CS264A Automated Reasoning Review Note

2020 Fall By Zhiping (Patricia) Xiao

Notation

variable	$x, \alpha, \beta, \dots$ (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.

$$\Delta = C_1 \wedge C_2 \wedge \dots$$

written in clausal form as:

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

where each clause C_i is a disjuntion of literals:

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

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*.

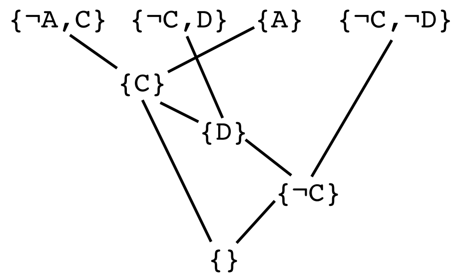
Refutation Theorem

$\Delta \models \alpha$ 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

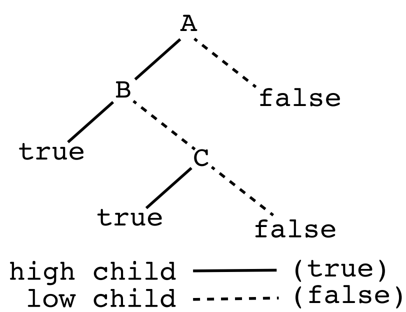
Directed resolution can be applied to forgetting / projecting.
When we do existential quantification on variables $P_1, P_2, \ldots P_m$, we:

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

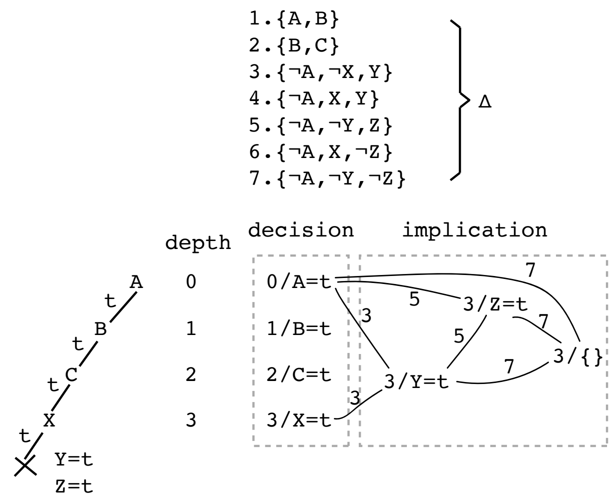
Mostly SAT II, plus unit-resolution.
UNIT-RESOLUTION is used at each node looking for entailed value, to save searching steps.
If there's any implication made by UNIT-RESOLUTION, we write down the values next to the node where the implication is made. (e.g. $A = t, B = f, \ldots$)
This is **NOT** a standard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

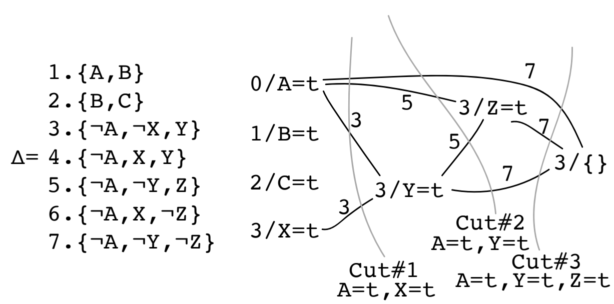
Implication Graph is used to find more clauses to add to the KB, so as to empower the algorithm.
An example of an implication graph upon the first conflict found when running DPLL+ for Δ:



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:



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 $3/Y = t$.

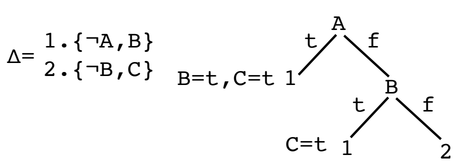
Exhaustive DPLL

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:



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

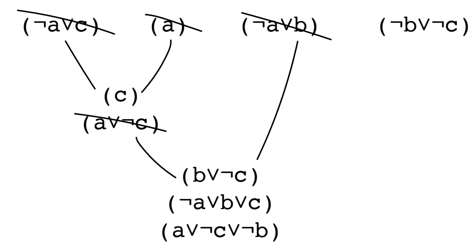
1. Pick an order of the variables, say, $x_1, x_2, \ldots, x_n$
2. For each x_i, exhaust all possible Max-SAT resolutions, the move on to x_{i+1}.
When resolving x_i, using only the clauses that does not mention any x_j, $\forall j < i$.

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.

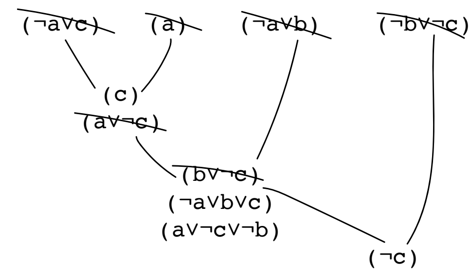
In the end, there remains k false (conflicts), and Γ (guaranteed to be satisfiable). k is the minimum cost, each world satisfying Γ achieves this cost.

Directed Max-SAT Resolution: Example

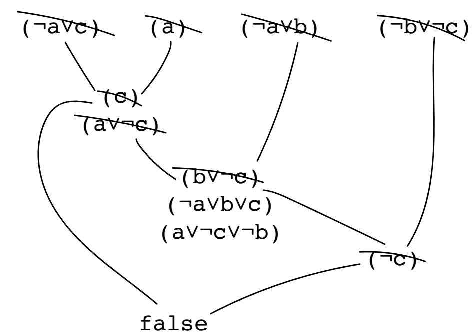
$\Delta = (\neg a \vee c) \wedge (a) \wedge (\neg a \vee b) \wedge (\neg b \vee \neg c)$
Variable order: a, b, c.
First resolve on a:



Then resolve on b:



Finally:



The final output is:

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

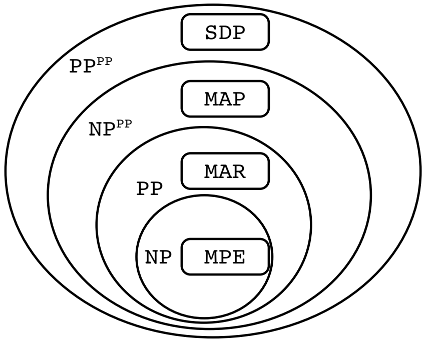
Where $\Gamma = (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$, and $k = 1$, indicating that there must be at least one clause in Δ that is not satisfiable.

Beyond NP

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

Complexity Classes

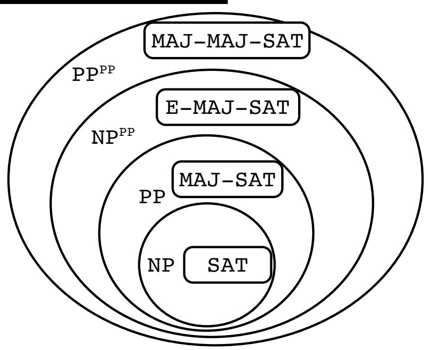
Shown in the figure are some example of the complete problems.



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

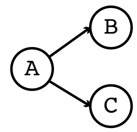
A MAJ-SAT problem consists of:

- #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:

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

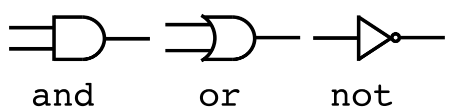
where $\Theta = \{\theta_A, \theta_{\neg A}\} \cup \{\theta_{B|A}, \theta_{\neg B|A}, \theta_{B|\neg A}, \theta_{\neg B|\neg A}\} \cup \{\theta_{C|A}, \theta_{\neg C|A}, \theta_{C|\neg A}, \theta_{\neg C|\neg A}\}$ are the parameters within the Bayesian network at nodes A, B, C respectively, indicating 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

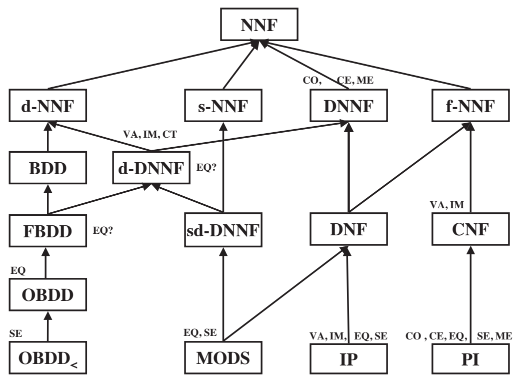
Decomposability: for any **and** node, any pair of its children must be on **disjoint** variable sets. (e.g. one child $A \vee B$, the other $C \vee D$)
Determinism: for any **or** node, any pair of its children must be **mutually exclusive**. (e.g. one child $A \wedge B$, the other $\neg A \wedge B$)
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	✓	✓	✓	✓	✓	○	✓	✓
$OBDD_<$	✓	✓	✓	✓	✓	✓	✓	✓
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	$\Delta \vert P$
FO	forgetting	$\exists P, Q, \dots \Delta$
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	✓	✗	✓	✗	○	✗	○	✓
$OBDD_<$	✓	✗	✓	✗	✓	✗	✓	✓
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
$OBDD_<$	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.
$OBDD_<$: 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 $OBDD_<$ languages. In **this course** we always use **OBDD** to refer to **$OBDD_<$**.
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 $\Delta \models \alpha$ ($\alpha = \ell_1 \vee \ell_2 \dots \ell_n$) 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. ($\Delta \rightarrow (\neg P \wedge \Delta|\neg P) \vee (P \wedge \Delta|P)$) 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 $\rightarrow$ 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 $\to$ FBDD $\to$ d-DNNF $\to$ 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.

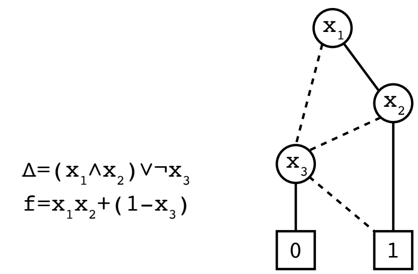
We need to deal with the redundancy of that circuit.

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.

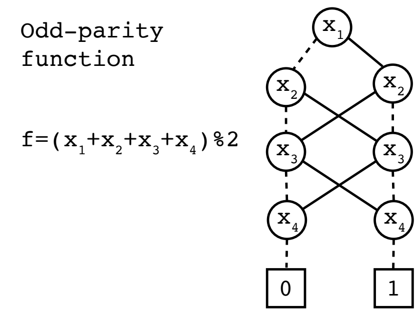
OBDD (Ordered Binary Decision Diagrams)

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).

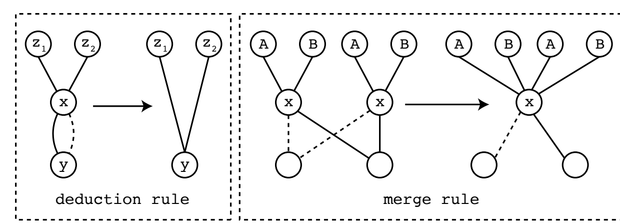


An example of a DNF

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:



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 $v_1, v_2, \dots, v_n$; 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.

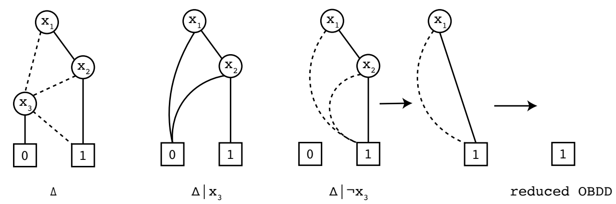
x_1	x_3	x_5	subfunction
0	0	0	0
0	0	1	x_6
0	1	0	x_4
0	1	1	$x_4 + x_6$
1	0	0	x_2
1	0	1	$x_2 + x_6$
1	1	0	$x_2 + x_4$
1	1	1	$x_2 + x_4 + x_6$

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

OBDD: Transformations

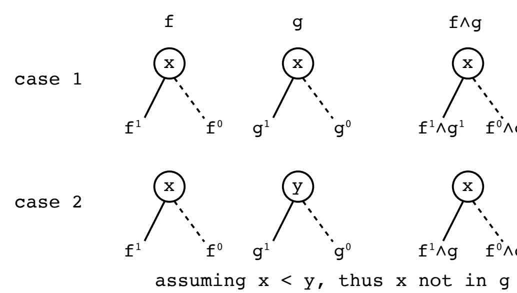
$\neg C$: **negation**. Negation on OBDD and on all BDD is simple. Just swapping the nodes 0 and 1 — turning 0 into 1 and 1 into 0, done. $\mathcal{O}(1)$ time complexity.

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.



SDDs (Sentential Decision Diagrams)

SDD is the most popular generalization of OBDD. It is also a circuit type.

- Order: needed, and matters
- Unique: when canonical / reduced

SDD: Structured Decomposability

Decomposability:

$$f(ABCD) =$$
$$f_1 \quad [g_1(AB) \wedge h_1(CD)] \vee$$
$$f_2 \quad [g_2(A) \wedge h_2(BCD)] \vee \dots$$

Structured Decomposability:

$$f(ABCD) =$$
$$f_1 \quad [g_1(AB) \wedge h_1(CD)] \vee$$
$$f_2 \quad [g_2(AB) \wedge h_2(CD)] \vee \dots$$

feature: variables split in the same way in each subfunction.

SDD: Partitioned Determinism

An $(\mathbf{X}, \mathbf{Y})$-partition of a function f goes like:

$$f(\mathbf{X}, \mathbf{Y}) = g_1(\mathbf{X})h_1(\mathbf{Y}) + \cdots + g_n(\mathbf{X})h_n(\mathbf{Y})$$

where $\mathbf{X} \cap \mathbf{Y} = \varnothing$ and $\mathbf{X} \cup \mathbf{Y} = \mathcal{V}$ where $\mathcal{V}$ are all the variables we have for function f.
It is called a **structured decomposability**.
g_i regarding $\mathbf{X}$ is called a **prime**, and h_i regarding $\mathbf{Y}$ is called a **sub**.
Requirements on the primes are:

$$\begin{cases} \forall i, j \; g_i \wedge g_j = \textbf{false} & \text{//mutual exclusiveness} \\ g_1 \vee \cdots \vee g_n = \textbf{true} & \text{//exhaustive} \\ \forall i \; g_i \neq \bot & \text{//satisfiable} \end{cases}$$

VTree

Vtree is a binary tree that denotes the order and the structure of a **SDD**. Each node's left branch refers to the element in the **primes**, and each node's right branch refers to that of the **subs**.

From OBDD to SDD

OBDD is a special case of **SDD** with right-linear [a] **vtree**.
SDD is a *strict superset* of **OBDD**, maintaining key properties of **OBDD** [b], and **could be** exponentially smaller than **OBDD**.

[a] Right-linear means that each node's left child is a leaf.
[b] What is called a path-width in **OBDD** is called a tree-width in **SDD**

SDD: Compression

$(\mathbf{X}, \mathbf{Y})$-partition is **compressed** if there is **no** equal **subs**. That is,

$$h_i \neq h_i, \ \forall i \neq j$$

Any f has a unique compressed $(\mathbf{X}, \mathbf{Y})$-partition.

Systematic Way of Building SDD: Example

Given: $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$
$\mathbf{X} = \{A, B\}$
$\mathbf{Y} = \{C, D\}$

Then we can have the sub-functions (**subs**) as conditioned on the primes:

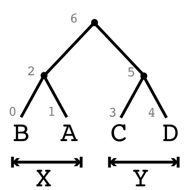
prime	*sub*
$A \wedge B$	**true**
$A \wedge \overline{B}$	$C \wedge D$
$\overline{A} \wedge B$	C
$\overline{A} \wedge \overline{B}$	$C \wedge D$

Resolving the primes with the same sub, to conduct **compression**:

prime	*sub*
$A \wedge B$	**true**
$\overline{A} \wedge B$	C
$\overline{B}$	$C \wedge D$

$$f = \underbrace{\underbrace{(A \wedge B)}_{\text{prime}} \underbrace{(\textbf{true})}_{\text{sub}}}_{f_1} + \underbrace{\underbrace{(\overline{A} \wedge B)}_{\text{prime}} \underbrace{(C)}_{\text{sub}}}_{f_2} + \underbrace{\underbrace{(\overline{B})}_{\text{prime}} \underbrace{(C \wedge D)}_{\text{sub}}}_{f_3}$$

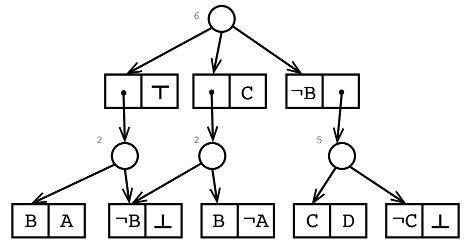
One possible **vtree** is:



Note that there other possible **vtree**s, but under this circumstance, where $\mathbf{X}$ and $\mathbf{Y}$ are fixed, the leaves under the left branch of the root has to contain and only contain variables belong to $\mathbf{X}$, and right branch for $\mathbf{Y}$. For intermediate nodes (neither leave nor root), do the same **recursively**.

Construct an SDD: Example

Following the **previous example**, using that specific vtree, the **SDD** we construct looks like:



where $\top$ stands for always **true** and $\bot$ for always **false**. [a][b] Each node consists of a head and a tail; for either a head or a tail, if it involves *more than one* variable (a.k.a representing an intermediate node in the vtree), we need to decompose it again (according to its left-right branches in the vtree).

OBDDs are **SDD**s where the partition at any node has $|\mathbf{X}| = 1$, being a Shanoon decomposition $(g_i(\mathbf{X})h_i(\mathbf{Y}|\mathbf{X}))$.

In a **SDD** circuit, the in-signals of any **or**-gate are either **one-high** or **all-low** (when, for example, the selected prime has a $\bot$ sub).

[a] https://en.wikipedia.org/wiki/List_of_logic_symbols
[b] https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols

Same Partition: Polytime Operation

$(\mathbf{X}, \mathbf{Y})$-partition of

$$f : (p_1, q_1) \dots (p_n, q_n)$$
$$g : (r_1, s_1) \dots (r_m, s_m)$$

which means that, for example,

$$f = p_1(\mathbf{X})q_1(\mathbf{Y}) + \dots + p_n(\mathbf{X})q_n(\mathbf{Y})$$

And then we have the $(\mathbf{X}, \mathbf{Y})$-partition of $f \circ g$ being:

$$(p_i \wedge r_j, q_i \circ s_j | p_i \wedge r_j \neq \textbf{false})$$

where there are $m \times n$ sub-functions in total.
Note: at this stage, *compression* is **not** guaranteed.

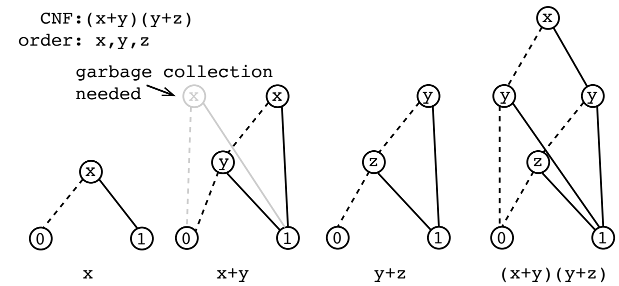
Bottom-Up Compilation (OBDD/SDD)

- To compile a CNF:

$$\begin{cases} \text{OBDD/SDD for literals} \\ \text{disjoint literals to clause} \\ \text{disjoint clauses to CNF} \end{cases}$$

- Similar to DNF
- Works for every Boolean formula

An example of the bottom-up compilation:



Note: I've directly omitted a lot of nodes that are **garbage-collected** in the middle. For instance, shown on the second step is where we do garbage collection for the first x literal node.
Garbage-collection: for the sake of **memory**.
Challenges: good variable order, apply (e.g. conjoint, disjoint) operations scheduler, etc.
Top-Down v.s. Bottom-Up: bottom-up approaches are typically more **space-consuming**, yet more **flexible**. (Sometimes, $f_1 \wedge f_2$ could be simple when f_1 and f_2 on their own are complex.)

Canonicity in Compilation

OBDDs are canonical:
fixed **variable order** $\rightarrow$ unique reduced OBDD
SDDs are canonical:
fixed **vtree** $\rightarrow$ unique trimmed & compressed SDD
Note: *variable ordering* has great impact on OBDD **size**; *vtree* has significant impact on SDD **size**.

Minimizing OBDD Size

n variables lead to $n!$ possibilities. We swap two adjacent variables to change variable order. (This can be done easily, and could explore all possibilities.)

Minimizing SDD Size

The key point of optimizing the SDD size is to find the best **vtree**. *A vtree **embeds** a variable order.* There are two approaches to find a good vtree:

- **statically**: by pre-examining the Boolean function
- **dynamically**: by searching for an appropriate one at runtime

Distinct sub-functions matter. Different vtrees can have exponentially different SDD sizes.

Counting Vtrees

A vtree **embeds** a variable order because the variable order can be obtained by a **left-right traversal** of the vtree. Vtree **dissects** a variable order, it tells the division among primes and subs explicitly.

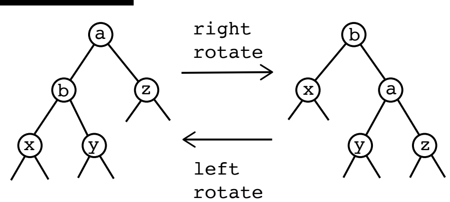
- # variable orders: $n!$ (n: #vars)
- # dissections: $C_{n-1} = \frac{\big(2(n-1)\big)!}{n!(n-1)!}$ (**Catalan** number, # full binary trees with n leaves.)
- # **vtree**s over n variables:

$$n! \times C_{n-1} = \frac{\big(2(n-1)\big)!}{(n-1)!}$$

Searching Over Vtrees

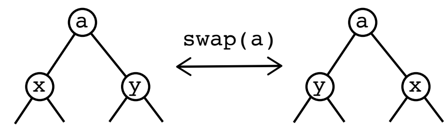
- a Double-search problem $\begin{cases}\text{variable order}\\ \text{dissection}\end{cases}$
- using tree operations $\begin{cases}\text{rotating}\\ \text{swapping}\end{cases}$

Tree Rotations



It preserves variable order; enumerates all dissections.

Tree Swapping



Searching Over Vtrees: in Practice

Vtree fragments [a]: root, child, left-linear fragment (beneath the left child), right-linear fragment (beneath the right child).
Fragment operations: next, previous, goto, etc.
swap + rotate: enough to explore all possible vtrees.
in practice: we need time limit to avoid exploding ourselves.

greedy search:

- enumerate all vtrees over a *window* (i.e. reachable via a certain amount of rotate/swap operations)
- greedily accept the best vtree found, and then move window

[a] Fragment: (possibly empty) connected subgraph of a binary tree; unlike subtree = root node + descendents of that node, a fragment need not include all descendents of its root.

SDD, PSDD and Conditional PSDD

These are circuits of learning from Data & Knowledge.

year	model	comments
	probability space	the truth table
2011	SDD	Tractable Boolean Circuit
2014	PSDD	P: Probabilistic
2018	Conditional PSDD	conditional probability

Impact of knowledge (supervised/unsupervised):

- reduce the **amount of data** needed (for training)
- improve **robustness** of ML systems
- improve **generality** of ML systems

Truth table: world, instantiation, 1/0.
Probability distribution: world, instantiation, $\Pr(\cdot)$.

Probabilistic: Review

- Marginal Probability: formally the marginal probability of X can always be written as an expected value:

$$\begin{aligned} p_X(x) &= \int_y p_{X|Y}(x \mid y)\, p_Y(y)\, \mathrm{d}y \\ &= \mathrm{E}_Y[p_{X|Y}(x \mid y)] \end{aligned}.$$

  computed by examining the conditional probability of $\mathbf{X}$ (some variables) given a particular value of $\mathbf{Y}$ (the remaining variables), and then averaging over the distribution of all $\mathbf{Y}$s.

  In our case it is usually the **sum** of some worlds' probabilities ($\sum_i \Pr(\omega_i)$).

- Conditional Probability:

$$\Pr(\alpha|\beta) = \frac{\Pr(\alpha,\beta)}{\Pr(\beta)}$$

To compute them efficiently/effectively, we can use circuits.
SDD (probability version): $(\mathbf{X},\mathbf{Y})$-Partition,

$$f(\mathbf{X},\mathbf{Y}) = g_1(\mathbf{X})h_1(\mathbf{Y}) + \cdots + g_n(\mathbf{X})h_n(\mathbf{Y})$$

$$\begin{cases}\forall i,\ g_i \neq 0 & \\ \forall i \neq j,\ g_i g_j = 0 & \text{(mutually exclusive)} \\ g_1 + g_2 + \cdots + g_n = 1 & \text{(exhaustive)}\end{cases}$$

where in this case g_i are the probabilities.
Compressed ($\forall i \neq j,\ h_i \neq h_j$) $(\mathbf{X},\mathbf{Y})$-Partition of f is **unique**.
e.g. Given α, we have

$$\Pr(\alpha) = \sum_{i=1}^{n} \Pr(\alpha|g_i)\Pr(g_i)$$

Structured Space: instead of considering all possible worlds, crossing-off some worlds for not satisfying some known **constraints**.

- e.g. Routes: nodes are cities, edges are streets. Assign to edge value 1 for being on the route and 0 for not. *Structure: being a route.* Unstructured assignment has 2^m possibilities where m is the number of possible streets (0/1 for each).

From SDD to PSDD

PSDD, compared to SDD, is almost the same, except that:

- **OR-gates**: having probability distributions over all inputs.
- Any two OR-gates may have **different** probability distributions.

The **AND-gates** are just kept the same and no probability applies.

PSDD: Probability of Feasible Instantiations

Evaluating the circuit top-side down — for each world, from the top, tracing one child at each OR-gate, tracing all children at each AND-gate. Then we have $\Pr(\omega_i)$.
Interpreting PSDD Parameters: At each **OR-gate**, it induces a normalized distribution satisfying assignments. The probability distribution corresponds to the probabilities of **primes**.

PSDD: Computing Marginal Probabilities

In this case, marginal probabilities refers to the probabilities of some ***partial assignments*** *(e.g.* $\Pr(A = t, B = f)$ *when variables are* A, B, C, D*).*
PSDDs are ACs (OR: +, AND: *).
The challenge is that: parameters (probability distribution) unknown.

PSDD: Learning Background Knowledge

We learn the **parameters** of PSDD via evidence.
Evidence: observed data sample.
First we have the SDD structure.
Then, we have **Data** such as:

L	K	P	A	#samples
0	0	1	0	6
0	0	1	1	10
		...		

Starting from the top, trace the **high-wired** (1, one under OR, all under AND) for each sample. Assign 1 for each sample along the trace, under OR-gate. [a]
Normalizing under each OR-gate. (Sum up to 1.)

[a] e.g. In this case, the OR-gates input high wires corresponding to $\neg L \wedge \neg K \wedge P \wedge \neg A$ are assigned $0 + 6 = 6$. If the same edge gets assignment ≥ 2 times, sum them up (e.g. $6+10 = 16$).

Likelihood

For model $\Pr(\cdot)$, and PSDD with parameters θ, the idea is that we evaluate the quality of the parameters by likelihood (e_i is a single observation — *the line with count* 6 *are actually* 6 *observations*).

$$L(\text{Data}|\theta) = \Pr_\theta(e_1) * \Pr_\theta(e_2) * \dots \Pr_\theta(e_n)$$

Dataset Incompleteness

Incomplete data means that for some worlds / observations, there are some variable instantiation missing.

Dataset Type	Algorithm
Classical Complete	Closed-form Solution [a]
Classical Incomplete	EM Algorithm (on PSDD)
Non-classical Incomplete	N/A in ML

Non-classical Incomplete Dataset example:

$$x_2 \wedge (y_2 \vee z_2), \quad x \Rightarrow y, \dots$$

It is good to define arbitrary events.
Missing in the ML literature, conceptually doable but there are computational reasons. (See extension readings mentioned in class.)

[a] Unique maximum-likelihood estimates.

PSDD Multiplication

factor $\leftarrow$ { distribution, normalization constant κ }
factor: worlds' instantiation, and sample count (integer)
distribution: worlds' instantiation, and probability
Consider the tables as matrices, then $\mathbf{F} = \mathbf{D} * \kappa$.
Normalization needs to be re-done after multiplication. (Multiplying two circuits.)
Aligning the rows of worlds in the factor table, the resulting factor table (of multiplication) is computed via multiplying each row's value (# samples multiplied).
Besides, it **doesn't** means that, when $\kappa_1 * PSDD_1 \times \kappa_2 * PSDD_2 = \kappa_3 * PSDD_3$, $\kappa_1, \kappa_2, \kappa_3$ have **any** correlation. (**Can't** expect to have $\kappa_3 = \kappa_1 \times \kappa_2$.)
The PSDD circuits involved $(PSDD_1, PSDD_2, PSDD_3)$ **doesn't** need to be similar at all.
An application: Compiling Bayesian Network into PSDDs. e.g. $PSDD_{all} = PSDD_A * PSDD_B * PSDD_{C|AB} * PSDD_{D|B} \dots$

Conditional PSDD

Conditional PSDD models $\Pr(\alpha|\beta)$.
Its circuit is always a **hybrid** — from root to leave, SDD on top and PSDD at the bottom. Meaning that condition β's probability is not important at all.
An application: **hierarchical map**. If we treat each part of the map as conditional PSDD *conditioning on the outer connections*, then we can solve a very big map by safely dividing it into smaller maps.

Conditional Vtrees

Conditional PSDDs of $\Pr(\mathbf{Y}|\mathbf{X})$ need **conditional vtrees**. $\mathbf{X}$, $\mathbf{Y}$ are sets of variables, $\mathbf{X}$ includes the conditions.
The **conditional vtree** must contain a node, with **precisely** the variables in $\mathbf{X}$ contained in the subtree beneath it. Then this node is called a $\mathbf{X}$-node, denoted as a $*$ instead of a $\cdot$ when drawing the vtree.
The $\mathbf{X}$-node must be reachable from the root of the vtree by *only following the* ***right*** *children.*

Prime Implicate (PI), Prime Implicant (IP)

The two concepts are closely-related. Δ is the knowledge base.

Prime Implicate (PI)	Prime Implicant (IP)
clauses	terms
CNF no subsumed clauses	DNF no subsumed terms
Implicate c of Δ: $\Delta \models c$	Implicant t of Δ: $t \models \Delta$
Resolution $\frac{\alpha \vee x, \beta \vee \neg x}{\alpha \vee \beta}$	Consensus $\frac{\alpha \wedge x, \beta \wedge \neg x}{\alpha \wedge \beta}$
$\wedge$ prime implicates of Δ	$\vee$ prime implicants of Δ

To obtain PI/IP: Close Δ Under Resolution / Consensus then drop subsumed clauses/terms.
Subsume = all-literals already contained:

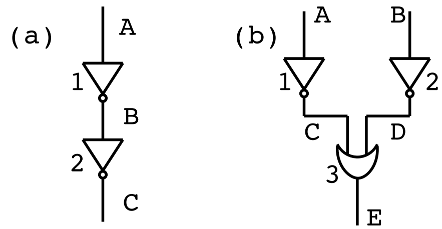
- Clauses: c_1 subsumes c_2, for $c_1 = A \vee \neg B$, $c_2 = A \vee \neg B \vee C$; $c_1 \models c_2$.
- Terms: t_1 subsumes t_2, for $t_1 = \neg A \wedge B$, $t_2 = \neg A \wedge B \wedge \neg C$; $t_2 \models t_1$.

For PI, existential quantification, and CE (clausal entailment check), are easy.
Prime means a clause/term is **not subsumed** by any any other clause/term.

Duality: $\begin{cases} \alpha \text{ is a prime implicate of } \Delta \\ \neg\alpha \text{ is a prime implicant of } \neg\Delta \end{cases}$

Model-Based Diagnosis

In a circuit, on each **edge** (connecting two gates) there is a signal (high or low), denoted as $X, Y, A, B, C, \ldots$, (α, could be directly observed).
For each **gate** (usually numbered $1, 2, \ldots$), there is one extra variable ($ok1, ok2, \ldots$) called **health variable**, representing whether or not the gate is correctly functioning.
Δ contains $A, B, C, \ldots ok1, ok2, \ldots$. Examples:



$$\Delta_a = \begin{cases} ok1 \Rightarrow (A \iff \neg B) \\ ok2 \Rightarrow (B \iff \neg C) \end{cases}$$

$$\Delta_b = \begin{cases} ok1 \Rightarrow (A \iff \neg C) \\ ok2 \Rightarrow (B \iff \neg D) \\ ok3 \Rightarrow ((C \vee D) \iff E) \end{cases}$$

Model-Based Diagnosis figure out what are the possible situations of **health variables** when given Δ and α (an observation, e.g. $\alpha_a = C, \alpha_b = \neg E$, etc.).
Δ here is called a **system**, and α is **system observation**.
For example: in case (a), if $\Delta \wedge \alpha \wedge ok1 \wedge ok2$ is **satisfiable** (using SAT solver) then health condition $ok1 = t, ok2 = t$ is **normal**, otherwise it is **abnormal**.
To do **diagnosis** we conclude **all** the normal assignments of the health variables.
e.g. Example (b), $\alpha = \neg A, \neg B, \neg E$, diagnosis:

$ok1$	$ok2$	$ok3$	normal?
✓	✓	✓	no
✓	✓	✗	yes
✓	✗	✓	no
✓	✗	✗	yes
✗	✓	✓	no
✗	✓	✗	yes
✗	✗	✓	yes
✗	✗	✗	yes

Concluding all **yes** and simplify: $\neg ok3 \vee (\neg ok1 \wedge \neg ok2)$.

Health Condition

Health condition of system Δ given observation α is:

$$\text{Health}(\Delta, \alpha) = \exists \underbrace{\ldots}_{\text{all except } oki} \Delta \wedge \alpha$$

— projection of $\Delta \wedge \alpha$ onto health variables oki.
Note: Could be done easily by bucket resolution + forgetting we've learned before.

Methods of Diagnosis

Based on health condition $\text{Health}(\Delta, \alpha)$ we can do model-based diagnosis.

CNF:
- **conflict**: implicates of $\text{Health}(\Delta, \alpha)$
- **min-conflict**: **PI** of $\text{Health}(\Delta, \alpha)$

DNF:
- **parlid**: implicant of $\text{Health}(\Delta, \alpha)$
- **kernel**: **IP** of $\text{Health}(\Delta, \alpha)$

Minimum Cardinality Diagnosis: turn the health condition $\text{Health}(\Delta, \alpha)$ into a DNNF and then compute the minCard. The path with minimum cardinality corresponds to the solution.

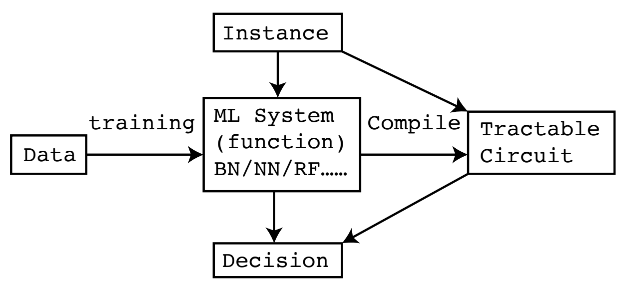
Current Topics

- Explaining decisions of ML systems. (see "Why Should I Trust You?" (KDD'16))
- Measuring robustness of decisions.

Readings:

- Three Modern Roles for Logic in AI (PODS'20)
- Human-Level Intelligence or Animal-Like Abilities? (CACM'18)

Explanation (Explaining Decisions)



(BN: Bayesian Nets, NN: Neural Nets, RF: Random Forests)

Classifiers: Review

Function version of a classifier:

$$f(x_1, \ldots, x_n)$$

where x_i are called features, all features $x_1, x_2, \ldots x_n$ together: instance; output of f: decision (classification); positive/negative decision refer to $f = 1/0$ respectively, while the corresponding instances are called positive/negative instantiation.

- Boolean Classifier: x_i, f have Boolean values
  - Propositional Formula as Classifier: $\omega \models \Delta$ positive and $\omega \models \neg\Delta$ negative.
- Monotone Classifier: positive instance remains positive if we flip some features from $-$ to $+$
  - e.g. $f(+--+) \rightarrow + \Rightarrow f(++++) \rightarrow +$

Minimum Cardinality of Classifiers: the number of false variables (negative features). **Note:** Computed on DNNF easily. Sometimes the circuit can be minimized when (1) smooth (2) prune some edges that aren't helpful to minCard.

- **Sub-Circuit**: **a model**; trace-down one child of OR-gates and all children of AND-gates.

MC Explanations and PI Explanations

MC Explanations (MC: Minimum Cardinality)

- which positive features are responsible for a yes decision? (negative: vice versa)
- computed in linear time on **DNNF*** (def: Δ, $\neg\Delta$ are both DNNF)
- to answer Q1: which positive features, def: minCard = # positive variables; condition on the negative features observed in the current case,; compute minCard; minimizing (kill unhelpful nodes, edges); enumerate (subcircuits)

PI Explanations (PI: Prime Implicant)

- characteristics make the rest irrelevant?
- compute PI; **sufficient reasons** are all the PI terms.

Decision and Classifier Bias: Definition

Protected features: we don't want them to influence the classification outcome. (e.g. gender, age)
Decision is biased if the result changes when we flip the value of a **protected** feature.
Classifier is biased if one of its decisions is biased.

Decision and Classifier Bias: Judgement

Theorem: Decision is biased iff each of its sufficient reasons contains at least one protected feature.

Complete Reason (for Decision)

Complete reason is the disjunction ($\vee$) of all **sufficient reasons**. (e.g. $\alpha = (E \wedge F \wedge G) \vee (F \wedge W)$ — we made the decision because of α)

Reason Circuit (for Decision)

Reason Circuit: tractable circuit representation of the **complete reason**.
If the classifier is in a special form (e.g. OBDD, Decision-DNNF), then reason circuit can be obtained directly in linear time. How:

1. compile the classifier into a circuit, and get a **positive instance** ready (otherwise work on **negation** of the classifier circuit);
2. add **consensus**:

$$\frac{(\neg A \wedge \alpha) \vee (A \wedge \beta)}{\alpha \wedge \beta}$$

   add all the $\alpha \wedge \beta$ terms into the circuit.

3. **filtering**: go to the branches incompatible with the instance and kill them.
   - the reason circuit thereby **monotone** (positive feature remains positive, negative feature remains negative)
   - because of monotone, can do existential quantification in linear time.

The reason circuit can be used to handle queries such as: sufficient reasons, necessary properties, necessary reason, because statement, ...

Reasoning about ML Systems: Overview

Queries	Explanation, Robustness, Verification, etc.
ML Systems	Neural Networks, Graphical Models, Random Forests, etc.
Tractable Circuits	OBDD, SDD, DNNF, etc.

For more: `http://reasoning.cs.ucla.edu/xai/`

Robustness (for Decision / Classifier)

Hamming Distance (between instances): the number of disagreed features. Denoted as $d(x_1, x_2)$.
Instance Robustness:

$$\text{robustness}_f(x) = \min_{x': f(x') \neq f(x)} d(x, x')$$

Model Robustness:

$$\text{model\_robustness}(f) = \frac{1}{2^n} \sum_x \text{robustness}_f(x)$$

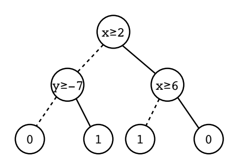
Instance Robustness is the minimum amount of flips needed to change decision. Model Robustness is the average of all instances' robustness. (2^n is the amount of instances.)
e.g. odd-parity: the model-robustness is 1.

Compiling I/O of ML Systems

By compiling the input/output behavior of ML systems, we can analyze classifiers by tractable circuits.
From easiest to hardest **conceptually**: RF, NN, BN
main challenge: **scaling** to large ML systems

Compiling Decision Trees and Random Forests

DT (decision tree): could transfer into multi-valued propositional logic



where $x \in (-\infty, 2) \rightarrow x = x_1$, $x \in [2, 6) \rightarrow x = x_2$, $x \in [6, +\infty) \rightarrow x = x_3$; $y \in (-\infty, -7) \rightarrow y = y_1$, $y \in [-7, +\infty) \rightarrow y = y_2$.
RF (random forest): majority voting of many DTs.

Compiling Binary Neural Networks

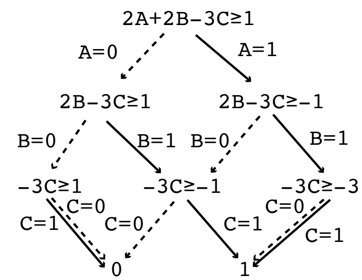
This is a very recent topic.
Binary: the whole NN represents a Boolean Function.

- Input to the NN (and to each neuron): Boolean (0/1)
- Step activation function:

$$\sigma(x) = \begin{cases} 1 & \sum_i w_i x_i \geq T \\ 0 & \text{otherwise} \end{cases}$$

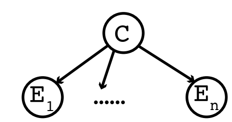
  where in this case the neuron has a threshold T and inputs from the last layer are: $x_1, x_2, \ldots, x_i, \ldots$, with corresponding weights $w_1, w_2, \ldots, w_i, \ldots$.

For instance, a neuron that represents $2A + 2B - 3C \geq 1$ can be reduced to a Boolean circuit:



Naïve Bayes Classifier

Naïve Bayes Classifier:



- Class: C (all E_i depend on C)
- Features: $E_1, \ldots E_n$ (conditional independent)
- Instance: $e_1, \ldots e_n = \mathbf{e}$
- Class Posterior: (note that) $\Pr(\alpha|\beta) = \frac{\Pr(\alpha \wedge \beta)}{\Pr(\beta)}$

$$\Pr(c|e_1, \ldots, e_n) = \frac{\Pr(e_1, \ldots e_n)\Pr(c)}{\Pr(e_1, \ldots e_n)} = \frac{\Pr(e_1|c) \ldots \Pr(e_n|c)\Pr(c)}{\Pr(e_1, \ldots e_n)}$$

Naïve Bayes: CPT

A Bayesian Network has **conditional probability tables (CPT)** at each of its node.
e.g. previous example node C CPT:

C	Θ_C
c_1	θ_{c_1} (e.g., 0.1)
...	
c_k	θ_{c_k} (e.g., 0.2)

where $\forall i\ \theta_{c_i} \in [0,1]$, $\sum_{i=1}^{k} \theta_{c_i} = 1$.
And at node E_j, the CPT:

C	E_j	$\Theta_{E_j\|C}$
c_1	$e_{j,1}$	$\theta_{e_{j,1}\|c_1}$ (e.g., 0.01)
c_1	$e_{j,2}$	$\theta_{e_{j,2}\|c_1}$ (e.g., 0.03)
...		
c_1	$e_{j,q}$	$\theta_{e_{j,q}\|c_1}$ (e.g., 0.1)
c_2	$e_{j,1}$	$\theta_{e_{j,1}\|c_2}$ (e.g., 0.01)
...		
c_k	$e_{j,q}$	$\theta_{e_{j,q}\|c_k}$ (e.g., 0.02)

where $\forall i,j,x\ \theta_{e_{j,x}|c_i} \in [0,1]$, $\forall i,j\ \sum_{x=1}^{q} \theta_{e_{j,x}|c_i} = 1$.
Under a condition, the marginal probability is 1.
$\forall i, e_i, c,\ \Pr(e_i|c)$ are all in CPT tables.

Odds v.s. Probability

Probability: $\Pr(c)$ (chance to happen, $[0,1]$)

Odds: $\mathrm{O}(c) = \dfrac{\Pr(c)}{\Pr(\bar{c})}$

$$\mathrm{O}(c|e) = \frac{\Pr(c|e)}{\Pr(\bar{c}|e)} \quad \text{(conditional odds)}$$

$$\log \mathrm{O}(c|e) = \log \frac{\Pr(c|e)}{\Pr(\bar{c}|e)} \quad \text{(log odds)}$$

In the previous example, if we use log odds instead of probability, $\Pr(\alpha) \geq p \iff \log \mathrm{O}(\alpha) \geq \rho = \frac{p}{1-p}$

$$\mathrm{O}(c|\mathbf{e}) = \log \frac{\Pr(c)\prod_{i=1}^{n} \Pr(e_i|c)/\Pr(\mathbf{e})}{\Pr(\bar{c})\prod_{i=1}^{n} \Pr(e_i|\bar{c})/\Pr(\mathbf{e})}$$
$$= \log(c) + \sum_{i=1}^{n} \log \frac{\Pr(e_i|c)}{\Pr(e_i|\bar{c})} = \log(c) + \sum_{i=1}^{n} w_{e_i}$$

w_{e_i} is weight of evidence e_i, depending on instance. $\log \mathrm{O}(c)$ is the **prior log-odds**. Changing class prior (shift $\log \mathrm{O}(c)$) shifts all $\log \mathrm{O}(c|e_i)$ the same amount.

Compiling Naive Bayes Classifier

Brutal force method: consider sub-classifiers — $\Delta|U$ and $\Delta|\neg U$, recursively.
Problem: can have exponential size (to # variables).
Solution: cache sub-classifiers.
Note: Naïve Bayesian Network has **threshold** T and **prior** (e.g. in the previous example, we have prior of C, and if $\Pr(C = c_i|E_j = e_{j,x}) \geq T$ then, for example, the answer is **yes**, otherwise **no**). We may have **different conditions, different conditional probabilities**, sharing the **same sub-classifier**.

Application: Solving MPE & MAR

MPE: most probable explanation
→ NP-complete
→ probabilistic reasoning, find the world with the largest probability
→ solved by **weighted** MaxSAT
→ compile to DNNF
MAR: marginal probability
→ PP-complete
→ sum of all worlds' probabilities who satisfy certain conditions
→ solved by **WMC (weighted model counting)**
→ compile to d-DNNF
conditional version: work on "shrunk table" where some worlds are removed

Solving MPE via MaxSAT

- Input: weighted CNF $= \alpha_1, \ldots, \alpha_n$ (with weights $w_1, \ldots, w_n$)
  - $(x \vee \neg y \vee \neg z)^3$, $(\neg x)^{10.1}$, $(y)^{.5}$, $(z)^{2.5}$
  - next to the clauses, $3, 10.1, 0.5, 2.5$ are the corresponding weights
  - W: the weight of **hard** clauses, greater than the sum of all soft clauses' weights
- find variable assignment with the highest **weight** / least **penalty**

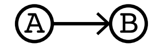
$$\mathrm{Wt} = \mathrm{weight}(x_1,\ldots,x_n) = \sum_{x_1,\ldots x_n \models \alpha_i} w_i$$

$$\mathrm{Pn} = \mathrm{penalty}(x_1,\ldots,x_n) = \sum_{x_1,\ldots x_n \not\models \alpha_i} w_i$$

$$\mathrm{Wt}(x_1,\ldots,x_n) + \mathrm{Pn}(x_1,\ldots,x_n) = \Psi \quad \text{(constant)}$$

Solving MPE via MaxSAT: Example

Given a Bayesian Network (with CPT listed):



A	θ_A
a_1	0.3
a_2	0.5
a_3	0.2

A	B	$\theta_{B\|A}$
a_1	b_1	0.2
a_1	b_2	0.8
a_2	b_1	1
a_2	b_2	0
a_3	b_1	0.6
a_3	b_2	0.4

- Indicator variables:
  - from A (values a_1, a_2, a_3): $I_{a_1}, I_{a_2}, I_{a_3}$
  - from B (values b_1, b_2): I_{b_1}, I_{b_2}
- **Indicator Clauses**:

$$A \begin{cases} (I_{a_1} \vee I_{a_2} \vee I_{a_3})^W \\ (\neg I_{a_1} \vee \neg I_{a_2})^W \\ (\neg I_{a_1} \vee \neg I_{a_3})^W \\ (\neg I_{a_2} \vee \neg I_{a_3})^W \end{cases} \qquad B \begin{cases} (I_{b_1} \vee I_{b_2})^W \\ (\neg I_{b_1} \vee \neg I_{b_2})^W \end{cases}$$

- **Parameter Clauses**: (= $\sum$# rows in CPTs)

$$A \begin{cases} (\neg I_{a_1})^{-\log(.3)} \\ (\neg I_{a_2})^{-\log(.5)} \\ (\neg I_{a_3})^{-\log(.2)} \end{cases} \qquad B \begin{cases} (\neg I_{a_1} \vee \neg I_{b_1})^{-\log(.2)} \\ (\neg I_{a_1} \vee \neg I_{b_2})^{-\log(.8)} \\ (\neg I_{a_2} \vee \neg I_{b_1})^{-\log(1)} \\ (\neg I_{a_2} \vee \neg I_{b_2})^{-\log(0)} \\ (\neg I_{a_3} \vee \neg I_{b_1})^{-\log(.6)} \\ (\neg I_{a_3} \vee \neg I_{b_2})^{-\log(.4)} \end{cases}$$

  where we define $W = \log(0)$.
- the weighted CNF contains all **Indicator Clauses** and **Parameter Clauses**
- **Evidence**: e.g. $A = a_1$, adding $(I_{a_1})^W$.

Given a certain instantiation Γ, e.g. $\neg I_{a_1}, \ldots \neg I_{b_2}$:

$$\mathrm{Pn}(\Gamma) = \sum_{\theta_{x|v} \sim \mathbf{x}} -\log \theta_{x|v}$$
$$= -\log \prod_{\theta_{x|v} \sim \mathbf{x}} \theta_{x|v} = -\log \Pr(\mathbf{x})$$

MaxSAT: Solving

Previously we've discussed methods of solving MaxSAT problems, such as searching.
MaxSAT could also be solved by compiling to DNNF and calculate the minCard.
An Example: (unweighted for simplicity)

$$\Delta : \underbrace{A \vee B}_{C_0}, \underbrace{\neg A \vee B}_{C_1}, \underbrace{\neg B}_{C_2}$$

- add selector variables: S_0, S_1, S_2

  $$\Delta' : \; A \vee B \vee S_0, \; \neg A \vee B \vee S_1, \; \neg B \vee S_2$$

  representing whether or not a clause is selected to be unsatisfiable / thrown away.
- assign weights:

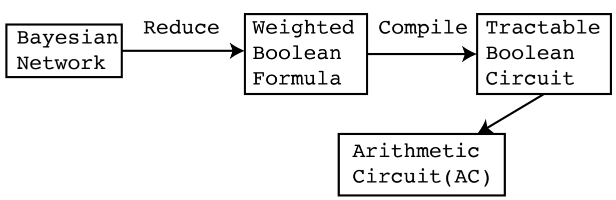
  $$\begin{cases} w(S_0) = 1, \; w(S_1) = 1, \; w(S_2) = 1 \\ w(\neg S_0) = 0, \; w(\neg S_1) = 0, \; w(\neg S_2) = 0 \\ w(A) = w(\neg A) = w(B) = w(\neg B) = 0 \end{cases}$$

- define cardinality: number of positive selector variables — computing minCard is the same with working on the **weights**
- compile Δ' into DNNF (hopefully)
- compute minCard, optimal solution minCard = 1 achieved when $S_0, \neg S_1, \neg S_2, \neg A, \neg B$; solution: $\neg A, \neg B$; satisfied clauses: C_1, C_2.

Factor v.s. Distribution

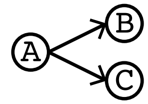
Factor sums up to anything;
Distribution sums up to 1.

Solving MAR via WMC



Reduction: using indicator and parameter variables.
More Reading: Modeling and Reasoning with Bayesian Networks

Solving MAR via WMC: Example



A	θ_A
a_1	0.1
a_2	0.9

A	B	$\theta_{B\|A}$
a_1	b_1	0.1
a_1	b_2	0.9
a_2	b_1	0.2
a_2	b_2	0.8

A	C	$\theta_{C\|A}$
a_1	c_1	0.1
a_1	c_2	0.9
a_2	c_1	0.2
a_2	c_2	0.8

- **Indicator Variables**: $I_{a_1}, I_{a_2}, I_{b_1}, I_{b_2}, I_{c_1}, I_{c_2}$
- **Parameter Variables**: $P_{a_1}, P_{a_2}, P_{b_1|a_1}, P_{b_2|a_1}, P_{b_1|a_2}, P_{b_2|a_2}, P_{c_1|a_1}, P_{c_2|a_1}, P_{c_1|a_2}, P_{c_2|a_2}$
- I_* and P_* are all **Boolean** variables.
- **Indicator Clauses**:

  $$\begin{cases} \text{A:} & I_{a_1} \vee I_{a_2}, \; \neg I_{a_1} \vee \neg I_{a_2} \\ \text{B:} & I_{b_1} \vee I_{b_2}, \; \neg I_{b_1} \vee \neg I_{b_2} \\ \text{C:} & I_{c_1} \vee I_{c_2}, \; \neg I_{c_1} \vee \neg I_{c_2} \end{cases}$$

- **Parameter Clauses**:

  $$\begin{cases} \text{A:} & I_{a_1} \iff P_{a_1}, \; I_{a_2} \iff P_{a_2} \\ \text{B:} & I_{a_1} \wedge I_{b_1} \iff P_{b_1|a_1}, \; I_{a_1} \wedge I_{b_2} \iff P_{b_2|a_1} \\ & I_{a_2} \wedge I_{b_1} \iff P_{b_1|a_2}, \; I_{a_2} \wedge I_{b_2} \iff P_{b_2|a_2} \\ \text{C:} & I_{a_1} \wedge I_{c_1} \iff P_{c_1|a_1}, \; I_{a_1} \wedge I_{c_2} \iff P_{c_2|a_1} \\ & I_{a_2} \wedge I_{c_1} \iff P_{c_1|a_2}, \; I_{a_2} \wedge I_{c_2} \iff P_{c_2|a_2} \end{cases}$$

  the rule is:

  $$I_{u_1} \wedge \cdots \wedge I_{u_m} \wedge I_x \iff P_{x|I_{u_1} \ldots I_{u_m}}$$

- Weights are defined as:

  $$\text{Wt}(I_x) = \text{Wt}(\neg I_x) = \text{Wt}(\neg P_{x|u}) = 1$$
  $$\text{Wt}(P_{x|u}) = \theta_{x|u}$$

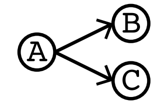
  e.g. $P_{b_2|a_2}$ has 0.8 weight.

$\mathcal{N}$: CNF encoding of BN $\Rightarrow \Delta_{\mathcal{N}}$
Any evidence $\mathbf{e} = e_1, \ldots e_k$:

$$\Pr(\mathbf{e}) = \text{WMC}(\Delta_{\mathcal{N}} \wedge I_{e_1} \ldots I_{e_k})$$

Network Instantiation: $(a_i b_j c_k)$:
e.g. $\text{Wt}(a_1 b_1 c_2) = .1 * .1 * .9 = .009$.

MAR as WMC: Example with Local Structure



A	θ_A
t	0.5
f	0.5

A	B	$\theta_{B\|A}$
t	t	1
t	f	0
f	t	0
f	f	1

A	C	$\theta_{C\|A}$
t	t	0.8
t	f	0.2
f	t	0.2
f	f	0.8

First we construct the clauses as before (this time denote e.g., $a_1 = a$ and $a_2 = \overline{a}$).
Local Structure: re-surfacing old concept; here parameter **values** matter.

- Zero Parameters (logical constraints): e.g.

  ~~$I_a \wedge I_{\overline{b}} \iff P_{\overline{b}|a}$~~ $\rightarrow \; \neg I_a \vee \neg I_{\overline{b}}$

- One Parameters (logical constraints): e.g.

  ~~$I_a \wedge I_b \iff P_{b|a}$~~

- Equal Parameters: e.g.

  $\begin{cases} \text{~~}I_a \wedge I_c\text{~~} \\ \text{~~}I_{\overline{a}} \wedge I_{\overline{c}}\text{~~} \end{cases}$ $\rightarrow \; (I_a \wedge I_c) \vee (I_{\overline{a}} \wedge I_{\overline{c}}) \iff P_1$

- Context-Specific Independence (CSI): independent only when considering some specific worlds

With local structure considered, the clauses:

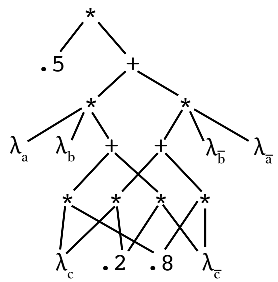
$$\begin{array}{ccc} I_a \vee I_{\overline{a}} & I_b \vee I_{\overline{b}} & I_c \vee I_{\overline{c}} \\ \neg I_a \vee \neg I_{\overline{a}} & \neg I_b \vee \neg I_{\overline{b}} & \neg I_c \vee \neg I_{\overline{c}} \\ & \neg I_a \vee \neg I_{\overline{b}} & \\ & \neg I_{\overline{a}} \vee \neg I_a & \end{array}$$

$$\begin{array}{rcll} (I_a \wedge I_c) \vee (I_{\overline{a}} \wedge I_{\overline{c}}) & \iff & P_1 & (0.8 \text{ prob}) \\ (I_a \wedge I_{\overline{c}}) \vee (I_{\overline{a}} \wedge I_c) & \iff & P_2 & (0.2 \text{ prob}) \\ I_a \vee I_{\overline{a}} & \iff & P_3 & (0.5 \text{ prob}) \end{array}$$

Could be compiled into **sd-DNNF**.
And we can build **AC** accordingly, by: (1) replacing I_x with λ_x (and $I_{\overline{x}}$ with $\lambda_{\overline{x}}$); (2) replacing P_y with θ_y; (3) replace **and** by $*$, **or** by $+$.
Evidence in AC: when there's no evidence, $\lambda_i = 1$; when there is an evidence, if compatible with it $\lambda_i = 1$, otherwise $\lambda_i = 0$. (e.g. given A: $\lambda_a = 1$, $\lambda_{\overline{a}} = 0$)

MAR as WMC: Example — AC

The AC generated from the previous example (considering local structure) is:



On AC we can do backpropagation.

$$\frac{\partial f}{\partial \lambda_x}(\mathbf{e}) = \Pr(x, \mathbf{e} - x)$$

$$\theta_{x|u}\frac{\partial f}{\partial \theta_{x|u}}(\mathbf{e}) = \Pr(x, u, \mathbf{e})$$

There are other possible reductions, such as minimizing the size of CNF, etc.

ACs with Factors

Motivation: avoid losing reference point etc. when **learning ACs from data**.
For instance, instead of listing $A, B, \Theta_{B|A}$ and use it, we list $A, B, f(A, B)$ where the f values are integers.
In the AC, because we use f instead of Θ, the values are **integers** as well.
We can build ACs to compute factor (f) in this way. (e.g. given instance A, B, compute $f(a, b)$ via the AC by setting $\lambda_a = \lambda_b = 1$ and $\lambda_{\bar{a}} = \lambda_{\bar{b}} = 0$)
Some of these ACs also computes:

- marginals: e.g. $f(a) = f(a, b) + f(a, \bar{b})$ can be computed via the AC setting $\lambda_a = \lambda_{\bar{b}} = \lambda_b = 1$ and $\lambda_{\bar{a}} = 0$.
- MPE: by replacing "+" with "max" in the AC.

Claim: If an AC:

1. computes a factor f,
2. and is **decomposable**, **deterministic** and **smooth**,

it computes marginals (2003) and MPE (2006).

Sum-Product Nets (SPN, 2011)

Claim: If an AC:

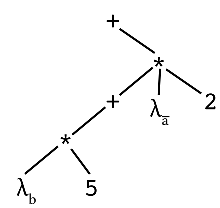
1. computes a factor f,
2. and is **decomposable** and **smooth**,

it computes marginals of f.
Known as **SPN**s (Sum-Product Nets). SPNs **can't** compute MPE in linear time.
Decomposable and smooth guarantee that sub-circuit term is a complete instantiation.
Determinism further guarantees one-to-one mapping between sub-circuit terms and complete instantiations.
An SPN that satisfies determinism is called **Selective-SPN**, and it computes MPE.

Parametric Completeness of Factors

Definition: Parameter Θ is complete for factor $f(\mathbf{x})$ iff for any instantiation $\mathbf{x}$, $f(\mathbf{x})$ can be expressed as a product of parameters in Θ.
Claim: The parameters of a Bayesian Network are complete for its factor.
Infer: When completeness of the parameters is guaranteed: $\exists$AC$(\mathbf{X}, \Theta)$ that is decomposable, deterministic and smooth.

Factor: Sub-circuit Term & Coefficient



e.g., the above sub-circuit:

$$\begin{cases} \text{term:} & \bar{a}b \\ \text{coefficient:} & 2 * 5 = 10 \end{cases}$$

An instantiation can have multiple sub-circuits; with the *same term*, but *different coefficients*. **Sum** the coefficients up to get the factor.

Finale: more topics

ACs:
- model-based supervised learning:
  in between AC-encoding with & without local structure; only part of the parameters (part of θ) are known and the rest to learn.
  - background knowledge (BK): (1) known parameters (2) functional dependencies (sometimes we know that $Y = f(X)$ but we don't know the identity of function f)
  - from compile model to **compile query**: e.g. evidence A,C, query B; AC's leaves: $\lambda_a, \lambda_{\bar{a}}, \lambda_c, \lambda_{\bar{c}}, \Theta$; output $P^*(b), P^*(\bar{b})$ can be trained from labeled data (GD etc.)
- tensor graphs
  new AC compilation algorithm
  key benefit: parallel
  - Structural Causal Models (SCMs): exogenous variables (distributions, e.g. U_x, it points to x), endogenous variables (functions, e.g. x, a node in a directed graph)

Solving PP^{PP}-complete problems with tractable circuits
- Maj-Maj-SAT is solvable in linear time (to the SDD size) if we can constrain its SDD (i.e. normalized for a constrained Vtree)
  - Vtree is x-constrained iff there's a node $\exists v$ that (1) appears on the right-most path (2) the set of variables **outside** v are equal to x.

Graph abstractions of KB
- primal, dual, incidence graphs; hyper-graph
- tree-width, cut-width, path-width

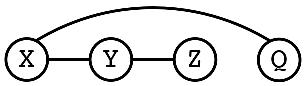
Auxiliary variables: basically, the idea is to add $X \iff \ell_1 \vee \ell_2$ where ℓ_1 and ℓ_2 are carefully-chosen literals.
- **Equivalent Modulo Forgetting** (**EMF**): A function $f(X)$ is EMF to function $g(X, Y)$ iff $f(X) = \exists Y g(X, Y)$.
- **Tseitin Transformation** (1968): convert Boolean formulas into CNF.

Graph Abstraction: Examples

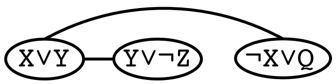
Given a CNF:

$$(X \vee Y) \wedge (Y \vee \neg Z) \wedge (\neg X \vee Q)$$

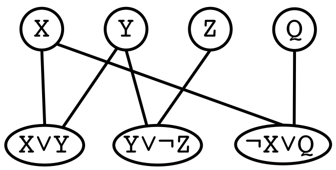
```
      1       2        3
CNF: (X∨Y)∧(Y∨¬Z)∧(¬X∨Q)
```



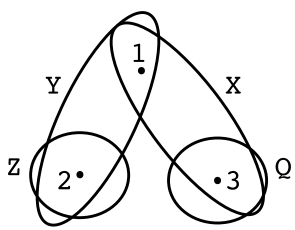
(a) primal graph



(b) dual graph



(c) incidence graph



(d) hypergraph

Graph Properties: Treewidth

Tree width of graph G: $\text{tw}(G)$ is the minimum width among all tree-decomposition of G. [a]

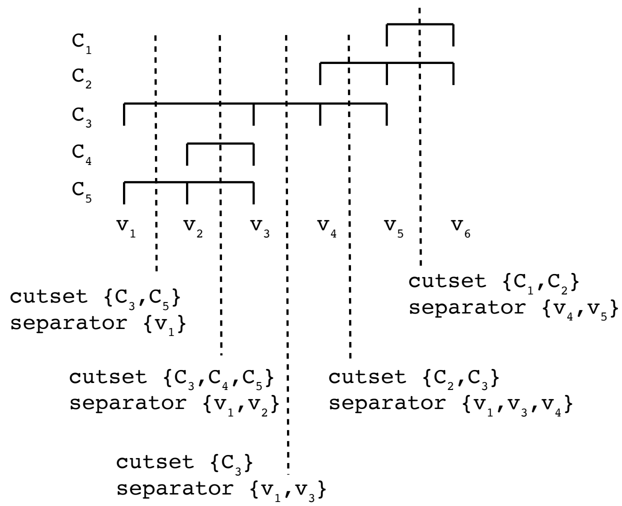
In many cases, good performance is guaranteed when there's a small treewidth.

[a] https://en.wikipedia.org/wiki/Treewidth

CNF Properties: Cutwidth and Pathwidth

Given the case:

$$\begin{array}{ll} C_1 & v_5 + v_6 \\ C_2 & v_4 + \neg v_5 + v_6 \\ C_3 & v_1 + v_3 + v_4 + v_5 \\ C_4 & v_2 + v_3 \\ C_5 & v_1 + v_2 + \neg v_3 \end{array}$$



Cutwidth and pathwidth are both influenced by variable ordering.

Cutwidth of a variable order: size of the largest **cutset**, e.g. 3 in this case. (cutset is the set of **clauses** that crosses a cut.)

Cutwidth of CNF: smallest cutwidth attained by any variable order.

Pathwidth of a variable order: size of the largest **separator**. e.g. 3 in this case. (separator is the set of **variables** that appear in the clauses within the cutset, and before the cut — according to the variable ordering.)

Pathwidth of CNF: smallest pathwidth attained by any variable order.

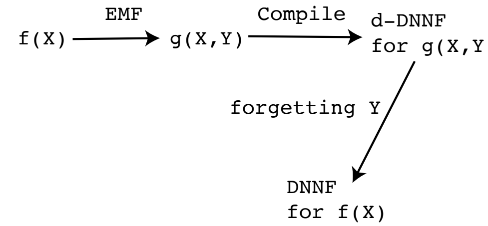
AC: Conclusions

Two fundamental **notations**:

1. Arithmetic Circuit (AC): indicator variables, constants, additions, multiplications
2. Evaluation AC (at evidence): set indicator → 1 if its subscript is consistent with evidence, otherwise 0.

Three fundamental **questions**: (1) reference factor $f(x)$? (2) marginal of factor? (3) MPE of factor?

Auxiliary Variables



There's no easy direct way from $f(X)$ to its DNNF. Sometimes $f_n(X)$ has exponential size when $g_n(X, Z)$ has polynomial size.

When adding auxiliary variables to Δ, we guarantee equal satisfiability.

An example:

$$\Delta = (A \vee D) \wedge (B \vee D) \wedge (C \vee D) \wedge (A \vee E) \wedge (B \vee E) \wedge (C \vee E)$$
$$\Sigma = (A \vee \neg X) \wedge (B \vee \neg X) \wedge (C \vee \neg X) \wedge (D \vee X) \wedge (E \vee X)$$

Here we have $\exists X \Sigma = \Delta$ by doing existential quantification (forgetting).

Extended Resolution: might reduce cost. (e.g. Pigeonhole: exponential to polynomial) e.g.:

resolution: (recall)

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

1.	$\{\neg A, C\}$	Δ
2.	$\{\neg B, C\}$	Δ
3.	$\{\neg C, D\}$	Δ
4.	$\{\neg D\}$	$\neg\alpha$
5.	$\{A\}$	$\neg\alpha$
6.	$\{\neg C\}$	3, 4
7.	$\{\neg A\}$	1, 6
8.	$\{\}$	5, 7

extension rule: (carefully choose literals ℓ_i and X is new/unseen to this CNF)

$$X \iff \ell_1 \vee \ell_2$$

which is equivalent with adding the following clauses:

$$\neg X \vee \ell_1 \vee \ell_2$$
$$X \vee \neg \ell_1$$
$$X \vee \neg \ell_2$$

Intuition: resolving multiple variables all at once.