Sat-Solvers

Satisfaction and Consequence

- **Satisfaction:** A Truth Assignment (TA) maps values to variables. A TA satisfies a formula Φ if Φ evaluates to 1 on that TA. A formula is "satisfiable" if at least one TA satisfies it
- **Logical Consequence (Entailment):** A formula Φ is a logical consequence of Ψ (written as $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ
- **The Golden Rule of SAT:** For any two propositional formulas, $\Psi \models \Phi$ if and only if the formula $\Psi \wedge \neg\Phi$ is unsatisfiable

Normal Forms (Recap):

- **Literals:** A variable (e.g., x, y, z) or its negation (e.g., $\overline{x}, \overline{y}, \overline{z}$)
- **Clauses:** A disjunction (OR) of literals, formulated as $l_1 \vee \cdots \vee l_n$. In SAT-solving, a clause is often represented simply as a set of literals, such as $\{l_1, \ldots, l_n\}$
- **Conjunctive Normal Form (CNF):** A conjunction (AND) of clauses, formulated as $C_1 \wedge \cdots \wedge C_m$. This can be thought of as a set of sets, or a clause-set: $\{C_1, \ldots, C_m\}$

Tseitin's Algorithm

- **Goal:** To convert an arbitrary propositional formula into CNF without suffering an exponential blow-up in size
- The algorithm produces a clause-set that is *equisatisfiable* to the original formula (may involve introducing extension variables), rather than logically equivalent

Equisatisfiability:

- Two formulas are either **both** satisfiable, or **both** unsatisfiable

Method:

1. Identify an innermost subformula of the form $l \circ m$, where $\circ$ is a placeholder for any binary logical connection $(\wedge, \vee, \Rightarrow, \Leftrightarrow)$
2. Add an "Extension variable" $(x_{l \circ m})$, to represent this specific subformula
3. Substitute the new variable into the main formula
4. Use the standard set of CNF clauses for that logical operator (logically equivalent to $x_{l \circ m} \iff (l \circ m))$
5. Repeat the process until the entire formula is a single literal, then combine all the generated clauses

Result:

- The resulting clause-set has a size linear to the number of binary connectives in the original formula $O(n)$
- Each clause contains at most 3 literals

Example:

- **Question:** Convert $\Psi = \neg(x \to (y \to x))$ into an equisatisfiable CNF clause-set, starting with an empty set $F = \emptyset$
  1. Innermost Subformula

$$\begin{aligned}&\text{Let } u \text{ be } y \to x \therefore u \Leftrightarrow (y \to x)\\&\text{The standard Transformations give us: } \{y, u\}, \{\overline{x}, u\}, \{\overline{y}, x, \overline{u}\}\\&\therefore \text{The main formula is } \Psi = \neg(x \to u)\end{aligned}$$

  2. Process the Next Subformula

$$\begin{aligned}&\text{Let } v \text{ be } x \to u\\&\text{The standard Transformations give us: } \{x, v\}, \{\overline{u}, v\}, \{\overline{x}, u, \overline{v}\}\\&\therefore \text{The main formula is } \Psi = \neg v\end{aligned}$$

  3. Final Clause-Set
- The remaining formula is a single literal $(\neg v)$, and we add it to the clause-set F

$$\{\{y, u\}, \{\overline{x}, u\}, \{\overline{y}, x, \overline{u}\}, \{x, v\}, \{\overline{u}, v\}, \{\overline{x}, u, \overline{v}\}, \{\overline{v}\}\}$$

The DPLL Algorithm

- Used to reduce the search space for SAT-solvers

Pure Literal Elimination

- **Pure Literals:** A literal x in a clause-set F is considered a "pure literal" if some clauses contain x, but no clause contains its exact opposite $\overline{x}$
- **Elimination:** Because $\overline{x}$ does not exist anywhere in the set, we can safely assign x to be true and completely remove all clauses that contain it, leaving a new clause-set F'
- **Equisatisfiability:** The original set F and the reduced set F' are equisatisfiable
- **Notation:** $PL(F)$ denotes the smallest possible clause-set obtained by applying pure literal elimination as many times as possible

Unit Propagation

- **Unit Clauses:** A unit clause is a clause containing only one literal $\{l\}$
- **Propagation:** If a clause-set contains $\{l\}$, that literal *must* be true $(l = 1)$ for the entire formula to be satisfied
  $\therefore$ We can propagate this forced assignment throughout the clause-set to obtain F
- **Notation:** $UP(F)$ is the clause-set obtained by applying unit propagation as often as possible until no clauses remain
- **Equisatisfiability:** F and $UP(F)$ are always equisatisfiable

Algorithm DPLL(F):

1. **Reduce with UP:** Set $F := UP(F)$
2. **Reduce with PL:** Set $F := PL(F)$
3. **Base Cases:** Check if there are any variables left to assign $(var(F) = \emptyset)$:
   - If the clause-set is empty $(F = \emptyset)$, the formula is **Satisfiable**
   - If F contains an empty clause (denoted as $\square$), it means a contradiction was reached, so it is **Unsatisfiable**
4. **Branching (Guessing):** If variables remain, pick a remaining branching variable $x \in var(F)$
5. **Recursive Search:** First, recursively run the algorithm assuming $x = 0$. If $DPLL(F[x = 0])$ returns "sat", then exit with **Satisfiable**
6. **Backtracking:** If $x = 0$ failed, try the other path. If $DPLL(F[x = 1])$ returns "sat", exit with **Satisfiable.** If both paths fail, exit with **Unsatisfiable**

DPLL Example

- **Question:** Determine the satisfiability of the following clause-set using the DPLL algorithm

$$F = \{\{x, y\}, \{\overline{x}, z\}, \{\overline{y}, \overline{z}, w\}, \{\overline{w}\}\}$$

Step 1: First Reduction

- **Unit Propagation (UP):** The algorithm always checks for unit clauses first. The unit clause $\{\overline{w}\}$ forces $w = 0$
  - Substitute $w = 0$ into F and the clause $\{\overline{y}, \overline{z}, w\}$ becomes $\{\overline{y}, \overline{z}\}$
  - The unit clause $\{\overline{w}\}$ is satisfied and removed.
  - **Result:** $F_1 = \{\{x, y\}, \{\overline{x}, z\}, \{\overline{y}, \overline{z}\}\}$
- **Pure Literal Elimination (PL):**
  - Looking at F_1, every variable (x, y, z) appears in both its positive and negated forms
  - **Results:** No pure literals exist, so F_1 remains unchanged
- **Base Case Check:** Since F_1 is neither empty, nor does it contain an empty clause $(\square)$, we proceed to branching

Step 2: Branching

- Let x be the variable to branch on (this could be any variable)
- DPLL first assumes $x = 1$ and creates a new branch
- **Substitute:** We plug $x = 1$ into our current set $F_1 = \{\{x, y\}, \{\overline{x}, z\}, \{\overline{y}, \overline{z}\}\}$
  - $\{x, y\}$ is completely satisfied because x is true, so we remove it
  - $\{\overline{x}, z\}$ becomes $\{z\}$ because $\overline{x}$ is false, forcing z to be true
  - **Result:** $F_2 = \{\{z\}, \{\overline{y}, \overline{z}\}\}$
- **Unit Propagation (UP):** DPLL runs UP on the new set. We have the unit clause $\{z\}$, forcing $z = 1$
  - Substitute $z = 1$ into F_2. The clause $\{\overline{y}, \overline{z}\}$ becomes $\{\overline{y}\}$
  - **Result:** $F_3 = \{\{\overline{y}\}\}$

- **Unit Propagation (Again):** DPLL runs UP again. We have the unit clause $\{\overline{y}\}$, forcing $y = 0$.
  -
    - Substitute $y = 0$ into F_3. The clause is satisfied and removed.
  - **Result:** $F_4 = \emptyset$

Step 3: The Conclusion

- **Base Case Hit:** Because the clause-set is now completely empty ($F_4 = \emptyset$), the algorithm immediately exits and returns **sat** (Satisfiable).
- **Note:** Because we found a satisfying assignment on our first guess ($x = 1$), the algorithm does not need to backtrack or check the $x = 0$ branch

---

Conflict-Driven Clause Learning (CDCL)

- Instead of recursion, CDCL is an iterative version of DPLL. When the solver makes a series of guesses that lead to a dead end (a conflict), clauses of conflict are cached and learnt. These learnt clauses allow us to prune the search space later on.

Method:

1. **Initialise:** Start with the given clause-set F and an empty assignment τ
2. **Propagate:** Apply Unit Propagation. If the clause-set is fully satisfied $(UP(F[\tau]) = \emptyset)$, exit and return **"sat"**
3. **Top-Level Conflict:** If unit propagation produces an empty clause $\left(\Box \in UP(F[\tau])\right)$, while our assignment is still empty ($\tau = \emptyset$), the formula is fundamentally broken. Exit and return **"unsat"**
4. **Learn and Backtrack:** If we hit an empty clause $\left(\Box \in UP(F[\tau])\right)$ after making some decisions, a conflict has occurred
   - We generate a new clause: $C = \text{CONFLICT-CLAUSE}(F, \tau)$
   - We add this learnt clause to our set: $F := F \cup \{C\}$
   - We undo our recent bad decisions: $\tau := \text{BACKTRACK}(\tau)$
5. **Decide:** If there is no conflict, choose an unassigned variable x, guess a value $b \in \{0, 1\}$, and add it to our current assignment $\tau := \tau \cup \{(x, b)\}$. Then Loop back to step 2

Implication Graph:

A directed graph built during Unit Propagation. Used to find *why* a conflict happened (to learn a useful clause before backtracking)

- **Nodes:** The nodes of graph G are the literals that are currently true
- **Decision Literals (Sources):** The variables we actively guessed (Step 5) from the sources of G and have no incoming edges
- **Implied Literals:** If we have a clause $\{l_1, \dots, l_k, l\}$ where $\overline{l_1}, \dots, \overline{l_k}$ are already true in our graph, then l is forced to be true via Unit Propagation. We add l to the graph and draw edges from $\overline{l_i}$ to l to show *why* it was forced
- **Conflict Literals:** If the graph ends up forcing two complementary literals to be true (e.g. x and $\overline{x}$), then a conflict has occurred. We draw edges from these conflict literals to a special "□" node

---

Conflict Graphs

When our Implication Graph reveals a conflict (two forced variables contradicting each other, like y and $\overline{y}$), we extract a smaller, focused graph called a **Conflict Graph**

- **Purpose:** It clearly represents a single, direct cause of the conflict, tracing back from the contradicting variables to the root decision variables
- **Key Properties:** Unlike the main implication graph (which can have loops), a conflict graph is always acyclic (no directed cycles). It must contain exactly one pair of conflict variables

---

Conflict Clauses

Once we have our Conflict Graph, we need to generate a new clause to "learn" so we never make this exact sequence of bad decisions again

- **The Cut:** We draw a line to divide the conflict graph into two halves: the **Reason Side (RS)** and the **Conflict Side (CS)**

- **Rules:** The Reason Side must contain all of the original decision literals (our guesses). The Conflict Side must contain at least one of the conflict literals
- **Forming the Clause:** Take the *opposites (complements)* of the literals in the Reason Side that have an arrow pointing to a literal in the Conflict Side (arrows that cross the line). And OR these literals to form the **Conflict Clause** (C)
- **Result:** This new clause is added to our main clause-set. Because there are many ways to draw the cut, sat-solvers use strategies to pick the best possible Conflict Clause

---

Decision Level

- **Definition:** A decision level consists of a decision literal (our guess) and all the implied literals that were forced to be true immediately after that guess was made
- **Tracking:** By tracking decision levels, the solver knows exactly which chain of deductions belongs to which specific guess

Unique Implication Points (UIPs)

Choosing where to draw the "cut" to find the most efficient bottleneck in the implication graph

- **Definition:** A Unique Implication Point (UIP) is a literal l on the *current* decision level such that every single path from the current decision literal to the conflict literals must run through l
- **Logic:** Think of a UIP as a single, fundamental reason for a conflict. If you remove the UIP, the conflict cannot happen. Note that the current decision literal itself is always considered a UIP
- **Strategy:** Some sat-solvers choose a cut that specifically places a UIP into the Reason Side (RS) and all of its successors into the Conflict Side (CS). This produces a highly effective conflict clause

---

Non-Chronological Backtracking (Backjumping)

Normal Backtracking simply undoes the most recent assignment and tries the opposite, this is slow when the root cause of the conflict was made many levels ago.

- **Solution:** Backjumping allows the solver to bypass recent, irrelevant guesses and jump further back up the decision tree

Mechanism: (Assuming we learnt a clause using a UIP)

1. If the newly learnt clause is a unit clause (only one literal), NCB backtracks to the beginning (an empty assignment, $\tau = \emptyset$)
2. Otherwise, NCB evaluates the literals in the newly learnt clause and backtracks to the second *latest* decision level found among them
   Result:

- When the solver jumps back, the newly learnt clause triggers Unit Propagation at that higher level, steering the search away from the entire failure state

---

CDCL Example

Question:

- Let F be the clause-set consisting of the following clauses:
  - $C_1 = \{x_1, x_2\}$
  - $C_2 = \{x_1, x_3, x_7\}$
  - $C_3 = \{\overline{x_2}, \overline{x_3}, x_4\}$
  - $C_4 = \{\overline{x_4}, x_5, x_8\}$
  - $C_5 = \{\overline{x_4}, x_6, x_9\}$
  - $C_6 = \{\overline{x_5}, \overline{x_6}\}$
    Assume the solver makes the following sequence of decisions (guesses) in this exact order: $\tau = \{(x_7, 0), (x_8, 0), (x_9, 0), (x_1, 0)\}$

Draw the Implication graph, identify a UIP, find a clause that can be learnt, and determine the level for non-chronological backtracking

---

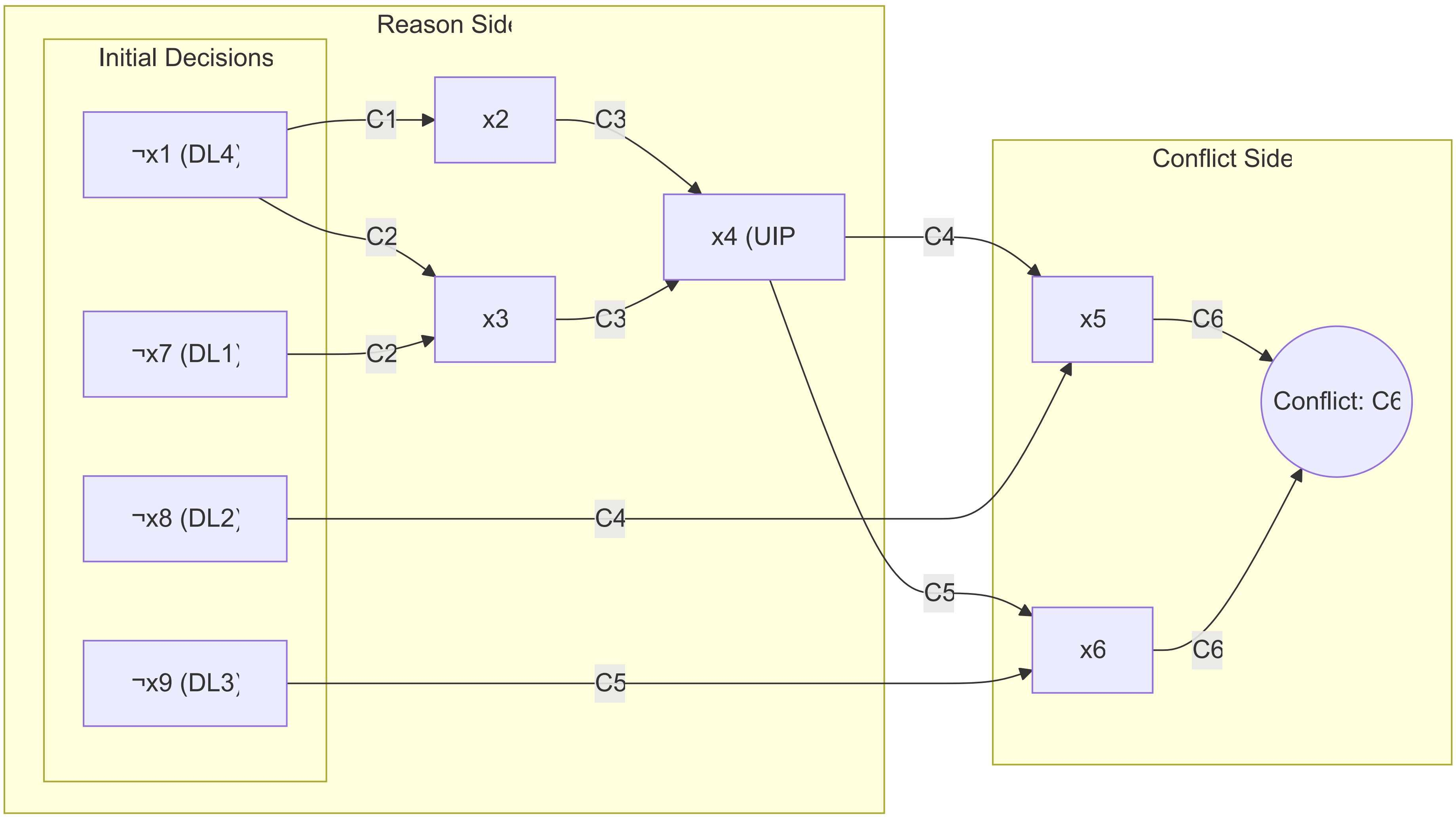
Step 1: Decision Levels & Unit Propagation

Map out the decisions and see what they force to happen. A value of 0 means the literal is negated (e.g. $x_7 = 0$ means $\overline{x_7}$ is true).

1. **Decision Level 1:** $\overline{x_7}$ is True (No unit clauses triggered)
2. **Decision Level 2:** $\overline{x_8}$ is True (No unit clauses triggered)
3. **Decision Level 3:** $\overline{x_9}$ is True (No unit clauses triggered)
4. **Decision Level 4:** $\overline{x_1}$ is True
   Unit Propagation:
   - Because $\overline{x_1}$ is true, C_1 forces x_2 to be true
   - Because $\overline{x_1}$ and $\overline{x_7}$ are true, C_2 forces x_3 to be true
   - Because x_2 and x_3 are true, C_3 forces x_4 to be true
   - Because x_4 and $\overline{x_8}$ are true, C_4 forces x_5 to be true
   - Because x_4 and $\overline{x_9}$ are true, C_5 forces x_6 to be true
   - **Conflict:** C_6 requires either $\overline{x_5}$ or $\overline{x_6}$ to be true, but we just forced both x_5 and x_6 to be true

---

Step 2: The Implication Graph



Step 3: Finding a UIP and Making the Cut

Identify the UIP

- A UIP is a node on the *current* decision level (DL4) that acts as a bottleneck. All paths from $\overline{x_1}$ to the conflict must pass through x_4
- $\therefore x_4$ **is a UIP**
  Note: *$\overline{x_1}$ is also a UIP, but x_4 is closer to the conflict*

The Cut:

- We draw a line placing the UIP (x_4) and all earlier decisions into the Reason Side (RS), and everything after it $(x_5, x_6,$ and the conflict) into the Conflict Side (CS)

Step 4: Learning the Conflict Clause

- Look at the nodes in the Reason Side that have arrows pointing directly into the Conflict Side $(x_4, \overline{x_8},$ and $\overline{x_9})$
- **The Clause:** We take the exact opposite (complement) of these nodes and OR them together to form the new learnt clause
- **Result:** The learnt clause is $C_{learnt} = \{\overline{x_4}, x_8, x_9\}$

Step 5: Non-Chronological Backtracking (Backjumping)

Undoing our mistakes with the clause we just learnt

- **Analyse the Learnt Clause:** Look at the decision levels of the variables in our new clause $\{\overline{x_4}, x_8, x_9\}$:
  - x_4 was forced during **Level 4**
  - x_8 was decided at **Level 2**
  - x_9 was decided at **Level 3**

The Backjump Rule:

- Because this is not a unit clause, Non-Chronological Backtracking tells us to jump to the **second latest** decision level present in the learnt clause

The Result:

- The latest level is 4. The second latest level is 3. Therefore, the solver **backtracks to Decision Level 3**

Heuristics & Optimisation

- Solvers prefer variables that appear in many short clauses, as this facilitates and triggers Unit Propagation much faster

1. MOMS

- Choose the literal with the **M**aximum number of **O**ccurrences in **M**inimum **S**ize clauses

2. Jeroslow-Wang

- Assigns a weight to each variable based on the length of the open clauses it appears in. It give exponentially more weight to variables in shorter clauses
- **Formula:** Let $C(x)$ be the set of open clauses containing either polarity of a given variable x. The weight $w(x)$ is defined as:

$$w(x) = \sum_{c \in C(x)} 2^{-|c|}$$

- **Note:** $|c|$ is the length of the clause, so the negative power is taken to penalise longer clauses
- **Decision:** The solver chooses the variable with the maximum weight $w(x)$ as the next branching variable

3. VSIDS (Variable State Independent Decaying Sum)

- Highly effective and biases the solver to recently learnt clauses
- **Counter:** For each literal, the solver keeps a counter of how many *learnt* clauses it appears in
- **Bias:** Periodically, the solver divides all counters by a constant. This "decays" the value of older clauses, ensuring the solver focuses on recent conflicts
- **Decision:** At each decision point, it chooses the literal with the highest counter

4. Optimising Unit Propagation (UP)

- 80% to 90% of a solver's run-time is spent executing Unit Propagation
- **Naïve Optimisation:** For each clause, the solver keeps a strict count of how many literals are satisfied, falsified, and unresolved. When a literal is assigned or unassigned, the solver visits *all* clauses containing that literal to update the counters
- **Drawback:** Continuously updating counters for every assignment and unassignment (during backtracking) is computationally very expensive

---

Resolution & Proof Complexity

Resolution Recap:

- Resolution
- **Example:** If we have $C = \{x, y, \overline{z}\}$ and $D = \{v, \overline{y}, \overline{z}\}$, we resolve them on the literal y. We drop y and $\overline{y}$, and combine the rest (removing duplicates) to get $\{v, x, \overline{z}\}$

Refutations:

- **Derivation:** A sequence of clauses $C_1, \ldots, C_s$ where every clause is either an original "axiom" (a clause from your starting set F) or a resolvent of two previous clauses in the sequence . We write $F \vdash_R C$ to say clause C can be derived from F

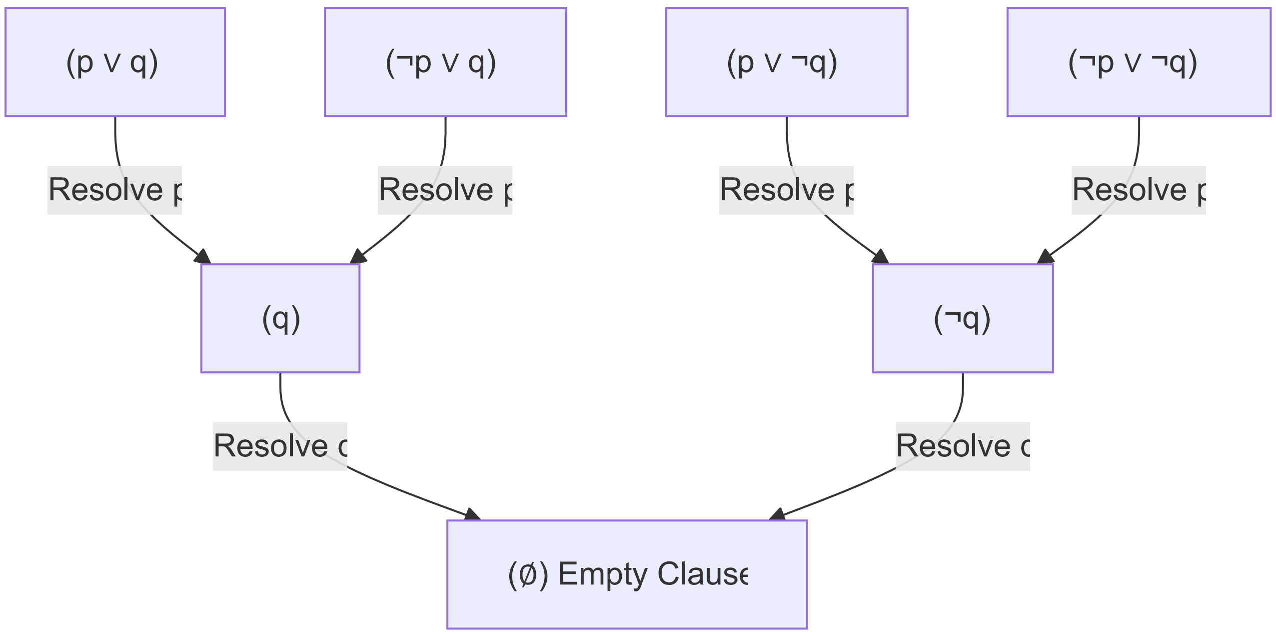
- **Refutation:** A specific type of derivation that successfully derives the empty clause ($\square$)
- **Theorem:** A clause-set F is unsatisfiable if and only if $F \vdash_R \square$
- **Soundness & Completeness:** This theorem proves two things. Resolution is **sound** (it will only refute sets that are actually unsatisfiable) and **refutationally complete** (if a set is unsatisfiable, there is guaranteed to be a resolution refutation for it)

Tree-like & DAG-like Resolution

We can visualise the derivations to form graphs

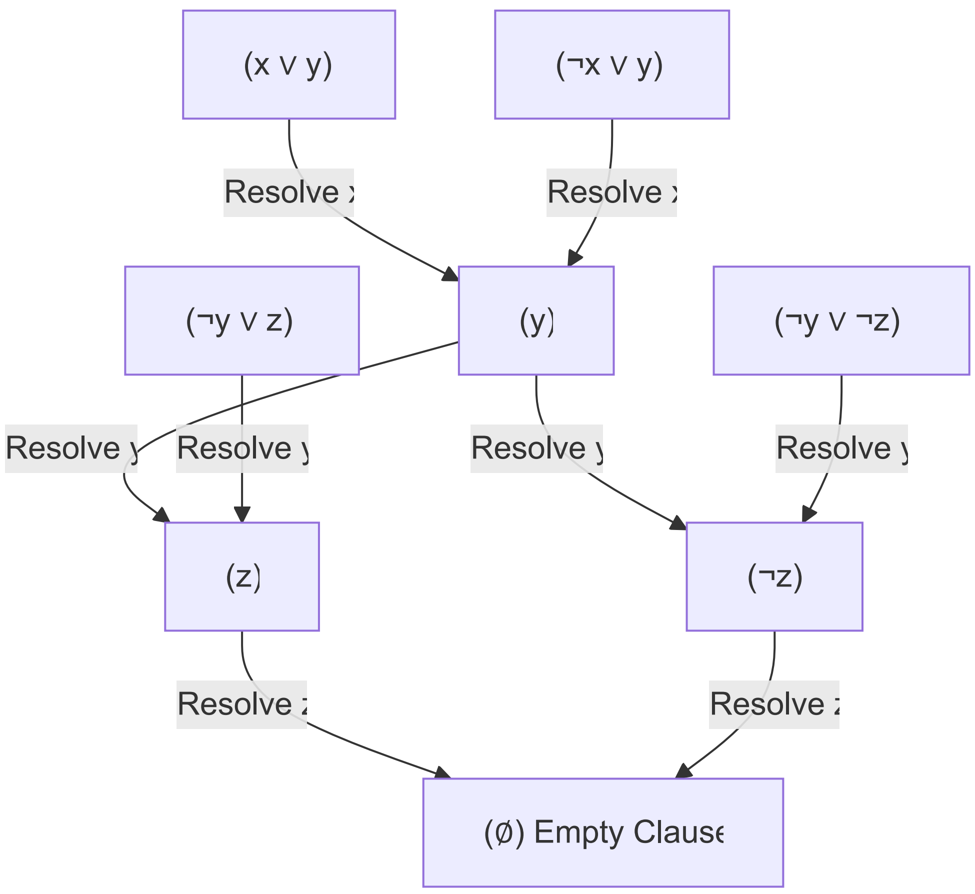
Tree-like Resolution

- **Structure:** Every derived clause is used exactly once to generate the next clause. The derivation branches out like a standard tree
- **Inefficiency:** If a specific derived clause is needed twice in different parts of the proof, the solver must recalculate it from scratch both times



DAG-like (General) Resolution

- **The Structure:** Derived clauses can be saved and reused multiple times as inputs for future resolution steps. This transforms the tree into a Directed Acyclic Graph (DAG)
- **The Efficiency:** DAG-like resolution derivations are significantly shorter and faster because they eliminate redundant calculations



- ***Note how the derived node (y) has two outgoing arrows, proving it is shared across multiple branches of the proof***

Resolution & Time Complexity

While the resolution system is strictly sound and complete, it has severe limitations regarding its computational complexity.

- **Worst-Case Scenario:** For specific hard problems, resolution requires at least an exponential number of steps to find an empty clause $\Omega(c^n)$, proved with **Haken's Theorem**

Restricted Resolution

To avoid exponential complexity, there are restricted versions of resolution, but these come at the cost of some completeness

- **Linear Resolution:** The derivation must form a strict linear sequence. Every new resolvent C_i must be created by resolving the immediately preceding resolvent C_{i-1} with another clause
- **Input Resolution:** A much stricter form where at least one of the two parent clauses in *every* resolution step must be an original input clause from the starting set F. It is exceptionally fast, but it loses completeness (it cannot resolve every possible general CNF formula)

Horn Clauses & SLD Resolution

- While Input resolution is incomplete for general CNF, it is perfectly complete for a specific, highly useful subset of logic

Horn Clause Definition:

- A propositional clause containing at most one positive literal $\left(\text{e.g. } \{\overline{p}, \overline{q}, r\}\right)$

Implication:

- Using the standard logical shortcut $\neg(A \to B) \equiv A \wedge \neg B$ in reverse, we can rewrite the Horn clause $\overline{p} \vee \overline{q} \vee r$ as $(p \wedge q) \to r$
Proof:

$$\boxed{\begin{array}{ll} (\overline{p} \vee \overline{q}) \vee r & \text{De Morgan's} \\ \neg(p \wedge q) \vee r & \text{Implication where } A \text{ is } (p \wedge q) \\ (p \wedge q) \to r & \end{array}}$$

SLD Resolution:

- This stands for *Selected, Linear, Definite* clause resolution. It is a highly efficient algorithm that applies **linear** and **input resolution** strictly to Horn clauses

---