MDD-Based Constraint Programming in Haddock

Laurent Michel (University of Connecticut)
Willem-Jan van Hoeve (Carnegie Mellon University)

CP 2022 Tutorial

Agenda

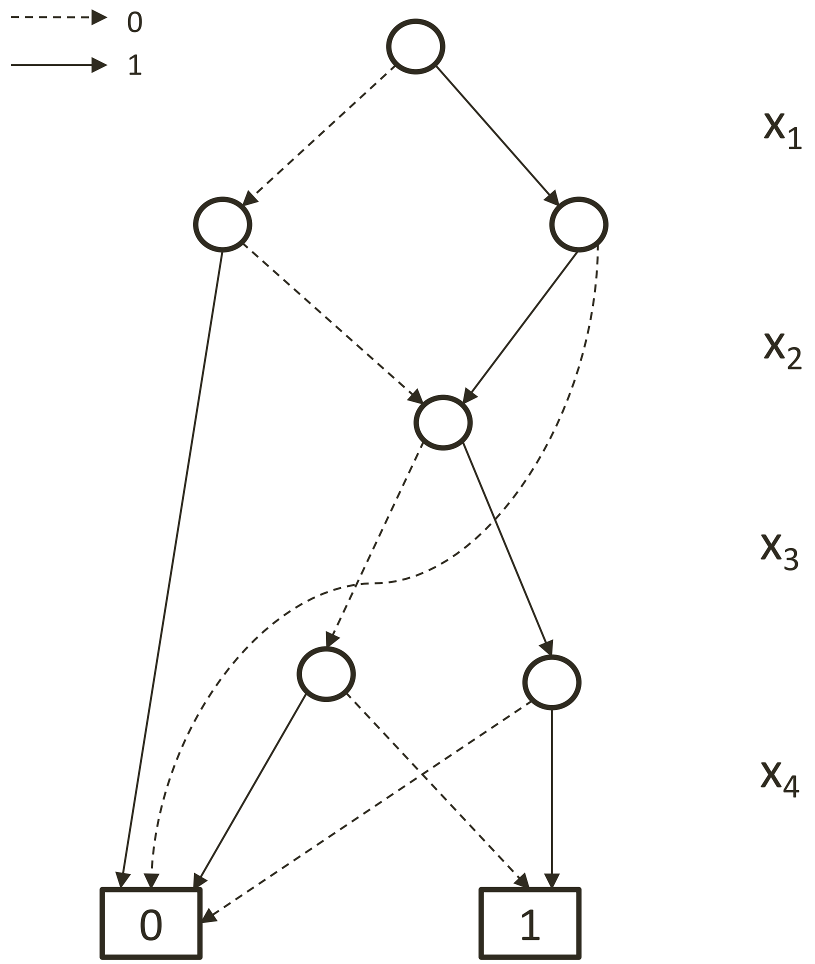
- Decision Diagrams: Background
- Constraint Programming with Decision Diagrams
- Decision Diagrams within Constraint Programming Solvers
- Applications

Decision Diagrams

- Graphical representation of **Boolean functions**

$$f(x) = \left(x_1 \Leftrightarrow x_2\right) \wedge \left(x_3 \Leftrightarrow x_4\right)$$

x_1	x_2	x_3	x_4	**f(x)**
0	0	0	0	**1**
0	0	0	1	**0**
0	1	1	0	**0**
0	0	1	1	**1**
…	…	…	…	**…**

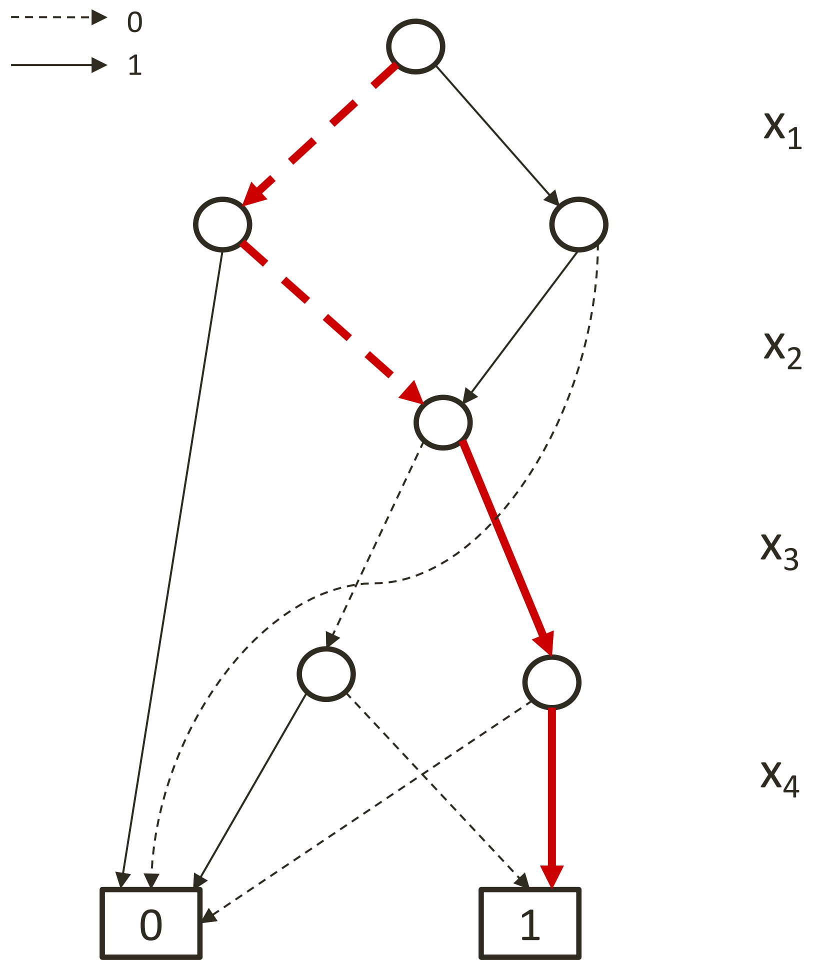


Decision Diagrams

- Graphical representation of **Boolean functions**

$$f(x) = \left(x_1 \Leftrightarrow x_2\right) \wedge \left(x_3 \Leftrightarrow x_4\right)$$

x_1	x_2	x_3	x_4	**f(x)**
0	0	0	0	**1**
0	0	0	1	**0**
0	1	1	0	**0**
0	**0**	**1**	**1**	**1**
…	…	…	…	**…**

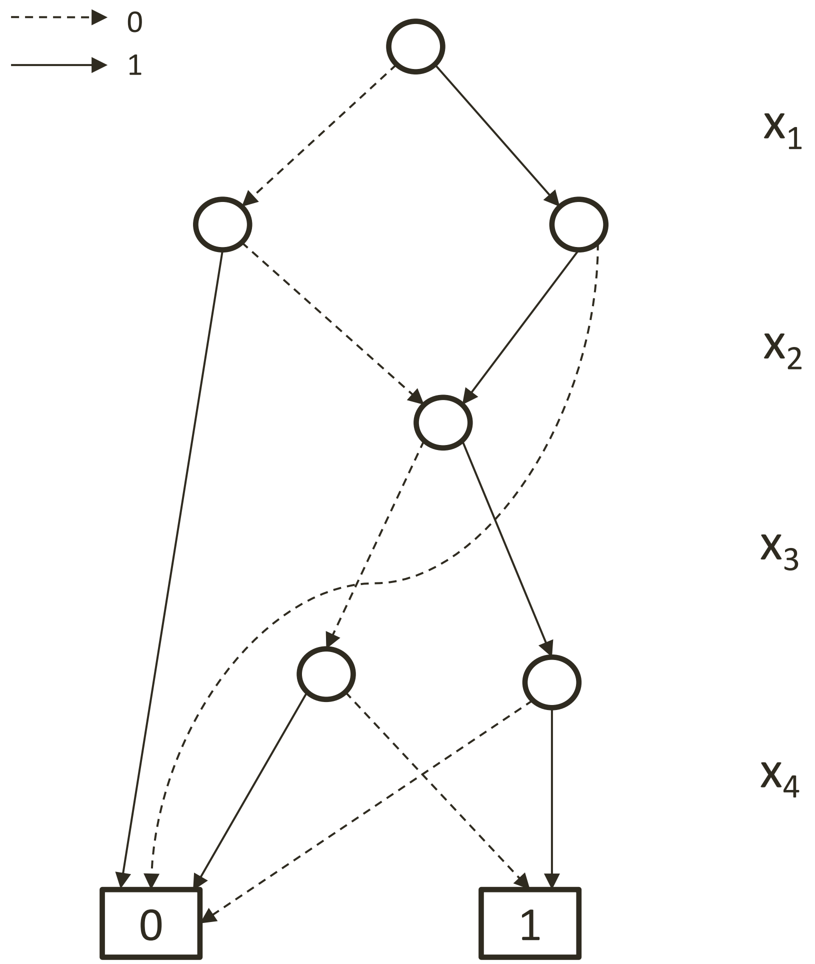


Decision Diagrams

- Graphical representation of **Boolean functions**

$$f(x) = \left(x_1 \Leftrightarrow x_2\right) \wedge \left(x_3 \Leftrightarrow x_4\right)$$

- BDD: binary decision diagram
- MDD: multi-valued decision diagram



Brief Historic Background

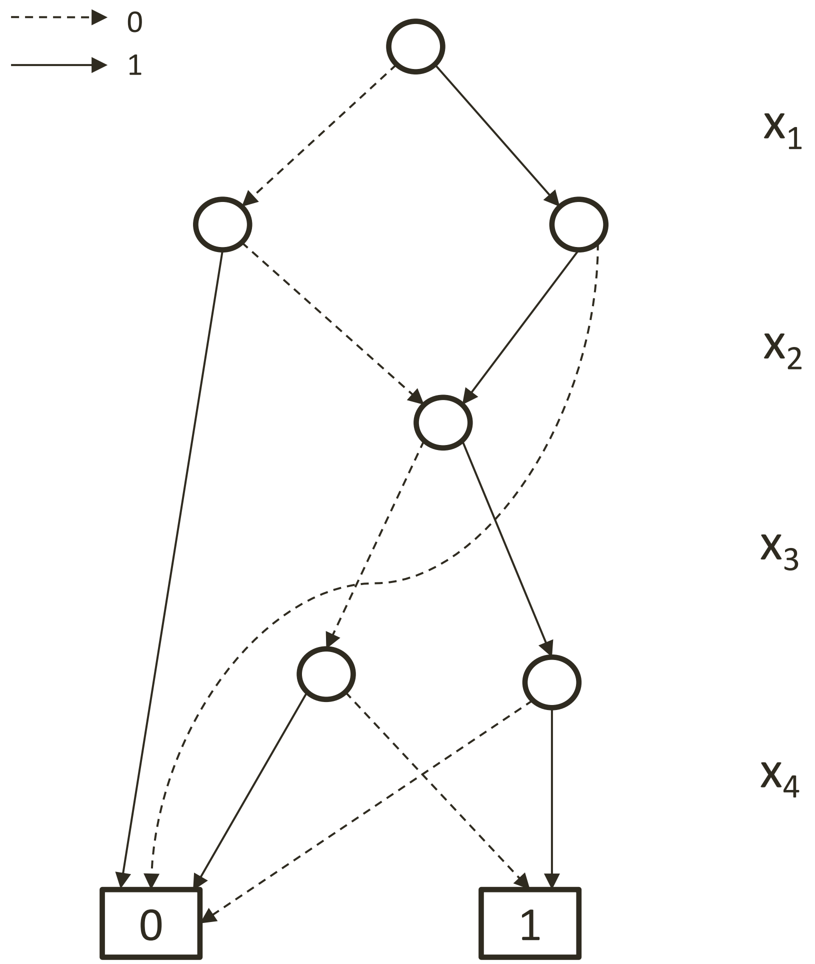
- Widely used in computer science [Lee, 1959; Akers, 1978; Bryant, 1986]
  - original application areas: circuit design, verification
- Usually *reduced ordered* BDDs/MDDs are applied
  - fixed variable ordering; minimal exact representation
- First applications to discrete optimization problems
  - BDD-based IP solver [Lai et al., 1994]
  - set bounds propagation in CP [Hawkins, Lagoon, Stuckey, 2005]
  - IP cut generation [Becker et al., 2005] [Behle & Eisenbrand, 2007] [Behle, 2007]
  - post-optimality analysis [Hadzic & Hooker, 2006, 2007]
- *Relaxed* Decision Diagrams [Andersen, Hadzic, Hooker & Tiedemann, CP 2007]

Decision Diagrams: Optimization View

- Graphical representation of **Boolean functions**

$$f(x) = \left(x_1 \Leftrightarrow x_2\right) \wedge \left(x_3 \Leftrightarrow x_4\right)$$

- Optimization perspective:
  - literals → variables
  - arcs → assignments
  - paths → solutions



Decision Diagrams: Optimization View

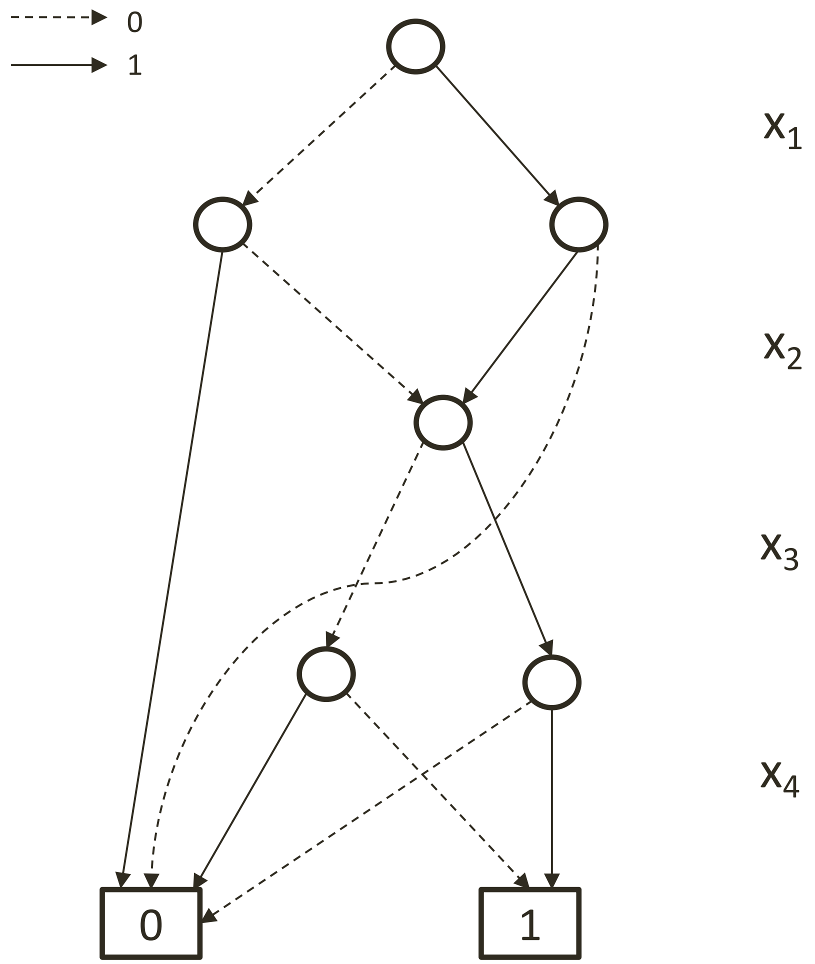
max $2x_1 + x_2 - 4x_3 + x_4$

subject to

$$x_1 - x_2 = 0$$

$$x_3 - x_4 = 0$$

$$x_1, x_2, x_3, x_4 \in \{0,1\}$$



Decision Diagrams: Optimization View

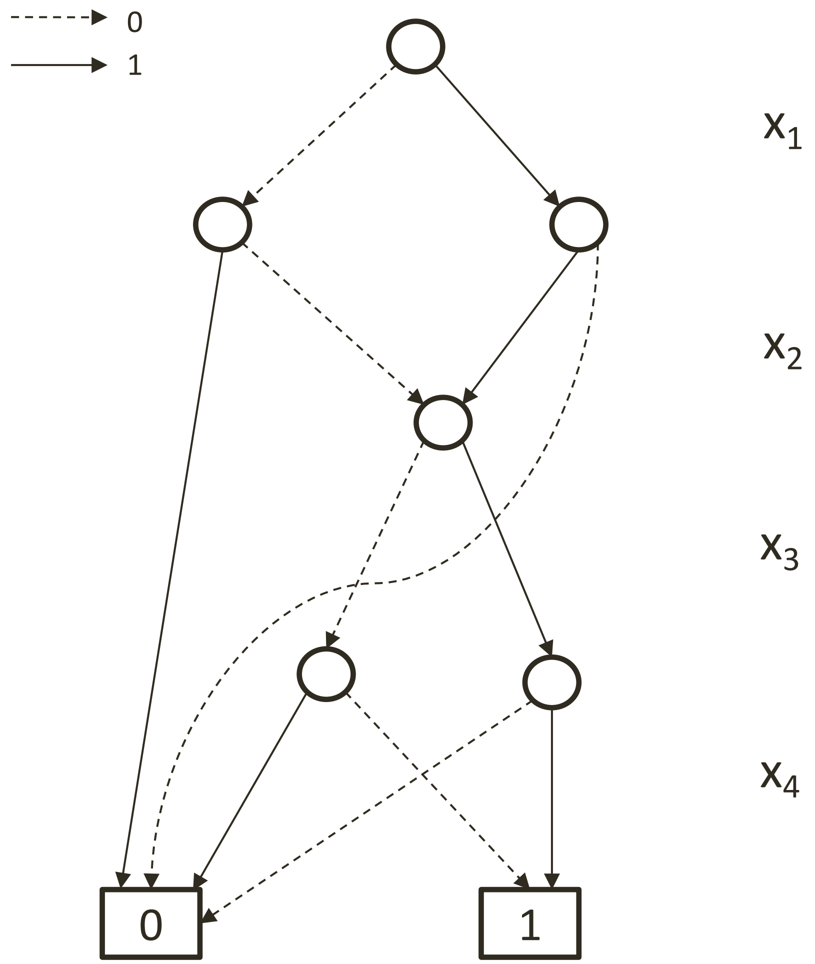
$\max \quad 2x_1 + x_2 - 4x_3 + x_4$

subject to

$$x_1 - x_2 = 0$$

$$x_3 - x_4 = 0$$

$$x_1, x_2, x_3, x_4 \in \{0,1\}$$



Decision Diagrams: Optimization View

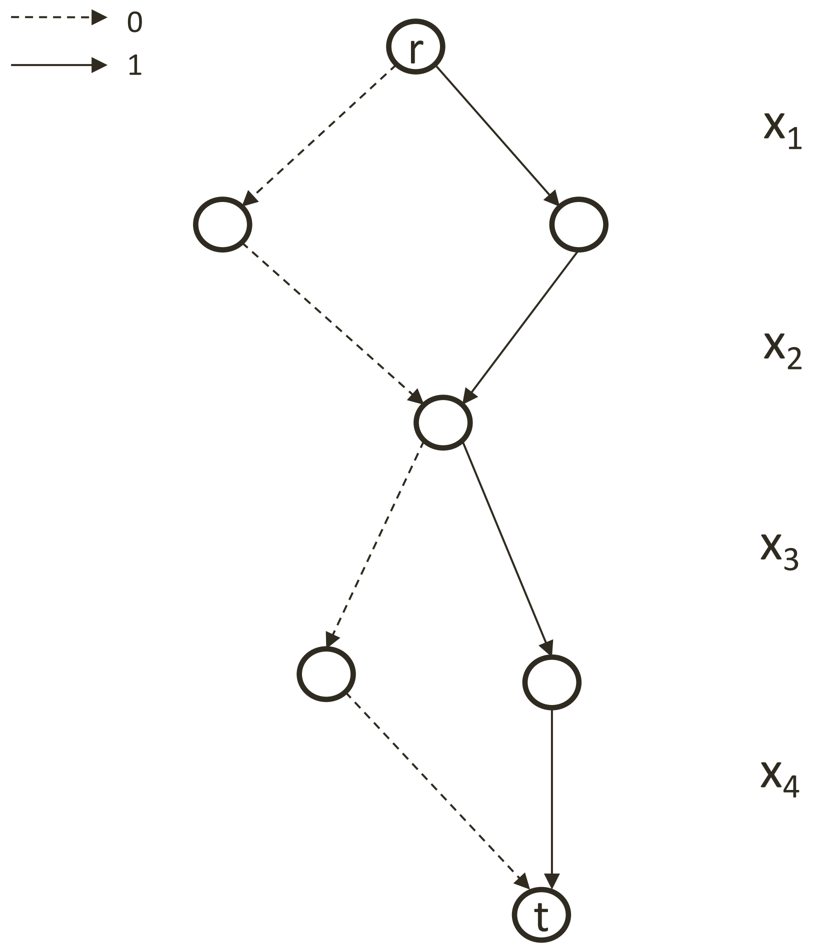
max $2x_1 + x_2 - 4x_3 + x_4$

subject to

$x_1 - x_2 = 0$

$x_3 - x_4 = 0$

$x_1, x_2, x_3, x_4 \in \{0,1\}$



Decision Diagrams: Optimization View

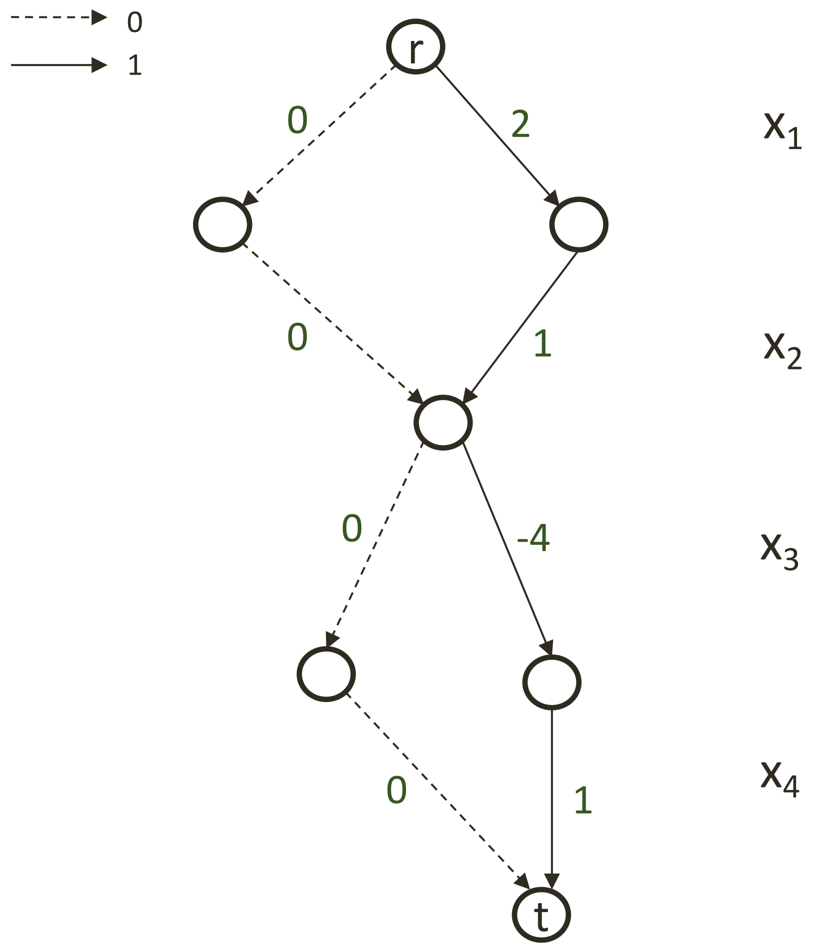
max $2x_1 + x_2 - 4x_3 + x_4$

subject to

$$x_1 - x_2 = 0$$

$$x_3 - x_4 = 0$$

$$x_1, x_2, x_3, x_4 \in \{0,1\}$$



- Maximizing a linear (or separable) function:
  - Arc lengths: contribution to the objective
  - Longest path: optimal solution

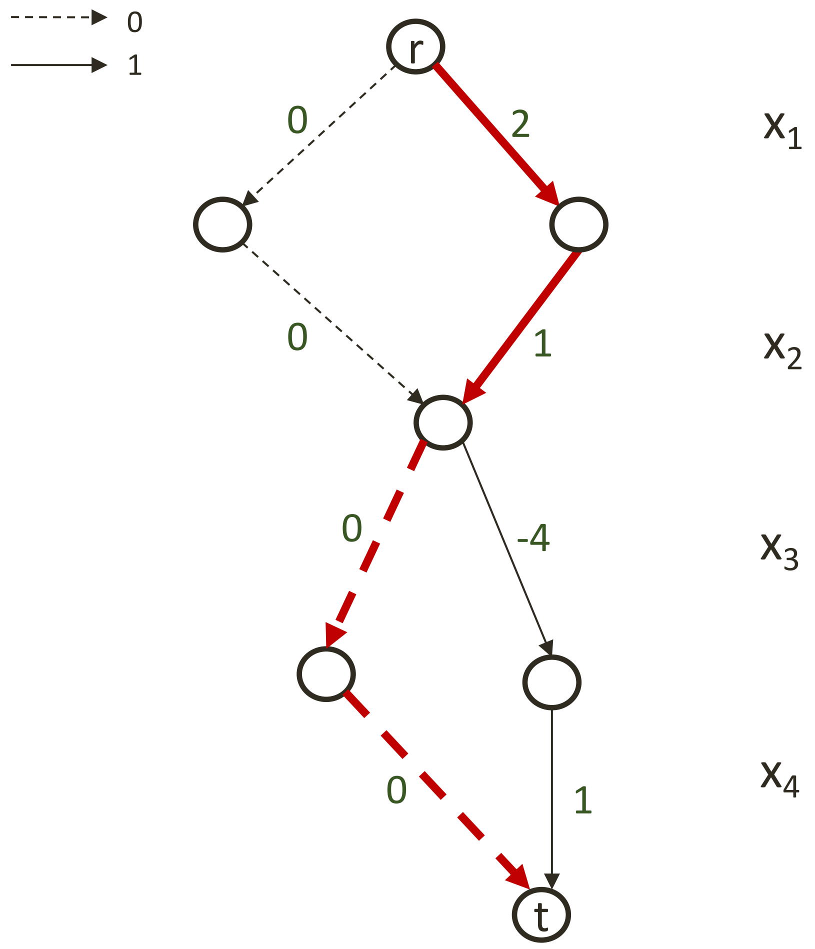
Decision Diagrams: Optimization View

max $2x_1 + x_2 - 4x_3 + x_4$
subject to

$x_1 - x_2 = 0$

$x_3 - x_4 = 0$

$x_1, x_2, x_3, x_4 \in \{0,1\}$



- Maximizing a linear (or separable) function:
  - Arc lengths: contribution to the objective
  - Longest path: optimal solution

  (MDD can also handle nonlinear functions)

Categories of Successful Applications

- Constraint Programming
  - DD-based constraint propagation
- Combinatorial optimization
  - MISP, MAX-CUT, graph coloring,…
- Scheduling, routing, planning
  - machine scheduling, TSPTW, SOP, AI robotic planning,…
- Decomposition and embedding in MIP
  - nonlinear objective functions, cutting planes, column generation,…

[Andersen et al. CP2007] [Hoda et al. CP2010] [Bergman, Cire, and/or vH, 2013-2022] [Perez&Régin 2015-2018] [Coppé et al., CP 2022] [Verhaeghe et al. IJCAI 2018, CPAIOR 2019] [Gentzel et al. CP 2020, 2022]

[Bergman, Cire, vH, Hooker, 2011-2016] [Gillard et al., IJCAI 2020] [vH, MP 2022] [Karahalios&vH, 2022] [Coppé et al., CP 2022]

[Cire&vH, OR2013], [Kinable et al. EJOR 2017] [O'Neil&Hoffman, ORL2019] [Bogaerdt&de Weerdt, 2019] [Gillard&Schaus, IJCAI2022] [Rudich et al. CP 2022] [Castro et al. 2019-2022] [Horn et al. 2019-2021]

[Bergman&Cire 2018] [Lozano et al. 2020-2022] [Morrison et al. IJOC 2016] [Kowalczyk & Leus IJOC 2018] [Tjandraatmadja&vH, 2019, 2021] [Davarnia&vH, MP 2021]

Excellent survey paper: Castro, Cire & Beck [IJOC 2022]

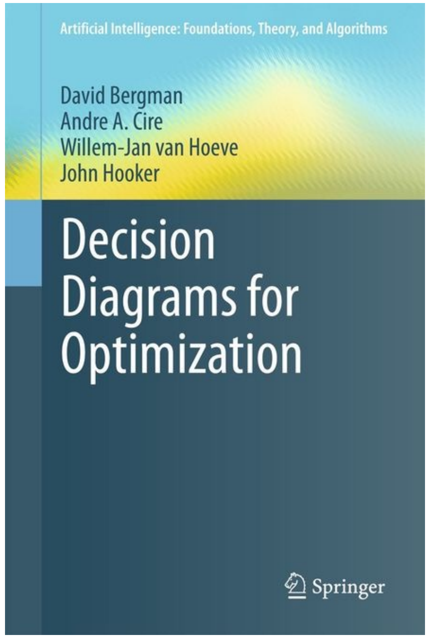
Systems for Decision Diagrams

- DDO: Gillard, Schaus & Coppé [IJCAI 2021]
  – Generic implementation of DD-based branch-and-bound
- Nextmv: Carolyn Mooney & Ryan O'Neil
  – https://www.nextmv.io/
  – Industrial vehicle routing system (MDD-based)
- Haddock: Gentzel, Michel, vH [CP 2020]
  – Generic implementation of MDD-based CP

MDD-Based Constraint Programming

Papers that are most relevant for this tutorial:

- Andersen, Hadzic, Hooker, and Tiedemann. A constraint store based on multivalued decision diagrams. CP 2007. LNCS 4741:118-132.
- Hoda, vH, and Hooker. A systematic approach to MDD-based constraint programming. CP 2010. LNCS 6308: 266-280.
- Gentzel, Michel and vH. HADDOCK: A Language and Architecture for Decision Diagram Compilation. CP 2020. LNCS 12333:531-547.
- Chapters 9-11 of “Decision Diagrams for Optimization” by Bergman, Cire, vH, Hooker. Springer 2016.



Constraint Programming with Decision Diagrams

Motivation

- Constraint Programming applies constraint propagation
  - Remove provably inconsistent values from variable domains
  - Propagate updated domains to other constraints

$\boxed{x_1 > x_2}$

$\boxed{x_1 + x_2 = x_3}$

$\boxed{\textit{alldifferent}(x_1,x_2,x_3\,,x_4)}$

$x_1 \in \{\cancel{1},2\}$, $x_2 \in \{\cancel{0},1,\cancel{2},\cancel{3}\}$, $x_3 \in \{\cancel{2},3\}$, $x_4 \in \{0,\cancel{1}\}$

domain propagation
can be weak, however…

Illustrative example

$alldifferent(x_1,x_2,x_3,x_4)$ (1)

$x_1 + x_2 + x_3 \geq 9$ (2)

$x_i \in \{1,2,3,4\}$

(1) and (2) are both domain consistent

(i.e., no propagation)

List of all solutions to *alldifferent*:

x_1	x_2	x_3	x_4
~~1~~	~~2~~	~~3~~	~~4~~
~~1~~	~~2~~	~~4~~	~~3~~
~~1~~	~~3~~	~~2~~	~~4~~
	…		
4	3	2	1

Suppose we could evaluate (2) on this list

domain projection: $D(x_i) = \{1,2,3,4\}$

Illustrative example

alldifferent(x_1,x_2,x_3,x_4) (1)

$x_1 + x_2 + x_3 \geq 9$ (2)

$x_i \in \{1,2,3,4\}$

(1) and (2) are both domain consistent

(i.e., no propagation)

List of all solutions to *alldifferent*:

	x_1	x_2	x_3	x_4
✓	2	3	4	1
✓	2	4	3	1
✓	3	2	4	1
		…		
✓	4	3	2	1

Suppose we could evaluate (2) on this list

domain projection: $D(x_4) = \{1\}$

$D(x_1) = D(x_2) = D(x_3) = \{2,3,4\}$

Illustrative example

$alldifferent(x_1,x_2,x_3,x_4)$ (1)

$x_1 + x_2 + x_3 \geq 9$ (2)

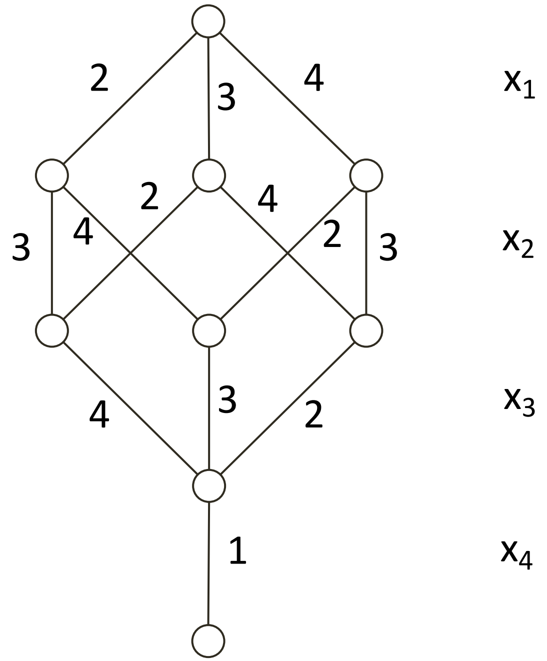
$x_i \in \{1,2,3,4\}$

(1) and (2) are both domain consistent

(i.e., no propagation)

List of all solutions to *alldifferent*:

	x_1	x_2	x_3	x_4
✓	2	3	4	1
✓	2	4	3	1
✓	3	2	4	1
		…		
✓	4	3	2	1



Use an MDD!

Motivation for MDD propagation

- Conventional domain propagation: all structural relationships among variables are lost after domain projection
- Potential solution space is implicitly defined by Cartesian product of variable domains (very coarse relaxation)

We can communicate more information between constraints using MDDs [Andersen et al. 2007]

- Explicit representation of more refined potential solution space
- Limited width defines *relaxed* MDD
- Strength is controlled by the imposed width

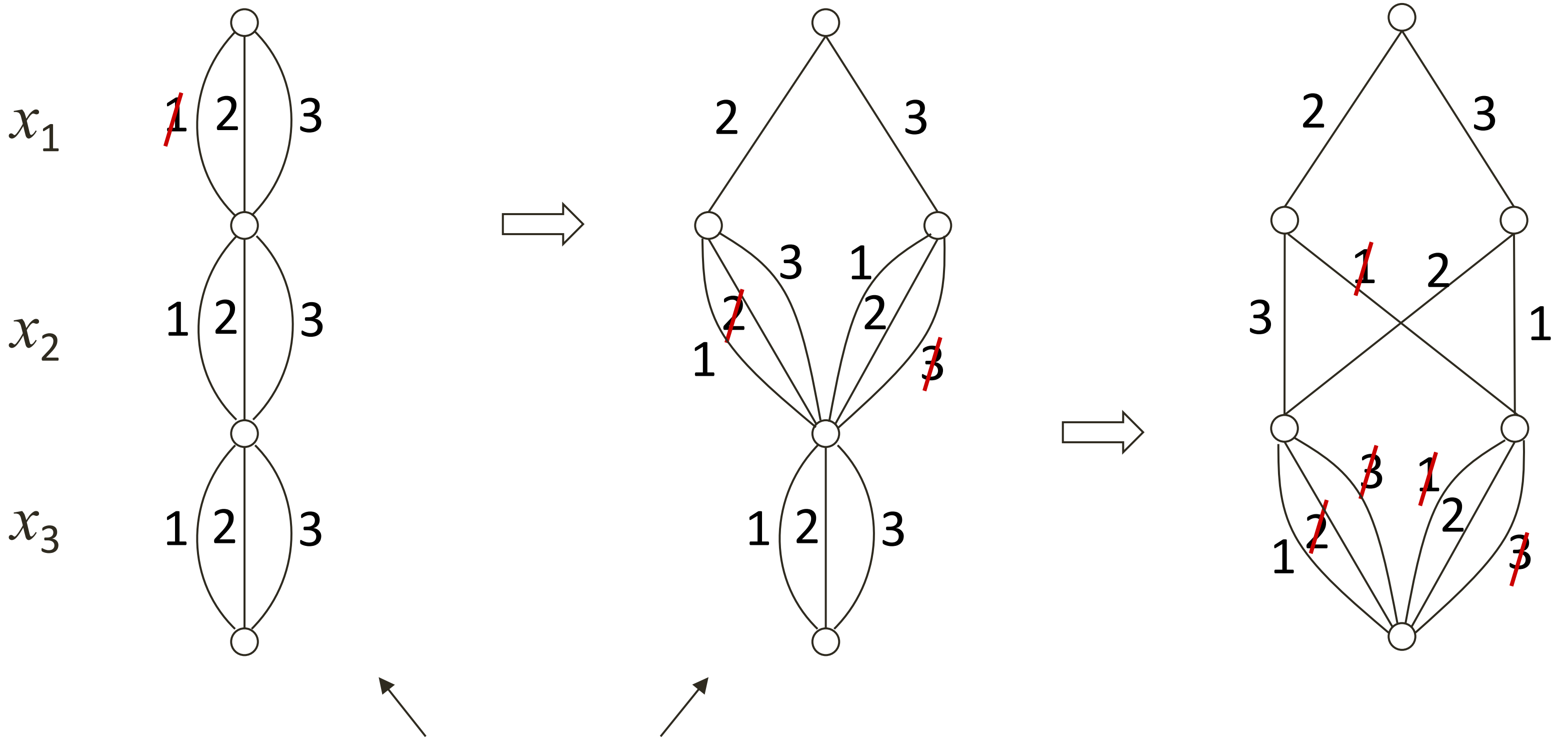
MDD-based Constraint Programming

- Maintain limited-width MDD
  - Serves as relaxation
  - Typically start with width 1 (initial variable domains)
  - Dynamically adjust (refine) the MDD, based on constraints
- Constraint Propagation
  - Arc filtering: Remove provably inconsistent arcs (those that do not participate in any solution)
  - Node refinement: Split nodes to separate information carried by the incoming arcs
- Search
  - As in classical CP, but may now be guided by MDD

Example of Top-Down Iterative MDD Refinement

alldifferent(x_1, x_2, x_3)

$x_1 > x_3$



relaxed MDDs
(strength is controlled by maximum width)

exact MDD

Characterization of Propagation

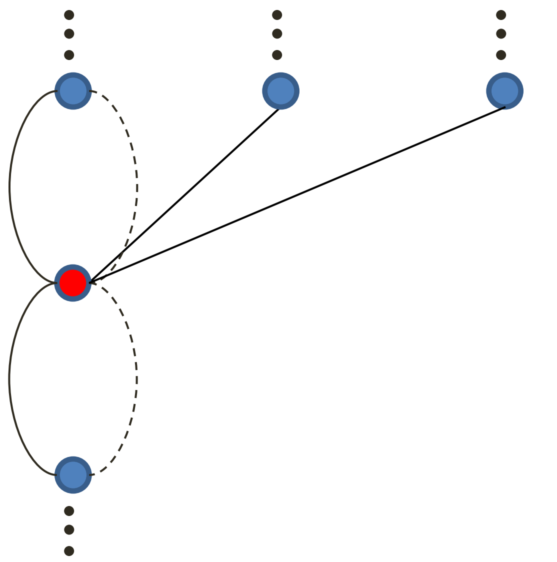
Domain consistency generalizes naturally to MDDs:

- Let C(X) be a constraint on variables X and let M be an MDD on X
- Constraint C is **MDD consistent** if for each arc in M, there is at least one path in M that represents a solution to C

Equivalent to domain consistency for MDD of width 1

Constraint Representation in MDDs

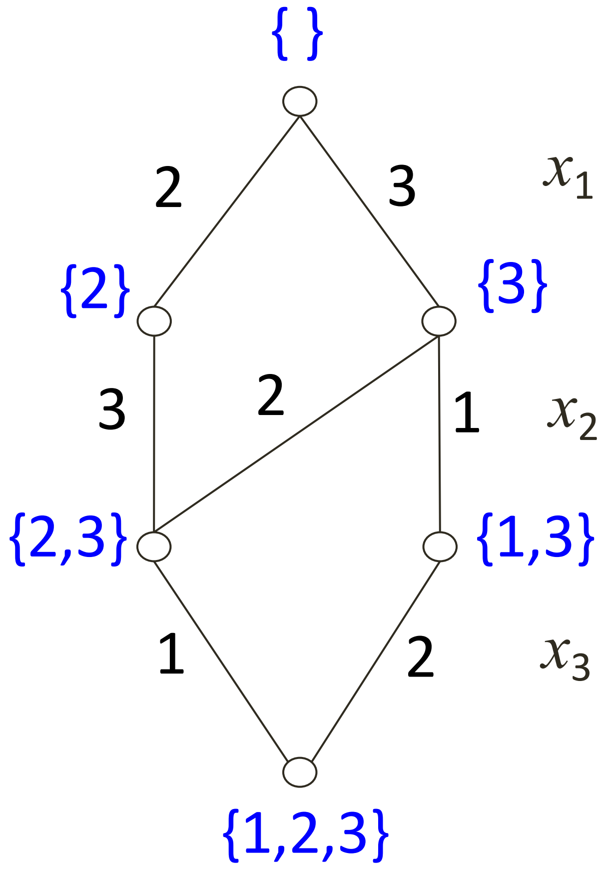
- For a given constraint type we maintain specific '**state information**' at each node in the MDD
- In Haddock, these are called '**Properties**'
  - their type can be binary, integer, …
- Computed from incoming arcs (both from top and/or from bottom)
- State information is basis for arc filtering and for node refinement



MDD State Information: Examples

alldifferent(x_1, x_2, x_3)
$x_1 > x_3$

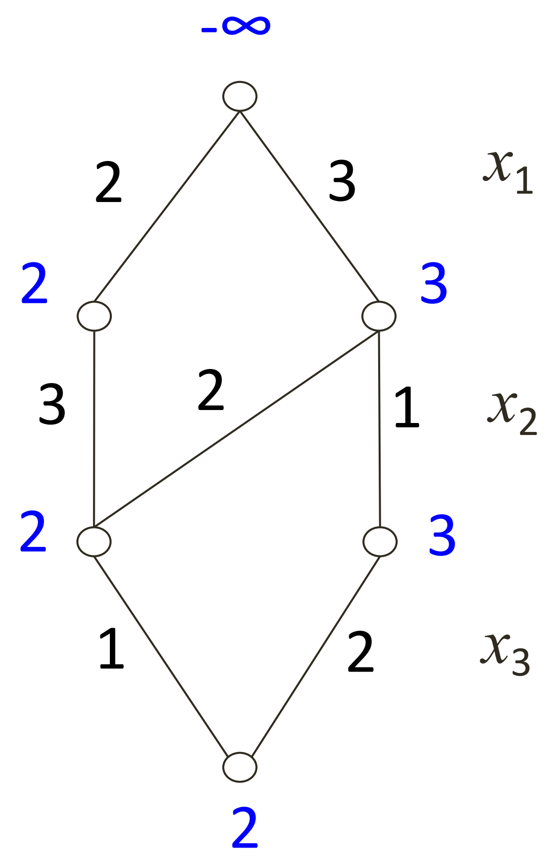
- State information for *alldifferent* constraint
  - set of values taken on paths from root (resp. terminal) to the state
- State information for linear inequality constraint



MDD State Information: Examples

alldifferent(x_1, x_2, x_3)
$x_1 > x_3$

- State information for *alldifferent* constraint
  - set of values taken on paths from root (resp. terminal) to the state
- State information for linear inequality constraint
  - minimum (resp. maximum) value along paths from root (resp. terminal) to the state



Specific MDD propagation algorithms

- Linear equalities and inequalities [Hadzic et al., 2008] [Hoda et al., 2010]
- *Alldifferent* constraints [Andersen et al., 2007] [Hoda et al., 2010]
- *Element* constraints [Hoda et al., 2010]
- *Among* constraints [Hoda et al., 2010]
- Disjunctive scheduling constraints [Hoda et al., 2010] [Cire & v.H., 2011, 2013]
- *Sequence* constraints (combination of *Amongs*) [Bergman et al., 2014]
- Generic re-application of existing domain filtering algorithm for any constraint type [Hoda et al., 2010]

First example: Among constraints

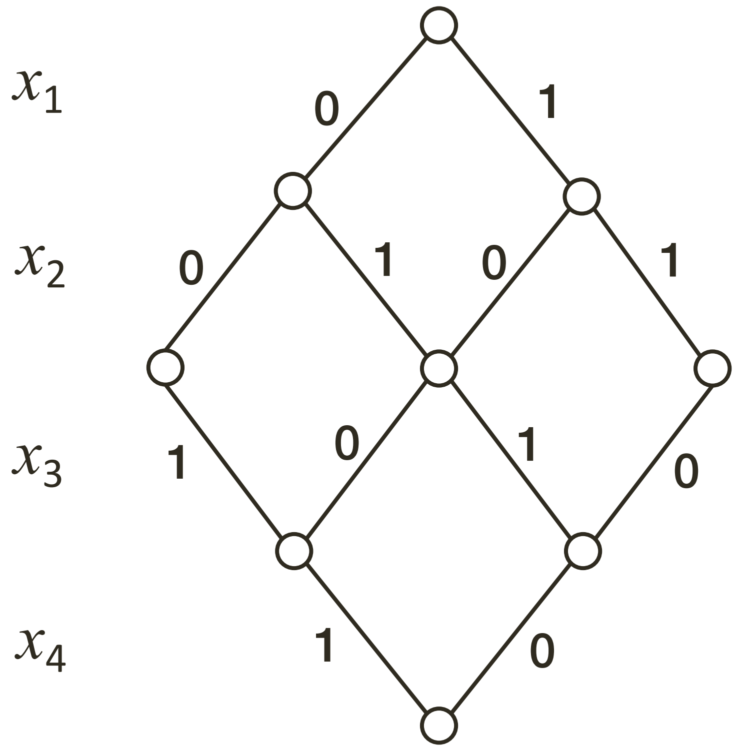
- Given a set of variables X, and a set of values *S*, a lower bound *l* and upper bound *u*,

$$\textit{Among}(\mathbf{X}, S, l, u) := \quad l \leq \sum_{x \in \mathbf{X}} (\, x \in S \,) \leq u$$

“among the variables in X, at least *l* and at most *u* take a value from the set *S*”

- Applications in, e.g., sequencing and scheduling
- WLOG assume here that X are binary and *S* = {1}

Example MDD for Among



<u>State information:</u>
path length from root
(and from terminal)

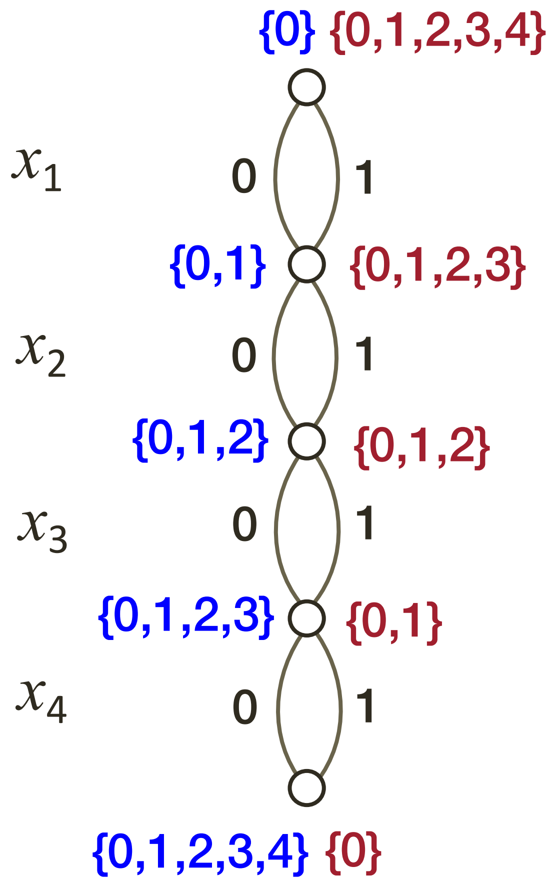
Exact MDD for *Among*($\{x_1,x_2,x_3,x_4\},\{1\},2,2$)

MDD Propagation for Among

$$\textit{Among}(\mathbf{X}, S, l, u) := l \leq \sum_{x \in X} (x \in S) \leq u$$

- Identify the state properties
  - Path lengths from root
  - Path lengths from terminal
- Remove inconsistent arcs
  - Each arc must be on a path with length between *l* and *u*
  - Ideally: Establish MDD consistency (in polynomial time)
- Refine (split nodes) if width permits to do so
  - Identify equivalence classes

MDD Propagation for Among: Example



Among($\{x_1,x_2,x_3,x_4\},\{1\},2,2$)

Identify the state properties:

- sets of path lengths

Observe that each arc is a labeled transition

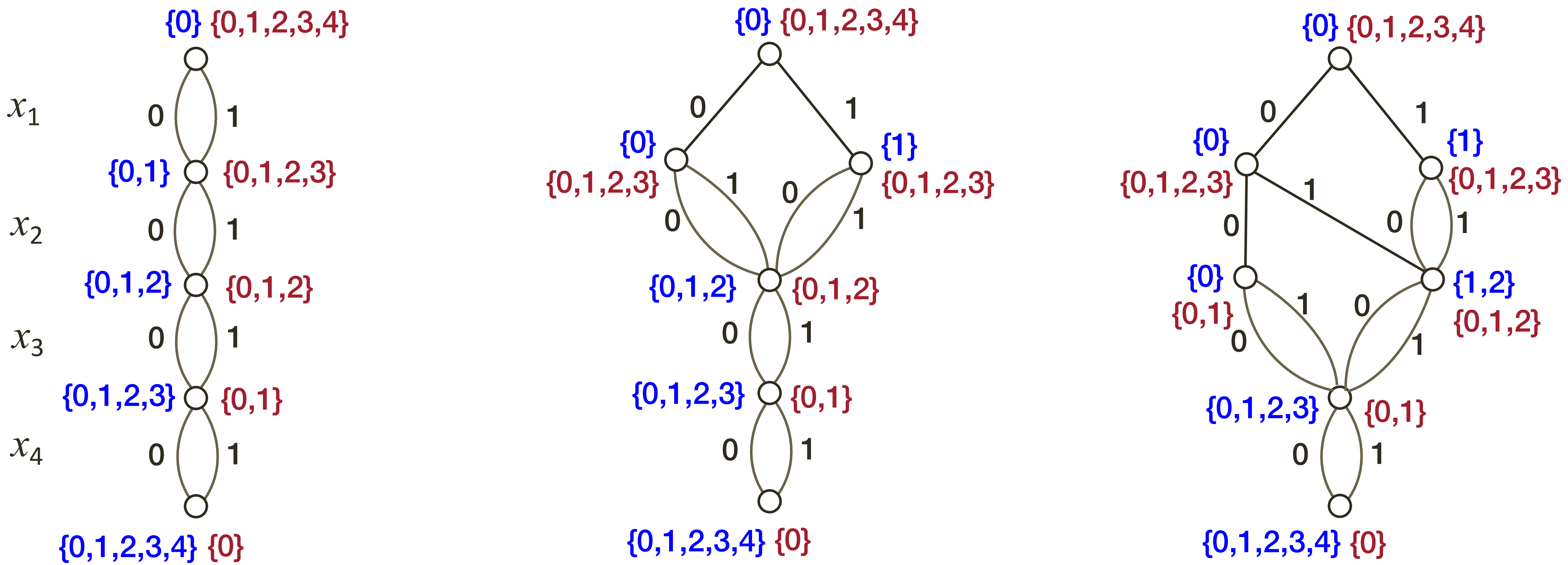
- each can lead to a different state
- the relaxed MDD *merges* their endpoints
- we need to define a "*state merging rule*" for each property (here: set union)

Node Refinement for Each Layer

For each layer in MDD, we first apply the arc filter and then try to **refine** (if the width is not yet maximal)

- consider the incoming arcs for each node
- split the node if there exist incoming arcs that are **not equivalent**
- in other words, need to identify *equivalence classes*
  - definition of equivalence class depends on the constraint, objective, …
  - it is usually effective to separate promising states
  - for *Among*: group together states with the minimum or maximum path length (this can trigger more arc filtering)

MDD Propagation for Among: Example (cont'd)



Among({x_1,x_2,x_3,x_4},{1},2,2)

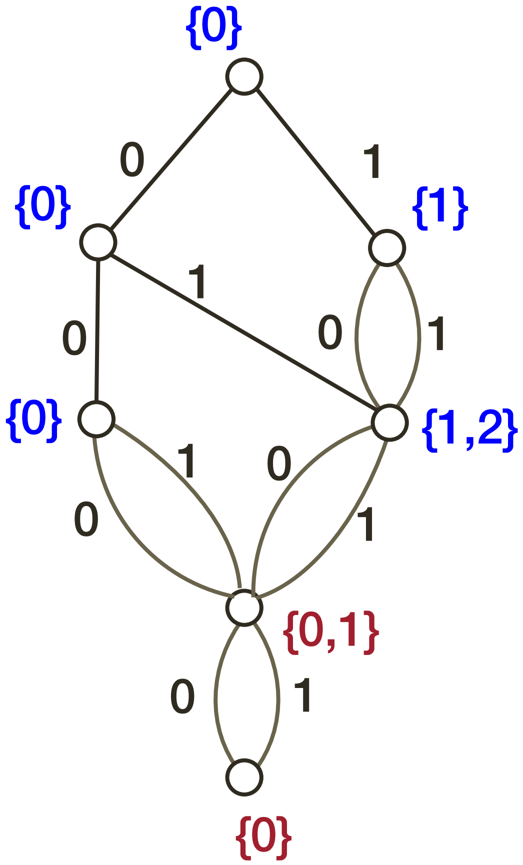
MDD Propagation for Among: Example (cont'd)

x_1

x_2

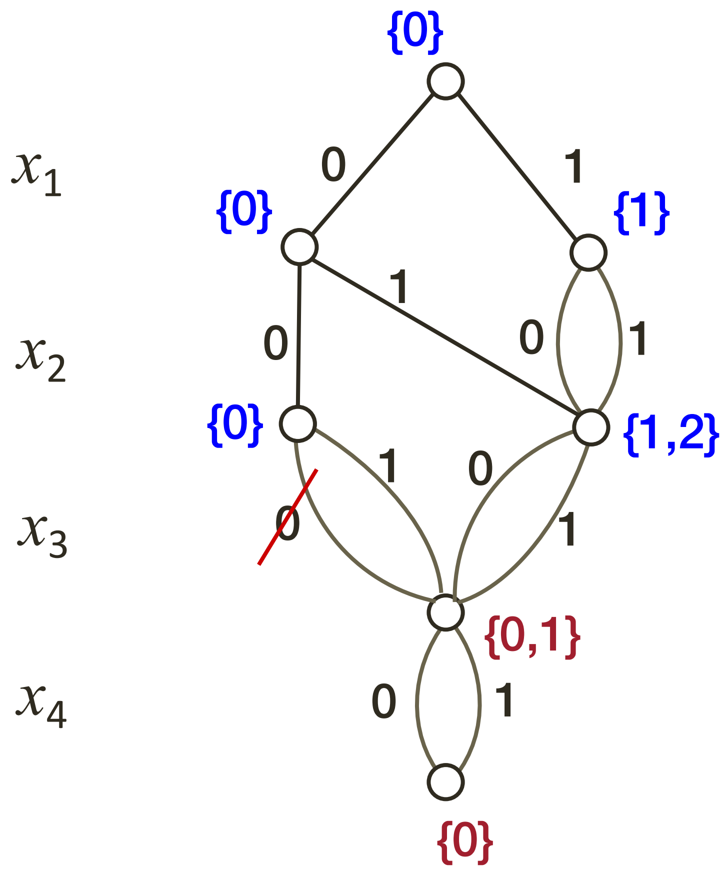
x_3

x_4



Among($\{x_1,x_2,x_3,x_4\},\{1\},2,2$)

MDD Propagation for Among: Example (cont'd)



Among($\{x_1,x_2,x_3,x_4\}$,{1},2,2)

- Remove inconsistent edges
  - Each arc must be on a path with length between *l* and *u*
- Define an *arc existence rule*
  - For *Among:*

$$l \leq \textit{path length from top} + \textit{arc value} + \textit{path length from bottom} \leq u$$

MDD Filtering for Among

Goal: Given an MDD and an *Among* constraint, remove *all* inconsistent edges from the MDD (establish MDD-consistency)

Theorem: Establishing MDD consistency for *Among* on an arbitrary MDD can be done in polynomial time [Hoda et al., CP 2010]

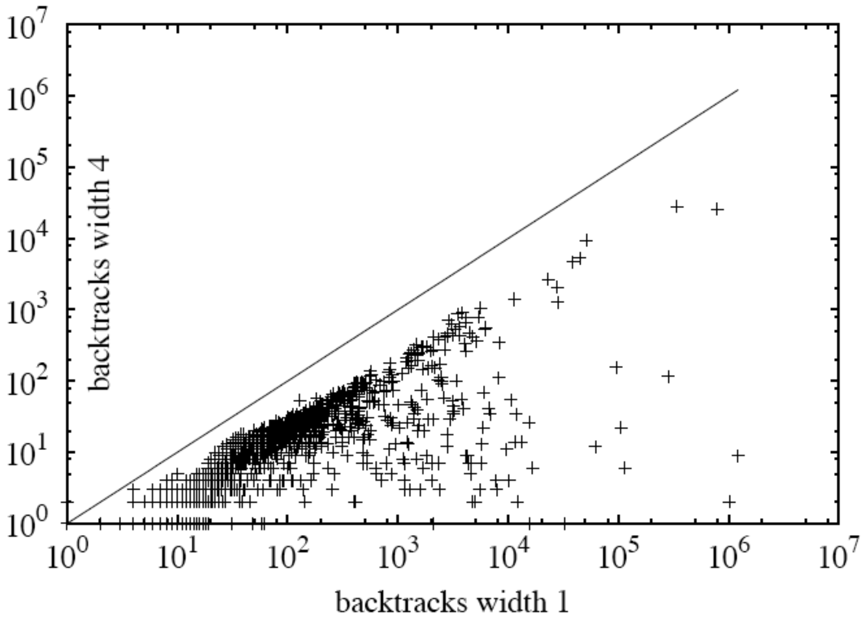
Proof:

- Compute path lengths from the root and from the terminal to each node in O(nW) time, where W is the maximum MDD width
- Complete (MDD-consistent) version: maintain all path lengths
  - Check consistency of arc takes $O(n^2)$ time
- Partial version (may not remove all inconsistent arcs)
  - Maintain and check bounds (longest and shortest paths); linear time

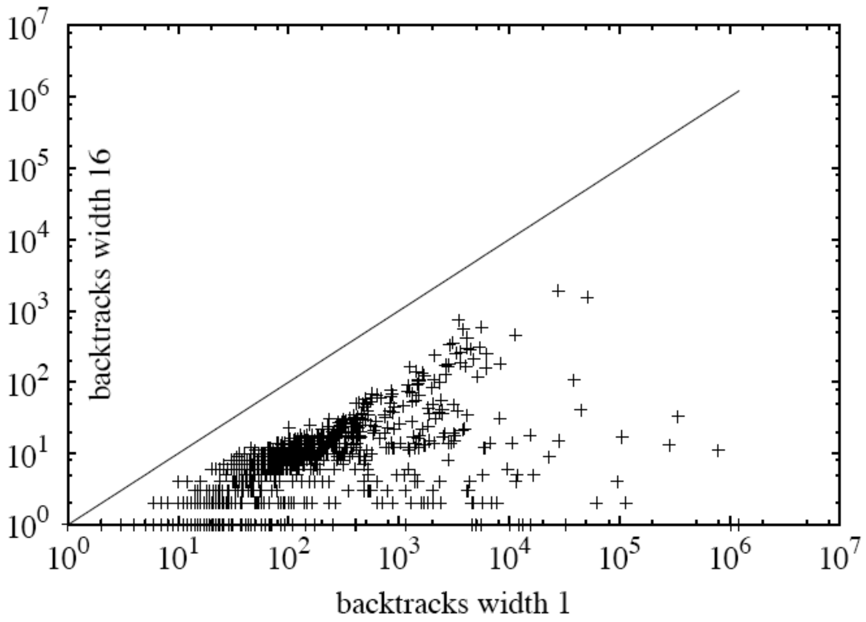
Experiments: Overlapping Among Constraints

- Multiple among constraints
  - 50 binary variables total
  - 5 variables per among constraint, indices chosen from normal distribution with uniform-random mean in [1..50] and stdev 2.5, modulo 50 (i.e., somewhat consecutive)
  - Classes: 5 to 200 among constraints (step 5), 100 instances per class
- Nurse rostering instances (horizon *n* days)
  - Work 4-5 days per week
  - Max A days every B days
  - Min C days every D days
  - Three problem classes
- Compare width 1 (traditional domains) with increasing widths

Domain vs MDD Propagation: Backtracks

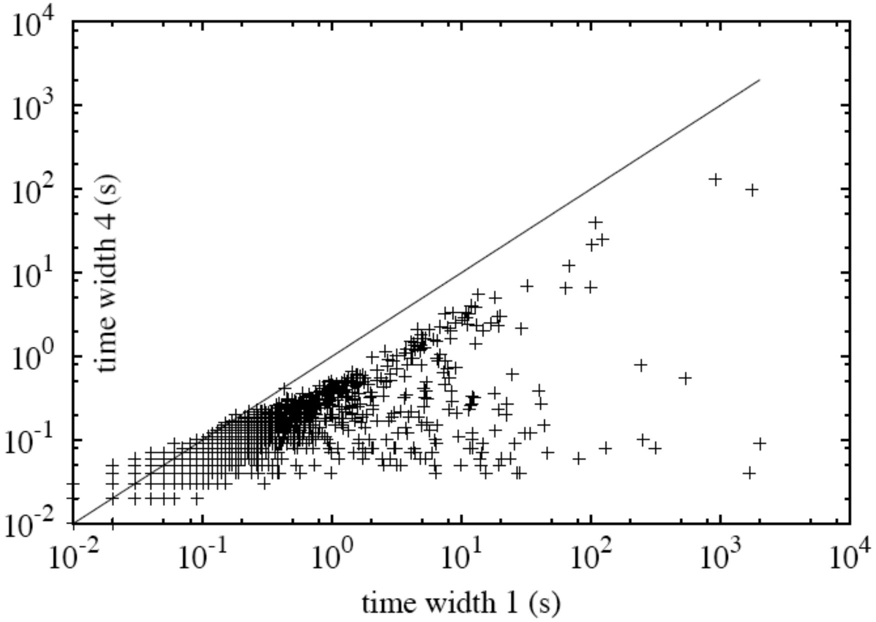


width 1 vs 4

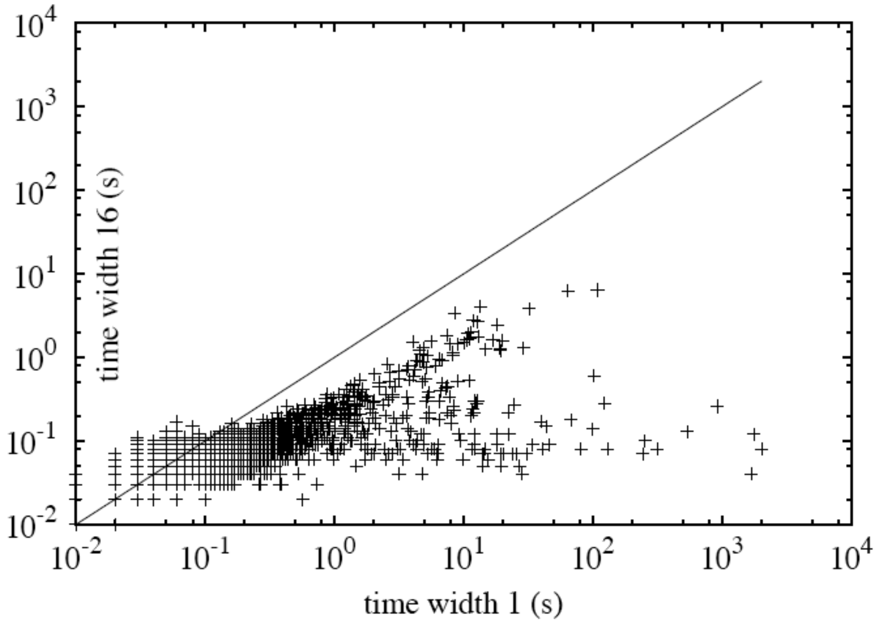


width 1 vs 16

Domain vs MDD Propagation: Time



width 1 vs 4



width 1 vs 16

Nurse rostering problems

		Width 1		Width 4		Width 32	
	Size	BT	CPU	BT	CPU	BT	CPU
Class 1	40	61,225	55.63	8,138	12.64	3	0.09
	80	175,175	442.29	5,025	44.63	11	0.72
Class 2	40	179,743	173.45	17,923	32.59	4	0.07
	80	179,743	459.01	8,747	80.62	2	0.32
Class 3	40	91,141	84.43	5,148	9.11	7	0.18
	80	882,640	2,391.01	33,379	235.17	55	3.27

Sequence Constraint

Employee must work between 2 and 7 days every 9 consecutive days

sun	mon	tue	wed	thu	fri	sat	sun	mon	tue	wed	thu
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}

$2 \le x_1 + x_2 + \ldots + x_9 \le 7$
$2 \le x_2 + x_3 + \ldots + x_{10} \le 7$
$2 \le x_3 + x_4 + \ldots + x_{11} \le 7$
$2 \le x_4 + x_5 + \ldots + x_{12} \le 7$

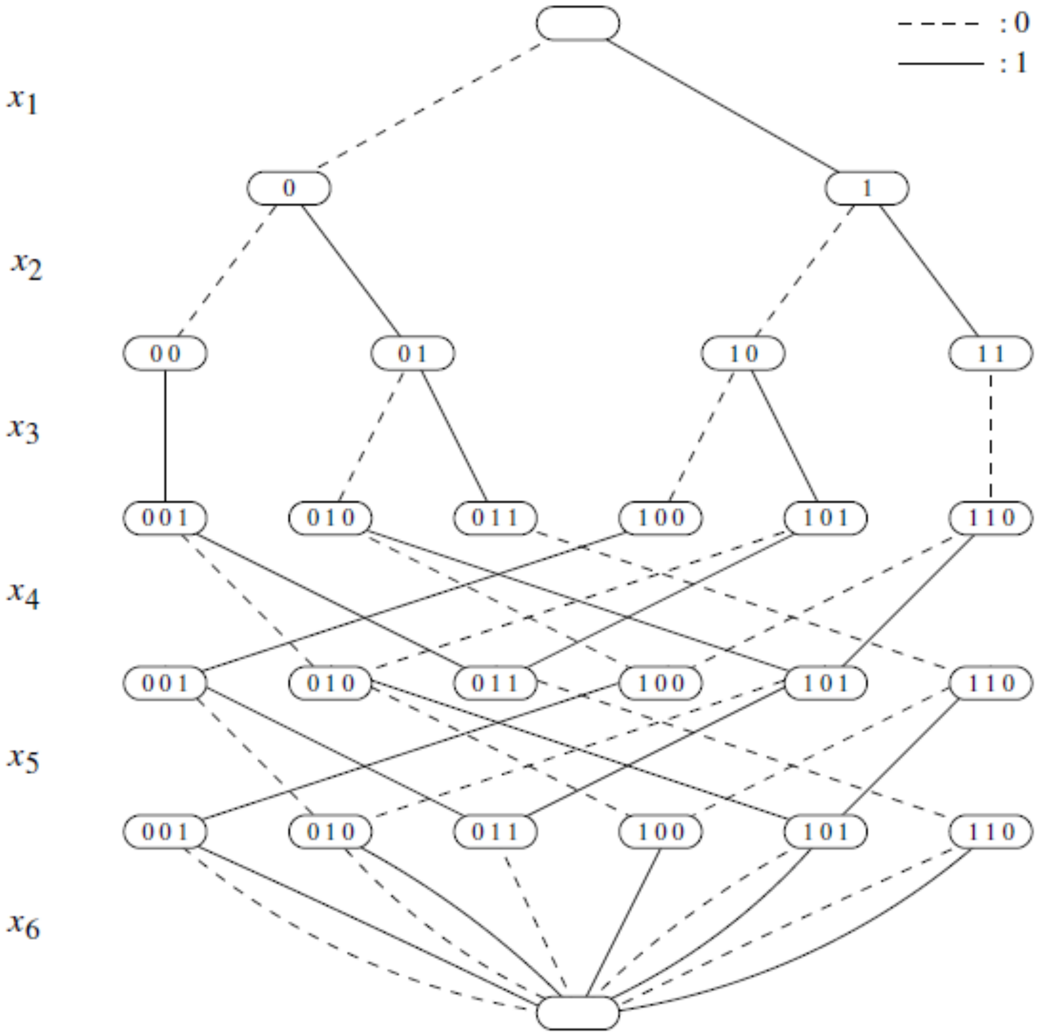
$=: \mathit{Sequence}([x_1, x_2, \ldots, x_{12}],\ q{=}9,\ S{=}\{1\},\ l{=}2,\ u{=}7)$

$$\mathit{Sequence}(X, q, S, l, u) := \bigwedge_{|X'|=q} l \le \sum_{x \in X'} (\, x \in S \,) \le u$$

$$\downarrow$$

$\mathit{Among}(X, S, l, u)$

MDD Representation for Sequence



- Similar to the DFA representation of *Sequence* for domain propagation

  [v.H. et al., 2006, 2009]

- Size $O(n2^q)$

Exact MDD for

Sequence(*X*, *q*=3, *S*={1}, *l*=1, *u*=2)

MDD Filtering for Sequence

Goal: Given an arbitrary MDD and a *Sequence* constraint, remove *all* inconsistent edges from the MDD (i.e., MDD-consistency)

Can this be done in polynomial time?

Theorem: Establishing MDD consistency for *Sequence* on an arbitrary MDD is NP-hard (even if the MDD order follows the sequence of variables X)

Proof: Reduction from 3-SAT [Bergman, Cire, vH, JAIR 2014]

Next goal: Develop a *partial* filtering algorithm, that does not necessarily achieve MDD consistency

Partial filter from decomposition

- Consider *Sequence*(X, q, S, l, u) with $X = x_1, x_2, \ldots, x_n$
- Introduce a 'cumulative' variable y_i representing the sum of the first i variables in X

$$y_0 = 0$$

$$y_i = y_{i-1} + (x_i \in S) \quad \text{for } i = 1..n$$

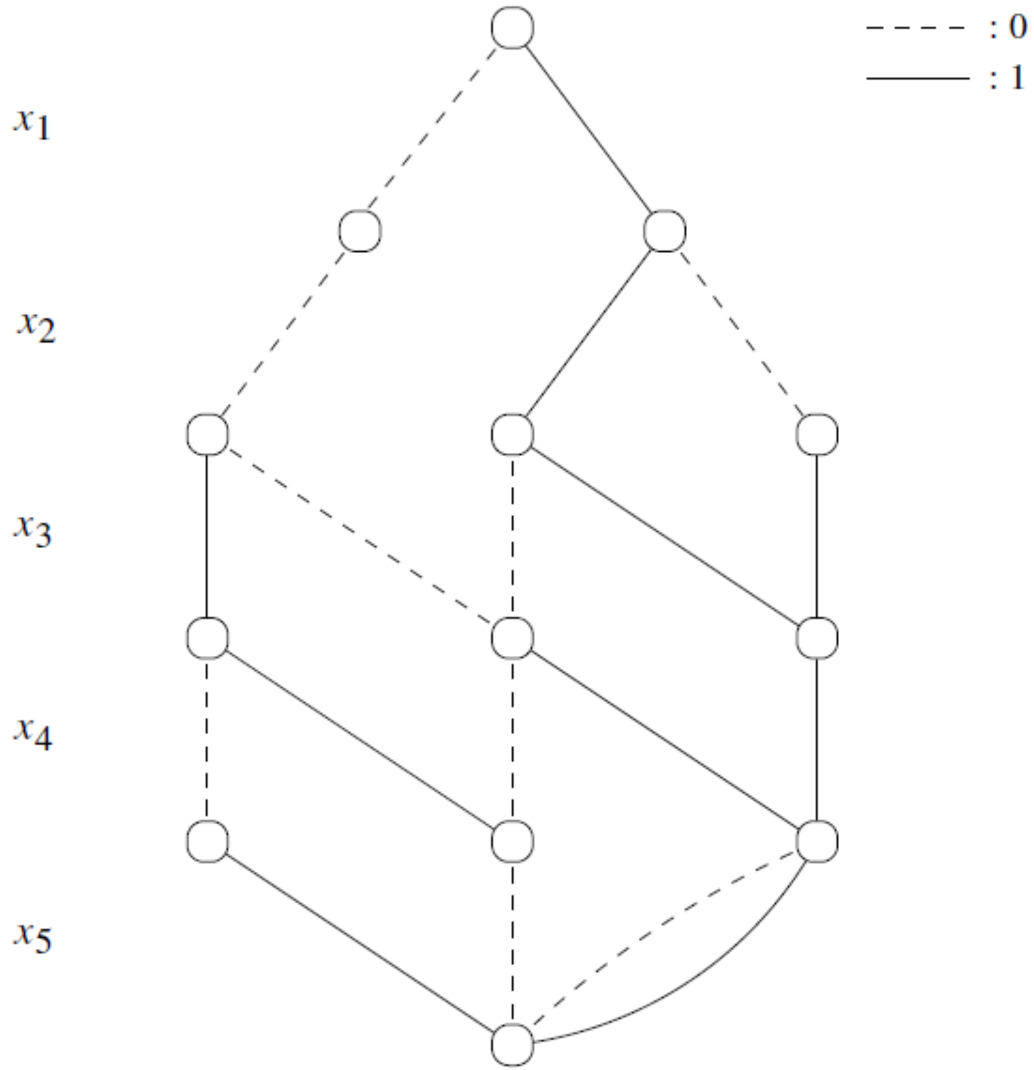
- Then the *Among* constraint on $[x_{i+1}, \ldots, x_{i+q}]$ is equivalent to

$$l \leq y_{i+q} - y_i$$

$$y_{i+q} - y_i \leq u \quad \text{for } i = 0, \ldots, n\text{-}q$$

[Brand et al., 2007] show that bounds reasoning on this decomposition suffices to reach domain consistency for *Sequence* (in poly-time)

MDD Filtering for Cumulative Sums Decomposition

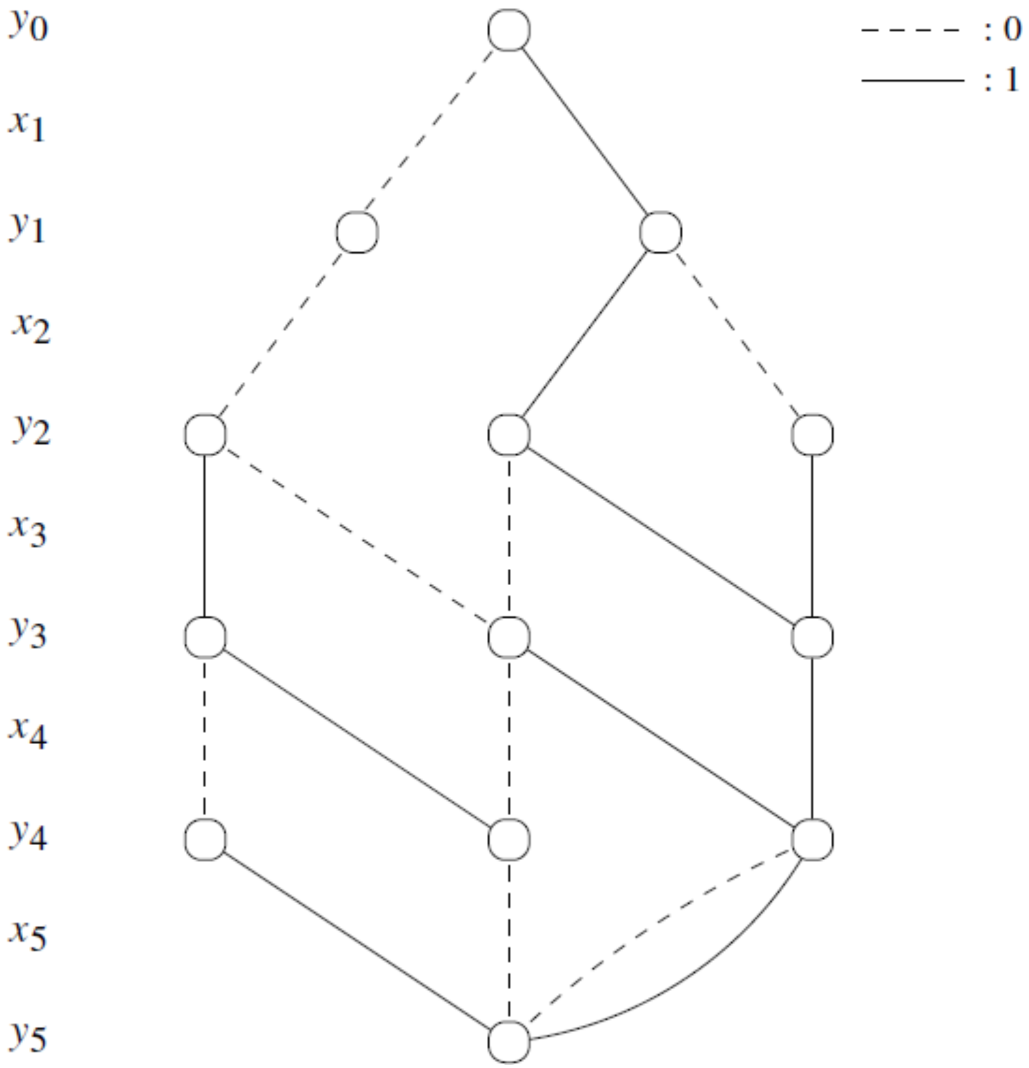


Sequence(X, q=3, S={1}, l=1, u=2)

Approach

- The auxiliary variables y_i can be naturally represented at the *nodes* of the MDD – this will be our state information
- We can now actively *filter* this node information (not only the edges)

MDD Filtering for Cumulative Sums Decomposition



Sequence(X, q=3, S={1}, l=1, u=2)

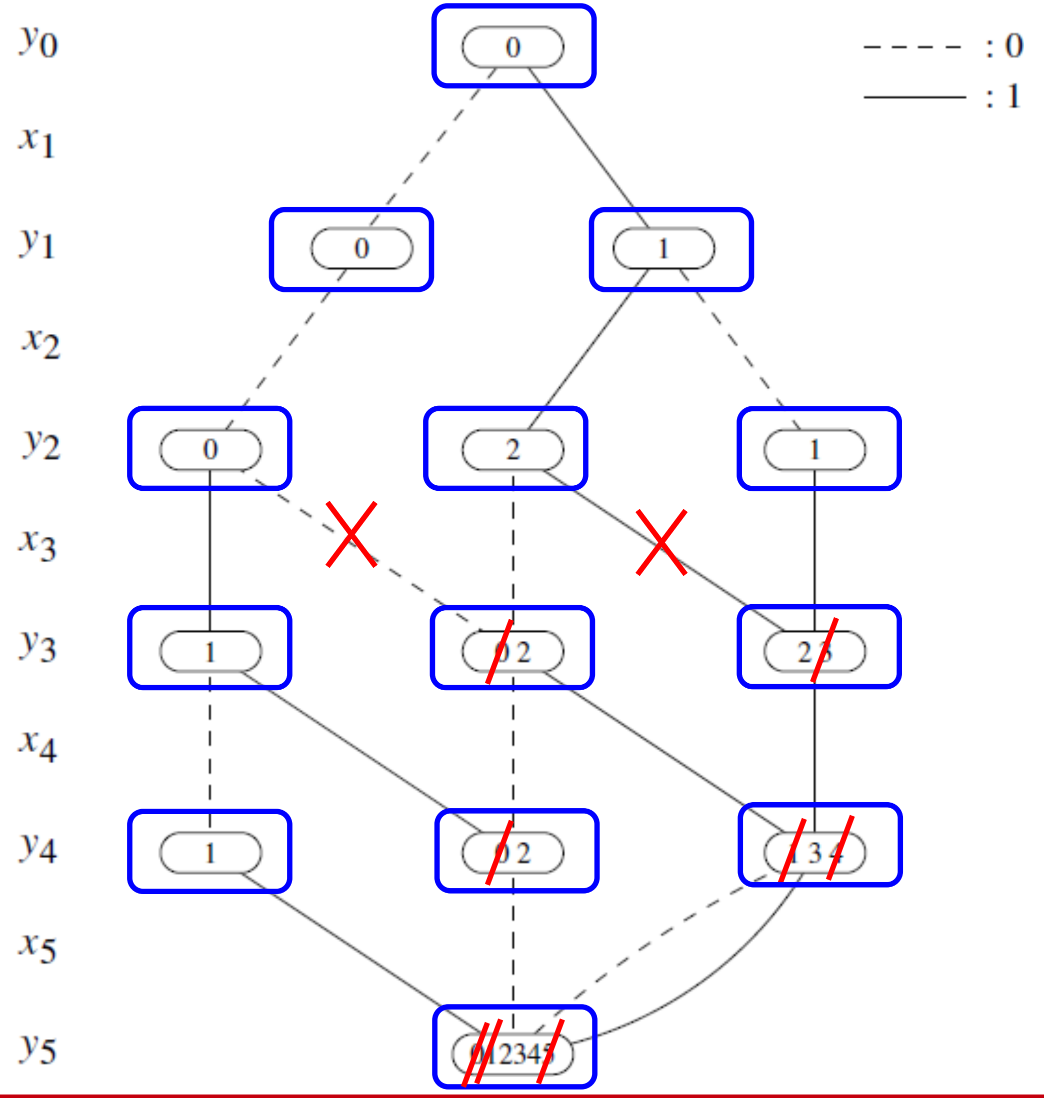
$$\boxed{y_i = y_{i-1} + x_i}$$

$$1 \le y_3 - y_0 \le 2$$

$$1 \le y_4 - y_1 \le 2$$

$$1 \le y_5 - y_2 \le 2$$

MDD Filtering for Cumulative Sums Decomposition



Sequence(X, q=3, S={1}, l=1, u=2)

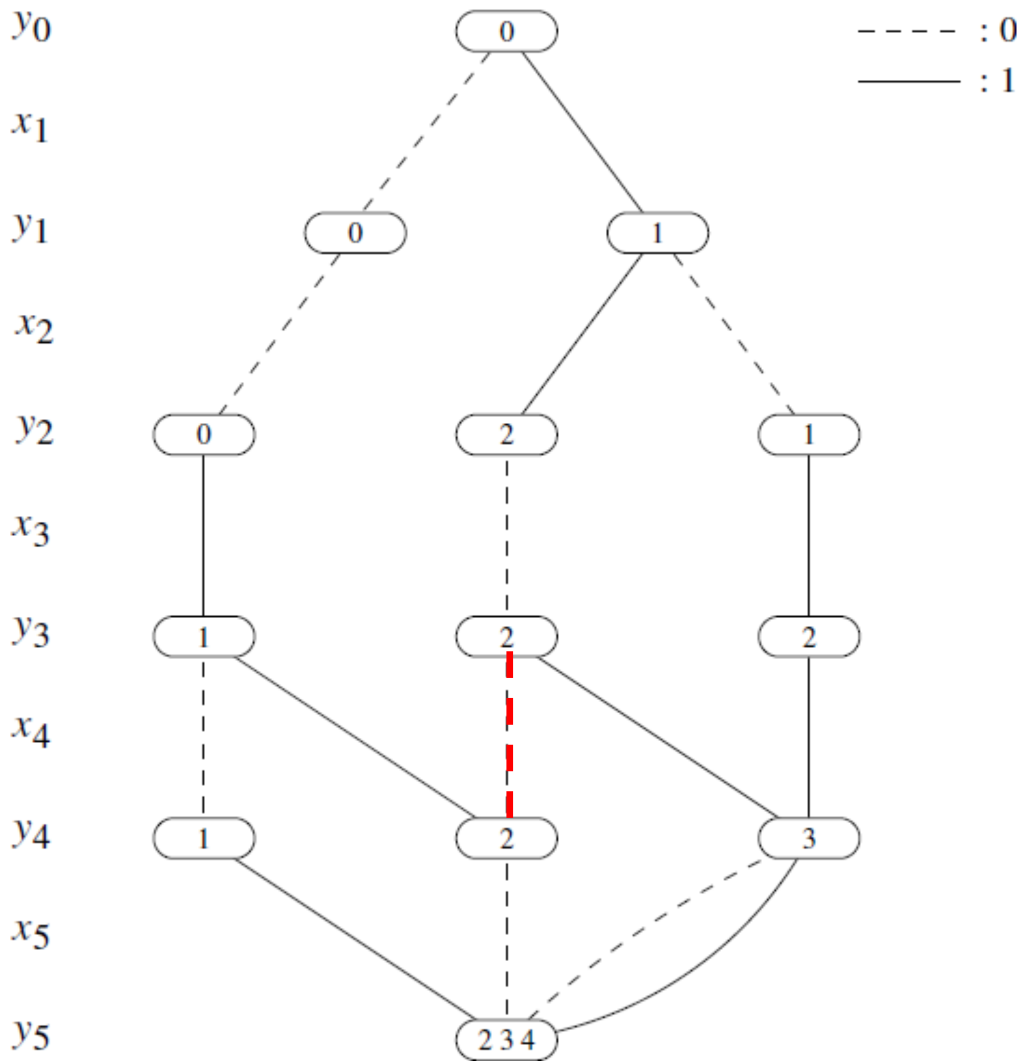
$y_i = y_{i-1} + x_i$

$1 \leq y_3 - y_0 \leq 2$

$1 \leq y_4 - y_1 \leq 2$

$1 \leq y_5 - y_2 \leq 2$

MDD Filtering for Cumulative Sums Decomposition



Sequence(X, q=3, S={1}, l=1, u=2)

$y_i = y_{i-1} + x_i$

$1 \leq y_3 - y_0 \leq 2$

$1 \leq y_4 - y_1 \leq 2$

$1 \leq y_5 - y_2 \leq 2$

This procedure does not guarantee MDD consistency

Analysis of Algorithm

- Initial population of node domains (y variables)
  - linear in MDD size
- Analysis of each state in layer k
  - maintain list of ancestors from layer k-q
  - direct implementation gives $O(qW^2)$ operations per state (W is maximum width)
  - need only maintain min and max value over previous q layers: $O(Wq)$
- One top-down and one bottom-up pass

Comparing MDD and Domain Consistencies

Proposition: MDD consistency on the ‘cumulative sums’ encoding is incomparable to MDD consistency on the *Among* encoding of *Sequence*

Follows from a result by [Brand et al., 2007]

Proposition: MDD consistency on the ‘cumulative sums’ encoding of *Sequence* is incomparable to domain consistency on *Sequence*

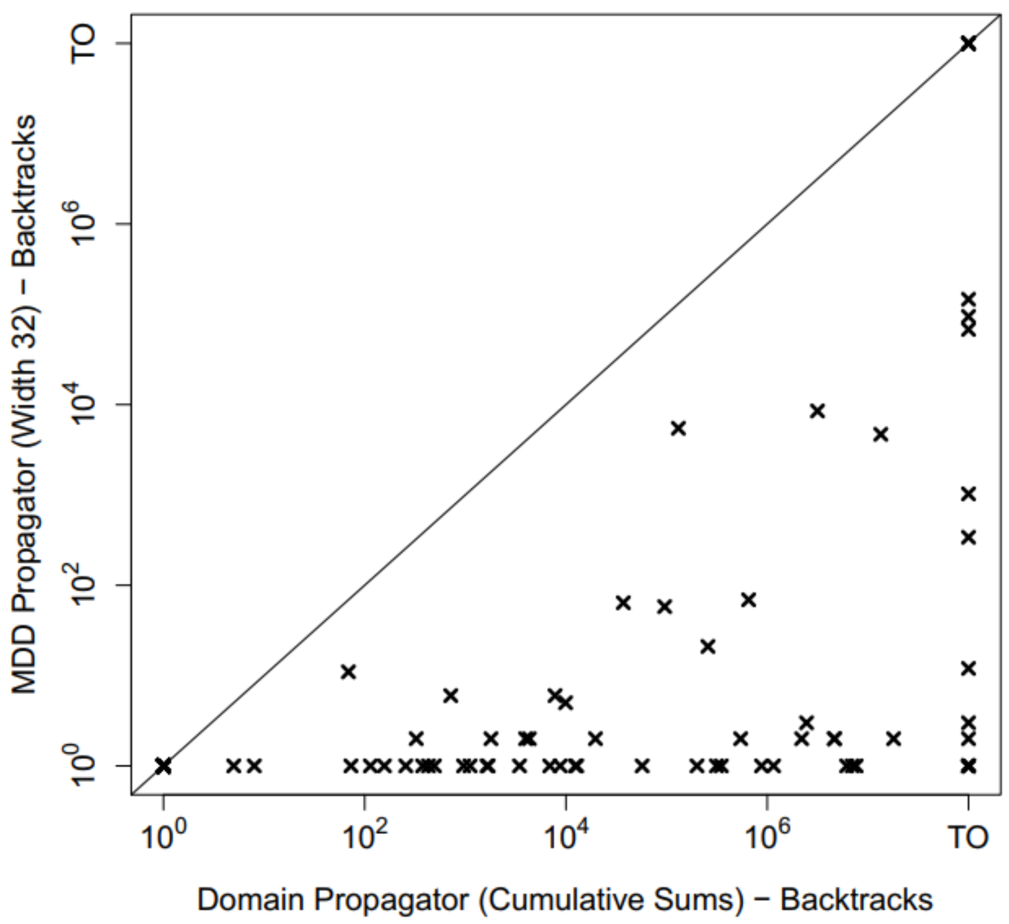
[Bergman et al., 2014]

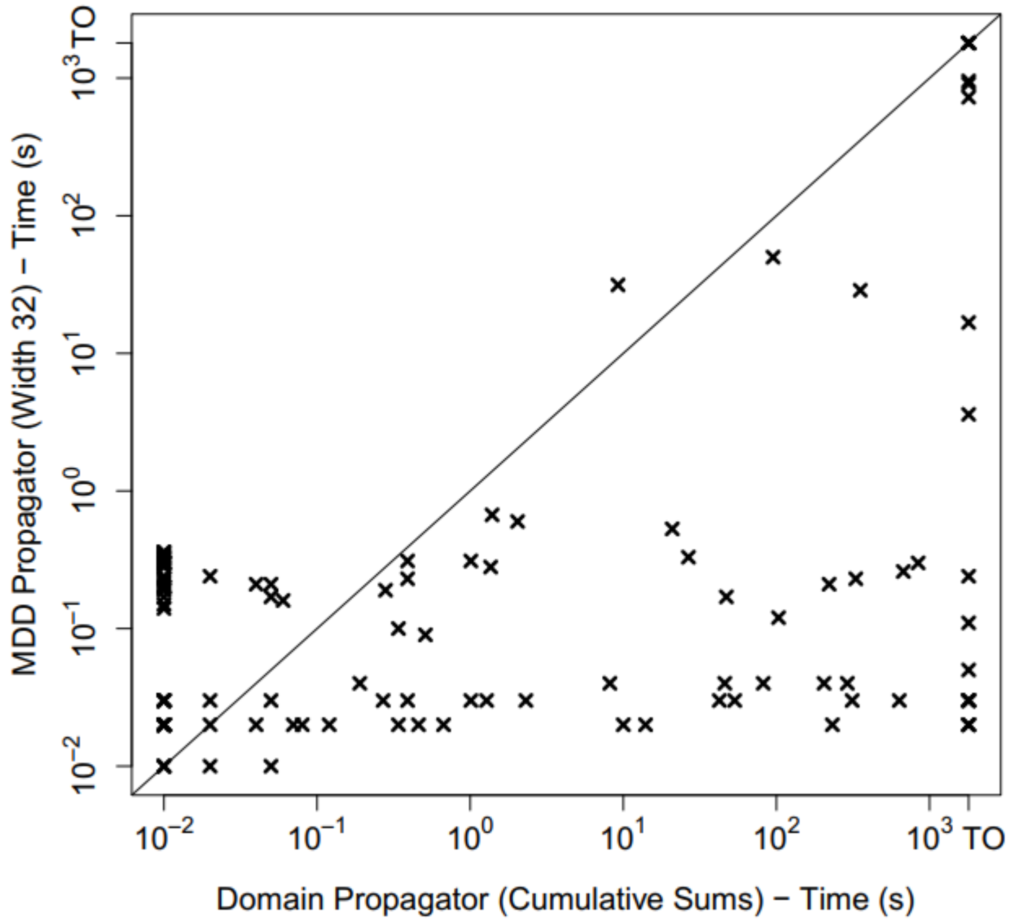
Experimental Evaluation

- Comparison of different encodings and consistencies
  - Among encoding: MDD consistency
  - Cumulative sums encoding: MDD consistency & Domain consistency
  - Sequence: Domain consistency (two different algorithms)
- Experimental setup
  - Implemented in IBM ILOG CPLEX CP Optimizer 12.3
  - 250 randomly generated instances
  - All methods apply the same fixed search strategy (lexicographic variable and value ordering; find first solution or prove that none exists)

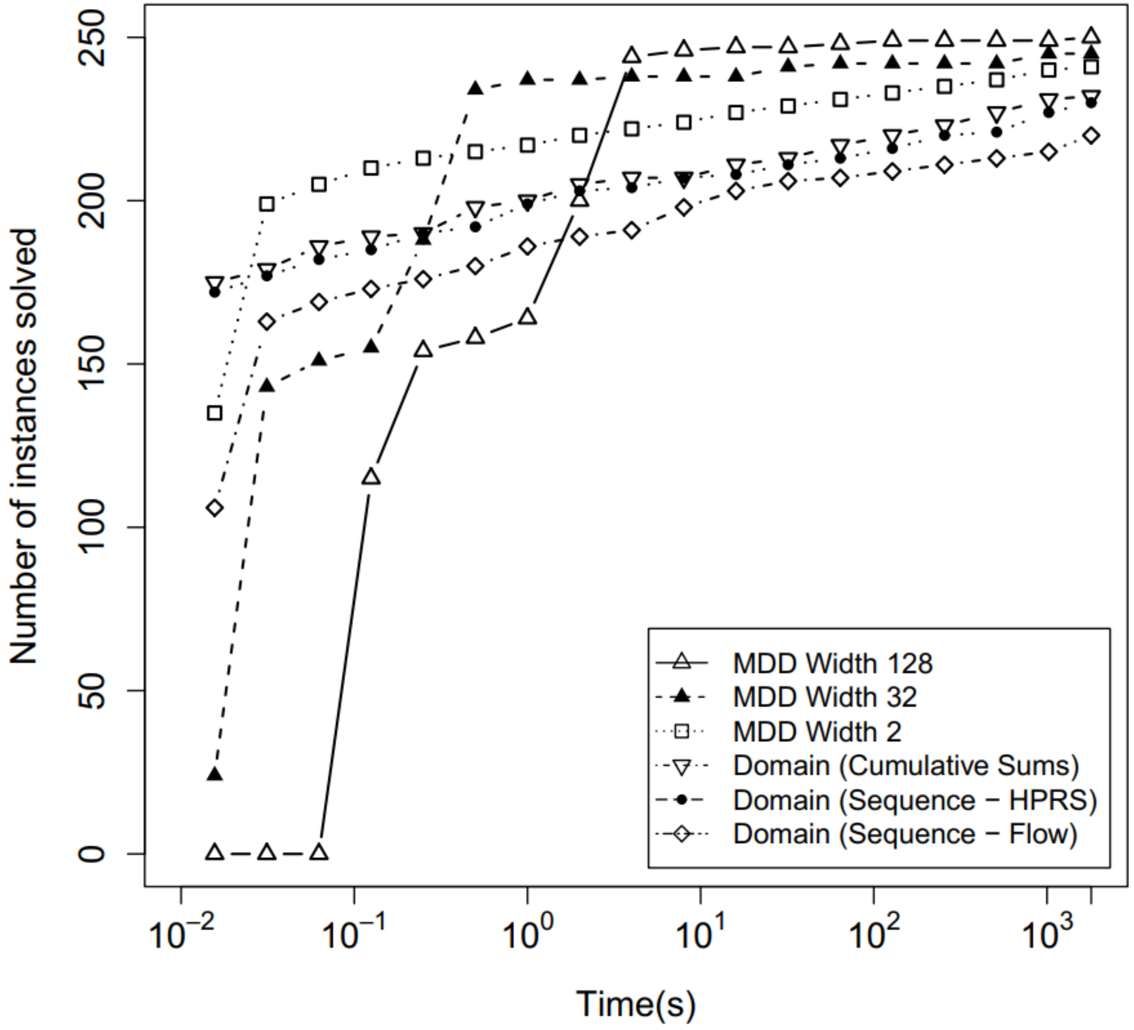
[Bergman, Cire, vH, JAIR 2014]

Cumulative Sums: MDD vs Domain Propagator



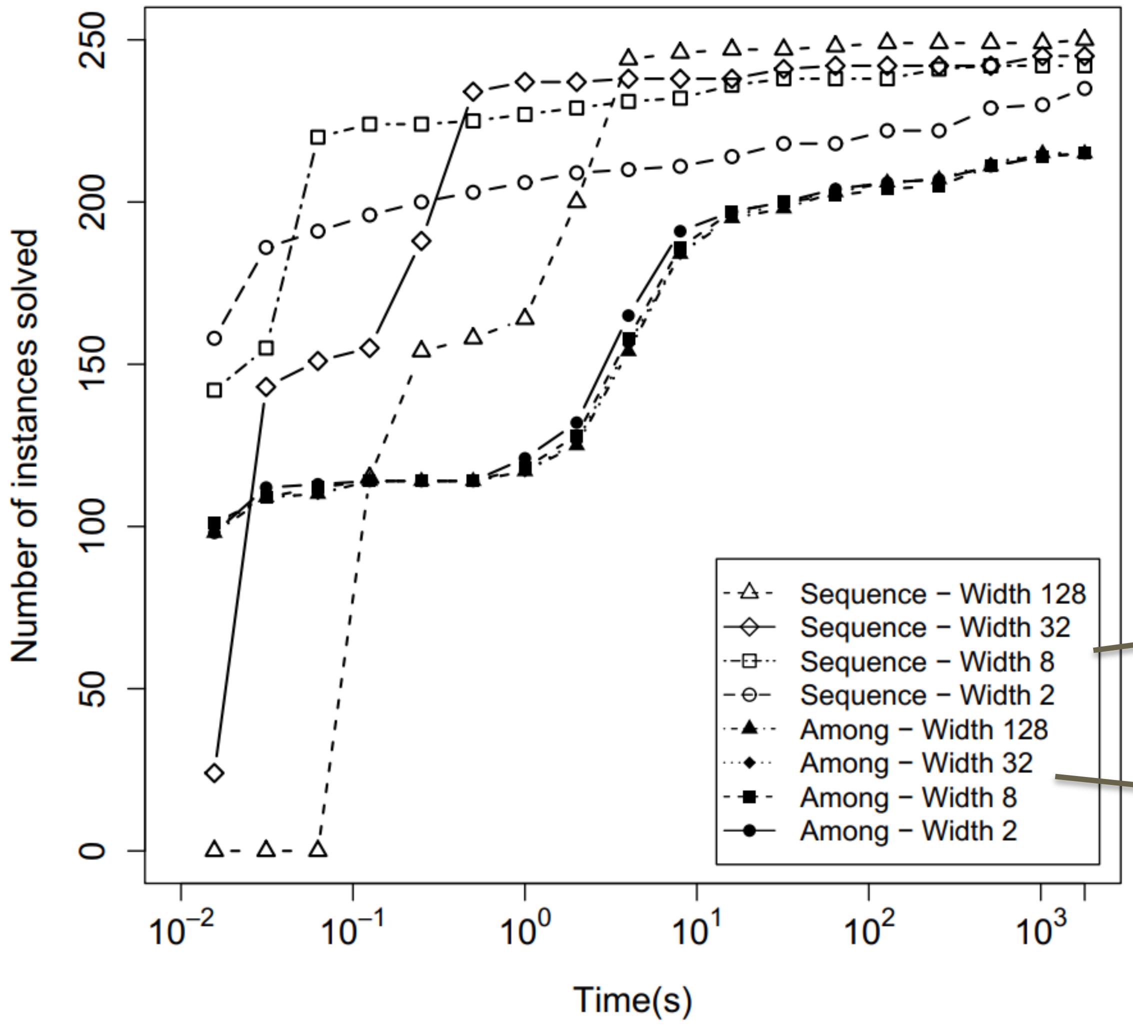


MDD outperforms Domain Propagation



- Two domain consistent *Sequence* propagators
  - HPRS: vH et al. (CP 2006)
  - Flow: Maher et al. (CP 2008)

Large MDD Widths Alone are not Enough



- **MDD-cumulative sums captures the *Sequence* structure better than MDD-Among**

MDD propagator on 'cumulative sums' encoding

MDD propagator on *Among* encoding

MDD-Based Constraint Programming in Haddock

Laurent Michel & Willem-Jan van Hoeve

CP'22
Haifa, August 1-7, 2022

Agenda

- What is Haddock ?
- Getting Started Example (MIS)
- Writing a simple sum
- A cardinality exercise
- Modeling AIS
- Absolute Value LTS
- Some demos

Integrating Decision Diagrams

- Embed an MDD as global constraint
  - You can have > 1!
- Interface seamlessly with domain variables
- Support filtering within the MDD
- Support optimization variable
- As incremental as possible
- Stateful MDD Nodes

Hosting CP Solver

- Architecture: MiniCP
- Implementation
  - C++ port of miniCP → MiniCPP
  - C++ 17 standard
  - Available on BitBucket
    - https://bitbucket.org/ldmbouge/minicpp/src/master/
  - Compiles with clang / gcc / cmake
  - Build and tested on macOS and Linux
  - Runs on Intel and ARM64 (M1)

```
git clone https://ldmbouge@bitbucket.org/ldmbouge/minicpp.git
```

CP = Model + Search

- Adding MDD
  - Same philosophy as pure CP
  - MDDs are global propagators
  - Can have several ones
  - Retain search writing ability
  - Coexist with all other constraints
  - Hooks up with classic domain variables

Stating a Simple "Pure CP" Model

- **MIS…**

Integer Programming Formulation:

$$
\begin{aligned}
\text{max} \quad & 5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4 \\
\text{subject to} \quad & x_0 + x_1 \leq 1 \\
& x_0 + x_4 \leq 1 \\
& x_1 + x_2 \leq 1 \\
& x_1 + x_3 \leq 1 \\
& x_2 + x_3 \leq 1 \\
& x_3 + x_4 \leq 1 \\
& x_0, x_1, x_2, x_3, x_4 \in \{0,1\}
\end{aligned}
$$

Stating a Simple "Pure CP" Model

- **MIS…**

Integer Programming Formulation:

$$\max\ 5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$$

$$\begin{aligned} \text{subject to}\quad & x_0 + x_1 \leq 1 \\ & x_0 + x_4 \leq 1 \\ & x_1 + x_2 \leq 1 \\ & x_1 + x_3 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_0, x_1, x_2, x_3, x_4 \in \{0,1\} \end{aligned}$$

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);
   auto obj = Factory::maximize(z);
   …
}
```

Solving a Simple “Pure CP” Model

```
int main(int argc,char* argv[]) {
  using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);
   auto obj = Factory::maximize(z);

   DFSearch search(cp,firstFail(cp,x));
   search.onSolution([&x,&z]() { std::cout << "Assignment:" << x << "\t OBJ:" << z << "\n";});
   auto stat = search.optimize(obj);
   std::cout << stat << std::endl;
   cp.dealloc();
   return 0;
}
```

Going MDD-style

Pure CP	MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

Going MDD-style

Pure CP	MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   auto mdd = Factory::makeMDDRelax(cp,4);
   mdd->post(sum(x,{5,4,2,6,8},z));
   mdd->post(sum({x[0],x[1]},0, 1));
   mdd->post(sum({x[0],x[4]},0, 1));
   mdd->post(sum({x[1],x[2]},0, 1));
   mdd->post(sum({x[1],x[3]},0, 1));
   mdd->post(sum({x[2],x[3]},0, 1));
   mdd->post(sum({x[3],x[4]},0, 1));
   cp->post(mdd);
   auto obj = Factory::maximize(z);
   …
}
```

Going MDD-style

Pure CP	MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   auto mdd = Factory::makeMDDRelax(cp,4);
   mdd->post(sum(x,{5,4,2,6,8},z));
   mdd->post(sum({x[0],x[1]},0, 1));
   mdd->post(sum({x[0],x[4]},0, 1));
   mdd->post(sum({x[1],x[2]},0, 1));
   mdd->post(sum({x[1],x[3]},0, 1));
   mdd->post(sum({x[2],x[3]},0, 1));
   mdd->post(sum({x[3],x[4]},0, 1));
   cp->post(mdd);
   auto obj = Factory::maximize(z);
   …
}
```

Going MDD-style

Pure CP	MDD

Pure CP

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   auto mdd = Factory::makeMDDRelax(cp,4);
   mdd->post(sum(x,{5,4,2,6,8},z));
   mdd->post(sum({x[0],x[1]},0, 1));
   mdd->post(sum({x[0],x[4]},0, 1));
   mdd->post(sum({x[1],x[2]},0, 1));
   mdd->post(sum({x[1],x[3]},0, 1));
   mdd->post(sum({x[2],x[3]},0, 1));
   mdd->post(sum({x[3],x[4]},0, 1));
   cp->post(mdd);
   auto obj = Factory::maximize(z);
   …
}
```

Going MDD-style

Pure CP	MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   auto mdd = Factory::makeMDDRelax(cp,4);
   mdd->post(sum(x,{5,4,2,6,8},z));
   mdd->post(sum({x[0],x[1]},0, 1));
   mdd->post(sum({x[0],x[4]},0, 1));
   mdd->post(sum({x[1],x[2]},0, 1));
   mdd->post(sum({x[1],x[3]},0, 1));
   mdd->post(sum({x[2],x[3]},0, 1));
   mdd->post(sum({x[3],x[4]},0, 1));
   cp->post(mdd);
   auto obj = Factory::maximize(z);
   …
}
```

Going MDD-style

Pure CP	MDD

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);

   cp->post(sum(x,{5,4,2,6,8}) == z);
   cp->post(sum({x[0],x[1]}) <= 1);
   cp->post(sum({x[0],x[4]}) <= 1);
   cp->post(sum({x[1],x[2]}) <= 1);
   cp->post(sum({x[1],x[3]}) <= 1);
   cp->post(sum({x[2],x[3]}) <= 1);
   cp->post(sum({x[3],x[4]}) <= 1);

   auto obj = Factory::maximize(z);
   …
}
```

```
int main(int argc,char* argv[]) {
   using namespace Factory;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, 5, 0, 1);
   auto z = Factory::makeIntVar(cp,0,10000);
   auto mdd = Factory::makeMDDRelax(cp,4);
   mdd->post(sum(x,{5,4,2,6,8},z));
   mdd->post(sum({x[0],x[1]},0, 1));
   mdd->post(sum({x[0],x[4]},0, 1));
   mdd->post(sum({x[1],x[2]},0, 1));
   mdd->post(sum({x[1],x[3]},0, 1));
   mdd->post(sum({x[2],x[3]},0, 1));
   mdd->post(sum({x[3],x[4]},0, 1));
   cp->post(mdd);
   auto obj = Factory::maximize(z);
   …
}
```

Remainder unchanged!

Behavior?

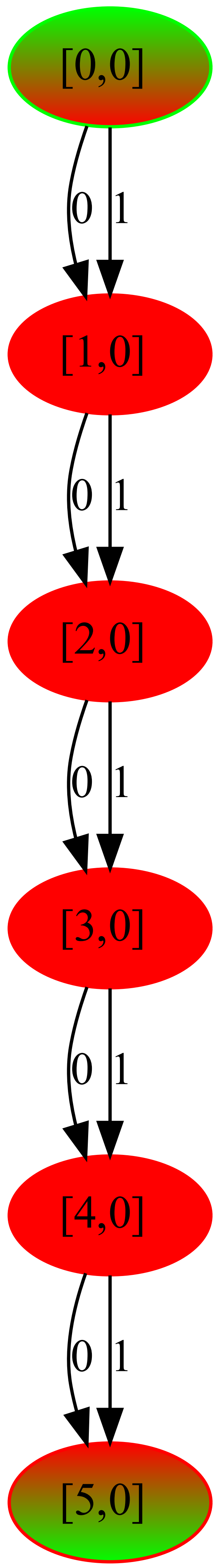
- MDD
  - Compiled behind the scene
- Filtering
  - Much smaller search trees
  - Benefits all the other constraints not in the MDD
- Search
  - Unchanged
  - Might be wise to label according to variables in MDD of course

What it does…

- Step 1
  - Create a w=1 MDD
- Step 2
  - Refine to desired width
- Step 3
  - As usual!

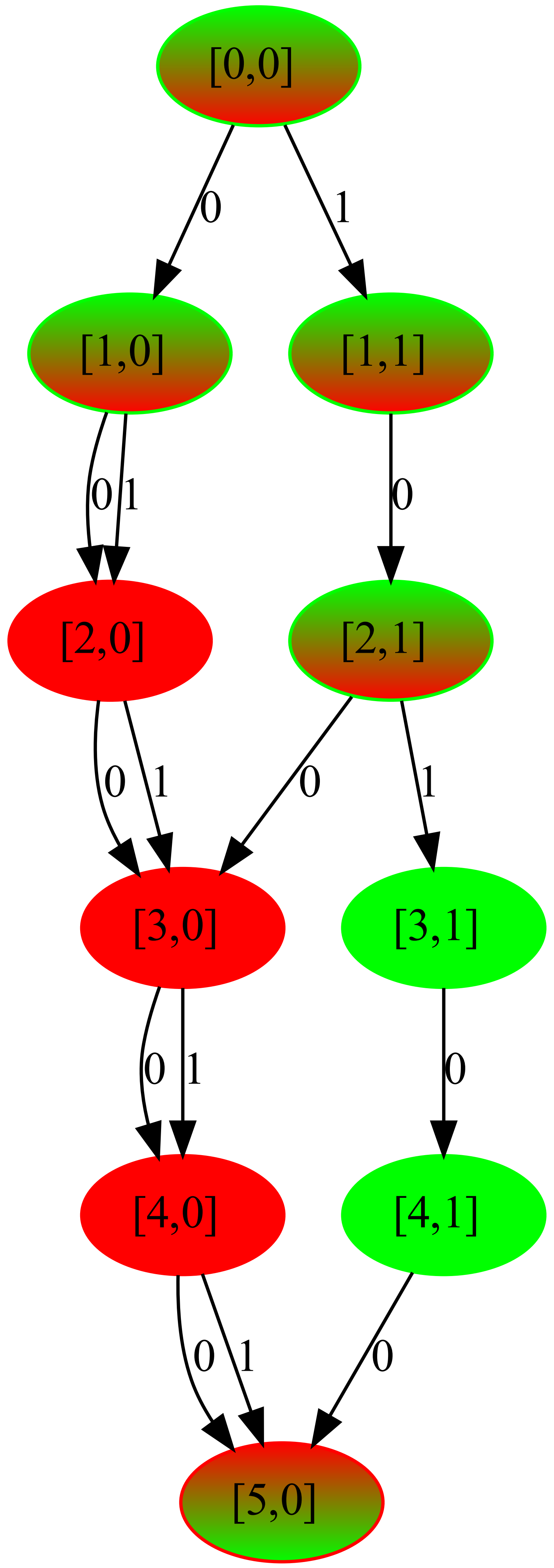
The initial MDD (w=1)

- Still same MIS



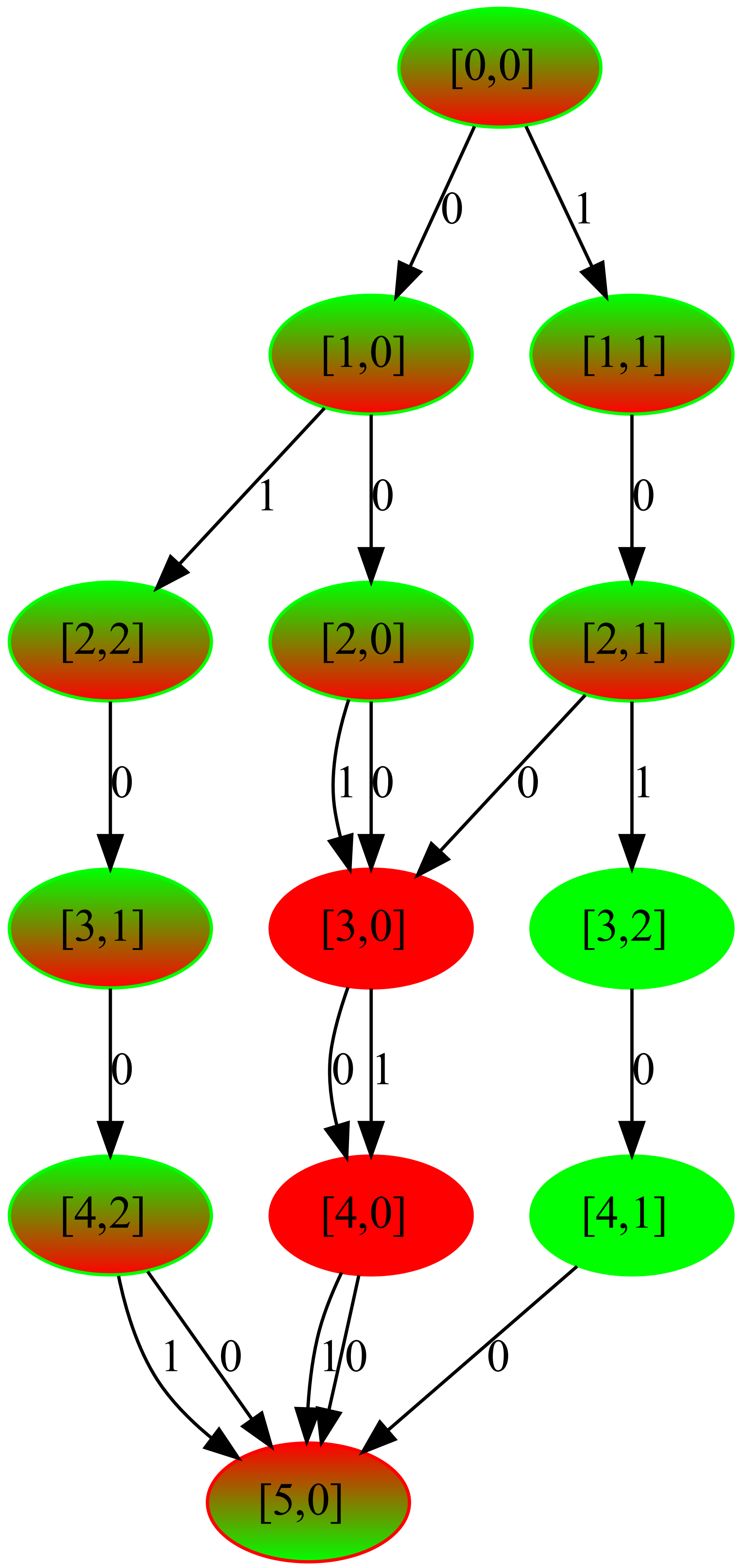
The refined MDD (w=2)

- Still same MIS



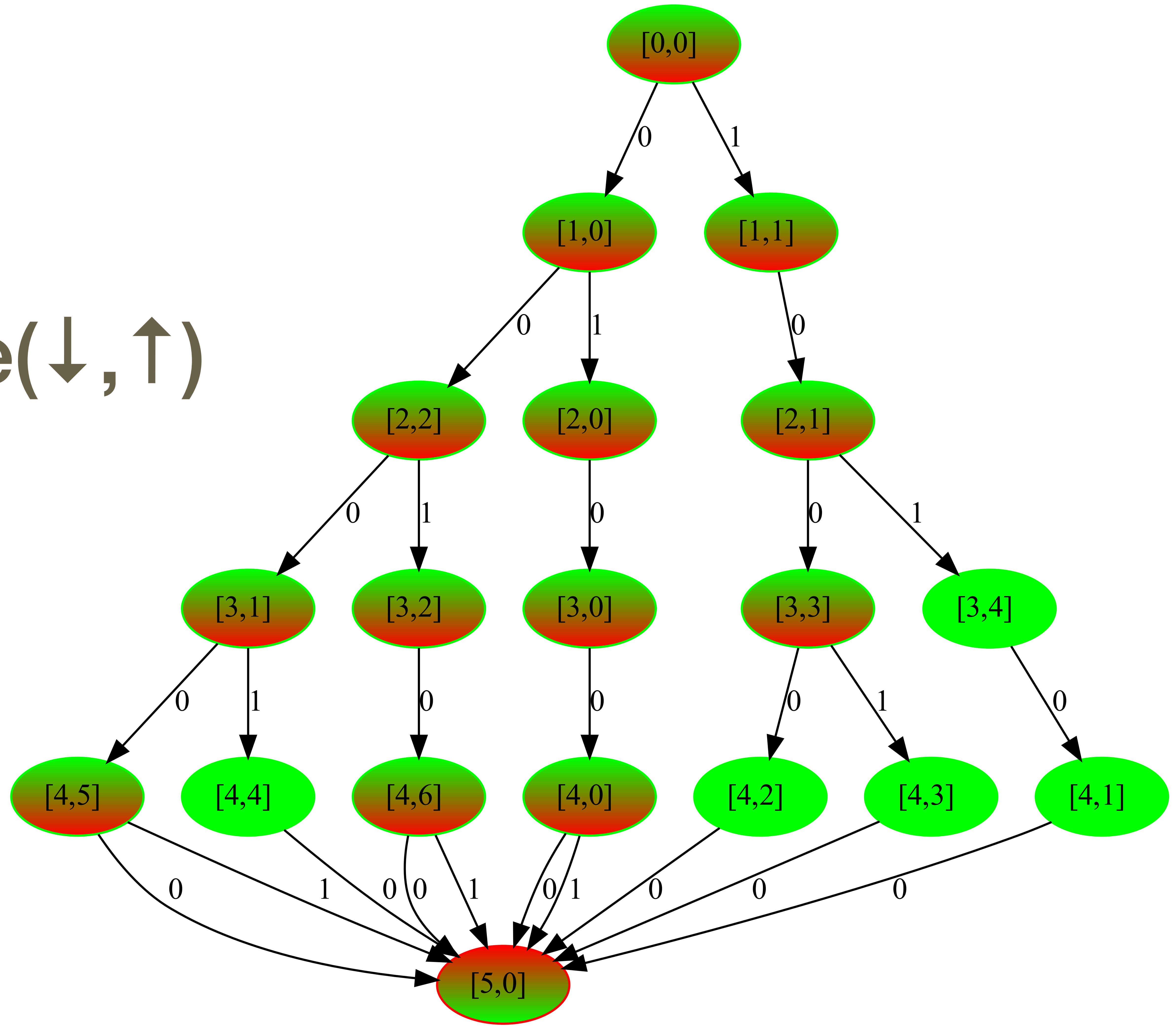
The refined MDD (w=3)

- Still same MIS



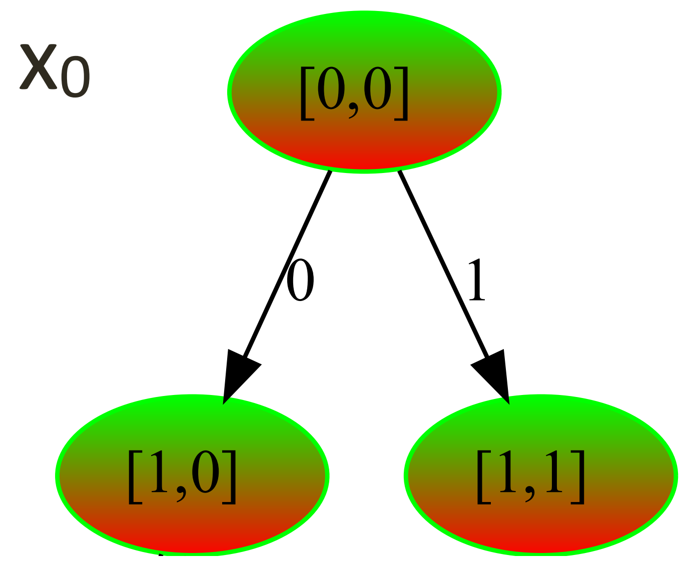
The refined MDD (w=8)

- Still same MIS
  - Not using full width
  - Colors convey approximate(↓,↑)
    - Exact : green
    - Approximate: red
    - Mixed: gradient
    - ➡Refining down only



Anatomy of a node…

- 7 constraints
  - sums
- 1 objective (z)



max $5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$

subject to $x_0 + x_1 \leq 1$

$x_0 + x_4 \leq 1$

$x_1 + x_2 \leq 1$

$x_1 + x_3 \leq 1$

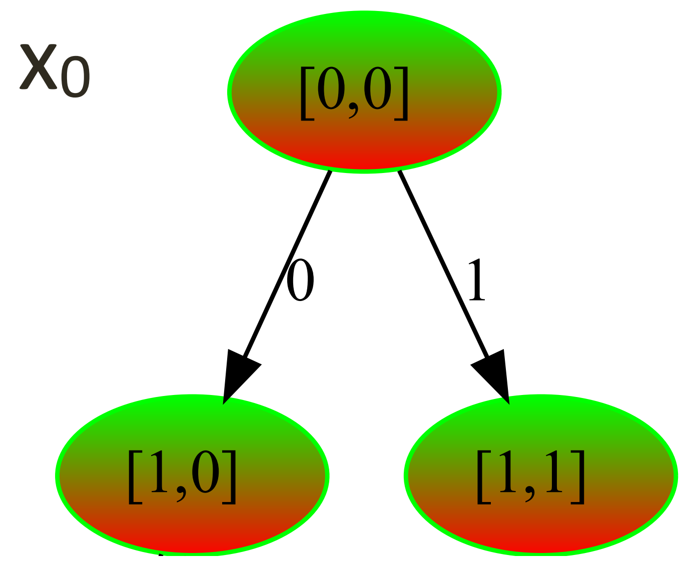
$x_2 + x_3 \leq 1$

$x_3 + x_4 \leq 1$

$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$

Anatomy of a node…

- 7 constraints
  - sums
- 1 objective (z)



max $5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$

subject to $x_0 + x_1 \leq 1$

$x_0 + x_4 \leq 1$

$x_1 + x_2 \leq 1$

$x_1 + x_3 \leq 1$

$x_2 + x_3 \leq 1$

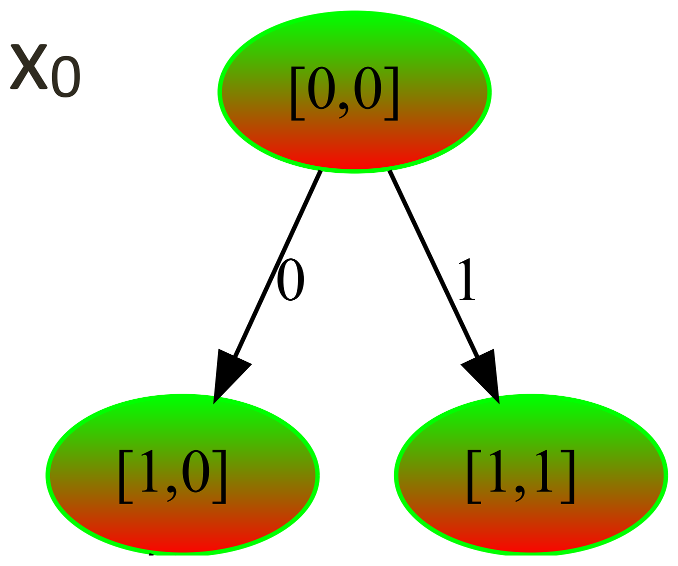
$x_3 + x_4 \leq 1$

$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$

```
[1,1] F[5  5 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ]
      T[0 14 4 0 0 1 0 1 1 0 1 2 0 1 2 0 1 2 0 2 2 ]
```

Anatomy of a node…

- 7 constraints
  - sums
- 1 objective (z)



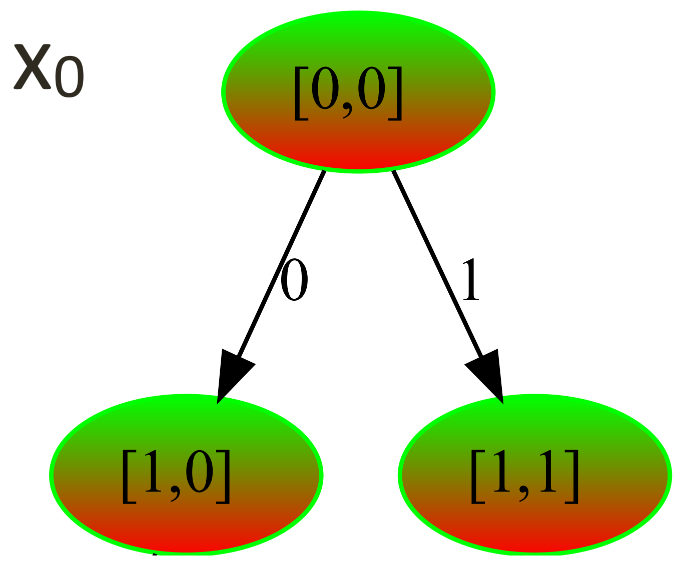
$$
\begin{aligned}
\text{max}\quad & 5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4 \\
\text{subject to}\quad & x_0 + x_1 \leq 1 \\
& x_0 + x_4 \leq 1 \\
& x_1 + x_2 \leq 1 \\
& x_1 + x_3 \leq 1 \\
& x_2 + x_3 \leq 1 \\
& x_3 + x_4 \leq 1 \\
& x_0, x_1, x_2, x_3, x_4 \in \{0,1\}
\end{aligned}
$$

$5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4 = z$
$x_0 + x_1 \leq 1$
$x_0 + x_4 \leq 1$
$x_1 + x_2 \leq 1$
$x_1 + x_3 \leq 1$
$x_2 + x_3 \leq 1$
$x_3 + x_4 \leq 1$

```
[1,1] F[5   5 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ]
      T[0 14 4 0 0 1 0 1 1 0 1 2 0 1 2 0 1 2 0 2 2 ]
```

Anatomy of a node…

- 7 constraints
  - sums
- 1 objective (z)



max $5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$

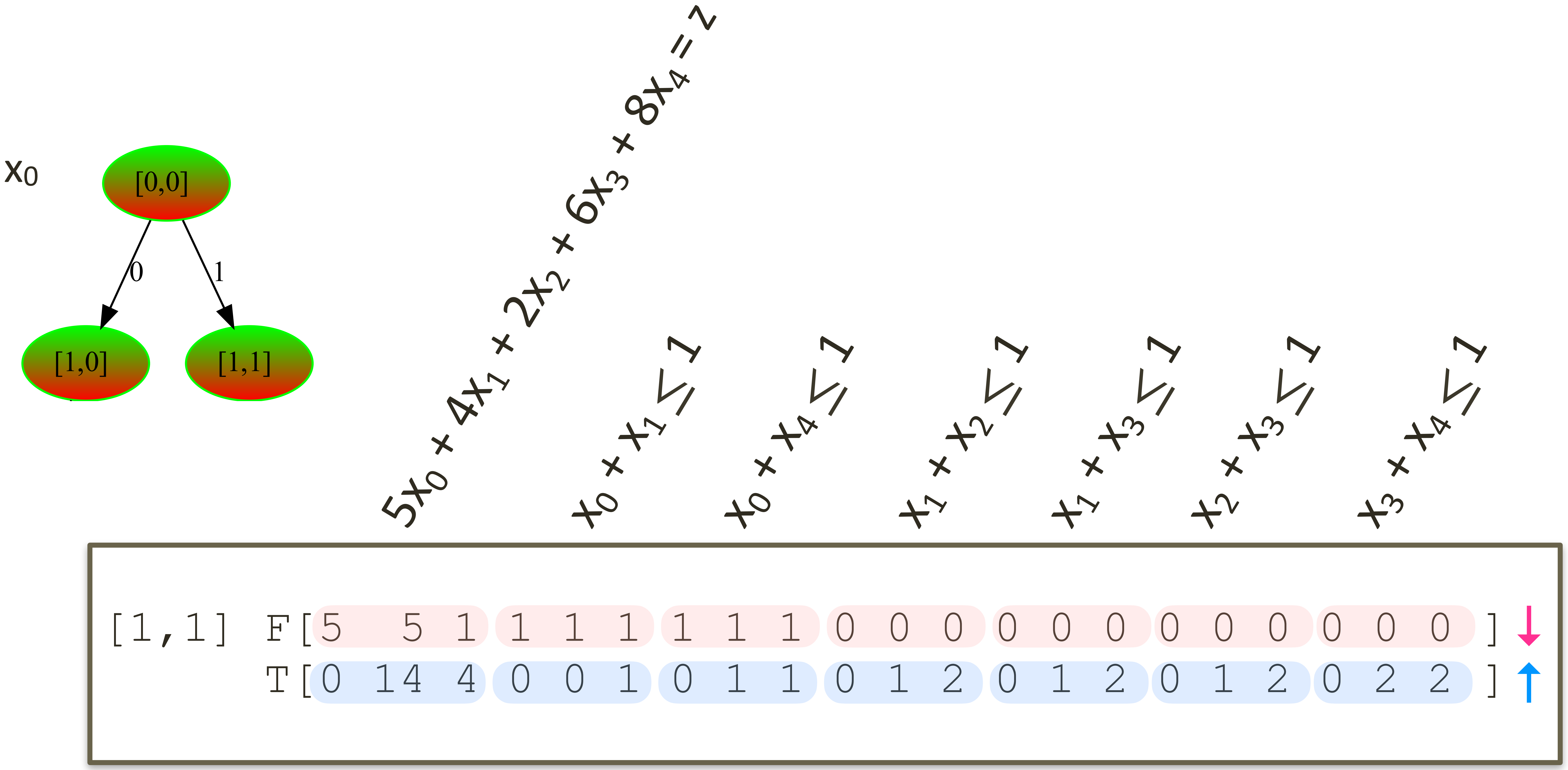
subject to
$$x_0 + x_1 \leq 1$$
$$x_0 + x_4 \leq 1$$
$$x_1 + x_2 \leq 1$$
$$x_1 + x_3 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$x_3 + x_4 \leq 1$$
$$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$$

$5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4 = z$ | $x_0 + x_1 \leq 1$ | $x_0 + x_4 \leq 1$ | $x_1 + x_2 \leq 1$ | $x_1 + x_3 \leq 1$ | $x_2 + x_3 \leq 1$ | $x_3 + x_4 \leq 1$

```
[1,1] F[5  5 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ]
      T[0 14 4 0 0 1 0 1 1 0 1 2 0 1 2 0 1 2 0 2 2 ]
```

Anatomy of a node…

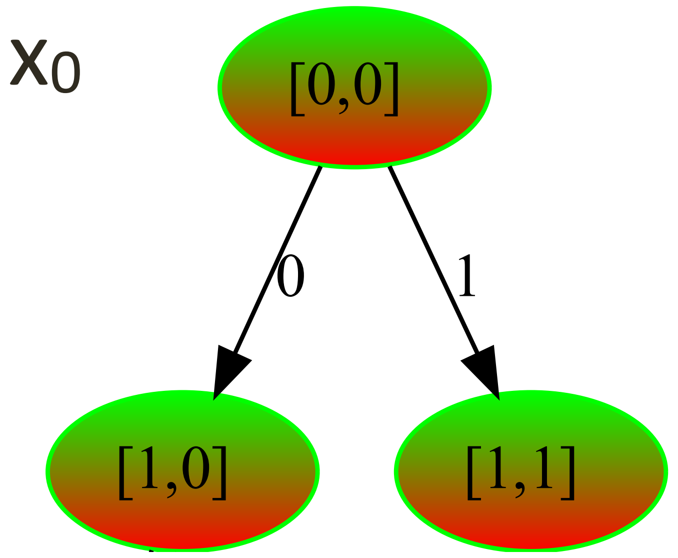
- 7 constraints
  - sums
- 1 objective (z)



$$
\begin{aligned}
\text{max}\quad & 5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4 \\
\text{subject to}\quad & x_0 + x_1 \le 1 \\
& x_0 + x_4 \le 1 \\
& x_1 + x_2 \le 1 \\
& x_1 + x_3 \le 1 \\
& x_2 + x_3 \le 1 \\
& x_3 + x_4 \le 1 \\
& x_0, x_1, x_2, x_3, x_4 \in \{0,1\}
\end{aligned}
$$

Anatomy of a node…

- 7 constraints
  - sums
- 1 objective (z)



max $5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$

subject to $x_0 + x_1 \leq 1$

$x_0 + x_4 \leq 1$

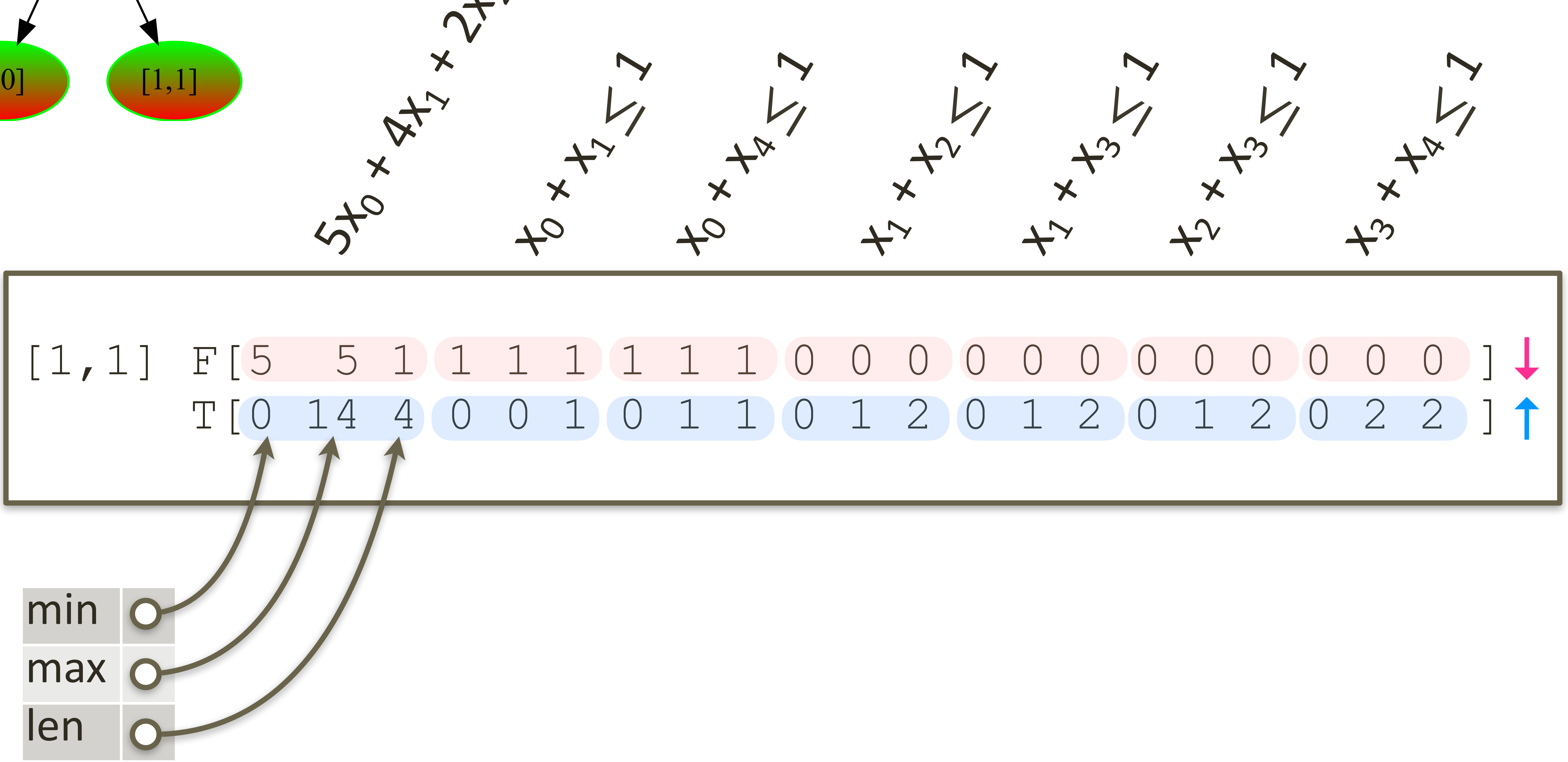
$x_1 + x_2 \leq 1$

$x_1 + x_3 \leq 1$

$x_2 + x_3 \leq 1$

$x_3 + x_4 \leq 1$

$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$

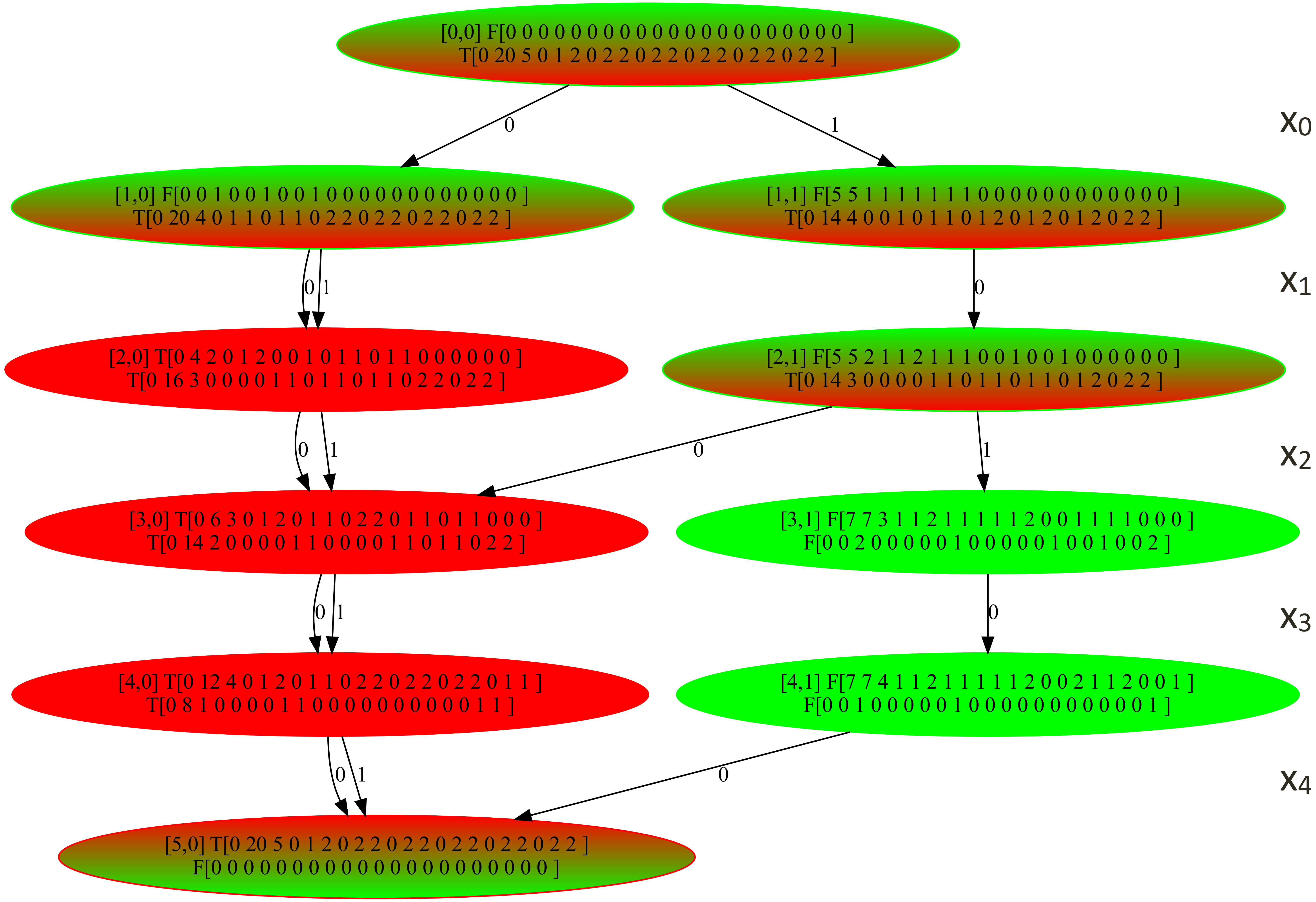


MIS MDD (w=2)

- With States

$$\max \; 5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$$

$$\begin{aligned}\text{subject to } & x_0 + x_1 \leq 1 \\ & x_0 + x_4 \leq 1 \\ & x_1 + x_2 \leq 1 \\ & x_1 + x_3 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_0, x_1, x_2, x_3, x_4 \in \{0,1\}\end{aligned}$$



Search

- Branching….
  - Variable selection
    - Can follow the layer order (or reverse layer order)
    - Can use other strategies (firstFail, semantic,…)
  - Value selection
    - Anything you wish
    - Can even look at the MDD to make decisions

Search

DEMO

- Branching….
  - Variable selection
    - Can follow the layer order (or reverse layer order)
    - Can use other strategies (firstFail, semantic,…)
  - Value selection
    - Anything you wish
    - Can even look at the MDD to make decisions

MDD Constraints available

- Arithmetic
  - linear (in)equalities
  - linear (in)equalities with objective
  - Absolute value
- Combinatorial
  - allDifferent
  - gcc
  - among
  - sequence

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Haddock Specification

- Use LTS formalism to specify an MDD

Given an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1)$ through $D(x_n)$, a *multi-valued decision diagram (MDD)* on X is an LTS $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which:

- the state set $\mathcal{S}$ is stratified in $n+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_n$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;
- the transition label set Λ is defined as $\bigcup_{i \in 1..n} D(x_i)$;
- a transition between two states $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ carries a label $v \in D(x_i)$ $(i \in 1..n)$, i.e., $a \xrightarrow{v} b$;
- the layer $\mathcal{L}_0$ consists of a single *source* state $s_\perp$;
- the layer $\mathcal{L}_n$ consists of a single *sink* state $s_\top$.

Writing a simple constraint

$$lb \leq \sum_{i=0}^{n-1} a_i \cdot x_i \leq ub$$

- Bounded sum (both sides)
  - Number of variables: ***n***
  - array of coefficients: ***a***
- We need
  - State (source / sink / general)
  - Transitions
  - Arc existence predicate
  - Node existence predicate
  - Relaxations

Sum State

- Ideas
  - Break down into prefix and suffix of $[a_0x_0, a_1x_1, \ldots, a_nx_n]$
  - Remember length of prefix / suffix
  - Maintain lower (L) and upper (U) bounds
  - Prefix via the *down* direction
  - Suffix via the *up* direction
- Thus
  - Internal: $\langle [L^{\downarrow}, U^{\downarrow}, len^{\downarrow}], [L^{\uparrow}, U^{\uparrow}, len^{\uparrow}] \rangle$
  - Source state: $\langle [0, 0, 0]^{\downarrow}, - \rangle$
  - Sink state: $\langle -, [0, 0, 0]^{\uparrow} \rangle$

Transitions (Down)

- Given
  - Parent node **p**
  - Child node **c**
  - Value labeled arc $p \xrightarrow{v} c$
- Down transitions are

$$
\begin{aligned}
L^{\downarrow}(c) &= L^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
U^{\downarrow}(c) &= U^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
len^{\downarrow}(c) &= len^{\downarrow}(p) + 1
\end{aligned}
$$

Transitions (Up)

- Given
  - Parent node **p**
  - Child node **c**
  - Value labeled arc $p \xrightarrow{v} c$
- Up transitions are

$$
\begin{aligned}
L^{\uparrow}(p) &= L^{\uparrow}(c) + a[|x| - 1 - len^{\uparrow}(c)] \cdot v \\
U^{\uparrow}(p) &= U^{\uparrow}(c) + a[|x| - 1 - len^{\uparrow}(c)] \cdot v \\
len^{\uparrow}(p) &= len^{\uparrow}(c) + 1
\end{aligned}
$$

Arc Existence

- Given
  - Parent node **p**
  - Child node **c**
  - Value labeled arc $p \xrightarrow{v} c$
- Determine whether the arc exist

$$exist(p \xrightarrow{v} c) = \begin{pmatrix} L^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +L^{\uparrow}(c) \leq ub \\ & \wedge & \\ U^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +U^{\uparrow}(c) \geq lb \end{pmatrix}$$

Node Existence

- Given
  - A node ***n***
- Determine if the node exist

$$exist(n) = \begin{pmatrix} L^{\downarrow}(n) & + & L^{\uparrow}(n) \leq ub \\ & \wedge & \\ U^{\downarrow}(n) & + & U^{\uparrow}(n) \geq lb \end{pmatrix}$$

Relaxations

- Fairly straightforward
  - When *merging* nodes
    - Take the ***min*** of the lower bounds (L)
    - Take the ***max*** of the upper bounds (U)
    - len is constant in layer, so it does not matter…

See it in code! (Slide 1/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   MDDSpec& mdd = m->getSpec();
   const int nbVars = (int)vars.size();
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");

   const auto L      = mdd.downIntState(d, 0, INT_MAX,MinFun);
   const auto U      = mdd.downIntState(d, 0, INT_MAX,MaxFun);
   const auto len    = mdd.downIntState(d,0,nbVars,MinFun);
   const auto Lup    = mdd.upIntState(d, 0, INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d, 0, INT_MAX,MaxFun);
   const auto lenUp  = mdd.upIntState(d, 0,nbVars,MinFun);









}
```

See it in code! (Slide 1/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   MDDSpec& mdd = m->getSpec();
   const int nbVars = (int)vars.size();
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");

   const auto L      = mdd.downIntState(d, 0, INT_MAX,MinFun);
   const auto U      = mdd.downIntState(d, 0, INT_MAX,MaxFun);
   const auto len    = mdd.downIntState(d,0,nbVars,MinFun);
   const auto Lup    = mdd.upIntState(d, 0, INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d, 0, INT_MAX,MaxFun);
   const auto lenUp  = mdd.upIntState(d, 0,nbVars,MinFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto, const auto& val) -> bool {
      return (parent.down[L] + val*array[parent.down[len]] + child.up[L] <= ub) &&
             (parent.down[U] + val*array[parent.down[len]] + child.up[U] >= lb);
   });
   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= ub) && (n.down[U] + n.up[Uup] >= lb);
   });
   …
}
```

See it in code! (Slide 1/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   MDDSpec& mdd = m->getSpec();
   const int nbVars = (int)vars.size();
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");

   const auto L      = mdd.downIntState(d, 0, INT_MAX,MinFun);
   const auto U      = mdd.downIntState(d, 0, INT_MAX,MaxFun);
   const auto len    = mdd.downIntState(d,0,nbVars,MinFun);
   const auto Lup    = mdd.upIntState(d, 0, INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d, 0, INT_MAX,MaxFun);
   const auto lenUp  = mdd.upIntState(d, 0,nbVars,MinFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto, const auto& val) -> bool {
      return (parent.down[L] + val*array[parent.down[len]] + child.up[L] <= ub) &&
             (parent.down[U] + val*array[parent.down[len]] + child.up[U] >= lb);
   });
   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= ub) && (n.down[U] + n.up[Uup] >= lb);
   });
   …
}
```

$$exist(p \xrightarrow{v} c) = \begin{pmatrix} L^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +L^{\uparrow}(c) \leq ub \\ & \wedge & \\ U^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +U^{\uparrow}(c) \geq lb \end{pmatrix}$$

See it in code! (Slide 1/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   MDDSpec& mdd = m->getSpec();
   const int nbVars = (int)vars.size();
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");

   const auto L      = mdd.downIntState(d, 0, INT_MAX,MinFun);
   const auto U      = mdd.downIntState(d, 0, INT_MAX,MaxFun);
   const auto len    = mdd.downIntState(d,0,nbVars,MinFun);
   const auto Lup    = mdd.upIntState(d, 0, INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d, 0, INT_MAX,MaxFun);
   const auto lenUp  = mdd.upIntState(d, 0,nbVars,MinFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto, const auto& val) -> bool {
      return (parent.down[L] + val*array[parent.down[len]] + child.up[L] <= ub) &&
             (parent.down[U] + val*array[parent.down[len]] + child.up[U] >= lb);
   });
   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= ub) && (n.down[U] + n.up[Uup] >= lb);
   });
   …
}
```

$$exist(p \xrightarrow{v} c) = \begin{pmatrix} L^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +L^{\uparrow}(c) \leq ub \\ & \wedge & \\ U^{\downarrow}(p)+ & v \cdot a[len^{\downarrow}(p)] & +U^{\uparrow}(c) \geq lb \end{pmatrix}$$

$$exist(n) = \begin{pmatrix} L^{\downarrow}(n) & + & L^{\uparrow}(n) \leq ub \\ & \wedge & \\ U^{\downarrow}(n) & + & U^{\uparrow}(n) \geq lb \end{pmatrix}$$

See it in code! (Slide 2/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   …
   mdd.transitionDown(d,L,{len,L},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[L] = parent.down[L] + array[parent.down[len]] * val.min();
   });
   mdd.transitionDown(d,U,{len,U},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[U] = parent.down[U] + array[parent.down[len]] * val.max();
   });
   mdd.transitionDown(d,len,{len},{},[=](auto& out,const auto& parent,const auto&, const auto&) {
      out[len]  = parent.down[len] + 1;
   });




}
```

See it in code! (Slide 2/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   …
   mdd.transitionDown(d,L,{len,L},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[L] = parent.down[L] + array[parent.down[len]] * val.min();
   });
   mdd.transitionDown(d,U,{len,U},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[U] = parent.down[U] + array[parent.down[len]] * val.max();
   });
   mdd.transitionDown(d,len,{len},{},[=](auto& out,const auto& parent,const auto&, const auto&) {
      out[len]  = parent.down[len] + 1;
   });

}
```

$$
\begin{aligned}
L^{\downarrow}(c) &= L^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
U^{\downarrow}(c) &= U^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
len^{\downarrow}(c) &= len^{\downarrow}(p) + 1
\end{aligned}
$$

See it in code! (Slide 2/2)

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   …
   mdd.transitionDown(d,L,{len,L},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[L] = parent.down[L] + array[parent.down[len]] * val.min();
   });
   mdd.transitionDown(d,U,{len,U},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[U] = parent.down[U] + array[parent.down[len]] * val.max();
   });
   mdd.transitionDown(d,len,{len},{},[=](auto& out,const auto& parent,const auto&, const auto&) {
      out[len]  = parent.down[len] + 1;
   });
}
```

$$
\begin{aligned}
L^{\downarrow}(c) &= L^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
U^{\downarrow}(c) &= U^{\downarrow}(p) + a[len^{\downarrow}(p)] \cdot v \\
len^{\downarrow}(c) &= len^{\downarrow}(p) + 1
\end{aligned}
$$

2 noteworthy observations

1. val is a set of labels
   → the set of labels on arcs from parent to out

2. Down transitions.
   → out is the down state of the child

See it in code! (Slide 2/2) [with both ↓,↑]

```
MDDCstrDesc::Ptr sum(MDD::Ptr m, const Factory::Veci& vars, const std::vector<int>& array, int lb, int ub)
{
   …
   mdd.transitionDown(d,L,{len,L},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[L] = parent.down[L] + array[parent.down[len]] * val.min();
   });
   mdd.transitionDown(d,U,{len,U},{},[=](auto& out,const auto& parent,const auto&,const auto& val) {
      out[U] = parent.down[U] + array[parent.down[len]] * val.max();
   });
   mdd.transitionDown(d,len,{len},{},[=](auto& out,const auto& parent,const auto&, const auto&) {
      out[len]  = parent.down[len] + 1;
   });

   mdd.transitionUp(d,Lup,{lenUp,Lup},{},[=](auto& out,const auto& child,const auto&, const auto& val) {
      out[Lup] = child.up[Lup] + array[nbVars - 1 - child.up[lenUp]] * val.min();
   });
   mdd.transitionUp(d,Uup,{lenUp,Uup},{},[=](auto& out,const auto& child,const auto&, const auto& val) {
      out[Uup] = child.up[Uup] + array[nbVars - 1 - child.up[lenUp]] * val.max();
   });
   mdd.transitionUp(d,lenUp,{lenUp},{},[=](auto& out,const auto& child,const auto&, const auto&) {
      out[lenUp] = child.up[lenUp] + 1;
   });
   return d;
}
```

Bottom line recap

- Haddock LTS
  - Requires 40 lines of C++ to specify a class of constraints
  - Code is very close to MDD specification
  - Propagator derived automatically
  - It composes automatically with other MDD LTS
  - Reasonable performance
    - W.r.t. other MDDs
    - W.r.t. domain-based models
  - Drastic search tree size reduction

What Can be hosted in a State ?

- States hold *typed* properties
- Types supported today
  - Integer (32-bit signed)
  - Bytes (8-bit signed)
  - Bit (1-bit)
  - BitSequence (a collection of n bits [0..n-1]) for sets!
  - IntegerWindow (window of *n* consecutive integers)

Handling an Objective

- Recall MIS again…
  - Linear sum for the objective (z)
  - CP solver maximizes z
  - What about the MDD ?

Handling an Objective

- Recall MIS again…
  - Linear sum for the objective (z)
  - CP solver maximizes z
  - What about the MDD ?

max $5x_0 + 4x_1 + 2x_2 + 6x_3 + 8x_4$

subject to
$x_0 + x_1 \leq 1$
$x_0 + x_4 \leq 1$
$x_1 + x_2 \leq 1$
$x_1 + x_3 \leq 1$
$x_2 + x_3 \leq 1$
$x_3 + x_4 \leq 1$
$x_0, x_1, x_2, x_3, x_4 \in \{0,1\}$

MDD and objective

- ⇢ Objective variable z affects the MDD
  - arcExist : validate all arc existence
  - nodeExist : validate all node existence
- ⇠ MDD affects the objective z
  - Use the bounds in the sink to tighten z

MDD and objective

- ⇢ Objective variable z affects the MDD
  - arcExist : validate all arc existence
  - nodeExist : validate all node existence
- ⇠ MDD affects the objective z
  - Use the bounds in the sink to tighten z

Approach

1. Listen on z for bound change events to review all nodes/arcs
2. When at fixpoint, tighten the bounds on z from sink

SumMDD and Objective

```
MDDCstrDesc::Ptr sum(MDD::Ptr m,const Factory::Veci& vars,const std::vector<int>& array,var<int>::Ptr z) {
   MDDSpec& mdd = m->getSpec();
   mdd.addGlobal({z});
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");
   const auto L   = mdd.downIntState(d,0,INT_MAX,MinFun);
   const auto U   = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto len = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto Lup   = mdd.upIntState(d,0,INT_MAX,MinFun);
   const auto Uup   = mdd.upIntState(d,0,INT_MAX,MaxFun);
   const auto lenUp = mdd.upIntState(d,0,INT_MAX,MaxFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto,int val) {
      return ((parent.down[L] + val * array[parent.down[len]] + child.up[Lup] <= z->max()) &&
              (parent.down[U] + val * array[parent.down[len]] + child.up[Uup] >= z->min()));
   });

   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= z->max()) && (n.down[U] + n.up[Uup] >= z->min());
   });
   ... // same as before
   mdd.onFixpoint([=](const auto& sink) {
      z->updateBounds(sink.down[L],sink.down[U]);
   });
   return d;
}
```

SumMDD and Objective

```
MDDCstrDesc::Ptr sum(MDD::Ptr m,const Factory::Veci& vars,const std::vector<int>& array,var<int>::Ptr z) {
   MDDSpec& mdd = m->getSpec();
   mdd.addGlobal({z});
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");
   const auto L    = mdd.downIntState(d,0,INT_MAX,MinFun);
   const auto U    = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto len = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto Lup    = mdd.upIntState(d,0,INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d,0,INT_MAX,MaxFun);
   const auto lenUp = mdd.upIntState(d,0,INT_MAX,MaxFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto,int val) {
      return ((parent.down[L] + val * array[parent.down[len]] + child.up[Lup] <= z->max()) &&
              (parent.down[U] + val * array[parent.down[len]] + child.up[Uup] >= z->min()));
   });

   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= z->max()) && (n.down[U] + n.up[Uup] >= z->min());
   });
   ... // same as before
   mdd.onFixpoint([=](const auto& sink) {
      z->updateBounds(sink.down[L],sink.down[U]);
   });
   return d;
}
```

MDD globally depends on z. Update MDD when z changes.

SumMDD and Objective

```
MDDCstrDesc::Ptr sum(MDD::Ptr m,const Factory::Veci& vars,const std::vector<int>& array,var<int>::Ptr z) {
   MDDSpec& mdd = m->getSpec();
   mdd.addGlobal({z});
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");
   const auto L    = mdd.downIntState(d,0,INT_MAX,MinFun);
   const auto U    = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto len = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto Lup    = mdd.upIntState(d,0,INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d,0,INT_MAX,MaxFun);
   const auto lenUp = mdd.upIntState(d,0,INT_MAX,MaxFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto,int val) {
      return ((parent.down[L] + val * array[parent.down[len]] + child.up[Lup] <= z->max()) &&
              (parent.down[U] + val * array[parent.down[len]] + child.up[Uup] >= z->min()));
   });

   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= z->max()) && (n.down[U] + n.up[Uup] >= z->min());
   });
   ... // same as before
   mdd.onFixpoint([=](const auto& sink) {
      z->updateBounds(sink.down[L],sink.down[U]);
   });
   return d;
}
```

MDD globally depends on z. Update MDD when z changes.

Conditioned on z

SumMDD and Objective

```
MDDCstrDesc::Ptr sum(MDD::Ptr m,const Factory::Veci& vars,const std::vector<int>& array,var<int>::Ptr z) {
   MDDSpec& mdd = m->getSpec();
   mdd.addGlobal({z});
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");
   const auto L    = mdd.downIntState(d,0,INT_MAX,MinFun);
   const auto U    = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto len = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto Lup    = mdd.upIntState(d,0,INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d,0,INT_MAX,MaxFun);
   const auto lenUp = mdd.upIntState(d,0,INT_MAX,MaxFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto,int val) {
      return ((parent.down[L] + val * array[parent.down[len]] + child.up[Lup] <= z->max()) &&
              (parent.down[U] + val * array[parent.down[len]] + child.up[Uup] >= z->min()));
   });

   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= z->max()) && (n.down[U] + n.up[Uup] >= z->min());
   });
   ... // same as before
   mdd.onFixpoint([=](const auto& sink) {
      z->updateBounds(sink.down[L],sink.down[U]);
   });
   return d;
}
```

MDD globally depends on z. Update MDD when z changes.

Conditioned on z

Conditioned on z

SumMDD and Objective

```
MDDCstrDesc::Ptr sum(MDD::Ptr m,const Factory::Veci& vars,const std::vector<int>& array,var<int>::Ptr z) {
   MDDSpec& mdd = m->getSpec();
   mdd.addGlobal({z});
   auto d = mdd.makeConstraintDescriptor(vars,"sumMDD");
   const auto L    = mdd.downIntState(d,0,INT_MAX,MinFun);
   const auto U    = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto len = mdd.downIntState(d,0,INT_MAX,MaxFun);
   const auto Lup    = mdd.upIntState(d,0,INT_MAX,MinFun);
   const auto Uup    = mdd.upIntState(d,0,INT_MAX,MaxFun);
   const auto lenUp = mdd.upIntState(d,0,INT_MAX,MaxFun);

   mdd.arcExist(d,[=] (const auto& parent,const auto& child,auto,int val) {
      return ((parent.down[L] + val * array[parent.down[len]] + child.up[Lup] <= z->max()) &&
              (parent.down[U] + val * array[parent.down[len]] + child.up[Uup] >= z->min()));
   });

   mdd.nodeExist([=](const auto& n) {
      return (n.down[L] + n.up[Lup] <= z->max()) && (n.down[U] + n.up[Uup] >= z->min());
   });
   ... // same as before
   mdd.onFixpoint([=](const auto& sink) {
      z->updateBounds(sink.down[L],sink.down[U]);
   });
   return d;
}
```

MDD globally depends on z. Update MDD when z changes.

Conditioned on z

Conditioned on z

tighten z at fixpoint (from the sink)

Exercise…

Exercise…

- Let’s do “atMost”!
  - Only the upper bound is given
  - Only reasoning with down information at first

Exercise…

- Let’s do “atMost”!
  - Only the upper bound is given
  - Only reasoning with down information at first
- Questions
  - What is the state representation?
  - What are the transitions?
  - What are the existence functions?

atMost

```
MDDCstrDesc::Ptr atMostMDD(MDD::Ptr m,const Factory::Veci& vars,const std::map<int,int>& ub)
{
   auto& spec = m->getSpec();
   auto [minFDom,minLDom] = domRange(vars);
   auto desc = spec.makeConstraintDescriptor(vars,"atMostMDD");

   std::map<int,MDDPInt::Ptr> pd;
   for(int i=minFDom; i <= minLDom;++i)
      pd[i] = spec.downIntState(desc,0,INT_MAX,MinFun);

   spec.arcExist(desc,[=](const auto& parent,const auto&,auto,int v) {
      return parent.down[pd.at(v)] < ub.at(v);
   });

   for(int i=minFDom; i <= minLDom;++i)
      spec.transitionDown(desc,pd[i],{pd[i]},{},[=](auto& out,const auto& parent,auto,const auto& val) {
         out[pd.at(i)] = parent.down[pd.at(i)] + (val.isSingleton() ? val.contains(i) : 0);
      });

   return desc;
}
```

Yet…

- This only handles the prefix…
  - Just reason on the suffix in the same way
  - Revise arc existence to use the suffix from the child

atMost

```
MDDCstrDesc::Ptr atMostMDD2(MDD::Ptr m,const Factory::Veci& vars,const std::map<int,int>& ub) {
   MDDSpec& spec = m->getSpec();
   auto [minFDom,minLDom] = domRange(vars);
   auto desc = spec.makeConstraintDescriptor(vars,"atMostMDD2");

   std::map<int,MDDPInt::Ptr> pd,pu;
   for(int i=minFDom; i <= minLDom;++i) {
      pd[i] = spec.downIntState(desc,0,INT_MAX,MinFun);
      pu[i] = spec.upIntState(desc,0,INT_MAX,MinFun);
   }

   spec.arcExist(desc,[=](const auto& parent,const auto& child,auto x,int v) {
      return parent.down[pd.at(v)] + child.up[pu.at(v)] < ub.at(v);
   });

   for(int i=minFDom; i <= minLDom;++i)
      spec.transitionDown(desc,pd[i],{pd[i]},{},[=](auto& out,const auto& parent,auto,const auto& val) {
         out[pd.at(i)] = parent.down[pd.at(i)] + (val.isSingleton() ? val.contains(i) : 0);
      });

   for(int i=minFDom; i <= minLDom;++i)
      spec.transitionUp(desc,pu[i],{pu[i]},{},[=](auto& out,const auto& child,auto,const auto& val) {
         out[pu.at(i)] = child.up[pu.at(i)] + (val.isSingleton() ? val.contains(i) : 0);
      });
   return desc;
}
```

There … and back again

- Model something a bit more unusual with CP+MDD
  - AIS : CSPLib-007
  - Find a serie of *n* numbers $(s_0,s_1,\ldots,s_{n-1})$
    - That is a permutation
    - For which the absolute values of consecutive pairs $(|s_1-s_0|,|s_2-s_1|, \ldots ,|s_{n-1} - s_{n-2}|)$ is a permutation of $(1,\ldots n-1)$

Questions to answer

- First, let's whip a pure CP model in MiniCPP
  - Two arrays of variables
    - serie : xVars
    - absolute values : yVars

AIS - Pure CP

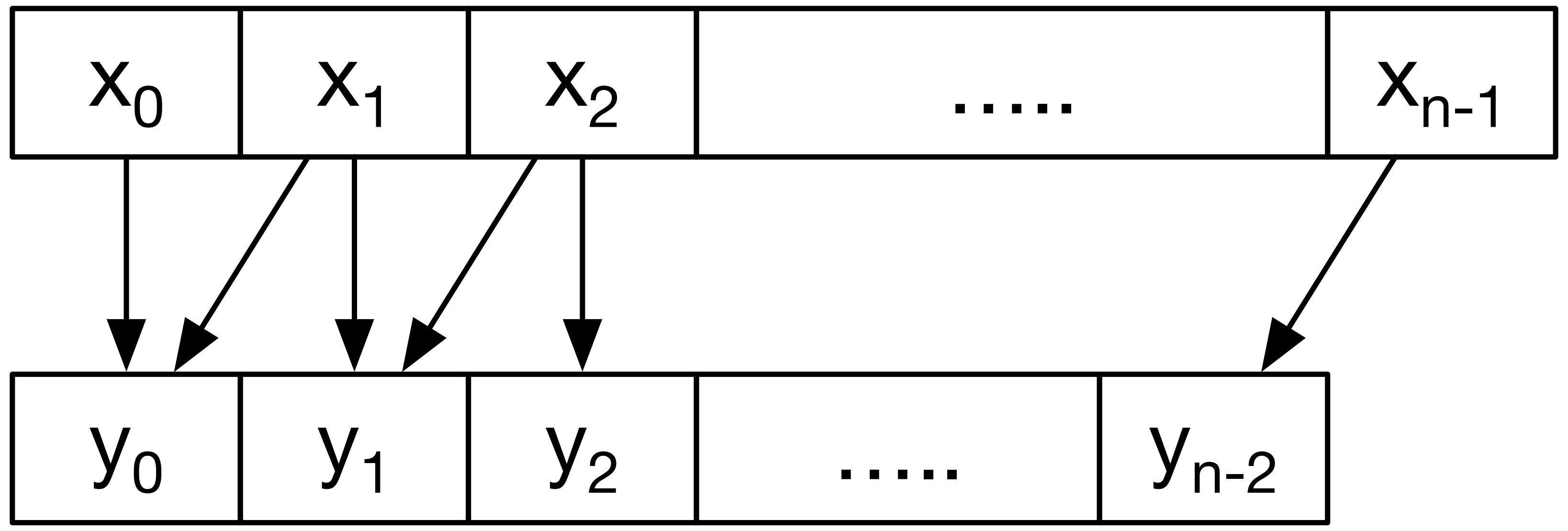
```
int main(int argc,char* argv[]) {
   using namespace Factory;
   int N     = (argc >= 2 && strncmp(argv[1],"-n",2)==0) ? atoi(argv[1]+2) : 8;

   CPSolver::Ptr cp  = Factory::makeSolver();
   auto x = Factory::intVarArray(cp, N,   0, N-1);
   auto y = Factory::intVarArray(cp, N-1,0, N-1);
   for (int i=0; i<N-1; i++)
      cp->post(y[i] != 0);

   cp->post(Factory::allDifferentAC(x));
   cp->post(Factory::allDifferentAC(y));
   for (int i=0; i<N-1; i++)
      cp->post(Factory::equalAbsDiff(y[i], x[i+1], x[i]));

   DFSearch search(cp,[=]() {
      return indomain_min(cp,selectFirst(x,[](const auto& xi) { return xi->size() > 1;}));
   });

   SearchStatistics stat = search.solve([](const SearchStatistics& stats) {
      return stats.numberOfSolutions() > INT_MAX;
   });
   std::cout << stat << "\n";
   return 0;
}
```

Going MDD…

- Decisions
  - We need an MDD LTS for allDifferent over n variables
  - We need MDD LTSs for all the $y_i=|x_i - x_{i-1}|$ (3 variables)
  - We need an MDD LTS for allDifferent over the y_i
  - We need to bundle them all in the same propagator

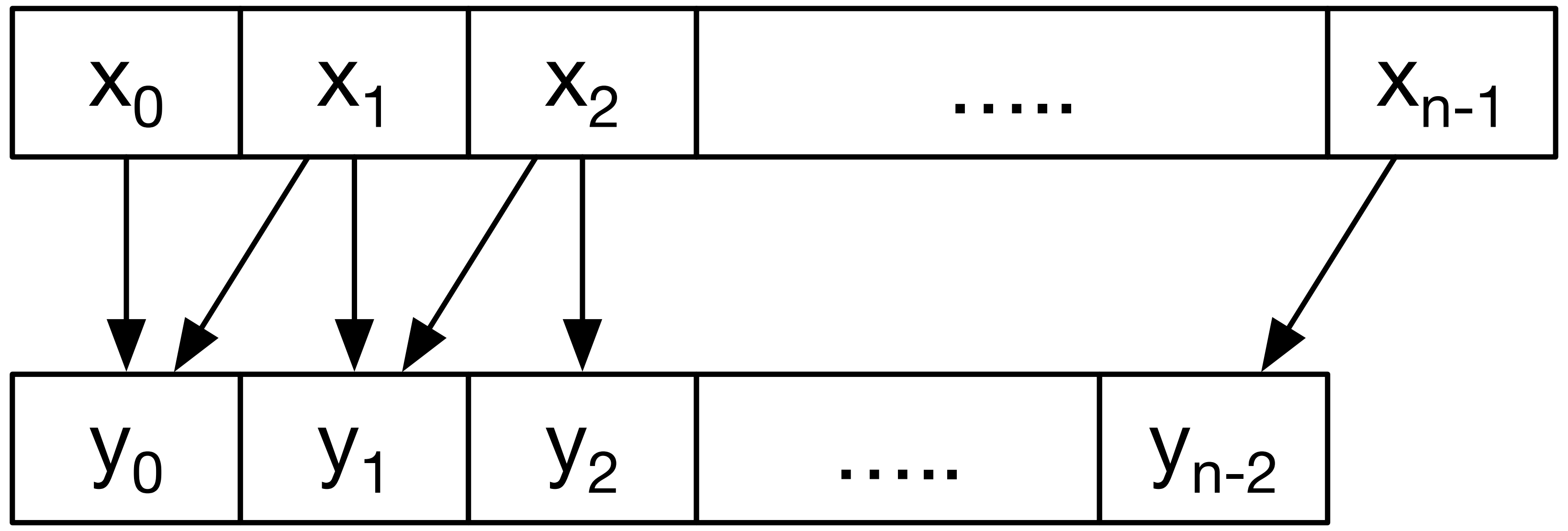
Variables and layers

- Classic CP representation



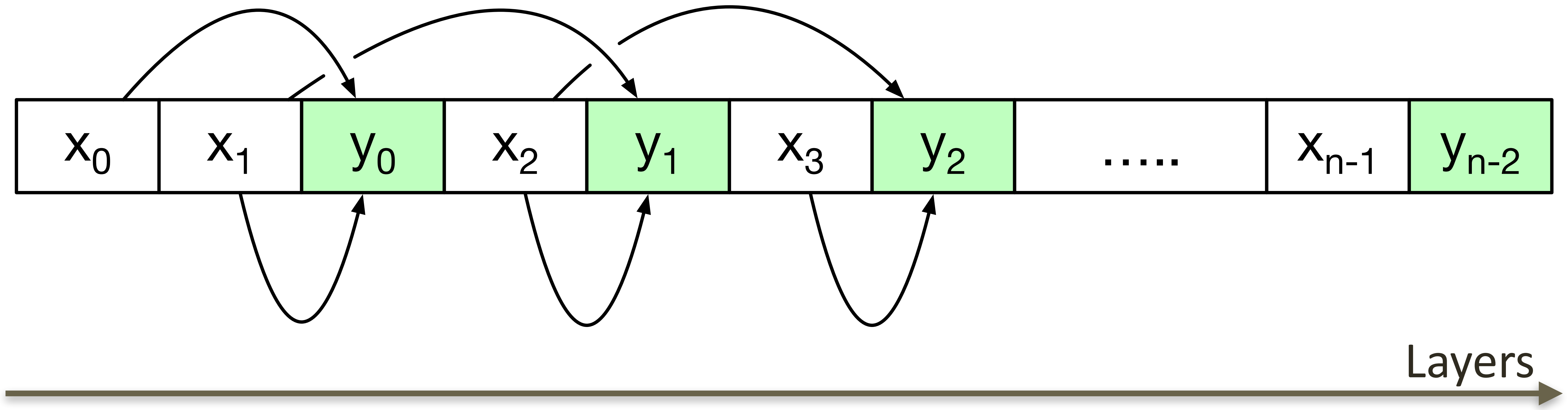
Variables and layers

- Classic CP representation



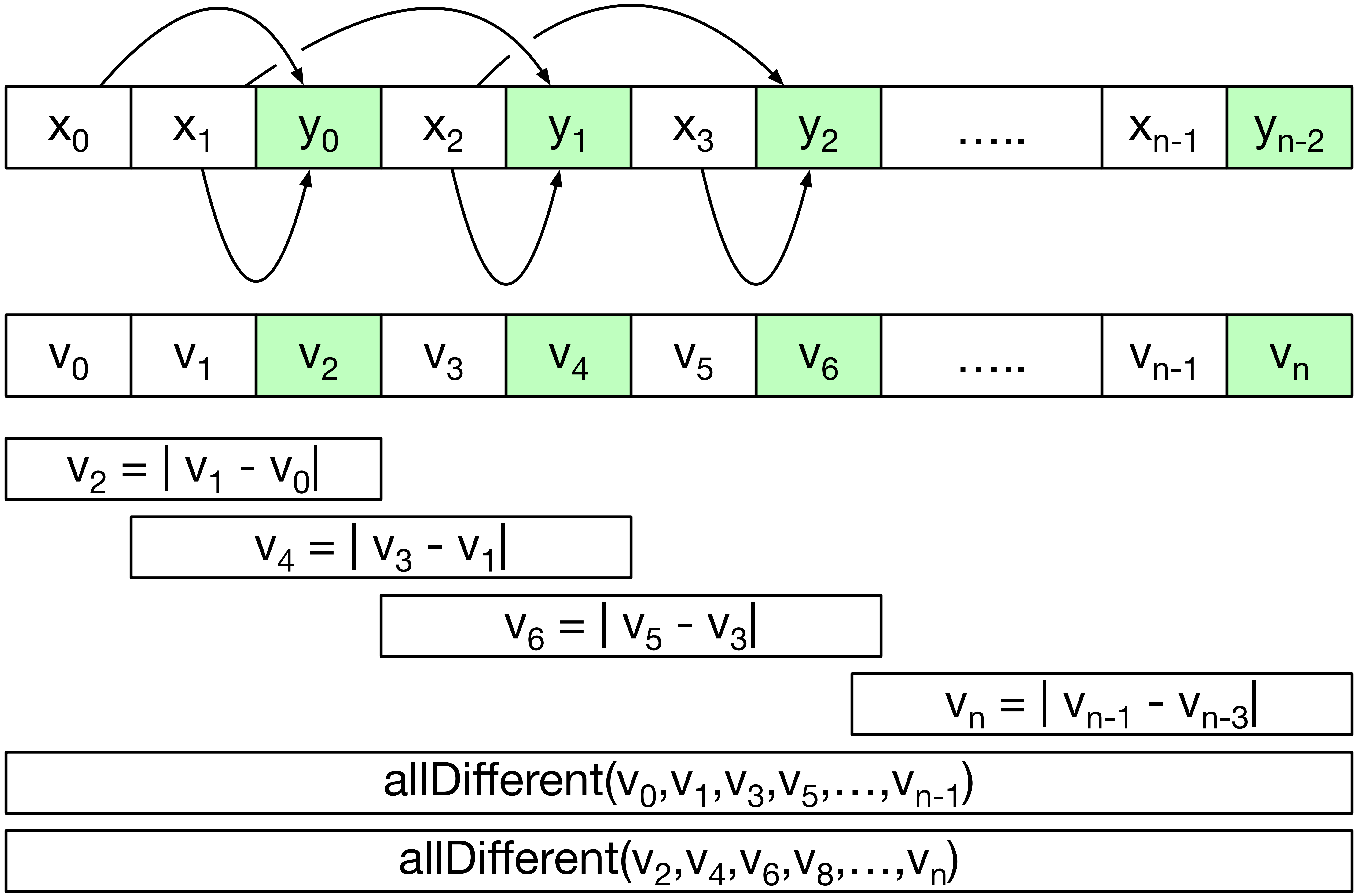
Variables and layers

- MDD Representation
  - Interleave x/y



Variables and layers

- MDD Representation
  - Interleave x/y
  - Lay down all MDDs
  - Some vars skipped



Observations

- Variable ordering
  - Driven by declaration order (now)
- Constraints in the MDD
  - Do not operate on consecutive variables
    - Variables not in scope are “skipped over”
  - Still have a local view for each MDD
  - Propagator does the “globalization”

The MDD-flavored model

```
int main(int argc,char* argv[]) {
   int N     = (argc >= 2 && strncmp(argv[1],"-n",2)==0) ? atoi(argv[1]+2)  : 8;
   int width = (argc >= 3 && strncmp(argv[2],"-w",2)==0) ? atoi(argv[2]+2)  : 1;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto v = Factory::intVarArray(cp, 2*N-1, 0, N-1);
   set<int> xIdx = filter(range(0,2*N-2),[](int i) {return i==0 || i%2!=0;});
   set<int> yIdx = filter(range(2,2*N-2),[](int i) {return i%2==0;});
   auto x = all(cp, xIdx, [&v](int i) {return v[i];});
   auto y = all(cp, yIdx, [&v](int i) {return v[i];});
   for (auto i=0u; i<y.size(); i++)
      cp->post(y[i] != 0);
   auto mdd = Factory::makeMDDRelax(cp,width,0);
   mdd->post(Factory::allDiffMDD(x));
   mdd->post(Factory::allDiffMDD(y));
   for(int i=0; i < N-1;i++)
      mdd->post(Factory::absDiffMDD(mdd,{y[i],x[i+1],x[i]}));
   cp->post(mdd);

   DFSearch search(cp,[=]() {
      return indomain_min(cp,selectFirst(x,[](const auto& x) { return x->size() > 1;}));
   });
   SearchStatistics stat = search.solve([](const SearchStatistics& stats) {
      return stats.numberOfSolutions() > INT_MAX;
   });
   cout << stat << "\n";
   return 0;
}
```

The MDD-flavored model

```
int main(int argc,char* argv[]) {
   int N     = (argc >= 2 && strncmp(argv[1],"-n",2)==0) ? atoi(argv[1]+2)  : 8;
   int width = (argc >= 3 && strncmp(argv[2],"-w",2)==0) ? atoi(argv[2]+2)  : 1;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto v = Factory::intVarArray(cp, 2*N-1, 0, N-1);
   set<int> xIdx = filter(range(0,2*N-2),[](int i) {return i==0 || i%2!=0;});
   set<int> yIdx = filter(range(2,2*N-2),[](int i) {return i%2==0;});
   auto x = all(cp, xIdx, [&v](int i) {return v[i];});
   auto y = all(cp, yIdx, [&v](int i) {return v[i];});
   for (auto i=0u; i<y.size(); i++)
      cp->post(y[i] != 0);
   auto mdd = Factory::makeMDDRelax(cp,width,0);
   mdd->post(Factory::allDiffMDD(x));
   mdd->post(Factory::allDiffMDD(y));
   for(int i=0; i < N-1;i++)
      mdd->post(Factory::absDiffMDD(mdd,{y[i],x[i+1],x[i]}));
   cp->post(mdd);

   DFSearch search(cp,[=]() {
      return indomain_min(cp,selectFirst(x,[](const auto& x) { return x->size() > 1;}));
   });
   SearchStatistics stat = search.solve([](const SearchStatistics& stats) {
      return stats.numberOfSolutions() > INT_MAX;
   });
   cout << stat << "\n";
   return 0;
}
```

Indexes of x/y variable extractions

The MDD-flavored model

```
int main(int argc,char* argv[]) {
   int N      = (argc >= 2 && strncmp(argv[1],"-n",2)==0) ? atoi(argv[1]+2)  : 8;
   int width = (argc >= 3 && strncmp(argv[2],"-w",2)==0) ? atoi(argv[2]+2)  : 1;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto v = Factory::intVarArray(cp, 2*N-1, 0, N-1);
   set<int> xIdx = filter(range(0,2*N-2),[](int i) {return i==0 || i%2!=0;});
   set<int> yIdx = filter(range(2,2*N-2),[](int i) {return i%2==0;});
   auto x = all(cp, xIdx, [&v](int i) {return v[i];});
   auto y = all(cp, yIdx, [&v](int i) {return v[i];});
   for (auto i=0u; i<y.size(); i++)
      cp->post(y[i] != 0);
   auto mdd = Factory::makeMDDRelax(cp,width,0);
   mdd->post(Factory::allDiffMDD(x));
   mdd->post(Factory::allDiffMDD(y));
   for(int i=0; i < N-1;i++)
      mdd->post(Factory::absDiffMDD(mdd,{y[i],x[i+1],x[i]}));
   cp->post(mdd);

   DFSearch search(cp,[=]() {
      return indomain_min(cp,selectFirst(x,[](const auto& x) { return x->size() > 1;}));
   });
   SearchStatistics stat = search.solve([](const SearchStatistics& stats) {
      return stats.numberOfSolutions() > INT_MAX;
   });
   cout << stat << "\n";
   return 0;
}
```

Indexes of x/y variable extractions

MDD-style!

The MDD-flavored model

```
int main(int argc,char* argv[]) {
   int N     = (argc >= 2 && strncmp(argv[1],"-n",2)==0) ? atoi(argv[1]+2)  : 8;
   int width = (argc >= 3 && strncmp(argv[2],"-w",2)==0) ? atoi(argv[2]+2)  : 1;
   CPSolver::Ptr cp  = Factory::makeSolver();
   auto v = Factory::intVarArray(cp, 2*N-1, 0, N-1);
   set<int> xIdx = filter(range(0,2*N-2),[](int i) {return i==0 || i%2!=0;});
   set<int> yIdx = filter(range(2,2*N-2),[](int i) {return i%2==0;});
   auto x = all(cp, xIdx, [&v](int i) {return v[i];});
   auto y = all(cp, yIdx, [&v](int i) {return v[i];});
   for (auto i=0u; i<y.size(); i++)
      cp->post(y[i] != 0);
   auto mdd = Factory::makeMDDRelax(cp,width,0);
   mdd->post(Factory::allDiffMDD(x));
   mdd->post(Factory::allDiffMDD(y));
   for(int i=0; i < N-1;i++)
      mdd->post(Factory::absDiffMDD(mdd,{y[i],x[i+1],x[i]}));
   cp->post(mdd);

   DFSearch search(cp,[=]() {
      return indomain_min(cp,selectFirs
   });
   SearchStatistics stat = search.solve
      return stats.numberOfSolutions()
   });
   cout << stat << "\n";
   return 0;
}
```

Indexes of x/y variable extractions

MDD-style!

Recall Pure CP

```
 for (int i=0; i<N-1; i++)
     cp->post(y[i] != 0);

  cp->post(Factory::allDifferentAC(x));
  cp->post(Factory::allDifferentAC(y));
  for (int i=0; i<N-1; i++)
     cp->post(Factory::equalAbsDiff(y[i], x[i+1], x[i]));
```

Peek behind the curtain

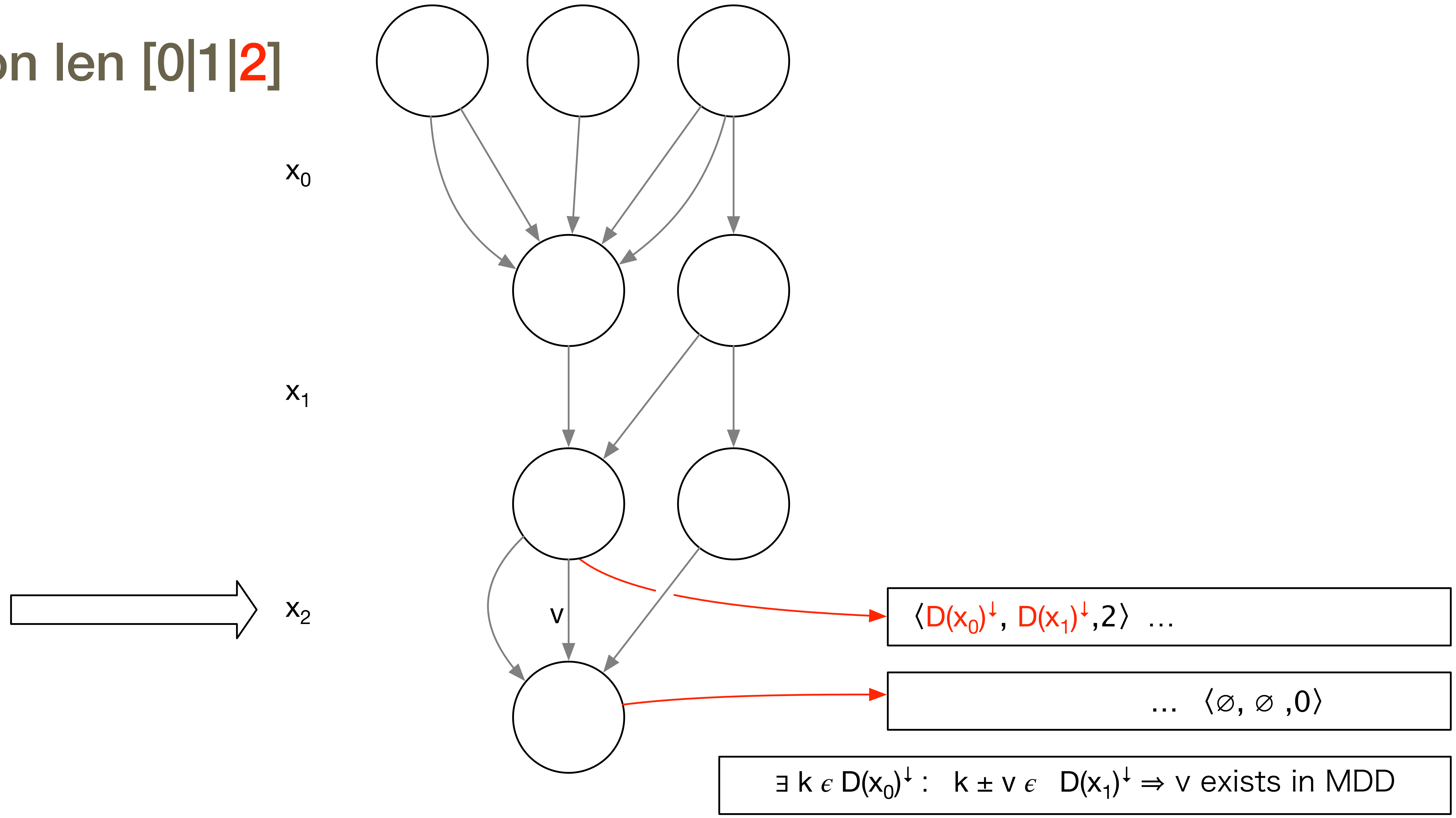
MDD for |x-y| = z ???

- Yes….
  - 3 variables
  - 1 ordering [x,y,z]
  - State definition:
    - down : <D(x),D(y),[0|1|2]>
    - up : <D(z),D(y),[0|1|2]>
  - Transitions
    - Down D(x) : Set a bit @ 1 for every value labeling inbound arcs
    - Down D(y) : Ditto
    - Down len : increase by 1 if variable is in scope
    - Up : same story for all three properties

MDD for |x-y| = z ???

- Arc Existence
  - Case analysis based on len [0|1|2]
- First case
  - Easy…
  - len = 2

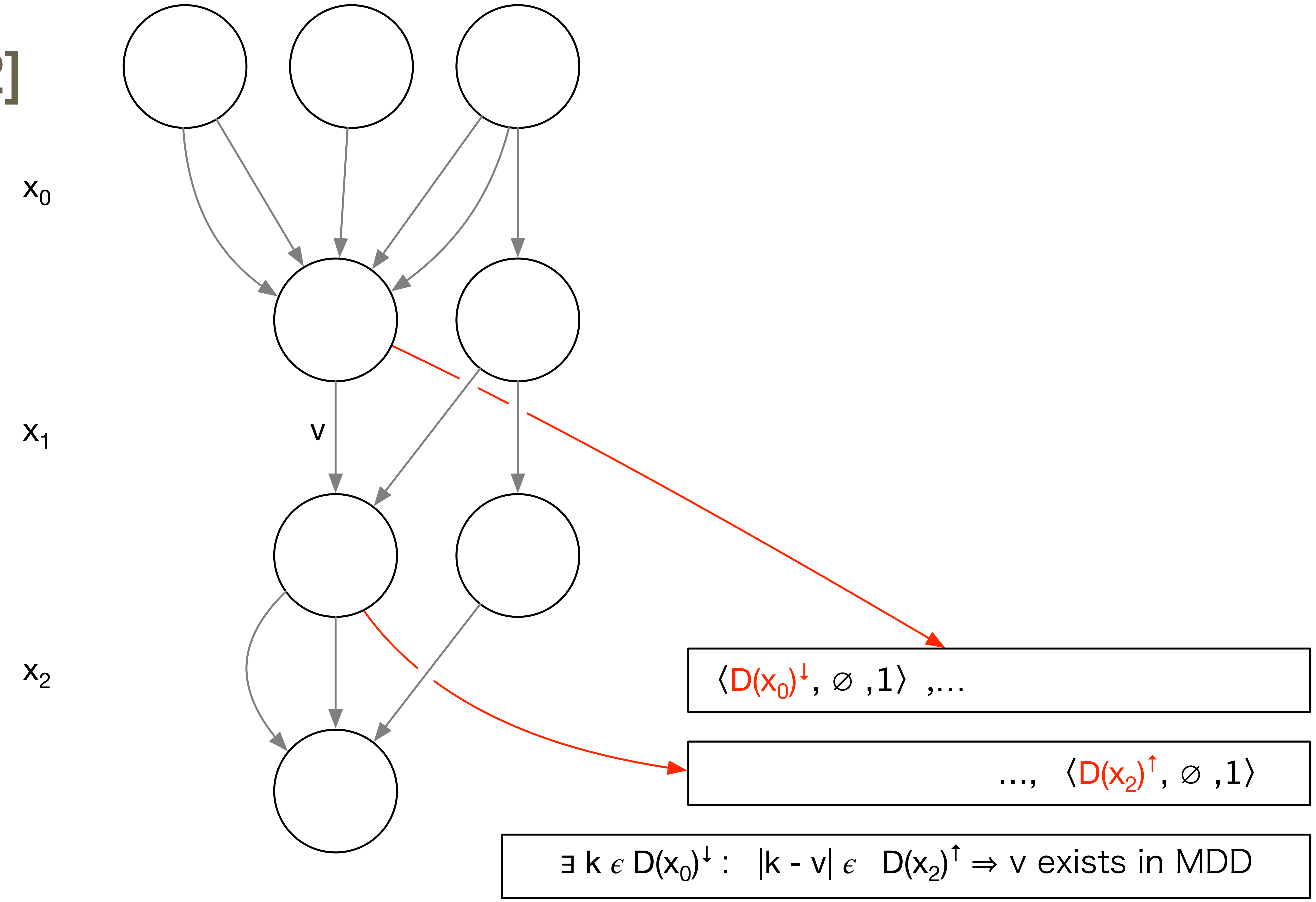
$$|x_0 - x_1| = x_2$$



MDD for |x-y| = z ???

- Arc Existence
  - Case analysis based on len [0|1|2]
- Second case
  - Use both!
  - len = 1

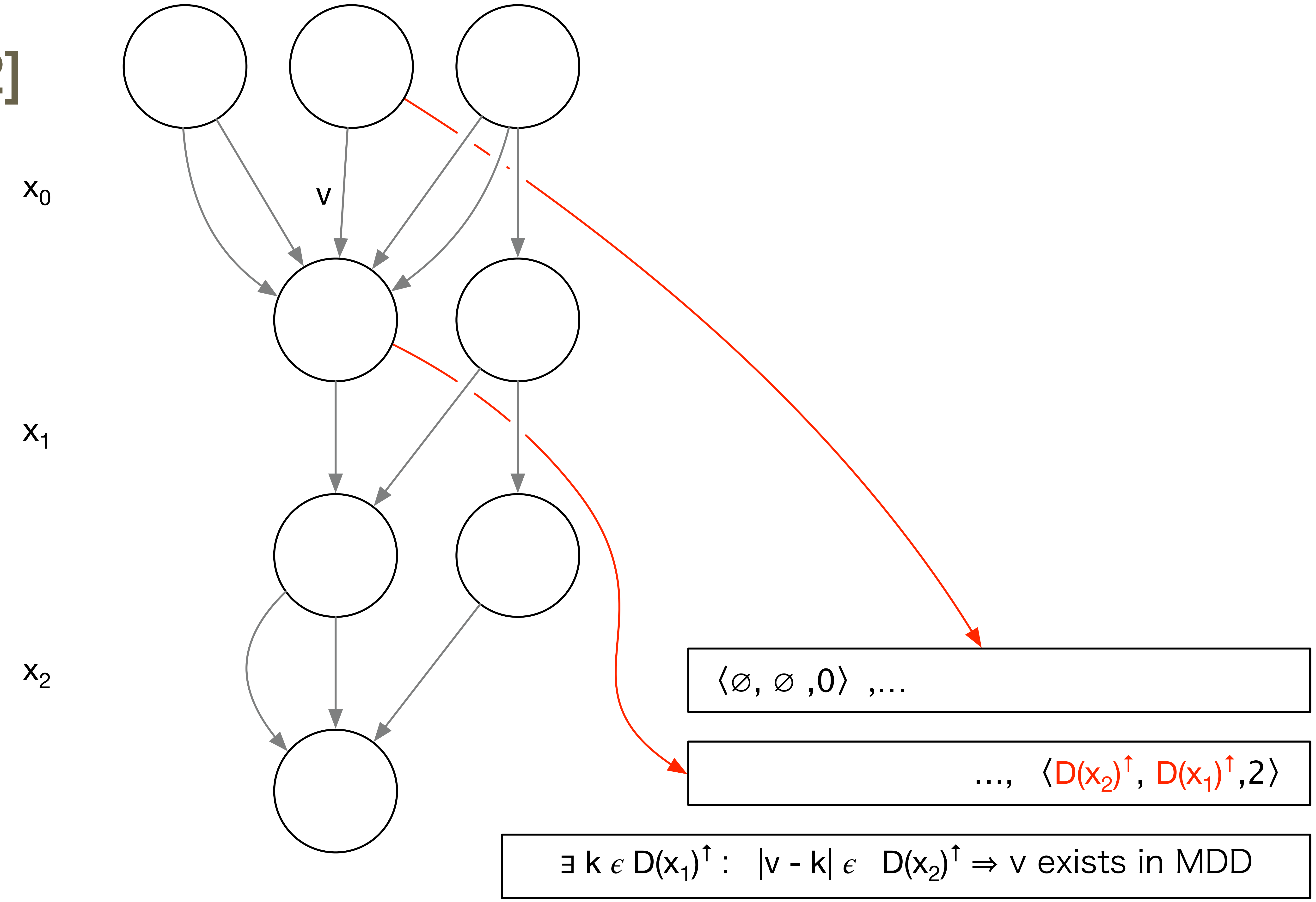
$$|x_0 - x_1| = x_2$$



MDD for |x-y| = z ???

- Arc Existence
  - Case analysis based on len [0|1|2]
- Third case
  - Just backward
  - len = 0

$$|x_0 - x_1| = x_2$$



Trying it out

- Demo time…

Wrap up

HADDOCK: A Language and Architecture for Decision Diagram Compilation. Rebecca Gentzel, Laurent Michel and Willem-Jan Van Hoeve, pp. 531–547, CP'20, LLN, Belgium

- Haddock
  - You can write MDD specifications (even for |x-y|==z)
  - You can compose in one (or more) propagators
  - You retain any custom search
  - You get the MDD benefit (smaller trees)
  - It all blends with traditional constraints (and small trees!)
- Going further
  - Customize with Heuristics ⇒

Heuristics for MDD Propagation in HADDOCK.
Rebecca Gentzel, Laurent Michel and Willem-Jan Van Hoeve
THURSDAY, AUGUST 4TH 11AM || 66A/Taub 7

- GIT
  - `git clone https://ldmbouge@bitbucket.org/ldmbouge/minicpp.git`

Agenda

- Decision Diagrams: Background
- Constraint Programming with Decision Diagrams
- Decision Diagrams within Constraint Programming Solvers
- Applications