Combinatorial Problem Solving with Serial and Nonserial Decision Diagrams

John Hooker
Carnegie Mellon University

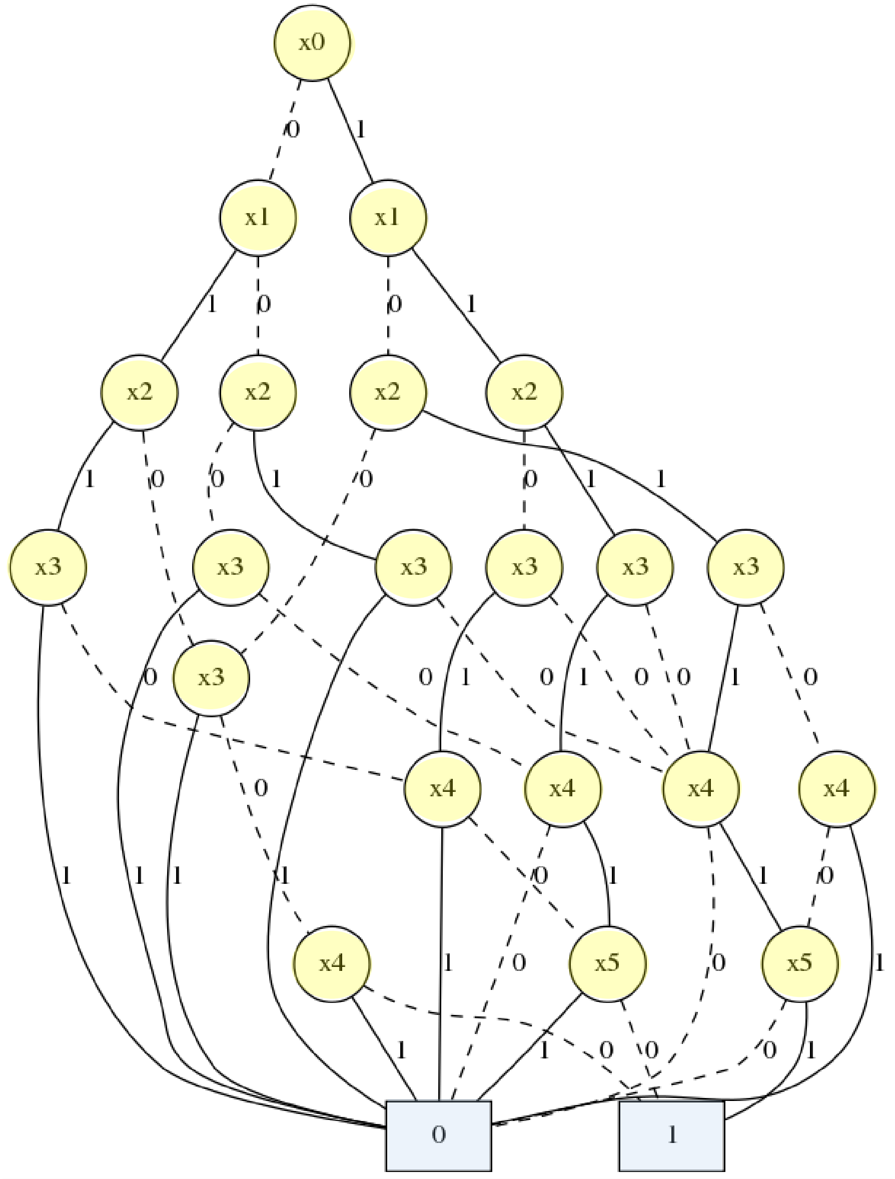
Workshop on SAT and Combinatorial Solving
Banff, Canada
January 2026

Carnegie Mellon University
Tepper School of Business

DD-based combinatorial problem solving

Decision diagrams (DDs) are an old idea (1970s).

They have been used for **logic circuit** design and **product configuration** since the 1980s.



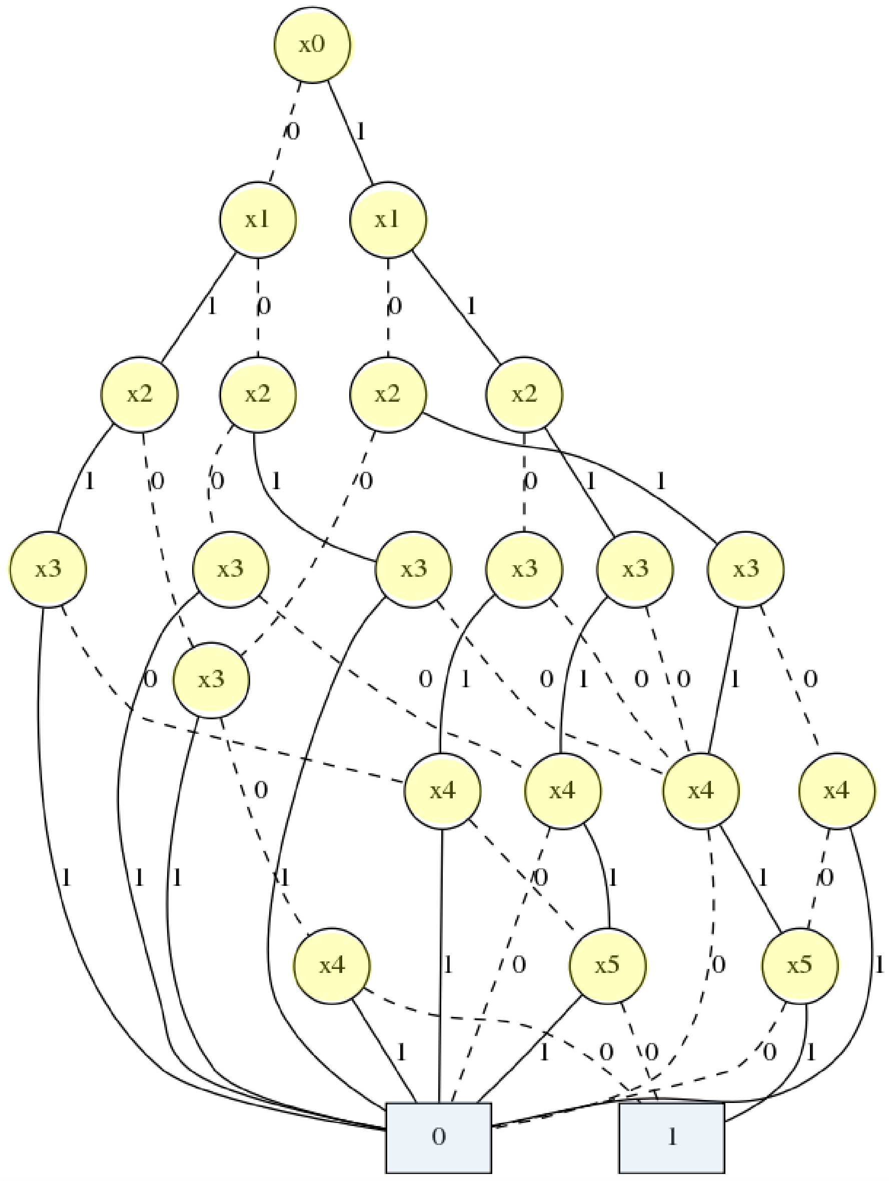
DD-based combinatorial problem solving

Decision diagrams (DDs) are an old idea (1970s).

They have been used for **logic circuit** design and **product configuration** since the 1980s.

DDs have recently been adapted to **combinatorial optimization** and **constraint solving**.

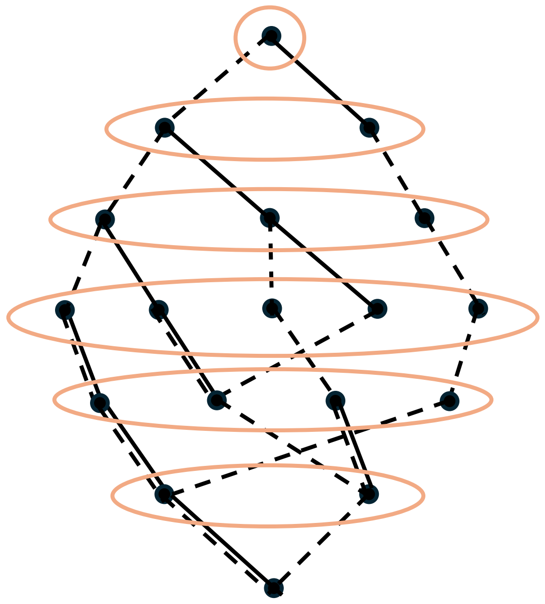
That is the **subject of this talk**.



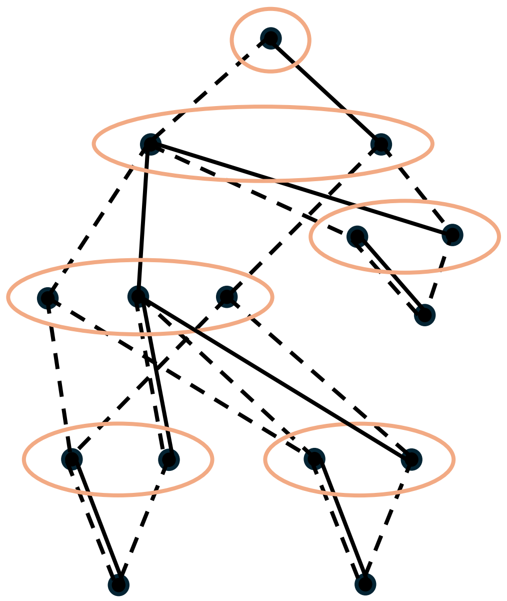
DD-based combinatorial problem solving

Part I. A brief **survey** of previous developments (**serial** DDs)

Part II. Recent work on **nonserial** DDs



Serial DD

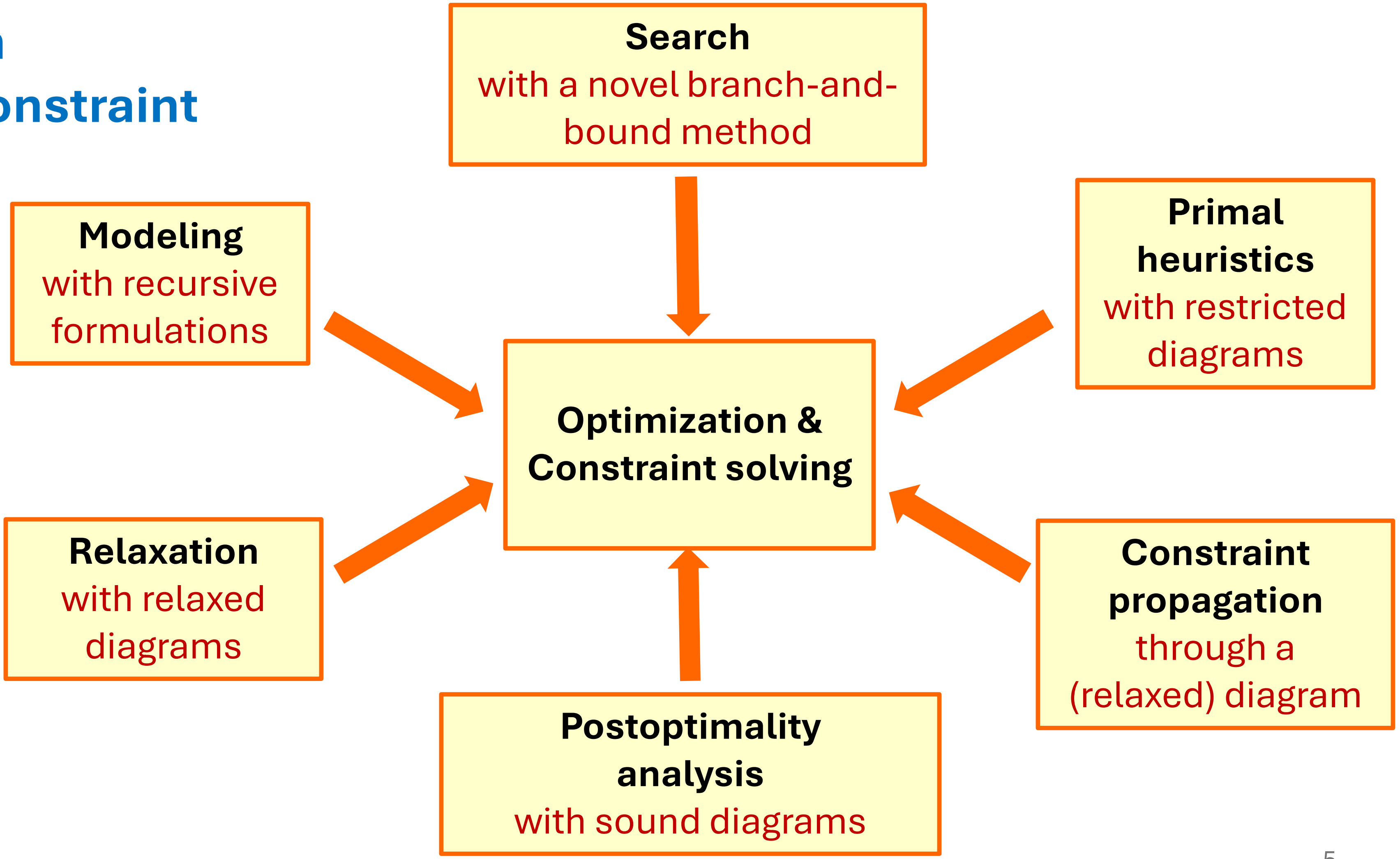


Nonserial DD

Carnegie Mellon University
Tepper School of Business

DD-based combinatorial problem solving

DDs can perform the **functions** one typically finds in **optimization** and **constraint solvers**.


Search with a novel branch-and-bound method
Modeling with recursive formulations
Primal heuristics with restricted diagrams
Optimization & Constraint solving
Relaxation with relaxed diagrams
Constraint propagation through a (relaxed) diagram
Postoptimality analysis with sound diagrams

DD-based combinatorial problem solving

Some advantages:

- Ideal for **recursive models** (dynamic programming)
- Discrete **problem relaxations** with adjustable tightness
- Fast **primal heuristics**
- No need for **linearity**, **convexity**, or **inequality** constraints
- Novel approach to **branch and bound**
- More effective domain propagation for **constraint programming**
- Highly **parallelizable**
- Comprehensive **postoptimality** analysis

DD-based combinatorial problem solving

Some advantages:

- Ideal for **recursive models** (dynamic programming)
- Discrete **problem relaxations** with adjustable tightness
- Fast **primal heuristics**
- No need for **linearity**, **convexity**, or **inequality** constraints
- Novel approach to **branch and bound**
- More effective domain propagation for **constraint programming**
- Highly **parallelizable**
- Comprehensive **postoptimality** analysis

Disadvantages:

- Unclear how to extend to **continuous variables**.*
- Reliance on good **heuristic choices** for tight relaxations

*But easily embedded in mixed discrete/continuous solvers

DD-based combinatorial problem solving

Some advantages:

- Ideal for **recursive models** (dynamic programming)
- Discrete **problem relaxations** with adjustable tightness
- Fast **primal heuristics**
- No need for **linearity**, **convexity**, or **inequality** constraints
- Novel approach to **branch and bound**
- More effective domain propagation for **constraint programming**
- Highly **parallelizable**.
- Comprehensive **postoptimality** analysis

Disadvantages:

- Unclear how to extend to **continuous variables**.*
- Reliance on good **heuristic choices** for tight relaxations

*But easily embedded in mixed discrete/continuous solvers

New advantage:

- Can exploit loosely coupled variables with **nonserial decision diagrams**

Origin of DDs

- **Boolean logic** Boole (1847,1854)
- **Switching circuits** interpreted as Boolean functions Peirce (1886) Shannon (1937)
- **Binary-decision programs** for representing switching circuits Lee (1959)
- **Graphical representation** of binary-decision programs (**BDD**s) Akers (1978)
- **Reduced ordered** BDDs Bryant (1986)
- **Applications** to circuit design and testing, product configuration, etc.
- DD-based **optimization** and **constraint programming** Hadžić & JH (2006,2007) Behle (2007)

Andersen, Hadžić, JH, Tiedemann (2007)

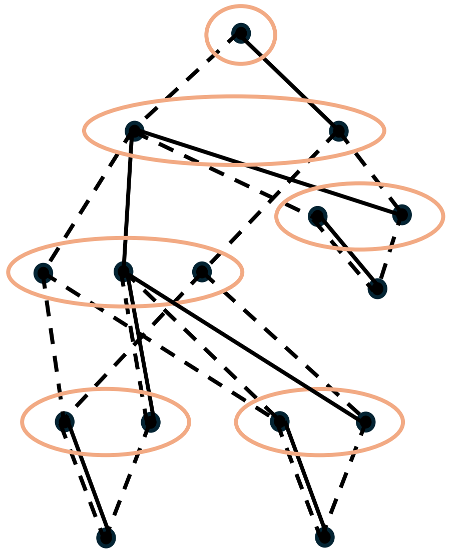
A comprehensive survey

M. P. Castro, A. A. Ciré, J. C. Beck, Decision diagrams for discrete optimization: A survey of recent advances, *INFORMS Journal on Computing* **34** (2022) 2271-2295

Outline

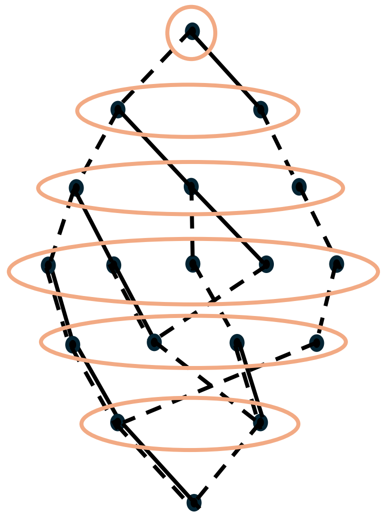
Part I – Survey of previous work
- DD basics
- Set packing example
- Reduced serial DDs
- Relaxed serial DDs
- Restricted serial DDs
- DD-based branch and bound
- DD-based constraint propagation
- DD-based Lagrangian relaxation
- Other developments

Part II – Nonserial DDs
- Treewidth
- Set packing example
- Computational results



Carnegie Mellon University
Tepper School of Business

Part I. Survey of Previous Work

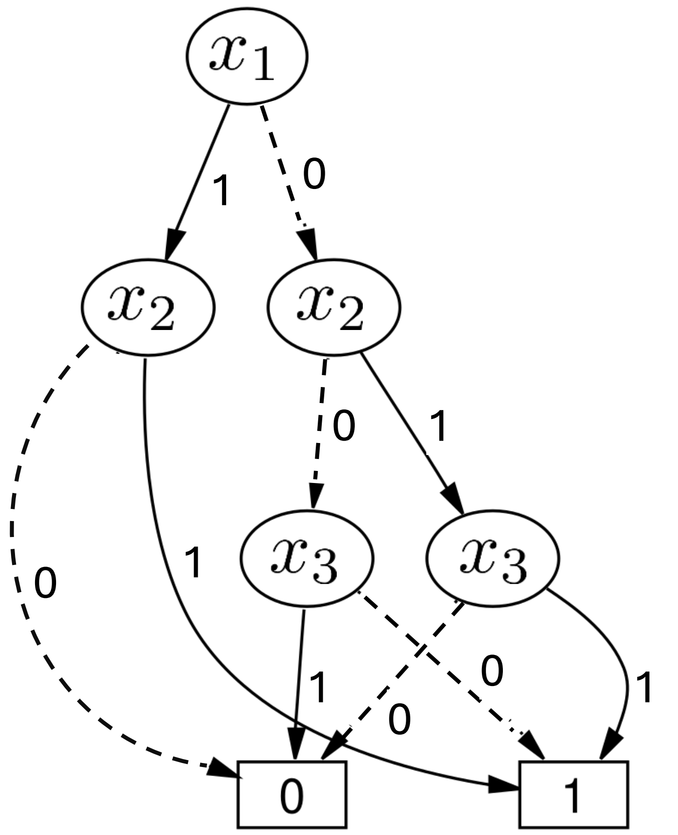


Carnegie Mellon University
Tepper School of Business

DD basics

Binary decision diagrams (BDDs) encode Boolean functions.

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



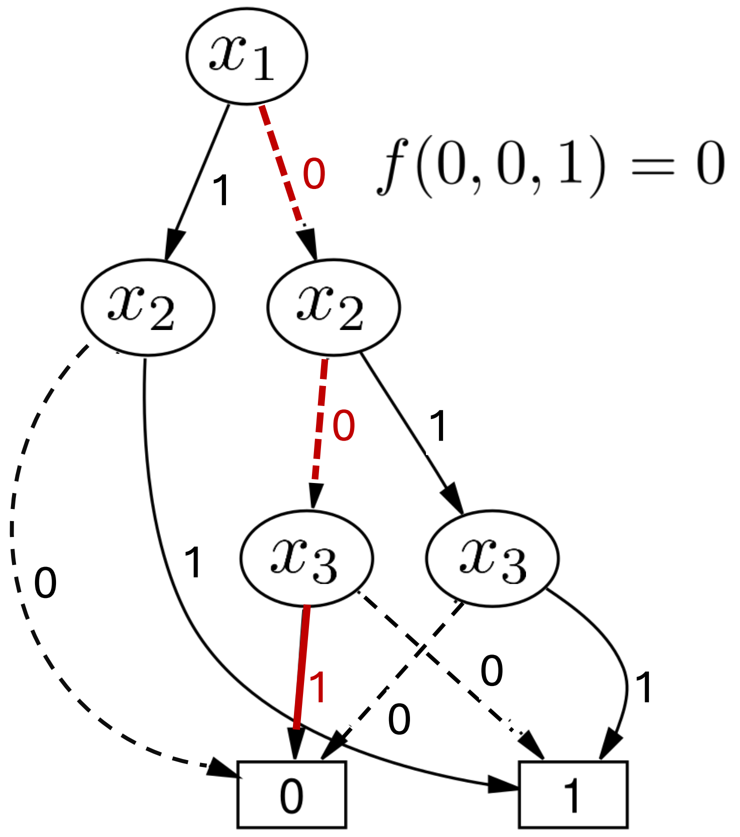
DD basics

Binary decision diagrams (BDDs) encode Boolean functions.

Paths to 1 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 1$

Paths to 0 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 0$

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



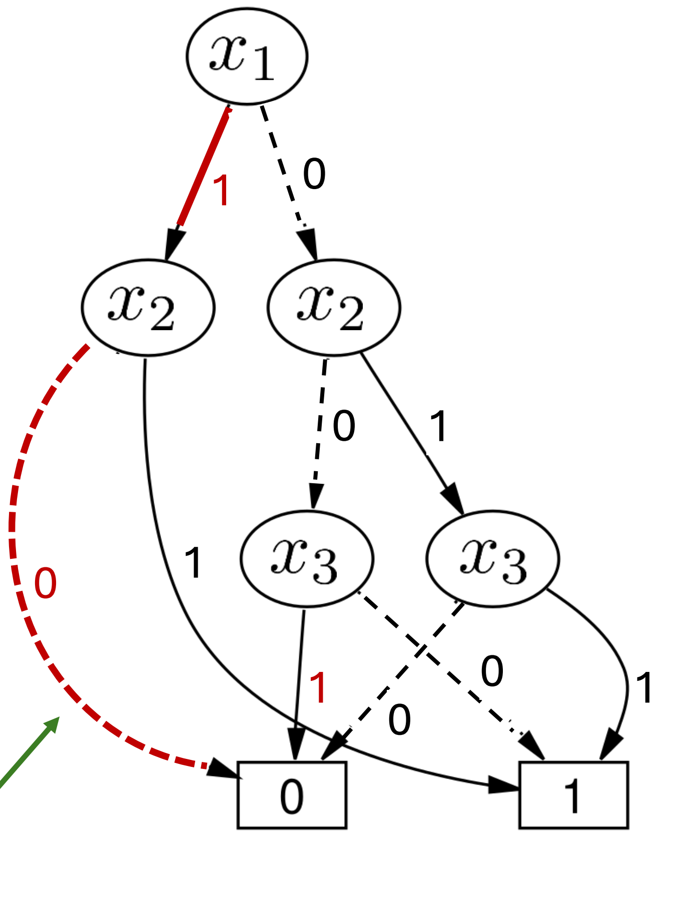
DD basics

Binary decision diagrams (BDDs) encode Boolean functions.

Paths to 1 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 1$

Paths to 0 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 1$

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Long arc indicates

$$f(1,0,0) = f(1,0,1) = 0$$

We will not use long arcs.

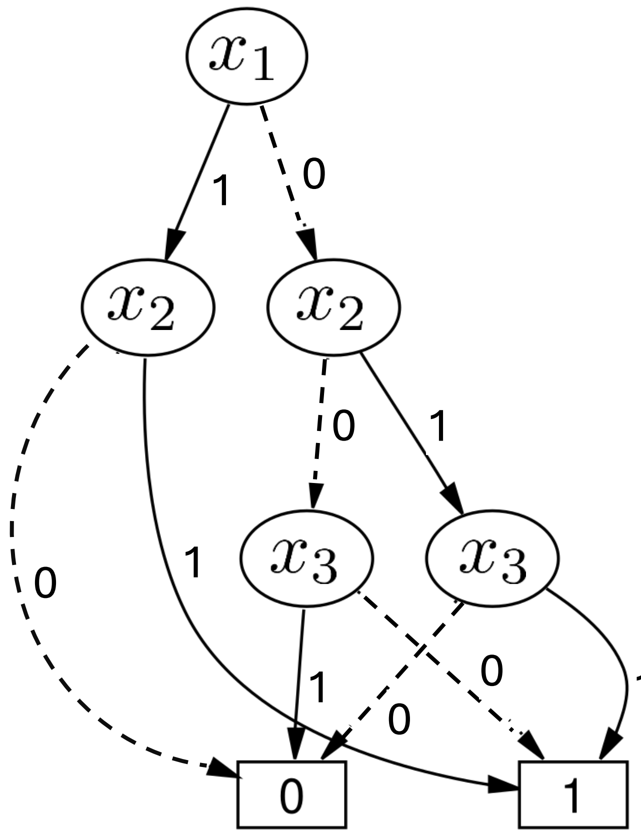
DD basics

Binary decision diagrams (BDDs) encode Boolean functions.

Paths to 1 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 1$

Paths to 0 node represent values of $\boldsymbol{x}$ for which $f(\boldsymbol{x}) = 1$

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



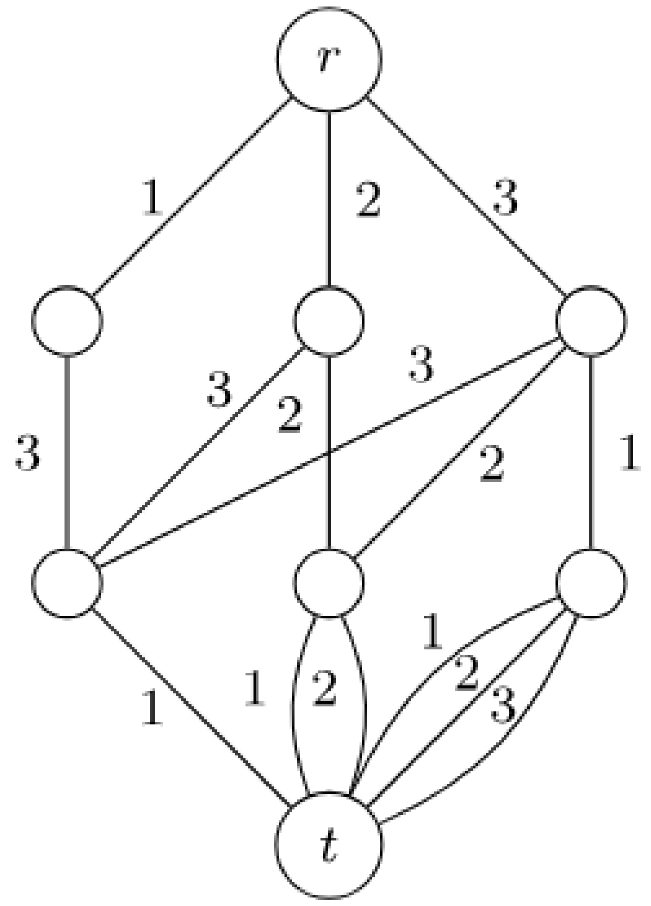
We will need paths only to the **1 terminal node**, to represent feasible solutions of a constraint set.

Carnegie Mellon University
Tepper School of Business

DD basics

Multivalued DDs allow for variables with multiple discrete values.

All results described here are valid for **both binary and multivalued DDs**.



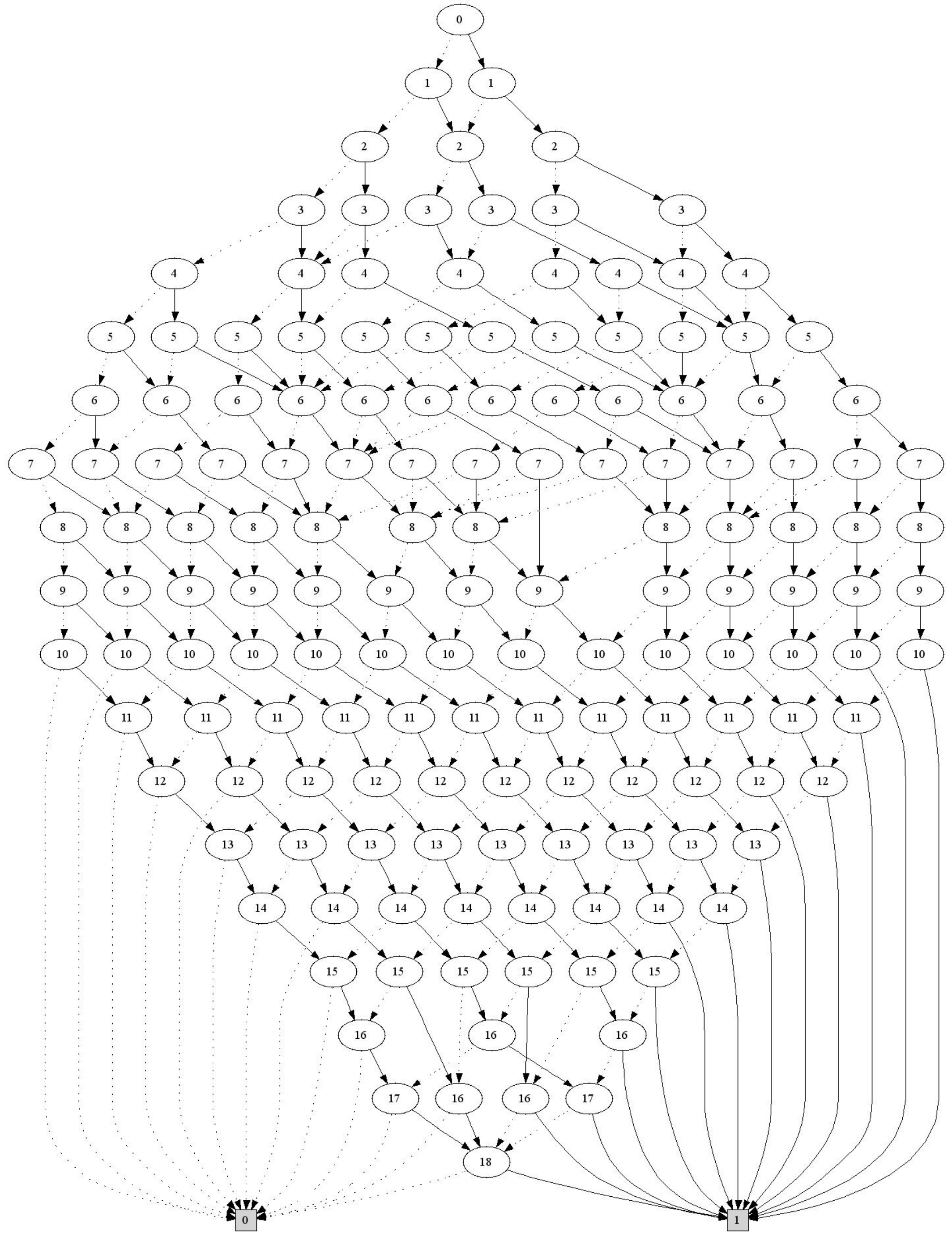
DD basics

DDs can compactly represent large feasible sets.

This BDD represents all 117,520 maximal 0-1 solutions of

$$300x_0 + 300x_1 + 285x_2 + 285x_3 + 265x_4$$
$$+ 265x_5 + 230x_6 + 230x_7 + 190x_8 + 200x_9$$
$$+ 400x_{10} + 200x_{11} + 400x_{12} + 200x_{13} + 400x_{14}$$
$$+ 200x_{15} + 400x_{16} + 200x_{17} + 400x_{18} \leq 2700$$

with only 152 nodes.



Carnegie Mellon University
Tepper School of Business

DD basics

DDs can compactly represent large feasible sets.

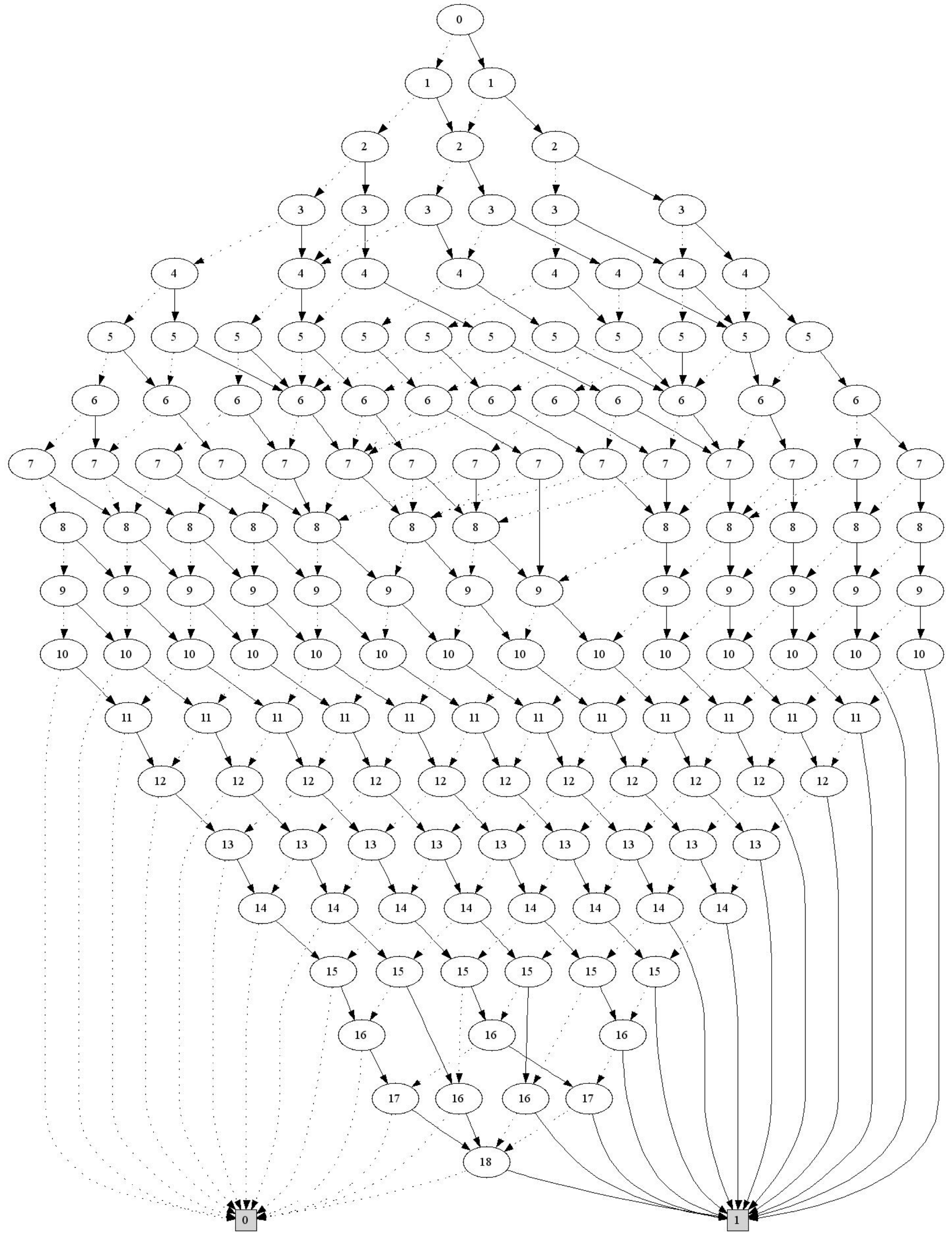
This BDD represents all 117,520 maximal 0-1 solutions of

$$300x_0 + 300x_1 + 285x_2 + 285x_3 + 265x_4 + 265x_5 + 230x_6 + 230x_7 + 190x_8 + 200x_9 + 400x_{10} + 200x_{11} + 400x_{12} + 200x_{13} + 400x_{14} + 200x_{15} + 400x_{16} + 200x_{17} + 400x_{18} \leq 2700$$

with only 152 nodes.

However, DDs can grow **exponentially** – for example, all permutations of 1, ..., *n*

Carnegie Mellon University
Tepper School of Business



Example
Set packing

Find a maximum subcollection of sets
in which no two sets have common elements.

```
{A,   C   }
{     C,D}
{A,B      }
{     C   }
{A        }
{   B,   D}
```

Example
Set packing

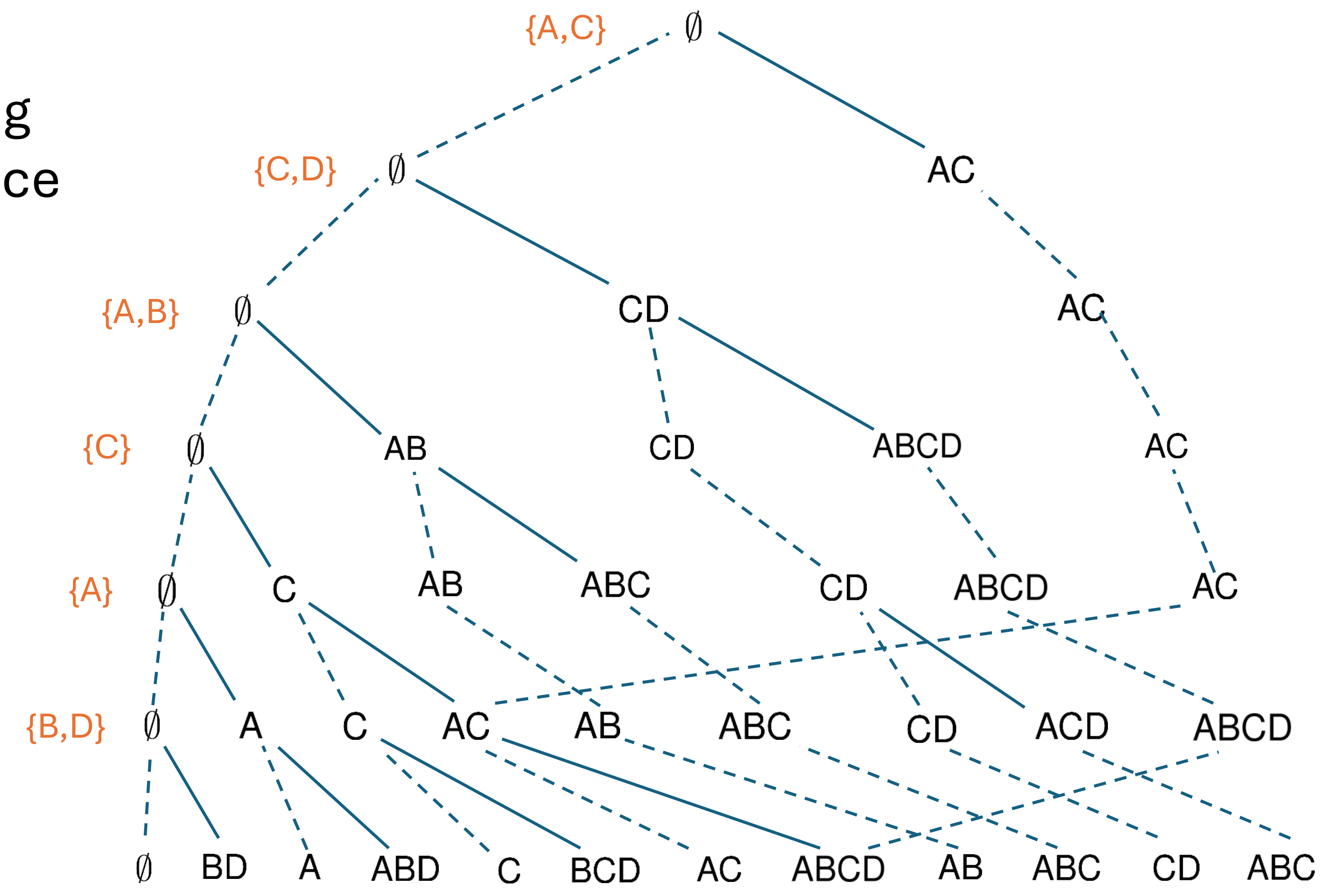
Find a maximum subcollection of sets
in which no two sets have common elements.

$$\begin{array}{l}\{A,\quad C\quad\}\\ \{\qquad C,D\}\\ \{A,B\qquad\}\\ \left.\begin{array}{l}\{\qquad C\quad\}\\ \{A\qquad\quad\}\\ \{\quad B,\quad D\}\end{array}\right\}\text{solution}\end{array}$$

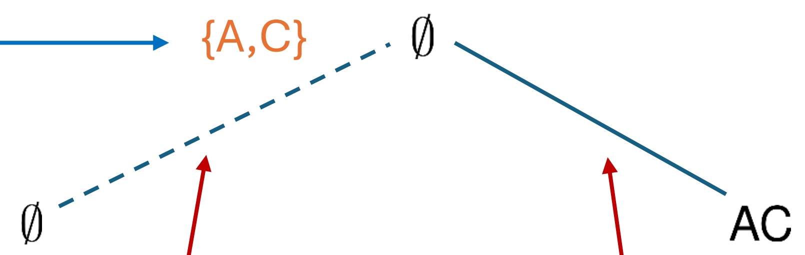
Serial DD
for a set packing problem instance

Layers correspond to selection decisions for each set.

Variables indicate the decisions (controls).



Decide whether to select set {A,C}

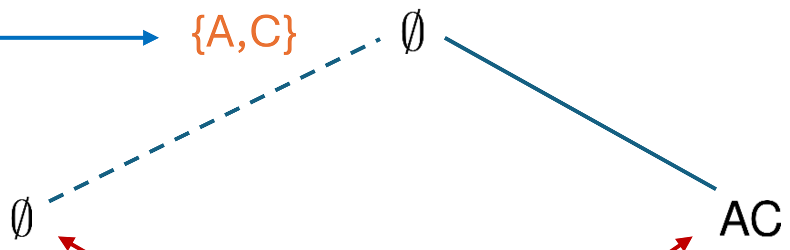


Don’t select set {A,C}

Select set {A,C}

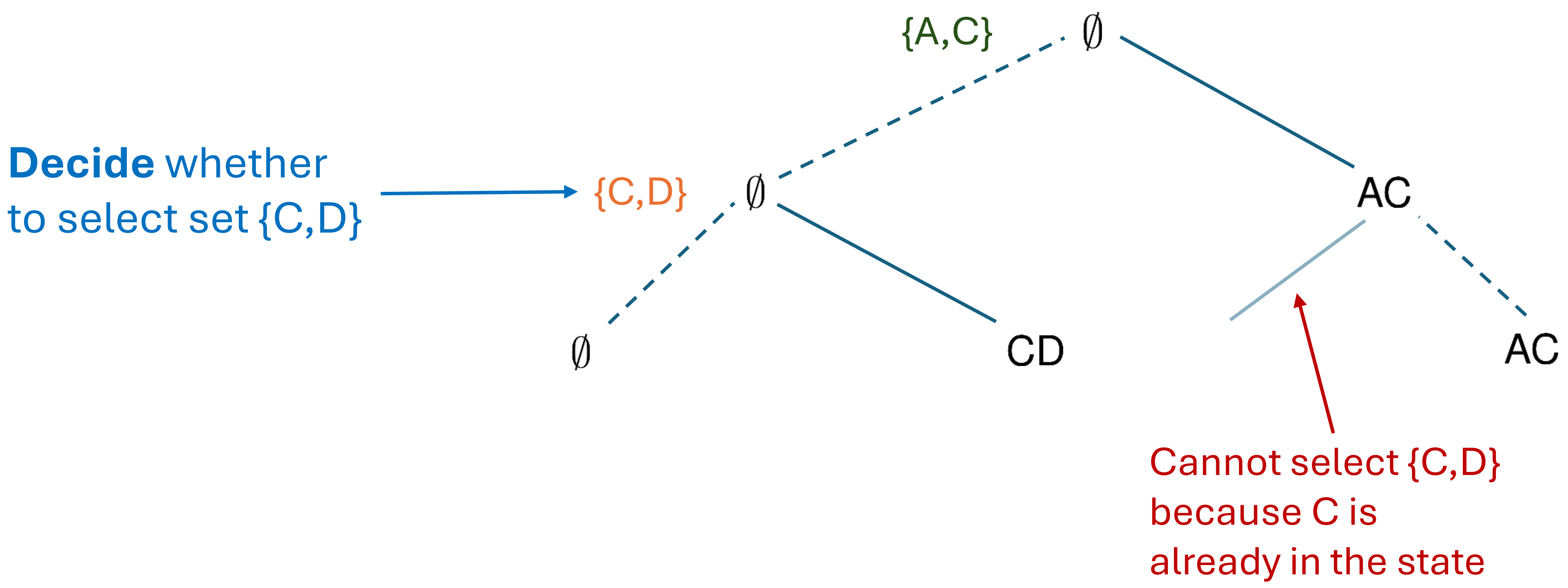
Carnegie Mellon University
Tepper School of Business

Decide whether to select set {A,C}



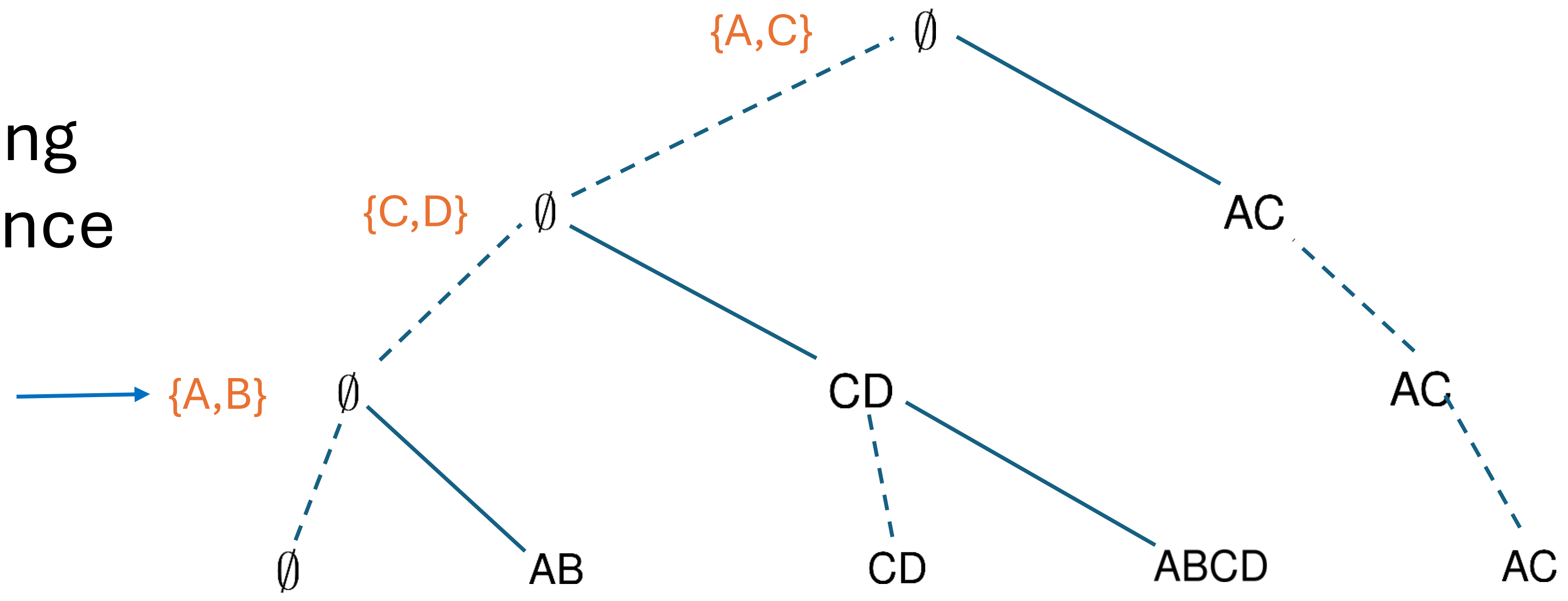
State consists of elements in sets so far selected.

As in dynamic programming.


{A,C}
∅
∅
AC
{C,D}
∅
CD
AC
Decide whether to select set {C,D}
Cannot select {C,D} because C is already in the state

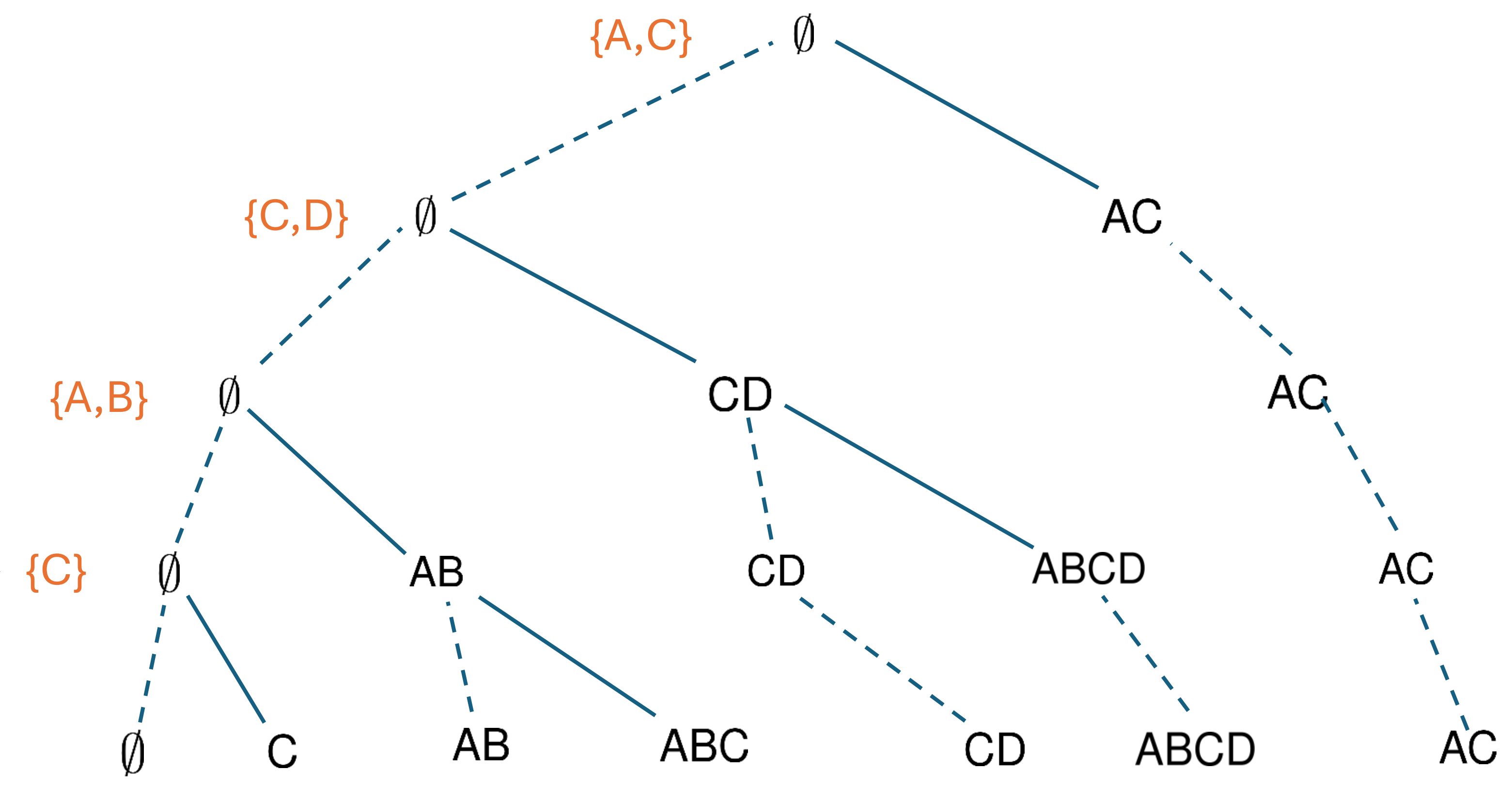
Serial DD
for a set packing problem instance

Decide whether to select set {A,B}



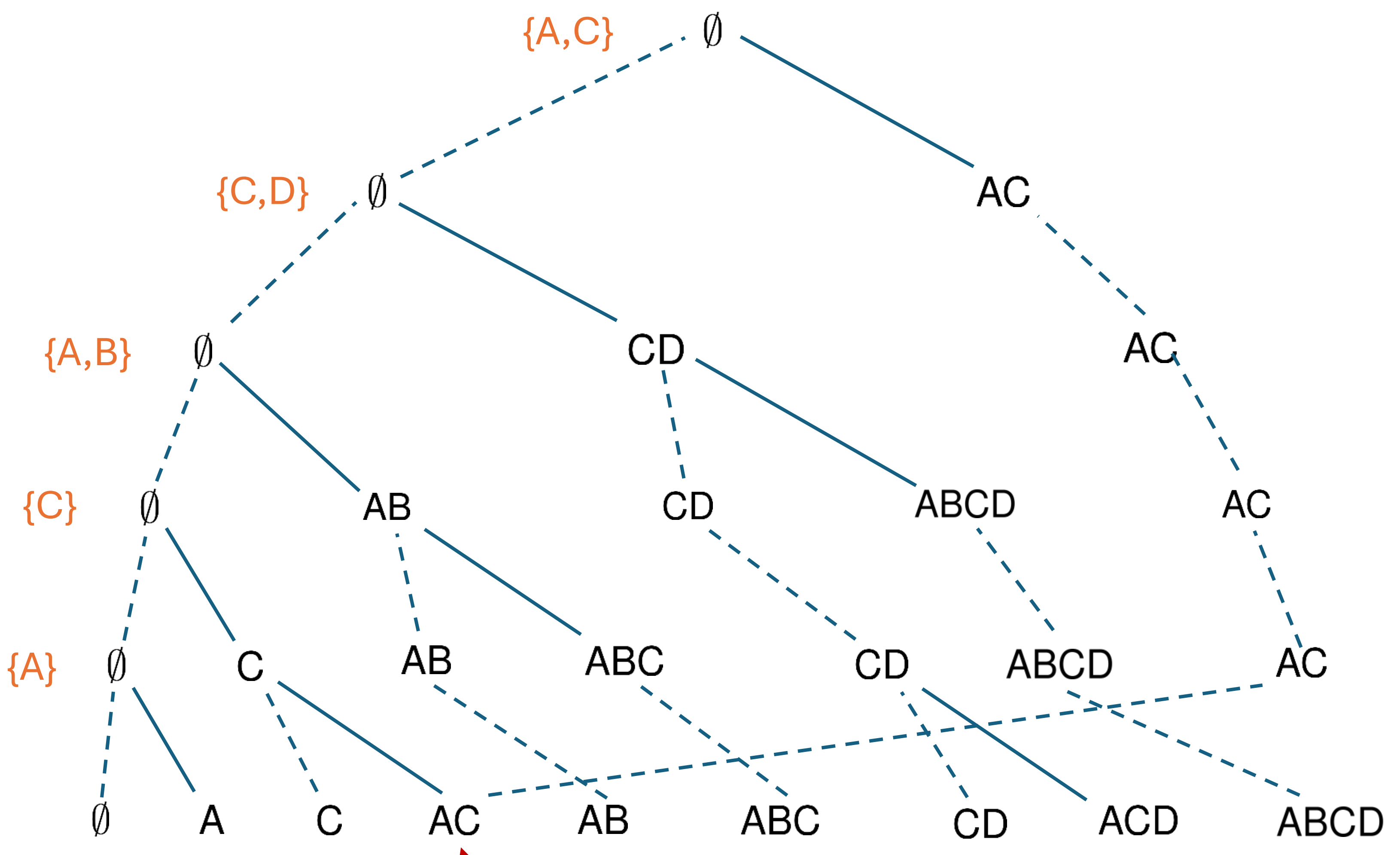
Serial DD
for a set packing problem instance

Decide whether to select set {C}



Carnegie Mellon University
Tepper School of Business

Serial DD
for a set packing problem instance

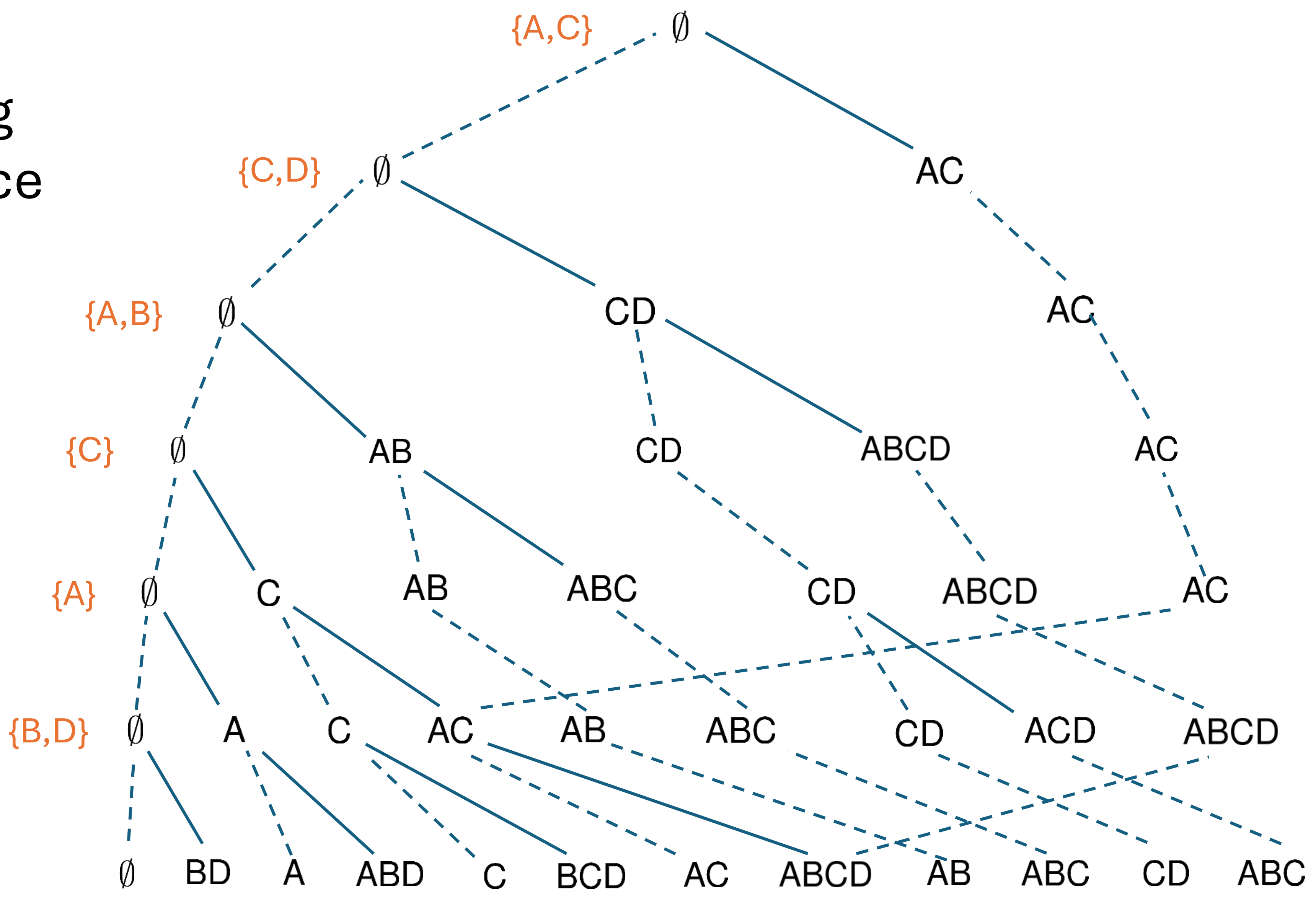


Decide whether to select set {A}

DD is not a tree because branches can terminate in the same state. This happens quite often in a DD.

Carnegie Mellon University
Tepper School of Business

Serial DD
for a set packing problem instance

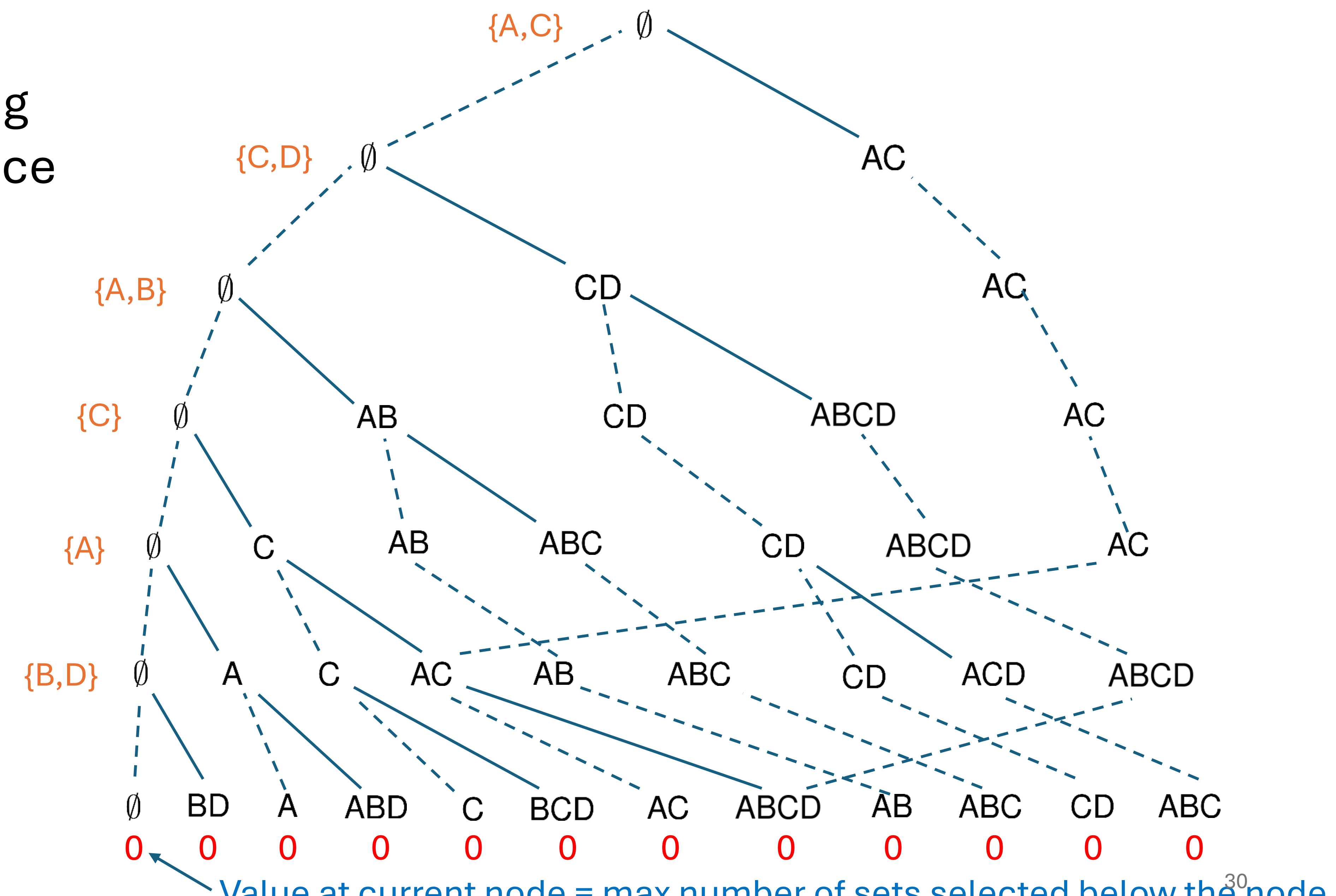


Decide whether to select set {B,D}

DD has 39 nodes

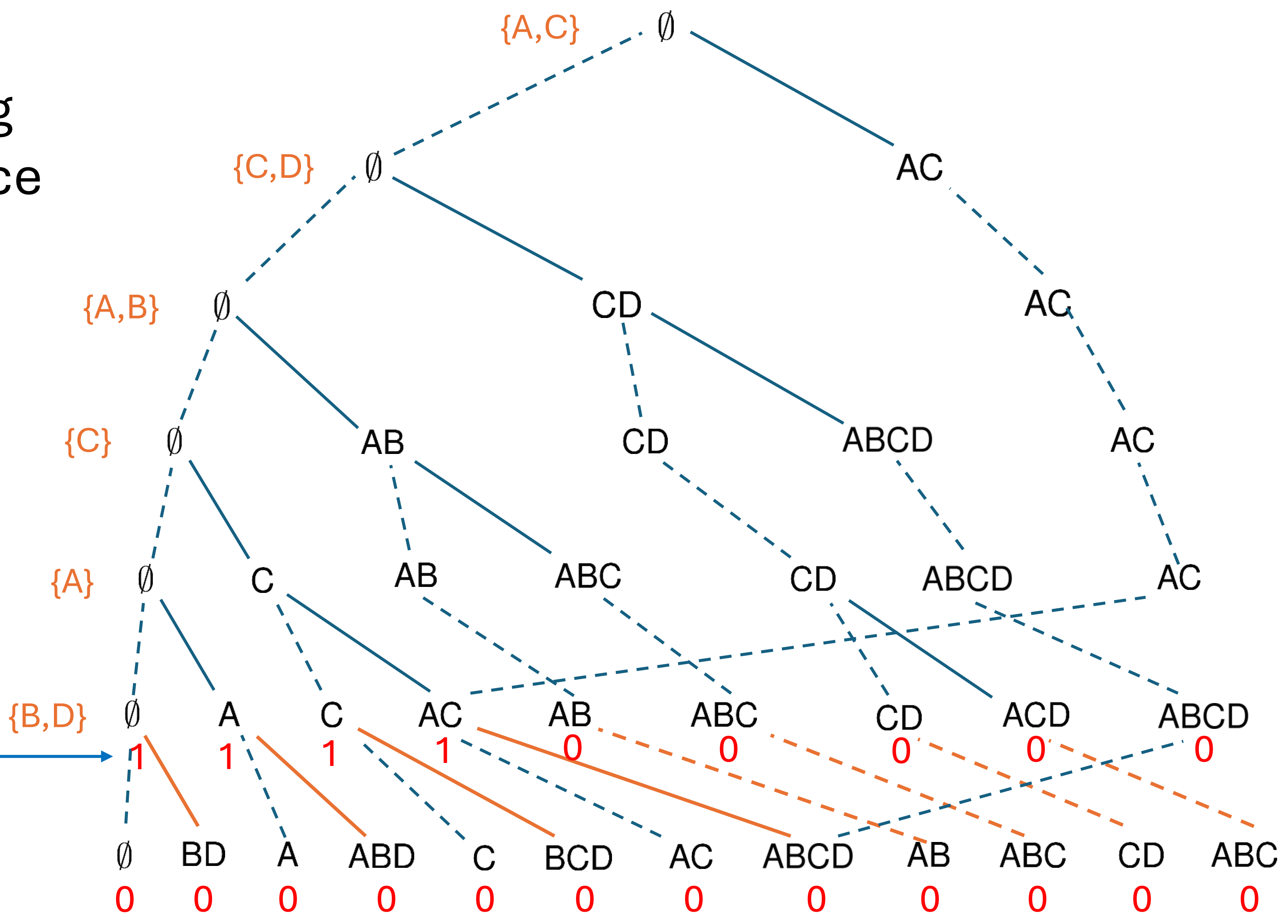
Serial DD
for a set packing problem instance

Now find an optimal solution recursively, using a **backward pass**, as in dynamic programming.



{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}
∅
AC
CD
AB
ABC
ABCD
ACD
BD
A
ABD
C
BCD
0
Value at current node = max number of sets selected below the node

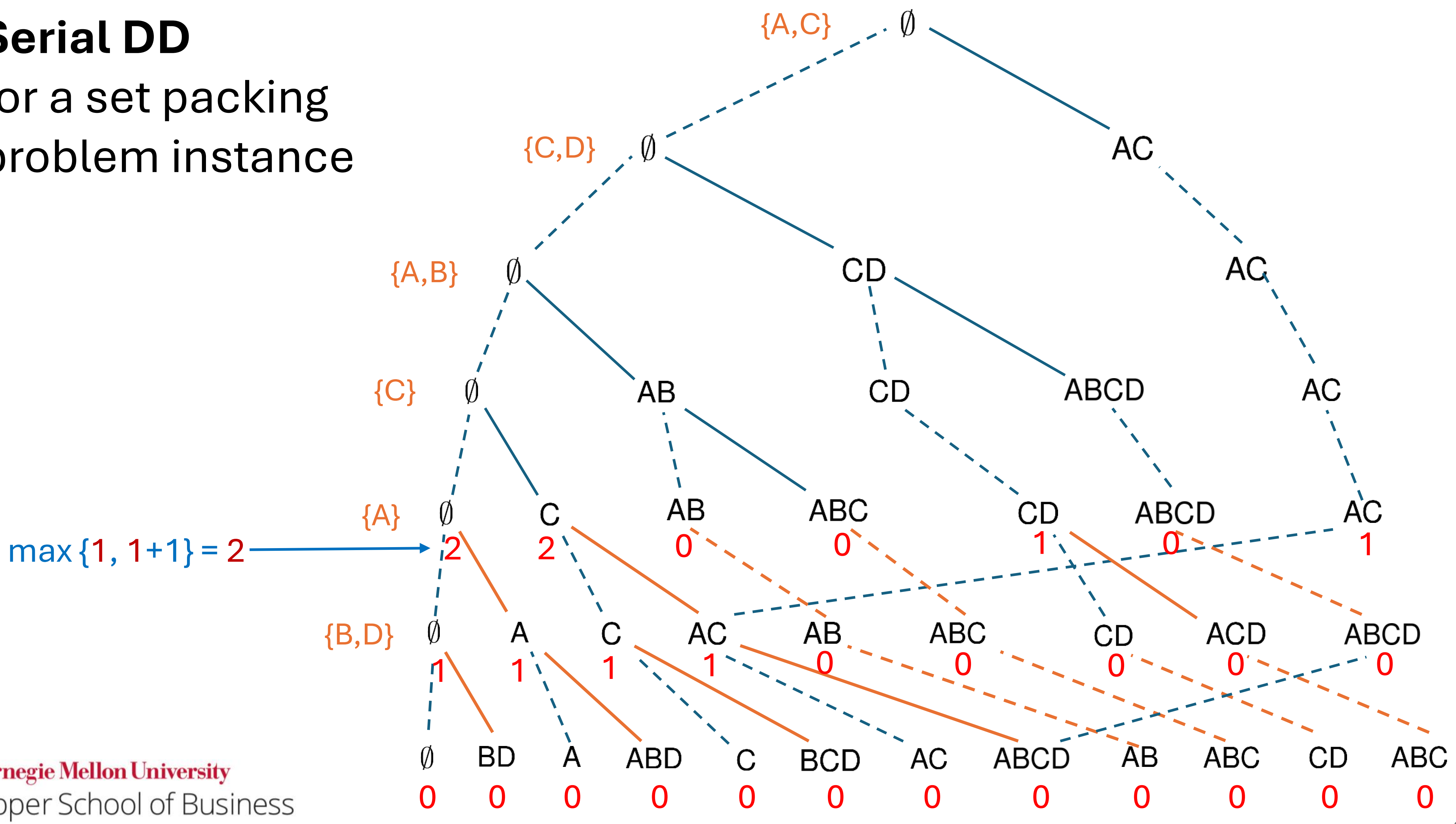
Serial DD
for a set packing problem instance



max {0, 0+1} = 1
Mark optimal decision with orange arc

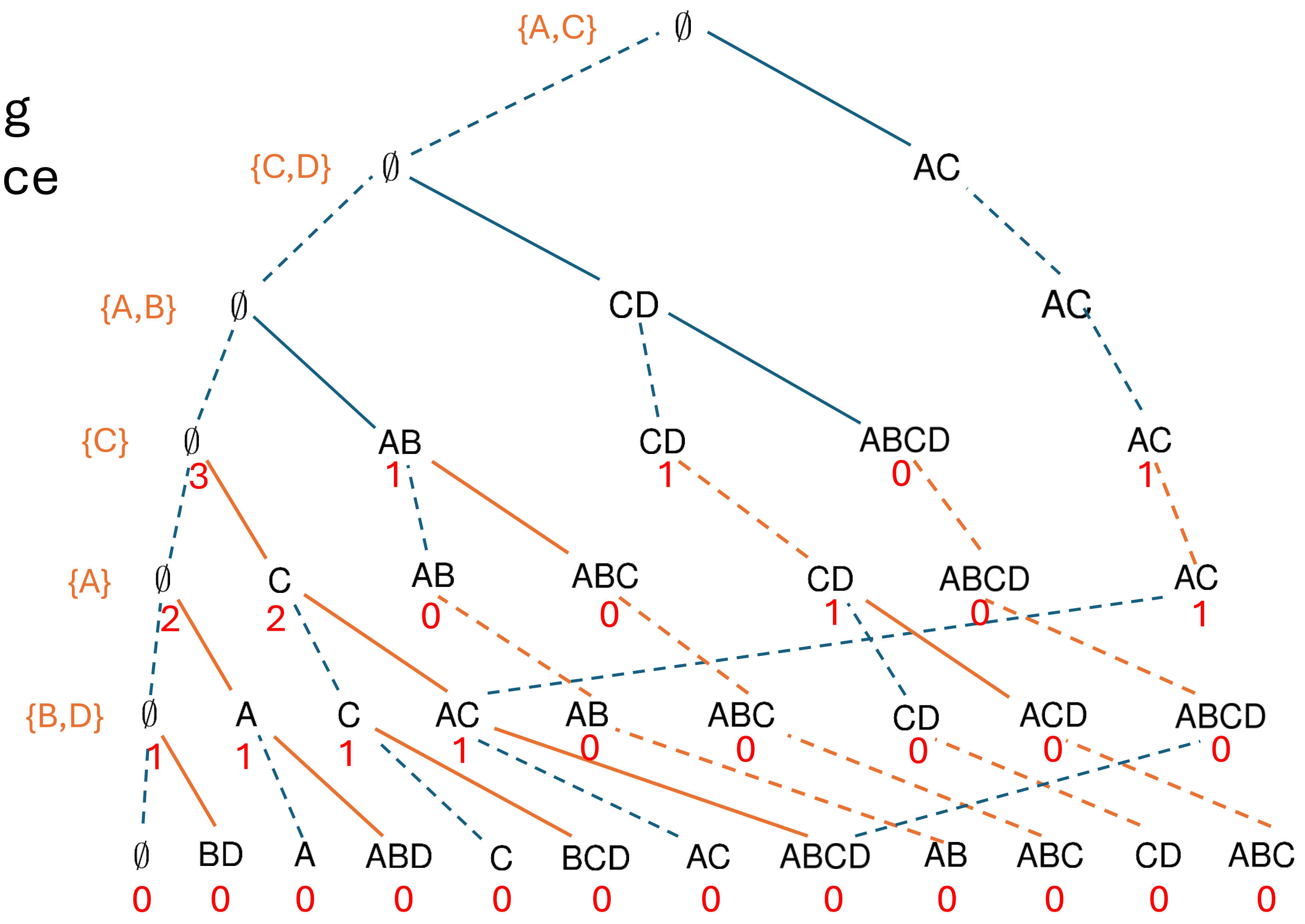
Carnegie Mellon University
Tepper School of Business

Serial DD
for a set packing problem instance



Carnegie Mellon University
Tepper School of Business

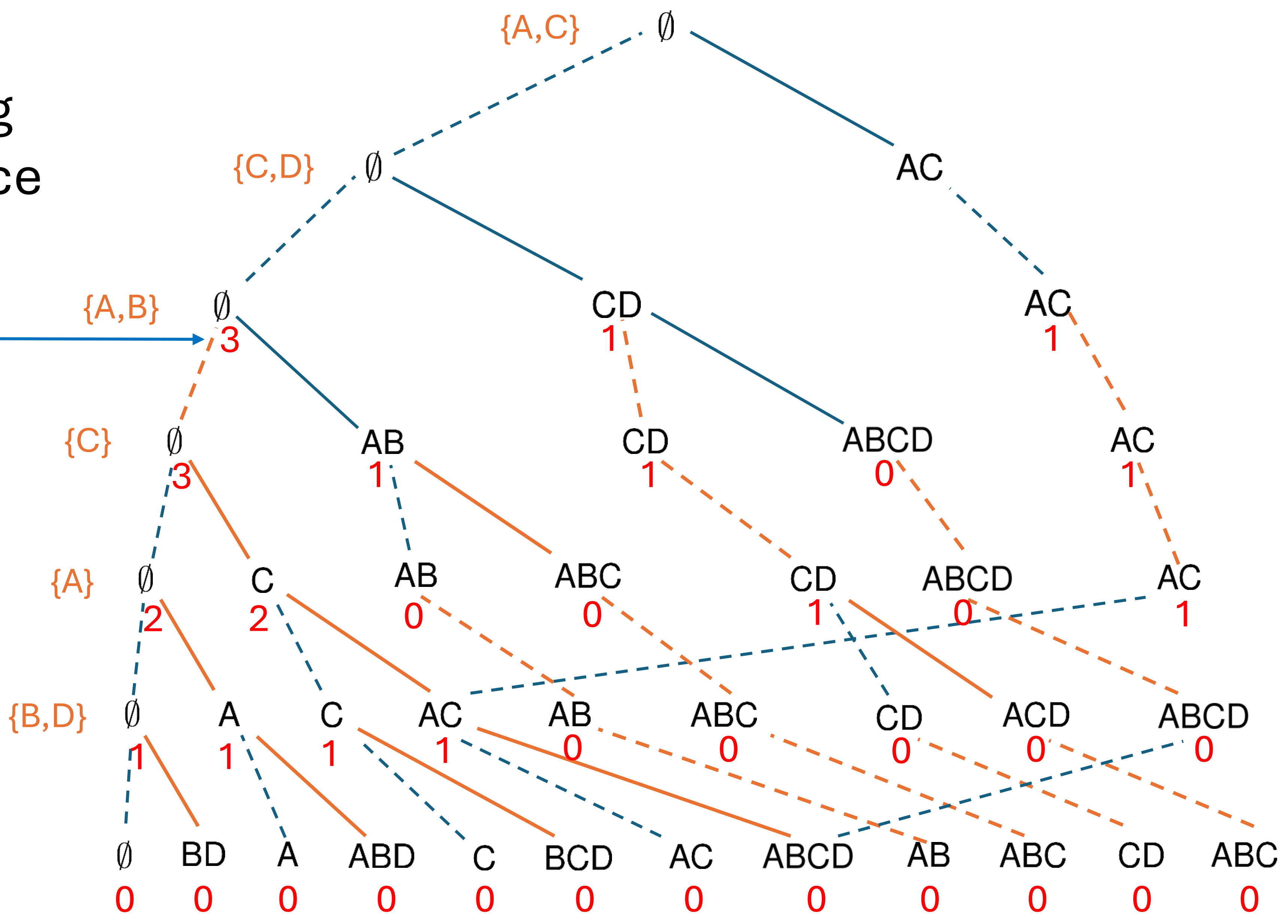
Serial DD
for a set packing problem instance



Serial DD

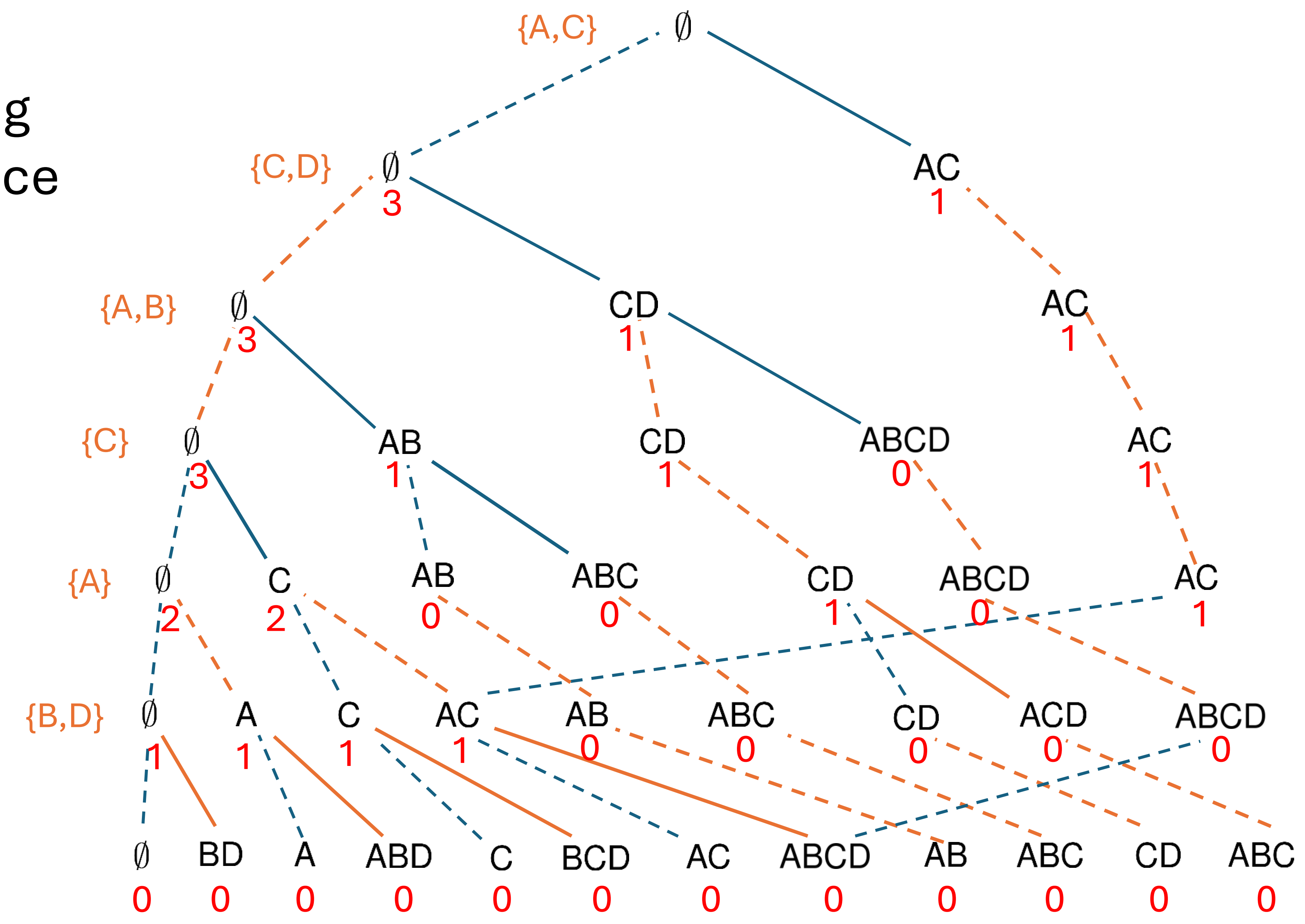
for a set packing problem instance

max {3, 1+1} = 3



Serial DD

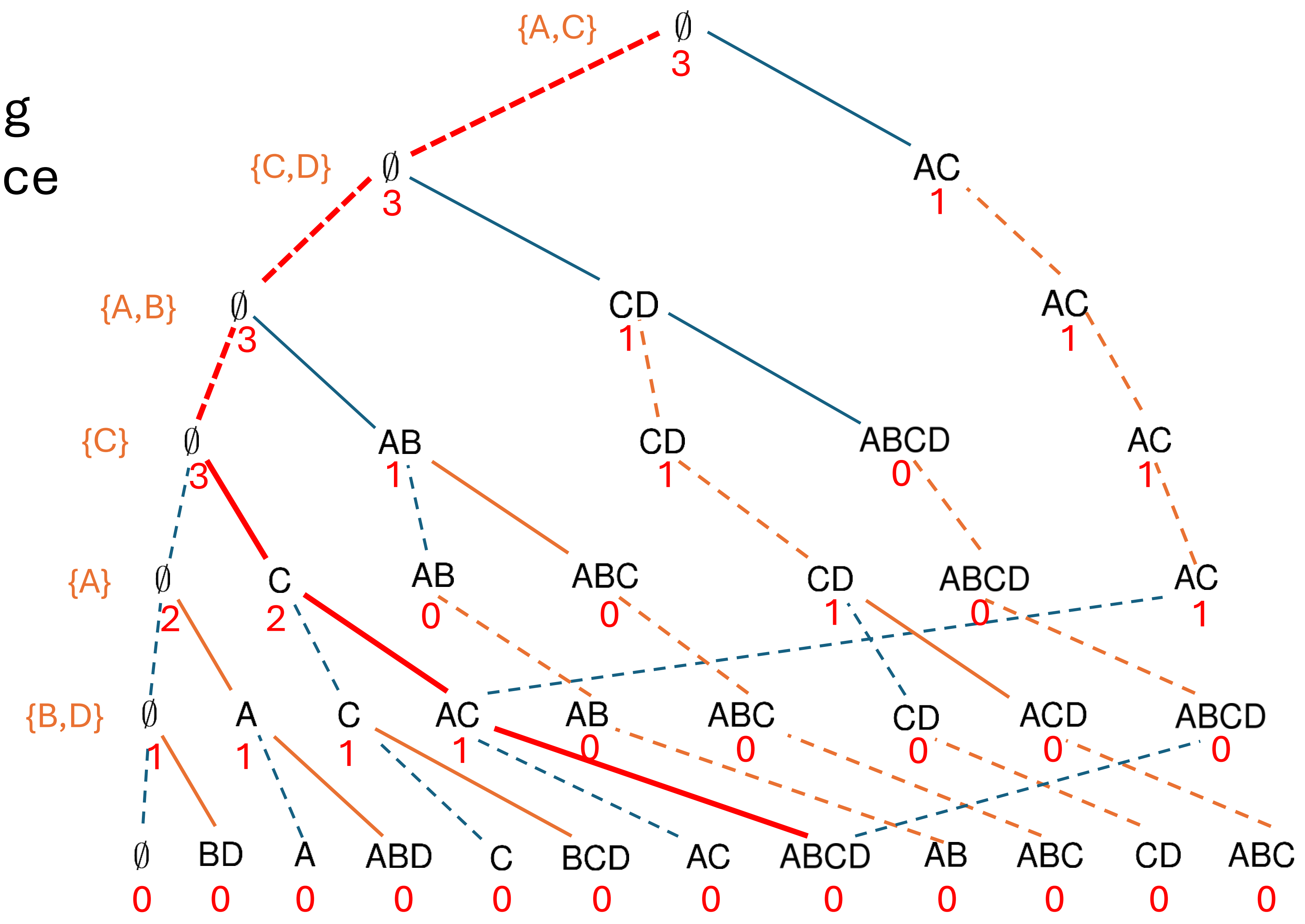
for a set packing problem instance



Carnegie Mellon University
Tepper School of Business

Serial DD for a set packing problem instance

Trace optimal choices top-down to find **optimal solution** (on longest path) {C}, {A}, {B,D}



Carnegie Mellon University
Tepper School of Business

Reduced DDs

A given Boolean function is represented by a **reduced DD** (minimize size DD) that is **unique** for a given variable ordering.

Bryant (1986)

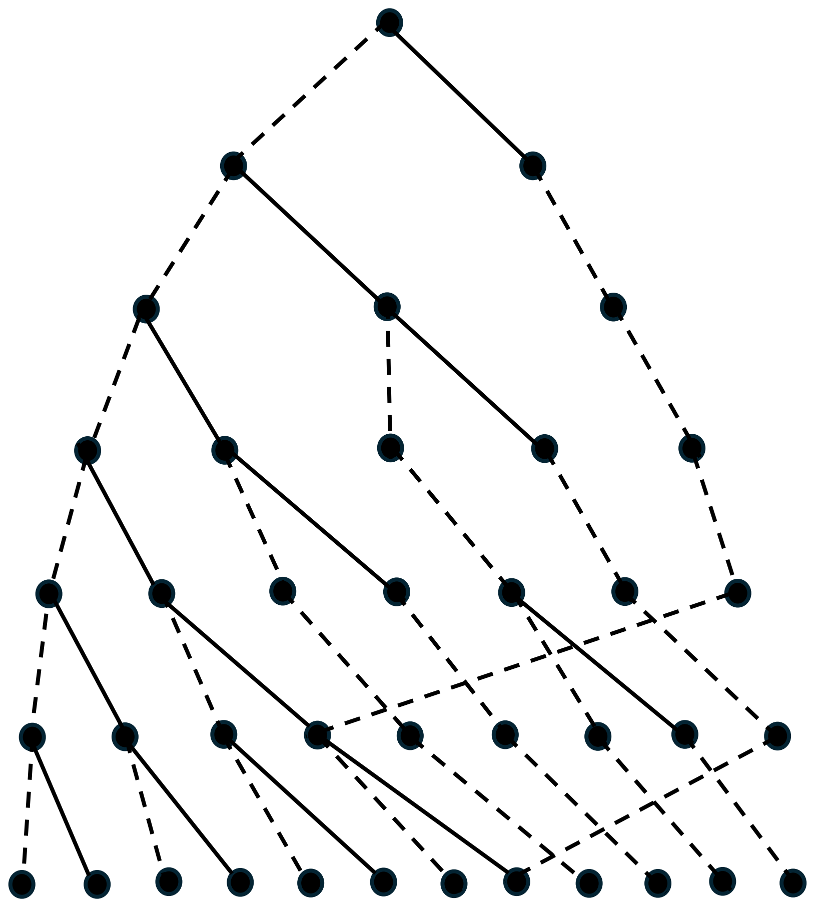
States become irrelevant after reduction.

Longest (shortest) path can be computed in the usual fashion.

Reduced DD
For set packing problem instance.

Begin with top-down compilation for set packing problem.

It can be reduced in bottom-up fashion.

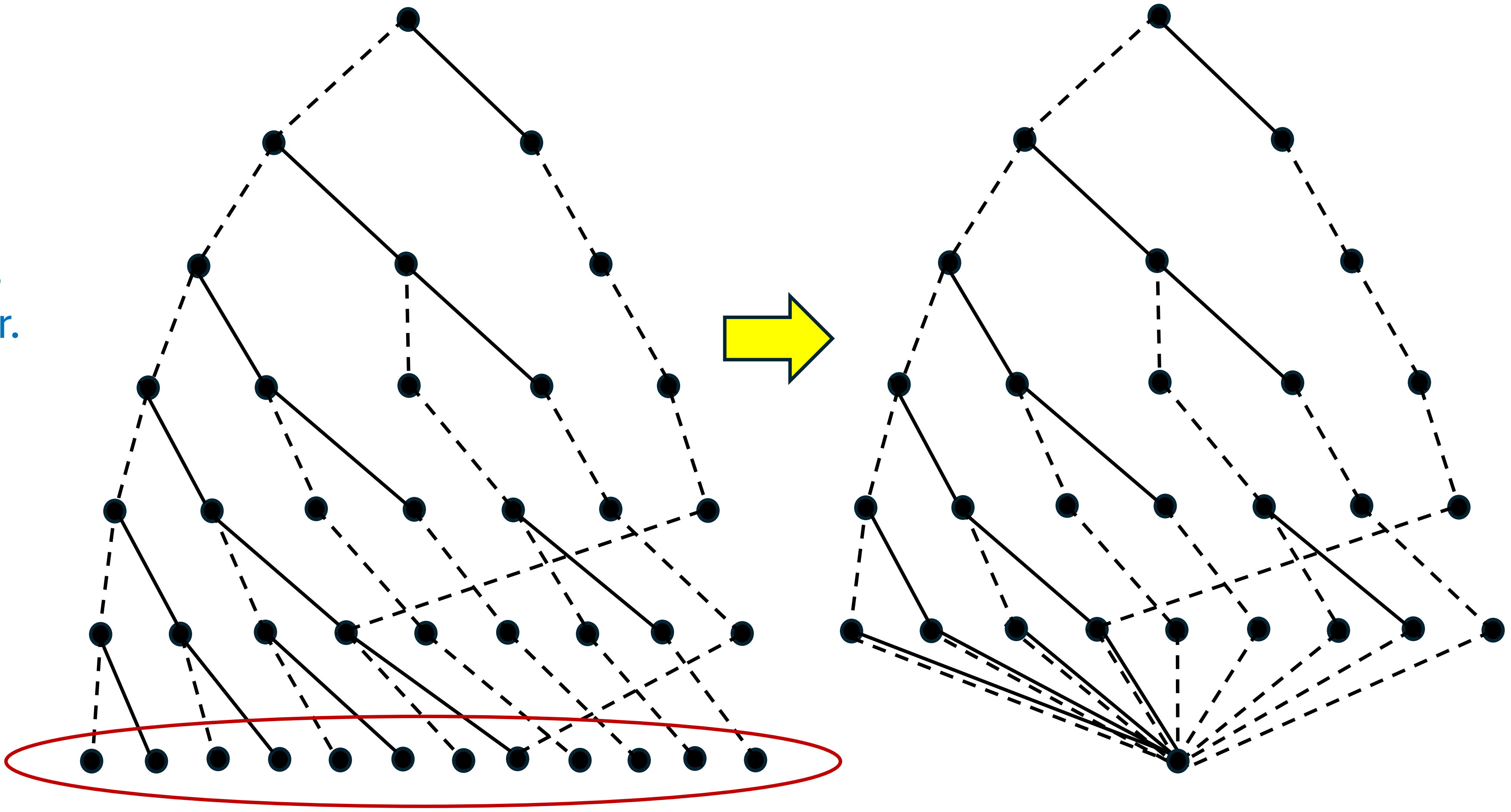


39 nodes

Carnegie Mellon University
Tepper School of Business

Reduced DD
For set packing problem instance.

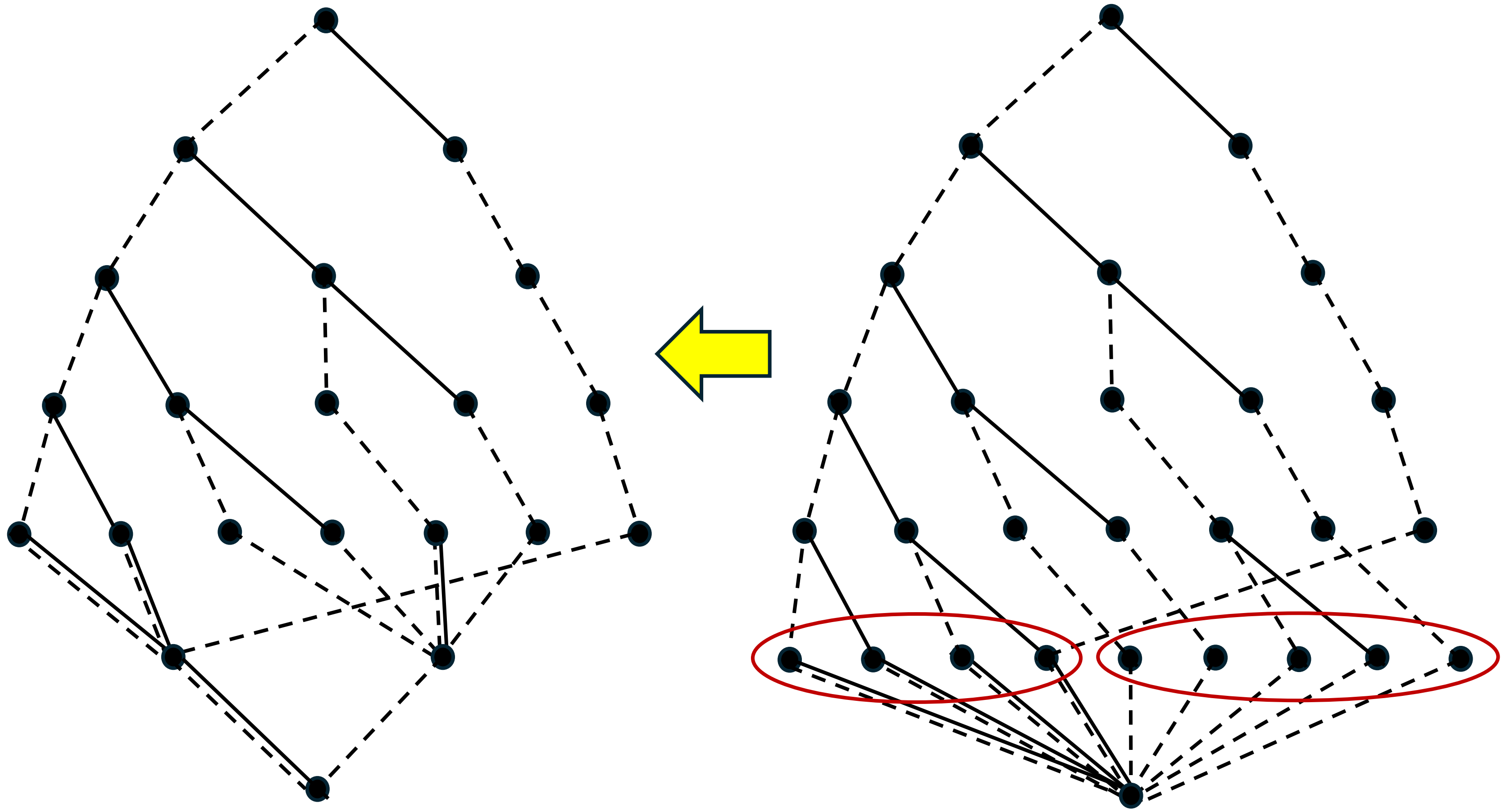
Superimpose nodes that are **roots of identical DDs**, beginning with bottom layer.



Carnegie Mellon University
Tepper School of Business

Reduced DD
For set packing problem instance.

Now, next layer.

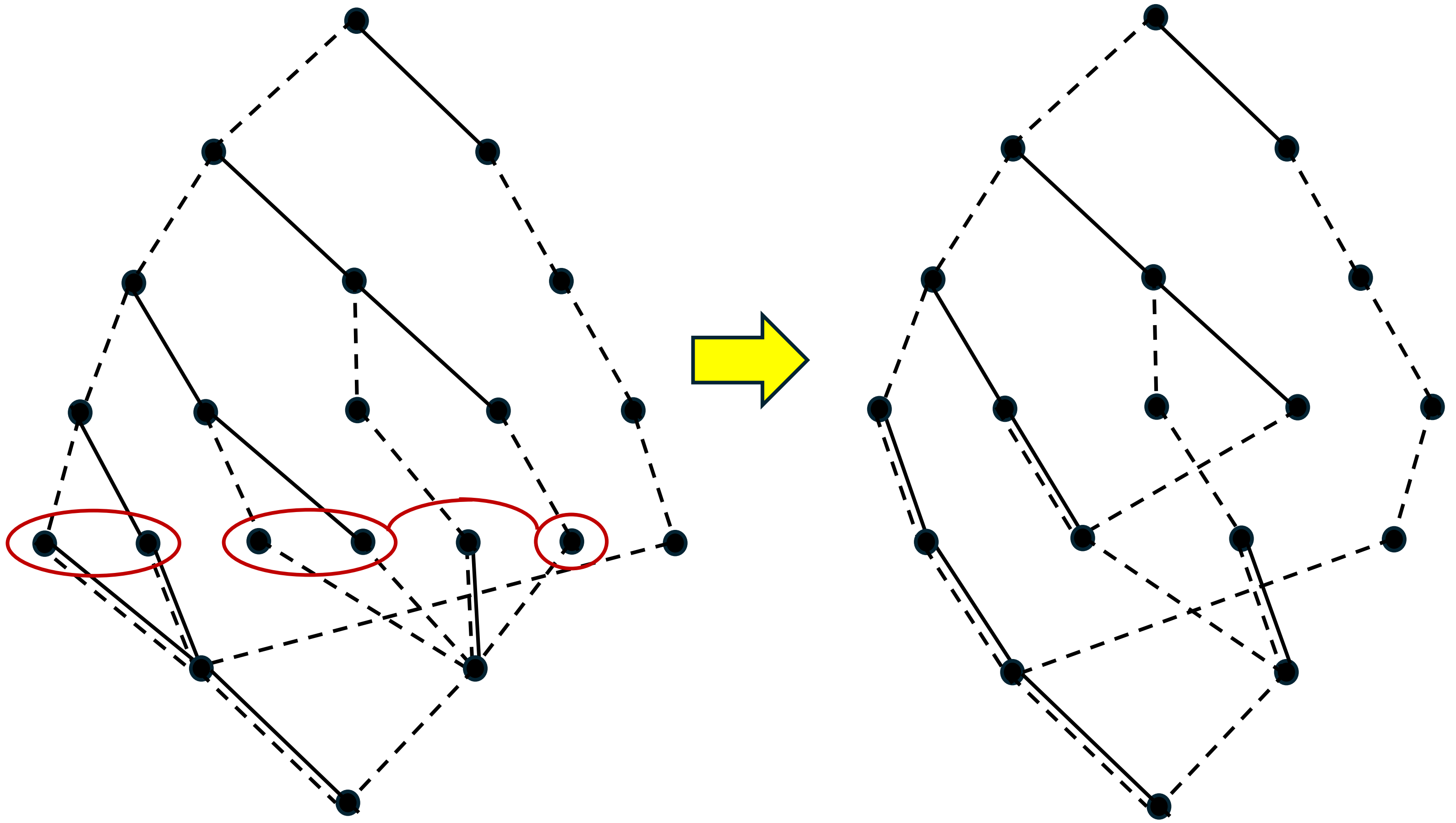


Reduced DD

For set packing problem instance.

Next layer

No more reduction possible.



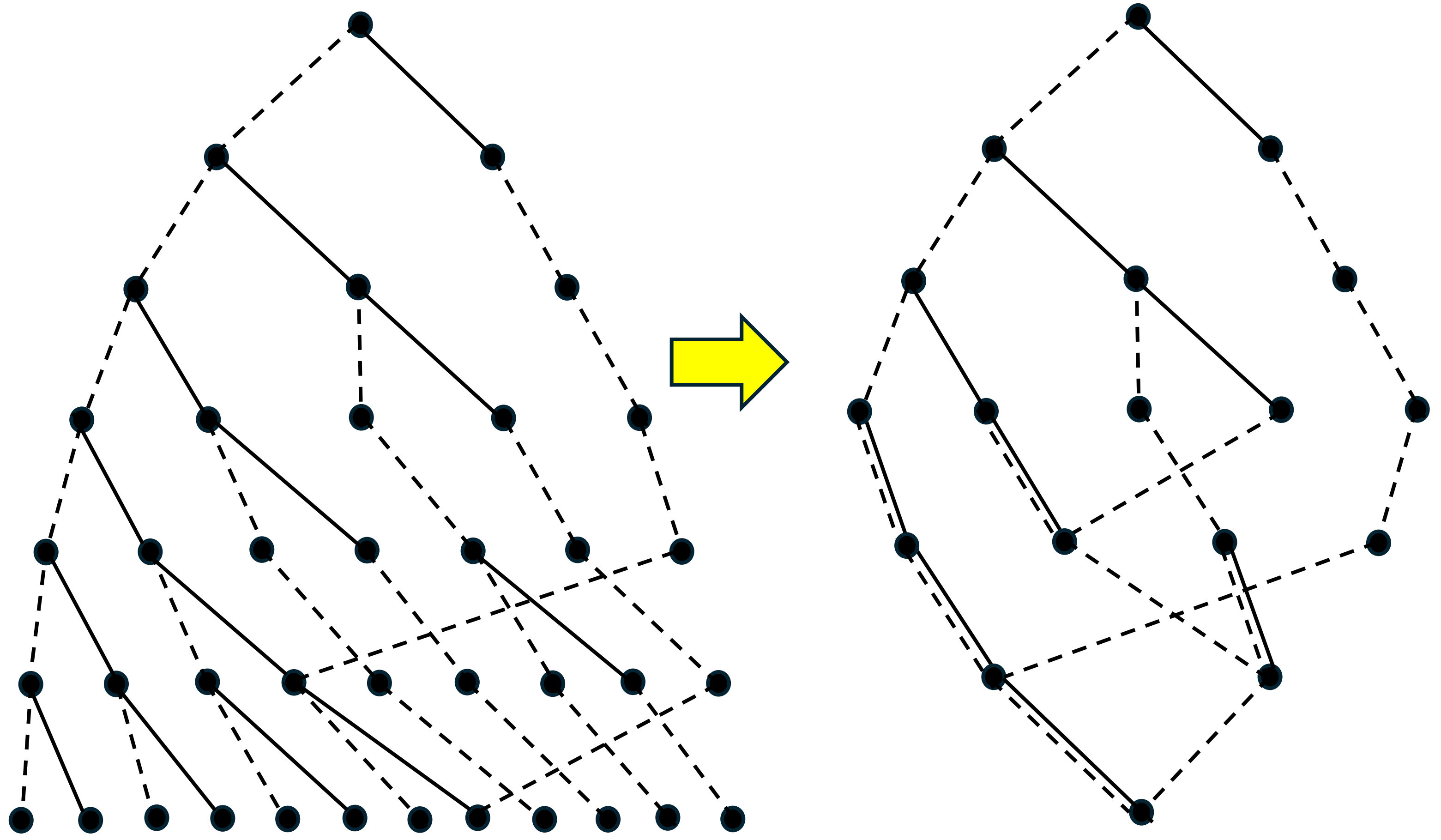
Carnegie Mellon University
Tepper School of Business

Reduced DD

For set packing problem instance.

Substantial size reduction.

Original and **reduced** serial DDs



39 nodes

18 nodes

Carnegie Mellon University
Tepper School of Business

Reduced weighted DDs

A **weighted** DD has **arc costs**, used to find min or min path length.

In previous example, all solid (and all dashed) arcs have the same cost.

Otherwise, one must **consider arc costs** during reduction.

There is a **unique reduced weighted DD,** which can again be found by a bottom-up procedure…

…provided the arc costs are **canonical** (easily achieved).

JH (2013)
Similar result for AADDs:
Sanner & McAllister (2005)

Carnegie Mellon University
Tepper School of Business

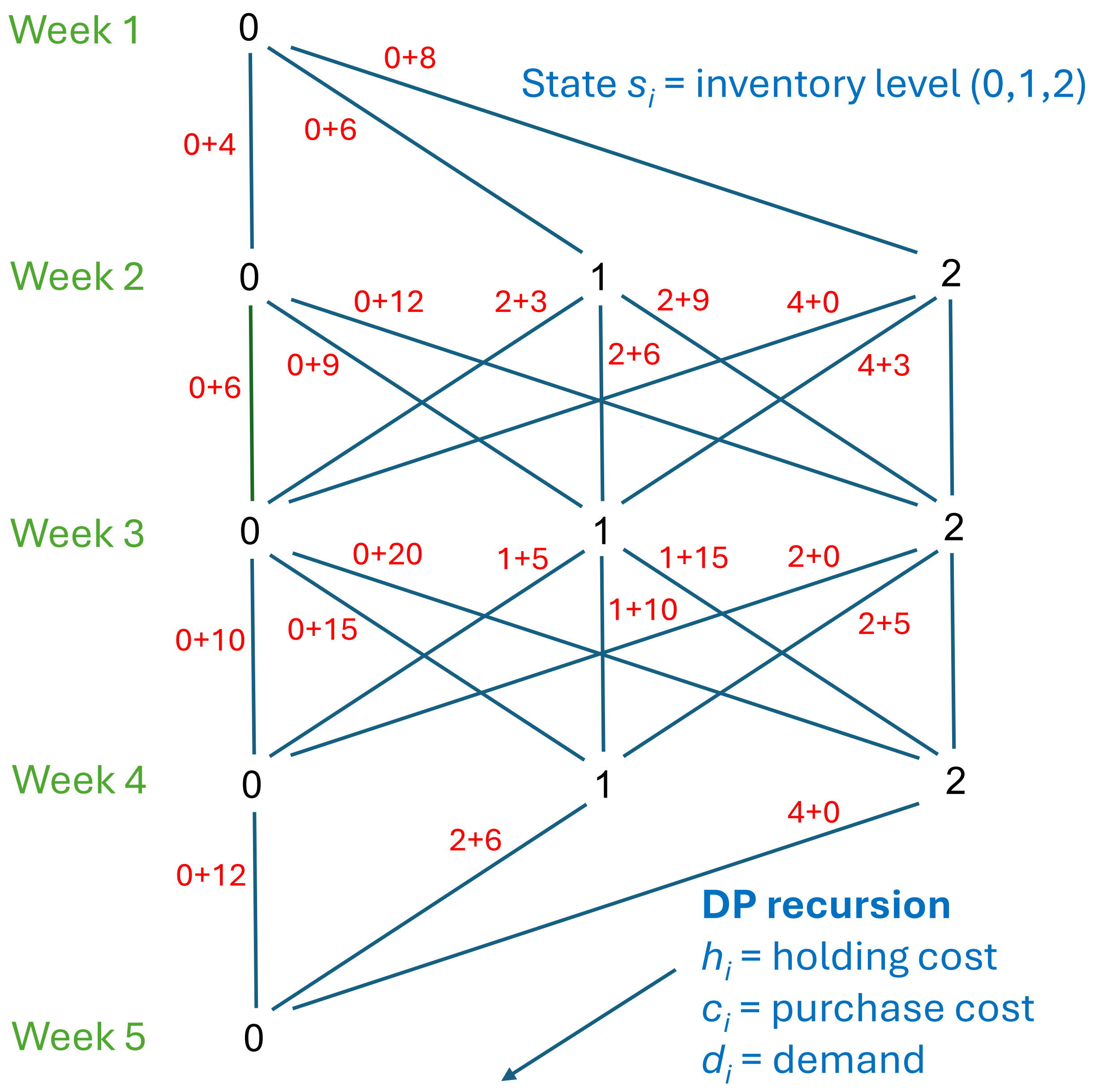
Generating a reduced DD

Reduction is usually **bottom-up** and requires that **entire DD** be available.

However, reductions can sometimes be identified **analytically** in advance.

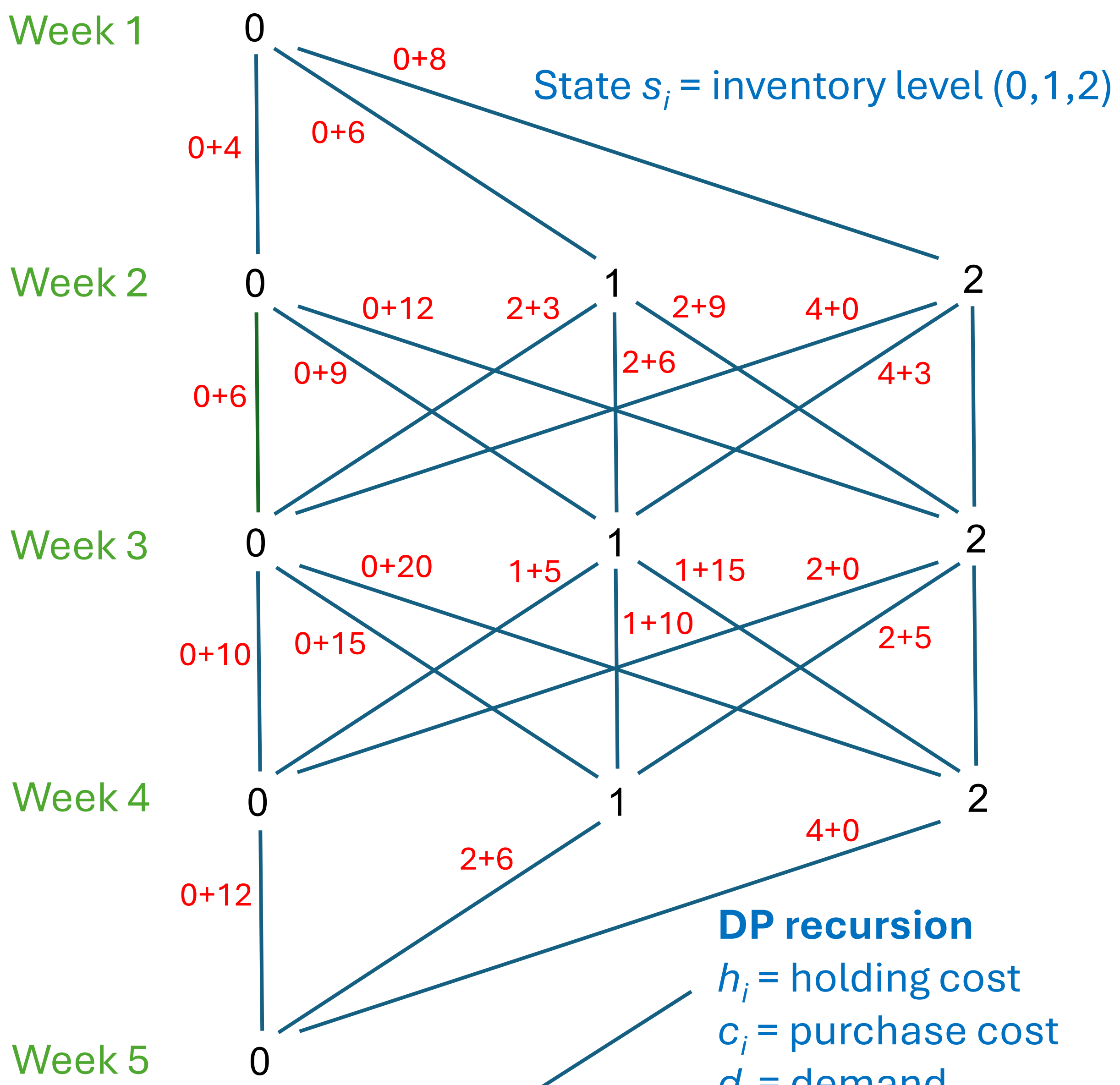
Example: a class of **inventory management problems**.

DP-based weighted DD



$$g_i(s_i) = \min_{x_i} \left\{ h_i s_i + c_i x_i + g_{i+1}(s_i + x_i - d_i) \right\}$$

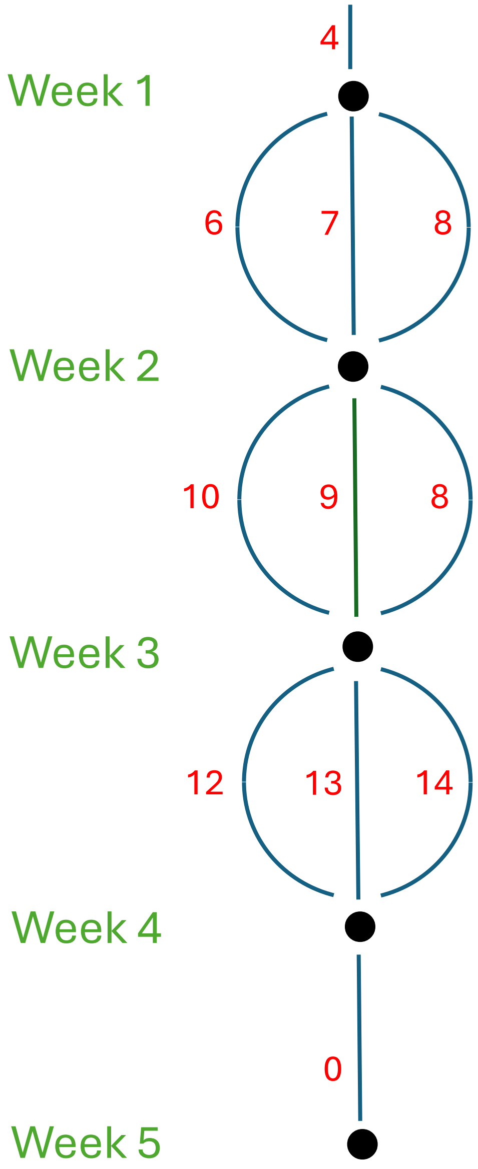
DP-based weighted DD



DP recursion
h_i = holding cost
c_i = purchase cost
d_i = demand

$$g_i(s_i) = \min_{x_i} \left\{ h_i s_i + c_i x_i + g_{i+1}(s_i + x_i - d_i) \right\}$$

Reduced weighted DD



Reformulated DP recursion results in **canonical** arc costs.

Reduced weighted DD is much smaller, computing shortest path is **trivial.**

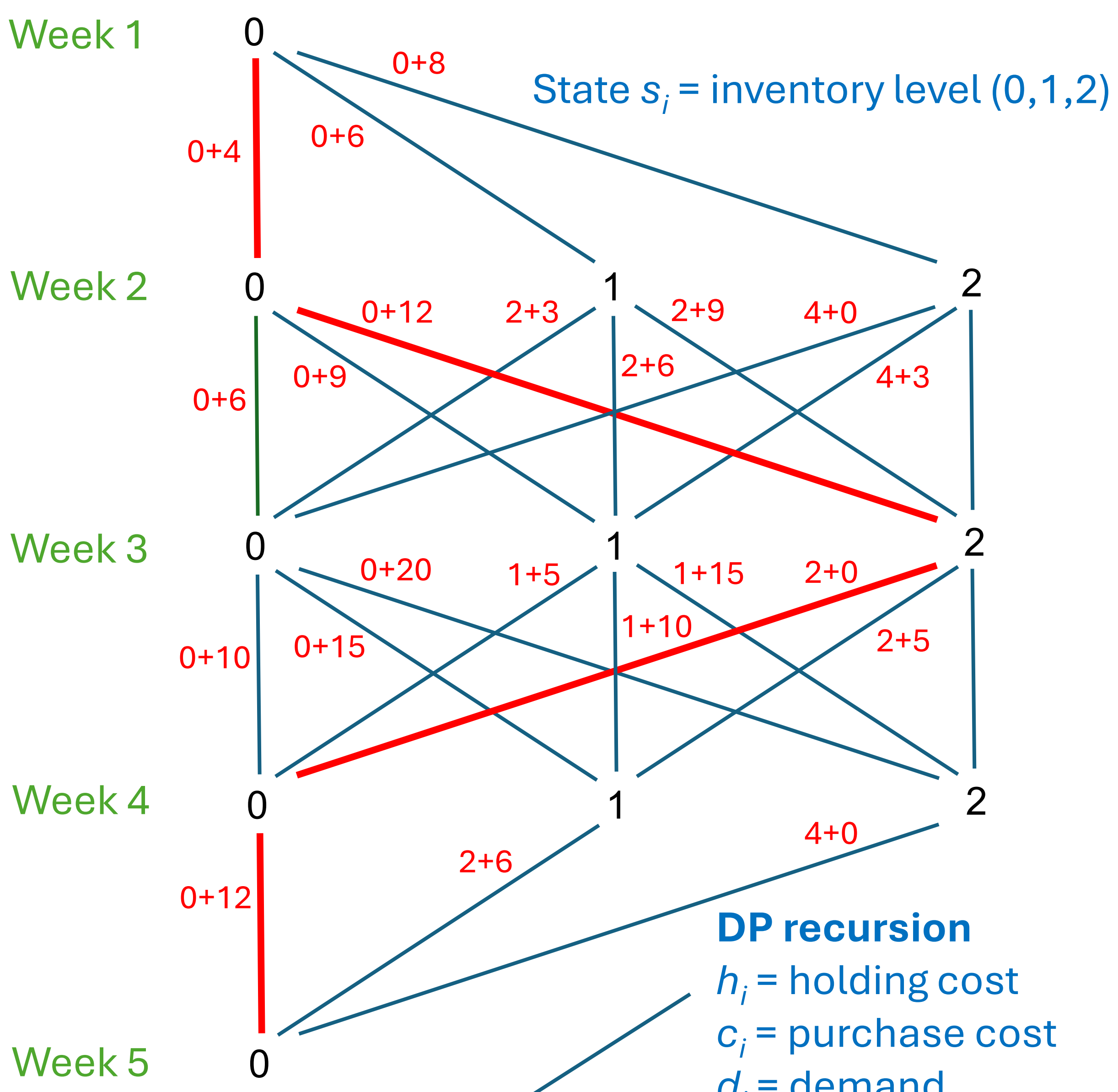
This simplification was apparently **never observed** over decades of research on inventory models.

JH (2013)

Reformulated DP recursion

$$g_i = \min_{x'_i} \left\{ h_{i+1} x'_i + c_i(x'_i - m + d_i) + c_{i+1}(m - x'_i) + g_{i+1} \right\}$$

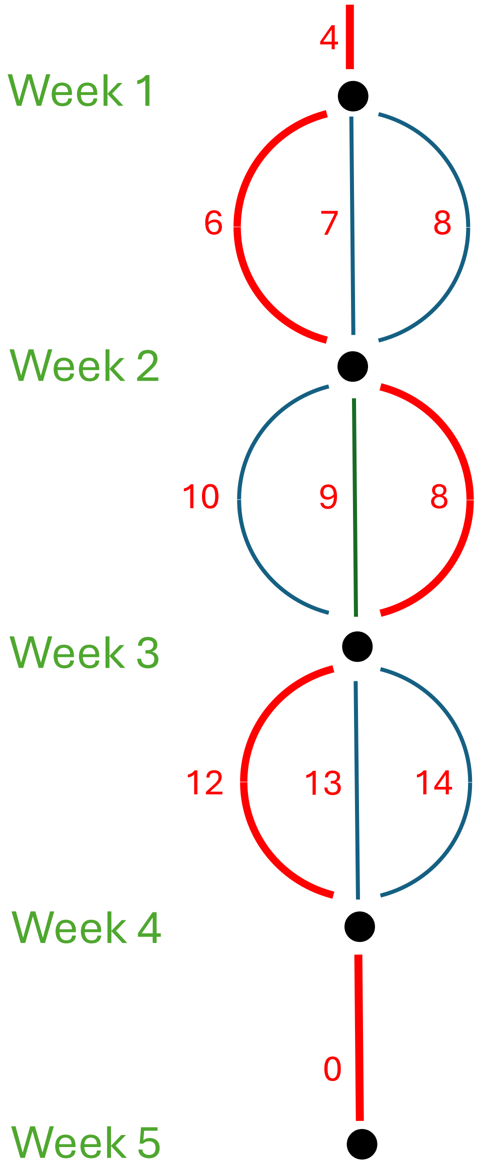
DP-based weighted DD



DP recursion
h_i = holding cost
c_i = purchase cost
d_i = demand

$$g_i(s_i) = \min_{x_i} \left\{ h_i s_i + c_i x_i + g_{i+1}(s_i + x_i - d_i) \right\}$$

Reduced weighted DD



Reformulated DP recursion results in **canonical** arc costs.

Reduced weighted DD is much smaller, computing shortest path is **trivial.**

This simplification was apparently **never observed** over decades of research on inventory models.

JH (2013)

Reformulated DP recursion

$$g_i = \min_{x_i'} \left\{ h_{i+1} x_i' + c_i(x_i' - m + d_i) + c_{i+1}(m - x_i') + g_{i+1} \right\}$$

DD vs state transition graph in DP

How does a **DD** differ from a **dynamic programming state transition graph**?

A state transition graph can be viewed as a DD, but:

- DD nodes need not be associated with **states**.
- The **reduced DD** can be much **smaller** than the state transition graph.
- Much smaller **relaxed DDs** provide **bounds***
- Much smaller **restricted DDs** provide a **primal heuristic**.

*DD-based relaxation ≠ "state space relaxation" in DP

Relaxed DDs

Even **reduced** DDs tend to **grow exponentially** for most problems.

However, **relaxed DDs** of limited width can be obtained by allowing some infeasible paths.

Relaxed DDs

Even **reduced** DDs tend to **grow exponentially** for most problems.

However, **relaxed DDs** of limited width can be obtained by allowing some infeasible paths.

Two **top-down** compilation methods generate relaxed DDs:

- **Node merger** reduces each layer by heuristically merging nodes and their associated states.
- **Node splitting** heuristically adds nodes on each layer to rule out some infeasible solutions.

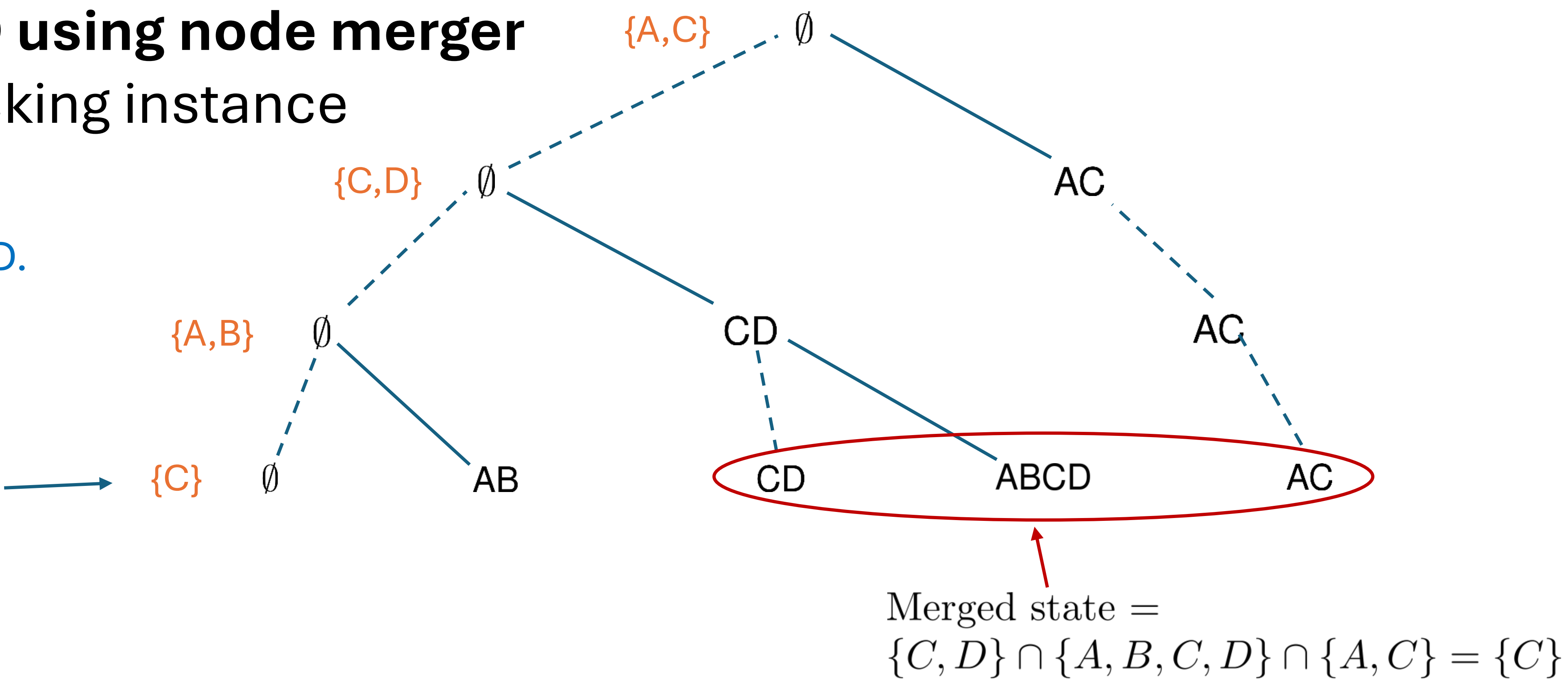
Hadžić & JH (2006)
Andersen, Hadžić, JH, Tiedemann (2007)
Hadžić, JH, O'Sullivan, Tiedemann (2008)
Bergman, van Hoeve, JH (2011)

Relaxed DD using node merger
for a set packing instance

Start building DD.
We want a
max width of 3.

Merge selected
states to keep
width ≤ 3.

Here, resulting
state is
intersection of
merged states.

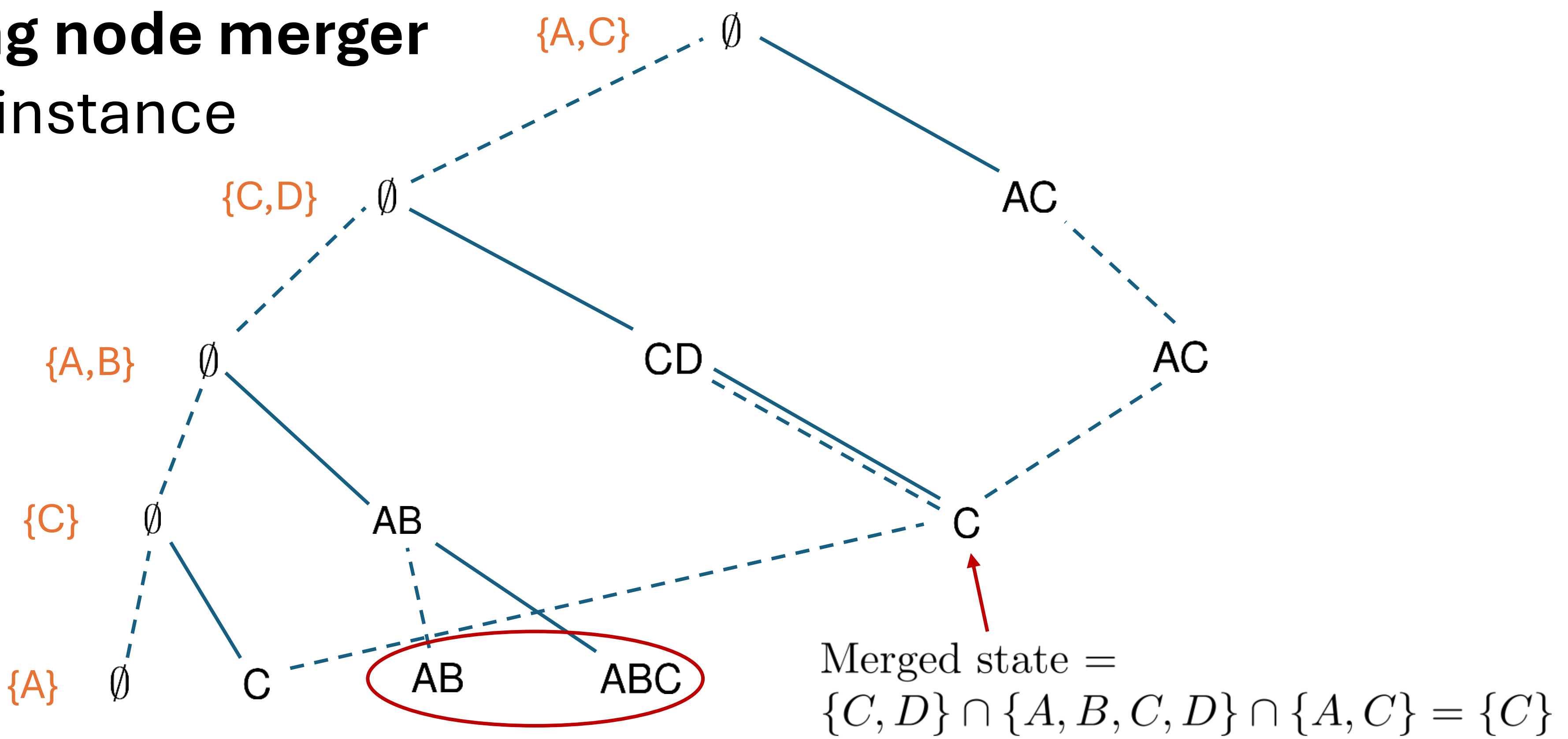


Carnegie Mellon University
Tepper School of Business

Relaxed DD using node merger
for a set packing instance

Continue building relaxed DD from reduced layer, using relaxed states.

Choice of nodes to merge is heuristic.

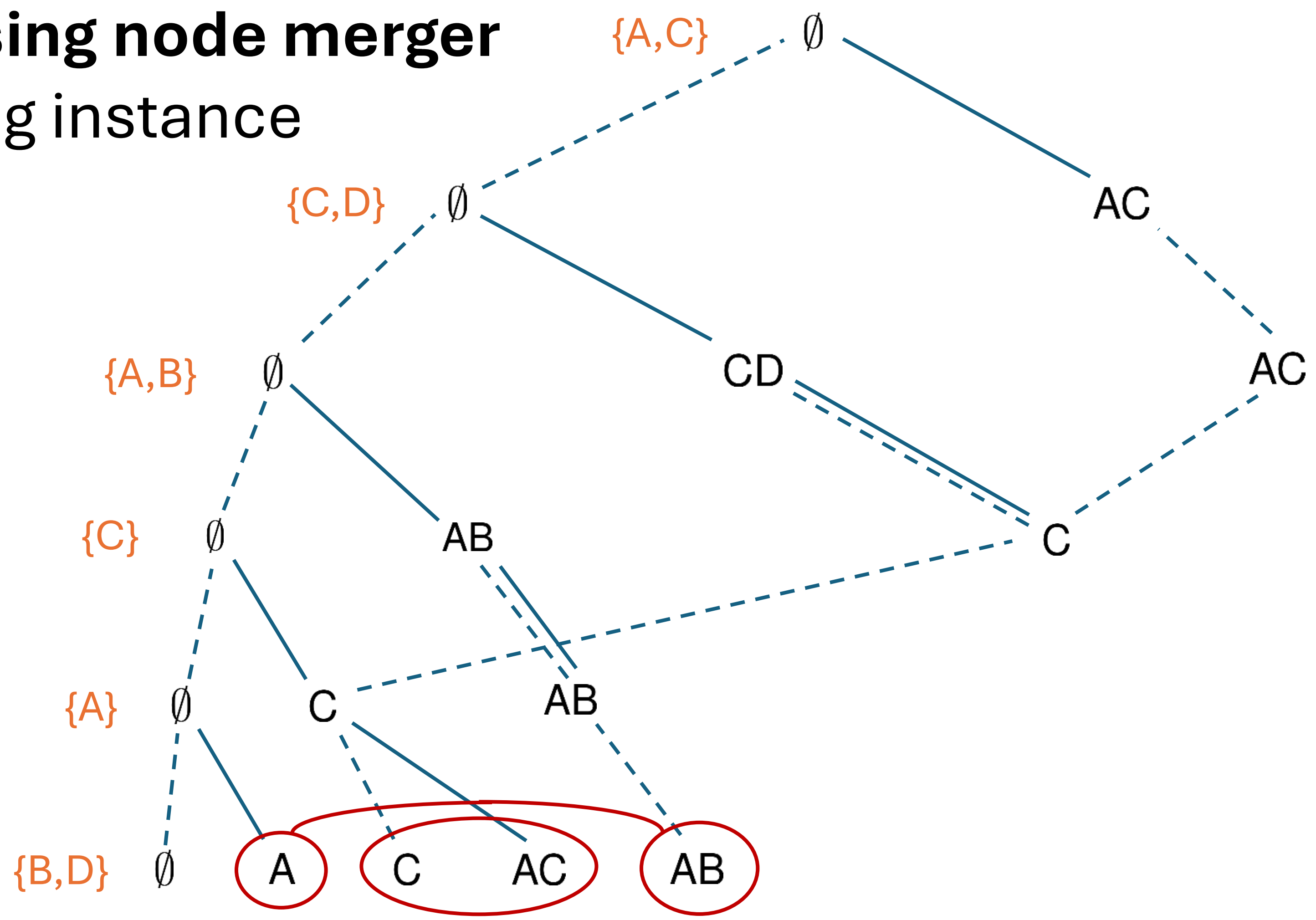


Carnegie Mellon University
Tepper School of Business

Relaxed DD using node merger
for a set packing instance

Continue building relaxed DD from reduced layer, using relaxed states.

Choice of nodes to merge is heuristic.

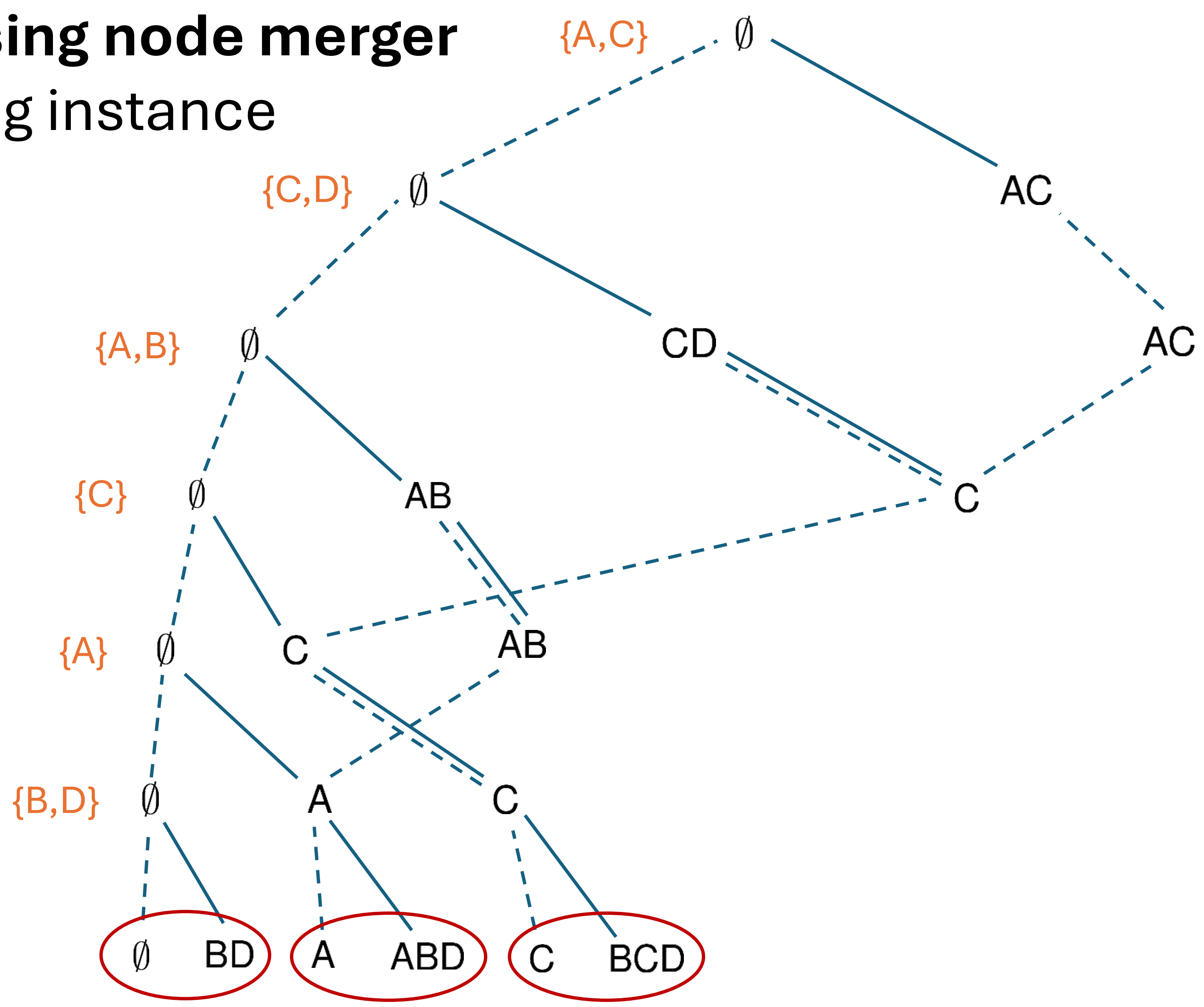


Carnegie Mellon University
Tepper School of Business

Relaxed DD using node merger
for a set packing instance

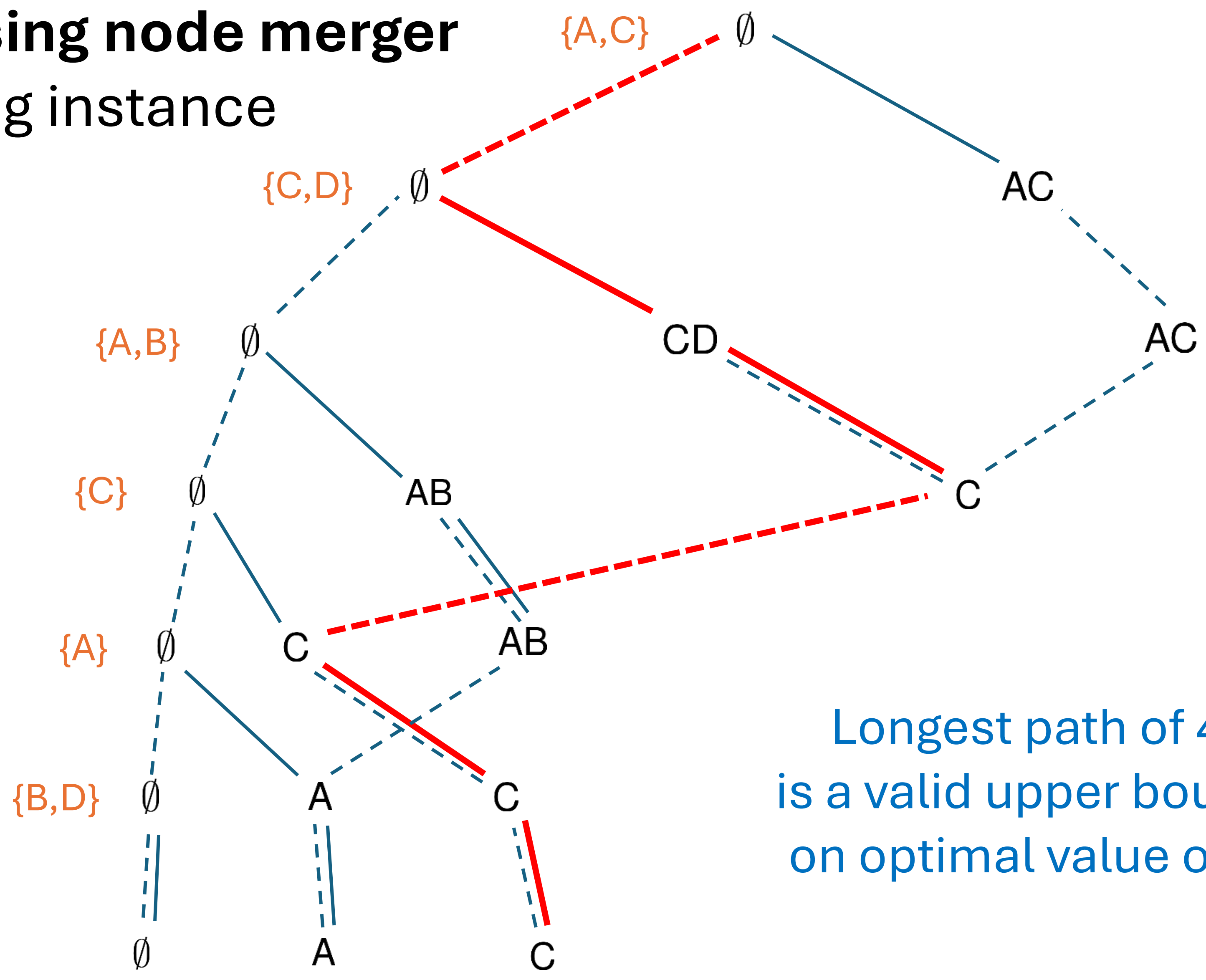
Continue building relaxed DD from reduced layer, using relaxed states.

Choice of nodes to merge is heuristic.



Carnegie Mellon University
Tepper School of Business

Relaxed DD using node merger
for a set packing instance



Longest path of 4
is a valid upper bound
on optimal value of 3

Carnegie Mellon University
Tepper School of Business

Relaxed DDs
Conditions for node merger

A state S' **relaxes** state S if and only if:

- Every control that is feasible in S is feasible in S'.
- The arc cost resulting from any feasible control in S is at least the cost of that control in S' (when minimizing).

A state merger operation generates a **valid relaxed DD** if

- The merger of two states is a relaxation of the merged states.
- State transition preserves relaxation. That is, If S' relaxes S, then $\phi(S')$ relaxes $\phi(S)$, for any given state transition ϕ.

JH (2017)

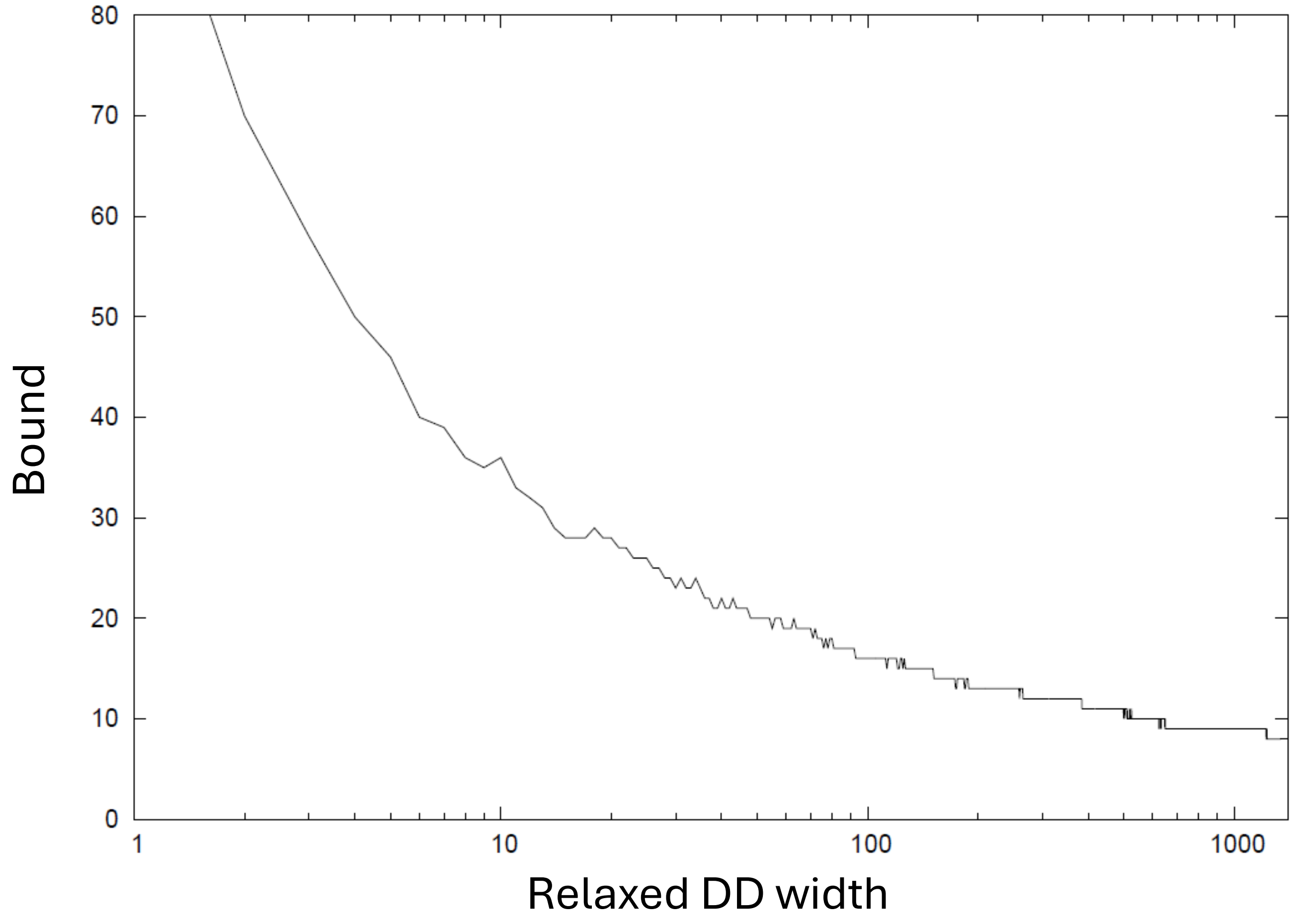
Carnegie Mellon University
Tepper School of Business

Relaxed DDs
Adjustable bound quality

Bound quality vs. relaxed DD width for **max stable set problem**.

Greater bound quality can be obtained by investing more time to generate a larger relaxed DD

Bergman, Ciré, van Hoeve, JH (2013)



Relaxed DDs
Experimental results for node merger

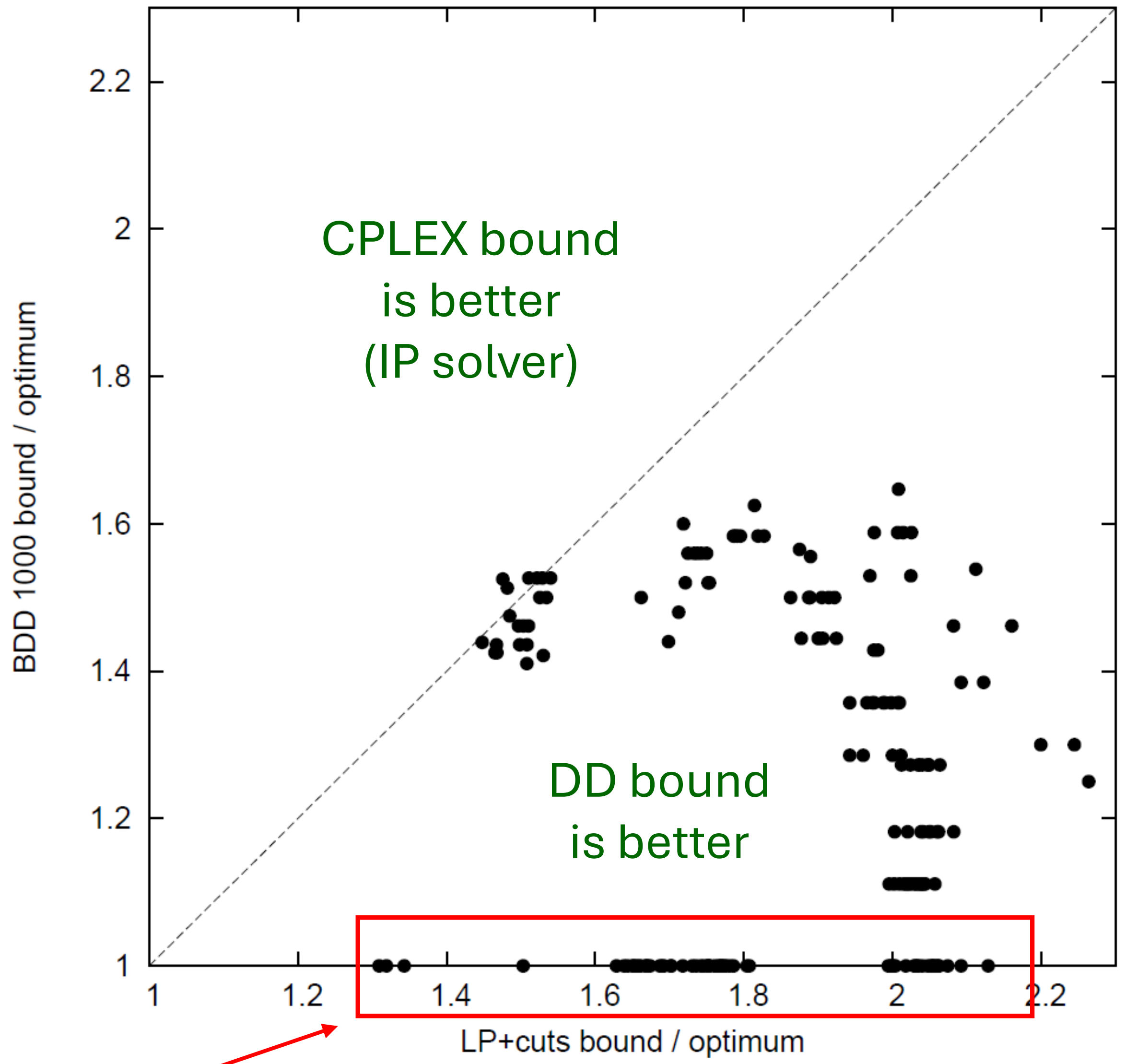
Bound quality, DDs vs IP for **max stable set problem**.

Relaxed DD width = 1000.

CPLEX bound based on 50 years of cutting plane research.

DDs require about **5% the computation time** of CPLEX.

Bergman, Ciré, van Hoeve, JH (2013)



Optimal value obtained

Carnegie Mellon University
Tepper School of Business

Restricted DDs

A **restricted DD** represents a **proper subset** of feasible solutions.

It can be compiled top-down by heuristically deleting nodes
as necessary to limit the DD width.

Bergman, Ciré,
van Hoeve, JH (2016)

Restricted DDs

A **restricted DD** represents a **proper subset** of feasible solutions.

It can be compiled top-down by heuristically deleting nodes
as necessary to limit the DD width.

Finding a shortest (longest) path in a restricted DD provides a
primal heuristic for generating good feasible solutions.

Primal heuristics are responsible for much of the remarkable
speedup of **IP solvers**.

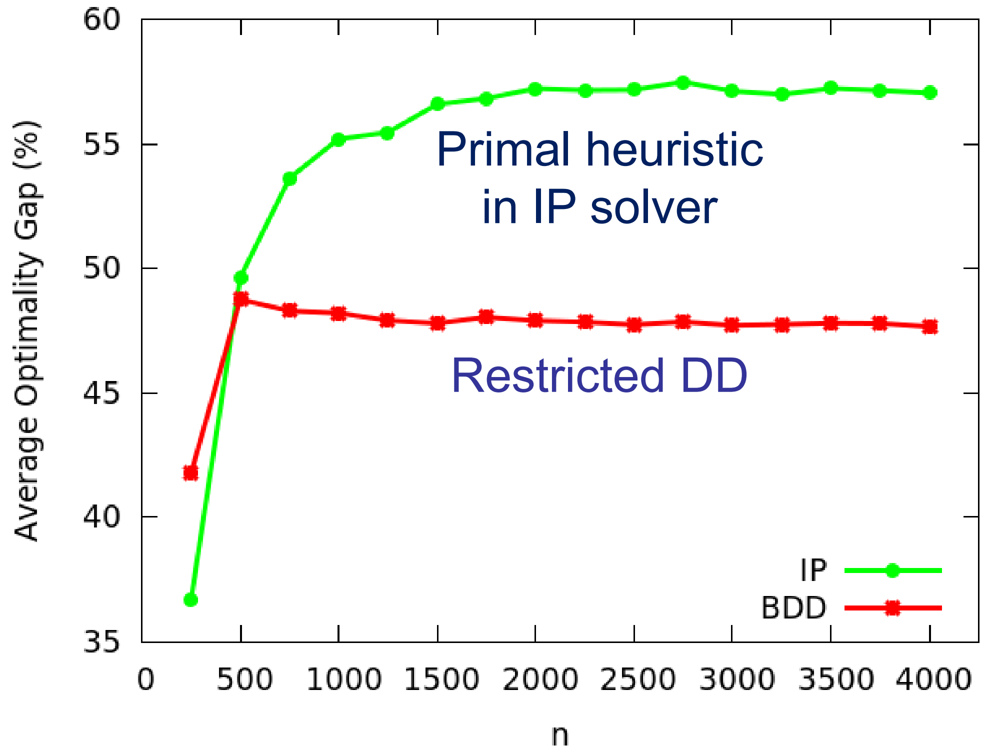
A restricted DD can be **superior** to state-of-the art primal heuristics.

Bergman, Ciré,
van Hoeve, JH (2016)

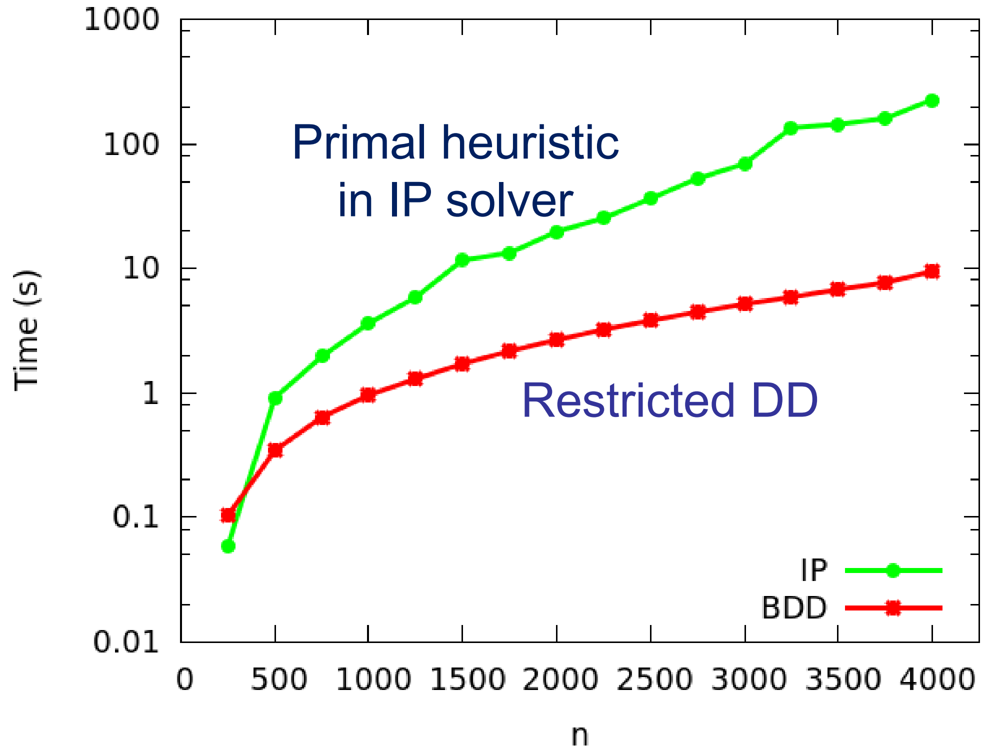
Restricted DDs
Experimental results

Primal heuristics for **set covering**

Quality of solution (smaller gap is better)



Time to generate solutions



Carnegie Mellon University
Tepper School of Business

Bergman, Ciré, van Hoeve, JH (2016)

DD-based Branch and Bound

Branch-and-Bound methods of **integer programming** prune a branching tree, using bounds on the optimal value from a **linear programming relaxation**.

DD-based Branch and Bound replaces the LP relaxation with a **relaxed DD**.

It **branches within a relaxed DD**, which eliminates many unnecessary branches.

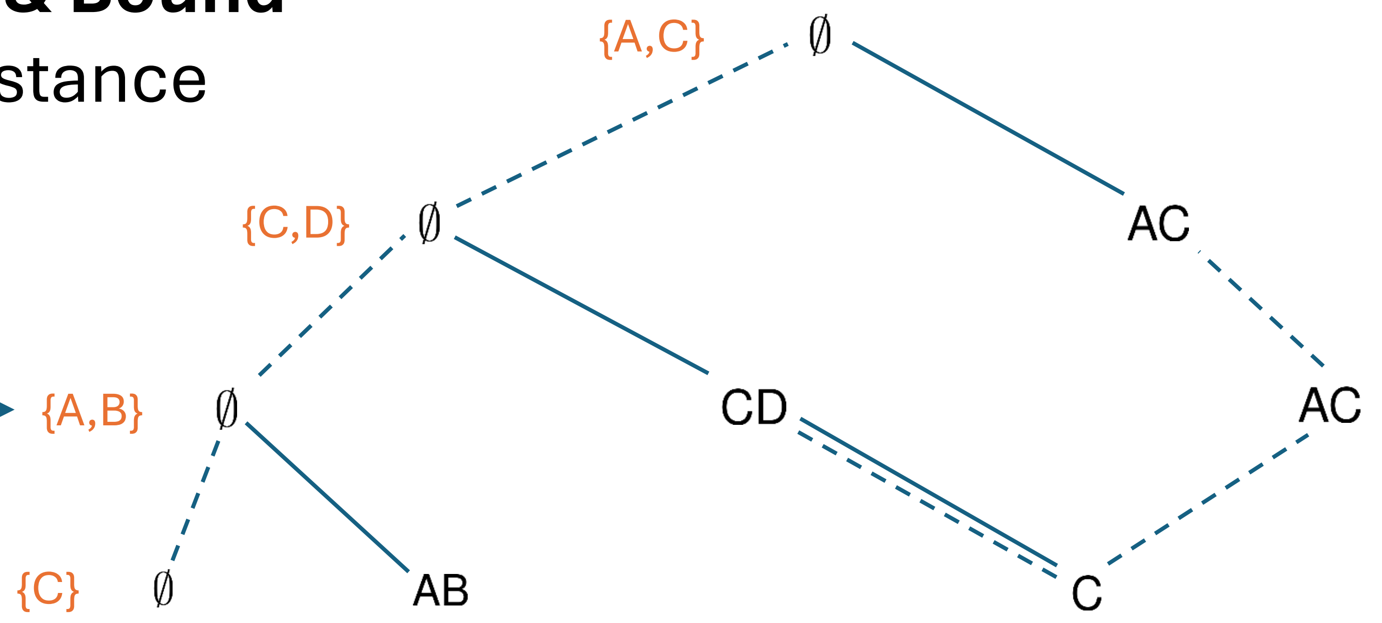
Bergman, Ciré,
van Hoeve, JH (2016)

DD-based Branch & Bound

for a set packing instance

Start building a relaxed DD.

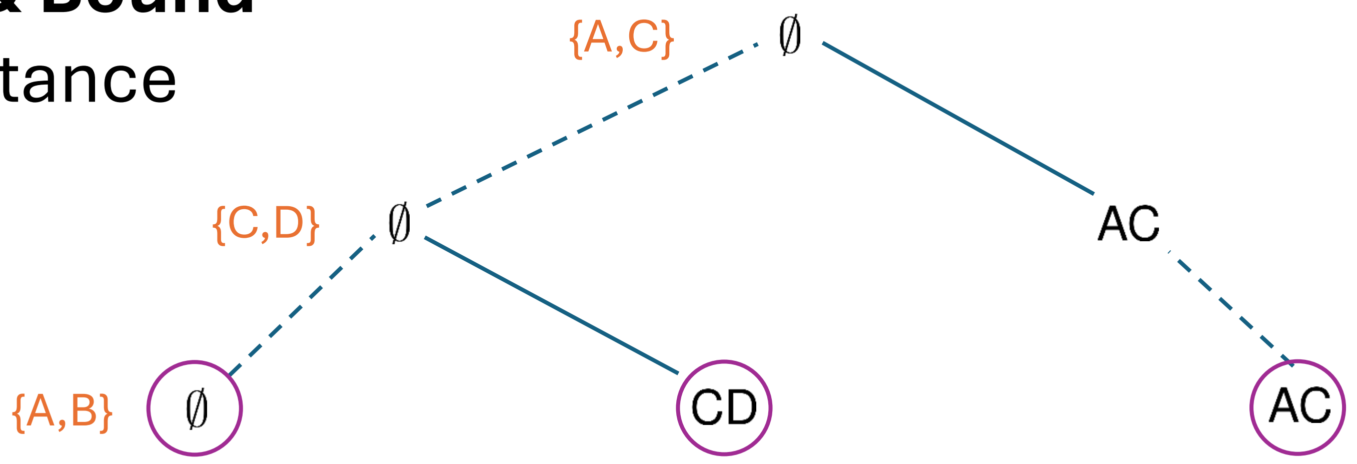
This is the **last exact layer** (no node mergers) →



DD-based Branch & Bound

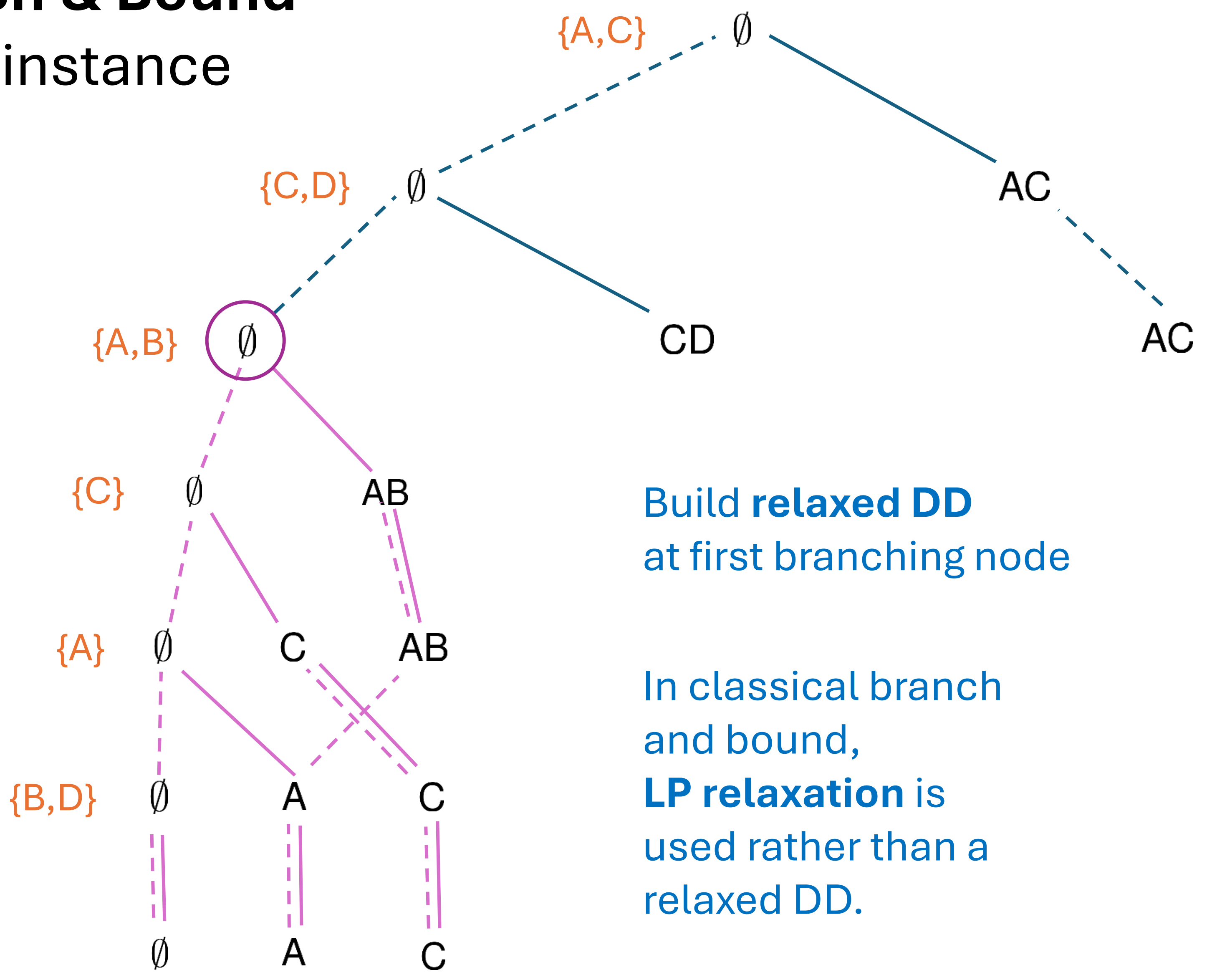
for a set packing instance

So **branch** on this layer



DD-based Branch & Bound

for a set packing instance



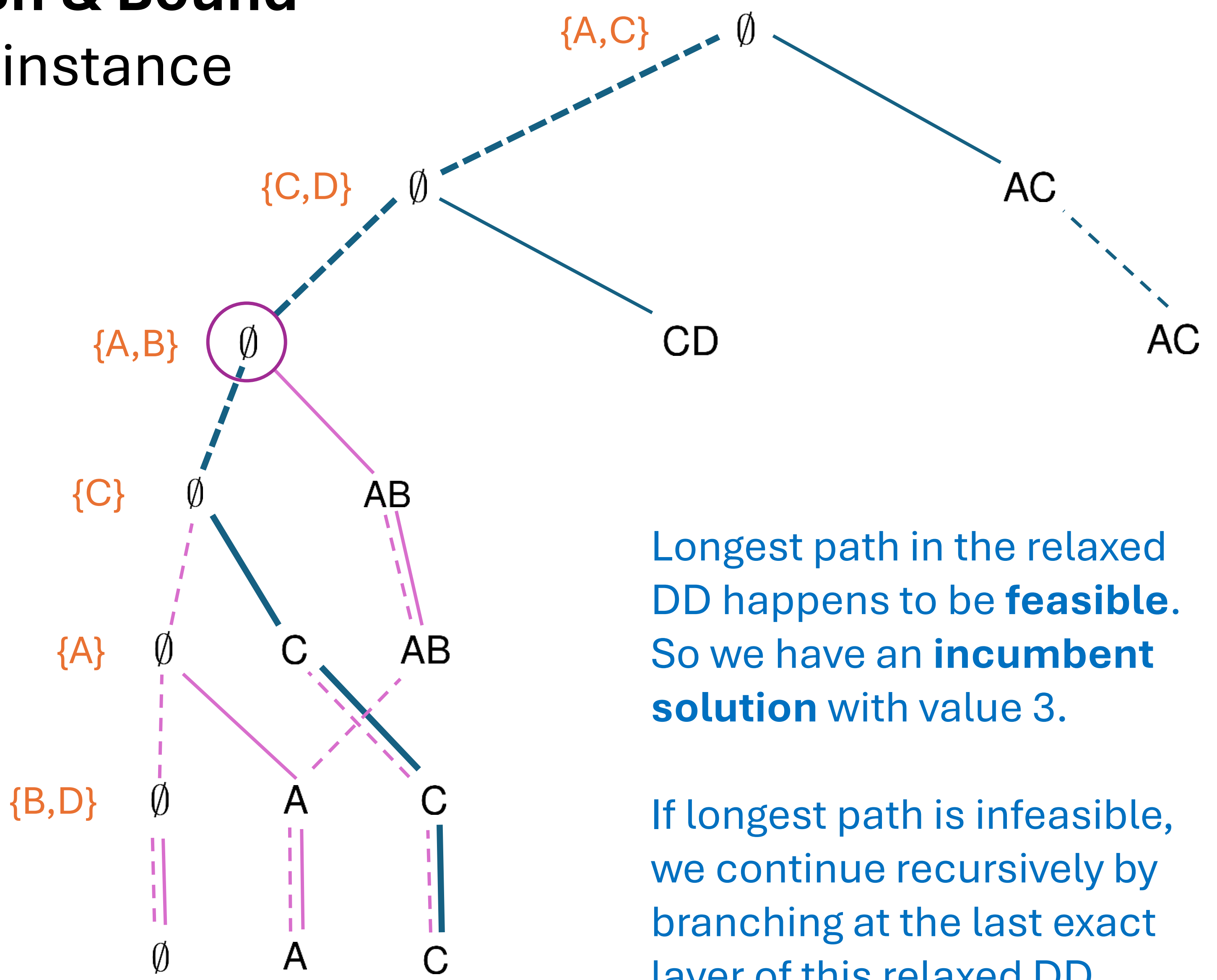
Build **relaxed DD** at first branching node

In classical branch and bound, **LP relaxation** is used rather than a relaxed DD.

Carnegie Mellon University
Tepper School of Business

DD-based Branch & Bound

for a set packing instance



Upper bound = 3
Incumbent solution

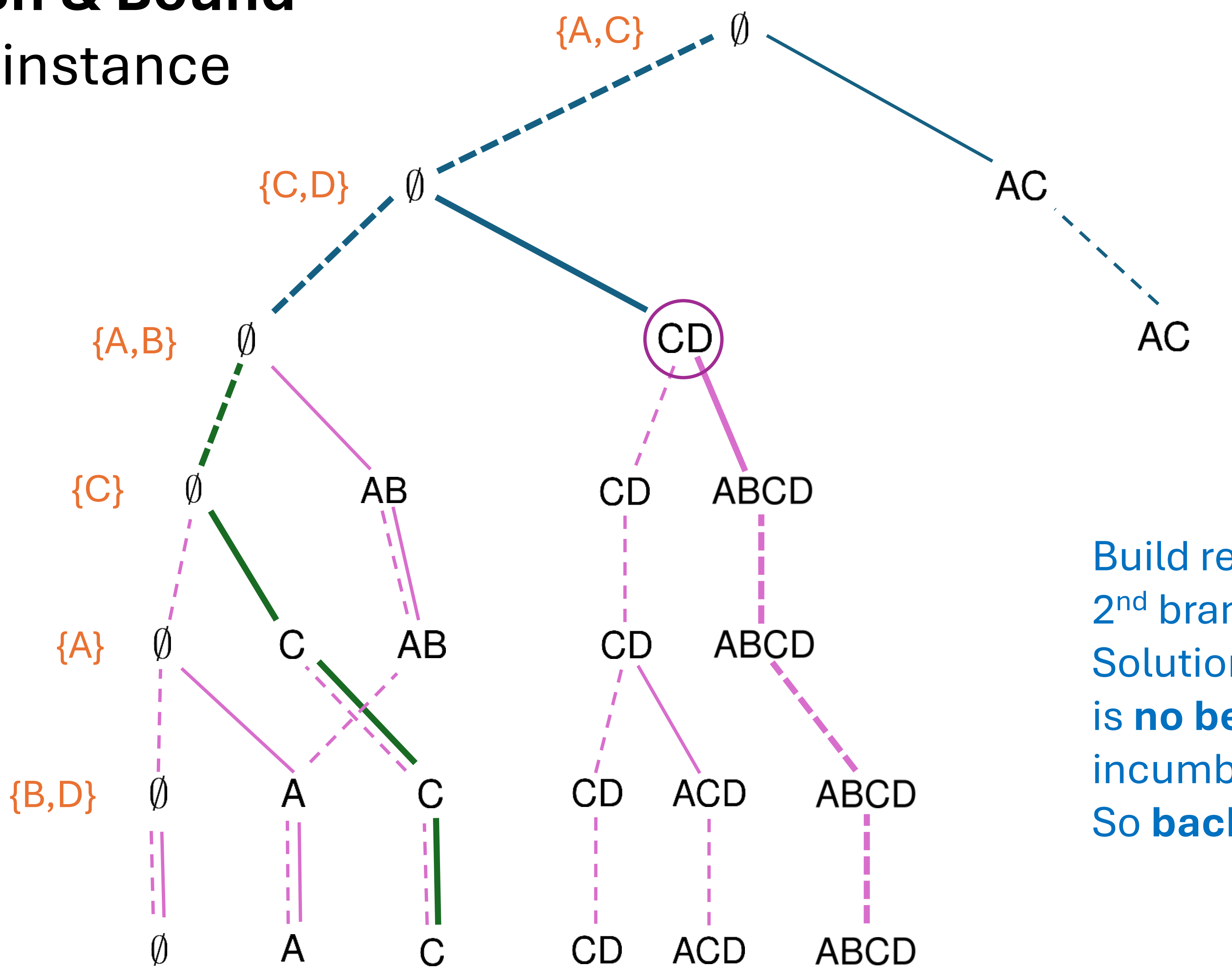
Longest path in the relaxed DD happens to be **feasible**. So we have an **incumbent solution** with value 3.

If longest path is infeasible, we continue recursively by branching at the last exact layer of this relaxed DD.

Carnegie Mellon University
Tepper School of Business

DD-based Branch & Bound

for a set packing instance



Decide whether to select set {B,D}

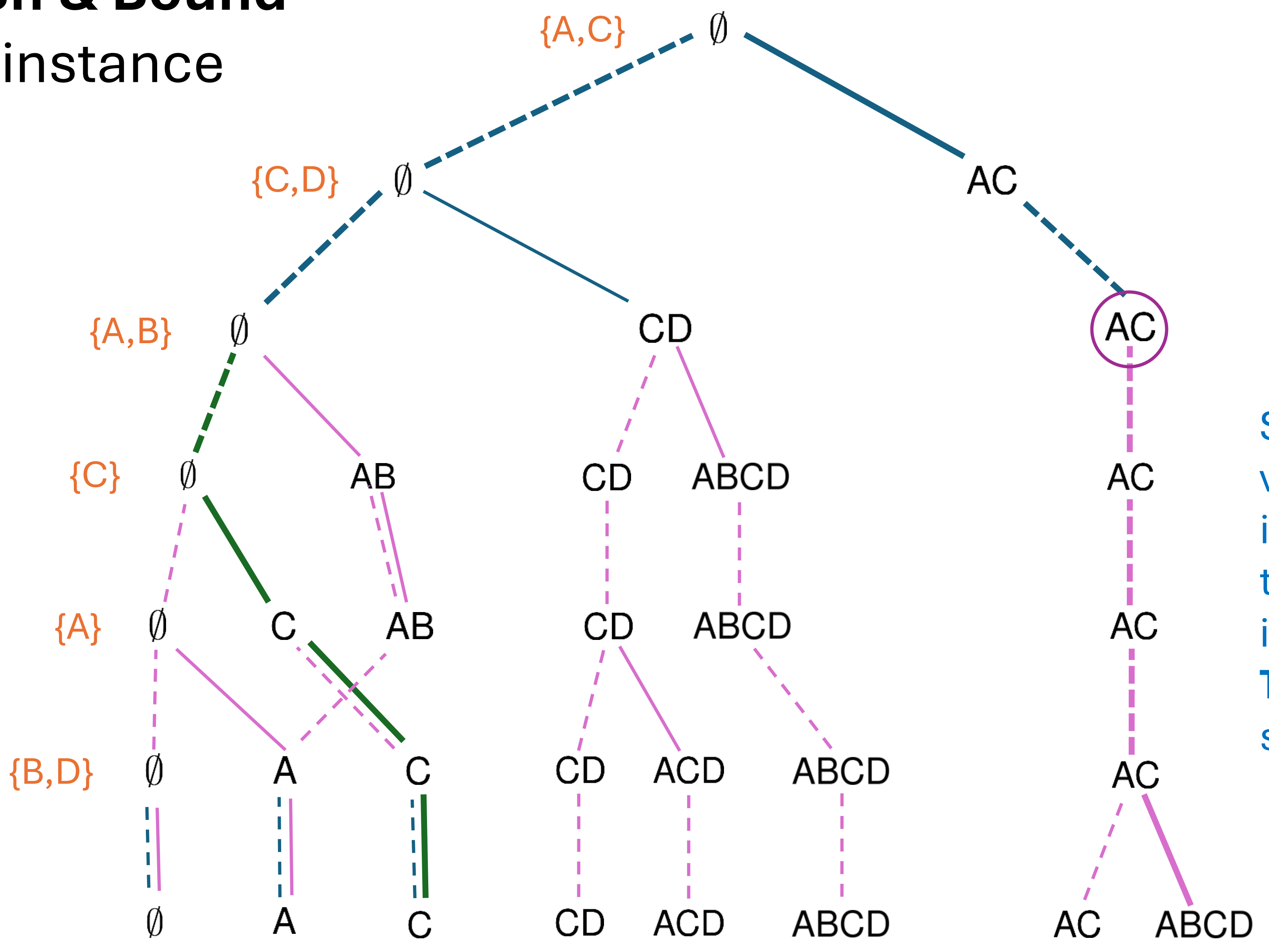
Upper bound = 3
Incumbent solution

Upper bound = 2
Backtrack

Build relaxed DD at 2nd branching node
Solution value 2 is **no better** than incumbent.
So **backtrack**.

Carnegie Mellon University
Tepper School of Business

DD-based Branch & Bound
for a set packing instance



Decide whether to select set {B,D}

Upper bound = 3
Incumbent solution

Upper bound = 2
Backtrack

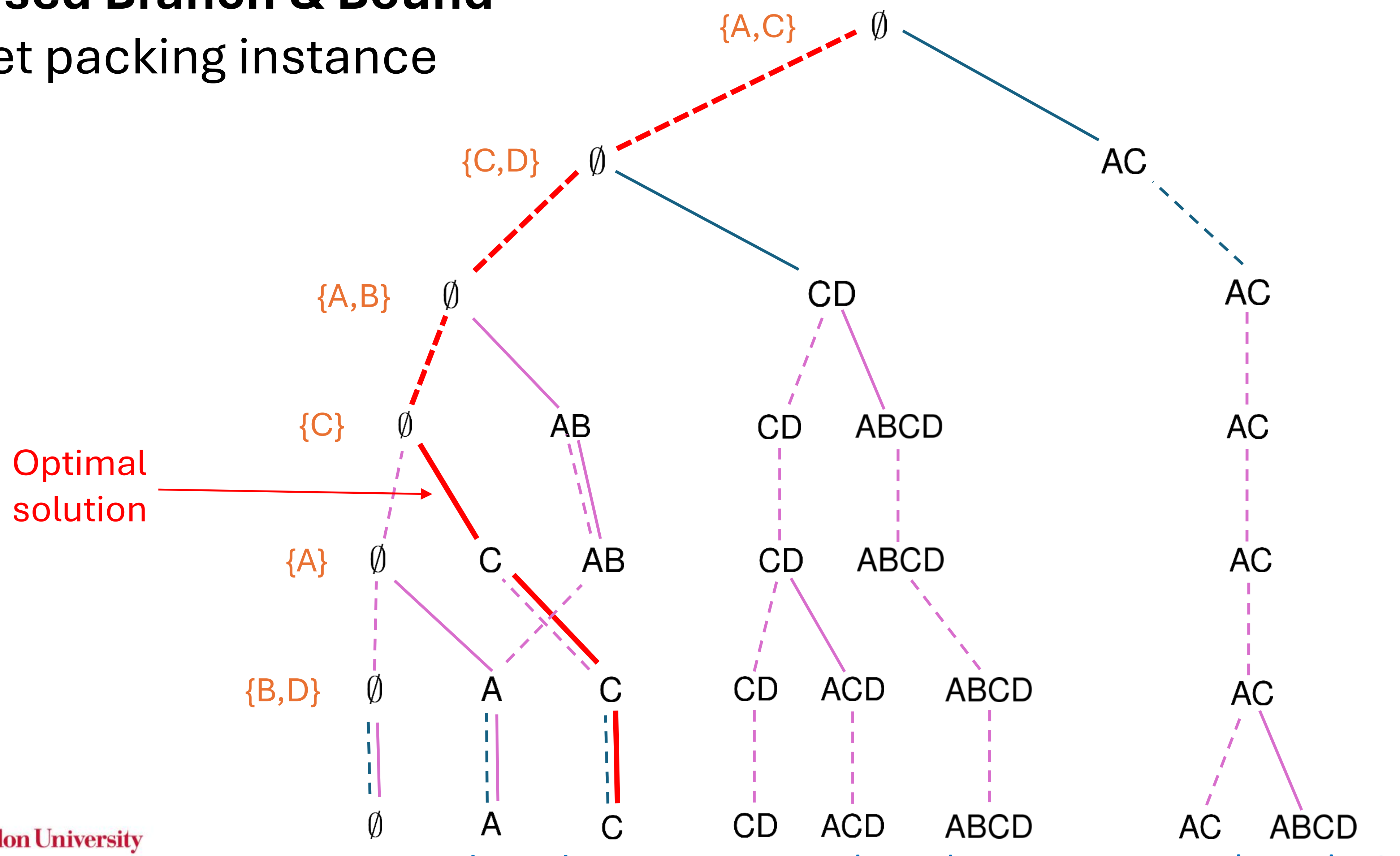
Upper bound = 2
Backtrack

Solution value 2 is **no better** than incumbent. **Terminate** search.

Carnegie Mellon University
Tepper School of Business

DD-based Branch & Bound

for a set packing instance



Upper bound = 3
Optimal solution

Upper bound = 2
Backtrack

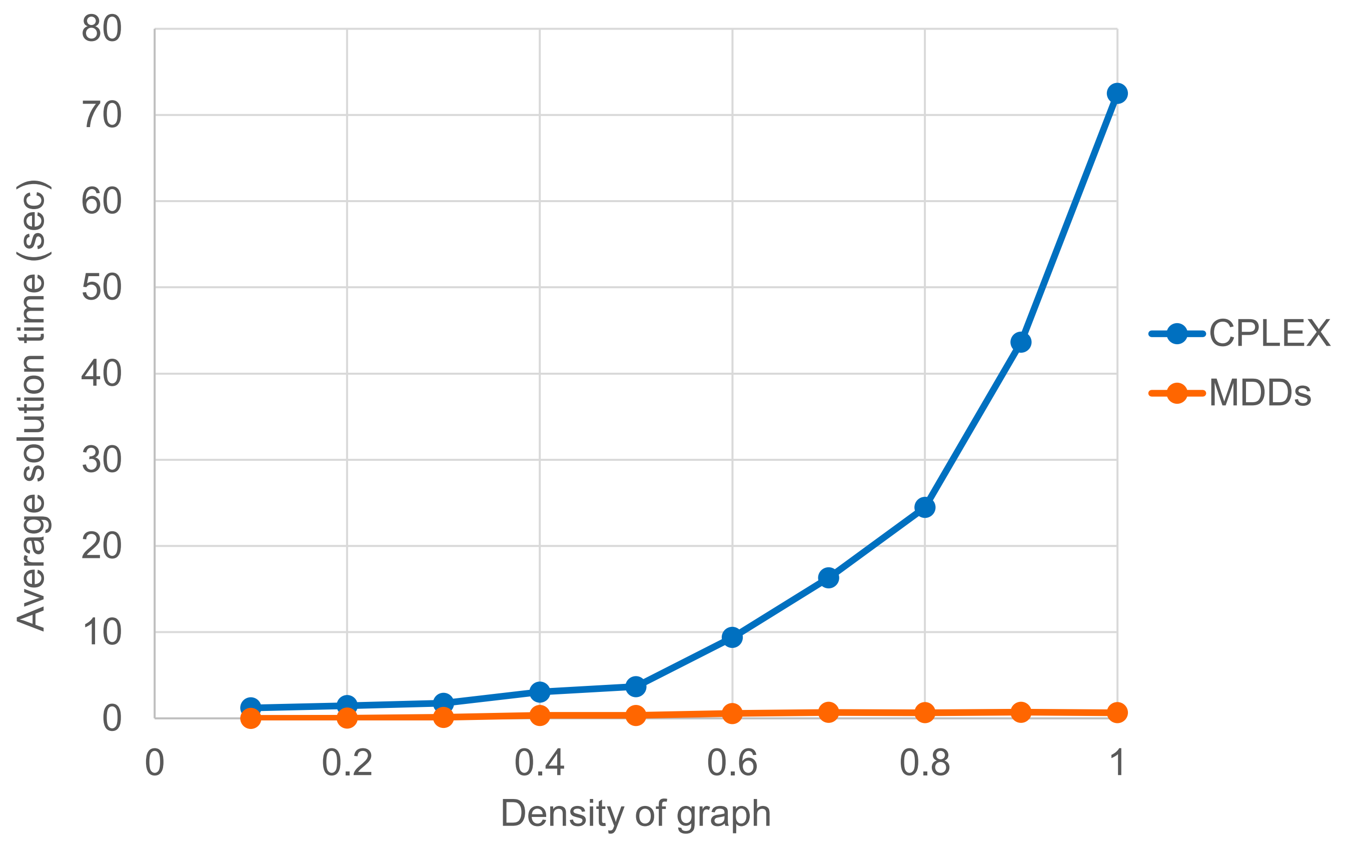
Upper bound = 2
Backtrack

Carnegie Mellon University
Tepper School of Business

DD-based Branch & Bound
Experimental results

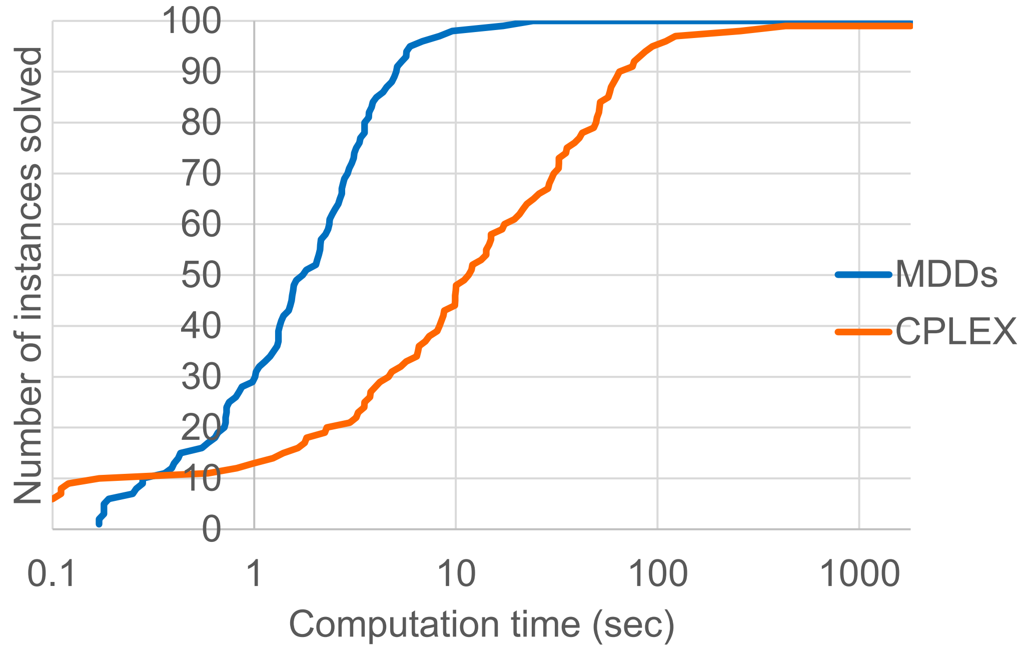
Computation time
for **max cut problem**
on a graph

Bergman, Ciré,
van Hoeve, JH (2016)

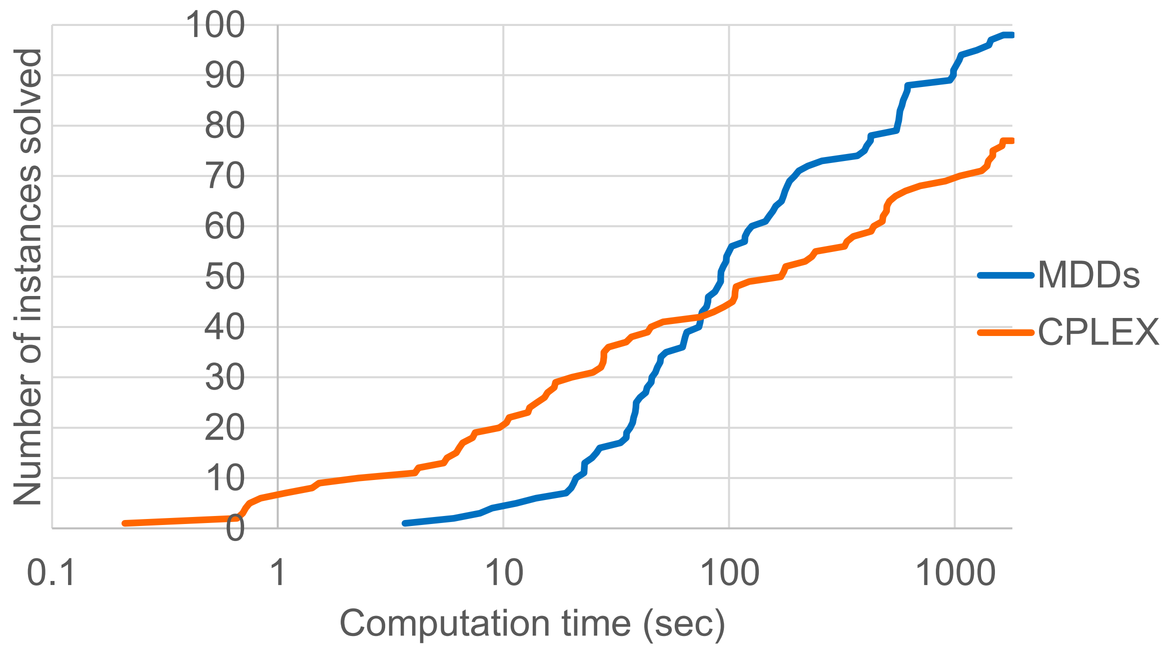


DD-based Branch & Bound
Experimental results

Performance profiles for **max 2SAT**



30 variables



40 variables

Carnegie Mellon University
Tepper School of Business

Bergman, Ciré,
van Hoeve, JH (2016)

DD-based constraint propagation

Domain filtering and propagation are key elements of **constraint programming**.

Filtering removes values from variable domains that are inconsistent with a given constraint.

The reduced domains are **propagated** to the next constraint for additional filtering.

DD-based constraint propagation

Domain filtering and propagation are key elements of **constraint programming**.

Filtering removes values from variable domains that are inconsistent with a given constraint.

The reduced domains are **propagated** to the next constraint for additional filtering.

Proposal: maintain a **relaxed DD,** rather than just variable domains, for each constraint.

Andersen, Hadžić, JH, Tiedemann (2007)

Propagation of a relaxed DD **conveys more information** than domains.

DD-based constraint propagation

Example

Standard **domain propagation**

$x_1 + 2x_2 + 3x_3 \leq 10$ $\longleftarrow$ filters domains to $x_1, x_2 \in \{1,2,3\},\ x_3 \in \{1,2\}$

$\text{all-different}(x_1, x_2, x_3)$ $\longleftarrow$ no more filtering possible for **propagated** domains

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

DD-based constraint propagation

Example

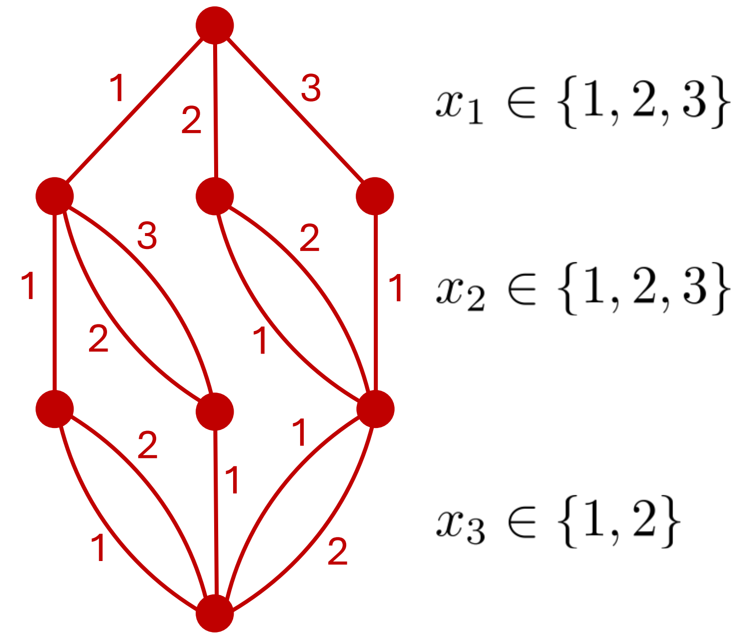
Standard **domain propagation**

$$x_1 + 2x_2 + 3x_3 \leq 10$$ ← filters domains to $x_1, x_2 \in \{1,2,3\}$, $x_3 \in \{1,2\}$

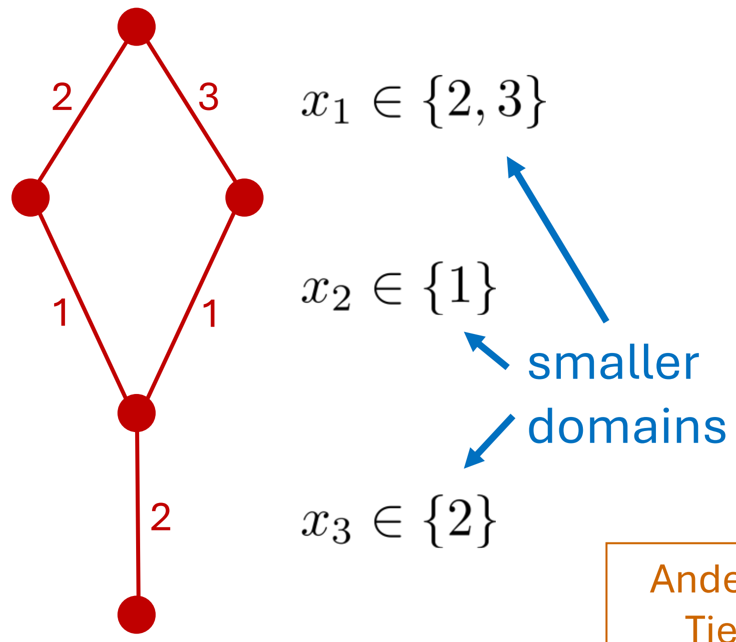
$$\text{all-different}(x_1, x_2, x_3)$$ ← no more filtering possible for **propagated** domains

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

Propagation through a **relaxed DD**



Relaxed DD for
$x_2 + 2x_2 + 3x_3 \leq 10$



Relaxed DD after applying all-different
to propagated DD rather than propagated domains

Andersen, Hadžić, JH, Tiedemann (2007)

DD-based constraint propagation
Experimental results

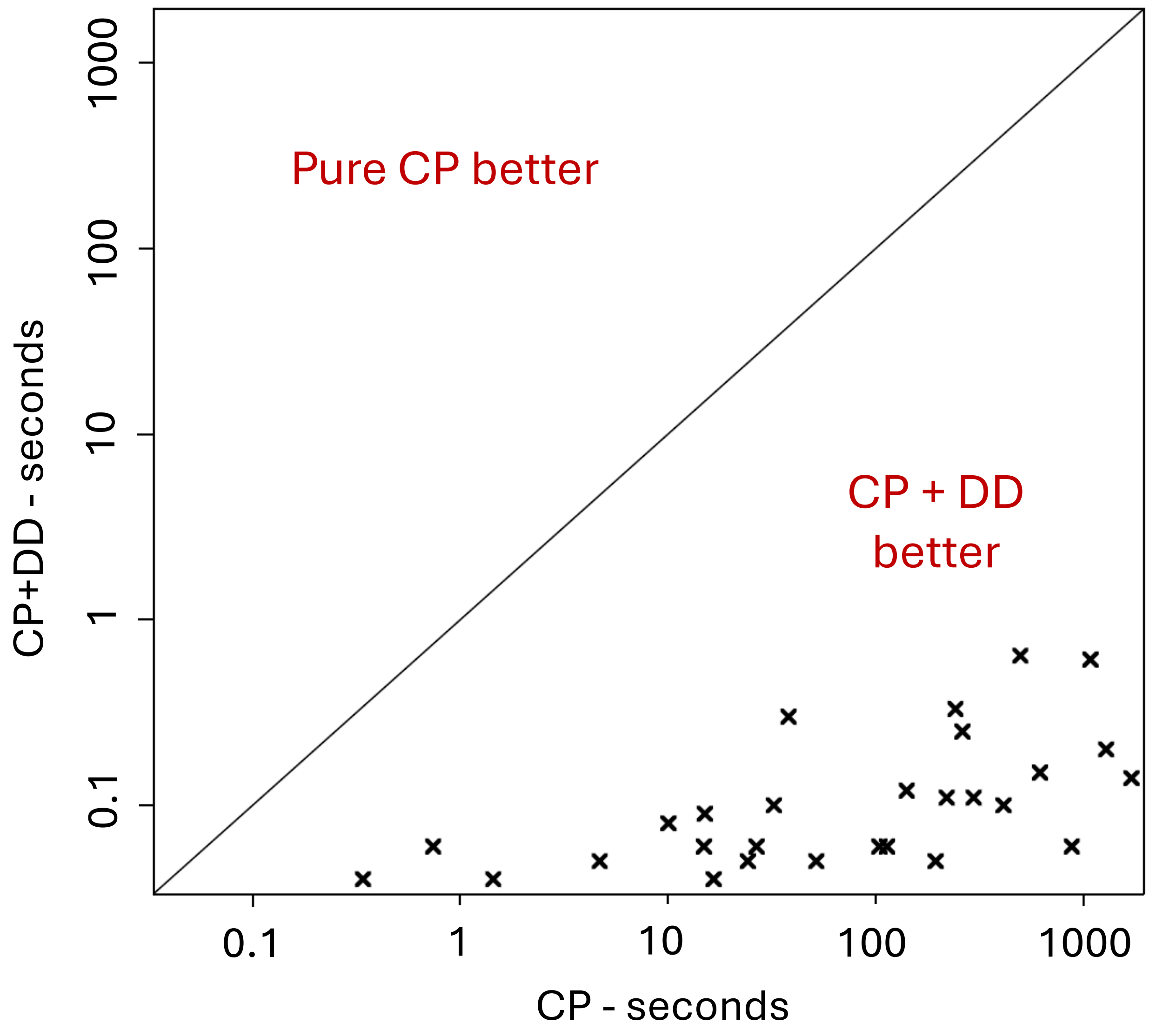
Traveling salesman problem with time windows

Intensely studied problem

Relaxed DD propagator for all-diff added to standard CP solver

Closed 3 long-standing open instances

Ciré & van Hoeve (2013)



DD-based Lagrangian relaxation

Lagrange multipliers can be added to arc costs to obtain **tighter DD-based bounds** on the optimal value.

Bergman, Ciré,
van Hoeve (2015)

Classical methods can then be used to solve the Lagrangian dual on the DD.

This takes time, but the resulting extremely tight bounds can be used to **assess the quality of heuristic solutions**.

DD-based Lagrangian relaxation

Example: Job sequencing

Let x_i be the *i*-th job in the sequence.
A **path** in the DD corresponds to an **assignment of jobs** to positions 1, ..., *n*.
Each job must occur **exactly once** in a feasible path.

So for each job *j* we must have exactly one x_i equal to *j*

$$-1 + \sum_i \boxed{[x_i = j]} = 0$$

$$= \begin{cases} 1 & \text{if } x_i = j \\ 0 & \text{otherwise} \end{cases}$$

Carnegie Mellon University
Tepper School of Business

DD-based Lagrangian relaxation

Example: Job sequencing

Let x_i be the *i*-th job in the sequence.
A **path** in the DD corresponds to an **assignment of jobs** to positions 1, ..., *n*.
Each job must occur **exactly once** in a feasible path.

So for each job *j* we must have exactly one x_i equal to *j*

$$-1 + \sum_i \boxed{[x_i = j]} = 0$$

$$= \begin{cases} 1 & \text{if } x_i = j \\ 0 & \text{otherwise} \end{cases}$$

The Lagrangian relaxation of the problem is

$$\min \sum_i \underbrace{\boxed{c_{ix_i}}}_{\text{Original arc cost from layer } i} + \sum_j \underbrace{\lambda_j}_{\text{Lagrange multiplier}} \Big(\underbrace{\boxed{-1 + \sum_i [x_i = j]}}_{\text{Should be zero}} \Big)$$

Original arc cost from layer *i*

Lagrange multiplier

Should be zero

DD-based Lagrangian relaxation

Example: Job sequencing

Let x_i be the *i*-th job in the sequence.
A **path** in the DD corresponds to an **assignment of jobs** to positions 1, ..., *n*.
Each job must occur **exactly once** in a feasible path.

$$= \begin{cases} 1 & \text{if } x_i = j \\ 0 & \text{otherwise} \end{cases}$$

So for for each job *j* we must have exactly one equal to *j*

$$-1 + \sum_i \boxed{[x_i = j]} = 0$$

The Lagrangian relaxation of the problem is

$$\min \sum_i c_{ix_i} + \sum_j \lambda_j \Big(-1 + \sum_i [x_i = j] \Big) = \sum_i \Big(c_{ix_i} + \lambda_{x_j} \Big) - \sum_j \lambda_j$$

Rearranging

Carnegie Mellon University
Tepper School of Business

DD-based Lagrangian relaxation

Example: Job sequencing

Let x_i be the *i*-th job in the sequence.
A **path** in the DD corresponds to an **assignment of jobs** to positions 1, ..., *n*.
Each job must occur **exactly once** in a feasible path.

$$= \begin{cases} 1 & \text{if } x_i = j \\ 0 & \text{otherwise} \end{cases}$$

So for each job *j* we must have exactly one equal to *j*

$$-1 + \sum_i \boxed{[x_i = j]} = 0$$

The Lagrangian relaxation of the problem is

$$\min \sum_i c_{ix_i} + \sum_j \lambda_j \Big(-1 + \sum_i [x_i = j] \Big) = \sum_i \Big(\boxed{c_{ix_i} + \lambda_{x_j}} \Big) - \boxed{\sum_j \lambda_j}$$

This becomes arc cost in relaxed DD

Offset penalty at top of DD

Carnegie Mellon University
Tepper School of Business

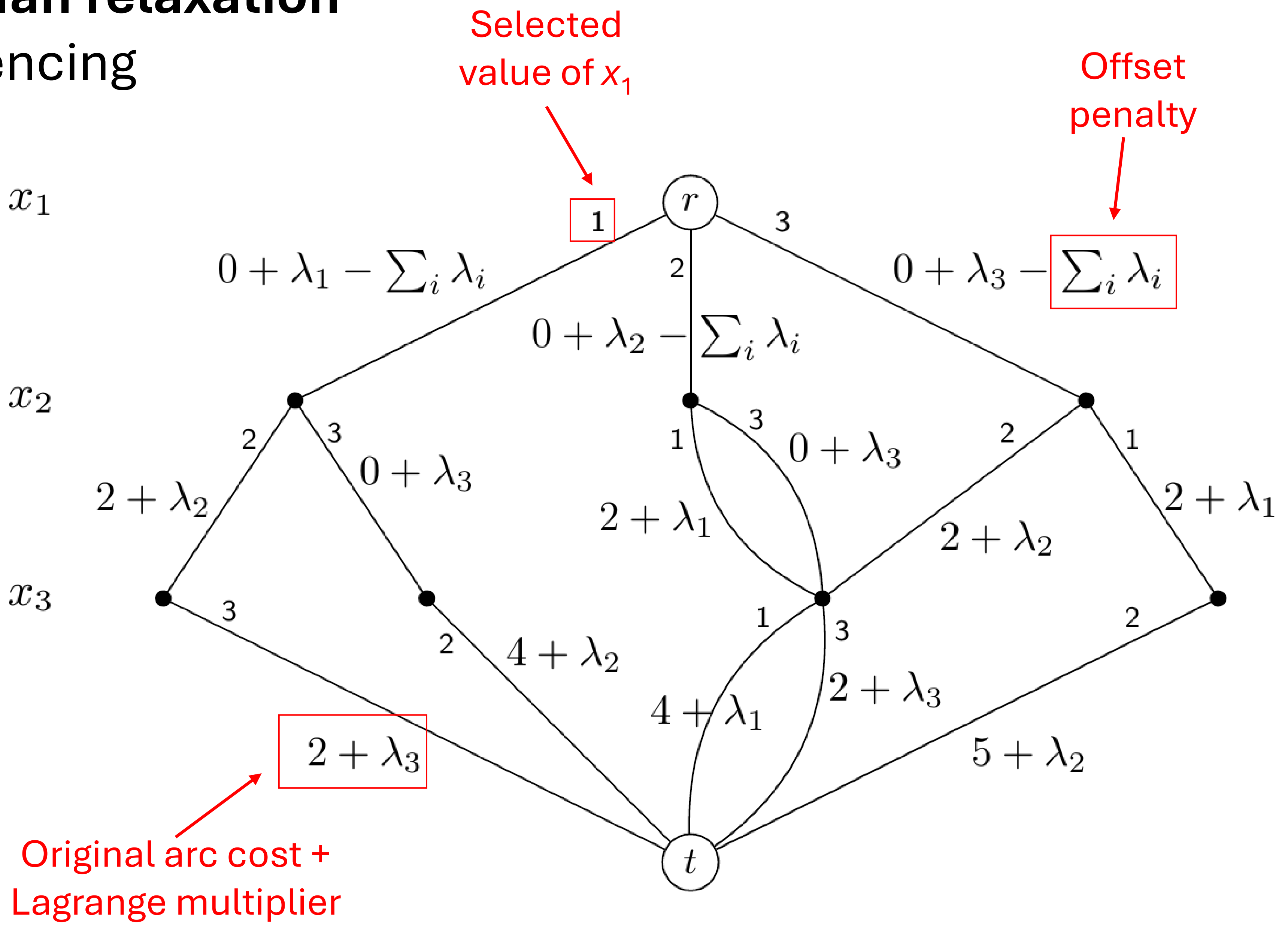
DD-based Lagrangian relaxation

Example: Job sequencing

Relaxed sequencing DD with Lagrange multipliers (3 jobs)

This is a **multivalued** DD

Shortest path is solution of Lagrangian relaxation for a given set of λs.



Carnegie Mellon University

Tepper School of Business

DD-based Lagrangian relaxation
Computational experiments

A set of 60 **hard job sequencing instances** have been studied for 25 years.

Biskup & Feldman (2001)

As of 2019, **none** had been solved to proven optimality, although heuristic algorithms had been proposed.

DDs + Lagrangian relaxation obtain **extremely tight bounds**, showing that the heuristic solutions are **very close to optimal**.

6 solutions are proved optimal.

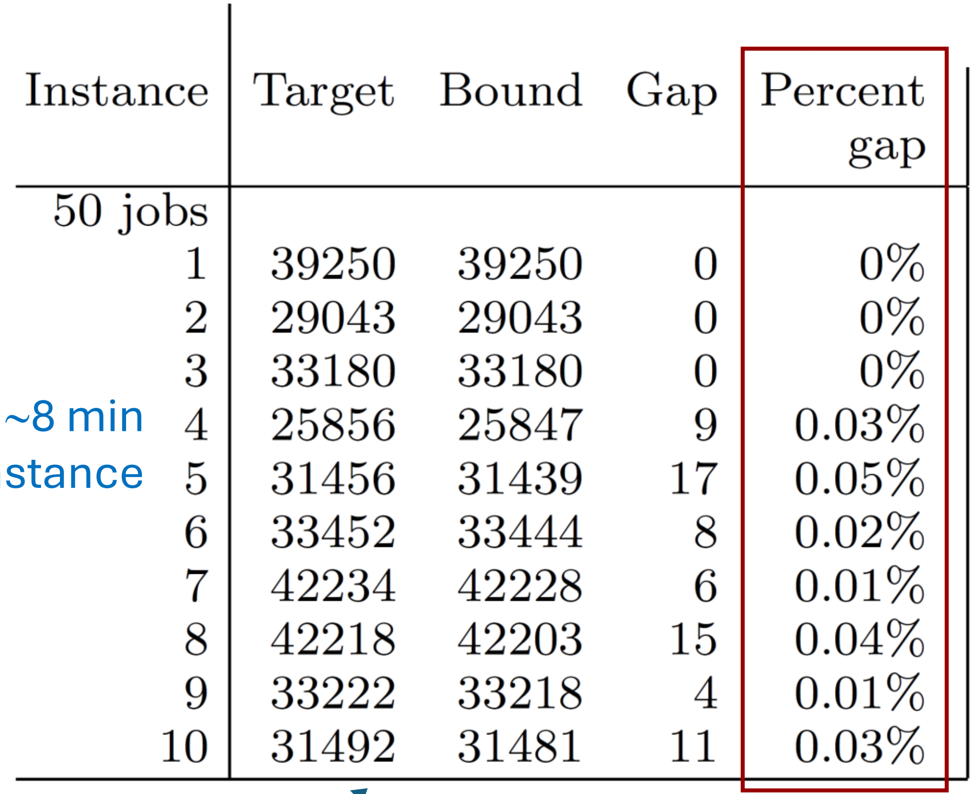
JH (2019)

DD-based Lagrangian relaxation

Computational experiments

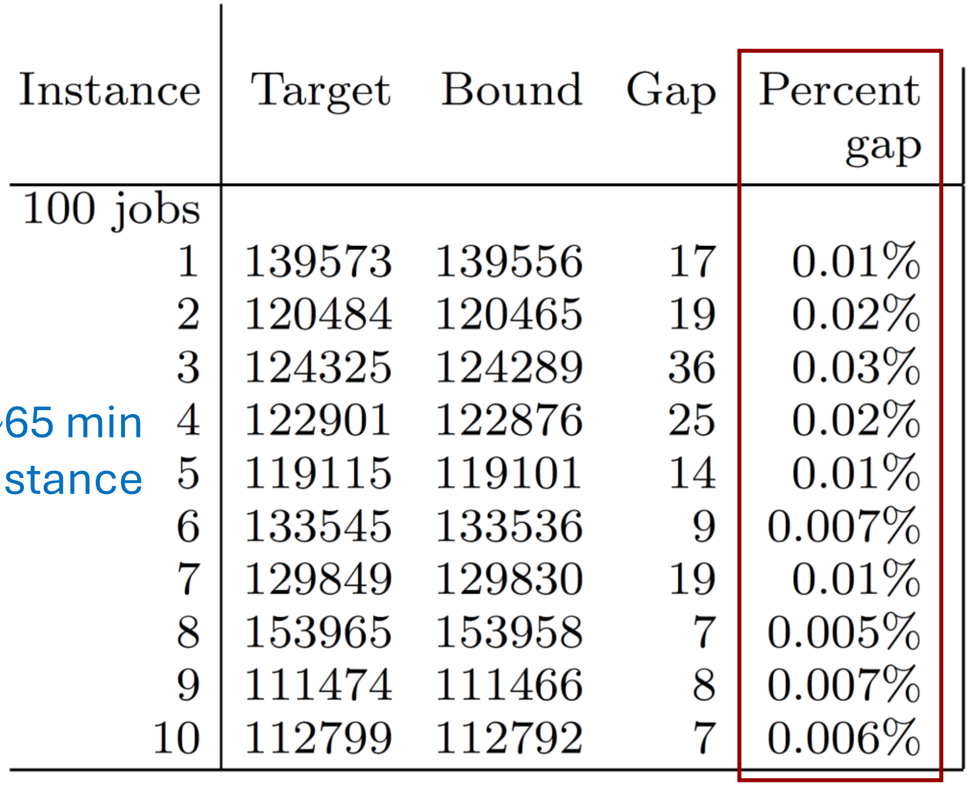
Sampling of results, Biskup-Feldman instances

Time: ~8 min per instance



Instance	Target	Bound	Gap	Percent gap
50 jobs				
1	39250	39250	0	0%
2	29043	29043	0	0%
3	33180	33180	0	0%
4	25856	25847	9	0.03%
5	31456	31439	17	0.05%
6	33452	33444	8	0.02%
7	42234	42228	6	0.01%
8	42218	42203	15	0.04%
9	33222	33218	4	0.01%
10	31492	31481	11	0.03%

Time: ~65 min per instance



Instance	Target	Bound	Gap	Percent gap
100 jobs				
1	139573	139556	17	0.01%
2	120484	120465	19	0.02%
3	124325	124289	36	0.03%
4	122901	122876	25	0.02%
5	119115	119101	14	0.01%
6	133545	133536	9	0.007%
7	129849	129830	19	0.01%
8	153965	153958	7	0.005%
9	111474	111466	8	0.007%
10	112799	112792	7	0.006%

Best known solution

JH (2019)

Carnegie Mellon University
Tepper School of Business

Other developments

- **Network flow** model of DD
    - Allows DD to be integrated into linear or integer programming model.

Becker et al. (2005)
Behle (2007)
Bergman & Lozano (2021)

- **Cutting planes** from DD network flow models
    - Focus on separation algorithms

Becker et al. (2005)
Behle (2007)
Tjandraatmadja & van Hoeve (2019)
Davarnia & van Hoeve (2021)

- **Multiple** network flow DDs with linked variables
    - LP/MILP model provides linking constraints.

Bergman & Ciré (2016)
Bergman, Cardonha, Mehrani (2019)
Lozano, Bergman, Smith (2020)
Nadaraja & Ciré (2020)
Castro, Cire, Beck (2022)

- Flow-based DDs for **nonlinear** problems

Bergman & Ciré (2018)
Lozano, Bergman, Smith (2020)
Bergman & Lozano (2021)

Carnegie Mellon University
Tepper School of Business

Other developments

- DDs for **probabilistic constraints**
  - Uses sentential DDs, maps problem into MILP.

Latour et al. (2017)
Latour, Babaki, Nijssen (2019)

- Solving **2-stage stochastic** programs with DDs
  - Also maps to MILP.

Haus, Michini, Laumanns (2017)
Guo, Bodur, Alema, Urbach (2021)
Lozano & Smith (2022)

- **Stochastic** exact and relaxed DDs
  - Can solve stochastic DP problems by branch and bound.

JH (2022)

- DD for **continuous** variables

Davarnia (2021)
Salemi & Davarnia (2021)

Other developments

- DDs in **Benders decomposition**
  - DD can represent either master problem or subproblem.

Bergman & Lozano (2021)
Lozano & Smith (2019)
Salemi & Davarnia (2021)

- **Feasibility checking** in constraint programming.
  - Nogood generation.

Subbarayan (2008)
Gange, Stuckey, Szymanek (2013)
Jung and Régin (2021)

- **Parallel computation** with DD-based branch and bound.
  - Much more effective than parallelization of IP solvers.

Bergman et al. (2014)

- **Postoptimality analysis** for IP
  - Much more comprehensive than traditional methods.

Hadžić & JH (2006)
Serra & JH (2020)

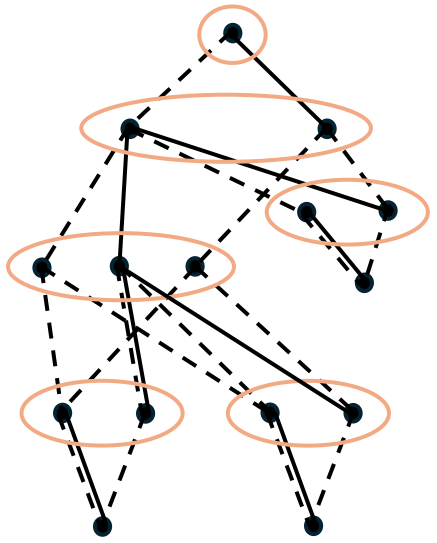
Carnegie Mellon University
Tepper School of Business

Other developments

- **General DD-based solver** for combinatorial optimization
    - **CODD**, based on DD compilation software **Ddo** and **HADDOCK**
    - Uses dynamic programming problem formulations

Gillard, Schaus, Coppé (2020)
Gentzel, Michel, van Hoeve (2020)
Michel & van Hoeve (2024)

Part II. Nonserial Decision Diagrams



Carnegie Mellon University
Tepper School of Business

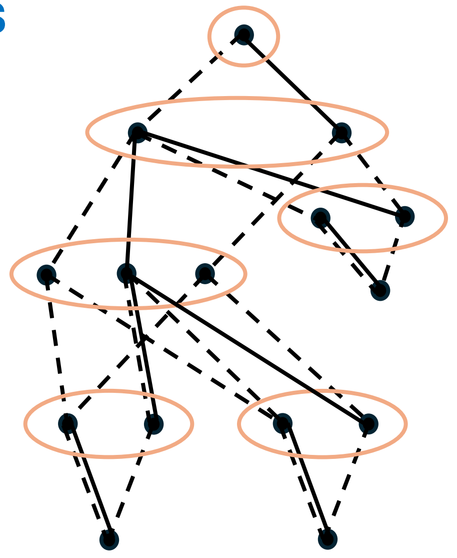
Nonserial DDs

They exploit structure of problem instances with **small treewidth**.

Treewidth (with respect to an ordering) = **max in-degree** of nodes in the **induced** dependency graph.

Complexity of a problem **instance** is at worst exponential in its minimum **treewidth** over all orderings.

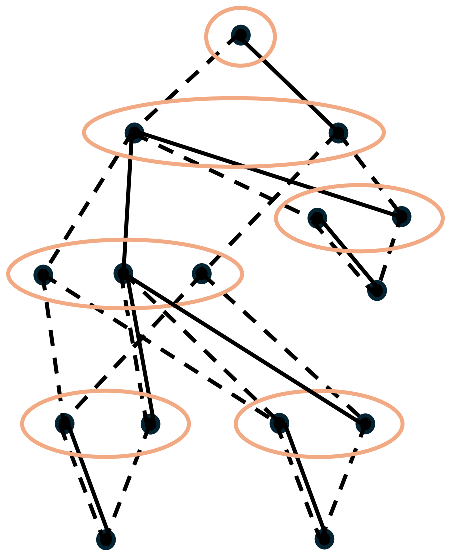
Instances with **small treewidth** generate **much smaller nonserial DDs** and are **much easier to solve**.



Nonserial DDs

Why nonserial DDs?

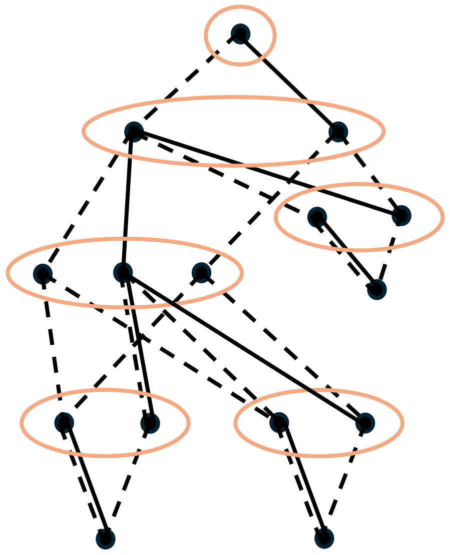
- They exploit structure of problem instances whose variables **partially decouple**.
- They combine **nonserial dynamic programming** ideas with **DD solution technology** – reduction, relaxation, restriction, flow models, etc.
- They can be **dramatically smaller** than serial DDs.
- Reduction in **compilation time** is **even greater**.



Nonserial DDs

When exact DDs are **smaller**....

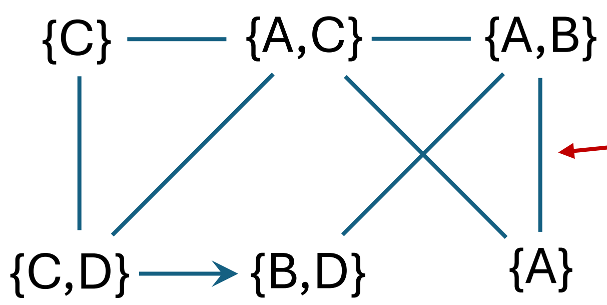
- **Relaxed DDs** of a given size provide **tighter bounds**.
- **Restricted DDs** of a given size are more likely to yield **feasible solutions**.
- **Flow models** are more likely to be **tractable**.



Dependency graph
For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

First, build **dependency graph** that shows variable coupling.
Here, 0-1 variables indicate whether each set is included in packing.



Arc indicates one or more elements in common

Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

First, build **dependency graph** that shows variable coupling.
Here, 0-1 variables indicate whether each set is included in packing.

$$\begin{tikzcd}
\{C\} \arrow[r] \arrow[d] & \{A,C\} & \{A,B\} \arrow[l] \\
\{C,D\} \arrow[ur] & \{B,D\} \arrow[l] \arrow[ur] & \{A\} \arrow[ul] \arrow[u]
\end{tikzcd}$$

Arc indicates one or more elements in common

We generally don't know the min-treewidth ordering.
As a heuristic, we use a **min-degree ordering**.

Carnegie Mellon University
Tepper School of Business

Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.

$$\begin{tikzcd}
\{C\} \arrow[r] \arrow[d] & \{A,C\} & \{A,B\} \arrow[l] \\
\{C,D\} \arrow[ur] & \boxed{\{B,D\}} \arrow[l] \arrow[ur] & \{A\} \arrow[ul] \arrow[u]
\end{tikzcd}$$

Remove

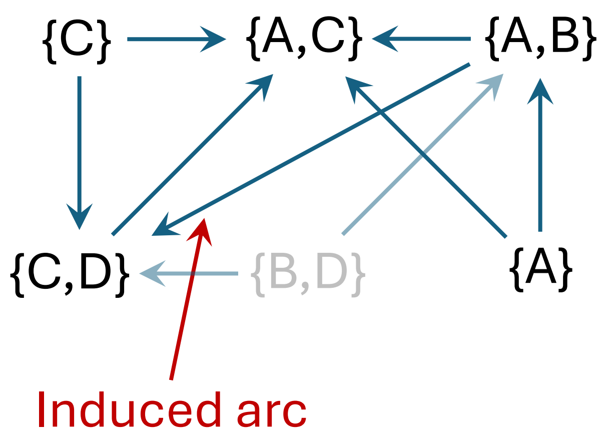
Build tree of layers for nonserial DD $\longrightarrow$ {B,D}

Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.



{B,D}

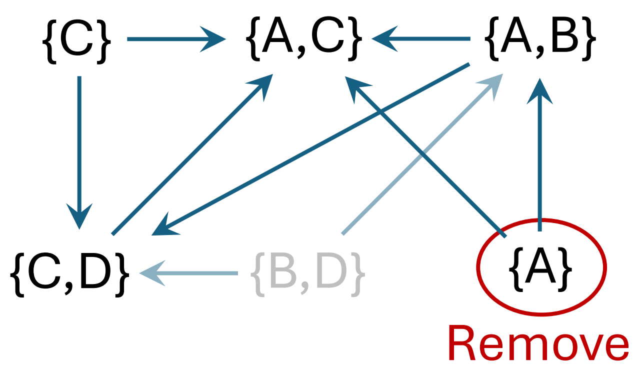
Build tree of layers for nonserial DD

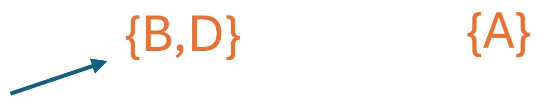
Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.





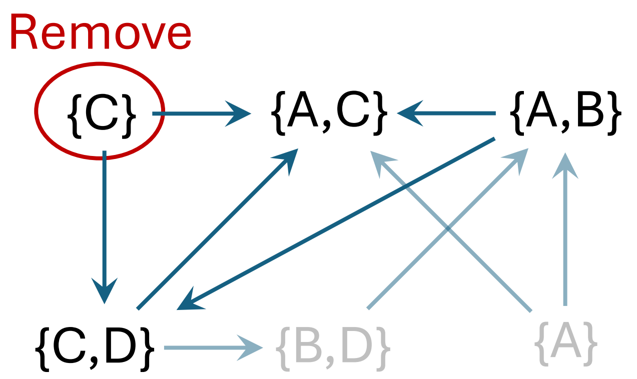
Build tree of levels for nonserial DD

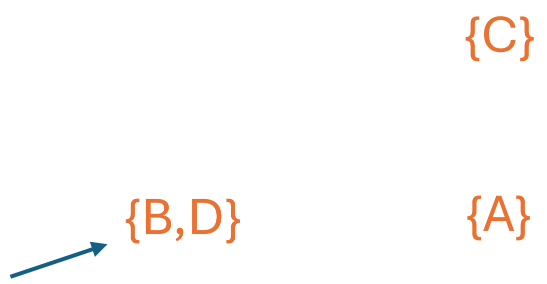
Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.





Build tree of levels
for nonserial DD

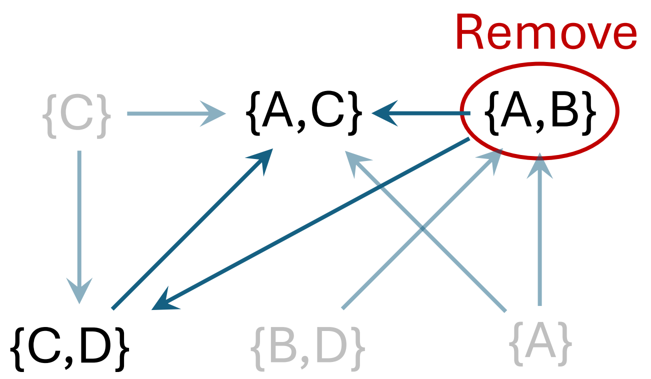
Carnegie Mellon University
Tepper School of Business

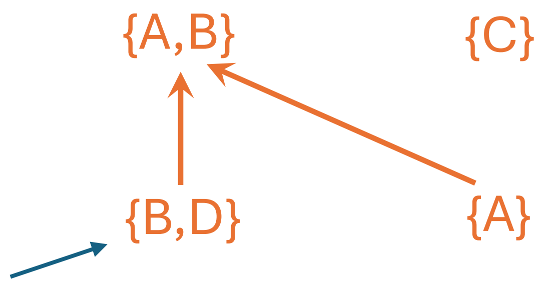
Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.





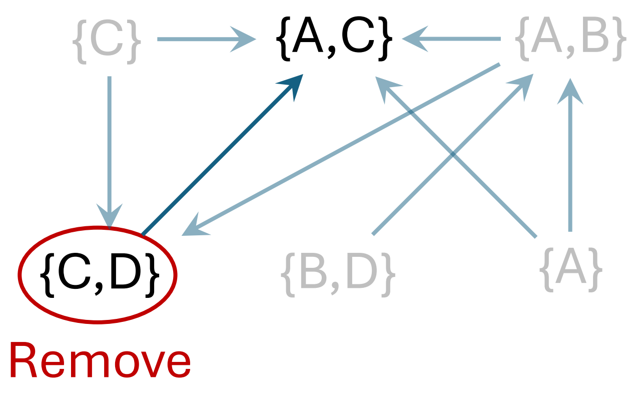
Build tree of levels for nonserial DD

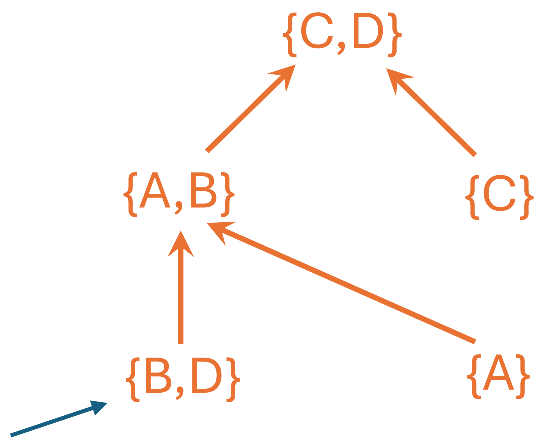
Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.





Dependency graph

For set packing example

{A,C}
{C,D}
{A,B}
{C}
{A}
{B,D}

Now, build **induced** dependency graph by removing nodes in min degree order, adding arcs to connect all neighbors.

$$\begin{tikzcd}
\{C\} \arrow[r] \arrow[d] & \{A,C\} & \{A,B\} \arrow[l] \arrow[dll] \\
\{C,D\} \arrow[ur] \arrow[r] & \{B,D\} \arrow[ur] & \{A\} \arrow[ull] \arrow[u]
\end{tikzcd}$$

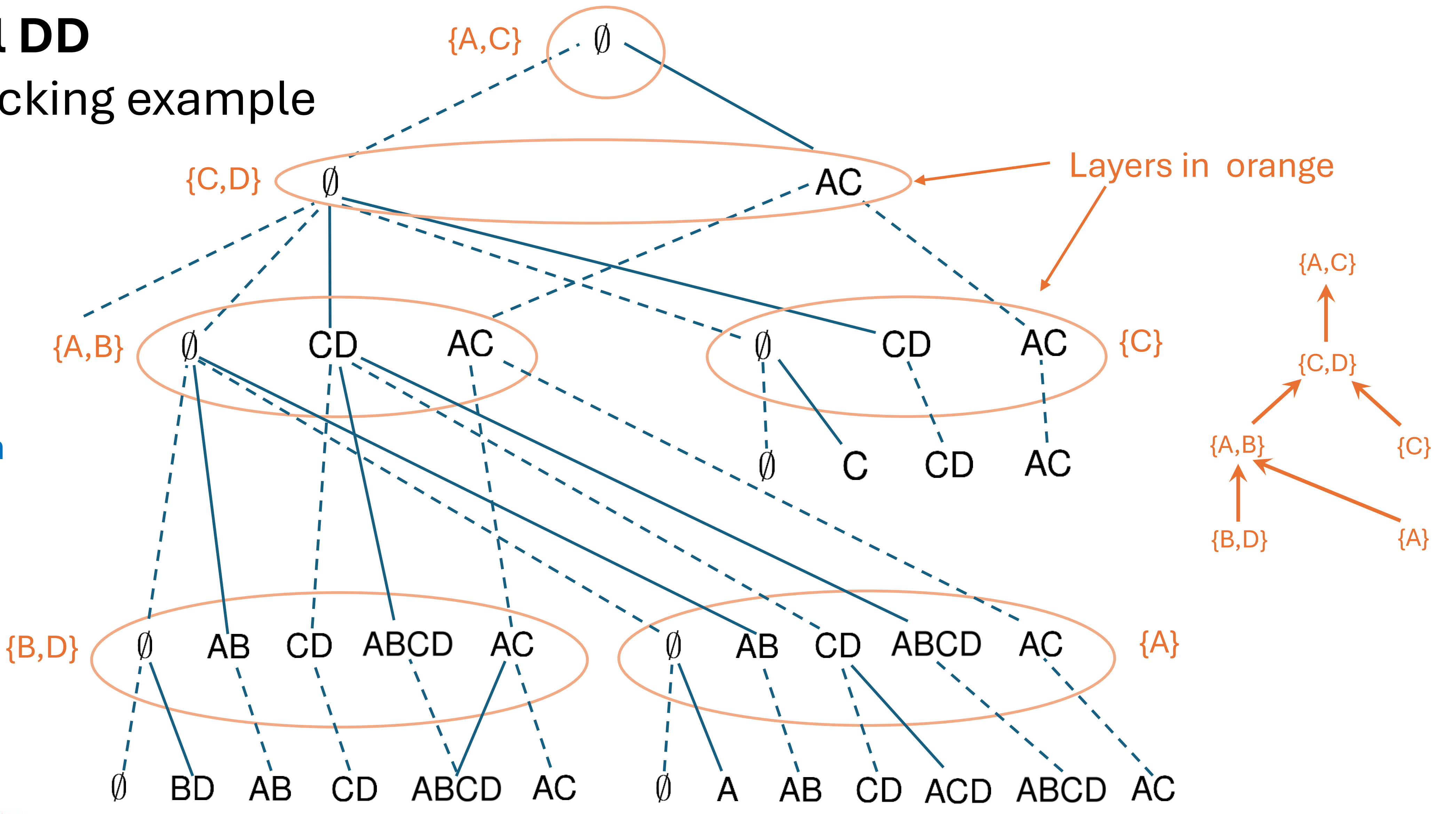
$$\begin{tikzcd}
& \{A,C\} & \\
& \{C,D\} \arrow[u] & \\
\{A,B\} \arrow[ur] & & \{C\} \arrow[ul] \\
\{B,D\} \arrow[u] & & \{A\} \arrow[ull]
\end{tikzcd}$$

Treewidth =
max in-degree = 2

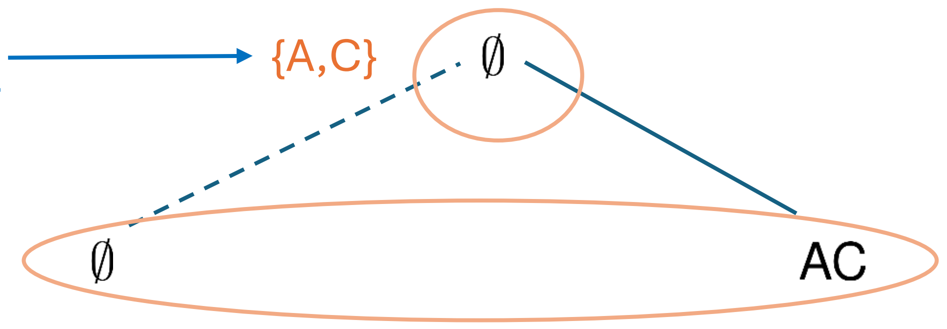
Nonserial DD

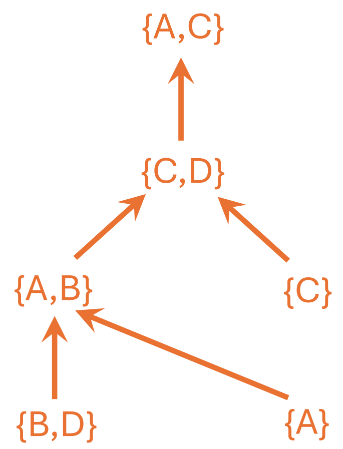
For set packing example

Layers form a **tree** rather than an ordered sequence



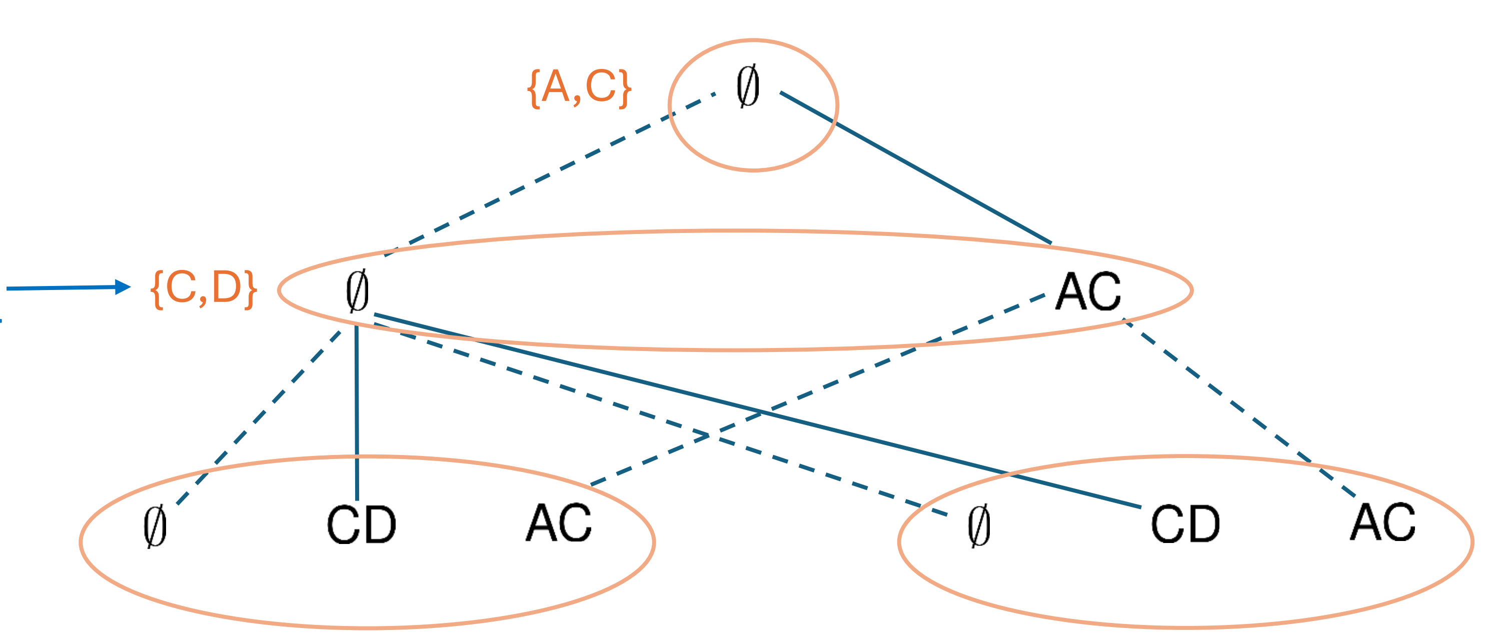
Decide whether to select set {A,C}

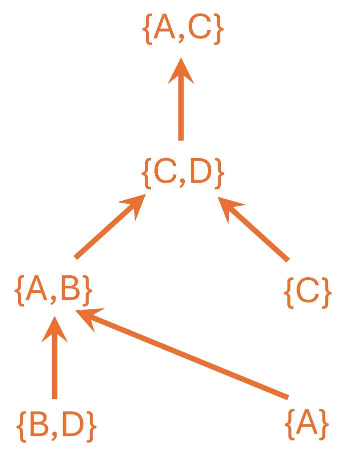


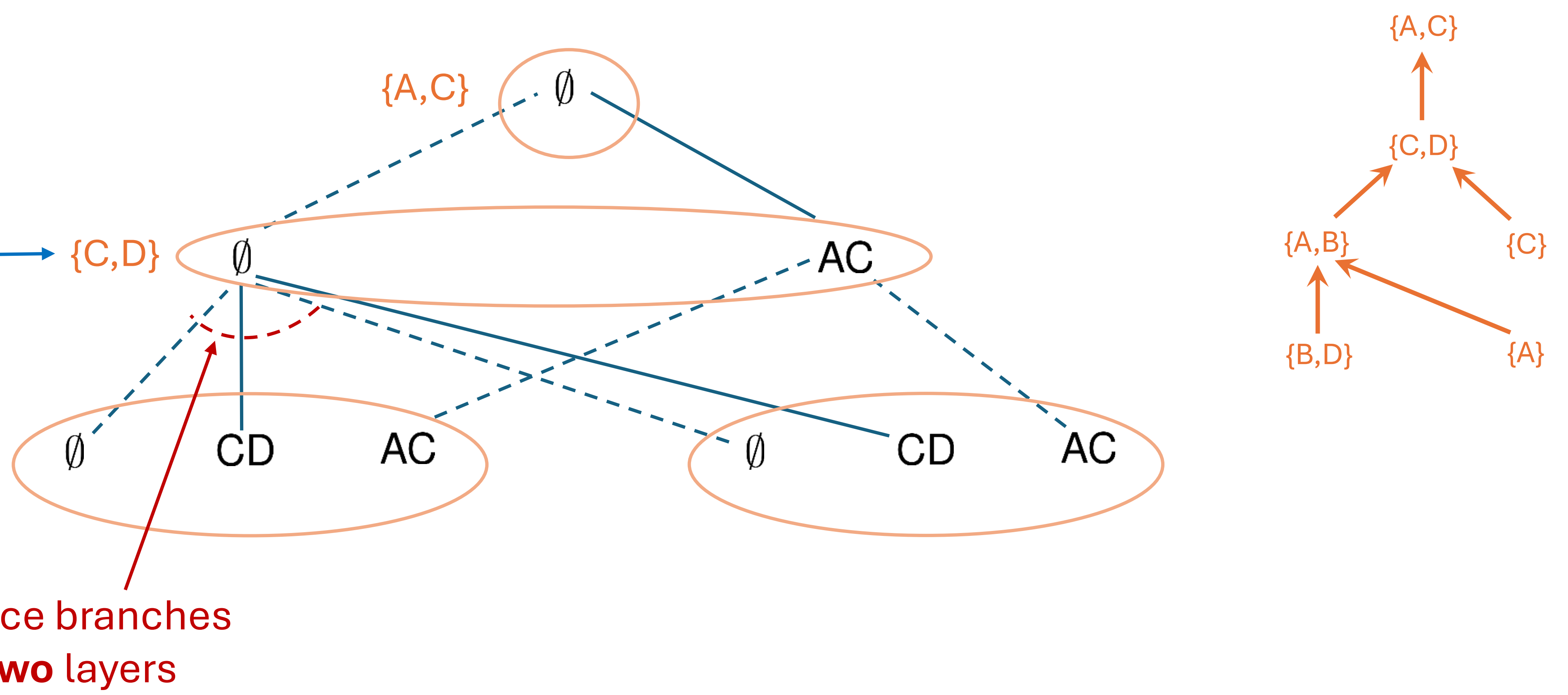


Carnegie Mellon University
Tepper School of Business

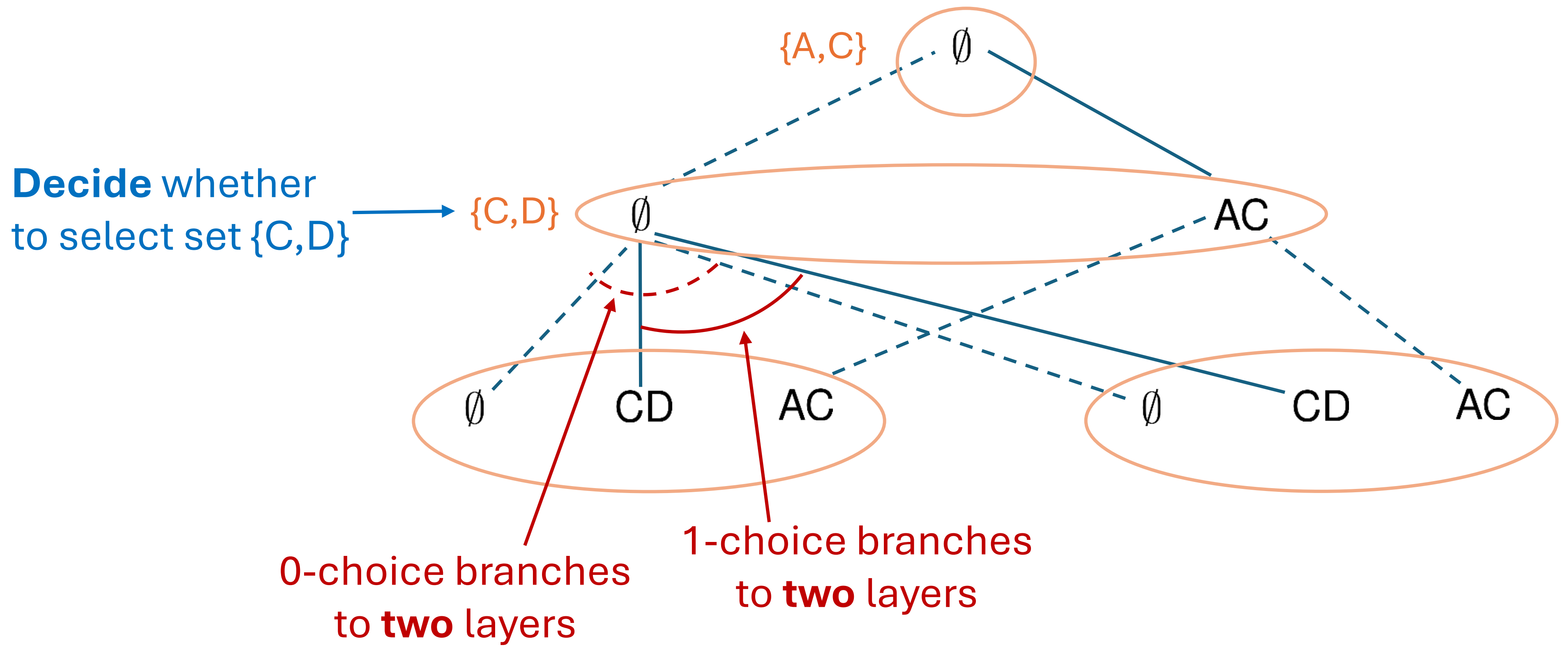
Decide whether to select set {C,D}

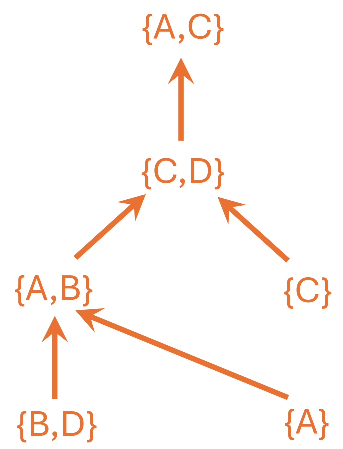


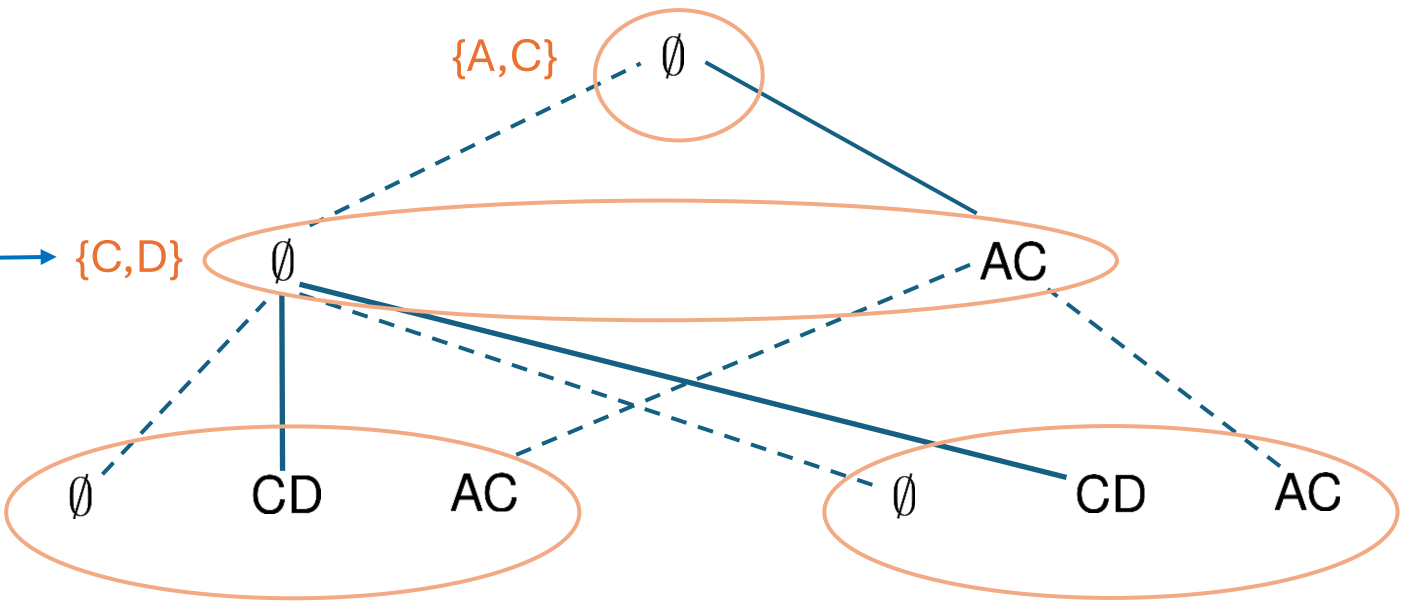



{A,C}
∅
Decide whether to select set {C,D}
{C,D}
∅
AC
∅
CD
AC
∅
CD
AC
0-choice branches to two layers
{A,C}
{C,D}
{A,B}
{C}
{B,D}
{A}

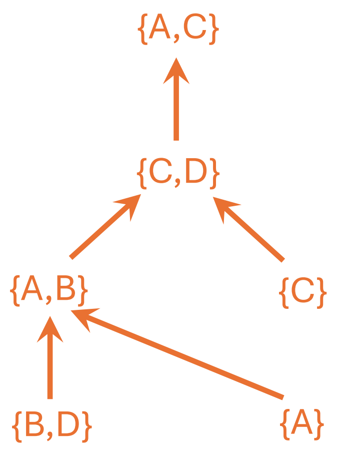
Carnegie Mellon University
Tepper School of Business


{A,C}
∅
Decide whether to select set {C,D}
{C,D}
∅
AC
∅
CD
AC
∅
CD
AC
0-choice branches to two layers
1-choice branches to two layers

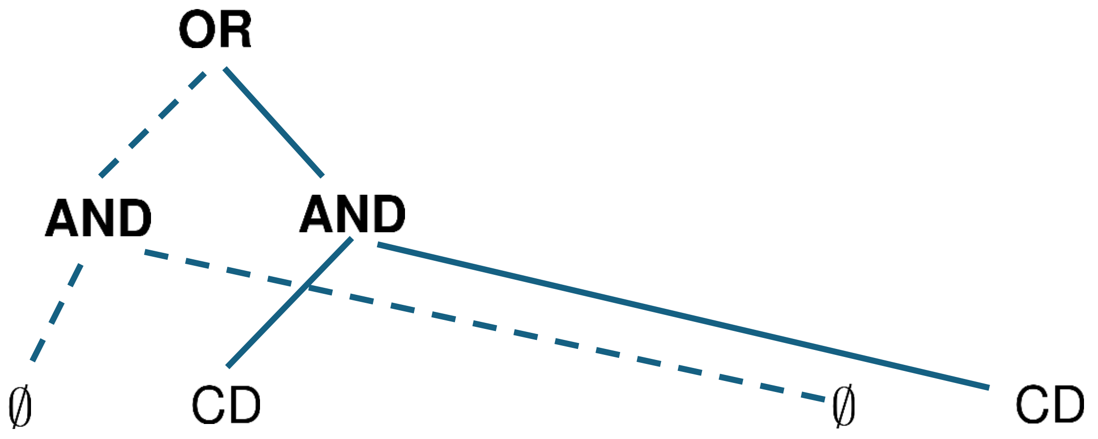

{A,C}
{C,D}
{A,B}
{C}
{B,D}
{A}



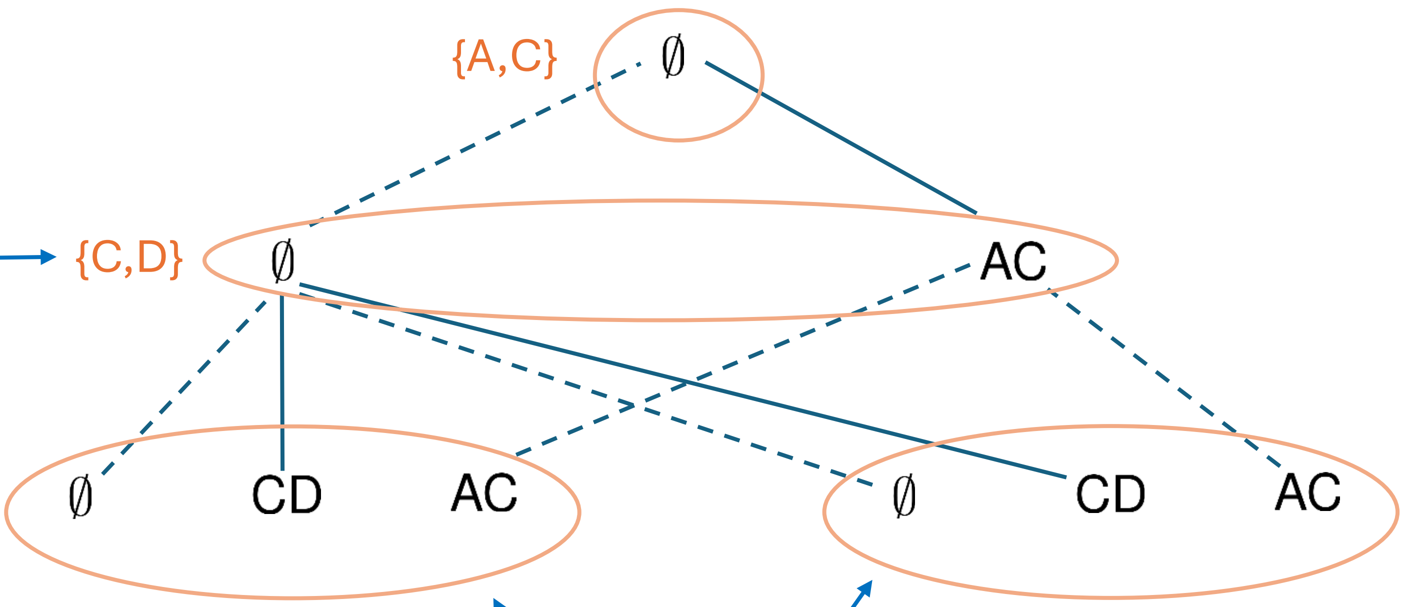
Decide whether to select set {C,D}



Can be viewed as **and-or** DD

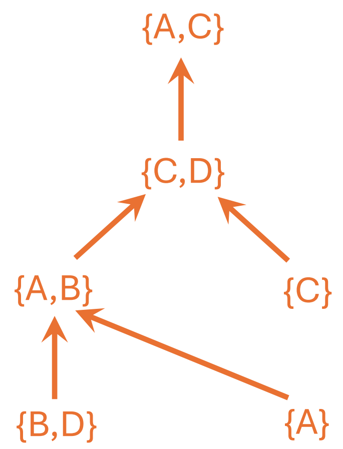


Carnegie Mellon University
Tepper School of Business



Decide whether to select set {C,D}

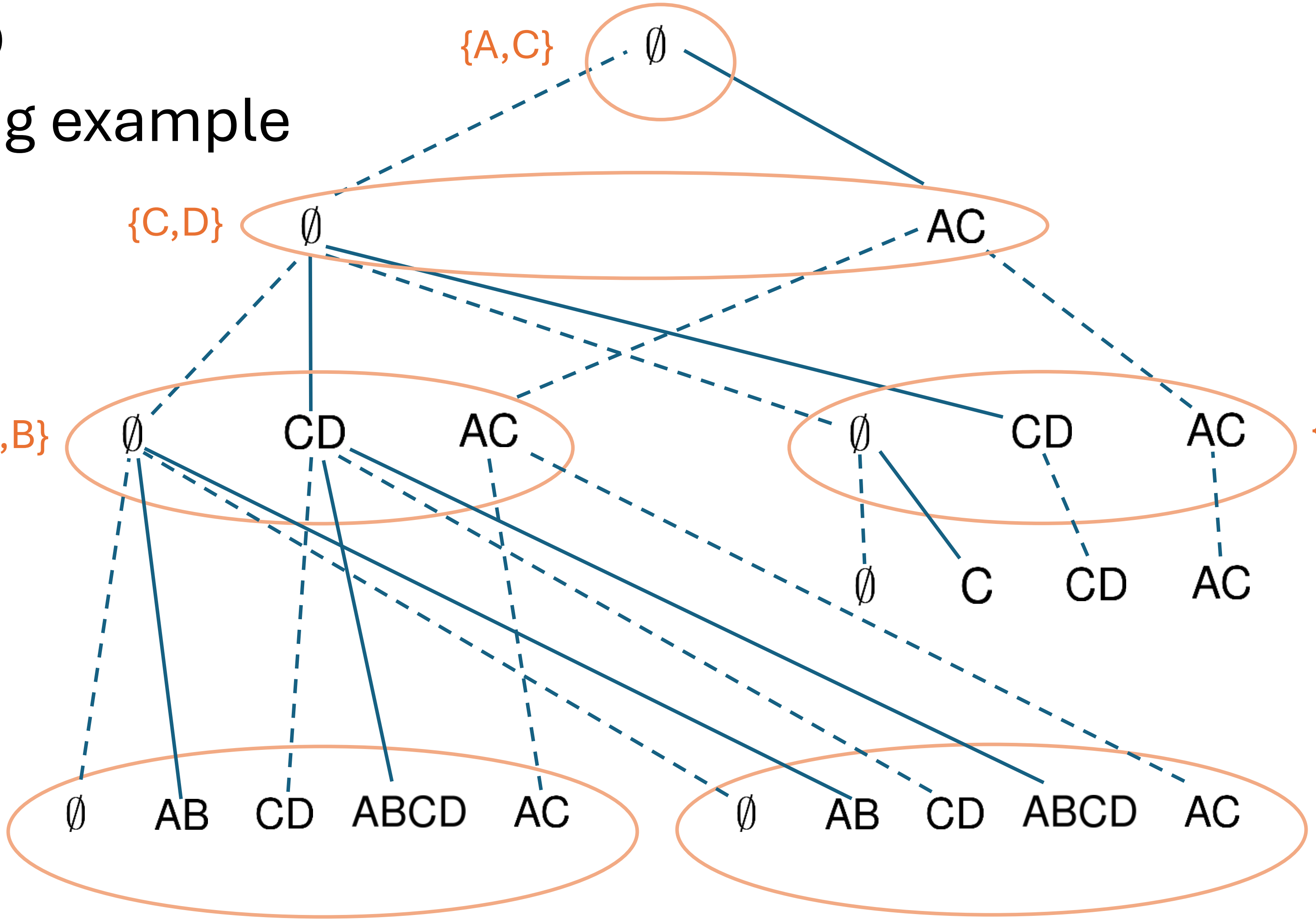
Duplication of states creates some overhead, but this will be offset by smaller width of layers.



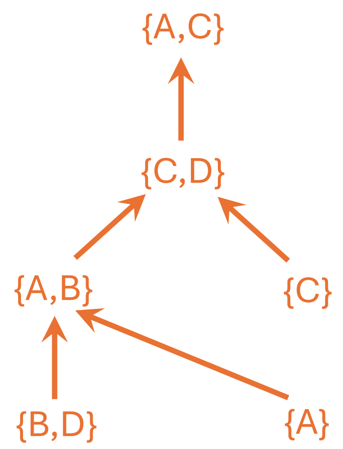
Carnegie Mellon University
Tepper School of Business

Nonserial DD

For set packing example



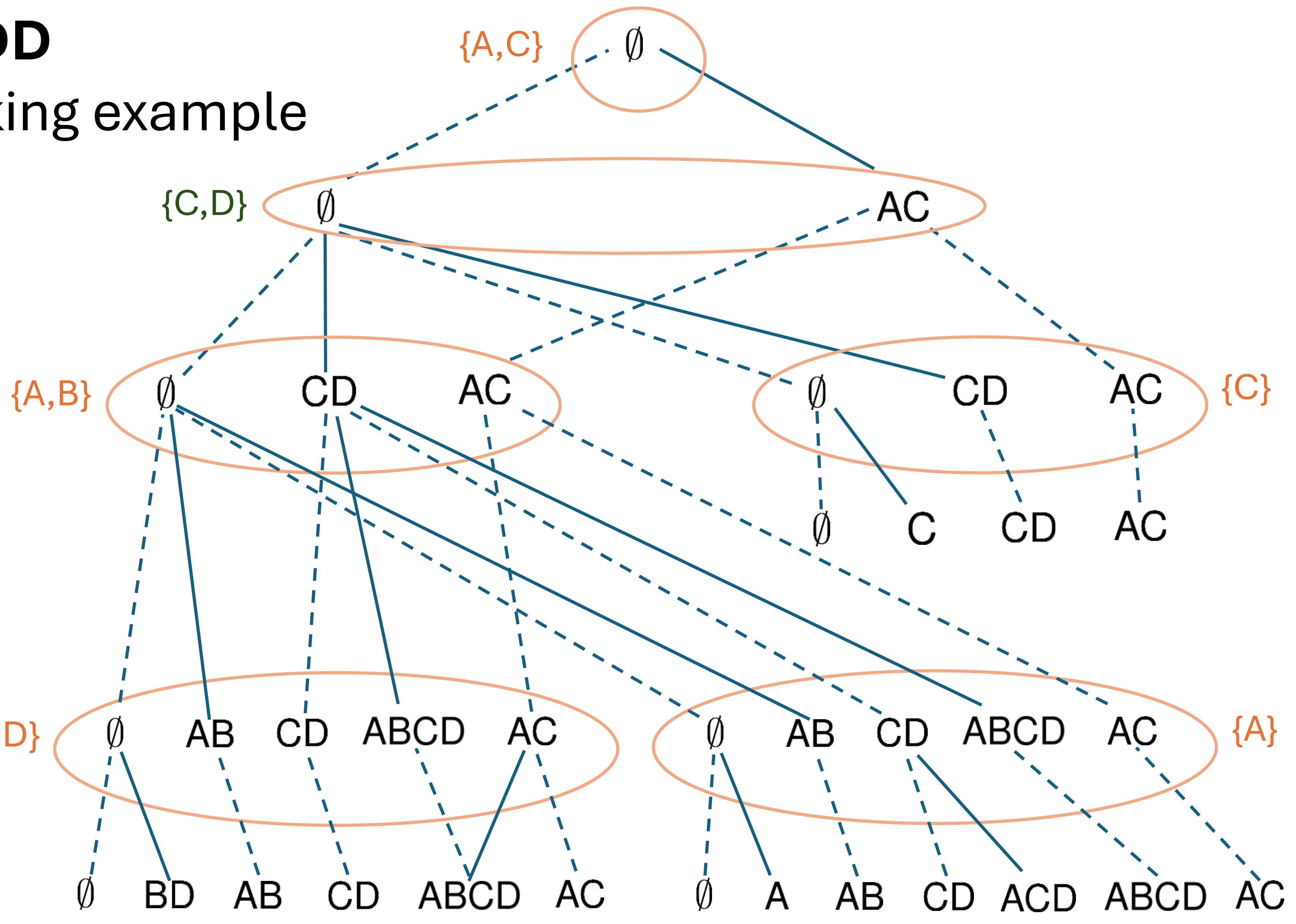
Decide whether to select set {A,B}

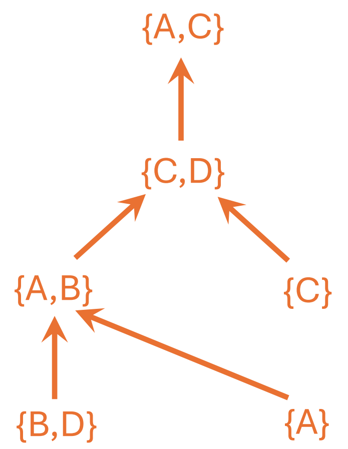


Decide whether to select set {C}

Nonserial DD

For set packing example





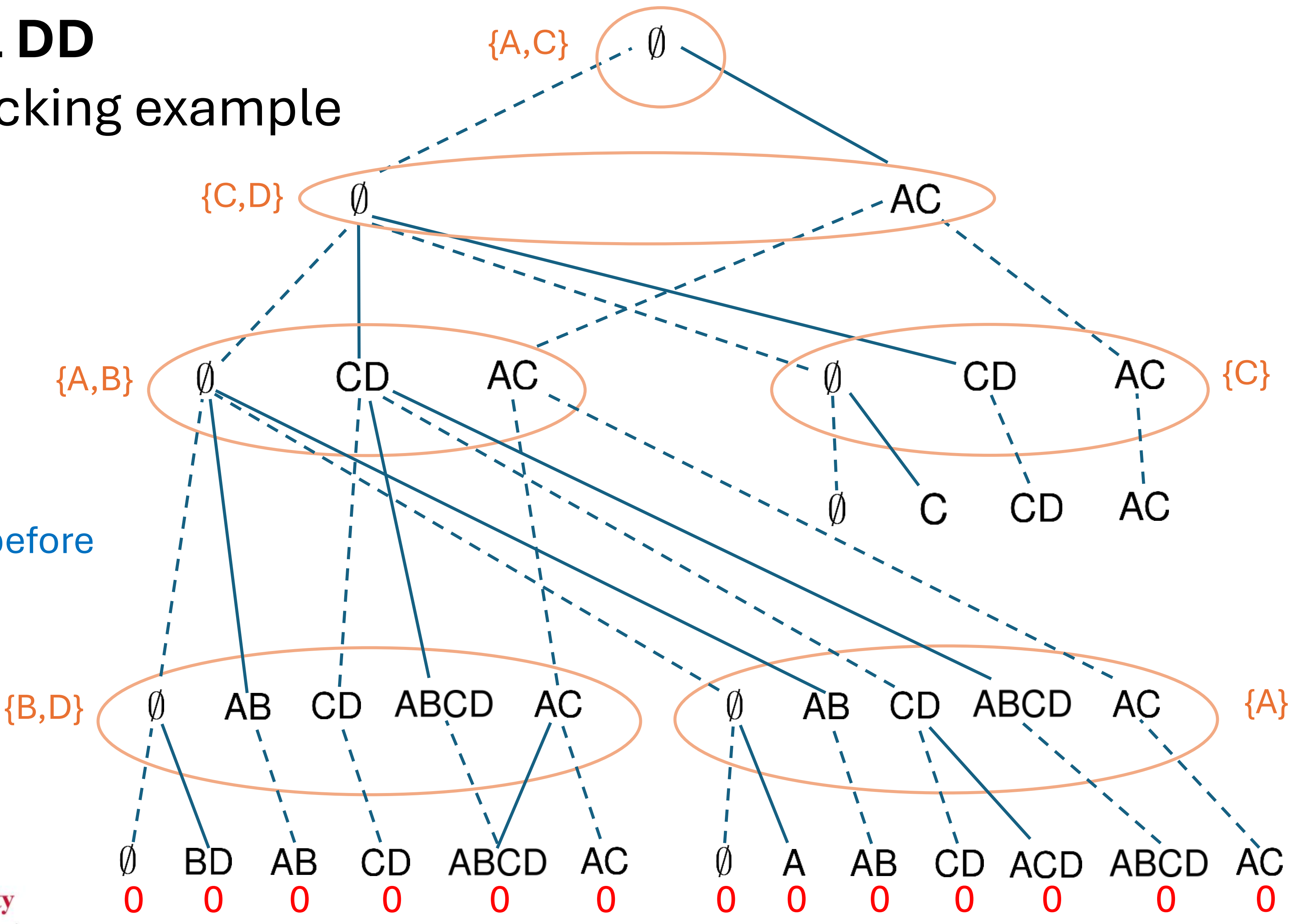
Decide whether to select set {B,D}

Decide whether to select set {A}

DD has 36 nodes

Nonserial DD

For set packing example

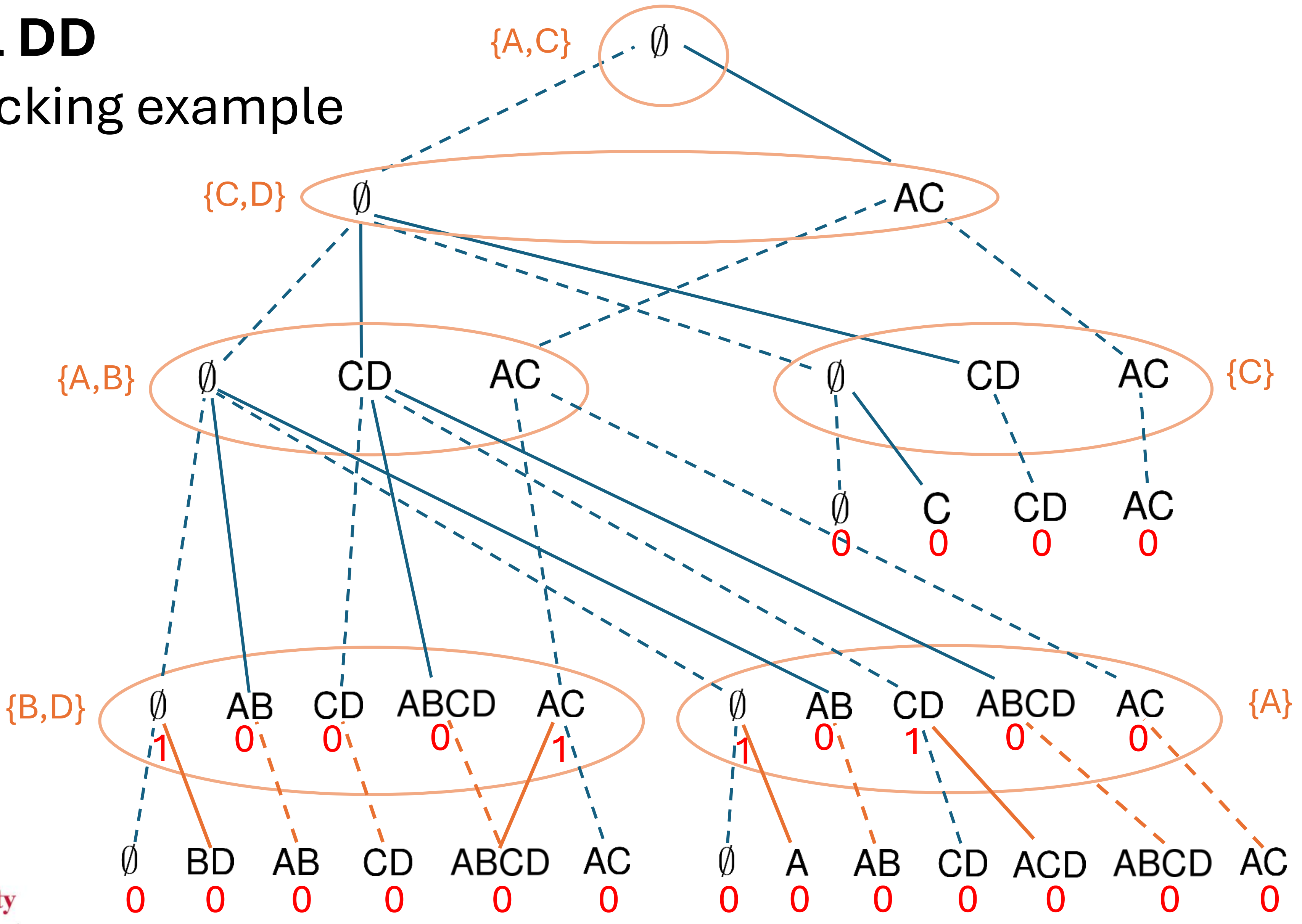


Evaluate the DD bottom-up as before

Carnegie Mellon University
Tepper School of Business

Nonserial DD

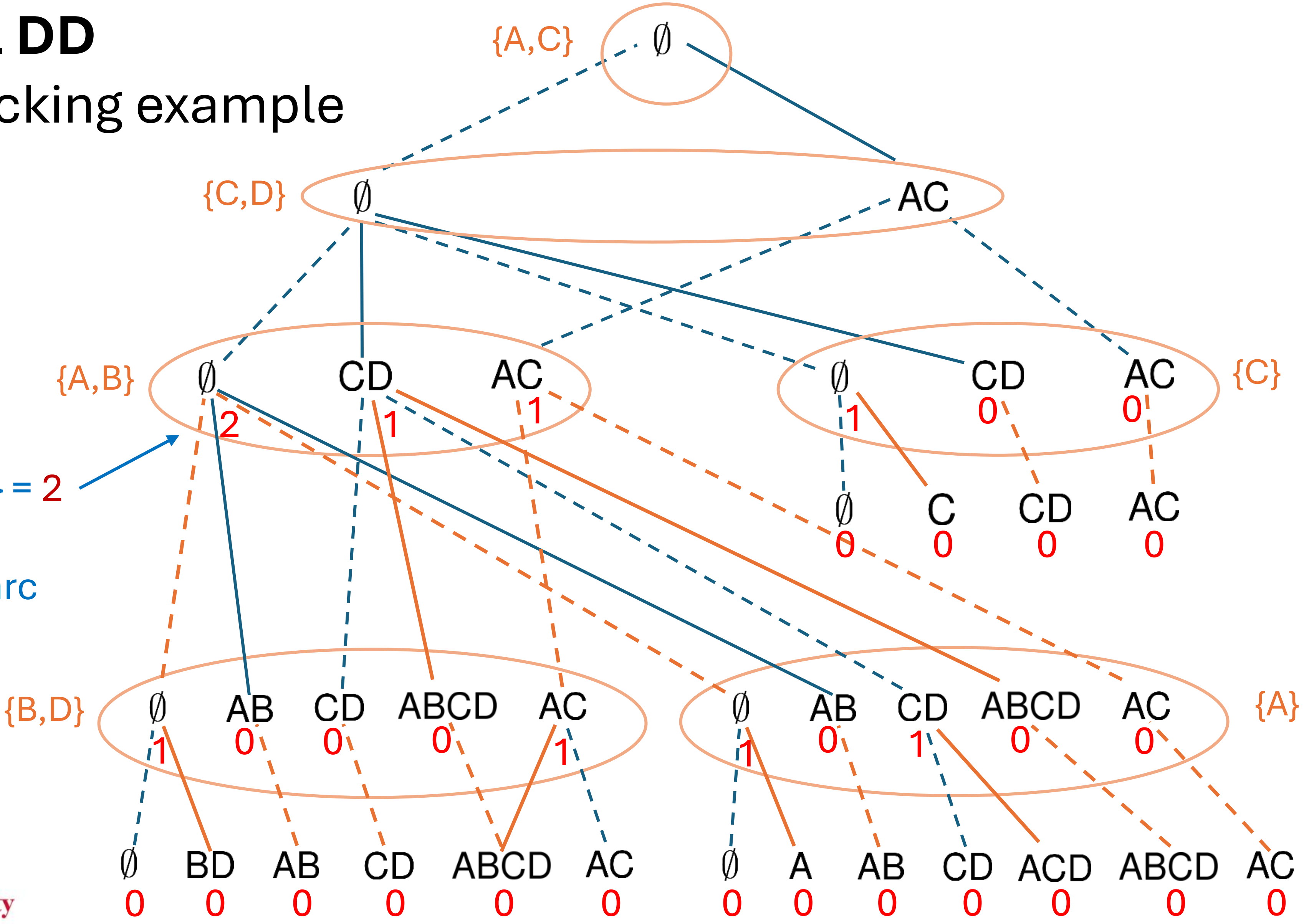
For set packing example



Carnegie Mellon University
Tepper School of Business

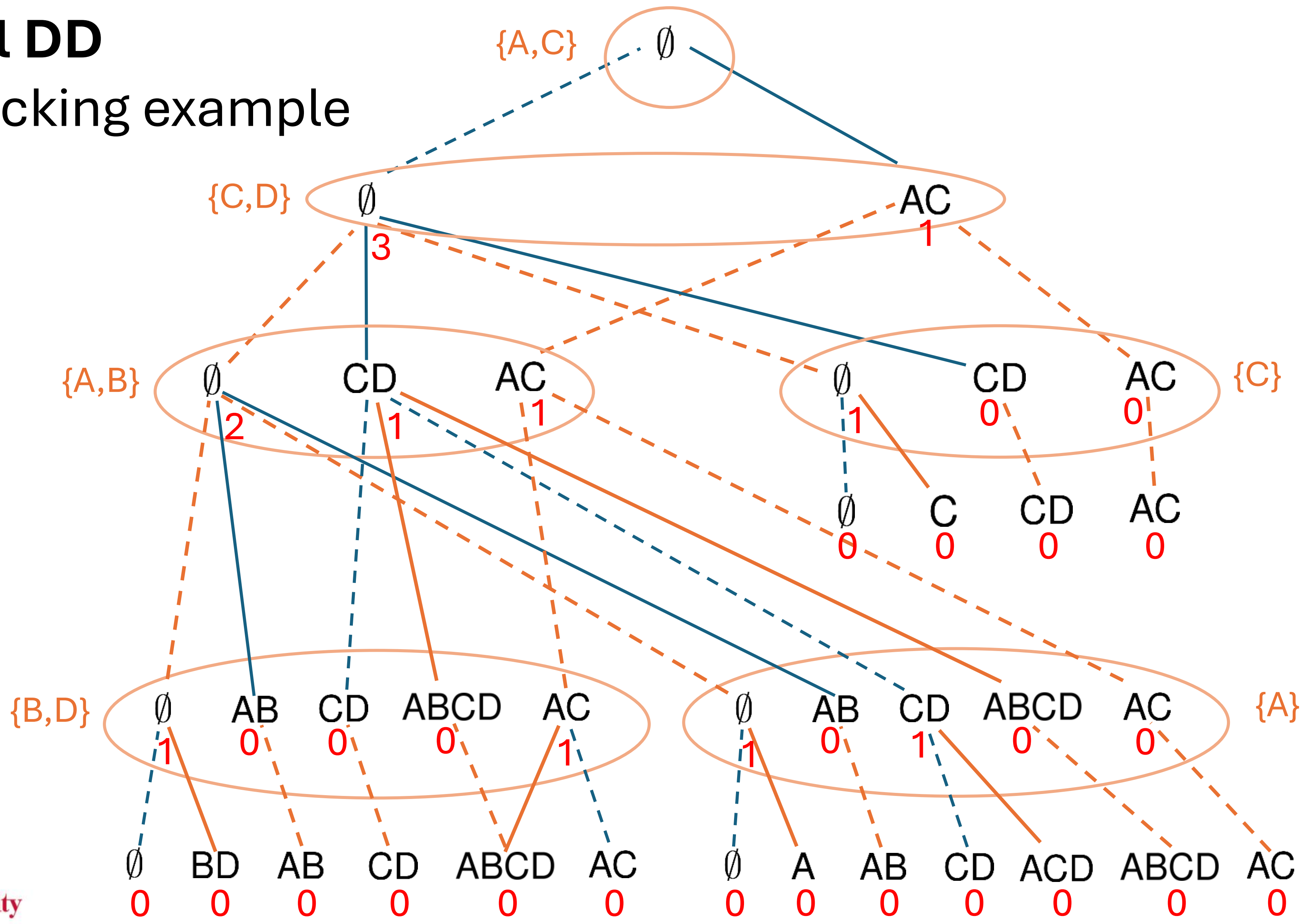
Nonserial DD

For set packing example



max {1+1, 0+0+1} = 2
Outgoing 1-arcs counted as one arc (as in and-or DD)

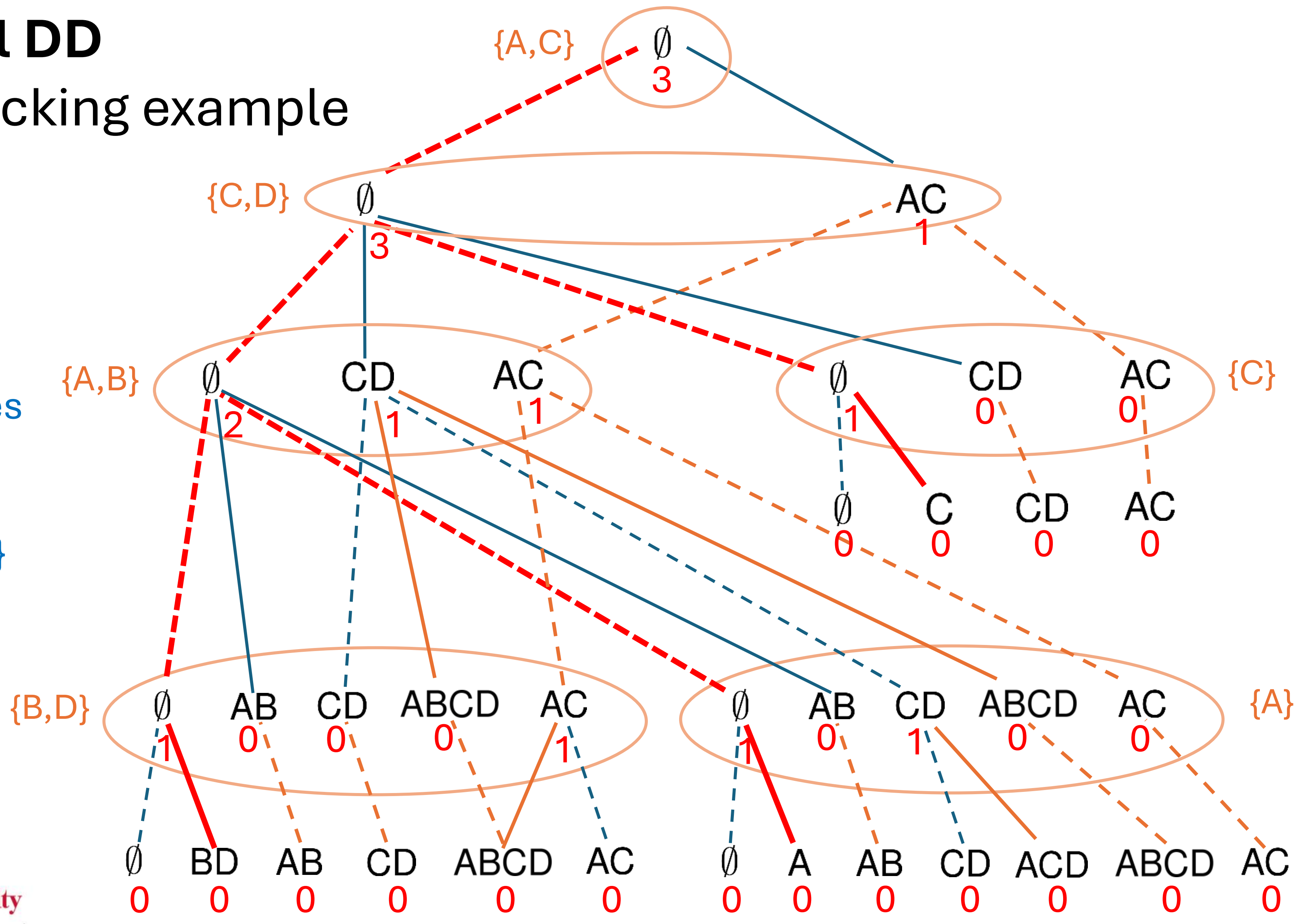
Nonserial DD
For set packing example



Nonserial DD

For set packing example

Trace **tree** of optimal choices to find optimal solution
{C} + {A} + {B,D}

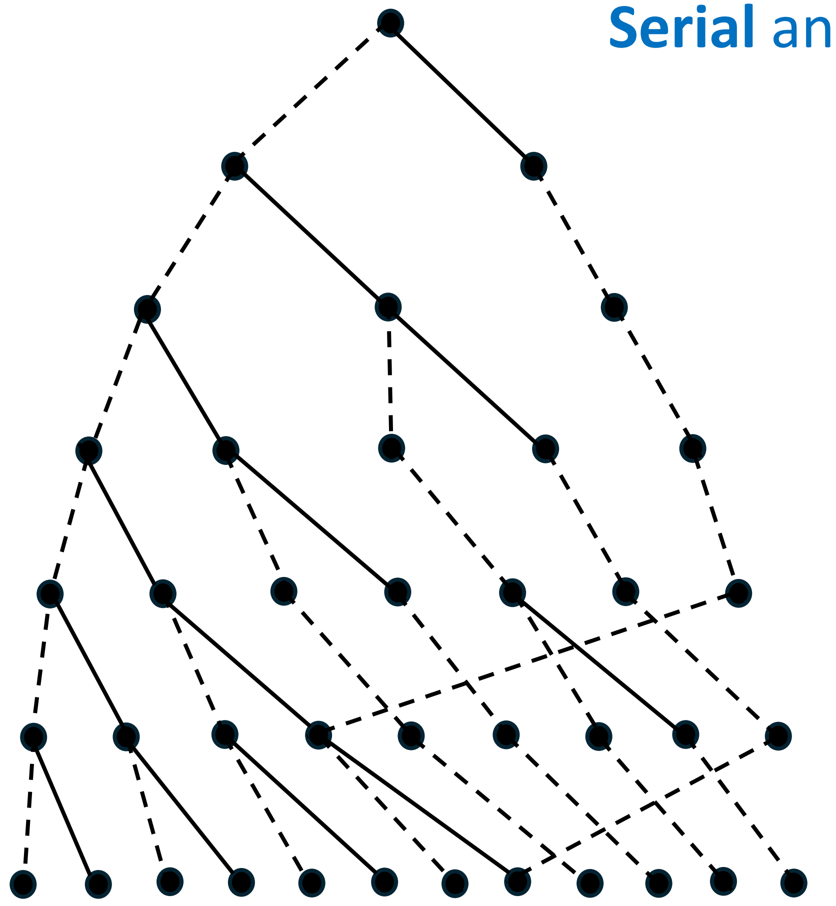


Carnegie Mellon University
Tepper School of Business

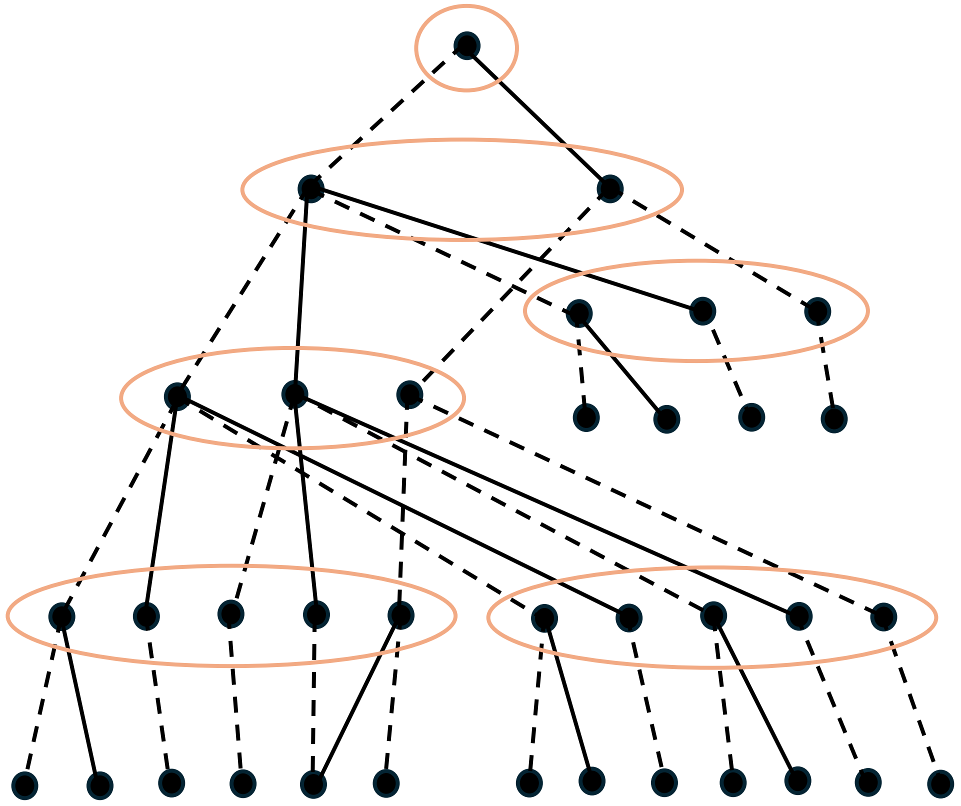
Nonserial DD
For set packing example

Difference can be **much greater** in larger instances.

Serial and **nonserial** DDs



39 nodes

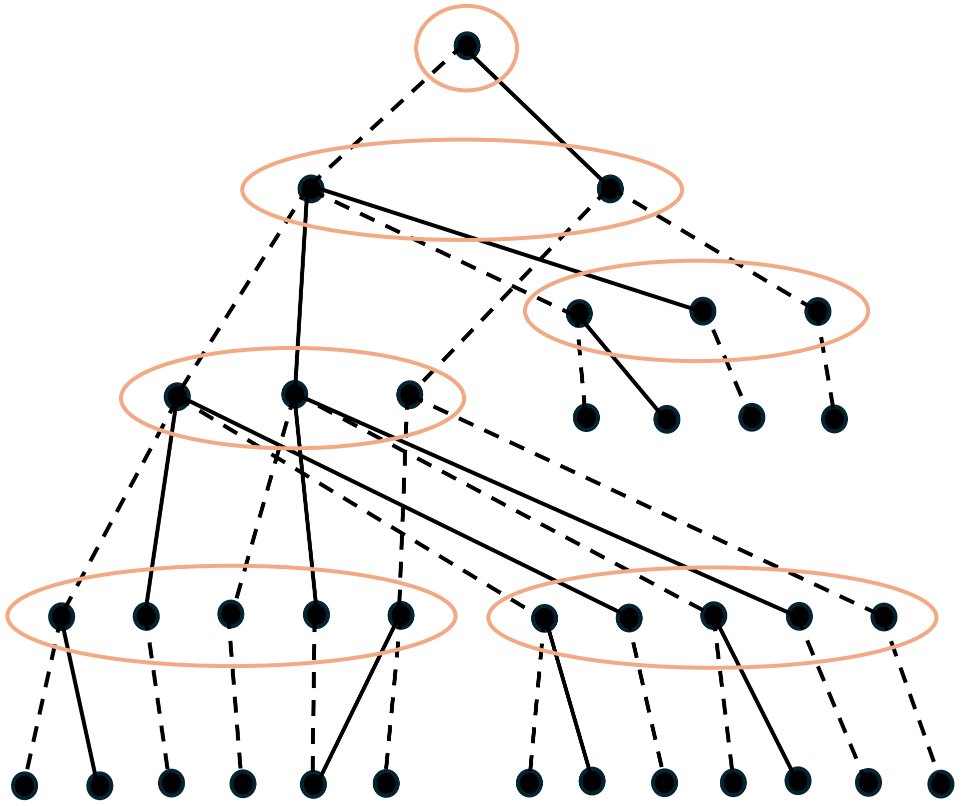


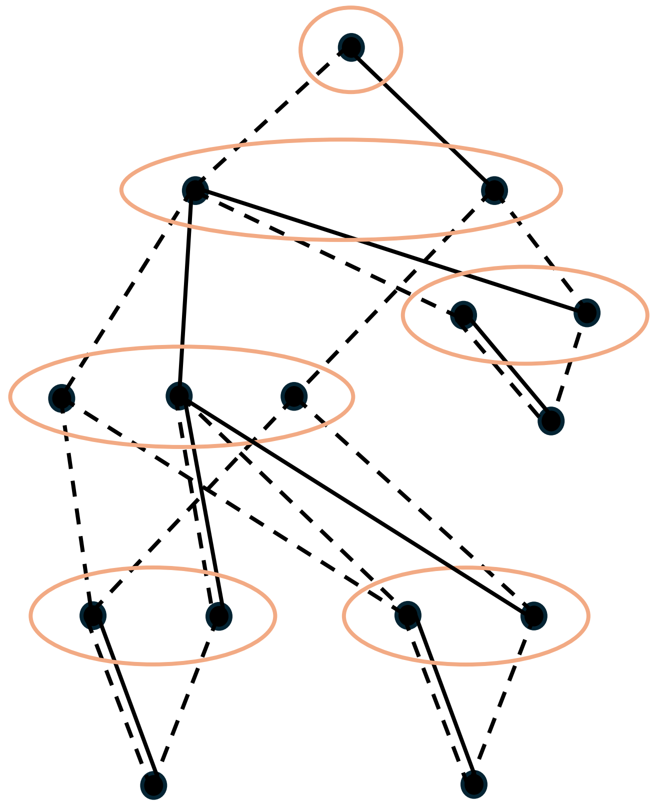
36 nodes

Carnegie Mellon University
Tepper School of Business

Nonserial DD
For set packing example

Original and **reduced** nonserial DDs





36 nodes

15 nodes

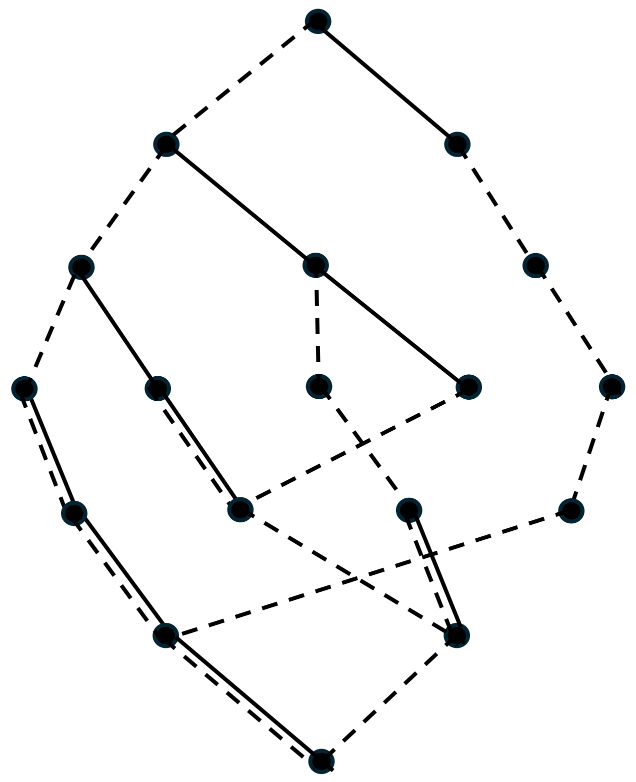
Carnegie Mellon University
Tepper School of Business

Nonserial DD

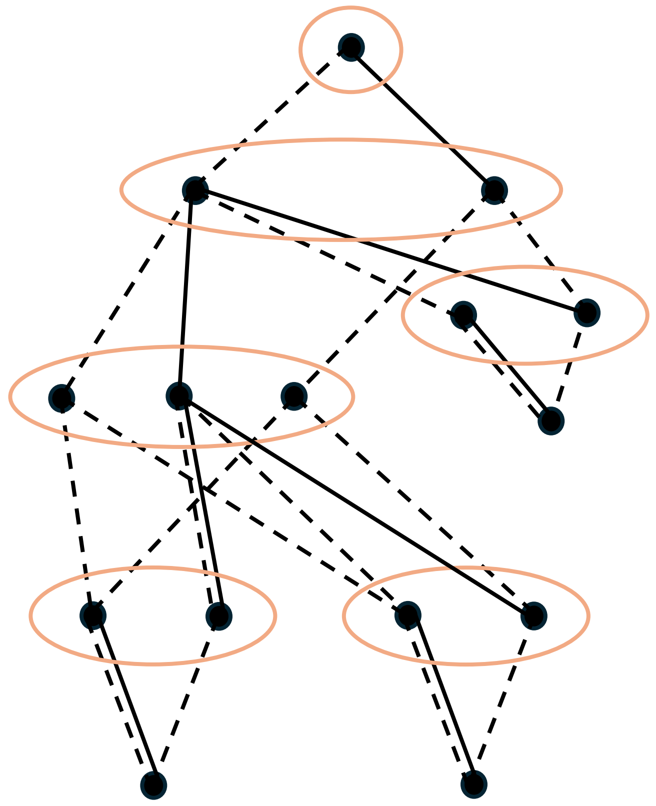
For set packing example

Reduced **serial** and **nonserial** DDs

Difference can be **much greater** in larger instances.



18 nodes



15 nodes

Carnegie Mellon University
Tepper School of Business

Nonserial DDs
Computational experiments

Compare size of non-reduced **serial** and **nonserial DDs** for randomly generated **set packing** instances of various treewidths.

Use **min-degree ordering** for serial and nonserial DDs, as it benefits both.

Let each element occur in a given set with **probability *p***.

Discard random instances with a **disconnected** dependency graph.

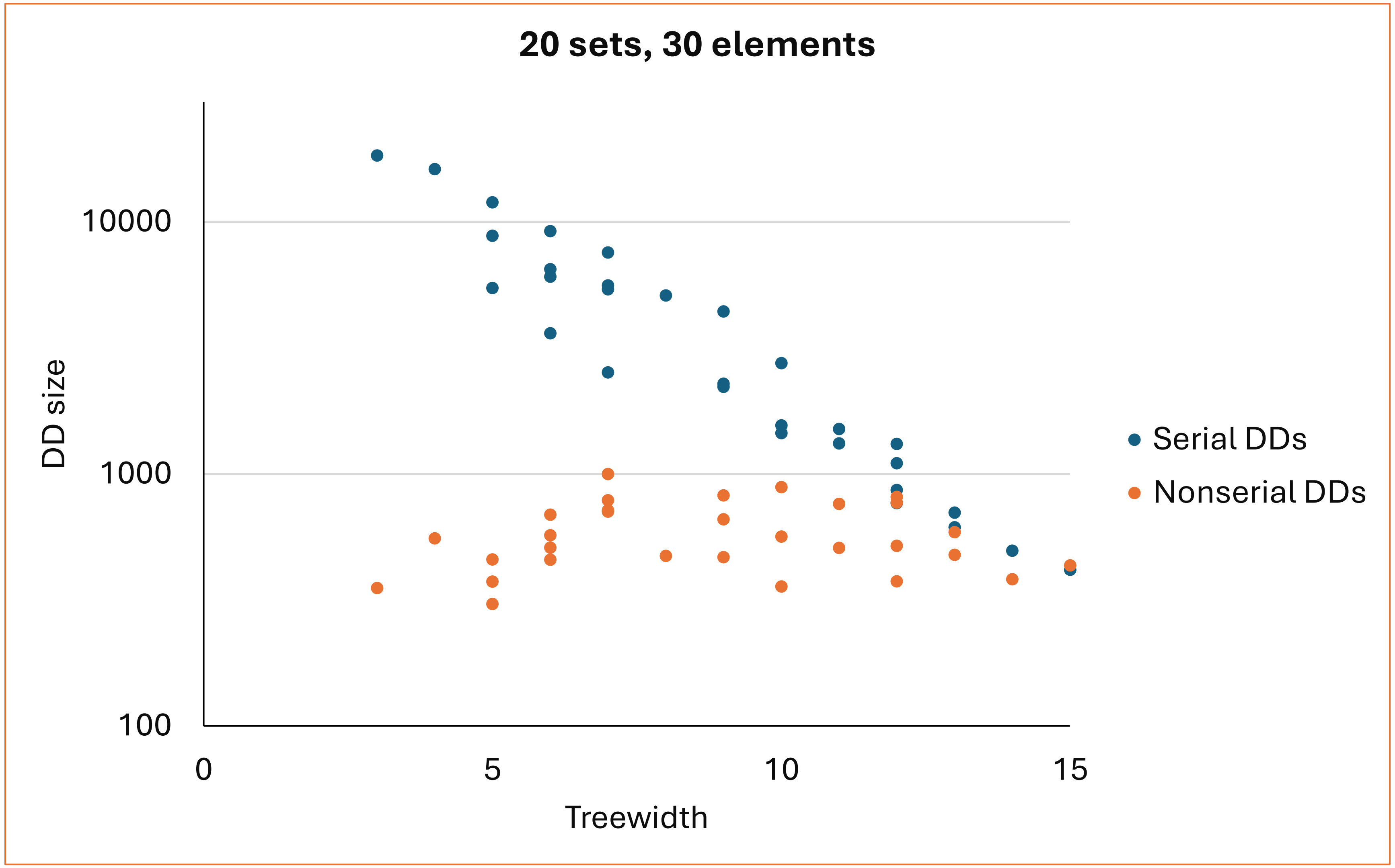
Use smaller **values of *p*** to get smaller **treewidths**.

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Each instance is represented by **two data points**.

Instances with **many elements per set** are **easier to solve** due to fewer feasible solutions.



Average 2.4-6 elements/set

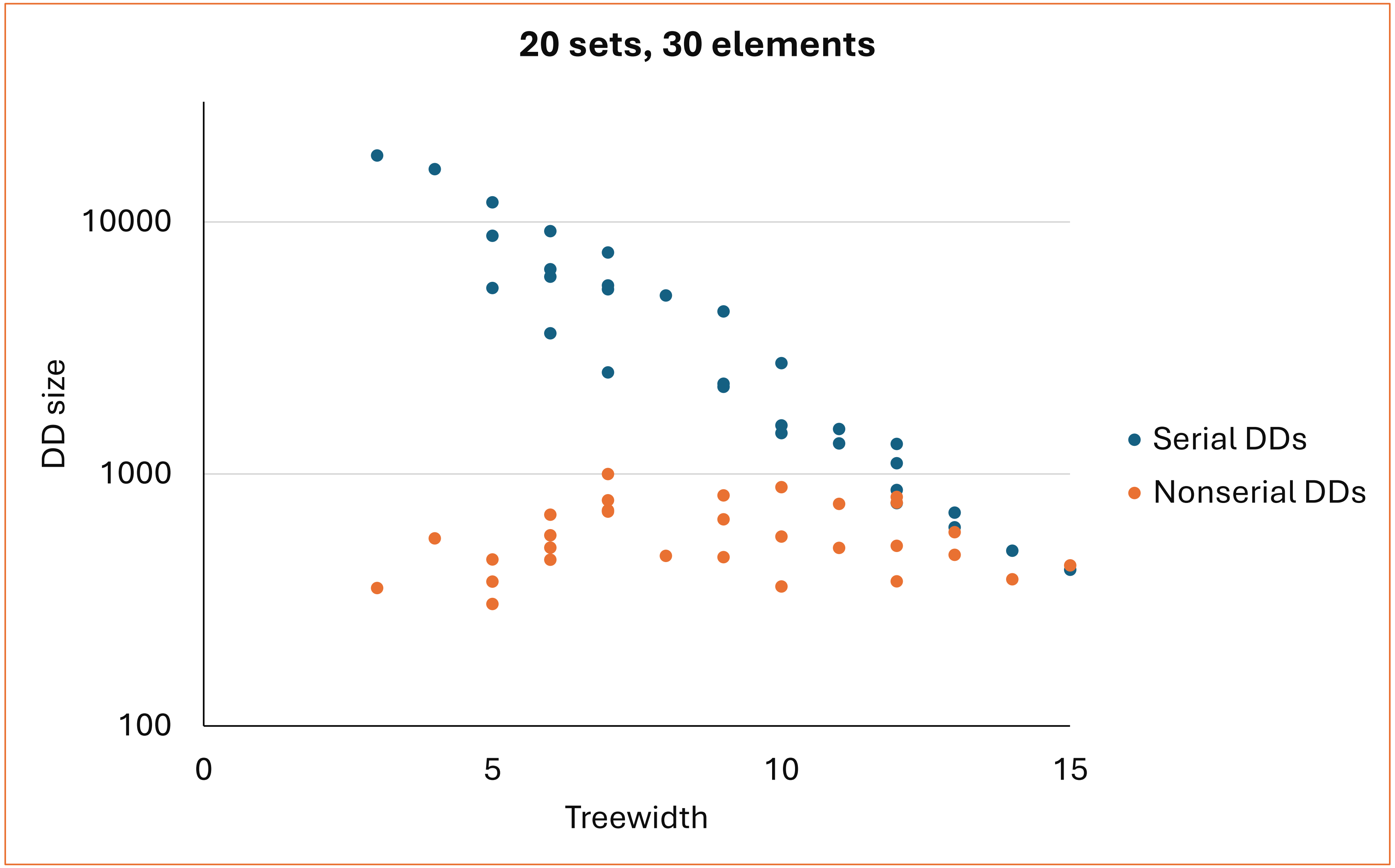
Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Smaller bandwidths result in **much larger serial DDs** (instances are harder).

Nonserial DD size is fairly **constant.**

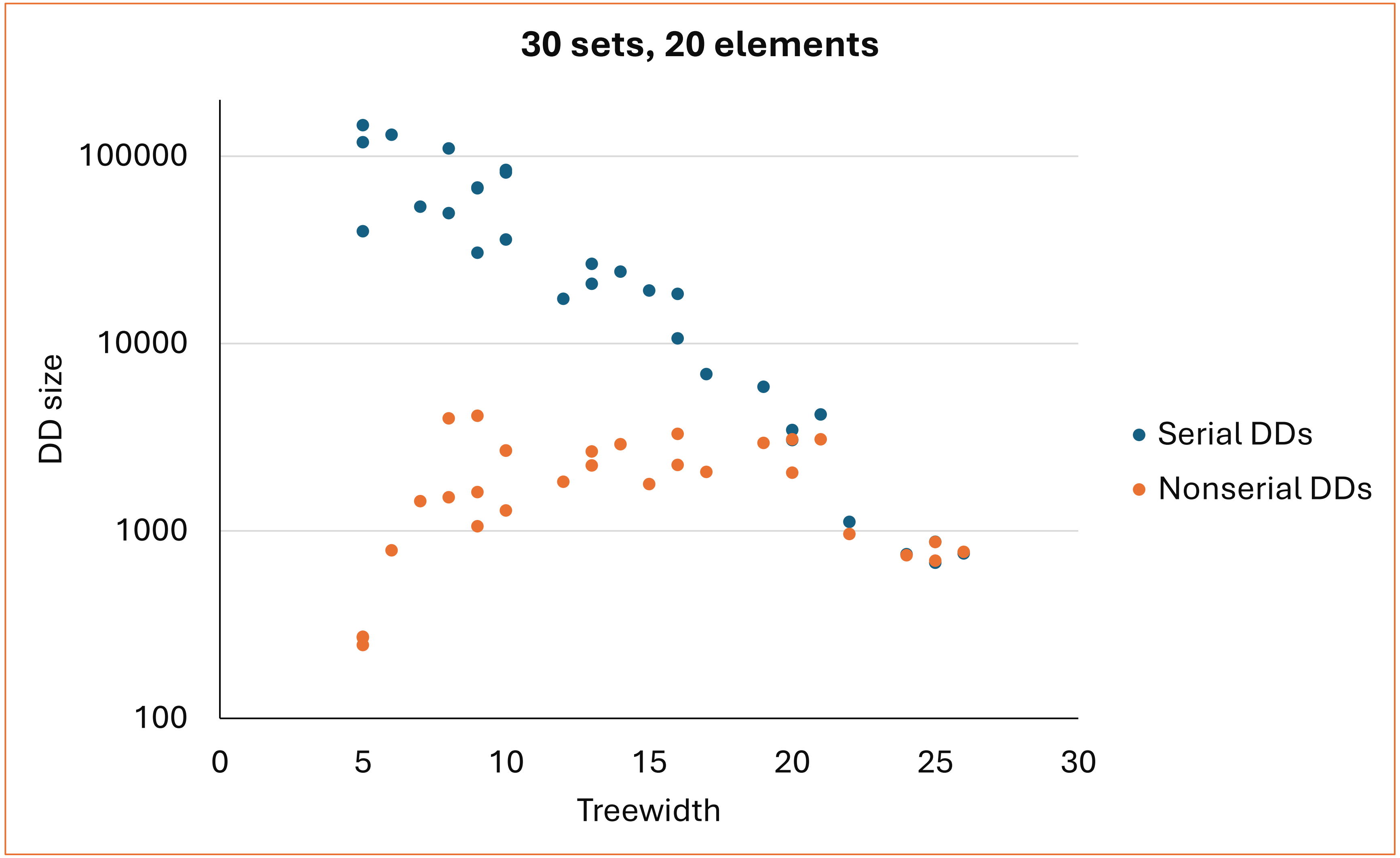
Nonserial DD's exploitation of small bandwidth **offsets** greater difficulty of the instance.



Average 2.4-6 elements/set

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial**
DD size vs treewidth



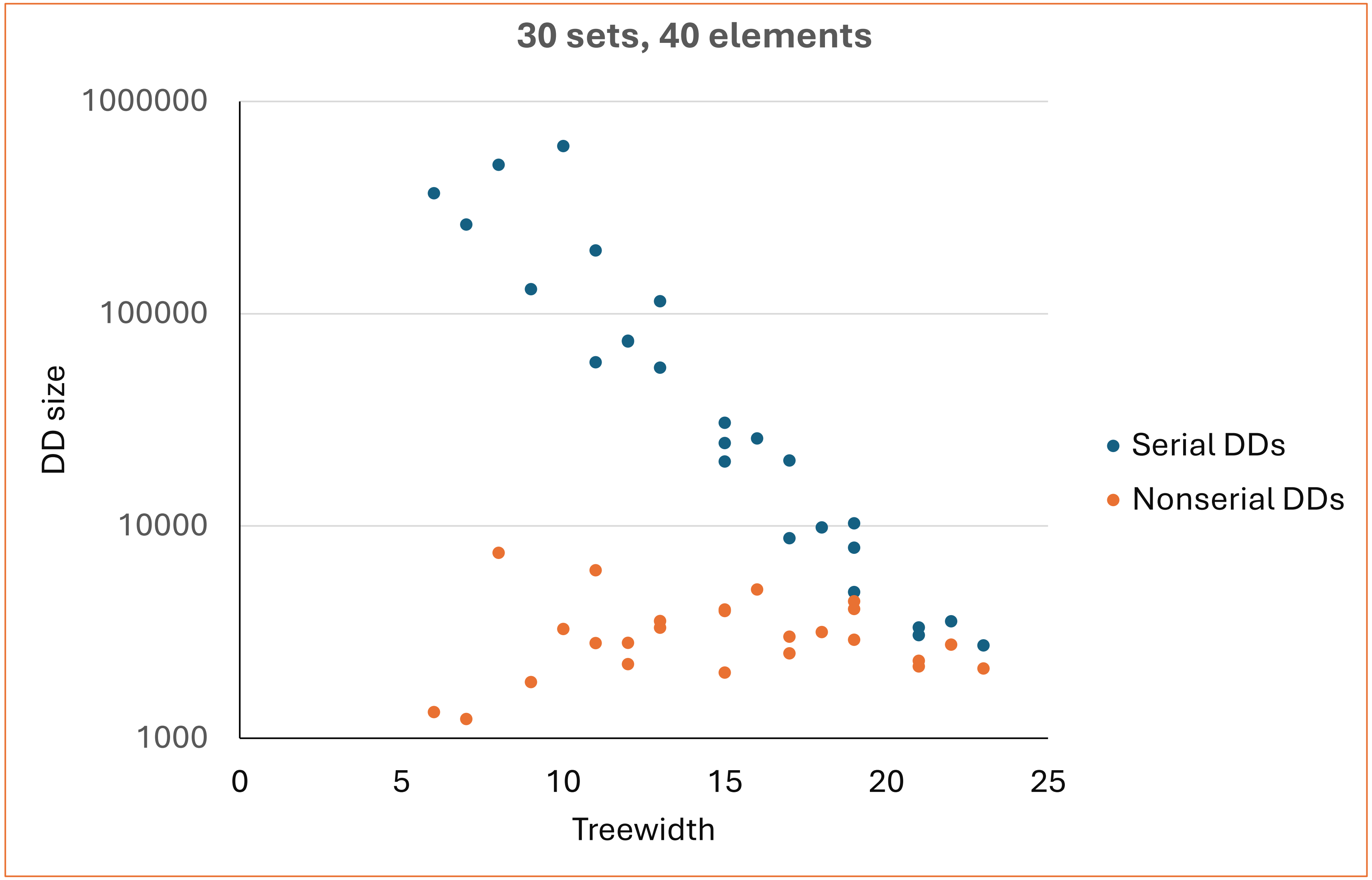
Similar pattern,
except for inverted-U shape of nonserial data points

Average 1.6-6 elements/set

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Larger DDs, but otherwise **similar pattern**



Average 2.4-6 elements/set

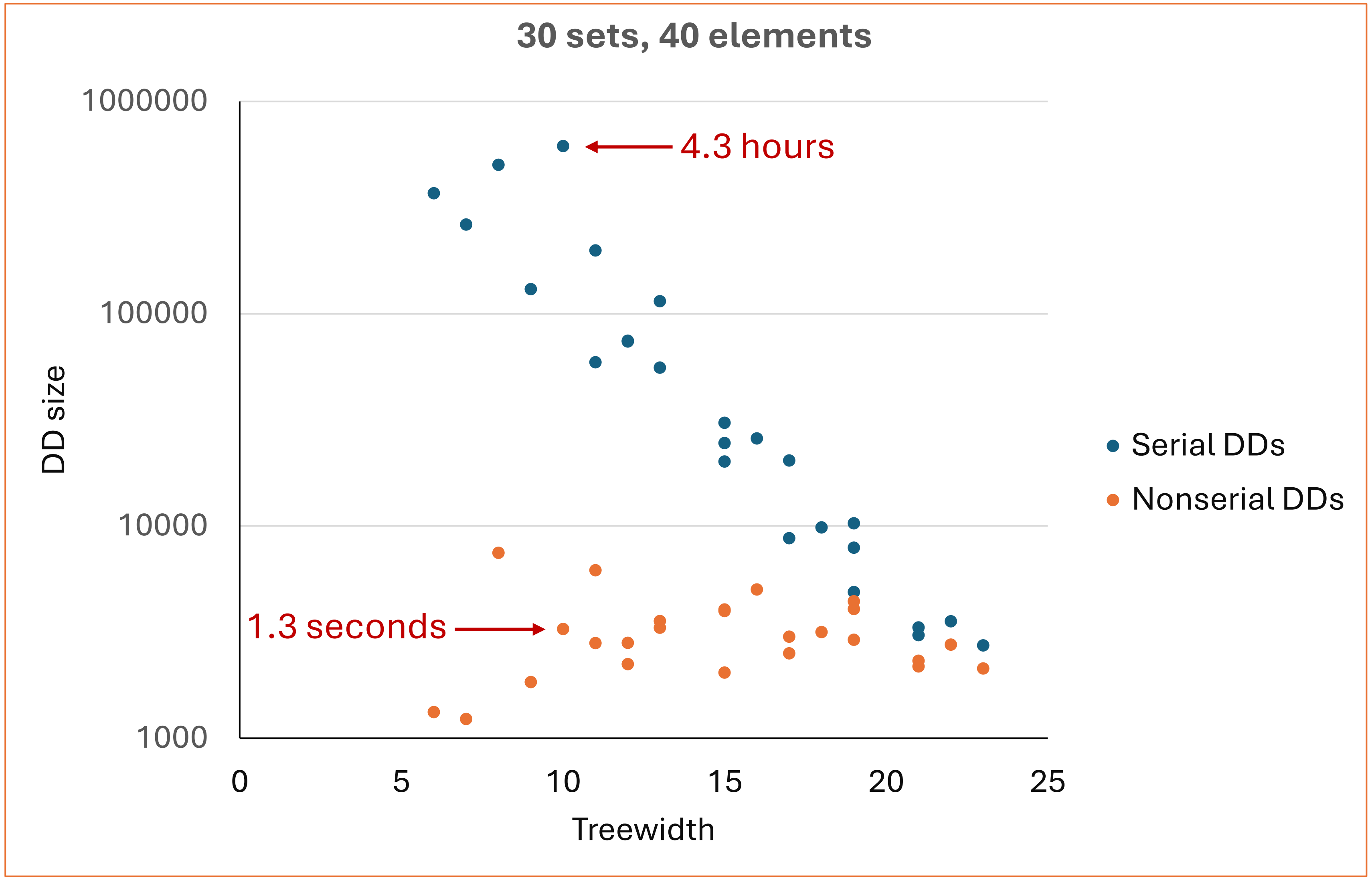
Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Difference in **compile time** is even more dramatic than DD size.

Compile time is roughly **quadratic** in max **layer** size.

Serial DD layers are much **larger**.

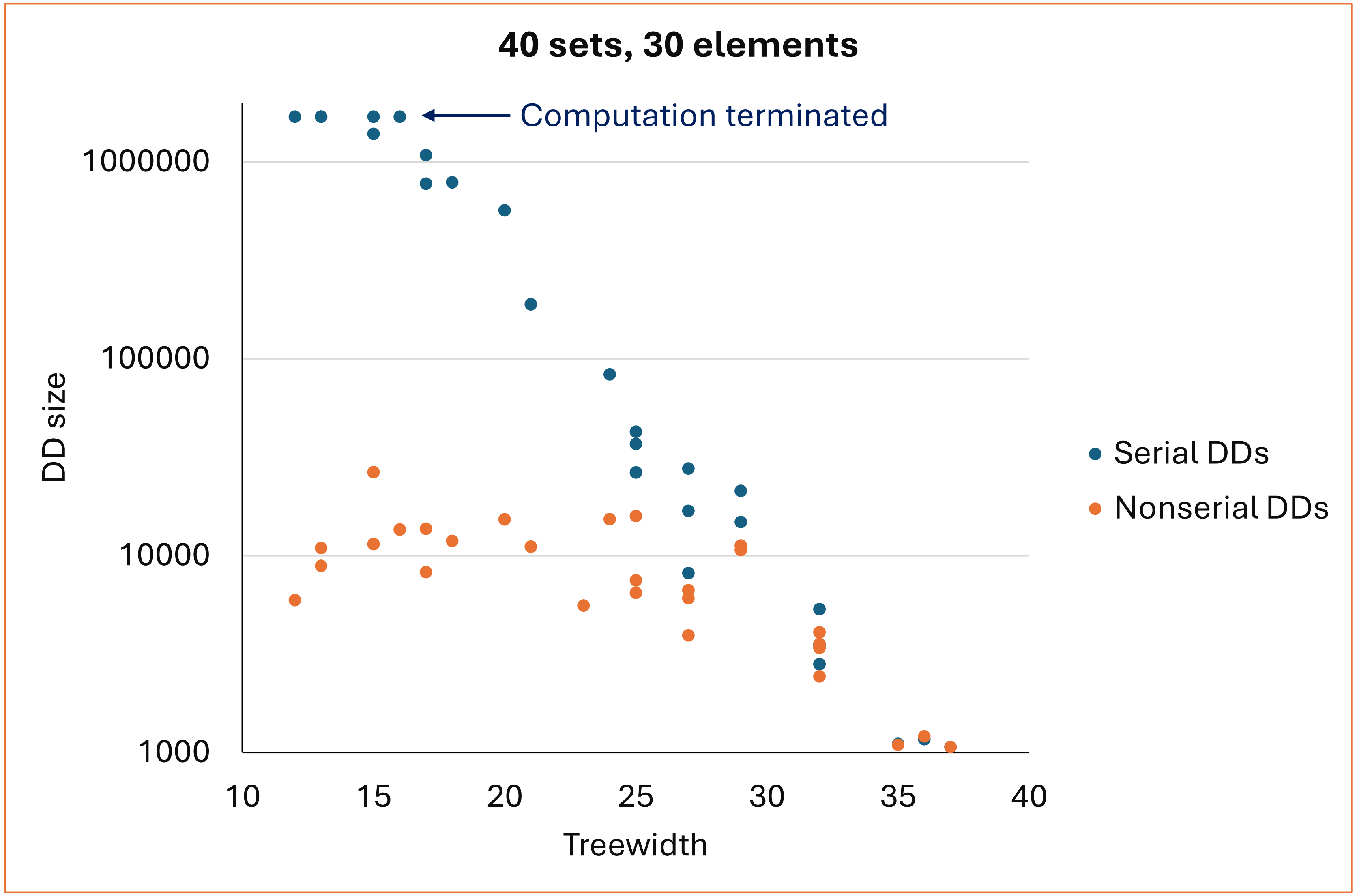


Average 2.4-6 elements/set

Serial and **nonserial** DD size vs treewidth

Some **serial** DDs are too large to build.

Nonserial DD size again levels off with smaller treewidths

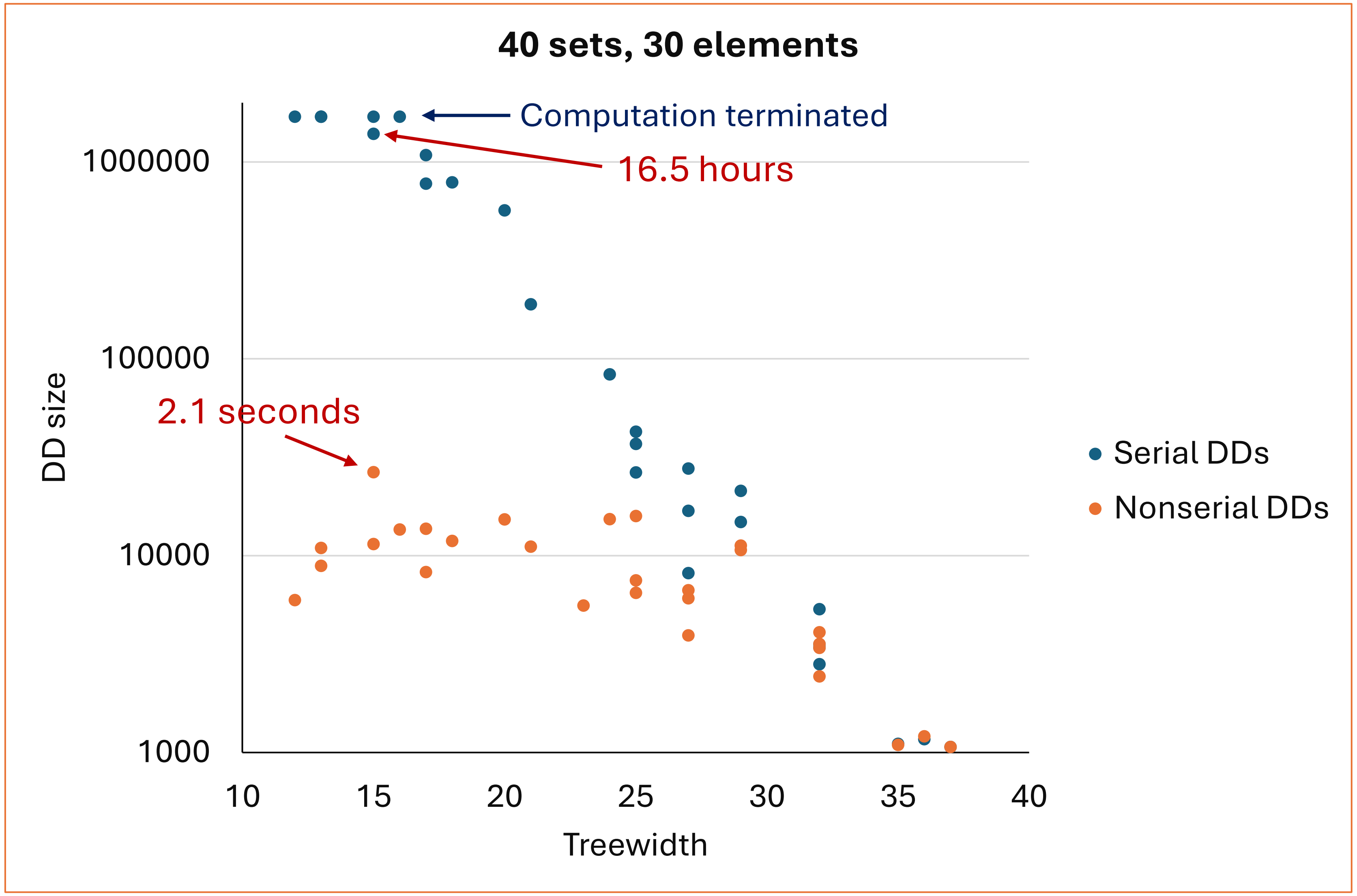


Average 2.4-9 elements/set

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Compile time advantage of nonserial DD is again even greater than **size** advantage.

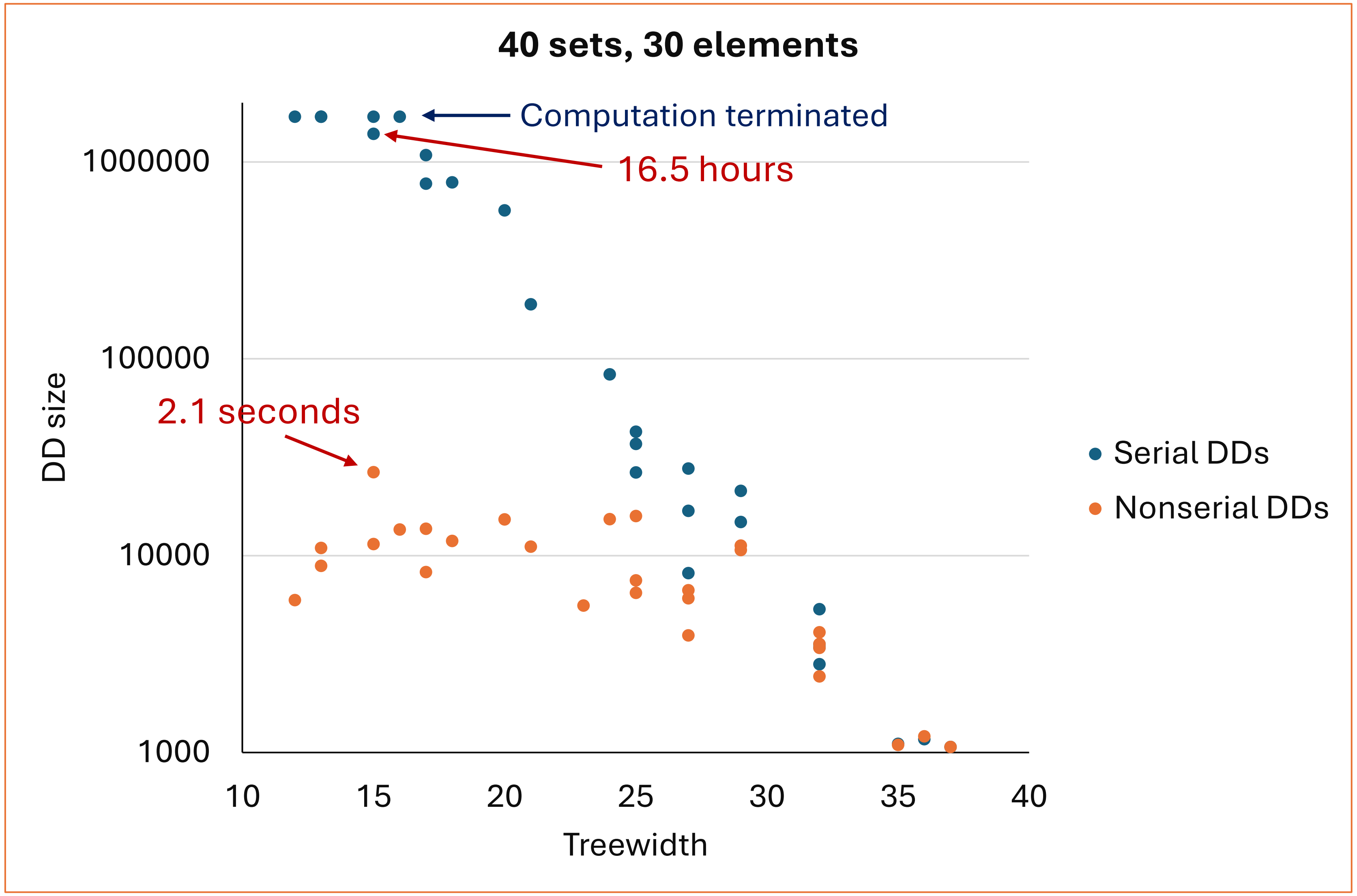


Average 2.4-9 elements/set

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth

Compile time advantage of nonserial DD is again even greater than **size** advantage.



Average 2.4-9 elements/set

Nonserial DDs
Computational experiments

Preliminary results for **0-1 programming**

Use sparser coefficient matrices to get smaller treewidths.

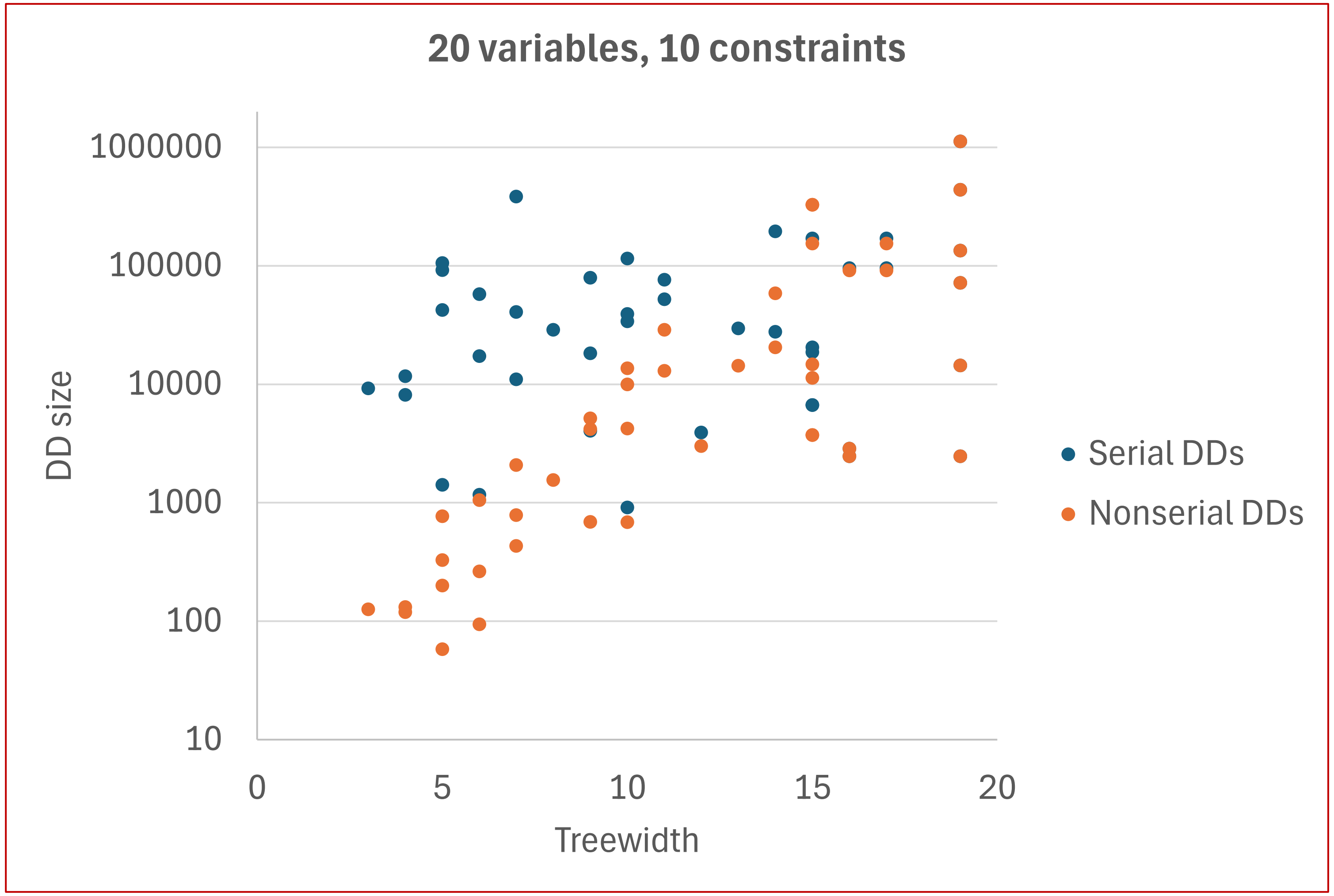
Nobody suggests solving 0-1 problems this way. Use a MIP solver.

But... 0-1 inequality constraints may be a **subset** of the problem

and... results are the same for **any set of constraints** (linear or nonlinear) in which each constraint has the same feasible solutions.

Serial and **nonserial** DD size vs treewidth 0-1 programming

Much scatter, because random instances vary widely in difficulty



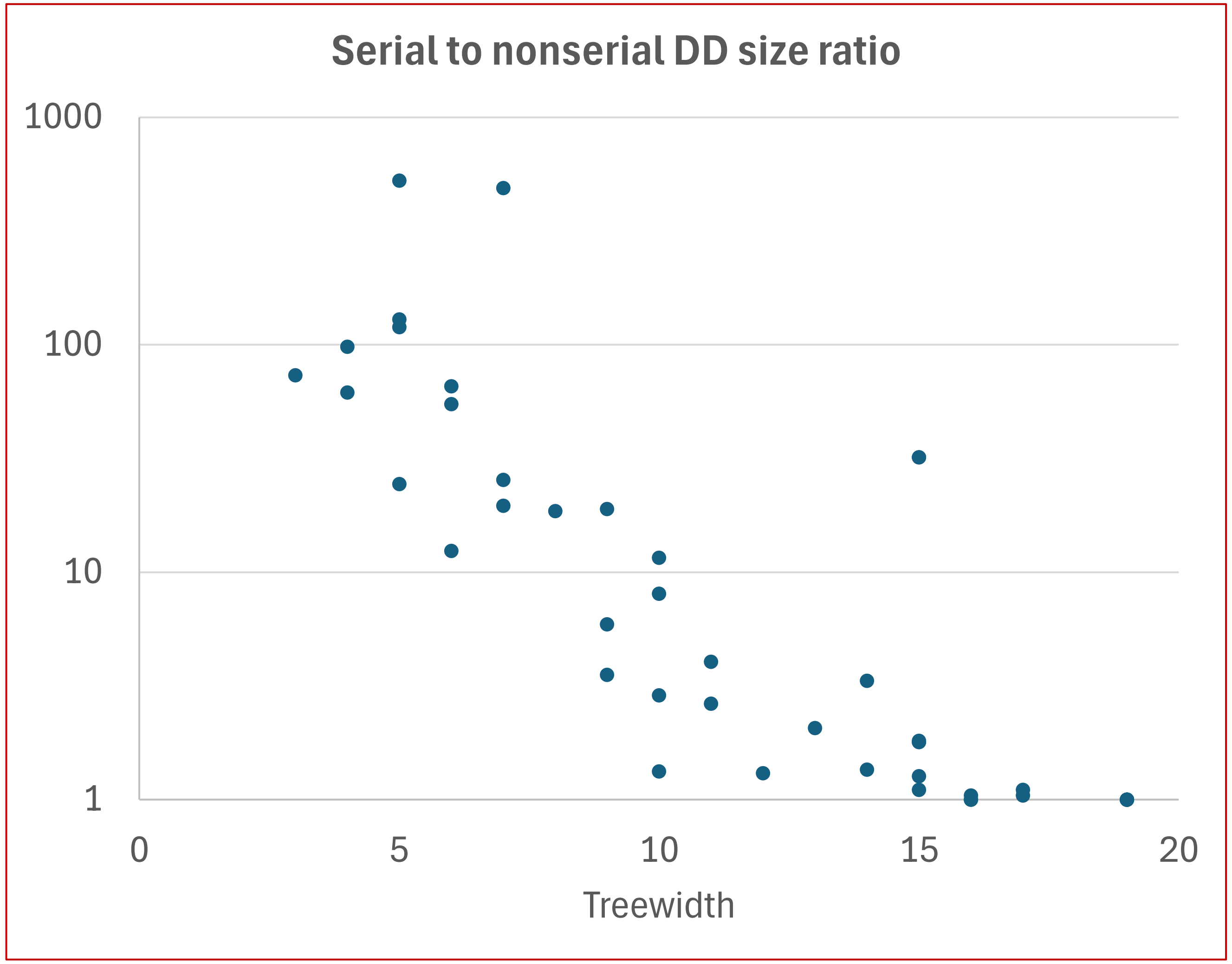
Average 2-15 nonzeros/row

Carnegie Mellon University
Tepper School of Business

Serial and **nonserial** DD size vs treewidth 0-1 programming

Pattern is clearer when plotting **ratio** of serial to nonserial DD size.

Nonserial DDs impose 20% overhead when there is **no** decoupling (treewidth = # variables)



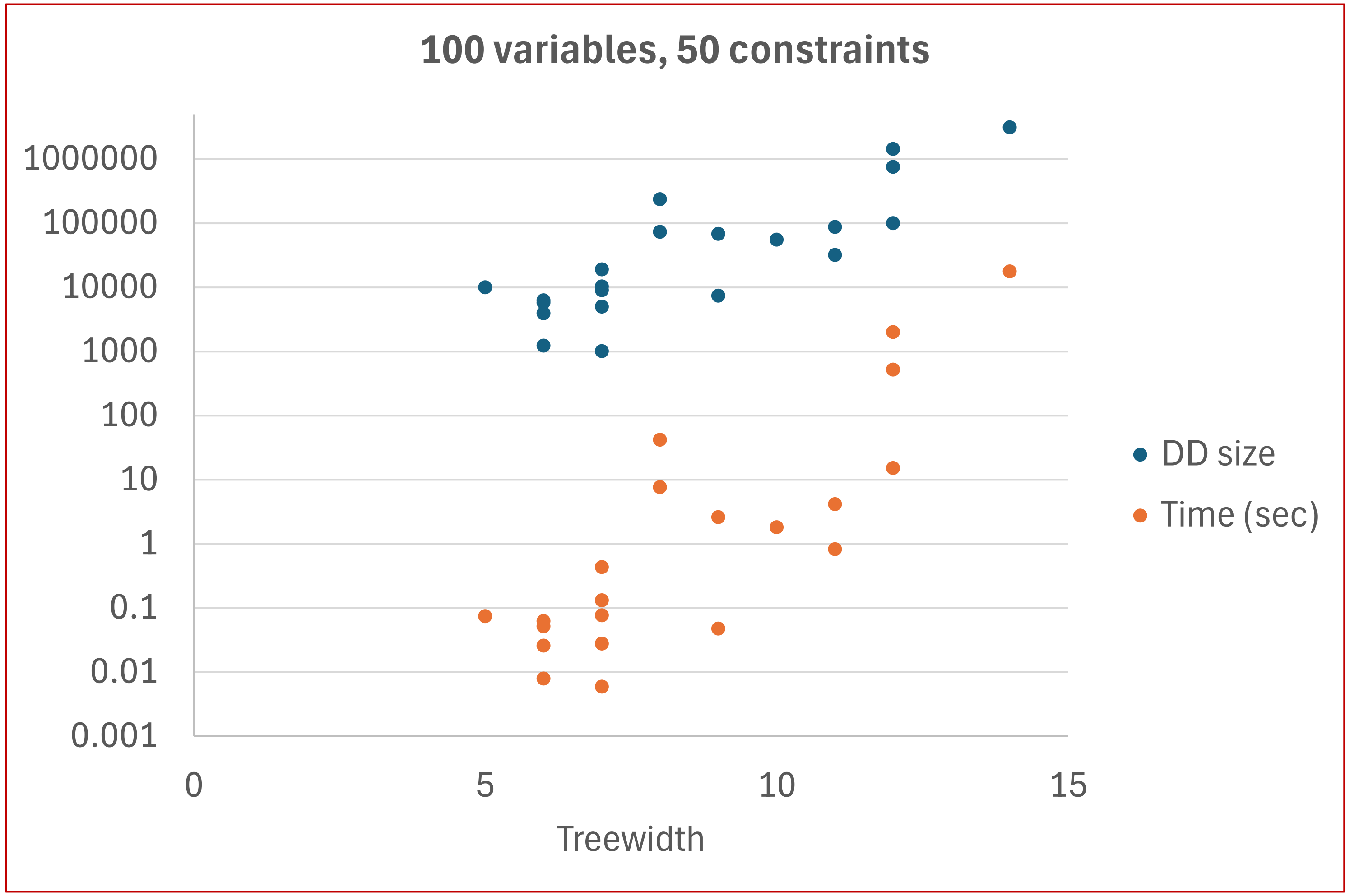
Average 2-15 nonzeros/row

Carnegie Mellon University
Tepper School of Business

Nonserial DDs

DD size & build time

0-1 programming



Serial DDs **too large to build** even for smallest treewidths.

Nonserial DD build time is **very small** for treewidth ≤ 12

Carnegie Mellon University
Tepper School of Business

Average 2-3 nonzeros/row

Conclusions

Set packing problem

Nonserial DDs are **very helpful** when you **need them,** and are **not helpful** when you **don’t need them.**

Problem class containing 0-1 programming

Nonserial DDs radically **smaller** than serial DDs, easy to build when treewidth $\le$ 12 or so.

Tentative conclusions

We should **always use nonserial DDs** in DD applications.

There is only a **small computational overhead** for doing so.

There are **enormous computational benefits** when treewidth is limited.

All DD technologies easily **generalize** to the nonserial case (reduction, relaxation, restriction, flow models)

Congratulations!

You survived 133 slides!