Dynamic Programming on Tree Decompositions in Practice

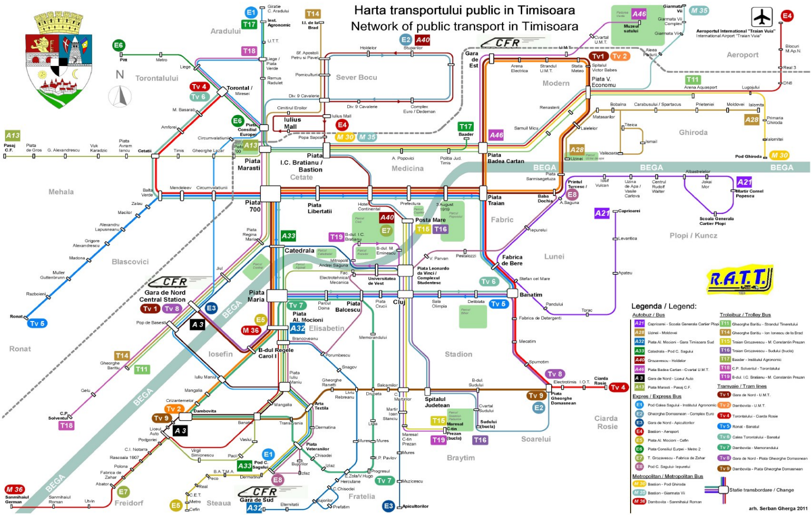
Some Lessons Learned

Stefan Woltran

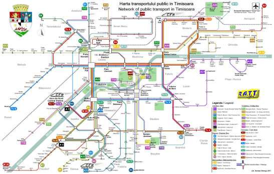
TU Wien (Vienna University of Technology)

Sept 23, 2014

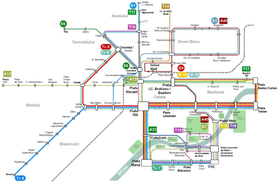
Graphs are Everywhere ...

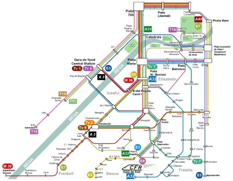


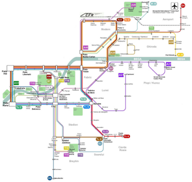
Let’s Decompose them ...



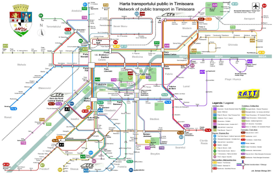
$\Longrightarrow$



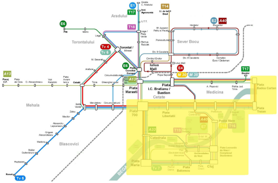


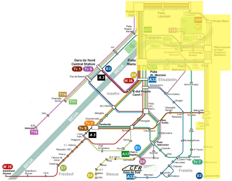


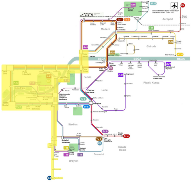
Let’s Decompose them ...



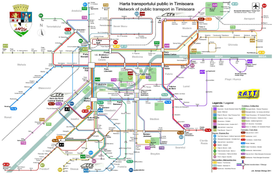
$\Longrightarrow$



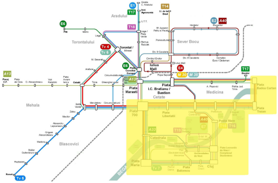


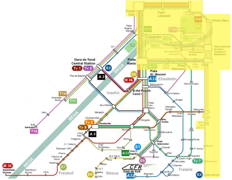


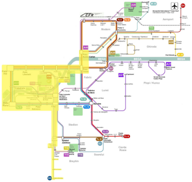
Let’s Decompose them ...

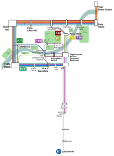


$\Longrightarrow$

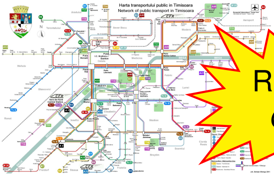






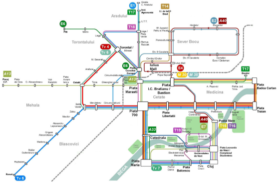


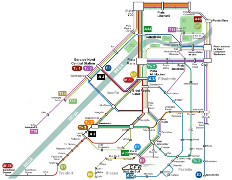
Let's Decompose them ...

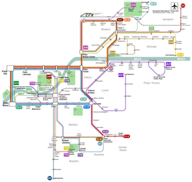


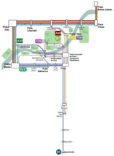
Runtime:
$\mathcal{O}(2^n)$

$\Longrightarrow$

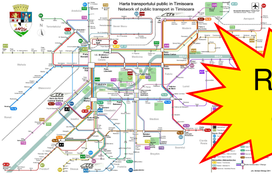






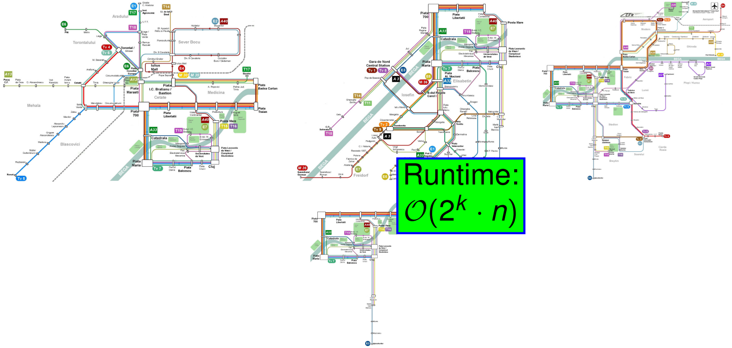


Let's Decompose them ...



Runtime:
$\mathcal{O}(2^n)$

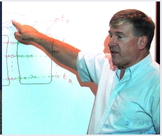
$\Longrightarrow$



Runtime:
$\mathcal{O}(2^k \cdot n)$

The Whole Story in 3 Minutes ...

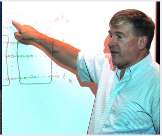
Tree Decomposition and Treewidth

By-product in the theory of graph minors due to Robertson and Seymour (1984); similar notions appeared even earlier (Bertelè and Brioschi, 1972; Halin, 1976).

The Whole Story in 3 Minutes ...

Tree Decomposition and Treewidth



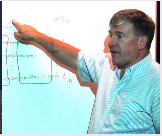
By-product in the theory of graph minors due to Robertson and Seymour (1984); similar notions appeared even earlier (Bertelè and Brioschi, 1972; Halin, 1976).

Courcelle's Theorem (1990)

Any property of finite structures which is definable in MSO can be decided in time $O(f(k) \cdot n)$ where n is the size of the structure and k is its treewidth.

The Whole Story in 3 Minutes ...

Tree Decomposition and Treewidth

By-product in the theory of graph minors due to Robertson and Seymour (1984); similar notions appeared even earlier (Bertelè and Brioschi, 1972; Halin, 1976).

Courcelle's Theorem (1990)

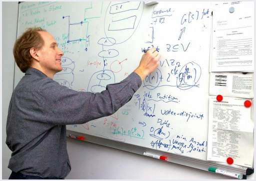
Any property of finite structures which is definable in MSO can be decided in time $O(f(k) \cdot n)$ where n is the size of the structure and k is its treewidth.

SEQUOIA (2011)

A system developed by Rossmanith's group at RWTH Aachen; SEQUOIA takes a graph and MSO description of problem and does decomposition and dynamic programming “inside”.

The Whole Story in 3 Minutes ...

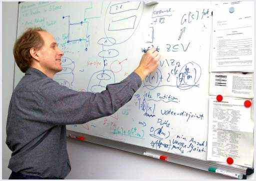
But ...



"*. . . rather than synthesizing methods indirectly from Courcelle's Theorem, one could attempt to develop practical direct methods.*" (Niedermeier, 2006)

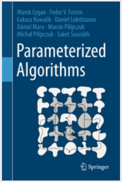
The Whole Story in 3 Minutes ...

But ...



"*...rather than synthesizing methods indirectly from Courcelle's Theorem, one could attempt to develop practical direct methods.*" (Niedermeier, 2006)

... and, more recently ...



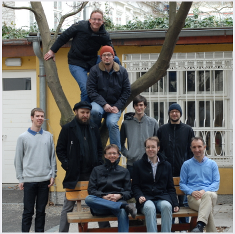
"*Courcelle's theorem [...] should be regarded primarily as classification tool, whereas designing efficient dynamic programming routines on tree decompositions requires 'getting your hands dirty' and constructing the algorithm explicitly.* " (Cygan et al., 2015)

The Whole Story in 3 Minutes ...

Our Vision

A system that

- supports declarative specifications of dynamic programming on tree decompositions
- performs reasonably efficient
- bothers the user only with the actual algorithm design



Quick thanks to all collaborators...

Michael Abseher, *Bernhard Bliem*, *Günther Charwat*, Frederico Dusberger, Johannes Fichte, *Markus Hecher*, *Marius Moldovan*, Michael Morak, Nysret Musliu and Reinhard Pichler.

Outline

Treewidth

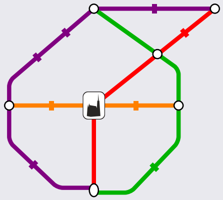
- Some graphs are more “tree-like” than others.
- Treewidth measures “tree-likeness”.
  - Trees have treewidth 1.
  - The higher the treewidth, the more complex the graph.
- Often “easy on trees” implies “easy on tree-like graph”.
  - Many problems are fixed-parameter tractable w.r.t. treewidth w, i.e. can be decided in $O(2^w \cdot n)$.
  - That is, they become easy when putting a bound on the treewidth.

Treewidth

- Some graphs are more “tree-like” than others.
- Treewidth measures “tree-likeness”.
  - Trees have treewidth 1.
  - The higher the treewidth, the more complex the graph.
- Often “easy on trees” implies “easy on tree-like graph”.
  - Many problems are fixed-parameter tractable w.r.t. treewidth w, i.e. can be decided in $O(2^w \cdot n)$.
  - That is, they become easy when putting a bound on the treewidth.
- It works for many hard problems.
- Real-world applications often have small treewidth.

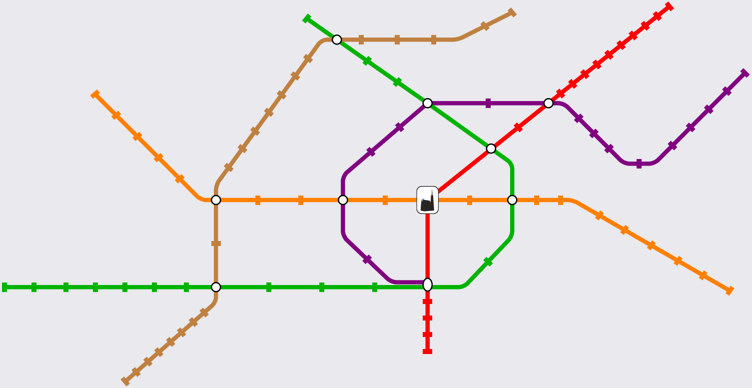
Treewidth (ctd.)

Example: Treewidth 3.



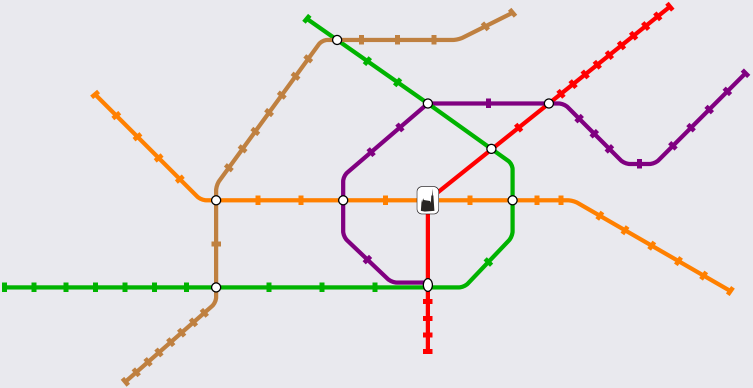
Treewidth (ctd.)

Example: Treewidth 3. Still.



Treewidth (ctd.)

Example: Treewidth 3. Still.



Treewidth is defined in terms of tree decompositions.

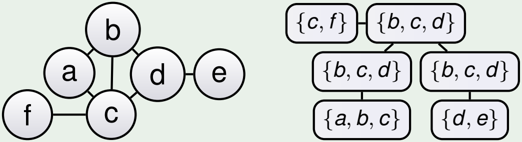
Tree Decompositions

Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some *bag*.
2. For each edge, there is a bag containing both endpoints.
3. If vertex v appears in bags of nodes n_0 and n_1, then v is also in the bag of each node on the path between n_0 and n_1.

Example



- *Decomposition width*: size of the largest bag (minus 1)
- Treewidth: minimum width over all possible tree decompositions

Tree Decompositions (ctd.)

Constructing a Tree Decomposition

- Any graph admits at least a trivial tree decomposition.
- But finding a *minimum-width* tree decomposition is difficult.
- However, there are good heuristics!

Tree Decompositions (ctd.)

Constructing a Tree Decomposition

- Any graph admits at least a trivial tree decomposition.
- But finding a *minimum-width* tree decomposition is difficult.
- However, there are good heuristics!

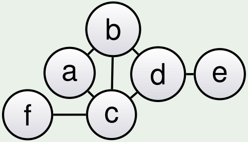
Dynamic Programming on Tree Decompositions

- Traverse tree decomposition from leaves to root and compute partial solutions in each node by
- suitably combining partial solutions of child nodes.
- Algorithms often exponential only in decomposition width but *linear* in the input size.

Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

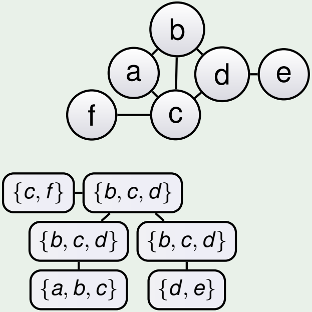


Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance

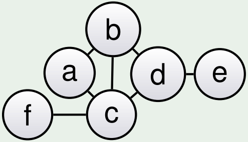


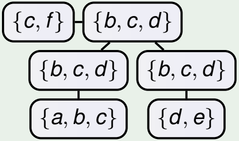
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

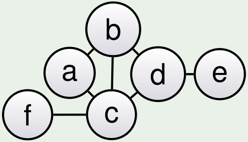
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

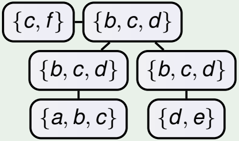
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

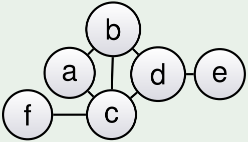
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

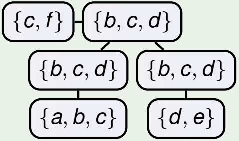
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

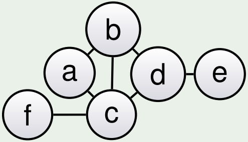
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

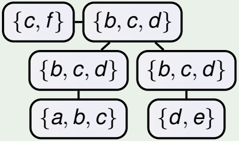
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems

	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

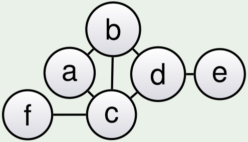
	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

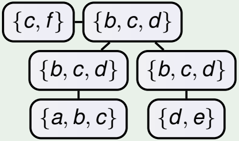
	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0



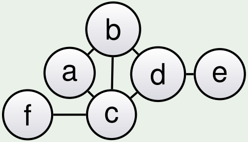


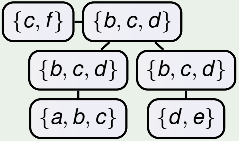
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

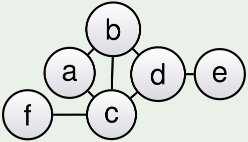
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

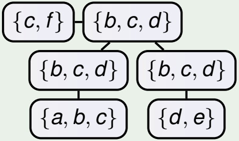
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

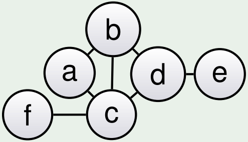
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

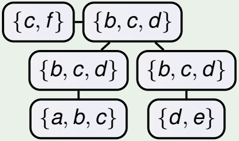
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

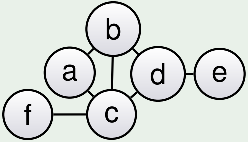
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

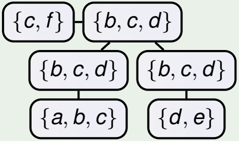
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

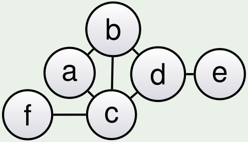
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

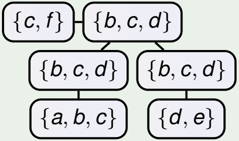
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

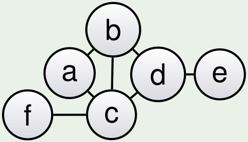
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

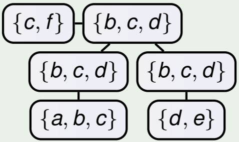
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

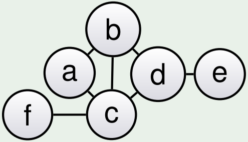
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

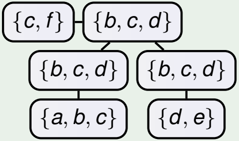
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

1. Decompose instance
2. Solve partial problems





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

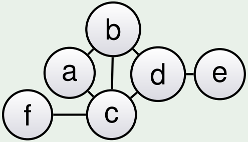
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

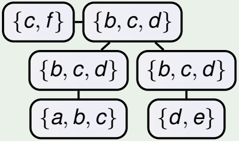
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

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





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

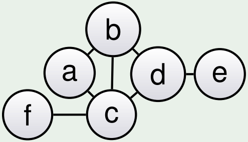
	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

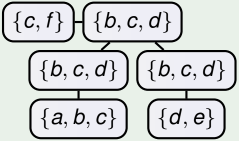
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:

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





	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

Outline

D-FLAT

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

What does it do?

1. Constructs a tree decomposition of the input structure
2. In each node: Executes user-supplied logic program that describes the dynamic programming algorithm
3. Decides the problem (or materializes solutions)

Properties

- Relies on Answer-Set Programming (ASP) paradigm
- Users only need to write an ASP program
- Communication with the user's program via special predicates
- Uses external libraries for ASP solving, tree decomposition, etc.

Answer-Set Programming (ASP)

- Successful declarative programming paradigm in AI
- Has its roots in nonmonotonic reasoning and datalog
- Systems have been developed since the late 90s
- Applications in many diverse areas
  - Bio-Informatics
  - Diagnosis
  - Configuration
  - Linguistics
  - ...

Answer Set Programming (ctd.)

- ASP provides a convenient Guess & Check method
  1. Guess a candidate solution non-deterministically
  2. Check if the candidate is indeed a solution
- Any search problem in NP (even in Σ_2^P) can be solved with ASP

Minimum Independent Dominating Set

Input:
Graph $G = (V, E)$ via predicates `vertex/1` and `edge/2`.

```
{ in(X) : vertex(X) }.
← in(X), in(Y), edge(X,Y).
dominated(X) ← in(Y), edge(Y,X).
← vertex(X), not in(X), not dominated(X).
#minimize{ 1,X : in(X) }.
```

Why ASP for Dynamic Programming?

- Compact declarative description of combinatorial problems
- ASP typically delivers *all* solutions
- Powerful systems available

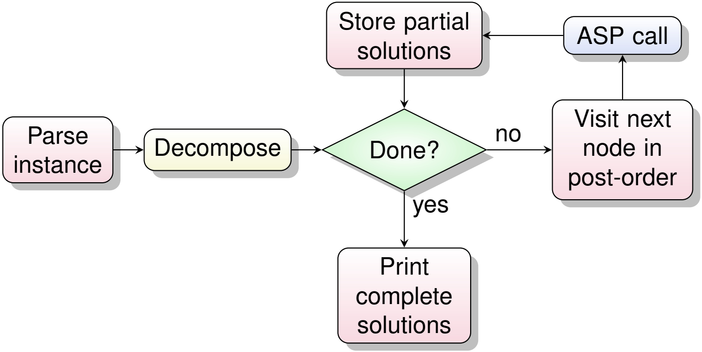
Practical Observation:

- If ASP is well suited for a problem, it is usually also well suited for the subproblems required in a decomposition

  $\Longrightarrow$ allows for rapid prototyping of dynamic programming on tree decompositions

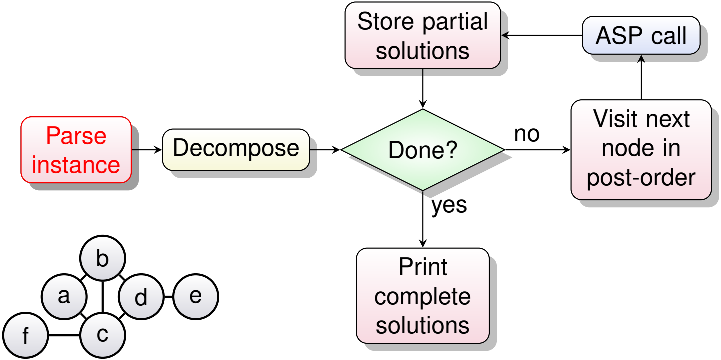
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



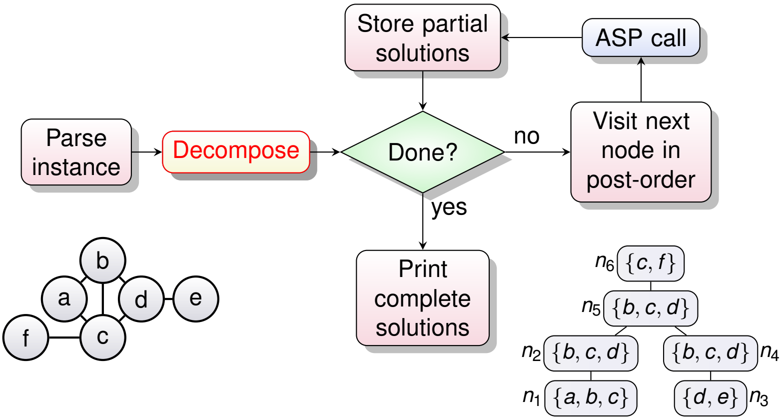
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



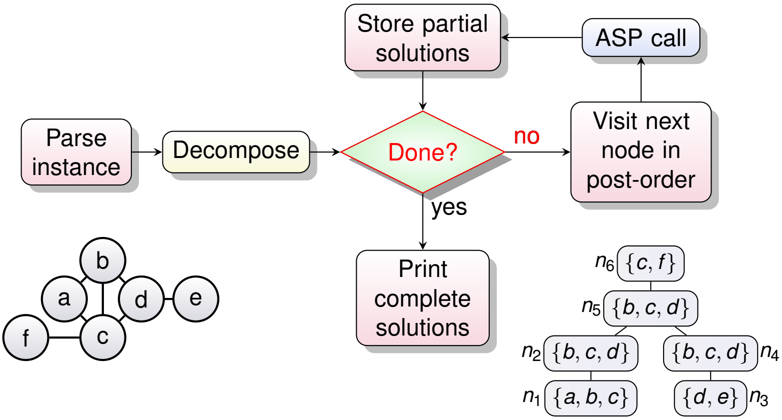
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



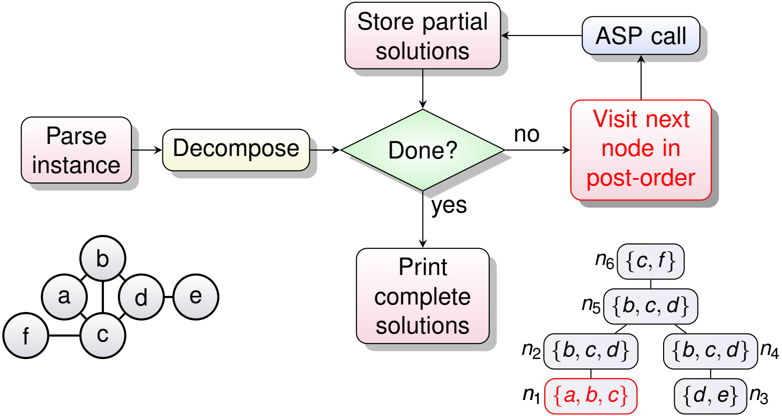
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



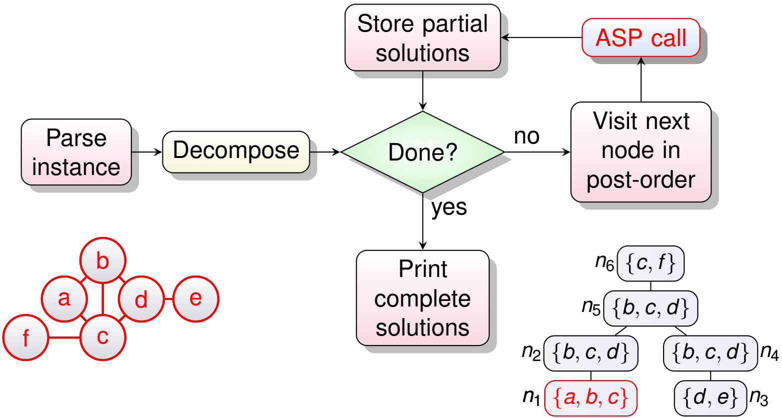
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET

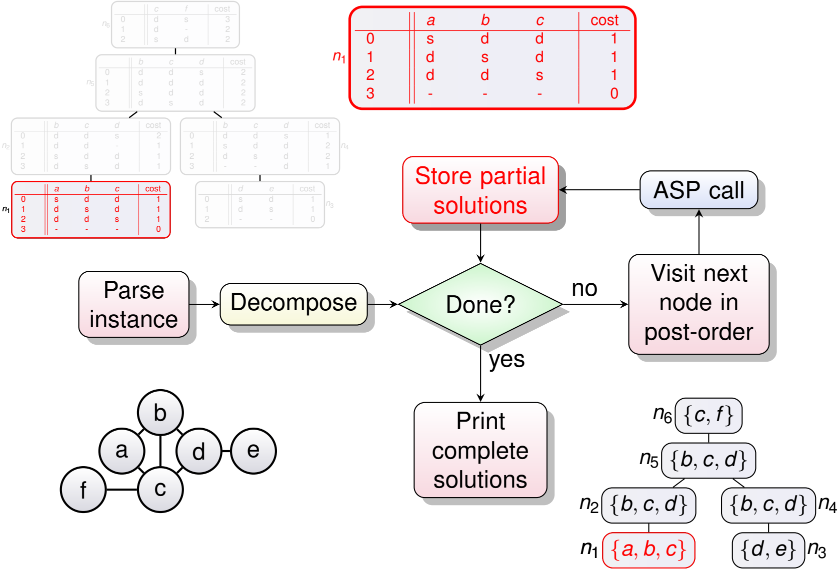


D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET

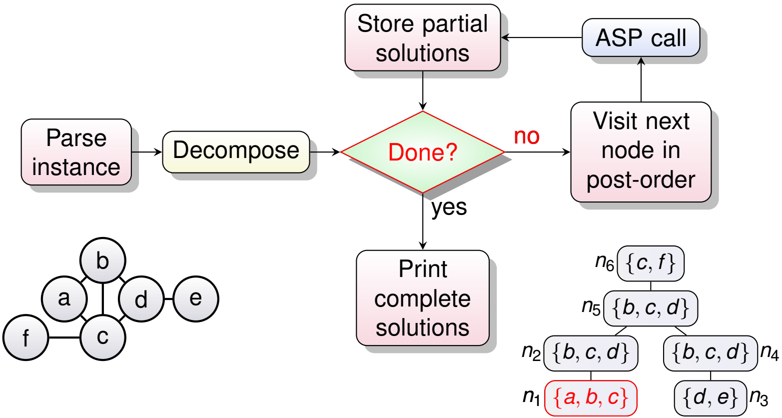
n_1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0



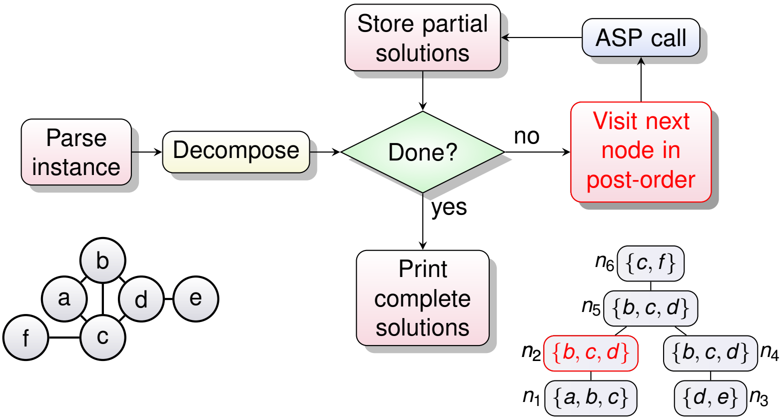
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



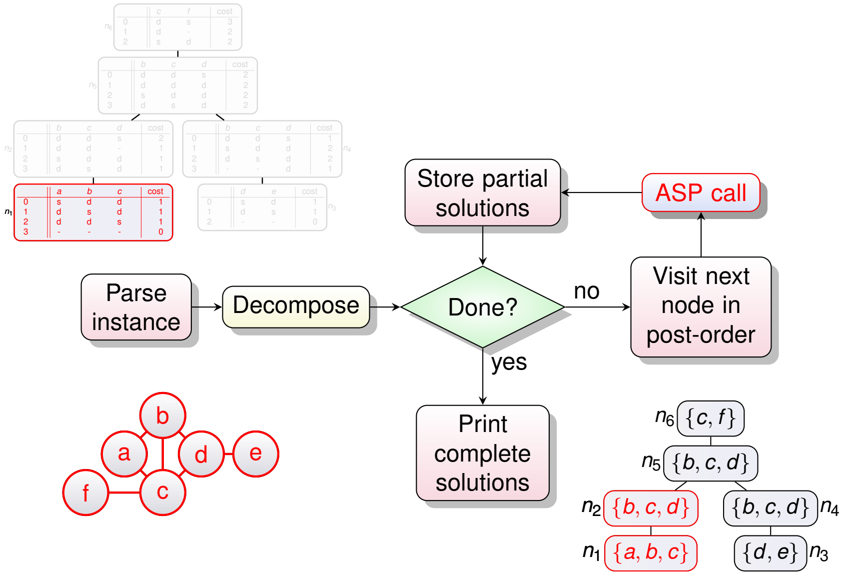
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



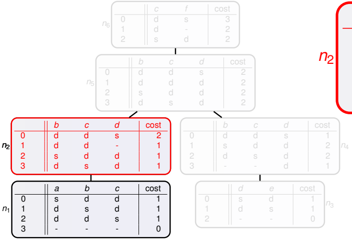
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



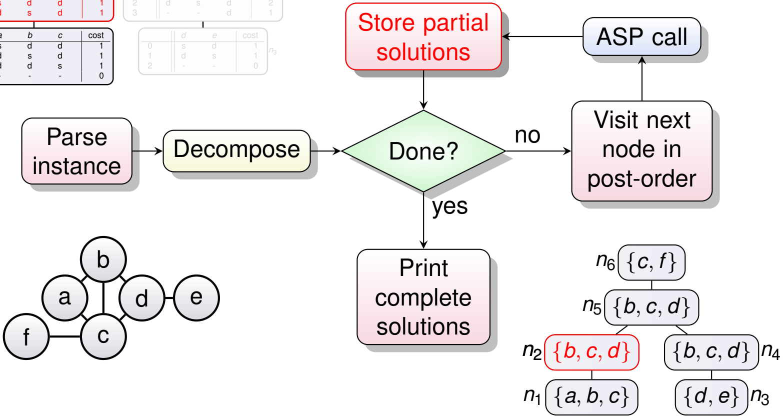
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



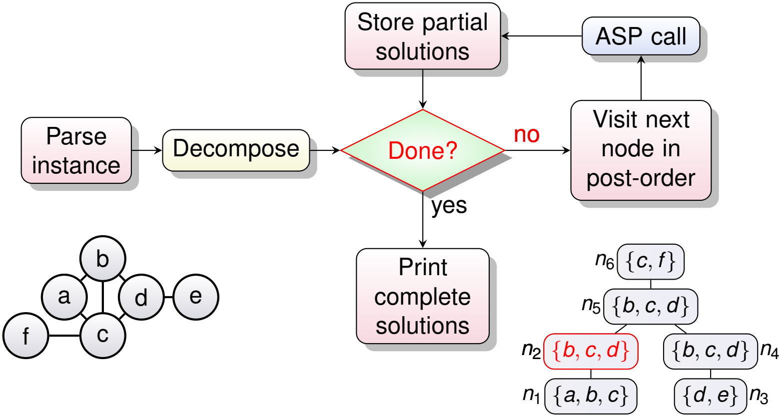
n_2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1



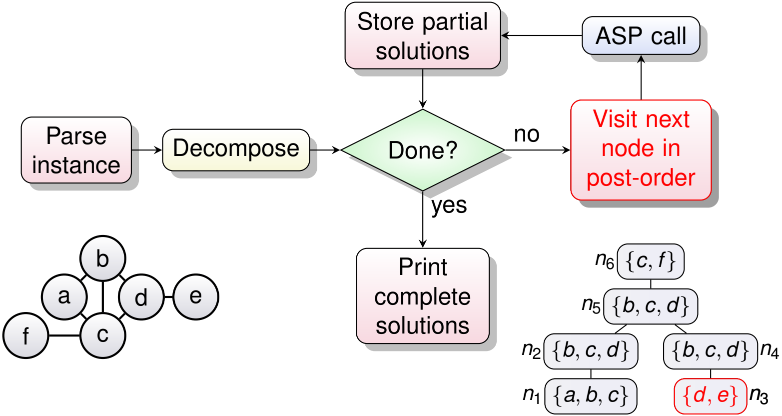
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



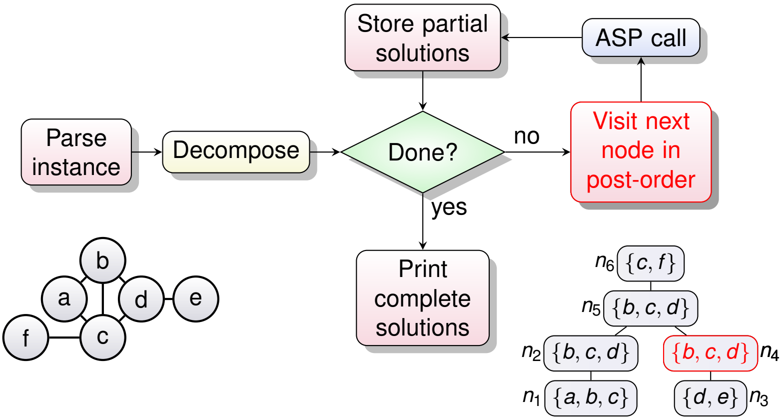
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



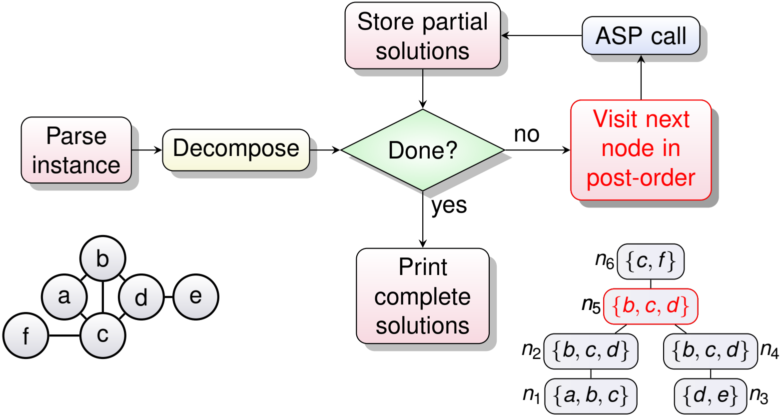
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET

n_6

	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

n_5

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

n_2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

n_4

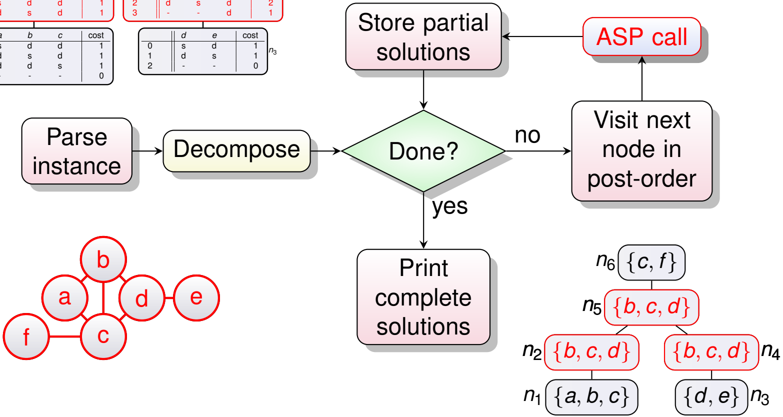
	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

n_1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

n_3

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET

n_6

	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

n_5

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

n_2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

n_4

	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

n_1

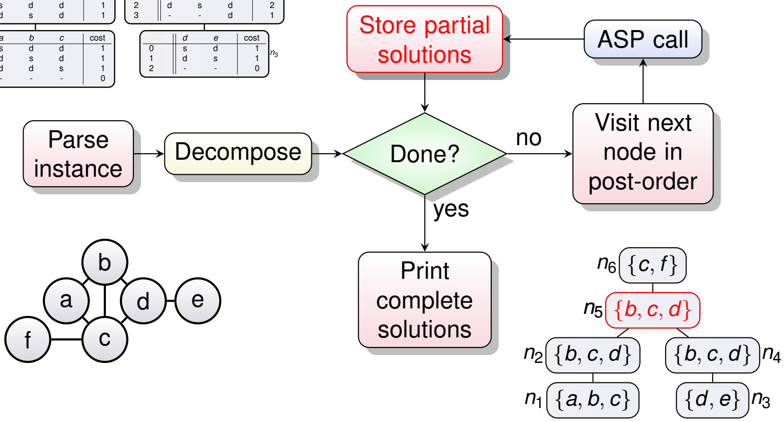
	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

n_3

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0

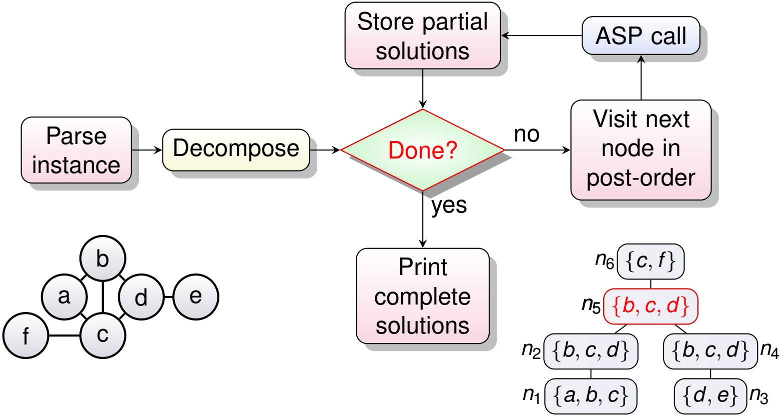
n_5

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2



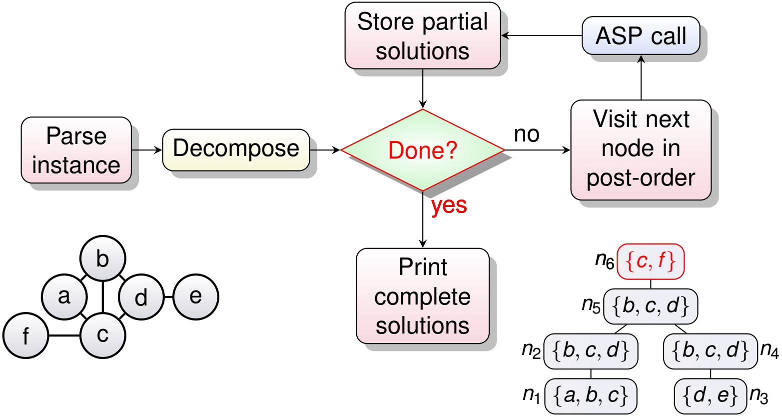
D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET



D-FLAT at Work

Illustrated by means of INDEPENDENT DOMINATING SET

n_6

	c	f	cost
0	d	s	3
1	d	-	2
2	s	d	2

n_5

	b	c	d	cost
0	d	d	s	2
1	d	d	d	2
2	s	d	d	2
3	d	s	d	2

n_2

	b	c	d	cost
0	d	d	s	2
1	d	d	-	1
2	s	d	d	1
3	d	s	d	1

n_4

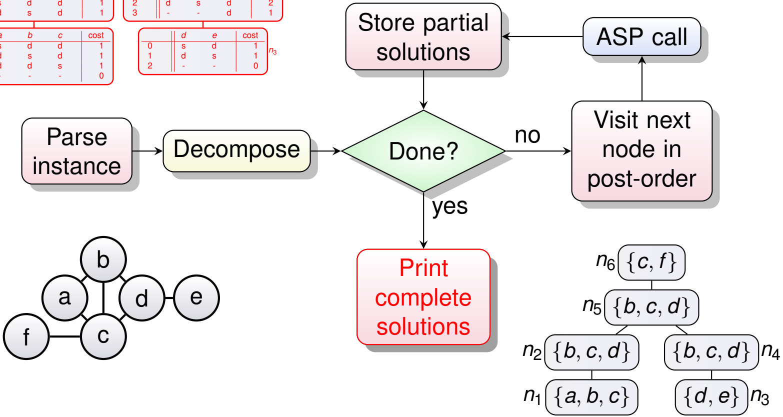
	b	c	d	cost
0	d	d	s	1
1	s	d	d	2
2	d	s	d	2
3	-	-	d	1

n_1

	a	b	c	cost
0	s	d	d	1
1	d	s	d	1
2	d	d	s	1
3	-	-	-	0

n_3

	d	e	cost
0	s	d	1
1	d	s	1
2	-	-	0



D-FLAT at Work (ctd.)

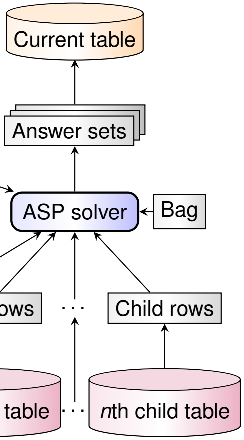
Illustrated by means of INDEPENDENT DOMINATING SET

User-supplied program

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
← extend(R1;R2), childItem(R1,in(X)),
              not childItem(R2,in(X)).
← removed(X), extend(R),
   not childItem(R,in(X)), not childItem(R,dom(X)).
item(in(X))    ← extend(R), childItem(R,in(X)),
                 current(X).
item(dom(X))   ← extend(R), childItem(R,dom(X)),
                 current(X).
{ item(in(X)) : introduced(X) }.
item(dom(X))   ← item(in(Y)), edge(Y,X),
                 current(X).
← edge(X,Y), item(in(X;Y)).
```

Instance

```
vertex(a;b;c;d;e).
edge(a,b). edge(a,c). edge(b,c).
edge(b,d). edge(c,d). edge(d,e).
```



Another Example: Boolean Satisfiability (SAT)

Although SAT is not a graph problem, we can still decompose it.

- Use the incidence graph of the formula:
- One vertex for each variable and each clause.
- Edge (v, c) if variable v occurs in clause c.

D-FLAT encoding

```
% Extend compatible rows from child nodes.
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
← extend(R;S), atom(A), childItem(R,A), not childItem(S,A).
% Retain extended assignment and guess on introduced atoms.
item(X) ← extend(R), childItem(R,X), current(X).
{ item(A) : atom(A), introduced(A) }.
% Additional clauses might have become satisfied.
item(C) ← current(C;A), pos(C,A), item(A).
item(C) ← current(C;A), neg(C,A), not item(A).
% Kill assignments that leave some clause unsatisfied.
← clause(C), removed(C), extend(R), not childItem(R,C).
```

What about Performance?

"About your cat, Mr. Schrödinger—I have good news and bad news."

What about Performance?

"About your cat, Mr. Schrödinger—I have good news and bad news."

Time for a Demo!

D-FLAT Features

- Special predicates in LP allow the user to delegate tasks to D-FLAT
- Different modes for decision, counting, optimization and enumeration problems
- Support of different normalizations of the decomposition
- Support of hypergraphs
- “Default Join”
- Two modes for storing and handling solutions of subproblems

D-FLAT Features (ctd.)

"Table-Mode" for Problems in NP

- We compute a table at each node
- We guess rows using ASP
- ... yields all accepting computation branches of an NTM

D-FLAT Features (ctd.)

“Table-Mode” for Problems in NP

- We compute a table at each node
- We guess rows using ASP
- ... yields all accepting computation branches of an NTM

“Tree-Mode” for Problems in the Polynomial Hierarchy

- We compute a tree at each node
- We guess branches using ASP
- ... yields all accepting computation branches of an ATM (D-FLAT appropriately handles the trees inside).

General Applicability

Recall Courcelle's theorem

Any problem definable in MSO can be solved in linear time
on graphs of bounded treewidth.

It is such problems that decomposition is usually employed for.

General Applicability

Recall Courcelle's theorem

Any problem definable in MSO can be solved in linear time on graphs of bounded treewidth.

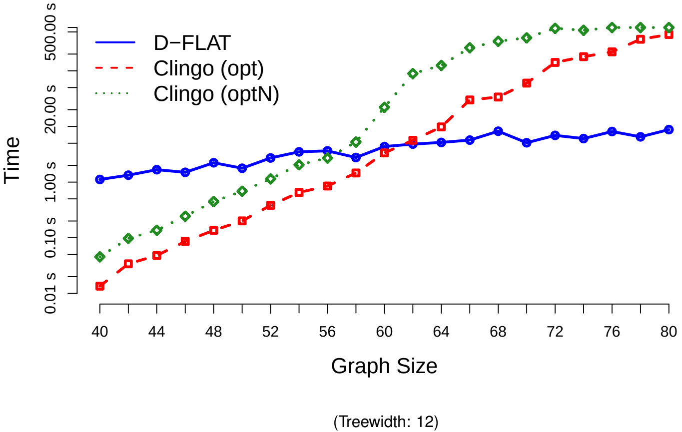
It is such problems that decomposition is usually employed for.

Good news

D-FLAT can be effectively used for all such problems

- It can evaluate MSO formulas in linear time if the treewidth is bounded
- Encoding for MSO is not overly complicated (approx. 30 lines of ASP code)
- However, expressing the problem at hand via MSO and then feed to D-FLAT is not recommended
  - instead, D-FLAT is designed for problem-specific dynamic programming solutions

Experimental Evaluation: #Maximum Independent Set



Comparison between D-FLAT and the ASP solver clingo 4.3.0

A First Conclusion

Summary

- Hard problems often become tractable when instances exhibit certain properties.
- Especially bounded treewidth often leads to tractability (problems expressible in MSO).
- The "D-FLAT" method [TPLP 2012, JELIA 2014] allows to specify dynamic programming algorithms in a declarative way.
  - This works for all MSO-definable problems [IPEC 2013]

Next Steps

- additional D-FLAT features for arithmetics
- lazy D-FLAT

Outline

Outline

Motivation

Lesson Learnt

- DP algorithms often show recurring patterns ...
  - In particular, DP algorithms for problems on the 2^{nd} level of PH often require treatment of subset-minimization or maximization
  - This leads to quite involved DP specifications.

Motivation

Lesson Learnt

- DP algorithms often show recurring patterns ...
  - In particular, DP algorithms for problems on the 2^{nd} level of PH often require treatment of subset-minimization or maximization
  - This leads to quite involved DP specifications.

Goals

- Provide a simple mechanism for the user
- Improve performance for 2^{nd}-level problems

Motivation (ctd.)

D-FLAT program for MINSAT

```
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).
```

D-FLAT^2

DP **F**ramework with **L**ocal Execution of **A**SP on **T**Ds for **2**nd-Level Subset-Optimizations

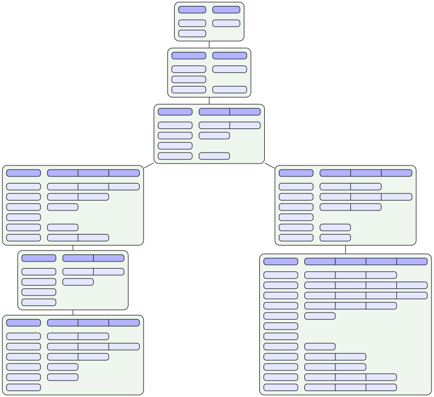
What does it do?

1. Constructs a tree decomposition of the input structure
2. First pass executes user-supplied program and stores partial solutions (as before)
3. Second pass (in each node)
   - Executes our native subset optimization algorithm
   - Stores counter candidate pointers by reusing partial solutions

Properties

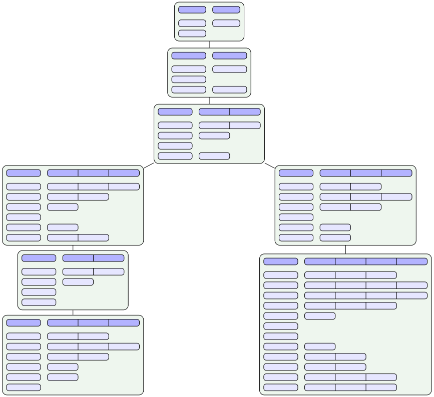
- Users only need to write an ASP program
- Subset optimization on user-specified items via optItem/1 done "inside"

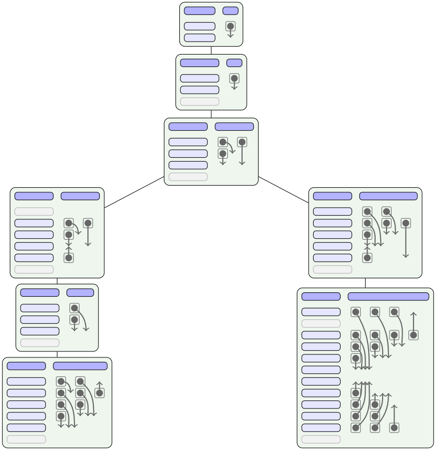
Comparison



Dynamic Programming in D-FLAT

Comparison





Dynamic Programming in D-FLAT   Dynamic Programming in D-FLAT^2

D-FLAT^2 (ctd.)

Recall encoding for SAT

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
← extend(R;S), atom(A), childItem(R,A), not childItem(S,A).
item(X) ← extend(R), childItem(R,X), current(X).
{ item(A) : atom(A), introduced(A) }.
item(C) ← current(C;A), pos(C,A), item(A).
item(C) ← current(C;A), neg(C,A), not item(A).
← clause(C), removed(C), extend(R), not childItem(R,C).
```

D-FLATˆ2 (ctd.)

Recall encoding for SAT

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
← extend(R;S), atom(A), childItem(R,A), not childItem(S,A).
item(X) ← extend(R), childItem(R,X), current(X).
{ item(A) : atom(A), introduced(A) }.
item(C) ← current(C;A), pos(C,A), item(A).
item(C) ← current(C;A), neg(C,A), not item(A).
← clause(C), removed(C), extend(R), not childItem(R,C).
```

For MINSAT, we just need to add

```
optItem(X) ← item(X), atom(X).
```

D-FLAT^2 (ctd.)

Recall encoding for SAT

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
← extend(R;S), atom(A), childItem(R,A), not childItem(S,A).
item(X) ← extend(R), childItem(R,X), current(X).
{ item(A) : atom(A), introduced(A) }.
item(C) ← current(C;A), pos(C,A), item(A).
item(C) ← current(C;A), neg(C,A), not item(A).
← clause(C), removed(C), extend(R), not childItem(R,C).
```

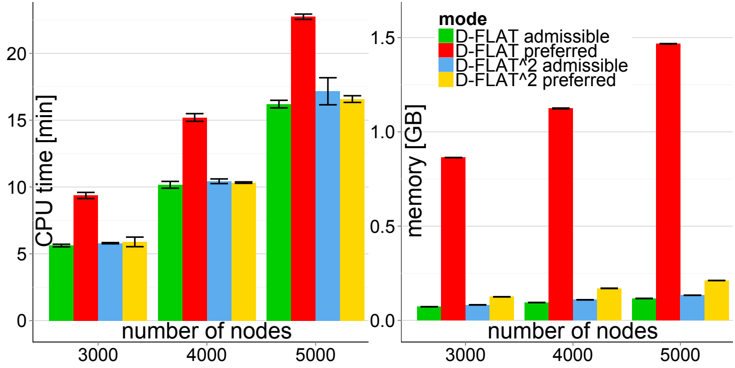
For MINSAT, we just need to add

```
optItem(X) ← item(X), atom(X).
```

For Circumscription, we just need to add

```
optItem(X) ← item(X), minatom(X).
optItem(t(X)) ← item(X), varyatom(X).
optItem(f(X)) ← not item(X), varyatom(X).
```

D-FLAT vs. D-FLAT^2



(Counting, Treewidth: 4)

Comparison between D-FLAT and D-FLAT^2

D-FLATˆ2 – Discussion

Summary

- D-FLATˆ2 [ASPOCP 2015] is an extension of D-FLAT for rapid prototyping of 2^{nd}-level DP algorithms on tree decompositions involving subset optimization
- Preliminary results indicate that optimization is almost for free in case of small treewidth

Next Steps

- D-FLATˆ2 $\Longrightarrow$ D-FLATˆn (generalize D-FLATˆ2 to handle problems on the n^{th} level of the polynomial hierarchy)
- Implement further problems and improve D-FLATˆ2 towards more general specifications of optimization task

Outline

Motivation

Lesson Learnt

- Bottleneck of D-FLAT (resp. DP in general): size of tables
  - size grows exponentially with treewidth
- Can we find a match to logic (truth-table vs. formula)?

Motivation

Lesson Learnt

- Bottleneck of D-FLAT (resp. DP in general): size of tables
  - size grows exponentially with treewidth
- Can we find a match to logic (truth-table vs. formula)?

Idea

- Employ Binary Decision Diagrams (BDDs):
  - compact representation of truth-tables
  - can be treated like formulas

Motivation

Lesson Learnt

- Bottleneck of D-FLAT (resp. DP in general): size of tables
  - size grows exponentially with treewidth
- Can we find a match to logic (truth-table vs. formula)?

Idea

- Employ Binary Decision Diagrams (BDDs):
  - compact representation of truth-tables
  - can be treated like formulas

Goals

- Understand feasibility of this approach
- Understand limits in describing DPs as formula manipulation

Binary Decision Diagrams

Example (OBDD representation)

Let formula $\varphi = (a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$.

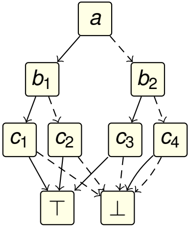


Figure: OBDD of φ.

Binary Decision Diagrams

Example (OBDD representation)

Let formula $\varphi = (a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$.

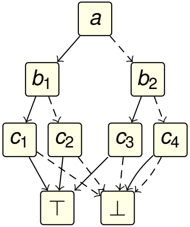


Figure: OBDD of φ.

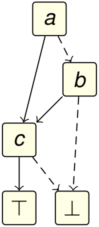


Figure: ROBDD of φ.

Binary Decision Diagrams (ctd.)

Advantages of BDDs:

- Well-studied and mature concepts that are successfully applied to planning, verification, etc.
- Efficient implementations available
- Delegate burden of memory-efficient implementation to data structure
- Logic-based algorithm specification

Comparison

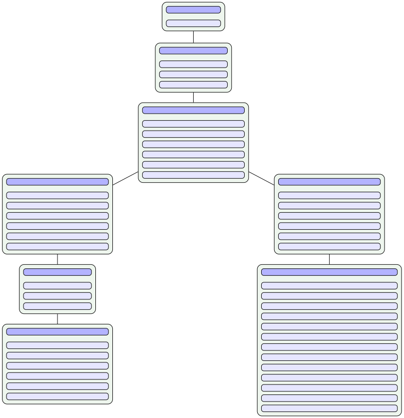


Table-based Dynamic Programming

Comparison

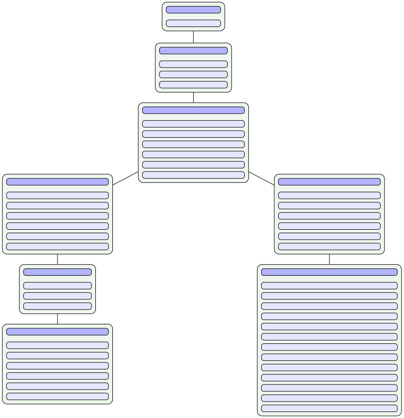
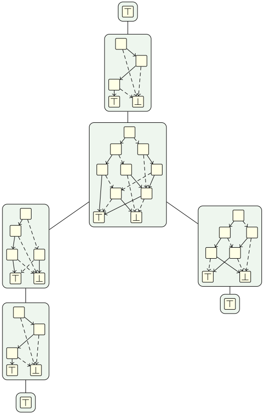


Table-based Dynamic Programming



BDD-based Dynamic Programming

DP of Independent Dominating Set (on Digraphs) via BDDs

$$\mathcal{B}_t^I = \bigwedge_{(x,y) \in E_t} (\neg i_x \vee \neg i_y) \wedge \bigwedge_{y \in V_t} \left(d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right)$$

DP of Independent Dominating Set (on Digraphs) via BDDs

$$\mathcal{B}_t^l = \bigwedge_{(x,y)\in E_t} (\neg i_x \vee \neg i_y) \wedge \bigwedge_{y\in V_t} \left(d_y \leftrightarrow \bigvee_{(x,y)\in E_t} i_x\right)$$

$$\mathcal{B}_t^i = \exists D'_{t'} \Bigg[\mathcal{B}_{t'}[D_{t'}/D'_{t'}] \wedge \bigwedge_{(u,y)\in E_t} (\neg i_u \vee \neg i_y) \wedge \left(d_u \leftrightarrow \bigvee_{(x,u)\in E_t} i_x\right) \wedge$$

$$\bigwedge_{\substack{(u,y)\in E_t \wedge \\ u\neq y}} \left(d_y \leftrightarrow d'_y \vee i_u\right) \wedge \bigwedge_{y\in V_t \wedge (u,y)\notin E_t} \left(d_y \leftrightarrow d'_y\right)\Bigg]$$

DP of Independent Dominating Set (on Digraphs) via BDDs

$$\mathcal{B}_t^l = \bigwedge_{(x,y)\in E_t} (\neg i_x \vee \neg i_y) \wedge \bigwedge_{y\in V_t} \Big(d_y \leftrightarrow \bigvee_{(x,y)\in E_t} i_x \Big)$$

$$\mathcal{B}_t^i = \exists D'_{t'} \Big[\mathcal{B}_{t'}[D_{t'}/D'_{t'}] \wedge \bigwedge_{(u,y)\in E_t} (\neg i_u \vee \neg i_y) \wedge \Big(d_u \leftrightarrow \bigvee_{(x,u)\in E_t} i_x \Big) \wedge \bigwedge_{\substack{(u,y)\in E_t \wedge \\ u\neq y}} (d_y \leftrightarrow d'_y \vee i_u) \wedge \bigwedge_{y\in V_t \wedge (u,y)\notin E_t} (d_y \leftrightarrow d'_y) \Big]$$

$$\mathcal{B}_t^r = \mathcal{B}_{t'}[i_u/\top, d_u/\bot] \vee \mathcal{B}_{t'}[i_u/\bot, d_u/\top]$$

DP of Independent Dominating Set (on Digraphs) via BDDs

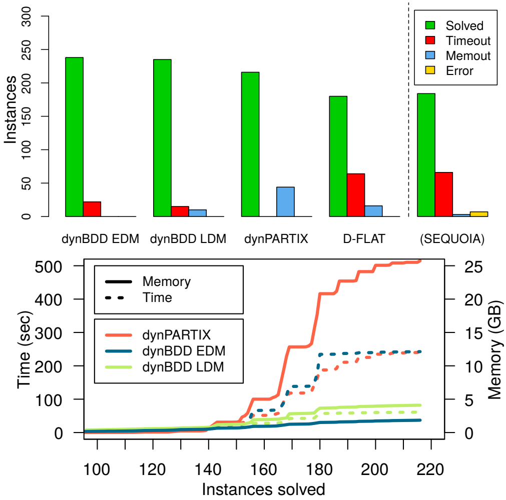
$$\mathcal{B}_t^l = \bigwedge_{(x,y)\in E_t} (\neg i_x \vee \neg i_y) \wedge \bigwedge_{y\in V_t} \Big(d_y \leftrightarrow \bigvee_{(x,y)\in E_t} i_x\Big)$$

$$\mathcal{B}_t^i = \exists D'_{t'} \Big[\mathcal{B}_{t'}[D_{t'}/D'_{t'}] \wedge \bigwedge_{(u,y)\in E_t} (\neg i_u \vee \neg i_y) \wedge \Big(d_u \leftrightarrow \bigvee_{(x,u)\in E_t} i_x\Big) \wedge$$
$$\bigwedge_{\substack{(u,y)\in E_t \wedge \\ u\neq y}} (d_y \leftrightarrow d'_y \vee i_u) \wedge \bigwedge_{y\in V_t \wedge (u,y)\notin E_t} (d_y \leftrightarrow d'_y)\Big]$$

$$\mathcal{B}_t^r = \mathcal{B}_{t'}[i_u/\top, d_u/\bot] \vee \mathcal{B}_{t'}[i_u/\bot, d_u/\top]$$

$$\mathcal{B}_t^j = \exists D'_t D''_t \Big[\mathcal{B}_{t'}[D_t/D'_t] \wedge \mathcal{B}_{t''}[D_t/D''_t] \wedge \bigwedge_{x\in V_t} (d_x \leftrightarrow d'_x \vee d''_x)\Big]$$

Experiments: Independent Dominating Set



Dynamic Programming with BDDs – Discussion

Summary

- dynBDD is a first prototype that performs DP algorithms on tree decompositions via manipulation of BDDs [LPNMR 2015]
- allows for realization of more advanced DP algorithms (“wild cards” etc)
- preliminary results indicate significant decrease of space used
- currently, algorithms have to be implemented in C++ on top of CUDD

Next Steps

- user front-end
- so far, methodology only tested for “table-mode”;
  generalization to arbitrary DP is also theoretically challenging

Outline

Motivation

Lesson Learnt

- Generation of decompositions rather cheap (compared to the runtime of dynamic programming)
- Shape of decomposition crucial for performance (it's not the width only!)
- Better understanding needed how "good tree decompositions" look like

Motivation

Lesson Learnt

- Generation of decompositions rather cheap (compared to the runtime of dynamic programming)
- Shape of decomposition crucial for performance (it's not the width only!)
- Better understanding needed how "good tree decompositions" look like

Goal

- Identification of features for tree decompositions (rather than on the actual input instance)
- Understand how machine learning can help us to select a good decomposition from a set of decompositions

Methodology

Given a specific problem

- Training data: 90 small random instances with rather low treewidth (10 decompositions for each instance)
- Obtain regression models (5 different methods) for ranking decompositions using specific decomposition features
- Apply model to real-world instances (treewidth up to 8)
  - Generate 10 tree decompositions per instance
  - Model selects the best-ranked decomposition

Experimental Set-Up (ctd.)

Features (Selection)

Decomposition Size:

- BagSize*
- BagSize$^*_{NL}$
- ContainerCount*
- Σ BagSize
- NodeCount

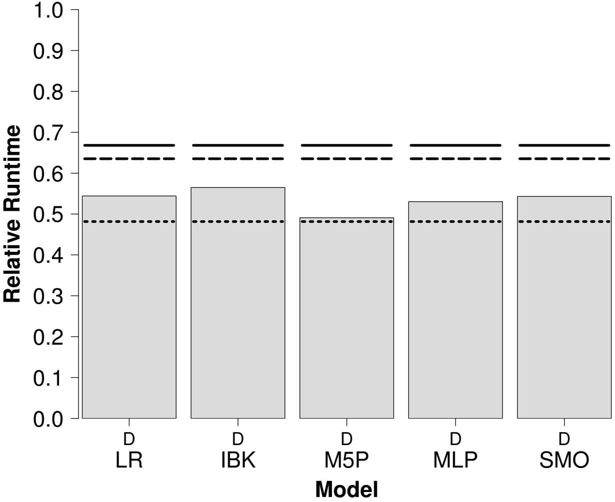
Introduce / Forget / Join / Leaf Nodes:

- Depth*
- BagSize*
- NodeCount (#)
- Percentage

Structural Features:

- JoinNodeDistance*
- ItemLifetime*
- NumberOfChildren*
- BalancednessFactor
- AdjacencyRatio*
- BagConnectednessRatio*
- NeighbourCoverageRatio*

Experimental Results



Decomposition Features – Discussion

Summary

- We conducted huge test series [IJCAI 2015] for several problems and two systems (D-FLAT and SEQUOIA)
- Feature-based ML successfully identified good decompositions
- However, crucial features are in general not problem independent

Next Steps

- We need to get a precise picture on crucial features
- Use gained insights to tailor tree decomposition heuristics

Outline

Summary

- Tree-Decompositions known as a promising tool to exploit structure in hard problems
- D-FLAT: a system for rapid prototyping of DP algorithms
  - takes care of the decomposition task
  - declarative specifications of dynamic programming via ASP
  - ASP systems used to solve subproblems
  - general applicability
  - able to outperform standard technology
- Many ongoing developments

The D-FLAT Suite

- D-FLAT System
- D-FLAT Debugger (new and improved visualization tool currently under development)
- D-FLATˆ2
- dynBDD

Ongoing + Future Work

- Automatic generation of D-FLAT code from “standard” encoding
  - D-FLATˆ2 as a first step towards a library for DP designers
- Exploit smarter ways to store solutions
  - BDDs a promising option
  - easy-to-use interface still missing
- Tailor tree decomposition heuristics
  - observation: shape of decomposition crucial for performance
  - huge test series showed the potential of ML methods
- Tighter integration of D-FLAT with ASP solvers
  - communication between D-FLAT and ASP solver is bottleneck
  - exploit recent ASP technology (“multishot solving”)

Try it out! D-FLAT is free software, available at

```
http://dbai.tuwien.ac.at/proj/dflat/
```

Try it out! D-FLAT is free software, available at

```
http://dbai.tuwien.ac.at/proj/dflat/
```

...and have fun with decompositions ...

Thanks for your attention!

Main References

ASPOCP 2015 B. Bliem, G. Charwat, M. Hecher, and S. Woltran: “D-FLAT^2: Subset minimization in dynamic programming on tree decompositions made easy”. *Proceedings ASPOCP 2015*, 2015.

JELIA 2014 M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran: “The D-FLAT system for dynamic programming on tree decompositions”. *Proceedings JELIA 2014*, pp. 558–572. Springer, 2014.

IJCAI 2015 M. Abseher, F. Dusberger, N. Musliu, and S. Woltran: “Improving the efficiency of dynamic programming on tree decompositions via machine learning”. *Proceedings IJCAI 2015*, pp. 275–282, AAAI Press, 2015.

IPEC 2013 B. Bliem, R. Pichler, and S. Woltran: “Declarative dynamic programming as an alternative realization of Courcelle’s theorem”. *Proceedings IPEC 2013*, pp. 28–40, Springer, 2013.

LPNMR 2015 G. Charwat and S. Woltran: “Efficient problem solving on tree decompositions using binary decision diagrams”. *Proceedings LPNMR 2015*, pp. 213–227, Springer, 2015.

TPLP 2012 B. Bliem, M. Morak, and S. Woltran: “D-FLAT: Declarative problem solving using tree decompositions and answer-set programming”. *Theory and Practice of Logic Programming*, vol. 12, pp. 445–464, 2012.