Propositional logic

Propositional formulas

- A proposition is a statement to which exactly one truth value (either true or false) can be assigned
- $\top$ (truth) and $\bot$ (falsehood) are the nullary connectives
- id and ¬ are the unary connectives
- For n ≥ 1, there are 2^{2^n} n-ary connectives corresponding to all possible valuations of the truth table with 2^n rows
    - The binary connectives XOR and NXOR are equivalent to not-iff and iff respectively and so are denoted $\not\equiv$ and ≡ in this module
- Parse tree of an atomic formula x is a single node x
- Parse tree for a formula ¬X is a ¬ node with child the parse tree for X
- Parse tree for a formula X○Y is a ○ node with left child the parse tree for X and right child the parse tree for Y
- **Degree of a formula = number of internal nodes in its parse tree**
    - deg(x) = 0; deg(¬X) = 1 + deg(X); deg(X○Y) = 1 + deg(X) + deg(Y)

Propositional formulas: Valuations

- A valuation is a mapping from propositional formulas to truth values that respects logical connectives or equivalently a mapping from variables to truth values
  - Let v be a valuation. Then, $v(\top) = \top$; $v(\bot) = \bot$; $v(\neg X) = \neg v(X)$; $v(X \circ Y) = v(X) \circ v(Y)$
- **A formula is a tautology iff it is $\top$ under all valuations**
- **A formula is a contradiction iff it is $\bot$ under all valuations**
- **A formula is satisfiable iff there exists a valuation under which it is $\top$** — tautology ⇒ satisfiable

Non-trivial laws of boolean algebra (Equational reasoning)

- De Morgan: $\neg(A \wedge B) = \neg A \vee \neg B$
- De Morgan: $\neg(A \vee B) = \neg A \wedge \neg B$
- Distributivity: $X \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$
- Distributivity: $X \vee (Y \wedge Z) = (X \vee Y) \wedge (X \vee Z)$
- Contrapositive: $X \rightarrow Y = \neg X \vee Y = \neg Y \rightarrow \neg X$
- Exportation: $(X \wedge Y) \rightarrow Z = X \rightarrow (Y \rightarrow Z)$

Normal forms

- A formula is in disjunctive normal form (DNF) iff it is a disjunction of conjunctive clauses (an $\vee$ of $\wedge$s of literals)
- A formula is in conjunctive normal form (CNF) iff it is a conjunction of disjunctive clauses (an $\wedge$ of $\vee$s of literals)
- We abbreviate $X_1 \wedge \ldots \wedge X_n$ to $\mathbf{<X_1, \ldots, X_n>}$ (<onjunction)
  - $<> = \top$
- We abbreviate $X_1 \vee \ldots \vee X_n$ to $\mathbf{[X_1, \ldots, X_n]}$ ([ɔisjunction)
  - $[] = \bot$

Normal form algorithms

- **Theorem: Any boolean function f can be written in DNF**
  Proof:

```
clauses := {⊥}
for v ∈ valuations(f)
    continue if v(f) is false
    clause := {⊤}
    for x in variables(f)
        if v(x) is true
            Literal := x
        else // v(x) is false
            literal := ¬x
        clause := clause ∪ {literal}
    clauses := clauses ∪ {∧ clause}
return ∨ clauses
```

- **Theorem: Any boolean function f can be written in CNF**
  Proof:
  Let DNF be an oracle

```
return ¬(DNF(¬f))
```

- *These are proofs of the completeness of {¬, ∧, ∨}*

α-/β-formulas

- 10 (all except ≡ and ≢) of the 12 boolean connectives can be written in exactly one of the following forms: $X \circ Y = (\neg)X \wedge (\neg)Y$ or $X \circ Y = (\neg)X \vee (\neg)Y$
- **Conjunctive formulas (those that can be written using and) are called α-formulas**
- **Disjunctive formulas are called β-formulas**

More efficient normal form algorithms

- Using the laws of boolean algebra we can rewrite a formula into a normal form without having to compute the exponentially large truth table
- **CNF expansion algorithm for an arbitrary boolean formula F:**

```
acc := <[F]>
While True
    Pick arbitrary D ∈ acc such that D contains a non-literal
    return acc if there was no D
    Pick an arbitrary non-literal N ∈ D
    switch(N):
        case ¬⊤: Replace N in D with ⊥
        case ¬⊥: Replace N in D with ⊤
        case ¬¬X: Replace N in D with X
        case β1 ∨ β2: Replace N in D with β1, β2
        case α1 ∧ α2: Replace D in acc with two copies of D, one
with N replaced by α1 and one with N replaced by α2
```

  - **For DNF start with [<F>] and in-place replace for α, duplicate for β**

First order logic

Syntax

- Every first-order logic has the connectives of propositional logic, and the first-order quantifiers ($\forall$, $\exists$)
  - Each first-order logic has its own **truth functions (predicates (relations))**, and **operators (functions)**, and **constants** (or equivalently operators that don't make use of their arguments)
- **A first-order language's syntax is entirely determined by: countable set R of predicate symbols, countable set F of function symbols, and countable set C of constant symbols** — we denote this language L(R, F, C)
- F_t = family of terms = {x: x is a variable or x is a constant or $\exists f \in F$: $\exists t_1, \ldots, t_n \in F_t$: $x = f(t_1, \ldots, t_n)$}
- A relation $A(t_1, \ldots, t_n)$ is an atomic formula iff $A \in R$ and $t_1, \ldots, t_n \in F_t \cup \{\bot, \top\}$ — let F_a = the set of atomic formulas
- F_f = family of formulas = {B: $B \in F_a$ or $\exists B' \in F_f$: $B = \neg B'$ or $\exists B_1, B_2 \in F_f$: $B = B_1 \circ B_2$ where $\circ$ is a propositional connective or $\exists B' \in F_f$: ($B = \forall x.\ B'$ or $B = \exists x$: B') where x is a variable}
- **A variable if bound iff it is attached to a quantifier**
  - Note variables with the same name can be both bound and free as **quantifiers create local scopes**
- **A variable is free iff it is not attached to a quantifier**
- **A formula is a sentence iff it is a closed formula iff there are no free variables**

Semantics: Models

- **A model for L(R, F, C) is M = (D, I) where the non-empty set D is the domain and I is the interpretation, $\forall c \in C: c^I \in D$ and $\forall f \in F. f^I = f: D^n \mapsto D$ where n is the arity of f and $\forall A \in R. R^I = R' \subseteq D^n$ where n is the arity of A**
    - *We cannot actually use quantifiers and functions to define first-order logic as this would be a circular definition, but this is a compact way of notating the idea at play*
- **x^A = the image (which will be a value in the domain D) of the variable x under the assignment A**
- Given a first-order language L(R, F, C), a model M=(D, I) for this language, and an assignment A in this model, we can associate a value $t^{I, A}$ to each term:
    - If t is a constant c, $t^{I, A} = c^I$
    - If t is a variable x, $t^{I, A} = x^A$
    - If t is an n-ary function f with arguments $t_1, \ldots, t_n$, $t^{I,A} = f^I(t_1^{I, A}, \ldots, t_n^{I, A})$

Semantics: Truth values

- Truth value of a formula $\phi = \phi^{I,A}$
- If $\phi \in \{\top, \bot\}$, then $\phi^{I,A} = \phi$
- If $\phi = R(t_1, \ldots, t_n)$, then $\phi^{I,A} = \top$ iff $(t_1^{I,A}, \ldots, t_n^{I,A}) \in R^I$ [so $\phi^{I,A} = \bot$ iff $(t_1^{I,A}, \ldots, t_n^{I,A}) \notin R^I$]
- If $\phi = \forall x.\ \sim$, then $\phi^{I,A} = T$ iff $\sim^{I,A'} = T$ for all A' that differ from A at most by the value assigned to X [so $\phi^{I,A} = \bot$ iff $\sim^{I,A'} = \bot$ for some assignment A' that differs from A at most by the value assigned to X]
- If $\phi = \exists x$: $\sim$, then $\phi^{I,A} = T$ iff $\sim^{I,A'} = T$ for some A' that differs from A at most by the value assigned to X [so $\phi^{I,A} = \bot$ iff $\sim^{I,A'} = \bot$ for all assignments A' that differ from A at most by the value assigned to X]
- If $\phi = \neg X$, then $\phi^{I,A} = \neg(X^{I,A})$
- If $\phi = X \circ Y$, then $\phi^{I,A} = X^{I,A} \circ Y^{I,A}$
- A set S of formulas is satisfiable in a given model M = (D, I) iff there exists an assignment A such that $\forall \phi \in S.\ \phi^{I,A} = \top$
- **A formula ϕ is true <u>in a given model</u> M = (D, I) iff for all assignments A $\phi^{I,A} = \top$**
- **A formula ϕ is <u>valid</u> iff it is true in all models of the language** that is that it is true for all definitions and all domains
- Proof systems tend to only deal with validity as then semantics can be abstracted away

γ-/δ-formulas

- **Formulas of the form ∀x. ϕ or ¬∃x: ϕ are γ-formulas** — gamma for forall
- **Formulas of the form ∃x. ϕ or ¬∀x: ϕ are δ-formulas** — delta for exists
- Let γ be an arbitrary γ-formula, then **γ(t) = (¬)ϕ{x/t} = (¬)ϕ with all occurrences of x that were bound to the outermost quantifier replaced with the value t**
  - Symmetrically, let δ be an arbitrary δ-formula, then δ(p) = (¬)ϕ{x/p}
- Par = countable set of parameters — Par ∩ C = ∅
  - $L(R, F, C)^{Par} = L(R, F, C \cup Par)$

Proof systems

Misc

- **$S \models X$ (formula X is a consequence of the set of formulas S) iff $X = \top$ under every model** (i.e. valuation if not in FOL) **for which every member of $S = \top$**
- **X is a tautology iff $\varnothing \models X$** which we abbreviate to $\models X$
- Refutation systems show that X is a tautology by showing that $\neg X$ is a contradiction — do not forget to negate X under exam pressure!
  - Tableau and resolution are refutation systems, natural deduction is not
- Theorem: Tableau, resolution, and natural deduction are each sound and complete. That is $S \models X$ iff $S \vdash_t X$ iff $S \vdash_r X$ iff $S \vdash_d X$
  Proof: Out of scope

Semantic tableau

- **A branch = a path from the root to a leaf**
- Each branch represents the conjunction of its nodes
- The tree represents the disjunction of its branches
- **Start with a single node (the root) ¬X**
- **If a branch contains ¬⊤ we can add ⊥ to the end of it**
- **If a branch contains ¬⊥ we can add ⊤ to the end of it**
- **If a branch contains ¬¬X we can add X to the end of it**
- **If a branch contains $\alpha_1 \wedge \alpha_2$ we can add α_1 to the end of it with α_2 as a child of α_1**
- **If a branch contains $\beta_1 \vee \beta_2$ we can add children β_1 and β_2 to the end of it**
- **If a branch contains γ, we can add γ(t) to the end of it where t is a <u>chosen</u> closed (uses no variables) <u>term</u> of L^{par} (in practice probably a used parameter)**
- **If a branch contains δ, we can add δ(p) to the end of it where p is an <u>arbitrary parameter</u> (must have not yet been used)**

Semantic tableau

- S-introduction rule: **When showing S ⊨ X, any Y ∈ S can be added to any branch at any point**
- Rules of thumb: Prefer doing α-expansions before β-expansions to reduce branching; Prefer doing δ-expansions before γ-expansion to inform the choice; Do S-introductions early to cover many branches in one go
    - With the obvious tweaks, these hold for all our proof systems
- Strictness condition (removes risk of cycling (except on γ nodes) without affecting completeness): **Each node can be expanded at most once <u>in each branch except for γ nodes</u>** which can have countably infinitely many values assigned
- **A branch is closed iff a formula and its negation occur on the branch or ⊥ occurs on the branch**
    - Atomically closed is defined the same way but atom instead of formula
- **We have a tableau proof for X ((S) $\vdash_t$ X) iff the tableau for X is closed iff <u>all</u> its branches are closed**

Tableau: An example

Proposition: $(\forall x.(P(x) \vee Q(x))) \rightarrow (\exists x; P(x) \vee \forall x. Q(x))$

Proof: 1. $\neg[\forall x(P(x) \vee Q(x)) \rightarrow \exists x(P(x)) \vee \forall x(Q(x))]$

2. $\forall x.(P(x) \vee Q(x))$

3. $\neg[\exists x(P(x)) \vee \forall x(Q(x))]$

4. $\neg\exists x(P(x))$

5. $\neg\forall x(Q(x))$

6. $\neg Q(a)$

7. $\neg P(a)$

8. $P(a) \vee Q(a)$

9. $P(a)$ $\quad\bot$

10. $Q(a)$ $\quad\bot$

α-E on 1 gives 2+3

γ-E on 3 gives 4+5

δ-E on 5 gives 6

$\forall$-E on 4 gives 7

$\forall$-E on 2 gives 8

β-E on 8 gives 9+10

7+9 gives $\bot$

6+10 gives $\bot$

Resolution

- Start with <[¬X]>
- **Run CNF expansion algorithm, appending the new clause on each expansion step**
- **If a clause contains γ, we can create a new clause with γ replaced with γ(t)**
- **If a clause contains δ, we can create a new clause with δ replaced with δ(p)**
- Resolution rule: **If there are clauses D_1, D_2 such that D_1 contains a formula X and D_2 contains ¬X, we can create a new clause of D_1 with X removed concatenated with D_2 with ¬X removed**
    - Trivial resolution: **If a clause contains ⊥, we can create a new clause of it with ⊥ removed**
- S-introduction rule: **When showing S ⊨ X, any Y ∈ S can be added as a singleton clause at any point**
- Strictness condition: **Each clause is expanded at most once, <u>except for γ expansions</u>** which can occur countably infinitely many times
    - <u>We can though apply the same clause to different clauses in the resolution rule</u>
- **We have a resolution proof for X ((S) $\vdash_r$ X) iff we have the empty clause []**

Resolution: An example

Proposition: $(\forall x.\, P(x) \wedge Q(x)) \rightarrow (\forall x.\, P(x) \wedge \forall x.\, Q(x))$

Proof:

1. $[\neg((\forall x(P(x) \wedge Q(x)) \rightarrow (\forall x(P(x)) \wedge \forall x(Q(x)))))]$
2. $[\forall x(P(x) \wedge Q(x))]$
3. $[\neg(\forall x(P(x)) \wedge \forall x(Q(x)))]$
4. $[\neg\forall x(P(x)), \neg\forall x(Q(x))]$
5. $[\neg P(a), \neg Q(b)]$
6. $[P(a) \wedge Q(a)]$
7. $[P(a)]$
8. $[Q(a)]$
9. $[P(b), Q(b)]$
10. $[P(b)]$
11. $[Q(b)]$
12. $[\neg Q(b)]$
13. $[\,]$

α-E on 1 gives 2+3
β-E on 3 gives 4
2x δ-E on 4 gives 5
γ-E on 2 gives 6
α-E on 6 gives 7+8
γ-E on 2 gives 9
α-E on 9 gives 10+11
Resolution on 5+7 gives 12
Resolution on 11+12 gives 13

Natural deduction

- Each box is a lemma
- **The first line of each box is an assumption**
- **Formulas that are outside a box or in a box that is currently open are active (in scope to be used)**
- When depicting inference rules, everything above the line is a precondition and everything below the line is a consequence
  - For ease of production, in these notes we will rewrite vertically stacked $X_1 \ldots X_n$ LINE vertically stacked $Y_1 \ldots Y_n$ as $((X_1) \wedge \ldots \wedge (X_n)) \Rightarrow ((Y_1) \wedge \ldots \wedge (Y_n))$
  - In natural deduction, the precondition may be a box with the line before the consequence being the bottom of the box
    - For ease of production, in these notes we will rewrite this as $\underline{\text{BOX}}((X_1) \wedge \ldots \wedge (X_n)) \Rightarrow ((Y_1) \wedge \ldots \wedge (Y_n))$

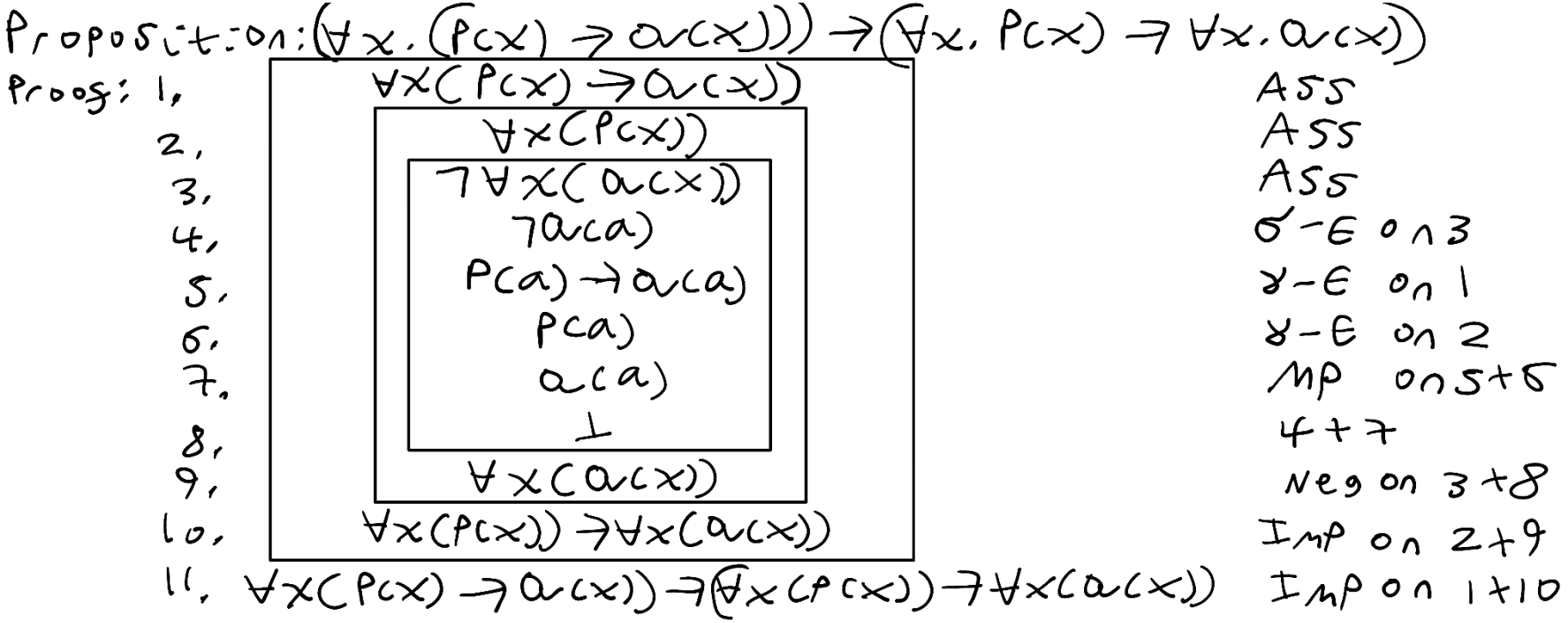
Natural deduction: Rules: Base rules

- Constant rules: $\bot \Rightarrow X$, $X \Rightarrow \top$
- **Negation rules: $(X \wedge \neg X) \Rightarrow \bot$, $\underline{BOX}(X \wedge \ldots \wedge \bot) \Rightarrow \neg X$, $\underline{BOX}(\neg X \wedge \ldots \bot) \Rightarrow X$**
- **Introduction rules: $((X) \wedge (Y)) \Rightarrow (X \wedge Y)$, $\underline{BOX}(\neg X \wedge \ldots \wedge Y) \Rightarrow (X \vee Y)$**
- **Elimination rules: $((X \wedge Y)) \Rightarrow X$, $((X \wedge Y)) \Rightarrow Y$, $\underline{BOX}(\neg X \wedge (X \vee Y)) \Rightarrow Y$, $\gamma \Rightarrow \gamma(t)$, $\delta \Rightarrow \delta(p)$**
- **S-introduction rule: When showing $S \vDash X$, any $Y \in S$ can be added as a <u>premise</u>** (like an assumption but don't open a new box)
  - Easiest to do before opening any boxes to maximize scope

Natural deduction: Rules: Derived rules

- **Double negation: ¬¬X ⇛ X, X ⇛ ¬¬X**
- Copy: X ⇛ X (can write an active formula again at any point)
- **Implication: BOX(X ∧ … ∧ Y) ⇛ X → Y**
- **Modus ponens: (X ∧ (X → Y)) ⇛ Y**
- Modus tollens: (¬Y ∧ (X → Y)) ⇛ ¬X
- Excluded middle: X ⇛ (Y ∨ ¬Y)

Natural deduction: An example



Satisfiability (probably skip in exam)

Reductions

- **Theorem: For all k, k-SAT $\leq_P$ 3-SAT**
  Proof:
  k=1, 2, 3 are trivial (they are 3-SAT). Suppose k>3.
  We can decrease the size of a clause by replacing some x, y with a new variable z and adding new clauses to capture $z \leftrightarrow (x \vee y) = (z \rightarrow (x \vee y)) \wedge (z \leftarrow (x \vee y)) = (x \vee y \vee \neg z) \wedge ((\neg x \wedge \neg y) \vee z) = (x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$. That is, **we can rewrite <..., [a, b, x, y], …> as <..., [a, b, z], [x, y, ¬z], [¬x, z], [¬y, z], …>.** We can repeat this process for as long as we have clauses with more than 3 literals and will eventually stop as we are not adding any new such clauses
- Theorem: For all k, k-COLOURING $\leq_P$ SAT
  Proof: Let $x_{v, k} = \top$ iff vertex v has been assigned colour k. Let V be the set of vertices, E be the set of edges, and C={1, …, k} be the set of colours.
  At most one colour is assigned to each vertex: $\wedge_{v \in V} \wedge_{(i, j) \in C^2} (i=j \vee \neg x_{v, 1} \vee \neg x_{v, 2})$
  At least one colour is assigned to each vertex: $\wedge_{v \in V} \vee_{i \in C} x_{v, i}$
  No neighbours have the same colour: $\wedge_{(u, v) \in E} \wedge_{i \in C} (\neg x_{u, i} \vee \neg x_{v, i})$
  Taking the conjunction of the above 3 constraints gives a CNF formula which is satisfiable iff G=(V, E) is k-colourable

Resolution

- Theorem: **2-SAT ∈ P**
  Proof:
  As our formula is already in CNF we only need to apply the (strict) resolution rule exhaustively and see whether we get the empty clause. Over n variables, there are exactly $2^i \times nCi$ possible clauses of size exactly i [nCi combinations of variables, each element may be negated]. **The maximum clause size generated by applying the resolution rule is 2-1+2-1 = 2 meaning we generate at most $1+2n+4 \times nC2 \in O(n^2)$ clauses** and so take $O(n^4)$ time overall as each application of the resolution rule is $O(n^2)$ and we apply it $O(n^2)$ times.
  - Applying this technique to 3-SAT gives us a maximum clause size of 3-1+3-1 = 4 which could then lead to a clause of size 4-1+3-1 = 5 and so on leading to exponential work
- *3-SAT ∈ NP*
- Proposition: 3-SAT can be solved in $O(\text{cuberoot}(7)^n)$
  Proof: **A subset of clauses is called independent iff they are pairwise disjoint with respect to variables. Suppose we have a 3-SAT formula F and a maximal (cannot add anymore clauses with violating independence) set of independent clauses S.**
  $|S| \leq n/3$ as each clause added to S removes 3 variables from being able to occur in any other clauses in S.
  As S is maximal every clause contains at least one variable that occurs in S, **So, for any valuation of S, we can construct a 2-SAT instance that is satisfiable iff F is satisfiable under the partial assignment made by S, The construction algorithm is if a literal is true delete every clause it occurs in, if a literal is false delete it from every clause it occurs in].**
  Finally, there are $7^{|S|} \leq 7^{n/3}$ assignments to G we need to consider, as G is an independent set and there are 8 possible valuations of each clause but we know setting every literal in a clause to $\perp$ will not be a satisfying assignment. As 2-SAT ∈ P, this time to generate the assignments dominates.

Implication graph

- Alternative proof for 2-SAT $\in$ P:
  Create a directed graph with 2n nodes, one for each atom and its negation, and an edge (x, y) iff x $\rightarrow$ y is a clause. Note that an edge (¬x, x) means ¬¬x $\vee$ x = x is a clause. Note that there is an edge (x, y) iff there is an edge (¬y, ¬x) [contrapositive].
  As there is at most 2 edges for each clause, we can construct the graph in polynomial time.
  The formula is satisfiable iff there for every variable x, x and ¬x are not strongly connected (there does not exist a directed cycle containing both x and ¬x). This is because edges correspond to implications and implication and reachability are both transitive relations.

Horn clauses

- **A clause is a horn clause iff it contains at most one positive literal** e.g. an implication with antecedent of a single (possibly negated) literal *(called the head of the clause)* and consequent of conjunction of positive literals *(called the body of the clause)*
    - *Prolog statements are horn clauses*
- HORN-SAT: Determine whether a CNF that contains only horn clauses (but of unbounded length) is satisfiable
- **HORN-SAT ∈ P:**
  If there are no clauses, the formula is trivially satisfiable (this holds for any form of SAT). If there is an empty clause [], the formula is trivially unsatisfiable (this holds for any form of SAT).
  If there is no clause of size less than 2, every clause must contain a negative literal so setting all variables to false is a satisfying assignment. If there is a clause of size 1, the corresponding literal must be set to true, this will give a new horn-cnf so we can recurse to determine satisfiability.
    - In fact with some implementation tricks we can get linear time

SAT-Solvers: DPLL

- Naive test-and-set SAT-solver (returns the set of all literals that are assigned ⊤):
  Let F|l denote F with l set to ⊤ (all clauses that contain l removed, and all occurrences of ¬l removed)

```
Back(F) throws Failure
    return ∅ if F.length == 0 # any assignment would work
    fail if [] ∈ F
    Pick an arbitrary literal l that occurs in F
    try
        A = Back(F|l)
        return A ∪ {l}
    except Failure
        A = Back(F|¬l)
        return A ∪ {¬l}
```

- **DPLL** is test-and-set with pre-processing of the formula at the start of each call to definitely reduce the number of recursive calls made:
  - **Unit clause progoation: Clauses of length 1 immediately tell us what the necessary assignment for their literal is**
  - **Pure literal elimination: Literals that only occurs positively (or equivalently negatively) in the formula are called pure and tell us a sufficient assignment for themself**

SAT-Solvers: Heuristics

- We can use heuristics to determine which literal to pick and whether to set it to $\top$ or $\bot$ first
- Idea: Try to create unit clauses quickly
- **DLIS** (Dynamic Largest Individual Sum) **always chooses the literal with the most occurences to hopefully reduce the number of recursive calls made**
- **MOMS** (Maximum Occurrence In Clauses Of Minimum Size) **always chooses the literal with maximal occurrence count over the clauses of minimum size to hopefully reduce the number of recursive calls made**

SAT-Solvers: Conflicts

- At each stage of the backtracking search we can produce an implication graph
  - **Draw solid circles for each literal that has been assigned $\top$ (decision literals)**
  - If we have a clause in which every literal is set to false except one which is not yet assigned, an assignment is satisfying only if the unset literal is set to true: **If we have a clause $[l_1, \ldots, l_k, l]$ such that $\neg l_1, \ldots, \neg l_k$ are in the graph but l is not, then l should be added to the graph (as an unfilled circle) and along with edges $(\neg l_1, l), \ldots, (\neg l_k, l)$**
    - Apply exhaustively
- **A literal l is a conflict literal iff l and $\neg l$ both appear in the implication graph** and so our current partial assignment cannot lead to a satisfying assignment
- A **subgraph of the implication graph is a conflict graph iff it contains <u>exactly</u> one conflict literal and every node can reach either the conflict literal or its negation and no node has incoming edges from more than one clause**
- **Given a cut-set S that partitions the graph into (R, C) s.t. All decision literals are in the R (reason) partition and the conflict literal and its negation are in C (conflict) partition we can construct a conflict clause $\neg\bigwedge_{(a,\, b) \in S} a = \bigvee_{(a,\, b) \in S} \neg a$**
  - **Conjuncting in a conflict clause preserves satisfiability** (in fact logical equivalence as it is a form of the resolution rule) and tells us that a certain subset of our current partial assignment will never work

Monte-Carlo SAT-Solving

- *Starting with a random total assignment and flipping a random variable involved in an unsatisfied clause at each step until the formula is satisfied or we have done 3N (N = number of trials) flips will find a satisfying assignment for a satisfiable 3-SAT formula with n variables with probability $\Omega((3/4)^n/n)$ with n trials — we can run this many times to drive up our probability of success*

Program verification (Hoare logic) (sketch notes)

Principles

- Post-condition = a first-order logic formula that is true when the program terminates
- Pre-condition = a first-order logic formula that will be true when the program starts
- Any program can be rewritten as a sequence of instructions C_1; …; C_n where each C_i is an assignment or a while loop or a if/else
- A Hoare triple ⦇Pre⦈Prog⦇Post⦈ is a logical assertion that if Prog is carried out on a state satisfying Pre then the resulting state (if it exists) will satisfy Post — a Hoare triple is called valid iff it is true (however if the precondition is ⊥ it will not be useful)
- wp(Prog, Post) = weakest (least restrictive) possible precondition
  - ⦇wp(P, Post)⦈Prog⦇Post⦈ and ∀Pre ∈ {Pre: ⦇Pre⦈Prog⦇Post⦈}. Pre ⇒ wp(P, Post)
- To produce proofs we will work our way backwards thinking about the rules (next slide) to derive wp(Prog, Post) and show that Pre ⇒ it
- To check proofs we will work our way forwards to derive the strongest (most restrictive) possible post condition for Pre, Prog using the rules (next slide) and verify that it ⇒ Post

Inference rules for Hoare triples

- **Assignment rule: (|Post[x/E]|)x=E(|Post|)**
    - **Post[x/E] = Post with all occurrences of x replaced by E** — to simplify the result of this substitution, use the implied rule
        - Note we replace the LHS with the RHS when going backwards and so the RHS with the LHS when going forwards
- **Implied rule: (Pre ⇒ P ∧ (|P|)Prog(|Q|) ∧ Q ⇒ Post) ⇒ (|Pre|)Prog(|Post|)**
    - The other rules give us the weakest precondition for the given postcondition but this tell us **we can strengthen** (make more restrictive) **preconditions and weaken** (make less restrictive) **postconditions**
- **Composition rule: ((|Pre|)$Prog_1$(|Mid|) ∧ (|Mid|)$Prog_2$(|Post|)) ⇒ (|Pre|)$Prog_1$; $Prog_2$(|Post|)**
- **Conditional rule: ((|Pre ∧ B|)C_1(|Post|) ∧ (|Pre ∧ ¬B|)C_2(|Post|)) ⇒ (|Pre|)if B {C_1} else {C_2}(|Post|)**
- **Loop rule: (|B ∧ L|)C(|L|) ⇒ (|L|)while B {C}(|¬ B ∧ L|)**
    - B is the loop break condition and so will not hold at the end of the loop (and may never hold) but will hold at the start of each execution of C
    - L is the loop invariant and so must hold at the start of the loop and at the instant it ends and at every execution of C
    - Requires insight to **pick an L such that Pre ⇒ L and (¬B ∧ L) ⇒ Post**
        - Typically a condition that is at least as strong as B but unlike B still holds when the loop ends, and also captures how the loop is progressing towards the post condition

Conditional example

```
{ T }
{ x+1 = x+1 }  Implied
a = x+1
{ a = x+1 }  Assignment
if (a-1 = 0) {
    { a-1 = 0 ∧ a = x+1 }
    { 1 = x+1 }  Implied
    y = 1
    { y = x+1 }  Assignment
}
else {
    { ¬(a-1 = 0) ∧ a = x+1 }
    { a = x+1 }  Implied
    y = a
    { y = x+1 }  Assignment
}
{ y = x+1 }  Conditional
```

Loop example

$q \geq 1$ D

$q \geq 1 \wedge 0 = 0$ D Implied

$y := 0$

$q \geq 1 \wedge y = 0$ D Assignment

$x := z$

$q\ x \geq 1 \wedge y = 0$ D Assignment

while $(x > 0)$ {

$q\ x > 0 \wedge x \geq 0 \wedge 2x + y = 2z$ D

$q\ x \geq 1 \wedge 2x + y = 2z$ D Implied

$q\ x - 1 \geq 0 \wedge (2x - 1 + 1) + y + 2 - 2 = 2z$ D Implied

$x := x - 1$

$q\ x \geq 0 \wedge 2(x+1) + y + 2 - 2 = 2z$ D Assignment

$y := y + 2$

$q\ x \geq 0 \wedge 2(x+1) + y - 2 = 2z$ D Assignment

$q\ x \geq 0 \wedge 2x + y = 2z$ D Implied

}

$q \wedge \neg(x > 0) \wedge x \geq 0 \wedge 2x + y = 2z$ D Loop

$q\ x = 0 \wedge 2x + y = 2z$ D Implied

$q\ y = 2z$ D Implied

Prolog (sketch notes)

Prolog

- Each line of a prolog program x :- y, z, … is a Horn clause $x \Leftarrow y \wedge z \wedge \ldots$
- Strings that start with lowercase are a predicate (function which is true and may take arguments), or a constant (a symbol)
- Strings that start with lowercase are a variable — prolog will calculate a list of all bindings to it that make the predicate it occurs in true
- Knowledge base is checked from top to bottom, left to right for a match, backtracking to continue on to the next match once the query fails (or succeeds and continues to try to find another such binding)
- =:= is equality comparison
- =\= is negation of equality comparison
- X = … binds X to RHS (assigns RHS to X) without evaluating RHS
- X is … evaluates RHS and binds the result to X
- _ is used where we are not using a value and so don't need to give it a name
- ! is the shortcut operator, it prevents backtracking through the point it occurs, thus committing to the choice made when it was encountered
- [H|T] matches H to the first element in a list and T to the list with H removed — will match a one element list with T = [] but will not match the empty list

Examples

- ```
  mod(X, Y, X) :- X<Y, !.
  mod(X, Y, R) :- XSY is X-Y, mod(XSY, Y, R)
  ```
- ```
  get([H | _], 0, H) :- !.
  get([_| T], I, X) :-  J is I-1, get(T, J, X).
  ```
- ```
  prepend(L, X, [X | L]).
  ```
- ```
  append([], X, [X]).
  append([H | T], X, [H | R]) :- append(T, X, R).
  ```