CS264A Automated Reasoning Review Note

2020 Fall By Patricia Xiao

Notation

variable	$x, \alpha, \beta, \ldots$ (a.k.a. propositional variable / Boolean variable)
literal	$x, \neg x$
conjunction	conjunction of α and β: $\alpha \wedge \beta$
disjunction	disjunction of α and β: $\alpha \vee \beta$
negation	negation of α: $\neg\alpha$
sentence	variables are sentences; negation, conjunction, and disjunction of sentences are sentences
term	conjunction ($\wedge$) of literals
clause	disjunction ($\vee$) of literals
normal forms	**universal** format of **all** logic sentences (everyone can be transformed into CNF/DNF)
CNF	conjunctive normal form, conjunction ($\wedge$) of clauses ($\vee$)
DNF	disjunctive normal form, disjunction ($\vee$) of terms ($\wedge$)
world	ω: truth assignment of all variables (e.g. $\omega \models \alpha$ means sentence α holds at world ω)
models	$\text{Mods}(\alpha) = \{\omega : \omega \models \alpha\}$

Main Content of CS264A

- Foundations: logic, quantified Boolean logic, SAT solver, MAX-SAT etc., compiling knowledge into tractable circuit (the book chapters)
- Application: three modern roles of logic in AI
  1. logic for computation
  2. logic for leaning from knowledge / data
  3. logic for meta-learning

Syntax and Semantics of Logic

Logic syntax, "how to express", include the literal, etc. all the way to normal forms (CNF/DNF).
Logic semantic, "what does it mean", could be discussed from two perspectives:

- properties: consistency, validity etc. (of **a sentence**)
- relationships: equivalence, entailment, mutual exclusiveness etc. (of **sentences**)

Existential Quantification Useful Equations

$$\alpha \Rightarrow \beta = \neg\alpha \vee \beta$$
$$\alpha \Rightarrow \beta = \neg\beta \Rightarrow \neg\alpha$$
$$\neg(\alpha \vee \beta) = \neg\alpha \wedge \neg\beta$$
$$\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$$
$$\gamma \wedge (\alpha \vee \beta) = (\gamma \wedge \alpha) \vee (\gamma \wedge \beta)$$
$$\gamma \vee (\alpha \wedge \beta) = (\gamma \vee \alpha) \wedge (\gamma \vee \beta)$$

Models

Listing the 2^n worlds w_i involving n variables, we have a **truth table**.
If sentence α is true at world ω, $\omega \models \alpha$, we say:

- sentence α holds at world ω
- ω satisfies α
- ω entails α

otherwise $\omega \not\models \alpha$.
$\text{Mods}(\alpha)$ is called **models/meaning** of α:

$$\text{Mods}(\alpha) = \{\omega : \omega \models \alpha\}$$

$$\text{Mods}(\alpha \wedge \beta) = \text{Mods}(\alpha) \cap \text{Mods}(\beta)$$
$$\text{Mods}(\alpha \vee \beta) = \text{Mods}(\alpha) \cup \text{Mods}(\beta)$$
$$\text{Mods}(\neg\alpha) = \overline{\text{Mods}(\alpha)}$$

$\omega \models \alpha$: world ω entails/satisfies sentence α.
$\alpha \vdash \beta$: sentence α derives sentence β.

Semantic Properties

Defining $\varnothing$ as empty set and W as the set of all worlds.
Consistency: α is consistent when

$$\text{Mods}(\alpha) \neq \varnothing$$

Validity: α is valid when

$$\text{Mods}(\alpha) = \text{W}$$

α is valid iff $\neg\alpha$ is inconsistent.
α is consistent iff $\neg\alpha$ is invalid.

Semantic Relationships

Equivalence: α and β are equivalent iff

$$\text{Mods}(\alpha) = \text{Mods}(\beta)$$

Mutually Exclusive: α and β are equivalent iff

$$\text{Mods}(\alpha \wedge \beta) = \text{Mods}(\alpha) \cap \text{Mods}(\beta) = \varnothing$$

Exhaustive: α and β are exhaustive iff

$$\text{Mods}(\alpha \vee \beta) = \text{Mods}(\alpha) \cup \text{Mods}(\beta) = \text{W}$$

that is, when $\alpha \vee \beta$ is valid.
Entailment: α entails β ($\alpha \models \beta$) iff

$$\text{Mods}(\alpha) \subseteq \text{Mods}(\beta)$$

That is, satisfying α is stricter than satisfying β.
Monotonicity: the property of relations, that

- if α implies β, then $\alpha \wedge \gamma$ implies β;
- if α entails β, then $\alpha \wedge \gamma$ entails β;

it infers that adding more knowledge to the existing KB (knowledge base) never recalls anything. This is considered a limitation of traditional logic. Proof:

$$\text{Mods}(\alpha \wedge \gamma) \subseteq \text{Mods}(\alpha) \subseteq \text{Mods}(\beta)$$

Quantified Boolean Logic: Notations

Our discussion on **quantified Boolean logic** centers around *conditioning* and *restriction*. ($|$, $\exists$, $\forall$) With a *propositional sentence* Δ and a *variable* P:

- condition Δ on P: $\Delta|P$
  i.e. replacing all occurrences of P by true.
- condition Δ on $\neg P$: $\Delta|\neg P$
  i.e. replacing all occurrences of P by false.

Boolean's/Shanoon's Expansion:

$$\Delta = \Big(P \wedge (\Delta|P)\Big) \vee \Big(\neg P \wedge (\Delta|\neg P)\Big)$$

it enables recursively solving logic, e.g. DPLL.

Existential & Universal Qualification

Existential Qualification:

$$\exists P \Delta = \Delta|P \;\vee\; \Delta|\neg P$$

Universal Qualification:

$$\forall P \Delta = \Delta|P \;\wedge\; \Delta|\neg P$$

Duality:

$$\exists P \Delta = \neg(\forall P \neg\Delta)$$
$$\forall P \Delta = \neg(\exists P \neg\Delta)$$

The quantified Boolean logic is different from first-order logic, for it does not express everything as *objects* and *relations* among objects.

Forgetting

The right-hand-side of the above-mentioned equation:

$$\exists P \Delta = \Delta|P \;\vee\; \Delta|\neg P$$

doesn't include P.
Here we have an example: $\Delta = \{A \Rightarrow B, B \Rightarrow C\}$, then:

$$\Delta = (\neg A \vee B) \wedge (\neg B \vee C)$$
$$\Delta|B = C$$
$$\Delta|\neg B = \neg A$$
$$\therefore \exists E \Delta = \Delta|B \vee \Delta|\neg E = \neg A \vee C$$

- $\Delta \models \exists P \Delta$
- If α is a sentence that does not mention P then $\Delta \models \alpha \iff \exists P \Delta \models P$

We can safely remove P from Δ when considering existential qualification. It is called:

- **forgetting** P from Δ
- **projecting** P on all units / variables but P

Resolution / Inference Rule

Modus Ponens (MP):

$$\frac{\alpha, \alpha \Rightarrow \beta}{\beta}$$

Resolution:

$$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

equivalent to:

$$\frac{\neg\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$$

Above the line are the known conditions, below the line is what could be inferred from them.
In the resolution example, $\alpha \vee \gamma$ is called a "**resolvent**". We can say it either way:

- resolve $\alpha \vee \beta$ with $\neg\beta \vee \gamma$
- resolve over β
- do β-resolution

MP is a special case of resolution where $\alpha = \text{true}$.
It is always written as:

$$\Delta = \{\alpha \vee \beta, \neg\beta \vee \gamma\} \vdash_R \alpha \vee \gamma$$

Applications of resolution rules:

1. existential quantification
2. simplifying KB (Δ)
3. deduction (strategies of resolution, directed resolution)

Completeness of Resolution / Inference Rule

We say rule R is complete, iff $\forall \alpha$, if $\Delta \models \alpha$ then $\Delta \vdash_R \alpha$.
In other words, R is complete when it could "discover everything from Δ".
Resolution / inference rule is **NOT complete**. A counter example is: $\Delta = \{A, B\}, \alpha = A \vee B$.
However, when applied to CNF, resolution is **refutation complete**. Which means that it is sufficient to discover **any inconsistency**.

Clausal Form of CNF

CNF, the Conjunctive Normal Form, is a conjunction of clauses.

$$\Delta = C_1 \wedge C_2 \wedge \ldots$$

written in clausal form as:

$$\Delta = \{C_1, C_2 \ldots\}$$

where each clause C_i is a disjuntion of literals:

$$C_i = l_{i1} \vee l_{i2} \vee l_{i3} \vee \ldots$$

written in clausal form as:

$$C_i = \{l_{i1}, l_{i2}, l_{i3}\}$$

Resolution in the clausal form is formalized as:

- Given clauses C_i and C_j where literal $P \in C_i$ and literal $\neg P \in C_j$
- The resolvent is $(C_i \backslash \{P\}) \cup (C_j \backslash \{\neg P\})$ (Notation: removing set $\{P\}$ from set C_i is written as $C_i \backslash \{P\}$)

If the clausal form of a CNF contains an **empty clause** ($\exists i, C_i = \varnothing = \{\}$), then it makes the CNF inconsistent / unsatisfiable.

Existential Quantification via Resolution

1. Turning KB Δ into CNF.
2. To existentially Quantify B, do all B-resolutions
3. Drop all clauses containing B

Unit Resolution

Unit resolution is a special case of resolution, where $\min(|C_i|, |C_j|) = 1$ where $|C_i|$ denotes the size of set C_i. **Unit resolution** corresponds to **modus ponens** (MP). It is **NOT refutation complete**. But it has benefits in efficiency: could be applied in *linear time*.

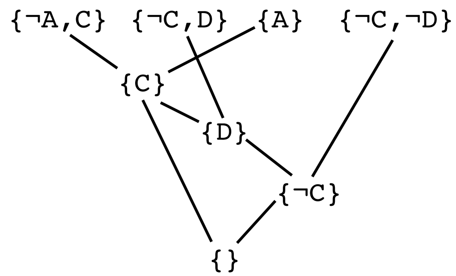
Refutation Theorem

$\Delta \models \alpha$ iff $\Delta \wedge \neg\alpha$ is inconsistent. (useful in proof)

- resolution finds contradiction on $\Delta \wedge \neg\alpha$: $\Delta \models \alpha$
- resolution does not find any contradiction on $\Delta \wedge \neg\alpha$: $\Delta \not\models \alpha$

Resolution Strategies: Linear Resolution

All the clauses that are originally included in CNF Δ are **root** clauses.
Linear resolution resolved C_i and C_j only if one of them is **root** or an **ancestor** of the other clause.
An example: $\Delta = \{\neg A, C\}, \{\neg C, D\}, \{A\}, \{\neg C, \neg D\}$.



Resolution Strategies: Directed Resolution

Directed resolution is based on bucket elimination, and requires pre-defining an order to process the variables. The steps are as follows:

1. With n variables, we have n buckets, each corresponds to a variable, listed from the top to the bottom in **order**.
2. Fill the clauses into the buckets. Scanning top-side-down, putting each clause into the first bucket whose corresponding variable is included in the clause.
3. Process the buckets top-side-down, whenever we have a P-resolvent C_{ij}, put it into the first **following** bucket whose corresponding variable is included in C_{ij}.

An example: $\Delta = \{\neg A, C\}, \{\neg C, D\}, \{A\}, \{\neg C, \neg D\}$, with variable order A, D, C, initialized as:

A:	$\{\neg A, C\}, \{A\}$	
D:	$\{\neg C, D\}, \{\neg C, \neg D\}$	
C:		

After processing finds $\{\}$ ($\{C\}$ is the A-resolvent, $\{\neg C\}$ is the B-resolvent, $\{\}$ is a C-resolvent):

A:	$\{\neg A, C\}, \{A\}$	
D:	$\{\neg C, D\}, \{\neg C, \neg D\}$	
C:		$\{C\}, \{\neg C\}, \{\}$

Directed Resolution: Forgetting

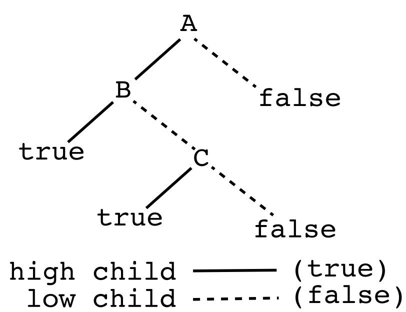
Directed resolution can be applied to forgetting / projecting.
When we do existential quantification on variables $P_1, P_2, \ldots P_m$, we:

1. put them in the first m places of the variable order
2. after processing the first m ($P_1, P_2, \ldots P_m$) buckets, remove the first m buckets
3. keep the clauses (*original clause* or *resolvent*) in the remaining buckets

then it is done.

Utility of Using Graphs

Primal Graph: Each node represents a variable P. Given CNF Δ, if there's at least a clause $\exists C \in \Delta$ such that $l_i, l_j \in C$, then the corresponding nodes P_i and P_j are connected by an edge.
The *tree width* (w) (a property of graph) can be used to estimate time & space complexity. e.g. complexity of directed resolution. e.g. Space complexity of n variables is $\mathcal{O}(n \exp(w))$.
For more, see textbook — **min-fill heuristic**.
Decision Tree: Can be used for model-counting. e.g. $\Delta = A \wedge (B \vee C)$, where $n = 3$, then:



for counting purpose we assign value $2^n = 2^3 = 8$ to the *root* (A in this case), and $2^{n-1} = 4$ to the next level (its direct children), etc. and finally we sum up the values assigned to all true values. Here we have: $2 + 1 = 3$. $|\text{Mods}(\Delta)| = 3$. Constructing via:

- If inconsistent then put false here.
- Directed resolution could be used to build a decision tree. P-bucket: P nodes.

SAT Solvers

The SAT-solvers we learn in this course are:

- requiring modest space
- foundations of many other things

Along the line there are: SAT I, SAT II, DPLL, and other modern SAT solvers.
They can be viewed as optimized searcher on all the worlds ω_i looking for a world satisfying Δ.

SAT I

SAT-I (Δ, n, d):
  If $d = n$:
    If $\Delta = \{\}$, return $\{\}$
    If $\Delta = \{\{\}\}$, return FAIL
  If $\mathbf{L} = \text{SAT-I}(\Delta | P_{d+1}, n, d+1) \neq$ FAIL:
    return $\mathbf{L} \cup \{P_{d+1}\}$
  If $\mathbf{L} = \text{SAT-I}(\Delta | \neg P_{d+1}, n, d+1) \neq$ FAIL:
    return $\mathbf{L} \cup \{\neg P_{d+1}\}$
  return FAIL

Δ: a CNF, unsat when $\{\} \in \Delta$, satisfied when $\Delta = \{\}$
n: number of variables, $P_1, P_2 \ldots P_n$
d: the depth of the current node

- root node has depth 0, corresponds to P_1
- nodes at depth $n-1$ try P_n
- leave nodes are at depth n, each represents a world ω_i

Typical DFS (depth-first search) algorithm.

- DFS, thus $\mathcal{O}(n)$ space requirement (moderate)
- No pruning, thus $\mathcal{O}(2^n)$ time complexity

SAT II

SAT-II (Δ, n, d):
  If $\Delta = \{\}$, return $\{\}$
  If $\Delta = \{\{\}\}$, return FAIL
  If $\mathbf{L} = \text{SAT-II}(\Delta | P_{d+1}, n, d+1) \neq$ FAIL:
    return $\mathbf{L} \cup \{P_{d+1}\}$
  If $\mathbf{L} = \text{SAT-II}(\Delta | \neg P_{d+1}, n, d+1) \neq$ FAIL:
    return $\mathbf{L} \cup \{\neg P_{d+1}\}$
  return FAIL

Mostly SAT I, plus early-stop.

Termination Tree

Termination tree is a sub-tree of the complete search space (which is a depth-n complete binary tree), including only the nodes visited while running the algorithm.
When drawing the termination tree of SAT I and SAT II, we put a cross (X) on the failed nodes, with $\{\{\}\}$ label next to it. Keep going until we find an answer — where $\Delta = \{\}$.

Unit-Resolution

UNIT-RESOLUTION (Δ):
  $\mathbf{I}$ = unit clauses in Δ
  If $I = \{\}$: return $(\mathbf{I}, \Delta)$
  $\Gamma = \Delta|\mathbf{I}$
  If $\Gamma = \Delta$: return $(\mathbf{I}, \Gamma)$
  return UNIT-RESOLUTION(Γ)

Used in DPLL, at each node.

DPLL

DPLL (Δ):
  $(\mathbf{I}, \Gamma)$ = UNIT-RESOLUTION(Δ)
  If $\Gamma = \{\}$, return $\mathbf{I}$
  If $\{\} \in \Gamma$, return FAIL
  choose a literal l in Γ
  If $\mathbf{L}$ = DPLL$(\Gamma \cup \{\{l\}\}) \neq$ FAIL:
    return $\mathbf{L} \cup \mathbf{I}$
  If $\mathbf{L}$ = DPLL$(\Gamma \cup \{\{\neg l\}\}) \neq$ FAIL:
    return $\mathbf{L} \cup \mathbf{I}$
  return FAIL

Mostly SAT II, plus unit-resolution.
UNIT-RESOLUTION is used at each node looking for entailed value, to save searching steps.
If there's any implication made by UNIT-RESOLUTION, we write down the values next to the node where the implication is made. (e.g. $A = t, B = f, \dots$)
This is **NOT** a standard DFS. UNIT-RESOLUTION component makes the searching flexible.

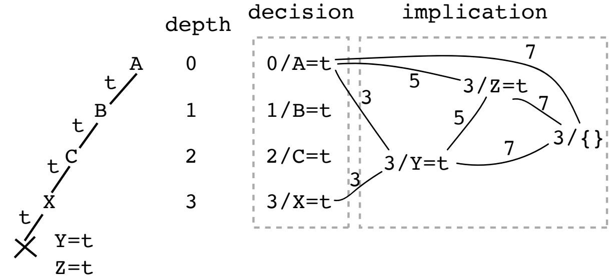
Non-chronological Backtracking

Chronological backtracking is when we find a contradiction/FAIL in searching, backtrack to parent.
Non-chronological backtracking is an optimization that we jump to earlier nodes. a.k.a. **conflict-directed backtracking**.

Implication Graphs

Implication Graph is used to find more clauses to add to the KB, so as to empower the algorithm.
An example of an implication graph upon the first conflict found when running DPLL+ for Δ:

```
1.{A,B}
2.{B,C}
3.{¬A,¬X,Y}
4.{¬A,X,Y}      } Δ
5.{¬A,¬Y,Z}
6.{¬A,X,¬Z}
7.{¬A,¬Y,¬Z}
```

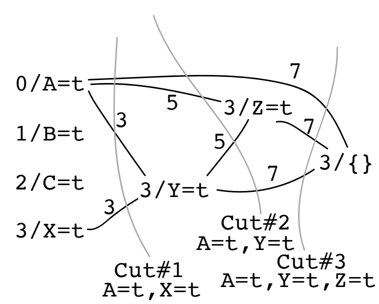


There, the decisions and implications assignments of variables are labeled by the **depth** at which the value is determined.
The edges are labeled by **the ID of the corresponding rule** in Δ, which is used to generate a unit clause (make an implication).

Implication Graphs: Cuts

Cuts in an Implication Graph can be used to identify the conflict sets. Still following the previous example:

```
    1.{A,B}
    2.{B,C}
    3.{¬A,¬X,Y}
Δ= 4.{¬A,X,Y}
    5.{¬A,¬Y,Z}
    6.{¬A,X,¬Z}
    7.{¬A,¬Y,¬Z}
```



Here Cut#1 results in learned clause $\{\neg A, \neg X\}$, Cut#2 learned clause $\{\neg A, \neg Y\}$, Cut#3 learned clause $\{\neg A, \neg Y, \neg Z\}$.

Asserting Clause & Assertion Level

Asserting Clause: Including **only one** variable at the **last** (highest) decision level. (The last decision-level means *the level where the last decision/implication is made.*)
Assertion Level (AL): The **second-highest** level in the clause. (Note: 3 is higher than 0.)
An example (following the previous example, on the learned clauses):

Clause	Decision-Levels	Asserting?	AL
$\{\neg A, \neg X\}$	$\{0, 3\}$	Yes	0
$\{\neg A, \neg Y\}$	$\{0, 3\}$	Yes	0
$\{\neg A, \neg Y, \neg Z\}$	$\{0, 3, 3\}$	No	0

DPLL+

DPLL+ (Δ):
  $D \leftarrow ()$
  $\Gamma \leftarrow \{\}$
  While true Do:
    $(\mathbf{I}, \mathbf{L})$ = UNIT-RESOLUTION$(\Delta \wedge \Gamma \wedge D)$
    If $\{\} \in \mathbf{L}$:
      If $D = ()$: return false
      Else (backtrack to assertion level):
        $\alpha \leftarrow$ asserting clause
        $m \leftarrow$ AL(α)
        $D \leftarrow$ first $m+1$ decisions in D
        $\Gamma \leftarrow \Gamma \cup \{\alpha\}$
    Else:
      find ℓ where $\{\ell\} \notin \mathbf{I}$ and $\{\neg\ell\} \notin \mathbf{I}$
      If an ℓ is found: $D \leftarrow D; \ell$
      Else: return true

true if the CNF Δ is satisfiable, otherwise false.
Γ is the learned clauses, D is the decision sequence.
Idea: Backtrack to the assertion level, add the conflict-driven clause to the knowledge base, apply unit resolution.
Selecting α: find **the first UIP**.

UIP (Unique Implication Path)

The variable that set on every path from the last decision level to the contradiction.
The **first UIP** is the closest to the contradiction.
For example, in the previous example, the **last** UIP is $3/X = t$, while the **first UIP** is $3/Y = t$.

Exhaustive DPLL

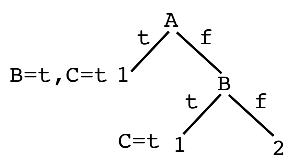
Exhaustive DPLL: DPLL that doesn't stop when finding a solution. Keeps going until explored the whole search space.
It is useful for model-counting.
However, recall that, DPLL is based on that Δ is satisfiable iff $\Delta|P$ is satisfiable or $\Delta|\neg P$ is satisfiable, which infers that we do not have to test both branches to determine satisfiability.
Therefore, we have smarter algorithm for model-counting using DPLL: CDPLL.

CDPLL

CDPLL (Γ, n):
    If $\Gamma = \{\}$: return 2^n
    If $\{\} \in \Gamma$: return 0
    choose a literal l in Γ
    $(\mathbf{I}^+, \Gamma^+) = \text{Unit-Resolution}(\Gamma \cup \{\{l\}\})$
    $(\mathbf{I}^-, \Gamma^-) = \text{Unit-Resolution}(\Gamma \cup \{\{\neg l\}\})$
    return CDPLL$(\Gamma^+, n - |\mathbf{I}^+|)+$
        CDPLL$(\Gamma^-, n - |\mathbf{I}^-|)$

n is the number of variables, it is very essential when counting the models.
An example of the termination tree:

Δ= 1.{¬A,B}
2.{¬B,C}



Certifying UNSAT: Method #1

When a query is satisfiable, we have an answer to certify.
However, when it is unsatisfiable, we also want to validate this conclusion.
One method is via verifying UNSAT directly (example Δ from implication graphs), example:

level	assignment	reason
-1		
0	A	
1	B	
2	C	
3	X	
	Y	$\neg A \vee \neg X \vee Y$
	Z	$\neg A \vee \neg Y \vee Z$

And then learned clause $\neg A \vee \neg Y$ is applied. Learned clause is asserting, $AL = 0$ so we add $\neg Y$ to level 0, right after A, then keep going from $\neg Y$.

Certifying UNSAT: Method #2

Verifying the Γ generated from the SAT solver after running on Δ is a correct one.

- Will $\Delta \cup \Gamma$ produce any inconsistency?
  - Can use Unit-Resolution to check.
- CNF $\Gamma = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ comes from Δ?
  - $\Delta \wedge \neg\alpha_i$ is inconsistent for all clauses α_i.
  - Can use Unit-Resolution to check.

Why **Unit-Resolution** is enough: $\{\alpha_i\}_{i=1}^n$ are generated from cuts in an **implication graph**. The implication graph is built upon conflicts found by **Unit-Resolution**. Therefore, the conflicts can be detected by **Unit-Resolution**.

UNSAT Cores

For CNF $\Delta = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, an UNSAT core is any subsets consisting of some $\alpha_i \in \Delta$ that is inconsistent together. There exists at least one UNSAT core iff Δ is UNSAT.
A **minimal UNSAT core** is an UNSAT core of Δ that, if we remove a clause from this UNSAT core, the remaining clauses become consistent together.

More on SAT

- Can SAT solver be faster than linear time?
  - 2-literal watching (in textbook)
- The "phase-selection" / variable ordering problem (including the decision on trying P or $\neg P$ first)?
  - An efficient and simple way: "try to try the phase you've tried before". — This is because of the way modern SAT solvers work (cache, etc.).

SAT using Local Search

The general idea is to start from a random guess of the world ω, if UNSAT, move to another world by flipping one variable in ω (P to $\neg P$, or $\neg P$ to P).

- Random CNF: n variables, m clauses. When m/n gets extremely small or large, it is easier to randomly generate a world (thinking of $\binom{n}{m}$: when $m/n \to 0$ it is almost always SAT, $m/n \to \infty$ will make it almost always UNSAT). In practice, the split point is $m/n \approx 4.24$.

  Two ideas to generate random clauses:
  - 1^{st} idea: variable-length clauses
  - 2^{nd} idea: fixed-length clauses (k-SAT, e.g. 3-SAT)
- Strategy of Taking a Move:
  - Use a cost function to determine the quality of a world.
    - Simplest cost function: the number of unsatisfied clauses.
    - A lot of variations.
    - Intend to go to lower-cost direction. ("hill-climbing")
  - Termination Criteria: No neighbor is better (smaller cost) than the current world. (Local, not global optima yet.)
  - Avoid local optima: Randomly restart multiple times.
- Algorithms:
  - GSAT: hill-climbing + side-move (moving to neighbors whose cost is equal to ω)
  - WALKSAT: iterative repair
    - randomly pick an unsatisfied clause
    - pick a variable within that clause to flip, such that it will result in the fewest previously satisfied clauses becoming unsatisfied, then flip it
  - Combination of logic and randomness:
    - randomly select a neighbor, if better than current node then move, otherwise move at a probability (determined by how much worse it is)

Max-SAT

Max-SAT is an optimization version of SAT. In other words, Max-SAT is an optimizer SAT solver.
Goal: finding the assignment of variables that **maximizes the number of satisfied clauses** in a CNF Δ. (We can easily come up with other variations, such as Min-SAT etc.)

- We assign a weight to each clause as the score of satisfying it / cost of violating it.
- We maximize the score. (This is only one way of solving the problem, we can also do it by minimizing the cost. — **Note**: score is different from cost.)

Solving Max-SAT problems generally goes into three directions:

- Local Search
- Systematic Search (branch and bound etc.)
- Max-SAT **Resolution**

Max-SAT Example

We have images I_1, I_2, I_3, I_4, with weights (importance) $5, 4, 3, 6$ respectively, knowing: (1) I_1, I_4 can't be taken together (2) I_2, I_4 can't be taken together (3) I_1, I_2 if overlap then discount by 2 (4) I_1, I_3 if overlap then discount by 1 (5) I_2, I_3 if overlap then discount by 1.
Then we have the knowledge base Δ as:

$$
\begin{aligned}
\Delta :& (I_1, 5) \\
& (I_2, 4) \\
& (I_3, 3) \\
& (I_4, 6) \\
& (\neg I_1 \vee \neg I_2, 2) \\
& (\neg I_1 \vee \neg I_3, 1) \\
& (\neg I_2 \vee \neg I_3, 1) \\
& (\neg I_1 \vee \neg I_4, \infty) \\
& (\neg I_2 \vee \neg I_4, \infty)
\end{aligned}
$$

To simply the example we look at I_1 and I_2 only:

I_1	I_2	**score**	**cost**
✓	✓	9	0
✓	✗	5	4
✗	✓	4	5
✗	✗	0	9

In practice we list the truth table of I_1 through I_4 ($2^4 = 16$ worlds).

Max-SAT Resolution

In Max-SAT, in order to **keep the same cost/score** before and after resolution, we:

- Abandon the resolved clauses;
- Add compensation clauses.

Considering the following two clauses to resolve:

$$
\begin{aligned}
x \vee & \overbrace{\ell_1 \vee \ell_2 \vee \cdots \vee \ell_m}^{c_1} \\
\neg x \vee & \underbrace{o_1 \vee o_2 \vee \cdots \vee o_n}_{c_2}
\end{aligned}
$$

The results are the resolvent $c_1 \vee c_2$, and the compensation clauses:

$$
\begin{aligned}
& c_1 \vee c_2 \\
& x \vee c_1 \vee \neg o_1 \\
& x \vee c_1 \vee o_1 \vee \neg o_2 \\
& \vdots \\
& x \vee c_1 \vee o_1 \vee o_2 \vee \cdots \vee \neg o_n \\
& \neg x \vee c_2 \vee \neg \ell_1 \\
& \neg x \vee c_2 \vee \ell_1 \vee \neg \ell_2 \\
& \vdots \\
& \neg x \vee c_2 \vee \ell_1 \vee \ell_2 \vee \cdots \vee \neg \ell_m
\end{aligned}
$$

Directed Max-SAT Resolution

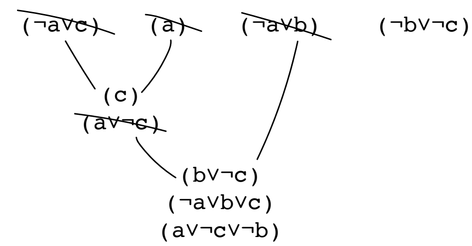
1. Pick an order of the variables, say, $x_1, x_2, \ldots, x_n$
2. For each x_i, exhaust all possible Max-SAT resolutions, the move on to x_{i+1}.
When resolving x_i, using only the clauses that does not mention any x_j, $\forall j < i$.

Resolve two clauses on x_i only when there isn't a $x_j \neq x_i$ that x_j and $\neg x_j$ belongs to the two clauses each. (Formally: do not contain complementary literals on $x_j \neq x_i$.)
Ignore the resolvent and compensation clauses when they've appeared before, as original clauses, resolvent clauses, or compensation clauses.

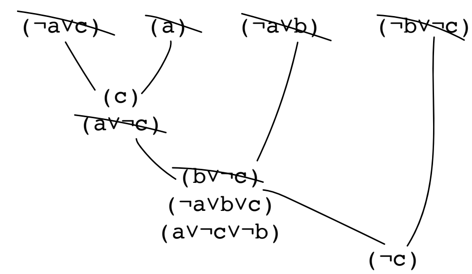
In the end, there remains k false (conflicts), and Γ (guaranteed to be satisfiable). k is the minimum cost, each world satisfying Γ achieves this cost.

Directed Max-SAT Resolution: Example

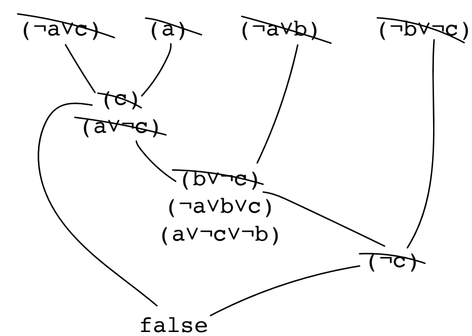
$\Delta = (\neg a \vee c) \wedge (a) \wedge (\neg a \vee b) \wedge (\neg b \vee \neg c)$
Variable order: a, b, c.
First resolve on a:



Then resolve on b:



Finally:



The final output is:

$$\text{false}, \big[(\neg a \vee b \vee c), (a \vee \neg b \vee \neg c)\big]$$

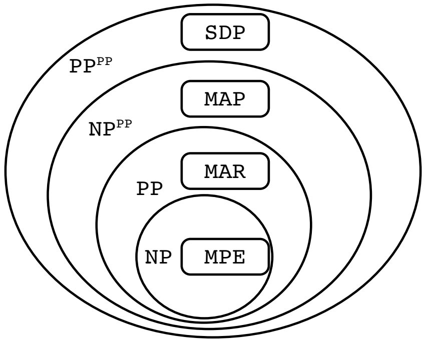
Where $\Gamma = (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$, and $k = 1$, indicating that there must be at least one clause in Δ that is not satisfiable.

Beyond NP

Some problems, even those harder than NP problems can be reduced to logical reasoning.

Complexity Classes

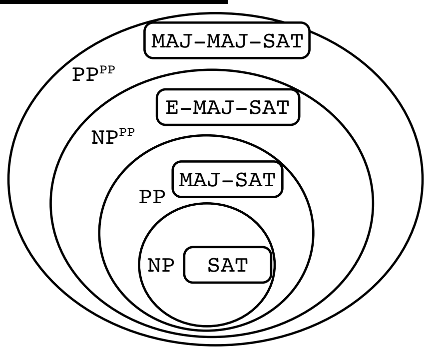
Shown in the figure are some example of the complete problems.



abbr.	meaning
SDP	Same-Decision Probability
MAP	Maximum A Posterior hypothesis
MAR	MArginal Probabilities
MPE	Most Probable Explanation

A **complete** problem means that it is one of the hardest problems of its complexity class. e.g. NP-complete: among all NP problem, there is not any problem harder than it.
Our goal: Reduce **complete problems** to **prototypical problems** (Boolean formula), then transform them into **tractable Boolean circuits**.

Prototypical Problems



abbr.	meaning
SAT	satisfiability
MAJ-SAT	majority-instantiation satisfiability
E-MAJ-SAT	with (X, Y)-split of the variables, exists an X-instantiation that satisfies the majority of Y-instantiation.
MAJ-MAJ-SAT	with (X, Y)-split of the variables, the majority of X-instantiation satisfies the majority of Y-instantiation.

Again, those are all **complete** problems.

Bayesian Network to MAJ-SAT Problem

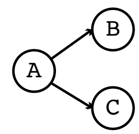
A MAJ-SAT problem consists of:

- #SAT Problem (model counting)
- WMC Problem (weighted model counting)

Consider WMC (weighted model counting) problem, e.g. three variables A, B, C, weight of world $A = t, B = t, C = f$ should be:

$$w(A, B, \neg C) = w(A)w(B)w(\neg C)$$

Typically, in a Bayesian network, where both B and C depend on A:



And we therefore have:

$$\text{Prob}(A = t, B = t, C = t) = \theta_A \theta_{B|A} \theta_{C|A}$$

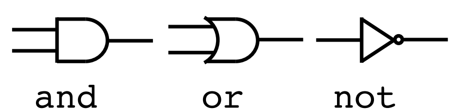
where $\Theta = \{\theta_A, \theta_{\neg A}\} \cup \{\theta_{B|A}, \theta_{\neg B|A}, \theta_{B|\neg A}, \theta_{\neg B|\neg A}\} \cup \{\theta_{C|A}, \theta_{\neg C|A}, \theta_{C|\neg A}, \theta_{\neg C|\neg A}\}$ are the parameters within the Bayesian network at nodes A, B, C respectively, indicating the probabilities.
Though slightly more complex than treating each variable equally, by working on Θ we can safely reduce any Bayesian network to a MAJ-SAT problem.

NNF (Negation Normal Form)

NNF is the form of **Tractable Boolean Circuit** we are specifically interested in.
In an **NNF**, leave nodes are **true**, **false**, **P** or **¬P**; internal nodes are either **and** or **or**, indicating an operation on all its children.

Tractable Boolean Circuits

We draw an NNF as if it is made up of logic. From a circuit perspective, it is made up of gates.



NNF Properties

Property	On Whom	Satisfied NNF
Decomposability	and	DNNF
Determinism	or	d-NNF
Smoothness	or	s-NNF
Flatness	whole NNF	f-NNF
Decision	or	BDD (FBDD)
Ordering	each node	OBDD

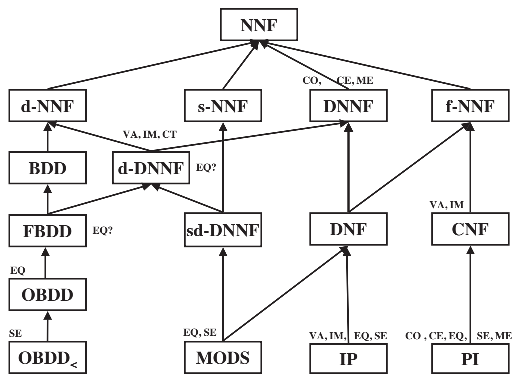
Decomposability: for any **and** node, any pair of its children must be on **disjoint** variable sets. (e.g. one child $A \vee B$, the other $C \vee D$)
Determinism: for any **or** node, any pair of its children must be **mutually exclusive**. (e.g. one child $A \wedge B$, the other $\neg A \wedge B$)
Smoothness: for any **or** node, any pair of its children must be on **the same** variable set. (e.g. one child $A \wedge B$, the other $\neg A \wedge \neg B$)
Flatness: the height of each sentence (sentence: from root — select one child when seeing **or** ; all children when seeing **and** — all the way to the leaves / literals) is at most 2 (depth $0, 1, 2$ only). (e.g. CNF, DNF)
Decision: a **decision node** N can be **true**, **false**, or being an **or**-node $(X \wedge \alpha) \vee (\neg X \wedge \beta)$ (X: variable, α, β: decision nodes, decided on $\text{dVar}(N) = X$).
Ordering: make no sense if not decision (FBDD); variables are decided following a fixed order.

NNF Queries

Abbr.	Spelled Name	description
CO	consistency check	$SAT(\Delta)$
VA	validity check	$\neg SAT(\neg\Delta)$
SE	sentence entailment check	$\Delta_1 \models \Delta_2$
CE	clausal entailment check	$\Delta \models$ clause α
IM	implicant testing	$\Delta \models$ term ℓ
EQ	equivalence testing	$\Delta_1 = \Delta_2$
CT	model counting	$\lvert\text{Mods}(\Delta)\rvert$
ME	model enumeration	$\omega \in \text{Mods}(\Delta)$

Our goal is to get the above-listed **queries** done on our circuit within **polytime**.
Besides, we also seek for polytime **transformations**: Projection (existential quantification), Conditioning, Conjoin, Disjoin, Negate, etc.

The Capability of NNFs on Queries



	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
d-NNF	○	○	○	○	○	○	○	○
s-NNF	○	○	○	○	○	○	○	○
f-NNF	○	○	○	○	○	○	○	○
DNNF	✓	○	✓	○	○	○	○	✓
d-DNNF	✓	✓	✓	✓	?	○	✓	✓
FBDD	✓	✓	✓	✓	?	○	✓	✓
OBDD	✓	✓	✓	✓	✓	○	✓	✓
$\text{OBDD}_<$	✓	✓	✓	✓	✓	✓	✓	✓
BDD	○	○	○	○	○	○	○	○
sd-DNNF	✓	✓	✓	✓	?	○	✓	✓
DNF	✓	○	✓	○	○	○	○	✓
CNF	○	✓	○	✓	○	○	○	○
PI	✓	✓	✓	✓	✓	✓	○	✓
IP	✓	✓	✓	✓	✓	✓	○	✓
MODS	✓	✓	✓	✓	✓	✓	✓	✓

✓: can be done in polytime
○: cannot be done in polytime unless $P = NP$.
✗: cannot be done in polytime **even if** $P = NP$
?: remain unclear (no proof yet)

NNF Transformations

notation	transformation	description
CD	conditioning	$\Delta \| P$
FO	forgetting	$\exists P, Q, \ldots \Delta$
SFO	singleton forgetting	$\exists P.\Delta$
$\wedge C$	conjunction	$\Delta_1 \wedge \Delta_2$
$\wedge BC$	bounded conjunction	$\Delta_1 \wedge \Delta_2$
$\vee C$	disjunction	$\Delta_1 \vee \Delta_2$
$\vee BC$	bounded disjunction	$\Delta_1 \vee \Delta_2$
$\neg C$	negation	$\neg\Delta$

Our goal is to **transform** in **polytime** while still keep the properties (e.g. DNNF still be DNNF).
Bounded conjunction / disjunction: KB Δ is bounded on conjunction / disjunction operation. That is, taking any two formula from Δ, their conjunction / disjunction also belong to Δ.

The Capability of NNFs on Transformations

	CD	FO	SFO	$\wedge C$	$\wedge BC$	$\vee C$	$\vee BC$	$\neg C$
NNF	✓	○	✓	✓	✓	✓	✓	✓
d-NNF	✓	○	✓	✓	✓	✓	✓	✓
s-NNF	✓	○	✓	✓	✓	✓	✓	✓
f-NNF	✓	○	✓	✗	✗	✗	✗	✓
DNNF	✓	✓	✓	○	○	✓	✓	○
d-DNNF	✓	○	○	○	○	○	○	?
FBDD	✓	✗	○	✗	○	✗	○	✓
OBDD	✓	✗	✓	✗	○	✗	○	✓
$\text{OBDD}_<$	✓	✗	✓	✗	✓	✗	✓	✓
BDD	✓	○	✓	✓	✓	✓	✓	✓
sd-DNNF	✓	✓	✓	✓	?	○	✓	✓
DNF	✓	✓	✓	✗	✓	✓	✓	✗
CNF	✓	○	✓	✓	✓	✗	✓	✗
PI	✓	✓	✓	✗	✗	✗	✓	✗
IP	✓	✗	✗	✗	✓	✗	✗	✗
MODS	✓	✓	✓	✗	✓	✗	✗	✗

✓: can be done in polytime
○: cannot be done in polytime unless $P = NP$.
✗: cannot be done in polytime **even if** $P = NP$
?: remain unclear (no proof yet)

Variations of NNF

Acronym	Description
NNF	Negation Normal Form
d-NNF	Deterministic Negation Normal Form
s-NNF	Smooth Negation Normal Form
f-NNF	Flat Negation Normal Form
DNNF	Decomposable Negation Normal Form
d-DNNF	Deterministic Decomposable Negation Normal Form
sd-DNNF	Smooth Deterministic Decomposable Negation Normal Form
BDD	Binary Decision Diagram
FBDD	Free Binary Decision Diagram
OBDD	Ordered Binary Decision Diagram
$\text{OBDD}_<$	Ordered Binary Decision Diagram (using order $<$)
DNF	Disjunctive Normal Form
CNF	Conjunctive Normal Form
PI	Prime Implicates
IP	Prime Implicants
MODS	Models

FBDD: the intersection of DNNF and BDD.
$\textbf{OBDD}_<$: if N and M are **or**-nodes, and if N is an ancestor of M, then $\text{dVar}(N) < dVar(M)$.
OBDD: the union of all $\text{OBDD}_<$ languages. In **this course** we always use **OBDD** to refer to $\textbf{OBDD}_<$.
MODS is the subset of DNF where every sentence satisfies determinism and smoothness.
PI: subset of CNF, each clause entailed by Δ is subsumed by an existing clause; and no clause in the sentence Δ is subsumed by another.
IP: dual of PI, subset of DNF, each term entailing Δ subsumes some existing term; and no term in the sentence Δ is subsumed by another.

DNNF

CO: check consistency in polytime, because:

$$\begin{cases} \text{SAT}(A \vee B) = \text{SAT}(A) \vee \text{SAT}(B) \\ \text{SAT}(A \wedge B) = \text{SAT}(A) \wedge \text{SAT}(B) \quad // \text{ DNNF only} \\ \text{SAT}(X) = \textbf{true} \\ \text{SAT}(\neg X) = \textbf{true} \\ \text{SAT}(\textbf{true}) = \textbf{true} \\ \text{SAT}(\textbf{false}) = \textbf{false} \end{cases}$$

CE: clausal entailment, check $\Delta \models \alpha$ ($\alpha = \ell_1 \vee \ell_2 \ldots \ell_n$) by checking the consistency of:

$$\Delta \wedge \neg\ell_1 \wedge \neg\ell_2 \wedge \cdots \wedge \neg\ell_n$$

constructing a new NNF of it by making NNF of Δ and the NNF of $\neg\alpha$ direct child of root-node **and**.
When a variable P appear in both α and Δ, the new NNF is not DNNF. We fix this by conditioning Δ's NNF on P or $\neg P$, depending on either P or $\neg P$ appears in α. ($\Delta \to (\neg P \wedge \Delta|\neg P) \vee (P \wedge \Delta|P)$) If P in α, then $\neg P$ in $\neg\alpha$, we do $\Delta|\neg P$.
Interestingly, this transformation might turn a non-DNNF NNF (troubled by A) into DNNF.
CD: conditioning, $\Delta|A$ is to replace all A in NNF with **true** and $\neg A$ with **false**. For $\Delta|\neg A$, vice versa.
ME: model enumeration, CO + CD $\to$ ME, we keep checking $\Delta|X$, $\Delta|\neg X$, etc.

DNNF: Projection / Existential Qualification

Recall: $\Delta = A \Rightarrow B, B \Rightarrow C, C \Rightarrow D$, existential qualifying B, C, is the same with forgetting B, C, is in other words projecting on A, D.
In **DNNF**, we existential qualifying $\{X_i\}_{i \in \mathcal{S}}$ ($\mathcal{S}$ is a selected set) by:

- replacing all occurrence of X_i (both positive and negative, both X_i and $\neg X_i$) in the DNNF with **true** (Note: result is still DNNF);
- check if the resulting circuit is consistent.

This can be done to DNNF, because:

$$\begin{cases} \exists X.(\alpha \vee \beta) = (\exists x.\alpha) \vee (\exists x.\alpha) \\ \exists X.(\alpha \wedge \beta) = (\exists x.\alpha) \wedge (\exists x.\alpha) \quad // \text{ DNNF only} \end{cases}$$

In DNNF, $\exists X.(\alpha \wedge \beta)$ is $\alpha \wedge (\exists X.\beta)$ or $(\exists X.\alpha) \wedge \beta$.

Minimum Cardinality

Cardinality: in our case, by default, defined as the number of false in an assignment (in a world, how many variables' truth value are **false**). We seek for its minimum. [a]

$$
\begin{aligned}
\text{minCard}(X) &= 0\\
\text{minCard}(\neg X) &= 1\\
\text{minCard}(\textbf{true}) &= 0\\
\text{minCard}(\textbf{false}) &= \infty\\
\text{minCard}(\alpha \vee \beta) &= \min\big(\text{minCard}(\alpha), \text{minCard}(\beta)\big)\\
\text{minCard}(\alpha \wedge \beta) &= \text{minCard}(\alpha) + \text{minCard}(\beta)
\end{aligned}
$$

Again, the last rule holds only in DNNF.
Filling the values into DNNF circuit, we can easily compute the **minimum cardinality**.

- minimizing cardinality requires smoothness;
- it can help us optimizing the circuit by "killing" the child of **or**-nodes with higher cardinality, and further remove dangling nodes.

[a] Could easily be other definitions, such as defined as the number of **true** values, and seek for its maximum.

d-DNNF

CT: model counting. $\text{MC}(\alpha) = |\text{Mods}(\alpha)|$
(decomposable) $\text{MC}(\alpha \wedge \beta) = \text{MC}(\alpha) \times \text{MC}(\beta)$
(deterministic) $\text{MC}(\alpha \vee \beta) = \text{MC}(\alpha) + \text{MC}(\beta)$
counting graph: replacing $\vee$ with $+$ and $\wedge$ with $*$ in a d-DNNF. Leaves: $\text{MC}(X) = 1$, $\text{MC}(\neg X) = 1$, $\text{MC}(\textbf{true}) = 1$, $\text{MC}(\textbf{false}) = 0$.
weighted model counting (**WMC**): can be computed similarly, replacing 0/1 with weights.
Note: **smoothness** is important, otherwise there can be wrong answers. Guarantee smoothness by adding trivial units to a sub-circuit (e.g. $\alpha \wedge (A \vee \neg A)$).
Marginal Count: counting models on some conditions (e.g. counting $\Delta|\{A, \neg B\}$) **CD**+CT.
It is not hard to compute, but the marginal counting is bridging CT to some structure that we can compute **partial-derivative** upon (input: the conditions / assignment of variables), similar to Neural Networks.
FO: forgetting / projection / existential qualification.
Note: a problem occur — the resulting graph might no longer be deterministic, thus d-DNNF is **not** considered successful on polytime FO.

Arithmetic Circuits (ACs)

The **counting graph** we used to do **CT** on d-DNNF is a typical example of Arithmetic Circuits (ACs).
Other operations could be in ACs, such as by replacing "+" by "max" in the counting graph, running it results in the most-likely instantiation. (MPE)
If a Bayesian Net is *decomposable*, *deterministic* and *smooth*, then it could be turned into an Arithmetic Circuits.

Succinctness v.s. Tractability

Succinctness: not expensive; Tractability: easy to use.
Along the line: OBDD $\rightarrow$ FBDD $\rightarrow$ d-DNNF $\rightarrow$ DNNF, succinctness goes up (higher and higher space efficiency), but tractable operations shrunk.

Knowledge-Base Compilation

Top-down approaches:

- Based on exhaustive search;

Bottom-up approaches:

- Based on transformations.

Top-Down Compilation via Exhaustive DPLL

Top-down compilation of a circuit can be done by keeping the trace of an exhaustive DPLL.
The trace is automatically a circuit equivalent to the original CNF Δ.
It is a decision tree, where:

- each node has its high and low children;
- leaves are SAT or UNSAT results.

We need to deal with the redundancy of that circuit.

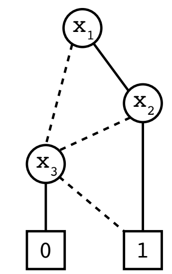
1. Do not record redundant portion of trace (e.g. too many SAT and UNSAT — keep only one SAT and one UNSAT would be enough);
2. Avoid equivalent subproblems (merge the nodes of the same variable with exactly the same out-degrees, from bottom to top, iteratively).

In practice, formula-caching is essential to reduce the amount of work; trade-off: it requires a lot of space.
A limitation of exhaustive DPLL: some conflicts can't be found in advance.

OBDD (Ordered Binary Decision Diagrams)

In an OBDD there are two special nodes: 0 and 1, always written in a square. Other nodes correspond to a variable (say, x_i) each, having two out-edges: high-edge (solid, decide $x_i = 1$, link to high-child), low-edge (dashed, decide $x_i = 0$ link to low-child).

```
Δ=(x1∧x2)∨¬x3
f=x1x2+(1-x3)
```

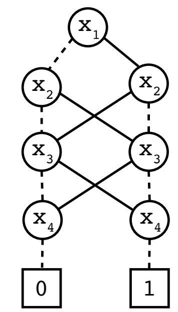


An example of a DNF

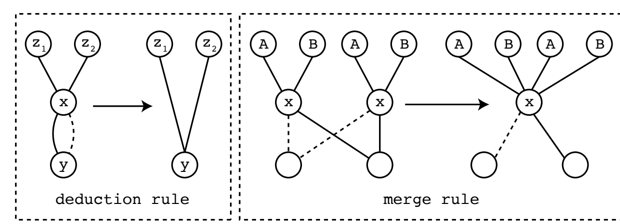
We express KB Δ as function f by turning all $\wedge$ into multiply and $\vee$ into plus, $\neg$ becomes flipping between 0 and 1. None-zero values are all 1. Another example says we want to express the knowledge base where there are odd-number positive values:

```
Odd-parity
function

f=(x1+x2+x3+x4)%2
```



Reduction rules of OBDD:



An OBDD that can not apply these rules is a reduced OBDD. **Reduced OBDDs are canonical**. i.e. Given a fixed variable order, Δ has **only one** reduced OBDD.

OBDD: Subfunction and Graph Size

Considering the function f of a KB Δ, we have a fixed variable order of the n variables $v_1, v_2, \ldots, v_n$; after determining the first m variables, we have up to 2^m different cases of the remaining function (given the instantiation).

The **number of distinct subfunction** (range from 1 to 2^m) **involving** v_{m+1} determines the number of nodes we need for variable v_{m+1}. Smaller is better.

An example: $f = x_1x_2 + x_3x_4 + x_5x_6$, examining two different variable orders: $x_1, x_2, x_3, x_4, x_5, x_6$, or $x_1, x_3, x_5, x_2, x_4, x_6$. Check the subfunction after the first three variables are fixed.

The first order has 3 distinct subfunction, only 1 depend on x_4, thus next layer has 1 node only.

x_1	x_2	x_3	subfunction
0	0	0	x_5x_6
0	0	1	$x_4 + x_5x_6$
0	1	0	x_5x_6
0	1	1	$x_4 + x_5x_6$
1	0	0	x_5x_6
1	0	1	$x_4 + x_5x_6$
1	1	0	1
1	1	1	1

The second order has 8 distinct subfunction, 4 depend on x_2, thus next layer has 4 nodes.

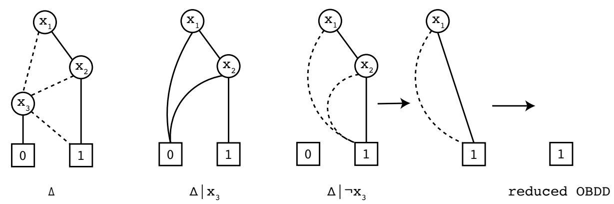
x_1	x_3	x_5	subfunction
0	0	0	0
0	0	1	x_6
0	1	0	x_4
0	1	1	$x_4 + x_6$
1	0	0	x_2
1	0	1	$x_2 + x_6$
1	1	0	$x_2 + x_4$
1	1	1	$x_2 + x_4 + x_6$

Subfunction is a reliable measurement of the OBDD graph size, and is useful to determine which variable order is better.

OBDD: Transformations

$\neg C$: **negation**. Negation on OBDD and on all BDD is simple. Just swapping the nodes 0 and 1 — turning 0 into 1 and 1 into 0, done. $\mathcal{O}(1)$ time complexity.

CD: **conditioning**. $\mathcal{O}(1)$ time complexity. $\Delta|X$ requires re-directing all parent edges of X be directed to its high-child node, and then remove X; similarly $\Delta|\neg X$ re-directs all parent edges of X-nodes to its low-child node, and then remove itself.



$\wedge C$: **conjunction**.

- Conjoining BDD is super easy ($\mathcal{O}(1)$): link the root of Δ_2 to where was node-1 in Δ_1, and then we are done.
- Conjoining OBDD, since we have to keep the order, will be quadratic. Assuming OBDD f and g have the same variable order, and their size (i.e. #nodes) are n and m respectively, time complexity of generating $f \wedge g$ will be $\mathcal{O}(nm)$. This theoretical optimal is achieved in practice, by proper caching.