MRFs, Inference Properties, Factor Structure
Part I:
Graphical Model Variants and Inference Properties
Directed Graphical Models

Bayesian Network:

- compact (factored) specification of joint probability
- e.g., have binary variables B, F, A, H, P:

\[ P(B,F,A,H,P) = P(H|B,F) \cdot P(P|F,A) \cdot P(B) \cdot P(F) \cdot P(A) \]
Undirected Graphical Models

- Markov Random Fields (MRFs)

Variables take on possible configurations

Factors denote “compatibility” of configurations

\[ P(V_1, V_2, V_3, V_4) = \frac{1}{Z} F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4) \]

- Conditional MRFs (CRFs)

\[ P(V_1, V_2|V_3, V_4) = \frac{1}{Z(V_3, V_4)} F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4) \]

Note: representation above is **Factor Graph** (FG), which works for directed models as well. For FGs of directed models, what conditions also hold?
Remark: Inference in Undirected Models

- (VE) Inference same as for BN

\[ P(V_1|V_3) = \frac{\sum_{V_2} \sum_{V_4} F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4)}{\sum_{V_1} \sum_{V_2} \sum_{V_4} F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4)} \]

- Query example on previous MRF

- Previous CRF

\[ P(V_1|V_3, V_4) = \frac{\sum_{V_2} F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4)}{\sum_{V_1} \sum_{V_2} \sum_{V_4} Z(V_3, V_4)F(V_1, V_2)F(V_2, V_4)F(V_1, V_3)F(V_3, V_4)} \]
Dynamical Models & Influence Diagrams

• Dynamic Bayes Nets (DBNs) ...
  – Represent state @ times t, t+1
    • Assume stationary distribution

• Influence Diagrams (IDs)...
  – Action nodes [squares]
    • Not random variables
    • Rather “controlled” variables
  – Utility nodes <diamonds>
    • A utility conditioned on state, e.g.
      \[ U(X_1',X_2') = \text{if } (X_1' = X_2') \text{ then 10 else 0} \]
Remark: Inference in Bayes Nets vs. IDs

• Probabilistic Queries in Bayesian networks:
  • Want $P(\text{Query}|\text{Evidence})$, e.g.

  \[
  P(E|X) = \frac{P(E,X)}{P(X)} \propto \sum_A \sum_B P(A,B,E,X) = \sum_A \sum_B P(A|B,E) P(X|B) P(E) P(X)
  \]

• Maximizing Exp. Utility in Influence Diagrams:
  • Want optimal action $a^* = \arg\max_a E[U|A=a,...]$, e.g.

  \[
  a^* = \arg\max_a E_X[U(X')|A=a,X=x] = \arg\max_a \sum_{X'} U(X')P(X'|A=a,X=x)
  \]

Note: $\sum_a, \max_a, \arg\max_a$, are all compatible with variable elimination
Recall Marginalization ($\sum_b$)

- Marginalization

$$\sum_b P(A, b) = P(A)$$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$Pr$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>.1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>.3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>.4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A$</th>
<th>$Pr$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.4</td>
</tr>
<tr>
<td>1</td>
<td>.6</td>
</tr>
</tbody>
</table>
Similar for Maximization \((\max_a, \arg\max_a)\)

- Maximization

\[
\max_b \ P(A, b) = P(A)
\]

\[
\begin{array}{|c|c|c|}
\hline
A & B & Pr \\
\hline
0 & 0 & .1 \\
0 & 1 & .3 \\
1 & 0 & .4 \\
1 & 1 & .2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
A & Pr \\
\hline
0 & .3 \ (B=1) \\
1 & .4 \ (B=0) \\
\hline
\end{array}
\]

Note: can track \(\arg\max_b\) while computing \(\max_b\)
Associative-Commutative Semirings

• VE just exploits the *reverse distributive* property
  – The reverse distributive property holds for any two operators that form an AC Semiring
    • Examples: $\Sigma \Pi$; $\max \Pi$; $\max \Sigma$

• So efficient VE holds for
  – Marginal queries ($\Sigma \Pi$)
    • $\Sigma_X C F(X) = C \Sigma_X F(X)$
  – MPE queries over $\Pi$ or $\Sigma$ ($\max \Pi$; $\max \Sigma$)
    • $\max_X C F(X) = C \max_X F(X)$
    • $\max_X C + F(X) = C + \max_X F(X)$
  – MAP ($\max \Sigma \Pi$), provided that $\max$, $\Sigma$ not swapped
Treewidth (TW)

• Refers to clique tree compilation of graphical model \textit{given variable order}
  – Suffices to know that TW corresponds to largest factor produced during VE minus 1 (for variable order)

• Exact inference is $O(n^{2^{TW}})$ for graphical models with n binary random variables
  – TW relative to variable order
    • All exact inference needs some variable order
  – Should be obvious why this holds for VE

Note: $O$ and not $\theta$ or $\Omega$
Part II: Structured Factors

(how to avoid exponential blowup of factor representation vs. # of variables)
Noisy-Or

• One way to deal with large number of parents $X_1, \ldots, X_n$ that independently cause $Y$ with only $n+1$ parameters $\lambda_i$

$$P(Y=\text{true}|X_1, \ldots, X_n) = 1 - (1 - \lambda_0) \prod_{i \in \{X_i = \text{true}\}} (1 - \lambda_i)$$

- $\lambda_0$ is base probability of $Y=\text{true}$ if no $X_i=\text{true}$
- $\lambda_i$ is probability $X_i=\text{true}$ “causes” $Y=\text{true}$
- Reduces to OR if $\lambda_0=0$ (default off) and $\lambda_i=1$ (deterministic causes)

• Caveat: VE cannot directly exploit Noisy-Or
  - Unless introduce extra latent variables to encode, or
  - Represent with Affine ADDs (AADDs)… coming up
General class of factors that work well with VE...
What’s needed for exact inference?

- **Observation 1**: all discrete functions *can* be tables

\[
P(A,B) = 
\begin{array}{ccc}
A & B & Pr \\
0 & 0 & .1 \\
0 & 1 & .3 \\
1 & 0 & .4 \\
1 & 1 & .2 \\
\end{array}
\]

- **Observation 2**: all operations computable in closed-form
  - \( f_1 + f_2 \)
  - \( f_1 \cdot f_2 \)
  - \( \sum_x f(x) \)
  - (arg)\text{max}_x f(x), (arg)\text{min}_x f(x)

Are there any other data structures as alternatives to tables that efficiently support these operations?
Can we do better than tables?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>( F(a,b,c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Function Representation (Trees)

• How about a tree? Sure, can simplify.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>F(a,b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Context-specific independence!
Function Representation (ADDs)

- Why not a directed acyclic graph (DAG)?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>F(a,b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Exploits context-specific independence (CSI) (e.g. $c \perp b \mid a=\text{true}$) and shared substructure.
Function Representation (ADDs)

- Why not a directed acyclic graph (DAG)?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>F(a,b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Exploits context-specific independence (CSI) (e.g. \( c \perp b \mid a = \text{true} \)) and shared substructure.
Trees vs. ADDs

- Trees can compactly represent AND / OR
  - But not XOR (linear as ADD, exponential as tree)
  - Why? Trees must represent every path
Binary Operations (ADDs)

- Why do we order variable tests?
- Enables us to do efficient binary operations…

Result: ADD operations can avoid state enumeration
Summary

• We need $B^n \rightarrow B / Z / R$
  – We need compact representations
  – We need efficient operations

→ DDs are a promising candidate

• Great, tell me all about DDs…
  – OK 😊

Not claiming DDs solve all problems… but often better than tabular approach
Decision Diagrams: Reduce

(how to build canonical DDs)
How to Reduce Ordered Tree to ADD?

• Recursively build bottom up
  – Hash terminal nodes $R \rightarrow ID$
    • leaf cache
  – Hash non-terminal functions $(v, ID_0, ID_1) \rightarrow ID$
    • special case: $(v, ID, ID) \rightarrow ID$
    • others: keep in (reduce) cache
GetNode

- Removes redundant branches
- Maintains cache of internal nodes

Algorithm 1: \( \text{GetNode}(v, F_h, F_l) \rightarrow F_r \)

input : \( v, F_h, F_l \) : Var and node ids for high/low branches
output: \( F_r \) : Return values for offset, multiplier, and canonical node id

\[
\text{begin} \\
\quad \text{// If branches redundant, return child} \\
\quad \text{if } (F_l = F_h) \text{ then} \\
\quad \quad \text{return } F_l; \\
\quad \text{// Make new node if not in cache} \\
\quad \text{if } (\langle v, F_h, F_l \rangle \rightarrow \text{id is not in node cache}) \text{ then} \\
\quad \quad \text{id} := \text{currently unallocated id;} \\
\quad \quad \text{insert } \langle v, F_h, F_l \rangle \rightarrow \text{id in cache;} \\
\quad \text{// Return the cached, canonical node} \\
\quad \text{return id;} \\
\text{end}
\]
Algorithm 1: Reduce($F$) \rightarrow $F_r$

input : $F$ : Node id
output: $F_r$ : Canonical node id for reduced ADD

begin

// Check for terminal node
if (F is terminal node) then
   return canonical terminal node for value of $F$;

// Check reduce cache
if (F \rightarrow $F_r$ is not in reduce cache) then
   // Not in cache, so recurse
   $F_h := \text{Reduce}(F_h)$;
   $F_l := \text{Reduce}(F_l)$;

   // Retrieve canonical form
   $F_r := \text{GetNode}(F^{\text{var}}, F_h, F_l)$;

   // Put in cache
   insert $F \rightarrow F_r$ in reduce cache;

// Return canonical reduced node
return $F_r$;

end
Reduce Complexity

• Linear in size of input
  – Input can be tree or DAG

• Because of caching
  – Explores each node once
  – Does not need to explore all branches
Canonicity of ADDs via Reduce

- Claim: *if two functions are identical, Reduce will assign both functions same ID*

- By induction on var order
  - **Base case:**
    - Canonical for 0 vars: terminal nodes
  - **Inductive:**
    - Assume canonical for k-1 vars
    - GetNode result canonical for k\textsuperscript{th} var
      (only one way to represent)
Impact of Variable Orderings

- **Good orders can matter**

- **Good orders typically have related vars together**
  - e.g., low tree-width order in transition graphical model

Graph-Based Algorithms for Boolean Function Manipulation
From this point on, slides for information only – not tested material
In-diagram Reordering

• Rudell’s sifting algorithm
  – Global reordering as pairwise swapping
  – Only need to redirect arcs
    • Helps to use pointers
      → then don’t need to redirect parents, e.g.,

![Diagram](image-url)

Can also do reorder using Apply... later
Beyond Binary Variables

• Multivalued (MV-)DDs
  – A DD with multivalued variables
  – straightforward k-branch extension
  – e.g., k=6

– Works for ADD extensions as well
Decision Diagrams: Apply

(how to do efficient operations on DDs)
Recap

• Recall the Apply recursion

Need to handle base cases

Need to handle recursive cases

Result: ADD operations can avoid state enumeration
Apply Recursion

• Need to compute $F_1 \ op \ F_2$
  – e.g., $op \in \{\oplus, \otimes, \land, \lor\}$

• Case 1: $F_1$ & $F_2$ match vars

  $F_h = \text{Apply}(F_{1,h}, F_{2,h}, op)$
  $F_l = \text{Apply}(F_{1,l}, F_{2,l}, op)$
  $F_r = \text{GetNode}(F_{1,\text{var}}, F_h, F_l)$
Apply Recursion

• Need to compute $F_1 \ op \ F_2$
  - e.g., $op \in \{\oplus, \otimes, \land, \lor\}$

• Case 2: Non-matching var: $v_1 \not< v_2$

\[
\begin{align*}
F_h &= \text{Apply}(F_1, F_{2,h}, op) \\
F_l &= \text{Apply}(F_1, F_{2,l}, op) \\
F_r &= \text{GetNode}(F_{2,\text{var}}, F_h, F_l)
\end{align*}
\]
Apply Base Case: ComputeResult

- Constant (terminal) nodes and some other cases can be computed without recursion

<table>
<thead>
<tr>
<th>Operation and Conditions</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1 \text{ op } F_2; \ F_1 = C_1; \ F_2 = C_2$</td>
<td>$C_1 \text{ op } C_2$</td>
</tr>
<tr>
<td>$F_1 \oplus F_2; \ F_2 = 0$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$F_1 \oplus F_2; \ F_1 = 0$</td>
<td>$F_2$</td>
</tr>
<tr>
<td>$F_1 \ominus F_2; \ F_2 = 0$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$F_1 \odot F_2; \ F_2 = 1$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$F_1 \lhd F_2; \ F_1 = 1$</td>
<td>$F_2$</td>
</tr>
<tr>
<td>$F_1 \oslash F_2; \ F_2 = 1$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$\min(F_1, F_2); \ \max(F_1) \cdot \min(F_2)$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$\min(F_1, F_2); \ \max(F_2) \cdot \min(F_1)$</td>
<td>$F_2$</td>
</tr>
</tbody>
</table>

Similarly for max

other | null

Table 1: Input and output summaries of ComputeResult.
Algorithm 1: \( \text{Apply}(F_1, F_2, \text{op}) \rightarrow F_r \)

<table>
<thead>
<tr>
<th>input</th>
<th>( F_1, F_2, \text{op} ): ADD nodes and op</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>( F_r ): ADD result node to return</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{begin} & \quad & & \text{// Check if result can be immediately computed} \\
& \text{if} & & (\text{ComputeResult}(F_1, F_2, \text{op}) \rightarrow F_r \text{ is not null}) \quad \text{then} \\
& & & \text{return } F_r; \\
& \text{// Check if result already in apply cache} \\
& \text{if} & & (\langle F_1, F_2, \text{op} \rangle \rightarrow F_r \text{ is not in apply cache}) \quad \text{then} \\
& & & \text{// Not terminal, so recurse} \\
& & & \text{var} := \text{GetEarliestVar}(F_1^{\text{var}}, F_2^{\text{var}}); \\
& & & \text{// Set up nodes for recursion} \\
& & & \text{if} & & (F_1 \text{ is non-terminal} \land \text{var} = F_1^{\text{var}}) \quad \text{then} \\
& & & & & F_l^{v_1} := F_1, l; \quad F_h^{v_1} := F_1, h; \\
& & & & \text{else} \\
& & & & & F_{l/h}^{v_1} := F_1; \\
& & & & \text{if} & & (F_2 \text{ is non-terminal} \land \text{var} = F_2^{\text{var}}) \quad \text{then} \\
& & & & & F_l^{v_2} := F_2, l; \quad F_h^{v_2} := F_2, h; \\
& & & & \text{else} \\
& & & & & F_{l/h}^{v_2} := F_2; \\
& & & \text{// Recurse and get cached result} \\
& & & F_l := \text{Apply}(F_l^{v_1}, F_l^{v_2}, \text{op}); \\
& & & F_h := \text{Apply}(F_h^{v_1}, F_h^{v_2}, \text{op}); \\
& & & F_r := \text{GetNode}(\text{var}, F_h, F_l); \\
& & & \text{// Put result in apply cache and return} \\
& & & \text{insert } \langle F_1, F_2, \text{op} \rangle \rightarrow F_r \text{ into apply cache;} \\
& & & \text{return } F_r; \\
\text{end} & \quad & & \text{// Apply works for any binary operation!} \\
\text{Why?} & & & \text{// Note: Apply works for any binary operation!} \\
\end{align*}
\]
Apply Properties

• Apply uses *Apply cache*
  - \((F_1, F_2, \text{op}) \rightarrow F_R\)

• **Complexity**
  - Quadratic: \(O(|F_1| \cdot |F_2|)\)
    - \(|F|\) measured in node count
  - Why?
    - Cache implies touch every pair of nodes at most once!

• **Canonical?**
  - Same inductive argument as Reduce
Reduce-Restrict

• Important operation

• Have
  – $F(x,y,z)$

• Want
  – $G(x,y) = F|_{z=0}$

• Restrict $F|_{v=value}$ operation performs a *Reduce*
  – Just returns branch for $v=value$ whenever $v$ reached
  – Need *Restrict-Reduce cache* for $O(|F|)$ complexity
Marginalization, etc.

- Use Apply + Reduce-Restrict
  - \( \sum_x F(x, \ldots) = F|_{x=0} \oplus F|_{x=1}, \) e.g.

- Likewise for similar operations...
  - **ADD:** \( \min_x F(x, \ldots) = \min( F|_{x=0}, F|_{x=1} ) \)
  - **BDD:** \( \exists x F(x, \ldots) = F|_{x=0} \lor F|_{x=1} \)
  - **BDD:** \( \forall x F(x, \ldots) = F|_{x=0} \land F|_{x=1} \)
Apply Tricks I

- Build $F(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$
  - Don’t build a tree and then call Reduce!
  - Just use indicator DDs and Apply to compute

- In general:
  - Build any arithmetic expression bottom-up using Apply!

\[
x_1 + (x_2 + 4x_3) \ast (x_4) \\
\rightarrow x_1 \oplus (x_2 \oplus (4 \otimes x_3)) \otimes (x_4)
\]
Apply Tricks II

• Build *ordered* DD from *unordered* DD

z is out of order

result will have z in order!
Affine ADDs
ADD Inefficiency

• Are ADDs enough?
• Or do we need more compactness?
• Ex. 1: Additive reward/utility functions
  \[ R(a,b,c) = R(a) + R(b) + R(c) = 4a + 2b + c \]

• Ex. 2: Multiplicative value functions
  \[ V(a,b,c) = V(a) \cdot V(b) \cdot V(c) = \gamma^{(4a + 2b + c)} \]
Affine ADD (AADD)

- Define a new decision diagram – **Affine ADD**

- Edges labeled by **offset** (c) and **multiplier** (b):

  \[ \langle c_1, b_1 \rangle \text{ and } \langle c_2, b_2 \rangle \]

- **Semantics**: if (a) then \( c_1 + b_1 F_1 \) else \( c_2 + b_2 F_2 \)
Affine ADD (AADD)

- Maximize sharing by normalizing nodes $[0,1]$

- Example: if (a) then (4) else (2)

Need top-level affine transform to recover original range
AADD Reduce

Key point: 
*automatically* finds additive structure
AADD Examples

• Back to our previous examples…
• Ex. 1: Additive reward/utility functions
  • \( R(a,b) = R(a) + R(b) \)
    \[ = 2a + b \]
• Ex. 2: Multiplicative value functions
  • \( V(a,b) = V(a) \cdot V(b) \)
    \[ = \gamma^{2a + b}; \gamma < 1 \]
### AADD Apply & Normalized Caching

- Need to normalize Apply cache keys, e.g., given

\[
\langle 3 + 4F_1 \rangle \oplus \langle 5 + 6F_2 \rangle
\]

- before lookup in Apply cache, normalize

\[
8 + 4 \cdot \langle 0 + 1F_1 \rangle \oplus \langle 0 + 1.5F_2 \rangle
\]

<table>
<thead>
<tr>
<th>Operation and Conditions</th>
<th>Normalized Cache Key and Computation</th>
<th>Result Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle c_1 + b_1 F_1 \rangle \oplus \langle c_2 + b_2 F_2 \rangle; \ F_1 \neq 0)</td>
<td>(c_r + b_r F_r = \langle 0 + 1F_1 \rangle \oplus \langle 0 + (b_2/b_1)F_2 \rangle)</td>
<td>(\langle (c_1 + b_2 c_r + b_1 b_r F_r) \rangle)</td>
</tr>
<tr>
<td>(\langle c_1 + b_1 F_1 \rangle \oplus \langle c_2 + b_2 F_2 \rangle; \ F_1 \neq 0)</td>
<td>(c_r + b_r F_r = \langle 0 + 1F_1 \rangle \oplus \langle 0 + (b_2/b_1)F_2 \rangle)</td>
<td>(\langle (c_1 - c_2 + b_1 c_r + b_1 b_r F_r) \rangle)</td>
</tr>
<tr>
<td>(\langle c_1 + b_1 F_1 \rangle \odot \langle c_2 + b_2 F_2 \rangle; \ F_1 \neq 0)</td>
<td>(c_r + b_r F_r = \langle (c_1/b_1) + F_1 \rangle \odot \langle (c_2/b_2) + F_2 \rangle)</td>
<td>(\langle (b_1/b_2)c_r + (b_1/b_2)b_r F_r \rangle)</td>
</tr>
<tr>
<td>max((\langle c_1 + b_1 F_1 \rangle, \langle c_2 + b_2 F_2 \rangle); (\ F_1 \neq 0), Note: same for min)</td>
<td>(c_r + b_r F_r = \max(\langle 0 + 1F_1 \rangle, \langle (c_2 - c_1)/b_1 + (b_2/b_1)F_2 \rangle))</td>
<td>(\langle (c_1 + b_1 c_r + b_1 b_r F_r) \rangle)</td>
</tr>
<tr>
<td>any (\langle op \rangle) not matching above: (\langle c_1 + b_1 F_1 \rangle \langle op \rangle \langle c_2 + b_2 F_2 \rangle)</td>
<td>(c_r + b_r F_r = \langle c_1 + b_1 F_1 \rangle \langle op \rangle \langle c_2 + b_2 F_2 \rangle)</td>
<td>(\langle c_r + b_r F_r \rangle)</td>
</tr>
</tbody>
</table>
ADDs vs. AADDs

- Additive functions: $\sum_{i=1..n} x_i$

Note: no context-specific independence, but subdiagrams shared: result size $O(n^2)$
ADDs vs. AADDS

• Additive functions: $\sum_i 2^i x_i$
  – Best case result for ADD (exp.) vs. AADD (linear)
Main AADD Theorem

• **AADD can yield exponential time/space improvement over ADD**
  – and never performs worse!

• **But…**
  – Apply operations on AADDs can be exponential
  – Why?
    • Reconvergent diagrams possible in AADDs (edge labels), but not ADDs →
    • Sometimes Apply explores all paths if no hits in normalized Apply cache
# Bayes Net Results: Various Networks

<table>
<thead>
<tr>
<th>Bayes Net</th>
<th>Table</th>
<th>ADD</th>
<th>AADD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Entries</td>
<td>Time</td>
<td># Nodes</td>
</tr>
<tr>
<td>Alarm</td>
<td>1,192</td>
<td>2.97s</td>
<td>689</td>
</tr>
<tr>
<td>Barley</td>
<td>470,294</td>
<td>EML*</td>
<td>139,856</td>
</tr>
<tr>
<td>Carpo</td>
<td>636</td>
<td>0.58s</td>
<td>955</td>
</tr>
<tr>
<td>Hailfinder</td>
<td>9,045</td>
<td>26.4s</td>
<td>4,511</td>
</tr>
<tr>
<td>Insurance</td>
<td>2,104</td>
<td>278s</td>
<td>1,596</td>
</tr>
<tr>
<td>Noisy-Or-15</td>
<td>65,566</td>
<td>27.5s</td>
<td>125,356</td>
</tr>
<tr>
<td>Noisy-Max-15</td>
<td>131,102</td>
<td>33.4s</td>
<td>202,148</td>
</tr>
</tbody>
</table>

*EML: Exceeded Memory Limit (1GB)

AADDs compactly model noisy-or/max!
Question: why not just bypass graphical model and compile entire distribution into data structure?

This is knowledge compilation… for a later lecture, but note, can do with AADDs…
AADDs for Compilation

- Additive functions: $\sum_{i=0..n-1} F(x_i, x_{(i+1) \mod n})$

Pairwise factoring evident in AADD structure