Efficient Problem Solving on Tree Decompositions using Binary Decision Diagrams*

Günther Charwat and Stefan Woltran

Institute of Information Systems, TU Wien, Austria
{gcharwat,woltran}@dbai.tuwien.ac.at

Abstract. Dynamic programming (DP) on tree decompositions is a well studied approach for solving hard problems efficiently. Usually, implementations rely on tables for storing information, and algorithms specify how tuples are manipulated during traversal of the decomposition. However, a bottleneck of such table-based algorithms is relatively high memory consumption. Binary Decision Diagrams (BDDs) and related concepts have been shown to be very well suited to store information efficiently. While several techniques have been proposed that combine DP with efficient BDD-based storage for some particular problems, in this work we present a general approach where DP algorithms are specified on a logical level in form of set-based formula manipulation operations that are executed directly on the BDD data structure. In the paper, we provide several case studies in order to illustrate the method at work, and report on preliminary experiments. These show promising results, both with respect to memory and run-time.

1 Introduction

For problems that are known to be intractable, one approach is to exploit structural properties of the given input. An important parameter of graph-based instances is "tree-width", which, roughly speaking, measures the tree-likeness of the input. Tree-width is defined on so-called tree decompositions [30], where the instance is split into smaller parts, thereby taking into account its structure. The problem at hand can then be solved by dynamic programming (DP). Many problems are fixed-parameter tractable (fpt) with respect to tree-width, i.e., solvable in time $f(k) \cdot n^{\mathcal{O}(1)}$ where k is the tree-width, n is the input size and f is some computable function. Note that here the explosion in run-time is confined to k instead of the input size. Courcelle showed that every problem that is definable in monadic second-order logic (MSO) is fixed-parameter tractable with respect to tree-width [12]. There, the problem is solved via translation to a finite tree automaton (FTA). However, the algorithms resulting from such "MSO-to-FTA" translation are oftentimes impractical due to large constants [29]. One approach to overcome this problem is to develop dedicated DP algorithms for the problems at hand (e.g., [6,21]). Such algorithms typically rely on tables for storing information, resulting in a large memory

* This is a pre-print of an article published in LPNMR 2015, LNAI 9345, pp. 213–227. The final authenticated version is available online at https://doi.org/10.1007/978-3-319-23264-5_19. This work has been supported by the Austrian Science Fund (FWF): Y698, P25607, P25518.

footprint. This problem has been addressed, e.g., by proposing heuristics [4] or reducing the number of simultaneously stored tables [2].

In this work we mitigate the problem by developing DP algorithms with *native* support for efficient storage. In our approach, Binary Decision Diagrams (BDDs) [9] serve as the data structure. BDDs have undergone decades of research and are a well-established concept used, e.g., in model-checking [27], planning [22] and software verification [5]. Our approach is in line with recent research that studies the effectiveness of exploiting tree-width by applying decomposition techniques in combination with decision diagrams. In the area of knowledge compilation, so-called "Tree-of-BDDs" [19,33] are constructed in an offline phase from a given CNF, and queried in the online phase to answer questions on this data structure in linear time. Furthermore, Algebraic Decision Diagrams (ADDs) [3] are used for compiling Bayesian networks in such a way that the structure of the network can be exploited in order to compute inference efficiently [11]. Combining DP and decision diagrams has been proven well-suited also for Constraint Optimization Problems (COPs) [31]. The key idea is to employ ADDs to store the set of possible solutions, and the branch-and-bound algorithm is executed on a decomposition of the COP instance. This was shown to be superior to earlier approaches in [8], where additionally (no)good recording is applied during computation.

In this work we continue this promising branch of research. However, from a conceptual perspective, our algorithms are specified on a logical level as formulae. This gives a compact and exact specification of algorithms, which are executed directly on the BDDs in form of BDD manipulation operations. In contrast to table-based DP algorithms, we do not manipulate tuples directly, but modify the *set* of models. Furthermore, in the course of this work we develop two different DP algorithm design paradigms, which we call *early decision method* (EDM) and *late decision method* (LDM). In EDM, information is incorporated in the BDD as soon as it becomes available when traversing the tree decomposition and is thus similar to the approach usually employed in standard table-based implementations. As we will see, LDM gives rise to novel DP algorithms where the BDD manipulation operations are delayed until just before the information is removed. We illustrate these concepts by providing several case studies that exemplarily show how DP algorithms can be implemented following our approach. These prototypical problems differ in that only fixed information, also changing information or even connectedness has to be handled appropriately. While we focus here on problems that are NP-complete, we plan to apply our method also to problems beyond NP (thus covering applications from the AI and LPNMR domain) with the long-term goal to extend our way of DP algorithm specification to all MSO-definable problems. To summarize, the main contributions of this paper are as follows:

- An approach for specifying DP algorithms on tree decompositions via formula manipulation, and two design patterns called early/late decision method (EDM/LDM).
- Case studies of 3-COLORABILITY, STABLE EXTENSION (from the field of argumentation) and HAMILTONIAN CYCLE to illustrate our method at work.

– A performance analysis that compares memory and time requirements of our approach with available DP implementations, indicating that our approach significantly reduces memory requirements and gives advantages in performance.[1]

2 Background

Tree Decompositions. Tree decompositions, introduced in [30], are defined as follows.

Definition 1. *A* tree decomposition *of an undirected graph $G = (V, E)$ is a pair $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree and $\mathcal{X} : V_\mathcal{T} \to 2^V$ assigns to every node $V_\mathcal{T}$ of the tree a set of vertices V from the original graph. The sets of vertices $\mathcal{X} = (X_t)_{t \in V_\mathcal{T}}$ have to satisfy the following conditions: a) $\bigcup_{t \in V_\mathcal{T}} X_t = V$. b) $\{x, y\} \in E \Rightarrow \exists t \in V_\mathcal{T} : \{x, y\} \subseteq X_t$. c) $x \in X_{t'} \wedge x \in X_{t''} \wedge t''' \in path(t', t'') \Rightarrow x \in X_{t'''}$. X_t is also called the* bag *for the vertex $t \in V_\mathcal{T}$. The* width *w of the decomposition is $max_{t \in V_\mathcal{T}} |X_t| - 1$. The* tree-width *$k$ of a graph is the minimum width over all its tree decompositions.*

Intuitively, this definition guarantees that every vertex of the graph is contained in some bag of the tree decomposition, adjacent vertices appear together in some bag, and that nodes that contain the same vertex are connected. For problems on directed graphs, Definition 1 can be naturally extended. We will denote an edge between two vertices x, y by $\{x, y\}$, and directed arcs by (x, y). Furthermore, for a decomposition node t, we denote by $E_t = \{\{x, y\} \in E \mid x, y \in X_t\}$ the edges of G induced by the vertices X_t, and analogously by A_t the arcs in t. It is well-known that obtaining an optimal decomposition (with respect to width) is NP-hard [1], but there are heuristics that provide a "good" decomposition in polynomial time [7,13,14]. For the ease of representation, we consider a special type of tree decomposition throughout this work.

Definition 2. *A tree decomposition $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is called* normalized *if each $t \in V_\mathcal{T}$ is of one of the following types: (1)* Leaf *node: t has no child nodes. (2)* Introduction *node: t has exactly one child node t' with $X_{t'} \subset X_t$ and $|X_{t'}| = |X_t| - 1$. (3)* Removal *node: t has exactly one child node t' with $X_t \subset X_{t'}$ and $|X_{t'}| = |X_t| + 1$. (4)* Join *node: t has exactly two child nodes t' and t'' with $X_t = X_{t'} = X_{t''}$.*

Furthermore, without loss of generality, we assume that $X_r = \emptyset$ for the root node r of $\mathcal{T}$. Note that such a normalized decomposition can be obtained in linear time from an arbitrary one without increasing the tree-width [23].

Example 1. Figure 1 shows an example graph G and a possible (normalized) tree decomposition $\mathcal{T}$ ($\mathcal{T}_n$) of width 2. The tree decompositions are optimal w.r.t. width.

(Reduced Ordered) Binary Decision Diagrams. In our approach, Reduced Ordered Binary Decision Diagrams (ROBDDs) [9] serve as the data structure for storing information during the traversal of the decomposition.

[1] A prototype system, called *dynBDD*, which is built on top of the BDD library CUDD [32] is available under `http://dbai.tuwien.ac.at/proj/decodyn/dynbdd`.

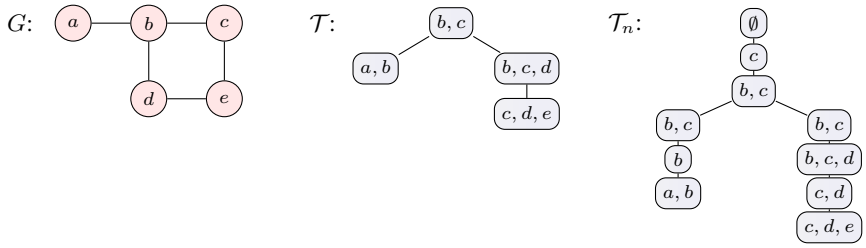

Fig. 1. Graph G and possible (normalized) tree decomposition $\mathcal{T}$ ($\mathcal{T}_n$) of G.

Definition 3. *An* Ordered Binary Decision Diagram $\mathcal{B} = (V_\mathcal{B}, A_\mathcal{B})$ *is a rooted, connected, directed acyclic graph where* $V_\mathcal{B} = V_T \cup V_N$ *and* $A_\mathcal{B} = A_\top \cup A_\bot$. *The following conditions have to be satisfied:*

1. V_T *may contain the terminal nodes* $\top$ *and* $\bot$.
2. V_N *contains the internal nodes, where each* $v \in V_N$ *represents a variable* v.
3. *Each* $v \in V_N$ *has exactly one outgoing arc in* $A_\top$ *and one in* $A_\bot$, *represented by a solid and a dashed arc respectively.*
4. *For every path from the root to a terminal node, each variable occurs at most once and in the same order (i.e., we have a strict total order over the variables).*

In Reduced OBDDs *(ROBBD), isomorphic nodes are merged into a single node with several incoming edges. Furthermore, nodes* $v \in V_N$ *where both outgoing arcs reach the same node* $v' \in V_\mathcal{B}$, *are removed.*

Given an OBDD $\mathcal{B}$, propositional variables V_N and an assignment A to V_N, the *corresponding path* in $\mathcal{B}$ is the unique path from the root node to a terminal node, such that for every $v \in V_N$ it includes the outgoing arc in $A_\top$ ($A_\bot$) iff A gets assigned true (false) for v. A is a satisfying assignment of the function represented by $\mathcal{B}$ iff the path ends in $\top$. With slight abuse of notation, in the following we will specify BDDs by giving the function in form of a logic formula.

Example 2. Figure 2 shows an OBBD $\mathcal{B}$ and the corresponding ROBBD $\mathcal{B}_{red}$ for formula $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$. Nodes c_1, c_2 and c_3 represent the same variable c and have arcs to the same terminal nodes. Hence, these isomorphic nodes are merged to a single node c. Then, both outgoing arcs of b_1 reach c, and b_1 is removed. Furthermore, c_4 is removed.

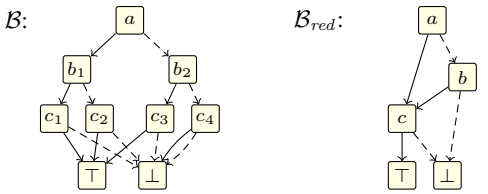

Fig. 2. OBBD and ROBBD of formula $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$.

BDDs support standard logical operators *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$) and *equivalence* ($\leftrightarrow$). Furthermore, for a BDD $\mathcal{B}$, *existential quantification* over a set of variables V, $V \subseteq V_N$ is denoted by $\exists V\mathcal{B}$. *Restriction* of a variable $v \in V_N$ to true ($\top$) or false ($\bot$) and *renaming* to a variable v' is denoted by $\mathcal{B}[v/\cdot]$ where $\cdot \in \{\top, \bot, v'\}$. For sets of variables $V \subseteq V_N$, $\mathcal{B}[V/\cdot]$ with $\cdot \in \{\top, \bot, V'\}$ and $V' = \{v' \mid v \in V\}$, denotes restriction or renaming of each $v \in V$ by applying $\mathcal{B}[v/\cdot]$.

3 DP on Tree Decompositions with BDDs

Our algorithms follow a general pattern of how the solution is constructed. First, the graph representation of the input instance is decomposed. Next, the decomposition is normalized in linear time. The resulting tree decomposition $\mathcal{T}$ is traversed in bottom-up order, and at each node $t \in V_\mathcal{T}$ the associated BDD, denoted by $\mathcal{B}_t$, is manipulated according to the problem at hand. In the root node r of the decomposition (where $X_r = \emptyset$), either $\mathcal{B}_r = \top$ or $\mathcal{B}_r = \bot$ holds, representing the solution to the problem.

We present two algorithm design choices, which we call the *early decision method* (EDM), where information is incorporated within introduction nodes, and the *late decision method* (LDM), where BDD manipulation is delayed until removal of vertices. For unsatisfiable instances, EDM potentially detects conflicts earlier during the traversal of the decomposition. However, LDM gives advantages when specifying more involved algorithms (see Section 4). Note that EDM is similar to the approach employed in standard table-based implementations, while LDM is usually harder to implement on tables.

For the case studies presented in the following, we will specify the manipulation operations on $\mathcal{B}_t$ based on the node type of t, with $\mathcal{B}_t^l$ representing the BDD resulting from a leaf node operation, $\mathcal{B}_t^i$ (introduction node), $\mathcal{B}_t^r$ (removal node), and $\mathcal{B}_t^j$ (join node). Nodes t', t'' denote child nodes, $\mathcal{B}_{t'}, \mathcal{B}_{t''}$ the BDDs constructed in the child nodes, and u the introduced or removed vertex (if any). All $\mathcal{B}_t$ for $t \in V_\mathcal{T}$ are required to share the same global variable ordering for efficiency during manipulation. In general, the size of the stored BDDs (i.e., the number of nodes in $\mathcal{B}_t$) is bounded by $\mathcal{O}(2^{wl})$ where w is the width of $\mathcal{T}$ and l the number of variables stored per bag element (i.e., vertex of the original input graph). However, in practice the size may be exponentially smaller, in particular in case a "good" variable ordering is applied [20]. Since finding an optimal variable ordering is in general NP-hard [9], we rely on BDD-internal heuristics for finding such a good ordering [32]. With this, the BDDs require much less space than an equivalent table representation as used in state-of-the-art systems (see Section 4).

3.1 3-Colorability

The 3-COLORABILITY problem ("*Given a graph G, is G 3-colorable?*") is very well-suited to illustrate how DP algorithms for problems that are FPT with respect to treewidth can be specified following our approach. As input, the algorithms expect a simple graph $G = (V, E)$. Furthermore, we define the set of colors $C = \{r, g, b\}$. The following variables are to be used within the BDDs: For all $c \in C$ and $x \in V$, the truth value of variable c_x denotes whether vertex x gets assigned color c.

EDM. The BDD manipulation operations given below are applied at the respective decomposition nodes. We have to guarantee that every vertex gets assigned exactly one color, and adjacent vertices do not have the same color. Intuitively, $\mathcal{B}_t^l$ and $\mathcal{B}_t^i$ are constructed by adding the respective constraints for introduced vertices. In $\mathcal{B}_t^r$, due to the definition of tree decompositions, we know that all constraints related to removed vertex u were already taken into account. Hence, we can abstract away the variables associated with u, thereby keeping the size of the BDD bound by the width of the decomposition. In join nodes, $\mathcal{B}_t^j$ combines the intermediate results obtained in the child nodes of the decomposition.

$$
\begin{aligned}
\mathcal{B}_t^l =& \bigwedge_{c \in C} \bigwedge_{\{x,y\} \in E_t} \neg(c_x \wedge c_y) \wedge \bigwedge_{x \in X_t} (r_x \vee g_x \vee b_x) \wedge \\
& \bigwedge_{x \in X_t} \Big(\neg(r_x \wedge g_x) \wedge \neg(r_x \wedge b_x) \wedge \neg(g_x \wedge b_x) \Big) \\
\mathcal{B}_t^i =& \mathcal{B}_{t'} \wedge \bigwedge_{c \in C} \bigwedge_{\{x,u\} \in E_t} \neg(c_x \wedge c_u) \wedge (r_u \vee g_u \vee b_u) \wedge \\
& \neg(r_u \wedge g_u) \wedge \neg(r_u \wedge b_u) \wedge \neg(g_u \wedge b_u) \\
\mathcal{B}_t^r =& \exists r_u g_u b_u [\mathcal{B}_{t'}] \qquad\qquad \mathcal{B}_t^j = \mathcal{B}_{t'} \wedge \mathcal{B}_{t''}
\end{aligned}
$$

LDM. Another possibility for specifying the algorithm is to incorporate information as late as possible, that is, when a vertex is removed from the decomposition. In leaf nodes the BDD $\mathcal{B}_t^l$ is initialized with $\top$, and in introduction nodes the BDD $\mathcal{B}_t^i$ corresponds to that of the child nodes. When a vertex u is removed, one variable of r_u, g_u, b_u is set to true, thereby assigning to the vertex exactly one color $c \in C$. Furthermore, neighboring vertices x with $\{x, u\} \in E_{t'}$ must not get assigned the same color, which is achieved by adding $\neg c_x$ to the formula. $\mathcal{B}_t^r$ is simply the disjunction over the three BDDs resulting from the choice of the color. As in EDM, it is sufficient to construct $\mathcal{B}_t^j$ via conjunction of the child BDDs.

$$
\begin{aligned}
\mathcal{B}_t^l =& \top \qquad\qquad \mathcal{B}_t^i = \mathcal{B}_{t'} \qquad\qquad \mathcal{B}_t^j = \mathcal{B}_{t'} \wedge \mathcal{B}_{t''} \\
\mathcal{B}_t^r =& \big(\mathcal{B}_{t'}[r_u/\top, g_u/\bot, b_u/\bot] \wedge \bigwedge_{\{x,u\} \in E_{t'}} \neg r_x\big) \vee \\
& \big(\mathcal{B}_{t'}[r_u/\bot, g_u/\top, b_u/\bot] \wedge \bigwedge_{\{x,u\} \in E_{t'}} \neg g_x\big) \vee \\
& \big(\mathcal{B}_{t'}[r_u/\bot, g_u/\bot, b_u/\top] \wedge \bigwedge_{\{x,u\} \in E_{t'}} \neg b_x\big)
\end{aligned}
$$

3.2 Stable Extension

The STABLE EXTENSION problem ("*Given an argumentation framework AF, does there exist a stable extension in AF?*") is a well-known problem from the area of abstract argumentation [16]. An argumentation framework $AF = (V, A)$ is a directed

graph where the vertices V represent arguments and the arcs A the so-called attack relation between arguments. A stable extension $\mathcal{E}$ of an AF is a set $\mathcal{E} \subseteq V$ that is (i) conflict-free, i.e., for all $x, y \in \mathcal{E} : (x, y) \notin A$ holds, and (ii) all arguments are either in the set or defeated, i.e., for all $x \in V : x \in \mathcal{E} \vee (\exists (y, x) \in A : y \in \mathcal{E})$ holds. In the area of argumentation, DP algorithms for various semantics have been studied in [17].

In our BDD-based approach we specify the following variables. For all $x \in V$, the truth value of variable i_x denotes whether argument x is in some $\mathcal{E}$. Furthermore, the assignment of true to variable d_x represents that x is defeated. Additionally, for a node t of the tree decomposition, we denote by $D_t = \{d_x \mid x \in X_t\}$ the defeated arguments in t. Here, i_x for $x \in V$ are variables with *fixed* truth assignment (i.e., containment in an extension is *guessed once*), while all d_x have a truth assignment that *changes* during the traversal of the tree decomposition (an argument may *become* defeated in a decomposition node).

$$
\begin{aligned}
\mathcal{B}_t^l =& \bigwedge_{(x,y)\in A_t} (\neg i_x \vee \neg i_y) \wedge \bigwedge_{y \in X_t} \Big(d_y \leftrightarrow \bigvee_{(x,y)\in A_t} i_x \Big) \\
\mathcal{B}_t^i =& \exists D'_{t'} \Big[\mathcal{B}_{t'}[D_{t'}/D'_{t'}] \wedge \bigwedge_{\{u,y\}\in E_t} (\neg i_u \vee \neg i_y) \wedge \Big(d_u \leftrightarrow \bigvee_{(x,u)\in A_t} i_x \Big) \wedge \\
& \bigwedge_{\substack{(u,y)\in A_t \wedge \\ u \neq y}} \big(d_y \leftrightarrow d'_y \vee i_u \big) \wedge \bigwedge_{y \in X_t \wedge (u,y) \notin A_t} \big(d_y \leftrightarrow d'_y \big) \Big] \\
\mathcal{B}_t^r =& \mathcal{B}_{t'}[i_u/\top, d_u/\bot] \vee \mathcal{B}_{t'}[i_u/\bot, d_u/\top] \\
\mathcal{B}_t^j =& \exists D'_t \exists D''_t \Big[\mathcal{B}_{t'}[D_t/D'_t] \wedge \mathcal{B}_{t''}[D_t/D''_t] \wedge \bigwedge_{x \in X_t} \big(d_x \leftrightarrow d'_x \vee d''_x \big) \Big]
\end{aligned}
$$

EDM. In leaf nodes, variable d_y for arguments $y \in X_t$ is true (i.e., defeated) iff one of its attacking arguments is in the stable extension, and adjacent arguments can not be both in the extension. In $\mathcal{B}_t^i$, for u the formula is constructed as in leaf nodes. In order to update the truth value of defeat variables, for any argument y we apply a general pattern of $\exists y'[\mathcal{B}_{t'}[y/y'] \wedge (y \leftrightarrow (y' \vee cond))]$, that is, renaming, potentially adding conditions ($cond$), and removing the renamed variable y'. Here, $cond$ contains i_u in case u is an incoming neighbor of y. By this, the size of the BDDs remains bounded by the width of the decomposition. In removal nodes, u must either be contained in the extension, or it is defeated. Note that the conflict-free property would be violated in case u is both in the extension and defeated. In $\mathcal{B}_t^j$, the defeat information is propagated via renaming, equivalence, and existential quantification.

$$\mathcal{B}^l_t = \bigwedge_{x \in X_t} \neg d_x \qquad \mathcal{B}^i_t = \mathcal{B}_{t'} \wedge \neg d_u$$

$$\mathcal{B}^r_t = \phi^r_t[i_u/\top, d_u/\bot] \vee \phi^r_t[i_u/\bot, d_u/\top] \text{ with}$$

$$\phi^r_t = \exists D'_{t'} \Big[\mathcal{B}_{t'}[D_{t'}/D'_{t'}] \wedge \bigwedge_{\{u,y\} \in E_{t'}} (\neg i_u \vee \neg i_y) \wedge \bigwedge_{y \in X_t} (d_y \leftrightarrow d'_y \vee_{(u,y) \in A_{t'}} i_u) \wedge (d_u \leftrightarrow d'_u \vee \bigvee_{(x,u) \in A_{t'}} i_x) \Big]$$

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

LDM. Here, all information is considered when a vertex is removed. Hence, introduced vertices cannot become defeated in leaf or introduction nodes, and the corresponding variables are initialized with $\bot$. In $\mathcal{B}^r_t$, we guess whether the removed vertex u is in the extension or defeated. Furthermore, we guarantee conflict-freeness with vertices adjacent to u. A vertex y becomes defeated if it is attacked by u and u is in the extension, and u is defeated if it is attacked by some vertex that was already removed, or by an in-vertex on an arc in $A_{t'}$. Note that we use a small disjunction symbol with condition whenever there is at most one disjunction in the instantiated formula, and a large symbol otherwise. $\mathcal{B}^j_t$ is specified as in EDM.

3.3 Hamiltonian Cycle

The HAMILTONIAN CYCLE problem ("*Given a graph $G = (V, E)$, does there exist a Hamiltonian Cycle in G?*") requires a more involved algorithm specification. Monolithic propositional encodings (where the whole instance is available at once) allow one to assign a global order over the variables that specifies the ordering over the vertices in the cycle. However, in our DP-based approach, we are restricted to information that is available in the current decomposition node. Hence, we consider a *relative* ordering as follows. The idea is to first specify exactly one incoming and one outgoing edge for each vertex. For $x \in V$, the truth value of variable i_x (o_x) denotes that it has an outgoing (incoming) edge. A selected edge $\{x, y\} \in E$ is represented by variable t_{xy}. Second, we have to guarantee that we have a *single* cycle that covers all vertices. Therefore we select a fixed vertex $f \in V$ that denotes where the cycle starts and ends. Variable a_{xy} for $x, y \in V$ denotes that x lies after y on the path from f to f. For a tree decomposition node t we have $S_t = \{i_x, o_x, a_{xy} \mid x, y \in X_t\}$. Furthermore, for a vertex $u \in X_t$, let $T_{t,u} = \{t_{xu}, t_{ux} \mid \{x, u\} \in E_t\}$. Due to space limitations in the following we only present the LDM version.

LDM. In leaf and introduction nodes all changing variables are initialized with $\bot$. In removal nodes, at least one incoming edge for removed vertex u is selected. Here, i'_u is true iff the incoming neighbor of u was already removed from the bag. Furthermore, at most one incoming edge from $E_{t'}$ is selected. Finally, if i'_u is true, we cannot select an additional incoming edge, and the incoming and outgoing edges for u have to be different. The same construction is used to guarantee exactly one outgoing edge for u.

For vertices $x \in X_t$, i_x and o_x are updated in case u was a neighbor of x. Again, at most one incoming (outgoing) edge must be selected. For $x, z \in X_t$, a_{xz} becomes true if $u \neq f$ and u lies on the path between x and z. With this, we keep information on the path (from f to f), restricted to X_t, where the truth value of t_{xy}-variables represents selected edges in X_t and a_{xy}-variables denote that x is before y on the path where intermediate vertices were already removed. Finally, in case a_{xx} for $x \neq f$ is true, we know that there is a cycle that does not cover f, and is therefore no Hamiltonian cycle. In join nodes, i_x, o_x and a_{xy} variables are propagated as usual. Here, whenever both i'_x and i''_x are true, due to the connectedness condition of tree decompositions and the fact that these variables are updated when a vertex is removed, x has different incoming edges and cannot be a solution. The same holds for outgoing edges.

$$
\begin{aligned}
\mathcal{B}^l_t =& \bigwedge_{x \in X_t} (\neg i_x \wedge \neg o_x) \wedge \bigwedge_{x,y \in X_t} \neg a_{xy} \\
\mathcal{B}^i_t =& \mathcal{B}_{t'} \wedge \neg i_u \wedge \neg o_u \wedge \bigwedge_{x \in X_t} (\neg a_{xu} \wedge \neg a_{ux}) \\
\mathcal{B}^r_t =& \exists T_{t',u} S'_{t'} \Big[\mathcal{B}_{t'}[S_{t'}/S'_{t'}] \wedge (i'_u \vee \bigvee_{\{x,u\} \in E_{t'}} t_{xu}) \wedge (o'_u \vee \bigvee_{\{u,y\} \in E_{t'}} t_{uy}) \wedge \\
& \bigwedge_{\substack{\{x',u\} \in E'_t \wedge \\ \{x'',u\} \in E'_t \wedge x' \neq x''}} \big(\neg(t_{x'u} \wedge t_{x''u}) \wedge \neg(t_{ux'} \wedge t_{ux''})\big) \wedge \\
& \bigwedge_{\{x,u\} \in E_{t'}} \big(\neg(i'_u \wedge t_{xu}) \wedge \neg(o'_u \wedge t_{ux}) \wedge \neg(t_{xu} \wedge t_{ux})\big) \wedge \\
& \bigwedge_{x \in X_t} \Big(\big(i_x \leftrightarrow (i'_x \vee_{\{u,x\} \in E_{t'}} t_{ux})\big) \wedge \big(o_x \leftrightarrow (o'_x \vee_{\{x,u\} \in E_{t'}} t_{xu})\big) \Big) \wedge \\
& \bigwedge_{\{x,u\} \in E_{t'}} \big(\neg(i'_x \wedge t_{ux}) \wedge \neg(o'_x \wedge t_{xu})\big) \wedge \\
& \bigwedge_{x,z \in X_t} \Big(a_{xz} \leftrightarrow a'_{xz} \vee_{u \neq f} \big((a'_{xu} \vee_{\{x,u\} \in E_{t'}} t_{xu}) \wedge (a'_{uz} \vee_{\{u,z\} \in E_{t'}} t_{uz})\big) \Big) \wedge \\
& \bigwedge_{x \in X_t \wedge x \neq f} \neg a_{xx} \Big] \\
\mathcal{B}^j_t =& \exists S'_t S''_t \Big[\mathcal{B}_{t'}[S_t/S'_t] \wedge \mathcal{B}_{t''}[S_t/S''_t] \wedge \\
& \bigwedge_{x \in X_t} \Big(\big(i_x \leftrightarrow (i'_x \vee i''_x)\big) \wedge \big(o_x \leftrightarrow (o'_x \vee o''_x)\big) \wedge \neg(i'_x \wedge i''_x) \wedge \neg(o'_x \wedge o''_x) \Big) \wedge \\
& \bigwedge_{x,y \in X_t} \big(a_{xy} \leftrightarrow (a'_{xy} \vee a''_{xy})\big) \Big]
\end{aligned}
$$

4 Experimental Analysis

The aforementioned algorithms were implemented in the prototype system *dynBDD* that utilizes the library CUDD [32] for efficient BDD management and the HTDECOMP library [14] for constructing the tree decompositions by applying the "min-degree" heuristics. We compare run-time and memory requirements to freely available

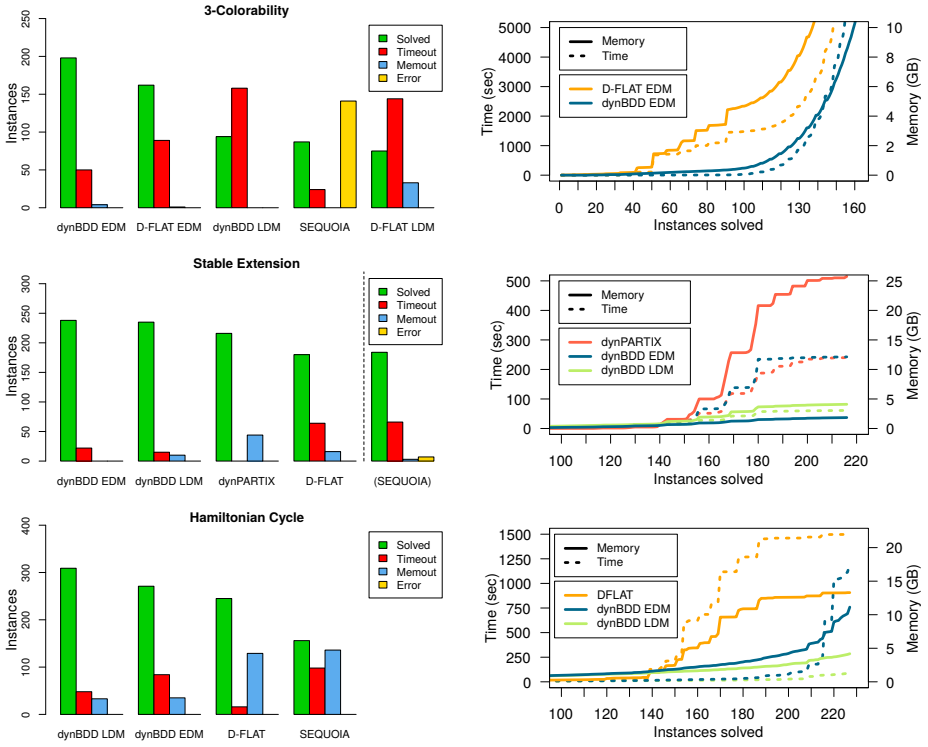

Fig. 3. Result Overview: System comparison and detailed results for best-performing systems.

implementations that also utilize the concept of DP on tree decompositions, namely SEQUOIA [25] (version 0.9) and D-FLAT [6] (version 1.0.0). Furthermore, for the area of abstract argumentation, the DP-based dynPARTIX system [10] (version 2.0) is available. SEQUOIA implements a game-theoretic approach [24]. As input, SEQUOIA expects the problem to be formulated as an MSO formula. The instance is decomposed and the DP algorithm automatically generated and executed. D-FLAT combines DP with answer-set programming (ASP). In contrast to SEQUOIA, the user specifies an ASP encoding that is executed at each node of the decomposition, thereby defining the DP algorithm explicitly. dynPARTIX comprises of implementations of reasoning tasks relevant to the field of argumentation.

All experiments were performed on a single core of an AMD Opteron 6308 (3.5GHz) processor running Debian GNU/Linux 7 (kernel 3.2.0-4-amd64). Each run was limited to 10 minutes (Timeout) and 4GB of memory (Memout). Instances were generated using the random graph model due to Erdös and Rényi [18]. This allows us to compare the implementations on various instances that cover a broad range of different widths. Below, we denote by n the number of vertices and by p the edge probability of an instance. Since run-time depends on the heuristically obtained tree decomposition, we run the algorithms on the same normalized tree decompositions (if not stated otherwise).

3-Colorability. To analyze the performance, the EDM and LDM versions of the algorithms were implemented in dynBDD and D-FLAT. For SEQUOIA, the performance of

the MSO-based algorithm is measured. Overall, 252 instances with n between 10 and 1000 and p between 0.001 (very sparse) and 0.2 (dense) were tested. Figure 3 (upper left) shows the number of solved instances per system and implementation variant. SEQUOIA implements a pre-check to tell whether it is capable of solving the instance. *Error* denotes the number of times this check failed. Results show that EDM is by far superior to LDM. Here, we observed that EDM detects conflicts in unsatisfiable instances earlier than LDM. In fact, additional analysis showed that for satisfiable instances dynBDD (EDM) performed only marginally better than dynBDD (LDM). In total, 161 instances were solved by both dynBDD (EDM) and D-FLAT (EDM). Figure 3 (lower left) gives details on the accumulated run-time and memory usage of the best-performing systems over these instances. The figure shows how many instances were solved after a certain amount of time. For example, D-FLAT (EDM) solved the first 100 instances in approx. 1500 seconds with a total of 4700 MB of memory usage, whereas dynBDD (EDM) required only 30 seconds and 500 MB. In total, dynBDD (EDM) required approx. 18% less time and 47% less memory on solved instances, and solved 36 instances more than D-FLAT (EDM). Note that we omitted dynBDD (LDM) since it solves significantly less instances than the best systems.

Regarding the width w of the tree decompositions, dynBDD (EDM) solved satisfiable instances up to $w = 48$ and unsatisfiable instances up to $w = 944$. While unsatisfiable instances of high width may be easily solvable due to early conflict detection, the measured width for solved satisfiable instances is quite large. Recall that the size of the BDD (or an equivalent table representation) may be up to $\mathcal{O}(2^{wl})$ which corresponds to $2^{48 \cdot 3} \approx 2.2 \cdot 10^{43}$ for $w = 48$ and 3 variables per bag element. This indicates that BDDs are indeed memory-efficient.

Stable Extension. We compare dynBDD with D-FLAT as well as dynPARTIX. Note that dynPARTIX constructs decompositions on the same heuristics as used for the other systems, but obtaining the same decomposition can not be guaranteed. Furthermore, we compare dynBDD to SEQUOIA. However, to the best of our knowledge, SEQUOIA is incapable of handling directed graphs. To give an impression on its performance, we decided to show results for the related INDEPENDENT DOMINATING SET problem.

260 instances with n between 10 and 100 and p between 0.001 and 0.1 were tested. Figure 3 (middle column) illustrates the overall number of solved instances per system. The run-time of the problem-tailored implementation in dynPARTIX (that implements an approach similar to EDM) is almost the same as that of dynBDD (EDM). However, the advantage of BDDs becomes evident with respect to memory, where dynBDD (EDM) requires less than 8% of that of dynPARTIX. Although dynBDD (LDM) solves slightly less instances and requires more memory than dynBDD (EDM), it is by far fastest implementation over all solved instances. One reason may be that due to the LDM specification being more compact than the EDM version, less BDD manipulation operations are to be executed. Compared to D-FLAT (restricted to solved instances), dynBDD (EDM) uses less than 0.6% of time and 9% of memory. dynBDD (LDM) requires less than 0.5% of time and 14% of memory compared to D-FLAT.

Hamiltonian Cycle. In our dynBDD implementation for this problem we selected the lexicographically smallest vertex as fixed vertex f. A study of how this selection influences run-time is deferred to future work. We tested 390 instances with n between

10 and 50 and p between 0.01 and 0.25. Here, generated instances had a width between 1 and 22. As depicted in Figure 3, dynBDD (LDM) solved most instances, followed by dynBDD (EDM) and D-FLAT. For this problem, it becomes apparent that width is crucial for the run-time. Considering instances solved by the three best-performing systems, we observed that instances up to width 12 were solved. Note that in this case we have BDDs with up to $2^{12\cdot(6+4\cdot12)} \approx 1.3 \cdot 10^{36}$ nodes since we have $3 \cdot 1$ i_x, $3 \cdot 1$ o_x, 12 t_{xy}, and $3 \cdot 12$ a_{xy}-variables per vertex in the bag (including renamed variables).

As also observed for the STABLE EXTENSION problem, our results indicate that for more complex problems the LDM variant pays off, especially for satisfiable instances.

5 Conclusion

In this work we showed how classical DP algorithms on tree decompositions can be reformulated in order to be executed on Binary Decision Diagrams. This gives rise to algorithms that are specified on a logical level where - opposed to manipulation of tuples in a table-based specification - the set of models is modified by executing operations directly on the BDD. Furthermore, we studied two algorithm design patterns, namely early (EDM) and late decision method (LDM), and illustrated the concepts by providing several case studies. The case studies are exemplary for NP-complete problems that are tractable w.r.t. tree-width. The corresponding algorithms are specified solely on fixed variables (3-COLORABILITY), additionally require changing variables (STABLE EXTENSION) and handling of connectedness within the DP algorithm (HAMILTONIAN CYCLE). From a practical perspective, our work is in line with the freely available systems D-FLAT [6], SEQUOIA [25] and dynPARTIX [10]. Our preliminary experiments showed that the implementation of dynBDD indeed mitigates performance and memory shortcomings of these systems. In particular, results indicate that for problems which require a more involved algorithm, LDM is superior to EDM. Note that our system currently does not implement any problem-specific shortcuts and that the libraries have been employed as black-box tools.

In the future, we want to tighten the integration of BDD handling and the tree decomposition (in particular, to obtain a good ordering of variables in the BDD from the structure of the decomposition) and to study how problem-specific shortcuts can be incorporated. Additionally, our approach natively supports parallel problem solving (over decomposition branches), which would be a complementary approach to recent developments on parallel BDD implementations [15,26]. Finally, our approach can directly be extended to problems that involve optimization. Here, ADDs as well as Multi-valued Decision Diagrams (MDDs) (see, e.g., [28]) and related data structures can serve as appropriate tools. Most importantly, we want to study our approach in the context of problems that are hard for the second level of the polynomial hierarchy (e.g., Circumscription, Abduction) and ultimately provide a tool-set for all MSO-definable problems.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods 8, 277–284 (1987)

2. Aspvall, B., Telle, J.A., Proskurowski, A.: Memory requirements for table computations in partial k-tree algorithms. Algorithmica 27(3), 382–394 (2000)
3. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebric decision diagrams and their applications. Formal Methods in System Design 10(2-3), 171–206 (1997)
4. Betzler, N., Niedermeier, R., Uhlmann, J.: Tree decompositions of graphs: Saving memory in dynamic programming. Discrete Optimization 3(3), 220–229 (2006)
5. Beyer, D., Stahlbauer, A.: BDD-based software verification - Applications to event-condition-action systems. STTT 16(5), 507–518 (2014)
6. Bliem, B., Morak, M., Woltran, S.: D-FLAT: declarative problem solving using tree decompositions and answer-set programming. TPLP 12(4-5), 445–464 (2012)
7. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Inf. Comput. 208(3), 259–275 (2010)
8. Boutaleb, K., Jégou, P., Terrioux, C.: (No)good recording and ROBDDs for solving structured (V)CSPs. In: Proc. ICTAI. pp. 297–304. IEEE Computer Society (2006)
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 100(8), 677–691 (1986)
10. Charwat, G., Dvořák, W.: dynPARTIX 2.0 - Dynamic programming argumentation reasoning tool. In: Proc. COMMA. FAIA, vol. 245, pp. 507–508. IOS Press (2012)
11. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: Proc. IJCAI. pp. 2443–2449 (2007)
12. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Inf. Comput. 85(1), 12–75 (1990)
13. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
14. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B.J., Musliu, N., Samer, M.: Heuristic methods for hypertree decomposition. In: Proc. MICAI. LNCS, vol. 5317, pp. 1–11. Springer (2008)
15. van Dijk, T., Laarman, A., van de Pol, J.: Multi-core BDD operations for symbolic reachability. Electr. Notes Theor. Comput. Sci. 296, 127–143 (2013)
16. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321 – 357 (1995)
17. Dvořák, W., Pichler, R., Woltran, S.: Towards fixed-parameter tractable algorithms for abstract argumentation. Artif. Intell. 186, 1–37 (2012)
18. Erdös, P., Rényi, A.: On random graphs, I. Publicationes Mathematicae (Debrecen) 6, 290–297 (1959)
19. Fargier, H., Marquis, P.: Knowledge compilation properties of Trees-of-BDDs, revisited. In: Proc. IJCAI. pp. 772–777 (2009)
20. Friedman, S.J., Supowit, K.J.: Finding the optimal variable ordering for binary decision diagrams. In: Proc. IEEE Design Automation Conference. pp. 348–356. ACM (1987)
21. Groër, C., Sullivan, B.D., Weerapurage, D.: INDDGO: Integrated network decomposition & dynamic programming for graph optimization. Tech. Rep. ORNL/TM-2012/176 (2012)
22. Kissmann, P., Hoffmann, J.: BDD ordering heuristics for classical planning. J. Artif. Intell. Res. (JAIR) 51, 779–804 (2014)
23. Kloks, T.: Treewidth, Computations and Approximations, LNCS, vol. 842. Springer (1994)
24. Kneis, J., Langer, A., Rossmanith, P.: Courcelle's theorem - A game-theoretic approach. Discrete Optimization 8(4), 568–594 (2011)
25. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: Proc. ALENEX. pp. 55–63 (2012)
26. Lovato, A., Macedonio, D., Spoto, F.: A thread-safe library for binary decision diagrams. In: Proc. SEFM. LNCS, vol. 8702, pp. 35–49. Springer (2014)

27. Męski, A., Penczek, W., Szreter, M., Woiźna-Szcześniak, B., Zbrzezny, A.: BDD-versus SAT-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance. Autonomous Agents and Multi-Agent Systems 28(4), 558–604 (2014)
28. Miller, D.M.: Multiple-valued logic design tools. In: Proc. MVL. pp. 2–11 (1993)
29. Niedermeier, R.: Invitation to fixed-parameter algorithms. Oxford Lecture Series in Mathematics and its Applications, vol. 31. OUP, Oxford (2006)
30. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
31. Sachenbacher, M., Williams, B.C.: Bounded search and symbolic inference for constraint optimization. In: Proc. IJCAI. pp. 286–291. PBC (2005)
32. Somenzi, F.: CU Decision Diagram package release 2.5.0. Department of Electrical and Computer Engineering, University of Colorado at Boulder (2012)
33. Subbarayan, S.: Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In: Proc. CPAIOR. LNCS, vol. 3524, pp. 351–365. Springer (2005)