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 ...
Let’s Decompose them ...
Let's Decompose them ...
Let’s Decompose them ...

Runtime: $O(2^n)$
Let’s Decompose them ...

Runtime: $O(2^n)$
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).
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”.
But ...

“...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...

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 is defined in terms of \textit{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
### 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!
### 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:

\[
\begin{array}{c|c|c|c}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
\text{f} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{d} & \text{e} & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Set</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b, c}</td>
<td>0</td>
</tr>
<tr>
<td>{d}</td>
<td>1</td>
</tr>
<tr>
<td>{e}</td>
<td>2</td>
</tr>
<tr>
<td>{f}</td>
<td>2</td>
</tr>
<tr>
<td>{b, c, d}</td>
<td>3</td>
</tr>
<tr>
<td>{c, f}</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
\text{f} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{d} & \text{e} & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Set</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b, c}</td>
<td>0</td>
</tr>
<tr>
<td>{d}</td>
<td>1</td>
</tr>
<tr>
<td>{e}</td>
<td>2</td>
</tr>
<tr>
<td>{f}</td>
<td>2</td>
</tr>
<tr>
<td>{b, c, d}</td>
<td>3</td>
</tr>
<tr>
<td>{c, f}</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
\text{f} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{d} & \text{e} & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Set</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b, c}</td>
<td>0</td>
</tr>
<tr>
<td>{d}</td>
<td>1</td>
</tr>
<tr>
<td>{e}</td>
<td>2</td>
</tr>
<tr>
<td>{f}</td>
<td>2</td>
</tr>
<tr>
<td>{b, c, d}</td>
<td>3</td>
</tr>
<tr>
<td>{c, f}</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
\text{f} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{d} & \text{e} & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Set</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b, c}</td>
<td>0</td>
</tr>
<tr>
<td>{d}</td>
<td>1</td>
</tr>
<tr>
<td>{e}</td>
<td>2</td>
</tr>
<tr>
<td>{f}</td>
<td>2</td>
</tr>
<tr>
<td>{b, c, d}</td>
<td>3</td>
</tr>
<tr>
<td>{c, f}</td>
<td>0</td>
</tr>
</tbody>
</table>
Dynamic Programming on Tree Decompositions

Example: **Minimum Independent Dominating Set**

Methodology:

1. Decompose instance

![Diagram of a graph with nodes labeled a, b, c, d, e, and f, and sets {c, f}, {b, c, d}, {b, c, d}, {b, c, d}, {a, b, c}, and {d, e}.]
Dynamic Programming on Tree Decompositions

Example: **MINIMUM INDEPENDENT DOMINATING SET**

Methodology:

1. Decompose instance
2. Solve partial problems

![Diagram of a tree decomposition and corresponding matrices for cost](image)
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

\begin{align*}
\{c, f\} & \rightarrow \{b, c, d\} \\
\{b, c, d\} & \rightarrow \{b, c, d\} \\
\{a, b, c\} & \rightarrow \{d, e\}
\end{align*}
```

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

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

```
\begin{array}{c|c|c|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|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

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

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

![Diagram of tree decomposition and cost matrices](image-url)
Dynamic Programming on Tree Decompositions

**Example: MINIMUM INDEPENDENT DOMINATING SET**

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

![Graph and Table Example]

- **Graph:** Nodes represent vertices, and edges represent connections. The graph is decomposed into smaller subgraphs.
- **Table:** The table shows the cost for different configurations of vertices. Each row represents a different combination of vertices, and the cost is calculated accordingly.
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
```

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

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

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>d</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>s</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>s</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

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
Dynamic Programming on Tree Decompositions

Example: **Minimum Independent Dominating Set**

Methodology:
1. Decompose instance
2. Solve partial problems

![Diagram showing tree decomposition and cost matrix]

---

<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>d</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>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>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

\[
\begin{align*}
\{c, f\} & \rightarrow \{b, c, d\} \\
\{b, c, d\} & \rightarrow \{b, c, d\} \\
\{a, b, c\} & \rightarrow \{d, e\}
\end{align*}
\]
Dynamic Programming on Tree Decompositions

Example: **MINIMUM INDEPENDENT DOMINATING SET**

Methodology:
1. Decompose instance
2. Solve partial problems

![Diagram]

- **Node Decomposition**
  - `{c, f}`
  - `{b, c, d}`
  - `{a, b, c}`
  - `{d, e}`

- **Cost 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>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>-</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>
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
3. Combine the solutions

![Tree Decomposition Diagram]

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

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

![Diagram of tree decompositions and tables](image)
Outline
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
  - ...
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`.

```prolog
{ 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
  
  \[ \implies \text{allows for rapid prototyping of dynamic programming on tree decompositions} \]
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Store partial solutions

Done? yes → Print complete solutions

Done? no → ASP call → Visit next node in post-order
D-FLAT at Work
Illustrated by means of **INDEPENDENT DOMINATING SET**

- **Parse instance** → **Decompose** → **Done?**
  - **no** → **ASP call**
  - **yes** → **Print complete solutions**

**ASP Call**

**Visiting next node in post-order**

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

**Print complete solutions**
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 → ASP call

Done? no → Visit next node in post-order

yes → Print complete solutions

{a, b, c} n₁
{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
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

Yes ➔ No

Parse instance diagram:
- a
- b
- c
- d
- e
- f

ASP call graph:
- \( n_1 \{a, b, c\} \)
- \( n_2 \{b, c, d\} \)
- \( n_3 \{d, e\} \)
- \( n_4 \{b, c, d\} \)
- \( n_5 \{b, c, d\} \)
- \( n_6 \{c, f\} \)
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

Parse instance → Decompose → Store partial solutions → ASP call

Done? no → Visit next node in post-order

Print complete solutions

Visit next node in post-order

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

Print complete solutions

Done?

no

Visit next node in post-order

yes

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

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

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

\[
\begin{array}{c|cccc|c}
0 & d & c & s & 1 \\
1 & s & d & d & 2 \\
2 & s & d & d & 2 \\
3 & - & - & d & 1 \\
\end{array}
\]

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

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**

Store partial solutions

Print complete solutions

ASP call

Visit next node in post-order

---

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

- **Yes** → **Print complete solutions**
- **No** → **Visit next node in post-order**

---

**Node n₁**
- **Parse instance:** {a, b, c}
- **Decompose:** {b, c, d}
- **Print solutions:**
  - {b, c, d}

**Node n₂**
- **Parse instance:** {b, c, d}
- **Decompose:** {b, c, d}
- **Print solutions:**
  - {b, c, d}

**Node n₃**
- **Parse instance:** {d, e}
- **Decompose:** {d, e}
- **Print solutions:**
  - {d, e}

**Node n₄**
- **Parse instance:** {b, c, d}
- **Decompose:** {b, c, d}
- **Print solutions:**
  - {b, c, d}

---

**Node n₅**
- **Parse instance:** {b, c, d}
- **Decompose:** {b, c, d}
- **Print solutions:**
  - {b, c, d}

**Node n₆**
- **Parse instance:** {c, f}
- **Decompose:** {c, f}
- **Print solutions:**
  - {c, f}
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 → Done? → Store partial solutions

- no: ASP call
- yes: Visit next node in post-order

Print complete solutions

\[
\begin{align*}
&\{a, b, c\} \quad n_1 \\
&\{b, c, d\} \quad n_2 \\
&\{b, c, d\} \quad n_5 \\
&\{c, f\} \quad n_6 \\
&\{d, e\} \quad n_3 \\
\end{align*}
\]
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

Decompose

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

Print complete solutions

Done?

yes

no
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET
D-FLAT at Work
Illustrated by means of INDEPENDENT DOMINATING SET

<table>
<thead>
<tr>
<th>n1</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>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>s</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>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n2</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>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>n3</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>

<table>
<thead>
<tr>
<th>n4</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>

Parse instance → Decompose → Store partial solutions

ASP call

Print complete solutions

Done?

no
Visit next node in post-order

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 → Done?

ASP call

Visit next node in post-order

Print complete solutions

Store partial solutions

Done?

yes

no

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
  - yes: Print complete solutions
  - no: ASP call, Visit next node in post-order
- Done?
  - yes: Print complete solutions
  - no: Visit next node in post-order

Graph:

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

Visit next node in post-order.

Print complete solutions.
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

\[ \{c, f\} \]
\[ \{b, c, d\} \]
\[ \{b, c, d\} \]
\[ \{b, c, d\} \]
\[ \{d, e\} \]
D-FLAT at Work (ctd.)
Illustrated by means of INDEPENDENT DOMINATING SET

User-supplied program

\[
\begin{align*}
&\{ \text{extend}(R) : \text{childRow}(R,N) \} \quad \text{1} \leftarrow \text{childNode}(N). \\
&\leftarrow \text{extend}(R_1;R_2), \text{childItem}(R_1,\text{in}(X)), \\
&\quad \text{not} \quad \text{childItem}(R_2,\text{in}(X)). \\
&\leftarrow \text{removed}(X), \text{extend}(R), \\
&\quad \text{not} \quad \text{childItem}(R,\text{in}(X)), \text{not} \quad \text{childItem}(R,\text{dom}(X)). \\
&\text{item}(\text{in}(X)) \leftarrow \text{extend}(R), \text{childItem}(R,\text{in}(X)), \\
&\quad \text{current}(X). \\
&\text{item}(\text{dom}(X)) \leftarrow \text{extend}(R), \text{childItem}(R,\text{dom}(X)), \\
&\quad \text{current}(X). \\
&\{ \text{item}(\text{in}(X)) : \text{introduced}(X) \}. \\
&\text{item}(\text{dom}(X)) \leftarrow \text{item}(\text{in}(Y)), \text{edge}(Y,X), \\
&\quad \text{current}(X). \\
&\leftarrow \text{edge}(X,Y), \text{item}(\text{in}(X;Y)).
\end{align*}
\]

Instance

\[
\text{vertex}(a;b;c;d;e). \\
\text{edge}(a,b). \text{edge}(a,c). \text{edge}(b,c). \\
\text{edge}(b,d). \text{edge}(c,d). \text{edge}(d,e).
\]
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.

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
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).
  - 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 2nd 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 2nd 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 2nd-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 Framework with Local Execution of ASP on TDs 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
Recall encoding for SAT

\[
\text{extend}(R) : \text{childRow}(R,N) \} 1 \leftarrow \text{childNode}(N).
\]
\[
\text{extend}(R;S), \text{atom}(A), \text{childItem}(R,A), \text{not} \text{childItem}(S,A).
\]
\[
\text{item}(X) \leftarrow \text{extend}(R), \text{childItem}(R,X), \text{current}(X).
\]
\[
\{ \text{item}(A) : \text{atom}(A), \text{introduced}(A) \}.
\]
\[
\text{item}(C) \leftarrow \text{current}(C;A), \text{pos}(C,A), \text{item}(A).
\]
\[
\text{item}(C) \leftarrow \text{current}(C;A), \text{neg}(C,A), \text{not} \text{item}(A).
\]
\[
\text{clause}(C), \text{removed}(C), \text{extend}(R), \text{not} \text{childItem}(R,C).
\]
Recall encoding for SAT

\[
1 \{ \text{extend}(R) : \text{childRow}(R,N) \} \quad 1 \leftarrow \text{childNode}(N).
\]
\[
\leftarrow \text{extend}(R;S), \ \text{atom}(A), \ \text{childItem}(R,A), \ \text{not} \ \text{childItem}(S,A).
\]
\[
\text{item}(X) \leftarrow \text{extend}(R), \ \text{childItem}(R,X), \ \text{current}(X).
\]
\[
\{ \text{item}(A) : \ \text{atom}(A), \ \text{introduced}(A) \}.
\]
\[
\text{item}(C) \leftarrow \text{current}(C;A), \ \text{pos}(C,A), \ \text{item}(A).
\]
\[
\text{item}(C) \leftarrow \text{current}(C;A), \ \text{neg}(C,A), \ \text{not} \ \text{item}(A).
\]
\[
\leftarrow \text{clause}(C), \ \text{removed}(C), \ \text{extend}(R), \ \text{not} \ \text{childItem}(R,C).
\]

For MINSAT, we just need to add

\[
\text{optItem}(X) \leftarrow \text{item}(X), \ \text{atom}(X).
\]
Recall encoding for SAT

\[
\begin{align*}
1 \{ \text{extend}(R) & : \text{childRow}(R,N) \} 1 \leftarrow \text{childNode}(N). \\
& \leftarrow \text{extend}(R;S), \ \text{atom}(A), \ \text{childItem}(R,A), \ \text{not} \ \text{childItem}(S,A). \\
\text{item}(X) & \leftarrow \text{extend}(R), \ \text{childItem}(R,X), \ \text{current}(X). \\ & \{ \text{item}(A) : \ \text{atom}(A), \ \text{introduced}(A) \}. \\
\text{item}(C) & \leftarrow \text{current}(C;A), \ \text{pos}(C,A), \ \text{item}(A). \\
\text{item}(C) & \leftarrow \text{current}(C;A), \ \text{neg}(C,A), \ \text{not} \ \text{item}(A). \\
& \leftarrow \text{clause}(C), \ \text{removed}(C), \ \text{extend}(R), \ \text{not} \ \text{childItem}(R,C). 
\end{align*}
\]

For MINSAT, we just need to add

\[
\text{optItem}(X) \leftarrow \text{item}(X), \ \text{atom}(X). 
\]

For Circumscription, we just need to add

\[
\begin{align*}
\text{optItem}(X) & \leftarrow \text{item}(X), \ \text{minatom}(X). \\
\text{optItem}(t(X)) & \leftarrow \text{item}(X), \ \text{varyatom}(X). \\
\text{optItem}(f(X)) & \leftarrow \text{not} \ \text{item}(X), \ \text{varyatom}(X). 
\end{align*}
\]
Comparison between D-FLAT and D-FLAT^2
D-FLAT^2 – Discussion

Summary

- D-FLAT^2 [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-FLAT^2 \iff D-FLAT^n (generalize D-FLAT^2 to handle problems on the n^{th} level of the polynomial hierarchy)
- Implement further problems and improve D-FLAT^2 towards more general specifications of optimization task
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)?
动机

教训
- D-FLAT（或广义的DP）的瓶颈：表格的大小
  - 大小随着treewidth指数增长
- 我们可以找到与逻辑匹配的表（truth-table vs. formula）吗？

想法
- 使用二叉决策图（BDDs）：
  - 真值表的紧凑表示
  - 可以像公式一样处理
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 $\varphi = (a \land b \land c) \lor (a \land \neg b \land c) \lor (\neg a \land b \land c)$.

Figure: OBDD of $\varphi$. 
Binary Decision Diagrams

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 $\varphi$. 
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

$$B_t^l = \bigwedge_{(x,y) \in E_t} (\neg i_x \lor \neg i_y) \land \bigwedge_{y \in V_t} \left(d_y \leftrightarrow \bigvee_{(x,y) \in E_t} i_x \right)$$
DP of Independent Dominating Set (on Digraphs) via BDDs

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

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

\[ \left. \bigwedge_{(u,y) \in E_t \land u \neq y} (d_y \leftrightarrow d'_y \lor i_u) \land \bigwedge_{y \in V_t \land (u,y) \notin E_t} (d_y \leftrightarrow d'_y) \right] \]
DP of Independent Dominating Set (on Digraphs) via BDDs

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

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

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

\[
\mathcal{B}_t^r = \mathcal{B}_t' \left[ i_u/\top, d_u/\bot \right] \lor \mathcal{B}_t' \left[ i_u/\bot, d_u/\top \right]
\]
DP of Independent Dominating Set (on Digraphs) via BDDs

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

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

\[
\mathcal{B}_t^r = \mathcal{B}_t'[i_u/\top, d_u/\bot] \lor \mathcal{B}_t'[i_u/\bot, d_u/\top]
\]

\[
\mathcal{B}_t^j = \exists D_t D_t'' \left[ \mathcal{B}_t'[D_t/D_t'] \land \mathcal{B}_t''[D_t/D_t''] \land \bigwedge_{x \in V_t} (d_x \leftrightarrow d_x' \lor d_x'') \right]
\]
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
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)

<table>
<thead>
<tr>
<th>Decomposition Size:</th>
<th>Structural Features:</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ BagSize*</td>
<td>▶ NodeCount (#)</td>
</tr>
<tr>
<td>▶ BagSize_{NL}</td>
<td>▶ Percentage</td>
</tr>
<tr>
<td>▶ ContainerCount*</td>
<td></td>
</tr>
<tr>
<td>▶ (\Sigma) BagSize</td>
<td></td>
</tr>
<tr>
<td>▶ NodeCount</td>
<td></td>
</tr>
</tbody>
</table>

Introduce / Forget / Join / Leaf Nodes:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Depth*</td>
<td></td>
</tr>
<tr>
<td>▶ BagSize*</td>
<td></td>
</tr>
<tr>
<td>▶ Percentage</td>
<td></td>
</tr>
</tbody>
</table>
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.

We need to get a precise picture on crucial features.

Use gained insights to tailor tree decomposition heuristics.
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-FLAT^2
- dynBDD
Ongoing + Future Work

- Automatic generation of D-FLAT code from “standard” encoding
  - D-FLAT^2 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**


