CS264A Automated Reasoning Review Note
2020 Fall By Zhiping (Patricia) Xiao

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)

Semantic Properties
- Consistency: α is consistent when Mods(α) ≠ ∅
- Validity: α is valid when Mods(α) = W
- α is valid iff ¬α is inconsistent.
- α is consistent iff ¬α is invalid.

Existential Quantification Useful Equations
- α → β = ¬α ∨ β
- α → β = ¬β → ¬α
- ¬(α ∨ β) = ¬α ∧ ¬β
- ¬(α ∧ β) = ¬α ∨ ¬β
- γ ∧ (α ∨ β) = (γ ∧ α) ∨ (γ ∧ β)
- γ ∨ (α ∧ β) = (γ ∨ α) ∧ (γ ∨ β)

Models
Listing the 2ⁿ worlds wᵢ involving n variables, we have a truth table.
If sentence α is true at world ω, ω ⊨ α, we say:
- sentence α holds at world ω
- ω satisfies α
- ω entails α
otherwise ω ⊭ α.

Semantic Relationships
Equivalence: α and β are equivalent iff Mods(α) = Mods(β)
Mutually Exclusive: α and β are equivalent iff Mods(α ∧ β) = Mods(α) ∩ Mods(β) = ∅
Exhaustive: α and β are exhaustive iff Mods(α ∨ β) = Mods(α) ∪ Mods(β) = W
that is, when α ∨ β is valid.

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: Δ|P
  i.e. replacing all occurrences of P by true.
- condition Δ on ¬P: Δ|¬P
  i.e. replacing all occurrences of P by false.

Boolean’s/Shanoon’s Expansion:
Δ = (P ∧ (Δ|P)) ∨ (¬P ∧ (Δ|¬P))
it enables recursively solving logic, e.g. DPLL.
### Existential & Universal Qualification

**Existential Qualification:**
\[ \exists P \Delta = \Delta | P \lor \Delta | \neg P \]

**Universal Qualification:**
\[ \forall P \Delta = \Delta | P \land \Delta | \neg P \]

**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:
\[ \exists P \Delta = \Delta | P \lor \Delta | \neg P \]
doesn’t include \( P \).

Here we have an example: \( \Delta = \{ A \Rightarrow B, B \Rightarrow C \} \), then:
\[ \Delta = (\neg A \lor B) \land (\neg B \lor C) \]
\[ \Delta | B = C \]
\[ \Delta | \neg B = \neg A \]
\[ \therefore \exists E \Delta = \Delta | B \lor \Delta | \neg E = \neg A \lor C \]

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

We can safely remove \( P \) from \( \Delta \) when considering existential qualification. It is called:
- **forgetting** \( P \) from \( \Delta \)
- **projecting** \( P \) on all units / variables but \( P \)

### Resolution / Inference Rule

**Modus Ponens (MP):**
\[
\frac{\alpha, \alpha \Rightarrow \beta}{\beta}
\]

**Resolution:**
\[
\frac{\alpha \lor \beta, \neg \beta \lor \gamma}{\alpha \lor \gamma}
\]

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, \( \alpha \lor \gamma \) is called a “resolvent”. We can say it either way:
- resolve \( \alpha \lor \beta \) with \( \neg \beta \lor \gamma \)
- resolve over \( \beta \)
- do \( \beta \)-resolution

**Existential Quantification via Resolution**

1. **Turning KB \( \Delta \) into CNF.**
2. **simplifying KB \( \Delta \)**
3. **deduction (strategies of resolution, directed resolution)**

**Completeness of Resolution / Inference Rule**

We say rule \( R \) is complete, if \( \forall \alpha \), if \( \Delta \models \alpha \) then \( \Delta \vdash R \alpha \).

In other words, \( R \) is complete when it could “discover everything from \( \Delta \)”. Resolution / inference rule is NOT complete. A counter example is: \( \Delta = \{ A, B \}, \alpha = A \lor B \).

However, when applied to CNF, resolution is refutation complete. Which means that it is sufficient to discover any inconsistency.

### Unit Resolution

Unit resolution is a special case of resolution, where \( \min(|C_1|, |C_2|) = 1 \) 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

\( \Delta \models \alpha \) if \( \Delta \land \neg \alpha \) is inconsistent. (useful in proof)

- resolution finds contradiction on \( \Delta \land \neg \alpha \): \( \Delta \models \alpha \)
- resolution does not find any contradiction on \( \Delta \land \neg \alpha \): \( \Delta \not\models \alpha \)

### Clausal Form of CNF

CNF, the Conjunctive Normal Form, is a conjunction of clauses.
\[ \Delta = C_1 \land C_2 \land \ldots \]

written in clausal form as:
\[ \Delta = \{ C_1, C_2 \ldots \} \]

where each clause \( C_i \) is a disjunction of literals:
\[ C_i = l_{i1} \lor l_{i2} \lor l_{i3} \lor \ldots \]

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 \( (C_i \setminus \{ P \}) \cup (C_j \setminus \{ \neg P \}) \) (Notation: removing set \( \{ P \} \) from set \( C_i \) is written as \( C_i \setminus \{ P \} \))

If the clausal form of a CNF contains an empty clause \( (\exists i, C_i = \emptyset = \{\}) \), then it makes the CNF inconsistent / unsatisfiable.
Resolution Strategies: Linear Resolution

All the clauses that are originally included in CNF $\Delta$ 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: $\Delta = \{\neg A, C\}, \{\neg C, D\}, \{A\}, \{\neg C, \neg D\}$. 

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.

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 the clause.

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, $\{\}$ is the B-resolvent, $\{\}$ is the C-resolvent):

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

Primal Graph: Each node represents a variable $P$.

Given CNF $\Delta$, if there’s at least a clause $\exists C \in \Delta$ such that $l_i, l_j \in C$, then the corresponding places $P_i$ 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 $O(n \exp(w))$.

Utility of Using Graphs

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.

Utility of Using Graphs

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.

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.

SAT I

1. SAT-I ($\Delta, n, d$):
   1. If $d = n$:
      1. If $\Delta = {}$, return $\{\}$
      2. If $\Delta = \{\}$, return FAIL
   2. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   3. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   4. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   5. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   6. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   7. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   8. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   9. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$

SAT II

1. SAT-II ($\Delta, n, d$):
   1. If $d = n$:
      1. If $\Delta = {}$, return $\{\}$
      2. If $\Delta = \{\}$, return FAIL
   2. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   3. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   4. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   5. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   6. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   7. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$
   8. If $\Delta = \{\}, \neg \neg \Delta \neq FAIL$:
      1. return $\{\}$
      2. return $\{\}$

Mostly SAT I, plus early-stop.
Decision-Levels

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

1. Unit-Resolution (\( \Delta \)):
2. \( I = \) unit clauses in \( \Delta \)
3. If \( I = \{\} \): return \((I, \Delta)\)
4. \( \Gamma = \Delta | I \)
5. If \( \Gamma = \Delta \): return \((I, \Gamma)\)
6. return Unit-Resolution(\( \Gamma \))

Used in DPLL, at each node.

DPLL

01. DPLL (\( \Delta \)):
02. \((I, \Gamma) = \) Unit-Resolution(\( \Delta \))
03. If \( \Gamma = \{\} \): return \(I\)
04. If \( \{\} \in \Gamma \): return FAIL
05. choose a literal \( l \) in \( \Gamma \)
06. If \( L = \) DPLL(\( \Gamma \cup \{\{l\}\} \)) \( \neq \) FAIL:
07. return \(L \cup I\)
08. If \( L = \) DPLL(\( \Gamma \cup \{\neg l\} \)) \( \neq \) FAIL:
09. return \(L \cup I\)
10. 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 \( \Delta \):

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 \( \Delta \), which is used to generate a unit clause (make an implication).

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 \( \{\neg A, \neg Y, \neg Z\} \).

Implication Graphs: Cuts

1. \( \{A, B\} \)
2. \( \{B, C\} \)
3. \( \{\neg A, \neg X, Y\} \)
4. \( \{A, X, Y\} \)
5. \( \{\neg A, Y, Z\} \)
6. \( \{A, X, \neg Z\} \)
7. \( \{\neg A, \neg Y, \neg Z\} \)

DPLL+

01. DPLL+ (\( \Delta \)):
02. \( D \leftarrow \{\} \)
03. \( \Gamma \leftarrow \{\} \)
04. While true Do:
05. \((I, L) = \) Unit-Resolution(\( \Delta \land \Gamma \land D \))
06. If \( \{\} \in L \):
07. If \( D = \{\} \): return \(false\)
08. Else (backtrack to assertion level):
09. \( \alpha \leftarrow \) asserting clause
10. \( m \leftarrow \) AL(\( \alpha \))
11. \( D \leftarrow D + m + 1 \) decisions in \( D \)
12. \( \Gamma \leftarrow D \land \Gamma \land \{\} \)
13. Else:
14. find \( \ell \) where \( \{\ell\} \notin I \) and \( \{\neg \ell\} \notin I \)
15. If an \( \ell \) is found: \( D \leftarrow D; \ell \)
16. Else: return \(true\)

true if the CNF \( \Delta \) is satisfiable, otherwise false. \( \Gamma \) 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 \( \alpha \): find the first UIP.

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/imputation 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):

<table>
<thead>
<tr>
<th>Clause</th>
<th>Decision-Levels</th>
<th>Asserting?</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>{¬A, ¬X}</td>
<td>{0, 3}</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>{¬A, ¬Y}</td>
<td>{0, 3}</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>{¬A, ¬Y, ¬Z}</td>
<td>{0, 3, 3}</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

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**

**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 $\Delta$ is satisfiable iff $\Delta \models P$ is satisfiable or $\Delta \models \neg P$ 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.

**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 $\Delta$ from implication graphs), example:

<table>
<thead>
<tr>
<th>level</th>
<th>assignment</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$A$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$B$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$C$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$\neg A \lor \neg X \lor Y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\neg A \lor \neg Y \lor Z$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$Y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$Z$</td>
<td></td>
</tr>
</tbody>
</table>

And then learned clause $\neg A \lor \neg Y$ 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 $\Gamma$ generated from the SAT solver after running on $\Delta$ is a correct one.

- Will $\Delta \cup \Gamma$ produce any inconsistency?
  - Can use Unit-Resolution to check.

- $\Delta \land \neg \alpha_i$ is inconsistent for all clauses $\alpha_i$.
  - 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 $\text{CNF } \Delta = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, 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 $\Delta$ is UNSAT.

A **minimal UNSAT core** is an UNSAT core of $\Delta$ 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 $\omega$, if UNSAT, move to another world by flipping one variable in $\omega$ ($P$ 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 $m/n \rightarrow 0$ it is almost always SAT, $m/n \rightarrow \infty$ will make it almost always UNSAT).
  
In practice, the split point is $m/n \approx 4.24$.

Two ideas to generate random clauses:

- 1st idea: variable-length clauses
- 2nd 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 $\omega$)
  - 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 $\Delta$. (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 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:

$$
\begin{align*}
& x \lor \ell_1 \lor \ell_2 \lor \cdots \lor \ell_m \\
& \neg x \lor o_1 \lor o_2 \lor \cdots \lor o_n \\
& c_1 \\
& o_2
\end{align*}
$$

The results are the resolvent $c_1 \lor o_2$, and the compensation clauses:

1. $c_1 \lor c_2$
2. $x \lor c_1 \lor \neg o_1$
3. $x \lor c_1 \lor o_1 \lor \neg o_2$

\[ \vdots \]
4. $x \lor c_2 \lor \neg \ell_1$
5. $x \lor c_2 \lor \ell_1 \lor \ell_2$

\[ \vdots \]
6. $x \lor c_2 \lor \ell_1 \lor \ell_2 \lor \cdots \lor \ell_m$

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 1 (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 $\Delta$ as:

$\Delta : (I_1, 5) \land (I_2, 4) \land (I_3, 3) \land (I_4, 6) \land (\neg I_1 \lor \neg I_2, 2) \land (\neg I_1 \lor \neg I_3, 1) \land (\neg I_2 \lor \neg I_3, 1) \land (\neg I_1 \lor \neg I_4, \infty) \land (\neg I_2 \lor \neg I_4, \infty)$

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

<table>
<thead>
<tr>
<th>$I_1$</th>
<th>$I_2$</th>
<th>score</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\times$</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

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

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 $\Gamma$ (guaranteed to be satisfiable). $k$ is the minimum cost, each world satisfying $\Gamma$ achieves this cost.

Directed Max-SAT Resolution: Example

$\Delta = (\neg a \lor c) \land (a) \land (\neg a \lor b) \land (\neg b \lor \neg c)$

Variable order: $a, b, c$.

First resolve on $a$:

The final output is:

false, $[(\neg a \lor b \lor c), (a \lor \neg b \lor \neg c)]$

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

Beyond NP

Some problems, even those harder than NP problems can be reduced to logical reasoning.
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 \( A \):

![Bayesian Network Diagram]

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 \( \Theta \) 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 \( \neg P \); 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.

![Tractable Boolean Circuit Diagram]
Our goal is to transform in polytime while still keep the properties (e.g. DNNF still be DNNF).

Bounded conjunction / disjunction: KB ∆ is bounded on conjunction / disjunction operation. That is, taking any two formula from ∆, their conjunction / disjunction also belong to ∆.

\[ \Delta \land \neg A \land \neg B \land \ldots \land \neg B \]

?- can be done in polytime: can be done in polytime unless P = NP.
\(\nabla\): cannot be done in polytime even if P = NP.

Recall: \(\Delta = A \Rightarrow B, B \Rightarrow C, C \Rightarrow D\), existential qualifying B, C, is the same with forgetting B, C, in other words projecting on A, D.

In DNNF, we existential qualifying \(\{x_i\}_{i \in S}\) (\(S\) 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:

\[ \exists X_i (\alpha \lor \beta) = (\exists x_i \alpha) \lor (\exists x_i \beta) \]

\[ \exists X_i (\alpha \land \beta) = (\exists x_i \alpha) \land (\exists x_i \beta) \]
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.

\[
\min \text{Card}(X) = 0 \\
\min \text{Card}(\neg X) = 1 \\
\min \text{Card}(\text{true}) = 0 \\
\min \text{Card}(\text{false}) = \infty
\]

\[
\min \text{Card}(\alpha \lor \beta) = \min (\min \text{Card}(\alpha), \min \text{Card}(\beta)) \\
\min \text{Card}(\alpha \land \beta) = \min \text{Card}(\alpha) + \min \text{Card}(\beta)
\]

Again, the last rule holds only in DNNF. Filling the values into DNF 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.

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

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 → FBDD → d-DNNF → 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 \( \Delta \).

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.

d-DNNF

CT: model counting. MC(\( \alpha \)) = |\text{Mods}(\alpha)|
(decomposable) MC(\( \alpha \land \beta \)) = MC(\( \alpha \)) \times MC(\( \beta \))
(deterministic) MC(\( \alpha \lor \beta \)) = MC(\( \alpha \)) + MC(\( \beta \))

counting graph: replacing \( \lor \) with + and \( \land \) with * in a d-DNNF. Leaves: MC(\( X \)) = 1, MC(\( \neg X \)) = 1, MC(\text{true}) = 1, MC(\text{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. \( \alpha \land (A \lor \neg A) \)).

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.

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).

\[ \Delta = (x_i \land x_j) \lor \neg x_j \]
\[ f = x_i x_j + (1-x_j) \]

An example of a DNF

We express KB \( \Delta \) as function \( f \) by turning all \( \land \) into multiply and \( \lor \) 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:

\[ \text{Odd-parity function} \]
\[ f = (x_1 + x_2 + x_3 + x_4) \% 2 \]

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, \( \Delta \) has only one reduced OBDD.
OBDD: Subfunction and Graph Size

Considering the function \( f \) of a KB \( \Delta \), 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 instantiations).

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.

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>subfunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( x_5x_6 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x_4 + x_5x_6 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( x_5x_6 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( x_4 + x_5x_6 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( x_5x_6 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( x_4 + x_5x_6 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_3 )</th>
<th>( x_5 )</th>
<th>subfunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x_6 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( x_4 + x_6 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( x_2 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( x_2 + x_6 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( x_2 + x_4 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( x_2 + x_4 + x_6 )</td>
</tr>
</tbody>
</table>

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. \( O(1) \) time complexity.

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

SDD: Structured Decomposability

Decomposability:

\[
\begin{align*}
&f(ABCD) = \\
&f_1 \ [g_1(AB) \land h_1(CD)] \lor \\
&f_2 \ [g_2(A) \land h_2(BCD)] \lor \ldots
\end{align*}
\]

Structured Decomposability:

\[
\begin{align*}
&f(ABCD) = \\
&f_1 \ [g_1(AB) \land h_1(CD)] \lor \\
&f_2 \ [g_2(A) \land h_2(BCD)] \lor \ldots
\end{align*}
\]

SDD: Partitioned Determinism

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

\[
f(X, Y) = g_1(X)h_1(Y) + \cdots + g_n(X)h_n(Y)
\]

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

\[
\begin{align*}
\forall i, j & \ g_i \land g_j = \text{false} & & /\text{mutual exclusiveness} \\
& g_1 \lor \cdots \lor g_n = \text{true} & & /\text{exhaustive} \\
\forall i \ h_i & \neq \bot & & /\text{satisfiable}
\end{align*}
\]

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 \( ^{\text{right-linear}} \) vtree.

SDD is a strict superset of OBDD, maintaining key properties of OBDD \(^{a}\) 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.

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
\textbf{SDD: Compression} \newline
\((X, Y)\)-partition is \textbf{compressed} if there is no equal \textit{subs}. That is, \[h_i \neq h_i, \forall i \neq j\]

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

\textbf{Systematic Way of Building SDD: Example} \newline

\textit{Given:} \(f = (A \land B) \lor (B \land C) \lor (C \land D)\) \newline
\(X = \{A, B\}\) \newline
\(Y = \{C, D\}\)

Then we can have the sub-functions (\textit{subs}) as conditioned on the primes:

\begin{align*}
\text{prime} & \quad \text{sub} \\
A \land B & \quad \text{true} \\
A \land \overline{B} & \quad C \land D \\
\overline{A} \land B & \quad C \\
\overline{A} \land \overline{B} & \quad C \land D
\end{align*}

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

\begin{align*}
\text{prime} & \quad \text{sub} \\
A \land B & \quad \text{true} \\
\overline{A} \land B & \quad C \\
\overline{A} \land \overline{B} & \quad C \land D
\end{align*}

\[f = (A \land B) (\text{true}) + (\overline{A} \land B) (C) + (\overline{B}) (C \land D)\]

\textit{One possible} \textit{vtree} is:

\begin{itemize}
  \item \textbf{Bottom-Up Compilation (OBDD/SDD)}
  \begin{itemize}
    \item To compile a CNF:
      \begin{itemize}
        \item OBDD/SDD for literals
        \item disjoint literals to clause
        \item disjoint clauses to CNF
      \end{itemize}
    \item Similar to DNF
    \item Works for DNF
  \end{itemize}
\end{itemize}

An example of the bottom-up compilation:

\begin{itemize}
  \item \textbf{Canonicity in Compilation}
    \begin{itemize}
      \item OBDDs are canonical:
        \begin{itemize}
          \item fixed \textit{variable order} \rightarrow unique reduced OBDD
        \end{itemize}
      \item SDDs are canonical:
        \begin{itemize}
          \item fixed \textit{vtree} \rightarrow unique trimmed & compressed SDD
        \end{itemize}
    \end{itemize}
  \item \textbf{Minimizing OBDD Size}
    \begin{itemize}
      \item \(n\) variables lead to \(n!\) possibilities. We swap two adjacent variables to change \textit{variable order}. (This can be done easily, and could explore all possibilities.)
    \end{itemize}
\end{itemize}
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: \( n! \) (\( n: \# \text{vars} \))
- # dissections: \( C_{n-1} = \frac{(2(n-1))!}{n!(n-1)!} \) (Catalan number, # full binary trees with \( n \) leaves.)
- # vtrees over \( n \) variables:
  \[
  n! \times C_{n-1} = \frac{(2(n-1))!}{(n-1)!}
  \]

### Searching Over Vtrees

- A Double-search problem \( \{ \text{variable order} \} \) dissection
- Using tree operations \( \{ \text{rotating dissection} \} \) swapping

### Tree Rotations

It preserves variable order; enumerates all dissections.

### Searching Over Vtrees: in Practice

Vtree fragments: 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

\( ^* \) 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.

<table>
<thead>
<tr>
<th>year</th>
<th>model</th>
<th>comments</th>
<th>year</th>
<th>model</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>SDD</td>
<td>Tractable Boolean Circuit</td>
<td>2014</td>
<td>PSDD</td>
<td>P: Probabilistic</td>
</tr>
<tr>
<td>2018</td>
<td>Conditional PSDD</td>
<td>conditional probability</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:
  \[
  \Pr(X) = \int p_{X\mid Y}(x \mid y) \Pr(Y) \, dy
  \]
  computed by examining the conditional probability of \( X \) (some variables) given a particular value of \( Y \) (the remaining variables), and then averaging over the distribution of all \( Y \)s.

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

- Conditional Probability:
  \[
  \Pr(\alpha \mid \beta) = \frac{\Pr(\alpha, \beta)}{\Pr(\beta)}
  \]

To compute them efficiently/effectively, we can use circuits.

SDD (probability version): (\( X, Y \))-Partition,

\[
\begin{align*}
  f(X,Y) &= g_1(X)h_1(Y) + \cdots + g_n(X)h_n(Y) \\
  \forall i, g_i &\neq 0 \\
  \forall i \neq j, g_i g_j &= 0 \quad \text{(mutually exclusive)} \\
  g_1 + g_2 + \cdots + g_n &= 1 \quad \text{(exhaustive)}
\end{align*}
\]

where in this case \( g_i \) are the probabilities.

**Compressed** (\( \forall i \neq j, h_i \neq h_j \)) (\( X, Y \))-Partition of \( f \) is unique.

E.g. Given \( \alpha \), we have

\[
\Pr(\alpha) = \sum_{i=1}^n \Pr(\alpha \mid g_i) \Pr(g_i)
\]

**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:

<table>
<thead>
<tr>
<th>L</th>
<th>K</th>
<th>P</th>
<th>A</th>
<th>#samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

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. Normalizing under each OR-gate. (Sum up to 1.)

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

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

Likelihood

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

\[ L(\text{Data}|\theta) = \Pr_\theta(e_1) \cdot \Pr_\theta(e_2) \cdot \ldots \cdot \Pr_\theta(e_n) \]

Dataset Incompleteness

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

<table>
<thead>
<tr>
<th>Dataset Type</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Complete</td>
<td>Closed-form Solution</td>
</tr>
<tr>
<td>Classical Incomplete</td>
<td>EM Algorithm (on PSDD)</td>
</tr>
<tr>
<td>Non-classical Incomplete</td>
<td>N/A in ML</td>
</tr>
</tbody>
</table>

Non-classical Incomplete Dataset example:

\[ x_2 \land (y_2 \lor z_2), \quad x \Rightarrow y, \ldots \]

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.)

**Unique maximum-likelihood estimates.**

### PSDD Multiplication

factor \( \leftarrow \{ \text{distribution, normalization constant } \kappa \} \)

factor: worlds’ instantiation, and sample count (integer)

distribution: worlds’ instantiation, and probability

Consider the tables as matrices, then \( F = D \ast \kappa \).

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 mean that, when \( k_1 \ast PSDD_1 \times k_2 \ast PSDD_2 = k_3 \ast PSDD_3 \), \( k_1, k_2, k_3 \) have any correlation. (Can’t expect to have \( k_3 = k_1 \times k_2 \).)

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. \( PSDD_{all} = PSDD_A \ast PSDD_B \ast PSDD_{C|AB} \ast PSDD_D|B \ldots \)

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 \( \beta \)'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 \( \Pr(Y|X) \) need conditional vtrees. \( X, Y \) are sets of variables, \( X \) includes the conditions.

The conditional vtree must contain a node, with precisely the variables in \( X \) contained in the subtree beneath it. Then this node is called a \( X \)-node, denoted as \( \ast \) instead of \( \cdot \) when drawing the vtree. The \( X \)-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. \( \Delta \) is the knowledge base.

<table>
<thead>
<tr>
<th>Prime Implicate (PI)</th>
<th>Prime Implicant (IP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF no subsumed clauses</td>
<td>DNF no subsumed terms</td>
</tr>
<tr>
<td>Implicate ( c ) of ( \Delta: \Delta \vdash c )</td>
<td>Implicant ( t ) of ( \Delta: \Delta \vdash t )</td>
</tr>
<tr>
<td>Resolution ( \frac{\alpha \lor \beta}{\alpha \lor \beta \lor \gamma} )</td>
<td>Consensus ( \frac{\alpha \lor \beta \land \gamma}{\alpha \lor \beta \land \gamma} \lor \text{prime implicants of } \Delta )</td>
</tr>
</tbody>
</table>

To obtain PI/IP: Close \( \Delta \) Under Resolution / Consensus then drop subsumed clauses/terms.

Subsume = all-literals already contained:

- Clauses: \( c_1 \) subsumes \( c_2 \), for \( c_1 = A \lor \neg B, \) \( c_2 = A \lor \neg B \lor C, c_1 \vdash c_2 \).
- Terms: \( t_1 \) subsumes \( t_2 \), for \( t_1 = \neg A \land B, t_2 = \neg A \land B \land \neg C, t_2 \vdash t_1 \).

For PI, existential quantification, and CE (clausal entailment check), are easy.

Prime means a clause/term is not subsumed by any other clause/term.

Duality: \( \neg \alpha \) is a prime implicate of \( \neg \Delta \)
In a circuit, on each **edge** (connecting two gates) there is a signal (high or low), denoted as $X, Y, A, B, C, \ldots$, $(\alpha)$ 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.

$\Delta$ contains $A, B, C, \ldots ok1, ok2, \ldots$. Examples:

![Circuit Diagram](image)

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

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

Model-Based Diagnosis figure out what are the possible situations of **health variables** when given $\Delta$ and $\alpha$ (an observation, e.g. $\alpha_a = C, \alpha_b = \neg E$, etc.). $\Delta$ here is called a system, and $\alpha$ is **system observation**. For example: in case (a), if $\Delta \land \alpha \land ok1 \land ok2$ is satisfiable (using SAT solver) then health condition for $ok1 = t, ok2 = t$ 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:

<table>
<thead>
<tr>
<th>$ok1$</th>
<th>$ok2$</th>
<th>$ok3$</th>
<th>normal?</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>no</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>yes</td>
</tr>
<tr>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>no</td>
</tr>
<tr>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>yes</td>
</tr>
<tr>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>yes</td>
</tr>
<tr>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>yes</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>yes</td>
</tr>
</tbody>
</table>

Concluding all yes and simplify: $\neg ok3 \lor (\neg ok1 \land \neg ok2)$.
### 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.

### Reasoning about ML Systems: Overview

<table>
<thead>
<tr>
<th>Queries</th>
<th>Explanation, Robustness, Verification, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML Systems</td>
<td>Neural Networks, Graphical Models, Random Forests, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tractable Circuits</th>
<th>OBDD, SDD, DNNF, etc.</th>
</tr>
</thead>
</table>

For more: [http://reasoning.cs.ucla.edu/xai/](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**:

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

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 I/O of ML Systems

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**: $e_1, \ldots, e_n = e$
- **Class Posterior**: (note that) $\Pr(C|\beta) = \Pr(C|\alpha) \frac{\Pr(\alpha|\beta)}{\Pr(\beta)}$

$$\Pr(c|e_1, \ldots, e_n) = \frac{\Pr(e_1, \ldots, e_n|c) \Pr(c)}{\Pr(e_1, \ldots, e_n)} = \frac{\Pr(e_1|c) \ldots \Pr(e_n|c) \Pr(c)}{\Pr(e_1, \ldots, e_n)}$$
Naive Bayes: CPT
A Bayesian Network has conditional probability tables (CPT) at each of its node.

\[
\begin{array}{c|c}
C & \Theta_C \\
\hline
c_1 & \theta_{c_1} \text{ (e.g., 0.1)} \\
\vdots \\
c_k & \theta_{c_k} \text{ (e.g., 0.2)} \\
\end{array}
\]

where \( \forall i \in [0,1], \sum_{i=1}^{k} \theta_{c_i} = 1 \).

And at node \( E_j \), the CPT:

\[
\begin{array}{c|c|c}
C & E_j & \Theta_{E_j|C} \\
\hline
c_1 & e_{j,1} & \theta_{e_{j,1}|c_1} \text{ (e.g., 0.01)} \\
c_1 & e_{j,2} & \theta_{e_{j,2}|c_1} \text{ (e.g., 0.03)} \\
\vdots \\
c_1 & e_{j,q} & \theta_{e_{j,q}|c_1} \text{ (e.g., 0.1)} \\
\vdots \\
c_k & e_{j,q} & \theta_{e_{j,q}|c_k} \text{ (e.g., 0.02)} \\
\end{array}
\]

where \( \forall i,j,x, \theta_{e_{j,x}|c_i} \in [0,1], \sum_{x=1}^{q} \theta_{e_{j,x}|c_i} = 1 \).

Under a condition, the marginal probability is 1.

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: Naive Bayesian Network has threshold \( T \) and

\[
\begin{align*}
\text{prior log-odds} & = \log(\frac{\text{Pr}(C=c_i|E_j=e_{j,x})}{\text{Pr}(C=c_j|E_j=e_{j,x})}) \\
\text{conditional odds} & = \text{max} \{ \log(\text{Pr}(C=c_i|E_j=e_{j,x})), \log(\text{Pr}(C=c_j|E_j=e_{j,x})) \}
\end{align*}
\]

where \( \forall i,j,x \in [0,1], \sum_{x=1}^{q} \theta_{e_{j,x}|c_i} = 1 \).

Under a condition, the marginal probability is 1.

For different conditions, different conditional probabilities, sharing the same sub-classifier.

Application: Solving MPE & MAR
MPE: most probable explanation
\( \rightarrow \) NP-complete
\( \rightarrow \) probabilistic reasoning, find the world with the largest probability
\( \rightarrow \) solved by weighted MAXSAT
\( \rightarrow \) compile to DNNF

MAR: marginal probability
\( \rightarrow \) PP-complete
\( \rightarrow \) sum of all worlds’ probabilities who satisfy certain conditions
\( \rightarrow \) solved by WMC (weighted model counting)
\( \rightarrow \) compile to d-DNNF

conditional version: work on “shrunk table” where some worlds are removed

Odds v.s. Probability

Probability: \( \text{Pr}(c) \) (chance to happen, \([0,1]\])

Odds: \( \text{O}(c) = \frac{\text{Pr}(c)}{\text{Pr}(\neg c)} \) (conditional odds)

\[
\log \text{O}(c|e) = \log \left( \frac{\text{Pr}(c|e)}{\text{Pr}(\neg c|e)} \right)
\]

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

\[
\text{O}(c|e) = \log \left[ \frac{\text{Pr}(c|e)}{\text{Pr}(\neg c|e)} \right] = \log(\frac{\text{Pr}(c|e)}{\text{Pr}(\neg c|e)})
\]

Solving MPE via MaxSAT
Input: weighted CNF = \( \alpha_1, \ldots, \alpha_n \) (with weights \( w_1, \ldots, w_n \))

\[
- (x \vee \neg y \vee \neg z)^3, \quad (\neg x)^{10.1}, \quad (y, z)^{2.5}
\]

\( \rightarrow \) 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

\( \text{Wt} \) and \( \text{Pn} \) find variable assignment with the highest weight / least penalty

\[
\begin{align*}
\text{Wt} = & \ \text{weight}(x_1, \ldots, x_n) = \sum_{x_1, \ldots, x_n \models \alpha_i} w_i \\
\text{Pn} = & \ \text{penalty}(x_1, \ldots, x_n) = \sum_{x_1, \ldots, x_n \not\models \alpha_i} w_i \\
\text{Wt}(x_1, \ldots, x_n) + \text{Pn}(x_1, \ldots, x_n) = & \ \Psi \quad (constant)
\end{align*}
\]

Solving MPE via MaxSAT: Example

Given a Bayesian Network (with CPT listed):

\[
\begin{array}{c|c|c|c|c|c}
\hline
A & B & \theta_{B|A} \\
\hline
a_1 & b_1 & 0.2 \\
\hline
a_2 & b_1 & 1 \\
\hline
a_3 & b_1 & 0.6 \\
\hline
a_3 & b_2 & 0.4 \\
\end{array}
\]

- Indicator variables:
  \( \rightarrow \) from \( A \) (values \( a_1, a_2, a_3 \)): \( I_{a_1}, I_{a_2}, I_{a_3} \)
  \( \rightarrow \) from \( B \) (values \( b_1, b_2 \)): \( I_{b_1}, I_{b_2} \)

- Indicator Clauses:

\[
\begin{align*}
A \left( I_{a_1} \lor I_{a_2} \lor I_{a_3} \right)^W \\
\neg I_{a_1} \land \neg I_{a_2} \land I_{a_3}^W \\
\neg I_{a_1} \land \neg I_{a_2} \land \neg I_{a_3}^W \\
\neg I_{a_2} \land \neg I_{a_3}^W
\end{align*}
\]

- Parameter Clauses: \( (\sum \# \text{ rows in CPTs}) \)

\[
\begin{align*}
\neg I_{a_1} \lor \neg I_{b_1} \text{ } & \text{log}(2) \\
\neg I_{a_2} \lor \neg I_{b_1} \text{ } & \text{log}(8) \\
\neg I_{a_2} \lor \neg I_{b_2} \text{ } & \text{log}(1) \\
\neg I_{a_3} \lor \neg I_{b_1} \text{ } & \text{log}(4) \\
\neg I_{a_3} \lor \neg I_{b_2} \text{ } & \text{log}(0)
\end{align*}
\]

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 \( \Gamma \), e.g. \( \neg I_{a_1}, \ldots, \neg I_{b_2} \):

\[
\text{Pn}(\Gamma) = \sum_{\theta_{x|v} \models \infty} - \text{log} \theta_{x|v} = - \text{log} \prod_{\theta_{x|v} \models \infty} \theta_{x|v}
\]
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)
\[
\Delta : A \lor B, \neg A \lor B, \neg B
\]
\[
\Delta' : A \lor B \lor S_0, \neg A \lor B \lor S_1, \neg B \lor S_2
\]

- add selector variables: \(S_0, S_1, S_2\)
- represent whether or not a clause is selected to be unsatisfiable / thrown away.

- assign weights:
\[
\begin{align*}
    w(S_0) &= 1, \quad w(S_1) = 1, \quad w(S_2) = 1 \\
    w(\neg S_0) &= 0, \quad w(\neg S_1) = 0, \quad w(\neg S_2) = 0 \\
    w(A) &= w(\neg A) = w(B) = w(\neg B) = 0
\end{align*}
\]

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

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

Solving MAR via WMC

![Diagram](image)

| \(A\) | \(\theta_A\) | \(\theta_{B|A}\) | \(\theta_{C|A}\) |
|-------|------------|----------------|------------|
| \(a_1\) | 0.1        | 0.1            | 0.1        |
| \(a_2\) | 0.9        | 0.2            | 0.8        |

- **Indicator Variables:** \(I_{a_1}, I_{a_2}, I_{b_1}, I_{b_2}, I_{c_1}, I_{c_2}\)
- **Parameter Variables:** \(P_{a_1}, P_{a_2}, P_{b_1|a_1}, P_{b_2|a_1}, P_{b_1|a_2}, P_{b_2|a_2}, P_{c_1|a_1}, P_{c_2|a_1}, P_{c_1|a_2}, P_{c_2|a_2}\)
- \(I_s\) and \(P_s\) are all **Boolean** variables.

- **Indicator Clauses:**
\[
\begin{align*}
    A: \quad & I_{a_1} \lor I_{a_2}, \neg I_{a_1} \lor \neg I_{a_2} \\
    B: \quad & I_{b_1} \lor I_{b_2}, \neg I_{b_1} \lor \neg I_{b_2} \\
    C: \quad & I_{c_1} \lor I_{c_2}, \neg I_{c_1} \lor \neg I_{c_2}
\end{align*}
\]

- **Parameter Clauses:**
\[
\begin{align*}
    A: \quad & I_{a_1} \iff P_{b_1}, I_{a_2} \iff P_{a_2} \\
    B: \quad & I_{a_2} \land I_{b_1} \iff P_{b_1|a_1}, I_{a_1} \land I_{b_2} \iff P_{b_2|a_1} \\
    C: \quad & I_{a_1} \land I_{c_1} \iff P_{c_1|a_1}, I_{a_2} \land I_{c_2} \iff P_{c_2|a_1}, \\
    & I_{a_2} \land I_{c_1} \iff P_{c_1|a_2}, I_{a_2} \land I_{c_2} \iff P_{c_2|a_2}
\end{align*}
\]

the rule is:
\[
I_{a_1} \land \cdots \land I_{a_m} \land I_{b} \iff P_{x|a_1 \cdots a_m}
\]

- Weights are defined as:
\[
Wt(I_x) = Wt(\neg I_x) = Wt(\neg P_{x|a}) = 1
\]
\[
Wt(P_{x|a}) = \theta_{x|a} \\
e.g. P_{b_1|a_1} \text{ has 0.8 weight.}
\]

\(N\): CNF encoding of BN \(\Rightarrow \Delta_N\)

- Any evidence \(e = e_1 \cdots e_k\):
\[
Pr(e) = \text{WMC}(\Delta_N \land I_{e_1} \land \cdots I_{e_k})
\]

Network Instantiation: \((a_1 b_1 c_1)\):
e.g. \(Wt(a_1 b_1 c_1) = .1 \ast .1 \ast .9 = .009\).

MAR as WMC: Example with Local Structure

![Diagram](image)

| \(A\) | \(\theta_A\) | \(\theta_{B|A}\) | \(\theta_{C|A}\) |
|-------|------------|----------------|------------|
| \(t\) | 0.5        | 1              | 0.8        |
| \(f\) | 0.5        | 0              | 0.2        |

First we construct the clauses as before (this time denote e.g., \(a_1 = a\) and \(a_2 = \pi\)).

Local Structure: re-surfacing old concept; here parameter values matter.

- Zero Parameters (logical constraints): e.g.
\[
\begin{align*}
    I_a \land I_{\pi} & \iff P_{b_{a\pi}} \\
    \neg I_a \lor \neg I_{\pi}
\end{align*}
\]

- One Parameters (logical constraints): e.g.
\[
\begin{align*}
    I_a \land I_{\pi} & \iff P_{b_{a\pi}} \\
    \neg I_a \lor \neg I_{\pi}
\end{align*}
\]

- Equal Parameters: e.g.
\[
\begin{align*}
    I_a \land I_{\pi} & \iff P_{b_{a\pi}} \\
    \neg I_a \lor \neg I_{\pi}
\end{align*}
\]

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

With local structure considered, the clauses:
\[
\begin{align*}
    I_a \land I_{\pi} & \iff P_{b_{a\pi}} (0.8 \text{ prob}) \\
    \neg I_a \lor \neg I_{\pi}
\end{align*}
\]

Could be compiled into sd-DNNF.

And we can build AC accordingly, by: (1) replacing \(I_x\) with \(\lambda_x\) (and \(I_{\pi}\) with \(\lambda_{\pi}\)); (2) replacing \(P_y\) with \(\theta_y\); (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 \(A\); \(\lambda_1 = 1, \lambda_{\pi} = 0\))
The AC generated from the previous example (considering local structure) is:

\[
\begin{array}{c}
\lambda_a \\
+ \\
\lambda_b \\
+ \\
\lambda_c \\
+ \\
\lambda_d \\
+ \\
\lambda_e \\
\end{array}
\]

On AC we can do backpropagation.

\[
\frac{\partial f}{\partial \lambda_x}(e) = \Pr(x, e - x)
\]

\[
\theta_x|u \frac{\partial f}{\partial \theta_x|u}(e) = \Pr(x, u, e)
\]

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 \( \Theta \), 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 \( \lambda_\pi = \lambda_\tau = 0 \))

**Some** of these ACs also computes:

- marginals: e.g. \( f(a) = f(a, b) + f(a, b) \) can be computed via the AC setting \( \lambda_a = \lambda_\pi = \lambda_b = 1 \) and \( \lambda_\tau = 0 \).
- MPE: by replacing “+” with “max” in the AC.

**Claim:** If an AC:

1. computes a factor \( f \),
2. and is decomposable and smooth,


**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 SPNs (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 \( \Theta \) is complete for factor \( f(x) \) iff for any instantiation \( x, f(x) \) can be expressed as a product of parameters in \( \Theta \).

**Claim:** The parameters of a Bayesian Network are complete for its factor.

**Infer:** When completeness of the parameters is guaranteed: \( \exists AC(X, \Theta) \) that is decomposable, deterministic and smooth.

**Factor: Sub-circuit Term & Coefficient**

\[
\begin{array}{c}
\lambda_a \\
+ \\
\lambda_b \\
+ \\
\lambda_c \\
+ \\
\lambda_d \\
+ \\
\lambda_e \\
\end{array}
\]

e.g., the above sub-circuit:

\[
\begin{align*}
\text{term:} & \quad \pi b \\
\text{coefficient:} & \quad 2 \times 5 = 10
\end{align*}
\]

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 \( \Theta \)) 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: \( \lambda_a, \lambda_B, \lambda_c, \lambda_\pi, \Theta \); output \( P^*(b), P^*(B) \) 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-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 \( X \leftrightarrow \ell_1 \lor \ell_2 \) where \( \ell_1 \) and \( \ell_2 \) 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.
Given a CNF:

\[(X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)\]

**Graph Abstraction: Examples**

- **(a) primal graph**
- **(b) dual graph**
- **(c) incidence graph**
- **(d) hypergraph**

**Graph Properties: Treewidth**

Tree width of graph \(G\): \(tw(G)\) is the minimum width among all tree-decomposition of \(G\). In many cases, good performance is guaranteed when there’s a small treewidth.

**CNF Properties: Cutwidth and Pathwidth**

Given the case:

\[
\begin{align*}
C_1 & : v_5 + v_6 \\
C_2 & : v_4 + v_5 + v_6 \\
C_3 & : v_1 + v_3 + v_4 + v_5 \\
C_4 & : v_2 + v_3 \\
C_5 & : v_1 + v_2 + v_3 \\
\end{align*}
\]

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 \(\rightarrow 1\) 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.

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

\[
\frac{X \lor \alpha, \neg X \lor \beta}{\alpha \lor \beta}
\]

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

**Extension rule**: (carefully choose literals \(\ell_{i}\) and \(X\) is new/unseen to this CNF)

\[
X \leftrightarrow \ell_{1} \lor \ell_{2}
\]

which is equivalent with adding the following clauses:

\[
\begin{align*}
\neg X \lor \ell_{1} \lor \ell_{2} \\
X \lor \neg \ell_{1} \\
X \lor \neg \ell_{2}
\end{align*}
\]

**Intuition**: resolving multiple variables all at once.