Decision Diagrams in Automated Planning and Scheduling

Scott Sanner
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 ⊕ g, f ⊗ g, max(f, g)
  • BDD: ¬f, f ∧ g, f ∨ g
  • ZDD: f \ g, f ∩ g, f ∪ 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, (F)EVBDDs …

• 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: $D_i=\{\text{closed}/0, \text{open}/1\}; i=1..7$

- 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
    • \( (R=1, D_1=0, D_2=c, \ldots, D_7=0) \)
  – Planning operators efficiently update state
  – Satisficing tracks dominated by search-based algorithms
    • But representation of \( B^n \rightarrow B / Z / R \) important for optimal tracks

• 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)

  \[
  \begin{array}{cc|cc}
    x_1 & x_2 & x_1' & T/P \\
    0 & 0 & 0 & 1 \\
    0 & 0 & 1 & 0 \\
    0 & 1 & 0 & 0 \\
    \vdots & \vdots & \vdots & \vdots 
  \end{array}
  \]

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

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 $x_1=0, x_2=1$
    » $\text{FR}(x_1', x_2') = \text{T}(x_1' \mid x_1=0, x_2=1) \land \text{T}(x_2' \mid x_2=1)$
  – given a set of possible states $S: (x_1, x_2) \rightarrow B$
    » $\text{FR}(x_1', x_2') = \exists x_1 \exists x_2 \text{T}(x_1' \mid x_1, x_2) \land \text{T}(x_2' \mid x_2) \land S(x_1, x_2)$
  – Note: $\exists x \text{ F}(x, ...) = F(x=1, ...) \lor F(x=0, ...)$

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

• Probabilistic transition systems

\[ P(x_1, x_2) \]

- 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: \( \sum_x F(x, \ldots) = F(x=1, \ldots) \oplus F(x=0, \ldots) \)
  
  - 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 \( E[V](x_1, x_2) \)
  
  \[ V(x_1, x_2) = \sum_{x_1'} \sum_{x_2'} P(x_1'| x_1, x_2) \otimes P(x_2'| x_2) \otimes V'(x_1', x_2') \]
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

- Classical and Adversarial planning
  - Classical: recent work by Torralba, Alcázar, *et al*
  - Games: Gamer (*Kissmann, Edelkamp*)

- (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](http://mbp.fbk.eu/AIPS02-tutorial.html)

- Probabilistic planning
  - MDPs: SPUDD (*Hoey, Boutilier et al*)
  - POMDPs: Symbolic Perseus (*Poupart et al*)

All use of Bn → 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?
Function Representation (Tables)

- How to represent functions: $B^n \rightarrow \mathbb{R}$?
- How about a fully enumerated table…
- …OK, how to do operations?

<table>
<thead>
<tr>
<th></th>
<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</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<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</td>
<td>0.00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Manipulating Discrete Distributions

- Marginalization

\[ \sum \limits_{b} P(A, b) = P(A) \]

\[ \sum \limits_{b} \begin{array}{|c|c|c|} 
\hline 
A & B & Pr \\
\hline 
0 & 0 & .1 \\
0 & 1 & .3 \\
1 & 0 & .4 \\
1 & 1 & .2 \\
\hline 
\end{array} = \begin{array}{|c|c|} 
\hline 
A & Pr \\
\hline 
0 & .4 \\
1 & .6 \\
\hline 
\end{array} \]
Manipulating Discrete Distributions

- Maximization

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

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

\[
\begin{array}{cc}
A & Pr \\
0 & .3 (B=1) \\
1 & .4 (B=0) \\
\end{array}
\]
Manipulating Discrete Distributions

- Binary Multiplication

\[ P(A|B) \cdot P(B|C) = P(A, B|C) \]

<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>.9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>.2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</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>.9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>.2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>.8</td>
</tr>
</tbody>
</table>

- Same principle holds for all binary ops
  - +, -, /, max, etc…
Discrete Inference & Optimization

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Pr</td>
</tr>
<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>

\[ P(A, B) = \]

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

Are we done?

Why do we need DDs?
Why DDs for Planning?

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

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

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

• 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
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
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

  $\rightarrow$ 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

$$(x_1, 1, 0) \rightarrow 2$$

```
\begin{align*}
(x_1, 1, 0) & \rightarrow 2 \\
1 & \rightarrow 2 \\
2 & \rightarrow 2 \\
0 & \rightarrow 2 \\
x_1 & \rightarrow 2 \\
x_1 & \rightarrow 2 \\
1 & \rightarrow 2 \\
0 & \rightarrow 2 \\
x_1 & \rightarrow 2 \\
1 & \rightarrow 2 \\
0 & \rightarrow 2 \\
x_1 & \rightarrow 2 \\
\end{align*}
```
Reduce Algorithm

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

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 $F_l$;
  end

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

  // 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\(^{th}\) 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)
Recap

• Recall the Apply recursion

Result: ADD operations can avoid state enumeration

Need to handle recursive cases

Need to handle base cases
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: $v_1 \prec v_2$

\[
F_h = Apply(F_1, F_2,h, op)
\]
\[
F_l = Apply(F_1, F_2,l, op)
\]
\[
F_r = GetNode(F_{\text{var}}^2, F_h, F_l)
\]
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>$F_1 \ominus F_2; \ F_2 = 1$</td>
<td>$F_1$</td>
</tr>
<tr>
<td>$F_1 \ominus F_2; \ F_1 = 1$</td>
<td>$F_2$</td>
</tr>
<tr>
<td>$F_1 \ominus F_2; \ F_2 = 1$</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.
Apply Algorithm

Note: Apply works for any binary operation!

Why?

Algorithm 1: \( \text{Apply}(F_1, F_2, \text{op}) \rightarrow F_r \)

\begin{verbatim}
input : F_1, F_2, \text{op} : ADD nodes and op
output: F_r : ADD result node to return
begin
  // Check if result can be immediately computed
  if (\( \text{ComputeResult}(F_1, F_2, \text{op}) \rightarrow F_r \) is not null) then
    return \( F_r \);
  // Check if result already in apply cache
  if (\( \langle F_1, F_2, \text{op} \rangle \rightarrow F_r \) is not in apply cache) then
    // Not terminal, so recurse
    var := GetEarliestVar(F_1^{\text{var}}, F_2^{\text{var}});
    // Set up nodes for recursion
    if (\( F_1 \) is non-terminal \( \land \) var = \( F_1^{\text{var}} \)) then
      \( F_1^{v1} := F_1, l; \)\( F_1^{v1} := F_1, h; \)
    else
      \( F_1^{v1} := F_1; \)
    if (\( F_2 \) is non-terminal \( \land \) var = \( F_2^{\text{var}} \)) then
      \( F_2^{v2} := F_2, l; \)\( F_2^{v2} := F_2, h; \)
    else
      \( F_2^{v2} := F_2; \)
    // Recurse and get cached result
    \( F_l := \text{Apply}(F_1^{v1}, F_2^{v2}, \text{op}); \)
    \( F_h := \text{Apply}(F_1^{v1}, F_2^{v2}, \text{op}); \)
    \( F_r := \text{GetNode}(\text{var}, F_h, F_l); \)
    // Put result in apply cache and return
    insert \( \langle F_1, F_2, \text{op} \rangle \rightarrow F_r \) into apply cache;
  return \( F_r \);
end
\end{verbatim}
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
  \[ \sum_x F(x, \ldots) = F|_{x=0} \oplus F|_{x=1}, \text{ e.g.} \]

- 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

\[
\begin{align*}
  x_1 \oplus x_2 \oplus \ldots \oplus x_n
\end{align*}
\]

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

\[
\begin{align*}
  x_1 + (x_2 + 4x_3) \times (x_4) \\
  \rightarrow x_1 \oplus (x_2 \oplus (4 \otimes x_3)) \otimes (x_4)
\end{align*}
\]
Apply Tricks II

• Build *ordered* DD from *unordered* DD

z is out of order

result will have z in order!
ZDDs
(zero-suppressed BDDs)

Represent sets of subsets
ZDDs for Sets of Subsets

• Example BDD and ZDD

![Diagram showing BDD and ZDD for sets of subsets]

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

An Introduction to Zero-Suppressed Binary Decision Diagrams
Alan Mishchenko
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 $x$
  – $F\setminus x$ is sub-ZDD for set without $x$

• $F \cap G$:
  – if (x in set)
    • then $F_x \cap G_x$
    • else $F\setminus x \cap G\setminus x$

• 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{align*}
\text{Semantics: } \text{if} \ (a) \ \text{then} \ (c_1 + b_1 F_1) \ \text{else} \ (c_2 + b_2 F_2)
\end{align*}
```
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>((c_1 + c_2 + b_1 c_r) + b_1 b_r F_r)</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>((c_1 - c_2 + b_1 c_r) + b_1 b_r F_r)</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 (c_1/b_1) + F_1 \rangle \oplus \langle (c_2/b_2) + F_2 \rangle)</td>
<td>((b_1 b_2 c_r + b_1 b_2 b_r F_r))</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)</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>((c_1 + b_1 c_r) + b_1 b_r F_r)</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>(c_r + b_r F_r)</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)
ADDs vs. AADDs

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

Pairwise factoring evident in AADD 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

![Diagram of a multivalued DD with nodes ID1, ID2, ..., ID6 and branches R=1, R=2, ..., R=6.]

Obvious generalizations to Apply and Reduce
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
Further Afield

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

- **FODDs, FOADDs**
  - Support first-order logical decision tests
    (Wang, Joshi, Khardon, JAIR-08)
    (Sanner, Boutilier, AIJ-09)

- **XADDs: continuous variables →**
  (Sanner, UAI-11)
Teaser: XADD Maximization

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

May introduce new decision tests
Approximation

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

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

- 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

For operations, see “interval arithmetic”:
http://en.wikipedia.org/wiki/Interval_arithmetic
More Compactness? AADDs?

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

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

• Merge ≈ nodes from bottom up:

```
    (0 + 7.11 *)
      /
     /
x1  <0 + 0.852 * >  <0.142 + 0.858 * >
      /
     /
x2  <0.332 + 0.668 * >  <0 + 0.665 * >
      /
     /
x3  <0 + 0 * >  <1 + 0 * >
    0
```
Decision Diagram Software

Work with decision diagrams in < 1 hour!
Software Packages

- **CUDD**
  - BDD / ADD / ZDD
  - [http://vlsi.colorado.edu/~fabio/CUDD/](http://vlsi.colorado.edu/~fabio/CUDD/)
  - Hands down, the best package available

- **JavaBDD (native interface to CUDD / others):**

- **NuSMV – Model Based Planner (MBP)**

- **SPUDD – ADD-based value iteration for MDPs**
  - [http://www.computing.dundee.ac.uk/staff/jessehoey/spudd/index.html](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**
  - [https://code.google.com/p/dd-inference/](https://code.google.com/p/dd-inference/)
  - Scott’s code, not high performance, but functional
  - Includes Java version of SPUDD factored MDP solver & variable elimination
Example Applications using Decision Diagrams

Do they really work well?
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:
    \[ P(a|b,c) = \text{if } b \ ? \ 1 : \text{if } c \ ? \ .7 : .3 \]
  – Additive CPTs
    • Probability is discretized linear function:
    \[ P(a|b_1 \ldots b_n) = c + b \cdot \sum_i 2^{i} b_i \]
  – Multiplicative CPTs
    • Noisy-or (multiplicative AADD):
    \[ P(e|c_1 \ldots c_n) = 1 - \prod_i (1 - p_i) \]

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

<table>
<thead>
<tr>
<th>Bayes Net</th>
<th>Table</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)
Application: MDP Solving

- SPUDD Factored MDP Solver (Hoey et al, 99)
  - Originally uses ADDs, can use AADDs
  - Implements factored value iteration…

\[ V^{n+1}(x_1 \ldots x_i) = R(x_1 \ldots x_i) + \gamma \cdot \max_a \sum_{x_1' \ldots x_i'} \prod_{F1 \ldots Fi} P(x_1' \mid x_1 \ldots x_i) \ldots P(x_i' \mid x_1 \ldots x_i) \cdot V^n(x_1' \ldots x_i') \]
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

[Bar chart showing comparison between APRICODD and MADCAP for time and space across different scenarios involving 20 and 24 variables, exact and approximate methods.]
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>
Inference with Decision Diagrams vs. Compilations (d-DNNF, etc.)

Important Distinctions
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)

Definitions / Diagrams from “A Knowledge Compilation Map”, Darwiche and Marquis. JAIR 02
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

Definitions / Diagrams from “A Knowledge Compilation Map”, Darwiche and Marquis. JAIR 02

Ordered BDD, in previous slides I call this a BDD

Children inherit polytime operations of parents

Size of children \(\geq\) parents

<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>

Table 4: Notations for queries.
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.