Optimizing Clasp by Exploiting Program Structure via Tree Decompositions

Towards D-FLAT0: Motivation and Ideas

Markus Hecher

Joint work with SW & JF

TU Wien

April 27^{th}, 2016

Outline

Why Tree Decompositions (TDs)?

- Many problems are (comput.) hard on graphs, but simpler on trees
- Is there a way to generalize from trees to complex graphs and still retain the good properties?

Why Tree Decompositions (TDs)?

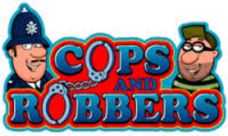
- Many problems are (comput.) hard on graphs, but simpler on trees
- Is there a way to generalize from trees to complex graphs and still retain the good properties?
- There is a way to capture how “tree-like” a graph is – the treewidth.
- The *treewidth* of a graph is defined in terms of tree decompositions . . .

Why Tree Decompositions (TDs)?

- Many problems are (comput.) hard on graphs, but simpler on trees
- Is there a way to generalize from trees to complex graphs and still retain the good properties?

- There is a way to capture how “tree-like” a graph is – the treewidth.
- The *treewidth* of a graph is defined in terms of tree decompositions . . .

- . . . but we can also use



Cops and Robber Games

- Round-based board (graph) game between **cops** and a **robber**.
- The **cops** are placed (and removed) freely on the vertices of G.
- The **robber** starts the game and can only be moved along paths in G (which currently are not “blocked” by a cop).

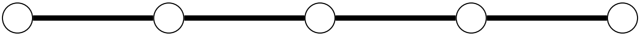
Cops and Robber Games

- Round-based board (graph) game between **cops** and a **robber**.
- The **cops** are placed (and removed) freely on the vertices of G.
- The **robber** starts the game and can only be moved along paths in G (which currently are not “blocked” by a cop).
- The **cops** win if a cop is placed on the robbers position and the robber can not move away; otherwise the **robber** wins.

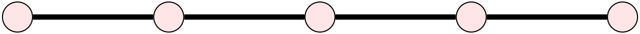
Treewidth

The number of cops required to capture the robber on G equals the treewidth of G (plus one).

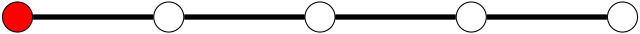
Examples



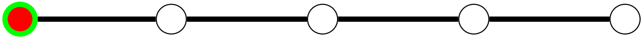
Examples



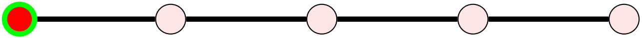
Examples



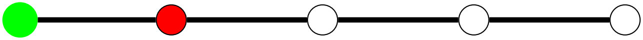
Examples



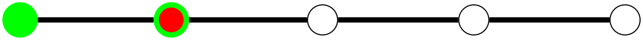
Examples



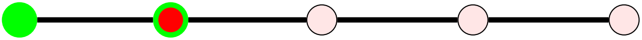
Examples



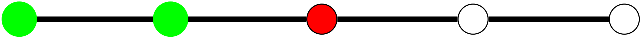
Examples



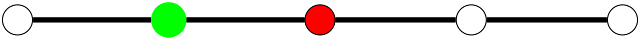
Examples



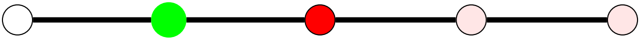
Examples



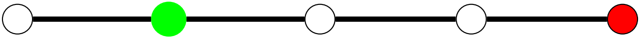
Examples



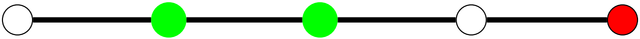
Examples



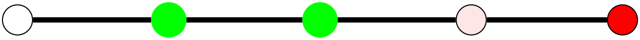
Examples



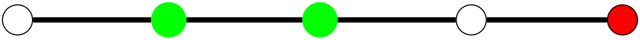
Examples



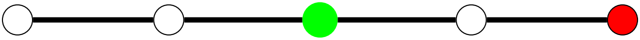
Examples



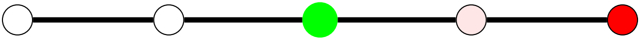
Examples



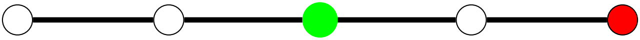
Examples



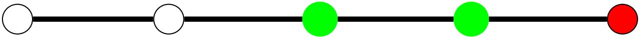
Examples



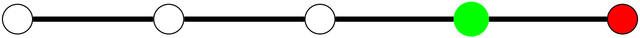
Examples



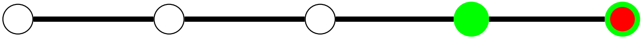
Examples



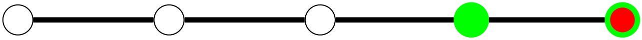
Examples



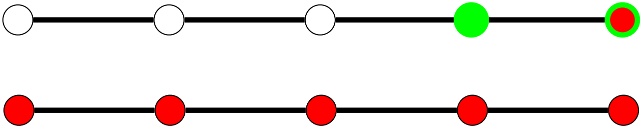
Examples



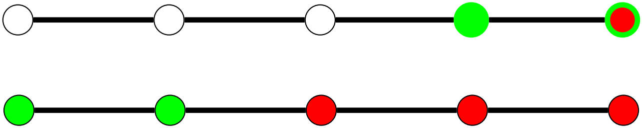
Examples



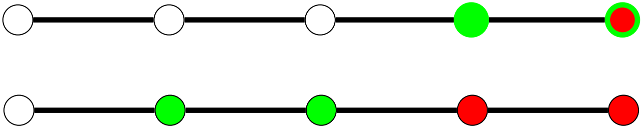
Examples



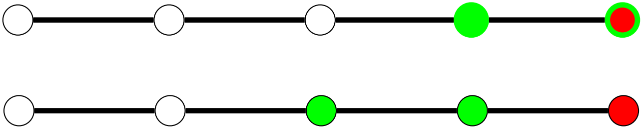
Examples



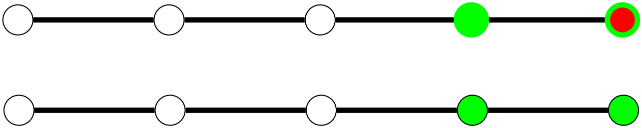
Examples



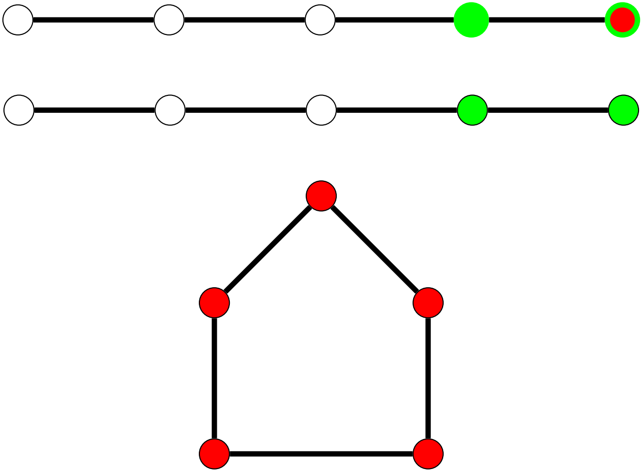
Examples



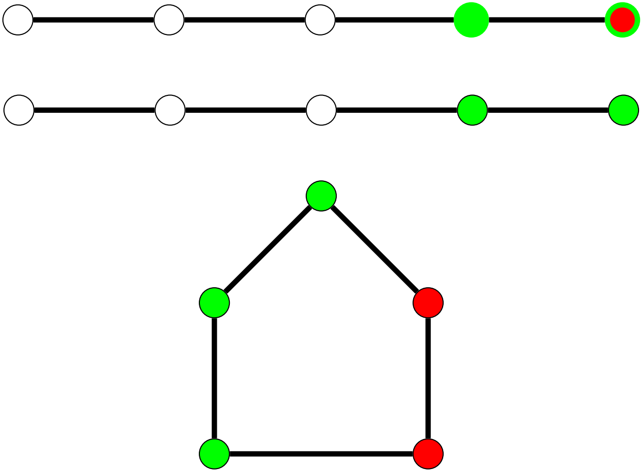
Examples



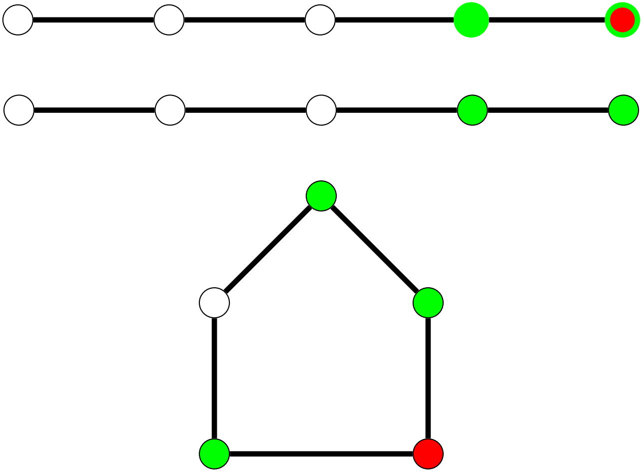
Examples



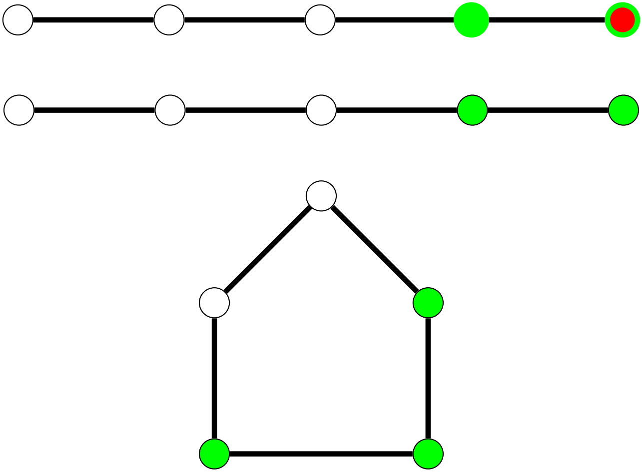
Examples



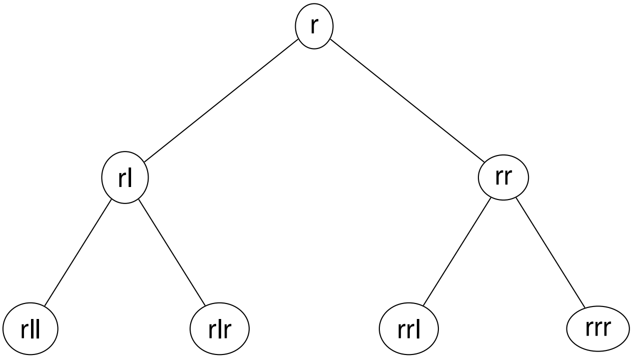
Examples



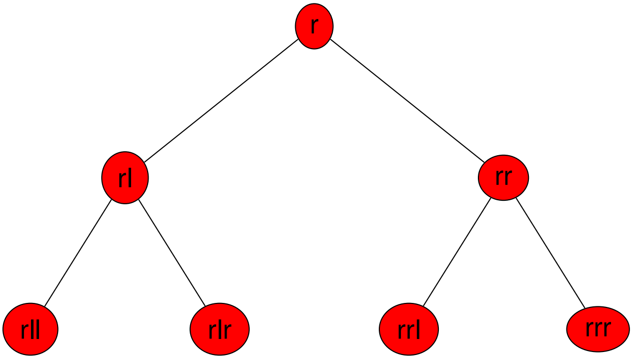
Examples



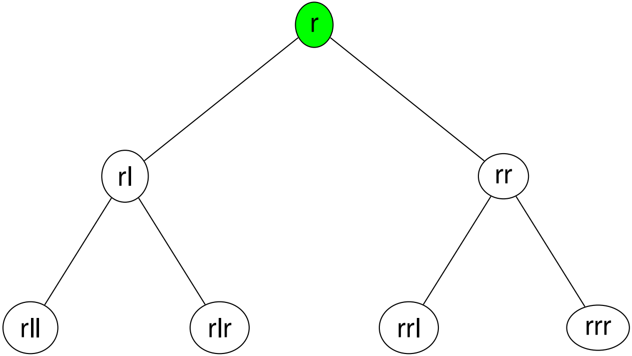
Example: Trees



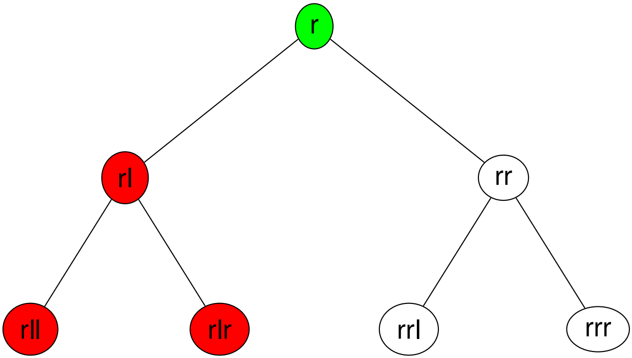
Example: Trees



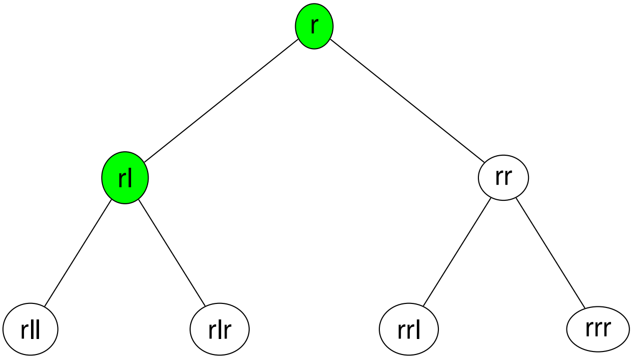
Example: Trees



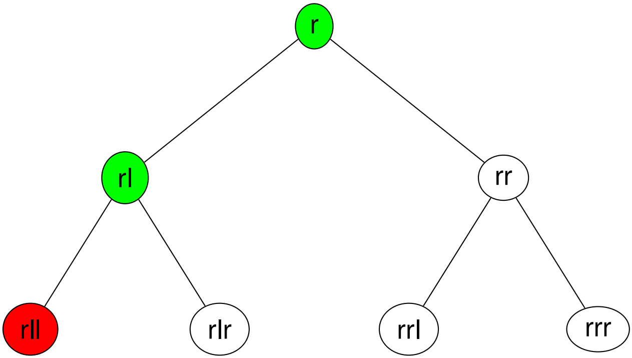
Example: Trees



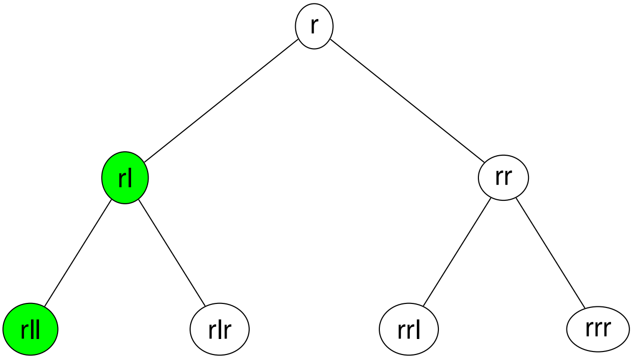
Example: Trees



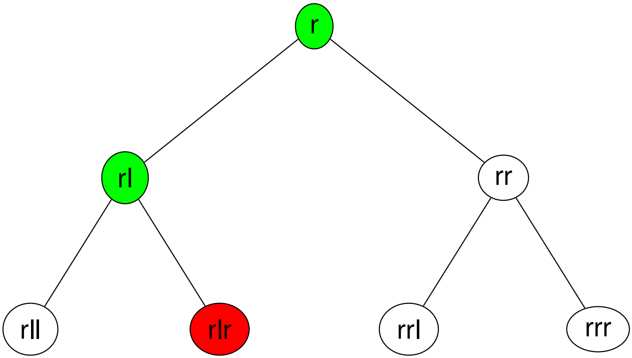
Example: Trees



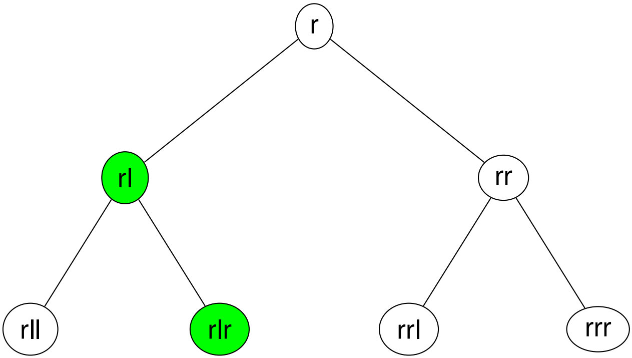
Example: Trees



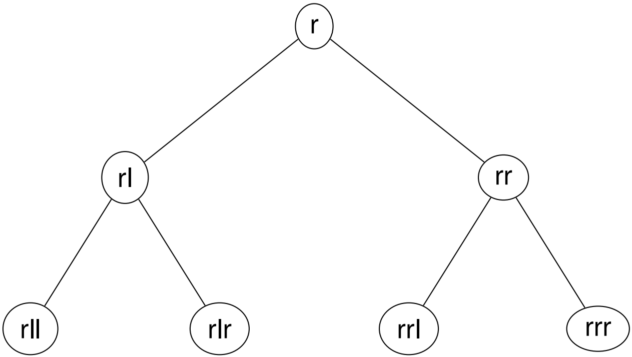
Example: Trees



Example: Trees



Example: Trees



Using Tree Decompositions (TDs)

Task: Solve NP-complete problem instances $\mathcal{I}$

Problem: Standard approaches require in general $\mathcal{O}(2^{|\mathcal{I}|})$ runtime

Using Tree Decompositions (TDs)

Task: Solve NP-complete problem instances $\mathcal{I}$

Problem: Standard approaches require in general $\mathcal{O}(2^{|\mathcal{I}|})$ runtime

Idea: Dynamic Programming (DP) on Tree Decompositions

- Exploit the structure of the problem instance $\mathcal{I}$
- Confine complexity to the parameter "treewidth"
- Goal: Fixed-parameter tractability (FPT) w.r.t. treewidth t, i.e. solvability in time

$$f(t) \cdot |\mathcal{I}|^{\mathcal{O}(1)}$$

Using Tree Decompositions (TDs)

Task: Solve NP-complete problem instances $\mathcal{I}$

Problem: Standard approaches require in general $\mathcal{O}(2^{|\mathcal{I}|})$ runtime

Idea: Dynamic Programming (DP) on Tree Decompositions

- Exploit the structure of the problem instance $\mathcal{I}$
- Confine complexity to the parameter “treewidth”
- Goal: Fixed-parameter tractability (FPT) w.r.t. treewidth t, i.e. solvability in time

$$f(t) \cdot |\mathcal{I}|^{\mathcal{O}(1)}$$

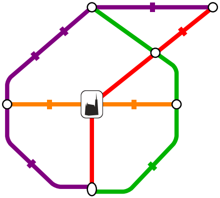
The D-FLAT Approach

1. Decompose instance $\mathcal{I}$
2. Specify DP algorithm via ASP
3. Combine results of subsequent calls to Clasp

Why This Approach Is Reasonable

- It works for many hard problems
- Real-world applications often have small treewidth
- In many cases, we can thus avoid the explosion of runtime!

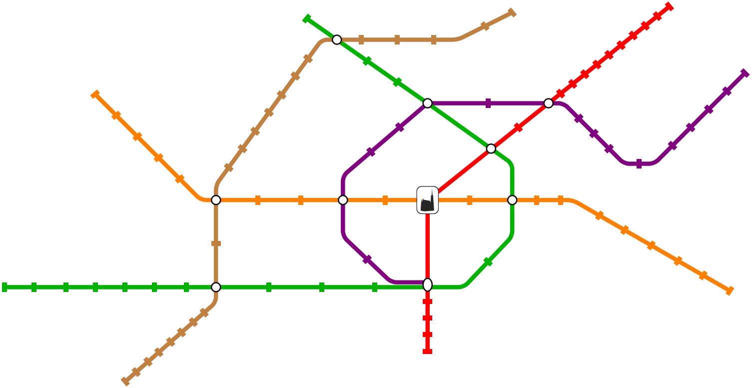
Example: Treewidth 3 (i.e., runtime $\mathcal{O}(2^{(3+1)} \cdot |\mathcal{I}|)$).



Why This Approach Is Reasonable

- It works for many hard problems
- Real-world applications often have small treewidth
- In many cases, we can thus avoid the explosion of runtime!

Example: Treewidth 3 (i.e., runtime $\mathcal{O}(2^{(3+1)} \cdot |\mathcal{I}|)$). Still.

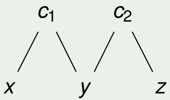


Outline

Solving Decomposed Problems

Example: SAT

$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$

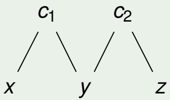


Methodology:

Solving Decomposed Problems

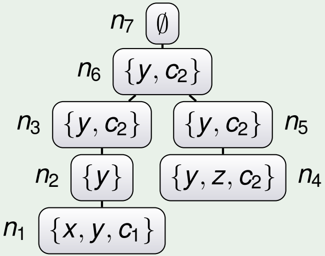
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

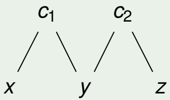
1. Decompose instance



Solving Decomposed Problems

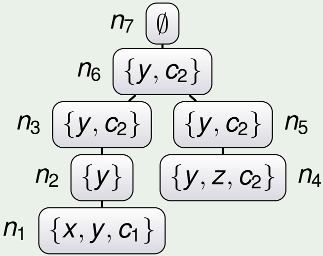
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

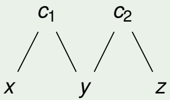
1. Decompose instance
2. Solve partial problems



Solving Decomposed Problems

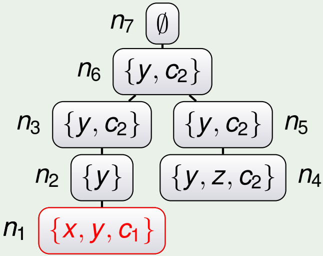
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61),(63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31,51)\}$
62	T		$\{(32,52)\}$
63	T	S	$\{(32,53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41),(42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12),(14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

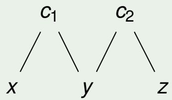
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

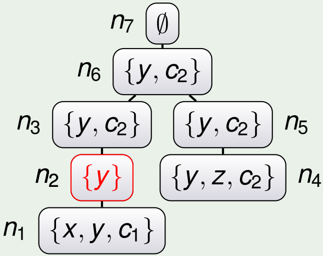
Example: SAT

$$\underbrace{(x \lor y)}_{c_1} \land \underbrace{(\neg y \lor z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

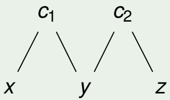
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

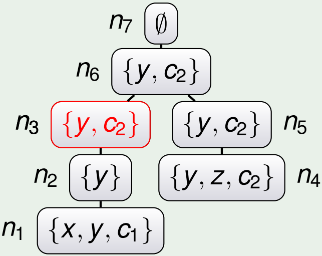
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61),(63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31,51)\}$
62	T		$\{(32,52)\}$
63	T	S	$\{(32,53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41),(42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12),(14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

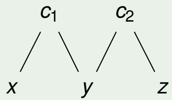
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

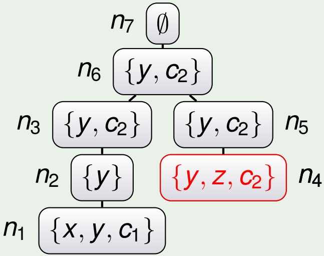
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

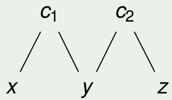
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

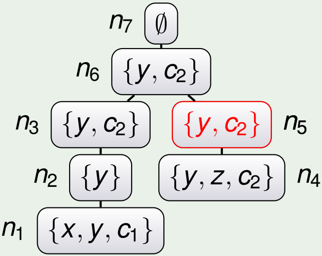
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

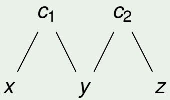
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

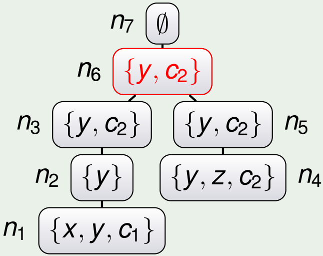
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

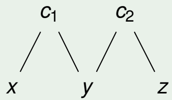
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

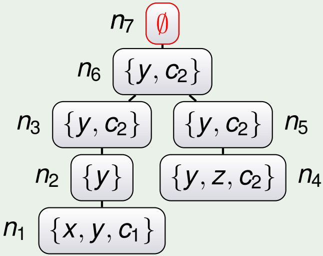
Example: SAT

$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$



Methodology:

1. Decompose instance
2. Solve partial problems



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

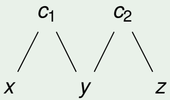
n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving Decomposed Problems

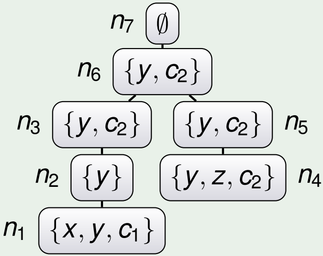
Example: SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$



Methodology:

1. Decompose instance
2. Solve partial problems
3. Combine solutions



n_7

id	origin
71	$\{(61), (63)\}$

n_6

id	y	c_2	origin
61	F	S	$\{(31, 51)\}$
62	T		$\{(32, 52)\}$
63	T	S	$\{(32, 53)\}$

n_3

id	y	c_2	origin
31	F	S	$\{(21)\}$
32	T		$\{(22)\}$

n_5

id	y	c_2	origin
51	F	S	$\{(41), (42)\}$
52	T		$\{(43)\}$
53	T	S	$\{(44)\}$

n_2

id	y	origin
21	F	$\{(13)\}$
22	T	$\{(12), (14)\}$

n_4

id	y	z	c_2
41	F	F	S
42	F	T	S
43	T	F	
44	T	T	S

n_1

id	x	y	c_1
11	F	F	
12	F	T	S
13	T	F	S
14	T	T	S

Solving 2^{nd}-level Problems

Example: $k, \subseteq$-MINIMAL SAT

$$\underbrace{(x \vee y)}_{c_1} \wedge \underbrace{(\neg y \vee z)}_{c_2}$$

n_7

id	origin	Γ
71	$\{(61), (64)\}$	$\emptyset$
72	$\{(62), (65)\}$	$\{71\}$

n_6

id	y	c_2	origin	Γ
61	F	S	$\{(31, 51)\}$	$\emptyset$
62	F	S	$\{(31, 52)\}$	$\{61\}$
63	T		$\{(32, 53), (33, 53)\}$	$\emptyset$
64	T	S	$\{(32, 54)\}$	$\emptyset$
65	T	S	$\{(33, 54)\}$	$\{61, 64\}$

n_3

id	y	c_2	origin	Γ
31	F	S	$\{(21)\}$	$\emptyset$
32	T		$\{(22)\}$	$\emptyset$
33	T		$\{(23)\}$	$\{31, 32\}$

n_5

id	y	c_2	origin	Γ
51	F	S	$\{(41)\}$	$\emptyset$
52	F	S	$\{(42)\}$	$\{51\}$
53	T		$\{(43)\}$	$\{51\}$
54	T	S	$\{(44)\}$	$\{51\}$

n_2

id	y	origin	Γ
21	F	$\{(13)\}$	$\emptyset$
22	T	$\{(12)\}$	$\emptyset$
23	T	$\{(14)\}$	$\{21, 22\}$

n_4

id	y	z	c_2	Γ
41	F	F	S	$\emptyset$
42	F	T	S	$\{41\}$
43	T	F		$\{41\}$
44	T	T	S	$\{41, 42, 43\}$

n_1

id	x	y	c_1	Γ
11	F	F		$\emptyset$
12	F	T	S	$\{11\}$
13	T	F	S	$\{11\}$
14	T	T	S	$\{11, 12, 13\}$

Outline

Introduction

Background

D-FLAT
- What's that?
- How does it work?

D-FLAT0

Conclusion

D-FLAT

Dynamic Programming **F**ramework with **L**ocal Execution of **A**SP on **T**ree Decompositions[1]

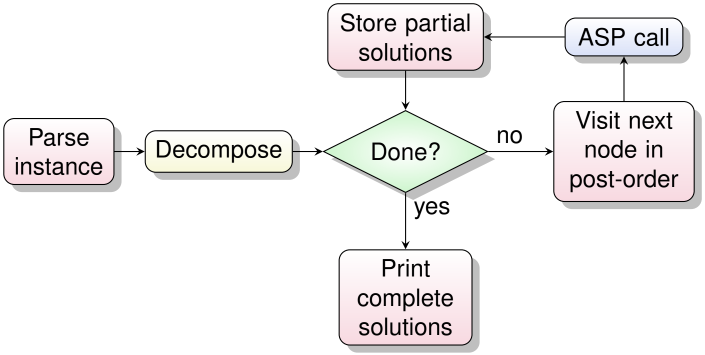
What does it do?

1. Constructs a tree decomposition of the input structure
2. In each node: Executes user-supplied program with ASP solver
   - Stores partial solutions specified by the answer sets
3. Decides the problem (or materializes solutions)

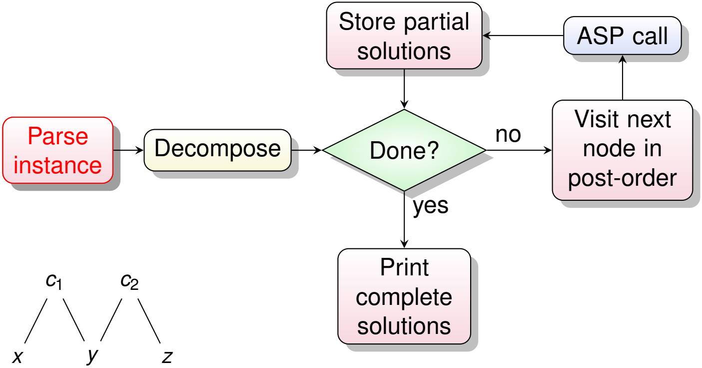
- Users only need to write an ASP program

[1] D-FLAT is open source: `https://www.github.com/bbliem/dflat`

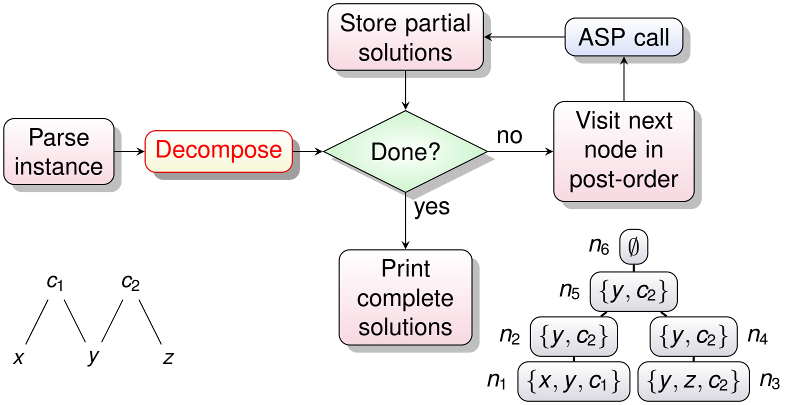
D-FLAT Algorithm Outline



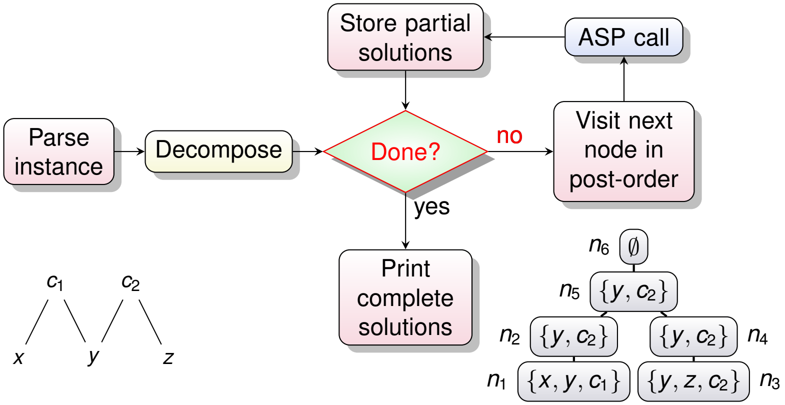
D-FLAT Algorithm Outline



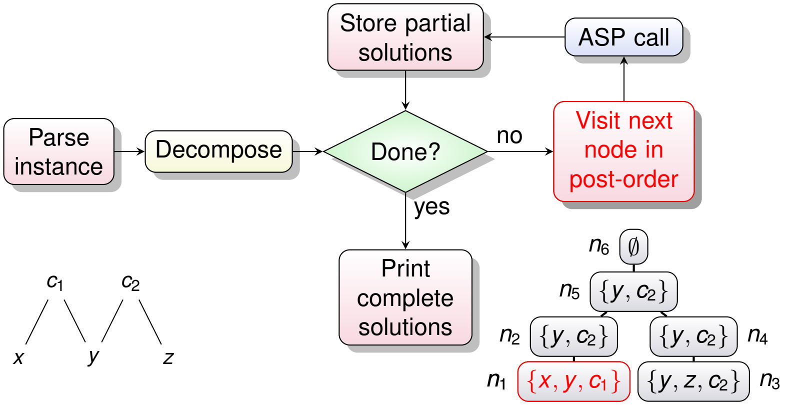
D-FLAT Algorithm Outline



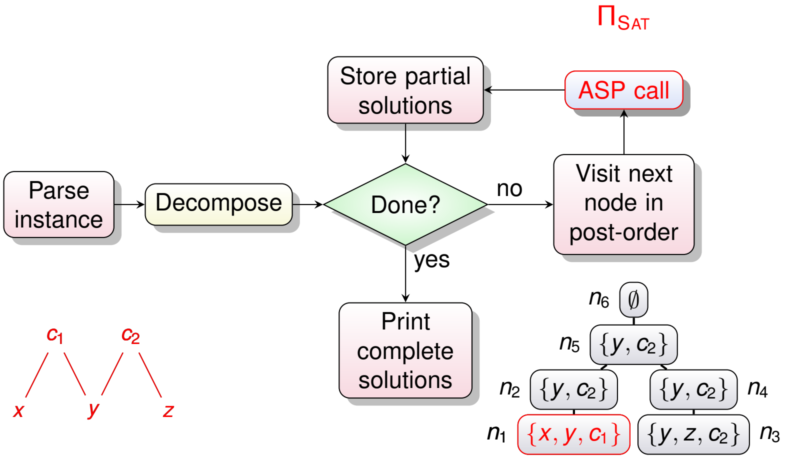
D-FLAT Algorithm Outline



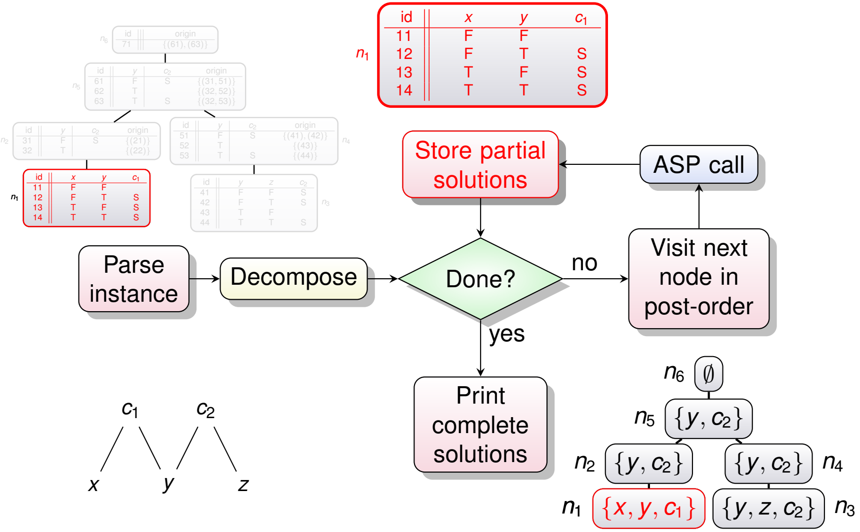
D-FLAT Algorithm Outline



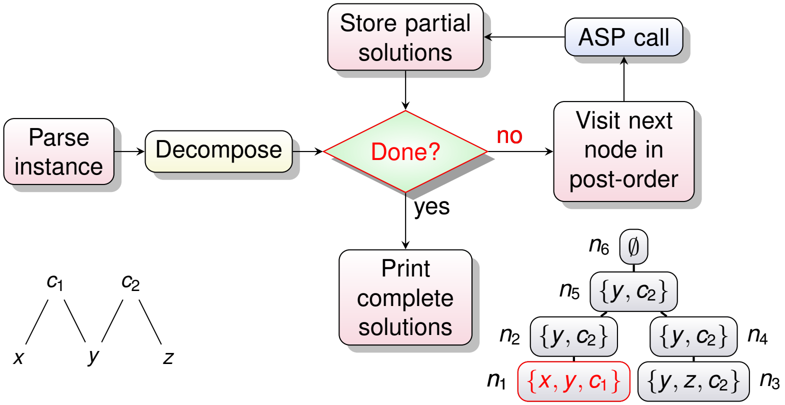
D-FLAT Algorithm Outline



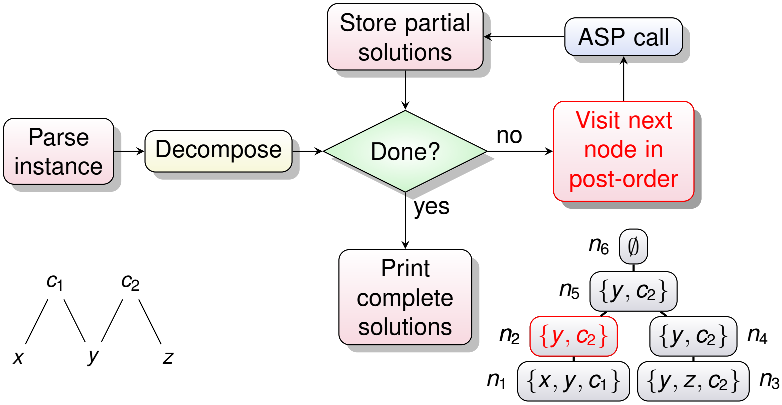
D-FLAT Algorithm Outline



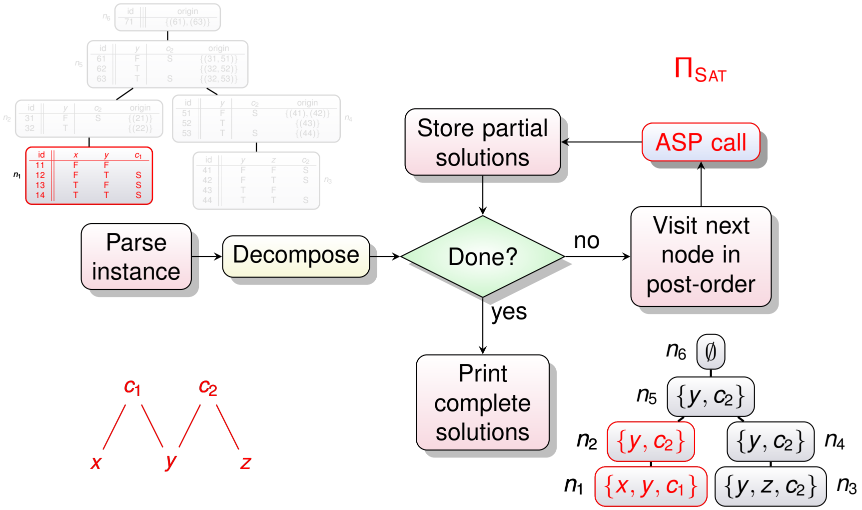
D-FLAT Algorithm Outline



D-FLAT Algorithm Outline

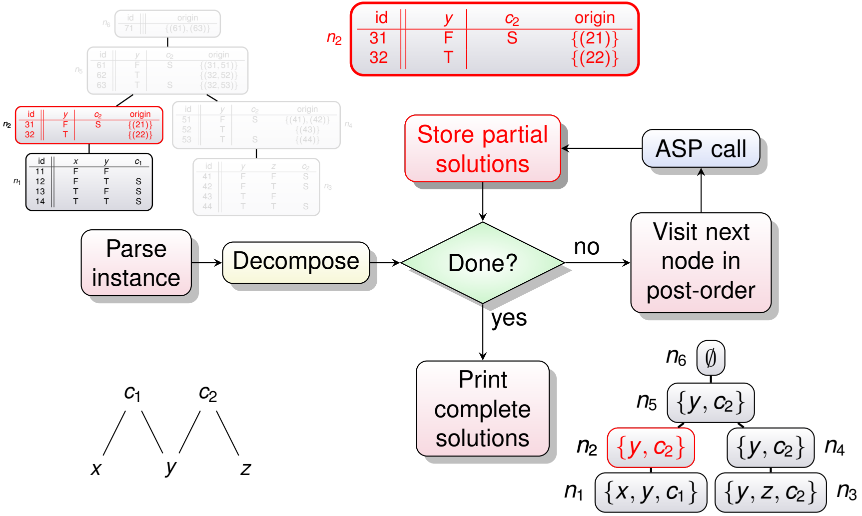


D-FLAT Algorithm Outline

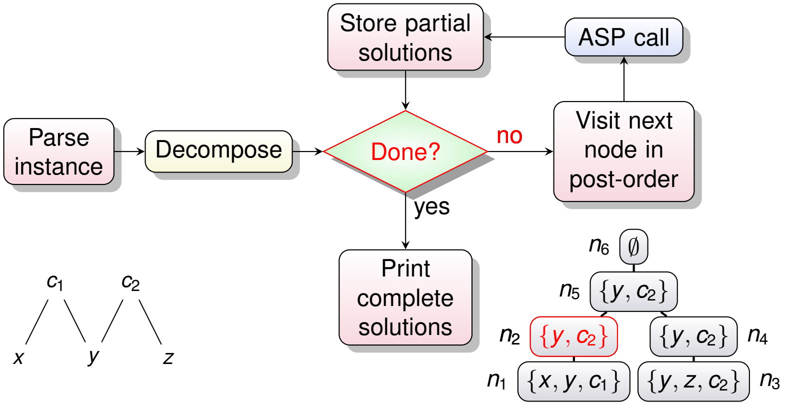


D-FLAT Algorithm Outline

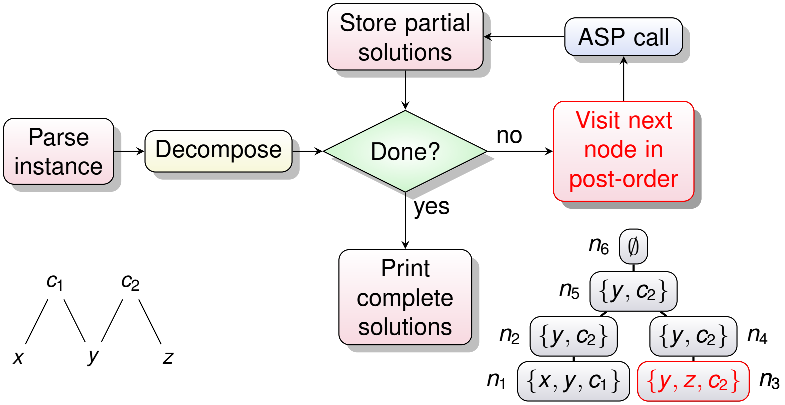
id	y	c_2	origin
31	F	S	{(21)}
32	T		{(22)}



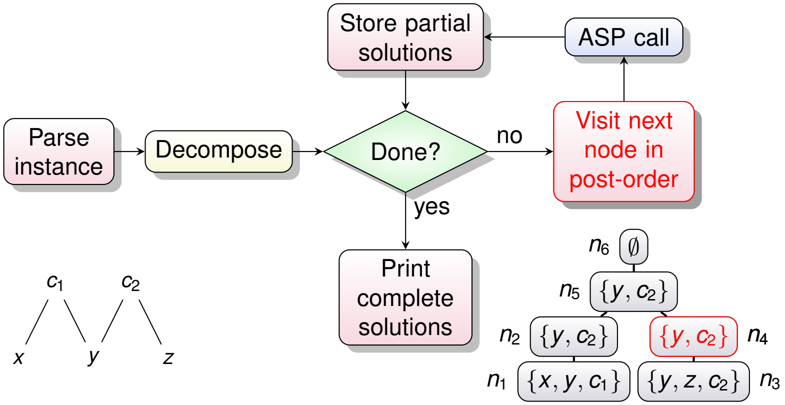
D-FLAT Algorithm Outline



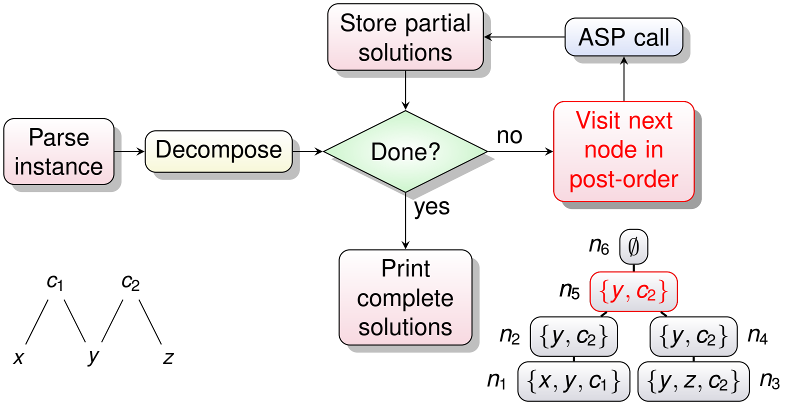
D-FLAT Algorithm Outline



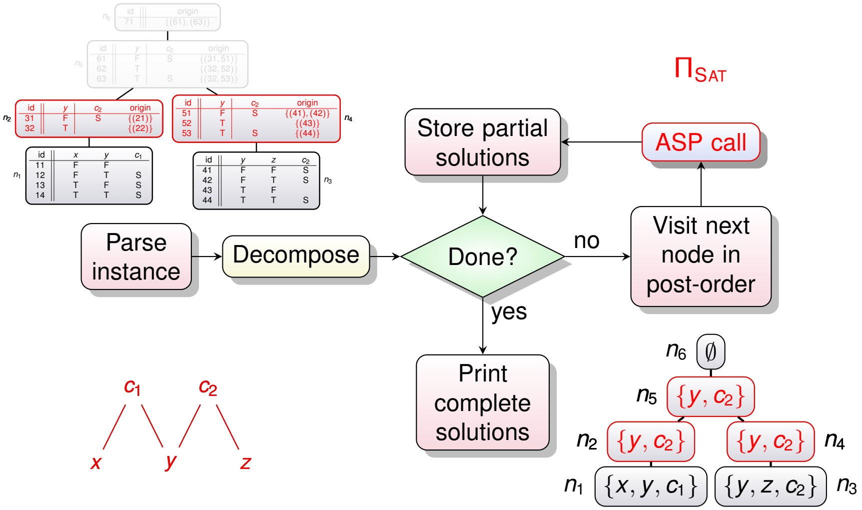
D-FLAT Algorithm Outline



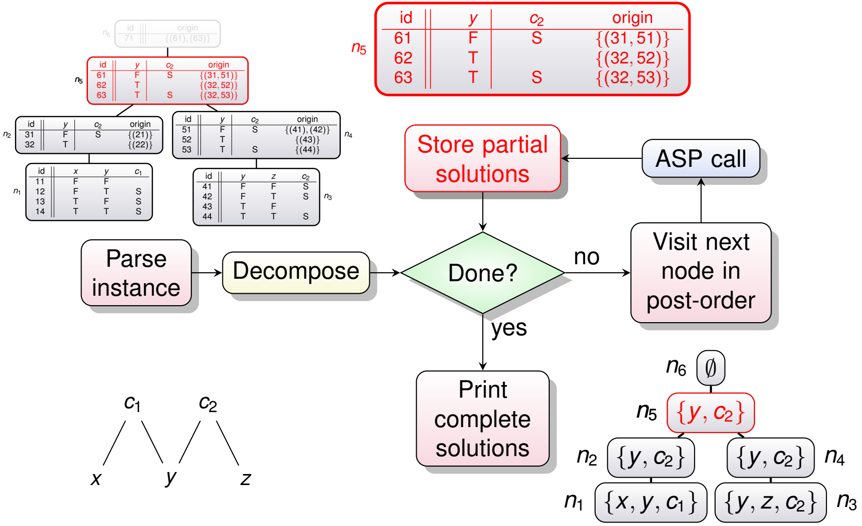
D-FLAT Algorithm Outline



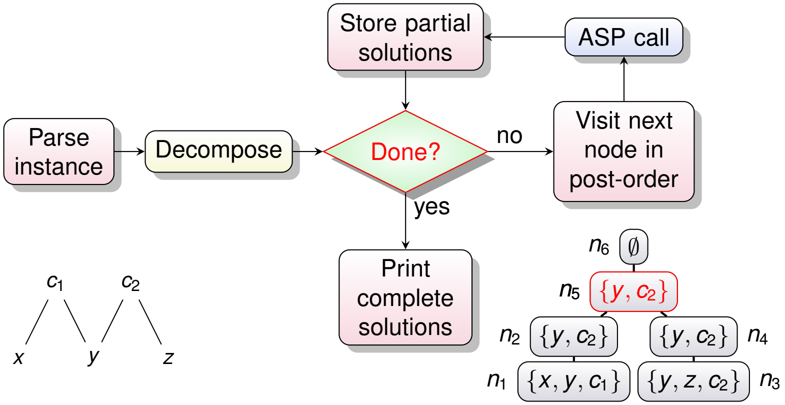
D-FLAT Algorithm Outline



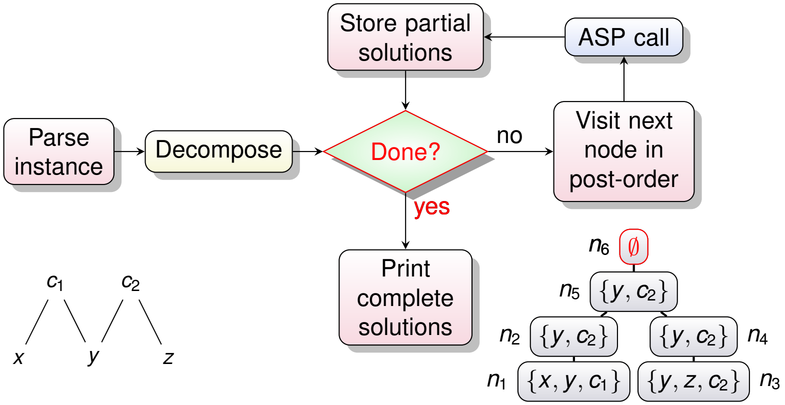
D-FLAT Algorithm Outline



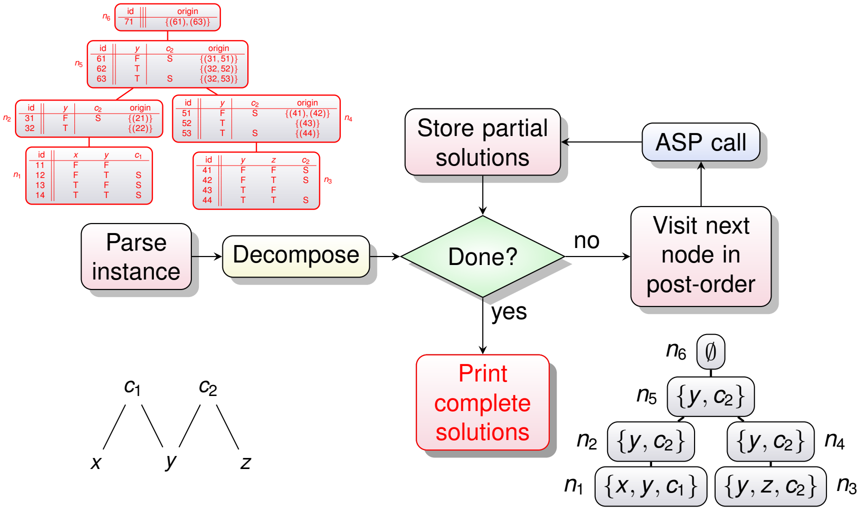
D-FLAT Algorithm Outline



D-FLAT Algorithm Outline



D-FLAT Algorithm Outline



Computing Partial Solutions via ASP

Illustrated by means of SAT

User-supplied program Π_{SAT}

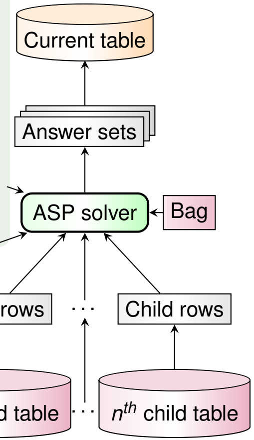
```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).

{ item(A) : atom(A), introduced(A) }.
item(X) ← extend(R), childItem(R,X), current(X).
item(C) ← current(C;A), pos(C,A),     item(A).
item(C) ← current(C;A), neg(C,A), not item(A).

← extend(R),   clause(C), removed(C),   f(R,C).
← extend(X;Y), atom(A), childItem(X,A), f(Y,A).
f(R,X) ← childRow(R,N), bag(N,X), not childItem(R,X).
```

Instance

```
atom(a;b;c;d). clause(c1;c2;c3).
pos(c1,a). neg(c1,b). pos(c2,c).
neg(c2,a). neg(c3,d).
```



Solving 2^{nd}-level Problems via D-FLAT

Demonstration of user-supplied program $\Pi_{k,\subseteq\text{-MINIMAL SAT}}$

```
length(2). level(1..2). or(0). and(1).
extend(0,R) ← root(R).
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L<2.

{ item(2,A;1,A) : atom(A), introduced(A) }.
auxItem(L,C) ← current(C;A), pos(C,A), item(L,A), level(L).
auxItem(L,C) ← current(C;A), neg(C,A), not item(L,A), level(L).
item(L,X) ← extend(L,R), childItem(R,X), current(X), level(L).
auxItem(L,C) ← extend(L,R), childAuxItem(R,C), current(C), level(L).

false(S,X) ← atNode(S,N), childNode(N), bag(N,X), sub(_,S), not childItem(S,X).
unsat(S,C) ← atNode(S,N), childNode(N), bag(N,C), sub(_,S), not childAuxItem(S,C).
unsat(R) ← clause(C), removed(C), unsat(R,C).
← extend(L,X;L,Y), atom(A), childItem(X,A), false(Y,A), level(L).
← extend(L,R), unsat(R), level(L).

reject ← final, extend(1,R), sub(R,S), childAuxItem(S,smaller), not unsat(S).
accept ← final, not reject.
auxItem(2,smaller) ← extend(2,S), childAuxItem(S,smaller).
auxItem(2,smaller) ← atom(A), item(1,A), not item(2,A).
← atom(A), item(2,A), not item(1,A).
```

Simple Optimizations with D-FLAT^2[2]

Example: User-supplied program $\Pi^2_{k,\subseteq\text{-MINIMAL SAT}}$

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).

{ item(A) : atom(A), introduced(A) }.
item(X) ← extend(R), childItem(R,X), current(X).
item(C) ← current(C;A), pos(C,A),     item(A).
item(C) ← current(C;A), neg(C,A), not item(A).

← extend(R),   clause(C), removed(C),   f(R,C).
← extend(X;Y), atom(A), childItem(X,A), f(Y,A).
f(R,X) ← childRow(R,N), bag(N,X), not childItem(R,X).

optItem(X)  ←   item(X), atom(X).
```

[2] D-FLAT^2 is available at `https://www.github.com/hmarkus/dflat-2`.

Outline

Motivation

Lessons Learnt: D-FLAT

- is well suited for instances of small bounded treewidth
- can solve many problems
- was extended to a competitive tool for 2^{nd}-level DP algorithms [1]
- is now capable of deriving solutions in a *lazy* fashion [2]

Bliem, Charwat, Hecher and Woltran: D-FLATˆ2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy, ASPOCP 2015

Bliem, Kaufmann, Schaub and Woltran: ASP for Anytime Dynamic Programming on Tree Decompositions, IJCAI 2016

Motivation & Idea

However . . .

- it is well suited only for instances of small, bounded treewidth
- it has quite some overhead
  - (Re-)grounding in every TD node
  - Several calls to Clasp (destroys learning process)
  - Finding the first solution is effectively enumerating
  - Actual materialization of partial, intermediate results
  - . . .

Motivation & Idea

However ...

- it is well suited only for instances of small, bounded treewidth
- it has quite some overhead
  - (Re-)grounding in every TD node
  - Several calls to Clasp (destroys learning process)
  - Finding the first solution is effectively enumerating
  - Actual materialization of partial, intermediate results
  - ...

Idea

- Reduce overhead by putting a little bit of D-FLAT into Clasp ;-)
- Improve applicability of the result

Possible Approaches

- ...

Step by step . . .

Let's just merge both worlds?

1. Improve Clasp by exploiting the structure of problem instance Π_I via extracting properties of some (heuristically computed) TD T_{Π_I}
2. . . . and let Clasp do all the work ;-)

Step by step . . .

Let's just merge both worlds?

1. Improve Clasp by exploiting the structure of problem instance Π_I via extracting properties of some (heuristically computed) TD T_{Π_I}
2. . . . and let Clasp do all the work ;-)

Goals

- Derive a method, which goes beyond a simple variable ordering
- Maybe even get better performance with (still decomposable!) instances of higher treewidth

Step by step ...

Let's just merge both worlds?

1. Improve Clasp by exploiting the structure of problem instance Π_I via extracting properties of some (heuristically computed) TD T_{Π_I}
2. ... and let Clasp do all the work ;-)

Goals

- Derive a method, which goes beyond a simple variable ordering
- Maybe even get better performance with (still decomposable!) instances of higher treewidth
- Result shall be useable in a portfolio-based solver
- Exploit performance of Clasp

Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)

Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values

Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values
- produces varying runtime

[a] Generalization not yet implemented

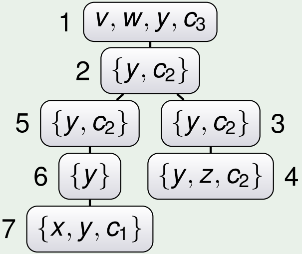
Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values
- produces varying runtime

[a] Generalization not yet implemented




```
_h(true(x), init, 7).  _h(true(y), init, 7).
```

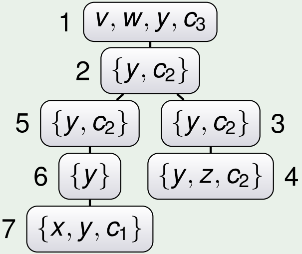
Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values
- produces varying runtime

[a] Generalization not yet implemented




```
_h(true(x), init, 7).   _h(true(y), init, 7).
_h(true(z), init, 4).
```

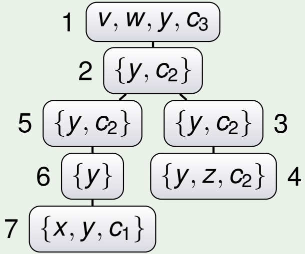
Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values
- produces varying runtime

[a] Generalization not yet implemented




```
_h(true(x), init, 7).  _h(true(y), init, 7).
_h(true(z), init, 4).
_h(true(v), init, 1).  _h(true(w), init, 1).
```

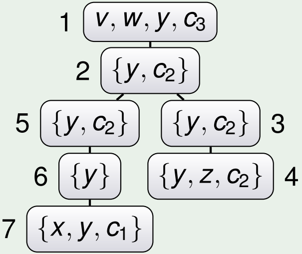
Baby steps

What has been going on?

Current prototype

- works for graph instances[a] (decompose input graph)
- modifies Clasp via providing heuristic (`_h`) values
- uses only a static initial modification of heuristic values
- produces varying runtime

[a]Generalization not yet implemented




```
_h(true(x), init, 3).  _h(true(y), init, 3).
_h(true(z), init, 2).
_h(true(v), init, 1).  _h(true(w), init, 1).
```

Troubles

Why this is not so easy . . .

1. Known problems
   - Runtime depends on the computed TD
   - TD of smallest width is not always the best one

     Abseher, Dusberger, Musliu and Woltran: Improving the Efficency of Dynamic Programming on Tree Decompositions via Machine Learning, IJCAI 2015
   - Modifying `level` via `_h` is too much
2. Does the TD actually help?
   - Same effect by arbitrary variable ordering?
   - . . .

Troubles

Why this is not so easy . . .

1. Known problems
   - Runtime depends on the computed TD
   - TD of smallest width is not always the best one

     Abseher, Dusberger, Musliu and Woltran: Improving the Efficency of Dynamic Programming on Tree Decompositions via Machine Learning, IJCAI 2015

   - Modifying `level` via `_h` is too much
2. Does the TD actually help?
   - Same effect by arbitrary variable ordering?
   - . . .

Why we require your expertise

- Look into Clasp to find out why some heuristic configurations work well and others don't
- Also add dynamics by modifying `factor`

Troubles

Why this is not so easy . . .

1. Known problems
   - Runtime depends on the computed TD
   - TD of smallest width is not always the best one

     Abseher, Dusberger, Musliu and Woltran: Improving the Efficency of Dynamic Programming on Tree Decompositions via Machine Learning, IJCAI 2015
   - Modifying `level` via `_h` is too much
2. Does the TD actually help?
   - Same effect by arbitrary variable ordering?
   - . . .

Why we require your expertise

- Look into Clasp to find out why some heuristic configurations work well and others don't
- Also add dynamics by modifying `factor`
- Maybe later even modify Clasps interna

Outline

Summary

- D-FLAT allows to specify DP algorithms on tree decompositions using ASP
- Combinatorial explosion is bounded by the treewidth
- The idea is to guide Clasp towards DP on TDs
  1. via heuristics
  2. via direct implementation
  3. ...
- It might help to take a deep breath and look inside Clasp

Thanks to Sebastian Ordyniak for his slides!