Dynamic Programming on Tree Decompositions in Practice

Stefan Woltran

TU Wien (Vienna University of Technology)

August 30, 2016
Graphs are Everywhere ...
Let’s Decompose them ...
Let’s Decompose them ...
Let’s Decompose them ...

Runtime: $O(2^n)$
Let's Decompose them ...
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."
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”.
“…rather than synthesizing methods indirectly from Courcelle’s Theorem, one could attempt to develop practical direct methods.” (Niedermeier, 2006)
“…rather than synthesizing methods indirectly from Courcelle’s Theorem, one could attempt to develop practical direct methods.” (Niedermeier, 2006)

“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...

Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
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 graphs”.

- Many problems are fixed-parameter tractable w.r.t. treewidth $k$, i.e. can be decided in $O(2^k \cdot n)$.
- 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 graphs”.
  - Many problems are fixed-parameter tractable w.r.t. treewidth $k$, i.e. can be decided in $O(2^k \cdot n)$.
  - 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.
Example: Treewidth 3.
Treewidth (ctd.)

Treewidth (ctd.)


Treewidth is defined in terms of tree decompositions.
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 $n_1$.

Decomposition width: size of the largest bag (minus 1)

Treewidth: minimum width over all possible tree decompositions
## 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:
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

\[
\begin{array}{ccc}
\{c, f\} & \{b, c, d\} & \{b, c, d\} \\
\{a, b, c\} & \{b, c, d\} & \{d, e\}
\end{array}
\]

\[
\begin{array}{ccc|ccc|c}
& c & f & \text{cost} \\
0 & d & s & 3 \\
1 & d & - & 2 \\
2 & s & d & 2 \\
\end{array}
\]

\[
\begin{array}{ccc|ccc|c}
& b & c & d & \text{cost} \\
0 & d & d & s & 2 \\
1 & d & d & d & 1 \\
2 & s & d & d & 1 \\
3 & d & s & d & 1 \\
\end{array}
\]

\[
\begin{array}{ccc|ccc|c}
& b & c & d & \text{cost} \\
0 & d & d & s & 2 \\
1 & s & d & d & 1 \\
2 & d & s & d & 2 \\
3 & - & - & d & 1 \\
\end{array}
\]

\[
\begin{array}{ccc|ccc|c}
& a & b & c & \text{cost} \\
0 & s & d & d & 1 \\
1 & d & s & d & 1 \\
2 & d & d & s & 1 \\
3 & - & - & - & 0 \\
\end{array}
\]

\[
\begin{array}{ccc|ccc|c}
& d & e & \text{cost} \\
0 & s & d & d & 1 \\
1 & d & s & 1 \\
2 & - & - & 0 \\
\end{array}
\]
Dynamic Programming on Tree Decompositions

**Example: MINIMUM INDEPENDENT DOMINATING SET**

Methodology:
1. Decompose instance
2. Solve partial problems

---

**Diagram:**

- **Vertices:** a, b, c, d, e, f
- **Sets:** 
  - \{c, f\}
  - \{b, c, d\}
  - \{b, c, d\}
  - \{a, b, c\}
  - \{d, e\}

**Cost Table:**

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>f</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Dynamic Programming on Tree Decompositions

Example: **Minimum Independent Dominating Set**

Methodology:
1. Decompose instance
2. Solve partial problems

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>f</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Dynamic Programming on Tree Decompositions

**Example: MINIMUM INDEPENDENT DOMINATING SET**

Methodology:
1. Decompose instance
2. Solve partial problems

![Diagram of tree decomposition and cost tables]
Example: **MINIMUM INDEPENDENT DOMINATING SET**

Methodology:

1. Decompose instance
2. Solve partial problems

---

**Graph Visualization**

- **b** connected to **c**, **d**, **e**
- **c** connected to **b**, **d**
- **d** connected to **b**, **c**, **e**
- **e** connected to **d**

**Subgraphs**

- **{c, f}**
- **{b, c, d}**
- **{b, c, d}**
- **{a, b, c}**
- **{d, e}**

**Cost Matrix**

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

|   | d  | e  | cost |
|---|----|------|
| 0 | s  | 1    |
| 1 | d  | 1    |
| 2 |    | 0    |
Dynamic Programming on Tree Decompositions

Example: **Minimum Independent Dominating Set**

Methodology:
1. Decompose instance
2. Solve partial problems

![Diagram of a tree decomposition with sets and costs]

- **Table 1**: Costs for different combinations of elements.
- **Table 2**: Costs for different combinations with elements in the set.
- **Table 3**: Costs for different combinations with elements outside the set.
Dynamic Programming on Tree Decompositions

Example: Minimum Independent Dominating Set

Methodology:
1. Decompose instance
2. Solve partial problems
Example: MINIMUM INDEPENDENT DOMINATING SET

Methodology:
1. Decompose instance
2. Solve partial problems

Dynamic Programming on Tree Decompositions
Dynamic Programming on Tree Decompositions

Example: MINIMUM INDEPENDENT DOMINATING SET

Methodology:
1. Decompose instance
2. Solve partial problems

{c, f}
{b, c, d}
{b, c, d}
{a, b, c}
{d, e}

{b, c, d}
{b, c, d}
{a, b, c}
{d, e}

\[
\begin{array}{c|c|c|c|c}
 & b & c & d & \text{cost} \\
\hline
0 & d & d & s & 2 \\
1 & d & d & d & 2 \\
2 & s & d & d & 2 \\
3 & d & s & d & 2 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
 & a & b & c & \text{cost} \\
\hline
0 & s & d & d & 1 \\
1 & d & s & d & 1 \\
2 & d & s & d & 1 \\
3 & - & - & - & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
 & d & e & \text{cost} \\
\hline
0 & s & d & 1 \\
1 & d & s & 1 \\
2 & - & - & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
 & c & f & \text{cost} \\
\hline
0 & d & s & 3 \\
1 & d & - & 2 \\
2 & s & d & 2 \\
\end{array}
\]
Dynamic Programming on Tree Decompositions

Example: **MINIMUM INDEPENDENT DOMINATING SET**

Methodology:
1. Decompose instance
2. Solve partial problems

```
Methodology:
1. Decompose instance
2. Solve partial problems

Problem instance:

```

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>f</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
</tbody>
</table>

```

```

```

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

```

```

```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

```
Example: **MINIMUM INDEPENDENT DOMINATING SET**

Methodology:
1. Decompose instance
2. Solve partial problems
Dynamic Programming on Tree Decompositions

**Example: MINIMUM INDEPENDENT DOMINATING SET**

**Methodology:**

1. Decompose instance
2. Solve partial problems

```
\{c, f\}
\{b, c, d\}
\{b, c, d\}
\{a, b, c\}
\{d, e\}

\begin{array}{|c|c|c|c|}
\hline
& c & f & \text{cost} \\
\hline
0 & d & s & 3 \\
1 & d & - & 2 \\
2 & s & d & 2 \\
\hline
\end{array}

\begin{array}{|c|c|c|c|}
\hline
& b & c & d & \text{cost} \\
\hline
0 & d & d & s & 2 \\
1 & d & d & d & 2 \\
2 & s & d & d & 2 \\
3 & d & s & d & 2 \\
\hline
\end{array}

\begin{array}{|c|c|c|c|}
\hline
& b & c & d & \text{cost} \\
\hline
0 & d & d & s & 1 \\
1 & s & d & d & 2 \\
2 & d & s & d & 2 \\
3 & - & - & d & 1 \\
\hline
\end{array}

\begin{array}{|c|c|c|c|}
\hline
& a & b & c & \text{cost} \\
\hline
0 & s & d & d & 1 \\
1 & d & s & d & 1 \\
2 & d & s & s & 1 \\
3 & - & - & - & 0 \\
\hline
\end{array}

\begin{array}{|c|c|c|c|}
\hline
& d & e & \text{cost} \\
\hline
0 & s & d & 1 \\
1 & d & s & 1 \\
2 & - & - & 0 \\
\hline
\end{array}
```
Dynamic Programming on Tree Decompositions

Example: MINIMUM INDEPENDENT DOMINATING SET

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

```
{c, f}
{b, c, d}
{b, c, d}
{a, b, c} {d, e}

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>f</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>s</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```
Dynamic Programming on Tree Decompositions

Example: **MINIMUM INDEPENDENT DOMINATING SET**

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

```
{c, f}       {b, c, d}
{b, c, d}    {b, c, d}
{a, b, c}    {d, e}
```

```
<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
D-FLAT

Dynamic Programming Framework with Local Execution of ASP on Tree 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 $\Sigma_2^P$) can be solved with ASP

**Minimum Independent Dominating Set**

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

\[
\begin{align*}
\{ \text{in}(X) : & \text{vertex}(X) \}. \\
\leftarrow & \text{in}(X), \text{in}(Y), \text{edge}(X,Y). \\
\text{dominated}(X) & \leftarrow \text{in}(Y), \text{edge}(Y,X). \\
\leftarrow & \text{vertex}(X), \text{not} \text{ in}(X), \text{not} \text{ dominated}(X). \\
\text{#minimize} & \{ 1,X : \text{in}(X) \}.
\end{align*}
\]
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
  
  \[ \rightarrow \text{ allows for rapid prototyping of dynamic programming on tree decompositions} \]
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

- Parse instance
- Decompose
- Store partial solutions
- ASP call
- Visit next node in post-order
- Done?
  - yes: Print complete solutions
  - no: Repeat process
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance \rightarrow \text{Decompose} \rightarrow \text{Done?} \rightarrow \text{ASP call} \rightarrow \text{Visit next node in post-order}

\text{Print complete solutions}

\text{Store partial solutions}

\text{Done?}

\text{yes}

\text{no}

\{b, c, d\} \rightarrow \text{n} \rightarrow \text{n} \rightarrow \text{n} \rightarrow \text{n} \rightarrow \text{n} \rightarrow \text{n}
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

Parse instance → Decompose → Store partial solutions

Done? yes → Print complete solutions

No → Visit next node in post-order

ASP call

Visit next node in post-order

Print complete solutions

n₁ \{a, b, c\} → n₂ \{b, c, d\} → n₃ \{d, e\} → n₄ \{b, c, d\} → n₅ \{b, c, d\} → n₆ \{c, f\}
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Done?

Store partial solutions

Print complete solutions

ASP call → Visit next node in post-order

Done?

no

yes

n₁ {a, b, c} n₂ {b, c, d} n₃ {d, e} n₄ {b, c, d} n₅ {b, c, d} n₆ {c, f}
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Store partial solutions

ASP call

Visit next node in post-order

Done?

Print complete solutions

\[
\begin{align*}
\text{n}_1 &: \{a, b, c\} \\
\text{n}_2 &: \{b, c, d\} \\
\text{n}_3 &: \{d, e\} \\
\text{n}_4 &: \{b, c, d\} \\
\text{n}_5 &: \{b, c, d\} \\
\text{n}_6 &: \{c, f\}
\end{align*}
\]
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Done?

Store partial solutions

ASP call

Try next node in post-order

Print complete solutions

\[
\begin{align*}
\text{Decompose} = \{c, f\} \\
\{b, c, d\} & \rightarrow n_2 \rightarrow n_4 \\
\{b, c, d\} & \rightarrow n_5 \\
\{a, b, c\} & \rightarrow n_1 \\
\{d, e\} & \rightarrow n_3
\end{align*}
\]
D-FLAT at Work
Illustrated by means of Independent Dominating Set

Parse instance ➔ Decompose

Done? no ➔ Visit next node in post-order

ASP call

Print complete solutions

Store partial solutions

Decompose

\{a, b, c\}
\{b, c, d\}
\{b, c, d\}
\{b, c, d\}
\{c, f\}
\{d, e\}

0 1 2 3
\{a\}
\{b\}
\{c\}
\{d\}
\{e\}
\{f\}
**D-FLAT at Work**

Illustrated by means of **INDEPENDENT DOMINATING SET**

**ASP call**

**Visit next node in post-order**

**Done?**

**Print complete solutions**

**Parse instance** → **Decompose**

**Store partial solutions**

**Print complete solutions**

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>

**Decompose**

- \( n_1 \): \{a, b, c\}
- \( n_2 \): \{b, c, d\}
- \( n_3 \): \{d, e\}
- \( n_4 \): \{b, c, d\}
- \( n_5 \): \{b, c, d\}
- \( n_6 \): \{c, f\}

**Visited nodes**

- \( n_1 \)
- \( n_2 \)
- \( n_3 \)
- \( n_4 \)
- \( n_5 \)
- \( n_6 \)

**Visit next node in post-order**

- \( n_2 \)
  - \( n_1 \)
  - \( n_3 \)
  - \( n_4 \)
  - \( n_5 \)
  - \( n_6 \)
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Store partial solutions

Done?

no → Visit next node in post-order

yes → Print complete solutions

ASP call

Nodes:
- $n_1 = \{a, b, c\}$
- $n_2 = \{b, c, d\}$
- $n_3 = \{d, e\}$
- $n_4 = \{b, c, d\}$
- $n_5 = \{b, c, d\}$
- $n_6 = \{c, f\}$

Edges:
- $a \rightarrow b$
- $c \rightarrow d$
- $c \rightarrow e$
- $f \rightarrow c$

Legend:
- $\{a, b, c\}$
- $\{b, c, d\}$
- $\{d, e\}$
- $\{c, f\}$
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Store partial solutions

ASP call

Visit next node in post-order

Done?

yes → Print complete solutions

no → Visit next node in post-order

Print complete solutions

\n\{a, b, c\} n_1
\{b, c, d\} n_2
\{b, c, d\} n_3
\{b, c, d\} n_4
\{c, f\} n_5
\{b, c, d\} n_6
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

- Parse instance
- Decompose
- Store partial solutions
- ASP call
- Visit next node in post-order
- Print complete solutions

Graph:
- Node a
  - Connections: b, c, d
- Node b
  - Connections: c, d, f
- Node c
  - Connections: a, b, d
- Node d
  - Connections: c, e
- Node e
  - Connections: d
- Node f
  - Connections: b, c

Nodes:
- $n_1 = \{a, b, c\}$
- $n_2 = \{b, c, d\}$
- $n_3 = \{d, e\}$
- $n_4 = \{b, c, d\}$
- $n_5 = \{b, c, d\}$
- $n_6 = \{c, f\}$
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

```
 Parse instance → Decompose → Store partial solutions → ASP call
                |               | no |
                | yes → Print complete solutions
                | Visit next node in post-order
```

```
{a, b, c} \n1  \n{b, c, d} \n2  \n{b, c, d} \n3  \n{d, e} \n4  \n{c, f} \n5  \n{b, c, d} \n6
```
D-FLAT at Work
Illustrated by means of Independent Dominating Set

Parse instance → Decompose → Store partial solutions → Done?

- Yes: Print complete solutions
- No: ASP call → Visit next node in post-order

Decompose \{c, f\} → n_6
Decompose \{b, c, d\} → n_5
Decompose \{b, c, d\} → n_4
Decompose \{a, b, c\} → n_1
Decompose \{d, e\} → n_3

ASP call: n_6, n_5, n_4, n_3
D-FLAT at Work
Illustrated by means of Independent Dominating Set

Parse instance → Decompose → Done?

Done? no → Visit next node in post-order

Print complete solutions

ASP call

Store partial solutions

{a, b, c} {d, e} {b, c, d} {c, f}

n1 n2 n3 n4 n5 n6
D-FLAT at Work
Illustrated by means of Independent Dominating Set

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Done?

Print complete solutions

no

yes

\{a, b, c\} \text{ n}_1

\{b, c, d\} \text{ n}_2

\{b, c, d, e\} \text{ n}_4

\{c, f\} \text{ n}_6

\{b, c, d\} \text{ n}_5

\{d, e\} \text{ n}_3

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?

no

yes

Parse instance → Decompose

Store partial solutions

ASP call

Visit next node in post-order

Print complete solutions

Done?
D-FLAT at Work
Illustrated by means of **INDEPENDENT DOMINATING SET**
D-FLAT at Work
Illustrated by means of **INDEPENDENT DOMINATING SET**

Choose nodes in a tree.

**Parse instance** → **Decompose** → **Done?**

1. **Print complete solutions**
2. **ASP call** → **Visit next node in post-order**

**Nodes**:
- **n1**: \{a, b, c\}
- **n2**: \{b, c, d\}
- **n3**: \{d, e\}
- **n4**: \{b, c, d\}
- **n5**: \{b, c, d\}
- **n6**: \{c, f\}

**Costs**:
- **n1**: cost = 0
- **n2**: cost = 2
- **n3**: cost = 1
- **n4**: cost = 1
- **n5**: cost = 1
- **n6**: cost = 2

**Ownership**:
- **n1**: \{a, b, c\}
- **n2**: \{b, c, d\}
- **n3**: \{d, e\}
- **n4**: \{b, c, d\}
- **n5**: \{b, c, d\}
- **n6**: \{c, f\}
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) }.
← edge(X,Y), item(in(X;Y)).
item(dom(X)) ← item(in(Y)), edge(Y,X),
              current(X).
```

### 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

```plaintext
% 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
- Additional D-FLAT features for arithmetics
- 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**
- D-FLAT^2 frontend
  - designed for minimization problems on top of “table-mode”
  - DP is automatically obtained from simpler principles
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^2 frontend
  - designed for minimization problems on top of “table-mode”
  - DP is automatically obtained from simpler principles

### “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.
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
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).
  - This works for all MSO-definable problems [JLC 2016]
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
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)
- Development of system that allows to customize tree decompositions
**Methodology**

<table>
<thead>
<tr>
<th>Given a specific problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Training data:</td>
</tr>
<tr>
<td>▶ 70 random instances with rather low treewidth</td>
</tr>
<tr>
<td>▶ 40 decompositions for each problem instance</td>
</tr>
<tr>
<td>▶ Obtain regression models (16 different methods) for ranking decompositions using specific tree decomposition features</td>
</tr>
<tr>
<td>▶ Apply model to real-world instances (treewidth up to 8)</td>
</tr>
<tr>
<td>▶ Generate 50 tree decompositions per instance</td>
</tr>
<tr>
<td>▶ Model selects the best-ranked decomposition</td>
</tr>
</tbody>
</table>
Experimental Results

- Accelerating Minimum Dominating Set using Machine Learning

### Minimum Dominating Set

<table>
<thead>
<tr>
<th>D-FLAT</th>
<th>Minimum Improvement: 7.39 %</th>
<th>Average Improvement: 21.80 %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum Improvement: 31.15 %</td>
<td>Median Improvement: 24.25 %</td>
</tr>
<tr>
<td></td>
<td>Statistical Significance: 99.95 %</td>
<td></td>
</tr>
</tbody>
</table>

#### Predicted Rank vs. Model

- Predicted Rank vs. Model
- Runtime Improvement vs. Model
Experimental Results

- Accelerating Minimum Dominating Set using Machine Learning

<table>
<thead>
<tr>
<th>Minimum Dominating Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUOIA</td>
</tr>
</tbody>
</table>

| Minimum Improvement: 11.39 % | Average Improvement: 16.24 % |
| Maximum Improvement: 19.65 % | Median Improvement: 17.39 %   |

<table>
<thead>
<tr>
<th>Statistical Significance:</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥ 99.95 %</td>
</tr>
</tbody>
</table>

![Graph showing predicted rank and runtime improvement](image-url)
Towards exploiting Decomposition Features

<table>
<thead>
<tr>
<th>New decomposition library: htd</th>
</tr>
</thead>
<tbody>
<tr>
<td>htd provides efficient implementations of well-known algorithms</td>
</tr>
<tr>
<td>htd allows to fully customize the tree decomposition via several strategies</td>
</tr>
<tr>
<td>htd offers a wide range of convenience functions like the possibility to access the subgraph induced by each bag at almost no cost (Performance boost for large graphs!).</td>
</tr>
<tr>
<td>3rd place in recent tree-decomposition competition</td>
</tr>
<tr>
<td><a href="https://pacechallenge.wordpress.com/">https://pacechallenge.wordpress.com/</a></td>
</tr>
<tr>
<td>Available at: <a href="https://github.com/mabseher/htd">https://github.com/mabseher/htd</a></td>
</tr>
</tbody>
</table>
Discussion

- We conducted huge test series [IJCAI 2015] for several problems and two state-of-the-art systems (D-FLAT and SEQUOIA).
- Feature-based ML successfully identified good decompositions.
- However, crucial features are in general not problem independent.
- New decomposition library allows the user to specify what kind of tree decomposition she prefers.
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
Motivation

Lesson Learnt

- Drawback of classical DP on TDs: Always computes all solutions even if only one is required.
- Optimization problems: Sometimes table rows have higher costs than optimal solution.
Motivation

Lesson Learnt

- Drawback of classical DP on TDs: Always computes all solutions even if only one is required.
- Optimization problems: Sometimes table rows have higher costs than optimal solution.

Idea

- Materialize tables “in parallel”.
- Realization in D-FLAT: modern ASP technology (external atoms)
- Use coexisting ASP solvers that communicate with each other.
Motivation

Lesson Learnt

- Drawback of classical DP on TDs: Always computes all solutions even if only one is required.
- Optimization problems: Sometimes table rows have higher costs than optimal solution.

Idea

- Materialize tables “in parallel”.
- Realization in D-FLAT: modern ASP technology (external atoms)
- Use coexisting ASP solvers that communicate with each other.

Goals

- Anytime behavior (ability to report solutions when interrupted)
- Understand feasibility of this approach
Example: “Lazy” DP on TDs

DP specification in ASP

```lp
#external childItem(in(X)) : childNode(N), bag(N,X).
#external childAuxItem(dom(X)) : childNode(N), bag(N,X).

item(in(X)) ← childItem(in(X)), not removed(X).
auxItem(dom(X)) ← childAuxItem(dom(X)), not removed(X).
{ item(in(X)) : introduced(X) }.
auxItem(dom(Y)) ← item(in(X)), edge(X,Y), current(X;Y).
← removed(X), not childItem(in(X)), not childAuxItem(dom(X)).
← edge(X,Y), item(in(X)), item(in(Y)).
```

Avoiding Re-grounding via Assumption-based Solving

- The `clingo` system supports external atoms.
- Truth value of externals can be set “from the outside”.
  1. Freeze a certain truth assignment on externals.
  2. Compute all answer sets under this assumption.
  3. Repeat with different assumption.
- Grounding only happens once.
Experimental Results

Search and optimization problems on real-world graphs

“Lazy” vs. “eager”

- Search problems: “Lazy” usually finds a solution much quicker.
- Optimization problems: “Lazy” mostly finds optimum faster (and able to print solutions along the way)

Comparison to clingo (without decomposition)

- Search problems: Clingo finds a solution much quicker.
- **Dominating Set, Vertex Cover**: Clingo is clearly faster.
- **Steiner Tree**: “Lazy” is faster . . .
  - “Lazy” often finds optimum when clingo times out.
  - “Lazy” offers better suboptimal solutions until timeout.
Discussion

- DP on TDs via “lazy evaluation”
- At each table, an ASP solver is used for computing rows
  - Multiple coexisting ASP solvers that communicate with each other
  - *Assumption-based solving*: avoids excessive re-grounding
- “Lazy” outperforms “eager”
- Outperforms state-of-the-art ASP systems on some problems (w.r.t. anytime performance)
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
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
Example (OBDD representation)

Let formula $\varphi = (a \land b \land c) \lor (a \land \neg b \land c) \lor (\neg a \land b \land c)$. 

Figure: OBDD of $\varphi$. 

Figure: ROBDD of $\varphi$. 

Example (OBDD representation)

Let formula \( \varphi = (a \land b \land c) \lor (a \land \neg b \land c) \lor (\neg a \land b \land c) \).
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 via BDDs

\[ B_t^l = \bigwedge_{(x,y) \in E_t} (\neg i_x \lor \neg i_y) \land \bigwedge_{y \in V_t} \left( d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right) \]
\[
B_t^l = \bigwedge_{(x,y) \in E_t} (\neg i_x \lor \neg i_y) \land \bigwedge_{y \in V_t} \left( d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right) \\
B_t^i = \exists D'_t \left[ B_t'[D_t'/D_t'] \land \bigwedge_{(u,y) \in E_t} (\neg i_u \lor \neg i_y) \land \left( d_u \leftrightarrow \bigvee_{(x,u) \in E_t} i_x \right) \land \bigwedge_{(u,y) \in E_t \land u \neq y} (d_y \leftrightarrow d'_y \lor i_u) \land \bigwedge_{y \in V_t \land (u,y) \notin E_t} (d_y \leftrightarrow d'_y) \right]
\]
DP of Independent Dominating Set via BDDs

\[ B_t^l = \bigwedge_{(x,y) \in E_t} \left( \neg i_x \lor \neg i_y \right) \land \bigwedge_{y \in V_t \land (x,y) \in E_t} \left( d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right) \]

\[ B_t^i = \exists D_t' \left[ B_t'[D_t'/D_t'] \land \bigwedge_{(u,y) \in E_t} \left( \neg i_u \lor \neg i_y \right) \land \left( d_u \leftrightarrow \bigvee_{(x,u) \in E_t} i_x \right) \land \right. \]

\[ \left. \bigwedge_{(u,y) \in E_t \land u \neq y} \left( d_y \leftrightarrow d'_y \lor i_u \right) \land \bigwedge_{y \in V_t \land (u,y) \notin E_t} \left( d_y \leftrightarrow d'_y \right) \right] \]

\[ B_t^r = B_t'[i_u/\top, d_u/\bot] \lor B_t'[i_u/\bot, d_u/\top] \]
DP of Independent Dominating Set via BDDs

\[ \mathcal{B}_t^l = \bigwedge_{(x,y) \in E_t} \neg i_x \lor \neg i_y \land \bigwedge_{y \in V_t} \left( d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right) \]

\[ \mathcal{B}_t^i = \exists D'_t \left[ \mathcal{B}_t'[D_t'/D'_t] \land \bigwedge_{(u,y) \in E_t} \left( \neg i_u \lor \neg i_y \land \left( d_u \leftrightarrow \bigvee_{(x,u) \in E_t} i_x \right) \land \bigwedge_{(u,y) \in E_t \land u \neq y} \left( d_y \leftrightarrow d'_y \lor i_u \right) \land \bigwedge_{y \in V_t \land (u,y) \notin E_t} (d_y \leftrightarrow d'_y) \right] \right] \]

\[ \mathcal{B}_t^j = \exists D'_t D''_t \left[ \mathcal{B}_t'[D_t/D'_t] \land \mathcal{B}_t''[D_t/D''_t] \land \bigwedge_{x \in V_t} \left( d_x \leftrightarrow d'_x \lor d''_x \right) \right] \]
<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the presented ideas for solving quantified Boolean formulas in prenex CNF form</td>
</tr>
<tr>
<td>( \exists ab \land cd \exists ef (a \lor c \lor e) \land (\neg b \lor d) \land (e \lor f) \land (c \lor \neg e) \land (\neg d \lor f) )</td>
</tr>
<tr>
<td>We consider primal graph of the CNF</td>
</tr>
<tr>
<td>Datastructure used is a recursive set of BDDs (recursion depth depends on number of quantifier alternations)</td>
</tr>
<tr>
<td>Some further optimizations required to be competitive</td>
</tr>
</tbody>
</table>
Experimental Results

2-QBF (∀∃) competition instances (#instances = 200)

![Graph showing instances solved vs. time (sec).]

Solved instances with small width (w ≤ 50, #instances = 55):
- dynQBF: 54
- EBDDRES: 31
- DepQBF: 28
- RAReQS: 19

Uniquely solved (#instances = 200):
- DepQBF: 43
- dynQBF: 41
- RAReQS: 5
- EBDDRES: 2
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\# instances = 200$)

- Solved instances with small width ($w \leq 50$, $\# instances = 55$):
  - $\diamond$ dynQBF: 54, $\otimes$ EBDDRES: 31, $\blacktriangleleft$ DepQBF: 28, $\times$ RAReQS: 19
- Uniquely solved ($\# instances = 200$):
  - $\blacktriangleleft$ DepQBF: 43, $\otimes$ dynQBF: 41, $\times$ RAReQS: 5, $\times$ EBDDRES: 2

![Graph showing the number of instances solved against time (sec) for different solvers.]
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\# instances = 200$)

Solved instances with small width ($w \leq 50$, $\# instances = 55$):
- dynQBF: 54, EBDDRES: 31, DepQBF: 28, RAReQS: 19

Uniquely solved ($\# instances = 200$):
- DepQBF: 43, dynQBF: 41, RAReQS: 5, EBDDRES: 2
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\#\text{instances} = 200$)

- Solved instances with small width ($w \leq 50$, $\#\text{instances} = 55$):
  - $\triangledown \text{dynQBF}: 54$, $\diamond \text{EBDDRES: 31}$, $\square \text{DepQBF: 28}$, $\circ \text{RAReQS: 19}$

- Uniquely solved ($\#\text{instances} = 200$):
  - $\bigdiamond \text{DepQBF: 43}$, $\blacklozenge \text{dynQBF: 41}$, $\triangle \text{RAReQS: 5}$, $\ast \text{EBDDRES: 2}$
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\#\text{instances} = 200$)

Solved instances with small width ($w \leq 50$, $\#\text{instances} = 55$):
- dynQBF: 54, EBDDRES: 31, DepQBF: 28, RAReQS: 19

Uniquely solved ($\#\text{instances} = 200$):
- DepQBF: 43, dynQBF: 41, RAReQS: 5, EBDDRES: 2
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\#\text{instances} = 200$)

- Instances solved
  - dynQBF (current)
  - dynQBF (QBFEval'16)
  - BDD (naive)
  - EBDDRES 1.2
  - RAReQS 1.1

- Solved instances with small width ($w \leq 50$, $\#\text{instances} = 55$):
  - dynQBF: 54, EBDDRES: 31, DepQBF: 28, RAReQS: 19

- Uniquely solved ($\#\text{instances} = 200$):
  - DepQBF: 43, dynQBF: 41, RAReQS: 5, EBDDRES: 2
Experimental Results

2-QBF ($\forall\exists$) competition instances ($\# instances = 200$)

Solved instances with small width ($w \leq 50$, $\# instances = 55$):
- dynQBF: 54
- EBDDRES: 31
- DepQBF: 28
- RAReQS: 19

Uniquely solved ($\# instances = 200$):
- DepQBF: 43
- dynQBF: 41
- RAReQS: 5
- EBDDRES: 2
Experimental Results

2-QBF ($\forall \exists$) competition instances ($\# \text{instances} = 200$)

[Solved instances with small width ($w \leq 50$, $\# \text{instances} = 55$):]
- dynQBF: 54, EBDDRES: 31, DepQBF: 28, RAReQS: 19
Experimental Results

2-QBF (∀∃) competition instances (#instances = 200)

Solved instances with small width (w ≤ 50, #instances = 55):
- dynQBF: 54, EBDDRES: 31, DepQBF: 28, RAReQS: 19

Uniquely solved (#instances = 200):
- DepQBF: 43, dynQBF: 41, RAReQS: 5, EBDDRES: 2
## Experimental Evaluation

QBF Gallery 2014 competition instances ($\#\text{instances} = 276$)

<table>
<thead>
<tr>
<th>System</th>
<th>Solved</th>
<th>SAT</th>
<th>UNSAT</th>
<th>Timeout</th>
<th>Memout</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepQBF 5.0</td>
<td>103</td>
<td>48</td>
<td>55</td>
<td>169</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>RAReQS 1.1</td>
<td>83</td>
<td>36</td>
<td>47</td>
<td>193</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>dynQBF (current)</td>
<td>21</td>
<td>6</td>
<td>15</td>
<td>250</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>EBDDRES 1.2</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>265</td>
<td>2</td>
</tr>
<tr>
<td>BDD (naive)</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>273</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Experimental Evaluation

QBF Gallery 2014 competition instances (#instances = 276)

<table>
<thead>
<tr>
<th>System</th>
<th>Solved</th>
<th>SAT</th>
<th>UNSAT</th>
<th>Timeout</th>
<th>Memout</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepQBF 5.0</td>
<td>103</td>
<td>48</td>
<td>55</td>
<td>169</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>RAReQS 1.1</td>
<td>83</td>
<td>36</td>
<td>47</td>
<td>193</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>dynQBF (current)</td>
<td>21</td>
<td>6</td>
<td>15</td>
<td>250</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>EBDDRES 1.2</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>265</td>
<td>2</td>
</tr>
<tr>
<td>BDD (naive)</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>273</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

dynQBF is not yet competitive:

- 27 out of 276 instances were not decomposed within the time limit
- Solved instances have an average width of 55, 3 quantifiers, 4711 atoms and 16409 clauses
Discussion

- **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
- particularly successful for QBF solving
- currently, algorithms have to be implemented in C++ on top of CUDD
- **Systems available:**
  - dbai.tuwien.ac.at/proj/decodyn/dynbdd/
  - dbai.tuwien.ac.at/proj/decodyn/dynqbf/
Outline

Motivation

Tree Decompositions + Dynamic Programming

The D-FLAT System

Further Developments
  Customizing Tree Decompositions
  Anytime Optimization
  Towards Space Efficiency

Conclusion
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
Ongoing + Future Work

- Automatic generation of D-FLAT code from “standard” encoding
- Exploit smarter ways to store solutions
  - BDDs a promising option
  - Easy-to-use interface still missing
- Tighter integration of D-FLAT with ASP solvers
  - Communication between D-FLAT and ASP solver is bottleneck
- Incorporation of other decomposition methods
  - Straight forward for clique width, branch width, . . .
  - Lack of efficient heuristics for obtaining decomposition
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


