ICAPS 2012 Tutorial

Decision Diagrams in Discrete and Continuous Planning

Scott Sanner

NICTA

THE AUSTRALIAN NATIONAL UNIVERSITY
DD Definition

• Decision diagrams (DDs):
  – DAG variant of decision tree
  – Decision tests ordered
  – Used to represent:
    • $f: B^n \rightarrow B$ (boolean – BDD, set of subsets $\{\{a,b\},\{a\}\}$ – ZDD)
    • $f: B^n \rightarrow Z$ (integer – MTBDD / ADD)
    • $f: B^n \rightarrow R$ (real – ADD)

more expressive domains / ranges possible – @ end
What’s the Big Deal?

• More than compactness
  – Ordered decision tests in DDs support efficient operations

• ADD: \(-f, f \oplus g, f \otimes g, \max(f, g)\)

• BDD: \(\neg f, f \land g, f \lor g\)

• ZDD: \(f \setminus g, f \cap g, f \cup g\)

  – Efficient operations key to planning / inference
Tutorial Outline

• Need for $B^n \rightarrow B / Z / R$ & operations in planning

• DDs for representing $B^n \rightarrow B / Z / R$
  – Why important?
  – What can they represent compactly?
  – How to do efficient operations?

• Extensions and Software
  – ZDDs, AADDs, FOADDs, XADDs, …

• DDs vs. Compilation (d-DNNF)
Factored Representations

- Natural state representations in planning

- State is inherently factored
  - Room location: \( R = \{1,2,3,4,5,6\} \)
  - Door status: \( D_i = \{\text{closed/0, open/1}\}; \ i=1..7 \)

- Relational fluents, e.g., \( \text{At}(r_1,6) \), (STRIPS) are ground variable templates: \( \text{at-r1-6} \)

For simplicity we will assume all state vars are boolean \( \{0,1\} \) – all DD ideas generalize to multi-valued case
Using Factored State in Planning

• Classical planning
  – State given by variable assignments
    • (R=1, D_1=o, D_2=c, ..., D_7=o)
  – Planning operators efficiently update state
  – Dominated by search-based algorithms
    • Explicit representation of $B^n \rightarrow B / Z / R$ not always crucial

• Non-det. / probabilistic planning, temporal verification
  – To compute *progressions* and *regressions*, often need:
    • State sets: $B^n \rightarrow B$ (states satisfying condition)
    • Policies: $B^n \rightarrow Z$ (action ids $\rightarrow Z$)
    • Value functions: $B^n \rightarrow R$
  – And operations on these functions
Factored Transition Systems I

- If have factored state
  - exploit factored transition systems with *graphical model* (arcs encode dependences)

- Can represent
  - (Non-)deterministic transitions
    - \( T(x_2' \mid x_1, x_2): (x_2', x_1, x_2) \rightarrow B \)
  - Probabilistic transitions
    - \( P(x_2' \mid x_1, x_2): (x_2', x_1, x_2) \rightarrow R \) (really \([0,1]\))

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_2'</th>
<th>T/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How is table different for det / non-det cases?
Factored Transition Systems II

- (Non-)det. transition systems
  - Forward reachability (FR) / backward reachability (BR)

- Progression:
  - given a single state \( x_1=0, x_2=1 \)
    \[ \text{FR}(x_1', x_2') = T(x_1' | x_1=0, x_2=1) \land T(x_2' | x_2=1) \]
  - given a set of possible states \( S: (x_1, x_2) \rightarrow B \)
    \[ \text{FR}(x_1', x_2') = \exists x_1 \exists x_2 \ T(x_1' | x_1, x_2) \land T(x_2' | x_2) \land S(x_1, x_2) \]
  - Note: \( \exists x \ F(x, \ldots) = F(x=1, \ldots) \lor F(x=0, \ldots) \)

- Regression: given goal function \( G: (x_1', x_2') \rightarrow B \)
  - \( \text{BR}(x_1, x_2) = \exists x_1' \exists x_2' \ T(x_1' | x_1, x_2) \land T(x_2' | x_2) \land G(x_1', x_2') \)
Factored Transition Systems III

- **Probabilistic transition systems**

  - **State updates**: given $P(x_1, x_2)$
    - State sample: $x_1' \sim P(x_1')$: $\sum_{x_1} \sum_{x_2} P(x_1'| x_1, x_2) \otimes P(x_1, x_2)$
    - $x_2' \sim P(x_2')$: $\sum_{x_1} \sum_{x_2} P(x_2'| x_2) \otimes P(x_1, x_2)$

  - **Note**: $\sum_x F(x, ...) = F(x=1, ...) \oplus F(x=0, ...)$
  - **State belief update**:
    $$P(x_1', x_2') = \sum_{x_1} \sum_{x_2} P(x_1'| x_1, x_2) \otimes P(x_2'| x_2) \otimes P(x_1, x_2)$$

- **DTR**: given value $V'(x_1', x_2')$, compute $E[V](x_1, x_2)$
  - $V(x_1, x_2) = \sum_{x_1'} \sum_{x_2'} P(x_1'| x_1, x_2) \otimes P(x_2'| x_2) \otimes V'(x_1', x_2')$

$P(x_1, x_2)$ can be $\{0,1\}$ if prev. state known.
Factored Transition Systems IV

- Adversarial transition systems

- Adversarial DTR
  - Given value $V'(x_1', x_2')$, compute $E[V](x_1, x_2)$
  - Opponent chooses *non-det.* transitions to minimize $V$
    
    $$V(x_1, x_2) = \min_{x_1', \min_{x_2'}} T(x_1' | x_1, x_2) \otimes T(x_2' | x_2) \otimes V'(x_1', x_2')$$

  - Note: $\min_x F(x, \ldots) = \min( F(x=1, \ldots), F(x=0, \ldots) )$

- Many other multi-agent formalizations
  - Often alternating turns with action variables…
Factored/Symbolic Planning Approaches

- (Non-det) planning
  - Planning as model checking
  - Conformant planning
  - Temporal verification, e.g., $x_1$ Until $x_2$?
    \[(Bertoli, Cimatti, Pistore, Roveri, Traverso, ...)
    \]
    see refs @ http://mbp.fbk.eu/AIPS02-tutorial.html

- Probabilistic planning
  - MDPs: SPUDD (Hoey, Boutilier et al)
  - POMDPs: Symbolic Perseus (Poupart et al)
    http://www.cs.uwaterloo.ca/~ppoupart/software.html

- Adversarial planning
  - GDL: Gamer (Kissmann, Edelkamp)
    http://www.tzi.de/~kissmann/publications/

All use of Bn $\rightarrow$ B / Z / R in representation
All planning as operations on these functions
OK, we need $B^n \rightarrow B / Z / R$ for Planning

But why Decision Diagrams?
Why DDs for Planning?

• For *symbolic / factored planning*, we need:
  – Compact representations?
  – Efficient operations: $\neg$, $\land$, $\lor$, $\max(F)$, $\oplus$, $\otimes$, $\max(F_1,F_2)$?

• **Reason 1: Space considerations**
  – $V(\text{Door-1-open, \ldots, Door-40-open})$ requires
    $\sim 1$ terabyte if all states enumerated

• **Reason 2: Time considerations**
  – With 1 gigaflop/s. computing power, binary operation on above function requires $\sim 1000$ seconds
Function Representation (Tables)

• How to represent functions: \( B^n \rightarrow R \)?

• How about a fully enumerated table…

• …OK, but can we be more compact?

<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>

Think of BDDs as \{0,1\} subset of ADD range
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)
Trees vs. ADDs

- Trees can compactly represent AND / OR
  - But not XOR (linear as ADD, exponential as tree)
  - Why? Trees must represent every path
How to Reduce Ordered Tree to ADD?

- Recursively build bottom up
  - Hash terminal nodes $R \rightarrow ID$
    - leaf cache
  - Hash non-terminal functions $(v, ID_0, ID_1) \rightarrow ID$
    - internal node cache
    - collectively referred to as reduce cache
Algorithm 1: \( \text{Reduce}(F) \rightarrow F_r \)

**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
GetNode

- Returns unique ID for internal nodes
- Removes redundant branches

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

```plaintext
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

begin
  // If branches redundant, return child
  if ($F_l = F_h$) then
    return $F_l$;

  // Make new node if not in cache
  if ($\langle v, F_h, F_l \rightarrow id \rangle$ is not in node cache) then
    $id :=$ currently unallocated id;
    insert $\langle v, F_h, F_l \rangle \rightarrow id$ in cache;

  // Return the cached, canonical node
  return $id$;
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
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.,

Can also do reorder using Apply… later
Decision Diagrams: Apply

(how to do efficient operations on DDs)
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 \bigoplus F_2; F_1 = C_1; F_2 = C_2 )</td>
<td>( C_1 \bigoplus 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 \oplus F_2; F_2 = 0 )</td>
<td>( F_1 )</td>
</tr>
<tr>
<td>( F_1 \bigotimes F_2; F_2 = 1 )</td>
<td>( F_1 )</td>
</tr>
<tr>
<td>( F_1 \bigotimes F_2; F_1 = 1 )</td>
<td>( F_2 )</td>
</tr>
<tr>
<td>( F_1 \bigodot 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>
<tr>
<td>similarly for max</td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 1: Input and output summaries of **ComputeResult**.
Apply Recursion

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

- $F_1$ & $F_2$ matching var

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

- Non-matching var: $v_1 \prec v_2$

  - $F_h = \text{Apply}(F_1, F_{2,h}, \text{op})$
  - $F_l = \text{Apply}(F_1, F_{2,l}, \text{op})$
  - $F_r = \text{GetNode}(F_{2\text{ var}}, F_h, F_l)$
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>

\[
\begin{align*}
\text{begin} \\
&\text{// Check if result can be immediately computed} \\
&\text{if } (\text{ComputeResult}(F_1, F_2, \text{op}) \rightarrow F_r \text{ is not null }) \text{ then} \\
&\quad \text{return } F_r; \\
&\text{// Check if result already in apply cache} \\
&\text{if } (\langle F_1, F_2, \text{op} \rangle \rightarrow F_r \text{ is not in apply cache}) \text{ then} \\
&\quad \text{// Not terminal, so recurse} \\
&\quad \text{var} := \text{GetEarliestVar}(F_1^{\text{var}}, F_2^{\text{var}}); \\
&\text{// Set up nodes for recursion} \\
&\quad \text{if } (F_1 \text{ is non-terminal } \land \text{var} = F_1^{\text{var}}) \text{ then} \\
&\quad \quad F_l^{\text{v1}} := F_1,; \quad F_h^{\text{v1}} := F_1,; \\
&\quad \text{else} \\
&\quad \quad F_{l/h}^{\text{v1}} := F_1; \\
&\quad \text{if } (F_2 \text{ is non-terminal } \land \text{var} = F_2^{\text{var}}) \text{ then} \\
&\quad \quad F_l^{\text{v2}} := F_2,; \quad F_h^{\text{v2}} := F_2,; \\
&\quad \text{else} \\
&\quad \quad F_{l/h}^{\text{v2}} := F_2; \\
&\quad \text{// Recurse and get cached result} \\
&\quad F_l := \text{Apply}(F_l^{\text{v1}}, F_l^{\text{v2}}, \text{op}); \\
&\quad F_h := \text{Apply}(F_h^{\text{v1}}, F_h^{\text{v2}}, \text{op}); \\
&\quad F_r := \text{GetNode}(\text{var}, F_h, F_l); \\
&\text{// Put result in apply cache and return} \\
&\quad \text{insert } \langle F_1, F_2, \text{op} \rangle \rightarrow F_r \text{ into apply cache;} \\
&\text{return } F_r;
\end{align*}
\]

Note: Apply works for any binary operation!
Why?
Apply Example

• (#)’s represent order of Apply recursions
  – And what nodes they are applied to
Apply Properties

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

- Complexity
  - Quadratic: \(O(|F_1|,|F_2|)\)
    - \(|F|\) 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
What about $\exists x, \forall x, \min_x, \sum_x$?

- Use Apply + Reduce-Restrict
  - $\exists x \ F(x, \ldots) = F|_{x=0} \lor F|_{x=1}$
  - $\forall x \ F(x, \ldots) = F|_{x=0} \land F|_{x=1}$
  - $\min_x \ F(x, \ldots) = \min( F|_{x=0}, F|_{x=1} )$
  - $\sum_x \ F(x, \ldots) = F|_{x=0} \oplus F|_{x=1}$, e.g.

![Diagram](image)
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

$$x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

– In general:

• Build any arithmetic expression bottom up using Apply!

$$-x_1 + (x_2 + 4x_3) \times (x_4) \rightarrow x_1 \oplus (x_2 \oplus (4 \otimes x_3)) \otimes (x_4)$$
Apply Tricks II

- Build *ordered* DD from *unordered* DD

\[ z \text{ is out of order} \quad \rightarrow \quad \text{result will have } z \text{ in order!} \]
ZDDs
(zero-suppressed BDDs)

Represent sets of subsets
ZDDs for Sets of Subsets

- Example BDD and ZDD

Figure 2. The BDD and the ZDD for the set of subsets \{\{a,b\}, \{a,c\}, \{c\}\}.

Variables not assigned value on path (e.g., c) assumed false! (0-suppressed)
ZDDs vs. BDDs

- But ZDDs not universal replacement for BDDs…

Figure 1. BDD and ZDD for $F = ab + cd$. 

Vars that aren’t false need this marked explicitly.
How to Modify Apply for ZDDs?

- Simple
  - $F_x$ is sub-ZDD for set with $x$
  - $F_{\backslash x}$ is sub-ZDD for set without $x$

- $F \cap G$:
  - if $(x$ in set$)$
    - then $F_x \cap G_x$
    - else $F_{\backslash x} \cap G_{\backslash x}$

- This is just standard *Apply*
  - With properly defined GetNode, leaf ops: $\cap = \land$, $\cup = \lor$
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 \( b \):

\[
\begin{array}{c}
\langle c_1, b_1 \rangle \quad a \\
F_1 \\
\langle c_2, b_2 \rangle \\
F_2
\end{array}
\]

- **Semantics**: if \( a \) then \( (c_1 + b_1 F_1) \) else \( (c_2 + b_2 F_2) \)
- Maximize sharing by **normalizing** nodes \([0,1]\)
- Example: if \( a \) then \( 4 \) else \( 2 \)

\[
\begin{array}{c}
\langle 4,0 \rangle \\
0 \\
\langle 2,0 \rangle
\end{array}
\quad \text{Normalize}
\quad \begin{array}{c}
\langle 1,0 \rangle \\
0 \\
\langle 0,0 \rangle
\end{array}
\]

\[
\begin{array}{c}
\langle 2,2 \rangle
\end{array}
\]
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>(\langle c_r + b_r F_r \rangle = \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 \oplus \langle c_2 + b_2 F_2 \rangle; \ F_1 \neq 0)</td>
<td>(\langle c_r + b_r F_r \rangle = \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 \oplus \langle c_2 + b_2 F_2 \rangle; \ F_1 \neq 0)</td>
<td>(\langle c_r + b_r F_r \rangle = \langle (c_1/b_1) + F_1 \rangle \oplus \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>(\langle c_r + b_r F_r \rangle = \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>(\langle c_r + b_r F_r \rangle = \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>
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

  • \( V(a, b) = V(a) \cdot V(b) \)
    \[ = \gamma^{2a + b}; \quad \gamma < 1 \]
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)
ADDs vs. AADDs

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

Pairwise factoring evident in structure
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
Other DDs
Multivalued (MV-)DD

• A DD with multivalued variables
  – straightforward k-branch extension
  – e.g., k=6
Multi-terminal (MT-)BDD

- Imagine terminal is 3 bits… use 3 BDDs

- MT-BDD – combine into single diagram
  - Same as ADD using bit vector (integer) leaves
(F)EV-BDDs

- **EdgeValue-BDD** is like AADD where only additive constant subtracted
  - Not a full affine transform
  - Better numerical precision properties than AADD
    - Additive, but no multiplicative compression like AADD

- **Factor-EVBDD** is for integer leaves \( Z \)
  - Instead of dividing by range… factors out largest prime factor as multiplier
Other Discrete DDs

- **K*DDs, BMDs, K*BMDs**
  - Like ZDD, AADD, different ways to do decomposition
  - Mainly used in digital verification literature

- **FODDs**
  (Wang, Joshi, Khardon, JAIR-08)
  - Support first-order logical decision tests
  - Can be very compact
  - Require non-trivial reduction operations

- **FOADDs**
  (Sanner, Boutilier, AIJ-09)
  - Alternate semantics to FODDs
First-order ADDs (FOADDs)

- Want to compactly represent:

\[
\text{case } = \begin{array}{|c|c|}
\hline
\exists x. [A(x) \lor \forall y. A(x) \land B(x) \land \neg A(y)] & 1 \\
\neg " & 0 \\
\hline
\end{array}
\]

- Push down quantifiers, expose prop. structure:

\[
[\exists x. A(x)] \lor ([\exists x. A(x) \land B(x)] \land [\forall y. \neg A(y)])
\]

- Convert to first-order ADD:

\[
\text{case } = \begin{array}{|c|c|}
\hline
a \lor (b \land \neg a) & 1 \\
\neg " & 0 \\
\hline
\end{array}
\]

- Convert to first-order ADD:

\[
\text{case } = \begin{array}{|c|c|}
\hline
a \lor (b \land \neg a) & 1 \\
\neg " & 0 \\
\hline
\end{array}
\]
XADDs: Extend DDs to continuous variables?

What are we representing?

Piecewise functions!
$z = f(x, y) =\begin{cases} (x > 3) \land (y \cdot x) : x + y \\ (x \cdot 3) \lor (y > x) : x^2 + xy^3 \end{cases}$
What operations for Cases?

• Can define all of the following case operations:
  • $f_1 \oplus f_2$, $f_1 \otimes f_2$
  • $\max(f_1, f_2)$, $\min(f_1, f_2)$
  • $\int_x f(x) \, dx$, $\int_x f(x) \, \delta(x-g(y)) \, dx$
  • $\max_x f(x)$, $\min_x f(x)$

• Makes possible
  – Exact inference in continuous graphical models
  – New paradigms for optimization and sequential control
  – New formalizations of machine learning problems

(see Sanner et al, UAI-11 and AAAI-12)
Case → XADD

- Extended ADD representation of case statements

\[ V = \begin{cases} 
  x_1 + k > 100 \land x_2 + k > 100 : & 0 \\
  x_1 + k > 100 \land x_2 + k \cdot 100 : & x_2 \\
  x_1 + k \cdot 100 \land x_2 + k > 100 : & x_1 \\
  x_1 + x_2 + k > 100 \land x_1 + k \cdot 100 \land x_2 + k \cdot 100 \land x_2 > x_1 : & x_2 \\
  x_1 + x_2 + k > 100 \land x_1 + k \cdot 100 \land x_2 + k \cdot 100 \land x_2 \cdot x_1 : & x_1 \\
  x_1 + x_2 + k \cdot 100 : & x_1 + x_2 
\end{cases} \]
XADD Maximization

\[
\text{max}(y > 0, x > 0) = x > y
\]

May introduce new decision tests
Maintaining XADD Orderings I

• Max may get variables out of order

Decision ordering (root→leaf)

- \( x > y \)
- \( y > 0 \)
- \( x > 0 \)

\[ \max( x, y ) = \]

\[ y > 0, \quad x > 0 \]

Newly introduced node is out of order!
Maintaining XADD Orderings II

• Substitution may get vars out of order

Decision ordering (root→leaf):
• x > y
• y > 0
• x > z

\[ \sigma = \{ z/y \} \]

Substituted nodes are now out of order!
Correcting XADD Ordering

• Obtain *ordered* XADD from *unordered* XADD
  – key idea: binary operations maintain orderings

\[ z \text{ is out of order} \]

\[ \text{result will have } z \text{ in order!} \]

Inductively assume ID\(_1\) and ID\(_0\) are ordered.

All operands ordered, so applying \(\otimes, \oplus\) produces ordered result!
XADD Pruning

Node unreachable – \( x + y < 0 \) always false if \( x > 0 \) & \( y > 0 \)

If \textbf{linear}, can detect with feasibility checker of LP solver & prune

Similar to Penberthy & Weld, AAAI-94
Approximation

Sometimes no DD is compact, but approximation is…
Problem: Value ADD Too Large

• Sum: \((\sum_{i=1..3} 2^i \cdot x_i) + x_4 \cdot \varepsilon\text{-Noise}\)

• How to approximate?
Solution: APRICODD Trick

- Merge $\approx$ leaves and reduce:

- Error is bounded!
Can use ADD to Maintain Bounds!

• Change leaf to represent range \([L, U]\)
  – Normal leaf is like \([V, V]\)
  – When merging leaves…
    • keep track of min and max values contributing
More Compactness? AADDs?

- Sum: \( \sum_{i=1..3} 2^i \cdot x_i \) + \( x_4 \cdot \varepsilon \)-Noise

- How to approximate? Error-bounded merge
Solution: MADCAP Trick

- Merge $\approx$ nodes from bottom up:

```
ROOT
\[<0 + 7.11 \times>\]
\[\times 1\]
\[<0 + 0.852 \times> <0.142 + 0.858 \times>\]
\[\times 2\]
\[<0.332 + 0.668 \times> <0 + 0.665 \times>\]
\[\times 3\]
\[<0 + 0 \times> <1 + 0 \times>\]
\[0\]
```
Approximation

- Bounded (interval) approximation

- This XADD has > 1000 nodes!
- Should only require < 10 nodes!
Open Problem for XADDs

• How to extend APRICODD trick to
  – expressions for decisions
  – expressions for leaves
Example Results
Empirical Comparison: Table/ADD/AADD

- Sum: $\sum_{i=1}^{n} 2^i \cdot x_i \oplus \sum_{i=1}^{n} 2^i \cdot x_i$

- Prod: $\prod_{i=1}^{n} \gamma^{(2^i \cdot x_i)} \otimes \prod_{i=1}^{n} \gamma^{(2^i \cdot x_i)}$
Application: Bayes Net Inference

• Use variable elimination
  – Replace CPTs with ADDs or AADDs
  – Could do same for clique/junction-tree algorithms

• Exploits
  – Context-specific independence
    • Probability has logical structure:

\[
P(a|b,c) = \begin{cases} 1 & \text{if } b \ \text{?} \\ .7 & \text{if } c \ \text{?} \\ .3 & \text{otherwise} \end{cases}
\]

  – Additive CPTs
    • Probability is discretized linear function:

\[
P(a|b_1...b_n) = c + b \cdot \sum_i 2^i b_i
\]

  – Multiplicative CPTs
    • Noisy-or (multiplicative AADD):

\[
P(e|c_1...c_n) = 1 - \prod_i (1 - p_i)
\]

• If factor has above compact form, AADD exploits it
## Bayes Net Results: Various Networks

<table>
<thead>
<tr>
<th>Bayes Net</th>
<th>Table # Entries</th>
<th>Time</th>
<th>ADD # Nodes</th>
<th>Time</th>
<th>AADD # Nodes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>1,192</td>
<td>2.97s</td>
<td>689</td>
<td>2.42s</td>
<td>405</td>
<td>1.26s</td>
</tr>
<tr>
<td>Barley</td>
<td>470,294</td>
<td>EML*</td>
<td>139,856</td>
<td>EML*</td>
<td>60,809</td>
<td>207m</td>
</tr>
<tr>
<td>Carpo</td>
<td>636</td>
<td>0.58s</td>
<td>955</td>
<td>0.57s</td>
<td>360</td>
<td>0.49s</td>
</tr>
<tr>
<td>Hailfinder</td>
<td>9,045</td>
<td>26.4s</td>
<td>4,511</td>
<td>9.6s</td>
<td>2,538</td>
<td>2.7s</td>
</tr>
<tr>
<td>Insurance</td>
<td>2,104</td>
<td>278s</td>
<td>1,596</td>
<td>116s</td>
<td>775</td>
<td>37s</td>
</tr>
<tr>
<td>Noisy-Or-15</td>
<td>65,566</td>
<td>27.5s</td>
<td>125,356</td>
<td>50.2s</td>
<td>1,066</td>
<td>0.7s</td>
</tr>
<tr>
<td>Noisy-Max-15</td>
<td>131,102</td>
<td>33.4s</td>
<td>202,148</td>
<td>42.5s</td>
<td>40,994</td>
<td>5.8s</td>
</tr>
</tbody>
</table>

*EML: Exceeded Memory Limit (1GB)
Application: POMDPs

- Provided an AADD implementation for Guy Shani’s factored POMDP solver
- Final value function size results:

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>AADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Management</td>
<td>7000</td>
<td>92</td>
</tr>
<tr>
<td>Rock Sample</td>
<td>189</td>
<td>34</td>
</tr>
</tbody>
</table>
Application: MDP Solving

- SPUDD Factored MDP Solver (HSHB99)
  - Originally uses ADDs
  - Can use AADDs as well...

\[
V^{n+1}(x_1...x_i) = R(x_1...x_i) + \\
\gamma \cdot \max_a \sum_{x_1'...x_i'} \prod_{F_1...F_i} P_1(x_1'|...x_i) ... P_i(x_i'|...x_i) \cdot V^n(x_1'...x_i')
\]
Application: SysAdmin

- SysAdmin MDP (GKP, 2001)
  - Network of computers: $c_1, \ldots, c_k$
  - Various network topologies
  - Every computer is running or crashed
  - At each time step, status of $c_i$ affected by
    - Previous state status
    - Status of incoming connections in previous state
  - Reward: $+1$ for every computer running (additive)
Results I: SysAdmin (10% Approx)
Results II: SysAdmin
Traffic Domain

- Binary **cell transmission model (CTM)**

- Actions
  - Light changes

- Objective:
  - Maximize empty cells in network
Results Traffic

Time (s)

- APRICODD
- MADCAP

EML

20 Vars, Exact
20 Vars, Approx (10%)
24 Vars, Exact
24 Vars, Approx (10%)

Space (# Nodes)

- APRICODD
- MADCAP

EML

20 Vars, Exact
20 Vars, Approx (10%)
24 Vars, Exact
24 Vars, Approx (10%)
Symbolic Dynamic Programming

• Value Iteration for $h \in 0..H$
  
  - Regression step:
    
    $$Q^{h+1}_a(\vec{b}, \vec{x}) = R_a(\vec{b}, \vec{x}) + \gamma \cdot \sum \int_{\vec{x}'} \left( \prod_{i=1}^{n} P(b'_i|\vec{b}, \vec{x}, a) \prod_{j=1}^{m} P(x'_j|\vec{b}, \vec{b'}, \vec{x}, a) \right) V^h(\vec{b'}, \vec{x'}) d\vec{x'}$$

  - Maximization step:
    
    $$V_{h+1} = \max_{a \in A} Q^{h+1}_a(\vec{b}, \vec{x})$$

Exact for any reward, discrete noise transition dynamics! (Sanner et al UAI-110)
Results: XADD Pruning vs. No Pruning

Summary:
- without pruning: superlinear vs. horizon
- with pruning: linear vs. horizon

Worth the effort to prune!
Exact XADD Value Functions

Knapsack

Mars Rover Linear

Mars Rover Nonlinear

Exact value functions in case form:
• linear & nonlinear piecewise boundaries!
• nonlinear function surfaces!
Decision Diagram Software

Work with decision diagrams in < 1 hour!
Software Packages

• CUDD
  – BDD / ADD / ZDD
  – http://vlsi.colorado.edu/~fabio/CUDD/
  – Hands down, the best package available

• JavaBDD (native interface to CUDD / others):
  – http://javabdd.sourceforge.net/

• NuSMV – Model Based Planner (MBP)
  – http://mbp.fbk.eu/

• SPUDD – ADD-based value iteration for MDPs
  – http://www.computing.dundee.ac.uk/staff/jessehoey/spudd/index.html

• Symbolic Perseus – Matlab / Java ADD version of value PBVI for POMDPs

• Java BDDs / ADDs / AADDs / XADDs
  – Scott’s code, not high performance, but functional
  – Includes Java version of SPUDD factored MDP solver & variable elimination
Compilation vs. Decision Diagrams
BDDs in NNF

- Can express BDD as NNF formula
- Can represent NNF diagrammatically

Definitions / Diagrams from "A Knowledge Compilation Map", Darwiche and Marquis. JAIR 02
**d-DNNF**

- **Decomposable NNF:** sets of leaf vars of conjuncts are disjoint

- **Deterministic NNF:** formula for disjuncts have disjoint models (conjunction is unsatisfiable)
d-DNNF

- D-DNNF used to **compile single formula**
  - d-DNNF does not support efficient binary operations ($\lor, \land, \neg$)
  - d-DNNF shares some polytime operations with OBDD / ADD
    - (weighted) model counting (CT) – used in many inference tasks
    - $\rightarrow$ Size(d-DNNF) $\leq$ Size(OBDD) so more efficient on d-DNNF

---

**Diagram**

- **NNF**
  - d-NNF
  - s-NNF
  - DNNF
  - f-NNF
- **BDD**
- **d-DNNF**
  - VA, IM, CT
- **EQ?**
- **Orderd BDD, in previous slides I call this a BDD**
  - VA, IM, EQ, SE
- **OBDD**
- **OBDD<**
- **MODS**
- **DNF**
  - VA, IM, EQ, SE
  - CO, CE, EQ, SE, ME
- **CNF**
  - VA, IM

---

**Defintions / Diagrams from**

"**A Knowledge Compilation Map**", Darwiche and Marquis. JAIR 02

---

**Table 4: Notations for queries.**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>polytime consistency check</td>
</tr>
<tr>
<td>VA</td>
<td>polytime validity check</td>
</tr>
<tr>
<td>CE</td>
<td>polytime clausal entailment check</td>
</tr>
<tr>
<td>IM</td>
<td>polytime implicant check</td>
</tr>
<tr>
<td>EQ</td>
<td>polytime equivalence check</td>
</tr>
<tr>
<td>SE</td>
<td>polytime sentential entailment check</td>
</tr>
<tr>
<td>CT</td>
<td>polytime model counting</td>
</tr>
<tr>
<td>ME</td>
<td>polytime model enumeration</td>
</tr>
</tbody>
</table>
Compilations vs Decision Diagrams

• Summary
  – **If** you can compile problem into **single formula** then compilation is likely preferable to DDs
    • provided you only need ops that compilation supports

  – Not *all* compilations efficient for *all binary* operations
    • e.g., all ops needed for progression / regression approaches
    • fixed ordering of DDs help support these operations

• Note: other compilations (e.g., arithmetic circuits)
And that’s a crash course in DDs!

Take-home point:

• If your problem is factored
• and you’re currently using a tabular representation
• and you need binary operations on these tables
  → consider using a DD instead.