to (\neg\perp \to r)) & \Rightarrow \\
\neg((\perp \to r) \to (\neg\perp \to r)) & \Rightarrow \\
\neg(\top \to (\neg\perp \to r)) & \Rightarrow \\
\neg(\neg\perp \to r) & \Rightarrow \\
\neg(\top \to r) & \Rightarrow \\
\neg r & \Rightarrow \\
\neg\perp & \Rightarrow \\
\top &
\end{array}$$

Clause (lec 2)

- Empty clause: num_var = 0, it is horn and both positve and negative polarity.
- Unit clause: num_var = 1.
- Horn clause: at most 1 positive literal.

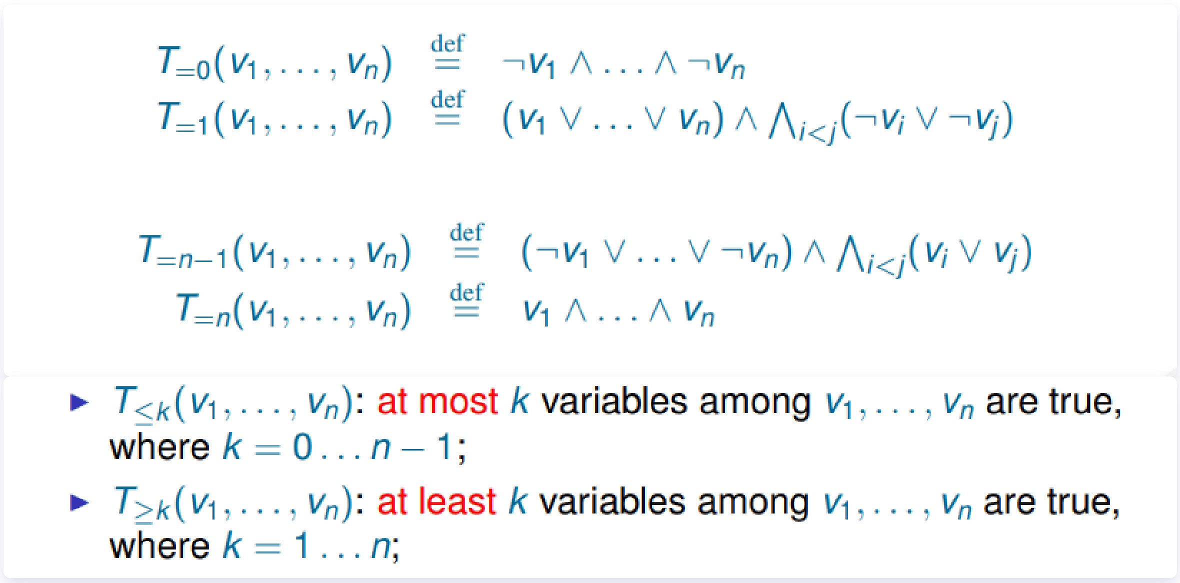
clause	unit	Horn	positive	negative
$\square$		✓	✓	✓
p	✓	✓	✓	
$\neg p$	✓	✓		✓
$p \vee q$			✓	
$\neg p \vee q$		✓		
$\neg p \vee \neg q$		✓		✓

CNF Transformation (lec 3)

$$\begin{array}{l}
\neg((p \to q) \wedge (p \vee q \to r \vee s) \to (p \to r)) \Rightarrow \\
\neg(\neg((p \to q) \wedge (p \vee q \to r \vee s)) \vee (p \to r)) \Rightarrow \\
\neg\neg((p \to q) \wedge (p \vee q \to r \vee s)) \wedge \neg(p \to r) \Rightarrow \\
(p \to q) \wedge (p \vee q \to r \vee s) \wedge \neg(p \to r) \Rightarrow \\
(p \to q) \wedge (p \vee q \to r \vee s) \wedge \neg(\neg p \vee r) \Rightarrow \\
(p \to q) \wedge (p \vee q \to r \vee s) \wedge \neg\neg p \wedge r \Rightarrow \\
(p \to q) \wedge (p \vee q \to r \vee s) \wedge p \wedge \neg r \Rightarrow \\
(p \to q) \wedge (\neg(p \vee q) \vee r \vee s) \wedge p \wedge \neg r \Rightarrow \\
(p \to q) \wedge ((\neg p \wedge \neg q) \vee r \vee s) \wedge p \wedge \neg r \\
(\neg p \vee q) \wedge ((\neg p \wedge \neg q) \vee r \vee s) \wedge p \wedge \neg r \\
(\neg p \vee q) \wedge (\neg p \vee r \vee s) \wedge (\neg q \vee r \vee s) \wedge p \wedge \neg r
\end{array}$$

$$\begin{array}{rcll}
A \leftrightarrow B & \Rightarrow & (\neg A \vee B) \wedge (\neg B \vee A), & \\
A \to B & \Rightarrow & \neg A \vee B, & \\
\neg(A \wedge B) & \Rightarrow & \neg A \vee \neg B, & \\
\neg(A \vee B) & \Rightarrow & \neg A \wedge \neg B, & \\
\neg\neg A & \Rightarrow & A, & \\
(A_1 \wedge \ldots \wedge A_m) \vee B_1 \vee \ldots \vee B_n & \Rightarrow & (A_1 \vee B_1 \vee \ldots \vee B_n) & \wedge \\
& & \ldots & \wedge \\
& & (A_m \vee B_1 \vee \ldots \vee B_n). &
\end{array}$$

Representing k out of n variables are true (lec 3)

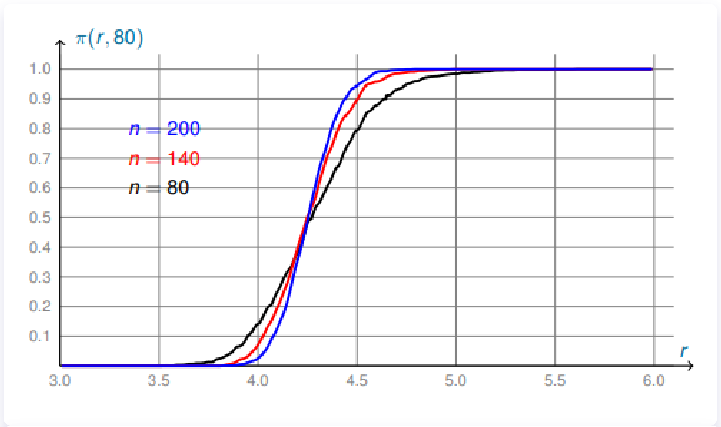


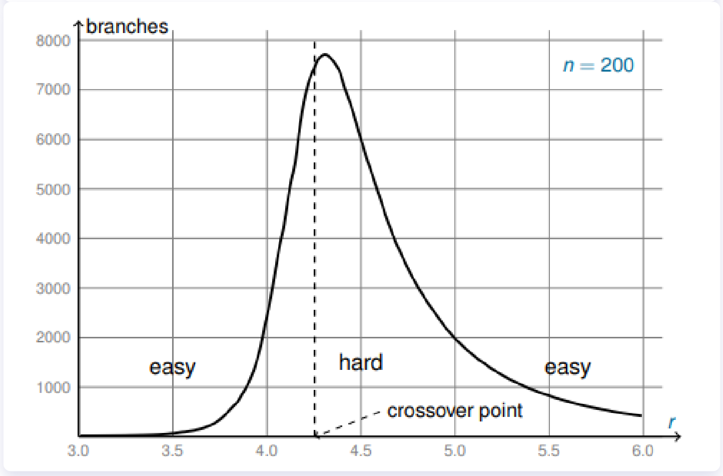
$$T_{=0}(v_1,\dots,v_n) \overset{\text{def}}{=} \neg v_1 \wedge \dots \wedge \neg v_n$$
$$T_{=1}(v_1,\dots,v_n) \overset{\text{def}}{=} (v_1 \vee \dots \vee v_n) \wedge \bigwedge_{i<j}(\neg v_i \vee \neg v_j)$$

$$T_{=n-1}(v_1,\dots,v_n) \overset{\text{def}}{=} (\neg v_1 \vee \dots \vee \neg v_n) \wedge \bigwedge_{i<j}(v_i \vee v_j)$$
$$T_{=n}(v_1,\dots,v_n) \overset{\text{def}}{=} v_1 \wedge \dots \wedge v_n$$

- $T_{\leq k}(v_1,\dots,v_n)$: at most k variables among $v_1,\dots,v_n$ are true, where $k = 0 \dots n-1$;
- $T_{\geq k}(v_1,\dots,v_n)$: at least k variables among $v_1,\dots,v_n$ are true, where $k = 1 \dots n$;

Transition & Easy-Hard-Easy Pattern for 3-SAT





We see that there is a sharp threshold transition of $\pi = m/n$ from sat to unsat with *crossover point* $\simeq 4.25$

Extras

Russian Spy

There are three people **A B C**, 2 Germans **G**, 1 Russian spy **R**:

$(RA \wedge GB \wedge GC) \vee (GA \wedge RB \wedge GC) \vee (GA \wedge GB \wedge RC)$

Russian is **Spy**:

$(RA \rightarrow SpyA) \wedge (RB \rightarrow SpyB) \wedge (RC \rightarrow SpyC)$

A is as German as **B** is Russian

$GA \leftrightarrow RB$

hidden: Russian is not German

$(RA \leftrightarrow \neg GA) \wedge (RB \leftrightarrow \neg GB) \wedge (RC \leftrightarrow \neg GC)$

we need to show **C** is not a Russian spy

$RC \wedge SpyC$

$\therefore$ If set of formulas is not satisfiable, then Person C cannot be a Russian Spy

Sudoku

Introduce $9 \times 9 \times 9 = 729$ variables , use V_{rcd} to represent board V at row r and column c contains digit d.

Each cell contains at least 1 digit

$(V_{rc1} \vee \cdots \vee V_{rc9})$

Each cell contains at most 1 digit

$(\neg V_{rc1} \vee \neg V_{rc2}) \wedge (\neg V_{rc1} \vee \neg V_{rc3}) \wedge \cdots \wedge (\neg V_{rc8} \vee \neg V_{rc9})$

Every row must contain different digits

$\{\neg V_{r',c,d} \vee \neg V_{r',c,d} \mid r, r', c, d \in \{1, \ldots, 9\}, c < c'\}$

Every column must contain different digits

$\{\neg V_{r,c,d} \vee \neg V_{r,c',d} \mid r, c, c', d \in \{1, \ldots, 9\}, c < c'\}$

Every 3x3 square must contain one of each digits

similar.

Finally we add initial configuration

e.g. V_{129}.

$\therefore$ If the set of formulas is satisfiable then the given sudoku with initial config has a solution.

Loop the Loop

We first number the rows and columns.

Let V_{ij} be there is a vertical line between $node_{ij}$ and $node_{ij+1}$

Let H_{ij} be there is a horizontal line between $node_{ij}$ and $node_{ij+1}$

Then we can encode exactly n lines around a number p like

$T_{=n}(v_{15}, v_{25}, h_{15}, h_{25})$

However, there is no simple way of expressing single loop in propositional logic