ProCount:
Weighted Projected Model Counting with Graded Project-Join Trees

Jeffrey M. Dudek - Vu H. N. Phan - Moshe Y. Vardi

Rice University

July 8, 2021

RICE

SAT 2021

In This Talk

Problem: Weighted Projected Model Counting $\#_X \exists_Y \, \varphi(X, Y)$

Applications in planning, formal verification, and infrastructure reliability

Our Solution: Two-phase algorithm based on graded project-join trees

1. Planning: Use ***tree decompositions*** to build a graded project-join tree
2. Execution: Process graded project-join tree with **algebraic decision diagrams (*ADDs)*** to get the count

Experiments: Our tool ProCount is fastest on 34% of solved benchmarks

https://github.com/vardigroup/DPMC

The Problem: Projected Model Counting

Satisfiability (SAT)		Projected Satisfiability (∃SAT)
Does φ have a solution?	=	Does $\exists Y.\varphi$ have a solution?
NP-Complete		NP-Complete

Polynomial Hierarchy (PH)

$\leq$ [Toda, 91]

Model Counting (#SAT)		Projected Model Counting (#∃SAT)
How many solutions does φ have?	[Zawadzki et al, 13] < Unless PH collapses	How many solutions does $\exists Y.\varphi$ have?
#P-Complete		$\#\mathrm{P}^{\mathrm{NP}[1]}$-Complete

φ is a CNF formula, $Y \subseteq \mathrm{Vars}(\varphi)$

Background: Projected Model Counting

Problem: Projected Model Counting

Input: A CNF formula $\varphi(X, Y)$ over disjoint variable sets X and Y

Output: $\#_X \exists_Y \, \varphi(X, Y)$

The number of $\vec{\boldsymbol{x}} \in 2^X$ s.t. there exists $\vec{\boldsymbol{y}} \in 2^Y$ where $\varphi(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) = 1$

Everything in this work generalizes to literal-weighted projected model counting.

Example:

$X = \{x_1, x_2\}$

$Y = \{y_1, y_2\}$

$\varphi(X, Y) = (x_1 \lor \neg x_2 \lor y_1) \land (x_1 \lor y_2) \land (\neg y_1 \lor \neg y_2)$

Solutions to $\exists Y. \varphi$ are: $(x_1 = 0,\ x_2 = 0)$, $(x_1 = 1,\ x_2 = 0)$, and $(x_1 = 1,\ x_2 = 1)$

Thus $\#_X \exists_Y \, \varphi(X, Y) = 3$

Background: Projected Model Counting

Problem: Projected Model Counting
Input: A CNF formula $\varphi(X, Y)$ over disjoint variable sets X and Y
Output: $\#_X \exists_Y \; \varphi(X, Y)$
The number of $\vec{\boldsymbol{x}} \in 2^X$ s.t. there exists $\vec{\boldsymbol{y}} \in 2^Y$ where $\varphi(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) = 1$

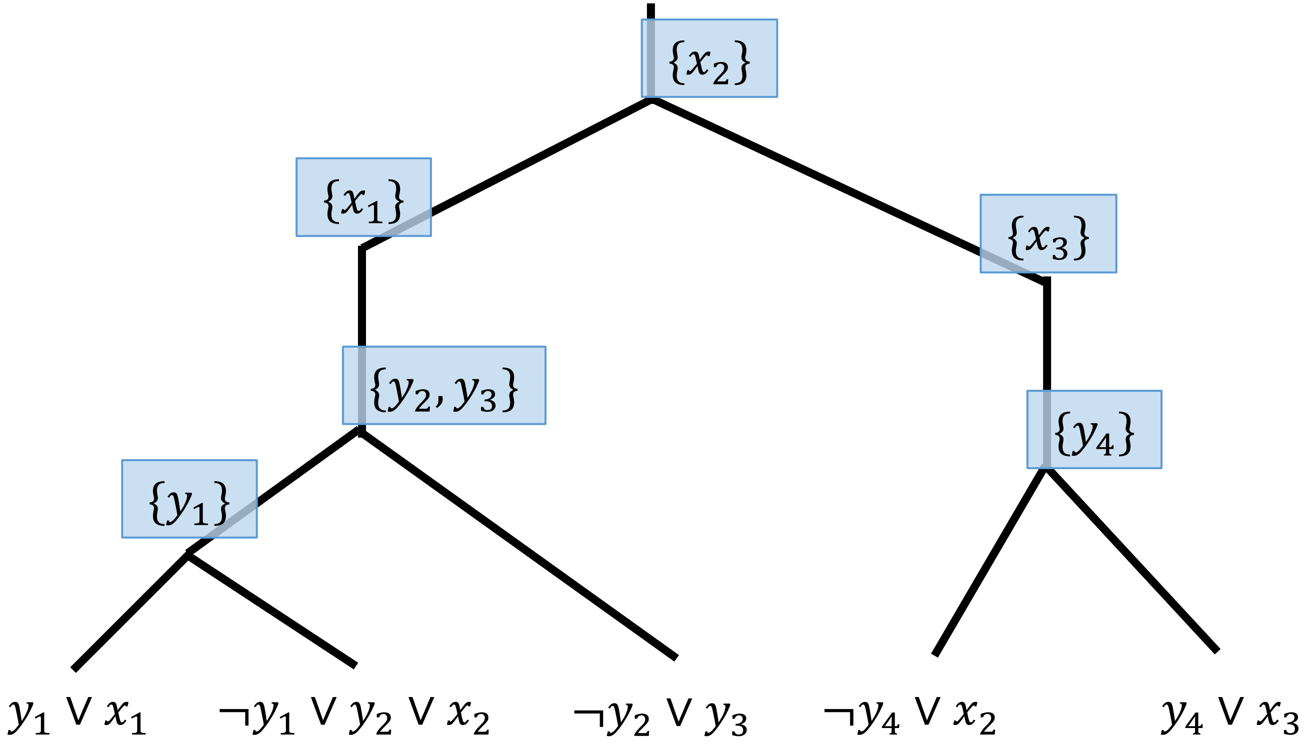
Techniques for *exact* projected model counting:

1. Search: Reason directly about φ using a SAT solver
   - projMC [Lagniez & Marquis, 19], reSSAT [Lee et al., 17]
2. Knowledge Compilation: Compile φ to a representation where counting is easy
   - $D4_P$ [Lagniez & Marquis, 19]
3. Dynamic Programming: Reason about the clause structure of φ
   - nestHDB [Hecher et al., 20]
   - **This work**

There is also *approximate* projected model counting, but we focus on exact.

DPMC: Model Counting Algorithm [**Dudek** et. al, 20]

1. **Planning**: Build a project-join tree of $\varphi(X)$.
2. **Execution**: Process project-join tree from leaves up to compute $\#_X\ \varphi(X)$.

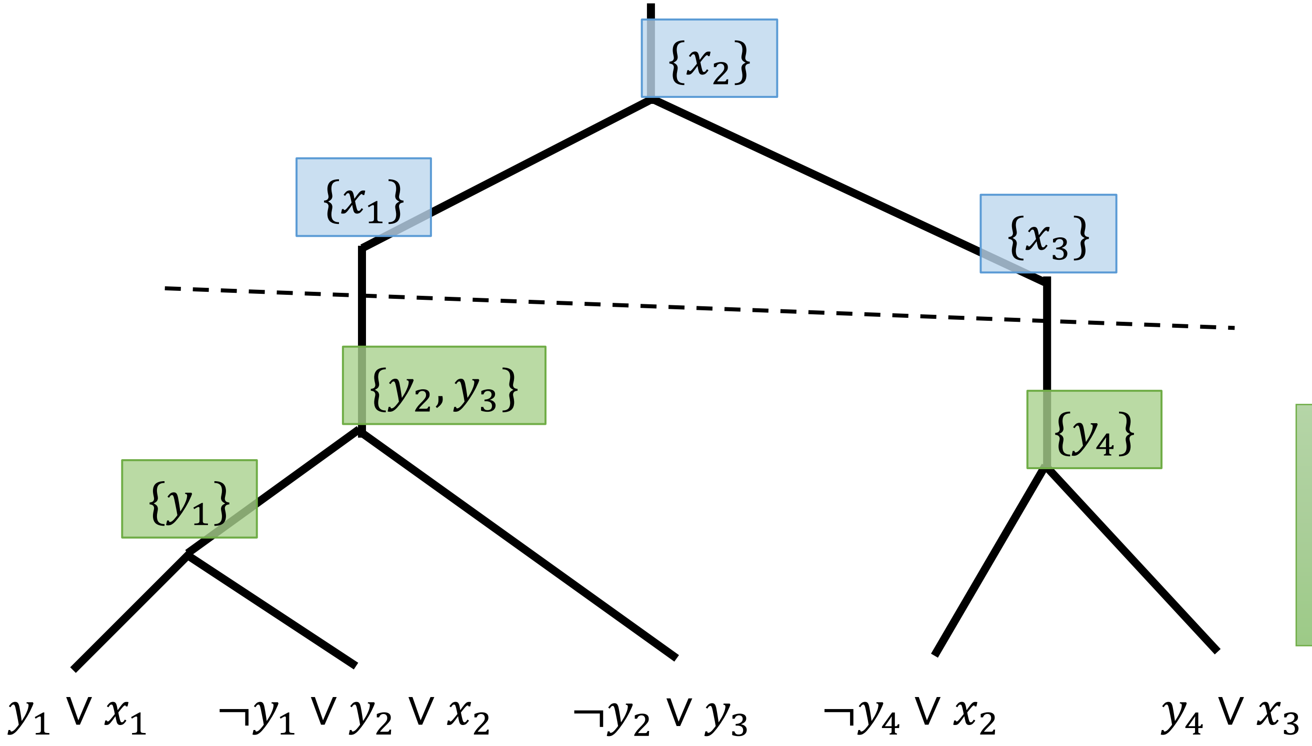


Definition: A ***project-join tree*** for φ is a tree where

1. Each clause of φ is assigned a (unique) leaf node.
2. Each variable of φ is assigned an internal node.
3. For all clauses C and variables z that appear in C, the z node is an ancestor of the C node.

Our Algorithm for Projected Model Counting

1. **Planning**: Build an **(X,Y)-graded** project-join tree of $\varphi(X,Y)$.
2. **Execution**: Process graded project-join tree from leaves up to compute $\#_X \exists_Y\ \varphi(X,Y)$.



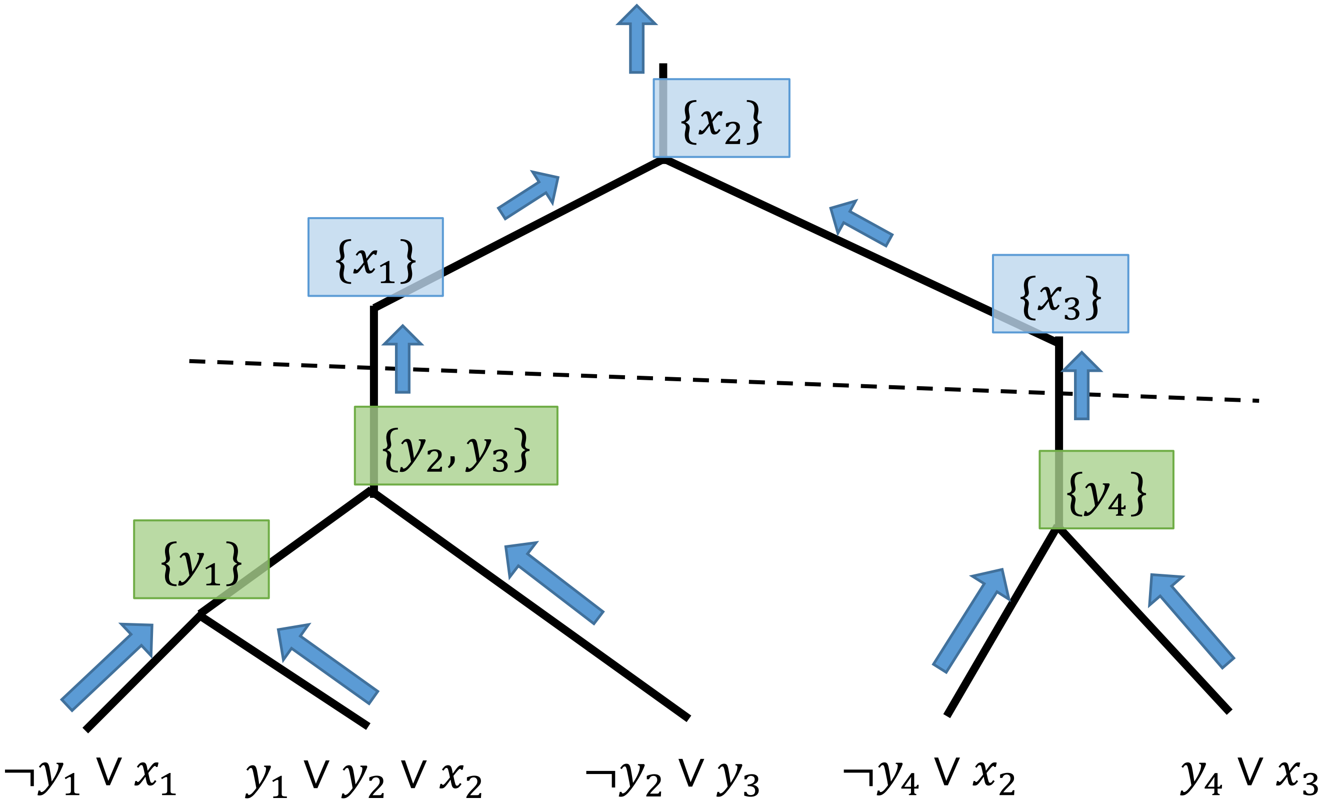
Definition: A ***project-join tree*** for φ is a tree where

1. Each clause of φ is assigned a (unique) leaf node.
2. Each variable of φ is assigned an internal node.
3. For all clauses C and variables z that appear in C, the z node is an ancestor of the C node.

Definition: A project-join tree is ***(X,Y)-graded*** if:

4. For all variables $x \in X$ and $y \in Y$, the y node is not an ancestor of the x node.

2. Execution

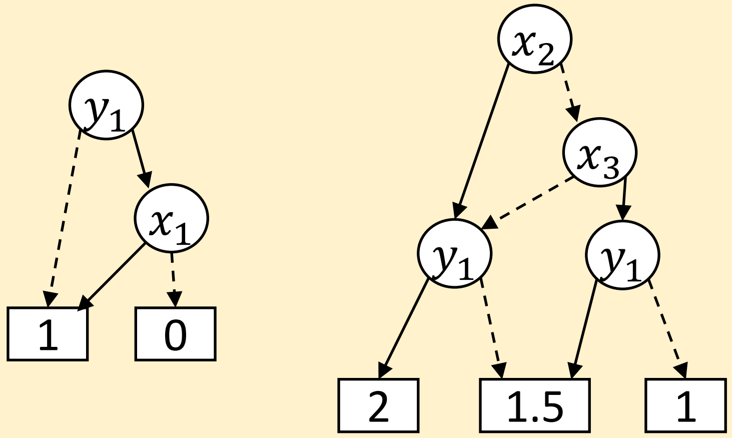


y_1	x_1	
0	0	1
0	1	1
1	0	0
1	1	1

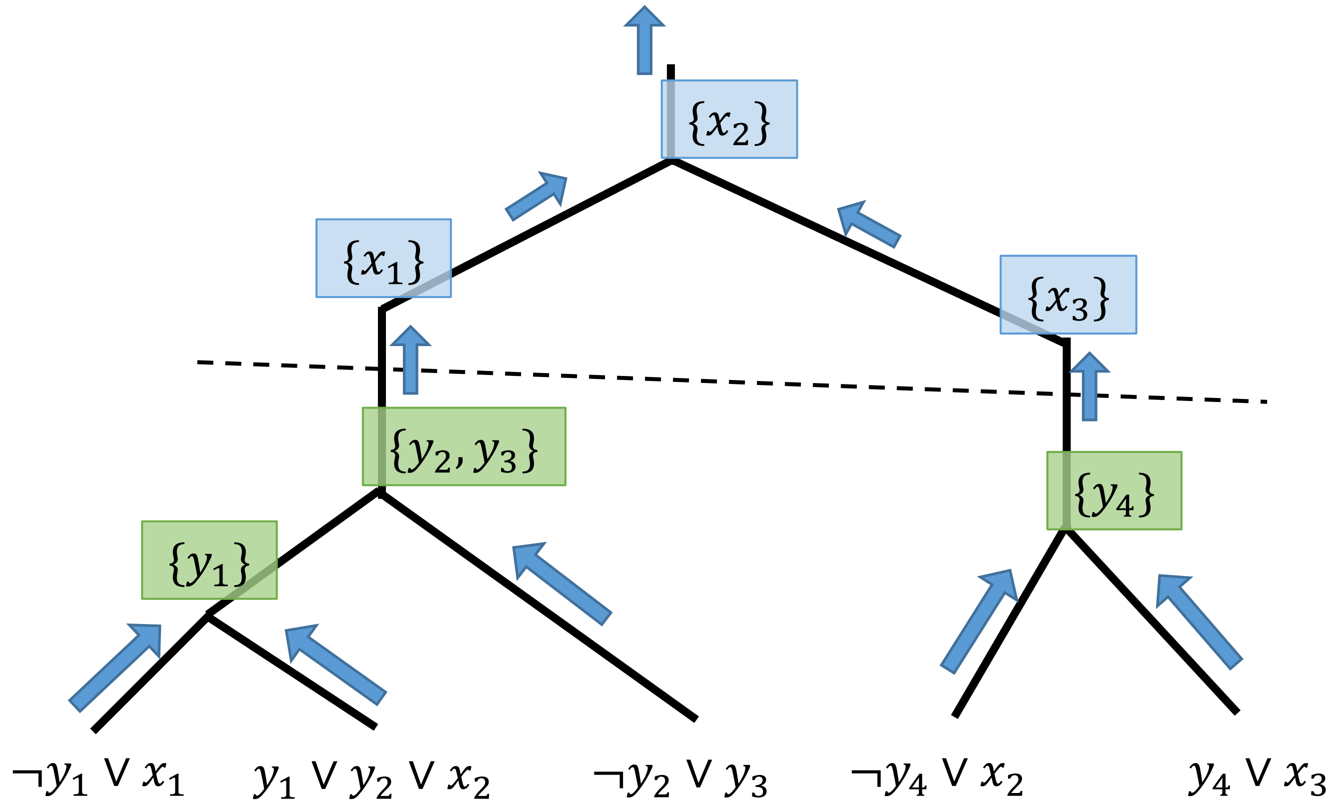
Idea:

Pass ADDs through tree from leaves to root, projecting away variables according to the labels.

An **Algebraic Decision Diagram (ADD)** represents a function $2^A \to \mathbb{R}$ as a (sparse) directed acyclic graph.



2. Execution: Running Time



Idea:
Pass ADDs through tree from leaves to root, projecting away variables according to the labels.

Key performance measure:

- *Width* of the project-join tree
- I.e., the maximum number of variables needed for a single ADD
- Width can be computed upfront

Theorem: Given:

- A CNF formula $\varphi(X, Y)$
- An (X,Y)-graded project-join tree of $\varphi(X, Y)$ of width w

This procedure computes $\#_X \exists_Y \, \varphi(X, Y)$ in time $O(2^w \cdot \text{poly}(|\varphi|))$.

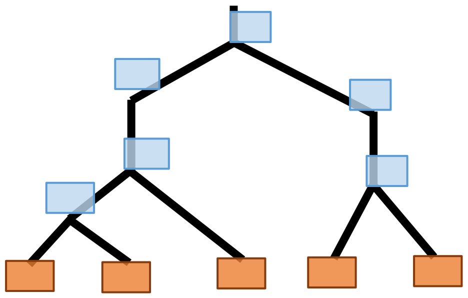
Projected counting (parameterized by width) with *ungraded* project-join trees is $\Omega\left(2^{2^w}\right)$ assuming ETH [Fichte et al., 18]

1. Planning (Model Counting)

How to find a low-width project-join tree of φ?

[McMahan et al., 04]
[Kask et al., 05]
[Markov and Shi, 05]

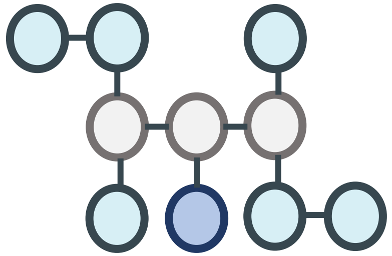
Project-Join Tree of φ



width w

poly. time

Tree Decomposition of $\mathbf{primal}(\boldsymbol{\varphi})$



treewidth $w + 1$

Decompositions show a "good" way to reason about a graph.
Black-box, heuristic tree-decomposition solvers:

- FlowCutter [Hamann and Strasser, 18]
- Tamaki [Tamaki, 17]
- htd [Abseher et al., 17]

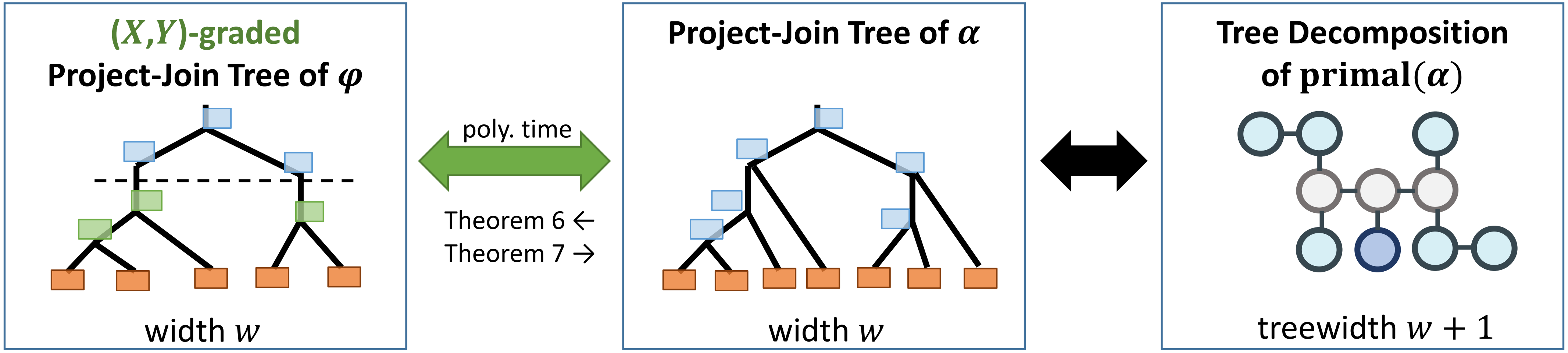
1. Planning (Projected Model Counting)

How to find a low-width **(X,Y)-graded** project-join tree of φ?

Possible Approach? Modify tree decomposition tools to take into account different variable types.

Better Idea: Use previous planners as a black box.

Add "virtual" clauses to φ to construct a new formula α so that:



1. Planning: The Reduction

Constructing α:

1. Build the ***primal graph*** of φ

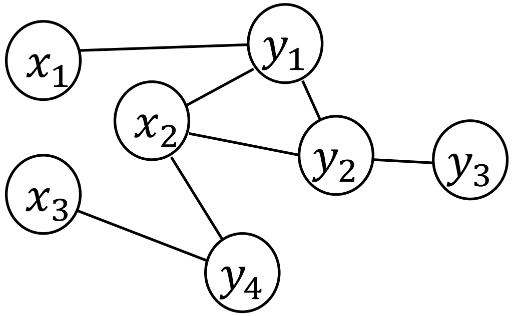
   A vertex for every variable, and an edge if two variables appear together in a clause.

2. Examine the connected components of Y variables in the primal graph

3. For each connected component, add a "virtual clause" of the adjacent X variables to α.

Example:

$$\varphi = \left\{ \begin{array}{c} \neg y_1 \lor x_1 \\ y_1 \lor y_2 \lor x_2 \\ \neg y_2 \lor y_3 \\ \neg y_4 \lor x_2 \\ y_4 \lor x_3 \end{array} \right\}$$



1. Planning: The Reduction

Constructing α:

1. Build the ***primal graph*** of φ

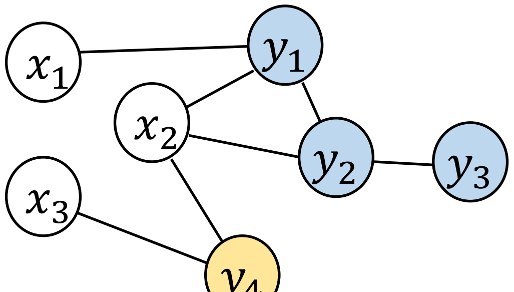
   A vertex for every variable, and an edge if two variables appear together in a clause.

2. Examine the connected components of Y variables in the primal graph

3. For each connected component, add a “virtual clause” of the adjacent X variables to α.

Example:

$$\varphi = \left\{ \begin{array}{c} \neg y_1 \vee x_1 \\ y_1 \vee y_2 \vee x_2 \\ \neg y_2 \vee y_3 \\ \neg y_4 \vee x_2 \\ y_4 \vee x_3 \end{array} \right\}$$



1. Planning: The Reduction

Constructing α:

1. Build the ***primal graph*** of φ

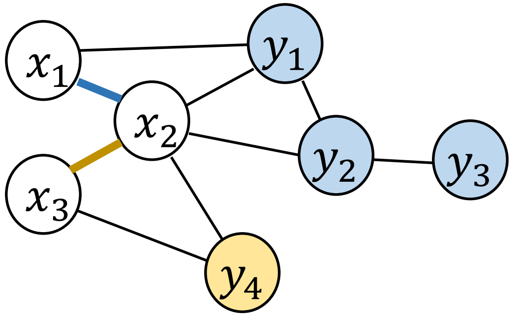
   A vertex for every variable, and an edge if two variables appear together in a clause.

2. Examine the connected components of Y variables in the primal graph

3. For each connected component, add a “virtual clause” of the adjacent X variables to α.

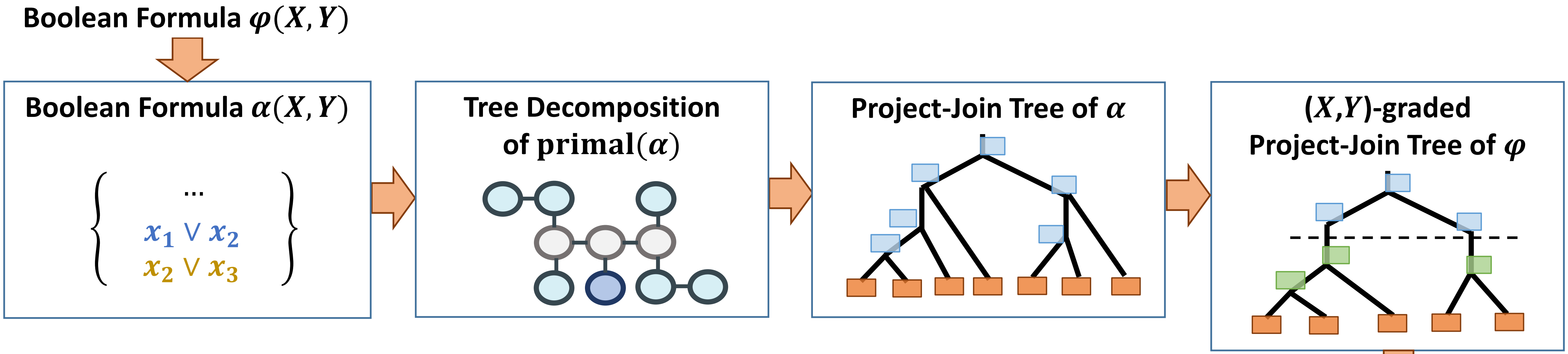
Example:

$$\varphi = \left\{ \begin{array}{c} \neg y_1 \lor x_1 \\ y_1 \lor y_2 \lor x_2 \\ \neg y_2 \lor y_3 \\ \neg y_4 \lor x_2 \\ y_4 \lor x_3 \end{array} \right\}$$



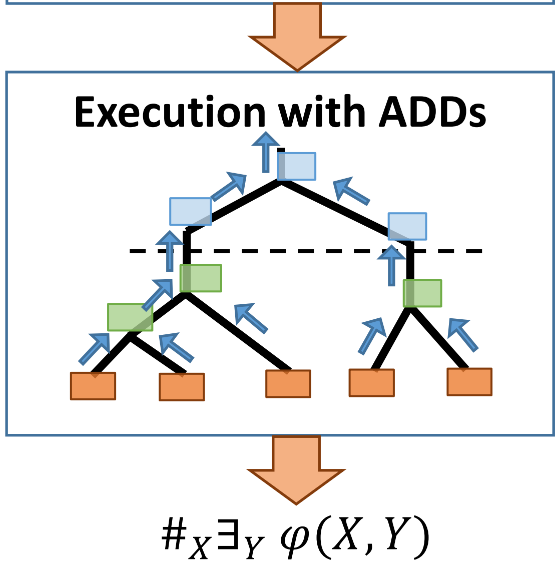
$$\alpha = \left\{ \begin{array}{c} \neg y_1 \lor x_1 \\ \dots \\ y_4 \lor x_3 \\ \boldsymbol{x_1 \lor x_2} \\ \boldsymbol{x_2 \lor x_3} \end{array} \right\}$$

Algorithm Overview: ProCount



ProCount: Implemented in C++

https://github.com/vardigroup/DPMC

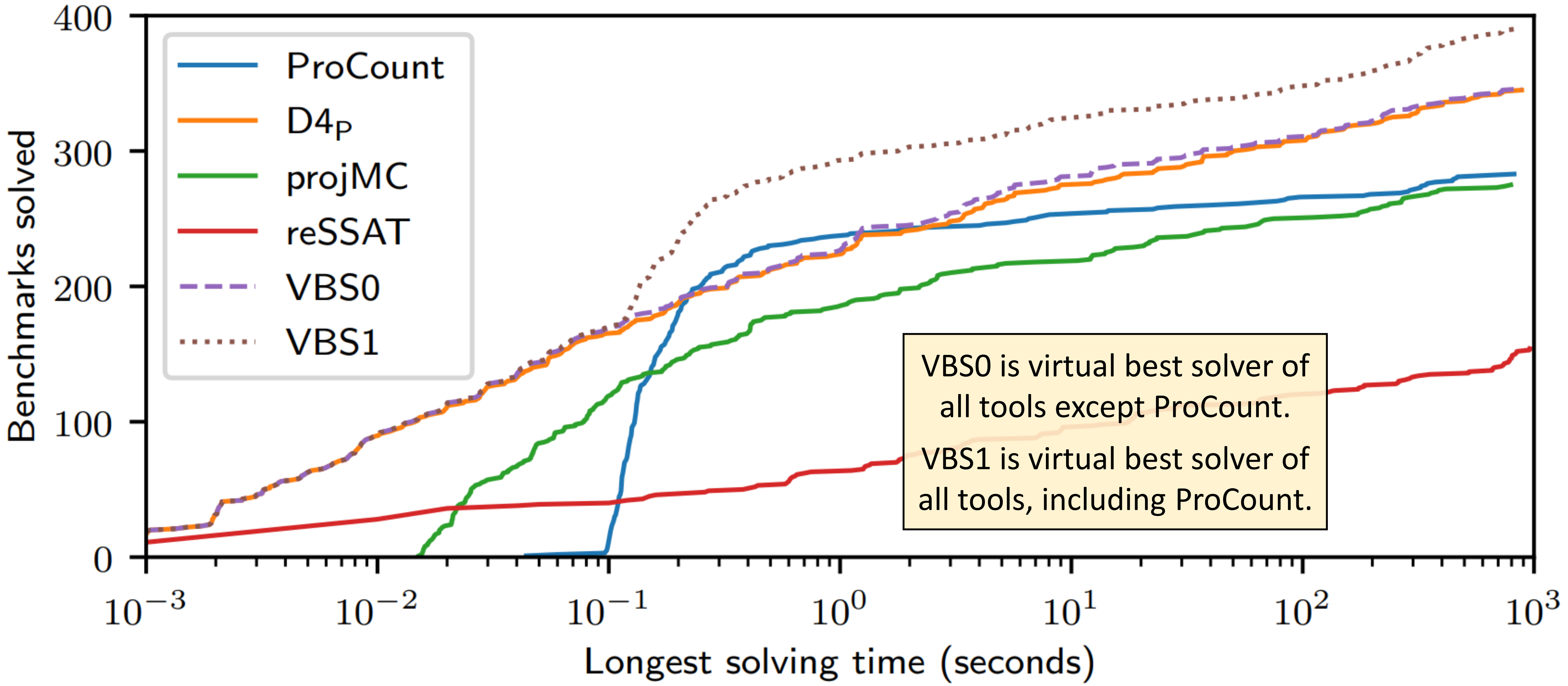


Planning: Black box tree-decomposition solvers

- FlowCutter, Tamaki, htd

Execution: ADDs with CUDD

Experimental Evaluation on Weighted #∃SAT



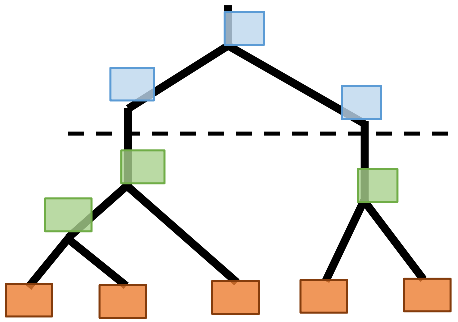
Our counter ProCount is the fastest tool on 131 benchmarks (34% of solved benchmarks).

* Run on a single 2.60 GHz core with 30 GB RAM. Used 849 #∃SAT benchmarks from [Gupta et al., 19] and [Soos and Meel, 19].

Summary and Conclusion

Problem: Weighted Projected Model Counting: $\#_X \exists_Y \varphi(X, Y)$

Our Solution: Two-phase algorithm based on graded project-join trees



1. **Planning**: Use tree decompositions to build a graded project-join tree
   Using previous planning as a black box lets us use *unmodified* tree-decomposition tools

2. **Execution**: Process graded project-join tree with ADDs to get the count
   Using graded projected-join trees lets us avoid a double-exponential dependency on width

Experiments: ProCount improves the VBS on 34% of benchmarks solved by at least one tool

https://github.com/vardigroup/DPMC

Future Work:

- More quantifier alternation (#QBF, MAP inference, FAQ problems)
- Planning with other graph decompositions
- Parallelization

jeffreydudek@gmail.com
vhp1@rice.edu
vardi@rice.edu